RFP 198—Managing External Code

Draft

10 Pages

Abstract

The purpose of this RFP is to enable the management of external code or processes by an OSGi framework. The goal is to bring the security verification, versioning and version management, and life cycle management of OSGi bundles to code that runs outside the Java Virtual Machine. Target code examples include FPGA bitstream code, GPGPU code, unmanage language code running in a separate process, Android apps, and firmware for external devices. The standard should not just be how they are managed, downloaded, verified, started, and stop, but also how bundles can communicate with such software.

0 Document Information

Table of Contents

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 .

`Source code is shown in this typeface.`

Revision History

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

Revision	Date	Comments
Initial	Jan. 24, 2020	Initial versions from aicas GmbH.

1 Introduction

OSGi has a well-defined life cycle and version management model for software running in an OSGi container. This works well for general purpose software, but cannot be used for programming external resources such as GPGPU processors and FPGA. It can also not be used for code written in unmanaged languages such as C and C++ without compromising the integrity of the framework. Since a pointer can be generated to any memory location in such a language, the isolation offered by OSGi and the Java Virtual Machine can be circumvented intentionally or unintentionally through program error. Still, to be able to manage such code via an OSGi framework could provide the version and life-cycle management to code outside the framework.

External code could be provided to an OSGi framework using the same signed JAR mechanism that standard OSGi bundles use. One must only extend the metadata for the bundle to differentiate external bundle from standard ones and provide enough information that the system can manage such a bundle. Thus version and permission information could be handled consistently.

Of course, an external bundle will require additional OSGi services to communicate with standard bundles, manage the life cycle of each bundle (load, resolve, start, stop), and manage resources. External bundles will be more dependent on the OS environment, but a Java runtime with RTSJ extensions provides several interfaces that could address these issues. Standard communication mechanisms can be provided via shared memory, memory ports, and sockets. Each type of resource, and possibly each implementation, will need one or more adapters for the OSGi framework to manage it.

A successful standard would extend the applicability of OSGi to a wider community of software developers and applications. Programs developed outside of the Java ecosystem could benefit from the advances in life cycle, security, and version management that OSGi has developed over the past two decades. It could also resolve some of the dissonance between OSGi and techniques such as DOCKER containers.

2 Application Domain

OSGi is used in a wide variety of applications from enterprise systems down to embedded systems. Extending OSGi to manage external code is applicable for all the same domains. Any system that contains a GPGPU or an FPGA should reap the advantages of this extension. Code written in unmanaged languages should also be usable in any of these domains without compromising the integrity of the OSGi framework.

2.1 What is an external bundle?

An external bundle is any code that is pack in a JAR file as a bundle but runs outside of the Java virtual machine, either in the same address space or in another process, but is still managed by an OSGi framework.

2.2 What are important external bundle types?

The most important types of external bundles are those that extend the OSGi programming model beyond the CPU. These include programs for GPGPU, FPGA, and DSP hardware. Other types are code that runs in an external process, container, or virtual machine.

2.3 What communication mechanisms could be used?

The communication mechanisms available depend on where the external bundle runs and whether or not common memory is available. A GPGPU, FPGA, or DSP might use shared memory to communicate via RTSJ raw memory, NIO buffers, or DMA. On the other hand, a process in a container might only be reachable via a network connection (socket). On might even need to use a device driver to load code onto an external device.

2.4 Terminology and abbreviations

Application
A set of bundles needed to render a full application to the user.

External Bundle
An entity that looks like an OSGi bundle, but runs outside the OSGi framework.

NIO Buffer
A Java class for accessing memory outside the JVM

RawMemory
An RTSJ interface for communicating with a device.

Managed Code
Code written in a language, such as Java, that requires exact garbage collection. A managed language environment can give isolation guarantees that an unmanage language,

such as C or C++, cannot give. For example, there is no way a bundle can read or write an object to which it does not have reference. In an unmanaged language, this can be do intentionally by creating a pointer or unintentionally by freeing and reallocating an object that is still reachable or running of the end of an array.

JNI
The so called Java Native Interface that enables one to call code outside the JVM. This is often used to call code written in an unmanaged language, but could also be used to call into another langauge VM running in the same address space.

3 Problem Description

For an OSGi framework to be able to manage external code, the OSGi framework must know a bit about each type of code that it can manage. It must be able to at least install the code, start the code, and stop the code. Additionally, it should also know whether or not each of its dependencies has already been resolved and started. Ideally, it would be able to communicate with the code and provide a means to call back into the Framework for registering itself and obtaining other services in accordance with its security profile. This may mean that each external code bundle must also have an internal stub to managing that communication.

3.1 Base Mechanism

Each type of external bundle must have a service for installing, resolving, starting, and stopping an external bundle of that type. This mechanism will probably needs to be system dependent, as different operating systems and external bundle types may provide different ways of managing programs. There also needs to be some sort of internal handle for bundle management and resolution.

3.2 OSGi-Aware Applications

OSGi-aware external bundles may also need a library to access OSGi services, such as the service registry and to communicate with other bundles.

4 Use Cases

The types of external bundles that could be supplied by OSGi is probably unlimited. The more fine grained the types are, the more support can be provided, but at the cost of a more complex interface. There are a small number of obvious candidates with which one could begin.

4.1 FPGA

Managing the bitstream of an FPGA is an important use case. The code for an FPGA is radically different than normal program code. It would be quite difficult to timeshare an FPGA as one does with a conventional CPU. The code is not a program in the normal sense, rather a description of how the hardware in an FPGA should be connected. An FPGA can be dynamically loaded and reloaded, but it is then a dedicated resource. An FPGA would not consume OSGi resources, but might have a stub bundle that could. It would require a dedicated, bitstream dependent, stub for communication. Data is usually transferred over memory addresses that act as “pins” or DMA. RTSJ facilities can be used to map these into the JVM for use by other bundles. So called native code should not be used to avoid dynamic unlinking problems with the OSGi framework when the FPGA program changes.

4.2 GPGPU or DSP

GPGPU and DSP programming is similar to FPGA programming except that the interface to the GPGPU or DSP can remain the same even when the bundle running on the hardware changes. Both RTSJ and NIO mechanisms could be used for communication. NIO is particularly good for shared memory communication.

4.3 External Process

The simplest example of this would be the management of an external OSGi framework. The second framework already knows how to talk to other OSGi frameworks. Only a service for starting and stopping it would be necessary.

However, the external process need not be another OSGi framework. It could be a single service. Here, a library for talking to an OSGi framework may be necessary, along with something for communication. Naturally, any standard communication protocol could be used.

This would be a better way to manage unmanaged code, e.g. C and C++ code, then using JNI, since running the code in an external process protects the integrity of the OSGi framework.

4.4 Container or VM

A container of VM is just a more isolated version of an external process. Communication with the container of VM is more limited, since some mechanisms, such as shared memory could not be used. How such as systems is started and stopped may also be different.

4.5 External Device

Embedded systems often have more than one processor, not all of which are powerful enough to run a full OSGi system, or even a Java program. A good example is an automobile, which may have a few strong processors and many small ones that need to be managed as well. The strong processors include the infotainment system, the telematics gateway, and domain controllers. These are generally 32bit or 64bit systems with large memories. Other processor are for dedicated functions, such as the airbag controller, or the brake controller. These are often much smaller and run a single application.

The assumption is that this external bundle type is for managing such small processors. Here the dedicated program is completely replaced at each update. That means that program load is generally a complete reflashing of the device. Communication is most likely over a field bus, such as CAN. Resolution is against programs running on other small processors.

Here the OSGi framework need to be able to flash these external devices in addition to resolving dependencies, starting them, and stopping them.

5 Requirements

R1: The solution MUST be able to identify an external bundle and its type.

R2: The solution MUST be able to track the version and dependencies of each external bundle.

R3: The solution MUST be able to control the life cycle of an external bundle including

- loading an external bundle into its correct environment,
- resolving its dependencies,
- starting it, and
- stopping it.

R5: The solution MUST be able to subject the external bundle to resource constraints.

R6: The solution MUST be able to manage the access of the external bundle to OSGi services and other external bundles.

R7: The solution MUST be able to reuse all resources used by the external bundle once the external bundle is stopped.

R8: The solution SHOULD be as platform independent as possible.

6 Document Support

References

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

[2]. Software Requirements & Specifications. Michael Jackson. ISBN 0-201-87712-0

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

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

[5]. *The Realtime Specification for Java 2.0., JSR 282, http://www.aicas.com/cms/rtsj*

Author’s Address

Name	James J. Hunt
Company	aicas GmbH
Address	Emmy-Noethe-Straße 9, 76131 Karlsruhe, Germany
Voice	+49 721 663 968-22
E-mail	jjh@aicas.com

End of Document