The SATIN Component System—A Metamodel for Engineering Adaptable Mobile Systems

Stefanos Zachariadis, Cecilia Mascolo, and Wolfgang Emmerich, *Member*, *IEEE Computer Society*

Abstract—Mobile computing devices, such as personal digital assistants and mobile phones, are becoming increasingly popular, smaller, and more capable. We argue that mobile systems should be able to adapt to changing requirements and execution environments. Adaptation requires the ability to reconfigure the deployed code base on a mobile device. Such reconfiguration is considerably simplified if mobile applications are component-oriented rather than monolithic blocks of code. We present the SATIN (System Adaptation Targeting Integrated Networks) component metamodel, a lightweight local component metamodel that offers the flexible use of logical mobility primitives to reconfigure the software system by dynamically transferring code. The metamodel is implemented in the SATIN middleware system, a component-based mobile computing middleware that uses the mobility primitives defined in the metamodel to reconfigure both itself and applications that it hosts. We demonstrate the suitability of SATIN in terms of lightweightedness, flexibility, and reusability for the creation of adaptable mobile systems by using it to implement, port, and evaluate a number of existing and new applications, including an active network platform developed for satellite communication at the European Space Agency. These applications exhibit different aspects of adaptation and demonstrate the flexibility of the approach and the advantages gained.

Index Terms—Distributed objects, components, containers, mobile systems, middleware, pervasive computing, mobile code.

1 Introduction

Mobile devices, such as mobile phones, Personal Digital Assistants (PDAs), MP3 players, and palmtop and laptop computers, are becoming increasingly popular. This has lead to a further and rapid decentralization of computing, with devices becoming more capable, cheaper, more mobile, and even fashionable personal items. The recent advances in wireless networking (UMTS, Bluetooth, IEEE 802.11, etc.), combined with the popularity of mobile devices, allow users to carry sophisticated computing environments that facilitate access to local and remote information *on the move*.

In traditional computing systems, application developers can often assume that their software will be executed by a powerful machine that is always connected to a centralized network using a high bandwidth link. Similarly, traditional distributed systems are usually composed of powerful hardware, interconnected using high bandwidth network links over a fixed topology. Mobility breaks this *static* model as mobile devices are considerably less powerful and stable in terms of computational resources available, such as CPU speed, battery, network bandwidth, and volatile and persistent memory. Moreover, the mobile network topology is considerably more *dynamic*, as mobile devices may come and go freely. Mobile devices frequently aggregate dynamically into various hybrid, independent, and even incompatible (Infrared and Bluetooth, for example) networks. Although devices are becoming increasingly more capable, they will, for the foreseeable future, lag behind their fixed counterparts with respect to their available resources, particularly in power provision and network connectivity, which is often fluctuating in bandwidth and intermittent.

Mobile computing systems may also be *highly heterogeneous*. This heterogeneity occurs in software and hardware. Mobile devices host a large number of different applications. We refer to these mobile devices as *hosts* in the remainder of this article. Mobile hosts use different operating systems and middleware and often have more than one network interface. Moreover, mobile applications are exposed to a *highly dynamic environment* in terms of their local and remote context. The current state of practice in engineering software for mobile systems offers little flexibility to accommodate such heterogeneity and variation. Application developers have to decide at design time what possible uses their applications may have; the applications do not change or adapt once they are deployed on a mobile host. Mobile applications are currently developed in monolithic architectures, which are more suitable to a fixed execution context rather than a dynamic, mobile one. Interaction with their environment and peers is either not considered or is very constrained. Moreover, mobile applications are usually monolithic [1], composed of a single large file making little use of libraries; thus, maintenance and updating of an application is difficult.

We argue that more flexible solutions are required that empower systems to automatically adapt to changes in the environment and to users' needs. Power [2] postulated more than a decade ago that it is common in distributed systems that "*when something unanticipated happens in the environment, such as changing user requirements and/or resources, the goals may appear to change. When this occurs the*

• *The authors are with the Department of Computer Science, University College London, Gower Street, WC1E 6BT, London, UK.*
E-mail: {s.zachariadis, c.mascolo, w. emmerich}@cs.ucl.ac.uk.

Manuscript received 18 Aug. 2005; revised 13 Aug. 2006; accepted 13 Aug. 2006; published online 6 Nov. 2006.
Recommended for acceptance by R. Schlichting.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0221-0805.

system lends itself to the biological metaphor in that the system entities and their relationships need to self-organize in order to accommodate the new requirements." Along those lines, this work considers a *self-organizing* or *adaptive* system as a system that is able to mutate to accommodate changes to its requirements. As a highly dynamic system, a mobile system encounters, by definition, changes to its requirements to which it needs to adapt. We investigate how logical mobility primitives can be used and implemented to support such adaptation.

Logical mobility is defined as the migration of a partial or complete application or process from one host to another. Logical mobility, commonly implemented using *code mobility* [3] techniques, allows systems to send and receive information that includes binary code compiled for a specific architecture, interpreted code and bytecode compiled for a virtual platform, such as the Java Virtual Machine (JVM), but also application data such as profiles, remote procedure call parameters, etc. Logical mobility has been classified into the following paradigms: *Remote evaluation* (REV) suggests that a host sends code for execution to another host. This paradigm is employed by Distributed.NET [4] and similar distributed computing environments, which work using the divide and conquer paradigm to break large computational tasks into smaller, more manageable tasks and distribute those to machines around the world. The results are then sent back to the server orchestrating the problem, which can recompose the answer to the original challenge. The *Code on Demand* (COD) paradigm enables a host to request code from another machine. Following the request, the code is transferred to the requesting host and can then be executed there. This is an example of dynamic code update, whereby a host or application can update its libraries and available code base at runtime. A *Mobile Agent* (MA) is an autonomous piece of code. It is injected into the network by a host to perform some tasks on behalf of a user or an application. The agent can autonomously migrate from one execution environment to another. COD, REV, and MAs are defined in [3]. We have identified in [5] that mobile systems require the flexible use of all the above logical mobility primitives and we show in this paper that these primitives should be applied to components [6].

Component-based development decouples a system into a set of interacting components with well-defined interfaces. Traditional component metamodels, such as Microsoft's Component Object Model (COM), OMG's CORBA Component Model, or Sun's Enterprise Java Bean component model, provide only very limited support for logical mobility. In particular, they do not allow the code that implements a component to be deployed at runtime onto a remote host. In this paper, we present a component metamodel that supports logical mobility as a first-class concept and we show how this combination supports the construction of adaptable mobile systems.

Component metamodels for distributed computing are often implemented in middleware. We follow the same approach and discuss a middleware that implements our component metamodel. Hence, the novel contribution of this paper is the flexible integration of logical mobility primitives into a component metamodel for mobile environments and an account of its implementation using a middleware approach. We evaluate the approach by means of a number of case studies that develop applications using the component metamodel and its middleware-based implementation.

The paper is structured as follows: Section 2 motivates this work by presenting an industrial example that shows the usefulness of mobile adaptation. Section 3 presents the SATIN component metamodel, which can be used to engineer adaptable component-based mobile systems. Section 4 presents the SATIN middleware and shows how the abstractions it provides can be used to engineer adaptable mobile systems. Section 5 describes the implementation of the SATIN middleware system and evaluates it by means of replicated experimentation. The evaluation addresses both quantitative and qualitative aspects and relies on a number of existing and new applications and projects that use SATIN. Section 6 presents a comparison with related work, while Section 7 concludes the paper and highlights ideas for future work.

2 Background and Motivation

In this section, we illustrate a motivating example and define the concept of adaptation, which will then be used throughout the paper.

2.1 Case Study: Mobile Application Development, Deployment, and Maintenance in Practice

The following paragraphs sketch the current state of practice in engineering mobile applications. We discuss the state of the research literature on mobile applications in Section 6. We then discuss its limitations and show how an adaptive approach based on components together with logical mobility can help in overcoming them.

Consider one of the recently released smart phones, the Palm Treo 650. The device, which runs PalmOS [1], has a 312 MHz ARM processor, wireless (Bluetooth, infrared, cellular, and optional WiFi), and wired (serial) network connectivity. It has 32 megabytes of RAM and a memory expansion mechanism. The current version of PalmOS allows for the creation of event driven, single-threaded applications. All files (applications and data) are stored in main memory. Developers compile an application into a single Palm Resource File (PRC). The operating system allows for limited use of libraries. Applications are identified by a unique 4 byte identifier, the Creator ID. A PalmOS device usually ships with personal information management (PIM) software installed.

A device like this can be used to connect to the centralized network using WiFi or cellular connectivity, but also to various ad hoc networks using the infrared or Bluetooth interfaces. The potential for interaction with its environment is great. However, PalmOS only provides limited primitives for this. The result is that such devices are still seen as stand-alone and independent computers, which interact mainly with a desktop to synchronize changes to shared data—interaction with their environment and peers is either not considered or is very limited. Thus, although *physically mobile*, they are *logically static* systems.

This model has various disadvantages: There is little code sharing between applications running on the same device. There is no middleware providing higher level interoperability and communication primitives for applications running on different devices. Applications are monolithic, composed of a single PRC, which makes it impossible to update part of an application. The procedure needed to install third party applications is difficult. It involves either locating a desktop computer and performing the installation there or having the application sent by another device directly, a procedure which is not automated. In fact, the only popular updates of mobile phone software are the download of ring tones and games. The source of the download is usually the network operator or another centralized agency. Cellular bandwidth, which is expensive for both user and operator, is used for the transfer.

A component-based approach using logical mobility primitives would have several advantages:

- Decomposition of applications as interoperable components would allow for updating individual parts, rather than replacing the application completely.
- Componentization would promote sharing of implementations at runtime, which preserves limited resources of mobile devices.
- Logical mobility primitives would facilitate discovery and retrieval of components existing on any host that is in reach, in a peer-to-peer fashion. This would make application installation much more scalable.
- A component model would provide interaction and communication primitives between components at an abstraction level that is higher than the network protocol stack.
- A component model could support the removal of infrequently used components when the system is running out of resources. The components could be transparently retrieved from peers or a centralized host when needed again.

Note that, in other, less popular mobile operating systems, such as Windows CE-based environments and Linux, the use of components is more prevalent, especially by parts of the operating system. This paper, however, does not advocate the use of general component systems for physical mobility, but rather the use of component systems with logical mobility. Thus, most of the problems outlined above still exist in these systems as those devices also do not interact with services available in their environment and applications are usually monolithic and static. It can be concluded that the monolithic nature of current mobile computing systems contributes to their rigidity: Mobile software is deployed once and is very rarely updated. Moreover, mobile systems lack a generalized infrastructure or middleware system to support interaction with their environment and adaptation to its changes.

2.2 Mobile Adaptation

This work considers adaptation to be *the process by which a system dynamically acquires or discards functionality*. Mobile adaptation is always a *reactive* process. It is an *action* that occurs as a consequence of a particular *event* or group of events. An event can be a *change to the environment*, a *change to the local context*, or a direct result of a *user action*.

The importance of mobile adaptation is highlighted in [7], where it is argued that *mobility exacerbates the tension between autonomy and interdependence that is characteristic of all distributed systems. To function successfully, mobile elements must be adaptive*.

Assuming that events have been emitted that should trigger an adaptation process, it is important to decide on how the system will adapt. A taxonomy for adaptation is defined in [7]. At one end of the spectrum is laissez-faire adaptation. Systems that offer laissez-faire adaptation provide the mechanisms needed to adapt but lack a central arbitrator that encapsulates the decision logic behind the adaptation process. This is delegated to the application, which is fully *aware* of and guides the adaptation process. At the other end is *application-transparent* adaptation, where the application is not aware of the adaptation process; the latter happens internally in the underlying system. In between, there are various types of *application-aware* adaptation, where the application can influence the central arbitrator that handles the adaptation process. We adopt a laissez-faire approach, i.e., we do not provide the decision logic on *how* to adapt, but, rather, a structured way to *engineer* a system for adaptation. We also provide the primitives needed to adapt. A laissez-faire adaptable system can be combined with various decision logic layers that can be used to arbitrate the adaptation process. In fact, we have built a decision logic layer for SATIN in [8].

The reader may consider the requirement for adaptation contradictory to the current success of nonadaptive mobile systems. This paper claims, however, that adaptation primitives are necessary for the emergence of a new class of mobile systems, the pervasive computing systems that adapt to changes in their environment. In fact, sales of traditional mobile devices (such as PDAs) are falling dramatically [9], and this has been explained as a saturation of the market with devices that do not offer anything new.

3 The SATIN Component Metamodel

Although component-based systems are widely used in desktop and client/server-type applications, their use in mobile applications is very limited. This can be attributed to many factors. In particular, the use of existing component models implies a computational, memory, and storage[1] cost over traditional monolithic systems; until recently, mobile devices had limited resources that could not accommodate such cost. With devices becoming increasingly more capable, this barrier has been effectively lifted and mobile devices are able to support lightweight component models.

As distributed component systems already address issues of heterogeneity which are inherent in mobile computing, it would appear that they are suitable for mobile devices. A comparison between distribution and collocation for object systems is presented in [10]. Distributed systems, such as the OMG's Common Object Request Broker Architecture [11] (a comparison of this work

1. Depending on the actual system, a component-based approach can lead to a decreased storage cost because of code (component) reusability.

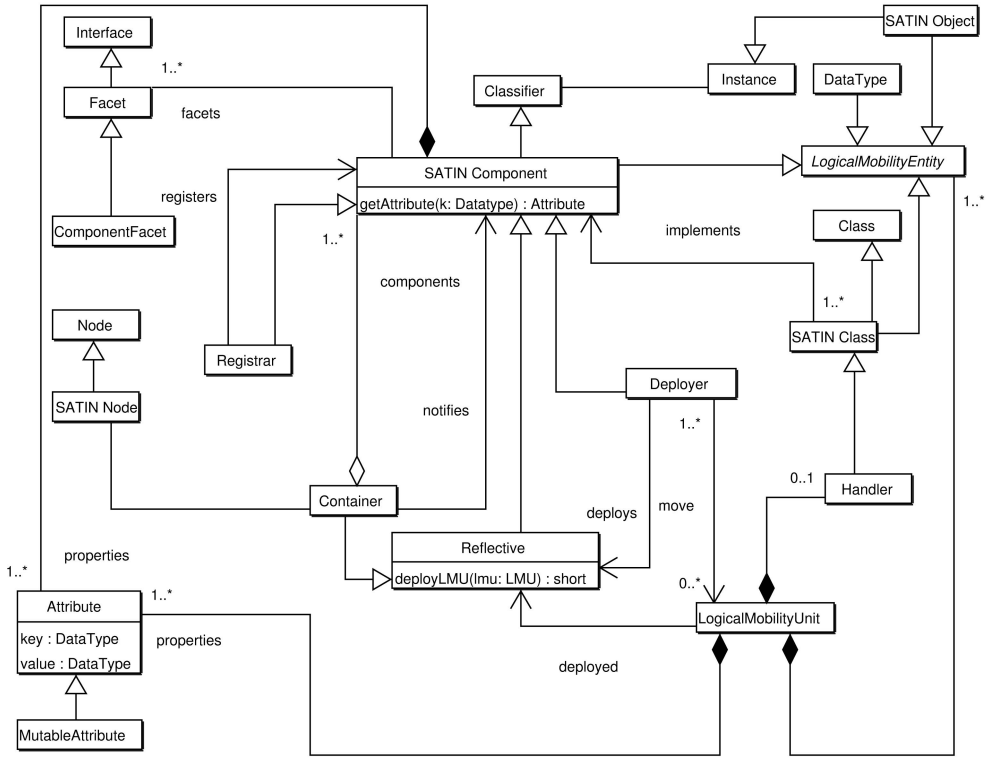


Fig. 1. The SATIN metamodel.

with the CORBA Component Model is presented in Section 6), Microsoft's Component Object Model [12], or the Enterprise Java Bean component model [13], are not suitable for mobile adaptation in a dynamic environment for the following reasons:

1. *Size.* Mobile devices have very limited resources. Distributed component model implementations are memory and CPU intensive to deliver functionality such as transactions, persistence, and concurrency control, which are often not essential in a mobile setting. These primitives can be provided at a higher level (i.e., built using the component model) if needed.
2. *Synchronization Primitives.* The most widely available synchronization primitive in distributed component models is a remote procedure call (RPC). RPCs assume continuous, reliable, and low-latency network connections to deliver synchronous invocations and fail with exceptions if these conditions are not met. In mobile networks, intermittent, unreliable, and delayed network connectivity is the norm and, as a result, the synchronous interaction paradigm supported by RPCs in distributed component models is unsuitable [14].
3. *Static deployment.* Distributed component models assume that an administrator deploys components across hosts. The physical mobility and volatile nature of the network connectivity of mobile devices dictate that the devices form highly dynamic networks, which may even be without a central administrator who takes deployment decisions. Even when the latter is not the case, mobile devices form significantly less predictable topologies than distributed systems in fixed networks. As a result, mobile applications are hardly comparable to standard distributed systems in terms of structure and deployment.

We now present the SATIN component metamodel, which is a local component metamodel for mobile adaptive systems. The rationale for SATIN's design decisions is given in terms of the case study presented in Section 2.1.

3.1 SATIN Component Metamodel Overview

The SATIN component metamodel is a *local*, or *in process*, *reflective* component metamodel for applications hosted on mobile devices. The model uses logical mobility primitives to provide distribution services and offers the flexible use of those primitives to applications; instead of relying on the invocation of remote services via the network, SATIN components are collocated on the same address space. The model supports the remote cloning of components between hosts, providing for system autonomy when network connectivity is missing or is unreliable. As such, an instance of SATIN is represented as a collection of local components, interconnected using local references and well-defined interfaces, deployed on a single host. The model also offers support for *structural reflection* [15] so that applications can introspect which components are available locally, choose components to perform a particular task, and dynamically change the system configuration by adding or removing components.

The SATIN component metamodel, as shown in Fig. 1, is a Meta Object Facility (MOF) [16]-compliant extension of the UML metamodel [17]. It builds upon and extends the

UML concepts of Classifier, Node, Class, Interface, DataType, and Instance. We have chosen the MOF metamodel rather than using the UML extension mechanisms because the concepts introduced by SATIN are too radically different from existing component models to be defined by stereotypes or tagged values. The most novel aspect of the component model is the way in which it offers distribution services to local components, allowing instances to dynamically send and receive components at runtime. We describe the model informally in the remainder of this section in order to make the model more accessible to practitioners who wish to use SATIN. A formal definition of the metamodel semantics using a process algebra is provided in [18].

3.2 Components

A SATIN *component* encapsulates particular functionality, such as, for instance, a user interface, a service advertisement protocol, a service, an audio codec, or a compression library. SATIN components separate interfaces and implementations. A component is implemented by one or several SATIN classes. It can implement one or more interfaces, called *facets* (a term borrowed from the CORBA component model [19]), with each facet offering any number of operations. A metamodel for components that are going to be deployed across autonomous domain boundaries needs to ensure that interfaces that have once been defined cannot be changed. For that reason, SATIN facets are immutable and application designers who wish to change an interface will have to create a new facet that the component is going to implement as well. A similar choice has been made in Microsoft's Distributed Component Object Model.

3.2.1 Component Metadata

SATIN is designed to support applications on heterogeneous set of mobile devices and architectures. Hence, the SATIN component abstraction must be rich enough to describe components that may be deployed across a large number of platforms. A device must be able to reason about whether a component is compatible with its hardware and whether it depends on the existence of another component to verify its authenticity, etc. To this end, SATIN uses attributes to describe a component, similarly to the Debian Project's [20] `.deb` packaging system.

A SATIN component *attribute* is a *key/value* pair. The set of all attributes of a component is the *properties* of the component. A component uses properties to express its dependencies on both the underlying software and hardware infrastructure. A set of attribute keys, or an ontology for both keys and values, is not defined by the metamodel but is determined by an application designer using the metamodel. Examples of attributes are an `ID` attribute, which acts as a component identifier, and a `VER` attribute, which denotes the version of the component implementation. As such, a component implementation can be uniquely identified using the `ID` and `VER` attributes. This also allows for differentiating between different versions of a component implementation.

Generally, the component properties allow for attaching arbitrary metadata to a component. Attributes can be mutable. As such, the component properties are a *memory segment* assigned to each component, managed by the SATIN container (see below). When updating a component, some state may need to be maintained. As attributes can be used by a component implementation to encapsulate such state separately from the component logic, the latter can be updated while maintaining the former.

Each SATIN component implements at least one facet, the `Component` facet. The purpose of this facet is to allow a device to reason about the component and its attributes. As such, it permits access to the properties of the component by retrieving, adding, removing, and modifying attributes. The component facet also contains a *constructor*, which is used to initialize the component, and a *destructor*, which is used when removing the component from the system (see below). Finally, the component facet allows for enabling or disabling a component (see Section 3.5).

3.3 Components and Containers

To reason about how to adapt, a device must *reflect* on what it can currently do or what its capabilities are. The central component of every SATIN system is the *container* component. A container is a component specialization that acts as a registry of components installed on the system. As such, a reference to each component is available via the container. The container component implements a specialization of the component facet that exports functionality for searching components that match a given set of attributes.

An adaptive system must also be able to react to changes in component availability. For example, a media player in a smart phone must be able to reason about which streams it can decode—if a new codec is installed, the media player should allow the user to connect to a new service, which is now decodable. Hence, the container permits the registration of listeners (represented by components that implement the `ComponentListener` facet) to be notified when components matching a set of attributes given by the listener are added or removed. As such, the `ComponentListener` facet provides a very simple event notification service, which is implemented by components that need to react to the availability of new components.

To allow for dynamic adaptation, the container can dynamically add or drop components to and from the system. Registration and removal of components is delegated to one or more *registrars*. A registrar is a component that implements a facet that defines primitives for loading and removing components, validating dependencies, executing component constructors, and adding components to the registry. When removing a component, a registrar is responsible for checking that the removal of the particular component will not invalidate the dependencies of others, calling its destructor, and removing it from the registry. This allows the system to be left in a consistent state after a component removal. Different registrars can have different policies on loading and removing components (from different sources, for example) and verifying that the dependencies are satisfied. For example, a smart phone with limited resources can run implementations of the container and registrar to keep track of how often components are used—this frequency-based approach can be used to drop the least-recently used components when

the system runs out of memory. Moreover, implementations of the registrar can emit events to notify interested listeners of component registration failures. Finally, registrar implementations may offer atomic registration and removal of groups of components.

The use of a container allows for introspecting the status of the platform, as callers of the container facet can reason about the current availability of functionality, encapsulated in components. Combined with the use of registrars to allow dynamic addition and removal of components, the SATIN container offers structural reflection at the component level, that is, the ability to reason about the components in the system.

3.4 Distribution and Logical Mobility

A system built using SATIN can reconfigure itself by using logical mobility primitives, as proposed in Section 1. As different paradigms can be applied to different scenarios, our metamodel does not build distribution into the components themselves, but it provides it as a service; implementations of the SATIN metamodel can, in fact, dynamically send and receive components and employ any of the above logical mobility paradigms. This functionality is provided using Logical Mobility Entities and Units, as well as Deployer and Reflective components. Their relationships are outlined in Fig. 1 and discussed below.

We consider four aspects of Logical Mobility: Components, Classes, Instances, and DataTypes; the last is defined as a bit stream that is not directly executable by the underlying architecture. One such, the *Logical Mobility Entity* (LME), is defined as an abstract generalization of a Class, Instance, or DataType.

The *Logical Mobility Unit* (LMU) is defined as a container that can encapsulate various constructs and representations of code and data. As such, an LMU is a composition of an arbitrary number of LMEs; in effect, the LMU is used to encapsulate Logical Mobility Entities, which were defined as either classes, data, instances, or components. The LMU provides operations that permit inspection of its content. This allows a recipient to inspect an LMU before using it.

The LMU can potentially encapsulate a *Handler* class. The Handler can be instantiated and the resulting object used by the recipient to deploy and manipulate the contents of the LMU. This can allow sender-customized deployment and binding. The Handler concept and name are taken from MuCode [21], which is a code mobility toolkit written in Java. An earlier prototype implementation of SATIN was built using this toolkit. Handlers and deployment in general are discussed further in the next paragraphs.

Similarly to SATIN components, an LMU also encapsulates a set of *attributes* called the *properties* of the LMU. The properties are used to describe the LMU they are associated with. For example, logical (software) or physical (hardware) dependencies, digital signatures, and even end-user textual descriptions can be expressed as attributes. As such, they can be used to express the heterogeneity of the target environment. For example, an LMU that contains Java classes may specify that it requires a Java Virtual Machine that implements version 2 of the appropriate specification as an attribute. An ontology for attribute keys and values is not defined by the metamodel, but will be defined by the application designer.

The LMU and its contents can be serialized and deserialized. Logical mobility is then equivalent to composing the LMU, serializing it, transferring it to a remote device, deserializing it, and deploying it.

We now need to define how and where to deploy an LMU in the local system. In the SATIN component metamodel, an LMU is always deployed in a *Reflective* component. A Reflective component is a component specialization that can be *adapted* at runtime by receiving LMUs from the SATIN migration services. By definition, the container is always a reflective component, as it can receive and host new components at runtime.

A SATIN LMU has two required attributes; `TARG`, which specifies the intended target node, and `LTARG`, which specifies the logical target or reflective component in the host specified by `TARG` to which the LMU is going to be deployed. For example, when deploying a component into the system, the value of the `LTARG` attribute points to the instance of the container in that system. As such, `TARG` is referred to as the physical destination of the LMU, whereas `LTARG` is the logical destination. Hence, the use of the location attributes allows for deploying components in specific targets uniquely identified by the (`TARG`, `LTARG`) tuple. LMUs can be implemented using JAR archives, for example.

A SATIN application cannot send an LMU directly. The functionality of sending, receiving, and deploying LMUs is handled by the *Deployer*. The Deployer is a SATIN component specialization that manages requesting, creating, sending, receiving, and deploying LMUs to the appropriate reflective components. A Deployer is directly accessible to any application through the container. The Deployer can be used to implement any logical mobility paradigm. Remote Evaluation becomes composing, sending, and deploying an LMU; Code on Demand becomes requesting, receiving, and deploying an LMU; and Mobile Agents become sending an LMU that encapsulates a Handler, which is used to represent the active agent.

A Deployer will reject any request to send LMUs that do not specify a logical and a physical destination as it would otherwise be responsible for serializing and sending the LMU to the Deployer component instance located at the physical destination. When receiving an LMU, the Deployer uses the container to verify that the component identified by the logical destination of the LMU exists in the local SATIN instance and that it is a reflective component. The LMU is then moved to its logical destination, which has the option of inspecting the contents before deployment; by using the methods exported by the LMU, a reflective component can access the properties and contents of the LMU before accepting it. As such, the inspection can result either in *full acceptance*, which means that the contents of the LMU are accepted in their entirety; *partial acceptance*, which means that parts of the LMU are accepted and others discarded; *rejection*, which means that the LMU is rejected and dropped; or *handler instantiation*, which means that the reflective component instantiates the Handler, encapsulated in the LMU, to perform the deployment. The result is

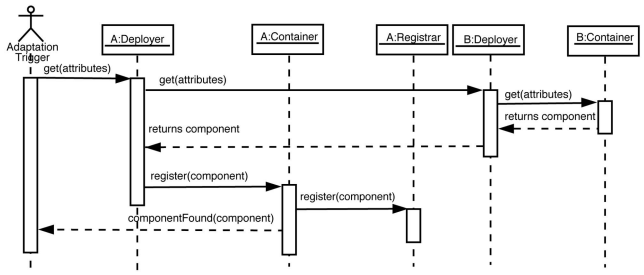


Fig. 2. A sequence diagram showing the adaptation process.

determined by the reflective component, based on the contents of the LMU. This gives flexibility to the reflective component implementer and is part of the laissez-faire nature of SATIN.

Fig. 2 shows a sequence diagram depicting an adaptation sequence. In it, an adaptation trigger (a user, a component reacting to an environmental change, an adaptation logic engine, etc.) asks the local Deployer for a component satisfying a set of attributes. The Deployer contacts a remote Deployer instance (we do not discuss how the remote Deployer is located at this stage but come back to it in Section 4) for the component. The remote Deployer asks for the component from the Container instance in the remote node, packs it inside an LMU, and sends it back to the original Deployer. The latter deploys it to the local Container, which delegated the registration process to the default Registrar. The Container then notifies the original Adaptation trigger, provided that it is a ComponentListener that has registered itself with the container for the event.

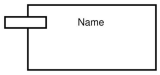
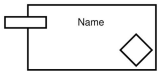
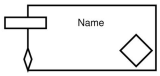
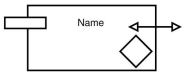
3.5 Component Life Cycle

SATIN supports a very simple and lightweight component life cycle. When a component is passed on to the container for registration by loading it from persistent storage, using a Deployer, etc., the container delegates registration to a registrar component. The registrar is responsible for checking that the dependencies of the component are satisfied, instantiating the component using its constructor, and adding it to the registry. Note that the component facet prescribes a single constructor. An instantiated component can use the container facet to get references to any other components that it may require.

A component deployed and instantiated in the container may be either enabled or disabled. The semantics of those and the initial state of the component depend on the component implementation. The functionality needed to manipulate the state of the component is exported by the component facet.

SATIN does not distinguish between multiple instances of the same component. This can be achieved by using specializations of the container. When removing a component, a registrar is responsible for verifying that the removal of the component does not break any dependencies in the system, disabling it, calling the destructor of the component, and then removing it from the registry. Instantiations of the metamodel may choose to associate the registrar that registered the component with the component itself. In this way, the same registrar can be automatically called to remove the component when requested.

TABLE 1
A Fragment of the SATIN Metamodel Notation

Notation	Description
	A SATIN Component called `Name`.
	A SATIN facet called `Name`.
	A reflective SATIN component called `Name`.
	A SATIN container called `Name`. Note that it is also a reflective component.
	A SATIN deployer called `Name`. Note that it is also a reflective component.

We have defined a notation that can be used to describe systems that instantiate the metamodel by extending the UML notation. The notation is used in the next sections. The full notation is available in [18], while a fragment including only the entities used in the diagrams of the next sections is presented in Table 1.

4 The SATIN Mobile Computing Middleware System

The previous section described a lightweight component metamodel that provides logical mobility primitives in support of adaptation of mobile systems. Metamodels for distributed computing are only useful in practice if they are implemented by some middleware. The aim of middleware in general is to provide higher level interaction primitives than those provided by the network operating system as a layer upon which applications are then constructed. In doing so, the middleware hides the complexities of addressing distribution, heterogeneity, and failures. This section shows how the SATIN middleware implements the SATIN metamodel. Since the aim of the paper is to demonstrate the flexible use of logical mobility primitives, we focus on distribution.

It is good practice in middleware design to rely on and use the primitives provided by the metamodel in the design of the middleware itself. The services provided by any CORBA implementation, for example, use the interaction primitive of remote object requests defined by the CORBA object metamodel also for the internal communication across higher level CORBA services. In this tradition, the SATIN middleware uses the adaptation primitives defined by the SATIN component model to build a flexible and adaptable middleware platform for mobile computing. Hence, while describing the design of the SATIN middleware, we also validate the SATIN metamodel by showing how it can be used to build a complete middleware system, which offers dynamically adaptable services. Thus, the middleware itself and all applications built with it are represented as a

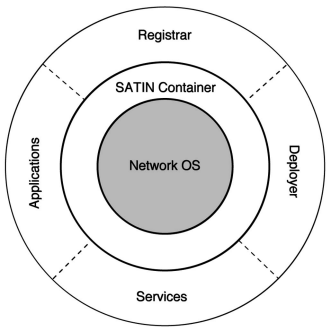


Fig. 3. The architecture of the SATIN middleware.

collection of SATIN components registered with the SATIN middleware. The interfaces to these components are specified in a number of facets.

4.1 Middleware Overview

Fig. 3 shows a high-level overview of the SATIN middleware. By instantiating the metamodel, the middleware itself is adaptable and allows applications to use any logical mobility primitive.

The system is built on top of the network operating system and provides an instance of the SATIN container, as defined in Section 3. This container is the central aspect of every instance of the middleware system. Registered with the container are all the components that are part of the system. This includes application components (such as a media player application), libraries (such as audio codecs), and system services (such as any registrars, deployers, service advertising, and discovery components, etc.). All components make their dependencies explicit through their properties. The circular notation used in Fig. 3 denotes that, from the point of view of the container, all other components available to the system are *equal*. The core of every SATIN system is the container, with every other service (including logical mobility) or application components built on top of it. Thus, even though components may build complex dependency graphs expressed via their properties, to the container, they all implement components facets. Components can be added and removed at runtime. This allows the middleware itself to adapt.

4.2 Advertising and Discovery

The SATIN middleware provides a number of services to applications. The services themselves are seen as regular components built on top of the container. As such, they can be dynamically added and removed. In the following paragraphs, we outline how the SATIN metamodel primitives are used to provide adaptable advertising and discovery services for SATIN components.

In general, one of the pivotal requirements of mobile and adaptable pervasive computing is the ability to reason about the environment. The environment is defined as the network of hosts that can, at a specific point in time, communicate with each other. In order to adapt, a mobile system needs to be able to detect changes that occur in its environment. As the host itself is also part of that environment, it needs to advertise its presence. A mobile host, however, may be able to connect to different types of

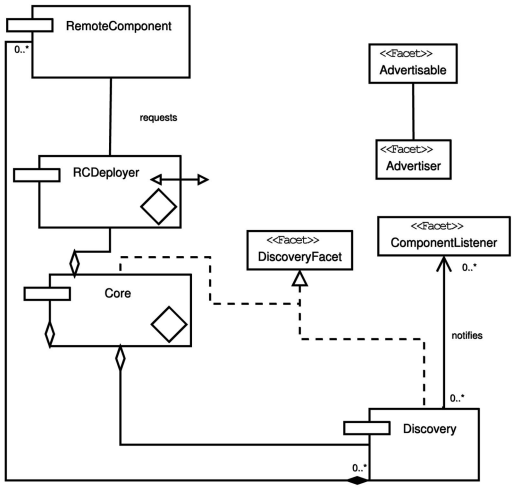


Fig. 4. The advertising and discovery services.

networks, either concurrently or at different times, with different networking interfaces. There also are many different approaches to advertising and discovery. Imposing a particular advertisement and discovery mechanism can hinder interoperability with other systems, making assumptions about the network, the nodes, and the environment, which may be violated at some later stage or simply not be optimal in a future setting—something which is likely to happen, given the dynamism of the target area of this work.

From the point of view of SATIN, the ability to reason about the environment is addressed by the ability to discover components currently in reach and to advertise the components installed in the local system. This is achieved via the use of *Remote* and *Discovery* components, as well as *Advertiser*, *Advertisable*, *DiscoveryFacet*, and *ComponentListener* facets. The design of the advertising and discovery service using the SATIN component metamodel concepts is shown in Fig. 4. The notation, outlined in Table 1, denotes that RemoteComponent, RCDeployer, Core, and Discovery are Components; RCDeployer and Core are Reflective Components; DiscoveryFacet, ComponentListener, Advertiser, and Advertisable are facets; and Discovery and Core both implement the DiscoveryFacet. The service is described in detail below.

Components that wish to advertise their presence to the environment must implement the `Advertisable` facet. Examples of advertisable components include codec repositories, services, etc. The `Advertisable` facet exports a method that returns a message that is used for advertising; thus, the advertising message allows the Advertisable component to express information that it requires to be advertised. An advertising technique is represented by an *Advertiser* component, which is a component implementing the `Advertiser` facet. An advertiser component is responsible for accepting the message of advertisable components, potentially transforming it into another format and using it to advertise them. An advertiser allows components that wish to be advertised to register themselves with it to be advertised. The combination of component availability notification and advertiser registration allows an advertisable component

to register with the container to be notified when specific advertisers are added to the system. The advertisable component can then register to be advertised by them. Moreover, an advertisable component can express that it requires a particular advertiser in its dependencies. Thus, the semantics of the advertisable message are not defined and depend on the advertisable component and on the advertising technique (i.e. the advertiser component) used. Note, that a component can implement both the `Advertiser` and the `Advertisable` facets. This allows for the advertising of advertising techniques; in this way, for example, the existence of a multicast advertising group can be advertised using a broadcast advertiser. Combined with the use of logical mobility primitives, this allows a host to dynamically acquire a different advertising and discovery mechanism for a network that was just detected. For example, upon approaching a Jini network [22], a node can request and download the components that are needed to advertise to, and use functionality from, the network.

The design of discovery techniques is analogous. A *Discovery* component is a component that implements a discovery service and is able to locate remote advertisable components. The latter are represented on the local instance (i.e., the instance of the container that hosts the Discovery component instance) as *RemoteComponents*. Note that a Remote Component is not a component that can be accessed remotely (as SATIN is a local component model)—it just provides methods to access the information in the advertising message and properties of the original advertisable component. The component can, however, be passed on to a Deployer instance (represented by *RCDeployer*) to be requested for deployment on the local host. As such, Discovery components act as a registry of Remote Components. Given the similarity with the container, both Discovery techniques and the container instance in the SATIN middleware (denoted by *Core* in Fig. 4b) implement the same facet, called the *DiscoveryFacet*, providing a common interface to the programmer.

The SATIN advertising and discovery service is general. It supports the implementation of facets and abstractions on top of existing techniques. As such, implementations of the service can use Jini, UPnP, etc., as their underlying service advertising and discovery techniques. Section 5.2 describes two very different implementations using IP Multicast and Publish-Subscribe.

5 Implementation and Evaluation

This section details the implementation of the SATIN middleware. The implementation is benchmarked and its performance on mobile devices is evaluated by measuring memory overhead and time needed to adapt. Both metamodel and middleware were used to develop a number of applications, some of which are adaptations of existing software. The suitability of SATIN for mobile adaptation is evaluated by illustrating how each application adapts and how this is made possible through the use of SATIN.

5.1 On Evaluating SATIN

The methodology used to evaluate SATIN follows the established combination of qualitative and quantitative evaluation. We use a combination of replicated experiments, dynamic analysis, case studies, assertions, and comparisons with legacy data [23] to evaluate the metamodel and its implementation. Thus, this section evaluates SATIN using the following approach:

- *Feasibility.* How practical are the metamodel abstractions, their instantiation, and their actual implementation for the constraints of mobility? To answer this question, this section begins by defining a class of devices which will be specifically targeted and continues by detailing the middleware implementation. We show that this implementation is lightweight enough to run on mobile devices despite the added functionality. We measure the memory footprint as well as the time needed to adapt. We compare the size of the implementation to that of other approaches in this area of research.
- *Utility and Completeness.* Can complete systems be built using the abstractions defined? Are they sufficient to support system level, as well as application level adaptation? To answer these questions, this section details the design of numerous systems and applications using SATIN and shows how they address specific issues identified in the motivating example presented in Section 2.1.
- *Usability and Complexity.* How easy is it to program using SATIN? How easily can existing projects be converted to use SATIN? Are the abstractions and design provided by SATIN usable by third parties? To answer these questions, this section describes the conversion of existing open source applications into SATIN components. By doing so, we can quantify the overhead created by SATIN and weigh it against the advantages in terms of adaptability of the applications.

We decided against using simulation as an evaluation technique because SATIN does not propose any new networking protocols the behavior of which would be advantageous to simulate. Rather, this paper presents a new approach to engineering mobile systems. As the approach itself is inherently modular, the details of the implementation of each individual module or component (which can be replaced) are not deemed important to simulate or benchmark extensively against. As such, we present an extensive quantitative evaluation of only one of the example applications. The other two applications are presented in order to illustrate the ease of refactoring existing code as SATIN components and to show the advantages that its port to the SATIN framework brings.

5.2 Implementation

While implementing SATIN, we had to make design choices regarding the class of mobile device on which it would run. Considering the rate at which hardware changes, this may be considered immaterial; however, by defining an actual target, quantifiable assertions about speed and resource requirements can be made. With these statements in mind, the devices that the implementation presented in this section targets are PDAs. More specifically, the target was one of the most popular PDAs of 2001, the Compaq iPAQ H3600. It uses a 206 MHz ARM CPU and 64 MB of RAM (32 of which are used as storage, so, in effect, the PDA has 32 MB of RAM). An IEEE 802.11b wireless adaptor was also connected to it. This device was chosen because the hardware that it offers is very mediocre by today's

TABLE 2
Details on the SATIN Implementation

Item	Size (in bytes)	Lines of Code
MiToolkit	89,072	800
Meta Model	13,607	233
Midleware (Model)	47,650	925
Advertising and Discovery Framework	6,668	118
Publish Subscribe	22,797	574
IP Multicast	22,161	418
Total	201,955	3,068

standards. In fact, entry-level PDAs are significantly more powerful, and mobile phones are starting to offer equivalent (or better) resources. Thus, by showing that the current implementation of SATIN can scale down to run on an old device like this, we can infer that it will run even better on more recent devices, like the one presented in Section 2.1.

SATIN has been implemented using Java 2 Micro Edition (Connected Device Configuration (CDC), Personal Profile). There are many reasons for this choice: Java, the Java 2 Micro Edition in particular, is a portable language and virtual machine for mobile devices. The virtual machine and the Java bytecode are used for binary-level interoperability between components. The CDC and personal profile were specifically chosen because they allow the use of the Java Object Serialization framework and Reflection API for the deployer implementation; this enables the dynamic sending and receiving of Java classes and objects. A mobile code toolkit, MiToolkit [24], was used to realize a logical mobility platform that is encapsulated by the SATIN Deployer. MiToolkit is the library that can be used to send and receive Java-based LMUs. The Advertising and Discovery framework was implemented using IP Multicast and also by using a simple centralized publish/subscribe protocol we devised.

Table 2 gives details on the size of the SATIN implementation. Note, that the size figure presented represents the *uncompressed* size, which is against the typical Java tradition. The Source Lines of Code (SLOC) were calculated using SLOCCount [25] and the number does not include comments or blank lines. The metamodel implementation represents the abstract classes and interfaces that represent the SATIN metamodel. Those are reified as a middleware system to provide the actual functionality.

These numbers compare favorably to other related projects. For example, OpenCOM [26] requires 28,160 or 18,432 bytes for its ARM and x86 implementation, respectively, versus the 13,607 bytes that the SATIN metamodel requires. The complete SATIN implementation requires 201,955 bytes of storage versus 609,700 bytes that, for instance, Lime [27], a data sharing middleware system that uses mobile agents, requires. Starting up the middleware and registering the deployer takes 1,846 milliseconds on a Pentium II 266 MHz machine with 64 megabytes of RAM. At that time, the middleware objects require 113,872 bytes of heap memory.

5.3 Code Fragments

This section includes some actual code fragments from the applications described below. For more details, the reader is referred to the SATIN open source project at [28].

The following shows how to initialize SATIN and register a Deployer and a discovery service with the container:

```
new Core(); //initialize the container
//get a reference to it
Container container =
          Container.getContainer();

//get a reference to the registrar
RegistrarFacet registrar =
          container.getDefaultRegistrar();

//Initialize the deployer
DeployerFacet d = new MiToolkitDeployer();

//register it with the middleware
registrar.registerComponent((Component)d);

d.setEnabled(true); //enable it

//initialize the discovery component
CentralDiscovery disc=new CentralDiscovery();

//register it
registrar.registerComponent(disc);
```

The following fragment shows how to initialize and send the Ogg Vorbis codec (see below) to a node with the IP address 192.168.0.1:

```
ComponentFacet c=new OggVorbisCodec();
registrar.registerComponent(c);

//gets a reference to the deployer, using its
identifier
DeployerFacet
   d=container.getComponent("STN:MITKDEP");

/* The following statement creates a new LMU,
 * with target "192.168.0.1" and
 * defines its destination as "STN:CONTAINER",
 * which is the container of the recipient host.
 *
 * This effectively defines that the LMU should
 * be sent to the
 * container of 192.168.0.1.
 *
 * The container will try to register it.
 */
LMU lmu=new HashLMU ("192.168.0.1",
   "STN:CONTAINER");
//adds the component to the LMU.
lmu.addComponent(c);

d.send(lmu);//sends the LMU
```

The following fragment shows requesting to be notified when a component with the value of the BITRATE attribute greater than or equal to 32 and the value of VER equal to 1 is found. This specification matches the Ogg Vorbis codec encapsulation.

TABLE 3
Evaluation Coverage Matrix

	Program Launcher	Music Player	Active Networking
Application Level Adaptation	x		
Middleware Level Adaptation	x		x
Legacy Code Adaptation		x	x
Advertising & Discovery Framework	x		
Code On Demand	x	x	
Remote Evaluation		x	x
Limited Resources	x	x	x

```
Hashtable template=new Hashtable();
template.put("BITRATE",
     new MatchAttribute("BITRATE",
     new Integer(32),
     new GreaterEqualThanFilter()));
template.put("VER",
     new GenericAttribute("VER",
     new Integer(1),true));

//adds a listener for a component
based on that template */
container.addListener(
     (ComponentListener)this,template);
```

SATIN was used to implement a number of applications, some of which are outlined below. Table 3 outlines what aspects of SATIN each example illustrates. The Program Launcher example illustrates how SATIN can be used to offer a middleware level adaptable service that allows end user applications to adapt. It demonstrates the use of the advertising and discovery framework. Moreover, it includes an extensive qualitative evaluation, examining how SATIN runs on a device with limited capabilities. The purpose of the Music Player and Satellite Active Networking applications is not to further benchmark SATIN (though some numbers are presented for completeness), but to illustrate how the same metamodel and middleware can be used to develop two vastly different systems, one exhibiting application level adaptation and another offering a low level networking service, but both adhering to the same principles. Both also exhibit how legacy code can be fitted to run under SATIN and be deployed dynamically. All three examples illustrate how SATIN has a minimal overhead and is suitable for scenarios with devices that offer limited resources.

5.4 The SATIN Program Launcher

Inspired by the problems discussed in Section 2.1, this application is a Dynamic Program Manager, or Launcher, for mobile devices. It is similar to the PalmOS Launcher in that its basic purpose is to display and launch applications that are registered with the container. The applications installed are shown as buttons, with the component identifiers as labels. The Launcher also manages and controls all installed components. Applications are components that implement the `Application` facet. As such, the program launcher registers itself with the container, to be notified when a component implementing the `Application` facet (i.e., a new application) is registered, in order to automatically redraw its interface to be able to launch it.

The dynamic program launcher offers the following services: Using the deployer, it can install any component from any discoverable source (i.e., through any discovery service). Using the same mechanism, it can update the components installed in the system, either transparently or as a result of a user command. If the device running the Launcher runs out of memory, it can discard unused components based on their frequency of use.

The application caters to the scenario presented in Section 2.1: Mobile devices roam through a dynamic context, able to transparently update their libraries and install new applications available in their current environment. Application are componentized, which facilitates maintenance.

The components of the launcher use 39,749 bytes as an uncompressed jar file and were written in 778 SLOC. The launcher is fully componentized; as such, the updating mechanism and the remote install mechanism are encapsulated as separate components. Fig. 5 shows the architecture of the Launcher expressed as a collection of interdependent components.

Testing. There were three devices in our tests: node alpha, which is a PDA (200 MHz StrongARM CPU, 32 MB of RAM as described above); node beta, which is a laptop (266 MHz Pentium II CPU, 64 MB of RAM); and node gamma which, is another laptop (1.5 GHz PowerPC G4 CPU, 1.25 GB of RAM). Alpha is connected to beta using an IEEE 802.11b card in ad hoc mode, while beta was connected to gamma using a fast Ethernet connection. Thus, alpha and gamma cannot connect to each other. Note that the connections are all single hop, with no artificial latency induced in either the wireless or the Ethernet network.

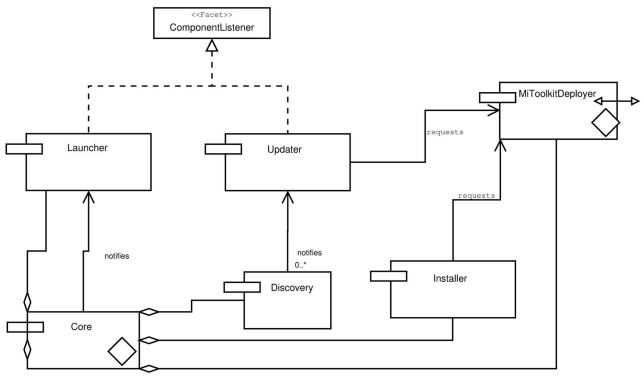


Fig. 5. The SATIN Launcher as a collection of components.

TABLE 4
Quantitative Evaluation Results of Program Launcher

Test	Mean	Standard Deviation	Variance	Confidence Interval
Startup time (alpha)	955.56 ms	0.5014	0.2514	0.1390
Memory usage (alpha)	1065784.48 bytes	3452.4369	11919320.65	956.9491
Discovery time (beta)	10.44 ms	2.2782	5.1902	0.6315
Receiving time (beta)	495.96 ms	32.3242	1044.8556	8.9597
Deployment time (beta)	2.54 ms	0.5035	0.2535	0.1395
Discovery time (alpha)	5.86 ms	0.4046	0.1637	0.1121
Receiving time (alpha)	1777.5 ms	57.4787	3303.8061	15.9320
Deployment time (alpha)	4.46 ms	0.5034	0.2535	0.1395

Alpha and beta are running Linux (familiar 0.6 and Debian 3.1, respectively), while gamma is running MacOSX 1.4.4. Note that node alpha is running a beta version of Java, with no Just In Time (JIT) compilation. This virtual machine was chosen because it is the only connected device configuration/personal profile compliant virtual machine for this platform. Moreover, by using interpretation rather than JIT compilation, we further restrict, artificially, the capabilities of our underlying platform, thus demonstrating how our implementation can scale down.

Our testing scenario tries to illustrate how an application can be transparently updated on heterogeneous nodes connected in heterogeneous networks. In the fast Ethernet network, there is a publish/subscribe advertising and discovery service. Node alpha cannot access it, as it does not have Ethernet connectivity. On the ad hoc wireless interface, however, there is a multicast advertising and discovery service running. Only node alpha and beta can access it, as gamma does not have a wireless interface. Alpha, beta, and gamma have the components needed to access the respective services available. All discovery services are preconfigured to access their respective advertisers. The multicast advertiser is configured to advertise new components every five seconds, while the publish subscribe server publishes messages to interested parties with the registration of every matching component.

A sample "Hello World" component was created, which is Advertisable. The component requires 1,257 bytes. Initially, alpha and beta are the only hosts online, and they have version 0 of the component. At some point, gamma comes online, advertising (over the publish subscribe service), the availability of version 1 of the component. The Launcher libraries of beta locate the component and request it, resulting in the deployment of version 1 on beta. However, version 1 of the sample component is now also advertised on the wireless network through beta, via the IP multicast service. Node alpha discovers this, requests it, receives it, and deploys it. Table 4 shows

1. the startup time of SATIN (with the multicast advertising and discovery service registered but without the launcher) on alpha, which excludes the Java Virtual Machine startup time;
2. the memory (heap) usage on alpha once the launcher and the sample component are also loaded;
3. the time node beta takes to discover and request version 1 of the sample component, after node gamma starts advertising it;
4. the time taken to receive, deserialize and load version 1 into the Java Virtual Machine of beta;
5. the time taken to deploy the component into the container of beta (checking any dependencies, etc.);
6. the time alpha takes to discover version 1 after it has been deployed on beta;
7. the time alpha takes to receive, deserialize and load version 1 into its Java Virtual Machine; and
8. the time alpha takes to load the component into the container (checking any dependencies, etc.).

The tests were run 50 times. The table includes the mean, variance, standard deviation, and confidence intervals with a 95 percent confidence level of each test.

The results obtained above show that the current implementation, even though unoptimized, is reasonably efficient. The tests illustrate that the most expensive operation is receiving, unpacking, and loading the classes that constitutes a component into the Java Virtual Machine of a node. One reason for this could be the Java Security Model, which verifies at runtime all incoming classes. Moreover, MiToolkit checks the incoming classes for namespace conflicts. The large difference in the receiving time between nodes alpha and beta is notable, with beta being approximately 2 to 3 times faster. This can be attributed to three main factors: First, node beta is much more capable, with twice as much memory and a faster CPU (even megahertz per megahertz, the Pentium II is faster than the StrongARM, as the latter lacks an FPU and is optimized for power consumption, rather than speed). Second, nodes beta and gamma are connected by a much faster and more reliable network. Finally, the Java Virtual Machine running on alpha interprets all the code running, rather than compiling it at runtime.

5.5 The SATIN Music Player

We implemented a simple music player using SATIN. This is a relevant application, as different media formats exist and new ones emerge constantly, demanding updates of the media player code. Media formats are commonly implemented in *codecs*. In our media player, SATIN components that implement audio codecs must implement the `AUDIOFORMAT` facet. The Music Player then uses the notification service to be notified whenever a component that provides this facet is registered. Moreover, it uses the deployer and the discovery components to download any codecs that are found remotely. The application itself occupies 6,568 bytes as an uncompressed jar file and was written in 133 SLOC.

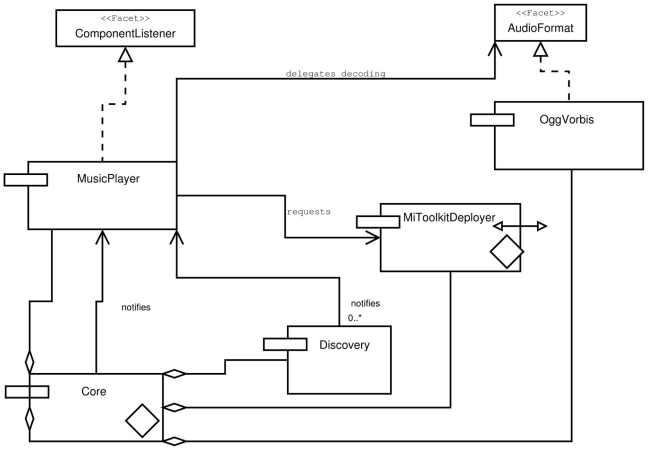


Fig. 6. The SATIN music player as a collection of components.

Note that this excludes any codec—it provides a basic user interface and an API for codecs to control the player.

JOrbis,[2] an open source Ogg Vorbis [29] implementation, was also adapted to run as a SATIN audio codec component. Ogg Vorbis is an open source, lossy compression audio codec. Both the music player and the componentized audio codec can be dynamically sent and deployed on SATIN nodes. The application is automatically notified when a codec component is found and adapts its interface accordingly. The JOrbis component occupies 169,978 bytes as an uncompressed Java archive and is composed of 50 classes. The Music Player application is a Java 2 Standard Edition application and not a Micro Edition application. This is denoted in the component attributes. Java 2 Standard Edition was used for this application because there are very few open implementations of the Java Mobile Media API for the Connected Device Configuration of Java 2 Micro Edition. JOrbis occupies 169,454 bytes as an uncompressed jar file.

The component abstraction for JOrbis occupies 1,371 bytes as an uncompressed Java archive. It was written in 60 SLOC. It is outlined as a collection of SATIN components in Fig. 6.

A number of tests were conducted using the Ogg Vorbis codec component. The startup times needed for playing an audio stream via the componentized codec were measured and compared with those of the JOrbis stand-alone implementation. Moreover, the time needed to send the componentized codec from one node to another was measured. The figures are shown in Table 5.

The tests were run using the two laptops described above. They show that the overhead (which includes the full SATIN middleware running) is relatively small. Moreover, the time needed to send and deploy the LMU containing the componentized Ogg Vorbis codec is quite small. The sending of an LMU took considerably more time than its deployment. This is consistent with the results of the Program Launcher tests described above. Also note that the Ogg Vorbis component is a significantly more complex component than the sample component described above as it is composed of 50 classes. This explains why it takes much more time to receive and deploy.

2. The version of JOrbis used was 0.0.14.

TABLE 5
Quantitative Evaluation Results of Music Player

Component-Based Overhead (includes SATIN)	19000 bytes
Time to Send LMU:	2682 ms
Time to Deploy LMU	860 ms

The Music Player demonstrates an application that uses the container to listen to the arrival of new components and then adapts its interface and functionality to reflect the arrival of a new component. It also demonstrates reaction to context changes as the application monitors the discovery services for new codec components and schedules them for download as soon as they appear. The operation is transparent to the end user. Finally, it illustrates the ease with which existing code can be adapted to run under SATIN. The adapted code gains the ability to be deployed dynamically at runtime and to be used as part of a component-based application, such as the music player. Hence, it can express dependencies on other components or platforms, it can be required by other components, and a component-based application can be built with it. These abilities impose minimal overhead.

5.6 Media Transcoding for Satellite Active Networks

As part of the PANAMAS project [30], funded by the European Space Agency, we investigated the use of SATIN to implement an Active Network [31] platform for satellite systems. The scenario of the project is that media producers broadcast a single stream through the satellite system; active components can dynamically adapt the stream based on network conditions (e.g. packet loss), potentially offering different versions of the stream to users, depending on from which spot beam (or link) they are accessing the satellite. Hence, a lower-resolution version of the satellite data stream could be viewed on a PDA or a lower bit-rate version by users on a slow link, thus helping solve issues of network congestion and offering a good quality of service. Active components can be dynamically shipped and used to reprogram the satellite system for types of streams that have not been envisioned at design time. Given the longevity and cost of a satellite system, reprogrammability and adaptation are very desirable characteristics. It is also important to note that, because of durability and weight restrictions, the hardware capabilities of satellite systems are very limited.

Note that we did not have access to a real satellite—this project was a feasibility study for technologies for future satellites. As such, we created a testbed that involved a media producer, a satellite ingress host acting as the host that manages the satellite, and a host emulating the satellite itself. We then had a number of clients accessing media served by the producer, connected through the satellite on different spot beams. SATIN was running on the ingress and satellite nodes.

To emulate access to the satellite, we implemented an HTTP proxy as a Reflective SATIN component. The proxy is responsible for getting the content from the media producer and forwarding it to the requesting client. As it is a Reflective component, it can dynamically accept LMUs.

Thus, it allows an arbitrary number of transcoding components to be deployed into it. A transcoding component is a component that actively manipulates the multimedia stream by either recoding (for example, into a smaller resolution) or transcoding into a different format. A transcoder can thus be used to send different qualities of the same stream to different users. For example, a PDA user can receive a lower resolution video than a high definition television user, or a user on a congested link can receive a video at a lower bit rate (i.e., with more artifacts). We encapsulated the Java reference implementation of the JPEG 2000 encoder and decoder as a SATIN transcoding component. We also implemented a supervisor component, responsible for spawning multiple proxies on the satellite, as SATIN does not, by default, support multiple instances of components. The different spot beams were emulated using the Linux traffic conditioning framework, which allowed us to introduce delays and limit the bandwidth. An outline of the satellite platform, represented as a collection of SATIN components is presented in Fig. 7.

The satellite control platform deployed the supervisor and proxy components to the satellite dynamically. Two instances of the proxy were spawned, simulating client connection on two different spot beams, one slower than the other. The clients on the slow link experienced problems in viewing the media. The transcoder was dynamically deployed and reprogrammed the appropriate proxy to recode the stream to a lower bit rate. Hence, both spot beams allowed users to watch the stream, albeit one with lower quality.

The component encapsulation of the transcoder required 155 SLOC. The PANAMAS project showed the practicality of developing an Active Network system using SATIN. It demonstrated the utility of the abstractions defined by the metamodel, in dynamically reprogramming a network architecture to adapt in response to changes to link quality and perceived user experience. Hence, the purpose of this example was to illustrate the ease of adaptation of existing programs to run under SATIN and the ability of SATIN to be used to offer an active networking system—a system that is radically different from the examples described above.

6 Related Work

There is a substantial body of work on self-organizing, self-healing, and adaptable systems, component deployment and middleware systems. This section presents a comparison of the SATIN approach with related work in all three areas. Note that we will not extensively compare with advertising and discovery mechanisms; the purpose of the SATIN advertising and discovery framework and its implementation is to show how the metamodel can be used to build middleware-level services and how mobile code and components can be used to offer dynamicity in service discovery mechanisms, an area which is traditionally fixed and inflexible. The middleware systems with which this section compares SATIN were chosen because of their use of logical mobility or related adaptation primitives.

6.1 Advertising and Discovery Mechanisms

Universal Plug And Play (UPnP) [32] is a set of networking protocols that allows devices to describe, advertise, and control services using XML and SOAP. It also includes an event-based mechanism that allows devices to communicate state changes to each other. Implementations of the SATIN Advertising and Discovery framework can use UPnP to describe, advertise, and discover components that implement services. The developer will also have to implement the control and event-based communication mechanisms that UPnP provides. The SATIN migration primitives can be used to dynamically deploy a UPnP implementation to any SATIN-enabled node.

Web Services and the Web Services Description Language (WSDL) [33] can be used by SATIN advertisable components to describe services and allow clients to communicate with them. The clients of Web Services use XML and HTTP to convey requests to the service, the interface of which is described using WSDL. Note that both Web Services and UPnP would be more suitable service discovery and communication paradigms for hosts that do not change location very frequently.

The OMG Data Distribution Service (DDS) [34] is a QoS-aware data-centric public subscribe communication model for distributed nodes. Implementations of the SATIN Advertising and Discovery framework can use DDS to advertise and discover services. Moreover, components that offer data dissemination services can use DDS to distribute the data to subscribers. The SATIN logical mobility primitives can be used to dynamically deploy a DDS implementation to any SATIN-enabled node.

6.2 Component Systems

OpenCOM [26] is a lightweight component model based on Microsoft COM, the purpose of which is to implement a popular component model with reflective and adaptation capabilities. OpenCOM components export interfaces and have "required interfaces," or *Receptacles*. A Receptacle represents a dependency of a Component to another. OpenCOM is similar to SATIN in that it provides a lightweight component model with support for reflection. The OpenCOM Capsule is also similar to the SATIN Container. It does not, however support logical mobility and, therefore, has limited adaptability.

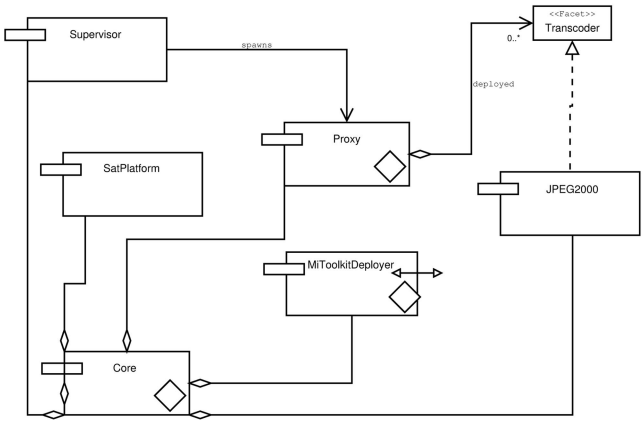


Fig. 7. The satellite platform as a collection of components.

The CORBA Component Model [19] (CCM) is an extension of the CORBA Object Model that appeared with the third version of the CORBA specification. It is a distributed and language-independent component model that allows multiple containers to host multiple components on each server. Components offer services expressed as interfaces and require other services as defined in receptacles. The CCM also provides an event-based communication system, an attribute system, and structural reflection capabilities. Components can be dynamically deployed into component servers. A number of services, such as transactions, security, persistence, etc., are offered.

Related to this is the CORBA Deployment and Configuration (D&C) [35] specification, which prescribes, in an MDA-compliant way, how CORBA Component Model systems can be reconfigured by dynamically deploying components or by choosing which component to use to perform a particular task at runtime. Deployment and configuration also allows applications to define themselves in terms of assemblies, which are component packages and assorted metadata. The D&C specification defines concepts similar to the SATIN Deployer and Logical Mobility Unit, which are able to serialize, send, receive, and deploy components. It offers various nonfunctional services, such as QoS support, hierarchical assemblies, etc.

There are two major differences between SATIN and the D&C specification. CORBA D&C does not allow for sending arbitrary classes and objects and, thus, cannot be used to offer the flexible use of any logical mobility paradigm as a computational primitive. In contrast, SATIN can deploy any class, object, or component to any Reflective target. Another difference is that implementations of the CCM and the D&C are too heavyweight to run on mobile devices, as will be discussed below. In contrast, SATIN has been implemented and is fully operational on 200 MHz ARM PDAs, and we have deployed a prototype for mote sensors.

The CORBA Component Model requires too many resources to be implemented on mobile devices. The Lightweight CORBA Component Model (LwCCM) was created as a subset of the full specification. It still offers point to point communication, event-based communication, and multiple container types. Despite the similarity of the CCM and, in particular, the LwCCM to SATIN, SATIN is fundamentally different in that it is a *local* component metamodel, with distribution built on top as a service, as justified in Section 3. The distributed nature, multiple interaction paradigms, and multiple container types offered by the LwCCM by default make it far more resource-demanding than SATIN by definition. As an example, CIAO [36], a popular implementation of the LwCCM, requires multiple megabytes to be installed with all its dependencies on a Debian Linux system.

Beanome [37] and Gravity [38] are component models built on top of the Open Services Gateway Initiative (OSGi) Framework [39]. OSGi is a commercial framework for the Java platform that allows service providers to deliver services to consumer devices attached to a residential network and to manage those devices remotely. Beanome and Gravity have been built for a different setting that can assume constant network connectivity at relatively low latency. This justifies a strict client/server architecture, which would not work in a mobile setting.

The Dynamically Programmable and Reconfigurable Software (DPRS) architecture [40] discusses a design for dynamic programmable and reconfigurable systems. The notion of a Micro-Building Block (MBB) is defined, which is, essentially, a minimal component that inputs data in the form of tuples, performs some action on it, and outputs a result in tuples. The state of the MBB is stored as tuples in a state storage area that is provided by the system. The main difference to SATIN is that the latter defines a way to dynamically reconfigure the system by sending and receiving components via the use of the Deployer and Reflective components. The DPRS architecture does not define any logical mobility mechanism. Finally, the DPRS implementation is rather heavyweight, as witnessed in its testing procedure that requires multi-GHz machines, whereas Section 5 shows that SATIN is lightweight.

DACIA [41] is an adaptable distributed component based system for groupware applications that allows for the reconfiguration of the system in the event of user mobility. The main differences to SATIN are SATIN's focus on device mobility and limited resource consumption. Moreover, SATIN component interconnections are local.

PCOM [42] is a distributed component model for pervasive computing. Built on top of BASE, a middleware system that allows for dynamically selecting communication protocol stacks, PCOM treats an application as a collection of potentially distributed components, which make their dependencies explicit. If those dependencies are invalidated, PCOM can attempt to automatically adapt by detecting alternatives according to various strategies.

The FarGo-DA [43] distributed component model uses logical mobility to allow disconnected operations. As such, when a FarGo component is disconnected, it has a number of options to allow the remote reference to remain valid. These options include cloning and replacing the reference. The SATIN component metamodel provides a much more general use of logical mobility primitives and focuses on the reconfiguration of autonomous hosts.

In [44], a software architecture-based, distributed component model is proposed that has the ability to update the components that constitute applications engineered using it. Using the concepts of components, which describe the logic and state of the system, connectors, which are responsible for interconnecting local and remote components, and configurations, which define topologies of components and connectors, the approach requires preloading of the software architecture skeleton (or metalevel configuration) on all hosts where the component-based application is to be deployed. SATIN offers more fine-grained use of Logical Mobility built into the model (whereas the approach in [44] can only send and receive components) and allows for reflection and late binding without requiring that any architecture description be preloaded on any node.

6.3 Middleware Systems

Lime [27] is a mobile computing middleware system that allows mobile agents to roam to various hosts sharing tuple spaces. PeerWare [45] allows mobile hosts to share data,

using logical mobility to ship computations to the remote sites that host the data. Jini [46] is a distributed networking system that allows devices to enter a federation and offer services to other devices or use code on demand to download code allowing them to utilize services that are already being offered. The Software Dock [47] is an agent-based software deployment network that allows negotiation between software producers and consumers. The one.world system [48] for pervasive applications facilitates dynamic service composition, migration of applications, and discovery of context, using remote evaluation and code on demand. A limitation of these approaches is that their use of logical mobility is focused to solving specific problems of a particular scope, such as data sharing, distributed computations, or disconnected operations. In contrast, SATIN allows for the flexible use of logical mobility by applications for any purpose. Moreover, these approaches are not suitable for heterogeneity and mobility as they usually predefine advertising and discovery services, making interoperability with different middleware systems and networks particularly difficult.

ReMMoC [49] is a middleware platform which allows reconfiguration through reflection and component technologies. It provides a mobile computing middleware system which can be dynamically reconfigured to allow the mobile device to interoperate with any middleware system that can be implemented using OpenCOM components. The generic request broker UIC [50] defines a skeleton of abstract components, which have to be specialized to the particular properties of each middleware platform the device wishes to interact with. The limitation of these approaches is that they do not provide adaptation primitives or the use of logical mobility primitives to the applications running on the middleware; they only allow for the reconfiguration of the middleware system itself.

7 Discussion and Conclusions

In this paper, we presented SATIN, a lightweight component metamodel instantiated as a middleware system for adaptable mobile systems. SATIN offers logical mobility primitives as first-class citizens. SATIN was used to create and port a number of applications that show different aspects of adaptation. The overhead of the SATIN implementation was shown to be minimal, despite the added flexibility.

SATIN was designed to be very flexible and lightweight. This flexibility was demonstrated in the breadth of different applications and systems that were developed using it. As such, it makes a number of design choices that affect this. To begin with, SATIN does not differentiate between different component instances by default. This behavior was found to be adequate in our testing. To support different instances, an extension of the container and default component life cycle would be needed and was built for the ESA project. Similarly, SATIN does not constrain adaptation in any way. This means that adaptation can potentially leave the system in an undesirable state (by removing, for example, the only Deployer). This will be addressed in future work via the use of component frameworks [6].

SATIN supports the construction of laissez-faire adaptable systems. It provides the architecture to engineer an adaptable system and the means to adapt it, but does not provide for the mechanism to select *how* a system should adapt (thus becoming an adaptation-aware system). To that end, we have created Q-CAD (QoS and Context Aware resource Discovery) [8], which employs application profiles and utility functions to decide on how to adapt.

Last, an issue that was not discussed in this paper was trust and security. At the current stage, our architecture provides for the use of digital signatures embedded in LMUs. This assumes the existence of a trusted third party, such as the ISP of the user. We consider it important in future work to investigate the use of Proof Carrying Code [51] techniques or even a trust-based mechanism [52], which may alleviate this need.

To conclude, SATIN provides a novel way to build adaptable mobile systems. The use of logical mobility, as offered by SATIN, allows a mobile system to adapt to changes in context. SATIN breaks the monolithic nature of mobile systems by representing them as collections of interacting collocated components, which can mutate using logical mobility techniques. Reflecting on the work presented, it can be concluded that mobile systems can benefit from the flexibility offered by component systems offering logical mobility primitives *without suffering significant performance penalties*. In particular, SATIN allows for building novel adaptable systems and also permits the porting of existing systems and applications, with the ports benefiting from the reconfiguration primitives that SATIN provides. The performance of an unoptimized version of SATIN running both new applications and ports of existing systems on mobile computing hardware that is several years old was found to be adequate and the time needed to adapt was measured to be minimal. An alternative approach to using components and logical mobility would be to create a programming language that allows the specification of modular systems but that also offers built-in logical mobility primitives. The major drawback of such an approach would be, however, that the complete software development chain, from the linker to the compiler to the development environment to the adaptable software itself, would have to be written from scratch. In contrast, SATIN allows the use of existing software and tools.

Moreover, the porting of the SATIN architecture to even smaller devices such as sensors is under consideration—the severely limited resource availability of a sensor platform offers a new set of research challenges. That said, we already have a prototype of the platform running as a Contiki [53] Service on the TelosB sensor platform. This is a testament to how implementations of the SATIN metamodel can scale down to devices that only offer 10 KB of RAM and an 8 MHz 16 bit CPU. We are planning to research further into code deployment consistency, dependency, and management issues, which are related to code mobility, the discarding of parts of a component received by a host, and the deployment of new code on a host which does not have the required functionality. We are also researching the use of model-driven architecture in specifying SATIN systems. Finally, SATIN is currently being used in the SEINIT [54]

and RUNES [55] EU projects. It has been released as an open source project under the GNU LGPL and is available online at [28].

ACKNOWLEDGMENTS

The authors thank Saleem Bhatti, Licia Capra, Wolfgang Fritsche, Gerhard Gessler, Stephen Hailes, Peter Kirstein, Karl Mayer, Lionel Sacks, and Mirco Musolesi for their help in producing this work. The authors also acknowledge the support of the EPSRC through project GR/R70460, the European Union through Project RUNES, and the European Space Agency through contract number 18376/04/NL/AD.

REFERENCES

[1] Palmsource Developers Program, http://www. palmsource.com/developers/, 2004.
[2] J. Power, "Distributed Systems and Self-Organization," *Proc. 1990 ACM Ann. Conf. Cooperation,* pp. 379-384, Feb. 1990.
[3] A. Fuggetta, G. Picco, and G. Vigna, "Understanding Code Mobility," *IEEE Trans. Software Eng.*, vol. 24, no. 5, pp. 342-361, May 1998.
[4] The Distributed.net Project, http://www.distributed.net, 1995.
[5] S. Zachariadis, C. Mascolo, and W. Emmerich, "SATIN: A Component Model for Mobile Self-Organisation," *On the Move to Meaningful Internet Systems 2004: Proc. CoopIS, DOA, and ODBASE,* pp. 1303-1321, Oct. 2004.
[6] C. Szyperski, *Component Software: Beyond Object-Oriented Programming.* Addison Wesley, 1998.
[7] M. Satyanarayanan, "Accessing Information on Demand at Any Location. Mobile Information Access," *IEEE Personal Comm.,* vol. 3, no. 1, pp. 26-33, Feb. 1996.
[8] L. Capra, S. Zachariadis, and C. Mascolo, "Q-CAD: QoS and Context Aware Discovery Framework for Adaptive Mobile Systems," *Proc. Int'l Conf. Pervasive Services (ICPS '05),* pp. 453-456, July 2005.
[9] T. Kort and K. Dulaney, "Sony's Exit from the PDA Market Hurts PalmSource," June 2004.
[10] W. Emmerich, *Engineering Distributed Objects.* John Wiley & Sons, Apr. 2000.
[11] *Object Management Architecture Guide,* R.M. Soley, ed. Wiley, 1992.
[12] R. Grimes, *DCOM Programming.* Wrox, 1997.
[13] R. Monson-Haefel, *Enterprise Javabeans.* 1999.
[14] C. Mascolo, L. Capra, and W. Emmerich, "Principles of Mobile Computing Middleware," *Middleware for Comm.,* Q. Mahmoud, ed., pp. 261-280, John Wiley, 2004.
[15] B.C. Smith, "Reflection and Semantics in a Procedural Programming Language," PhD thesis, Massachusetts Inst. of Technology, Jan. 1982.
[16] "Meta Object Facility (MOF) Specification," technical report, Object Management Group, Mar. 2000.
[17] "Unified Modeling Language,"version 1.5, Object Management Group, http://www.omg.org/docs/formal/03-03-01.pdf, Mar. 2003.
[18] S. Zachariadis, "Adapting Mobile Systems Using Logical Mobility Primitives," PhD thesis, Univ. of London, 2005.
[19] "CORBA Component Model," Object Management Group, http://www.omg.org/cgi-bin/doc?orbos/97-06-12, 1997.
[20] I. Murdock, "Overview of the Debian GNU/Linux System," *Linux J.,* vol. 6, Oct. 1994.
[21] G.P. Picco, "μCode: A Lightweight and Flexible Mobile Code Toolkit," *Proc. Second Int'l Workshop Mobile Agents,* K. Rothermel and F. Hohl, eds., 1998.
[22] J. Waldo, "The Jini Architecture for Network-Centric Computing," *Comm. ACM,* vol. 42, no. 7, pp. 76-82, July 1999.
[23] M.V. Zelkowitz and D. Walace, "Experimental Validation in Software Engineering," *Proc. Conf. Empirical Assessment and Evaluation in Software Eng.,* Mar. 1997.
[24] M. Ijaha, "Mitoolkit," MS thesis, Univ. College London, U.K., 2004.
[25] D.A. Wheeler, "SLOCCount," 2004.
[26] M. Clarke, G.S. Blair, G. Coulson, and N. Parlavantzas, "An Efficient Component Model for the Construction of Adaptive Middleware," *Proc. IFIP/ACM Int'l Conf. Distributed Systems Platforms,* pp. 160-178, 2001.
[27] A.L. Murphy, G.P. Picco, and G.-C. Roman, "Lime: A Middleware for Physical and Logical Mobility," *Proc. 21st Int'l Conf. Distributed Computing Systems (ICDCS 21),* pp. 368-377, May 2001.
[28] The SATIN Open Source Project, "The SATIN Component Model," http://satin.sourceforge.net/, 2005.
[29] The OGG Vorbis Project, Xiph.org Foundation, http://xiph.org/ogg/vorbis/, 1998.
[30] The European Space Agency, "Programmable Active Networks for Next Generation Multimedia Services (PANAMAS)," 2005.
[31] D.L. Tennenhouse, J.M. Smith, W.D. Sincoskie, D.J. Wetherall, and G.J. Minden, "A Survey of Active Network Research," *IEEE Comm. Magazine,* vol. 35, no. 1, pp. 80-86, 1997.
[32] "Universal Plug and Play," UPnP Forum, http://www.upnp.org/, 1998.
[33] R. Chinnici, M. Gudgin, J.-J. Moreau, and W. Weerawarana, *Web Services Description Language (WSDL) 1.2.* World Wide Web Consortium, Mar. 2003.
[34] G. Pardo-Castellote, "OMG Data-Distribution Service: Architectural Overview," *Proc. 23rd Int'l Conf. Distributed Computing Systems (ICDCSW '03),* p. 200, 2003.
[35] "The CORBA Component Model Deployment & Configuration Specification," Object Management Group, http://www.omg.org/cgi-bin/doc?ptc/2005-01-07, 2004.
[36] D.C. Schmidt, "Component-Integrated Ace ORB," http://www.cs.wustl.edu/%7Eschmidt/CIAO.html, 2006.
[37] H. Cervantes and R. Hall, "BEANOME: A Component Model for the OSGi Framework," *Software Infrastructures for Component-Based Applications on Consumer Devices,* Sept. 2002.
[38] H. Cervantes and R. Hall, "Autonomous Adaptation to Dynamic Availability Using a Service-Oriented Component Model," *Proc. 26th Int'l Conf. Software Eng. (ICSE '04),* pp. 614-623, May 2004.
[39] The OSGi Framework, OSGi Alliance, http://www.osgi.org, 1999.
[40] M. Roman and N. Islam, "Dynamically Programmable and Reconfigurable Middleware Services," *Proc. Middleware Conf.,* pp. 372-396, Oct. 2004.
[41] R. Litiu and A. Parakash, "Developing Adaptive Groupware Applications Using a Mobile Component Framework," *Proc. 2000 ACM Conf. Computer Supported Cooperative Work (CSCW '00),* pp. 107-116, 2000.
[42] C. Becker, M. Handte, G. Schiele, and K. Rothermel, "PCOM—A Component System for Pervasive Computing," *Proc. Second Int'l Conf. Pervasive Computing and Comm.,* pp. 67-76, Mar. 2004.
[43] Y. Weinsberg and I. Ben-Shaul, "A Programming Model and System Support for Disconnected-Aware Applications on Resource-Constrained Devices," *Proc. 24th Int'l Conf. Software Eng.,* pp. 374-384, May 2002.
[44] M. Mikic-Rakic and N. Medvidovic, "Architecture-Level Support for Software Component Deployment in Resource Constrained Environments," *Proc. IFIP/ACM Working Conf. Component Deployment (CD '02),* pp. 31-50, 2002.
[45] G. Cugola and G. Picco, "Peer-to-Peer for Collaborative Applications," *Proc. IEEE Int'l Workshop Mobile Teamwork Support/Int'l Conf. Distributed Computing Systems (ICDCS '02),* pp. 359-364, July 2002.
[46] K. Arnold, B. O'Sullivan, R.W. Scheifler, J. Waldo, and A. Wollrath, *The Jini(TM) Specification.* Addison-Wesley, 1999.
[47] R.S. Hall, D. Heimbigner, and A.L. Wolf, "A Cooperative Approach to Support Software Deployment Using the Software Dock," *Proc. 1999 Int'l Conf. Software Eng.,* pp. 174-183, 1999.
[48] R. Grimm, T. Anderson, B. Bershad, and D. Wetherall, "A System Architecture for Pervasive Computing," *Proc. Ninth ACM SIGOPS European Workshop,* pp. 177-182, 2000.
[49] P. Grace, G.S. Blair, and S. Samue, "Middleware Awareness in Mobile Computing," *Proc. First IEEE Int'l Workshop Mobile Computing Middleware (MCM '03)/Int'l Conf. Distributed Computing Systems (ICDCS '03),* pp. 382-387, May 2003.
[50] M. Roman, F. Kon, and R.H. Campbell, "Reflective Middleware: From Your Desk to Your Hand," *IEEE Distributed Systems Online J.,* special issue on reflective middleware, July 2001.
[51] G.C. Necula, "Proof-Carrying Code," *Proc. 24th ACM SIGPLAN/SIGACT Symp. Principles of Programming Languages,* pp. 106-119, Jan. 1997.

[52] L. Capra, "Engineering Human Trust in Mobile System Collaborations," *Proc. SIGSOFT/12th Int'l Symp. Foundations of Software Eng. (FSE-12),* pp. 107-116, Nov. 2004.
[53] A. Dunkels, B. Groenvall, and T. Voigt, "Contiki—A Lightweight and Flexible Operating System for Tiny Networked Sensors," *Proc. First IEEE Workshop Embedded Networked Sensors,* Nov. 2004.
[54] The SEINIT Project: Security Expert Initiative, http://www.seinit.org, 2003.
[55] The RUNES Project: Reconfigurable Ubiquitous Network Embedded Systems," http://ist-runes.org, 2004.

Stefanos Zachariadis received the PhD in computer science from University College London. During his time there, he published in the area of data synchronization, peer to peer systems, mobile computing and midleware, mobile code, and sensor systems. His research has led him to participate in various research projects, and his work has been used by various parties, including the European Space Agency and sensor network operating system developers. His interests are in the areas of software architectures, testing, and mobile, distributed, and resource constrained systems. He currently works as a software engineer for the Zühlke Technology Group in London. More details of his profile are available at www.zachariadis.net.

Cecilia Mascolo received the MSc and PhD degrees in computer science from the University of Bologna (Italy). She is an EPSRC Advanced Research Fellow and a senior lecturer with the Department of Computer Science at University College London. She has published extensively in the areas of mobile middleware, delay tolerant routing, ad hoc networks, mobility models, software architectures for ubiquitous systems, and code mobility. Dr. Mascolo is currently working on projects in middleware for mobile and sensor networks, delay tolerant and opportunistic networking, publish-subscribe systems, and sensor networks. She is an investigator on projects in mobile computing middleware, pervasive middleware for health care, middleware for emergency applications, and mobility models. Dr. Mascolo has served as a program committe member in many middleware, mobile system, and delay tolerant network conferences and she has been cochair of a number of workshops and conferences focusing on mobile systems. She also delivered tutorials on mobile computing middleware. More details of her profile are available at www.cs.ucl.ac.uk/staff/c.mascolo.

Wolfgang Emmerich received the PhD in computer science from the University of Paderborn and an MSc in informatics from the University of Dortmund in Germany. He is a professor of distributed computing at University College London, a leading British university. He heads the Software Systems Engineering Research Group in the Department of Computer Science, where he is currently also director of research. He is a member of London Software Systems. Prior to joining UCL, he held a lectureship at the City University in London and was a visiting research fellow in the Software Verification Research Centre at the University of Queensland in Brisbane, Australia. His research interests are in the area of software architectures for large-scale distributed and mobile systems. He is a member of the editorial board of the *IEEE Transactions on Software Engineering*. He has served on numerous program committees of international conferences in software engineering and distributed systems. He currently serves as program cochair of the International Conference on Software Engineering, to be held in Minneapolis in 2007. He is a chartered engineer and a member of the IEEE Computer Society, the ACM, and the Institution of Engineering and Technology. He is also a cofounder and partner of the Zühlke Technology Group, a medium-sized pan-European service provider of systems engineering services.