Evaluation of Life Cycle Functionality of Java Platform

ROMAN ROELOFSEN
Faculty IV, Department of Computer Science
University of Applied Sciences and Arts Hannover
Ricklinger Stadtweg 120, 30459 Hannover
GERMANY
roman.roelofsen@googlemail.com

ARNE KOSCHEL
Faculty IV, Department of Computer Science
University of Applied Sciences and Arts Hannover
Ricklinger Stadtweg 120, 30459 Hannover
GERMANY
arne.koschel@fh-hannover.de

IRINA ASTROVA
InVision Software OÜ
Lõõtsa 2A, 11415 Tallinn
ESTONIA
irinaastrova@yahoo.com

Abstract: - This paper at first describes future scenarios of using web services in (vehicle) embedded systems, viz. a car tracking service and an advertising service. Next the paper establishes requirements for the life cycle of services driven by the scenarios, viz. dynamic availability of services, versioning of services, composition of services, remote control of services, platform independence, access protection, security, isolation of services, communication between services and encryption. Finally, the paper evaluates the Java platform to see if it can meet those requirements.

Key-Words: - Java platform, (vehicle) embedded systems, web services, car tracking service, advertising service, life cycle of services

1 Introduction
There is no doubt that embedded systems such as navigators and multi-information displays will continue to spread in vehicles. In the future, embedded systems will allow drivers to use the Internet, access the address books of their mobile phones and play MP3 files directly from their USB sticks. Furthermore, the appearance of always-on UMTS networks with medium-bandwidth and local hot-spot networks will enable embedded systems to communicate with external systems. The term “external system” refers to anything that is outside of the vehicle.

2 Embedded Systems
“Embedded systems are losing their original meaning, which referred to small computational isolated (stand-alone) systems that give functional support for devices that do not fit to the definition of a computer. Today we can define an embedded system as a micro-processed device, thus programmable, which uses its computing power for a specific purpose” [1].

An embedded systems typically consists of memory (such as RAM, EPROM, ROM or flash memory), a processor (such as Intel x86, PowerPC or ARM), a clock and an input/output device (see Fig. 1). In addition, the embedded system can have communication interface for connecting to the development host.

![Fig. 1. Embedded system [1].](image-url)
3 Motivation
The motivation that leads us to evaluate the life cycle functionality in (vehicle) embedded systems stems from the following facts:

1. Embedded systems comprise the biggest niche in the market today. “Of the 9 billion processors manufactured in 2005, less than 2% became the brains off new PCs, Macs, and Unix workstations. The other 8.8 billion went into embedded systems” [2].

2. Vehicles are a promising market for embedded systems. “Automotive industry analysts predict that by 2010, software and electronics will account for 40 percent of a vehicle’s content, and some vehicles may contain 100 million lines of code” [3].

3. Embedded systems found in vehicles today are navigators and multi-information displays. However, automotive industry has started to invest in more complex embedded systems (e.g. Ford’s dashboard control system [4]).

4. In the future, vehicles will communicate more and more with external systems. This communication will be initiated either by vehicles themselves or by external systems.

4 Example Scenarios
Traditionally, web services were used in large distributed systems. However, the market trend towards embedded systems gives rise to the idea of using web services in embedded systems. This brings up a number of future scenarios that will make high demands on the life cycle functionality in embedded systems. To illustrate this, we’ll consider two examples:

1. Car tracking service (see Fig. 2a).
2. Advertising service (see Fig. 2b).

4.1 Car tracking service
This service will be used by a car rental company to get the position of a car. Installation of the service can be initiated either by the driver or by the car rental company.

4.2 Advertising service
This service will be used by the driver (e.g. waiting in a traffic jam) to get advertising information. The service can be installed by the local advertising server just in time as the car enters a local hot-spot network. The service can be saved for later reuse or it can be deleted when the car leaves the network.

5 Requirements of Example Scenarios
The use of web services in embedded systems is a challenging task because embedded systems rarely have enough memory and processing power to run web services [6].

The example scenarios specify the following requirements for the life cycle of services:

1. Dynamic availability of services.
2. Versioning of services.
3. Composition of services.
4. Remote control of services.
5. Platform independence.
6. Access protection.
8. Isolation of services.
9. Communication between services.

5.1 Dynamic availability of services
Since embedded systems have small memory footprints, it is important to keep the memory footprint at runtime as low as possible. Therefore, it should be possible to install services on a running system only when they are really needed and uninstall them afterwards (e.g. when they are no longer needed) without having to restart the whole system, as this would also affect other services and temporarily stop them from running. In addition, since services may change over time, it should be possible to update them at runtime. However, only the smallest possible set of services should be affected by that update.

The term “dynamic availability” [7] refers to a situation where services may come and go at any point in time. An important aspect of dynamic availability is that it should not be under control of embedded systems.

5.2 Versioning of services
Since services can be updated at runtime, it should be possible to reflect that update as new versions of services; i.e. it should be possible to assign version numbers to services.
5.3 Composition of services
Service can be composed of several (small) parts in order for embedded systems to install only the necessary parts at runtime. Again, this helps to keep the memory footprint at runtime as low as possible. Therefore, it should be possible to install services as a composition of several parts. However, the order in which these parts will be installed should not be fixed.

5.4 Remote control of services
Since services can be installed, uninstalled and updated by external systems, it should be possible to control the life cycle of services from the outside of embedded systems.

5.5 Platform independence
Since services can be installed by external systems, they cannot know in advance all environments where they will run. Therefore, it should be possible to run services in many different environments.

5.6 Access protection
Since embedded systems cannot completely trust to all of the services (especially those installed by external systems), it should be possible to prevent undesirable access to resources and data by malicious or buggy services. E.g. an embedded system inside the car can store secure information such as the driver’s address book. This information should be protected from access by the advertising service.

5.7 Security
Since external systems are insecure, it should be possible to prevent undesirable intrusion by unauthorized external systems (e.g. by requiring that external systems execute authentication).

While the term “access protection” refers to what is being requested, the term “security” refers to who is making this request.

5.8 Isolation of services
Since several (different) services can be installed on embedded systems, it should be possible to isolate services from each other.

5.9 Communication of services
While some services can live in isolation, others may work together to combine their functionality. E.g. the car tracking service can communicate with a GSP service to get the GSP data of a car. Therefore, it should be possible to enable communication between services.

5.10 Encryption
Embedded systems can communicate with external systems. This communication may require transmitting secure information between the two. E.g. the car tracking service can send the GSP data over an insecure network such as the Internet. Therefore, it should be possible to encrypt this information.

6 Java Platform
For evaluation of the life cycle functionality, we selected the Java platform because of its popularity. There are several adaptations of the Java platform to embedded systems; e.g. Java Standard Edition Embedded and Java Micro Edition. However, the life cycle functionality of these platforms is more restricted than that of the Java platform.

The Java platform provides the following concepts that help to meet the requirements of the example scenarios:
1. Java virtual machine.
2. Class path.
3. Class loader.
4. Sandbox.

6.1 Java virtual machine
The Java platform is based on a Java virtual machine (JVM) (see Fig. 3). This machine provides interaction between software and hardware. In particular, the JVM takes and analyses the instructions from the software, and maps them to the instructions for the underlying processor. To exchange these instructions, both should agree on a common “language”. In the JVM terminology, this language is called Java bytecode.

Java bytecode can be viewed as machine code that is targeted to the JVM rather than to the hardware. It is interpreted by the JVM at execution time. A big disadvantage of Java bytecode compared to machine code is that the execution of Java bytecode is much slower. To overcome this problem, the JVM usually uses a Just in Time Compiler that takes and analyses the instructions from the software by blocks, and caches the result of mapping.
6.2 Class path
Typically a class path is just an ordered list of directories or archive files (e.g., ZIP files). It is used as the index of all classes. E.g., the JVM uses the class path to load system library classes whereas the application uses the class path to load user-defined classes.

6.3 Class loader
The class path specifies the location of a class only. The JVM uses this information to look up and load classes. However, the actual task of doing lookup and loading the classes is accomplished by a class loader. The class loader is part of the JVM.

To look up a class, the class loader uses the fully qualified class name. This name consists of the package name and the class name. The class loader is then responsible for mapping the fully qualified class name to a resource (such as a file).

It is common to have several class loaders at the same time. Each class loader will have one specific class path to look up classes but different class loaders may use the same class path to look up classes.

There are two kinds of class loaders. One is the primordial class loader also called the bootstrap class loader. It is always part of the JVM; i.e., each JVM will consist of exactly one. The primordial class loader is responsible for loading classes from system libraries.

Another kind of class loaders is a class loader object. Class loader objects can be created at runtime; they are used to extend system library classes. To add classes to an application from a previously unknown location, the application could create a new class loader object with a class path that references the new location. Usually, at least one class loader object exits that is responsible for loading user-defined classes from the application.

Each class loader object has a reference to the primordial class loader. When a class loader object receives a request to load a class, it will first delegate this request to the primordial class loader. This delegation will always take place, even if the class loader object would be able to load the class by itself.

6.4 Sandbox
The JVM uses a sandbox as a security mechanism to control the execution of an application and access to resources such as files and sockets.

Since the JVM wraps the application and abstracts it completely from the hardware platform and operating system, the application must request access to resources from system libraries of the JVM. If this access is not allowed, the JVM will reject the request.

7 Evaluation of Java Platform
We evaluated the Java platform against the requirements of the example scenarios. Table 1 summarizes the results of our evaluation.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Is requirement met?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic availability of services</td>
<td>Yes/No</td>
</tr>
<tr>
<td>Versioning of services</td>
<td>No</td>
</tr>
<tr>
<td>Composition of services</td>
<td>Yes</td>
</tr>
<tr>
<td>Remote control of services</td>
<td>No</td>
</tr>
<tr>
<td>Platform independence</td>
<td>Yes</td>
</tr>
<tr>
<td>Access protection</td>
<td>Yes/No</td>
</tr>
<tr>
<td>Security</td>
<td>No</td>
</tr>
<tr>
<td>Isolation of services</td>
<td>Yes</td>
</tr>
<tr>
<td>Communication between services</td>
<td>No</td>
</tr>
<tr>
<td>Encryption</td>
<td>No</td>
</tr>
</tbody>
</table>

7.1 Dynamic availability of services
Class paths and class loaders enable to install services at runtime. The primordial class loader would represent the basic functionality and for each service a new class path and class loader could be created.

Each service would be loaded by class loader. This loading could take place either during the start of an application or at runtime. Therefore, loading services could be deferred until they are really needed. However, services could not be uninstalled or updated at runtime.

7.2 Versioning of services
Class loaders would use the fully qualified class names to identify services. However, the primordial class loader does not provide any capabilities to assign version numbers to services. Therefore, a custom solution will be necessary.

7.3 Composition of services
Class paths and class loaders enable to install services as composite applications. Each part of a service could be represented as an archive file, which is an element of the class path. The class loader would then provide a unified view on this file.

The term “a composite application” refers to an application that is composed of several (small) components such as applets, servlets and JavaBeans. These components can be loaded at runtime (e.g. to add new functionality to a running application).

7.4 Remote control of services
The JVM does not provide any capabilities to control class loaders remotely. Therefore, a custom solution will be necessary to control the life cycle of services from external systems.

7.5 Platform independence
The JVM uses bytecode to represent an application. By doing so, the JVM abstracts from the hardware platform and operating system. Therefore, services would run on any environment where the JVM is implemented.

7.6 Access protection
The JVM uses the sandbox to prevent undesirable access to resources such as files and sockets. But it does not provide any capabilities to control access to memory, processor and data. Therefore, malicious or buggy services can allocate all available memory and thus arise OutOfMemoryException. They can start more and more threads and thus prevent the JVM from working normally. Or they can overwrite static variables such as System.out [9].

7.7 Security
As said above, a custom solution will be necessary to control the life cycle of services remotely. This solution should be able to authenticate and authorize external systems.

7.8 Isolation of services
Class loaders enable to isolate services from each other. In particular, services loaded by one class loader would not have access to services loaded by another class loader.

7.9 Communication between services
As said above, services would be isolated by class loaders. But such isolation would complicate communication between services. In particular, this communication becomes possible only through a “bridge” in a shared, parent class loader. However, this bridge is not part of the standard JVM and therefore requires an additional custom solution.

7.10 Encryption
The JVM does not provide any capabilities to encrypt information. Therefore, a custom solution will be necessary.

8 Conclusion
The Java platform is a promising platform for embedded systems and its concepts (such as the JVM, class path, class loader and sandbox) help to meet some of the requirements of the example scenarios. However, further work is necessary to provide versioning of services, remote control of services, access protection, security, communication between services and encryption.

Moreover, the Java platform’s landscape is now subject to a large segmentation. This is reflected by the current state of system libraries and JVM versions used in embedded systems (e.g. mobile phones vs. server applications).

9 Future Work
In the future, we’ll evaluate the OSGi platform against the requirements of the example scenarios. This platform is based on the Java platform but it is more advanced in respect with the life cycle functionality.

References: