Using a Secure Java Micro-kernel on Embedded Devices for the Reliable Execution of Dynamically Uploaded Applications

Walter Binder and Balázs Lichtl

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

Abstract. This paper presents the architecture of an autonomous, multi-purpose station, which executes dynamically uploaded applications. The station hardware is based on an embedded Java processor, which runs the system software and applications. The system software is built on top of a flexible, lightweight, efficient, and secure mobile object platform, 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 of the autonomous station. Applications executing on the station may be charged for their resource consumption. This paper also outlines an initial application of the autonomous station, which has been recently deployed in a pilot project.

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 within a specific environment 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 [9]. The system software is based on 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.

In the past years numerous research works have focussed on mobile object technology. This has resulted in a better understanding of the problems inherent

J. Blieberger and A. Strohmeier (Eds.): Ada-Europe 2002, LNCS 2361, pp. 125–135, 2002.

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 objects, which may seriously damage the execution environment itself or other mobile objects in the system [2,3,7,8].

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) [14], dynamic and customizable class-loading, multi-threading, built-in support for object serialization, and language safety [22]. Since the proliferation 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 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

[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. We are using this term (although it is overloaded with different semantics in various contexts), because it is also employed by the Java Community Process JSR 121 [13], which aims at specifying an API to isolate applications executing within the same Java runtime system.

bytecode programs, which supersedes a Java Virtual Machine (JVM) [14] implemented in software, as well as an underlying operating system layer. Therefore, the executive overhead is significantly reduced when compared to a JVM implemented in software. 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 following 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 application communication. In section 4 we discuss possible hardware configurations and the Java runtime system executing on the station. Section 5 shows how the J-SEAL2 mobile object kernel is adapted to serve as a kind of operating system for the autonomous station. Section 6 outlines an application of the autonomous station, which has been deployed in a pilot project. The last section concludes the 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. In this section 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 stations*, where the bus only passes the station, if a customer has explicitly ordered the bus. In section 6 we present some details of this particular application.
- In the context of industrial sensing autonomous stations offer a cost-effective alternative to a system of wired sensing elements.
- The general architecture of our autonomous station is also well-suited for cheap multi-purpose satellites, as well as for spacecrafts.
- 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.
- Autonomous stations can be used for traffic monitoring and for tracing.

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 rechargable battery, as well as application-specific sensors and actuators (input and output devices). Details concerning the hardware configuration of the station are presented in section 4. Depending on the concrete application, the hardware components may 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 rechargable battery have to be selected according to the expected insolation. On a place with low insolation a more expensive power supply system is needed.
- 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 usage site, 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, such as Ada 95 [19]. 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 without interrupting the execution of applications.
- 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.

3 Application Upload and Communication

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, 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.

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 identifier of the destination station, 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, quality-of-service (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). 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 kernel 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 re-packaged application is transmitted to the destination station through a wireless network. All messages originating from the application will be tagged with the AID, allowing the SuSe to dispatch them to the client.

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. Based on the AID, the SuSe is able to determine the recipient. Finally, the message is delivered to the network address, which the client has

[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.

[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 6 the autonomous station is connected to a public GPRS (General Packet Radio Service) network [1].

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.

4 Hardware Configuration and Java Environment

The autonomous station is an embedded system with application-specific peripherals. The core of the station is a Java processor that natively executes JVM bytecode instructions. Current Java processors offer sufficiently high performance at rather low clock rates, because there is no overhead due to a JVM implemented in software. The low clock rate helps to preserve power, which is crucial if the station has a limited power supply. The reference implementation of our autonomous station is based on a Java processor from aJile Systems[4] operating at 80MHz. All the system software and applications are implemented in pure Java; aJile provides a Java development kit including support for the Java 2 Micro Edition (J2ME), as well as special APIs to access hardware registers, serial ports, etc.

The J2ME has been designed for devices with limited resources like our autonomous station. Currently, J2ME defines two separate specification, the *Connected Device Configuration* (CDC) [12] and the *Connected, Limited Device Configuration* (CLDC) [11]. The CLDC has been designed for mobile devices with very limited computing resources, such as 120–500KB of memory. Therefore, the CLDC omits several APIs that are part of the Java 2 Standard Edition (J2SE), but not required by typical Java applications. In contrast, the CDC offers all core APIs of the J2SE, but requires considerably more memory (e.g., 2–16MB).

Our system software is based on the J-SEAL2 mobile object kernel, which is discussed in more detail in the following section. J-SEAL2 relies on several APIs of the J2SE, which are supported by the CDC, but not by the CLDC. Hence, the autonomous station has to be equipped with enough memory to use the CDC. In practice, at least 4MB of memory will be needed to run the CDC, the J-SEAL2 kernel, system services, as well as some (small) applications.

The peripherals of the autonomous station largely depend on the application context. The input and output devices must be chosen accordingly, but also the power supply system and the communication module highly depend on the environment. Typically, the power supply system consists of solar cells and a rechargable battery to ensure the autonomy of the station. However, sometimes a simpler power supply may be appropriate, ranging from an AC/DC adaptor to an uninterruptible power supply (UPS). Furthermore, in some settings mon-

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

itoring of the state of the power supply is needed, in order suspend low priority applications in case of a power shortage.

The requirements for the communication module are also application dependent. In some industrial environments a simple ethernet interface may be sufficient. Other settings may require a private wireless LAN, a bluetooth network, or even proprietary radio links. However, in the most common commercial setting, we assume that some kind of public wireless network will be used. Our reference implementation of the autonomous station employs a GSM module, where SMS, GSM data calls, or GPRS [1] may serve as communication bearers.

For some applications, the station may have to provide a user interface, allowing customers to interact with the station. The type of the user interface is not limited by our system; it can range from no user interface at all to a color touch-screen. The power consumption of the station and the required processing performance highly depend on the hardware peripherals in use. Some sensors are very simple to interact with, they may require only one bit digital input or output, while other devices may expose sophisticated interfaces. Thus, the system software of the station must be scalable and cause only little overhead, to support the execution of simple applications on low-cost and low-performance processors, but also of complex tasks on a station with the processing power of a workstation.

5 Adaption of the J-SEAL2 Mobile Object Kernel

The system software of the autonomous station is based on J-SEAL2[5] [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 [21], 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

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

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 does rely 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.
- J-SEAL2 has a modular and extensible architecture. Special system services, such as device drivers, can be provided by mobile objects.
- 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 [4,20]. 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[6]. 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 Application of the Autonomous Station

The individual components of the autonomous station (i.e., the integrated station hardware, the J-SEAL2 kernel, the communication module, etc.) have been separately developed and deployed in the context of different academic and com-

[6] 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].

mercial projects. For instance, J-SEAL2 has been used in a Swiss e-commerce project[7], providing us with valuable feedback.

Recently, we have used autonomous stations based on the architecture described in this paper to provide on-demand bus stations 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 non-urban area and to improve the QoS. The customer has to press a button on the bus station to order the next bus. If there is no request by a customer, the bus will not pass the station. The on-demand bus stations 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 station allowing customers to order buses, the stations' SuSe, a server hosting an application to coordinate the routes of the buses, as well as a simple application running on cell phones to interact with the bus drivers. The server application communicates with the autonomous stations through the SuSe. In our setting the communication between the server application and the bus drivers' cell phones is managed by the SuSe as well. As the focus of this paper is the architecture of the autonomous station and its applications, details concerning the applications running on the server and on the cell phones are not covered here.

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 station. Charging of the application for consumed resources is not necessary in this pilot project, because the bus operator does not allow the uploading of foreign applications.

In order to reduce the hardware costs, the user interface of the station 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 lighting. The LEDs on the buttons help to signal the possible input choices to the user. Furthermore, the station contains a beeper for acoustic feedback of user actions, as well as a motion detection sensor to activate the background lighting of the display when a potential customer approaches the station.

The communication with the SuSe is based on UDP [15] packets on top of a GPRS [1] connection. The autonomous 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. The development of the communication software was one of the most challenging tasks in this project.

[7] `http://anaisoft.unige.ch/`

The low-level programming of the Motorola g18 is based on the standard GSM AT command interface to initialize the GSM module and to connect it to the GPRS network. After the connection has been established, the point-to-point protocol (PPP) is used [18]. Since we could not find any implementations of the PPP / IP [16] / UDP protocol stack in pure Java, we had to develop our own implementation from scratch. We have chosen UDP over TCP [17], because the implementation of the complete TCP protocol would have been much more complicated. Another important issue is the reliability and fault tolerance of the communication. The low-level communication software of the station consists of three layers:

- The lowest layer maintains the device itself; it resets the device upon a failure and initializes it when the station reboots.
- The second layer maintains the GPRS connection. It reconnects whenever the connection breaks down and signals an error of the device to the lower layer, ensuring that the device will be initialized again.
- The upper layer is a packet delivery service, which handles acknowledgement messages and the resending of packets after a timeout. To compensate for the bad reliability of current GPRS networks (no QoS supported, high and varying latency, etc.), the acknowledgement and resending of UDP packets is crucial. This layer is used by the high-level communication services of the autonomous station, which provide stream communication, encryption, compression, etc.

7 Conclusion

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. Secondly, we show how a mobile object platform can be adopted 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 station, which is based on the architecture of our autonomous station.

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, Oct. 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, Oct. 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, Oct. 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'01)*, Tampa Bay, Florida, Oct. 2001.
9. J. Gosling, B. Joy, G. L. Steele, and G. Bracha. *The Java language specification*. Java series. Addison-Wesley, Reading, MA, USA, second edition, 2000.
10. R. N. Horspool and J. Corless. Tailored compression of Java class files. *Software Practice and Experience*, 28(12):1253–1268, Oct. 1998.
11. Java Community Process. JSR-000030 J2ME Connected, Limited Device Configuration. Web pages at `http://jcp.org/aboutJava/communityprocess/final/jsr030/index.html`.
12. Java Community Process. JSR-000036 J2ME Connected Device Configuration. Web pages at `http://jcp.org/aboutJava/communityprocess/final/jsr036/index.html`.
13. Java Community Process. JSR 121 – Application Isolation API Specification. Web pages at `http://jcp.org/jsr/detail/121.jsp`.
14. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, second edition, 1999.
15. J. Postel. RFC 768: User Datagram Protocol, Aug. 1980.
16. J. Postel. RFC 791: Internet Protocol, Sept. 1981.
17. J. Postel. RFC 793: Transmission Control Protocol, Sept. 1981.
18. W. Simpson. RFC 1661: The point-to-point protocol (PPP), July 1994.
19. S. T. Taft. Programming the Internet in Ada 95. *Lecture Notes in Computer Science*, 1088:1–16, 1996.
20. 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.
21. J. Vitek and G. Castagna. Seal: A framework for secure mobile computations. In *Internet Programming Languages*, 1999.
22. F. Yellin. Low level security in Java. In *Fourth International Conference on the World-Wide Web*, MIT, Boston, USA, Dec. 1995.