RFC 153 - Classloading and Marshalling

Draft

17 Pages

Abstract

This document primarily addresses the problem of marshalling Java objects to and from external formats such as those based on XML, OO-RDBMS mappings or Java's built in object serialization mechanism. In particular it deals with the problem of locating and providing the classes and other resources required by marshalling processes.

0 Document Information

0.1 Table of Contents

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 8.1.

`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	12/08/09	Initial document from RFP 103 David Savage, Paremus Ltd, david.savage@paremus.com
0.1	15/03/10	Added discussion of technical solution and considered alternatives
0.2	17/05/10	Updated based on notes from previous F2F meeting
~~Revision~~	~~Date~~	~~Comments~~
~~Initial~~	~~12/08/09~~	~~Initial document from RFP 103~~ ~~David Savage, Paremus Ltd, david.savage@paremus.com~~
~~0.1~~	~~15/03/10~~	~~Added discussion of technical solution and considered alternatives~~

1 Introduction

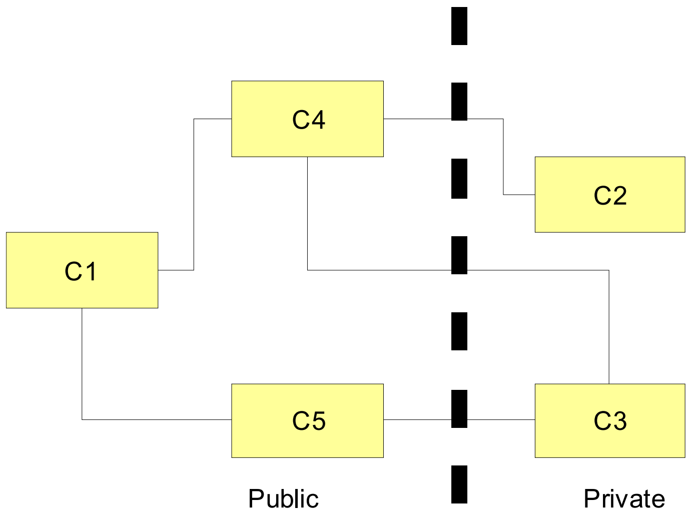
Introduce the RFC. Discuss the origins and status of the RFC and list any open items to do.

TODO

2 Application Domain

2.1 Marshalling/Unmarshalling

Marshalling is the term needed when a java object graph must be stored or communicated so that later, potentially in another process, it can be *unmarshalled* back into an object graph. Together this process is called *marshalling process*. The marshalling process is quite straightforward from a class loading point of view because the exact classes of the objects are known at that time. However, during unmarshalling the marshaled data only refers to a class *name*. This means that the unmarshaller must convert this name into a class again. The unmarshaller must load the class dynamically. This requires visibility not only of the root class of the graph structure, but to every class reachable in the graph, the so called *deep type structure*.



Deep Type Structure

Marshalling exposes classes that are otherwise private to the code implementations. The exposure of implementation classes is not arbitrary, but is determined using marshalling process specific configuration. For example, when using OO-RDBMS mapper based marshalling the publicly exposed types are typically specified in some configuration file. On the other hand, when using Java serialization the publicly exposed types are those marked as Serializable. To address the visibility problem, many subsystems in Java use the *context loader* (defined later) or require the parametrization of the unmarshaller with class loaders.

It is very often that case that the versions of types used during unmarshalling differ from those used during marshalling. This occurs for example when a client and server communicating remotely are upgraded independently, or when the types to which long lived data in an RDBMS are mapped are updated. This makes it important to understand which versions of the types are compatible with respect to marshalling.

In order for two versions of a bundle or package to be compatible with respect to marshalling, the types they contain must be compatible, as must the deep type structure of the types they refer to. One issue that arises here is how to deal with ambiguous situations, in which more than one version of a bundle is able to unmarshal a particular object. This is further complicated when object graphs contain objects that have the same class but different versions.

For different marshalling/unmarshalling mechanisms the types required become known at different times. For examples at opposite ends of the spectrum, consider:

1. An Object-Relational mapping framework that registers all types and mappings in advance of any marshalling using a context class loader.

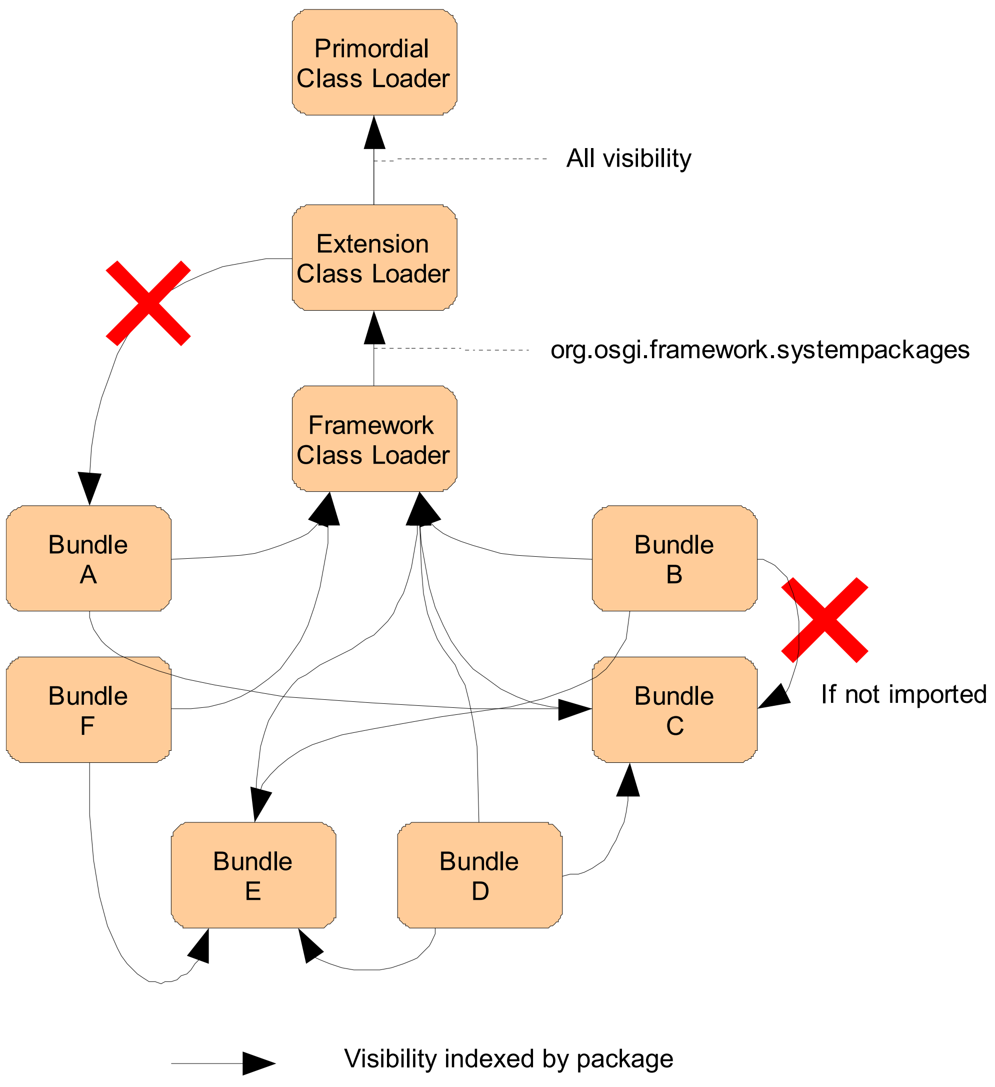
2. An RMI invocation for which type information only becomes available during the unmarshalling process as objects' codebase annotations are read.

If required types cannot be known in advance then the bundles providing the types cannot be installed in advance. In such situations bundles may have to be dynamically installed and resolved as the need for them becomes known.

The amount of type information available from a given marshalling process can also vary. Some marshalling techniques only use class names, leaving versioning concerns to be dealt with out of band, e.g. by an operator, whereas others can store arbitrary information, enabling greater automation of the marshalling process.

2.2 OSGi Class Loading

The OSGi Framework used the power of class loaders to provide a dynamic modularization system based on *bundles*. Bundles provide the source for a *Bundle class loader*. These class loaders have a much more formal approach to class loading. Each bundle must declare its imports and exports in the manifest, strictly reducing its visibility to other bundles. This restriction allows the OSGi framework to *wire up* the bundles in different ways, depending on their constraints. Wires are created on a package basis. That is, a package is a first class concept in OSGi. When a class load is done through a bundle class loader, the loader will first extract the package and will know from the package the exporter, or if there is no exporter, that it has to consult its own sources. In effect, this model creates a network of class loaders that are wired via their imported and exported packages. To minimize class loading issues, the OSGi used the service model to extend applications. This model allows the provider of the extension to do the service object creation instead of the consumer. This is a form of Inversion of Control that is considered good practice today.



OSGi Network Class Loader model

The reduced visibility of classes in OSGi is a good thing; however, there are many issues with *legacy code*, code not specifically written for OSGi.

In OSGi, public code gets its identity from its package name and package version number, and private code gets its identity from its bundle symbolic name and bundle version. Type compatibility in OSGi relies in the integrity of these identities being maintained. A particular version of a package can be exported by more than one bundle. The framework ensures class compatibility by selecting only one of the bundles as the package's provider for any given importing bundle. On the other hand, only one copy of each version of a bundle can be installed into the framework at a time, so the bundle's private code only ever has a single provider.

An important consequence of this is that when bundles are dynamically generated from other bundles, they can only guarantee compatibility of the public packages they export. Their private code's identity changes as soon as it is put into a new bundle, and is therefore guaranteed *not* to have type compatibility with the private code from the original bundle. This must be taken into account for code marshalling strategies that dynamically install bundles because it means that to get type compatibility with private classes the original bundles need to be used, not dynamically generated bundles that contain some of the original bundles' classes.

A bundle's class loader is created after the bundle is resolved and before the first class load. On the other hand, neither uninstalling a bundle, nor a refresh after it has been uninstalled can guarantee that its class loader is no longer available. This is because other bundles may hold strong references to instances of types defined by the defunct bundle.

3 Problem Description

OSGi bundles can share code via package imports and exports. If a bundle has a direct static reference to a shared type then it must import the package that provides the type. It is not necessary to import the package containing parent or member classes of directly referenced types unless these are themselves directly referenced. This approach ensures that all types referenced by a bundle can be seen from its class space. It does not make any guarantees about the deep structure of those types. Indeed, because of version ranges on imports, package wiring is not fully determined until bundles are resolved. The deep structure of types in public packages depends on the way these packages are wired, so this structure can vary between different runtime instances of types, even when the package defining them has the same version. In summary, OSGi guarantees local type structure, at least up to version, but in general makes no guarantees about deep type structure.

Some marshalling processes involve dynamic installation and resolution of bundles as the need for particular types becomes known. In order for the resolving process to succeed additional bundles may have to be installed. Deducing and installing the correct bundles will require access to a bundle repository that understands bundle dependencies and package constraints. At present there is a partial solution in this area in the form of the OBR proposal [5]. although at present this does not address enforcement of package constraints. The requirements in this area will be dealt with in a separate RFP. When bundles are installed in this way, i.e. dynamically and without operator intervention, it is unreasonable to expect an operator to be responsible for uninstalling them, or even to know when uninstallation is appropriate. As a result, dynamically installed bundles may stay installed for an indefinite period of time, making them a potential resource leak. Some means of determining when bundles are no longer in use and automatically uninstalling them is needed.

Java Object Serialization is Java's built in marshalling mechanism. It is widely used, notably in RMI. Serialization using OSGi suffers from all of the problems of marshalling in general with OSGi. RMI has a solution to dynamic installation of code, and to removing it when it is no longer in use. However, this is not compatible with OSGi. There is a need for a mapping of the general OSGi approach to marshalling to Serialization and RMI. This will be dealt with in a separate RFP.

Various class loading and marshalling processes expose types that are not otherwise exported by bundles. Bundle authors may want to limit the types available to such processes. Bundle authors and operators may want to limit use of such types to bundles with appropriate permissions.

- No way to prevent unexported classes from being marshalled
- java.io.Serializable is enough SerialVersionUID is ultimate decider of class identity
- Serialization update have consequences on package versions as internal changes can now imply non backwards compatible API changes

- Algorithm for uid is not standardised between JVMs

4 Requirements

- M001 – The solution SHOULD provide an approach to ensuring deep compatibility of types used in marshalling processes.
- M002 – The solution SHOULD address the need to avoid ambiguous unmarshalling.
- M003 – A mapping of the solution to Java's built in object serialization mechanism SHOULD be provided.
- M004 – The solution MUST provide a way for a bundle's author to decide which of its unexported types and resources are available to class loading and marshalling processes.
- M005 – Permissions limiting access to bundles' private types and resources for class loading and marshalling purposes MUST be defined.
- M006 – The solution MUST address the class loading and marshalling needs of OSGi oblivious library code, and describe approaches available to integrators of such code.
- M007 – The solution MUST provide best practice approaches to class loading and marshalling that can be used by OSGI aware code.

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.

5.1 Class Identity

Objects created in an OSGi framework are served by a single class loader that is associated with the package that supplies the underlying class. At the time of marshaling an object the identity of the class of an object is known. However at unmarshaling time a *marshaling provider* must resolve the class loader that is able to realize the marshaled representation. There are several cases that need to be considered:

1. The the exact same bundle that marshaled the object is already installed in the framework
2. A different bundle is installed that is capable of providing a class that is compatible with the marshaling scheme
3. No bundles are installed that serve the marshaled object

In case 3 it is necessary to resolve a bundle that is capable of providing a realization of the marshaled object. Here there are three possible outcomes to this resolution:

4. It is possible to install the exact same version of the bundle in this framework.
5. It is possible to install a different but compatible bundle in this framework
6. No valid bundle resolution can be performed.

To provide the marshaling provider with the information to find the appropriate classloader within the framework the following set of identity characteristics are defined, Bundle-Identity – bid, Package-Identity – pid, Class-Identity – cid.

```
bid: <Bundle-SymbolicName>#<Bundle-Version>
pid: <Export-Package>#<Package-Version>
cid: <hash>
```

5.2 Fragment Bundles

Classes that are defined within a fragment bundle have an extra dependency on unmarshal as they require a resolved host bundle in order to resolve the fragment bundle.

Two options here:

It is possible to model this dependency as a require bundle semantic

We could introduce a secondary identity characteristic: Host-Identity – hid:

```
hid: <Bundle-SymbolicName>#<Bundle-Version>
```

5.3 Marshaled-Packages Header

A new header “Marshaled-Packages” is defined that enumerates the packages that a marshaling provider is allowed to marshal. The format of this header matches the Export-Package header:

```
Marshaled-Packages ::= export ( ',' export )*
export              ::= package-names ( ';' parameter ) *
package-names       ::= package-name ( ';' package-name ) *
```

The Marshaled directives are:

- include – a comma separated list of class names that must be visible to an unmarshaller
- exclude – a comma separated list of class names that must not be visible to an unmarshaller

The Marshaled package also supports the following attribute:

- version – the version of the named package with the syntax as defined in the OSGi the core OSGi specification.

If no “Marshaled-Packages” header is present then the default is to assume that all packages are available for marshaling to allow for backwards compatibility with bundles that do not specify this header.

The Marshaled-Packages header is defined by the bundle, but for consistency the Marshaled classes of a public package *should* be the same for each exporter of the package.

5.4 ~~Marshalled-Classes Header~~

~~A new header “Marshalled-Classes” is defined that enumerates the classes that a marshaling provider is allowed to marshal. The format of this header is a comma separated list~~

```
~~Marshalled-Classes ::= pattern ( ',' pattern ) *~~
~~pattern          ::= ('-') token ('*')~~
```

~~This header provides a comma separated list of class names that may be marshaled from this bundle. The '-' character indicates that this is a negative pattern. The '*' character indicates a wild card pattern. Examples of this might include:~~

```
~~# any classes from this bundle may be marshaled~~
~~Marshalled-Classes: *~~

~~# any classes in the com.example package and~~
~~# any nested package from this bundle may be marshaled~~
~~Marshalled-Classes: com.example.*~~

~~# Any class not from the com.example.sensitive package~~
~~# from this bundle may be marshaled~~
~~Marshalled-Classes: *, com.example.sensitive*~~
```

~~If no “Marshalled-Classes” header is present then the default value of '*' is implied to allow for backwards compatibility with bundles that do not specify this header.~~

~~The Marshaled-Classes header is defined by the bundle, but for consistency the Marshaled classes of a public package *should* be the same for each exporter of the package.~~

5.5 Marshaled Object Resolution

When resolving the bundle that supplies the class loader for a marshaled object it is necessary to resolve not just the top level bundle that supplies the class but also the transient dependencies of this bundle in order to allow the marshaled object to create objects at runtime that are not part of the marshaled representation.

As the Marshaled-Packag~~Class~~es header may not be the same between bundles the resolver must also check that should a bundle be found that supplies the class that it is available for marshaling purposes.

~~DS: Is the above check needed? Marshaled-Classes header is intended to prevent classes being marshaled not being unmarshaled?~~

The resolution should take account of environmental factors in the framework, this includes:

- Installed bundles – uses clashes
- Security constraints that would prevent the bundle serving classes

TODO fill out info on deep class compatibility and maximiz~~s~~ing reuse within the framework.

5.6 MarshalAdmin

To provide common marshaling related tasks a new service is defined that allows different marshaling providers to reuse common functionality within the framework:

```
package org.osgi.marshal;

public interface MarshalAdmin {
  /**
   * Calculate the marshal identity of the supplied class.
   * Returns null if this class may not be marshaled.
   */
  MarshalIdentity getMarshalIdentity(Class<?> clazz);

  /**
   * Finds the classloader that is able to unmarshal an object
   * with the specified marshal identity. Returns null if no
   * valid classloader can be found.
   */
  ClassLoader getClassLoader(MarshalIdenity identity, boolean resolve);
}

package org.osgi.marshal;

public class MarshalIdentity {
  public String getBundleIdentity() {
  }

  public String getPackageIdentity() {
  }

  public String getClassIdentity() {
  }

  public boolean isCompatible(MarshallIdentity identity) {
  }
}
```

If the resolve parameter on getClassLoader is set to true the MarshalAdmin should attempt to resolve a bundle that can unmarshal the specified MarshalIdentity using an external resolver such as OBR. When a bundle is resolved on demand the BundleAdmin should use new Bundle-Lifecycle features specified in 5.8 to ensure that the bundle may be removed when it is no longer required.

5.7 Class-Identity Algorithm

TODO

5.8 Bundle Lifecycle

When dealing with marshaled objects it is important to differentiate between bundles that have been installed explicitly by the user and those that have been resolved by the MarshalAdmin. Bundles resolved by the MarshalAdmin are only required as long as any objects created as a result of unmarshaling are referenced within the framework.

Once all hard references to these objects have been removed the object is available for garbage collection by the JVM. At this point there is no need to keep these resolved bundles installed in the framework. In order to prevent gradual leakage of resources in long lived frameworks it is necessary to remove these bundles.

In order to track unmarshaled objects within the framework it is possible to use java.lang.reflect.WeakReference to track when all hard references to the specified object are released marking the object available for garbage collection.

5.9 BundleGarbageCollector

The task of tracking and garbage collecting bundles that are no longer required in the framework is provided by a service that implements the BundleGarbageCollector interface:

```
package org.osgi.marshal;

public interface BundleGarbageCollector {
  /**
   * Attempt to uninstall any bundles with the specified symbolic name
   * from the framework.
   *
   * Returns the set of bundles that were uninstalled as a result
   * of this sweep.
   */
  Bundle[] sweep(String pattern);
}
```

The BundleGarbageCollector service should track bundle installations using the BundleTracker or equivalent mechanism. ~~defines three distinct states for Bundles installed within the framework:~~

The BundleGarbageCollector defines two distinct states for Bundles installed within the framework:

- ROOT
- LEAF

Bundles installed by the MarshalAdmin should be marked as LEAF, bundles installed by the user should be marked as ROOT.

When the sweep method is called the BundleGarbageCollector should find any LEAF bundles that are no providing objects within the framework. Once they are no longer providing objects the BundleGarbageCollector should check if any other ROOT or LEAF bundles are wired to this bundle. If there are no other wires to this bundle then it may then be uninstalled.

- ~~EPHEMERAL~~
- ~~TRANSIENT~~
- ~~PERMANENT~~
- ~~The BundleGarbageCollector service should track bundle installations using the BundleTracker or equivalent mechanism. When a bundle is installed by a user via the ordinary method BundleContext.installBundle(String location) the BundleGarbageCollector should mark the bundle as PERMANENT. Bundles installed by the MarshalAdmin should be marked as either EPHERMERAL or TRANSIENT depending on the following factors:~~
- ~~If a bundle is installed to directly provide a class loader for a marshaled object then it is marked as EPHEMERAL~~
- ~~If a bundle is installed to satisfy a transient dependency of an EPHEMERAL bundle then it should be marked as TRANSENT.~~
- ~~Bundles that are marked as TRANSIENT as part of one resolve request should remain TRANSIENT even if a secondary resolution would have marked them as EPHEMERAL.~~
- ~~The distinction between EPHEMERAL and TRANSIENT bundles is required to cope with the situation where an object is unmarshalled that requires supporting bundles to be available within the framework to provide classes but these classes are not immediately loaded within the framework.~~
- ~~When the sweep method is called the BundleGarbageCollector should find any EPHEMERAL bundles that are no longer providing objects within the framework. These may now be uninstalled from the framework. Once EPHEMERAL root bundles have been uninstalled the BundleGarbageCollector should then look for TRANSIENT bundles that are no longer connected to a root and uninstall these as well.~~
- Bundle ClassLoader Reference

Tracking every unmarshaled object is too expensive (see considered alternatives) but it is possible to calculate if a given bundle resolved by the MarshalAdmin is required by storing a WeakReference to the ClassLoader that created the object. The ClassLoader is strongly referenced by the Class which is in turn strongly referenced by Object. Once the Object is garbage collected this chain is available for garbage collection and can be polled from a java.lang.reflect.ReferenceQueue using the following code:

```
ReferenceQueue<ClassLoader> queue = new ReferenceQueue<ClassLoader>();

void mark(ClassLoader loader) {
 new WeakReference<ClassLoader>(loader, queue);
}

ClassLoader next() throws InterruptedException {
  Reference<? extends ClassLoader> l = queue.remove();
  return l.get();
}
```

For this scheme to work the framework implementation must also store a WeakReference to the bundle classloader to allow for garbage collection.

5.10 Backwards compatibility

Storing a weak reference to the bundle class loader is problematic for a number of use cases including:

- Classes that store state in static variables
- Bundles that use native code

In order to maintain backwards compatibility with applications that expect bundles to be installed for all time it must be possible to mark at install time whether the framework should keep a strong or a weak reference to the bundle class loader. This requires the addition of two new framework methods:

```
BundleContext.installBundle(String location, int state)
Bundle.uninstall(int state)
```

The value of the state parameter may be one of the following:

- Bundle.STRONG_REFERENCE
- Bundle.WEAK_REFEERENCE

The framework must hold a weak reference to the bundle classloader until such time as Bundle.uninstall(WEAK_REFERENCE) is called.

The framework must only keep a strong reference to the bundle class loader after a call to installBundle has made with a STRONG_REFERENCE flag. The STRONG_REFERENCE flag is set until such time as a call to Bundle.uninstall(STRONG_REFERENCE) is made. When the STRONG_REFERENCE flag is removed then the framework should null the strong reference to allow for garbage collection to take place.

Calling BundleContext.installBundle(String location) should implicitly create a STRONG_REFERENCE to the bundle. Calling Bundle.uninstall should result in a delegated call to uninstall(STRONG_REFERENCE).

DS: Should this be uninstall(WEAK_REFERENCE) – current proposal though backwards compatible from a functionality point of view implies that a bundle can be uninstalled even if it is in use in the framework – which may be a mistake??

TODO need to work through potential race conditions in this mechanism.

5.11 Non OSGi aware marshaling

TODO - Use cases that need thinking about:

- Unmarshaling objects from non OSGi aware schemas – i.e. no MarshalIdentity defined by sender
- Support for non osgi aware code – need to look at some usecases

6 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.

6.1 Proxy bundles

R-OSGi: It is possible to create bundles on the fly to provide a class loader for a marshaled object and uninstall this bundle when the bundle that brings the marshaled object into the JVM is removed. One downside to this approach is that objects may out live the bundles that originally unmarshaled them this can lead to a situation where the framework is an inconsistent state where objects are stored in data structures that are not backed by their proxy bundles.

DS: Note I've put this in considered alternatives for the time being, feel free to flesh out info on this approach and we can move up to solution etc...

6.2 Distributed Update

The HotSwap solution [6]. (Relleymeyer etal) provides a mechanism to perform an update of all classes in a distributed software environment. This is beyond the scope of this RFC and could still be layered on top of the approach laid out here. It is not clear that a distributed update is always desirable, as there may be cases where it is useful to be able to update classes in one framework without effecting other frameworks. Also this scheme is aimed primarilly at over the wire marshalling schemes such as remote method invocation or messaging systems and does not work so well in storage related scenarios.

DS: Again meant really as a placeholder feel free to flesh out this section and we can merge any desirable features into technical solution...

6.3 Object References

Storing/weak references to the Bundle class loader involves some invasive changes to the OSGi framework which could be avoided if the solution to this RFC tracked object references of marshaled classes instead of their class loader. Unfortunately in any reasonably sized application this would be a prohibitively expensive task which would effect user experience and this solution is rejected on these grounds.

6.4 Bundle Intents

An alternative to the install(String location, int state) would be to support an intent based scheme such as install(String location, String[] intents) or install(String[] location, Map config)/Bundle.INTENT_CONFIG this allows for clients to extend the data that is stored with the bundle installation beyond the fixed set of states that are defined by this spec and use this data for their own purposes. Rejected because...(to complex?) (not obvious what uninstall semantics should be?)

7 Security Considerations

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

Bundles installed as a result of implicit unmarshaling are installed without direct operator interaction. It should be possible to set up security policy via the ConditionalPermissionAdmin to limit the capability of these bundles.

New UnmarshaledCondition?

Implies creation of Bundle.getState() method to allow condition to test state of installed bundle.

8 Document Support

8.1 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]. RFP 83 Class Loading and Marshalling
https://www2.osgi.org/members/svn/documents/trunk/rfps/rfp-0083-Classloading-and-Marshalling.doc

[4]. Serialization issues in Eclipse
https://bugs.eclipse.org/bugs/show_bug.cgi?id=85116

[5]. RFC-0112 Bundle Repository.
http://www2.osgi.org/div/rfc-0112_BundleRepository.pdf

[6]. Consistently Applying Updates to Compositions of Distributed OSGi Modules
http://www.iks.inf.ethz.ch/publications/files/HotSWUp08.pdf

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.***

8.2 Author's Address

Name	David Savage
Company	Paremus Ltd
Address	107-111 Fleet Street, London, EC4A 2AB
Voice	+44 20 7993 8321
e-mail	david.savage@paremus.com

8.3 Acronyms and Abbreviations

Marshaling Provider – a class that wishes to marshal or unmarshal an object from/to the framework using an external data representation.

DS: Not an acronym or abbreviation ~~or abb~~ but wan't sure where to put glossary terms...

8.4 End of Document