Exploiting Logical Mobility Techniques in Physically Mobile Environments

Stefanos – Zacharias Zachariadis

First Year PhD Viva

PhD start date: September 2001

Ext: 37190
s.zachariadis@cs.ucl.ac.uk

Executive Summary

With the recent developments in wireless networks (802.11, Bluetooth) and the sales of mobile computers of any kind (such as laptop computers, Personal Digital Assistants (PDAs), mobile phones etc.) soaring, we are experiencing the availability of increasingly powerful and mobile computing environments, roaming between different types of network connectivity. We have also recently witnessed the acceptance of logical mobility techniques, or the ability to ship part of an application or even a complete process from one host to another. The increasing popularity of the Java programming language and environment has largely been correlated with the acceptance of logical mobility techniques, due to the inherent code mobility infrastructure that Java provides.

To facilitate application developers programme for mobile devices, mobile middleware systems have been developed, which rise the programmer abstraction beyond the network os layer, and tackle issues of heterogeneity, communication primitives, concurrency, security etc. Whereas various mobile middleware systems have been developed the use of logical mobility has been very limited.

The purpose of this work is to investigate the use of logical mobility in mobile computing environments, our hypothesis being that it can bring tangible benefits to both developers and users which cannot be supported by the current state of the art. While investigating why the use of logical mobility has not been widely adopted by the mobile industry, we have found the following problems: There is currently no design stage methodology which allows application developers to model the effects of logical mobility in their applications at the design stage. This limits the development of mobile applications powered by logical mobility to simply trial-and-error scenarios, inhibiting their popularity. Moreover, we have found that there exists no mobile middleware that offers flexible use of logical mobility primitives to application developers, whilst allowing devices to traverse dynamically different types of networks, handling heterogeneity, service advertisement and discovery, communication etc. As such, applications using logical mobility primitives have to constantly reinvent the wheel in order to implement the solutions. We believe that these deficiencies in this area of research are closely related, being aspects of the same issue, one affecting the design stage, and the other the implementation stage.

This work intends to address these deficiencies, by researching a design-stage methodology, which allows developers to evaluate the use of logical mobility paradigms in their application, and developing a mobile middleware, which will be targeted by the aforementioned methodology and will offer the use of logical mobility primitives to applications. We finally intend to build a number of applications using our middleware and methodology, to evaluate our hypothesis.

Background and Motivation

We are rapidly approaching the era of ubiquitous computing; People are starting to interact daily with multiple computing environments. From the proliferation of mobile telephones, to the widespread acceptance of personal digital assistants and the tendency to replace desktop machines with portable laptop computers, users are increasingly starting to own and/or interact with a number of computing devices, a primary characteristic of which being that they are mobile. As these devices become increasingly more powerful, users are starting to carry with them mobile processing environments of respectful computing ability. Mobile devices are being equipped with various networking interfaces, such as conventional modems, IrDA adapters, 802.11b compliant cards, GSM adapters, Bluetooth chips and the like; As such, we expect these devices to dynamically connect to different types of networks. At some point, the device might have a slow GSM connection to the Internet; later in time, the same device may only be part of a Bluetooth piconet, being able to communicate with devices that are within a short distance, without fixed infrastructure; while later on, it might connect to a corporate network via a high speed Ethernet adapter or be devoid of any network access.

In recent years, we have also witnessed the growth of logical mobility techniques. We define logical mobility, as the process of moving parts of an application, either a code fragment (be it binary object code of a compiled language, interpreted fragments of a scripting language or compiled code targeting a virtual machine), or application data (or even migrating a whole process) from one processing environment, or a device, to another. Consider Java Applets for example, which operate as follows: The client machine (a host), requests via a web browser (the user interface) a code fragment (the applet), which the server (another host) sends back. The web browser executes and displays the applet transparently and without user intervention in an instance of the JVM (the processing environment). The mechanism effectively allows for dynamically changing the interaction interface on the user, on a per website (or, alternatively, per location) basis. We consider the following forms of logical mobility based interactions[8]: Client/Server interactions (CS), whereby the request of a client triggers the execution of a unit of code in a server returning the results, Remote Evaluation (REV), where a device sends a code unit to another host, has it executed and retrieves the result, Code On Demand (COD), where a host requests a unit of code from another device to be retrieved and executed, and Mobile Agents (MA), where an agent is an autonomous unit of code that decides when and where to migrate. Moreover, we consider devices that can be nomadically connected to a fixed network (e.g. a laptop dialling up to an ISP), devices that are constantly connected to a fixed network over a wireless connection (e.g. a GPRS-enabled mobile phone), devices that are connected to ad-hoc networks (e.g. Bluetooth) and any combinations of the above.

A primary demand of users, as witnessed by the flourishing of mobile networking hardware, mobile networks and the associated industries, is to have access to networked information and services using their mobile devices wherever they might be located. Application developers must cope with the variability of networking techniques and types of mobility, as well as eventualities such as no connection or frequent disconnection, lack of a fixed infrastructure, expensive network links, limited bandwidth etc. The issues arriving from the heterogeneity of such distributed systems have been traditionally resolved by the use of middleware systems. Middleware systems are situated between the network operating system (which may vary amongst devices) and the applications and facilitate the development of the latter by providing higher level abstraction primitives than the network operating system. While many middleware systems have been developed and are used in the industry for traditional distributed systems (e.g. CORBA, Java/RMI etc), mobile computing middleware is slowly and only recently being developed, with ad-hoc solutions being the norm. Mobile Computing Middleware allows for the systematic development of mobile applications, allowing developers to handle the heterogeneity of devices and environments, communication primitives, data sharing, location services etc. There are, however, no middleware systems developed which give developers the flexibility to utilise any logical mobility paradigm in their application, while allowing the device to traverse through heterogeneous networks.

We wish to prove that logical mobility techniques can be applied to mobile computing scenarios to bring solutions to problems that the current state of the art cannot support or cannot do so sufficiently. We believe that logical mobility offers the possibility to create a new class of flexible mobile applications, delivering innovative experiences to end-users and lead us to the era of ubiquitous computing. We believe that the reasons for which logical mobility has not been sufficiently exploited in physically mobile scenarios are twofold: a) There is currently no middleware that can offer to application developers the ability to choose between mobile code paradigms and the ability to seamlessly move between different types of IP-based networks, as described above. This limits the benefit of using mobile code techniques, as developers need to constantly re-invent the wheel in order to employ them in their application, inhibiting application interoperability as well. b) There is currently no methodology for evaluating which paradigm (or even local execution) would function best for a particular computing task. This limits the development and testing of application to trial and error scenarios, where an application may be built employing a particular methodology, only to find through testing that it does not perform as expected.

There are several factors that need to be taken into consideration by application developers when applying logical mobility in physically mobile scenarios. These considerations must take place at the application design stage, structuring the application accordingly to take advantage of the appropriate features and paradigms that the underlying middleware provides in each particular context, making the application function best in each scenario. However, in our target environment, there can be various definitions of how an application functions best: this may be when it uses underlying technologies to perform the given task rapidly or when it minimizes the network interactions for a specific task. This can be important to the end user, considering that some types of wireless connection are expensive. An application can function best when it minimises user-interaction, thus saving the user time. An application can also function best when, in performing a function potentially involving code mobility, it minimises the information needed to be sent over the network thus decreasing the total cost of the operation for the user. We believe that these are issues, which need to be thought of at the application design stage.

We plan to investigate this area of research, specifically by a) Providing a flexible mobile computing middleware offering logical mobility techniques to application developers and b) Deriving a design methodology, which, based on developer data, will evaluate the different paradigms at the design stage for particular applications. Thus we wish to prove that logical mobility can bring tangible benefits to mobile applications over traditional approaches.

.

Deficiencies in existing work

Having reviewed the literature in this area of research, we found that the current state of the art offers a rather limited solution to the exposed problem. Current approaches aim at use particular paradigms of logical mobility to (often transparently) offer specific solutions in a mobile environment. There is currently no mobile computing middleware providing the flexible use of logical mobility to applications and application developers and no formal support that helps use application design and performance evaluation targeting such a middleware. What follows is a critical review of some approaches of existing work. **Sun Microsystems Jini**[3] is a distributed networking system, which allows devices to enter a federation and offer services to other devices or utilise services already offered. Jini, based on the Java programming language, exploits the inherent code mobility capabilities of that language in allowing devices to transparently locate and operate services offered. A service in a Jini system is an object or a collection of objects provided by a computer, which can be utilised by other devices in the Jini federation. Examples include device drivers (such as printer drivers), whereby objects controlling the device can be operated remotely, time services, computational facilities (such as language parsers for example) and more. Jini relies on the existence of at least one lookup service, with which other services register, and which can be used by devices to locate services offered. Upon locating a service, a proxy object can be transferred using Code on Demand (COD) techniques to the requesting device, using it to utilise the service. Jini can be used to deliver distributed services in high-speed and long lived (preferably wired) networks. However, Jini is very much centralised, needing lookup services to operate. Jini does not really scale well on low bandwidth highly dynamic ad-hoc networks. The reference implementation relies on an http server to be used by the lookup service and is based on Java/RMI, making it unsuitable for deployment over resource constraint devices. There is, however, a lightweight implementation, JMatos[10], which does not rely on Java/RMI. This has been specifically developed to be used over devices with minimal resources, whilst being compatible with the reference implementation. Jini can be used to deliver context aware services to mobile devices, connected to a fixed network nomadically. It is not, on the other hand, particularly suitable for allowing mobile devices to offer services themselves, particularly in ad-hoc environments, which lack a centralised lookup service. **Lime & µCode**: Linda in a Mobile Environment [14] (Lime) is a middleware implemented in Java, which allows the development of applications which exhibit logical and/or physical mobility characteristics, by providing a coordination model based on Linda tuple spaces. Lime is primarily geared for ad-hoc networks, although it is not limited to such configurations only. Lime exploits the decoupled nature of tuple spaces to provide coordination primitives and information sharing for mobile components. The unit of logical mobility considered in Lime is a mobile agent, with mobile hosts acting as simple containers for the agents. An agent is in reach with other agents if it resides on the same host as they, or if the hosts of the agents are in reach. Each agent can have a set of tuple spaces, which are identified by their name, a String. The tuple spaces are bound to the agents and as the agent migrates, the tuple spaces migrate as well. The agent can choose whether to share a tuple space it owns. Lime makes all tuple spaces with the same name, marked as shared by agents in reach, transparently appear as a single tuple space. Lime also extends the standard Linda operations to support location based computing and allows agents to react to changes in context, by defining reaction primitives. The current implementation of Lime uses the µCode[15] toolkit, which is a lightweight mobile code toolkit to implement the Mobile Agent abstraction. Lime is related to our work, as it is a data-sharing and coordination middleware based on mobile agents for mobile computing. It does not, however provide the generic cross-paradigm logical mobility middleware that we wish to develop. Moreover, it uses an unstructured data unit, the tuple space and has no notion of security. There also appear to be some scalability issues with misplaced tuples. **PeerWare**[6] is a mobile computing middleware offering peer-to-peer communication, event subscription and a shared data space. PeerWare binds the models of event-based communication and data sharing into a single middleware, using REV, to distribute operations on remote data. The PeerWare system is based around the concept of a Global Virtual Data Structure (GVDS), a communication and coordination meta-model for mobile environments. A GVDS provides a global data space that is made dynamically by the local data spaces of each peer in range. It is virtual, as it does not exist on any host as a single entity. The GVDS meta-model does not specify how the GVDS is structured, leaving it to the implementer. The PeerWare data structure is a graph of nodes and documents, which are collectively referred to as items. Nodes are simple containers of items, and are structured as a forest of trees, with a distinct root. Nodes have a label, which cannot be shared with another node if both are roots or contained directly into the same node. This classification and organisation of nodes allows for expressing complex document organisation schemes, the resulting model resembling a standard file system. Each PeerWare-enabled host has a data structure stored locally. PeerWare dynamically constructs a GVDS by superimposing all the nodes of the local data structures of all peers in range. This function is completely hidden to applications that must only be aware that the contents of the data structured can eventually change. PeerWare makes a sharp distinction between operations that can be performed on the local data structure and those that can be performed on the GVDS. This distinction, even if it lacks transparency, gives the application programmer explicit knowledge of whether he/she is operating locally or globally. PeerWare exploits logical mobility, by considering the execution of an action on the GVDS, as a distributed execution of the action on the local data structures of the connected peers. It is designed specifically to provide a minimal set of operations, with application-specific primitives and abstractions left to the developer. PeerWare exploits logical mobility (REV in particular) effectively, to move computations to the data sources in a peer-to-peer environment. It is, in a sense, the next generation of the Lime system, providing a hierarchical data structure instead of the Lime's flat tuple space as a GVDS. However, the PeerWare model does not prescribe anything about routing and networking infrastructures and as such multiple different implementations must and are being provided to work in different settings, such as ad-hoc, nomadic, etc. It does not provide a single middleware, which can seamlessly operate in various different networks. Moreover, it can been argued that moving the code to the data is not always better than the traditional communication model of moving the data to code. Current implementations utilise μCode as a logical mobility layer to move code to the peers. **PRIMAmob-UML**[25] is a UML-based methodology for performance analysis of logically mobile software architectures. UML sequence and collaboration diagrams, which are usually available in the early stages of the software's lifecycle are annotated with mobility-related stereotypes, allowing the developer to model code migration. The diagrams are then annotated with probabilities and communication & computation cost information, and a performance model of the application is obtained, allowing the designer to evaluate the logical mobility choices made. This methodology, allows application developers to evaluate the various logical mobility paradigms when used on traditional distributed systems over fixed networks, using a modelling language, which is widely accepted in the industry and academia. Our work will differentiate from this, in that it will cater for physically mobile software architectures, taking into account the resource constraint nature of some the devices we are targeting, the different network topologies and architectures that we expect devices to use etc.

Hypothesis statement

We believe that the use of logical mobility in mobile computing middleware can deliver significant advantages to mobile applications, which cannot be delivered by other approaches. Advantages include having the application use underlying middleware technologies to perform a given task rapidly, minimising the network interactions for a specific task (as wireless network connectivity can be expensive), minimising user-interaction, thus saving the user time and more. Clearly, these are different goals, which need to be evaluated by application developers at the design stage. We support our hypothesis with a number of case studies, one of which is given below, in which logical mobility can yield significant advantages over legacy approaches.
Devices with Limited Resources and Dynamically Updating the System: Code On Demand offers a way to dynamically and (depending on connectivity) securely update the capabilities of a device. It also allows the manufacturer or developer not to cater for all the possible scenarios that the device might encounter, as behaviours needed for different scenarios can be learnt at runtime. Consider, for example, user A, who has an MP3 (a digital music format) player, the firmware of which can be updated. Moreover, assume that the mp3 player is equipped with a Bluetooth or infrared adaptor. Along comes B, who has an Ogg Vorbis (another digital music format) player, which features the same networking hardware as A. Now, even if both players are in reach and they can transfer files to/from each other, they cannot actually play the transferred music, as they don't have the appropriate codec to decode the files. However Code On Demand allows A's player to transparently request the Ogg codec from B, initialise it and play the music, the whole operation being hidden from the user. If memory becomes full, the player's middleware can decide which codec to drop (based on frequency of use); after all, it can always retrieve it when needed from a peer. Logical mobility can offer significant benefits to other scenarios as well: It can be used to offer location based reconfigurability and services, communication in disaster scenarios, faster execution by distributing computations and minimize user interaction.

We believe that the reason why logical mobility has not been widely adopted in the mobile industry is twofold: To begin with, there is no mobile computing middleware developed offering the flexible use of logical mobility paradigms to application developers, targeting resource constraint devices and being able to traverse different IP-based networks. Moreover, there is no design-stage methodology that allows application developers to evaluate the performance and functionality of their application when employing logical mobility techniques.

We will try to address these deficiencies in this area of research, by designing a mobile computing middleware that offers this flexibility to application developers, a design stage methodology to design applications targeting this middleware and sample applications to evaluate our hypothesis.

Testing the hypothesis 1:

What is proposed

We plan to design and implement a modular and flexible mobile computing middleware with the characteristics mentioned above. The middleware will allow devices to traverse between different types of networks, such as ad-hoc and fixed infrastructure networks and will allow applications to use any mobile code paradigm. The middleware will also allow users to store information such as the maximum connectivity cost the user is prepared to pay on profiles, which will be exposed to applications at runtime, among with other context information, such as the hosts that are currently in reach, the services available, the current networking structure etc. The middleware will be targeted towards mobile and resource constraint devices, although the protocols will be able to interoperate with other versions of the middleware, targeted and running on more powerful machines.

Why it is necessary to do this?

The current state of the art does not offer an adaptable middleware, which exposes logical mobility to applications. This is probably one of the reasons that the use of code mobility has not taken off in mobile environments. These are environments in which devices encounter various different networking configurations. We have identified two types of metrics that need to be quantified and considered with regards to application performance using logical mobility: Parameters regarding physical mobility and the underlying network (speed and congestion, cost of access, network longevity, routing and network configuration, reliability) and parameters regarding code mobility (size of mobile code unit, how to contact the user with a result, origin or destination of the code, available resources). We believe that these parameters can be re-organised further to mobility and user related parameters. Mobility parameters can be modelled and evaluated at the application design stage. User related parameters will be stored in the user profiles mentioned above and will be made accessible to the application. It follows that the middleware must give the application information on the current context.

What assumptions will you make?

The first assumption that will be made is that we can get the context information from a mobile device[4]. This is a different field of research and such issues will not be addressed in this work. Moreover, conflict resolution in user-profiles either within the same device or amongst different devices involved in a transaction will not be addressed, for the reasons given above. The implementation of the middleware will be Java-based, initially targeting J2SE with the possibility of scaling down to Personal Java and J2ME. The devices we will be initially targeting are HP Compaq Ipaq PDAs.

The mobile setting we are targeting will be both ad-hoc networks, based on 802.11b hardware and potentially Bluetooth, as well as nomadic networks, based on Ethernet and managed-mode 802.11b hardware. On the nomadic network setting, we will assume that there exist some centralised (possibly ISP-provided) servers which offer hosting facilities for REV requests and mobile agents.

How do you propose to do the testing?

Initially, the host and service discovery and advertising mechanisms will be tested using a number of machines from desktop computers to laptops to the PDAs. This will be tested using 802.11b compliant networking cards, Ethernet cards, possibly GSM/GPRS wireless connectivity and combinations of the above.

We will then investigate how to send and receive code over the network efficiently. This will include testing the sending of individual classes and objects, scriptable text (e.g. Python through JPython or even Java code through BeanShell) or groups of classes and objects. We will also investigate sending the code compressed or uncompressed, using various algorithms optimised for speed or size of the compressed code. Once the code mobility aspects are implemented, test/dummy applications will be developed to test them over the hardware mentioned above. Scalability issues when the numerous machines are involved will also be investigated. The sandbox serving mobile agents and remote evaluation requests will be tested last.

When this is implemented and tested, the ability to add user profiles will be designed and added. We will consider either adding a single per-device profile, or a profile per application and design the semantics accordingly.

Major testing will take part when the real applications, as mentioned in part 3, are implemented.

What results do you expect?

We expect that this middleware will be able to work well on resource constraint devices, given our experience in implementing XMIDDLE[12], a java-based mobile computing middleware (XMIDDLE currently weighs at around 125KB[2]). We believe that there will be some scalability issues with multicast and broadcast advertising, which we plan to investigate and we also plan to investigate the feasibility of the hosting sandbox using the java security framework. We believe that the most efficient mechanism to transfer code in general, will be groups of classes and objects. There will probably be cases however, where sending text-based scripts will be more efficient, as changes between two versions of the same script can be calculated and sent over the network as opposed to sending the full script or class and object representing it.

Timescale

6 months

Testing the hypothesis 2:
What is proposed

We will model the development of mobile applications using logical mobility techniques in order to construct a design-stage methodology based on UML and using its extension mechanisms. This will allow application developers to develop mobile applications evaluating the use of the various code mobility paradigms in a mobile environment early in the design stage. The diagrams will be annotated with data given by the developers, such as expected user location, estimated size of mobile code unit, interactions of the unit with other hosts, etc. Processing the diagrams will produce estimates, which the developers can use to evaluate how the application will perform when employing a particular paradigm. This is related to our original hypothesis, as it will allow us to model the benefits of logical mobility to a mobile application.

Why it is necessary to do this?

The current state of the art does not allow application developers to evaluate the use of the different code mobility paradigms in mobile environments at the design stage. As such, mobile applications employing logical mobility techniques have to be developed on a trial-and-error basis. We believe that this is one of the reasons for which code mobility is not widely adopted in the mobile computing industry.

A design-stage methodology such as the one we are proposing, based on an industry standard (UML) will allow application developers to model the effects of logical mobility in mobile applications, evaluate its benefits and potentially lead to an increase in its use in mobile scenarios. It would also allow us to evaluate the performance of our middleware, when developing mobile applications.

What assumptions will you make?

We believe that it is reasonable to expect the developer to supply some estimated data regarding the application in development. This includes estimates on the size of the unit of code that might be transferred to other hosts (which can be obtained by the number of classes that compose the unit and the complexity of those classes), estimates on the number of different hosts that the code will need to contact, estimates on the expected location of the user etc.

Further assumptions are still under investigation.

How do you propose to do the testing?

To test the validity of our approach, sample applications will be designed and implemented to run on our middleware. More concrete testing will be performed by designing and evaluating applications developed for the middleware, as described in part 3.

We expect to use a variety of tools to test our approach. We are considering developing a UML design tool, or possibly extending current open-source tools such as DIA or ArgoUML in order to support and test our approach. We are also proposing using process algebra and queuing networks to model process and network interactions.

What results do you expect?

We expect to be able to model application performance sufficiently, and be able to show to the developer which logical mobility paradigm (if any) will be better under a particular context. We believe that we will be able to offer an enhanced design tool for the analysis of logical mobility in mobile computing application scenarios. The tool will be based on performance evaluation techniques and will encode parameters important in deciding which paradigm to use in a particular context.

Timescale

7 months

Testing the hypothesis 3:

What is proposed

We are going to design and implement some applications using our design methodology that target our middleware. We currently plan to build an audio player which can dynamically request codecs to learn how to play new audio files, an e-shopping application using mobile agents and a mobile data querying application using remote evaluation and mobile agents, with more to follow. The applications will also be able to function without using any logical mobility paradigm. We will be able then to extract metrics concerning the application performance and evaluate them against those projected by our methodology as outlined before.

Why it is necessary to do this?

Building these applications will be essential to proving the validity of our main hypothesis. Evaluating their performance will prove whether logical mobility can give applications tangible advantages over legacy approaches. We will also be able to evaluate the performance and effectiveness of our middleware and design methodology and we will use these applications to further optimise their operation.

What assumptions will you make?

We will assume that, apart from the ones our middleware provides, there are centralised environments which offer hosting for mobile agents and remote evaluation requests. Moreover, we will assume that there are going to be some centralised code repositories, which can be used to download code from, or to check the hashes of code obtained from other hosts. We believe that this is a reasonable assumption and that once the validity of employing logical mobility in mobile applications is established, this could be a service provided by internet service providers, mobile device vendors, etc.

How do you propose to do the testing?

Similarly to the testing methodology above, we will run these applications on our middleware on a variety of machines, including PDAs, laptop computers and desktop machines. We will use variations of our middleware to create hosting environments for mobile agents and remote evaluation requests in desktop machines, to act as centralised servers offering hosting services to mobile devices. Having our devices equipped with Ethernet cards, Bluetooth adaptors and 802.11b compliant cards, we will be able to simulate the execution of our applications on a variety of distributed networked environments.

What results do you expect?

We expect that logical mobility will yield significant advantages in some situations. These advantages will include speeding up the application operation, decreasing the cost of access to the network, minimising user-interaction etc. We do expect, however, that traditional approaches such as local execution and client/server communication mechanisms will be better in some circumstances, which we believe we will be able to evaluate and predict using our design stage methodology.

Timescale

3 months

Why is the above set of work sufficient to test the hypothesis?

The above set of work is sufficient to test the hypothesis, as it manages to tackle the deficiencies of the research in Logical Mobility over Mobile Networks and builds upon the work to develop mobile applications employing logical mobility techniques and evaluate them. With this work, we believe we will be able to address the two main reasons for which logical mobility is not widely adopted in mobile computing.

Section 1 addresses the lack of a mobile computing middleware system, which gives application developers the flexibility of employing logical mobility in their application. Our middleware will allow the implementation of application which employ logical mobility techniques in their operation, on mobile environments, handling the issues of the heterogeneity of the devices targeted and of the networks encountered. Having completed the work of section 1, we will have proved that a mobile computing middleware that allows application developers to use logical mobility techniques is feasible.

Section 2 addresses the lack of a design-stage methodology which allows application developers to evaluate the operation of their applications when employing logical mobility paradigms, early in the design stage. As such, this methodology will allow for the design and development of applications targeted and implemented for the middleware developed in section 1.

Section 3 will allow as to engineer mobile applications using our design methodology and middleware, which will employ logical mobility techniques. Evaluating their performance and their effectiveness in various mobile scenarios, will allow as to verify the hypothesis, that logical mobility can bring tangible benefits to mobile applications that current state of the art approaches cannot.

Appendix A: Work to date

Work thus far has concentrated on background reading on approaches involving logical mobility over mobile environments. We have also come up with a number of different case studies whereby logical mobility would be able to give solutions to issues occurring in mobile environments that legacy approaches would not be able to:

Limited Resources and Dynamically Updating the System
To approach the stage of ubiquitous computing, mobile application developers and device manufacturers need to offer to consumers devices that can be used very easily, without having the user do any sort of software installation on the device to allow it to perform another function, thus giving consumers a device with the functionality and configurability of a computer, without forcing them to learn how to use one. These computers must be "invisible"; end users should not have to know and realise that they are using one. As such, it is very difficult for a middleware or even operating system developer to estimate in advance all the possible uses that the device might have; Moreover, resource (memory) restrictions on such appliances imposed by costs might not even allow the required code for all scenarios to be stored on the device. Here's how code mobility, and in particular Code On Demand can help in this case. Consider for example user A, who has an MP3 (a digital music format) player the firmware of which can be updated. Moreover, assume that the mp3 player is also equipped with a Bluetooth or infrared adaptor. Along comes B, who has an Ogg Vorbis (another digital music format) player, which features the same networking hardware as A. Now, even if both players are in reach and they can transfer files to/from each other, they cannot actually play the transferred music, as they don't have the appropriate codec to decode the files. However Code On Demand allows A's player to transparently request the Ogg codec from B, initialise it and play the music, the whole operation being hidden from the user. If memory becomes full, the player's middleware can decide (based on frequency of use) which codec to drop; after all, it can always retrieve it when needed from a peer.
Let's take the scenario a bit further. Suppose that user A has a Bluetooth adaptor and a GPRS modem equipped PDA. Along comes B, that offers a document, C, which is of interest to A. However A's device does not have the appropriate parser that is needed to display this file. Depending on the cost, the middleware has two choices: If connectivity to the GPRS network is cheap, it can connect to a centralised server, give it the file info, and request the appropriate parser. Alternatively, it can request the parser from B, generate a hash from it, and contact the centralised server to verify the hash (thus protecting the user from malicious code). In both cases, A will be able to see the document of interest, and all interactions will be transparent to the user. As it can be seen, Code On Demand offers a way to dynamically and (depending on connectivity) securely update the capabilities of a device. It also allows the manufacturer or developer not to cater for all the possible scenarios that the device might have, as behaviours needed for different scenarios can be learnt at runtime.
We have had some experience with this setting. In developing xmiddle[12], a data-sharing middleware for mobile computing, we designed and are implementing a mechanism using Java to allow a host to dynamically update the list of protocols that were available to the platform. A protocol is simply a networked interaction between two or more hosts and is defined by an abstract superclass which all protocols can inherit. This will allow xmiddle hosts to acquire new behaviours from peers as they come in reach, when needed.

Location-Based Reconfigurability and Services
Personal digital assistants or other similar portable computing environments, are meant to be mobile, unobtrusive and accompany the user everywhere. As such, these devices are exposed to a variety of environments and hardware resources which might exist in these environments and it is possible that the user would like to exploit some of these resources as transparently as possible. With the advent of Bluetooth technology, the likelihood of this scenario increases as Bluetooth allows devices to "pair", enabling one device to remotely operate the other. Code mobility can enhance the effectiveness of this scenario in the following ways:

Assume that a user, A, with a PDA and a rudimentary word processor, comes in the vicinity of a printer resource, say B. This printer could be a friend's printer, a pay-per-page printer at a university etc. In traditional systems, if A would like to print a document to B, he/she would have to undergo a relatively complex printer driver installation, which would involve locating the printer model and the appropriate driver for A's platform, a process which is not intuitive. Assume however, that both A's PDA and B are equipped with a networking technology, such as Bluetooth. Upon realising that the printer is in range, the middleware running on A's device automatically requests the code (a driver, complying to a generic interface) from B. After the transfer, the user would then be able to print the document without any intervention. Alternatively (for enhanced security, similarly to the scenario above) the code could be requested from a trusted centralised server using a resource signature (a model number for example) retrieved from B, assuming that A can connect to the server (using a GPRS modem for example). In both of these cases, the end result is that using Code On Demand and ad-hoc networking or base station mobility, the user is able to transparently operate a device that comes within his or her reach, without any intervention Alternatively, A can print the document on B using client/server interactions. The middleware on A's PDA can negotiate with the printer a common format that both devices understand (such as Postscript), translate the document into this format, send it to B which would initiate a printing service resulting in the document being printed. Again these interactions would be completely transparent to the user.

Messaging and Communication in Disaster Scenarios
The popularity of systems such as the Short Message Service (SMS) that GSM networks provide proves that consumers wish to leave asynchronous messages to people using their mobile device. Taking this one step further: Why limit the devices in only allowing them to leave a message to a particular person? An alternative would be leaving a message to a particular place for all passers by to see, like an electronic billboard. Drivers could leave, for example, warning messages at some particular spots where driving is particularly difficult, to be picked up by the computer of other drivers, warning them to be careful. Or diners could leave comments regarding a restaurant's quality to subsequent visitors. It has already been demonstrated that one can use Global Positioning System (GPS) technology to leave messages "in space" for passers by to pick up. We show an alternative way of doing this, using code mobility. Assuming the existence of a network of computers which offer these messaging services (similar to the SMS messaging servers), users of would be able to leave messages in the form of Mobile Agents to a messaging server provider, which would migrate on all devices that come in reach (and have registered as interested to this type of message) and display the encapsulated message. This is not limited to simple text messages, as the user could potentially encapsulate other information such as audio or images with a mobile agent. To take this scenario a bit further, let us consider ad-hoc networking and messaging. In this scenario there is no centralised computer network offering messaging services. This could be the case in a disaster or military scenario, which offers no fixed infrastructure. Messaging in this scenario could be pivotal; however locating the recipient is very difficult in an ad-hoc network. However, if the message were to be encapsulated in a mobile agent, the agent could roam the network, migrating or even cloning in other hosts that come in reach, until it reaches its intended destination (or its time to live expires). Some variation of ad-hoc networking messaging is already taking place in devices such as the Cybiko, which has already proved quite popular. These devices use RF networking to send messages to peers in reach. They can also do basic routing so that you can use an intermediate device as a router to send your message.

Electronic Shopping, Interacting on Behalf of the User and Limiting Connectivity Costs
The popularity of the Internet in recent years has led to the growth of electronic shopping: Large numbers of consumers now shop online, from electronic goods to daily groceries, and these numbers are bound to increase. On the other hand, electronic shopping using mobile devices such as PDAs has not been as popular as expected, relative to the proliferation of

these devices. The two primary factors that have been blamed for this are the following: To begin with, wireless network connectivity is prohibitively expensive. Moreover a great number of Internet sites are not optimised for portable devices (which are usually equipped with a very small screen and a sub-standard web browser). This forces users to stay for prolonged periods of time online, so that they can navigate the website, adding to the cost. We believe that code mobility in the form of Mobile Agents can resolve this situation. Assume a PDA with either a base station mobility or nomadic connectivity. This represents the majority of portable computing environments in production today and the numbers are bound to increase. The PDA could have a software application which would allow the user to input the product that he or she wishes to buy, the delivery address, and the payment information. Optionally, the user could input the highest price that he or she is willing to pay. The application would then encapsulate this information in a Mobile Agent, connect to the network (when connectivity is available) and inject the Agent into the network, which would transact with various online shops for the user, find the best price and order the item. The agent could migrate into the user's Internet Service Provider (ISP), or another independent third party, and conduct the required transactions from there, where connectivity would be much cheaper. This solution helps consumers save time and money, and could be extended to various other transactions and interactions. For example users could use a similar solution to this to perform a query for some information. The mobile agent would migrate to the users ISP, perform the search, and when the user reconnects to the network (using base station mobility or even nomadic computing) the agent would send the results back.

Distributing Computations and Exploiting Computational Resources
Mobile computing environments such as PDAs, mobile phones and the like, are underpowered, compared to typical desktop computers and workstations. Moreover they suffer from limited battery power which inhibits their ability to exploit their processing ability, however limited, for prolonged periods of time. Remote Evaluation (REV) can help in this scenario, over a variety of networking architectures. Consider a user, A, that wishes to perform a complex mathematical computation (estimating a complex integral for example) with his or her PDA and calculus software. If A chooses to perform the computation on the PDA, he or she will be forced to wait for a significant amount of time (CPUs such as the Strong ARM which are very popular on PDAs usually do not usually feature an Floating Point Unit), and possibly lose a good deal of the battery power. Assume however, that the user had nomadic or base station mobility connectivity to a larger network from the PDA. The application could package the computation to be done, ship it to a more powerful computer which would execute it and send the result back to the PDA, when connectivity is available. The computational environment could be provided by a third party, such as the user's ISP.
Another scenario could include ad-hoc networking, where powerful peers advertise processing cycles. Continuing the example above, A's calculus application could package and ship the computation to a peer B, which advertises enough free CPU cycles. The result of the computation would later be returned back to A's PDA. The problem with this scenario is that the user would need to stay in reach of B whilst the computation took place. The PDAs middleware could however do an estimation of how long it would take based on the information that B advertised, and inform the user accordingly.

Securing Communications over Potentially Hostile Networks
Sending and receiving sensitive data over any network can be dangerous. Third parties could read the traffic, forge replies, etc. This scenario becomes even more aggravated in wireless networks. Instead of sending the data over the network, we can inject the computation that needs to be performed over the sensitive data, to the network as a mobile agent. The agent could migrate to the site of the data and perform the computation there. This solution could be implemented over any networking architecture, although one with a fixed structure (such as the one offered by nomadic computing and base station mobility) would make implementation easier.

We have also researched into how to select a logical mobility paradigm to be used in a mobile application:

There are several factors that need to be taken into consideration by application developers when applying logical mobility in physical mobility scenarios. These considerations must take place at the application design stage, structuring the application accordingly to take advantage of the appropriate features and paradigms that the underlying middleware provides in each particular context, making the application function best in each scenario. However, in a our target environment, there can be various definitions of how an application functions best: An application can function best when it uses underlying technologies to perform the given task rapidly. An application can function best when it minimizes the network interactions for a specific task. This can be important to the end user, considering that some types of wireless connection are expensive. An application can function best when it minimises user-interaction, thus saving the user time. This is true for example, in the mobile agent shopping scenario given above. An application can also function best when in performing a function potentially involving code mobility, it protects the user's privacy by minimizing the information needed to be sent over the network. Clearly, these are very different goals that need to be identified by developers at design stage, realising which are being targeted. We identify the following metrics that need to be quantified and considered with regards to mobile software performance:
• Parameters regarding physical and network mobility
We believe that through reflective techniques[4], a mobile middleware should be able to provide the application with the following information:
– Speed and Congestion. We believe that the application developer should be able to retrieve some information on the current network throughput and speed.
– Network Longevity. It is also important that the application has some information on how permanent the current configuration is. Is it reasonable to expect that the device will stay connected in the current network for some time? This is information is essential if the application needs to be contacted as a result of a query or computation sent through the network.
– Cost of Access. The application should also be aware of how the user is being charged for the current connection.
– Network Configuration and Routing. It is important for the application to know to which type of network it is connected as this defines which hosts it can contact and how. Application behaviour on an ad-hoc network with limited routing support can be radically different than when connected to a centralised network such as the Internet, where all hosts can be contacted via the networking infrastructure.
– Network Reliability. Another consideration for application behaviour is the reliability of the network, i.e. whether the application can expect that a message or code fragment sent over the network will reach its destination, thus choosing its communication semantics.
• Parameters regarding code mobility
The following is information which we believe applications should be able to retrieve either through the middleware or calculate using available information:
– Size of the Mobile Code Unit. The application must be able to calculate what is the size of the unit that will be sent through the network for a particular mobile code paradigm. Combined with information on the network, this can determine how much the user will be charged, or how long will the particular computation take.
– Contacting the User. It is also important to know whether the user will need to be contacted with the result of the computation and whether it is desirable that the user be contacted in a short time interval. Given enough resources, if the user needs to be contacted with the result of the computation very quickly, it might be preferable for the application to execute a computation locally, if such a solution is feasible, rather than sending it through the network. Another consideration is how the user will be contacted if needed.
– Origin and Destination of Code. In a solution involving the transfer of code to and/or from another host, it is important to know where and how the code will be retrieved from or

transferred to. Apart from transfer issues, which are related to the underlying network connectivity, this can also determine whether the code received can be trusted.
– Available Resources. The application must also be aware of whether the device it is running on is capable to execute the code received, if the paradigm chosen for a particular function involves receiving code fragments over the network.

These are considerations that an application developer must take into account to determine the feasibility of using a mobile code paradigm, and choosing which one according to run-time information, to deliver the best result to the user. We believe that these parameters can be further identified into two main categories: Network-dependent parameters, which must be taken into account, and user-dependent considerations.

Moreover, work has begun on a designing a mobile computing middleware which has been named SATIN. Based around the concept of a very small registry of modules that provide different capabilities, we are aiming to allow modules to be dynamically added to and removed from the system at runtime. As it is still at the early design stage, it is desirable not to go into further detail that might change in the future.

Finally, what follows is a list of publications that have been made in the first year:

[1] Stefanos Zachariadis, Cecilia Mascolo and Wolfgang Emmerich. "Exploiting Logical Mobility in Mobile Computing Middleware". In Proc. of 22nd Int. Conf. on Distributed Computing Systems - WORKSHOPS (ICDCS 2002 Workshops). July 2002, Vienna, Austria.

[2] Stefanos Zachariadis, Licia Capra, Cecilia Mascolo, and Wolfgang Emmerich. "XMIDDLE: Information Sharing Middleware for a Mobile Environment". In Demo Session of ACM Proc. Int. Conf. Software Engineering (ICSE02). May 2002. Orlando , FL.

[3] Cecilia Mascolo, Licia Capra, Stefanos Zachariadis and Wolfgang Emmerich. "XMIDDLE: A Data-Sharing Middleware for Mobile Computing". In Personal and Wireless Communications Journal, Kluwer. April 2002.

[4] Licia Capra, Cecilia Mascolo, Stefanos Zachariadis and Wolfgang Emmerich. "Towards a Mobile Computing Middleware: a Synergy of Reflection and Mobile Code Techniques". In Proc. of the 8th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS'2001). Bologna, Italy. October 200

Appendix B: Literature survey

Sun Jini Network Technology
Sun Microsystems' Jini[3] technology is a distributed networking system, which allows devices to enter a federation and offer services to other devices or utilize services already offered. Jini, based on the Java programming language, exploits the inherent code mobility capabilities of that language in allowing devices to transparently locate and operate services offered. This distributed system of services is called in Jini terminology a djinn. Jini provides the infrastructure to build distributed services. A service in a Jini system is an object or a collection of objects provided by a computer, which can be utilised by other devices in the Jini federation. Examples include device drivers (such as printer drivers), whereby objects controlling the device can be operated remotely, time services, computational facilities (such as language parsers for example) and more. A Jini service can even include a user interface for the client to operate the service directly. Essentially, any object or collection o objects that perform a certain task or control a device can become a Jini service.

The Lookup Service; Discovering and Joining
The Jini architecture relies on the operation of servers called the lookup services. Lookup services are centralised indexes where objects advertise their services and clients can search for particular ones and receive the needed object reference to communicate with the service provider. This section considers what a lookup service is, how to discover it and how a service can join the lookup service. Lookup services are centralised indexing and managing mechanisms in a djinn. When a service provider connects to the network, it registers itself with a lookup service. Devices that need to find a service to perform a task must query the lookup service for it. There can be more than one lookup service in a djiin, each having a set of groups with which it is associated. A group is a simple string which categorises a set of services. Examples of group names can include "printers", although Sun suggests that implementers try to adhere to DNS-like names, such as "printers.cs.ucl.ac.uk". One special group is the public group, with which all lookup services should be associated. Group names have a similar functionality to DNS names; instead of looking up for particular groups via URLs and IP addresses, a device can find a lookup service for a particular group based on the groups name, even after a network reconfiguration. When a Jini device is plugged into the network, it looks for a lookup service for a group. This is part of the discovery protocol, which will be more thoroughly discussed below. If the device has a service to offer to the djinn, it registers it with the lookup service. Registration includes supplying to the lookup service an object that can be used to operate the service (a proxy object, an RMI stub etc) offered, as well as a set of entries. A Jini entry is an attribute aggregation system, which can include information such as the service's name, its location etc. Entries can be used by clients in looking up services. The lookup service and the service provider also negotiate a universally unique 128 bit service ID for the service being registered. Service providers are expected to employ a thread-safe implementation of the service, as multiple clients may by accessing the service concurrently.
Jini includes three different lookup service discovery mechanisms, outlined below:
• The Multicast Request Protocol is used by a device that wishes to discover a djiin. The device sets up a TCP server socket, and sends a multicast packet requesting lookup services associated with certain groups (or the public group). Lookup services listening for multicast requests and that are associated with the requested groups will connect to the TCP server socket of the device and proceed to use the unicast discovery protocol (see below) to send an object reference of the lookup service to the device. Note that the device may receive replies from multiple lookup servers matching the given group criteria.
• The Multicast announcement protocol is the reverse of the request protocol; In this scenario, devices interested in finding lookup services listen to multicast announcements sent periodically by the lookup services themselves. To obtain an object reference to an announcing lookup service, clients must then use the unicast discovery protocol, as described bellow. This mechanism can be useful in allowing clients to re-discover lookup services after

a network reconfiguration, a failure, etc. or to simply discover new lookup services as they are being started.
• The unicast discovery protocol operates as follows: Lookup services must listen for incoming requests. Upon locating a lookup service, either via one of the protocols above, or by using data such as user supplied information, a device sends a request to the particular host and port where the lookup service resides. It then receives a marshalled object (the Service Registrar) which can be used to operate the lookup service.

Locating and Utilising Services; Proxy Objects.
When a client wishes to utilise a service offered in a djinn, it must first obtain a reference to the service registrar object of a lookup service, using the discovery protocol as outlined above. When it does, it can query the registrar for the service needed, based on a service template, comprised by a set of optional information, such as the service ID, Entries etc. The lookup process returns the matches that satisfy the query. Note that the client can query multiple lookup services, to select the service which is more suitable to the task.
The result of the query can be null, if no service was found, or a set of objects representing the services found. Before using one of the demarshaled objects, the client needs to download all classes associated with the objects. The Java object serialisation framework[18] annotates the streams representing the objects with the location where the classes can be retrieved. This is done at the service provider's site. The annotation can be any URI parsable by Java. The usual choice is HTTP, and as such the Jini framework includes a minimal web server which can be used by service providers to allow clients to download any classes needed to operate the service. The JVM of the service provider must by provided with a value for the java.rmi.server.codebase property; the URI providing clients with the appropriate classes. Assuming that the operation of retrieving an object reference and downloading all classes needed to operate it was successful, the client can now use the object using the methods available. The object is usually a proxy object; the way it handles communication with the actual service is hidden from view. Examples include using the Java Remote Method Invocation (RMI)[19] mechanism, TCP sockets, etc.

Leasing, the Event System and Transaction Support
The Jini framework offers a few more services to clients and service providers. The Jini distributed leasing system allows clients to acquire access to a resource for a limited period of time. Upon requesting access to a service, the client can negotiate a lease with the service provider. The lease is time based, although the client can request an indefinite lease. The lease can be granted for the period asked, a smaller time period, or denied altogether. It is up to the service provider to make the choice. Leasing guarantees that the client will have access to the service for the period of time negotiated, assuming that there are no hardware or networking failures. The client can terminate the lease at will before it expires, and it can also negotiate a renewal of the lease. Leases can also be exclusive, meaning that only the client granted the lease will be able to access the service. Note that when a service provider registers with the lookup service, it too can negotiate a lease with the lookup service on how long the latter will advertise the service's existence. This service leasing model gives the Jini system resilience on failures. The traditional programming model that assumes that a service (or device) is in use until explicitly freed fails on a distributed system, as a client accessing a service may fail and never get a chance to free the provider to service other clients. Moreover, the leasing system allows providers to garbage-collect object references to clients no longer leasing the service. Once a lease expires, the provider can delete any such references and can refuse requests from these clients. Note that Jini offers helper classes which can manage negotiating leases automatically.
The Jini Distributed Events system allows programs running in one virtual machine to be notified of events occurring in a program running on another virtual machine. Jini defines a set of abstract classes and interfaces to allow applications to register for such events. Remote event registration is leased, and as such can expire in time. An object interested in an event can register a remote event listener to a remote event generator for a specific time period. The

remote event listener will then receive remote events by the generator. A remote event contains information such as the eventID which is used to identify the type of event, a reference to the event generator, a sequence number, which is always increasing and counts the number of events of this type, and more. Note that the remote event listener can choose whether to pass the remote event to the object interested in the event. This filtering process can be used if, for example, the object is running on a machine with limited resources which might not be able to cope with constant notification.
Jini also offers support for distributed transactions. The framework provides support for distributed objects to coordinate, via the completion protocol which consists of a two-phase commit (2PC) protocol. The default transaction semantics provide a way for the 2PC protocol to provide ACID properties, although objects are not obliged to follow the default semantics.

Networking and the Current Implementation
Sun's implementation of the Jini architecture requires hosts that want to participate in a djinn to have a functioning JVM and a TCP/IP protocol stack supporting TCP and UDP multicast. Multicast is only used in the discovery process. Moreover, service providers need to provide a mechanism for allowing clients to download the code needed, with a simple HTTP server being the recommended solution. This can also be provided by some cooperating party. The current implementation of Jini is heavily RMI-based (for example the lookup service, "reggie", is implemented as an RMI proxy), although the specifications do not fundamentally require this technology. Although the implementation currently weighs in at around 400KB, starting and running the lookup service, a service and the web server required to download classes, can be quite demanding.

Jini and Mobile Services
Jini technology can be used to provide context-aware mobile services. A user with a Jini-enabled PDA can walk into a djinn and instantly be greeted with list of available services. For example, booking tickets for the cinema, ordering food at a restaurant etc, using a graphical user interface given by the service provider. We feel that this is the environment that the application of Jini is suited for, at the mobile services domain; Nomadic computing devices, that are connected to a djinn temporarily, and can utilise the services offered. Jini technology could in theory also be applied in a ad-hoc scenario, but that assumes that at least one of the peers involved will take the role of the lookup service. It is not suitable for rapidly changing ad-hoc networks. One of the major hurdles in the acceptance of Jini in mobile environments, is that the reference implementation offered by Sun is very demanding in resources. Psinaptic, has recently developed a version of Jini called JMatos[9], which is specifically geared for mobile devices. Occupying only around 100KB of storage on the device, JMatos does not use the RMI technology, and does not implement a proxy-based lookup service, resulting in a more efficient fully compliant jini implementation.

A Service Example: A Time Server
This section presents an example of a Jini Time Server and a client that utilizes it. It is implemented using RMI. Some code has been omitted for clarity reasons.
```
Service Interface TimeServiceInterface.java
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface TimeServiceInterface extends Remote {
/**
Returns the time at the server in the form
dow mon dd hh:mm:ss zzz yyyy
*/
public String getTime() throws RemoteException;
}
Service Implementation TimeService.java
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import net.jini.core.lookup.ServiceID;
import net.jini.core.discovery.LookupLocator;
import net.jini.core.entry.Entry;
import net.jini.core.lookup.ServiceRegistrar;
import java.rmi.RMISecurityManager;
import net.jini.lookup.entry.Name;
import net.jini.discovery.DiscoveryListener;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceItem;
import java.util.Calendar;
import net.jini.discovery.LookupDiscovery;
import net.jini.discovery.DiscoveryEvent;
import net.jini.core.lookup.ServiceTemplate;
public class TimeService extends UnicastRemoteObject
implements DiscoveryListener, TimeServiceInterface {

/**
* Returns the Local Time
*/
public String getTime() throws RemoteException {
System.out.println("TimeService.getTime() called");
return(Calendar.getInstance().getTime().toString());
}
public TimeService() throws RemoteException {
super();
}
public static void main(String[] args) {
LookupDiscovery discovery;
TimeService service;
try {
System.setSecurityManager(new RMISecurityManager());
service=new TimeService();
discovery=new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
discovery.addDiscoveryListener(service);
System.out.println("Searching for any LookupServices in
reach...");
Thread.currentThread().join();
discovery.terminate();
}
catch (Exception e) {
e.printStackTrace();
}
}
/**
* Registers with the given ServiceRegistrar
*/
private void register(ServiceRegistrar registrar) {
System.out.println("Registering with registrar "+registrar);
Entry[] entries=new Entry[1];
entries[0]=new Name("TimeService");
ServiceItem service=new ServiceItem(timeID,this,entries);
try {
//registers the service with a maximum leas time
registrar.register(service,Long.MAX_VALUE);
}
catch(Exception e) {
e.printStackTrace();
}
}
/**
* A Lookup service has been discovered and we're registering with it
*/
public void discovered(DiscoveryEvent event) {
ServiceRegistrar[] results=event.getRegistrars();
System.out.println("Found and registering with the following:");
for(int counter=0;counter<results.length;counter++) {
System.out.println("* "+results[counter]);
register(results[counter]);
}
}
/**
* What to do when a Lookup service has been descarded.
*/
public void discarded(DiscoveryEvent param1) {
}
/**
* This is the TimeService's ID
*/
private ServiceID timeID=new ServiceID(000l,000l);
}
```

Client Implementation TimeClient.java

```
import java.rmi.RMISecurityManager;
import net.jini.core.discovery.LookupLocator;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.entry.Entry;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.lookup.entry.Name;
class TimeClient {
public static void main (String[] args) {
try {
System.setSecurityManager (new RMISecurityManager ());
//Unicast Discovery Protocol
LookupLocator lookup = new LookupLocator ("jini://localhost");
ServiceRegistrar registrar = lookup.getRegistrar ();
Entry[] query = new Entry[1];
query[0] = new Name ("TimeService");
ServiceTemplate template = new ServiceTemplate (null, null, query);
TimeServiceInterface timeServer =
(TimeServiceInterface)  registrar.lookup  (template);  //searches  for  the
service
if (timeServer instanceof TimeServiceInterface) {
System.out.println("Calling server...");
System.out.println(timeServer.getTime());
}
else {
System.out.println("no object found :( "+timeServer);
}
}
catch (Exception e) {
e.printStackTrace();
}
}
}
```

Summary and Evaluation

We feel that Jini is a very interesting technology which employs Code on Demand and Remote Evaluation techniques to deliver distributed services in high-speed and long lived (preferably wired) networks. However, Jini is very much centralised, needing lookup services to operate. Jini does not really scale well on low bandwidth highly dynamic ad-hoc networks. Jini and especially the JMatos implementation can be used by mobile devices to deliver context aware services to mobile devices, connected to a fixed network nomadically. It is not, on the other hand, particularly suitable for allowing mobile devices to offer services themselves, particularly in ad-hoc environments which lack a centralised lookup service.

µCode

µCode[15] is a lightweight Java library which provides a minimal set of primitives allowing code mobility. µCode was specifically designed to provide programmers with a set of primitives to move Java Classes and Objects between hosts. The idea being that higher abstractions such as mobile agents can be built using µCode, but programmers would not be forced to use a heavyweight mobile agent platform simply to be able to use REV for example in a program. As such, µCode can be used to implement all code mobility paradigms described above. That being said, µCode does include a set of abstractions, including a mobile agent implementation, in a separate package. The core µCode package is very small, occupying around 20KB of memory. µCode can essentially enable applications to ship and fetch code as well as link classes dynamically. As µCode is a library on top of the JVM, it only offers support for weak mobility, not being able to transfer the state of execution. µCode also does not rely on the existence of an RMI implementation. It supports synchronous and asynchronous invocation together with immediate and deferred execution of code received. µCode also allows programmers to compress the code being sent, increasing the computational power required, but reducing the time required for the actual transfer.

Groups, Objects, Classes and the µCode Architecture

The only unit of mobility in µCode, is the group. A group is a collection of classes and objects, defined by the programmer. There is no restriction on the objects and classes that can

be contained in a group, apart from the fact that anonymous classes are not supported. Additional µCode utilities enable computing and adding to a group the full closure of a class. Furthermore, the programmer can specify a location from which additional classes, not contained in the group, can be downloaded, if needed, enabling dynamic class loading. A group can contain two special classes: The group handler and the root class. The group handler is defined by a special interface, and is used to instantiate an object which will be used to unpack and manipulate the contents of the group. The root class can be used to provide additional information on the group, for example information on how to spawn a new thread of execution at the destination. Note that the handler and the root class can actually be the same class and it is not necessary for them to be included in the group. It is however the responsibility of the programmer to ensure that they will be available at the destination. µCode groups must be bound to a specific MuServer (see below).
The destination of a µCode group is the MuServer. The MuServer provides the runtime support for the µCode platform. A MuServer can create a µCode group and can receive groups sent over the network. Receiving groups or requests for dynamic linking is optional, and has to be enabled specifically by the programmer, by calling the boot() method. This starts a thread in the MuServer, which opens a TCP server socket waiting for connections from incoming groups or dynamic linking requests. Upon receiving and handling a group, the classes are kept in a private class space, which is the group's private namespace. A MuServer also provides a shared class space, to which classes from other namespaces (including private ones) can be published. The use of class spaces disallows name clashing and overriding of the classes in the default Java Class Library, while at the same time permitting sharing classes between groups and hosts. In order to achieve this, µCode implements a customized version of the Java Class Loader, the MuClassLoader. When a thread, t, requests a class, c, MuClassLoader takes the following steps:
• Checks whether c is a ubiquitous class. Ubiquitous classes are classes that are available in the all µCode servers, such as classes belonging to the Java or the µCode API.
• If not, it checks whether c is in the private class space of t.
• Otherwise, it checks whether c is in the shared class space of the MuServer.
• If not, the MuServer can download the code remotely
• Otherwise the class was not located and an exception is thrown.
If a dynamic linking request is received, the MuServer sends the class requested, if it is published in the shared class space.

Evaluation

µCode is particularly interesting in the way that it offers a very lightweight set of primitives to support code mobility. Its non-obtrusiveness allows it to be easily integrated with various middleware systems, and its small footprint makes it suitable for mobile middleware. For example, it is basis of the mobile agent system used in Lime. The µCode unit of mobility, the group, allows programmers to ship variably sized of code depending on the network the device receiving or sending the code is currently connected to. For example, if a portable device that has enough memory is connected to a wired, high-bandwidth network, it might make sense to download all objects and classes that might be needed for a particular function, if it is anticipated that the device will be disconnected from the high-speed network in the near future. The fact that classes can be embedded in a group is an added benefit, as it can allow clients to use the objects contained in a group without requiring a separate download of their respective classes. The ability to optionally compress the group is also beneficial to the programmer. Moreover, the use of the class spaces is an elegant way to solve class naming and overriding problems, as well as to allow groups to share classes with others. It is also worth noting, that µCode is licensed under the GNU Lesser General Public Licence (LGPL), a Free Software license. The main disadvantage we can find with µCode, is its lack of support for security. Anyone can connect to a MuServer without any authentication. This issue is however, currently being worked on.

Lime
Linda in a Mobile Environment (Lime)[14, 16] is a middleware implemented in Java, which allows the development of applications which exhibit logical and/or physical mobility characteristics, by providing a coordination model based on the Linda tuple space model. Lime is primarily geared for ad-hoc networks, although it is not limited only to such configurations.

Linda
Linda is a communication model for concurrent processes, which was developed at Yale University in the mid-1980s. Linda processes communicate using a shared repository of tuples, the tuple space. A tuple is an elementary data structure, a sequence of typed parameters, such as ("bar",1), and represents the information being communicated. A tuple, t, can be deposited to the tuple space using the out(t) operation, and can be retrieved using the in(p) operation, where p is a pattern matching the tuple returned. If no tuple currently matches the pattern, the requesting thread is blocked, until the pattern is matched. If more than one tuples match the given pattern, the one returned is selected non-deterministically. in(p) removes the resulting tuple from the tuple space. The model includes an operation to read a tuple from the tuple space without removing it, rd. The tuple space can be accessed concurrently by threads and processes, the resulting model providing spatial and temporal decoupling.

Agents, Tuple Spaces and Sharing Tuples
Lime exploits the decoupled nature of tuple spaces to provide coordination primitives and information sharing for mobile components. The unit of mobility in Lime is a mobile agent, with mobile hosts acting as simple containers for those agents. Special scenarios of this configuration are stationary agents, which are also supported by Lime. An agent is in reach with other agents if it resides on the same host as they, or if the hosts of the agents are in reach. Each agent can have a set of tuple spaces which are identified by their name, a String. The tuple spaces are bound to the agents and as the agent migrates, the tuple spaces migrate as well. The agent can choose whether to share a tuple space it owns. Lime makes all tuple spaces with the same name, marked as shared by agents in reach, transparently appear as a single tuple space. Thus, agents who have a local tuple space named "DOCUMENTS", will transparently be able to access data in any tuple space named "DOCUMENTS" that has been shared by any agent that is currently in reach, through operations on their local tuple space. Lime includes the "SYSTEM" tuple space which is a read only tuple space containing information such as host information, the agents and tuple spaces on the host etc. Hosts are identified by their LimeServer id, and agents are identified by an agent id. The LimeServer is the Lime runtime support that a host needs to be running and the identifiers must be globally unique. Lime communities are dynamically formed as hosts and their corresponding agents become in reach. Engaging a community, as this operation is termed, dynamically reconfigures the tuple spaces of all hosts in the community to reflect the new contents available. Both engaging and disengaging (or disconnecting) are atomic actions. Note that hosts with no shared tuple spaces are considered to be disengaged from the community, even if they are in reach. Lime extends the standard Linda operations, to support location based computing. To place a tuple, t, in a tuple space of a specific agent, a, out[a](t) can be used by an application. As such, each tuple is augmented by the Lime run-time support by two identifiers: The current location and the intended destination location. If an agent deposits a tuple into one of its tuple spaces using the out(t) operation, then the current and intended destination locations are equal. Using out[a](t) does not guarantee delivery of t to a. It might be the case, for example, that agent a might not be currently in reach. This results in a tuple that its current location is different from its intended location. When the intended agent engages the community, the runtime will automatically transfer the tuple to the intended tuple space of the agent. Lime also extends the Linda in and rd operations, annotating them with location information. in[c,d](p) and rd[c,d](p) retrieve and read respectively a tuple the current location of which is c, but the intended destination location is d, matching a pattern p.

c may be equal to d, and either can be substituted with the wildcard _, meaning any. For example, in[_,_](p) is equivalent to the standard Linda in(p) operation. Lime also allows agent to react to changes in context, by defining the reaction primitive. A reaction r(c,p) specifies a code fragment, c, that is executed when the tuple matching pattern p is found in the tuple space. As such, whenever the contents of the tuple space change, as a result, for example, of the engagement or disengament of a host, if a tuple matching p is found, c is executed. Moreover, a reaction is annotated with location parameters, similar to in or rd. However, the current location of the pattern must be limited to a specific agent. These are called strong reactions whereby the execution of c happens synchronously with the detection of a tuple matching p. As such, blocking operations are not allowed. Reactions over federated tuple spaces are allowed, but the execution of c is guaranteed to take place some time after a matching tuple is detected, provided that connectivity is preserved.

Implementation & Evaluation
Lime currently uses μCode as a mobile agent library. The implementation does not support ad-hoc disconnection, and as such, disengagement from a Lime community can only happen through an explicit API call. Moreover, in order to form a Lime community, the current implementation requires a form of bootstrapping, by selecting a community leader, which is then free to disengage from the community if needed. Lime provides application developers with a data-sharing middleware, geared for mobile agents and ad-hoc networks, although it can operate under fixed networking structures as well. It does not currently provide any form of security and the fact that it only offers a flat tuple space as the only common data structure, limits the processing that can be made on the shared information. Moreover, there appear to be scalability issues with the concept of misplaced tuples.

PeerWare
PeerWare[6] is a new mobile middleware model, offering peer-to-peer communication, event subscription and a shared data space. PeerWare binds the models of event-based communication and data sharing into a single middleware, using mobile code and REV in particular, to distribute operations on remote data.

Global Virtual Data Structures and the PeerWare Data Structure The PeerWare system is based around the concept of a Global Virtual Data Structure (GVDS). A GVDS is a communication and coordination meta-model for mobile environments. It is basically a generalisation of the Lime coordination model. A GVDS provides a global data space that is made dynamically by the local data spaces of each peer in range. It is virtual, as it does not exist on any host as a single entity. The GVDS meta-model does not specify how the
GVDS is structured, leaving it to the implementer. The PeerWare data structure is a graph of nodes and documents, which are collectively referred to as items. Nodes are simple containers of items, and are structured as a forest of trees, with a distinct root. Nodes have a label which cannot be shared with another node if both are roots or contained directly into the same node. This classification and organisation of nodes allows for expressing complex document organisation schemes, the resulting model resembling a standard file system. Each PeerWare-enabled host has a data structure stored locally. PeerWare dynamically constructs a GVDS by superimposing all the nodes of the local data structures of all peers in range. This function is completely hidden to applications, that must only be aware that the contents of the data structured can eventually change.

PeerWare Operations and Mobile Code
PeerWare makes a sharp distinction between operations that can be performed on the local data structure and on the GVDS. This distinction, even if it lacks transparency, gives the application programmer explicit knowledge of whether he/she is operating locally or globally. Hiding this difference could make programmers use the available operations in inefficient ways. The following operations are available only on the local data structure:

• createNode(n, n2) and removeNode(n) add and remove nodes (n) to and from the local data structure respectively.
• placeIn(d, n) and removeFrom(d, n) place and remove documents (d) to and from a node (n) in the local data structure respectively.
• publish(e, i) issues a notification that a given event, e, has occurred on a given item, i.

The following are the operations available on both the local and global data structures:
• I = execute(FN, FD, a). This operation performs the following steps:
– Executes FN, the node filter function, on the nodes of the data structure, and returns a set, MN of the matching nodes
– Executes FD, the document filter function, on all the documents contained in the nodes of MN and returns a set MI of matching items
– Executes action a on MI and returns the results, I, to the caller.
• subscribe(FN, FD, FE, c) allows the caller to subscribe to an event defined by FE, the event filter function, on the data structure identified by FN and FD (as above), executing code fragment c, when the event occurs.
• I=executeAndSubscribe(FN, FD, FE, a, c) performs both execute and subscribe operations as defined above, but in a single atomic step.
PeerWare exploits logical mobility, by considering the execution of an action on the GVDS, as a distributed execution of the action on the local data structures of the connected peers. It is designed specifically to provide a minimal set of operations, with application-specific primitives and abstractions constructible via the execute mechanism, giving flexibility to the developer.

Evaluation
PeerWare exploits logical mobility, REV in particular, effectively, to move computations to the data sources in a peer-to-peer environment. It is, in a sense, the next generation of the Lime system, providing a hierarchical data structure instead of the Lime's flat tuple space as a GVDS. However, the PeerWare model does not prescribe anything about routing and networking infrastructures and as such multiple different implementation must and are being provided to work in different settings, such as ad-hoc, nomadic, etc. It does not provide a single middleware which can seamlessly operate in various different networks. Moreover, it has been argued [8] that moving the code to the data is not always better than the traditional communication model of moving the data to code. Current implementations utilise µcode as a logical mobility layer to move code to the peers.

An Augmented Reality System
In this system[10], it is proposed to use COD techniques to allow adding virtual objects into the real world view of the mobile user, using computers projecting into the user's sensory systems. The core idea behind this project, is that physical objects which are to implement a virtual object will be equipped with an active tag. The tag contains a wireless communication device, such as a Bluetooth adaptor, a micro controller equipped with some memory and an optional interface to the physical object, which can be used by the virtual object to interact with the former. Mobile users equipped with mobile computers implementing the augmented reality (AR) system will locate the active tags. The location mechanism is not specified. An object as specified by this project, is considered to be either an active tag, an AR system or some other device. Each object is characterised by a locally unique address: This implies that no two tags with the same address can be in reach. Both the AR system and the tags implement the Mobile Code protocol, which allows for querying object identification and information, requesting mobile code, requesting permission to send code and sending arbitrary data. The mobile code implementing the virtual object is stored in the tags memory. Upon locating a tag, the AR system can request the code contained in the tag and execute it. Alternatively, the tag can request permission to send the code to the system; this allows for approaches where physical and virtual objects inform and/or notify all visitors of a specific

event. Upon receiving the code, the virtual object can interact with the tag via a communication link established between them.

Evaluation
This system offers an approach of using COD over an adhoc network to implement an AR system. This system is considered to be more scalable than traditional approaches in this field of research relying in centralised servers providing the virtual objects. The design is open-ended, not specifying the form of the mobile code that is going to be used. The system can support multiple forms of mobile code, and each tag identifies the type of code that it contains. However this approach is also very expensive, requiring a computing environment for each different object.

FarGo
FarGo[23] is a system that provides dynamic application layout support and a monitoring service that allows applications to register and react to specific system events. Dynamic application layout allows for the mapping of the application logic onto hosts to be manipulated at runtime. FarGo, which is implemented as an extension of Java, provides component mobility, allowing components to be attached to the same address space (or host), or conversely, detached into different address spaces.
The basic unit of mobility in FarGo is the complet. A complet, analogous to an application module, is a collection of objects with a FarGo application being typically comprised of a collection of complets. Each complet has an object, the anchor, the interface of which is the interface of the complet. Objects in a complet always reside within the same memory space and as such, normal references and method invocations are used for intra-complet communication. Inter-complet communication and complet interconnection happens using complet references, which are the major abstraction mechanism that is used for layout programming in FarGo. Complet references are used to interconnect two complets, which may reside on the same host or two different hosts. Although syntactically, using complet references is similar to local references, there do exist semantic differences, as parameters are passed by value.
When a complet moves from one host to another, all complet references are updated so that they remain valid. Furthermore, complet references can be augmented with semantics that identify the relationship between two complets when one of them moves. For example, the programmer can have a pull complet reference, which translates to having a complet follow the other when it relocates. What FarGo does, is provide a runtime infrastructure on all hosts that is used to keep complet references valid, even after a relocation has occurred. This is provided by a set of distributed objects called the core objects, which also provide support for naming and mobility. Each complet is associated with exactly one Core at any given time, although as the complet relocates, the Core association may change. The Core services are usually abstracted from the application programmer.
FarGo also provides monitoring facilities, which allow the application programmer to register for events in the current context (for example bandwidth) and react accordingly (for example move the complet to another host). This behaviour can be programmed directly from within the application using an API that is provided, or, alternatively, it can be encoded outside the application using a rule-based scripting language.

FarGo-DA
FarGo-DA[24] is an extension of FarGo, providing a mobile framework for resource-constrained devices that allows disconnected operations.
FarGo-DA works on the assumption that resource constrained mobile devices, named Networked Lightweight Portable Computing (NLPC) devices access services that are located on various servers. However, as NLPCs are portable, it is assumed that the servers will at some point be out of the reach of the devices. It extends FarGo to allow for complet migration, replication, replacement and merging, which will all be examined later on. It

allows application developers to add to their application disconnection and reconnection semantics.

FarGo-DA extends the original Fargo complet to a disconnected aware (DA) complet. A DA-complet can prepare for disconnection and reconnection to the network by implementing the preDisconnect and postReconnect methods that are exported by the DA-Listener interface. These are executed by the local Core before a disconnection and after a reconnection. As for complet references, FarGo-DA provides a set of DA reference types, which can be used by application programmers to describe how to maintain the validity of complet references on a connection, disconnection or both. The types available are the following:

• Clone denotes that prior to disconnection, the target complet should be duplicated and migrated to the local NLPC.

• Replace changes a reference to a remote complet with a reference to a local complet that has the same interface as the remote one, but usually a different, more lightweight implementation and as such may offer reduced functionality.

• Store and Forward queues all invocations made for the remote complet after disconnection and forwards them to it when it is again available. This assumes that requests are one-way only thus allowing the local complet to continue to operate as normal.

• Depart offers a batch-mode approach to invocations that is very similar to store and forward. When in disconnected mode, it operates exactly as store and forward. In connected mode however, it queues all requests to the remote complet, and transfers them prior to disconnection.

• Purge implies that upon reconnection, the replaced or clone complet will be overwritten by the original complet in the reference.

• Overwrite on the other hand, implies that the cloned object overwrites the original one.

• Last keeps the complet with the latest timestamp. This assumes that the NLPC and the server have synchronised clocks and that the modification times of the complets are stored in the system.

• Merge allows for merging the two complets. FarGo-DA provides two merging mechanisms, one based one callback methods and another based on merging operators, which are considered outside the scope of this report.

Evaluation

FarGo and especially FarGo-DA provide a methodology based on COD and CS interactions that allows applications to be distributed using a dynamic layout and allows for thin clients, or NLPCs to still have (somewhat limited) access to the services that servers provide, even when disconnected from those services. However, we can identify three main disadvantages of FarGo with respect to our approach: FarGo-DA assumes that disconnections are pre-announced. This limits its applicability in mobile networks, where disconnections are frequent and involuntary. Moreover, FarGo-DA does not provide any support for mobile devices that offer services themselves to other peers. Finally the current implementation of FarGo-DA is based on RMI, which we have found to be very heavyweight to be used on mobile devices.

Appendix C: Timescales

The following 2 months of the project will be spent trying to research into different ways of sending code from one host to another. This will involve investigating the types of code sent (interpreted, virtual, etc.), how to group code together and when and how to send it.

After this work has been completed, 1 month will be spent to design and implementing logical mobility modules to SATIN, with an additional 3 months needed to complete the design of the our mobile middleware.

Completing this part of the research, work will be focused on the second part of the testing of the hypothesis. 2 months will be spent in background reading of literature in design methodologies, queuing networks and modelling techniques. The following 4 months will then be spent in designing and testing our methodology and one more month will be needed to implement it in a design tool.

The remainder of the time will be spent in designing and developing applications using our methodology and our middleware, and testing their operation in various environments. We expect that 3 months will be needed to write the thesis.

Appendix D: Bibliography

[1] Stefanos Zachariadis, Cecilia Mascolo and Wolfgang Emmerich. "Exploiting Logical Mobility in Mobile Computing Middleware". In Proc. of 22nd Int. Conf. on Distributed Computing Systems - WORKSHOPS (ICDCS 2002 Workshops). July 2002, Vienna, Austria.

[2] Stefanos Zachariadis, Licia Capra, Cecilia Mascolo, and Wolfgang Emmerich. "XMIDDLE: Information Sharing Middleware for a Mobile Environment". In Demo Session of ACM Proc. Int. Conf. Software Engineering (ICSE02). May 2002. Orlando , FL.

[3] K. Arnold, B. O'Sullivan, R. W. Scheifler, J. Waldo, and A. Wollrath. The Jini[tm] Specification. Addison-Wesley, 1999.

[4] Licia Capra, Cecilia Mascolo, Stefanos Zachariadis, and Wolfgang 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), Bologna, Italy, October 2001.

[5] Compaq Computer Corporation. Compaq iPAQ H3600 Hardware Design Specification. Specification http://www.handhelds.org/Compaq/iPAQH3600/iPAQ H3600.html, Compaq Computer Corporation, May 2000.

[6] Gianpaolo Cugola and Gian Pietro Picco. Peerware: Core middleware support for peer-to-peer and mobile systems.

[7] J. Reilly D. Axtman, A. Ogus. IrDA Infrared LAN Access Extensions for Link Management Protocol. Specification http://www.irda.org/standards/pubs/IrLAN.PDF, Infared Data Association, July 1997.

[8] A. Fuggetta, G.P. Picco, and G. Vigna. Understanding Code Mobility. IEEE Trans. on Software Engineering, 24(5).

[9] S. Hashman and S Knudsen. The application of jini technology to enchance the delivery of mobile services. http://www.sun.com/jini/whitepapers/PsiNapticMIDs.pdf, December 2001.

[10] J ning K. Kangas. Using code mobility to create ubiquitous and active augmented reality in mobile computing. In Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM), 1999.

[11] D. B. Lange and M. Oshima. Programming and Deploying JavaTM Mobile Agents with AgletsTM. Addison-Wesley, 1998.

[12] C. Mascolo, L. Capra, S. Zachariadis, and W. Emmerich. XMIDDLE: A Data-Sharing Middleware for Mobile Computing. Int. Journal on Personal and Wireless Communications, April 2002.

[13] R. Mettala. Bluetooth Protocol Architecture. http://www.bluetooth.com/developer/whitepaper/, August 1999.

[14] Amy L. Murphy, Gian Pietro Picco, and Gruia-Catalin Roman. Lime: A Middleware for Physical and Logical Mobility. In Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS-21), May 2001.

[15] Gian Pietro Picco. Mucode: A lightweight and flexible mobile code toolkit.

[16] G.P. Picco, A. Murphy, and G.-C. Roman. Lime: Linda meets Mobility. In Proc. 21st Int. Conf. on Software Engineering (ICSE-99), pages 368–377. ACM Press, May 1999.

[17] Sun Microsystems. The Java Language Specification, Oct 1995.

[18] Sun Microsystems. Java Object Serialization Specification, 1998.

[19] Sun Microsystems. Java Remote Method Invocation Specification, Revision 1.50, JDK 1.2 edition, October 1998.

[20] S. Kerry V. Hayes. IEEE P802.11. Specification, Institute of Electrical and Electronics Engineers, Inc. (IEEE), 1996.

[21] J. White. Telescript Technology: Mobile Agents. In J. Bradshaw, editor, Software Agents. AAAI Press/MIT Press, 1996.

[22] D. Wong, N. Paciorek, and D. Moore. Java-based Mobile Agents. Communications of the ACM, 42(3):92–102, 1999

[23] H. Gazit O. Holder, I. Ben-Shaul. Dynamic layout of distributed applications in fargo. In Proceedings of International Conference on Software Engineering, pages 163–173, 1999.

[24] Y. Weinsberg, 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, 2002.

[25] V.Grassi, R.Mirandola. PRIMAmob-UML: a Methodology for Performance Analysis of Mobile Software Architectures. In Proceedings of the Third International Workshop on Software and Performance, 2002.

Appendix E: Table of contents