SATIN: A Component Model for Mobile Self Organisation

Stefanos Zachariadis, Cecilia Mascolo and Wolfgang Emmerich

Dept. of Computer Science, University College London
Gower Street, London WC1E 6BT, UK
{s.zachariadis,c.mascolo,w.emmerich}@cs.ucl.ac.uk

Abstract. We have recently witnessed a growing interest in self organising systems, both in research and in practice. These systems re-organise in response to new or changing conditions in the environment. The need for self organisation is often found in mobile applications; these applications are typically hosted in resource-constrained environments and may have to dynamically reorganise in response to changes of user needs, to heterogeneity and connectivity challenges, as well as to changes in the execution context and physical environment. We argue that physically mobile applications benefit from the use of self organisation primitives. We show that a component model that incorporates code mobility primitives assists in building self organising mobile systems. We present SATIN, a lightweight component model, which represents a mobile system as a set of interoperable local components. The model supports reconfiguration, by offering code migration services. We discuss an implementation of the SATIN middleware, based on the component model and evaluate our work by adapting existing open source software as SATIN components and by building and testing a system that manages the dynamic update of components on mobile hosts.

1 Introduction

The recent advances in mobile computing hardware, such as laptop computers, personal digital assistants (PDAs), mobile phones and digital cameras, as well as in wireless networking (with UMTS, Bluetooth and 802.11), deliver sophisticated mobile computing platforms. We are observing a further and rapid decentralisation of computing, with computers becoming increasingly capable, cheaper, mobile and even fashionable personal items. Mobile computers are exposed to a highly dynamic context and can connect to information on different networks through wireless links.

Mobile computing systems are *highly dynamic systems*. They dynamically form networks of various different topologies, they are heterogeneous both on the software and hardware layers, and resource constrained and are exposed to a dynamic environment. Consequently, the requirements for applications deployed on a mobile device are a moving target: The context in which a mobile application is embedded can be highly dynamic and changes in the environment may require changes to the application (such as integration with a new service).

The current state-of-practise for developing software for mobile systems offers little flexibility to accommodate such heterogeneity and variation. Currently, application developers have to decide at design time what possible uses their applications will have and the applications do not change or adapt once they are deployed on a mobile host. In fact, mobile applications are currently developed with monolithic architectures, which are more suitable for a fixed execution context.

We argue that more flexible solutions are required that empower applications to automatically adapt to changes in the environment and to the users' needs. Power [33] 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 system lends itself to the biological metaphor in that the system entities and their relationships need to self organise in order to accommodate the new requirements."

Along those lines, we define a *self organising system* as a system which is able to adapt to accommodate changes to its requirements. As a highly dynamic system, a mobile system will encounter changes to its requirements; We therefore argue that mobile systems can benefit from the usage of primitives for self organisation. However, the literature on self organising systems largely focuses on the application of genetic algorithms, expert and agent-based systems [33, 32, 20]. Other approaches focus on using self organising primitives for reliability and service availability of legacy systems [21]. These approaches tend to be heavyweight and appear unsuitable for mobile applications as they are executed on hosts that are by orders of magnitude more resource-scarce than the fixed systems for which these self organisation primitives have been devised.

In this work, we exploit logical mobility and components to offer self organisation to mobile systems. Logical Mobility is defined as the ability to ship part of an application or even to migrate a complete process from one host to another. Logical mobility primitives have been successfully used to enhance a user's experience (Java Applets), to dynamically update an application (Anti-Virus software etc.), to utilise remote objects (RMI [40], Corba [28], etc), to distribute expensive computations (Distributed.net [34]) etc. Component Models on the other hand, argue for the decoupling of a system into a set of interacting components with well defined interfaces. Components promote decomposition and reusability of software. There are numerous component models already developed and discussed in the literature [38, 36, 22, 29], offering various services such as transactions and concurrency control and which have been used to represent systems as a collection of either local or remote components.

The novel contribution of this paper is threefold: We argue for the advantages that self organisation brings to mobile computing and how this compares to other approaches. We develop and discuss a lightweight component model that uses logical mobility to offer self organisational abilities to mobile systems. Finally, we introduce an implementation of the SATIN middleware based on the component model and evaluate it, by converting existing open source projects into SATIN components and by developing a component deployment and update system for mobile hosts. The applications show system adaptation responding to context changes, demonstrate end-user ease of use and show how new functionality can be integrated into the system.

This paper is structured as follows: Section 2 presents the motivation for our work and gives some background into our area of research. Section 3 describes the SATIN component model and middleware. Section 4 evaluates our system, while Section 5 is a critical summary of related work. Finally, Section 6 concludes the paper, giving some ideas for future research.

2 Background and Motivation

2.1 Motivating Example

This section presents an industrial example in order to motivate our work. We give an overview of how it works, highlight its limitations and describe how a self organising approach based on components and the systematic use of logical mobility primitives can help in overcoming them.

Case Study: Industry State-of-the-Art mobile application development PalmOS [31] is the most widely used operating system for PDAs; it powers more than 30 million devices worldwide, including mobile phones, GPS receivers, PDAs and sub notebooks. For example, a popular device running PalmOS has 64MB of RAM (which is used both as storage and heap memory), Bluetooth, infrared and 802.11 wireless networking and wired (serial) networking interfaces, as well as a 400MHz ARM processor. 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) and application data can be stored in Palm Databases (PDBs). The operating system allows for limited use of libraries. Applications are identified by a unique 4 byte identifier, the Creator ID. Developers register Creator IDs for each individual application with the operating system vendor. A PalmOS device usually ships with personal information management (PIM) software installed. Installing new applications requires 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. Statistics show that users rarely install any 3rd party applications, even though there is a plethora available.

This model has various disadvantages: there is very 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 fact that users rarely install any third party applications is usually attributed to the fact that it is difficult to do so. Palm-based computers can be deployed in both a nomadic and ad hoc networking settings. The potential for interaction with their environment is great, however PalmOS does not provide any primitives to do this. The result is that PalmOS based PDAs are still seen as stand-alone independent devices which interact mainly with a desktop computer to synchronise changes to shared data - interaction with their environment and peers is either not considered or is very limited.

A component based approach using logical mobility primitives would have several advantages:
- Representation of applications as interoperable components allows for updating individual parts.
- Componentisation promotes code reusability, preserving the limited resources of mobile devices.
- Logical mobility primitives allow for transferring components existing in any host that is in reach, in a peer to peer fashion. This makes application installation and updating easier.
- A component model can provide higher level interaction and communication primitives between components, located either on the same or on different hosts.

Please note that in other, less popular PDA operating systems, such as Windows CE and Linux, the use of components is more prevalent, especially by aspects of the operating system. However, most of the problems outlined above are still relevant, as those devices also do not interact with their environment, applications are usually monolithic, not taking advantage of the component mechanism offered and the use of logical mobility primitives is not provided.

2.2 Logical Mobility and Components for Self Organisation

Logical mobility is defined as the ability to move parts of an application or migrate a complete process from one processing environment to another. Commonly implemented using *code mobility* [12] techniques, information transfered can include binary code, compiled for a specific architecture, interpreted code, 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. We define a *Logical Mobility Unit* (LMU), as a container that can encapsulate any combinations of binary or interpreted code and application data and that can be serialised on one host, or *execution environment*, transfered to another one and get deserialised and used there. As such, logical mobility primitives can be expressed by *composing* the LMU, *transferring* it from one execution environment to another, and then *deploying* it. The execution environments can range from different physical nodes, to different sandbox processes residing on the same host.

In the work presented in this paper, we use logical mobility to assist in the construction of self organising systems because:

- Logical mobility allows applications to update their codebase, thus acquiring new abilities.
- Logical mobility permits interoperability with remote applications and environments, which have not been envisioned at the design time.
- Logical mobility potentially achieves the efficient use of peer resources, as computationally expensive calculations can be offloaded to the environment.
- Logical mobility facilitates the efficient use of local resources, as infrequently used functionality can be removed to free some of the limited memory that mobile devices are equipped with, to be retrieved later when needed.
- Functionality acquired by a self organising system can be represented as an LMU, transferable to other hosts.

Components, Distribution and Collocation: Although component based systems are widely used in business client/server type applications, as well as in desktop systems [13], their use in mobile devices has been very limited. Section 2.1 has shown some of the limitations of current approaches; namely, mobile systems are monolithic and fail to interact with their environment and to adapt to changes to it.

We believe that logical mobility should be combined with a component-based approach to structure systems for the following reasons:
- Components break the monolithic structures that currently prevail in mobile systems by promoting the decomposition of applications into interacting components.
- Components logically structure applications into distinct units, which can be moved around the network. As such, they can provide a coarse-grained guide onto how a system can reorganise.
- Components encourage reusability, which is particularly important in the resource constrained settings of mobile devices. For example, a component that implements a compression algorithm can be reused by multiple applications.
- By representing a system as a set of interoperable components, logical mobility primitives allow us, in principle, to dynamically add, remove and replace components, thus adapting the overall behaviour of the system.

At first glance, it would seem that distributed component models are ideal for mobile devices, since they already address issues of heterogeneity, which are inherent in mobile computing. The comparison between distribution and collocation for object systems is made in [9]. There are three main reasons for which we argue that component model systems that enforce distribution are unsuitable for mobile devices:
Size: mobile devices have very limited resources. Distributed component model implementations usually require large amounts of memory and significant CPU power to deliver functionalities such as transactions, persistence and concurrency control, which are not considered essential in a mobile setting.
Network Connectivity: A reference to a component in a local, shared memory system, is usually a pointer, which is a lightweight data structure. In distributed systems however, the reference is usually a more substantial data structure, that

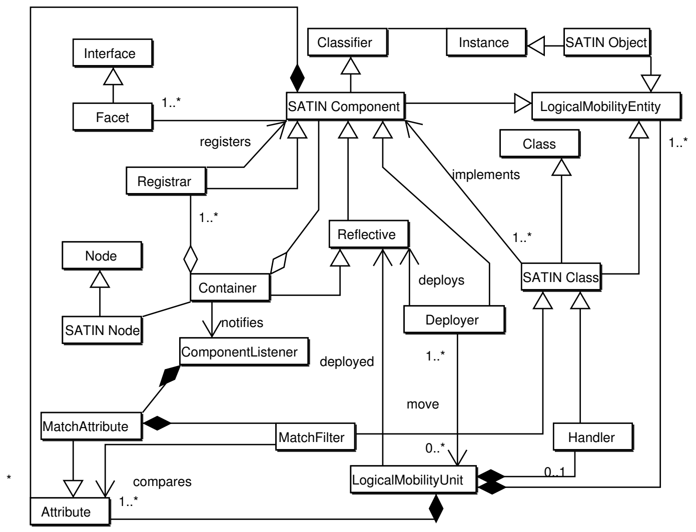

Fig. 1. The SATIN Meta Model. Note that Node, Classifier, Class, Interface and Instance are taken from UML.Core.

encodes location and security information. The process of calling a function in a distributed object, involves marshaling and unmarshaling both request and reply. Most distributed component model implementations assume continuous network connection with a high bandwidth and low latency to deliver synchronous remote procedure calls. On the other hand, mobile devices usually have intermittent network connectivity at low bandwidth and high latency. Invalidating those assumptions usually implies invalidating the remote component reference. As such, network references are often unsuitable for mobile applications, not providing for system autonomy when invalidated.
Complexity: Distributed component models usually assume a client / server architecture, with a predictable number of clients accessing one or more servers. An individual application is seen as a collection of components distributed in a predictable number of potentially heterogeneous devices. The physical mobility and temporal nature of the networking connectivity of mobile devices means that the devices form highly dynamic networks, which may even be completely structureless (ad hoc). Even when the latter is not the case, mobile devices form significantly less predictable topologies than standard distributed systems. Given this, mobile applications are hardly comparable to standard distributed systems, in terms of structure and complexity.

The next section presents the SATIN component model and its middleware system instantiation, which tackles the problems outlined above. A more thorough comparison of our model and traditional distributed component models can be found in Section 5.

3 SATIN

3.1 Component Model Overview

The SATIN component model is a local component model, targeting mobile devices, that uses logical mobility primitives to provide distribution services; Instead of relying on the invocation of remote services via the network, the component model supports the cloning and migration of components between hosts, providing for system autonomy when network connectivity is missing or is unreliable. 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 SATIN component model 1 is a Meta Object Facility [15]-compliant extension of the UML [27] meta model. We build upon and extend the concepts of Classifier, Interface, Class, Instance, and Node. The most novel aspect of the model is the way it offers distribution services to local components, allowing instances to dynamically send and receive components at runtime. We are now going to describe the model in detail as well as our implementation of the SATIN middleware system.

3.2 Components

A SATIN *component* encapsulates particular functionality, such as, for instance, a user interface, an advertising mechanism, a service, an audio codec or a compression library. SATIN components separate interfaces and implementations. A component has one or more interfaces, called *facets*, with each facet offering any number of operations. The SATIN component model does not support abstract components (which cannot be instantiated) as the objective of abstract components can also be achieved using facets. The component implementation is achieved by one or several SATIN classes.

Component Metadata Although the SATIN component model is a local one, it is used to represent a largely heterogeneous set of devices and architectures. As such, the SATIN component abstraction must be rich enough to be able to describe components that will be deployed over a large number of platforms. To this end, we draw parallels with the Debian Project's [23] package system. Debian is an operating system the packages of which are deployed over twelve different hardware architectures – a Debian system may run the Linux, Hurd, NetBSD or FreeBSD kernels; it is composed of hundreds of different installable packages, most of which have various inter-dependencies, to create a complete system. The Debian package format uses metadata to describe the heterogeneity of these platforms. SATIN follows a similar approach using attributes to describe a component.

A SATIN *attribute* is a tuple containing a *key* and a *value*. Attributes can be immutable. The set of attributes for a component is not fixed, but can be extended. SATIN requires that each component has an ID attribute, that acts as a component identifier, similar to the PalmOS Creator ID (see Section 2.1) and a VER attribute, which denotes the version of the component implementation. As such, a component implementation is uniquely identified using the ID and VER attributes. A SATIN component can also depend on other components. These dependencies are expressed as a component attribute.

Components and Containers The central component of every instance of SATIN is the *container* component. A container is a component specialisation that acts as a registry of components that are installed on an instance of SATIN. As such, a reference to each component is available via the container. Components can query the container for components satisfying a given set of attributes.

SATIN components can register listeners with the container to be notified when components satisfying a set of attributes given by the listener is added or removed. When querying the container or notifying listeners, satisfiability of the given set of attributes is verified by a *Match Filter*, which is a SATIN interface implemented by the listener. As such, satisfiability verification is highly customisable and allows for complex semantics for matching component implementations based on attributes. This allows components to react to changes in local component availability. For example, media player applications can be notified when components implementing the `AUDIOFORMAT` facet are installed in the system (see Section 4).

The container delegates registration and de-registration of components to one or more *registrars*. A registrar, which is also a component specialisation, is responsible for loading the component, validating its dependencies and adding it 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. Different registrars can have different policies on loading and removing components (from different sources for example) and verifying that dependencies are satisfied. For example, we have developed an implementation of the container and registrar, that keeps track of how often components are used - This frequency based approach is used to drop least used components when the system runs out of memory.

SATIN does not allow for the existence, on the same instance, of two components with the same identifier, unless they are two different versions of the same component implementation. As such, instances of SATIN can host different versions of the same component.

Distribution and Logical Mobility SATIN provides for the reconfiguration of applications via the use of logical mobility primitives. Distribution is not built into the components themselves, as SATIN is a local component model, but it is provided by the model as a service. This allows SATIN instances to dynamically send and receive components. We define a Logical Mobility Entity (LME), as a generalisation of a SATIN object, class, or component. As such, a SATIN *LMU*, as defined in Section 2.2, is a container, which is able to store arbitrary numbers of Logical Mobility Entities (see Figure 1 for the representation of these relationships). An LMU can therefore be used to represent various granularities of logical mobility, from complete applications and components, to individual classes and objects. The LMU has a set of attributes, the *LMU properties*, which are the union of the attributes of its contents. The LMU properties set is extensible. An LMU is always deployed in a *Reflective* component, a component specialisation that can be *adapted* at runtime, i.e., can receive new code or application data from the SATIN migration services. By definition, the container is always a reflective component, as it can receive and host new components at runtime.

The LMU has two special attributes, *TARG*, which specifies the intended recipient host and *LTARG*, which specifies the reflective component in the TARG host the LMU is going to be deployed into. A reflective component may inspect an LMU before accepting or rejecting it. Moreover, it can also accept parts of it and reject others. An LMU can optionally contain a *Handler* class, which can be instantiated and used by the receiver to automatically deploy the LMU to the reflective component; This mechanism can be used if the latter lacks knowledge of how to deploy and use the unit received. Finally, an LMU can be digitally signed.

A component cannot send an LMU directly. The functionality of sending, receiving and deploying components is abstracted and handled by the *Deployer*. The Deployer is a SATIN component specialisation that manages requesting, creating, sending, receiving and deploying LMUs to the appropriate reflective components. A Deployer is directly accessible to any component through the container.

When sending an LMU, a Deployer will reject any requests to send LMUs that do not have a TARG or LTARG attributes. Otherwise, it is responsible for serialising and sending the LMU to the Deployer component instance located at TARG. When receiving an LMU, the Deployer uses the container to verify that the component identified by LTARG exists in the local SATIN instance and that it is a reflective component. The LMU is then moved to the component it is destined to (identified by the LTARG attribute), which has the option of inspecting the component before deployment; The inspection results either in *full acceptance*, which means that the content of the LMU is accepted; *partial acceptance*, which means that parts of the LMU are accepted and others discarded; *rejection*, which means that the LMU is rejected and dropped; *Handler instantiation*, which means that the reflective component instantiates the Handler, encapsulated in the LMU, to perform the deployment. The result is determined by the reflective component, based on the content of the LMU.

A Deployer also listens to requests for components from other hosts. Upon receiving a request, the deployer checks if the component exists in the host. If it does, it clones it, encapsulates it in an LMU, serialises it and sends it to a Deployer instantiated at the requester.

Implementations of the Deployer could check for digital signatures and verify that an LMU is not malicious. This is further discussed in Section 6.

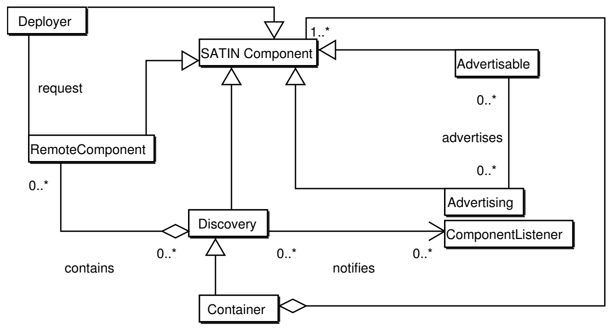

Fig. 2. The SATIN middleware system advertisement and discovery framework.

3.3 The SATIN Middleware

This section describes a possible middleware system instantiating the SATIN component model. The middleware, as well as any services and any applications deployed over it, are represented as collections of SATIN components, which can be dynamically added and removed. In particular, this section discusses advertising and discovery, which are represented as components themselves.

Advertisement and Discovery Services One of the pivotal requirements of mobile and pervasive computing, is the ability to reason about the environment. The environment is defined as the network of devices that can, at a specific point in time, communicate with each other. The devices can be both mobile and stationary - with the presence of mobile devices, however, the environment can be rapidly changing. In order to self organise, a mobile application needs to be able to detect changes to its environment. As the device itself is also part of that environment, it needs to advertise its presence. A mobile device, however, may be able to connect to different types of networks, either concurrently or at different times, with different hardware. There are many different ways to do advertisement and discovery. Imposing a particular advertisement and discovery mechanism can hinder interoperability with other systems, making assumptions about the network, the hosts and even the environment, which may be violated at some stage or not be optimal in a future setting - something which is likely to happen, given the dynamicity of the environment.

From the point of view of SATIN, the ability to reason about the environment is translated into the ability to discover components currently in reach and to advertise the components installed in the local system. Components that wish to advertise their presence in the environment are *advertisable* components. Examples include codec repositories, services, etc. An advertisable component provides a message that is used for advertising. An advertising technique is represented by an *advertiser* component. An advertiser component is responsible for taking 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. As such, an advertisable component can register with the container to be notified when new advertisers are added to the system. It can then register to be advertised by them.

Similarly, discovery techniques are encapsulated using *discovery* components. There can be any number of discovery components installed in a system. A discovery component is a registry of advertisable components located remotely. A *Remote Component* cannot directly be used by local components. It only provides methods to access its attributes, location and advertising message. Discovery components emit events representing components found remotely. Local components can register listeners with a discovery component, to be notified when components satisfying a given set of attributes are located or are out of reach. Satisfiability is verified using match filters. For the time being, we have implemented match filters for "greater than" and "lesser than" for numerical values, exact matching and string matching based on regular expressions.

Given the similarities between the container and a discovery component, the container is a specialisation of a discovery component in the SATIN middleware system, as it "discovers" components located and registered locally. This is shown in Figure 2.

4 Implementation and Evaluation

SATIN has been implemented using Java 2 Micro Edition (Connected Device Configuration, Personal Profile) [41]. It occupies 84 kilobytes, as a compressed Java archive, and includes a deployer implementation, multicast and centralised publish/subscribe advertising and discovery components and numerous match filters. We have used SATIN to implement the following:

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 components installed. 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 is registered. The dynamic program launcher offers the following services: Using a deployer, it can install any component from any discoverable source (through any discovery service). Figure 3 shows the Launcher displaying the components that are currently advertised by hosts in reach and installing one with identifier `STN:TESTAPP`. More-

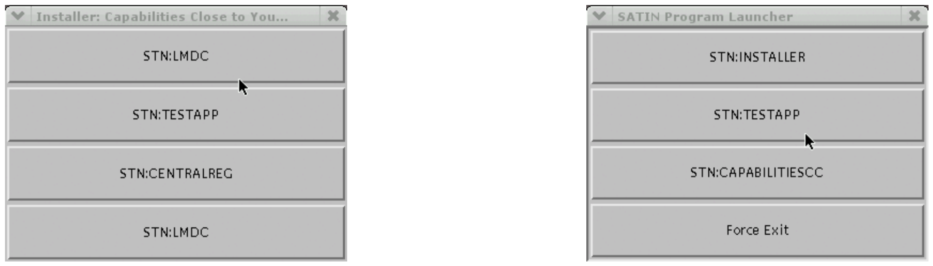

Fig. 3. (a) Showing what components are advertised on all networks, including those of the local host. (b) Component "STN:TESTAPP" was installed from a remote host and is displayed by the Launcher.

over, using the same mechanism, it can update the components installed in the system, either transparently or as a result of a user command. We deployed an implementation of the container, that monitors the usage of the components installed: If the device running the Launcher runs out of resources, it can delete unused components based on their frequency of use. The SATIN Launcher is implemented as a collection of interdependent components.

We have tested the application with three devices: a PDA equipped with an 802.11b card in ad hoc mode, a laptop equipped with an 802.11b card (again in ad hoc mode) and an Ethernet card, and a desktop with an Ethernet card. As such, the laptop could communicate with both the desktop and the PDA, whereas the PDA and the desktop could only communicate with the laptop. All three machines were running Linux. The PDA specifically, was running a beta version of JDK-1.3, with no Just In Time (JIT) compilation.

The laptop and PDA used the multicast advertising and discovery service to communicate over the wireless network, whereas the laptop and the desktop used the centralised advertising and discovery services over Ethernet. In our tests, the desktop was advertising the availability of version 2 of a component with identifier STN:TESTAPP, version 1 of which was installed on the PDA. The laptop installed version 1 of the component from the PDA and updated it to version 2 from the desktop. The PDA then discovered the availability of version 2 on the laptop and updated its copy. The table below shows the Java heap memory usage and the startup time for the Launcher on the PDA, the time it took for STN:TESTAPP to be installed from the PDA to the laptop, the time it took for the laptop to update STN:TESTAPP to version 2 from the desktop and the time it took for the PDA to update to version 2 from the laptop.

Startup Time on PDA	21 seconds
Memory Usage on PDA	1155KB
Time to install component from PDA to Laptop:	1998ms
Update time from Desktop to Laptop	1452ms
Update time on from Laptop to PDA	2063 ms

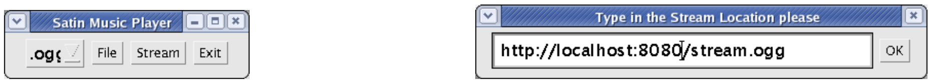
Satin Music Player
.ogg
File
Stream
Exit
Type in the Stream Location please
http://localhost:8080/stream.ogg
OK

Fig. 4. The SATIN Music Player.

The results obtained above show that the system implementation is reasonably lightweight. The components that make up the launcher occupy 22 kilobytes as a compressed Java archive. Please note that SATIN is not optimised yet. Moreover, note that as the container allows for multiple version of the same component implementation, updating does not break references to the previous version of a component implementation. We attribute the large time difference between the tests when the PDA was involved (installation time from the PDA to the laptop and update time from the laptop to the PDA) and when it was not (update time on from the desktop to the laptop) to the fact that the PDA runs a beta version of an interpreted JVM and to the nature of the wireless network that was used. We attribute the time difference between installing from the PDA to the laptop and updating from the laptop to the PDA to the fact that the PDA discovery component had to discover the updated version of the component in reach.

The SATIN **Music Player:** We have implemented a simple music player for SATIN. Components that implement audio codecs have the `AUDIOFORMAT` attribute defined. As such, the Music Player uses the notification service to be notified whenever a component that has this attribute implemented is registered. Moreover, it uses the deployer and the discovery components to download any codecs that are found remotely. The application itself occupies 3.6 kilobytes as a compressed Java archive.

We have also adapted JOrbis [18], an open source Ogg Vorbis [11] implementation to run as a SATIN audio codec component. As such, we are able to send and receive either the music player application or the audio codec. The application is automatically notified when the component is found and adapts its interface appropriately. The JOrbis component occupies 105 kilobytes as a compressed Java archive. Please note that the Music Player application is a Java 2 Standard Edition application. This is denoted in the component attributes. We used Java 2 Standard Edition, because there are very few open implementations of the Java Mobile Media API [39] for the Connected Device Configuration of Java 2 Micro Edition.

The Music Player demonstrates an application that uses the container to listen to the arrival of new components, adapting its interface and functionality upon new component arrival. 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.

```
-=Initialising the Container=-
-=Container (ID=STN:CONTAINER,FACETS=Discovery,VER=1)
  initialised=-
-=Creating Self=-
-=Registering Self (ID=STN:SHELL)=-
-=This is SATIN version 0.8=-
-=Running on Linux 2.6.5-1.358 / i386=-
-=Hostname: hamsalad.cs.ucl.ac.uk=-
-=Java 1.4.2_04 / Sun Microsystems Inc.=-
-=A reference to the container will be made available via the
  object reference container=-
-=Starting the beanshell...=-
BeanShell 2.0b1.1 - by Pat Niemeyer (pat@pat.net)
bsh % Component c=container.getComponent("STN:SHELL");
```

Fig. 5. The SATIN Shell.

The SATIN **Scripting Framework:** We have adapted BeanShell [26], an open source Java source interpreter and scripting mechanism as a SATIN component. This allows SATIN components to use scripts and to be scripted. Using this, we have created a "shell" for SATIN, which allows developers to manipulate the container and its contents by typing Java statements at runtime. The Scripting Framework component and the shell component occupy 280.6 kilobytes as a compressed jar file. Figure 7 shows sample output from the shell. The last line in particular, shows how to get a reference of a component from the container.

The SATIN shell demonstrates how a library is added into the system, promoting reusability between components. Moreover, the scripting framework can be expressed as a component dependency, for components (such as the shell) that require it and can be registered dynamically, when needed.

We believe that the implementation of the SATIN middleware system and the applications confirm that SATIN is reasonably lightweight, despite the offered features and added flexibility. The examples demonstrate applications that can monitor their context and adapt to changes to it. The next section presents a critical summary of related work.

5 Related Work and Discussion

Despite the evident suitability of logical mobility to the dynamicity of a mobile computing environment, its use to support self organisation has been very limited. Most approaches employ logical mobility to provide specific functionality to applications. Lime [24] is a mobile computing middleware system that allows mobile agents to roam to various hosts sharing tuple spaces. PeerWare [8] allows mobile hosts to share data, using logical mobility to ship computations to the remote sites that host the data. Jini [2, 35] is a distributed networking system, which allows devices to enter a federation and offer services to other devices, or use code on demand to download code allowing them to utilise services that are already being offered. The Software Dock [16] is an agent-based software deployment network that allows negotiation between software producers and consumers. one.world [14] is a system for pervasive applications, that allows for dynamic service composition, migration of applications and discovery of context, using remote evaluation and code on demand. FarGo-DA [42] is an extension of FarGo [17], providing a mobile framework for resource-constrained devices that uses remote procedure calls and code on demand to offer disconnected operations. The limitation of these approaches is in the fact 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 pre-define advertising and discovery services, making interoperability with different middleware systems and networks particularly difficult.

Other approaches focus on building reconfigurable middleware systems, using logical mobility primitives. ReMMoC [30] 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. UIC [37] a generic request broker, defines a skeleton of abstract components which have to be specialised 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 reconfigurational abilities 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.

There has also been some work on component based reconfiguration systems. Beanome [6] is a component model for the OSGi Framework [1], allowing Beanome applications to retrieve new components at runtime. Gravity [7] allows for reconfiguration of user-oriented applications. These approaches are limited: Beanome makes a clear distinction between hosts that can send components and between hosts that can receive them. SATIN on the other hand allows mobile devices to form large peer to peer networks of offered components. Gravity does not allow for system reconfiguration.

P2PComp [10] and PCOM [3] are mobile distributed component models. These approaches suffer from the problems examined in Section 2.2. Networking references in the absence of connectivity are invalidated, making autonomous operation of the system problematic.

Compared to related work, our approach does not limit how applications use logical mobility techniques; As SATIN takes a finer grained approach to logical mobility, allowing components to send and receive individual objects and classes as well as complete components, it can be used to implement the solutions of previous approaches, but its use and applicability is much more general. For example, existing middleware systems such as Lime can be implemented on top of SATIN, giving SATIN applications interoperability with hosts running Lime. Moreover, the way in which ReMMoc tackles heterogeneity through discovery and adaptation to different services can also be emulated with SATIN. The general adaptability and flexibility through logical mobility allows SATIN-based applications to heal and mutate according to context, which they can monitor, making them suitable for mobile computing. Moreover, the complete componentisation of all system aspects, including advertising and discovery, makes SATIN demonstrably suitable for roaming. The collocation of SATIN components allows a system to be autonomous; As SATIN focuses on the reconfiguration of local components, it allows for applications to function in the event of disconnection from remote hosts - This is particularly important, given the dynamicity of the network connectivity of mobile devices. Moreover, SATIN allows for devices to both send and receive LMUs; By not making any distinction between server and client, SATIN allows for the potential creation of a large peer to peer network of offered functionality.

SATIN is not the first project in which we use logical mobility techniques in a mobile environment. An earlier approach was used in XMIDDLE [19], a mobile computing middleware system, which allows for the reconciliation of changes to shared data. In the development of XMIDDLE, we realised that it would be advantageous to be able to chose at runtime which protocol to use to perform the reconciliation of changes and designed an architecture which allowed for deciding upon, retrieving and using a reconciliation protocol at runtime. In previous work [44], we also identified a number of examples showing that logical mobility can bring tangible benefits to mobile applications. Our efforts in designing SATIN are based on that experience. Initial work on SATIN [45, 43] focused on providing a middleware system that allowed for the flexible use of logical mobility primitives. We realised, however, that our approach can be formalised and offered in terms of a component model, with all the advantages for mobility that this entails, as shown in Section 2.2. This paper reflects the evolution of this work in this way.

It is worth considering that SATIN can easily be turned into a distributed component model, by allowing two types of containers in one instance of the system: One that would host the local components and another that would host the remote ones. Moreover, the completely decoupled nature of SATIN allows for strong customisation. For example, a registrar implementation could offer notification services when a component fails to register because of an unmet dependency. Similarly, a Deployer implementation could offer notification of failed or malicious LMU transfers. We believe that these examples show the flexibility of our architecture.

6 Conclusions

In this paper we have argued that mobile systems suffer from a number of problems related to their nature, heterogeneity and dynamicity in particular. We argue that these are problems that cannot be tackled by static applications and that mobile systems can benefit from the use of self organising primitives. We proposed the use of SATIN, a component model offering distribution services.

We have designed and implemented SATIN as a generic platform that offers self organisation through logical mobility and componentisation, with reasonable performance trade-offs as Section 4 shows. Unlike other approaches, SATIN allows applications to use any logical mobility paradigm and can be used to tackle the dynamicity inherent in this setting. The applications built demonstrate this functionality.

We intend to investigate a number of ideas in the future. An issue of great importance in a mobile system that can receive code from other parties is 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. In future implementations, we plan to investigate the use of Proof Carrying Code [25] techniques, which may alleviate this need. We are considering implementing CARISMA [4], a platform that uses reflection to allow applications to reason about the local execution context, as a collection of SATIN components. We have already identified the advantages of combining logical mobility and reflection primitives in previous work [5]. In essence, this will allow applications to use the SATIN adaptive mechanism to adapt to changes (through code download and upload), while using CARISMA to monitor and reason about the local execution context, such as battery power levels. Finally, we intend to continue testing with other networking interfaces, such as Bluetooth, to examine the behaviour and performance of SATIN on those interfaces. We are also considering implementing routing and communication components, that could be dynamically loaded when needed. This would allow us to route packets between networks, and attach metadata (that could contain security information, for instance) identifiers to the routing layer, thus introducing more flexibility over traditional routing techniques in terms of self organisation in the dynamic environment imposed by mobile computing.

Acknowledgements
We would like to thank Licia Capra for her suggestions and help in producing this paper and Martin Ijaha for his work in implementing a Deployer for SATIN. This work is kindly sponsored by EPSRC grant GR/R70460.

References

1. The OSGi Alliance. The OSGi framework. http://www.osgi.org.
2. K. Arnold, B. O'Sullivan, R. W. Scheifler, J. Waldo, and A. Wollrath. *The Jini[tm] Specification*. Addison-Wesley, 1999.
3. C. Becker, M. Handte, G. Schiele, and K. Rothermel. Pcom - a component system for pervasive computing. In *Proceedings of the 2nd International Conference on Pervasive Computing and Communications*, Orlando, Florida, March 2004.
4. L. Capra, W. Emmerich, and C. Mascolo. CARISMA: Context-Aware Reflective mIddleware System for Mobile Applications. *IEEE Transactions on Software Engineering*, 2003.
5. L. Capra, C. Mascolo, S. Zachariadis, and W. Emmerich. Towards a Mobile Computing Middleware: a Synergy of Reflection and Mobile Code Techniques. In *In Proc. of the 8th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS'2001)*, pages 148–154, Bologna, Italy, October 2001.
6. H. Cervantes and R. Hall. Beanome: A component model for the OSGi framework. In *Software Infrastructures for Component-Based Applications on Consumer Devices*, Lausanne, September 2002.
7. H. Cervantes and R. Hall. Autonomous adaptation to dynamic availability using a service-oriented component model. In *Proceedings of the 26th International Conference of Software Engineering (ICSE 2004)*, pages 614–623, Edinburgh, Scotland, May 2004. ACM Press.
8. G. Cugola and G. Picco. Peer-to-peer for collaborative applications. In *Proceedings of the IEEE International Workshop on Mobile Teamwork Support, Collocated with ICDCS'02*, pages 359–364, July 2002.
9. W. Emmerich. *Engineering Distributed Objects*. John Wiley & Sons, April 2000.

10. A. Ferscha, M. Hechinger, R. Mayrhofer, and R.Oberhauser. A light-weight component model for peer-to-peer applications. In *2nd International Workshop on Mobile Distributed Computing*. IEEE Computer Society Press, March 2004.
11. The Xiph.org Foundation. The OGG vorbis project. http://xiph.org/ogg/vorbis/.
12. A. Fuggetta, G. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, May 1998.
13. Kurt Granroth. Using KDE components (KParts), 2000. Unpublished invited talk at Annual Linux Showcase 2000.
14. Robert Grimm, Tom Anderson, Brian Bershad, and David Wetherall. A system architecture for pervasive computing. In *Proceedings of the 9th workshop on ACM SIGOPS European workshop*, pages 177–182. ACM Press, 2000.
15. Object Management Group. Meta Object Facility (MOF) specification. Technical report, Object Management Group, March 2000.
16. R. S. Hall, D. Heimbigner, and A. L. Wolf. A cooperative approach to support software deployment using the Software Dock. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 174–183. IEEE Computer Society Press / ACM Press, 1999.
17. O. Holder, I. Ben-Shaul, and H. Gazit. Dynamic layout of distributed applications in FarGo. In *Proceedings of International Conference on Software Engineering*, pages 163–173, May 1999.
18. JCraft. Jorbis – pure java ogg vorbis decoder. http://www.jcraft.com/jorbis/.
19. C. Mascolo, L. Capra, S. Zachariadis, and W. Emmerich. XMIDDLE: A Data-Sharing Middleware for Mobile Computing. *Int. Journal on Personal and Wireless Communications*, 21(1), April 2002.
20. P. Mathieu, J. C. Routier, and Y. Secq. Dynamic organization of multi-agent systems. In Maria Gini, Toru Ishida, Cristiano Castelfranchi, and W. Lewis Johnson, editors, *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'02)*, pages 451–452. ACM Press, July 2002.
21. Sam Michiels, Lieven Desmet, Nico Janssens, Tom Mahieu, and Pierre Verbaeten DistriNet. Self-adapting concurrency: the dmona architecture. In *Proceedings of the first workshop on Self-healing systems*, pages 43–48. ACM Press, 2002.
22. Richard Monson-Haefel. *Enterprise Javabeans*. O'Reilly & Associates, March 2000.
23. Ian Murdock. Overview of the Debian GNU/Linux system. *Linux Journal*, 6, October 1994.
24. A. L. Murphy, G.P. Picco, and G.-C. Roman. LIME: A Middleware for Physical and Logical Mobility. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS-21)*, May 2001.
25. G. C. Necula. Proof-carrying code. In *The 24TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119. ACM SIGACT and SIGPLAN, ACM Press, January 1997.
26. P. Niemeyer. BeanShell - Lightweight Scripting for Java.
27. Object Management Group. *Unified Modeling Language*, March 2003. statut : Version 1.5 , http://www.omg.org/docs/formal/03-03-01.pdf.
28. OMG. *The Common Object Request Broker: Architecture and Specification Revision 2.0*. 492 Old Connecticut Path, Framingham, MA 01701, USA, July 1995.
29. OMG. CORBA Component Model. http://www.omg.org/cgi-bin/doc?orbos/97-06-12, 1997.
30. P. Grace and G. S. Blair and Sam Samuel. Middleware Awareness in Mobile Computing. In *Proceedings of First IEEE International Workshop on Mobile Computing Middleware (MCM03) (co-located with ICDCS03)*, pages 382–387, May 2003.
31. PalmSource. Palmsource developers program. http://www.palmsource.com/developers/.
32. H. Van Dyke Parunak and Sven Brueckner. Entropy and self-organization in multi-agent systems. In *Proceedings of the fifth international conference on Autonomous agents*, pages 124–130. ACM Press, May 2001.
33. J. Power. Distributed systems and self-organization. In *Proceedings of the 1990 ACM annual conference on Cooperation*, pages 379–384. ACM Press, 1990.
34. The Distributed.net Project. Distributed.NET. http://www.distributed.net.
35. Psinaptic. JMatos. http://www.psinaptic.com/, 2001.
36. D. Rogerson. *Inside COM*. Microsoft Press, 1997.
37. M. Roman, F. Kon, and R. H. Campbell. Reflective middleware: From your desk to your hand. *IEEE Distributed Systems Online Journal, Special Issue on Reflective Middleware*, July 2001.
38. Sun Microsystems, Inc. JavaBeans. http://java.sun.com/products/javabeans/.
39. Sun Microsystems, Inc. Mobile Media API.
40. Sun Microsystems, Inc. *Java Remote Method Invocation Specification*, Revision 1.50, JDK 1.2 edition, October 1998.
41. Sun Microsystems, Inc. Java Micro Edition. http://java.sun.com/products/j2me/, 2001.
42. Y. Weinsberg and I. Ben-Shaul. A programming model and system support for disconnected-aware applications on resource-constrained devices. In *Proceedings of the 24th International Conference on Software Engineering*, pages 374–384, May 2002.
43. S. Zachariadis and C. Mascolo. Adaptable mobile applications through satin: Exploiting logical mobility in mobile computing middleware. In *1st UK-UbiNet Workshop*, September 2003.
44. S. Zachariadis, C. Mascolo, and W. Emmerich. Exploiting logical mobility in mobile computing middleware. In *Proceedings of the IEEE International Workshop on Mobile Teamwork Support, Collocated with ICDCS'02*, pages 385–386, July 2002.
45. S. Zachariadis, C. Mascolo, and W. Emmerich. Adaptable mobile applications: Exploiting logical mobility in mobile computing. In *5th Int. Workshop on Mobile Agents for Telecommunication Applications (MATA03)*, pages 170–179. LNCS, Springer, October 2003.