RFP 126 - OSGi ME: An OSGi Profile for Embedded Devices

Confidential, Draft

16 Pages

Abstract

The OSGi R4 specification [1] relies on techniques that are costly on most of the embedded devices today and tomorrow. The embedded world effectively requires some features targeted by OSGi specifications – e.g., application code structuring and openness to third parties – while demanding small cost and energy footprint. Therefore, a profile compliant with more embedded execution environments is required.

0 Document Information

0.1 Table of Contents

All Page Within This Box

0.2 Terminology and Document Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in [2].

```
Source code is shown in this typeface.
```

0.3 Revision History

The last named individual in this history is currently responsible for this document.

Revision	Date	Comments
Initial	July 26, 2009	First Draft Andre Bottaro PhD, Orange Labs
0.1	August 5, 2009	Review Fred Rivard PhD, IS2T
0.2	August 10, 2009	Integration and completion (mainly use cases) Andre Bottaro PhD, Orange Labs

All Page Within This Box

Revision	Date	Comments
0.3	August 11, 2009	Integration and review tentatives Added security requirements Fred Rivard PhD, IS2T
0.4	August 12, 2009	First version ready for review by external parties Andre Bottaro PhD, Orange Labs
0.5	August 14, 2009	Integration of the comments of reviewers Fred Rivard PhD, IS2T Added security paragraphs in the Application Domain, Problem Description, Use cases to introduce already written requirements Andre Bottaro PhD, Orange Labs

1 Introduction

More and more intelligence is inserted into embedded devices today. The number of applications is growing fast in market like the Home, the Vehicule, the Mobile markets. In order to deliver applications fast, application code structuring and flexibility are key points in Embedded Computing. Actors also think to open the devices to third party applications and accelerate service delivery through an open application store.

Device openness to third-party application demands the ability to deploy application modules developed by distinct service providers on the same device on a robust basis. Many execution platform technologies enable the management of modular software applications [3][4]. They define the structure of deployment units called modules, specify code sharing and isolation mechanisms between them and common programming interfaces to deploy and activate these entities. The openness of such models to the deployment of modules that may have been designed at distinct time and by distinct providers strengthen the requirements on robustness. Code sharing and isolation mechanisms must be strictly defined and controlled. The deployment - i.e., the installation, the update, the uninstallation, the activation and the deactivation – of modules must be performed with precise sound semantic. Transactional guaranties must be allowed on critical pieces of software. Security means must allow the definition of usage rights by and to modules. The OSGi R4 specification [1] answers part of these requirements with application modularity at design, development, and deployment phase.

However, OSGi specifications have drawbacks that are not compatible with embedded environments. First, OSGi modularity relies on heavy mechanisms, e.g., user-defined Java class loaders. These mechanisms are commonly available on execution environments that are often considered overweight for embedded devices where cost and energy consumption matter. The OSGi Minimum Execution Environment relies on Java ME CDC whereas the most spread Java execution environment on embedded devices is Java ME CLDC without file system. All OSGi releases assume Java virtual machines that weigh few megabytes both in static memory and volatile memory

while embedded devices have adopted Java ME virtual machines which footprint may be as low as few tens of kilobytes instead of few megabytes.

Second, OSGi modularity leave many flaws that developers have to tackle themselves, thus lowering the expected robustness of applications. For instance, stopping an OSGi module, i.e., a bundle, remains a delicate action. For example, there is no guarantee that bundle internal threads are stopped, and worse, the OSGi specification also warns the developer that *potentially harmful stale references* – references to an object that belongs to the bundle that is stopped or is associated with a service object that is unregistered – may remain.

Third, OSGi specifications have progressively adopted more and more features that present drawbacks without answering requirements that are mandatory on embedded devices. Some can be considered as too complex such as the support of multiple versions of a same class defined by its name. Some can be considered as security threats, e.g., dynamic imports, bundle requirements, bundle fragments, bundle extensions.

Therefore, this RFP is calling for the specification of OSGi ME, an OSGi profile that is aligned with embedded constraints. It enumerates the requirements of embedded devices ready to be open to third party applications without strongly threatening their robust and embedded nature. Thanks to OSGi ME, OSGi applications will have a simpler development process and will be delivered with more guarantees on embedded devices where costs do matter.

2 Application Domain

2.1 Concepts

This RFP takes into account the entities defined in the OSGi Core specification, e.g., Framework, Bundle, Service, Event. Other concepts used by this RFP rely on Java specifications .

2.2 Software Modularity

Software modularity first enables the separate download of system parts, and thus the adaptation of device software to the environment requirements. The software provider maintains a set of modules without maintaining a set of complete systems. Deployment features comprise the installation, uninstallation, update, activation and deactivation of software entities on a common platform.

Second, these technologies also offer code sharing and isolation mechanisms between deployed software units. Code sharing optimizes memory usage and facilitates the re-use and evolution of the units, which is one effective way to reduce software costs. Code isolation allows the definition of application parts that can not be used by other units and remain private to associated service providers.

Software modularity is claimed by several execution environments today, e.g., Linux distributions, .NET, OSGi, Java MIDP. The concept is however implemented with various levels of completion. Here are some examples at a glance [3]. Linux distributions such those of Debian family define dependencies between modules called 'packages' without defining finer-grain for code sharing and isolation. .NET enables the installation and activation of modules called 'assemblies' at runtime without allowing for the individual assembly unload and update [4]. Java

MIDP3 defines strict code sharing and isolation mechanisms but sharing is only static and no code is shared dynamically between active-able applications, i.e. the MIDlet Suites.

The OSGi R4 specification [1] defines code sharing and isolation at runtime at the fine-grain of Java class, a bundle being a set of classes. It also defines common APIs to deploy, i.e., install, uninstall, update, start, stop, deployment units called bundles. OSGi core mechanisms enable these operations at runtime. However, the embedded nature and the robustness of these core mechanisms can be improved.

2.3 Highly-constrained environments

The “embedded world” has two characteristics that makes it appealing and challenging for software engineers: (1) resources are scarce (sometimes really scarce), (2) the software cannot fail.

Embedded software are not just "software on small machines". Embedded applications usually is about optimizations that strive to create a software for an hardware. And system integration is one of the most important aspect which heavily impacts testing and debugging. This is one big difference between PC-based system and embedded system. For PC-based systems, the development machine is the machine that will execute the application (or a machine really similar). For embedded systems, there are actually two systems: one used to design the application (a PC-like desktop) and one to run the application (the device).

It is important to qualify with figures what is “scarce” from what is not. This RFP is meant for embedded devices where robustness must be ensured and where resources are scarce. Typical small hardware range from 8-bits to 32-bits (AVR, MC16, ARM7, ...) running as low as 8Mhz, with less than 256KB of flash, less than 64KB of ram. Even on more powerful devices, small footprint is often synonym of high execution speed.

OSGi versions have evolved to target more and more powerful systems and is pervasively available on PCs and PC-like devices. All OSGi releases assume capabilities available in Java virtual machines that weigh few megabytes both in static memory and volatile memory. OSGi ME would be designed for Java ME virtual machines which footprint are few tens of KB instead of few MB.

2.4 Java Garbage Collector

One of the most important feature of the Java technology is the garbage collector (see [5] and [6]). Thanks to the latter, software engineers are free to allocate objects without taking care of their de-allocations.

The philosophy of OSGi ME may be summarized by: “A faulty service cannot cause the failure of the entire platform”. OSGi ME needs to avoid any “manual-free-mechanism”. So as the de-allocation of objects is an activity that may jeopardize the whole system if not done correctly, such activity cannot be left into the hands of the software engineer.

2.5 Java User-Defined Class Loaders

Java Virtual Machine specification [5] defines class loaders for Java SE and EE editions. Java ME CLDC prevents the use of user-defined class loaders. Java ME CLDC forces applications to only use the classes and interfaces that are explicitely declared as Java imports at development time. While this effectively forbids the flexible user-defined class loaders, code sharing and isolation between applications could be allowed if the sharing capabilities are closed enough and strictly controlled by mechanims in the JVM.

2.6 Java Class Initialization

If a class needs to be initialized, it posses a “hidden” <clinit> method [5] generated by the compiler. Basically, the compiler goes sequentially through the class source file and concatenates in sequence all static fields

initializations and static block statements as they appear into one method called <clinit>: this method is the set of initializers.

2.7 Security

Openness to third-Party applications can be moderated with an access rights management system. Distinct actors obtain different trust levels depending on their knowncompetencies and their role. Access rights management on Java platforms is specified in Java 2 Security [8].

3 Problem Description

3.1 Code Sharing and Isolation Mechanisms for Embedded Software

A fine-grained code sharing and isolation model is defined by the OSGi R4 specification [1]. Bundles share classes, mainly service interfaces, by declaring them 'exported' while keeping other types private. Bundles that need to use them have to declare them 'imported'. Classes are named with their qualified name. These declarations are packaged with the bundles before deployment time. Instantiated objects registered as services are shared at runtime, which defines a service based collaboration model. Code sharing and isolation mechanisms are the basis of service-oriented programming.

A service is a regular object that is type-compliant with an exported type and that is registered in a common registry, named the Service Registry. Service requesters (objects that require a service) share the same service object registered by a service provider. The sharing and isolation model enables provider to share instantiated objects known by their exported interfaces and, at the same time, isolate private classes that implement these interfaces. Good service oriented practices advocate the separation of bundles exporting the service interfaces – Service API – from bundle that implement the service provider and the service requester (see Figure 1).

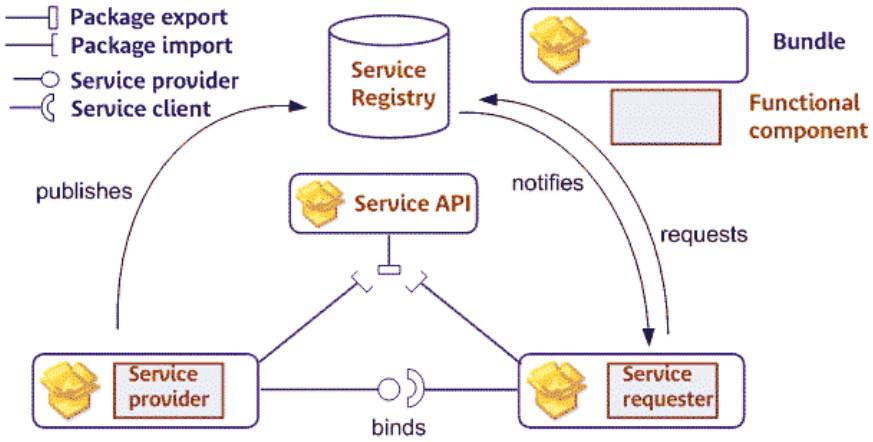

Figure 1 OSGi service oriented programming relying on a code sharing and isolation model

These code sharing and isolation features, as well as the service-oriented programming that it enables, are demanded by the embedded world. Java is already well adopted by embedded designers as it brings application

All Page Within This Box

portability and as it allows for a better evolution with object-oriented structuring. However, the OSGi R4 specification [1] first makes them tightly linked with an architecture made of user-defined class loaders, a Java feature that is heavy. It also adds some sharing mechanisms that require less strictly defined declarations than the basic mechanism described above.

User-defined class loaders are heavy. That is partly why Java ME CLDC [7], the Java execution environment that is well-spread on embedded devices, does not provide them. Another reason is that they are too much flexible and enables the wide openness of applications to any code snippet, a feature that can be considered as a major security threat. Actually, this is not the only technical solution for the definition of the strict class sharing and isolation features that are targeted here. Thus, an OSGi profile for embedded devices must not impose any implementation style such as an architecture made of user-defined class loaders for this core feature (Req. 2).

The OSGi R4 specification [1] supports the load of multiple versions of the same class identified by his full name. However, this feature is not desirable since this RFP targets very constrained devices where the memory storage is limited. Code sharing between applications at runtime is an asset for minimizing the static and volatile memory on the devices. Code interoperability is a constraint that arises from this policy. Thus, the load of only one version of a package is requested (Req. 6).

Other features of the OSGi R4 specification [1] can be considered as security threats that are not wished on embedded devices: dynamic imports, optional imports, bundle requirements, bundle fragments, bundle extensions (Req. 5). Wildcard dynamic imports enables developers to bypass the strict sharing and isolation declarative model that is targeted, which is not acceptable on embedded devices. These imports can be avoided by the share of a common service interface implemented by bundles with the classes that would have been dynamically imported otherwise. Moreover, even without wildcards, dynamic and optional imports are not desirable on embedded software. They make developers responsible of checking the completeness of bundle code resolution in internal bundle code. Indeed, the OSGi framework enables the start of a bundle when optional imports are not resolved. In case a code section using related classes is entered, resolution errors are thrown at runtime, a situation that has to be avoided on embedded devices. Therefore, for the sake of robustness, all the declared imports have to be checked by the OSGi ME framework before starting time and any unresolved import has to prevent the start of a bundle. Bundle requirements are a shortcut at development time that breaks the fine-grained code sharing model. This mechanism can be avoided by setting the right package imports and exports. Fragments are mainly meant for internationalization in [1]. It can be considered as a hack to add code to a bundle that was not meant for that. If extensibility is needed, then either bundle refactoring/versioning must be done or the service collaboration model must be used. Extensions are meant to add packages and resources to the boot delegation and to the system bundle. It allows for the Java platform and the OSGi framework modularity. However, modularity at this level could be considered as a too flexible feature and a major security threat.

3.2 Dynamic Deployment of Modules on Embedded Software

The dynamic deployment of bundles is highly wished on most embedded devices that are ready to be open to third-party applications (Req. 3). It covers not only the installation of bundles for an increase of functionalities but also the update and the uninstallation of them for resource saving and substitution. Modular code downloading at runtime is however not widely spread on embedded devices, which often only enables firmware upgrade. Modular code substitution and uninstallation are even more difficult, the .NET platform showing the example of a platform that is able to load modules called assemblies without being able to unload them individually [3][4] .

3.3 Native Code Linkers in Embedded Environments

An OSGi execution Environment for embedded devices is only about Java. It does not have to mention any mechanism related to different OS/RTOS[1], C and other native languages linkers (Req. 4). Indeed, the semantic of the linker may vary a lot, depending on the hardware capabilities, and can go from all statically linked software on

[1] There are roughly 150 commercial different RTOS, and at least as many in-house ones.

a device that does not support MMU (Memory Management Unit) and execute code in flash, to a fully dynamic linked at runtime software that execute in ram with MMU support.

3.4 Embedded Java Execution Environments

The Java execution environment that is the most widely spread on embedded devices is Java ME CLDC [7]. An OSGi profile for embedded devices would be compliant with this target (Req. 8). However, the OSGi R4 specification [1] defines a minimum execution environment that is a subset of Java ME CDC/Foundation Profile. The heavy use of class loaders, in particular, forbids the specification to be implemented on a Java ME CLDC execution environment.

3.5 Human-Readable Metadata in Constrained Environments

The memory constraints of some devices are such that gaining some hundreds of bytes is important. On the contrary, the openness of a modular programming model advocate the attachment of sometimes verbose metadata to every deployment unit, here OSGi bundles. In some cases, a trade-off could be the storage of metadata on a server distinct from the embedded device where applications are installed. In these cases, metadata would not be stored on the device (Req. 7). Although Strings have convenient support in the Java language [6], it comes at a price of high memory footprint (both flash and ram).

3.6 General Robustness

Even more than PC-based applications, embedded applications require robustness. The embedded applications are required to be stable in every situation. Some development tasks are error-prone and must be, as much as possible, be implemented by the underlying execution environment itself.

First, the implementation of an OSGi framework has to be thread-safe (Req. 12). The OSGi ME specification has to state this main requirement for the implementations.

Part of the clean-up associated to service management is left to developers in the OSGi R4 specification [1] (Req. 13, Req. 14). The problem is well described by a section about *stale references* – references to an object that belongs to the bundle that is stopped or is associated with a service object that is unregistered. It rises when a service is unregistered and when some living objects still maintain references to objects belonging to the service (graph of objects). Memory usage remains high as the garbage collector does not detect that the service is not used any more. It can happen when a bundle is stopped since associated services are automatically unregistered at this time thanks to the OSGi specification.

Another error-prone task that is left to bundle developers (Req. 15): Bundle developers are required to call a method – the BundleContext.ungetServiceReference() – to indicate as soon as a service reference is not used anymore. The framework maintains a count of used service references by every bundle. This task has often no visible impact for the developer, who is therefore given to forgetting it. Thus, this mechanism renders this count unreliable and the bundle life cycle is therefore not handled correctly.

3.7 Robustness and Class Initialization

The Java virtual machine specification [5] (section 2.17.4) defines its intent for the initialization of the classes and interfaces: “The intent here is that a type[2] have a set of initializers that put it in a consistent state and that this state be the first state that is observed by other classes”.

[2] A type is here a class or an interface

All Page Within This Box

Although the JVM specification [5] defines a clear intent, the semantic to archive it is not sound. In particular, there are more than one “first-state” (possibly one per startup) and, the way to get that “one-of-first-state” may cause non obvious deadlocks.

This RFP emphases the initial intent of [5] by demanding a stricter obedience to the philosophy it implies. Indeed, the requirements enforces the uniqueness of the “first state” and the fact that deadlocks may not appear while classes and interfaces get initialized. The initialization of a device must be unambiguous.

3.8 Transactions

In order to enforce the robustness of the platform, an API for transaction is needed to complete the OSGi R4 specification [1]. Transactions enable developers to define stable states and indicate part of the code which results have to be rolled back if an error happens during the execution of this part (Req. 16).

Furthermore, the adequate transaction mechanisms would include the modifications on objects according to the all-or-nothing rule (Req. 17). And while in a transaction, all modifications on objects and static variables cannot be seen by other threads until the successful end of the transaction (Req. 18). These mechanisms are to be applied especially on bundle deployment operations (Req. 19).

3.9 Security

Java ME CLDC recommends not to use Java 2 Security [8] because of the cost of this API and its implementation on embedded devices. It can be effectivelly considered reasonable not to specify rights management for small applications developed by only one party (Req. 21).

However, such a security layer is mandatory on top of platforms hosting evolving applications comprising several bundles developed by distinct parties (Req. 20). For these complex embedded applications, Java 2 Security seems to be the right security layer. For cost sake, some features like the external human-readable representation is not essential.

4 Use Cases

4.1 General Use Cases

4.1.1 Installing bundles at runtime

The user wishes to augment the features provided by an embedded device without stopping an application running on it. For instance, the device is an home automation platform sending alerts upon the detection of smoke or intrusion. The associated service providers inform the user that the alarm of the smoke detector can be also used by the intrusion application after the installation of an application module. Service providers confirm that this installation can be done at any time even when the user is outside the home since it will not interrupt smoke and intrusion detection. When the user accepts the installation, the download and the installation of a group of bundles is performed.

Requirements: Req. 3.

4.1.2 Uninstalling Bundles at Runtime

The installation of a set of features requires the installation of some bundles and the uninstallation of some others. The uninstallation of the latter is demanded for storage constraints and to avoid some interferences between the new bundles and the previous ones. The installation and the uninstallation are performed at runtime.

Requirements: Req. 3.

4.1.3 Updating Bundles at Runtime

For an always improved behavior, the embedded device requires the update of part of the application. And since a critical application is running on the platform, the updates that are not related to this application are performed without stopping the overall application.

Requirements: Req. 3.

4.1.4 Starting Bundles at Runtime

The OSGi ME framework administrator wishes to start some services demanded by the user. For these services to be available, some bundles have to be started. The administration server manages standard OSGi platforms.

Requirements: Req. 1, Req. 3

4.1.5 Stopping Bundles at Runtime

A bundle is stopped and the services that were registered by the bundle are automatically unregistered and are further detected by the garbage collector to be collected.

Requirements: Req. 1, Req. 3, Req. 13, Req. 15.

4.1.6 Forbidding Bundle Metadata on an OSGi ME Platform

A embedded device with typical resources constraints (256Kb flash / 64Kb ram) runs OSGi ME with a dozen of bundles on a small JVM. The flash and ram capabilities are so constrained that the platform is configured with bundle metadata declared as not available at runtime. The metadata has been interpreted to resolve dependencies and to link binary code ; no deployed bundle requires metadata for its execution.

Requirements: Req. 7.

4.2 Minimum Execution Environment

4.2.1 Java CLDC being the Execution Environment

Java CLDC is the underlying Execution Environment. Bundles that have their imports resolved exports their classes that are declared exported.

Requirements: Req. 2, Req. 5, Req. 6, Req. 8

4.3 Java Class Initialization Order

4.3.1 Starting and Restarting the OSGi ME Framework

The Java class initialization is made once at starting time. If the framework is restarted, the same initialization order is executed.

All Page Within This Box

Requirements: Req. 9, Req. 10, Req. 11

4.3.2 Installing and Updating a Bundle

Installing and updating a bundle or a group of bundles at runtime is possible on some embedded devices with OSGi ME depending on hardware capabilities. The class initialization is then performed on the classes deployed in the group of bundles after every installation or update.

Requirements: Req. 3, Req. 9, Req. 10, Req. 11

4.3.3 Uninstalling a Bundle

Uninstalling a bundle or a group of bundles at runtime is possible on some embedded devices with OSGi ME depending on hardware capabilities. The class initialization is changed at least after restart. It is performed as if the uninstalled bundles were never deployed. Bundles that rely on uninstalled bundle get recursively Unresolved.

Requirements: Req. 3, Req. 9, Req. 10, Req. 11

4.4 Robustness Use Cases

4.4.1 Multi-threading and no Deadlock

While Java applications Java are by nature multi-threaded, the OSGi ME has to be thread-safe. The user application should be able to synchronized on any object without risking mysterious deadlocks.

Requirements: Req. 12

4.4.2 Using an Object attached to a Service that is Unregistered

When calling a method on an object that belongs to the graph of objects that defines a service, an exception is thrown when the following reason occurs: the service is dead because it has been unregistered. Same applies if the method is called on an object that is part of the dead service object that has been unregistered.

Requirements: Req. 13

4.4.3 Detaching an Object from a Service

A service defines a set of cooperative objects that provide the service all together. One is elected and is registered as the service object: the root of the service. Java objects are created as being from no service. An object gets attached to a service as soon as an object of such service links (points) to that object. The explicit detach action done by the application allow to detach the object from the service from which it does not belong to.

Requirements: Req. 14

4.5 Transaction Use Cases

4.5.1 An Error occurs when executing a Code Section marked as a Transaction

The developer indicates that a transaction begins at a particular line and that it ends after some lines of code. Whenever an error occurs during this execution of this code section, all the intermediary modifications done on both any static variables and any object during the transaction are rolled back as if the code execution has never entered the transactional section.

Requirements: Req. 16, Req. 17, Req. 18.

4.5.2 An Error occurs when starting a Bundle

The start() method of a bundle is badly written; a service object is called in this method whereas the service is not registered yet. An exception is thrown and all the intermediary results are rolled back as if the start() method has not been called. The bundle is in the OSGi ME Stopped state.

Requirements: Req. 16, Req. 17, Req. 18, Req. 19.

4.6 Security Use Cases

4.6.1 Different Permissions for Distinct Bundles

Configuring security permissions is possible on some OSGi ME implementations depending on the hardware constraints. The administrator is able to configure different permissions to distinct bundles. Some bundles have the permission to use some packages and services while others do not.

Requirements: Req. 20, Req. 21, Req. 22.

5 Requirements

5.1 General Requirements

Req. 1. OSGi ME SHOULD be compliant with the most recent version of the OSGi core specification unless some features are specified not to be so. This RFP assumes this version to be OSGi R4 v4.1 [1].

Req. 2. OSGi ME MUST not mention any specific mechanism to ensure the name scoping of code sharing between bundles.

Req. 3. OSGi ME MUST define the installation, start, stop, update, uninstallation capabilities at runtime according to the underlying device capabilities and targeted use cases of a specific OSGi ME implementation.

Req. 4. OSGi ME MUST not mention any mechanism related to different OS/RTOS and their related native code linkers.

Req. 5. OSGi ME MUST NOT support dynamic imports, optional imports, bundle requirements, fragments nor extensions. OSGi ME MUST ignore DynamicImport-Package, Require-Bundle and Fragment-Host manifest headers as well as the import resolution directive.

Req. 6. OSGi ME MUST support the load of only one version of a package identified by his name.

Req. 7. OSGi ME MUST allow OSGi framework administrators to forbid manifest properties to be installed with bundles. Bundles MAY NOT have their properties accessible at runtime.

5.2 Minimum Execution Environment

Req. 8. OSGi ME MUST be compliant with Java ME CLDC [7].

5.3 Java Class Initialization Order

Req. 9. Although the precise order of the sequence of <clinit> calls is not necessary known, it MUST be defined once for all, before any code execution and it MUST remain the same across several restarts of the system.

Req. 10. The order MUST be compatible with the Java semantic [6]. When a class depends on other classes, those classes SHOULD be initialized first. We list a few of these dependencies: object creation, superclass, interfaces, methods receiver, arguments and fields types, ... [5] completely describes the initialization process and its implications on the order of the <clinit> sequence.

Req. 11. This order of <clinit> sequence does not rely on runtime behavior, but only on the application code. The constraint is: if the application code does not change, the order remains the same. Dependencies of classes upon themselves define a graph of dependencies. This graph may depict cycles. The graph MUST be made linear in an order which depends only on the graph itself, that is, a same graph always produces a same linear order for a given implementation of OSGi ME.

5.4 Robustness

Req. 12. An OSGi implementation MUST be thread-safe in the sense that: (i) any method can be called several times from several threads concurrently on the same receiver without jeopardizing the state of that receiver. This definition allows the number and the size of critical sections to be minimized, without compromising the robustness of the framework. (ii) the user application may be synchronized on any accessible object of the framework without causing a deadlock with the implementation of the underlying platform. The user is free to synchronize its application design according to its need without risking mysterious deadlocks.

Req. 13. Stale references MUST NOT appear in any case. The use of an object attached to a unregistered service MUST throw an exception (with a name like DeadObjectException).

Req. 14. An API MUST allow to detach objects from registered services.

Req. 15. The count of service references MUST NOT rely on manual tasks at development time.

5.5 Transactions

Req. 16. An API must allow the management of user-defined transactions.

Req. 17. Transaction mechanisms SHOULD include the modifications on objects follow according to the all-or-nothing rule.

Req. 18. While in a transaction, all modifications on objects and static variables SHOULD NOT be visible by other threads until the successful end of the transaction (which is success).

Req. 19. Deployment actions – i.e., installation, uninstallation, update, start, stop – on bundles must be performed with transactional guaranties.

5.6 Security

Req. 20. OSGi ME MUST provide the minimal core foundation of the Java 2 Securiy APIs [8] which is based on stack introspection. The core feature is the so-called Basic Algorithm with Privileged Operations

(section 6.4.4 in [8]). Pragmatically, two methods implements it: checkPermission and doPrivileged.

Req. 21. If security is not supported by an OSGi ME implementation, the code MUST run as if all bundles have AllPermission.

Req. 22. A bundle MUST have a single Bundle Protection Domain [8].

6 Document Support

6.1 References

[1]. OSGi™ Alliance, OSGi™ Service Platform, Core Specification and Service Compendium, Release 4, Version 4.1, April 2007

[2]. Bradner, S., Key words for use in RFCs to Indicate Requirement Levels, RFC2119, March 1997.

[3]. André Bottaro, Levent Gürgen, Maxime Vincent, François-Gaël Ottogalli, "Software Management of Heterogeneous Platforms", 3rd IEEE International Workshop on Telecommunication Networking, Applications and Systems (Tenas'09), Bradford, UK, May 2009

[4]. Clément Escoffier, Didier Donsez, Richard S. Hall, "Developing an OSGi-like Service Platform for .NET", 3rd IEEE Consumer Communications and Networking Conference (CCNC'05), Las Vegas, NV, USA, January 2006

[5]. The Java Virtual Machine Specification, Second Edition by Tim Lindholm and Frank Yellin. Addison-Wesley, 1999, ISBN 0-201-43294-3

[6]. The Java Language Specification, Third Edition, May 2005, ISBN 0-321-24678-0

[7]. Connected, Limited Device Configuration 1.1 (JSR-139), Sun Microsystems, Inc, http://jcp.org/jsr/detail/139.jsp.

[8]. Li Gong, Gary Ellison and Mary Dageforde, Inside Java 2 Platform Security: Architecture, API Design, and Implementation, Sun, 2nd Edition, 2003

6.2 Author's Address

Name	André Bottaro, PhD
Company	Orange Labs, France Telecom Group
Address	28 Chemin du Vieux Chêne 38243 Grenoble Meylan Cedex, FRANCE
e-mail	andre.bottaro -at- orange-ftgroup -dot -com

All Page Within This Box

Name	Fred Rivard, PhD
Company	IS2T – Industrial Smart Software Technology SA
Address	1 rue de la Noë 44321 Nantes Cedex 3, FRANCE
e-mail	fred.rivard -at- is2t -dot- com

6.3 Acronyms and Abbreviations

API: Application Programming Interface

CDC: Connected Device Configuration

CLDC: Connected Limited Device Configuration

EE: Execution Environment

JVM: Java Virtual Machine

MMU: Memory Management Unit

OS: Operating System

RFP: Request For Proposal

RTOS: Reat-Time Operating System

6.4 End of Document