BJ Hargrave 3/7/09 4:48 PM
Formatted: Top: (No border), Bottom: (No border), Left: (No border), Right: (No border)

RFC 119 - Distributed OSGi

Unknown
Deleted: -

Final

Tim Diekmann 2/20/09 1:42 PM
Deleted: Under Review

65 Pages

Tim Diekmann 4/10/09 5:29 PM
Deleted: 62

Abstract

This RFC contains a design that meets the requirements described in RFPs 79 and 88. The solution defines a minimal level of feature/function for distributed OSGi processing, including service discovery and access to and from external environments. This solution is not intended to preclude any other solution and is not intended as an alternative to Java EE, SCA, JBI, or any other external API set that may be mapped onto OSGi.

Tim Diekmann 2/20/09 1:42 PM
Deleted: Under Review

0 Document Information

0.1 Table of Contents

Philipp Konradi 4/7/09 4:10 PM
Formatted: French

Philipp Konradi 4/7/09 4:10 PM
Formatted: French

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 [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	Jul 17, 2007	Initial draft with information based on RFP 88 Eric Newcomer, IONA, eric.newcomer@iona.com David Bosschaert, IONA, davidb@iona.com
0.1	July 27, 2007	Added parts related to RFP 79 (service discovery) Tim Diekmann, Siemens Communications, tim.diekmann@siemens.com
0.2	August 1-8	Added introductory information, incorporated edits, and filled in sections from relevant RFPs.
0.3	September, 2007	Changes following feedback from the August Face-to-Face *Tim Diekmann, Siemens Communications,* tim.diekmann@siemens.com Eric Newcomer, IONA, eric.newcomer@iona.com David Bosschaert, IONA, davidb@iona.com
0.4	October, 2007	Tim Diekmann, Siemens Communications, Philipp Konradi, Siemens Corporate Technologies
0.5	November, 2007	Tim Diekmann, Siemens Communications, Philipp Konradi, Siemens Corporate Technologies Klaus Kunte, Siemens Enterprise Networks GmbH & Co KG
0.6	February, 2008	Changes as a result of discussions at January F2F Deleted comments that seemed resolved or discussed. Eric Newcomer
0.7	March, 2008	Tim Diekmann, Siemens Communications. Accepted all changes.
0.8	May, 2008	Eric Newcomer, accepted changes, incorporated text about intents, cleaned up comments following their resolution. Graham Charters & Philipp Konradi, qualified intents section.

Philipp Konradi 4/7/09 4:09 PM
Formatted: English (UK)

Philipp Konradi 4/7/09 4:09 PM
Formatted: English (UK)

Philipp Konradi 4/7/09 4:09 PM
Formatted: English (UK)

Revision	Date	Comments
0.9	June, July 2008	Eric Newcomer, further editorial cleanup .
0.9.1	July 2008	Tim Diekmann, minor editorial changes, bug 719
0.9.2	July 2008	Eric Newcomer, changes from July F2F & bug list
0.9.3	August 2008	David Bosschaert, changes relating to bugs 689, 729 and 735.
0.9.4	August 2008	Graham Charters, changes relating to bugs 606, 685, 755, 756, 759
0.9.5	August 2008	Viktor Ransmayr & Philipp Konradi, update interface description for Discovery Service
0.9.6	September 2008	Graham Charters, minor updates to SCA section based on 119 authors call.
0.9.7	September 2008	David Bosschaert: Changes as outlined in the resolution to bug 730.
0.9.8	September 2008	Graham Charters: changes for bugs 606 (configuration.type), 685 (SCA section) and 725 (Intents schema)
0.9.9	October 2008	David Bosschaert: Changes relating to bugs 740, 856, 782, 808, 731
0.9.9.1	November 2008	Graham Charters: Changes for bugs 868 (SCA intent definitions) and 697 (best practices – remote vs local)
0.9.9.2	November 2008	Added response descriptions for messaging intents. Added <service-descriptions/> schema (bug 757) and updated associated example.
0.9.9.3	December 2008	Philipp Konradi: resolution for bug 761 (Discovery auto-publish design)
0.9.9.4	December 2008	David Bosschaert: resolution for bug 735 (removal of getPublishedServices() API from DistributionProvider service).
0.9.9.5	December 2008	Tim Diekmann (tdiekman@tibco.com): preparation for vote in EEG
0.9.9.6	December 2008	Graham Charters: bugs 853 (scenario clarification) and 758 (passByValue as the default semantic).
0.9.9.7	December 2008	David Bosschaert, rewrote section 5.3.2 (Discovery using local files) as discussed during the authors call.
0.9.9.8	December 2008	Philipp Konradi: updated Discovery service interface, updated section 5.6 (Collaboration of new and changed entities), deleted optional Discovery properties, moved chapter "Discovery Service federation and interworking" to Best Practices
0.9.9.9	December 2008	Philipp Konradi: changed white-space separated propertyvalues to String[], updated description of osgi.remote.configuration.type to contain multiple values, other minor clarifications in preparation for vote.
1.0	December 2008	Final version out for vote.
1.0.1	February 2009	Tim Diekmann, minor changes, updated osgi.intents to service.intents according to bug 807

Revision	Date	Comments
1.0.2	March 2009	David Bosschaert, Bug 757, as discussed in the Austin F2F.
1.03	April 2009	Tim Diekmann, bug 807, renamed service.intents deployment.intents as discussed in Austin F2F

Tim Diekmann 4/10/09 5:27 PM
Formatted Table

1 Introduction

This RFC is being created as a design document to meet the requirements described in RFPs 79 and 88. The focus is on defining a possible solution within the OSGi environment to provide a minimal level of feature/function for distributed OSGi processing, including service discovery and access to and from external environments. This solution is not intended to preclude any other solution and is not intended as an alternative to JEE, SCA, JBI, or any other external API set that may be mapped onto OSGi, although the solution is intended to enable interworking with external implementations of those and other technologies.

The solution is intended to allow a minimal set of distributed computing functionality to be used by OSGi developers without having to learn additional APIs and concepts. In other words, if developers are familiar with the OSGi programming model then they should be able to use the features and functions described in this solution very naturally and straight forwardly to configure a distribution software solution into an OSGi environment to meet requirements stated in RFPs 79 and 88. If developers need to use advanced distributed computing capabilities they can use any other supported APIs defined for OSGi deployment to augment or replace the basic functionality described in this RFC.

This RFC is based on describing the minimal extensions necessary to the existing OSGi environment for the purposes of allowing:

- An OSGi bundle deployed in a JVM to invoke a service in another JVM, potentially on a remote computer accessed via a network protocol
- An OSGi bundle deployed in a JVM to invoke a service (or object, procedure, etc.) in another address space, potentially on a remote computer, in a non OSGi environment)
- An OSGi service deployed in another JVM, potentially on a remote computer, to find and access a service running in the “local” OSGi JVM (i.e. an OSGi deployment can accept service invocations from remote OSGi bundle
- A program deployed in a non OSGi environment to find and access a service running in the “local” OSGi JVM (i.e. an OSGi deployment can accept service invocations from external environments)

Basic assumptions include that the default mode of distributed access is consistent with the current OSGi programming model (i.e. a service oriented request/response model) and that in most cases the use of distribution software can be accomplished through the use of configuration and deployment metadata. The configuration and deployment metadata is

based on the Service Component Architecture (SCA) intent model of abstracting distributed computing capabilities. The design is intended to work with any broadly adopted type of distributed computing software system, such as Web services, CORBA, or messaging.

Existing distributed computing technologies are used in all cases to meet the requirements. A further distinction is drawn between solutions that use the same distributed software system for all communications, and solutions that use multiple distributed software systems. When multiple distributed software systems are involved additional metadata may be required to ensure consistency and compatibility of the configurations.

This RFC does not define any new distributed communication protocols, data formats, or policies: it simply defines an extension to the OSGi programming model and metadata that defines how to access and load modules for existing communication protocols, data formats, and policies (i.e. qualities of service assertions and associated configurations) to meet the requirements of RFPs 79 and 88.

1.1 Terminology

***OSGi service platform*:** See OSGi core specification chapter 1.

OSGi bundle: See OSGi core specification chapter 3 and 4.

OSGi service: See OSGi core specification chapter 5.

OSGi service registry: See OSGi core specification chapter 5.

***Component*:** A piece of code (e.g. similar to a Spring bean or a POJO) that is packaged and deployed in a bundle.

***Application*:** A set of bundles that are logically coupled to perform a common task. The bundles of this application don't have to be deployed in the same service platform, but can be spread over multiple service platforms.

Distribution software (DSW): A software entity providing functionality to an *OSGi service platform* that supports the binding and injection of services in other address spaces or across machine boundaries, using various existing software systems.

Discovery service: A software entity providing functionality to an *OSGi service platform* that supports the publishing and lookup of services in other address spaces or across machine boundaries, using various existing discovery systems.

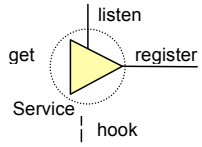
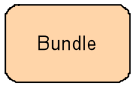
Service consumer: A bundle which requires a service from other service platforms.

Service provider: A bundle which provides a service to other service platfo

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

1.2 List of Symbols

The following symbols are used in the drawings in this document to illustrate the desired behavior of the distributed OSGi design.

Symbol	Term	Description
	OSGi Service	- can be registered by bundles (register) - can be looked for and used from other bundles (get) - can be listened on the service listener, e.g. a service tracker listens on service events (listen) - can be hooked into the process of service registration and lookup (hook) -can be configured to be accessed remotely
	OSGi Bundle	- provides modularization and encapsulation of components - is a deployment unit (software provisioning) for the OSGi runtime
	Extender	An (extender) bundle listens for life-cycle events of other bundles and synchronously acts if necessary e.g. to inject dependencies. The extender bundle is the one the arrow starts at.
	External Interface	- provides an interface outside of a local OSGi Service Platform - exposing transport or communication protocols, e.g. SOAP/HTTP, CORBA/IOP, RMI, etc.
	Non OSGi Platform	- provides a component based environment for enterprise applications - offering non OSGI technologies, e.g. SCA, Spring, etc.
	OSGi Service Platform	- provides a service-oriented, component-based environment - focused on the component integration and the software lifecycle

Also UML notation was used for some diagrams in this document. Please refer to www.uml.org for details on the notation.

2 Application Domain

[copied and combined from RFPs 79 & 88]

The primary design addressed by this RFC is intended to meet requirements for the heterogeneous enterprise IT environment that includes existing and new non-OSGi based applications that need to

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

communicate with OSGi based applications, and with which OSGi based applications need to communicate, including connecting embedded systems to enterprise systems.

Examples of such applications include internet banking applications connected to mainframe databases, travel applications with multiple providers of travel item reservations that all use different technologies, telecommunications industry services for broadband telephony and television that rely on legacy billing applications, and so on. Typical enterprise deployments include large scale applications, which require high availability, reliability, and scalability of the provided services.

Standalone or single technology applications (i.e. OSGi only) are also in scope, because of the fact that OSGi based applications might be deployed in more than just one OSGi platform and for scalability and availability purposes need to be able to find each other across the platform boundaries.

Some core features of heterogeneous enterprises:

- "Stove-piped" applications written using different languages and software systems, including but not limited to .NET, JEE, C++, CORBA, message oriented middleware, TP monitors, data base management systems, packaged applications, EDI, and Web technologies
- Applications built and maintained by separate departments and business divisions that were not designed or built using any consistent principles, and may or may not have integration points exposed.
- Multiple communication protocols and paradigms (i.e. synch and asynch) for interacting with different applications
- Multiple data formats for the same, or similar data items that need to be accessed consistently or reconciled for both read and update operations.
- Quality of service requirements inherent in existing applications, including security, transactionality, reliability, and performance service levels of agreement that need to be met, sometimes expressed in machine readable policies and configuration files

While OSGi has some of the capabilities in place for interaction with external systems, the requirements of interacting with heterogeneous IT environments is often dictated by the requirements of the existing applications, since they represent communications protocols, data formats, programming languages, software systems, and qualities of service agreements already in place for the business.

3 Problem Description

3.1 From RFPs 79 & 88:

Sometimes the objective of the interaction between new OSGi based applications and existing applications will be to perform retrieval and update operations directly on existing data resources. Other times the objective of the interaction will be to use an existing or new program to serve as a proxy or intermediary for another program's data operations. Other times the objective of an interaction will be to request the execution of some business logic, or to evaluate some data, or perform a complex calculation and return the results.

Independent of the interaction scenario, the services of the new OSGi based application need to be discovered by potential clients running outside of the hosting OSGi platform.

The problem space, therefore, has the following characteristics:

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

- Local OSGi services are only accessible from inside the same OSGi platform execution environment.
- Remote OSGi services need to be discovered and accessible from outside of the OSGi platform execution environment.
- Information about the distributed capability needs to be included in the registration and discovery of remote OSGi services. A mechanism needs to be defined for plugging in or binding to different communications protocols and data formats. A mechanism needs to be defined for plugging in or binding to different data formats – the requirement here in both cases can also be stated as how to bind an OSGi service to a transport layer and (potentially separately) a data format layer
- A mechanism that defines how to mix 'n' match communications protocols and data formats so that data formats can be reused over multiple transports (e.g.. allow SOAP over JMS or binary over HTTP)
- Existing legacy systems need to be able to locate and access OSGi services of new applications
- Embedded applications need to interoperate with enterprise applications
- Besides the pure interface definition additional metadata needs to be available about the services that are found remotely in order to assess their eligibility for reference binding. This metadata is part of the service contract and may include non-functional requirements.
- A mechanism to download a remote service
- A mechanism to configure or plug in quality of service capabilities
- A mechanism to interact with external (i.e. remote) data resources

The requirements of interacting with existing heterogeneous IT environments is often dictated by the requirements of the existing applications, since they represent communications protocols, data formats, programming languages, software systems, and qualities of service agreements already in place for the business.

Another requirement centers on interoperability:

1. A service published remotely through OSGi implementation A should be accessible from another Service that runs in OSGi implementation B.
2. Implementations A and B could be based on entirely different OSGi runtimes.
3. For a user of the OSGi runtime, it should be easy to identify that a certain OSGi runtime can interoperate with another OSGi runtime by examining the service properties and any associated metadata. So let's say the user already has an OSGi runtime that exposes its services using a certain wire protocol, e.g. SOAP/HTTP. If the user starts using another OSGi runtime that says that it supports SOAP/HTTP they should interoperate.

Whether or not an IDL or some other formalism like special use of Java Interfaces would be needed to satisfy this is certainly a valid discussion point, but it would be good to try and solve it within the boundaries of Java Interfaces, simply because this concept is already used in OSGi.

3.2 Scenario diagrams

Schematically, the problem domain centers on a solution to the following scenarios. Note that the non-OSGi clients and servers mentioned may represent existing and legacy applications that typically can't be modified.

The scenario illustrated in Figure 1 focuses on a client in one OSGi platform that needs to invoke on a Service that lives in another OSGi platform. Both client and server might initially not be written to be distributed. However, it may in certain cases be a possibility to tweak the client and/or service code to make them behave better in this distributed scenario.

Tim Diekmann 4/10/09 5:29 PM
Deleted: Figure 1

Note that in this case an implementation might choose to use an optimized protocol to communicate between the OSGi runtimes. Note also, that if the same distribution software (e.g. ESB) is used in both service platform instances, then the configuration required can also be optimized.

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

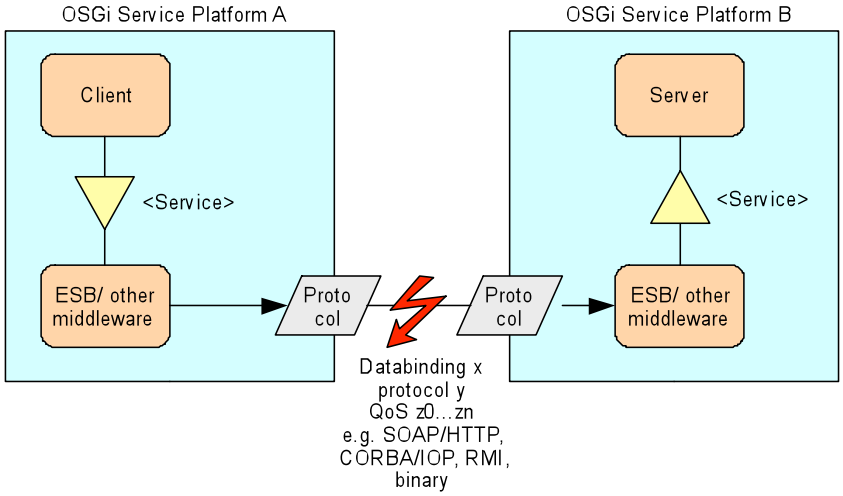


Figure 1 OSGi service consumer using a remote OSGi-service provider

The general use case of distributed OSGi is depicted in Figure 1. A client hosted in OSGi Service Platform A wants to use a service provided by another bundle hosted in OSGi Service Platform B. Since this is a remote service invocation spanning multiple framework processes (i.e., multiple JVMs), some intermediary bundle is required in both service platforms to marshal and unmarshal the communication objects. This RFC describes the mechanisms how to find and match client and server as well as how to implement the intermediary bundle to enable the remote invocations.

Tim Diekmann 4/10/09 5:29 PM
Deleted: Figure 1

It is the intent of this RFC to allow for any implementation of the distribution software as shown in the picture utilizing any protocol for the communication, associating metadata with the service to indicate that it's remotable, and with which distributed software characteristics (as expressed using "intents").

Note: As described in the requirements section, RFC 119 is also addressing the scenarios in which the client side is hosted in a non-OSGi environment. In this case, the left side would be replaced by another client hosting platform, e.g. .NET. Additionally, OSGi based clients should be able to remotely access services hosted in a non-OSGi environment, which would mean that the right side is replaced with a different hosting platform.

3.2.1 Consumer Side

The following diagram illustrates the detailed scenario from the consumer side.

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

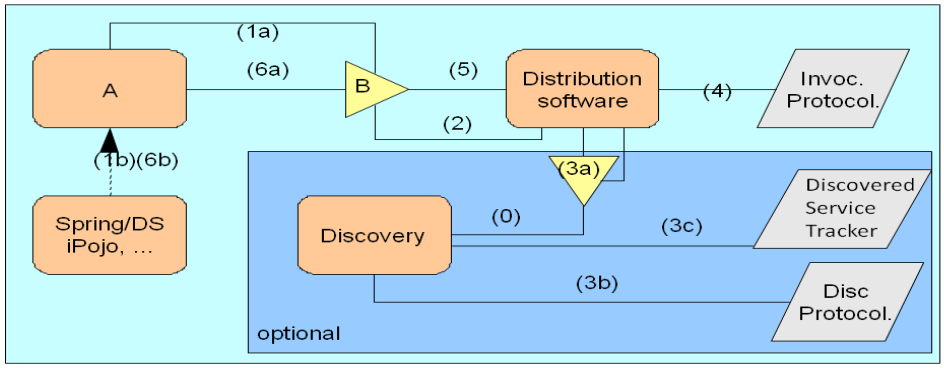


Figure 3 Service Consumer - in OSGi framework

Tim Diekmann 4/10/09 5:29 PM
Deleted: *2*

Figure 3 shows the OSGi implementation in the client OSGi platform A. Bundle A is interested in Service B and performs a lookup in the service registry (expressing the metadata intents it requires, if any (See Section 5.5.3 for the definition of intents)) or uses a ServiceTracker to listen for events regarding Service B – step (1a). Service B can have metadata properties associated with it to indicate that it's accessible remotely. Optionally, a dependency management mechanism such as Declarative Services or Blueprint Service based components (see RFC 124), or others could perform the dependency checking and register such a listener (1b).

Tim Diekmann 4/10/09 5:29 PM
Deleted: Figure 2

Step (2) in the diagram refers to RFC 126, service registry hooks (see also Section 5.4). It allows the distribution software to register a hook in the service registry, which is called when a service is being looked-up or requested.

In the optional step (3a), the distribution software could use the discovery service to perform the lookup of Service B over the network. The Discovery service is an optional service and registers its service upon startup. Discovery allows for synchronous as well as asynchronous discovery of services suitable for providing Interface B and meeting the requirements of Bundle A. Step (3b) illustrates the direct use of the discovery protocol, while step (3c) illustrates the use of a mechanism that tracks previously discovered services.

Step (3) is optional, because the distribution software may also acquire the information about Service B through other means, such as static configuration (wiring) as part of its implementation, or using a local file (see Section 5.3.2).

The distribution software and the Discovery service do not have to come from the same vendor and adhering to the OSGi specification allows for seamless integration of different discovery and distribution implementations.

In step (4) the distribution software creates a local endpoint for the discovered provider of Service B. The deployed protocol depends on the available protocols for Service B. (See discussion below for details about the provider side.)

In step (5), the distribution software registers the proxy with Interface B, which causes in step (6a) and (6b) the service reference to be returned to the calling party or injected by the dependency mechanism.

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

3.2.2 Provider Side

The following diagram illustrates the provider side.

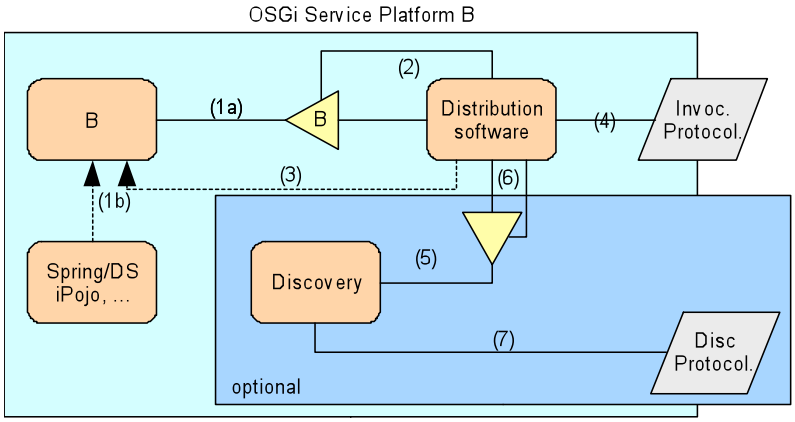


Figure 4: Service provider - OSGi framework

Tim Diekmann 4/10/09 5:29 PM
Deleted: *3*

In Figure 4 it is shown how Bundle B inside the OSGi Service Platform B registers a Service B in step (1a), including its metadata stating it's remotely accessible and with which characteristics (i.e. any specified properties and intents). Optionally, this step could also be performed by a dependency management mechanism such as Declarative Services, Blueprint Services, or any other non-standard implementation (1b).

Tim Diekmann 4/10/09 5:29 PM
Deleted: Figure 3

In step (2) the distribution software is notified about the registration of Service B and using additional information provided by step (3) in which the extender model can obtain any intents). This could be done by an extender or through properties as part of the registration of Service B. To make Service B reachable through a communication protocol the distribution software creates a local endpoint for the supported protocol(s) in step (4).

The OSGi Service Platform B may optionally also have deployed a discovery bundle as specified in this RFC. The discovery bundle registers is standard interface in step (5) and the distribution software is notified about the presence of the discovery service in step (6). Using the discovery service, the distribution software may then publish the information about the availability of Service B using any discovery protocol that the discovery service supports.

Note: It is entirely possible and encouraged that there are 0..n different discovery bundles deployed in the OSGi service platform. Multiple distribution software system types are also permitted.

3.2.3 A non-OSGi distributed client using an OSGi service

Figure 5 shows a legacy client that needs to invoke a service provided by an OSGi service. The client is written in a programming language such as C++ and uses a certain distributed protocol, such as SOAP/HTTP(S), CORBA/IIOP or RMI to access this service.

Tim Diekmann 4/10/09 5:29 PM
Deleted: Figure 4

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

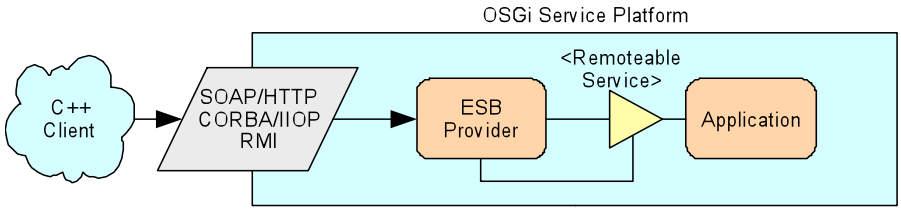


Figure 5 Remote non-OSGi service consumer using an OSGi service provider

Tim Diekmann 4/10/09 5:29 PM
Deleted: *4*

As illustrated in Figure 5, a C++ client deployed in a runtime environment external to OSGi accesses a service deployed in an OSGi platform using one of the communication protocols supported by a DSW deployed in the OSGi platform.

Tim Diekmann 4/10/09 5:29 PM
Deleted: Figure 4

Figure 7 illustrates additional detail of the OSGi runtime part of this scenario. While certain services could be distributed, and are therefore marked with the publish metadata property, it is also possible for other services to exist in the same OSGi platform that aren't distributed. A co-located client would be capable of making a direct invocation on services that are in the same OSGi platform regardless of their distribution status. Note, in doing this, care must be taken to ensure that the possible change in call semantics (e.g. from remote pass-by-value to local pass-by-reference in most cases) does not adversely alter the behavior of the service.

Tim Diekmann 4/10/09 5:29 PM
Deleted: Figure 5

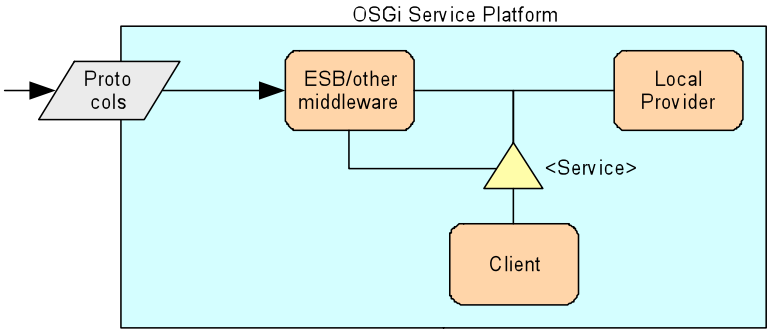


Figure 7 Service consumer uses a distributed but locally available service provider

Tim Diekmann 4/10/09 5:29 PM
Deleted: *5*

Figure 7 illustrates the scenario in which a remote service call from an external environment passes the call to a local OSGi service to invoke the actual service target from the remote invocation (and the actual service target could itself be in a remote OSGi platform).

Tim Diekmann 4/10/09 5:29 PM
Deleted: Figure 5

3.2.4 An OSGi client using a remote non-OSGi Service

Figure 9 illustrates how an OSGi client invokes a legacy service. The service is exposed using a particular type of middleware, e.g. SOAP/HTTP, CORBA/IIOP, RMI, etc. The service is also identified within the OSGi environment as remotely accessible using OSGi metadata (i.e. properties and intents).

Tim Diekmann 4/10/09 5:29 PM
Deleted: Figure 6

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

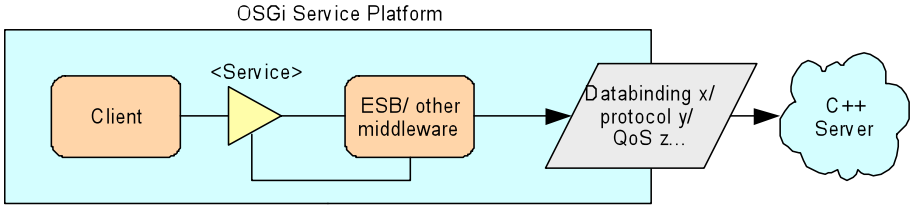


Figure 9 OSGi service consumer using a remote non-OSGi-service provider

Tim Diekmann 4/10/09 5:29 PM
Deleted: *6*

Figure 9 shows how an OSGi service client can access a service proxy created using a distributed software system that connects remotely to a C++ server deployed in an external runtime environment, using a distributed communications protocol and data format supplied by the DSW configured into the OSGi platform,

Tim Diekmann 4/10/09 5:29 PM
Deleted: Figure 6

3.3 Roles

When creating a distributed application people with different roles are involved, as shown in Figure 11. This section describes the roles relevant to this document.

Tim Diekmann 4/10/09 5:29 PM
Deleted: Figure 7

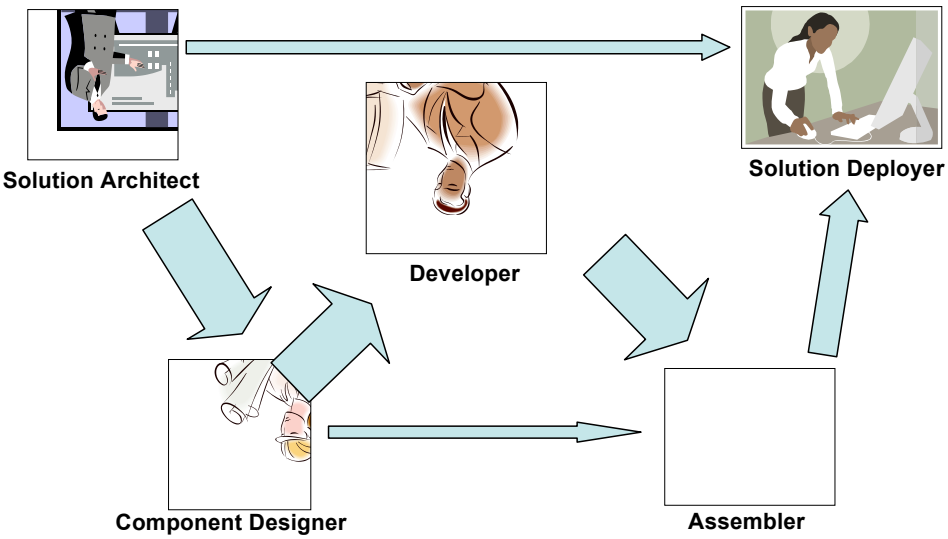


Figure 11 Relationships Among Designer, Developer, Architect, Assembler and Deployer Roles

Tim Diekmann 4/10/09 5:29 PM
Deleted: *7*

The following table shows how these roles collaborate with each other, which artifacts are required to perform the tasks in these roles, and which artifacts are produced:

Role	Required Artifacts	Performed Tasks	Collaborates With	Produced Artifacts
Solution Architect	Application Requirements	Analyses requirements	Solution Deployer	Solution Specification Component

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

		Defines SOA architecture Defines the Service Interface and some properties	Component Designer	Requirements Service Interface Definition (e.g. UML class diagram, etc.) and property definition (e.g. remoteable)
Component Designer	Component Requirements Service Interface Definition and property definition	Designs the service implementation Specifies the service interface Defines service-specific properties, intents and optional metadata	Solution Architect Developer Assembler	Component Specification Service Interface Specification (e.g. WSDL file, IDL file, etc.) Definition of Service Properties, intents and optional metadata (e.g. call by reference, call by value, remoteable, etc.)
Developer	Component Specification Service Interface Specification Definition of Service Properties, intents and optional metadata	Implements the business logic Defines implementation-specific properties	Component Designer Assembler	Components
Assembler	Components Definition of Service Properties, intents and optional metadata	Builds installable packages Creates service properties (types and defaults)	Components Designer Developer Solution Deployer	Bundles Service Properties, intents and optional metadata (i.e. bundle specific property file or XML file containing defaults)
Solution Deployer	Solution Specification Bundles Service Properties, intents and optional metadata	Deploys and configures the application Provides solution specific configuration properties	Solution Architect Assembler	Application Configuration Properties, intents and optional metadata (e.g. communication protocols, policies, etc.)

Note: within a role the members collaborate with each other, e.g. the component designer collaborates with other component designers

3.3.1 Solution Architect

The Solution Architect is responsible for defining the functional and non-functional component requirements and for providing the service interface definition. In addition he provides the solution specification to the solution deployer.

He analyses the application requirements and models an appropriate SOA architecture in which functionality is decomposed into services, which can be distributed over a network and can be combined together and reused to create applications.

The solution architect divides the required functions of the application between the components and specifies this architecture design in the component requirements in an informal manner. Furthermore he provides the service interface definitions in form of UML diagrams, which hides the concrete technology used for the service interface like WSDL for Web services, CORBA IDL, RPC IDL, etc.

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

3.3.2 Component Designer

The component designer is responsible for creating the service interface and for providing the component specification, which specifies the design of the service implementation. In addition he provides the service-specific properties, intents and optional metadata to the assembler and developer.

He works from the service interface definition provided by the solution architect, models the interface objects, e.g. request/response objects, data types, exceptions, etc, and maybe with support of a tool he creates the service interface, e.g. WSDL file, IDL file, etc.

The Component Designer also works with definitions of remote services to be consumed by the component. These are provided by the solution architect and will consist of a service interface description (e.g. Java, WSDL, IDL), and optionally service properties and intents. These remote service definitions may be defined by the solution, or could be external interfaces dictated by a third-party.

Analyzing the component requirements the component designer specifies the design of the service implementation in the component specification. He defines the service-specific properties, intents and optional metadata inherently associated with the component, such as 'osgi.remote.interfaces', call by reference / call by value semantics, required QoS, etc. He may define additional properties which depend on the concrete environment the component is used in and thus needs to be provided at runtime.

3.3.3 Developer

The developer is responsible for building the components, which is a set of classes comprising the business logic implementation, according to the component specification.

He works from the service interface provided by the component designer, codes the business logic and creates the business data objects.

3.3.4 Assembler

The assembler is responsible for assembling the components into bundles, which are installable packages, and for providing the defaults for service properties, intents and optional metadata.

He collects and validates the produced components and packages them appropriately together for an OSGi bundle, e.g. by analyzing which services communicate with each other he decides to package bundles for service consumers and service providers.

Based on the service property, intent and optional metadata definitions from the component designer and the developer the assembler creates the service properties, intents and optional metadata by means of types and defaults, and provides additional bundle specific properties, intents and optional metadata, e.g. reuse in multiple applications, etc. These properties are supplied either in a property file, which are configuration values managed as key/value pairs, or in XML files, specifying intents, bindings, policy sets and properties. (Depending on the property file versus XML file discussion)

3.3.5 Solution Deployer

The solution deployer is responsible for deploying the application, which is a set of bundles that are coupled together to perform the solution, and for providing distribution configuration.

He collects the OSGi bundles from the assemblers and configures the distribution software to distribute the services by providing configuration properties, intents and optional metadata as required by the solution specification, e.g. communication protocols to be used, policies which needs to be applied, etc. Additionally he can identify a component as 'remoteable' even it was not previously marked as such.

The solution deployer analyses the whole solution for performance issues, and diagnoses errors at the implementation / binding level.

Note: In the end the distribution software is responsible for providing a default for all those properties, intents and metadata that were not set in the steps performed by the previously described roles.

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

3.3.6 *Testing*

Testing is part of each role and is accompanied by each produced artifact to ensure performance, robustness and interoperability for the whole solution.

3.3.7 *Runtime (Framework)*

Controls the lifecycle of services and service dependencies (e.g. DS, Blueprint). Unresolved packages, class loading issues are indicators for improper configuration by the solution deployer.

4 Requirements

4.1 From RFP 79

1. The solution **MUST** provide means to discover OSGi services from outside the OSGi platform. This includes external clients as well as other OSGi services hosted in separate platforms.
2. The solution **MUST** support clients independent of the programming language and independent of the location they are at.
3. The solution **SHOULD** provide means to discover remote services through the local OSGi service registry and standard OSGi mechanisms.
4. The solution **MUST** be independent of the implementation of the discovery protocol. Multiple implementations must be possible in a single platform. It is understood that only those services will be discoverable that are actually discoverable by the discovery protocol implementation, i.e. a SLP implementation of the service discovery can only discover services that are advertised by SLP.
5. The solution **SHOULD** avoid or minimize the knowledge about the underlying implementation protocol of the discovery by any service in the local OSGi platform.
6. The OSGi service registry **SHOULD** contain information about the discovered OSGi services. The information available for the discovery as well as the registration and lookup **SHOULD** include
    a. Supported communication protocol(s).
    b. Meta-data about the OSGi service, defined by the service itself.
    c. Provided Interface(s)
    d. Quality-of-service information, e.g. transaction support, service specific policies, time constraints, etc.
    e. Transport information, e.g. support for IP V6
    f. Version information of the interface
7. The solution **SHOULD** support an OSGi service registering multiple interfaces.
8. The solution **SHOULD** support multiple OSGi services registering the same interface.

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

9. The solution **MUST** only expose information about those OSGi services that want to be discovered from external clients. Thus, NOT every OSGi service listed in the OSGi registry MAY automatically be included in the discovery for external services.
10. The solution **SHOULD** provide for limited visibility of services in the registry based on security mechanisms, e.g. authentication and authorization
11. The solution **SHOULD** ensure a reasonable response time for service lookup requests.

4.2 From RFP 88

1. The solution MUST enable interoperability between OSGi developed services (or components) and services (or components) developed using non OSGi environments.

The interoperability would typically be provided through the use of existing distributed data bindings and protocols such, e.g. SOAP/HTTP, CORBA/IIOP, JMS, RMI etc. Not all possible integrations need to be delivered, what is needed is an extensible framework that can hold these. The Reference Implementation should come with at least 2 implementations to prove the scenario and pluggability.

2. The solution SHOULD abstract protocols, data formats, and quality of service features in order to be easily adaptable to communication protocols, data formats, and qualities of service found in existing enterprise applications and software systems.

This means that the user code should not be required to be written against a particular type of protocol. This should be abstracted.

3. The solution SHOULD be compatible with multiple external programming languages and operating systems.

So it should allow interoperability with systems written in a variety of programming languages (e.g. C/C++, .NET, Cobol, scripting languages) running on a number of operating systems such as Windows, UNIX, Mainframes. Note that these external systems do not need to be running an OSGi platform. Interoperability would be provided through the distributed databinding & protocol used.

4. The solution SHOULD be extensible for custom developed interoperability solutions (i.e. users can add their own protocols, data formats, and quality of service extensions).

5. The solution SHOULD be configurable and understand policy expressions for the provisioning of the interoperability solutions, especially including the quality of service features.

The policy information could be for example expressed as WS-Policy expressions which should give the administrator the ability to define the distribution-related metadata in a declarative way.

6. The solution SHOULD support high availability and performance requirements typical of existing enterprise systems.

7. The solution SHOULD bridge the OSGi context sharing mechanism with external context sharing mechanisms (to support stateful failover, shared stateful sessions, etc.).

8. The solution SHOULD NOT introduce language specific, protocol specific, or quality of service specific dependencies.

9. The solution for external access SHOULD be as consistent as possible with the solution for accessing internal OSGi services, to minimize the amount of effort in moving from one to the other.

10. The solution SHOULD provide a consistent mechanism for simultaneously incorporating multiple protocols and data formats.

11. The solution SHOULD provide a consistent mechanism for quality of service enhancements. .

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

12. The solution MUST make it possible, but not necessary, for developers to interact with the distributed attributes of the system, such as distributed error conditions, and information around the data binding, transport and QoS.

To give application developers the option to find out the distributed properties of the Service and also be capable of detecting remoting-related specific error conditions if they wish to do so.

13. The solution MUST NOT prevent the use of asynchronous programming models if these are provided by the transport used.

In other words, if the transport provides an asynchronous protocol such as JMS, CORBA one-ways or other message queue or publish-and-subscribe model, it must be possible for the application programmer to take advantage of this asynchronous nature.

14. The solution SHOULD support the capability for a developer to declaratively specify the configuration requirements for a protocol layer.

15. The solution SHOULD allow a deployer to define wiring and configuration information for bundles and create distributed solutions with minimal or no code changes.

4.3 Further requirements

4.3.1 Levels of transparency

While it would nice to be able to turn an existing OSGi Service & Client into a distributed OSGi system without making code changes, it cannot be required that distributed OSGi is entirely transparent. The distributed nature of the system will introduce new scenarios (e.g. new failure scenarios) that were not relevant to non-distributed OSGi. If the program wishes to, it should be allowed to interact with the distributed nature of the system. Therefore, the following levels of transparency should be supported:

1. Completely transparent to the developer. No code changes needed in either Client or Service. Metadata changes will probably be necessary at this level.
2. The programmer should be able to influence the lookup of the Service in the Client based on properties provided in the metadata (e.g. transport, QoS, etc).
3. For any given distributed service it must be possible to find out what the distribution software is and obtain additional metadata that describes the data binding, protocol and QoS.
4. It must be possible to preserve distribution software specific exceptions and handle them as before, if desired. Another exception is defined to indicate a problem occurred in the mapping software.

5 Technical Solution

5.1 Overview of contributions to the OSGi standard

Distributed OSGi enhances the capabilities of the OSGi framework and opens deployment areas in the enterprise market. This section summarizes the changes to the existing specification as of R4.1, and summarizes the additional optional services in distributed OSGi. Subsequent sections provide more details on each.

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

5.1.1 Summary of Changes to the OSGi Core

The following changes to the core OSGi specification are contained within this design:

- Adaptation of RFC 126 regarding service registry hooks. The proposed solution for this RFC requires the ability to hook into the process of registration, see Section 5.4 for further details.
- Changes to the service programming model for distribution:
  - Reserved properties :
    - service.intents – list of intents satisfied by this service.
    - osgi.remote.interfaces – indicates that the provided service is to be made available remotely, which implies that it is suitable for remote invocations. The value of this property indicates which interface or interfaces implemented by the service are to be exposed remotely.
    - osgi.remote.requires.intents – list of intents that should be satisfied when publishing this service remotely. Provided by the component designer and changeable by the deployer.
    - osgi.remote.configuration.type – identifies the metadata type of additional metadata, if any, associated with the service provider or consumer, e.g. “sca”
    - osgi.remote – indicating that a service implementation is actually remote. This property is set on client side proxies so that the consumer can identify remote services if needed.
  - Metadata for configuring distribution software, which includes basic intents used when there’s a single type of distribution software, and additional metadata when multiple types of distribution software are required. This metadata is provided using service properties described above: deployment.intents, osgi.remote.requires.intents and osgi.remote.configuration.type.

Tim Diekmann 2/20/09 1:22 PM
Deleted: starting with *osgi.* including

Tim Diekmann 2/20/09 1:22 PM
Deleted: osgi

Tim Diekmann 2/20/09 1:23 PM
Deleted: osgi.intents

5.1.2 Summary of Additional Services

The distributed OSGi mechanism presented in this RFC 119 is an optional component to an existing OSGi Service Platform as described in the requirements Section 4. As such, the following new OSGi services are proposed to be added to the compendium document of the OSGi specification.

- Distribution software – provides remote invocation capability over one or more protocols; takes care of exposing a service remotely and also provides consumers of remote services with a local reference (proxy) to invoke the remote service. The distribution software will preserve the OSGi service programming model by making OSGi services available to external clients and allowing consumer bundles written in OSGi to bind to external services through OSGi service registry mechanisms. See Section 5.2 for further details.
- Discovery service – an optional service additional to distribution software to locate OSGi based and non-OSGi services over any available network protocol or other means used by the implementation. See Section 5.3 for further details.

5.2 Distribution software

5.2.1 Functionality

The distribution software is responsible for the actual network communication between a remotely available service and its consumer, including the data format (i.e. serialization) and communication protocol.

When a consumer invokes on the remote service the distribution software knows how to marshal the arguments and will then make the dispatch invocation on the remote entity. On completion it will unmarshal the response and return to the caller.

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

On the provider (service) side, the distribution software knows how to make an OSGi Service available over the network so that it can be invoked remotely. The distribution software may optimize on a particular distributed computing protocol, which may require the OSGi Service Java interface to be mapped onto that technology. Example target technologies include CORBA, RMI and Web Services technologies. However, this specification also allows an implementation to use other protocols and bindings, including proprietary ones.

The distribution software is responsible for interpreting the distribution-related metadata on an OSGi service and making the service available remotely if this is required by this metadata. This metadata can optionally include instructions about the actual remote data binding and transport to be used, as well as requirements around security, reliability, transactions and other Qualities of Service, depending on metadata type. Intents are in any case used to help consumers discover compatible services.

If the DSW detects distributed OSGi metadata it has to configure a proxy for the service, set the appropriate service properties (derived from the metadata), and optionally register it with the service registry for it to be detected by a Discovery service implementation.

On the consumer-side, the distribution software is responsible for creating proxies to remote services so that they can be invoked by the consumer, and supporting the filtering of services by the consumer to detect a remote service, if desired.

The distribution software is optionally responsible for interacting with the Discovery Service to publish and subsequently discover services that it has made available over the network.

Tim Diekmann
Comment: TODO update

Distribution software is an optional component in the OSGi framework that would typically be deployed as one or more OSGi bundles.

The following diagram illustrates a possible solution to the design using Apache Felix as the OSGi platform and Apache CXF as the distribution software.

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

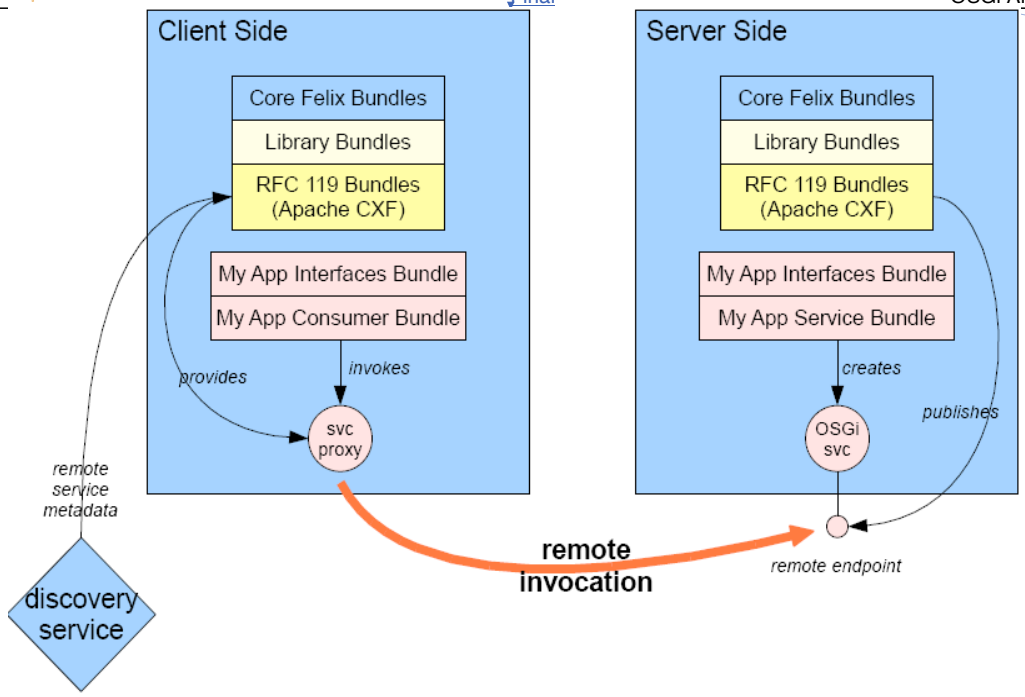


Figure 13 Example Implementation of Distributed OSGi using Web services

Tim Diekmann 4/10/09 5:29 PM
Deleted: *8*

Tim Diekmann 4/10/09 5:29 PM
Deleted: Figure 8

As illustrated in Figure 13, distributed communications between OSGi platform instances can be achieved by configuring a distributed software system such as Apache CXF into both client and server sides. In this example Apache Felix is used as the OSGi Framework implementation, and CXF is loaded into the framework. On the application side, common interface bundles are used, while consumer bundles are deployed on the client side and service bundles are deployed on the server side. A service proxy on the client side performs the communication with the remote endpoint deployed on the server side, which is created by CXF when the service is published. The Discovery service can be used on the client side to discover the location and additional properties of the remote service. In this example, the discover service references a metadata file in a directory local to the client, but another Discovery service would access a remote discovery mechanism such as UDDI or LDAP.

5.2.2 Interface description

The requirements for the distribution software state that the mechanism of how it implements the remote capabilities should not be defined in this document. Consequently, there is no mandatory functional interface to be implemented by a distribution software solution.

On the other hand, the need for identification of the deployed distribution software, its capabilities and version was raised and agreed upon. Therefore, any distribution software SHOULD implement the interface in Section 5.2.2.1 to return information about itself that is useful for identification and also in logging statements.

5.2.2.1 Distribution Software Interface

The distribution software implementation should be mandated to register a service in the local OSGi Service Registry that implements a standardized interface, which allows for obtaining static information

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

about the vendor, version, etc. as well as dynamic information about the remote service proxies it has created, the protocols it supports, and possibly other runtime statistics, which can be of value for a management console.

```
/**
 * Every Distribution Provider registers exactly one Service in the
 * ServiceRegistry implementing this interface. The service is registered with
 * extra properties identified at the beginning of this interface to denote the
 * Distribution Provider product name, version, vendor and supported intents.
 *
 * @ThreadSafe
 */
public interface DistributionProvider {
    /**
     * Service Registration property for the name of the Distribution Provider
     * product.
     */
    static final String PROP_KEY_PRODUCT_NAME =
                            "osgi.remote.distribution.product";

    /**
     * Service Registration property for the version of the Distribution
     * Provider product.
     */
    static final String PROP_KEY_PRODUCT_VERSION =
                            "osgi.remote.distribution.product.version";

    /**
     * Service Registration property for the Distribution Provider product
     * vendor name.
     */
    static final String PROP_KEY_VENDOR_NAME =
                            "osgi.remote.distribution.vendor";

    /**
     * Service Registration property that lists the intents supported by this
     * DistributionProvider. Value of this property is of type
     * Collection (<? extends String>).
     */
    static final String PROP_KEY_SUPPORTED_INTENTS =
                            "osgi.remote.distribition.supported_intents";

    /**
     * @return ServiceReferences of services registered in the local Service
     *         Registry that are proxies to remote services. If no proxies are
     *         registered, then an empty collection is returned.
     */
    Collection /*<? extends ServiceReference>*/ getRemoteServices();

    /**
     * @return ServiceReferences of local services that are exposed remotely
     *         using this DisitributionProvider. Note that certain services may be
     *         exposed and without being published to a discovery service. This
     *         API returns all the exposed services. If no services are exposed an
     *         empty collection is returned.
     */
    Collection /*<? extends ServiceReference>*/ getExposedServices();

    /**
     * Provides access to extra properties set by the DistributionProvider on
     * endpoints, as they will appear on client side proxies given an exposed
     * ServiceReference.
     * These properties are not always available on the server-side
```

```
     * ServiceReference of the exposed
     * service but will be on the remote client side proxy to this service.
     * This API provides access to these extra properties from the exposing
     * side.
     * E.g. a service is exposed remotely, the distribution software is configured
     * to add transactionality to the remote service. Because of this, on the
     * client-side proxy the property service.intents="transactionality" is set.
     * However, these intents are *not* always set on the original
     * ServiceRegistration on the server-side since on the server side the service
     * object is a local pojo which doesn't provide transactionality by itself.
     * This QoS is added by the distribution.
     * This API provides access to these extra properties from the server-side.
     *
     * @param sr A ServiceReference of an exposed service.
     * @return The map of extra properties.
     */
    Map /*<String, String>*/ getExposedProperties(ServiceReference sr);
}
```

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

Tim Diekmann 2/20/09 1:23 PM
Deleted: osgi

5.2.2.2 Exception Handling

There will be a new type of exception for the ServiceException: REMOTE. This type of exception is thrown when there is an issue with the distribution software used to covert between the protocol-specific and OSGi invocations.

Tim Diekmann
Comment: TODO define this exception

When using a specific type of distribution software, the exception handling system must allow distribution software specific exceptions to be captured and propagated to the client as if OSGi was not involved. For example, RMI exceptions can still be reported.

However since distributed OSGi is adding a mapping layer between a service and the distribution software, it's possible for an exception to occur within the mapping layer. The REMOTE exception is thrown to indicate a problem in this area, not to indicate problem within the distribution software itself.

5.3 Discovery Service

The Discovery service is an optional service, which enables publication of services running in a framework to remote consumers and discovery of services running outside a framework.

Publication of a service consists of publication of all the service metadata passed usually by the Distribution Software to Discovery. Discovery may use any internal protocol to transmit that service metadata. On the consumers side distribution software uses this metadata for filtering of potential candidates and creation of service proxies.

5.3.1 Functionality

There are two models for sharing information about distributed services in a system. Either remote services required by a Distribution Software on a node are known upfront and definitions of matching services are transmitted to it, or that Distribution Software autonomously looks for services available in the network. Distributed OSGi supports the latter 'discovery' model in the optional Discovery Service but allows the other model to be supported as an implementation option of Distribution Software.

The Discovery service allows the distribution software to actively discover services based on filter criteria. In addition, the discovery service may provide an asynchronous notification mechanism, which alerts interested clients about the availability of particular remote services.

The strategy and details of the discovery service is left to the implementers. For instance discovery can be performed eagerly (i.e. before anyone has asked for the service), or lazily (i.e. triggered as part of a request to use the service). The design is intended to be simple and flexible enough to allow for multiple different implementations to reside in the same OSGi service platform concurrently. Each Discovery

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

service implementation is expected to provide one or multiple discovery protocols, which are either well known (e.g. SLP, UDDI) or proprietary. Proprietary protocol implementations allow for reuse of existing mechanisms while open standard implementations allow for better integration with existing products in the enterprise market.

The distribution software on the service provider side passes the information about the service provider to Discovery services. For this purpose distribution software registers a `ServicePublication` object, whose registration makes all existing Discovery services publish the provided information and make the service discoverable by other OSGi service platforms as well as other external clients understanding the used protocol. A service may be published by a distribution software multiple times with different bindings. In the case of multiple distribution software implementations in the same platform, multiple publications of a particular service may occur with the same as well as with different bindings.

Figure 9 illustrates also how multiple Discovery services are used to publish a service via more than one type of discovery mechanism (such as SLP and UDDI).

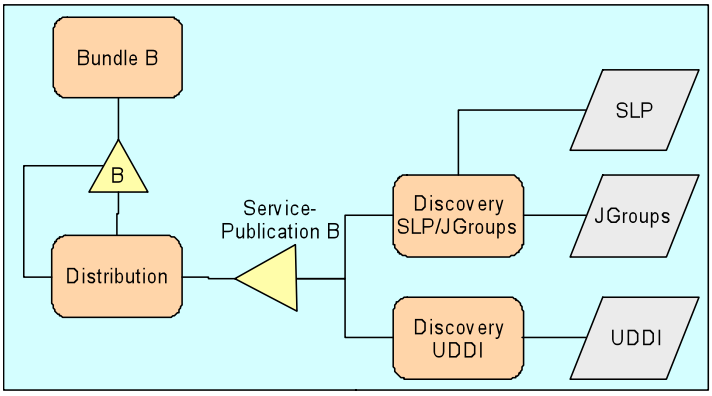


Figure 14 OSGi service published over multiple protocols

Tim Diekmann 4/10/09 5:29 PM
Deleted: *9*

By implementing a discovery protocol of any open standard, the discovery is not bound to OSGi services alone. This allows discovery of services implemented and offered in different technologies like .NET or C++. Likewise, OSGi services are published using the standard protocol to clients built on other technologies than OSGi.

5.3.2 Interface description

5.3.2.1 Discovery interface

```
/**
 * Discovery registers a service implementing this interface. This service is
 * registered with extra properties identified at the beginning of this
 * interface to denote the name of the product providing Discovery
 * functionality, its version, vendor, used protocols etc..
 *
 * Discovery allows to publish services exposed for remote access as well as to
 * search for remote services. Register a {@link ServicePublication} service in
 * order to publish service metadata and or a {@link DiscoveredServiceTracker}
 * service in order to search for remote services.<BR>
 * Discovery service implementations usually rely on some discovery protocols or
 * other information distribution means.
 *
 * @ThreadSafe
 * @version $Revision: 6686 $
```

Philipp Konradi 4/7/09 4:11 PM
Formatted: Font:9 pt, English (US)

Philipp Konradi 4/7/09 4:11 PM
Formatted: Adjust space between Asian text and numbers

```
 */
public interface Discovery {

		/**
		 * Service Registration property for the name of the Discovery product.
		 */
		static final String PROP_KEY_PRODUCT_NAME =
"osgi.remote.discovery.product";

		/**
		 * Service Registration property for the version of the Discovery product.
		 */
		static final String PROP_KEY_PRODUCT_VERSION =
"osgi.remote.discovery.product.version";

		/**
		 * Service Registration property for the Discovery product vendor name.
		 */
		static final String PROP_KEY_VENDOR_NAME = "osgi.remote.discovery.vendor";

		/**
		 * Service Registration property that lists the discovery protocols used
by
		 * this Discovery service. Value of this property is of type Collection
(<?
		 * extends String>).
		 */
		static final String PROP_KEY_SUPPORTED_PROTOCOLS =
"osgi.remote.discovery.supported_protocols";
}
```

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

Philipp Konradi 4/7/09 4:10 PM
Deleted: * @version $Revision: 6046 $.

5.3.2.2 ServicePublication Interface

```
/**
 * Register a service implementing the <code>ServicePublication</code> interface
 * in order to publish metadata of a particular service (endpoint) via
 * Discovery. Metadata which has to be published is given in form of properties
 * at registration. <br>
 * In order to update published service metadata, update the properties
 * registered with the <code>ServicePublication</code> service. Depending on
 * Discovery's implementation and underlying protocol it may result in an update
 * or new re-publication of the service. <br>
 * In order to unpublish the previously published service metadata, unregister
 * the <code>ServicePublication</code> service.<br>
 *
 * Please note that providing the {@link #PROP_KEY_SERVICE_INTERFACE_NAME}
 * property is mandatory when a <code>ServicePublication</code> service is
 * registered. Note also that a Discovery implementation may require provision
 * of additional properties, e.g. some of the standard properties defined below,
 * or may make special use of them in case they are provided. For example an
 * SLP-based Discovery might use the value provided with the
 * {@link #PROP_KEY_ENDPOINT_LOCATION} property for construction of a SLP-URL
 * used to publish the service.<br>
 *
 * Also important is that it's not guaranteed that after registering a
 * <code>ServicePublication</code> object its service metadata is actually
 * published. Beside the fact that at least one Discovery service has to be
 * present, the provided properties have to be valid, e.g. shouldn't contain
 * case variants of the same key name, and the actual publication via Discovery
 * mechanisms has to succeed.
 *
```

Philipp Konradi 3/5/09 4:08 PM
Formatted: Font:9 pt

Philipp Konradi 3/5/09 4:11 PM
Formatted: Font:9 pt, English (UK)

Philipp Konradi 3/5/09 4:08 PM
Formatted: Font:9 pt

Philipp Konradi 3/5/09 4:11 PM
Formatted: Font:9 pt, English (UK)

```
 * @ThreadSafe
 * @version $Revision: 6686 $
 */
public interface ServicePublication {

	/**
	 * Mandatory ServiceRegistration property which contains a collection of
	 * full qualified interface names offered by the advertised service
	 * endpoint. Value of this property is of type Collection (<? extends
	 * String>).
	 */
	public static final String	PROP_KEY_SERVICE_INTERFACE_NAME
	= "service.interface";

	/**
	 * Optional ServiceRegistration property which contains a collection of
	 * interface names with their associated version attributes separated by
	 * {@link #SEPARATOR} e.g. 'my.company.foo|1.3.5 my.company.zoo|2.3.5'. In
	 * case no version has been provided for an interface, Discovery may use
the
	 * String-value of <code>org.osgi.framework.Version.emptyVersion</code>
	 * constant. <br>
	 * Value of this property is of type Collection (<? extends String>).
	 */
	public static final String	PROP_KEY_SERVICE_INTERFACE_VERSION =
"service.interface.version";

	/**
	 * Optional ServiceRegistration property which contains a collection of
	 * interface names with their associated (non-Java) endpoint interface
names
	 * separated by {@link #SEPARATOR} e.g.:<br>
	 * 'my.company.foo|MyWebService my.company.zoo|MyWebService'.<br>
	 * This (non-Java) endpoint interface name is usually a communication
	 * protocol specific interface, for instance a web service interface name.
	 * Though this information is usually contained in accompanying properties
	 * e.g. a wsdl file, Discovery usually doesn't read and interprets such
	 * service meta-data. Providing this information explicitly, might allow
	 * external non-Java applications find services based on this endpoint
	 * interface.
	 *
	 * Value of this property is of type Collection (<? extends String>).
	 */
	public static final String	PROP_KEY_ENDPOINT_INTERFACE_NAME    =
"osgi.remote.endpoint.interface";

	/**
	 * Optional ServiceRegistration property which contains a map of
properties
	 * of the published service. <br>
	 * Property keys are handled in a case insensitive manner (as OSGi
Framework
	 * does). <br>
	 * Value of this property is of type <code>java.util.Map<code>.
	 */
	public static final String	PROP_KEY_SERVICE_PROPERTIES
			= "service.properties";

	/**
	 * Optional property of the published service identifying its location.
	 * Value of this property is of type <code>java.net.URL<code>.
	 */
```

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

Philipp Konradi 4/7/09 4:11 PM
Formatted: Font:9 pt, Font color: Custom Color(RGB(63,95,191)), English (UK)

Philipp Konradi 4/7/09 4:11 PM
Formatted: Font:9 pt, English (UK)

Philipp Konradi 4/7/09 4:11 PM
Formatted: Font:9 pt, Font color: Custom Color(RGB(63,95,191)), English (UK)

Philipp Konradi 4/7/09 4:11 PM
Formatted: Font:9 pt, English (UK)

Philipp Konradi 4/7/09 4:11 PM
Formatted: Font:9 pt, Font color: Custom Color(RGB(63,95,191)), English (UK)

Philipp Konradi 4/7/09 4:11 PM
Formatted: Font:9 pt, English (UK)

Philipp Konradi 4/7/09 4:11 PM
Formatted: Font:9 pt, Font color: Custom Color(RGB(63,95,191)), English (UK)

Philipp Konradi 4/7/09 4:11 PM
Formatted: Font:9 pt, English (UK)

Philipp Konradi 4/7/09 4:11 PM
Formatted: Font:9 pt, Font color: Custom Color(RGB(63,95,191)), English (UK)

Philipp Konradi 3/5/09 4:11 PM
Formatted: Font:9 pt, English (UK)

Philipp Konradi 3/5/09 4:08 PM
Formatted: Font:9 pt

Philipp Konradi 3/5/09 4:11 PM
Formatted: Font:9 pt, English (UK)


```
        public static final String          PROP_KEY_ENDPOINT_LOCATION
                = "osgi.remote.endpoint.location";

        /**
         * Optional property of the published service uniquely identifying its
         * endpoint. Value of this property is of type <code>String<code>.
         */
        public static final String          PROP_KEY_ENDPOINT_ID
                = "osgi.remote.endpoint.id";

        /**
         * Separator constant for association of interface-specific values with
the
         * particular interface name. See also
         * {@link #PROP_KEY_SERVICE_INTERFACE_VERSION} and
         * {@link #PROP_KEY_ENDPOINT_INTERFACE_NAME} properties which describe
such
         * interface-specific values.
         */
        public static final String          SEPARATOR
                                            = "|";

}
```

5.3.2.3 DiscoveredServiceTracker interface

```
/**
 * Interface of trackers for discovered remote services. <br>
 * When such a service is registered with the framework, then {@link Discovery}
 * will notify it about remote services matching one of the provided criteria
 * and will keep notifying it on changes of information known to Discovery
 * regarding this services.
 *
 * <code>Discovery</code> may deliver notifications on discovered services to a
 * <code>DiscoveredServiceTracker</code> out of order and may concurrently call
 * and/or reenter a <code>DiscoveredServiceTracker</code>.
 *
 * @ThreadSafe
 * @version $Revision: 6686 $
 */
public interface DiscoveredServiceTracker {

        /**
         * Optional ServiceRegistration property which contains service interfaces
         * this tracker is interested in. Value of this property is of type
         * Collection (<? extends String>). <br>
         * Property is optional, may be null.
         */
        public static final String          PROP_KEY_MATCH_CRITERIA_INTERFACES  =
"osgi.discovery.interest.interfaces";

        /**
         * Optional ServiceRegistration property which contains filters for
services
         * this tracker is interested in. <br>
         * Note that these filters need to take into account service publication
         * properties which are not necessarily the same as properties under which
a
         * service is registered. See {@link ServicePublication} for some standard
         * properties used to publish service metadata. <br>
         * The following sample filter will make Discovery notify the
         * DiscoveredServiceTracker about services providing interface
         * 'my.company.foo' of version '1.0.1.3':<br>
```

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

Philipp Konradi 3/5/09 4:08 PM
Formatted: Font:9 pt

Philipp Konradi 3/5/09 4:07 PM
Deleted:

```
/**
 * Register a service implementing the <code>ServicePublication</code> interface
 * in order to publish metadata of a particular service (endpoint) via
 * Discovery. Metadata which has to be published is given in form of properties
 * at registration. <br>
 * In order to update published service metadata, update the properties
 * registered with the <code>ServicePublication</code> service. Depending on
 * Discovery's implementation and underlying protocol it may result in an update
 * or new re-publication of the service. <br>
 * In order to unpublish the previously published service metadata, unregister
 * the <code>ServicePublication</code> service.<br>
 *
 * Please note that providing the {@link #PROP_KEY_SERVICE_INTERFACE_NAME}
 * property is mandatory when
```

... [1]

Philipp Konradi 3/5/09 4:09 PM
Formatted: Font:9 pt

Philipp Konradi 3/5/09 4:11 PM
Formatted: Font:9 pt, English (UK)

Philipp Konradi 4/7/09 4:12 PM
Formatted: Font:9 pt, Font color: Custom Color(RGB(63,95,191)), English (UK)

Philipp Konradi 4/7/09 4:12 PM
Formatted: Font:9 pt, English (UK)

Philipp Konradi 4/7/09 4:12 PM
Formatted: Font:9 pt, Font color: Custom Color(RGB(63,95,191)), English (UK)

Philipp Konradi 4/7/09 4:12 PM
Formatted: Font:9 pt, English (UK)

Philipp Konradi 4/7/09 4:12 PM
Formatted: Font:9 pt, Font color: Custom Color(RGB(63,95,191)), English (UK)

Philipp Konradi 4/7/09 4:12 PM
Formatted: Font:9 pt, English (UK)

Philipp Konradi 4/7/09 4:12 PM
Formatted: Font:9 pt, Font color: Custom Color(RGB(63,95,191)), English (UK)

Philipp Konradi 4/7/09 4:12 PM
Formatted: Font:9 pt, English (UK)

```
	 *
"(&(service.interface=my.company.foo)(service.interface.version=my.company.foo|1.0.1.3
))". <br>
	 * Value of this property is of type Collection (<? extends String>).
	 * Property is optional, may be null.
	 */
	public static final String		PROP_KEY_MATCH_CRITERIA_FILTERS
	= "osgi.discovery.interest.filters";

	/**
	 * Receives notification that information known to Discovery regarding a
	 * remote service has changed. <br>
	 * The tracker is only notified about remote services which fulfill the
	 * matching criteria, either one of the interfaces or one of the filters,
	 * provided as properties of this service. <br>
	 * If multiple criteria match, then the tracker is notified about each of
	 * them. This can be done either by a single notification callback or by
	 * multiple subsequent ones.
	 *
	 * @param notification the <code>DiscoveredServiceNotification</code>
object
	 *        describing the change.
	 */
	void serviceChanged(DiscoveredServiceNotification notification);
}
```

5.3.2.4 DiscoveredServiceNotification interface

```
/**
 * Interface for notification on discovered services.
 *
 * <code>DiscoveredServiceNotification</code> objects are immutable.
 *
 * @Immutable
 * @version $Revision: 6686 $
 */
public interface DiscoveredServiceNotification {

	/**
	 * Notification indicating that a service matching the listening criteria
has been
	 * discovered.
	 * <p>
	 * The value of <code>AVAILABLE</code> is 0x00000001.
	 */
	public final static int AVAILABLE				= 0x00000001;

	/**
	 * Notification indicating that the properties of a previously discovered
service
	 * have changed.
	 * <p>
	 * The value of <code>MODIFIED</code> is 0x00000002.
	 */
	public final static int MODIFIED				= 0x00000002;

	/**
	 * Notification indicating that a previously discovered service is no
longer known
	 * to discovery.
	 * <p>
	 * The value of <code>UNAVAILABLE</code> is 0x00000004.
```

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

Philipp Konradi 3/5/09 4:09 PM
Formatted: Font:9 pt

Philipp Konradi 3/5/09 4:10 PM
Formatted: Font:9 pt, English (UK)

Philipp Konradi 3/5/09 4:09 PM
Formatted: Font:9 pt

Philipp Konradi 3/5/09 4:09 PM
Deleted: /** ¶
* Interface of trackers for discovered remote services.
 ¶
* When such a service is registered with the framework, then {@link Discovery} ¶
* will notify it about remote services matching one of the provided criteria ¶
* and will keep notifying it on changes of information known to Discovery ¶
* regarding this services. ¶
* ¶
* <code>Discovery</code> may deliver notifications on discovered services to a ¶
*
<code>DiscoveredServiceTracker</code> out of order and may concurrently call ¶
* and/or reenter a <code>DiscoveredServiceTracker</code>. ¶
* ¶
* @version $Revision: 6037 $ ¶
*/ ¶
public interface
DiscoveredServiceTracker { ¶
¶
→ /** ¶
→ * Property describing service interfaces this tracker is interested in. ¶
→ * Value of this property is ... [2]

Philipp Konradi 3/5/09 4:10 PM
Formatted: Font:9 pt

Philipp Konradi 3/5/09 4:10 PM
Formatted: Font:9 pt, English (UK)

Philipp Konradi 4/7/09 4:13 PM
Formatted: Font:9 pt, Font color: Custom Color(RGB(63,95,191)), English (UK)

Philipp Konradi 4/7/09 4:13 PM
Formatted ... [3]

Philipp Konradi 4/7/09 4:13 PM
Formatted ... [4]

Philipp Konradi 4/7/09 4:13 PM
Formatted ... [5]

Philipp Konradi 4/7/09 4:13 PM
Formatted ... [6]

Philipp Konradi 3/5/09 4:10 PM
Formatted: Font:9 pt, English (UK)

```
     */
    public final static int UNAVAILABLE                    = 0x00000004;

    /**
     * Notification indicating that the properties of a previously discovered
service
     * have changed and the new properties no longer match the listener's
     * filter.
     * <p>
     * The value of <code>MODIFIED_ENDMATCH</code> is 0x00000008.
     */
    public final static int MODIFIED_ENDMATCH       = 0x00000008;

    /**
     * Returns information currently known to Discovery regarding the service
     * endpoint.
     * <p>
     *
     * @return metadata of the service this Discovery notifies about.
     */
    ServiceEndpointDescription getServiceEndpointDescription();

    /**
     * Returns the type of notification. The type values are:
     * <ul>
     * <li>{@link #AVAILABLE}</li>
     * <li>{@link #MODIFIED}</li>
     * <li>{@link #MODIFIED_ENDMATCH}</li>
     * <li>{@link #UNAVAILABLE}</li>
     * </ul>
     *
     * @return Type of notification regarding known service metadata.
     */
    int getType();

    /**
     * Returns interface name criteria of the {@link DiscoveredServiceTracker}
     * object matching with the interfaces of the ServiceEndpointDescription
and
     * thus caused the notification.
     *
     * @return matching interface name criteria of the
     *         {@link DiscoveredServiceTracker} object being notified.
     */
    Collection/* <String> */getInterfaces();

    /**
     * Returns filters of the {@link DiscoveredServiceTracker} object matching
     * with the ServiceEndpointDescription and thus caused the notification.
     *
     * @return matching filters of the {@link DiscoveredServiceTracker} object
     *         being notified.
     */
    Collection/* <String> */getFilters();

}
```

5.3.2.5 ServiceEndpointDescription interface

```
/**
 * This interface describes an endpoint of a service. This class can be
 * considered as a wrapper around the property map of a published service and
 * its endpoint. It provides an API to conveniently access the most important
```

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

Philipp Konradi 3/5/09 4:10 PM
Formatted: Font:9 pt, Italian

Philipp Konradi 3/5/09 4:10 PM
Formatted: Font:9 pt, English (UK)

Philipp Konradi 3/5/09 4:10 PM
Formatted: Font:9 pt, Italian

Philipp Konradi 3/5/09 4:10 PM
Formatted: Font:9 pt, English (UK)

Philipp Konradi 3/5/09 4:10 PM
Formatted: Font:9 pt

Philipp Konradi 3/5/09 4:10 PM
Deleted:

```
/**
 * Interface for notification on
discovered services.
 *
 * @version $Revision: 5970 $
 */
public interface
DiscoveredServiceNotification {

    /**
     * Notification indicating that a
service matching the listening
criteria has been
     * discovered.
     * <p>
     * The value of
<code>AVAILABLE</code> is
0x00000001.
     */
    public final static int AVAILABLE
= 0x00000001;

    /**
     * Notification indicating that
the properties of a previously
discovered service
     * have changed.
     * <p>
     * The value of
<code>MODIFIED</code> is
0x00000002.
     */
    public final static int MODIFIED
= 0x00000002;

    /**
     * Notification indicating that a
previously discovered service is no
longer known
     * to discovery.
     * <p>
     * The value of
<code>UNAVAILABLE</code> is
0x00000004.
     */
    public final static int
UNAVAILABLE = 0x00000004;

```

... [7]

Philipp Konradi 4/7/09 4:15 PM
Formatted: Font:9 pt

Philipp Konradi 4/7/09 4:15 PM
Formatted: Font:9 pt, English (UK)

```
 * properties of the service.
 *
 * <code>ServiceEndpointDescription</code> objects are immutable.
 *
 * @Immutable
 * @version $Revision: 6687 $
 */
public interface ServiceEndpointDescription {

		/**
		 * Returns the value of the property with key
		 * {@link ServicePublication#PROP_KEY_SERVICE_INTERFACE_NAME}.
		 *
		 * @return service interface names provided by the advertised service
		 *         (endpoint). The collection is never null or empty but contains
at
		 *         least one service interface.
		 */
		Collection /* <? extends String> */getProvidedInterfaces();

		/**
		 * Returns non-Java endpoint interface name associated with the given
		 * interface. Value of the property with key
		 * {@link ServicePublication#PROP_KEY_ENDPOINT_INTERFACE_NAME} is used by
		 * this operation.
		 *
		 * @param interfaceName for which its non-Java endpoint interface name
		 *        should be returned.
		 *
		 * @return non-Java endpoint interface name. Null, if it hasn't been
		 *         provided.
		 */
		String getEndpointInterfaceName(String interfaceName);

		/**
		 * Returns version of the given interface. Value of the property with key
		 * {@link ServicePublication#PROP_KEY_SERVICE_INTERFACE_VERSION} is used
by
		 * this operation.
		 *
		 * @param interfaceName for which its version should be returned.
		 * @return Version of given service interface. Null, if it hasn't been
		 *         provided.
		 */
		String getVersion(String interfaceName);

		/**
		 * Returns the value of the property with key
		 * {@link ServicePublication#PROP_KEY_ENDPOINT_LOCATION}.
		 *
		 * @return The URL of the service location. Null, if it hasn't been
		 *         provided.
		 */
		URL getLocation();

		/**
		 * Returns the value of the property with key
		 * {@link ServicePublication#PROP_KEY_ENDPOINT_ID}.
		 *
		 * @return Unique id of service endpoint. Null, if it hasn't been
provided.
		 */
		String getEndpointID();
```

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

Philipp Konradi 4/7/09 4:15 PM
Formatted: Font:9 pt

Philipp Konradi 4/7/09 4:15 PM
Formatted: Font:9 pt, English (UK)

```
        /**
         * Getter method for the property value of a given key.
         *
         * @param key Name of the property
         * @return The property value, null if none is found for the given key
         */
        Object getProperty(String key);

        /**
         * @return a <code>java.util.Collection</code> of property names available
         *         in the ServiceEndpointDescription. The collection is never null
         *         or empty but contains at least names of mandatory
         *         <code>ServicePublication</code> properties. Since
         *         <code>ServiceEndpointDescription</code> objects are immutable,
         *         the returned collection is also not going to be updated at a
         *         later point of time.
         */
        Collection/* <? extends String> */getPropertyKeys();

        /**
         * @return all properties of the service as a <code>java.util.Map</code>.
         *         The map is never null or empty but contains at least mandatory
         *         <code>ServicePublication</code> properties. Since
         *         <code>ServiceEndpointDescription</code> objects are immutable,
         *         the returned map is also not going to be updated at a later
point
         *         of time.
         */
        Map/* <String, Object> */getProperties();
}
```

5.3.3 Discovery using a local file(s)

The requirement for providing discovery type information in local files comes from the 'I want to connect to a Google Service' use-case. Basically, a mechanism is needed to specify the details of a service without access to an external discovery service.

To satisfy this requirement, an optional mechanism can be provided that uses resources inside a bundle providing Discovery type information. This mechanism is often provided by the Distribution Software, but can also be provided by another entity.

The mechanism checks bundles for the existence of xml files in the `OSGI-INF/remote-service` directory and if found it provides the information in these files to the Distribution Software, similar to how Discovery provides this information to the DSW.

The local files can be used to statically configure a client in case of not using the discovery model to share service information and distribute this information in a controlled way based on upfront knowledge of services required by a node.

The following is an example of such a file which shows a single Distribution Software approach used by the initial Reference Implementation. Section (add ref to section called “Bindings in Service Descriptions”) shows an example of how a Distribution Software can also optionally choose to use SCA metadata.

```
<?xml version="1.0" encoding="UTF-8"?>
<service-descriptions xmlns="http://www.osgi.org/xmlns/sd/v1.0.0">
  <service-description>
    <provide interface="com.iona.soa.pojo.hello.HelloService"/>
```

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

Philipp Konradi 4/7/09 4:15 PM
Formatted: Font:9 pt

Philipp Konradi 4/7/09 4:14 PM
Deleted:

```
/**
 * This interface describes an endpoint of a service. This class can be
 * considered as a wrapper around the property map of a published service and
 * its endpoint. It provides an API to conveniently access the most important
 * properties of the service.
 *
 * @version $Revision: 6037 $
 */
public interface ServiceEndpointDescription {

    /**
     * Returns the value of the property with key
     * {@link ServicePublication#PROP_KEY_SERVICE_INTERFACE_NAME}.
     *
     * @return service interface names provided by the advertised service
     *         (endpoint). The collection is never null or empty but contains at
     *         least one service interface.
     */
    Collection /* <? extends String> */getProvidedInterfaces();

    /**
     * Returns non-Java endpoint interface name associated with the given
     * interface. Value of the property with key
     * {@link ServicePublication#PROP_KEY_ENDPOINT_INTERFACE_NAME} is used by
     * this operation.
     *
     * @param interfaceName
     *            for which its non-Java endpoint interface name should be
     *            returned.
     *
```

... [8]

Unknown
Field Code Changed

Philipp Konradi 3/5/09 4:10 PM
Formatted: French

Unknown
Field Code Changed

Philipp Konradi 3/5/09 4:10 PM
Formatted: French

```
    <property name="service.intents">SOAP HTTP</property>
    <property name="osgi.remote.configuration.type">pojo</property>
    <property name="osgi.remote.configuration.pojo.address">
        http://localhost:9000/hello
    </property>
  </service-description>
  <service-description>
    <provide interface="com.iona.soa.pojo.hello.GreeterService"/>

    <property name="service.intents">SOAP HTTP</property>
    <property name="osgi.remote.configuration.type">pojo</property>
    <property name="osgi.remote.configuration.pojo.address">
        http://localhost:9005/greeter
    </property>
  </service-description>
</service-descriptions>
```

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

Tim Diekmann 2/20/09 1:25 PM
Deleted: osgi

Tim Diekmann 2/20/09 1:25 PM
Deleted: osgi

The XML files use the http://www.osgi.org/xmlns/sd/v1.0.0 namespace which can be found in section 5.8.

The solution should use a folder named "`OSGI-INF/remote-service`" and parse all files with the *.xml extension in this folder per default (i.e. adopt the Extender model).

The location for the service description folder and individual files within it can be overridden by a specific Manifest header named "`Remote-Service`". Multiple clauses in this header are allowed (comma separated). The format of the header should follow the `Bundle.findEntries()` approach. So the default value of this property would be `/OSGI-INF/remote-service/*.xml`. An example possible user-provided value for this property could be:

```
/META-INF/osgi/services.remote,/MyDirectory/osgi/*.xml
```

This matches a single file called services.remote in the /META-INF/osgi directory plus all files ending with *.xml in the /MyDirectory/osgi folder.

Besides being in a separate file, the static configuration as described here could also be part of a larger XML file. In that case the parser should ignore elements not part of the http://www.osgi.org/xmlns/sd/v1.0.0 namespace.

5.4 Service Registry Hooks

5.4.1 Registration of Remote Services in Local Service Registry

In the OSGi specification R4.1 the Service Registry serves as a central entity where one could register (locally available) services as well as search for them. Reusing the same mechanism for remote services would help to stay as much as possible in the established OSGi programming model and hence help developers in adopting the new capabilities coming with RFC 119. Using the Service Registry for both, local and remote, services offers also a certain degree of transparency for service providers and consumers.

The implementation of RFC 119 uses the ListenerHook as defined in RFC 126. This allows the distribution software to be informed when a consumer is looking for a service that potentially is not available in the local container (yet) and may therefore be discovered in the network.

5.4.2 Additional filtering

If additional filtering to what service consumers specify in their LDAP filter is required, this can be achieved by installing a FindHook as described in RFC 126. A FindHook allows the implementor to

restrict the visible set of services for one or more bundles. A possible use for the FindHook is to prevent a particular bundle from seeing remote services if use of remote services is not desired for this bundle.

For further details regarding the specification of this Service Registry Hook see RFC 126.

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

5.5 Service Programming Model

Sharing of a common service contract between service consumers and providers is fundamental for their interaction. Typically a service contract consists of two parts:

- Description of the functionality the service provider offers. That's mostly expressed by a service interface description e.g. a Java interface and the service's documentation.
- Description of the non-functional or quality of service (QoS) requirements regarding the way the agreed functionality is provided e.g. data has to be encrypted, call semantics.

An important point for RFC119 is its explicit support for dynamic wiring. In contrast to static wiring, where the concrete communication partners as well as their service contract are known beforehand (at the latest at deployment time) dynamic wiring allows service consumers and providers to establish contracts at runtime based on some criteria e.g. interface, supported communication protocols, or a set of QoS requirements (typically expressed using intents). An actual service contract results from requirements of a service provider and consumer as well as from the capabilities of distribution software on both sides.

The following kinds of metadata have been defined for service contracts:

- Service interface – describes the functionality of a service.
- Properties – provide information about the service object.
- Intents – state abstract requirements on service provider and consumer capabilities.

The above metadata may be sufficient when using the same distribution software on both client and service provider. To facilitate portability of configuration and interoperability in the case where multiple Distribution Software implementations are deployed, a service can be optionally configured using additional SCA metadata (see section **Error! Reference source not found.**, **Error! Reference source not found.**). Other metadata types are also permitted (standard or proprietary), through an extensibility mechanism but their integration into OSGi is not defined.

Tim Diekmann 4/10/09 5:29 PM
Deleted: Error: Reference source not found

Tim Diekmann 4/10/09 5:29 PM
Deleted: Error: Reference source not found

5.5.1 Service interface description

The service interface description defines the functionality, which a service provides. A service interface is the most basic service metadata and has to be well known by both interaction partners.

For RFC 119 a service interface is defined using a Java interface. The Java interface is typically used to derive a Distribution Provider interface, and some restrictions on the Java interface are therefore necessary to ensure compatibility across multiple DSW types (see Section 5.8).

5.5.2 Properties

Property – properties are used to describe a service while registering it in OSGi service registry. For more details on service properties, please refer to OSGi 4.1 core specification chapter 5.2.5.

Service properties can be provided statically by the bundle implementation and/or dynamically as configuration data that is used during the service registration, for example using the Configuration Admin service of OSGi R4.

Note that the properties defined in this section are for use with remote services only.

5.5.2.1 Definition of new Properties

Any custom service property can easily be defined. Please refer for more details to OSGi 4.1 core specification chapter 5.2.5. These have no bearing on the distribution of a service.

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

Tim Diekmann 2/20/09 1:25 PM
Deleted: osgi

5.5.2.2 Standard Properties

- service.intents – an optional list of intents provided by the service. The property advertises capabilities of the service and can be used by the service consumer in the lookup filter to only select a service that provides certain qualities of service. The value of this property is of type String[] and has to be provided by the service as part of the registration, regardless whether it's a local service or a proxy. The value on the proxy is a union of the value specified by the service provider, plus any remote-specific intents (see orgi.remote.require.intents, below), plus any intents which the Distribution Software adds that describe characteristics of the Distribution being mechanism. Therefore the value of this property can vary between the client side proxy and the server side.
- osgi.remote.interfaces – ["*" | interface_name [, interface_name]*]: A distribution software implementation may expose a service for remote access, if and only if the service has indicated its intention as well as support for remote invocations by setting this service property in its service registration. The value of this property is of type String[]. If the list contains only one value, which is set to "*", all of the interfaces specified in the BundleContext.registerService() call are being exposed remotely. The value can also be set to a comma-separated list of interface names, which should be a subset of the interfaces specified in the registerService call. In this case only the specified interfaces are exposed remotely.
- osgi.remote.requires.intents – an optional list of intents that should be provided when remotely exposing the service. If a DSW implementation cannot satisfy these intents when exposing the service remotely, it should not expose the service. The value of this property is of type String[].
- osgi.remote – this property is set on client side service proxies registered in the OSGi Service Registry.
- osgi.remote.configuration.type – service providing side property that identifies the metadata type of additional metadata, if any, that was provided with the service, e.g. "sca". Multiple types and thus sets of additional metadata may be provided. The value of this property is of type String[].

Both the osgi.remote.interfaces and osgi.remote.requires.intents should be modifiable by the deployer after the service has been developed. This can either be done through the Configuration Admin Service or through another mechanism.

Only the osgi.remote.interfaces property is required.

The service.intents property optionally defines the QoS capabilities that a published service provides, and allows a service requester to filter services according to its desired QoS capabilities.

Tim Diekmann 2/20/09 1:26 PM
Deleted: osgi

The osgi.remote.configuration.type optionally defines portable metadata to address the requirement for consistency across multiple DSW types. Distributed OSGi specifies how to use SCA metadata for this purpose.

The following example illustrates a potential XML file that could be used by Declarative Services to register the properties for distributed OSGi capability. This file would be installed through a bundle and identified by the bundle's Service-Component manifest header:

```
<?xml version="1.0" encoding="UTF-8"?>
<component name="OrderBeerService">
  <implementation .../>
  <service>
    <provide interface="org.beer.OrderBeerService" >
      <property name="osgi.remote.interfaces">*</property>
      <property name="osgi.remote.requires.intents">
          confidentiality
```

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

```
      </property>
    <provide />
   </service>
</component>
```

The example illustrates the `osgi.remote.interfaces` and `osgi.remote.requires.intents` properties specified for the `BeerOrderService`, which are associated with its interface. The `confidentiality` intent specifies the capability of the service to support encryption, such as through HTTP or IIOP/SSL.

5.5.3 Intents

An *intent* is an abstraction of a distributed computing capability that can be used to provision and select services. It describes one or more requirements of a service provider or consumer on the distribution software serving them.

An intent is a high-level, generic statement of 'what' a consumer may require from a provider. An intent is also a statement of what can be required of a deployer by a developer. An intent is associated with a service during deployment and can be used as a filter by a service consumer during the service discovery operation. When using an intent to filter a service, the consumer expects that the DSW has implemented the specified intent. The definition of intents comes from SCA but OSGi developers can also define their own intents, and any intent can be mapped by a given DSW to a DSW specific mechanism to fulfill the intent. Examples include intents for a reliable communication protocol, secure transmission, or a specific binding type.

The intent syntax is defined by the Service Component Architecture (SCA) and is extensible. This RFC references the SCA intents, defines two intents, and describes how to define additional intents.

A service requester can use an intent to help selecting a compatible service provider, and a service provider can use an intent to provision and deploy a service that advertises the intent.

When the same type of distributed software system is configured for both requester and provider, the DSW is not required to use the SCA mechanisms for defining the concrete instantiation of the intents, as long as its abstract meaning can be fulfilled by the DSW using another, similar mechanism. For example, instead of using WS-Policy as SCA does, a CORBA DSW might use CORBA policies. Any distribution detail undefined through intents or additional metadata is left to Distribution Software's interpretation and its (default) configuration. For example, a provided service may state that it requires the service to be 'confidential'. The Distribution Software may choose to distribute the service using a SOAP-based protocol with encryption. In doing so, it may optionally add the 'soap' intent so that clients selecting based on the 'soap' intent will also match this service.

The advantage of this Intents-based approach is that designers and developers can easily state requirements on service exposure or service reference (proxy) without the need to understand the complexities of mechanisms actually provided by the distribution software.

Intents may be provided by the component designer or by the deployer through configuration, i.e. Configuration Admin service, and used by the requester to select a service. For example, if a service requester requires 'integrity', only services which have been given the integrity intent (and provisioned accordingly) will be returned by the distribution software.

Intents that are provided by a service are listed in a service property named `deployment.intents`.

Tim Diekmann 2/20/09 1:27 PM
Deleted: `osgi.intents`

5.5.3.1 Example of using Intents

The example below shows a Declarative Service component called BeerOrderService which exposes a service with an `org.beer.BeerOrderService` interface, and consumes a BeerWarehouseService with an `org.beer.BeerWarehouseService` interface.

The service property osgi.remote.requires.intents is set to specify that remote communication with the BeerOrderService component should provide '`confidentiality`' and each delivery should occur

'exactlyOnce' (no duplicate messages or dropped messages). This might be desirable in order to prevent people snooping on messages and in order to ensure that orders are not lost or duplicated. These intents would typically be implemented using encryption and a reliable transport, respectively.

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

The example also shows how the component uses a `BeerWarehouseService` which is required to be available over a transport which assures messages are delivered '`exactlyOnce`'. This is expressed using the target filter "`(deployment.intents=exactlyOnce)`". Only a service which provides this intent will be injected for the BeerWarehouseService reference.

Tim Diekmann 2/20/09 1:27 PM
Deleted: `osgi.intents`

```
<?xml version="1.0" encoding="UTF-8"?>
<component name="BeerOrderService">
  <implementation .../>
  <service>
    <provide interface="org.beer.BeerOrderService" >
      <property name="osgi.remote.interfaces">*</property>
      <property name="osgi.remote.requires.intents">
        confidentiality
        exactlyOnce
      </property>
    <provide />
   </service>


  <reference .../>

  <reference name="BeerWarehouseService"
             interface="org.beer.BeerWarehouseService"
             target="(deployment.intents=exactlyOnce)" />
</component>
```

Tim Diekmann 2/20/09 1:27 PM
Deleted: `osgi.intents`

5.5.3.2 Defining Intents

An intent is a string with an associated abstract meaning. Their definition can be as simple as choosing a string name and documenting its meaning so that it can be shared between the various roles involved in creating the distributed system. Any user of Distributed OSGi is free to define their own intents using the mechanism defined in Section 5.7.6.

5.5.3.3 OSGi-defined Intents

OSGi defines two intents, `passByReference` and `passByValue` to allow services and clients to specify which type of call semantics they require.

- `passByReference` - states that the service requires pass-by-reference semantics. This restricts the subset of usable bindings to those that support pass-by-reference semantics, such as RMI.
- `passByValue` – states that the service requires pass-by-value semantics. This restricts the subset of usable bindings to those that support pass-by-value semantics.

When neither of these intents is used, then `passByValue` semantics are assumed, and a Distribution Software which publishes the service must ensure it adds this intent to the `ServiceEndpointDescription` to allow clients to explicitly select on it.

5.5.3.4 SCA-defined Intents

SCA defines a set of intents (strings and their associated abstract meaning). OSGi re-uses these intent definitions where appropriate (e.g. for defining service contracts QoS such as '`confidentiality`', or specific protocol requirements such as '`soap.1_1`').

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

SCA also defines a schema for adding new intents, the use of which is described in the section on Service Distribution using SCA Metadata (see section 5.7). A Distribution Software may choose to use this schema as a mechanism for adding new intent definitions. The types of intent relationships that schema allows and how they are exploited in Distributed OSGi are described below.

Below are the set of intents defined by the SCA Policy Framework specification Working Draft 09. Note, the '.' is used to define 'qualified intents' which are described in more detail in section 5.5.3.10. The expectation is this list will be updated to the most current document agreed by the SCA Policy TC (either a Community Draft or Public Review Draft) for inclusion in the final OSGi compendium specification.

5.5.3.5 Security Intents

authentication: the authentication intent is used to indicate that a client must authenticate itself in order to use a service. Typically, the client security infrastructure is responsible for the server authentication in order to guard against a "man in the middle" attack.

authentication.message: indicates that authentication should be realized at the message level of the communication.

authentication.transport: indicates that authentication should be realized at the transport layer of the communication.

confidentiality: the confidentiality intent is used to indicate that the contents of a message are accessible only to those authorized to have access (typically the service client and the service provider). A common approach is to encrypt the message, although other methods are possible.

confidentiality.message: indicates that confidentiality should be realized at the message level of the communication.

confidentiality.transport: indicates that confidentiality should be realized at the transport layer of the communication.

integrity: the integrity intent is used to indicate that assurance is required that the contents of a message have not been tampered with and altered between sender and receiver. A common approach is to digitally sign the message, although other methods are possible.

integrity.message: indicates that integrity should be realized at the message level of the communication.

integrity.transport: indicates that integrity should be realized at the transport layer of the communication.

5.5.3.6 Reliable Messaging Intents

atLeastOnce: the atLeastOnce intent is used to assure that a message that is successfully sent by a service consumer is delivered to the destination (i.e. service implementation). The message could be delivered more than once to the service implementation. A message that is successfully sent by a service implementation is also assured to be delivered to the destination (i.e. service consumer). The message could be delivered more than once to the service consumer.

atMostOnce: the atMostOnce intent is used to assure that a message that is successfully sent by a service consumer is not delivered more than once to the service implementation, however, the message is not guaranteed to be delivered to the service implementation. A message that is successfully sent by a service implementation is also assured not to be delivered more than once to the service consumer. The binding implementation does not guarantee that the message is delivered to the service consumer.

exactlyOnce: the exactlyOnce intent assures that a message sent by a service consumer is delivered to the service implementation. It also assures that the message is not delivered more than once to the service implementation. A message sent by a service implementation is also assured delivered to the service consumer. It also assures that the message is not delivered more than once to the service consumer.

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

ExactlyOnce is a profile intent of atLeastOnce and atMostOnce, the combination of which results in the exactlyOnce semantic.

ordered: the ordered intent assures that messages are delivered to the service implementation in the order in which they were sent by the service consumer. This intent does not guarantee that messages that are sent by a service consumer are delivered to the service implementation. Messages are also assured to be delivered to the service consumer in the order in which they were sent by the service implementation. This intent does not guarantee that messages that are sent by the service implementation are delivered to the service consumer.

5.5.3.7 Transactional Intents

propagatesTransaction: the propagatesTransaction intent indicates that the OSGi runtime must ensure that if the client of a service is running under a transaction, then that transaction context must be transmitted along with the invocation of an operation of the service and that the service invocation is dispatched under any propagated client transaction. Use of the propagatesTransaction intent implies that the Distribution Software must be capable of sending and receiving a transaction context and that a service with this intent specified will always join a propagated transaction, if present.

suspendsTransaction: the suspendsTransaction intent indicates that the OSGi runtime must ensure the service client must not send any transaction context with any service invocation and that the service is not dispatched under any propagated client transaction.

5.5.3.8 Miscellaneous Intents

SOAP: the SOAP intent specifies that the SOAP messaging model should be used for delivering messages. It does not require the use of any specific transport technology for delivering the messages, so for example, this intent can be supported by a binding that sends SOAP messages over HTTP, over bare TCP or over JMS. If the intent is required is an unqualified form of the intent, then any version of SOAP is acceptable.

SOAP.1_1: the SOAP.1_1 intent specifies the use of SOAP version 1.1 only. SOAP versions are mutually exclusive (e.g. it is an error to specify both SOAP.1_1 and SOAP.1_2).

SOAP.1_2: the SOAP.1_2 intent specifies the use of SOAP version 1.2 only. SOAP versions are mutually exclusive (e.g. it is an error to specify both SOAP.1_1 and SOAP.1_2).

JMS: The JMS intent does not specify a wire-level transport protocol, but instead requires that whatever binding technology is used, the messages should be able to be delivered and received via the JMS API.

NoListener: the NoListener intent is only applicable to a service reference. It indicates that the client is not able to handle new inbound connections. It requires that the Distribution Software be configured so that any response (or callback) comes either through a back channel of the connection from the client to the server or by having the client poll the server for messages.

Note, this intent does not follow the case convention of starting with lower-case and is expected to change to “noListener”.

5.5.3.9 Not Applicable Intents

In addition to the intents described above, SCA defines intents which cover advanced interaction patterns, such as Conversations (stateful protocol-style interactions) as well as service implementation requirements on a transactional runtime. These intents are not applicable to Distributed OSGi, but are listed for completeness. An OSGi-based runtime which supports these capabilities should consider re-using these intents where applicable:

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

Conversational: the conversation intent is used to indicate that a service is conversational. A conversational service is one which is stateful and may have a defined ordering in which service methods should be called.

Note, this intent does not follow the case convention of starting with lower-case and is expected to change to “conversational”. This has been raised with the Policy TC co-chair.

managedTransaction: the managedTransaction intent states that the service implementation requires a managed transaction environment in order to run. The specific type of managedTransaction required is not constrained. The valid qualifiers for this intent are mutually exclusive

managedTransaction.local: the managedTransaction.local intent states that the service implementation cannot tolerate running as part of a global transaction, and will therefore run within a local transaction containment (LTC) that is started and ended by the OSGi runtime. Any global transaction context that is propagated to the hosting OSGi runtime must not be visible to the target service implementation. Any interaction under this policy with a resource manager is performed in an extended resource manager local transaction (RMLT). Upon successful completion of the invoked service method, any RMLTs are implicitly requested to commit by the OSGi runtime. Note that, unlike the resources in a global transaction, RMLTs so coordinated in a LTC may fail independently. If the invoked service method completes with a non-business exception then any RMLTs are implicitly rolled back by the OSGi runtime. In this context a business exception is any exception that is declared on the service interface and is therefore anticipated by the service implementation. Local transactions cannot be propagated outbound across remote service invocations.

managedTransaction.global: the managedTransaction.glocal intent states that the service implementation requires an atomic transaction in order to run. The OSGi runtime must ensure that a global transaction is present before dispatching any method on the service. The OSGi runtime uses any transaction propagated from the client or else begins and completes a new transaction. See the ***propagatesTransaction*** intent below for more details.

noManagedTransaction: the noManagedTransaction intent states that the service implementation runs without a managed transaction, under neither a global transaction nor an LTC. A transaction that is propagated to the hosting OSGi runtime must not be joined by the hosting runtime on behalf of this service implementation. When interacting with a resource manager under this policy, the application (and not the OSGi runtime) is responsible for controlling any resource manager local transaction boundaries, using resource-provider specific interfaces (for example a Java implementation accessing a JDBC provider must choose whether a Connection should be set to autoCommit(true) or else must call the Connection commit or rollback methods).

transactedOneWay: the transactedOneWay intent indicates that OneWay invocations should be performed under transactional control. When applied to a service reference indicates that any OneWay invocation messages must be transacted as part of a client global transaction. If the client is not configured to run under a global transaction or if the Distribution Software does not support transactional message sending, then this is a deployment error. When transactedOneWay is applied to a service this indicates that any OneWay invocation message must be received from the transport binding in a transacted fashion, under the target service’s global transaction. The receipt of the message from the Distribution Software is not committed until the service transaction commits; if the service transaction is rolled back the the message remains available for receipt under a different service transaction. If the service is not configured to run under a global transaction or if the Distribution Software does not support transactional message receipt, then this is a deployment error.

immediateOneWay: the immediateOneWay intent indicates that OneWay invocations should be performed outside transactional control. When applied to a service reference indicates that any OneWay invocation messages is sent immediately regardless of any client transaction. When applied to a service indicates that any OneWay invocation is received immediately regardless of any target service transaction. The outcome of any transaction under which an immediateOneWay message is processed has no effect on the processing (sending or receipt) of that message.

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

5.5.3.10 Qualified Intents

An intent and the meaning it conveys can be specialized using a concept known as 'qualified intents'.

Example: Intent 'confidentiality' can be further qualified by extending it to '`confidentiality.message`' and would mean that '`confidentiality`' should be realized at the message level of the communication protocol e.g. by encrypting the messages. An alternative specialization of the intent '`confidentiality`' might be '`confidentiality.transport`' meaning that confidentiality should be realized through an encrypted transport.

Since qualification of intents is a specialization an intent '`confidentiality.message`' always fulfills the intent '`confidentiality`' but not necessarily the other way round.

Qualification of intents is a recursive model so qualified intents may be qualified again e.g. '`confidentiality.message`' to '`confidentiality.message.body`'.

5.5.3.11 Publishing of Qualified Intents

When publishing a service with qualified intents, the Distribution Software must make sure to list all appropriate intents for service selection. There are two aspects to this:

1. If a service has originally provided a qualified intent, then the Distribution Software should list also all the more general intents. A qualified intent is a specialization which means that a client looking for the more general intents should find a match. For example, '`confidentiality.message`' would be published as '`confidentiality confidentiality.message`' so that a client which does not care how confidentiality is provided will match the service which specifically provides it through the messages.
2. If a service has originally provided a general intent and Distribution Software has implemented it according to a qualified version of that intent then it should list also all the applicable qualified intents in addition to the original general intent. For example service which initially stated 'confidentiality' should be published as '`confidentiality confidentiality.message`' if the Distribution Software implemented '`confidentiality`' at the message level. So clients looking directly for qualified intents can also be served.

5.5.3.12 Profile Intents

A *profile intent* is an intent which is defined in terms of a set of intents. Using a profile intent is semantically equivalent to specifying all of the intents in the set. Profile intents are a convenience mechanism and remove the need to repeatedly specify the same intent sets in a solution.

The following example shows a new profile intent called '`communicationProtection`' which combines the '`confidentiality`' and '`integrity`' intents. Its purpose is to ensure that communications cannot be viewed or tampered with:

```
<intent name="communicationProtection"
 constrains="binding"
 requires="confidentiality integrity">
 <description>
  Ensure that communications cannot be seen or tampered with by
  unauthorized personnel.
 </description>
</intent>
```

5.5.3.13 Publishing of Profile Intents

When publishing a service with a profile intent, the Distribution Software must make sure to list all appropriate intents for service selection. For example, the profile intent '`communicationProtection`'

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

would be published as '`communicationProtection confidentiality confidentiality.transport confidentiality.message integrity integrity.message integrity.transport`'. Note, confidentiality and integrity are qualifiable intents and are therefore published according to the rules for qualified intents, described in Section 5.5.3.10.

If a Distribution Software natively understands intents then it must be configured with the profile intent definition. This enables the Distribution Software to know when all intents have been satisfied. If a Distribution Software does not natively understand intents, it is the responsibility of the deployer to ensure the Distribution Software in configured appropriately in order to satisfy the intents.

5.5.3.14 Exclusive Intents

Because intents are an expression of a service characteristic or requirement it is possible that two intents may be mutually exclusive and should not be used together. The existence of mutually exclusive intents on the same service is considered a deployment error. If a Distribution Software understands that two intents are mutually exclusive then it should not distribute the service.

5.5.4 Configuration type

The configuration type identifies the metadata used to describe additional DSW capabilities beyond intents, such as explicit communication protocol and data format bindings and quality of service policies. The main example in RFC 119 is SCA, but since RFC 119 is designed to support multiple DSW types, other metadata can be used and associated with additional configuration types.

The configuration type is specified using a property named `osgi.remote.configuration.type`. The property is set to string values which identify the type of additional metadata used. For example, the following shows how to specify that the additional metadata is in the form of SCA:

```
osgi.remote.configuration.type=sca
```

The configuration type is extensible so that other standard or proprietary types may be used. For example, the Foo Corporation may choose,

```
osgi.remote.configuration.type=foocorp
```

The configuration type property can be an array to allow multiple configuration types to be specified for the same service. For `example,osgi.remote.configuration.type=sca foocorp`

A naming convention is used to specify the service properties which provide the additional configuration. This follows the form `osgi.remote.configuration.<type string>.<sub-properties>`, so for foocorp, this would be,

```
osgi.remote.configuration.foocorp=...
```

It is recommended that each configuration type use sub-property names for their additional configuration as this is likely to make the properties more meaningful and allows multiple independent properties to be specified. For example Foo Corp may require two items of additional configuration, which could be specified as,

```
osgi.remote.configuration.foocorp.config1=...
osgi.remote.configuration.foocorp.config2=...
```

In general, metadata in a configuration type can be used to create a machine-readable description of a remoted service. This description can be compared with other descriptions for compatibility (i.e. is the service provider compatible with the service requester). That is, do they support the same communication protocols, encryption mechanisms, etc?

Service descriptions can be matched for compatibility initially at the intent level (i.e. intents can be compared for compatibility) but if the configuration type property is present and indicates additional metadata is available, it should be possible to perform an additional level of comparison for compatibility on the additional metadata.

If multiple matches are returned, matches based on intent properties are ranked lower than matches found using additional metadata. It should also be possible to rank services using a comparator.

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

5.5.5 Service Factories

In a non-distributed case Service Factories return a separate Service instance per consuming bundle.

For distributed OSGi Services, this behaviour is not extended into the remote case. For remoted services implemented using a ServiceFactory, the behaviour when a consumer calls the remote service by invoking on its remote endpoint should be similar to the Distribution Software bundle calling `Bundle.getBundleContext().getService()`. This means that a single instance of the remoted service will be used to serve remote invocations on it.

5.6 Collaboration of new and changed entities

In the following the interaction between service providers, consumers, distribution software and discovery is illustrated. Though involvement of discovery by distribution software is optional it's shown here for illustration reasons.

5.6.1 Interactions on the service provider side

5.6.1.1 Exposing a Service remotely

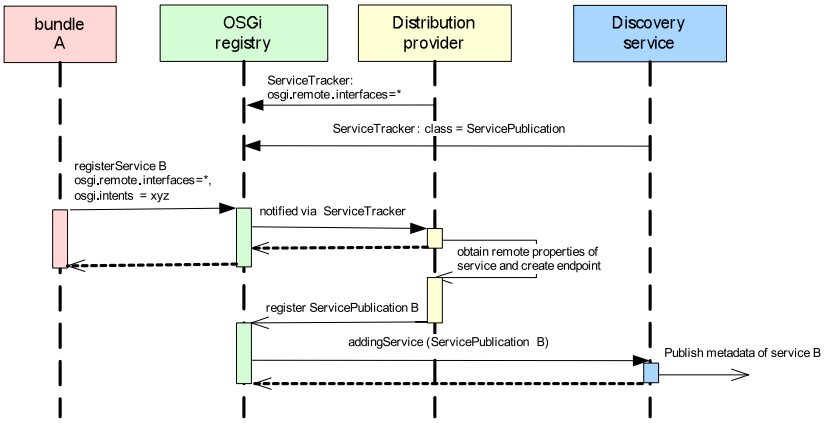


Figure 15: Server side service registration

Tim Diekmann 4/10/09 5:29 PM
Deleted: *10*

How the service B is registered in the OSGi framework and then made available for remote access is shown in **Error! Reference source not found.**. The important part in the picture is that service B is augmented with additional properties, which describe which interface of the service should be remotely exposed, required QoS, etc. (see section 5.5 for details about service metadata). This enables the distribution software to pick the appropriate protocol for service exposure and create an endpoint. In an optional step all service metadata required to communicate with that endpoint is published using the Discovery service. This happens by registering a ServicePublication service carrying service metadata to be published and which will be picked up by any existing Discovery service. The sequence diagram above shows the creation of only one endpoint by Distribution Provider, may be multiple, and publication by only one Discovery Service.

Tim Diekmann 4/10/09 5:29 PM
Deleted: Error: Reference source not found

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

5.6.1.2 Modification of service properties

The following diagram illustrates how Distribution software may react to modification of properties of a service which was exposed for remote access and published via Discovery.

The diagram shows only one possible scenario. Depending on which properties have changed and Distribution Providers implementation, it may choose not to update the existing endpoint but to create and publish a new one. The previous endpoint would be unpublished and destroyed in that case. Discovery service may also choose to make a new publication instead of updating an existing one depending on what the discovery protocol supports.

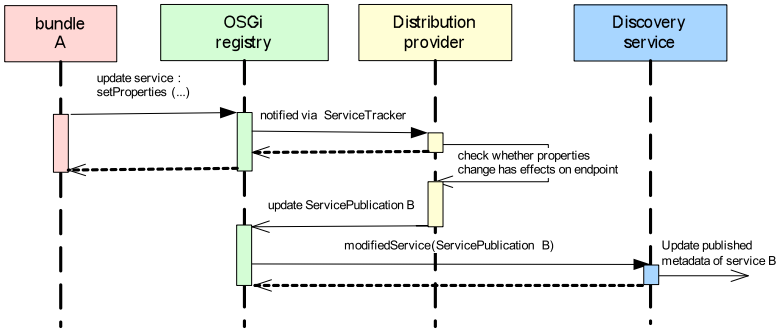


Figure 16 Update of properties of an exposed service

Tim Diekmann 4/10/09 5:29 PM
Deleted: *11*

5.6.1.3 Service Unregistration

The following figure depicts the flow of events in the case that a previously registered, remotely exposed and published service B is unregistered.

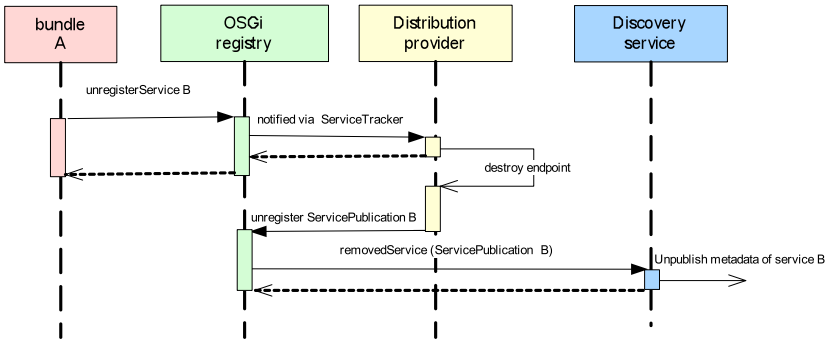


Figure 17: Unregistration of a service

Tim Diekmann 4/10/09 5:29 PM
Deleted: *12*

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

5.6.2 Interactions on the service consumer side

5.6.2.1 Lookup for a remote Service

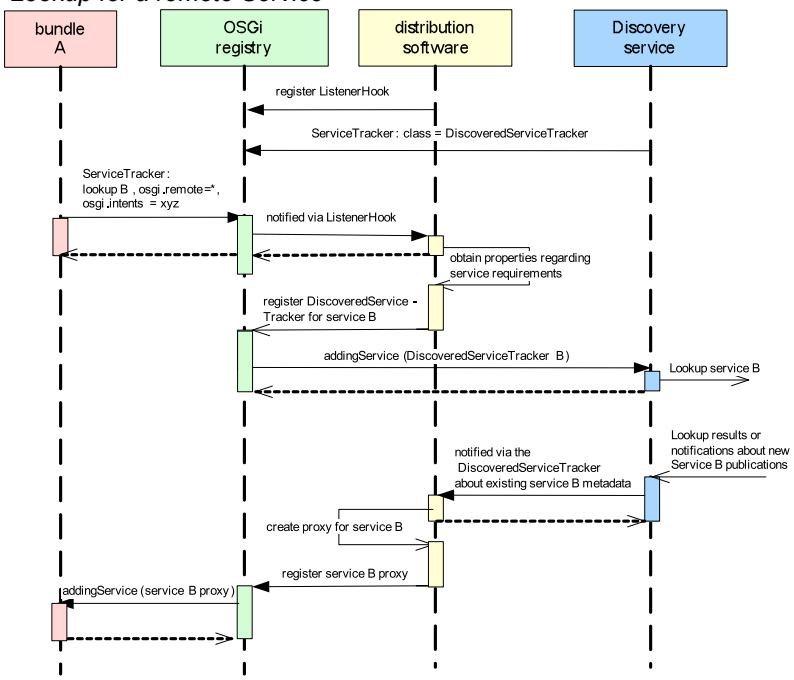


Figure 18: Client side service lookup

Tim Diekmann 4/10/09 5:29 PM
Deleted: *13*

In **Error! Reference source not found.** is shown what happens on the service consumer side, if the consumer is hosted in an OSGi platform. The distribution software is using the optional Discovery service to locate an implementation of service B, which satisfies the requirements specified in the service lookup. If found, the distribution software creates a proxy for the available protocol (binding) that implements the service interface as well as the required Qualities of Service. This proxy is then registered in the local OSGi service registry and thus is accessible by the service consumer. Proxy's properties reflect all concrete distribution-related service metadata like used binding, applied policies, service's host etc.

Tim Diekmann 4/10/09 5:29 PM
Deleted: Error: Reference source not found

On any subsequent lookup for the same interface this proxy may also be returned to other bundles provided that it's capable to fulfill their QoS requirements as well.

Note: The proxy implementation is entirely left to the distribution software.

5.6.2.2 Service invocation

Service invocation is exactly the way it is today with local OSGi services. The exception to this is that an invocation that goes to a remote service can potentially throw a new RuntimeException: `osgi.framework.ServiceException` with as exception type REMOTE in the event there is a problem with the remote invocation. This exception can wrap any distribution technology-specific exception. A technology-specific exception may also be thrown directly if Distribution Provider is configured in this way.

As the new exception is a RuntimeException, existing code is not required to check for it, however, distribution-aware code has the option to catch it and react appropriately.

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

5.6.2.3 Service Unregistration

The following figure depicts the flow of events in the case that a previously discovered and bound service B becomes unavailable. The scenario assumes that the Discovery Service is informed about the fact that the remote service is no longer a valid reference.

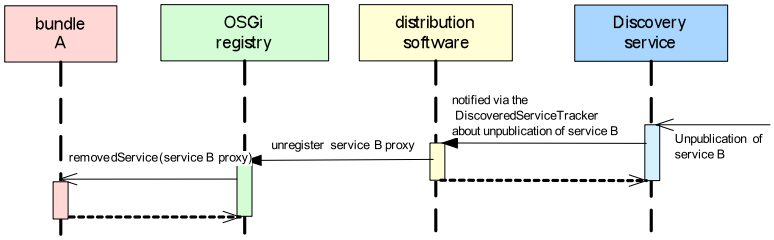


Figure 19: Unregistration of a service

Tim Diekmann 4/10/09 5:29 PM
Deleted: *14*

It is the Distribution Software's responsibility to ensure any proxies associated with unpublished services are unregistered.

Proxies to remote services which were configured from a locally installed bundle (e.g. through a local discovery service) are removed from the registry when the bundles which contributed them are stopped.

5.6.3 Interactions with Non-OSGi service providers and consumers

From interaction point of view there is no difference whether the other side uses OSGi or Non-OSGi technologies. Though it may put more restrictions on the exchanged data between service provider and consumer as well as Distribution Provider and Discovery e.g. no complex java objects.

5.6.4 Lifecycle dynamics

When a Distribution Provider is activated, it should:

- check for any services that are already registered in the OSGi Service Registry and have the property `osgi.remote.interfaces` specified and thus need to be exposed remotely.
- check for any active bundle which have unresolved references to remote services.

Before a Distribution Provider is deactivated, it should:

- unregister any proxies it has registered in the OSGi Registry (though Distribution Provider may rely on OSGi framework to unregister all Distribution Provider services when it's stopped).
- unpublish any services it has published to Discovery Services before (though Distribution Provider may rely on OSGi framework to unregister all Distribution Provider services when it's stopped). The Discovery Services will then propagate the change in availability of those particular services to other machines. This way other Distribution Providers consuming those services have a chance to gracefully react to this change.

When a new Discovery Service is activated, then it should:

- take care of already registered ServicePublication and DiscoveredServiceTracker services.

When a Discovery Service gets deactivated, then Distribution Providers should be aware that:

- They won't be notified about changes in availability or metadata of services previously located via this Discovery Service. If a previously discovered service does change or become unavailable, then the client will start to receive errors when trying to communicate with the service. Handling of this situation is application-dependent. For example, an application may choose to stop using a particular service after a number of failures.

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

- Any service metadata they published via this Discovery Service might still be remotely available e.g. because remote Discovery Services cached the information. So other remote Distribution Providers might still discover it and use to establish communication with the particular services. This means also that any Distribution Provider or client consuming remote services should be aware that the availability status of a service as well as the service metadata they get and use to choose appropriate services and to establish communication might be outdated and has changed already. This behavior is totally different from the local OSGi model where events regarding changes in service's lifecycle are guaranteed to be delivered and this in a synchronous way.

When a service changes its properties at runtime, then a Distribution Provider should check:

- Whether the property change affects the remote exposure of the service e.g. property *publish* was set to true and if yes then act as appropriate e.g. expose the service.
- Whether it's a service which is remotely exposed by this Distribution Provider and published to Discovery Services. In such a case the Distribution Provider should convey the change to other machines in the network and thus give them a chance to react to such a property change in a proper way e.g. the initial selection criteria may not be valid anymore. Distribution Provider may choose to republish the service with the new properties or to update the existing publication. This means also that it's not guaranteed that updating of a service will result in an modified event on the consumer side.

It is possible for the same distribution software to be configured to expose the service over multiple protocols, or for different distribution software types to expose the same service over the same or different protocols.

5.7 Service Distribution using SCA Metadata

Section 5.5.3 describes how intents are used to express distribution-related requirements. For example, a service provider can express the need for confidential communications using the 'confidentiality' intent, and a service client can use this same intent to select a service which provides 'confidentiality'. A deployer sees intent requirements and provisions the Distribution Software to support these (e.g. 'confidentiality' through an encrypted transport).

In section 5.5.3, the details of how intents are implemented are left up to the Distribution Software. This works well when the same Distribution Software is used at both ends of the communications, but when different Distribution Software systems are involved it is advantageous to use a standard mechanism for providing the additional configuration.

Distributed OSGi defines how to use SCA metadata for the concrete configuration of service distribution. It is not mandatory for a Distribution Software to support the use of this SCA metadata, but it is highly recommended as this enables greater portability and interoperability between Distribution Software types. There are a number of advantages to using the SCA metadata especially in deployments where different Distribution Software providers are involved. It enables portability of the metadata, which means a deployer does not need to learn new technologies in order to configure different Distribution Software types, and it also simplifies interoperability, where both parties involved understand the SCA metadata.

Any service using SCA metadata identifies this by setting the `osgi.remote.configuration.type` service property to "`sca`", as follows:

```
osgi.remote.configuration.type = sca
```

5.7.1 Bindings

SCA Bindings can be used to describe the access mechanism a client will use to call a service, or the access mechanism over which a service is made available. SCA defines a number of different binding types, for example, Web services, EJB, JMS, and also provides a mechanism for defining others.

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

5.7.1.1 Bindings on Services

A developer, or typically a deployer, can choose the bindings over which a service will be made available. This is done by setting the `osgi.remote.configuration.sca` service property to point to the detailed binding configuration. The value of this property is an array of URLs. If a URL does not have a protocol identifier, then it is assumed to be relative and refers to a resource within the contributing service's bundle. It is recommended that bindings located in a bundle be placed in the `OSGI-INF/bindings` folder. For example,

```
osgi.remote.configuration.sca.bindings =
  OSGI-INF/bindings/beer/OrderBeerService/bindings.xml
```

Where `OSGI-INF/bindings/beer/OrderBeerService/binding.xml` is an XML file in the bundle containing the definitions of one or more `binding` elements. The example below shows a document with two bindings:

```
<bindings>
   <binding.ws requires="soap.1_2" />
   <foocorp:binding.rmi />
</bindings>
```

Note, in this example, binding.rmi is a proprietary binding defined by foocorp and is therefore in the foocorp namespace (as denoted by the `foocorp` namespace prefix). See section 5.7.5**Error! Reference source not found.** for information on defining new binding types.

Tim Diekmann 4/10/09 5:29 PM
Deleted: Error: Reference source not found

A service configured with such a document would be made available via Web services (using soap 1.2) and RMI. The Distribution Software would expose the service at default endpoint URIs for both Web services and RMI. Bindings also allow specific endpoints URIs to be configured.

5.7.1.2 Bindings in Service Descriptions

A client component does not explicitly specify detailed distribution configuration for communicating with a service. The client will look up a service based on intents and business properties and then communicate with the matching services using whatever distribution configuration the target service published.

The description of the service with which the client will communicate is obtained via either Service Discovery or a static service description. Irrespective of the means by which the description is obtained, when SCA metadata is used, the description will contain properties for the detailed configuration of the target services. The following example illustrates this using static service descriptions as described in section 5.3.2.

Clients do not need to understand SCA metadata in order to be able to communicate with a service distributed using SCA metadata. For example, a service distributed using `<binding.ws />` can be called by a client using standard Web service technologies (WSDL, and soap/http).

```
<?xml version="1.0" encoding="UTF-8" ?>

<service-descriptions xmlns="http://www.osgi.org/xmlns/sd/v1.0.0">
  <service-description>
    <interface name="org.acme.MyReliableService" />
    <property key="osgi.remote.interfaces">*</property>
    <property key="osgi.remote.requires.intents">
      exactlyOnce reliable
    </property>
    <property key="osgi.remote.configuration.type">sca</property>
    <property key="osgi.remote.configuration.sca.bindings">
      OSGI-INF/bindings/MyReliableService/bindings.xml
    </property>
  </service-description>
  <service-description>
```

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

```
    <interface name="org.acme.MySecretService" />
    <property key="osgi.remote.interfaces">*</property>
    <property key="osgi.remote.requires.intents">
        confidential integrity
    </property>
    <property key="osgi.remote.configuration.type">sca</property>
    <property key="osgi.remote.configuration.sca.bindings">
      http://somemachine.acme.com:2808/com/acme/MySecretService/bindings.xml
    </property>
  </service-description>
</service-descriptions>
```

The property `osgi.remote.configuration.type=sca` identifies the service description as containing SCA metadata. The detailed binding information is located at the URL, or URLs, specified in the `osgi.remote.configuration.sca.bindings` property.

In the example above, the first service description is for a service available via bindings described in `OSGI-INF/bindings/MyReliableService/bindings.xml`. Note, these bindings will be configured to support the intents "reliable" and "exactlyOnce", specified in the service description's `osgi.remote.requires.intents` property. The second service description has bindings located on a separate machine, available via http, and supporting the "confidentiality" and "integrity" intents.

In both examples, the bindings must be fully configured. In other words, they must contain sufficient information for the Distribution Software to create a proxy to talk to the real service endpoint. For example, a service description (obtained through Service Discovery or static configuration) referencing the bindings described earlier in section 5.3.2 might result in the following fully configured bindings:

```
<bindings>
   <binding.ws uri="http://www.beercompany.org/BeerOrderService"
       requires="soap.1_2" />
   <foocorp:binding.rmi host="www.beercompany.org" port="8099"
       serviceName="BeerOrderService" />
</bindings>
```

In some cases it may be necessary to also include policy configuration through policySets. This is done by installing them separately as described in or referencing a definitions.xml file through a property called osgi.remote.configuration.sca.definitions. The format of this file is described in section 5.7.8 and the contents are the same as those described in section 5.7.4.

The combination of the service description and binding information is sufficient for a Distribution Software to create proxies for the remote service, one for Web services and one for RMI. These would be created and registered in the client framework's service registry. A client requesting a service with a filter which matches the service interface and service description properties would get one or both proxies from the service lookup.

5.7.2 PolicySets

PolicySets are used to define concrete policies that configure bindings. They configure a binding by providing a concrete definition of how an intent is to be fulfilled for the particular binding. PolicySets do not define their own policy language; they simple describe how to apply existing policy languages, such as WS-Policy, to bindings in support of specific intents.

The following example demonstrates this concept using the `OrderBeerService`, described earlier, and which specified intents of "`confidentiality`" and '`reliable`". A deployer might choose to configure a Web service binding to expose this service and need to configure that binding with the details of how to support "`confidentiality`" and "`reliable`". The following shows an example of how `confidentiality` might be configured:

```
<policySet name="SecurePolicy" provides="confidentiality.message"
  appliesTo="sca:binding.ws" xmlns="http://www.osoa.org/xmlns/sca/1.0"
```

```
  xmlns:wsp="http://schemas.xmlsoap.org/ws/2004/09/policy"
  xmlns:wssp="http://www.bea.com/wls90/security/policy">
  <wsp:Policy>
    <wssp:Confidentiality>
      <wssp:KeyWrappingAlgorithm
        URI="http://www.w3.org/2001/04/xmlenc#rsa-1_5" />
      <wssp:Target>
        <wssp:EncryptionAlgorithm
          URI="http://www.w3.org/2001/04/xmlenc#tripledes-cbc" />
        <wssp:MessageParts
          Dialect="http://schemas.xmlsoap.org/2002/12/wsse#part">
          wsp:Body()
        </wssp:MessageParts>
      </wssp:Target>
      <wssp:KeyInfo />
    </wssp:Confidentiality>
  </wsp:Policy>
</policySet>
```

The namespace of the policySet and the way in which PolicySets are deployed into the Distribution Software is described in section 5.7.8**Error! Reference source not found.**

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

Tim Diekmann 4/10/09 5:29 PM
Deleted: Error: Reference source not found

In this example, encryption is used to define how the intent "`confidentiality.message`" is implemented for the Web service binding (`binding.ws`). The rules for determining whether intents are satisifed by bindings and policySets are the same as those used to match client requirements to service providers (see 5.5.3). For example, the intent "`confidentiality`" can be satisfied by a policySet or bindings which implements a qualified version, in this case "`confidentiality.message`".

5.7.3 PolicySet Attachement

SCA defines two mechanisms for attaching policySets to bindings; direct attachment where the policySets are referenced from the bindings and external attachment where the policySets state which parts of SCA they apply to. OSGi makes use of the direct attachment approach only as this is the most appropriate to OSGi's use of SCA.

Direct attachment is done by referring to the policySet from a binding definition. This can be done either through an attribute or an element. The example below shows how the "SecurePolicy" is attached to the Web service binding using an attribute.

```
<binding.ws policySet="foocorp:SecurePolicy" />
```

The namspace of the policySet is the targetNamespace specified in the definitions file which provided the policySet (see section 5.7.8**Error! Reference source not found.**).

Tim Diekmann 4/10/09 5:29 PM
Deleted: Error: Reference source not found

Note, for a service to be properly configured, all intents must be satisfied either natively by the binding (e.g. "soap" supported by binding.ws) or by the attachment of policySets.

For more details on direct policy attachment, please refer to the SCA Policy specification.

5.7.4 Using the Discovery Service

A Distribution Software which supports the SCA metadata may also optionally use a Discovery Service. When publishing a services over Discovery, the Distribution Software should populate the ServiceEndpointDescription with the properties described in the <service-description /> elements above. Additionally, if any service descriptions require intent or policySet definitions, then an optional property called osgi.remote.configuration.type.sca.definitions should be set, which defines a URL to these definitions. For example,

osgi.remote.configuration.type.sca.definitions = http://somemachine.com/config/definitions.xml

The document available at the URL should be a definitions XML as defined in section 5.7.8Error! Reference source not found. below , which describes only those definitions which apply to the published service.

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

Tim Diekmann 4/10/09 5:29 PM
Deleted: Error: Reference source not found

When a Distribution Software receives a published service it is responsible for fetching the additional configuration via the URLs specified in the org.remote.configuration.type.sca.xxx properties. It must then ensure it can support the bindings intents and policySet specified. If the Distribution Software is capable of supporting these, then it registers a proxy representing the remote published service.

5.7.5 Defining New Binding Types

In addition to defining a set of Bindings, SCA also provides a means for defining new ones. For example, a Distribution Software may wish to support rmi by defining `<foocorp:binding.rmi />`. The details of how to define new bindings are described in the section on "Defining a Binding Type" in the SCA Assembly specification. Binding Types are contributed to an SCA metadata aware Distribution Software using a `definitions.xml` file as described in section 5.7.8.

5.7.6 Defining New Intents

Section 5.5.3 describes how intents are used in the OSGi service programming model. The concepts of qualified and profile intents are introduced. SCA provides a schema for defining intents, which an SCA metadata capable Distribution Software should use.

The SCA pseudo-schema for intent definition is as follows:

```
<intent name="xs:string" constrains ="list of QNames"
requires="list of QNames" excludes="list of QNames"?
mutuallyExclusive="boolean"?
   <description> xs:string.</description> ?
   <qualifier name = "xs:string"  default = "xs:boolean" ?>  *
       <description> xs:string.</description> ?
</qualifier>
</intent>
```

Where

- @name: specifies the name of the intent.
- @constrains: (optional) In SCA intents can apply to many specific SCA artifacts. In the OSGi usage of intents the relevant constrains value is 'sca:binding'. If this value is omitted then it is assumed that use of the intent is unrestricted.
- @requires: defines the set of all intents that the referring intent requires. This allows intents to be composed out of other intents (see section 5.5.3.12 on 'Profile Intents').
- @excludes: (optional) a list of intents that are incompatible with this intent. It is an error to deploy a service with incompatible intents. A Distribution Software should not distribute a service with incompatible intents.
- @mutuallyExclusive: (optional) 'true' signifies that the qualified intents are mutually exclusive. The default is 'false'.
- <qualifier>: (optional) used to define a qualifier for an intent (see section 5.5.3.10 on 'Qualified Intents').

The intent schema is described in more detail in the SCA Policy Specification.

The following example shows a new intent called '`communicationProtection`' which combines the '`confidentiality`' and '`integrity`' intents. Its purpose is to ensure the communications cannot be viewed or tampered with:

```
<intent name="communicationProtection"
  constrains="binding"
```

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

```
  requires="confidentiality integrity">
  <description>
    Ensure that communications cannot be seen or tampered with by
    unauthorized personnel.
  </description>
</intent>
```

5.7.7 Intents on Bindings

Intents can be directly configured on bindings. These are considered additions to those defined in the service property `osgi.remote.requires.intents`. This mechanism enables the deployer to provide additional restrictions which only apply to a particular binding on a service. For example, the following adds "`reliable`" to the Web service binding:

```
<binding.ws requires="reliable" />
```

The details on adding intents to bindings can be found in the SCA Policy specification.

5.7.8 Definition of Intents, Binding Types and PolicySets

Intents, policySets and binding types are not specific to a particular service implementation or client and are therefore provided separate from service and service description configuration. Definitions of these are provided to a Distribution Software in a definitions.xml file. The format of the file is defined by SCA and follows the following pseudo-schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://www.osoa.org/xmlns/sca/1.0"
           targetNamespace="xs:anyURI">
   <sca:intent/>*
   <sca:policySet/>*
   <sca:bindingType/>*
</definitions>
```

SCA defines other elements within the definitions element but these are not used in Distributed OSGi.

The targetNamespace defines the namspace in which the definitions belong, and is used to refer to them.

When a bundle containing a `definitions.xml` file is installed, the Distribution Software must read the file and react as follows:

- If there are services registered which refer to intents, policySets or binding types from the definitions.xml file, but which are not yet distributed, then the Distribution Software should use these definitions to distributed those services.
- If there are service descriptions (e.g service descriptions in a `<service-descriptions />` `xml` file, or as result from a discovery request) which refer to something defined in the definitions.xml, then the distribution software should use these to complete the detailed configuration of the description and create a proxy to the remote service.

When a bundle containing a `definitions.xml` file is uninstalled the Distribution Software should react as follows:

- If there are service endpoints available for services whose configuration depended on information in the `definitions.xml`, then those endpoints should be removed.
- If there are service proxies to remote services whose configuration depended on information in the `definitions.xml` file, then those proxy services should be remove from the service registry.

5.8 Service Descriptions XML schema

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
    /*
    * $Revision:$
    *
    * Copyright (c) OSGi Alliance (2008, 2008). All Rights Reserved.
    *
    * Licensed under the Apache License, Version 2.0 (the "License");
    * you may not use this file except in compliance with the License.
    * You may obtain a copy of the License at
    *
    *      http://www.apache.org/licenses/LICENSE-2.0
    *
    * Unless required by applicable law or agreed to in writing, software
    * distributed under the License is distributed on an "AS IS" BASIS,
    * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
implied.
    * See the License for the specific language governing permissions and
    * limitations under the License.
    */
-->
<schema xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:sd="http://www.osgi.org/xmlns/sd/v1.0.0"
    targetNamespace="http://www.osgi.org/xmlns/sd/v1.0.0"
    elementFormDefault="qualified"
    version="1.0.0">

    <annotation>
        <documentation xml:lang="en">
            This is the XML Schema for service descriptions used by
            Distributed OSGi. Service descriptions are used to describe
            remote services to a client. An extender, such as a local
            Discovery Service can look for service descriptions in
            installed bundles and then a Distribution Software can
            create proxies to the remote services based on the service
            descriptions.
        </documentation>
    </annotation>

    <element name="service-descriptions"
        type="sd:Tservice-descriptions" />

    <complexType name="Tservice-descriptions">
        <sequence>
            <element name="service-description"
                type="sd:Tservice-description" minOccurs="1"
maxOccurs="unbounded" />
                                <!-- It is non-deterministic, per W3C XML
Schema 1.0: http://www.w3.org/TR/xmlschema-1/#cos-nonambig
                                        to use namespace="##any" below. --
>
            <any namespace="##other" processContents="lax" minOccurs="0"
                maxOccurs="unbounded" />
        </sequence>
        <anyAttribute />
    </complexType>
```

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

Philipp Konradi 3/5/09 4:11 PM
Formatted: German

Philipp Konradi 3/5/09 4:11 PM
Formatted: German

Philipp Konradi 3/5/09 4:11 PM
Formatted: English (UK)

Philipp Konradi 3/5/09 4:11 PM
Formatted: English (UK)

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

```
    <complexType name="Tservice-description">
        <annotation>
            <documentation xml:lang="en">
                A Distribution Software is required to register a proxy
                with the interface(s) and any properties provided. If
                any 'intents' properties are specified then the
                Distribution Software should only register a proxy if it
                cann support those intents.
            </documentation>
        </annotation>
                        <sequence>
            <element name="provide" type="sd:Tprovide" minOccurs="1"
                maxOccurs="unbounded" />
            <element name="property" type="sd:Tproperty" minOccurs="0"
                maxOccurs="unbounded" />
                                <!-- It is non-deterministic, per W3C XML
Schema 1.0: http://www.w3.org/TR/xmlschema-1/#cos-nonambig
                                        to use namespace="##any" below. --
>
            <any namespace="##other" processContents="lax" minOccurs="0"
                maxOccurs="unbounded" />
        </sequence>
        <anyAttribute />
    </complexType>

          <complexType name="Tprovide">
                    <sequence>
                                <any namespace="##any" processContents="lax"
minOccurs="0"
                                        maxOccurs="unbounded" />
                    </sequence>
                    <attribute name="interface" type="token" use="required"
/>
                    <anyAttribute />
          </complexType>

          <complexType name="Tproperty">
                    <simpleContent>
                                <extension base="string">
                                        <attribute name="name"
type="string" use="required" />
                                        <attribute name="value"
type="string" use="optional" />
                                        <attribute name="type"
type="sd:Tjava-types"
                                                default="String"
use="optional" />
                                        <anyAttribute />
                                </extension>
                    </simpleContent>
          </complexType>

          <simpleType name="Tjava-types">
                    <restriction base="string">
                                <enumeration value="String" />
                                <enumeration value="Long" />
                                <enumeration value="Double" />
```

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

```
                        <enumeration value="Float" />
                        <enumeration value="Integer" />
                        <enumeration value="Byte" />
                        <enumeration value="Character" />
                        <enumeration value="Boolean" />
                        <enumeration value="Short" />
                    </restriction>
            </simpleType>

        <attribute name="must-understand" type="boolean" default="false">
            <annotation>
                <documentation xml:lang="en">
                    This attribute should be used by extensions to documents
                    to require that the document consumer understand the
                    extension.
                </documentation>
            </annotation>
        </attribute>
</schema>
```

Philipp Konradi 3/5/09 4:11 PM
Formatted: English (UK)

Best Practices

This section is non-normative.

Handling Distributed OSGi Services

The vast majority of OSGi bundles which consume services have been designed for local service invocations. With the introduction of Distributed OSGi it is important to consider whether or not a client bundle should be able to obtain a remote service. There are two options available to avoid accidentally resolving to a remote service:

1. Ensure the Distribution Software and/or Discovery Service are configured to avoid remote services being made available in the client's service registry.

2. Install a filter hook to filter out results for remote services. RFC 126 (Service Registry Hooks) and the requirement of a Distribution Software to add the service property osgi.remote=true to all remote proxies enables this to be done.

When designing a service or client for distributed operation, there are a number of factors to consider:

1. Communications failures: ensuring proper handling of failures and uncertain outcomes, such as a requeste reaching a service and being acted upon, but the response never reaching the client. Consider using a reliable transport for business-critical remote communications.

2. Latency issues which may lead to requests taking longer to complete, and in some cases could result in requests arriving out of sequence. Client and service implementation design can take this into account, and if necessary, a transport can be chosen which provides the desired QoS, such as assured message ordering.

3. Invocation semantics of a remote service are likely be passByValue, whereas OSGi local services are passByReference. If a service is likely to be distributed, consider designing and implementing it to be agnostic of the call semantics as this will maximize the opportunity for the service to be re-used.

Distribution-related limitations on service interface definitions

In OSGi, service interfaces are defined using Java interfaces. When exposing a service over a remote protocol, typically such an interface is mapped to a binding-specific interface definition which is then used to advertise the interface of the service. To make sure such a mapping to a distribution protocol would work, a few things should be taken into consideration, with regard to interface definition of remote services.

So it will probably be necessary to put some constraints on the possible usage of data types in service interfaces in order to be able to expose them over remote interfaces. As an example, an interface that has a `java.lang.Object` as an argument will probably not be allowed. The exact boundaries of the data fencing will need to be defined and it would be nice if a tool could or clear methodology could be defined that would allow the developer to test whether the interfaces at hand satisfy this requirement.

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

In general, the following rules should be adhered to. The Service interface should be defined in terms of:

Basic Types: *byte, short, int, char, long, float, double, string*
Arrays: *of basic types or a complex type which is part of the interface*
Complex types that are aggregations of the above.
[do we need to add more?]

The above is sometimes referred to as 'Data Fencing'.

Additionally, because most distribution transports use pass-by-value semantics, a developer should take care not to depend on any pass-by-reference semantics. In other words, if the caller passes an object to the Service and the Service modifies that object or makes an invocation on that object that causes a modification as a side-effect, the remote caller will not see this modification. Distributed Services should avoid such semantics.

The inverse is also an area where a developer should take care. For example, if a developer codes to a service interface assuming pass-by-value and therefore makes modifications to data which is passed in from a client or returned from a service, these modifications may become visible in the event the client and service are located in the same framework instance.

An implementation of Distributed OSGi could provide a tool that checks these constraints on your services and therefore informs the developer about the suitability for distribution.

Discovery Service Federation and Interworking

Tim Diekmann
Comment: TODO: move to best practices

This section describes potential scenarios for the use of the discovery service. The implementation of a discovery service may vary depending on the distributed software system involved. These scenarios are intended to illustrate desirable use cases for using the discovery service to fulfill enterprise requirements.

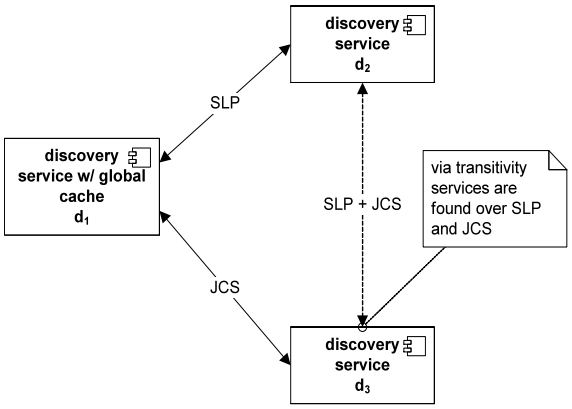


Figure 20 Multiple discoveries over different protocols

Tim Diekmann 4/10/09 5:29 PM
Deleted: *15*

The possible interaction of multiple OSGi service platforms over the discovery mechanism is shown in **Error! Reference source not found.** Since the discovery service implementation is not specified, it is possible that multiple different protocols may be deployed simultaneously. An implementation that maintains a cache of service information over all services discovered in a network, allows for building a

Tim Diekmann 4/10/09 5:29 PM
Deleted: Error: Reference source not found

transitive hull over the discovery mechanism. Thus, two OSGi service platforms may discover and reference each other even though there is no common protocol used in the discovery process.

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

The Discovery service implementation should not publish those services that it has discovered from other discovery instances over the network. This could lead to infinite loops. However, a Discovery service implementation should answer a request over network if it is aware of a suitable instance through its cache. This may include services discovered remotely.

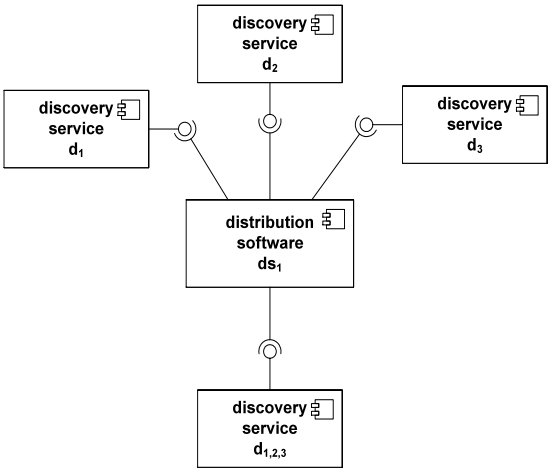


Figure 21 Possible discovery service implementation

Tim Diekmann 4/10/09 5:29 PM
Deleted: *16*

As shown in **Error! Reference source not found.** an implementation of the Discovery service interface may combine the implementation of multiple different protocols in a single bundle or provide separate bundles for different discovery protocols. This RFC does not make any assumption about the design choices.

Tim Diekmann 4/10/09 5:29 PM
Deleted: Error: Reference source not found

Bundle organization

For an OSGi client to be able to communicate with a remote Service, it will need access to a Java interface for the service. When the Service is implemented as an OSGi bundle, the easiest way to achieve this is to put the interface of the service in a separate bundle. This bundle should then both be installed in the OSGi client environment as well as on the Service's OSGi runtime. The service's implementation will have a dependency on this bundle.

Proxies

On the client side the DSW is expected to create and register local endpoints for the remote services. These endpoints are typically created as proxies. In these proxies additional logic with regard to caching and load balancing may be provided as appropriate. The definition of such smart proxies is left to a separate RFC.

Naming convention for communication-related properties

In order to make it easier for a Distribution Provider to decide whether modification of properties reported by Discovery is related to communication or not it's recommended to use the prefix "osgi.remote" for names of communication related service properties. Service properties defined by this RFC do already follow this recommendation. Depending on the fact whether the changed property describes some

communication aspects or not, Distribution Provider may decide to recreate the related service proxy or just to update its properties accordingly.

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

Reference Implementation

Installing Distribution Software in an OSGi platform

Both for Services and Consumers, the distribution software itself is rovided as an OSGi bundle, which is installed in the OSGi platform. Any configuration for the distribution software would be provided with this bundle, and will be automatically applied when the bundle is activated.

Considered Alternatives

Alternative: using simple properties to define service remoting

This alternative was not considered viable as the simple properties approach is most likely not expressive enough.

In this alternative the remoteness of the service is simply triggered by the `remote.profile` property on the Service Declaration. `remote.profile` is a simple property that specifies on a high level how the service is remoted, possible values could be:

- SOAP/HTTP
- SOAP/HTTPS
- CORBA
- RMI
- Other

The Distribution Software picks up this value and, if it supports this kind of profile, does the appropriate thing to expose the service remotely.

On the **Client-side**, the remote flag is set to true, as we are dealing with a proxy. The `remote.profile` contains the value of the profile as specified in the Service declaration.

Additionally, on the client side, the remote.url property holds the URL of the service. In most distribution technologies a URL can be used to point to the network location where the service can be contacted. Note that this URL is only provided to the consumer for informational purposes, the client does not need to deal with the URL as all the networking is taken care of by the proxy.

On the **Service-object** itself the remote property should either be set to false or not be set at all. (Note that it is not mandated, but allowed, to have the `remote.profile` property as set in the DS configuration file visible on the actual service object).

Pros:

- Very simple, easy to understand for the user.
- The RFC should list a number of known profiles, which could be implemented by vendors or open source products to provide interoperability.
- Vendors can add their own proprietary profiles.
- Interoperability on the wire for standardized profiles.

Cons:

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

- Not very flexible. Especially w.r.t. the specification of Qualities of Service. How will you specify that SOAP/HTTP with transactions is used? SOAP/HTTP/TX? How about reliability? It has the risk of becoming unmanageable when looking at all possible combinations. How will we distinguish between different versions of a binding, e.g. SOAP 1.1 and SOAP 1.2?
- Additional configuration is always needed, which will be vendor-specific.

A **variation** of this approach could be taken in which transport, binding and potentially QoS information are specified in separate properties.

Security Considerations

Vulnerabilities created by distributed OSGi include those in the bundles for the interfaces and proxies, and in the distributed software itself.

In the first case, distributed OSGi functionality must be implemented by trusted bundles, and deployment of distributed OSGi bundles must obtain the appropriate service permissions. Access to any resources required by the bundle or bundles also must be controlled via administrative permission. An implementation of distributed OSGi must prevent unauthorized deployment of bundles and unauthorized access to bundles and resources.

The two major security issues the DSW should address are authorized access to a service and the use of an encrypted communication protocol. When a remote service request is received, the DSW should check whether the request is authorized and also whether an encrypted protocol was used to transmit the request. If a request is not authorized for the service, the DSW should request authentication. If authentication isn't available, the DSW should return an error stating the requester is not authorized to access the service. Similarly, if the intent attribute of confidentiality is present on the service, the DSW should check whether an encrypted communication protocol was used and return an error to the requester if it was not.

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]. OSGi RFP 79: Remote Service Discovery. https://www2.osgi.org/members/svn/documents/trunk/rfps/rfp-0079-ServiceDiscovery.doc

[4]. OSGi RFP 88: Externalize OSGi Services.

https://www2.osgi.org/members/svn/documents/trunk/rfps/rfp-0088-ExternalServices.doc

[5]. OSGi RFC 126: https://www.osgi.org/members/svn/documents/trunk/rfcs/rfc0126/rfc-0126-ServiceRegistryHooks.odt

[6]. SCA Policy Framework 1.00, http://www.oasis-opencsa.org/SCA-policy-framework

[7]. SCA Web Service Binding Specification 1.00, http://www.oasis-opencsa.org/sca-bindings

[8]. SCA JMS Binding Specification 1.00, http://www.oasis-opencsa.org/sca-bindings

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

Unknown
Field Code Changed

Philipp Konradi 4/7/09 4:10 PM
Formatted: Italian

Philipp Konradi 4/7/09 4:10 PM
Formatted: Italian

Author's Addresses

Name	Eric Newcomer
Company	Progress Software Corp
Address	200 West Street, Waltham, MA 02451 USA
Voice	+1 781 902 8366
e-mail	ericn@progress.com, enewcomer@gmail.com

Name	David Bosschaert
Company	Progress Software Corp
Address	The IONA Building, Shelbourne Road, Ballsbridge, Dublin 4, Ireland
Voice	+353 1 637 2371
e-mail	dbosscha@progress.com

Name	Tim Diekmann
Company	TIBCO Software Inc.
Address	3303 Hillview Ave, Palo Alto, CA 94304, USA
Voice	+1 650 846 5521
e-mail	tdiekman@tibco.com

Name	Graham Charters
Company	IBM
Address	IBM United Kingdom Ltd, Hursley Park, Winchester, SO21 2JN, UK
Voice	+44 1962 816527
e-mail	charters@uk.ibm.com

Philipp Konradi 4/7/09 4:10 PM
Formatted: Italian

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

Name	Philipp Konradi
Company	Siemens AG
Address	Siemens AG, Otto-Hahn-Ring 6, 81739 Munich, Germany
Voice	+49 (89) 636-53802
e-mail	philipp.konradi@siemens.com

Name	Eoghan Glynn
Company	Progress Software Corp
Address	The IONA Building, Shelbourne Road, Ballsbridge, Dublin 4, Ireland
Voice	+353 1 637 2439
e-mail	eoglynn@progress.com

Tim Diekmann 2/20/09 1:53 PM
Deleted: Under Review

Acronyms and Abbreviations

OASIS	Organization for the Advancement of Structured Information Standards
Open CSA	Open Composite Services Architecture
SCA	Service Component Architecture
WSDL	Web Services Description Language