Using Mobile Code for Service Integration in Ubiquitous Computing

Kari Kangas and Juha Röning
University of Oulu, Department of Electrical Engineering
Computer Engineering Laboratory
P.O. BOX 4500, FIN-90401 Oulu, Finland
{macconen, jjr}@ee.oulu.fi

Abstract

As embedded systems become ubiquitous in the future, it will be a challenge to use in an integrated fashion the services they provide in other computing infrastructure. However, such cooperation is essential, as it facilitates some of the most interesting aspects of ubiquitous computing. The goal of this paper is to demonstrate that mobile code helps to implement service integration between ubiquitous embedded systems and the devices used to control them. To achieve this goal, we will first explain our approach to using a single device to control diverse embedded systems. After that, we will present our approach to service integration. We will demonstrate, by using print service as an example, how the control device can use the services provided by the embedded systems in its environment. In the print service example, we will describe two different ways to implement service integration: automatic and user-assisted integration. These two ways can be used to complement each other. The main advantage in our approach is that it allows the service structure in the control device to be highly dynamic. Our approach also makes it possible to implement rich service interfaces in a way that allows the embedded systems to remain relatively simple. Furthermore, each system may also contain several different service interfaces written in different mobile code languages, which are easy to update.

1. Introduction

Embedded systems are becoming an integral part of our life. Although we already encounter a large number of devices controlled by hidden computers every day, we take them for granted. However, it has been envisioned that, in the future, embedded systems will become so ubiquitous that we will feel downright uncomfortable without their presence [1].

When these ubiquitous embedded systems become reality, it will be challenging to use in an integrated fashion the services they provide in other computing infrastructure. However, the need to make services and systems cooperate is essential, as we feel that the most interesting aspects of ubiquitous computing will be lost if each system exists in isolation. The Internet is a prime example of the possibilities provided by cooperation between different computing systems.

In this paper, we will present an approach that will help to implement service integration in ubiquitous computing. Our approach draws much of its power from the utilization a of mobile code. For the purpose of this paper, a mobile code is an executable information object that originates from one computer system and is executed in another. These systems use a communication network for transfer. Once the mobile code has finished its execution and is no longer needed, the executing system usually discards it. A Java applet is an example of a mobile code component compatible with our definition. See [2] for a detailed discussion about the different aspects of mobile code. The reader should note, however, that we do not want to specify the contents of the mobile code. We only require that it contains information that is meaningful for a given mobile code execution environment, so that this environment can use the information for execution.

In our approach, we use a single handheld or wearable control device to control diverse embedded systems that we may encounter in our everyday life. This device contains significant processor, memory, and display capacities that are comparable with, or even exceed, those available in the current laptop computers. It also contains communication devices for both long and short-range communication. The long-range communication device is similar to the current cellular phones, and it is used to connect the control device to the Internet. However, we assume that the long-range link cannot be active all the time, maybe because it is expensive or energy-consuming. Because of this, the control device should use the short-range communication device whenever possible.

The short-range communication device has characteristics similar to those provided by Bluetooth [3]. It has relatively high bandwidth and low power consumption, but only a limited communication range. It also enables *ad hoc* connections that communication devices can establish when they come close enough to each other. Our idea of a control device resembles the possible future versions of current high-end cellular phones, such as Nokia 9110 Communicator [4]. Moreover, the CyPhone concept [5] is a detailed example of what such a device might be like.

The main assumption in our approach is that the control device uses the services provided by the different embedded systems present in its immediate vicinity. The device may also use other services, but, to simplify our presentation, we will ignore them. As the control device is potentially highly mobile, the set of available services at any given moment is notably dynamic. We try to solve the problems of dynamicity as follows. When the control device needs a service that is not currently available, it scans the environment for an embedded system that could provide the service. Once the device has located such a system, it downloads from the system a mobile code that, when executed, will provide the required service. The mobile code provides a standard service interface for the control device and carries out the possible communication with the embedded system. Once the service is no longer needed or the embedded system is no longer in the vicinity of the control device, the mobile code is terminated and discarded. Simply put, the mobile code operates as a device driver for that particular embedded system.

For example, consider a situation where we want to print a document from a text editor running in the control device. When we press the editor's print button, the control device scans its environment to see if an embedded system providing the print service is present. If the service is not available, the user can cancel the print job or spool it to be complete later. If the print service is available in some system, the control device downloads a certain type of mobile code from that system. When executed, this code provides the print service interface that the text editor can use for printing. When the service is invoked, the mobile code carries out the subsequent communication with the printer in order to complete the service, i.e. the code sends the print data to the printer. After the mobile code is no longer needed, the control device terminates and discards it.

The goal of this paper is to demonstrate that mobile code, if utilized as in our approach, will help to implement service integration between ubiquitous embedded systems and control devices. The main advantage in our approach is that it allows the service structure in the control device to be highly dynamic. It also makes it possible to implement rich service interfaces in a way that allows the embedded systems to remain relatively simple. Furthermore, one system may also contain several different service interfaces written in different mobile code languages, which are easy to update.

Finally, the reader should note that although our approach can be used to implement some service integration between embedded systems, we assume here that the interaction is asymmetric. Simply put, we assume that simple embedded systems provide services for powerful control devices. However, we can use our approach for service integration if simple embedded systems can use more powerful systems to execute the mobile code. For example, domestic embedded systems could send their mobile codes to the home server for execution. The mobile codes could then negotiate service integration between different embedded systems. This resembles somewhat the approach taken by HIVE [6], for example.

This paper is structured as follows. Section 2 illustrates related work. Section 3 describes briefly the CUES system that we use to implement service integration. Section 4 describes both automatic and user-assisted service integration. Section 5 discusses the benefits and drawbacks of our approach and illustrates our plans for future work. Section 6 draws a conclusion.

2. Related Work

Currently, normal desktop operating systems (OSes) use device drivers to integrate different devices (and embedded systems) into themselves. These drivers provide a level of abstraction by hiding the internals of each physical device under a well-defined device interface. For example, applications make no difference between a network printer and a printer connected to the printer port, although the drivers for these devices are internally quite different. The drivers are, however, usually static in the sense that the OS does not expect the set of available devices to be highly dynamic (although techniques like Plug-and-Play allow some dynamicity). The drivers are usually set up when the OS is installed or a new device is added to the system.

The approach used in desktop OSes, if used for ubiquitous computing, has several problems. Since different embedded systems (devices) usually require different drivers, we would need to store a large number of drivers in the control device for the possibility of encounterting such specific systems. For the same reason, we could never know when to discard the drivers that have not been used for a long time. Moreover, unknown or new systems would still cause problems. Finally, installing a driver for a new device is a disturbing event in most operating systems, and not something you would want to do frequently.

Another widely used approach to establish interaction with networked embedded systems is to install into the control device a separate program for each system. This program may then provide some form of service integration.

In essence, this is similar to, or even worse, than the device driver approach, and thus clearly not a feasible solution in ubiquitous computing.

We can eliminate the need for a separate program for networked embedded systems by defining standard device interfaces for different classes of systems and by exposing this interface to the network. For example, VCRs would implement a VCR interface, etc. In this way, we can use a single program to access the services of systems of certain type. In contrast to the device driver approach described above, the embedded system itself implements the device interface, and the control device accesses this interface via the network.

For example, we can define CORBA [7] or some other programming language independent interfaces for different types of embedded system. This is the approach used by Home API [8], and Inferno [9] also uses predefined device interfaces to integrate networked embedded systems into an operating system. Moreover, Ierusalimschy et al. [10] describe a technique that uses reflexivity to dynamically interface with CORBA objects. The control device could use this technique to inspect and use the services provided by an embedded system without using stubs or pre-defined IDL headers. We are not, however, aware of any system that would be actually using this technique to interact with embedded systems.

As stated earlier, Inferno [9] provides a way to integrate networked embedded systems into an operating system. However, this is most conveniently achieved by running a special Inferno virtual machine in embedded systems and by using a specific programming language, Limbo. The reader should note that Inferno also contains support for a mobile code, and it could therefore probably be used as HAVi or Jini (see later). In addition, Universal Plug-and-Play (UPnP) [11] is another possible way to achieve connectivity between networked embedded systems. Instead of fixed APIs, UPnP uses Extensible Markup Language (XML) to describe the device interface. This allows, for example, different vendors to implement different APIs on top of the XML layer.

The main problem with the standardized device interfaces used in ubiquitous computing is that the embedded system itself contains and executes the device interface. We must design the standard interfaces carefully, because too complex interfaces may require the embedded systems to possess excessive computing resources. Moreover, there will be a large number of different embedded systems, and the services they provide may have very little in common with each other. Likewise, new embedded systems and possible services are likely to be introduced all the time. Because of this, the interfaces will probably continue to evolve. Unless clever general-purpose interfaces can be constructed, perhaps by using the interface polymorphism, or there is a convenient way to update the device interfaces, this will eventually lead to problems.

Some of the problems related to fixed device interfaces that are run by the embedded system can be solved by using a mobile code. The mobile code, i.e. the embedded system's external interface, is usually downloaded from the system and executed by the control device, which may reduce resource requirements. Likewise, the mobile code is usually easier to update than the device interfaces executed by the embedded system, and embedded systems may easily contain several different interfaces defined by different mobile codes. HAVi [12] provides a mobile code approach using the services provided by various embedded systems that resembles our approach. However, HAVi assumes that Java is used as a mobile code, whereas our approach also allows other types of mobile code. Moreover, HAVi is intended to be used only in home networks, and it currently operates only with embedded systems supporting the IEEE 1394 and IEC 61883 interface standards. It should be noted that HAVi also supports device interfaces that are executed by the embedded system.

Jini [13] is another platform using mobile code and promising convenient integration of networked embedded systems. It emphasizes the use of mobile code in service integration. The Jini approach differs from our approach in that we do not use a lookup service as a mobile code mediator, but the control device requests the mobile code directly from the embedded system. However, Jini draws much of its power from the assumption that the embedded systems in Jini contain a Java Virtual Machine (JVM) and use the Java programming language.

The Java Embedded Server (JES) [14] also utilizes mobile code and provides dynamic service loading and removal. Although JES is usually targeted to be used in embedded systems, we assume that we could also use JES in the control device to implement similar functionality as provided by our CUES system. JES also draws much of its power from the JVM and Java. JES, when combined with Jini, may offer a very powerful platform for deploying networked embedded systems.

As described earlier, Inferno, HAVi, Jini, and Java Embedded Server draw their power from specific virtual machines and/or programming languages. We agree that it makes perfect sense to limit the number of different types of mobile code, even when using our approach, since after all, the mobile code makes integration possible. However, we feel that there should also be room for different types of mobile code, possibly even for a code that has not yet been invented. In addition, we feel that the embedded system manufacturers should not be forced to implement their systems by using specific virtual machines or programming languages, at least if they can implement the required functionality by other means better suited to their needs.

Java applets are also used to control embedded systems. For example, the GLASS system [15] uses applets for remote monitoring of embedded systems in power engineering. Applets do not, by default, allow service integration, but are similar to the virtual user interfaces we use for control. The reader should note, however, that signed applets can read files in the executing computer, which means that they could be used to implement similar functionality as we describe in user-assisted service integration.

Moreover, systems described by Kortuem et al. [16] and Hodes et al. [17] also use mobile code to control embedded systems. The system by Kortuem et al. currently provides functionality that is roughly equivalent to our virtual user interfaces. The system by Hodes et al. provides a mapping system for services provided and services needed, but their work focuses on control rather than service integration. Both of these systems use external code servers to store the mobile code, and the embedded systems are thus not self-contained. As our approach facilitates self-contained embedded systems, we feel that it is better suited to ubiquitous computing.

CyberDesk [18] is a system providing self-integrating services. CyberDesk automatically matches the services provided by some applications to the services needed by others. Although CyberDesk is currently used to integrate different programs running inside the CyberDesk framework, we assume that some of its ideas could be utilized to remove the necessity of fixed service interfaces in our current approach.

Finally, active networks [19] constitute one of the most active research areas that utilize mobile code. We also feel that some of the mobile code issues presented in active networks are very helpful for readers who wish to know more about the different aspects of mobile code. See [20] for references and for the most up-to-date survey on the issues in active networks.

In active networks, the network nodes, i.e. routers, contain programmable interfaces that enable dynamic modification of network behavior. This is usually achieved by using mobile code extensions that can be uploaded/downloaded to the network routes dynamically and/or by replacing normal network packets with capsules that contain not only the payload data but also a mobile code. The mobile code performs capsule-specific operations, such as special routing, when executed at network nodes.

However, the way mobile code is used in active networks differs conceptually from the approach we use in service integration, although [21] describes a system that also allows the mobile terminals to extend their functionality by downloading mobile code. In our system, the approach usually used in active networks would translate roughly into a situation where embedded systems could modify, by sending

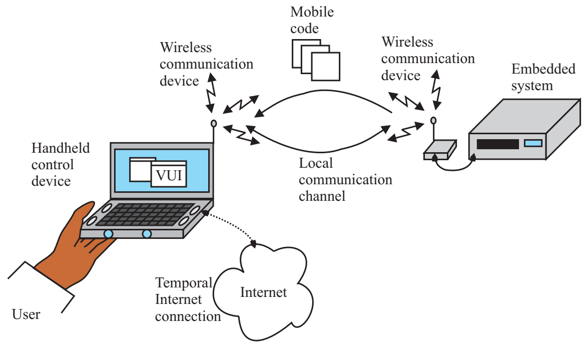

Figure 1. Components that take part in the CUES system

mobile code, the control device for their purpose. For example, the embedded system could use the control device as a gateway to the Internet by sending a mobile code that would operate as a communication relay. Although this approach is possible, we do not discuss it further in this paper.

3. CUES System

The CUES (Control for Ubiquitous Embedded Systems) system is a prototype platform that we used to implement service integration. We constructed CUES to operate as a test bench that we could use to investigate how different mobile code paradigms, code on demand, remote evaluation, and mobile agent paradigm [22], could be used in ubiquitous computing. CUES is a simple, flexible, and easily customizable platform, but it currently lacks the features and robustness required in a real-world system. In order to help the reader to understand the rest of this paper, we present a description of CUES here. Some of the aspects of CUES that are not presented have been discussed in [23].

3.1. CUES Overall Structure

Figure 1 presents the components that take part in CUES. We use a handheld (or wearable) control device to control various embedded systems, and we assume that it contains processor, memory, and display capacities that are comparable with those of current laptop computers. The device communicates with embedded systems by using a high-speed wireless communication link that has a limited range, perhaps only up to 10 meters. The control device also contains a temporary Internet connection. By ”temporary” we mean that the device cannot remain online for a very long time, perhaps because maintaining the connection is expensive or energy-consuming.

When the user wants to interact with an embedded system, the control device first downloads, by using a mobile code protocol and a local link, a mobile code from the selected system. Once executed in a corresponding execution environment, this code creates a virtual user interface (VUI) for the system. During its execution, the mobile code can use the services provided by the control device to communicate with the corresponding embedded system. This communication can follow any internal protocol that is used between the VUI and the embedded system. The reader should note that, although we currently use only one execution environment in CUES, the control device might contain several different environments that it could use with different types of mobile code. For example, some types of mobile code could contain URLs or HTML pages that would be 'executed' in a web browser.

3.2. Mobile Code Protocol

The mobile code protocol (MCP) is the most important component in the CUES system, as it enables communication between the control device and various embedded systems. Because all the devices and systems in CUES must share MCP, we tried to keep it as simple as possible, and thus MCP is a simple message-based protocol. The main principle in the design of MCP was to include into it only such functionality that could not be implemented in any other way. Simply put, if some functionality can be implemented by using internal communication between a VUI and the corresponding embedded system, it should not be included in MCP.

The services provided by MCP are:

Scan active systems; a CUES device (e.g. a control device or an embedded system) can use this service to locate other active devices within the communication range.

Query system information; a CUES device can use this service to query information from another device. The information includes id and description strings and a type number. Devices use the id and description strings mainly to describe themselves to the user, i.e. their purpose is not defined in MCP. The type number indicates what kind of mobile code the device prefers for execution. Zero value means that the device cannot (or will not) execute any mobile code. The meaning of different types of mobile code is defined in MCP.

Query mobile code information; a CUES device can use this service to ask information about a desired type of mobile code from the selected system. This information includes id and description strings, a type number, a time stamp indicating the moment of last modification, and a crc value. The actual meanings of the id and description strings may vary, depending on the corresponding execution environment. The type number indicates what kind of environment the code requires for execution. The control device uses the time stamp and the crc value to implement mobile code caching. With frequently encountered systems, such as domestic appliances, this prevents the repeated downloading of the same mobile code.

Query a mobile code; a CUES device can use this service to request a desired type of mobile code from the selected system. This data includes the same information as the mobile code information, but it also includes the executable mobile code.

Query permission to send a mobile code; a CUES device can use this service to ask for a permission before sending the mobile code. An embedded system should use this service to ask, before sending any code, the control device whether it will accept the code or not.

Create a mobile code; a CUES device uses this service to send a mobile code to another device for execution. We usually use it in conjunction with the *query permission* service described above.

Send data; a CUES device uses this service to encapsulate the communication between a VUI and the corresponding embedded system. The transmitted data may have arbitrary length and format. In this service, we use the notion of channels and channel numbers to identify the actual communication endpoints at the communicating systems. The channel number is similar to the port number in TCP/IP. When using this service, the communication will be, by default, message-based and connectionless. However, we feel that this form of communication is natural in CUES because the communication links are potentially highly volatile due the short communication range and the high mobility. We can, however, implement more complex communication systems on top of this service. For example, we could implement the Java Remote Method Invocation (RMI) [24] system by using the ideas presented in [25].

Figure 2 illustrates the Java API that hides the MCP implementation that we use in CUES.

Currently, the MCP implementation in CUES operates on top of our *ad hoc* Java-based protocol stack [23]. However, MCP requires only a protocol stack that provides services for locating other active systems and for sending reliably messages of arbitrary length. We will use a standard protocol stack to run MCP if we can find a suitable one, for example when Bluetooth is released.

3.3. Mobile Code Execution Environment

The control device in CUES uses the SMAF (Simple Mobile Agent Facility) system to execute the mobile code.

```
package fi.oulu.ee.Protocol.MCProtocol;
...
public class MCProtocol {
...
// sid = selected system.
// timeout = time after we assume the operation failed.

// Locate other active devices within the communication range.
public String[] querySystems(int timeout){...}

// Query information from another device, indicated by sid.
public SystemInfo requestInfo(String sid, int timeout) throws Exception{...}

// Ask information about a desired type of mobile code from the selected
// system. Type is indicated by mctype.
public MCInfo requestMCInfo(String sid, int mctype, int timeout) throws Exception{...}

// Request desired type of mobile code from the selected system.
public MCData requestMC(String sid, int mctype, int timeout) throws Exception{...}

// Ask for a permission before sending the mobile code.
public byte acceptMC(String sid, MCInfo mcinfo, int timeout) throws Exception{...}

// send mobile code to another device for execution.
public String createMC(String sid, MCData mcdata, int timeout) throws Exception{...}

// Send arbitrary data to selected system. cid indicates the channel.
public void sendData(String sid, int cid, byte[] data) throws Exception{...}
...
}
```

Figure 2. The Java API that hides the mobile code protocol implementation used in CUES

In the rest of this paper, we will use the term *SMAF system* to describe the execution environments within SMAF, and the term *SMAF* to describe a facility that may comprise several individual SMAF systems, as we will describe below.

SMAF is a simple mobile code test bench that we use for testing different mobile code paradigms [22]. The SMAF systems provide execution environments that offer basic services for a mobile code. These include a service that the mobile code can use to migrate from one SMAF system to another. Thus, the mobile code in SMAF is actually a mobile software agent, although we currently utilize only the code on demand and remote evaluation paradigms. In other words, the agent does not migrate from one SMAF system to another during its execution, although this is possible. We are currently investigating, however, a possibility to also use the mobile agent paradigm in CUES.

Moreover, the reader should note that the goal of SMAF is only to provide a test bench, not to compete in features or robustness with the existing mobile agent frameworks, such as IBM Aglets [26] or ObjectSpace Voyager [27]. For example, SMAF currently lacks the security features needed to shield the SMAF systems against defective or malicious agents. Although SMAF is simple, it follows roughly the guidelines outlined in the Mobile Agent System Interoperability Facility (MASIF) specification [28], formerly known as the Mobile Agent Facility (MAF) specification.

We used the Sun Java 2 platform [29], formerly known as the Java Developer Kit (JDK) version 1.2, to construct SMAF. We selected Java as the mobile code language mainly because it includes enough mobile code support to allow us to test all the mobile code paradigms with a moderate amount of work. Java is also platform-independent, and it comes with a standard class library with rich functionality.

Figure 3 outlines the components that take part in SMAF. As stated earlier, the SMAF systems operate as execution environments for the SMAF agents. The SMAF finder operates as a centralized registry that keeps track of the SMAF systems and agents, and it is used to find active systems and agents within a region. There should be only one active registry within a given region at a time. A region is a group of SMAF systems that operate, for example, under the same administrator domain. The reader should consult the MASIF specification for more information about the purpose of the finder and the region. In CUES, we run the SMAF finder in the control device. Although we do not currently use the services provided by the finder in CUES, we included it in the system for possible future use.

SMAF may also contain client programs that interact with the finder and the SMAF systems. For example, clients can create new agents, monitor active agents, and administer SMAF systems. Because we use Java in SMAF, the finder, the SMAF systems, and the clients use Java Remote Method Invocation [24] for communication.

SMAF operates as follows, Initially, the compiled Java class files used by the agent are stored in a Java Archive (JAR) file [30]. These JAR files can be used to create new agents, and they are stored in code repositories. We use the concept of code repository in SMAF to describe an arbitrary

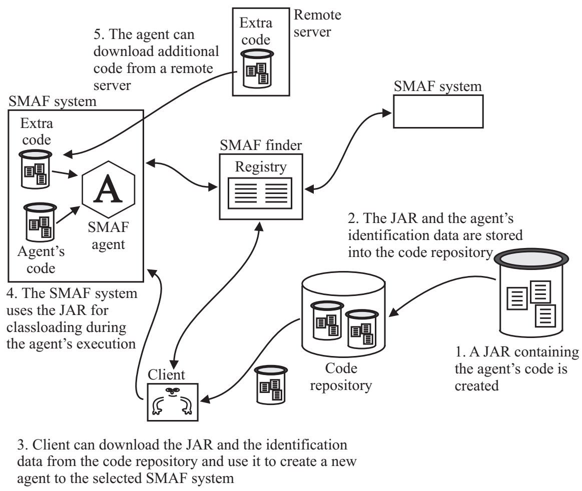

Figure 3. Components that take part in SMAF. (1) Initially, a JAR file containing the agent's code is created and (2) stored in a code repository. (3) A client can download the JAR and the agent's initialization data and use them to create a new agent in the selected SMAF system. (4) While the agent executes, the SMAF system uses the JAR file for class loading. (5) The agent can obtain parts of its classes from arbitrary sources, and thus dynamically extend itself. The SMAF finder keeps a track of the SMAF systems and agents, and it can be used to find other active systems and agents within a region

system storing the JAR files. For example, in CUES the embedded systems operate as repositories.

A new agent is instantiated by obtaining a corresponding JAR file from the repository, by converting it into a byte array, and by sending it together with the agent's initialization data to the selected SMAF system. The initialization data contains such information as the name of the bootstrap class that is the agent's execution entry point. To start agent execution, the SMAF system creates an instance of the bootstrap class and invokes its run method. For example, in CUES the control device uses the mobile code protocol to download a mobile code and its initialization data from the embedded system and sends them to the SMAF system that is running in the control device.

While the agent executes, the SMAF system uses the agent's JAR byte array for class loading. We do this by using our custom Java classloader. The agent hence does not need any external code servers for class loading during its lifetime. SMAF also allows an agent to add new classes into its JAR byte array. These new classes must also be in the form of JAR byte array. Once added, the new classes can be used by the SMAF system for subsequent class loading. This allows the agent to dynamically extend its functionality by obtaining parts of its classes from arbitrary sources during its execution. However, SMAF systems currently discard these additional classes when the agent migrates.

When the agent migrates from one SMAF system to another, part of the agent's execution state, along with the JAR byte array, is transferred. The receiver can then use this partial execution state to reconstruct some part of the complete execution state. When implementing the mobile code paradigms, SMAF utilizes the weakly mobile technology [22]. A system utilizing the weakly mobile technology does not preserve the exact execution state of the mobile code when the code is transferred from one system to another. Instead, only parts of the execution state are stored in the originating system and transferred with the mobile code to the receiving system. The receiving system uses this data to reconstruct some parts of the mobile code execution state. Part of the execution state can be reconstructed, for example, by using state variables that restore the desired execution branch at the receiving system.

The weakly mobile technology is implemented in SMAF by using Java Object Serialization [31]. The agent's state is serialized and transferred to the destination with the agent's JAR byte array. When the receiving system reconstructs the agent, the serialized state data can be used to restore only the member variable values of the object. The state of the local variables in the object's methods is lost.

Finally, in CUES, the agent can use the communication methods provided by the SMAF system to communicate with the embedded system where the control module downloaded the agent's code. The control device keeps a registry of what embedded system each agents was downloaded from and uses this information to relay the communication to the correct destination.

3.4. Hardware and Software

The CUES system currently operates in the following hardware. We use a desktop (Intel Pentium Pro 200, 96 Mbytes of RAM, and running Windows NT) to simulate the control device. This PC has a fixed Internet connection. We use a desktop computer instead of a more mobile computer, such as a laptop, because it provides a more convenient development environment. In the near future, we plan to use a wearable computer [32] as a control device in CUES.

Likewise, we currently use a desktop PC (Intel Pentium 75, 32 Mbytes of RAM, and running Windows 95) and various Java programs to simulate different embedded systems. We also simulate the printer used in this paper by using a corresponding Java program. This program relays the print requests to the actual printer attached to the PC, thus allowing us to see that the printing is successful.

We use Sequence DT10 radio modems [33] as the short-range communication devices. This modem provides both point-to-point (PP) and connectionless broadcast communication. The maximum data transfer rate is 115.2 Kbit/s in PP communication and somewhat lower in broadcast. The communication range is up to several hundreds of meters in line-of-sight conditions. Each modem is connected to the PC's serial port, and we use Java Communication API [34] to control them.

We use broadcast communication to locate other active systems and, as establishing the PP connection takes approximately 1 second, also to send small messages. The threshold between broadcast and PP is dynamically adjustable. Broadcast communication also allows several modems to transmit data to the same target at the same time, whereas in PP a new connection cannot be established if the modem is already online. This is similar to the busy line condition in normal modems.

The reason why we do not send large messages by using broadcast is performance. As we use a pure Java protocol stack below the mobile code protocol layer, the small size of the broadcast packets (up to 128 bytes) causes a large protocol overhead. The actual data transfer rate in the mobile code protocol is approximately 6 Kbytes/s in PP and 3 Kbytes/s in broadcast communication.

All the software in CUES is constructed by using the Sun Java 2 platform.

4. Print Service Integration

In this section, we will present the approach we use to integrate the service provided by a printer into the control device. We agree that a printer is not a prime example of a ubiquitous embedded system, mainly because the existing printers are already relatively complex and well-integrated into other computing systems. However, we will use it because the print service is easy to understand. Likewise, the principles presented here can be applied to other embedded systems that are more common in home-based computing. For example, a VCR could offer a service that would provide a video stream for the control device, a digital camera could offer a service that would provide still pictures for the control device, etc.

We use two different approaches for service integration, as also outlined in the Jini Architectural Specification [35]. The first approach is automatic service integration. This corresponds to the programmatic interfaces in Jini. The second approach is user assisted service integration, which corresponds to the normal user interfaces in Jini. The difference between these two approaches lies in the level of user involvement. In the first approach, integration is (almost) fully automatic. In the second approach, the user is responsible for the actual integration.

4.1. Automatic Service Integration

The automatic integration of print service operates as follows. When the control device needs print service, it automatically scans the environment to see if it contains an embedded system that can provide the service (e.g. a printer). Once the device has located such a system, it downloads from the system a mobile code that will, when executed, provide the print service interface. When the print service is invoked, the mobile code handles the communication with the embedded system. After the service is no longer needed or the control device is no longer in the vicinity of the embedded system, the mobile code is terminated and discarded.

Figure 4 presents schematically how the automatic integration of print service works. As shown, the control device contains five components relevant to this approach: a text editor (FooPad), a lookup service, a driver registry, a SMAF system, and a spooler. We also assume that a printer providing the print service is located in the immediate vicinity of the control device. In short, FooPad is the application that requests the print service from the control device. It uses the lookup service to find the driver registry. The driver registry operates as a centralized access point to the services that the device provides for applications. It also manages the available services. The control device uses the SMAF system to execute the mobile code downloaded from the printer. Print jobs can be spooled to the spooler to be completed later.

As stated above, we assume that the printer providing the print service is located in the immediate vicinity of the control device. To model our actual implementation (see Section 3), we assume that the printer itself is not a CUES-compatible embedded system. Instead, it uses a simple adapter (simulated by a PC) that implements the mo-

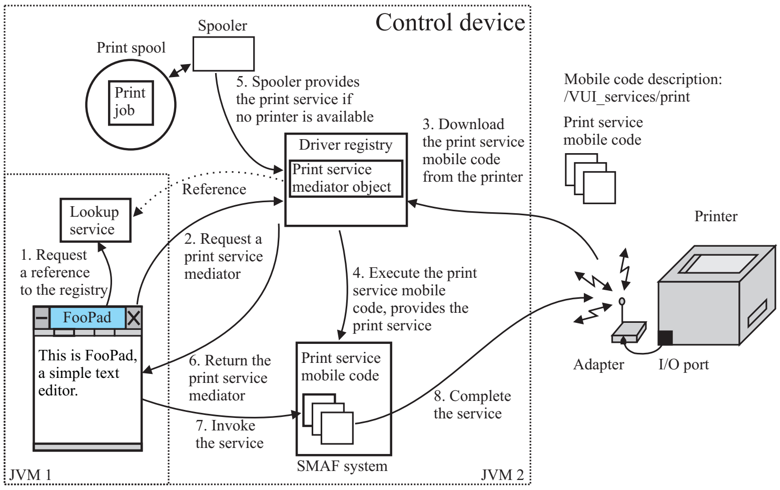

Figure 4. Automatic integration of print service. (1) FooPad requests from the lookup service a reference to the driver registry. (2) FooPad uses the reference to request a print service mediator object (SMO) from the registry. (3) The driver registry downloads the mobile code that provides a print service from the printer. (4) Once executed in the SMAF system, the mobile code provides the print service. (5) If no mobile code providing the print service can be located, a spooler can be used to complete the print job. (6) The registry binds the print SMO to the executing mobile code and returns it to FooPad. (7) FooPad uses the SMO to invoke the service. (8) The mobile code completes the service by interacting with the printer

bile code protocol and contains the required mobile code. The adapter also contains software used to interact with the printer. In this way, the printer requires only an I/O port that it can use to communicate with the adapter, which means that no drastic changes are needed in the existing printers. This adapter can be sold separately.

The FooPad text editor provides the basic functionality required for editing ASCII text files, and it can also be used to print text. FooPad is run in its own Java virtual machine (JVM), and this affected our design, as will be shown later. Although the same JVM could be used to run all the components in the control device by using the multi-processing support provided by Java [36], we selected to run FooPad in one JVM and the other components in another. We did this to be able to run conveniently several different applications that use the services provided by the control device. For example, we can now easily use Windows NT console windows to launch multiple instances of FooPad or some other application, so that each will run inside its own JVM. Had we run all the components inside one JVM, we would have needed to include into the current software some sort of application launcher utility or a shell-like command line tool to achieve similar functionality [36]. We feel that our solution is simpler in the current development stage.

When FooPad requires the print service, it first obtains a reference to the driver registry running in the control device. It does this by using the lookup service provided by the control device. Internally, the driver registry is a customized Java RMI registry [24], but the lookup service hides this from FooPad. We use RMI because it provides a convenient communication technique between different JVMs.

The driver registry is the most important component in the automatic service integration. It operates as a centralized access point to the services that the control device provides for applications. The driver registry also manages a mapping between service mediator objects (SMOs) and corresponding service providers. The structure of the driver registry is presented schematically in Figure 5.

Once FooPad has a reference to the driver registry, it can request the print SMO. Each SMO presents a capability for applications to perform the corresponding service [37]. The SMO also provides a fixed service interface, i.e. the print service has a fixed interface defined by the corresponding SMO. The print SMO interface that we currently use is shown in Figure 6. The SMOs are remote objects for FooPad, as they execute in different JVM. This means that RMI is also used for the communication between FooPad and the SMO. Currently, this involves a performance penalty, but we could eliminate much of it by running all the components in one JVM.

When FooPad requests the print SMO from the driver registry, the registry first checks if the SMO is already bound to a service provider. A service provider is usually an executing mobile code, although it can also be some other

```
package fi.oulu.ee.VUIDrivers.PrinterDrv;
...
public interface PrinterDrv extends Remote {
...
// Get printer status (currently not implemented).
public PrinterStatus getStatus() throws RemoteException;

// Print text file.
public PrintStatus printTxt(byte[] data) throws RemoteException;

// Print PostScript file.
public PrintStatus printPs(byte[] data) throws RemoteException;
...
}
```

Figure 6. The print service mediator object (SMO) interface

software component if, for example, the control device itself can provide some service. If the SMO is bound, the registry returns it to FooPad to be used to complete the service.

If the SMO is not bound to a service provider, the driver registry uses the mobile code protocol to scan the embedded systems that are located in the control device's environment. The systems that are found are selected for further investigation. The registry then queries mobile code information from each selected system. Only information of a mobile code of a predetermined type is queried. In other words, the driver registry assumes that the mobile code implementing a service or services in CUES has a predetermined type number defined by the mobile code protocol.

When the registry receives the mobile code information, it scans the description string and looks for predetermined service id strings. For example, if the mobile code provides the print service, it contains a string */VUI_services/print* in its description.

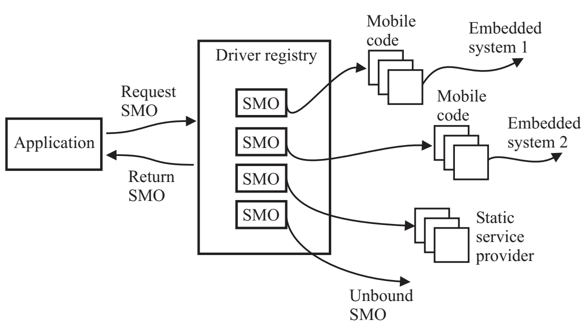

Figure 5. The structure of the driver registry. Applications use the driver registry to request service mediator objects (SMOs). The registry manages the binding between the service mediator objects (SMOs) and the service providers. The SMO can be bound to the executing mobile code or to a static service provider that is part of the control device

In case there are several embedded systems in the environment providing the print service, the driver registry selects, in a non-interactive mode, the first system. If the registry is in an interactive mode, it asks the user what system should be selected. Once the system has been selected, the registry downloads the mobile code and executes it in the SMAF system.

When the mobile code is executed, it indicates to the driver registry that it is a print service provider. The registry notices this, binds the print SMO to the mobile code, and returns the SMO to FooPad. When FooPad invokes the service, the mobile code receives the service data via the SMO. The mobile code then handles the details of how the service is actually implemented, e.g. what communication is required between the mobile code and the printer, etc. For example, the mobile code we currently use first compresses the service data and then sends it to the printer.

If the service is not requested again within a certain period, the driver registry removes it by unbinding the SMO and discarding the mobile code. However, the mobile code may remain in the control device's mobile code cache to be used later.

If the driver registry cannot find an embedded system providing the print service, it asks the user whether the print job should be spooled to be completed later. If the user refuses, an error is returned to FooPad. If the user agrees to spool the print job, the registry binds the print SMO to the spooler and returns the SMO. When FooPad uses the SMO, the service data is directed to the spooler. The spooler periodically checks if the print service is available in the environment, and if it is, asks the user whether the spooled job should be completed.

The size of the mobile code JAR implementing the print service is 11 Kbytes.

4.2. User Assisted Service Integration

The user-assisted print service integration operates as follows. When the user is in the vicinity of a printer and needs print service, he or she selects the printer in the con-

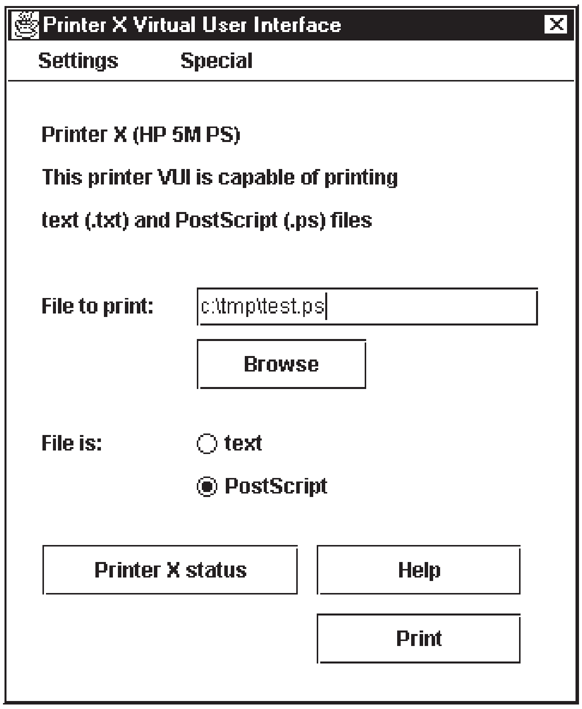

Figure 7. The virtual user interface for the printer

trol device's list of active systems. The control device then downloads a mobile code from the printer. This code, however, has a type number different from the one used in automatic service integration. Once executed, this mobile code implements a virtual user interface (VUI) for the printer, as presented in Figure 7. The user can use the VUI to interact with the printer. For example, the user can print a selected text or PostScript file that is stored in the control device. The size of the mobile code for the VUI is 28 Kbytes.

When the user wants to print a document by using the VUI, he or she must first use FooPad to print the document into a file, and then select this file to the VUI for printing. It should be noted, though, that when the mobile code implementing the VUI is executed, it also registers itself to the driver registry as a print service provider. In other words, although the VUI can provide the same functionality as the mobile code used in automatic service integration, it can also implement additional features. For example, the VUI could implement *a print on both sides* option even if the print SMO interface did not support that. By using both automatic and user-assisted service integration, the most commonly used functionality could be made available automatically, although there would be a backdoor that could be used when additional features are needed.

5. Discussion and Future Work

Our approach for service integration illustrates several promising aspects. Firstly, the service structure in the control device is highly dynamic. For example, as the user moves around, the control device can use the services provided by the embedded systems located in his or her immediate surroundings. We feel that this naturally maps to the characteristics of ubiquitous computing.

Secondly, embedded systems can remain relatively simple although the service interfaces they implement are complex. This is especially true if we use an adapter to interface with the actual embedded system, as described in Section 4. Thus, no drastic modifications are required for the existing embedded systems. This level of simplicity is achieved because the mobile code, the embedded system's primary interface to the outside world, is not itself executed in the embedded system. For example, the mobile code may provide a CORBA or Java RMI interface for the embedded system, although the embedded system does not execute any CORBA-related software or include a Java virtual machine. In addition, while we can program the embedded system by using low-level programming languages, such as C or Assembler, we can simultaneously program the service interface by using high-level programming languages, such as Java. This is especially helpful when the embedded system implements complex interfaces.

Thirdly, as the embedded system does not itself execute the mobile code, the code is relatively simple to update. As we described in [38], this can be done automatically by including a special mobile code providing the update service in the embedded system that, once executed, will perform the update. Furthermore, including additional service interfaces in the embedded system is merely a matter of including a new mobile code, and this can be easily done even if the system has already been released. For the embedded system, this requires only a small amount of additional non-volatile memory. In addition, some of the service interfaces can be implemented so that the embedded system itself only contains a small bootstrap code. Once executed by the control device, the bootstrap code downloads the rest of the code from an external server [38]. For example, the SMAF system provides for this purpose a service that allows the mobile code to dynamically extend itself by downloading additional code from arbitrary sources.

Finally, we feel that because the embedded system can contain several different service interfaces that may be written in different mobile code languages, in addition to the fact that these interfaces are easy to update, it is easier for the embedded system manufacturers to agree on standard service interfaces. Currently, it may be risky to settle on a standard interface too early, as it may soon become obsolete. Furthermore, service interfaces can be partly independent of the communication characteristics of the systems where they are used. We can construct the mobile code in such a way that it will handle the communication details to suit each particular system.

However, our approach also has disadvantages. The most obvious disadvantage is the problem with information security. The security requirements for a mobile code, when combined with mobile and ubiquitous computing, are still very much a research issue. This is the reason why we decided to ignore security in our current work. We agree that information security is probably the most essential problem that needs to be solved before ubiquitous computing can become a reality. We feel, however, that mobile code *per se* does not cause unsolvable problems, as the way in which we use mobile code in our approach is not so much different from the way Java applets operate. Another obvious problem is the communication overhead. Downloading a mobile code from the embedded system every time the control device needs a service causes a communication overhead. We can reduce some of this overhead by caching the most frequently used mobile code in the control device. This is especially useful when interacting with domestic or other frequently used appliances. However, caching is probably not very useful when the user roams in different environments, as he or she will constantly encounter previously unknown embedded systems.

The reader should note that we constructed both the service integration system and the underlying CUES system only to try out how we could use mobile code in ubiquitous computing. During our work, our goal was primarily to demonstrate what is possible and only secundarily to think how we could implement this possibility in a way that could be used in real-world systems.

Because of this, our future work will concentrate on refining the ideas we have obtained from this work. In addition, we will also concentrate on the information security aspects of ubiquitous computing in general.

One possible future work could also concentrate on investigating the possible techniques to eliminate the need for fixed service interfaces. Although we used fixed interfaces in this work, we feel that they are not an optimal solution for ubiquitous computing. Our virtual user interfaces can be used to control (almost) every possible embedded system mainly because they are used by humans, who can easily adapt to different user interfaces. However, when two or more mutually unrelated programs cooperate, the interface between them must be fixed, or some form of facilitator must match the provided and requested services. Interestingly, this problem is similar to the problem encountered in software agent interoperation [39].

It should be noted, however, that our approach offers an interesting technique to be used with the facilitator. A special mobile code (i.e. an information object) in addition to the normal service code could be downloaded from the embedded system. This special mobile code could contain information in a form that could be used by the facilitator to match capabilities to needs. The facilitator could also be located in a server available on the Internet. In such a case, the control device could send the information about the requested and provided services to the server and the facilitator could return the mapping to the device.

6. Conclusion

The goal of this paper was to demonstrate that mobile code helps to implement service integration between ubiquitous embedded systems and the devices used to control them. To achieve this goal, we presented our mobile code approach to service integration in ubiquitous computing. We used a single control device to interact with diverse embedded systems. We demonstrated, by using print service as an example, how the user of such a control device can conveniently use the services provided by the embedded systems located in the device's environment. We described two different ways, automatic and user-assisted, to implement service integration. These two ways can be used to complement each other.

The main advantage in our approach is that it allows the service structure in the control device to be highly dynamic. It also makes it possible to implement rich service interfaces in a way that allows embedded systems to remain relatively simple. Furthermore, one system may also contain several different service interfaces that are written in different mobile code languages and are easy to update.

In our current work, however, we ignored some of the problems, such as information security. Because of this, some of the implementation details should be redesigned if our work is to be applied in real-world systems.

In a wider sense, we hope that our work illustrates the possibilities that mobile code can provide for ubiquitous computing. We feel that, as computer systems are becoming extremely complex, widespread, and intertwined, there is a need for flexibility, adaptivity, and dynamicity. Mobile code provides all these features.

7. Acknowledgements

We would like to thank Elektrobit Ltd. (www.elektrobit.fi) for providing us the Sequence DT10 radio modems for our experimental system. We would also like to thank the anonymous reviewers for their comments.

References

[1] M Weiser. The computer for the 21st century. *Scientific American*, pages 66–75, September 1991.

[2] A Fuggetta, G P Picco, and G Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, May 1998.

[3] J Haartsen, M Naghshineh, J Inouye, O J Joeressen, and W Allen. Bluetooth: Vision, goals, and architecture. *Mobile Computing and Communications Review*, 2(4):38–45, 1998.

[4] Nokia. *Nokia 9110 Communicator*. http://www.nokia.com/products/phones/9110/

[5] P Pulli, T Pyssysalo, K Kuutti, J Similä, J-P Metsävainio, and O Komulainen. Cyphone—mobile multimodal personal augmented reality. In *Proc. of the 2nd Int. Conf. on Worldwide Computing and Its Applications (WWCA'98)*, pages 325–336, Tsukuba, Japan, March 4-5 1998.

[6] MIT Media Lab. *HIVE*. http://hive.www.media.mit.edu/projects/hive/

[7] Object Management Group (OMG). *The Common Object Request Broker: Architecture and Specification, version 2.2*. ftp://ftp.omg.org/pub/docs/formal/98-07-01.pdf

[8] Home API Working Group. *Home API*. http://www.homeapi.org/homeapi/whitepaper/

[9] S M Dorward, R Pike, D L Presotto, D M Ritchie, H W Tickey, and P Winterbottom. The inferno operating system. *Bell Labs Technical Journal*, 2(1):5–18, 1997.

[10] R Ierusalimschy, R Cerqueira, and N Rodriguez. Using reflexivity to interface with corba. In *Proc. of the 1998 Int. Conf. on Computer Languages (ICCL'98)*, Chicago, IL, USA, May 14-16 1998.

[11] Universal Plug and Play Forum. *Universal Plug and Play (UPnP)*. http://www.upnp.org/

[12] Home Audio/Video Interoperability. *HAVi*. http://www.havi.org/home.html

[13] Sun Microsystems, Inc. *Jini*. http://www.sun.com/jini/

[14] Sun Microsystems, Inc. *Java Embedded Server*. http://www.sun.com/software/embeddedserver/

[15] R Itschner, C Pommerell, and M Rutishauser. Glass: Remote monitoring of embedded systems in power engineering. *IEEE Internet Computing*, pages 46–52, May-June 1998.

[16] G Kortuem, Z Segall, and M Bauer. Context-aware, adaptive wearable computers as remote interfaces to intelligent environments. In *Proc. of the 2nd Int. Symp. on Wearable Computers (ISWC'98)*, Pittsburgh, Pennsylvania, USA, October 19-20 1998.

[17] T D Hodes, R H Katz, E Servan-Schreiber, and L Rowe. Composable ad-hoc mobile services for universal interaction. In *Proc. of the 3rd annual ACM/IEEE Int. Conf. on Mobile Computing and Networking (MOBICOM'97)*, pages 1–12, Budapest, Hungary, September 26-30 1997.

[18] A K Dey, G D Abowd, and A Wood. Cyberdesk: A framework for providing self-integrating context-aware services. In *Proc. of the 1998 Int. Conf. on Intelligent User Interfaces (IUI'98)*, pages 47–54, San Francisco, California, USA, January 6-9 1998.

[19] D L Tennenhouse and D J Wetherall. Towards an active network architecture. *Computer Communication Review*, 26(2):5–18, April 1996.

[20] J M Smith, K L Calvert, S L Murphy, H K Orman, and L L Peterson. Activating networks: A progress report. *Computer*, pages 32–41, April 1999.

[21] A Boulis, P Lettieri, and M B Srivastava. Active basestations & nodes for a mobile environment. In *Proc. of the 1st ACM Int. Workshop on Wireless Mobile Multimedia (WOWMON'98)*, pages 31–37, Dallas, Texas, USA, October 30 1998.

[22] A Carzaniga, G P Picco, and G Vigna. Designing distributed applications with mobile code paradigms. In *Proc. of the 1997 Int. Conf. on Software Engineering (ICSE'97)*, pages 22–32, Boston, Massachusetts, USA, May 17-23 1997.

[23] K Kangas and J Röning. Using mobile code interfaces to control ubiquitous embedded systems. In *Proc. of the USENIX Workshop on Embedded Systems (ES'99)*, Cambridge, Massachusetts, USA, March 29-31 1999.

[24] Sun Microsystems, Inc. *Java Remote Method Invocation (RMI)*. http://www.javasoft.com/products/jdk/rmi/

[25] V Krishnaswamy, D Walter, S Bhola, E Bommaiah, G Riley, B Topol, and M Ahamad. Efficient implementation of java remote method invocation (rmi). In *Proc. of the 4th USENIX Conf. on Object-Oriented Technologies and Systems (COOTS'98)*, Santa Fe, New Mexico, USA, April 27-30 1998.

[26] IBM Corp. *Aglets Software Development Kit*. http://www.trl.ibm.co.jp/aglets/

[27] ObjectSpace, Inc. *Voyager*. http://www.objectspace.com/developers/voyager/

[28] Object Management Group (OMG). *Mobile Agents Revision (MASIF Revision)*, July 28 1998. ftp://ftp.omg.org/pub/docs/orbos/98-03-09.pdf

[29] Sun Microsystems, Inc. *Java 2 Platform*. http://www.javasoft.com/products/jdk/1.2/

[30] Sun Microsystems, Inc. *Java Archive (JAR) File*. http:// www.javasoft.com/products/jdk/1.2/docs/guide/jar/

[31] Sun Microsystems, Inc. *Java Object Serialization*. http://www.javasoft.com/products/jdk/1.2/docs/guide/serialization/

[32] T Pyssysalo and T Turunen. A wearable terminal for mobile augmented reality based cscw applications. Submitted for publication.

[33] Elektrobit, Ltd. *Sequence DT10 radio modem*. http://www.elektrobit.fi/products/sequence.html

[34] Sun Microsystems, Inc. *Java Communications API*. http://www.javasoft.com/products/javacomm/

[35] Sun Microsystems, Inc. *Jini Architecture Specification*. http://www.sun.com/jini/specs/jini-spec.pdf

[36] D Balfanz and L Gong. Experience with secure multi-processing in java. In *Proc. of the 18th Int. Conf. on Distributed Computing Systems (ICDCS'98)*, Amsterdam, The Netherlands, May 26-29 1998.

[37] C Hawblitzel, C-C Chang, G Czajkowski, D Hu, and T von Eicken. Implementing multiple protection domains in java. In *Proc. of the USENIX Annual Tech. Conference*, New Orleans, Louisiana, USA, June 1998.

[38] K Kangas and J Röning. Using mobile code and the internet to implement value-added services for ubiquitous embedded systems. In *Proc. of the 12th ISCA Int. Conf. on Parallel and Distributed Computing Systems (PDCS'99)*, Fort Lauderdale, Florida, USA, August 17-20 1999. Accepted.

[39] M R Genesereth and S P Ketchpel. Software agents. *Communications of the ACM*, 37(7):48–53, July 1994.