Using a Secure Mobile Object Kernel as Operating System on Embedded Devices to Support the Dynamic Upload of Applications

Walter Binder and Balázs Lichtl

CoCo Software Engineering GmbH
Margaretenstr. 22/9, 1040 Vienna, Austria
{w.binder | b.lichtl}@cocosoftware.com

Abstract. In this paper we present the architecture of an autonomous, multi-purpose station which securely executes dynamically uploaded applications. The station hardware is based on an embedded Java processor running the system software and applications. The system software is built on top of a flexible, lightweight, efficient, and secure mobile object kernel, which is able to receive mobile code and to execute it, while protecting the station from faulty applications. Mobile code is used for application upload, as well as for remote configuration and maintenance. The autonomous station relies on resource accounting and control in order to prevent an overuse of its computing resources. Moreover, applications executing on the station may be charged for their resource consumption. This paper also describes an initial application of the autonomous station, which has been recently deployed in a pilot project: Based on the architecture of the autonomous station, we have designed and implemented an on-demand bus stop.

1 Introduction

This paper gives an overview of the design and architecture of an autonomous station, which is able to securely and reliably execute dynamically uploaded applications. The autonomous station does not rely on an external power supply system, but it comprises a unit for the generation of current in order to ensure its autonomy. It is equipped with application-dependent sensors and actuators, and it may be deployed in inaccessible environments. It offers its specific equipment to applications in a time-sharing fashion. The applications are not hard-coded in the station, but they are dynamically uploaded on demand. They are charged for their utilization of the resources provided by the autonomous station. In order to support application upload, transmission of results, and remote system maintenance, the autonomous station is connected to a public or private wireless network. The hardware of the station is based on an embedded Java processor running our system software, which is implemented in pure Java. The system software rests upon a lightweight, efficient, and secure mobile object platform, which is able to receive mobile code and to execute it, while protecting the autonomous station from malicious or badly programmed applications.

N. Suri (Ed.): MA 2002, LNCS 2535, pp. 154–170, 2002.

In the past years numerous research works have focussed on mobile object technology. This has resulted in a better understanding of the problems inherent to mobile objects and mobile code in general, which have to be solved in order to enable the widespread deployment of mobile code based solutions in various commercial settings, such as the embedding of a Java-based mobile object system within a proprietary hardware environment. In particular, significant research has concentrated on building secure environments for the execution of foreign, potentially malicious mobile code, which may seriously damage the execution environment itself or other components in the system [2,8,7,3].

Actually, Java has become the de facto standard implementation language for mobile object platforms due to its network centric approach, its runtime virtual machine, and features that ease the development of mobile object systems, such as a portable code format (Java bytecode) [11], dynamic and customizable class-loading, multi-threading, built-in support for object serialization, and language safety. Since the adoption of the real-time specification for Java [6], embedded Java processors conforming with this specifications have been developed, which are able to meet certain (soft) real-time requirements.

Despite of these advantages, Java has not been designed for multi-tasking[1]. Currently, Java offers no support for isolating mobile objects from each other. The lack of a task model in Java also makes it difficult to terminate applications in order to reclaim their allocated resources. Furthermore, Java has no support for resource accounting and control, which makes it vulnerable to denial-of-service (DoS) attacks and prevents the deployment of applications that shall be charged for their resource consumption. See [5] for an in-depth discussion of deficiencies of Java with respect to mobile code environments. Consequently, many recent research works have aimed at working around these shortcomings of Java related to mobile code. Different approaches have put restrictions on the programming model, used bytecode arbitration to control the execution of mobile code, and developed special runtime systems with enhanced functionalities usually found in operating systems.

In the meantime, Java-based mobile object systems are available that are secure and reliable enough to justify their deployment in commercial settings. As mobile object environments are perfectly suited for software distribution, installation, and remote maintenance, the system software of our autonomous station is based on J-SEAL2 [3], a lightweight Java-based micro-kernel, which makes Java safe for the execution of untrusted mobile code. J-SEAL2 offers a hierarchical task model allowing to isolate applications, to terminate them in a safe way, and to monitor and limit their resource consumption.

Our system software has control over a special hardware, the embedded Java processor and its peripherals (i.e., different sensors and actuators which are connected to the main station). We use a processor that natively executes Java bytecode programs, which supersedes a Java Virtual Machine (JVM) [11] implemented in software, as well as an underlying operating system layer. Therefore,

[1] In this paper the term 'task' refers to the concept of a *process* in an operating system, which allows to separate and isolate applications from each other.

the executive overhead is significantly reduced when compared to a JVM implemented in software. Furthermore, the rather low clock rates of current Java processors (about 40–400MHz) help to preserve power, which is crucial if the station has a limited power supply. Moreover, because all system components and applications are implemented within a Java-based, high-level, object-oriented programming model, the reliability of the overall system is greatly improved.

This paper is structured as follows: In the next section we discuss possible applications of the autonomous station and present our requirements and design goals. In section 3 we explain how the station is managed. We focus on application upload and on communication of application data. Section 4 outlines our business model and presents different options for charging applications executing on the station. In section 5 we explain why we selected the J-SEAL2 mobile object platform as operating system kernel for the autonomous station. Section 6 addresses technical issues regarding resource management on embedded Java systems. Using a special benchmark suite tailored to embedded devices we evaluate the overhead of CPU accounting based on program transformation techniques. Section 7 outlines an application of the autonomous station, which has been deployed in a pilot project. The last section concludes this paper.

2 Applications and Design Goals

The autonomous station is a multi-purpose, customizable, and extensible system, which may be deployed in many different configurations. Below we mention a few exemplary applications. Equipped with the necessary sensors and actuators, the same autonomous station may serve a series of applications at the same time.

- In a pilot project we have used autonomous stations to provide on-demand bus stops, where the bus only passes the stop, if a customer has explicitly ordered it. In section 7 we present some details of this particular application.
- Autonomous stations can be used for traffic monitoring and for tracing. Equipped with a radar unit and a high-speed digital camera, they are a cost-effective alternative to traditional radar units operated by the police, because they are maintained remotely. The uploaded monitoring application may employ techniques for pattern recognition to extract and transmit only the relevant portion of the image, thus saving communication costs.
- The general architecture of our autonomous station is also well-suited for cheap multi-purpose satellites, as well as for spacecrafts.
- In the context of industrial sensing autonomous stations offer a cost-effective alternative to a system of wired sensing elements.
- Autonomous stations may be deployed to monitor their environment. Equipped with sensors to detect toxic substances, autonomous stations can improve civil protection.
- Research institutions may use autonomous stations to collect environmental information.

Research on distributed wireless sensor networks [12] has been addressing some of these application domains, especially environmental monitoring. In sensor networks communication bandwidth and energy usually are limited, while computing power is comparatively plentiful and inexpensive. Due to bandwidth constraints and because communication over a wireless network consumes considerable energy, in-network processing, such as localized data aggregation, is essential to optimize the exploitation of resources [9]. Our autonomous station suffers from similar resource constraints, thus, driven by these limitations, we have developed a generic environment to move application code as close as possible to where the data is collected. Whereas many recent works on wireless sensor networks focus on routing protocols and on data distribution, a very simple communication architecture is sufficient for our purpose (for details see section 3). Our contribution is an extensible, multi-purpose, embedded system, where different applications are installed remotely on demand and execute in a controlled and secure way.

The hardware of the autonomous station comprises a main board with CPU and memory, a wireless communication module, a power supply system that typically consists of solar cells and a rechargeable battery, as well as application-specific sensors and actuators (input and output devices). Because of the wireless communication module and the integrated power supply, the operator may easily displace stations to new sites without incurring high installation costs. Depending on the concrete application, the hardware components have to meet the following requirements:

- The autonomous station has to be built from off-the-shelf components, in order to keep the hardware costs low.
- The hardware has to be resistant against variations in temperature.
- The power supply system must be adaptable to the concrete operational area of the station. The size of the solar cells and the capacity of the rechargeable battery have to be selected according to the expected insolation.
- Because of the limited power supply, the processor has to offer competitive performance as well as reduced power consumption. Since we require a safe high-level language for application programming, the autonomous station is based on a modern Java processor, which provides a standard JVM implemented in hardware.
- Depending on the location it may be necessary to protect the hardware against vandalism.

The design decision to employ a Java processor also implies that all system software, as well as the applications, have to be represented by JVM bytecode. The software may be implemented in pure Java or in any other language that can be compiled to JVM bytecode. For the system software, we have the following requirements:

- Because the station may not be easily accessible after deployment, and in order to reduce the maintenance overhead, the system is designed for remote

maintenance. This means that diagnostic programs may be uploaded in order to detect the reason for a malfunction. Furthermore, new system components may be installed remotely, and existing components may be replaced with new versions.
- Applications are installed and updated remotely. Applications may be terminated, freeing their allocated resources and leaving the station in a consistent state.
- Applications are charged for their resource consumption. The resources that may be charged for include power consumption, CPU and memory utilization, access to sensors and actuators, and communication.
- All system components are protected from faulty applications. Sensors and actuators cannot be directly accessed and programmed by an application, but a *device driver* (a system component) mediates access to the device.
- The autonomous station may offer its resources to multiple applications in a time-sharing fashion. Applications are isolated from each other, since they may execute on behalf of different parties.
- A *device manager* ensures that concurrent applications use multiple sensors and actuators in a consistent way. The device manager must support preemption and revocation, if a high-priority application requires access to a device occupied by a low-priority application.

3 Communication

In this section we give an overview of the communication model and infrastructure of the autonomous station, which is being used for application upload and for communication with running applications.

3.1 Supervising Server

Each autonomous station is managed and controlled by a single *supervising server* (SuSe). A SuSe may be in charge of multiple stations. The SuSe is the only communication partner of an autonomous station[2]. Clients who want to upload applications to an autonomous station or to communicate with an already uploaded application have to contact the station's SuSe, which acts as a gateway: It receives client requests from the wired network (e.g., through a TCP/IP connection) and dispatches the request to the corresponding station,

[2] In order to prevent the SuSe from becoming a single point of failure, the autonomous station may communicate with a set of backup SuSes, if its primary SuSe fails. The physical replication of a SuSe is crucial for applications with (soft) real-time guarantees, such as applications for civil protection, industrial sensing, or military purpose. Within this paper we do not elaborate issues concerning the replication of SuSes.

which is accessible only through a wireless network[3]. Vice versa, the SuSe receives messages from an application running on a station and forwards them to the client who has deployed the application.

This approach helps to protect the autonomous station, as all messages to the station are mediated by the SuSe. For instance, if an application is to be uploaded, the SuSe inspects the application code to ensure certain security properties. Moreover, this model simplifies the management and maintenance of the station, because the SuSe is its single authority, which also facilitates the charging of applications. In addition to this, the communication module within the autonomous station is significantly simplified by the fact that there is only a single communication partner. The autonomous station and its SuSe employ symmetric key encryption to protect and to authenticate the messages communicated over the wireless network. The symmetric key is a shared secret between the station and its SuSe[4]. Because cryptography based on symmetric keys can be implemented much more efficiently than public key cryptography, this approach saves processing on the autonomous station and, hence, helps to preserve power. Furthermore, there is no need for the station to utilize any public key infrastructure. Considering the limited CPU, memory, and power resources available to the station, the system software has to be kept simple, small, and efficient.

3.2 Directory and Registration

Each SuSe provides a directory of the autonomous stations it manages. Within the namespace of a SuSe, each station has its own unique identifier. For each autonomous station, the directory comprises the services offered by the station, its location (GIS coordinates), a specification of the APIs needed to program the special sensors and actuators attached to the station, a description of the different quality-of-service (QoS) levels supported by the station, and detailed pricing information. The directory of a SuSe may be publicly accessible, or it may be restricted to registered clients only.

Before a client can upload its first application to a station maintained by a particular SuSe, he has to register at the SuSe. The client has to post information necessary for billing (e.g., billing address, credit-card details, etc.), as well as his public key, which will be used by the SuSe to authenticate the client's requests (such as requests for application upload). If desired by the client, the public key may also be used to encrypt application results, which are forwarded by the SuSe to the client. The SuSe verifies the client's public key as well as the given billing information. Upon successful validation, the client will be allowed to upload applications to autonomous stations.

[3] The wireless network used to connect the autonomous station with its SuSe depends on the physical location of the station and the range of applications it shall support. For instance, in the configuration presented in section 7 the autonomous station is connected to a public GPRS (General Packet Radio Service) network [1].

[4] To protect against brute force attacks aimed at cracking the symmetric key, the autonomous station and its SuSe isochronously change the common key.

3.3 Application Upload

When a client wants to upload a new application to an autonomous station, he has to transmit the application to the station's SuSe. For this purpose, the client sends a signed message to the SuSe, including the identifiers of the destination stations, a Java archive (a JAR file) containing the application classes and a deployment descriptor, as well as the network location (e.g., host address, port, and protocol) where application results shall be routed to. The deployment descriptor specifies the resource requirements of the application, QoS parameters, etc.

The SuSe checks whether the requested QoS can be guaranteed. If the new application is accepted, it is assigned a unique *application identifier* (AID) within the SuSe's namespace. The application archive is opened, the application classes are verified (and eventually modified to guarantee certain security properties, such as resource accounting and control of the application [4]), and the application is re-packaged in a special application transfer format, which may yield better compression (an important aspect regarding the low bandwidth of many wireless networks) and which can be handled by the mobile object system executing within the autonomous station. We are relying on compression algorithms that are especially tailored to Java class files [10] and achieve significantly better compression than commonly used methods such as ZIP. The AID is part of the transfer format. The re-packaged application is transmitted to the destination stations through a wireless network. As mentioned before, symmetric key encryption is used to secure the communication. All messages originating from the application will be tagged with the AID, allowing the SuSe to dispatch them to the client. Upon succesful installation of the application, the AID is returned to the client, who can use it to direct messages to the application.

When an existing application is to be updated, the client has to provide the identifier of the application to be replaced. When the autonomous station receives an application which is already running (according to the AID), the old version is terminated before the new one is installed[5].

3.4 Communication of Application Data

Once installed, an application may transmit results to the client who has deployed the application. The communication module within the autonomous station tags the message with the correct AID. It is also responsible for buffering messages that cannot be transmitted immediately due to a network failure or crash of the SuSe. As with application upload, communication messages are encrypted with a symmetric key known only to the autonomous station and its SuSe.

[5] This simple mechanism for application update may not be suited for soft real-time applications that require a high QoS. A more sophisticated protocol for application update may allow both versions of the application to execute concurrently for a short period of time, allowing the new version to take over the functions of the old one, which will be terminated in a coordinated way.

The SuSe decrypts messages received from an autonomous station. Based on the AID, the SuSe is able to determine the recipient. If desired by the client, the SuSe signs the message and encrypts it with the receiver's public key. Finally, the message is delivered to the network address, which the client has provided during installation of the application. If the receiver is temporarily not available, the SuSe buffers the message.

The client may also send control messages to its application. For this purpose, he has to transmit the signed message to the SuSe, providing the destination AID, which the SuSe needs to route the message to the correct station. Within the autonomous station, the communication module dispatches the message to the corresponding application based on the AID.

3.5 Example

Figure 1 illustrates the deployment of an application on autonomous stations. Encryption details are not shown in this figure. In this example the SuSe controls the autonomous stations *AS 1* and *AS 2*. First, the client wishing to upload an application registers at the SuSe (1). Then he transmits a package containing application *App A* to the SuSe (2), which verifies and eventually rewrites the classes (3), before the application is uploaded to the autonomous stations (4). In this example, the client requests to upload the application to both stations. *AS 2* is already executing another application *App X*, while *AS 1* is idle before *App A* is installed. After *App A* has been installed and has started executing on *AS 2* (5), it transmits application results to its SuSe (6), which dispatches (7) and forwards (8) the data to the client owning the application. Finally, the client is able to process the application results (9).

4 Charging for Resource Consumption

In order to amortize the investment in autonomous stations and to make money, clients may be charged for applications they have deployed. The total charge for running an application on a station may comprise the following costs:

- A flat rate for application deployment.
- A general fee for using the station, which may be charged on a month-by-month basis. This charge will depend on the QoS (or priority) granted to the application.
- Variable costs depending on the resources consumed by the application, such as power, CPU time, and memory. For many applications, power will be the most precious resource, as it is limited by the available insolation and the capacity of the rechargeable battery. The power consumption may be roughly determined from measurements (i.e., voltage metering), as well as from the utilization of other resources, such as the number of executed JVM instructions (i.e., CPU consumption), access to the communication module, to sensors, and to actuators (which usually consumes extra power), etc.

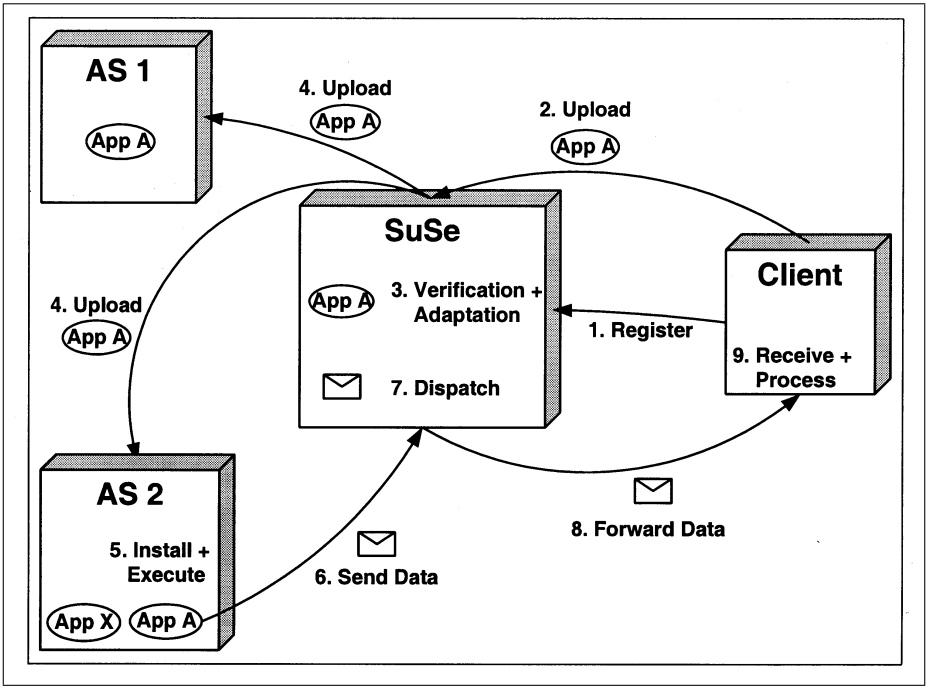

Fig. 1. Application upload and communication.

- Communication costs for messages originating from the application or sent to the application. This charge will depend on the costs of the underlying wireless network.
- An extra fee for application update.

The general fee and the variable costs for consumed resources help to amortize the investments in the station (hardware costs and license fees for the system software). Essentially, these investments are to be made by the station's owner before applications can be deployed. Nevertheless, there are also some non-negligible costs for maintenance: Apart from upgrades of the system software, which can be done remotely in a similar way as application upload, costs for repairs, for cleaning and calibration of sensors, etc. have to be considered. Furthermore, because the lifetime of the rechargeable battery is limited, it has to be replaced occasionally. The resource consumption of an application is monitored and accumulated by the system software within the autonomous station. Once it exceeds a given threshold, the information on consumed resources is communicated to the station's SuSe.

Communication causes additional costs for the owner of the autonomous station. Depending on the low-level transfer protocol offered by the wireless network and the communication frequency, these costs may become very high. Therefore,

it is important that the SuSe, which is involved in every message transfer over the wireless network, logs every communication on persistent storage. Thus, the charging for communication will be always accurate (except for lost messages sent by the autonomous station).

5 Adaption of the J-SEAL2 Mobile Object Kernel

The system software of the autonomous station is based on J-SEAL2[6] [3], a secure mobile object kernel. J-SEAL2 is a micro-kernel that offers a task model with strong isolation properties on top of standard Java runtime systems. It is based on the formal model of the Seal Calculus [14], which was first implemented by the JavaSeal kernel [7]. J-SEAL2 is able to securely and concurrently execute multiple Java applications in the same JVM, which are completely separated from each other. J-SEAL2 resembles the kernel of a traditional operating system, as it provides mechanisms for application isolation, efficient and mediated communication, safe termination, and resource control.

The architecture of J-SEAL2 is well suited as the basis for mobile code systems, because it offers the necessary level of host security, which is not found in current standard Java runtime systems: Executing applications and system services within separate tasks, J-SEAL2 protects the platform from malicious or badly programmed applications, as well as applications from each other. We are exploiting the advanced security mechanisms of J-SEAL2 to protect the autonomous station from faulty applications, to isolate applications from each other, and to account and control their resource consumption, which is necessary for charging. We selected J-SEAL2, because it offers several advantages with respect to our requirements:

- J-SEAL2 has been specially designed for increased host security. It provides a hierarchical task model, which allows to isolate system components (such as the communication module or device drivers) and applications, while they are executing within the same JVM. In the task model of J-SEAL2 a parent task acts as communication controller, access controller, and resource manager of its children. Applications that are uploaded to the autonomous station are installed as children of a trusted mediator task, which controls the execution of the applications and limits their resource consumption.
- J-SEAL2 is implemented in pure Java. In contrast to other secure mobile object platforms and Java operating systems, such as KaffeOS [2], J-SEAL2 relies neither on a special JVM nor on native code. This is of paramount importance, as the platform has to run on a Java processor, which provides a standard JVM implemented in hardware.
- J-SEAL2 is a small and efficient micro-kernel. The kernel offers only essential primitives to implement secure and reliable system software. This is important, since the memory available on the autonomous station is limited.

[6] `http://www.jseal2.com/`

- J-SEAL2 has a modular and extensible architecture. Special system services, such as device drivers, can be added dynamically.
- J-SEAL2 supports resource control for physical resources (i.e., CPU and memory), for logical resources (e.g., threads), and for access to service components. For CPU and memory control, the resource consumption of the application is reified [4,13]. I.e., the application is modified to expose its resource consumption to the system. This approach enables resource control, even if the underlying Java runtime system does not support it. In our setting the application is modified for resource control by the SuSe before it is uploaded to the station. As these modifications are complex and time consuming, they should not be carried out by the station, where the resources are limited.

System components (e.g., device drivers, device manager, communication module, etc.) as well as applications execute within separate tasks. Each task may be terminated at any time, which frees its allocated resources and is guaranteed to leave the system in a consistent state[7]. Consequently, we are able to implement application upload and update, installation of new system services, and update of system services in a similar way.

6 Resource Management on Embedded Java Systems

In order to support resource accounting and control on the autonomous station, the SuSe rewrites applications to keep track of the number of executed bytecode instructions (CPU accounting) and to update a memory account when objects are allocated or reclaimed by the garbage collector. Ideally, program transformations for resource management shall be compatible with existing Java runtime systems, shall cause only moderate overhead, and allow accurate accounting. While the accuracy of CPU accounting on a Java 2 Standard Edition JVM is limited, because of the significant part of native code that is not accounted for and due to optimizations performed by the compiler, the accounting precision on a Java processor can be much better, as the execution time of individual bytecode instructions can be measured and only very simple and well documented optimizations are performed, such as the combination of certain JVM instructions (instruction folding). However, regarding the overhead sophisticated optimizations can be beneficial, and consequently the relative overhead on an embedded Java processor may be significantly higher than on a JVM with a modern compiler, where the overhead for CPU accounting is about 15–30% [4, 13].

In order to evaluate the overhead caused by the rewriting of application and of JDK classes on an embedded Java processor, we created a special benchmark

[7] If the task to be terminated is executing a kernel operation, termination is delayed until completion of the kernel operation, in order to ensure the integrity of the kernel. Because kernel operations in J-SEAL2 are non-blocking and have a short and constant execution time, termination cannot be delayed arbitrarily. Details concerning task termination in J-SEAL2 are presented in [3].

suite including the Embedded Caffeine Mark 3.0[8] benchmarks, as well as a series of custom benchmarks focused on cryptography.[9] We measured encryption and decryption with the AES and RC6 symmetric key algortihms using 12kB of input data and a key length and block size of 128 bit. We also evaluated the performance of the RSA public key algorithm with 256 bytes of input data, a key length of 1024 bits for both the public and the private key, an exponent of 8 bits for the public key, and an exponent of 1016 bits for the private key. The cryptography benchmarks have high practical relevance for the autonomous station, as all communication with the station is encrypted. Our performance measurements were collected on a JStamp board by Systronix[10], which is based on an aJile aJ-80 processor[11] running at 80MHz. The memory resources of the JStamp are very limited, it only offers 512KB of SRAM and 512KB of flash memory. We measured the performance with the following 3 configurations:

Unmodified (U): Neither the runtime system nor the benchmarks are rewritten. This setting gives a reference value for comparison.

Rewritten (W): The runtime system and the benchmarks are rewritten for CPU accounting.

Rewritten, Optimized (W*): In this setting some simple optimizations are applied to reduce the overhead. A simple loop detection algorithm marks the beginning of loops in the control flow graph of each method. Basically, CPU accounting is limited to the first basic block of code in a method, in an exception handler, and in a JVM subroutine, as well as to the blocks marked by the loop detection algorithm. The accounting weight is determined by the longest path in the control flow graph until the next accounting site is reached. Depending on the required precision, additional accounting sites may be inserted. For our measurements we did not require precise accounting.

Since embedded systems tend to be memory constrained, we registered also the overhead regarding the size of the binary program image (including the Java runtime system as well as the benchmark classes) that is written into the flash memory of the board. Table 1 presents the size of the binary program image for each setting; the overhead is only 7–16%.

Tables 2 and 3 present our performance measurements. Comparing the performance of the crytography benchmarks backs our design decision to use only symmetric key encryption for communication with the autonomous station, which does not have the necessary computing resources for public key cryptography. Especially the RC6 algorithm is well suited for embedded systems with limited resources, as it is based on simple and efficient operations.

[8] `http://www.pendragon-software.com/pendragon/cm3/`

[9] We used the implementation of 'The Legion of the Bouncy Castle', available at `http://www.bouncycastle.org/`, which is a free implementation of cryptography protocols offering, in addition to the JCE API, also a lightweight encryption API for Java 2 Micro Edition environments.

[10] `http://www.systronix.com/`

[11] `http://www.ajile.com/`

Table 1. Size of binary program image (values in bytes).

Benchmark	U	W	W*
Image size	152192 (1,00)	176712 (1,16)	163260 (1,07)

Table 2. Overhead of CPU accounting: Custom benchmarks (time in milliseconds).

Benchmark	U	W	W*
Bubblesort	4429 (1,00)	7023 (1,59)	5264 (1,19)
Hashtable	141 (1,00)	251 (1,78)	210 (1,49)
AES Enc. 12kB	4561 (1,00)	7352 (1,61)	6168 (1,35)
AES Dec. 12kB	6295 (1,00)	11865 (1,88)	8893 (1,41)
RC6 Enc. 12kB	956 (1,00)	1689 (1,77)	1480 (1,55)
RC6 Dec. 12kB	991 (1,00)	1710 (1,73)	1500 (1,51)
RSA Enc. 256B	7276 (1,00)	15265 (2,10)	10514 (1,45)
RSA Dec. 256B	921967 (1,00)	2084941 (2,26)	1347205 (1,46)
Geometric mean	4286 (1,00)	7830 (1,83)	6096 (1,42)

Table 3. Overhead of CPU accounting: Embedded CaffeineMark benchmarks (values are scores).

Benchmark	U	W	W*
Sieve	33 (1,00)	12 (2,75)	25 (1,32)
Loop	26 (1,00)	16 (1,63)	23 (1,13)
Logic	42 (1,00)	7 (6,00)	38 (1,11)
String	57 (1,00)	26 (2,19)	42 (1,36)
Float	24 (1,00)	19 (1,26)	22 (1,09)
Method	41 (1,00)	16 (2,56)	27 (1,52)
Overall	35 (1,00)	14 (2,50)	28 (1,25)

Without optimizations the overhead of CPU accounting is excessive, about 80% for the custom benchmarks and even more for the CaffeineMark suite. The measurements W* show the best results optimizations can achieve by minimizing the number of accounting sites (without changing the structure of programs). We can see that optimizations have the potential to significantly reduce the accounting overhead to 25% for CaffeineMark and about 40% for the more realistic custom benchmarks.

For the moment, we have to accept this significant overhead, because resource accounting is essential for the proper protection of the autonomous station and for billing. However, since current Java processors do not perform sophisticated optimizations, we will also experiment with standard code optimization techniques typically applied by optimizing compilers, such as method inlining and loop unrolling. These techniques aim at increasing the average size of basic blocks of code, thus reducing the accounting overhead (in addition to eliminating branch instructions and method invocations). However, because embedded systems frequently are memory constrained, optimization techniques that increase the code size have to be applied with special caution.

7 Autonomous Stations as On-Demand Bus Stops

Recently, we have used autonomous stations based on the architecture described in this paper to provide on-demand bus stops within a pilot project. This application is the first deployment of our autonomous station under commercial settings. The project was initiated by a bus operator in Austria, in order to avoid empty buses in the rural area and to improve the QoS. The customer has to press a button on the bus stop to order the next bus. If there is no request by a customer, the bus will not pass the stop. The on-demand bus stops are deployed in areas where the bus service is used rarely and irregularly. Consequently, the bus driver may frequently select a shorter route to save fuel and time, which also helps to compensate for delays due to traffic jams. Therefore, the overall bus punctuality (and hence the QoS) is improved.

This application involves four major components: The autonomous stations allowing customers to order buses, the stations' SuSe, a logistics application to coordinate the bus routes, as well as a simple application running on cell phones to interact with the bus drivers. The logistics application communicates with the autonomous stations through the SuSe. In our setting the communication between the logistics application and the bus drivers' cell phones is managed by the SuSe as well.

In order to reduce the hardware costs, the user interface of the station (its peripherals) is kept as simple as possible: It comprises two vertically grouped buttons with built-in LEDs and a 20x4 character LCD display with green background illumination. The LEDs on the buttons signal the possible input choices to the user. See figure 2 for some pictures. Furthermore, the station contains a beeper for acoustic feedback of user actions, as well as a motion detection sensor to activate the illumination of the display when a potential customer approaches the station. The communication with the SuSe is based on UDP packets on top of a GPRS [1] connection. The station includes a Motorola g18 GPRS GSM embedded wireless module, which is connected to the mainboard of the station by the standard RS232 serial interface. Since we could not find a free implementation of the PPP/IP/UDP protocol stack in pure Java, we had to develop our own implementation from scratch.

In the current version the application executing on the autonomous station is able to serve the schedule of one bus in two directions. The configuration of the application (i.e., the bus schedule) is completely dynamic, it is communicated by the server application periodically. Therefore, the bus schedule may be changed at runtime without uploading a new version of the application. The application running on the station also logs all user actions and periodically transmits the logging data to the server. This information is important to evaluate the user behaviour and the acceptance of the on-demand bus stop. Charging the application for consumed resources is not necessary in this pilot project, because the bus operator does not allow the uploading of foreign applications so far.

The benefits of installing the flexible and extensible autonomous stations will become evident in the long term. If the number of deployed stations increases, managing software updates (e.g., bug fixes, improvements of the user dialog,

Fig. 2. Pictures of the on-demand bus stop, its solar panel (also showing the GSM antenna and the motion detection sensor), as well as its simple user interface.

etc.) remotely reduces the maintenance costs. Moreover, once a large number of stations has been installed in a wide area, this infrastructure becomes attractive to deploy additional applications in a cost-effective way, such as traffic or environmental monitoring. The operator has to add the required devices to the stations and may open his infrastructure to other parties that will be charged for this service.

8 Conclusions

The contributions of our work are threefold: Firstly, we present the autonomous station, a new application of embedded Java, which relies on mobile code for program upload and remote maintenance. The station has a flexible and extensible architecture to support a wide range of different applications. Secondly, we show how a mobile object kernel can be adopted as an embedded operating system for the distribution and installation of applications by mobile code. We

are exploiting the advanced security features of the J-SEAL2 mobile object kernel in order to provide a reliable and secure system to host foreign applications, which are charged for their resource consumption. Finally, we present a concrete application, the on-demand bus stop, which is based on the architecture of our autonomous station.

Since resource management is a missing feature in current versions of Java, we are relying on program transformation techniques to expose the resource consumption of applications. Our techniques for resource accounting are fully portable and also work on embedded Java systems. However, because current Java processors perform no sophisticated optimizations, the accounting overhead is significantly higher than on standard Java runtime systems. With the aid of traditional program optimizations we will continue to reduce the overhead of portable resource management in embedded applications like our autonomous station.

Acknowledgments. Many thanks to Klaus Rapf and to Volker Roth for their useful comments that helped us to improve the paper.

References

1. 3GPP. 3GPP Specifications Home Page. Web pages at http://www.3gpp.org/specs/specs.htm.
2. G. Back, W. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI'2000)*, San Diego, CA, USA, Ott. 2000.
3. W. Binder. Design and implementation of the J-SEAL2 mobile agent kernel. In *The 2001 Symposium on Applications and the Internet (SAINT-2001),* San Diego, CA, USA, Jan. 2001.
4. W. Binder, J. Hulaas, A. Villazón, and R. Vidal. Portable resource control in Java: The J-SEAL2 approach. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-2001)*, Tampa Bay, Florida, USA, Ott. 2001.
5. W. Binder and V. Roth. Secure mobile agent Systems using Java: Where are we heading? In *Seventeenth ACM Symposium on Applied Computing (SAC-2002)*, Madrid, Spain, Mar. 2002.
6. G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull. *The Real-Time Specification for Java.* Addison-Wesley, Reading, MA, USA, 2000.
7. C. Bryce and J. Vitek. The JavaSeal mobile agent kernel. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*, Palm Springs, CA, USA, Ott. 1999.
8. G. Czajkowski and L. Daynes. Multitasking without compromise: A virtual machine evolution. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 'Ul)*, Tampa Bay, Florida, Ott. 2001.

9. J. Heidemann, F. Silva, C. Intanagonwiwat, R. Govindan, D. Estrin, and D. Ganesan. Building efficient wireless Sensor networks with low-level naming. In G. Ganger, editor, *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP-01)*, volume 35, 5 of *ACM SIGOPS Operating Systems Review*, pages 146–159, New York, Ott. 21–24 2001. ACM Press.
10. R. N. Horspool and J. Corless. Tailored compression of Java class files. *Software Practice and Experience*, 28(12):1253–1268, Ott. 1998.
11. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, Reading, MA, USA, second edition, 1999.
12. G. J. Pottie and W. J. Kaiser. Embedding the Internet: wireless integrated network Sensors. *Communications of the ACM*, 43(5):51–51, May 2000.
13. A. Villazón and W. Binder. Portable resource reification in Java-based mobile agent systems . In *Fifth IEEE International Conference on Mobile Agents (MA-2001)*, Atlanta, Georgia, USA, Dec. 2001.
14. J. Vitek and G. Castagna. Seal: A framework for secure mobile computations. In *Internet Programming Languages*, 1999.