Programming-in-the-Many:
A Software Engineering Paradigm for the 21st Century

Nenad Medvidovic Marija Mikic-Rakic

Henry Salvatori Computer Science Center 338
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 argues for a concerted research effort needed to address the challenges of PitM. As "proof-of-concept" we highlight the results of a research project we have conducted in this area over the past two years. While the details of our work may not be universally applicable, we believe that our approach suggests a plausible software engineering research agenda for the future. 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 several applications. While a number of issues pertaining to PitM remain areas of future work, our experience to date has been very positive.

1 INTRODUCTION

The software systems of today are rapidly growing in size, complexity, amount of distribution, heterogeneity of constituent building blocks *(components*), and numbers of users. We have recently witnessed a rapid increase in the speed and capacity of hardware, a decrease in its cost, the emergence of the Internet as a critical worldwide resource, and a proliferation of hand-held consumer electronics devices (e.g., mobile phones, personal digital assistants). In turn, this has resulted in an increased demand for software applications, outpacing our ability to produce them, both in terms of their sheer numbers and the sophistication demanded of them.

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. In addition, the military relevance of such scenarios is rapidly increasing, as evidenced by the recent purchase of up to 80,000 Palm Pilot hand-held computers by the U.S. Army and Navy and the designation of the USS McFaul as a "test platform" for hand-helds [13]. 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.

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)* [2]. We argue 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 argues for the need to develop methods, techniques, and tools whose goal is to address the challenges of PitM.

We believe the most effective approach to PitM will be to interweave ideas that had been explored in the past with new ideas, adopting and, where necessary, adapting the results of existing software development techniques. At the same time, an effective approach to PitM will require novel solutions spanning at least six key areas:

- explicit design idioms comprising an architectural style [9,11];
- architectural self-awareness to enable continuous architectural monitoring, analysis, and evolution;
- tailorable architecture implementation, deployment, and evolution infrastructure;
- explicit, first-class software connectors [8];
- architecture-based support for distributed application deployment, mobility, and dynamic reconfiguration; and
- support for disconnected operation.

We have begun exploring these issues and applying our results on a handful of applications executing on a variety of desktop and mobile computing platforms. While several of the aspects of the proposed approach would be by themselves important contributions to the software engineering state-of-the-practice (e.g., architectural style for PitM, architectural self-awareness, support for disconnected operation), their true benefit will lie in their combination. In turn, this combination has the potential to set the long-term research and development agenda for PitM.

The remainder of the paper is organized as follows. Section 2 discusses the nature of the problem we are addressing while Section 3 introduces an example application. Section 4 argues for an architectural approach to PitM. Throughout the paper, we highlight the pertinent results from our own research into PitM.

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 [7]. In the interest of space, we only present a cross-section of the challenges here.

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. 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 system software. Although standardization efforts have been undertaken for the emerging new class of mobile devices (e.g., various wireless network protocols), the world of PitM is still characterized by proprietary operating systems (e.g., PalmOS), specialized dialects of existing programming languages (e.g., Java KVM [12] 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 suggested approach to supporting PitM is 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 may be able to gain a lot of leverage is software architecture [9,11]. 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 as further detailed below.

3 EXAMPLE APPLICATION

To illustrate the challenges posed by PitM we use an application for distributed military Troops Deployment and battle Simulations (TDS), depicted in Figure 1. We have designed, analyzed, implemented, deployed, migrated, and dynamically evolved TDS using a specific instance [7] of the architecture-based techniques for PitM advocated in this paper. 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 operating systems (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).

4 ARCHITECTURAL SUPPORT FOR PITM

A set of design idioms is needed to effectively capture application architectures found in the PitM setting. These idioms, comprising an architectural style, must be able to address the key characteristics of PitM discussed in Section 1: architectural monitoring and analysis, distribution, dynamism, mobility, and disconnected operation. A number of existing styles, particularly those supporting distributed applications [3], may inform the development of the PitM style.

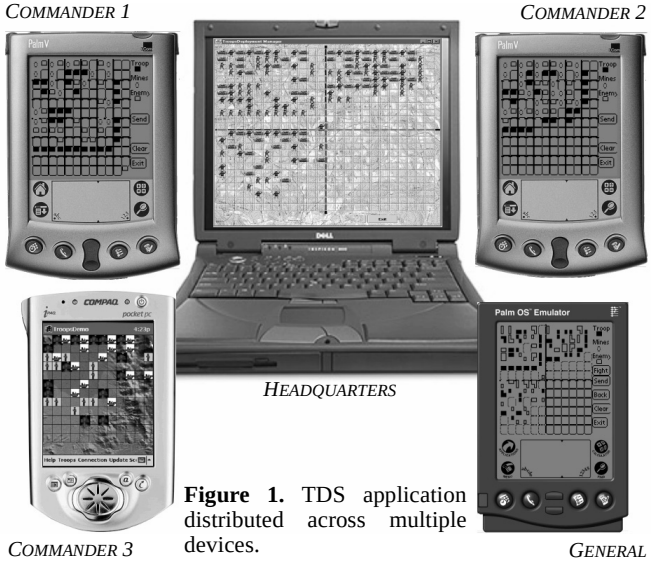

Figure 1. TDS application distributed across multiple devices.

In our investigation of PitM to date, we have leveraged our experience with the C2 architectural style [14]. 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) must be built on top of C2's existing facilities. Moreover, certain aspects of PitM simply cannot be supported by C2. For example, C2 mandates that components engage in asynchronous interactions only, making it ill suited for applications with real-time requirements. Furthermore, C2 imposes a strictly vertical topological orientation on architectures. 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 investigated seven major enhancements to the C2 style to account for the above shortcomings, as discussed below. It should be noted that, while we do not expect that all approaches to PitM would follow the same path we undertook (e.g., they would not be based on C2), we believe that these seven enhancements form a stable, generally applicable foundation for studying and addressing the challenges of PitM.

4.1 Peer-to-Peer Interaction

In order to support peer-to-peer interactions, we have introduced an extension to the C2-style by adding a new component port (called *side*) and message category (called *peer*). Side ports alleviate the relative topological rigidity of C2. At the same time, new composition rules are introduced to preserve the beneficial topological constraints of C2. For example, 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 inherited from C2 that components are independent of their substrate.

4.2 Architectural Self-Awareness

The nature of PitM applications demands support for their on-going monitoring, dynamic reconfiguration, deployment, mobility, and disconnected operation. To effectively address these needs, PitM should support 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 of application-level components.

In support of such a two-level architecture, we have identified the need for at least three types of communication messages. *ApplicationData* messages are used by application-level components to interact during execution. The other two message types, *ComponentContent* and *ArchitecturalModel*, are used by 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.

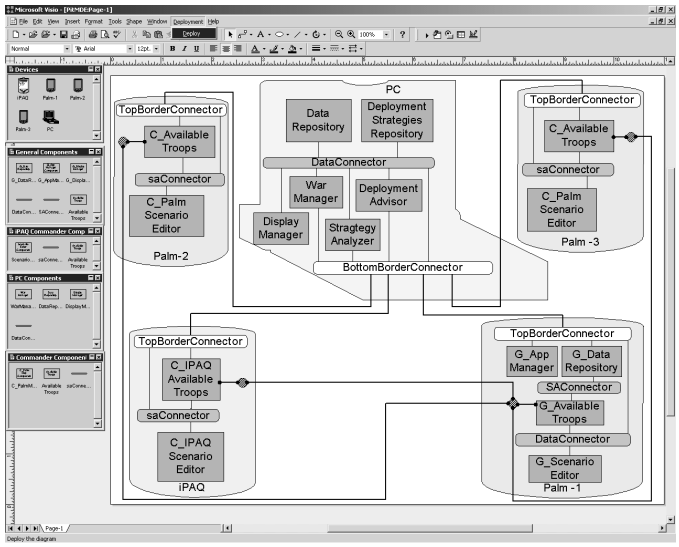

Figure 2. Architecture of the TDS application, displayed in *Prism*, the PitM deployment environment. The unlabeled circles connecting components across the hand-held devices represent *peer* connectors.

To date, we have developed several special-purpose meta-level components. For example, to facilitate deployment and migration of application components across devices (using *ComponentContent* messages) we have developed *Admin Component*s; to analyze the architectural models during the application's execution (using *ArchitecturalModel* messages), we have developed meta-level *Continuous Analysis* components.

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 Implementation

The proposed extensions to the C2 style outlined above provide 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. The implementation support must be highly efficient to account for the resource constrained nature of hardware platforms in the PitM setting. At the same time, we believe that the support must be provided in mainstream programming languages (e.g., as opposed to the frequently used special-purpose languages for embedded systems [7]) in order to be truly usable.

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 proposed style (e.g., architectures, components, connectors, ports, 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: Python, Java JVM and KVM, C++ and EVC++.

To support a variety of development situations in the PitM setting, we have implemented a library of PitM connectors on top of the framework [7]. These include *basic connectors* that support both synchronous and asynchronous message broadcast, multicast, and unicast; *border connectors* that encapsulate component interactions across process and machine boundaries (including infra-red and XML-based connectors); and *multi-versioning connectors* [10] that support reliable upgrading of application functionality at runtime.

4.5 Deployment

Support for software deployment is critical in highly distributed systems such as those found in PitM [5]. However, existing deployment techniques (e.g., Software Dock [5]) may not be suitable for this setting because of the much larger numbers of devices involved, their mobility, and their resource scarcity. Instead, much lighter-weight approaches are needed that will directly leverage the architectural basis of PitM.

Our support for deployment directly leverages the PitM implementation infrastructure. In order to deploy the desired configuration on a set of target hosts, we assume that a skeleton configuration, consisting of a border connector and the meta-level *AdminComponent* that supports code migration, is preloaded on each host. We have integrated and extended the COTS Microsoft Visio tool to develop *Prism*, the PitM architectural modeling and deployment environment (see Figure 2). Prism creates a description of the configuration 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 *ComponentContent* messages to *Admin Components* residing on the connected devices, which, in turn, perform the deployment of the specified configuration.

4.6 Mobility and Dynamic Reconfiguration

If, during the application's execution, a desired component- or system-level property is violated, the architecture may decide to reconfigure itself. We use the same technique for supporting runtime component mobility as we do for deployment: *Admin Components* exchanging *ComponentContent* messages that contain mobile code. This choice is made, again, because the nature of PitM applications and devices on which they execute will demand more efficient solutions than those provided by existing code mobility techniques (e.g., [4]). Performing the migration or dynamic reconfiguration results in the following process:

1. The migrant component is disconnected from its attached connectors.

2. *Admin Component* on the source device unloads the migrant component from the local subsystem.
3. *Admin Component* on the source device packages the migrant component and sends it as a *ComponentContent* message through its local *Border Connector.*
4. Once received by the *Border Connector* on the destination device, the *ComponentContent* message is forwarded to the local *Admin Component* which instantiates and attaches the received migrant component to the appropriate connectors in its local subsystem.

4.7 Disconnected Operation

Due to the nature of mobile devices, their network connections are intermittent, with periods of disconnection. In order to effectively support PitM applications in the face of connectivity losses, we should try to minimize the risks associated with disconnection. We propose maximizing the availability of an application during disconnection by migrating components from neighboring hosts to a local host before the disconnection occurs. The set of components to be migrated should be chosen such that it maximizes the autonomy of the local subsystem during disconnection, stays within the memory constraints imposed by the device, and can be migrated within the time remaining before disconnection occurs. This is a difficult problem in general [6], but we believe that we can leverage the architectural basis of the approach to PitM in solving it. We have done so in providing preliminary support for disconnected operation in our approach, as follows.

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 a given device if the component is migrated and (2) the *required memory* for loading the component. One possible approach to calculating the benefit of migration is proposed in [7]. The problem of selecting the best set of components for migration given a set of *n* candidate components is then stated as follows. Given the total available memory on the device, and benefit and required memory for each candidate component, select a subset of the candidate component set that maximizes the total benefit (as the sum of benefits of individual components). As stated, this problem represents a simplification of the actual problem: it assumes that the benefits of individual components are mutually exclusive, thus becoming an approximation in the case of highly-coupled components. At the same time, this simplification is a variant of the well studied *0-1-knapsack* problem, and can be solved in polynomial time using dynamic programming [1].

5 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) [2]. This paper has argued for solutions to 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).

As a "proof-of-concept" we have outlined a specific approach to PitM, which we have been developing over the past two years. The centerpiece of the proposed approach 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 handful of applications to date.

While our experience thus far has been very positive, a number of pertinent issues remain unexplored. These include ensuring trust in PitM applications, supporting configuration management of the many involved artifacts, automatically discovering the hardware devices and/or software components available on the network at a given time, and allowing architectures to (re)configure themselves "on-the-fly." A study of these and a number of related issues will frame our future work. We strongly believe that it should do the same for the software engineering community at large.

6 REFERENCES

1. T. H. Cormen, et al. Introduction to Algorithms. *MIT Press*, 2000.
2. F. DeRemer and H. Kron. Programming-in-the-Large Versus Programming-in-the-Small. *IEEE TSE*, June 1976.
3. R. Fielding. Architectural Styles and the Design of Network-Based Software Architectures. *Ph.D Thesis*, UCI, June 2000.
4. A. Fuggetta, G. P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, May 1998
5. 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.
6. J. S. Heidemann et al., Primarily Disconnected Operation: Experiences with Ficus. *Second Workshop on Management of Replicated Data*. IEEE, November 1992.
7. N. Medvidovic and M. Mikic-Rakic. Architectural Support for Programming-in-the-Many. Technical Report, USC-CSE-2001-506, University of Southern California, October 2001.
8. N. Mehta, N. Medvidovic, and S. Phadke. Towards a Taxonomy of Software Connectors. *ICSE 2000*, Limerick, June 2000
9. D.E. Perry, and A.L. Wolf. Foundations for the Study of Software Architectures. *Software Engineering Notes*, Oct. 1992.
10. M. Rakic, and N. Medvidovic, Increasing the Confidence in Off-the-Shelf Components: A Software Connector-Based Approach. *Symposium on Software Reusability*, May 2001.
11. M. Shaw, and D. Garlan. Software Architecture: Perspectives on an Emerging Discipline. *Prentice Hall*, 1996.
12. Sun Microsystems. K Virtual Machine (KVM). http://java.sun.com/products/kvm.
13. P.-W. Tam. U.S. Forces Pack Pocket Computers: Handhelds Track Troop Movements, Help Pinpoint Targets. *The Wall Street Journal*, Oct. 23, 2001. http://www.msnbc.com/news/646394.asp
14. R.N. Taylor, et al. A Component- and Message-Based Architectural Style for GUI Software. *IEEE TSE*, 22(6), June 1996.