Software architecture and mobility: A roadmap

Nenad Medvidovic, George Edwards *

Computer Science Department, University of Southern California, Los Angeles, CA 90089, USA

Article history:
Received 20 October 2009
Accepted 1 November 2009
Available online 14 November 2009

Keywords:
Software architecture
Mobility

Abstract
Modern software-intensive systems are characterized not only by the movement of data, as has been the case in traditional distributed systems, but also by the movement of users, devices, and code. Developing effective, efficient, and dependable systems in the mobile setting is challenging. Existing architectural principles need to be adapted and novel architectural paradigms devised. In this paper, we give an overview of the intersection of the areas of software architecture and mobility. We consider mobility from two related perspectives: (1) mobile software, which represents the computing functionality designed to migrate across hardware devices at runtime and execute on mobile hardware platforms, and (2) mobile systems, which are computing applications that include mobile software and hardware elements. We study the advances in both these areas, highlight representative existing solutions, and identify several remaining research challenges.

© 2009 Elsevier Inc. All rights reserved.

1. Introduction

Innovations in mobile computing technology are transforming how and why people use computers. First, from a human perspective, mobility allows people to access computing systems anytime, anywhere, which enables entirely new types of software applications. These mobile applications support a much wider range of activities than desktop applications and leverage information about the user’s environment to provide novel capabilities. Second, from a technology perspective, mobility shifts the global computing infrastructure from static, homogenous, powerful desktop computers to highly dynamic, heterogeneous, resource-constrained handheld and wearable computers. This new computing context demands entirely new software architectural paradigms that address the challenges of mobile software development, are specialized for the nature of mobile devices and wireless networks, and take advantage of the opportunities afforded by mobile systems. In this paper, we explain the core concepts and open research problems at the cross-section of the fields of software architecture and mobility.

Recent research has rapidly advanced the state-of-the-art in architectures for mobile software and systems. Researchers have established the technical basis for mobility by creating formal models of and engineering processes for mobile software. Mobile software is computing functionality designed to migrate across hardware devices at runtime and execute on mobile hardware platforms. The principles of software architecture are intrinsic to the development environments and run-time platforms that support mobile software models and processes. In parallel with (and supported by) these achievements, research activity that leverages this work to build mobile systems has expanded. Mobile systems are computing applications that include mobile software and hardware elements. These applications are characterized by customized software architectures that are designed for and intrinsically facilitate mobility.

Therefore, to structure and contextualize our discussion, we have organized the diverse aspects of software architecture and mobility touched on in this paper into two primary topics: mobile software and mobile systems. We begin with a brief summary of background information in Section 2. We then describe mobile software design, implementation, and deployment in Section 3, and highlight several classes of mobile systems that enable new computing applications in Section 4. For each topic discussed, we explain the key architectural concepts, illustrate those concepts with one or more representative examples from cutting-edge research, and outline areas for exploration. We conclude by offering our view on what we believe to be the most interesting open problems and promising research directions in software architecture and mobility.

2. Background and motivation

As software systems have become increasingly mobile, different solutions for supporting that mobility have emerged. Mascolo et al. (2002) have tried to understand the characteristics of mobile systems as a necessary first step to constructing them. They divide characteristics of mobile systems along three dimensions: device,
network connection, and execution context. An execution device can be fixed or mobile. A network connection can be permanent or intermittent. The execution context can be static or dynamic. A mobile system can have different combinations of these characteristics, but typically in this setting we are interested in software systems executing on mobile devices, with intermittent network connections, and with dynamic execution contexts. In other words, we are primarily focusing on a specific subset of the problem space, comprising

1. systems that are deployed on novel computing platforms that have emerged during the past decade – smart phones, mobile robots, motes, sensors, and so on;
2. novel usage scenarios frequently demanding computation anytime, anywhere – search-and-rescue, environment exploration, traffic management, medicine, assisted living, sensor networks, and so on, and thus
3. amplified software engineering challenges – distribution, decentralization, heterogeneity, resource constraints, context-awareness, real-time requirements, and so on.

Our objective is to look at mobile software systems specifically through the prism of software architecture (Taylor et al., 2009). In this section, we define and illustrate a number of relevant terms and concepts. Software architecture is defined as the set of principal design decisions $P$ about a software system. This means that a number of other system facets (e.g., low-level implementation support), which may otherwise be critical to supporting mobility, are outside the scope of this paper. Certain design-level solutions are better suited to mobile systems than others. Such solutions are usually embodied in architectural patterns and styles. A software architectural style is a named collection of architectural design decisions that (1) are applicable in a given development context, (2) constrain architectural design decisions that are specific to a particular system within that context, and (3) elicit beneficial qualities in each resulting system.

An issue relevant in the context of architecture-based software development that is particularly amplified in mobile systems is the relationship between a system’s “as-designed” and “as-implemented” architectures (Taylor et al., 2009). An as-designed architecture, also referred to as the system’s prescriptive architecture, is the set of architectural design decisions $P$ made at time $t$ that reflect the architects’ intent. The system’s as-implemented architecture, also referred to as its descriptive architecture, is the set of artifacts $A$ that realize the design decisions $P$. The artifact set $A$ could take the form of refinements of design decisions in a given modeling notation, models of employed architectural styles and patterns, existing off-the-shelf components, implementation frameworks and middleware, and so on. This dichotomy is at the heart of software architectural support for mobile systems. Consider the structural views of a simple mobile application’s prescriptive and descriptive architectures shown in Fig. 1. The application schedules and tracks a set of Vehicles as they route various cargo from a set of Delivery Ports to a set of Warehouses. The two architectures are clearly different in the connections between the Clock Conn connector on the one hand and the Vehicle component and Router Conn connector on the other.

Although the difference between the two architectures shown in Fig. 1 appears to be minor, the system’s engineers could make other important decisions that are based on that incorrect information. Such differences permeate many software systems. They result in two phenomena first identified by Perry and Wolf (1992) that reflect architectural degradation. The first phenomenon, architectural drift, is the introduction of principal design decisions into a system’s descriptive architecture that are not included in, encompassed by, or implied by the prescriptive architecture, but which do not actually violate any of the prescriptive architecture’s design decisions. The second phenomenon, architectural erosion, is the introduction of architectural design decisions into a system’s descriptive architecture that do violate its prescriptive architecture. For example, in the cargo routing application the engineers may expect each Vehicle’s clock to be synchronized with those of the Warehouses and Delivery Ports, as implied by the prescriptive architecture. Since this is not the case in the implemented system (i.e., in the system’s descriptive architecture), any design decisions

![Diagram](image-url)
based on this expectation will prove erroneous. Alternatively, they may expend system resources unnecessarily to maintain wireless connectivity between the Clock and the mobile Vehicles, even though the system’s developers may have found a more effective way of keeping track of time within the Vehicles, as reflected in the descriptive architecture.

Another notion relevant to software architectures of mobile systems is deployment. Deployment is the outcome of the activity of placing a system’s software components onto its hardware hosts. An example deployment for the cargo routing application is shown in Fig. 2. Deployment is an example architectural perspective, which is a non-empty set of architectural design decisions of a specific type. A relocation of a software system’s components across its hosts is component re-deployment or migration. Component migration is a primary way in which architectural mobility manifests itself. During migration, the overall application functionality may be preserved if, for example, a component is simply moved from a remote host to a host where it is frequently used. On the other hand, the system’s quality-of-service is likely to be affected when, for example, a previously remote interaction becomes local, and thus faster and more reliable.

3. Mobile software

A large number of techniques have been developed for addressing different facets of software system mobility. In this paper we are particularly interested in those that either take an explicit software architectural focus or have direct architectural ramifications. Since, as discussed in Section 2, a key concern is to prevent architectural degradation, i.e., to ensure the consistency between a mobile software system’s prescriptive and descriptive architectures, in this section we will overview techniques that address

- capturing intended architectural design decisions in a prescriptive architecture design,
- realizing those design decisions in a descriptive architecture via implementation, and
- injecting continuous changes into the system via system deployment and dynamic adaptation.

For each of the above three topics, we highlight one or more examples of state-of-the-art research and note open research challenges.

3.1. Mobile software design

The key software architectural abstractions are components, connectors, their interfaces, configurations, and constraints on system structure, behavior, composition, and interaction (Perry and Wolf, 1992; Taylor et al., 2009). Architectural styles are essentially named sets of such constraints: client–server, peer-to-peer (p2p), publish-subscribe (pub-sub), event notification, and so on. It is not readily obvious what level of support for mobility is yielded by, say, a given architectural style such as p2p or pub-sub. These traditional styles do provide many design guidelines that can prove to be useful in mobile systems. The guidelines include

- **component decoupling**, whereby the practical units of mobility are delimited – most architectural styles adhere to this guideline;
- **avoiding shared memory**, whereby potential side effects of computations are limited and interaction “back doors” eliminated – styles such as pub-sub and event notification adhere to this guideline;
- **insulating components from execution context**, whereby a single component may be effectively redeployed onto a number of sites – substrate independence is one of C2’s key design principles (Taylor et al., 1996);
- **implicit invocation**, whereby no component relies directly on the geographical presence or location of another – event notification and C2 are example styles that rely on event notifications;
- **asynchronous interaction**, whereby no component relies on the temporal presence of another – pub-sub and C2 make exclusive use of asynchronous interaction;
- **stateless components**, whereby the migration process and system consistency assurance are greatly simplified – clients in some client–server applications are examples of stateless components; and
- **stateless interactions**, whereby each interaction is self-contained and handled independently of the service requestor’s or provider’s location – the best known proponent of this guideline is REST (Fielding and Taylor, 2002; Taylor et al., 2009).

However, not all traditional architectural styles provide all of these guidelines simultaneously. This suggests that all of them are only partially suited for mobility and that that suitability will be context dependent. In this section we will instead focus on a set of architectural styles explicitly geared to supporting mobile software systems. We will then turn our attention to several research issues that remain open.
3.1.1. State-of-the-art

Carzaniga et al. (1997, 2007) have provided a foundational study of mobility “paradigms”, i.e., styles. They divide mobile systems into three categories: remote evaluation, code on demand, and mobile agent. They describe these three mobile systems paradigms in terms of a set of abstractions that are similar, but not identical, to those used in traditional architectural styles:

- **resource**, which is the data to be processed when delivering a system service,
- **know-how**, which is the code component (i.e., algorithms) whose execution delivers the required service,
- **computational component**, which is a processing thread,
- **interaction**, which is any event involving two or more components, and
- **site**, which is an execution environment that will host the data or run the code.

**Remote evaluation** – In this mobility paradigm, a computational component located on a client site has the know-how but does not have the resources to accomplish some needed functionality. The component sends the know-how to a computational component located on a remote site that has the required resources. Once the remote component completes the service request via the received know-how, it returns the results to the service client. Fig. 3 depicts this mobility paradigm schematically.

**Code on demand** – In this mobility paradigm, a computational component located on a client site has the resources but does not have the know-how to accomplish some needed functionality. The component requests the know-how from a computational component located on a remote site and uses that know-how to complete the service request. Fig. 4 depicts this mobility paradigm schematically.

**Mobile agent** – In this mobility paradigm, a computational component located on a client site has the know-how but does not have the resources to accomplish some needed functionality. This is similar to the remote evaluation scenario. However, the computational component itself (including the know-how and possibly intermediate results) migrates to the remote site, where it uses the local resources to complete the service request. Fig. 5 depicts this mobility paradigm schematically.

Ciancarini and Mascolo (1998) offer a slightly different, although related, characterization of architectural styles for mobile systems, in terms of the type of entities exchanged between sites:

**Data** – In this mobility style, data is sent from a requester to a supplier of a service. Even traditional architectural styles used in the construction of static distributed systems, such as client–server, engage in data mobility.

**Reference** – In this mobility style, neither data nor code are migrated, but only a reference to a resource. An example of such a reference is a URL denoting the location of some actual resource.

**Code and store** – In this mobility style, an executable component migrates from one site to another along with values of the variables to which the component refers.

**Code, store, and state** – In this mobility style, the code, store, and execution state of a computational component (i.e., thread) are migrated, allowing remote execution to continue from the point at which the component has been suspended at the local site. An example of this style is mobile agents.

**Closure** – In this mobility style, a mobile agent migrates to a remote site but maintains all links to the resources on which it depended on its previous site. In other words, the local resources automatically become remote, and some of the remote resources may conversely become local.

**Ambient** – In this mobility style, an entire execution environment required for providing a service would seamlessly move.

The seven styles are depicted schematically in Fig. 6.

---

**Fig. 3.** The remote evaluation mobility paradigm. The computational component in bold face is the one that executes the code. Components in italics are those that have been moved (adopted from Carzaniga et al. (1997)).

**Fig. 4.** The code on demand mobility paradigm. The computational component in bold face is the one that executes the code. Components in italics are those that have been moved (adopted from Carzaniga et al. (1997)).

**Fig. 5.** The mobile agent mobility paradigm. The computational component in bold face is the one that executes the code. Components in italics are those that have been moved (adopted from Carzaniga et al. (1997)).
In addition to the attempts to understand and categorize the software architectures of mobile code systems, several researchers have proposed specific architectural techniques with different facets of mobility. Thus, for example, Inverardi and Wolf (1995) showed that a software system can be modeled via a chemical abstract machine (CHAM), in which system components are represented as molecules. Highly dynamic system models could thus be produced via chemical solutions and reactions, and system mobility supported via molecular migration from one solution to another.

Another example approach focusing on mobility has been Fiadeiro et al.’s CommUnity (Fiadeiro and Lopes, 2003), which focuses on the role of software connectors in mobile systems’ architectures. Specifically, commUnity enables formal modeling of coordination primitives (i.e., connectors) for mobility.

A particularly interesting technique for supporting component mobility was proposed by Holder et al. (1999) in FarGo. FarGo models a software system as a collection of potentially mobile modules (complets), and provides dynamic layout constraints based on the specified relationships between two complet:

- **Link** mandates that a dependency between two complet A and B be maintained regardless of whether and where they are relocated. Thus, a local dependency may become a remote dependency.
- **Pull** is a uni-directional relationship which mandates that, whenever the “puller” complet A is relocated, the “pullee” complet B must follow.
- **Bi-directional pull** is a pull relationship in which each complet is both a puller and a pullee.
- **Duplicate** requires that a dependency between two complet be maintained such that if complet A migrates to a different host, a copy is made of complet B and deployed locally with A.
- **Finally, Stamp** requires that a migrating complet A find a local instance of the complet B’s type and connects to it.

### 3.1.2. Research challenges

The implications of mobility on architectural design have been studied by researchers in both the mobile systems and software architecture areas. However, to a significant degree each study reflects the dominant perspective of the researchers conducting it. This results in two still largely disjoint sets of architectural styles: traditional (e.g., client–server, p2p, pub-sub) and mobility (e.g., code on demand, remote evaluation, mobile agent). Clearly, some of the traditional styles are better suited to constructing mobile systems in specific contexts than others. This is primarily a function of the design heuristics they embody. However, this level of understanding is still very much lacking. It would be more meaningful to identify underlying “atomic” building blocks of styles that allow a uniform description of both categories of styles, and thereby a more meaningful characterization of their relationship.

Several other issues also require additional study. One of them is providing architecture-level support for migrating active components with active state. Note that, for example, Ciancarini and Mascolo’s ambient mobility style remains largely unsupported by existing solutions. Another issue is the potential overhead of explicit connectors. Existing mobility solutions typically “wither away” software connectors in implementation-level facilities such as procedure calls, RPC, and message-based communication facilities. It has been shown that explicit connectors yield many benefits in the resulting systems (Mehta et al., 2000; Taylor et al., 2009), but they do so at a cost. Understanding and quantifying that cost, especially in the context of mobile systems, remains an open issue. A related, and perhaps the biggest, challenge in supporting mobile systems from an architectural perspective is ensuring the consistency between the architectural model and the system’s implementation. The remainder of Section 3 will discuss this issue.

### 3.2. Mobile software implementation

Once a software system’s architectural design is reasonably complete, it is supposed to be realized faithfully in the system’s implementation. The objective of implementing an architecture is simple: realize the principal design decisions and achieve the key intended non-functional system qualities as a result. However, the system’s implementation frequently departs from the intended architecture and many, if not most, existing software systems suffer from architectural drift and erosion (recall Section 2).
Maintaining the consistency between a system’s prescriptive and descriptive architectures is particularly important in mobile systems, which depend on and interact with many facets of their external environment, and in which violations of architectural intent may have consequences that go far beyond the system’s inability to produce its intended behavior. For this reason, support for effective implementation of mobile systems is critical. This involves the provision of appropriate implementation-level abstractions for capturing the system structure and behavior that were captured in the architectural design, and an appropriate infrastructure for ensuring the desired system properties and the requisite facets of mobility.

3.2.1. State-of-the-art

As with any large, distributed, long-lived systems, mobile systems are most effectively developed with the help of middleware facilities. These facilities rest on top of operating systems and programming language runtimes, and provide the needed implementation-level abstractions and execution support for the desired services: distribution, resource management, service discovery, security, run-time adaptation, and so on. Many different middleware solutions have emerged over the past couple of decades with the proliferation of distributed computing. An increasing number of middleware solutions have been specifically geared to mobile systems.

Mascolo et al. (2002) have attempted to classify mobile middleware platforms. They divide these platforms based on their particular foci, as follows.

Traditional middleware – These are typically object-oriented middleware platforms adapted to mobile settings, in order to render mobile devices interoperable with fixed networks. Examples of traditional mobile middleware platforms applied in mobile computing are ALICE (Haahr et al., 1999), DOLMEN (Reynolds and Brangeon, 1996), and Mobile DCE (Schill et al., 1995).

Context-aware middleware – These middleware platforms specifically target the heterogeneity of hosts and networks, limited resources, and variations in the user environments in order to improve an application’s behavior. Examples of context-aware middleware platforms are UIC (Ubi-core, 2001), Gaia (Cerqueira et al., 2001), and Nexus (Fritsch et al., 2000).

Data sharing middleware – These middleware platforms in particular target disconnected operation, where the availability of data is ensured regardless of the network’s performance and reliability. Examples of data sharing middleware platforms are Coda (Satyanarayanan et al., 1990), Odyssey (Satyanarayanan, 1996), and XMIDDLE (Mascolo et al., 2002).

Tuple spaces middleware – Certain middleware platforms allow system components to communicate in a manner that is decoupled in both time and space. This is achieved via tuple spaces, which are globally shared memory spaces in which data structures are vectors of typed values. Examples of tuple space-based middleware platforms are Lime (Murphy et al., 2001), Tspaces (Wyckoff et al., 1998), and L2imbo (Davies et al., 1998).

While these platforms support the construction of different classes of and provide support for different aspects of mobile systems, the question of the extent to which any of them supports the architectural integrity of those systems remains. In fact, it can be argued that they do not help to capture architectural notions or enforce architectural design decisions pertaining to system computation, data, interaction, and styles. Thus, for example,

- software components can be realized in a number of different ways, including object classes, packages, modules;
- software connectors may not exist explicitly in system implementations (other than as procedure calls), or they may be embodied in facilities such as software buses and messaging services;
- interfaces are typically captured as the system modules’ API signatures; however, other aspects of interfaces, such as constraints on their use, are not readily transferred to the implementation;
- architectural configurations are typically hidden in system implementations due to the typical realization of interfaces (in particular, the required interfaces, which are distributed throughout the code in the form of method calls), use of function pointers, dynamic class loading and binding, and reflection;
- design rationale is not captured in the system explicitly, but rather is implicit in the comments embedded in the code as well as system documentation;
- behavior is embodied throughout the system in the form of algorithms, but a critical question is how an engineer is to translate formal notations used at the level of architecture, such as FSP (Magee et al., 1999), StateCharts (Harel, 1987), Z (Spivey, 1998), and so on into code;
- non-functional properties are a direct function of the implementation, but are not in any way visible in the implementation; rather, they are assessed indirectly and often partially via design rationale, code inspections, testing, user studies, and so on.

In order to address this discrepancy between the architectural intent and design on the one hand and system implementation on the other hand, another category of middleware platform has been recently proposed. This category has grown out of early work on architectural implementation frameworks (Medvidovic, 2002; Taylor et al., 2009). In this context we refer to it as architectural middleware:

Architectural middleware – A class of mobile middleware platforms focuses primarily on providing implementation-level constructs for the realization of architectural concepts and abstractions. In these platforms, developers explicitly declare software components, connectors, communication ports and events, and may even be able to specify named sets of implementation constraints that correspond to specific architectural styles. Example architectural middleware solutions are Aura (Sousa and Garlan, 2002) and Prism-MW (Malek et al., 2005a). As an illustration, we now elaborate on Prism-MW.

Prism-MW targets architecture-centric software development of highly distributed, resource constrained, embedded, and mobile systems. It has been implemented in several programming languages, including Java and C++, and has been applied across a number of application domains, including handheld computing (Medvidovic et al., 2003), wireless sensor networks (Malek et al., 2007), mobile grids (Mittmann et al., 2005), and mobile robotics (Malek et al., 2010). It provides basic architectural concepts as “core”, first-class middleware constructs realized via specialized programming language-level classes (see shaded rectangles in the right-hand diagram of Fig. 7). This allows engineers to “program” a system by declaring components, connectors, ports, events, explicit configurations, and even to restrict system designs to particular architectural styles.

Support for advanced facilities (e.g., real-time processing of events, data compression, or event delivery guarantees) is provided via extensions to the middleware core (see clear rectangles in the right-hand diagram of Fig. 7). These facilities are implemented on top of a virtual machine, which abstracts system resources in heterogeneous environments (see left-hand diagram in Fig. 7). Finally, the core middleware facilities are used to construct higher-order services targeted at mobile environments, such as resource discovery, system monitoring, dynamic adaptation, and deployment (further discussed in Section 3.3).
3.2.2. Research challenges

Selecting an appropriate middleware solution may help to control architectural degradation (recall Section 2). However, since the middleware is targeted at supporting any system in a given domain, it cannot by itself guarantee architectural integrity. At best, it provides a sort of “gentle persuasion” for engineers to respect the architecture. At the same time, the highly dynamic nature of mobile systems renders them particularly susceptible to architectural drift and erosion. Adding support for architectural styles only partially remedies the problem since many designs are possible within a single style. Relying on generative techniques will, in principle, result in systems that are architecturally “correct by construction”. However, any adaptation will require at least partial system re-generation and re-deployment, which is likely to be prohibitively expensive, especially in highly distributed and resource constrained execution settings.

One promising approach is to rely on continuous monitoring and analysis. It is possible to implement relatively inexpensive system monitoring facilities. Using the monitoring data, analysis can take place both locally, for simpler architectural consistency checks, and off-board, on more capacious hosts, for more sophisticated processing. In cases where discrepancies are discovered between the prescriptive and descriptive architectures, the system may need to be dynamically adapted to bring the two in line with one another. However, this may result in violations of desired behaviors and service quality levels. In order to ensure that, ideally an architectural model would first be constructed and/or modified, and then analyzed, before any system modification is allowed. This poses two significant additional research challenges: (1) architectural modeling approaches are currently not expressive enough to represent such concerns and (2) the time required to collect and interpret monitoring data, to cross-correlate them with desired system behaviors and quality levels, and finally to effect the necessary changes may be unacceptably long.

3.3. Mobile software deployment and dynamic adaptation

Deployment of software components onto hardware hosts is an instance of stateless or weak mobility (Fuggetta et al., 1998): code and data are migrated from one host to another, but not active system state. The nature of software deployment has changed in recent years. Traditionally, deployment has involved installation of an entire software system, or installation of specific subsystems in a manner opaque to the user, from CD ROMs or websites. Today, deployment is increasingly becoming a process of cooperation and negotiation between service producers and consumers, and system updates are transparent to the user (Taylor et al., 2009).

Once a system has been deployed and is in operation, subsequent changes to the system fall under the general rubric of adaptation. If the system cannot be brought down for updates, as is the case with many mobile systems, the adaptation will need to be dynamic: the system will need to continue delivering its services, perhaps in a degraded mode, during the adaptation process; furthermore, system users should be unaware of the adaptation to the greatest extent possible.

3.3.1. State-of-the-art

Carzaniga and colleagues (1998) have provided a useful categorization of deployment activities. According to them, deployment involves planning, modeling, analysis, and implementation.

Deployment planning consists of finding the deployment architecture (i.e., the mapping of software components and connectors to the hardware hosts) that achieves the desired system objectives.

Deployment modeling involves capturing, at the least, the details of the system’s hardware and software elements, their mappings, any constraints, relevant quality-of-service dimensions (e.g., security or reliability), and system users.

Deployment analysis ensures that the resulting system will have required properties, satisfy constraints, and deliver its services at the desired quality level.

Deployment implementation comprises activities required to effect the deployment plans and models: system release, installation, activation, updates, and so on.

Considering system deployment requires one to extend traditional notions of what is considered to be a part of a software architecture. In addition to components, connectors, events, etc., deployment also requires a suitable representation of hardware, service qualities, system users, and possibly other concerns. This is why extensible architecture description languages (ADLs) (Medvidovic and Taylor, 2000), such as xADL (Dashofy et al., 2002), present the most promising modeling and analysis platform for supporting deployment. In fact, xADL provides the structural core of an architecture modeling and analysis approach that natively incorporates system deployment characteristics, called XTEAM (Edwards et al., 2007). Fig. 8 shows an example XTEAM deployment model and the results of several analyses accomplished by simulating the model.
If the analysis results predict unsatisfactory quality-of-service patterns, the architect may employ a deployment improvement tool such as DeSi (Malek et al., 2005b), which treats deployment as a multidimensional optimization problem and tries to optimize an objective function (e.g., minimize the system’s battery power consumption as well as latency for a given service) by altering the mapping of components to hosts. Once an improved deployment is found, or some other trigger results in a necessary change to the architecture, the system must be adapted dynamically. From an architectural perspective, dynamic adaptation can be thought of as consisting of one or more of the following operations on system components: addition, removal, replacement, and reconnection. As a result of component relocation, system connectors may also need to be adapted. For example, a connector that supported only local interactions may need to support interactions across the network if a local component has been redeployed remotely.

A model of architecture-based dynamic adaptation proposed by Oreizy and colleagues (1999) is shown in Fig. 9. The model reflects two key aspects of architecture-based dynamic adaptation. First, dynamic adaptation must be properly planned, analyzed, and effected (as depicted in the top portion of the diagram). Second, the system’s architecture should always be the driver of dynamic adaptation, i.e., the implementation should never be modified without understanding the architectural implications of that modification (as indicated by the counter-clockwise flow of activities in the bottom portion of the diagram).

3.3.2. Research challenges

Deploying and dynamically adapting a system requires that its components be reasonably loosely coupled. One way of reducing component coupling is by supporting explicit connectors, not only at the level of architectural design (which is what many approaches do), but also in the implementation (which is much less prevalent). Furthermore, it must be possible to render the components quiescent, i.e., to ensure that it has finished its current processing and to disengage it temporarily from any further interaction with the rest of the system (Kramer and Magee, 1990). Moving such a component to a new location becomes more challenging if the component has active state, and especially if it shares that state with other components; this is what Fuggetta et al. (1998) refer to as strong mobility.

Another set of challenges inherent in dynamically adapting systems is the degradation in performance as well as the temporary loss of available services. Since the service-providing components...
are constantly moving, another critical issue is the provision of efficient and effective service registration and discovery utilities. Furthermore, the migration process itself may be prone to failure, so that a system ends up in an inconsistent state. Precisely understanding, or even quantifying, these issues is challenging. For example, Fig. 10 depicts what may happen with a system's quality during run-time adaptation: the “dips” in the quality levels are difficult to estimate accurately, both in terms of depth and duration. Estimating their impact on users remains an even greater challenge.

4. Mobile systems

Mobile computing devices are exploding in popularity as a handheld platform for traditional desktop applications (such as email and web browsing) and as an enabler of entirely new classes of applications (such as context-aware and activity-based software). In both cases, novel software architectures that are specialized for the domain of mobile systems are required because most of the assumptions, constraints, and goals of software development are radically shifted in the mobile setting. The principles of software architecture have played, and will continue to play, a central role in ensuring that software systems developed for mobile computing platforms meet requirements for quality dimensions that demand increased emphasis in mobile environments, including:

Scalability – The number of mobile devices in use worldwide dwarfs the number of other types of computing platforms; according to research by Nokia, 3.3 billion people worldwide have mobile phones (Chipchase, 2007). The potential for collaborative applications that harness the aggregated computation and communication power of these devices is enormous. Decentralized and autonomic software architectures – such as those employed by peer-to-peer and biologically-inspired systems – are the most promising way to achieve seamless scalability to billions of devices.

Heterogeneity – Mobile platforms comprise an extreme variety of hardware devices, operating systems, applications, and user interfaces. Systems that are optimized to run on a single platform will likely perform very poorly on others. Software architectures that contain pluggable and adaptable components and interfaces ensure the portability of applications across the heterogeneous mobile computing infrastructure.

Trustworthiness – The vulnerability of mobile devices and wireless networks to intrusion dictates that privacy and security are fundamental to mobile systems. Many envisioned “killer apps” for mobile systems, like location-aware search and opportunistic social networking, require access to sensitive data. Software architectures for mobile systems must provide structural guarantees of trustworthiness for users to believe their data is protected.

The properties discussed above are examples, and are far from a complete list. Requirements for system quality in more traditional areas, like efficiency and dependability, remain relevant.

In this section, we describe the processes and features of software architectures for systems that execute in mobile environments. We highlight three particularly compelling areas of research: mobile device management, context-aware mobile applications, and mobile robotics. For each area, we mention representative examples of state-of-the-art research and outline future research challenges.

4.1. Mobile device management

Large organizations and enterprises are equipping their employees with mobile devices to facilitate communication (e.g., email on-the-go), improve information gathering (e.g., in-person form data entry), enable resource management (e.g., tracking employee locations), and increase productivity (e.g., continuous access to applications). Mobile workforces are being used in the insurance industry for on-site claims handling, in the pharmaceutical industry for on-site sales, and in the consulting industry for on-site access to backend support applications.

To provision mobile devices for thousands of users, organizations must configure each device with user-specific software and settings and establish mechanisms that allow those users to access enterprise services and databases, while ensuring compliance with security and privacy policies. Most organizations rely on a team of administrators to provision, manage, and troubleshoot mobile devices. The number of mobile users per administrator depends on the nature of the enterprise, but it is generally much lower than the number of PC users per administrator because existing tools for mobile device management only access and configure one device at a time, and they force administrators to specify each configuration manually. As a result, the cost of equipping employees with mobile devices is high.

New, experimental mobile device management systems simplify the provisioning, maintenance, and evolution of mobile services and integrate mobile technology with business processes. These technologies leverage policy-driven device configuration and automated device provisioning processes to reduce the burden on system administrators and allow them to manage larger numbers of mobile users. Cutting-edge mobile device management systems include capabilities for transparent user authentication, security policy enforcement, and automatic software deployment, installation, and upgrade.

Software architectures for mobile device management are centered on the requirement for integration of heterogeneous platforms and services. Mobile device management systems must interact with (1) enterprise databases that contain user access credentials, email, and other data, (2) software repositories where application installation packages are stored, (3) help and troubleshooting systems that detect problems and enact fixes, and (4) device interfacing products that implement the low-level protocols required to communicate with mobile devices. The software architecture of a mobile device management system also commonly includes components and user interfaces for defining, storing and executing policies and processes.
4.1.1. State-of-the-art

The BlueStar system, developed by Matrianni et al. (2008), automates the management and delivery of enterprise mobile services. Automation is achieved through standards-based process and policy definitions for service deployment, dependencies, and content. Processes capture the procedures and workflows required to provision a mobile device for a user, install a mobile application, change the configuration of a mobile service, and so on. A process consists of a set of tasks, each of which may be a sub-process, an invocation of a web service, or a custom code fragment. BlueStar selects process definitions to be instantiated in two ways: either through an interaction with a system administrator using the BlueStar console or programmatically, through process invocation as a step in another workflow. Policies specify operational requirements that ensure compliance with business functions and objectives, such as corporate security and software versioning policies. Policies are used to derive a profile for each managed mobile device, which lists device- and user-specific constraints on communications are used to derive a profile for each managed mobile device, which lists device- and user-specific constraints on communication, software, and settings.

An overview of the components and structure of BlueStar is shown in Fig. 11. BlueStar integrates management tools that allow administrators to define, store, and execute processes and policies with external enterprise applications and databases. These BlueStar process and policy management tools comprise the top half of the diagram. Processes are customized to an organization's needs through policies saved in the policy store and executed in the policy engine. BlueStar utilizes a WS-BPEL-compliant process design tool, process builder, and process engine. The use of standards-based process tools permits integration with external components, as described below.

External components, shown on the bottom half of Fig. 11, are integrated with BlueStar using a plug-in API. BlueStar interacts with the following external components:

- A device control product, which handles all interaction with mobile devices such as data exchange and software installation.
- A help desk, which tracks trouble tickets and assigns them to administrators for resolution.
- A user directory, which stores user authorization and entitlement information, and
- A software repository, which stores installation packages to be deployed to mobile devices.

The implementations and interfaces of these components vary widely among organizations; for example, the user directory could be stored in an Oracle, IBM or Microsoft database. BlueStar integrates these heterogeneous external components using web service-based interfaces and custom-built adapters, where needed.

4.1.2. Research challenges

The capabilities of mobile devices are evolving rapidly. Furthermore, numerous entities, such as Research In Motion (BlackBerry), Microsoft, and Apple, are producing competing operating systems and platforms for mobile devices. Mobile device management technologies must be agile and flexible to stay ahead of the curve and easily accommodate new, unforeseen platforms and services. Otherwise, device platforms may become tightly integrated with proprietary backend servers and databases, making the construction of comprehensive mobile device management systems impractical.

4.2. Context-aware mobile applications

Mobile devices are more resource-constrained than PCs and servers in some ways (e.g., in terms of processing power), but are more powerful in other ways. Specifically, mobile devices have the ability to leverage context to provide a richer user experience. Context is information about a user's current state and history, such as the user's location and movement, activities, preferences, and past behavior. Context-aware applications are possible because of the integration of multiple context-sensing devices on mobile platforms like smartphones. Today's smartphones include video cameras, audio recorders, GPS receivers, and accelerometers. Furthermore, context-awareness is fundamentally intertwined with mobility because it is most useful when a user's situation is frequently changing.

Context-aware software services enable a number of potentially useful and powerful applications. Context-aware search applications utilize a user's location to narrow down and prioritize search query results. Context-aware social networking applications recognize possibilities for social interaction based on physical proximity and common interests. Context-aware collaborative work applications direct users toward solutions and goals based on current activities and business processes.

Software architectures for context-aware mobile applications are commonly based on event-based, publish-subscribe architectural styles and middleware infrastructures. Context publishers make information about changes in the user's environment, behavior, and goals available to interested applications and services. Context subscribers interpret and respond to the published information to tailor and customize their functionality and interfaces. The use of standardized publish-subscribe services and event buses permits the easy addition, upgrade, or removal of context-providing or context-consuming components.

4.2.1. State-of-the-art

The Context Toolkit, built by Dey et al. (2001), is a development infrastructure that adopts a component-based, architectural approach to the construction of context-aware mobile applications. The Context Toolkit provides specialized design and implementation constructs that directly address the unique requirements and constraints of context-aware applications. Specifically, the

[Fig. 11. Overview of the BlueStar mobile device management system (adopted from Matrianni et al. (2008)).]
Context Toolkit is oriented to provide elegant solutions to the following concerns:

Separation of concerns – To build adaptable, portable, reusable, and maintainable context-aware mobile applications, it is essential to effectively decouple components that obtain a user’s context from those that utilize the user’s context. Components that acquire context are generally sensor controllers that implement low-level interaction with hardware devices like cameras and GPS receivers. Coupling these low-level details with application logic and user interfaces that use context results in brittle systems.

Context interpretation – Context exists at a number of different levels of abstraction. At the lowest level, context is represented by raw sensor data, while the highest level, context is organized into ontologies with complex semantics. Bridging the gap between these two extremes are multiple layers of context interpretation and aggregation. Context information must be accessible by applications at each level of abstraction, as the requirements for such information are highly application-specific.

Transparent, distributed communications – The transport and delivery of context from context-sensing components to context-using components must account for the fact that context is acquired from a wide variety of sources. In many cases, context-sensing devices will be distributed across dynamic and heterogeneous networks. Therefore, a context-aware application framework must provide scalable and transparent communication of context data.

Constant availability of context acquisition – The mobility of people and devices necessitates that context-aware applications be capable of utilizing whatever resources are available nearby to determine context. This implies that context-providing devices and components cannot (and should not) anticipate or know when or what applications are currently leveraging the information they are providing. Clearly, the acquisition of context must be constantly available to meet these conditions.

Context storage – As noted above, history and past behavior are an important element of context. A record of previously gathered context information is necessary to make accurate and realistic predictions about current goals and future activities. Consequently, context must be continually stored for possible queries and analysis.

Resource discovery – As the location and requirements of context-aware applications change over time, they must be able to dynamically locate and access context providers at runtime. This capability includes the handling of queries that specify context requirements semantically. Query results should include not only the location of relevant resources but also information about the protocols and interfaces required to access those resources.

The Context Toolkit implements the following constructs for assembling context-aware mobile applications:

- Widget, which implements the widget abstraction,
- Server, which aggregates context,
- Interpreter, which interprets context,
- Service, which performs actions, and
- Discoverer, which supports resource discovery.

Interactions and data exchange among these elements are depicted graphically in Fig. 12. Instances of these components are created and deployed independently, and they manage their own state and threads of execution. The underlying communication infrastructure, which utilizes HTTP and XML, abstracts host and network boundaries between components to enable transparent distribution.

4.2.2. Research challenges

Although numerous prototypes of context-aware applications have been developed and demonstrated in the lab setting, relatively few have yet made the transition to production and wide distribution. Context-aware applications that have made this transition often employ location information exclusively, and do not utilize the much broader definition of context described in this section. This lack of adoption is likely due to both a lack of conceptual tools and processes for designing and modeling context-aware architectures, as well as a lack of robust and scalable implementation infrastructures for implementing those architectures.

First, context-aware application designers need specialized modeling languages for specifying exactly what context they depend on and how that context will be processed to achieve useful and practical solutions. These languages can benefit from reuse of the paradigms and concepts of software architectures in general, but they must also incorporate domain-specific concerns and requirements. Second, context-aware application developers need specialized middleware platforms and programming languages for supporting highly distributed and dynamic sensor-based computing environments. Again, the approaches that have worked well in more traditional distributed computing systems, such as transparency of location, programming language, and run-time platforms, must be applied with domain-specific enhancements and modifications.

4.3. Mobile robotics systems

Mobile robots have a wide range of uses. Mobile robots that perform environment exploration are used by scientists (e.g., underwater and interplanetary explorers), civilian groups (e.g., search-and-rescue teams), and military organizations (e.g., surveillance units). Mobile robots that perform navigation are used for transport of goods and materials (e.g., self-driving supply convoys) and intelligent traffic management (e.g., autonomous cars). Mobile robots that manipulate objects are used to support day-to-day activities (e.g., assisted living for the elderly and handicapped) and specialized functions (e.g., industrial mining operations).

Autonomous mobile robots carry out high-level goals by sensing their environment, planning responsive actions, and executing those plans. When situations arise that the robot is not equipped to handle, it must adapt by modifying or adding new behaviors. Since mobile robot behavior is implemented in software, adapting mobile robot behavior requires adapting the software that controls the robot.

Software architectures for adaptive mobile robotics generally adopt a layered approach. At each layer, components are responsible for creating and executing plans that are beyond the capability of components at the layers below. Thus, each additional layer of

![Fig. 12. Interactions among components of the Context Toolkit (adopted from Dey et al. (2001)).](Image)
the stack implements successively more complex behaviors. This often includes management of dynamic adaptations and new configurations of the “building block” behaviors implemented at the layers below.

Dynamic self-management and self-adaptation of this type requires an infrastructure that permits seamless addition, removal, and substitution of components. Software architecture-based middleware platforms that structure application functionality in independently-deployable components with precisely specified interfaces and dependencies provide such an infrastructure. These platforms are ideally suited for adaptive robotics applications because they enforce architectural constraints that ensure adaptation occurs in controlled and expected ways, and define architectural constructs as the units on which adaptation mechanisms operate.

4.3.1. State-of-the-art

Approaches proposed by Georgas and Taylor (2008) and Kramer and Magee (2007) recognize and argue for a central role for software architecture in constructing adaptive mobile robotics systems.

Georgas et al. (2008) defined the RAS architectural style depicted in Fig. 13. Architectures that conform to the RAS style are component-based and explicitly layered. Communication in RAS architectures is connector-based and event-based. RAS architectures are dynamic, as they may be evolved at run-time via modifications to an architectural model. Run-time adaptations are realized through a policy-based approach to architectural adaption management (PBAAM) infrastructure, which includes an Architectural Model Manager, Architectural Adaptation Manager, and Runtime Manager.

Kramer and Magee (2007) proposed the Three Layer Architecture Model for adaptive mobile robots shown in Fig. 14. This model, like the RAS style, is layered. The bottommost layer is the component control layer, where components that implement application functionality reside. The middle layer is the change management layer, which is responsible for monitoring and adapting components at the layer below. Components at the change management layer are only capable of executing pre-specified actions in response to specific conditions. For example, change management components may create components and change component connections in response to changes in the system state, such as possible anticipated failures. The topmost layer is the goal management layer, which is responsible for achieving high-level goals. Components at this layer handle changes in the system goals or situations that change management components are not equipped to handle. Usually, this involves adapting the set of components at the change management layer.

4.3.2. Research challenges

The field of adaptive mobile robotics is relatively new, so a number of challenging open problems remain largely unexplored. First, due to the embedded nature of robotics systems and their continual interaction with the real-world environment, many robotics applications demand fast and predictable performance from software controllers. However, permitting dynamic self-adaptation can reduce performance in unforeseen ways. Mechanisms that ensure that adaptation only takes place when the robot can afford slower or highly variable performance are required.

Second, mobile robots often work collaboratively to achieve their goals. In these settings, localizing and encapsulating all adaptation decisions within each individual robot likely cannot achieve optimal solutions. Instead, adaptation must be planned and executed on a global basis, with robots communicating and coordinating their actions to complement and enhance each other. Research on both the theoretical underpinnings of and practical solutions for global, cooperative adaptation is needed to make this a reality.

Third, models and languages for representing adaptation policies and constraints are not yet mature. Ultimately, such models should include precisely-defined, standardized semantics to permit interchangeability among vendor components. However, we must first determine exactly what concerns adaption policies must address and how they can be specific in an intuitive, mechanically-leverageable way.

5. Conclusions and future trends

The trend of software-intensive systems becoming increasingly mobile and ubiquitous will only accelerate in the future. Developing effective architectures for designing, implementing, verifying, deploying, and evolving such systems will become comparatively even more critical than today. Emerging novel computing metaphors for the mobile setting will offer great opportunities for developers and users to push mobility and ubiquity beyond the limits achievable today. At the same time, these metaphors will demand that engineers think “outside the box” and develop completely novel architectures. Two such new metaphors are cloud computing and biologically-inspired computing. We conclude the

![Fig. 13. The RAS architectural style (adopted from Georgas and Taylor (2008)).](image-url)

![Fig. 14. The three layer architecture model for self-management (adopted from Kramer and Magee (2007)).](image-url)
paper by briefly outlining the opportunities and challenges they present, and in particular focus on their potential synergistic relationship.

People and organizations are increasingly utilizing cloud computing to access and deliver software services. Cloud computing allows service clients to have a single, location- and machine-independent view of their applications and data, while freeing them from having to understand or manage the technology that implements those services. Naturally, as people perform more computing tasks on mobile devices, the incentive and benefits of expanding cloud computing to mobile devices will increase. This will likely include both (1) access to standard cloud computing services via mobile devices, and (2) new cloud computing infrastructures implemented on mobile devices. The former is a straightforward extension of current cloud computing approaches, and is already available to consumers in several (limited) forms. The latter requires entirely new cloud computing platforms that allow services to be transparently deployed to, and execute on, highly distributed, unreliable, resource-constrained mobile networks.

One possible enabling technology for mobile cloud computing is the organic computational grid. Like traditional computational grids, organic grids pool hardware and software resources to deliver computation as a standardized service. However, organic grids distribute computation in a completely decentralized and autonomous manner by employing biologically-inspired processes and structures. Organic grids use self-organizing and self-adapting behaviors from nature to permit virtually unlimited scalability and elegantly deal with malicious and unreliable nodes. The computational load on each node in an organic grid can be extremely small (on the order of complexity of simple two-input binary gates, for some systems), but the collective power of the network, when enough nodes participate, is tremendous. Consequently, organic grids will be ideal for deployment on networks consisting of millions of mobile devices, which comprise a great majority of the computing devices available today.

References


Haahr, M., Cunningham, R., Cahill, V., 1999. Supporting CORBA applications in a mobile environment (ALICE). In: Fifth Int. Conf. on Mobile Computing and Networking (MobiCom), Washington, DC.


Nenad Medvidović is an Associate Professor in the Computer Science Department and the Director of the Center for Systems and Software Engineering at the University of Southern California. He is the Program Co-Chair of the 2011 International Conference on Software Engineering (ICSE). He is a recipient of the National Science Foundation CAREER and IBM Real-Time Innovation Awards. His ICSE 1998 paper was named that conference’s Most Influential Paper. Medvidović’s research interests are in the area of software architecture, with a focus on distributed, mobile, and embedded computing environments. He is a co-author of a new textbook on software architectures.

George Edwards is a Ph.D. candidate in the Computer Science Department at the University of Southern California, where he is a USC Viterbi School of Engineering Fellow, an Annenberg Graduate Fellow, and a member of the Software Architecture Research Group in the Center for Systems and Software Engineering. Edwards has authored over twenty scholarly journal articles and conference papers on varied topics related to software engineering and distributed systems, and in 2008, he received the USC Computer Science Outstanding Student Research Award.