# RFC 231: Remote Service Intents

Final

11 Pages

## Abstract

10 point Arial Centered.

OSGi Remote Services are a powerful, flexible way to expose OSGi services from one framework into another. The model allows great flexibility in the serialization, transport and wire formats that are used by the Remote Services implementation, meaning that Remote Services can be used across many sorts of distributed system. Unfortunately this flexibility offers few guarantees about the types of object that may be safely used in the Remote Service API. This RFC aims to provide some basic remote service intents that can be relied upon to provide support for simple synchronous and asynchronous remote services.

# 0 Document Information

## 0.1 License

## 0.2 Trademarks

## 0.3 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/.

## 0.4 Table of Contents

## 0.5 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.
```

## 0.6 Revision History

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

| Revision | Date | Comments |
|---|---|---|
| Initial | Oct 06 2016 | Initial version of the Remote Service Intents RFC. Tim Ward, Paremus |
| 0.1 | Nov 25, 2016 | Added ‘private’ intent from RFC 183. David Bosschaert, Adobe |
| 0.2 | January, 2018 | Mark as Final |

# 1 Introduction

OSGi Remote Services were introduced in the OSGi Enterprise Specification version 4.2. Remote Services are a powerful, flexible way to expose OSGi services from one framework into another. The Remote Services specification chapter is deliberately short of implementation details, which allows great flexibility in the serialization, transport and wire formats that are used by the Remote Services implementation.

Remote Services can therefore be used across many sorts of distributed system. Unfortunately this flexibility offers few guarantees about the types of object that may be safely used in the Remote Service API. Service intents offer a mechanism for services to express their distribution requirements, however there are no standard intents that can be used.

The requirements for this RFC relate to two separate RFP/RFC documents, both of which require guarantees from the Remote Services implementation. The first RFC is RFC 183, which makes use of Remote Services as a means of interacting with remote OSGi frameworks in a distributed system, and the second is RFP 184, which provides support for natively asynchronous calls.

The purpose of this RFC is to define intents that can be relied upon to provide support for these simple synchronous and asynchronous remote service operations.

# 2 Application Domain

Remotable OSGi services declare a remotable service interface to other OSGi frameworks in a similar way to local OSGi services declaring their interfaces to other bundles. For local services the Java Virtual machine and the OSGi framework ensure that the method parameters and return types are able to be passed safely.

For remote services the Java virtual machine and OSGi framework are not able to make the same guarantees. The serialization and deserialization process may, or may not, be able to correctly handle the remote method call.

In addition, remote invocations are inevitably based on some form of asynchronous communication, even if this is made synchronous by an intermediate network layer. For transports which are truly asynchronous there can be advantages when handling methods with asynchronous return types.

# 3 Problem Description

Remote Services are required to support primitive types and Strings as arguments and return types, however the OSGi specifications also define DTOs as an easily serializable data representation.

Furthermore, asynchronous methods may return an OSGi Promise, Future or CompletableFuture. For asynchronous transports these types should be able to be handled natively, and as long as the result of the asynchronous method call is easily serializable this should also be possible to handle.

In the case of an asynchronous service, a communications failure or partition may prevent a response from ever being received. Synchronous services block at this point, and asynchronous services simply fail to progress.

Most remote systems typically have a heartbeat and timeout to prevent the system from hanging. Remote services should have a timeout configuration parameter to prevent the system from hanging in a state which is difficult to diagnose.

# 4 Requirements

RSI 0010: The solution MUST define a way for remote services to use DTOs as a remote return type

RSI 0020: The solution MUST define a way for remote services to return asynchronous results using OSGi Promises, Java Futures and Java CompletableFutures. Implementations MAY support this mode of operation.

RDI 0030: The solution MUST define a way for remote services to set an invocation timeout.

# 5 Technical Solution

Remote Services is a long-standing OSGi specification which provides a generic set of rules for the provision of remote OSGi service invocations. Remote Services are declared using the “service.exported.interfaces” property on a registered OSGi service.

Selection of a Remote Service Provider can be performed in one of two ways:

- The service.exported.configs can be used to limit the remote service to providers that support a given configuration type

- The service.exported.configs can be used to limit the remote service to providers that support a given set of intents type

These two mechanisms may superficially seem identical, however intents and configuration types are very different. An intent is a generic expression of a requirement that the remote service has for how it is distributed. For example the requirement for a secure transport layer would be expressed as an intent. Importantly an intent makes no further statement about the low-level implementation of the Remote Services provider. A configuration type, however, is an expression of interoperation between the two parties in a remote service invocation, for example a configuration type might be declared for SOAP over HTTP web services. The configuration type guarantees sufficient information will be supplied so that different remote services implementations can interoperate with one another.

The result of this difference is that a configuration type is a highly restrictive and detailed restriction. It is therefore typically used to identify a single remote services implementation. Whilst this does permit a remote service to select a provider which supports the features it requires, it is almost impossible to provide an exhaustive list of providers that can be used.

Remote Service Intents are therefore the most appropriate mechanism to use when identifying broad feature support required from a Remote Services Provider. This RFC therefore proposes to define several intents for common use cases.

## 5.1 Synchronous Remote Services

Remote Services Implementations are required to support services that:

- Expose a single Java interface (not a class type)

- Use only the OSGi scalar types, or arrays/Lists of the OSGi scalar types as method parameters and return values.

This model allows a significant number of synchronous remote services to be exported, but it is also quite limited. The lack of structured data transfer object types (DTOs), for example, means that most OSGi specifications are not able to depend on remote services.

This RFC therefore proposes the “osgi.basic” intent, with the following minimum levels of support:

### 5.1.1 Supported Service Interface

- The remote service exposes only a single Java interface declaring zero or more methods. The interface may extend other Java interfaces.

- The methods’ parameters and return types are permitted to be Java primitives or the OSGi scalar types

- The methods’ parameters and return types are permitted to be arrays of Java primitives or the OSGi scalar types.

- The methods’ parameters and return types are permitted to be Lists of the OSGi scalar types, however the List implementation may not be the same on both sides of the remote call (for example a Linked List from the client may be an ArrayList on the server or vice versa)

- The methods parameters and return types of the interface are permitted to be Sets of the OSGi Scalar types, using equals() to determine identity. Iteration order may not be maintained between the client and the server. SortedSet need not be supported as a parameter or return due to the difficulties in serializing a Comparator.

- The methods’ parameters and return types are permitted to be Collections or Iterables of the OSGi scalar types, however the implementation may not be the same on both sides (for example a Linked List from the client may be a HashSet on the server)

- The methods parameters and return types of the interface are permitted to be Maps where the keys and values are OSGi Scalar types, using equals() to determine identity for the keys. Iteration order may not be maintained between the client and the server. SortedMap need not be supported as a parameter or return due to the difficulties in serializing a Comparator.

- Additionally, method parameters and return types that conform to the OSGi DTO rules are supported. DTOs may also be used in Arrays, Lists, Sets, Collections, Iterables, and as the values in Maps.

### 5.1.2 Configurable timeout behaviour

The implementation of a Remote Services provider is entirely opaque, it may use a synchronous request response protocol over TCP, an asychronous UDP based messaging provider, or even by writing to the file system. In many cases there will be no feedback feedback mechanism if the remote call hangs, or if the remote node fails, and so the local client will must decide to either block indefinitely, or to fail with an exception after a certain amount of time has elapsed. As it is rarely a good idea to block indefinitely most remote services providers will throw an Exception after the timeout has elapsed.

A remote services provider must be able to service a wide variety of different remote service invocations across many services, therefore it is difficult to identify a sensible timeout for the remote service invocation. Some calls may be quick, and so a ten second timeout is desirable for rapid failure detection, other calls may be long-running, and a two minute timeout too short. It therefore makes sense to allow the remote service to declare its own timeout.

To declare a timeout the remoteable service may provide a property “osgi.basic.timeout” which provides a timeout value in milliseconds. The value may be declared as a String or as a Number, which will be converted into a Long. The timeout value is used to limit the maximum time for which a remote service client will be blocked waiting for a response when calling methods on this service. In the event that the invocation reaches the timeout value the client will receive a ServiceException with its type set to REMOTE.

Note that due to the inherent complexities of distributed systems it is not known whether a timed out invocation is finished, still executing, or never started executing. It is up to the client application to ensure that failures of this kind are resolved appropriately.

## 5.2 Asynchronous Remote Services

Some service invocations operate asynchronously, returning quickly and continuing to process in the background. For void methods with no completion notifications this is simple to achieve remotely, but more useful scenarios are difficult to support without using higher-level abstractions to represent the eventual result.

This RFC therefore proposes the “osgi.async” intent as an extension of the “osgi.basic” intent, with the following additional minimum levels of support:

### 5.2.1 Supported Service Interface

The methods’ return types are permitted to be one of the following “asynchronous holders”

- java.util.concurrent.Future
- java.util.concurrent.CompletableFuture
- org.osgi.util.promise.Promise

If the return type is an asynchronous holder then the eventual value must either be

- A supported “osgi.basic” type
- The java.lang.Void type, which will always be null.

### 5.2.2 Timeout behaviour

Asynchronous calls may time out in exactly the same way as synchronous calls do. In this case the asynchronous holder is failed with the same exception as would be thrown in a synchronous call.

## 5.3 Private Remote Services

In many deployment scenarios, including cloud, hosts are accessible via a public network and via a private network. In this case the host will have multiple IP addresses where public and private receive different IP addresses. Private IP addresses normally in one of the following blocks: 10.0.0.0/8, 172.16.0.0/12 or 192.168.0.0/16.

In some cases it can be desirable to expose remote OSGi services only on the private network so that these services cannot be accesses from the outside world. This is especially useful if this service is used as a microservice within a larger application. This RFC proposes the “osgi.private” intent that can be specified that the remote service is only exposed via the private networks – not via a public network.

# 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

# 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

## 10.1 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

*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.***

## 10.2 Author’s Address

| | |
|---|---|
| Name | Tim Ward |
| Company | Paremus |
| Address | |
| Voice | |
| e-mail | Tim.ward@paremus.com |

## 10.3 Acronyms and Abbreviations

## 10.4 End of Document