RFC0207 - Native OSGi

Draft

Pages

Text in Red is here to help you. Delete it when you have followed the instructions. E
The <RFC Title> can be set from the File>Properties:User Defined menu. To update it onscreen, press F9. To update all of the fields in the document Select All (CTRL-A), then hit F9. Set the release level by selecting one from: Draft, Final Draft, Release. The date is set automatically when the document is saved.

Abstract

10 point Arial Centered.

Put information about the purpose of the document and the information that it contains here. This text should not extend beyond this front page. If it does, revise the abstract.

1

Document Information

License

DISTRIBUTION AND FEEDBACK LICENSE, Version 2.0

Trademarks

Feedback

This document can be downloaded from the OSGi Alliance design repository at https://github.com/osgi/design The public can provide feedback about this document by opening a bug at https://www.osgi.org/bugzilla/.

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 10.1.

`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	*Oct 18 2013*	*Initial* *Steffen Kächele, University of Ulm, steffen.kaechele@uni-ulm.de*
	Oct 30 2013	*bundle format and execution environments* *Steffen Kächele, University of Ulm, steffen.kaechele@uni-ulm.de*

1 Introduction

Introduce the RFC. Discuss the origins and status of the RFC and list any open items to do.

TODO

2 Application Domain

This section should be copied from the appropriate RFP(s). It is repeated here so it can be extended while the RFC authors learn more subtle details.

Native OSGi can be applied in any domain which has/wants to rely on native code. Common examples are:

- Embedded software vendors
- (Medical) Imaging solutions
- Sensor Networks

Since this RFP focuses on a C/C++ OSGi Core Specification, it is difficult to list the final application domain. A few usage examples are:

- A software vendor has a large distributed (embedded) software stack, and wants to be able to dynamically (re)configure a running system. For this they need a module based system which has the ability to replace a module at runtime.

• A software vendor has a mixed software base of Java and C/C++ software. The Java software already uses OSGi. To be able to leverage the same benefits, they want to have a similar solution for their native code.

Advantages

Compared to the traditional way of writing shared libraries, a Native OSGi system has the following advantages:

1. Service – oriented modularity concepts for native developers, benefiting from years of experience gathered within the Java OSGi community.
2. Dynamic updates and reconfigurability of native code in a standardized way.
3. Alternative to JNI for Java and native code interoperability.
4. Having a standard bundle format to package and ship native code.

Disadvantages

1. C/C++ lacks development tools when compared to Java, which could make the usage of OSGi concepts difficult for the average programmer.
2. The lack of language features like reflection will limit the scope of a Native OSGi specification.

Terminology + Abbreviations

ABI: Application Binary Interface

Native OSGi: Working title for a C/C++ OSGi specification.

N-OSGi: Shorthand for Native OSGi.

Shared Library: This document consistently uses the term "shared library" for code loadable at runtime. On Windows, they are called Dynamic Link Libraries (DLL), on UNIX systems Dynamic Shared Objects (DSO), and on Mac OS platforms Dynamic Libraries (DyLib).

Loader: A platform-specific program responsible for loading shared libraries into memory and resolving their dependencies.

Platform: *A combination of processor architecture, operating system, shared library format, and compiler.*

3 Problem Description

This section should be copied from the appropriate RFP(s). It is repeated here so it can be extended while the RFC authors learn more subtle details.

The Native OSGi specifications are assumed to be mostly written in a platform-independent way. They should refer to the supported C and C++ language standard, with the exception of specifying the resolving process. The resolving process will likely need to take platform-specific and/or object file format features into account.

C and C++ Language Standard

As every language, the C and C++ languages evolve over time. A Native OSGi specification requires a definition of a language standard. Due to the large amount of different compilers and platforms, a suitable language version is required to not restrict the usage of Native OSGi too much.

Supported Platforms

A platform for native systems typically is a combination of processor architecture, operating system, and compiler. While the JVM provides a standardized runtime platform for OSGi, this will not be the case for Native OSGi. Native OSGi requires a minimum set of platforms on which compliant implementations are required to run, similar to the concept of Java *execution environments*.

Memory Management

Java relies on a garbage collector to reclaim memory from unused objects whereas C/C++ does not impose any kind of memory management system. Especially with regard to service objects, memory management of these objects is an important issue. It is also necessary to specify the life-time and ownership of all objects specified in Native OSGi.

Packaging and bundle format

OSGi ships bundles in the JAR format. However, JAR is explicitly bound to Java. Native OSGi has to specify a comparable ZIP-like bundle format.

As bundles contain native code, they are bound to a specific platform. Native OSGi bundles thus require meta data describing the platform architecture and needed runtime environment. Platform-independent bundles require source code. Native OSGi has to specify how to build such bundles.

In the native world, libraries (e.g. ELF format on Linux, Mach-O format on MacOSX, and PE format on Windows) are a standard way to split applications. Yet, there are multiple ways to package code into libraries. Libraries could represent a complete bundle, single packages (a logical group of classes within a bundle) or individual classes. Using libraries to represent packages in a bundle has a couple of advantages:

- Have multiple libraries per bundle
- Have exported and private libraries

Just like Java Packages this makes it possible to make a distinction between the private implementation and the exported services.

- Allow code sharing

Using exported libraries it is possible to share code between bundles.

Another issue is the search path of libraries. A Native OSGi solution has to set up the search path in a way that it can fully control wiring of libraries and bundles.

Dependencies and versioning

The Module Layer of the OSGi Core specification defines several dynamical code loading features, but relies on Java classloading mechanisms. The NOStrum project : Referenz nicht gefunden already presents solutions for many native specific issues. native OSGi has to take care of loading and wiring the different libraries by either relying on platform specific mechanisms ior by providing a custom mechanism on all supported platforms. The details of either approach must be invisible to the user and the wiring process must lead to the same results on all supported platforms.

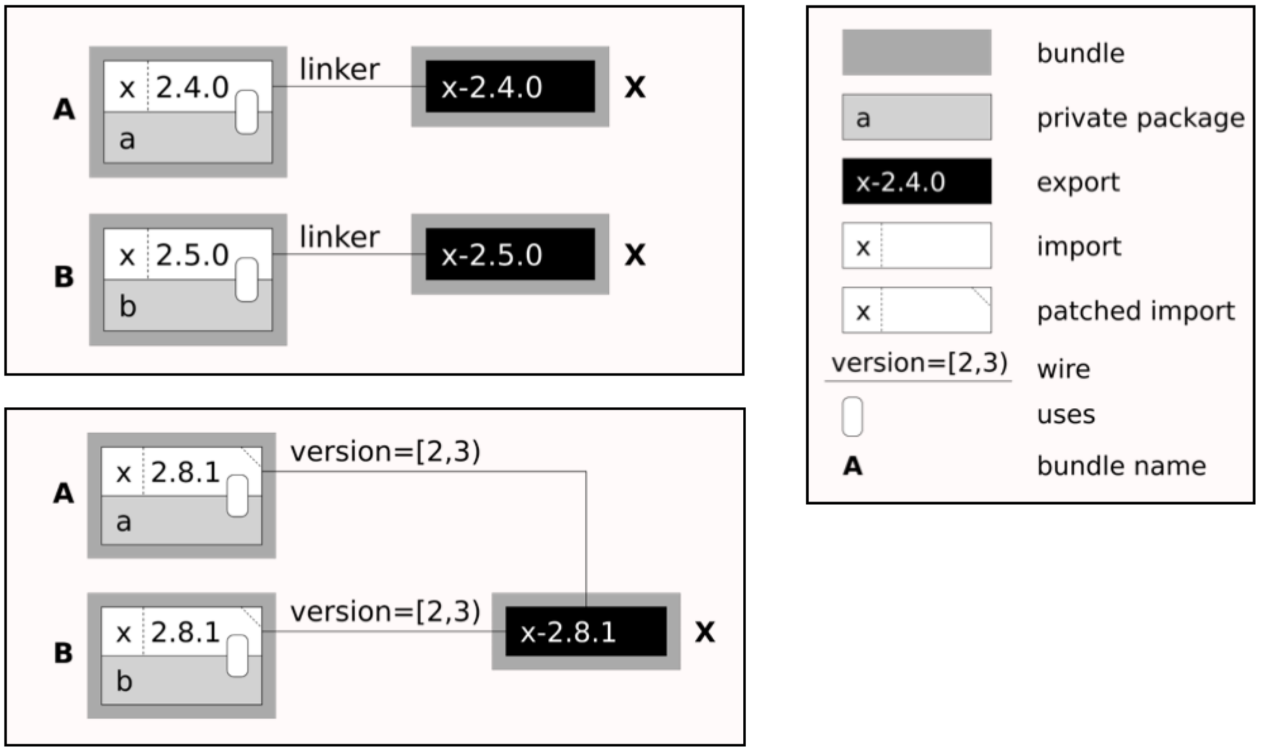
An important aspect of OSGi is versioning of packages, and the wiring needed to find the correct version of a dependency. Whereas Java by itself does not provide any default versioning mechanism, the different native linkers do. These versioning mechanisms usually allow to load and access code libraries in different version at the same time. Unix differentiates between a library's name (the SONAME) and the library file name on disk. By convention, the SONAME of a library includes a numerical major version number and at runtime, the dynamic loader resolves such names encoded in depending libraries by choosing a library with a matching file name and optional minor version number appended to it (if multiple such files exist, the one with the highest minor version number is loaded). Libraries which differ only in their minor version are assumed to be fully backwards compatible. MacOS mach-o file format also provides version information comparable to Unix. The windows Portable Executable format also has version information. Yet, it is not used during loading the DLL.

Native-OSGi should use an abstract version scheme that is compatible with popular platforms and allows libraries (packages) to be dynamically wired together at runtime, It must also support multiple library versions and dynamic bundle updates.

Exports And Imports

The following diagram depicts the wiring process for ELF shared libraries in NOStrum, where the full library version is additionally encoded in the library's file name . In the upper diagram the situation at link-time is detailed. The library dependencies are hard-coded by the linker during the linking process, using the SONAME of the dependencies (which includes the full version number). This results in a situation where Library A depends on version 2.4.0 of Library X, while Library B, developed with a different set-up, depends on version 2.5.0 of Library X.

If in the Meta-Data of the bundle providing library A or B the version range [2, 3) is provided, the module layer (resolver) looks for any bundles which fit in this range. Since the linker already has embedded the version info in the library itself (using the SONAME), this information has to be updated. So the version of the providing bundle/library (found by the resolver) is used to patch the version required by library A and B.



Alternative

Even though the model depicted above works, it might be worthwhile to investigate different solutions. A possible alternative is to extend the used linker (or linkers) to solve this. This would mean that instead of linking to a fixed version, the version range can be used. In this case there is no need to update the libraries at runtime. However, this might introduce restrictions to the target environment, e.g. a proper linker has to be supplied for every operating system, runtime environment and possible versions. This solution would also exclude Windows from the list of supported platforms, since the Windows dynamic loader is part of the Kernel and cannot be modified or substituted (see : Referenz nicht gefunden).

A much more complex solution might be the usage of one dedicated linker on all platforms. This linker can be bundled together with the framework which means the platform linker is ignored. Benefits of this are:

- No different library format, which means no different solutions (bundles or anything else) for different platforms.
- Only need for one common solution for wiring.

However, this would again exclude Windows from the list of supported platforms.

Security Layer

Isolation in native code is not a trivial task. There is no integrated security manager that can limit access to resources for parts of application code. Using pointers memory can be arbitrarily modified.

ABI Compatibility

C++ (in contrast to C) does not specify an ABI standard. In a Native OSGi setup, interoperability will therefore be reduced when using C++ compilers of different vendors or same vendors but different versions, which usually produce libraries with different ABIs. A careful library API design and refraining from using the STL or other third-party C++ libraries may in theory yield a cross-compiler compatible ABI. However, the restrictions on the API would severely limit its ease of use. As a consequence, it is expected that in a Native OSGi system only C++ bundles that were compiled with the same compiler (same vendor and compiler version) may share code via their C++ API. Hence bundles will need to be compiled from source for a specific Native OSGi target platform or provide appropriate meta-data when being published as pre-compiled binaries. Using an intermediate binary format such as LLVM may allow to mix bundles that were compiled with different compilers (see : Referenz nicht gefunden).

Compatibility of C and C++ Bundles

Although C is a subset of C++, Native OSGi should provide a native C++ API for bundles written in C++ (In addition to a pure C API). Code sharing between a C and a C++ bundle will only be straight forward for a C++ bundle depending on a C bundle. It is out of the scope of Native OSGi to specify a mechanism for sharing code of a C++ bundle with C bundles. However, bundles collaborating via the service layer should not need to be aware of the programming language (C or C++) the providing or consuming bundle is written in. Thus, Native OSGi has to specify a way to call a C service via a C++ interface and vice versa.

Using a mixed set of C and C++ bundles requires Native OSGi to provide both a C and C++ API. Bundle writers providing service interfaces also need to provide C and C++ versions of these interfaces for maximum interoperability. While the creation of a C service interface wrapper for a C++ service interface or vice versa can be supported by tools from Native OSGi and need only be done for the interface and not for each service implementation, such wrappers should remain optional.

4 Requirements

This section should be copied from the appropriate RFP(s)

OSGi Compatibility

Module Layer

OSGI-1 Native OSGi MUST detail how the Module Layer should be supported. This support SHOULD be based on the module layer described in a chosen specification [5] with at least r4.

OSGI-2 Native OSGi MUST use a well defined, ZIP-like platform independent format for bundles similar to the format described in a chosen specification : Referenz nicht gefunden with at least r4.

OSGI-3 As the bundle format will include specific native metadata that is as closest as possible to Java OSGi : Referenz nicht gefunden.

OSGI-4 The module layer in Native OSGi MUST support multiple libraries even of the same name but in different versions.

OSGI-5 The module layer in Native OSGi MUST support export control of libraries through the use of export/import headers in the bundle manifest. It MUST support exported and private libraries.

OSGI-6 The module layer MUST support versioning using export versions and import version ranges.

Life Cycle Layer

OSGI-7 Native OSGi MUST follow the life cycle for bundles as defined in a specification : Referenz nicht gefunden with at least r4.

Service Layer

OSGI-8 Native OSGi MUST detail a service layer and registry similar/equal to the chosen specification. This support SHOULD be based on the service layer described in a specification : Referenz nicht gefunden with at least r4.

Compatibility of C and C++ Bundles

CPPC-1 Native OSGi MUST support C and C++. This support has to be in a language natural manner. C++ developers should be able to use C++ constructs, whereas C developers should be able to do the same using C constructs.

CPPC-2 Native OSGi SHOULD support consumption of C services from C++ bundles and vice versa. Services SHOULD be handles in a transparent way without any additional work on the consuming end.

CPPC-3 The Native OSGi service registry SHOULD be able to handle C as well as C++ services.

Native language issues

LANG-1 Native OSGi MUST define a language standard. Due to the large amount of different compilers and platforms, a suitable language version must be chosen to not restrict the usage of Native OSGi too much.

LANG-2 The Native OSGi specification MUST state a minimum set of platforms on which compliant implementations are required to run, similar to the concept of Java *execution environments*.

LANG-3 Native OSGi MUST NOT rely on functions of specific operating systems NOR alter the standard system environment (e.g. standard runtime libraries).

LANG-4 Native OSGi MUST address memory management of all object specified in Native OSGi, in particular with regard to service objects. This covers both, the life-time and ownership of objects.

LANG-5 Native OSGi SHOULD introduce no or little overhead when issuing calls between bundles. Where possible wrappers must be avoided. The case of C – C++ interaction will be an exception to this requirement.

LANG-6 Native OSGi MUST define how to build platform independent bundles (i.e. source bundles).

LANG-7 The Module Layer in Native OSGi MUST take care of loading and wiring the different libraries.

LANG-8 Native OSGi SHOULD use shared libraries on the level of Java Packages.

LANG-9 Native OSGi SHOULD use a version scheme that is compatible with library version schemes of popular platforms.

5 Technical Solution

First give an architectural overview of the solution so the reader is gently introduced in the solution (Javadoc is not considered gently). What are the different modules? How do the modules relate? How do they interact? Where do they come from? This section should contain a class diagram. Then describe the different modules in detail. This should contain descriptions, Java code, UML class diagrams, state diagrams and interaction diagrams. This section should be sufficient to implement the solution assuming a skilled person.

Strictly use the terminology a defined in the Problem Context.

On each level, list the limitations of the solutions and any rationales for design decisions. Almost every decision is a trade off so explain what those trade offs are and why a specific trade off is made.

Address what security mechanisms are implemented and how they should be used.

Bundle format

The Core Specification defines a unit of modularization called a *bundle*. In Native OSGi a bundle is comprised of native code (i.e. libraries) and other resources. Bundles can share Libraries among an export bundle and an importer bundle in a well-defined way. In the Native OSGi bundles including libraries are the only entities for deploying native applications.

A native bundle is deployed as a ZIP file. These files include both the application and its resources.

Bundle Manifest Headers

A native bundle can carry description information about itself in the manifest file that is contained in its ZIP file under the name `MANIFEST/NATIVEMANIFEST.MF`.

A Framework implementation must:

- Process the main section of the manifest. Individual sections of the manifest are only used during bundle signature verification.
- Ignore unrecognized manifest headers. The bundle developer can define additional manifest headers as needed.
- Ignore unknown attributes and directives. All specified manifest headers are listed in the following sections. All headers are optional, unless specifically indicated.

Bundle-Activator: com.acme.fw.Activator
The Bundle-Activator header specifies the name of the class used to start and stop the bundle. See Starting Bundles in the Core Specification.

Bundle-Name: Firewall

The Bundle-Name header defines a readable name for this bundle. This should be a short, human-readable name that can contain spaces.

Bundle-NativeCodeFormat: osname = Linux; osversion = 3.1;
Bundle-NativeCodeFormat header contains a specification of native code libraries contained in this bundle.

Bundle loader

Wiring

Code sharing

Execution environments

The following attributes are architected:

- `osname` – Name of the operating system. The value of this attribute must be the name of the operating system upon which the native libraries run. A number of canonical names are defined in *Environment Properties* in the Core Specification.
- `osversion` – The operating system version. The value of this attribute must be a version range as defined in *Version Ranges* in the core specification.
- `processor` – The processor architecture. The value of this attribute must be the name of the processor architecture upon which the native code runs. see Environment Properties in the Core Specification.
- `compiler` – The compiler. This attribute must be the name of the compiler which was used to create the library.
- `selection-filter` – A selection filter. The value of this attribute must be a filter expression that indicates if the native code clause should be selected or not.

The following is a typical example of a native code declaration in a bundle's manifest:

```
Bundle-NativeCode:
            osname = Linux;
            processor = mips;
            compiler = gcc4;

Bundle-NativeCode:
            osname = Solaris;
            osname =
            processor = mips;
            compiler = gcc4;
```

6 Data Transfer Objects

RFC 185 defines Data Transfer Objects as a generic means for management solutions to interact with runtime entities in an OSGi Framework. DTOs provides a common, easily serializable representation of the technology.

For all new functionality added to the OSGi Framework the question should be asked: would this feature benefit from a DTO? The expectation is that in most cases it would.

The DTOs for the design in this RFC should be described here and if there are no DTOs being defined an explanation should be given explaining why this is not applicable in this case.

This section is optional and could also be provided in a separate RFC.

7 Javadoc

Please include Javadoc of any new APIs here, once the design has matured. Instructions on how to export Javadoc for inclusion in the RFC can be found here: https://www.osgi.org/members/RFC/Javadoc

C++ API

C API

8 Considered Alternatives

For posterity, record the design alternatives that were considered but rejected along with the reason for rejection. This is especially important for external/earlier solutions that were deemed not applicable.

9 Security Considerations

Description of all known vulnerabilities this may either introduce or address as well as scenarios of how the weaknesses could be circumvented.

10 Document Support

References

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

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

Add references simply by adding new items. You can then cross-refer to them by chosing <Insert><Cross Reference><Numbered Item> and then selecting the paragraph. ***STATIC REFERENCES (I.E. BODGED) ARE NOT***

ACCEPTABLE, SOMEONE WILL HAVE TO UPDATE THEM LATER, SO DO IT PROPERLY NOW.

Author’s Address

Name	Steffen Kächele
Company	University of Ulm
Address	Albert-Einstein-Allee 11 89081 Ulm
Voice	
e-mail	steffen.kaechele@uni-ulm.de

Acronyms and Abbreviations

End of Document

Discussion at Virtual Conference

- usual RFC structure?
- new infrastructure for specification production
  - API Code
  - reference implementation
  - compliance tests