RFC 183 – Cloud Ecosystems

Draft

25 Pages

Abstract

The Computing Cloud often provides a highly dynamic environment where the load of a system might change, the topology of the cloud nodes might change or the requirements on the application may change at runtime. This document describes an OSGi cloud environment where nodes and capabilities can be discovered dynamically through OSGi Services and the deployment topology can be changed at runtime to react in changes in the observed characteristics, topology or requirements. An OSGi cloud can also be repurposed which can save network bandwidth as VM images only need to be sent to cloud nodes once.

0 Document Information

0.1 License

DISTRIBUTION AND FEEDBACK LICENSE, Version 2.0

The OSGi Alliance hereby grants you a limited copyright license to copy and display this document (the “Distribution”) in any medium without fee or royalty. This Distribution license is exclusively for the purpose of reviewing and providing feedback to the OSGi Alliance. You agree not to modify the Distribution in any way and further agree to not participate in any way in the making of derivative works thereof, other than as a necessary result of reviewing and providing feedback to the Distribution. You also agree to cause this notice, along with the accompanying consent, to be included on all copies (or portions thereof) of the Distribution. The OSGi Alliance also grants you a perpetual, non-exclusive, worldwide, fully paid-up, royalty free, limited license (without the right to sublicense) under any applicable copyrights, to create and/or distribute an implementation of the Distribution that: (i) fully implements the Distribution including all its required interfaces and functionality; (ii) does not modify, subset, superset or otherwise extend the OSGi Name Space, or include any public or protected packages, classes, Java interfaces, fields or methods within the OSGi Name Space other than those required and authorized by the Distribution. An implementation that does not satisfy limitations (i)-(ii) is not considered an implementation of the Distribution, does not receive the benefits of this license, and must not be described as an implementation of the Distribution. "OSGi Name Space" shall mean the public class or interface declarations whose names begin with "org.osgi" or any recognized successors or replacements thereof. The OSGi Alliance expressly reserves all rights not granted pursuant to these limited copyright licenses including termination of the license at will at any time.

EXCEPT FOR THE LIMITED COPYRIGHT LICENSES GRANTED ABOVE, THE OSGi ALLIANCE DOES NOT GRANT, EITHER EXPRESSLY OR IMPLIEDLY, A LICENSE TO ANY INTELLECTUAL PROPERTY IT, OR ANY THIRD PARTIES, OWN OR CONTROL. Title to the copyright in the Distribution will at all times remain with the OSGi Alliance. The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted therein are fictitious. No association with any real company, organization, product, domain name, email address, logo, person, place, or event is intended or should be inferred.

THE DISTRIBUTION IS PROVIDED "AS IS," AND THE OSGi ALLIANCE (INCLUDING ANY THIRD PARTIES THAT HAVE CONTRIBUTED TO THE DISTRIBUTION) MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, OR TITLE; THAT THE CONTENTS OF THE DISTRIBUTION ARE SUITABLE FOR ANY PURPOSE; NOR THAT THE IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS.
NEITHER THE OSGi ALLIANCE NOR ANY THIRD PARTY WILL BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF OR RELATING TO ANY USE OR DISTRIBUTION OF THE DISTRIBUTION.

Implementation of certain elements of this Distribution may be subject to third party intellectual property rights, including without limitation, patent rights (such a third party may or may not be a member of the OSGi Alliance). The OSGi Alliance is not responsible and shall not be held responsible in any manner for identifying or failing to identify any or all such third party intellectual property rights.

The Distribution is a draft. As a result, the final product may change substantially by the time of final publication, and you are cautioned against relying on the content of this Distribution. You are encouraged to update any implementation of the Distribution if and when such Distribution becomes a final specification.

The OSGi Alliance is willing to receive input, suggestions and other feedback (“Feedback”) on the Distribution. By providing such Feedback to the OSGi Alliance, you grant to the OSGi Alliance and all its Members a non-exclusive, non-transferable,

worldwide, perpetual, irrevocable, royalty-free copyright license to copy, publish, license, modify, sublicense or otherwise distribute and exploit your Feedback for any purpose. Likewise, if incorporation of your Feedback would cause an implementation of the Distribution, including as it may be modified, amended, or published at any point in the future ("Future Specification"), to necessarily infringe a patent or patent application that you own or control, you hereby commit to grant to all implementers of such Distribution or Future Specification an irrevocable, worldwide, sublicenseable, royalty free license under such patent or patent application to make, have made, use, sell, offer for sale, import and export products or services that implement such Distribution or Future Specification. You warrant that (a) to the best of your knowledge you have the right to provide this Feedback, and if you are providing Feedback on behalf of a company, you have the rights to provide Feedback on behalf of your company; (b) the Feedback is not confidential to you and does not violate the copyright or trade secret interests of another; and (c) to the best of your knowledge, use of the Feedback would not cause an implementation of the Distribution or a Future Specification to necessarily infringe any third-party patent or patent application known to you. You also acknowledge that the OSGi Alliance is not required to incorporate your Feedback into any version of the Distribution or a Future Specification.

I HEREBY ACKNOWLEDGE AND AGREE TO THE TERMS AND CONDITIONS DELINEATED ABOVE.

0.2 Trademarks

OSGi™ is a trademark, registered trademark, or service mark of the OSGi Alliance in the US and other countries. Java is a trademark, registered trademark, or service mark of Oracle Corporation in the US and other countries. All other trademarks, registered trademarks, or service marks used in this document are the property of their respective owners and are hereby recognized.

0.3 Feedback

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

0.4 Table of Contents

0.5 Terminology and Document Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in 1.

```
Source code is shown in this typeface.
```

0.6 Revision History

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

Revision	Date	Comments
Initial	August, 2011	David Bosschaert, Initial Version
0.2	April, 2012	Richard Nicholson, Additional Input
0.3	April, 2012	David Bosschaert, Prepare for F2F
0.4	October, 2012	David Bosschaert, incorporate feedback from Basel F2F, introduce Ecosystems.
0,5	November, 2012	Marc Schaaf, Minor changes and some comments for Orlando F2F.
0.6	November, 2012	David Bosschaert, incorporate feedback from Orlando F2F.
0.7	December, 2012	Richard Nicholson – feedback / comments
0.8	January, 2013	David Bosschaert, some additional comments and clarification

Revision	Date	Comments
0.9	February, 2013	David Bosschaert, incorporate feedback from Austin F2F, split off Distributed Event Admin and Distributed Config Admin sections.
0.10	February, 2013	Steffen Kächele, feedback and comments
0.11	February, 2013	David Bosschaert, Richard Nicholson, cleanup for EA draft.
0.12	March, 2013	David Bosschaert, describe how the metadata provided by the FrameworkStatus service can be enhanced and add FrameworkStatus service diagram.
0.13	April, 2013	David Bosschaert, process feedback from Cologne F2F. Remove EndpointEventListener as it moves to bug 164.
0.14	June, 2013	David Bosschaert, feedback from Palo Alto F2F, plus feedback from Carsten.

1 Introduction

1st generation 'public Cloud' solutions have since their inception been built upon two enabler technologies

- Service orientation (increasingly REST centric) allowing allowing dynamic find / bind / use of deployed Services & Resources.
- The virtual Machine - this used as the mechanism to:
  - Partition physical compute resource.
  - Via the virtual machine image - provide a standard deployment artifact; a static opaque software blob.

However, it is increasingly accepted that deployment of opaque virtual machine images consumes unnecessary network bandwidth and storage. As the dependencies are not understood – such approaches have larger downstream maintenance implications. For highly centralized / monolithic Cloud offerings – brute force infrastructure approaches - at significant capital cost – are possible (e.g. Amazon / Google offerings). However the dependency / maintenance issue is not addressed. Recent trends involving the deployment of software artefacts into a Cloud environment (Puppet, Chef) are a step in the right direction; but the software components remain coarse grained with non standard approaches to dependency management and configuration of the deployed artifacts. Industry standards bodies are retrospectively attempting to standardize topology and life-cycle for traditional applications via initiatives such as OASIS TOSCA & CAMP.

Meanwhile the next generation of Cloud will be driven by the edge – meaning pervasive / federated cloud solutions with service components running in a variety of environments including: 3rd generation mobile and home

networks and more federated Cloud cores. For such environments only the minimum necessary software required to realize the Cloud service should be deployed, as required, to the appropriate location / device. Updates likewise limited to the necessary changes.

To achieve this, software modularity, sophisticated 'requirements 'and 'capabilities' driven dependency resolution and assembly - are the key enablers. Hence OSGi is uniquely placed within the industry to realize this vision; this especially so given the resurgence of Universal OSGi activities.

One of the key aspects of Cloud Computing is the fact that Resources are deployed on (virtual) machines, nodes, which are not pre-determined. When working with more complex cloud systems, where various components are deployed across different nodes, these components need to be connected with each other to form a working solution. Furthermore frequent changes in these deployments are common for such environments to allow dynamic scalability which again requires the discovery of newly added or removed components even after the initial deployment is finished.

Building upon the background research presented in RFP 133 [3]., this RFC explores the intersection between OSGi and Cloud with specific emphasis on discovery, configuration and 'wire-up' of multi-node OSGi based applications in a dynamic Cloud environment through the use of OSGi APIs and mechanisms.

2 Application Domain

This RFC relates to Cloud Computing domains and use-cases but can also be useful in non-cloud environments. Cloud Computing is to a certain degree a marketing term and many of its concepts are applicable where distributed computing is used.

This RFC aims at providing a baseline platform that can address use cases around discovery of OSGi frameworks and other resources, provisioning and re-provisioning of deployables and reacting to change in the system by providing the primitives to discover and monitor the topology of a system. Combined with a remote deployment mechanism and utilizing the dynamic capabilities of the OSGi framework this provides the capabilities to control cloud deployments and change their characteristics at runtime.

The dynamic aspects of OSGi frameworks and its Service Registry map quite naturally to the dynamic behavior of Cloud systems where deployments may need to be modified during operation because of changes in demand or the running environment. The small footprint of OSGi frameworks themselves and the fact that they can be highly customized to the task at hand also fits well with Cloud scenarios where memory, computing power and storage facilities are often constrained or charged for per usage.

OSGi Services and Distributed Systems

Currently we have three categories of OSGi Service that are / or may potentially be / distributed in some form: namely RSA, Event Admin and ConfigAdmin. However there is no coherent / over-arching 'distributed architecture' which encompasses all, and explains the inter-relationship between, these OSGi services.

This is highlighted when one considers 'discovery'.

In addition to the type of entity being discovered (a RPC Service, message source/sink, REST resource etc) we MUST also consider 'change' as entities will usually have a mixture of static, slowly and rapidly changing properties.

It is important to make the distinction between occasionally changing properties and rapidly changing properties as different ways to advertise these may be appropriate.

Discovery

The following 'discovery' use cases are suggested

1) Resources Discovery – Available OSGi Frameworks in the specified environment; also physical resource in the environment which might act as a host for an OSGi Framework.

Static properties might include attributes and capabilities such as location; ownership; access to other type of resource – i.e. required data, CPU/number of cores, OS, JVM or OSGi framework Type (all could be considered immutable),

Dynamic properties attributes include installed bundles / sub systems at each point in time, current available memory, current load etc.

2) Deployed Artifacts - Artifacts of a particular type that have been deployed into the environment. Usually a subset of discovered Resources.

Static properties might include name of bundle / sub-System, static configurations, requirements and capabilities.

Dynamic properties might include resource consumption / performance metrics (monitoring) , configurations which are dynamically configurable at runtime.

3) Available Services – Services that are available within the environment. Usually a subset of discovered / deployed artifacts.

Static: Properties of the host – including all – or a subset of - resource and artefact 'Capabilities'

Dynamic: Usually relating to Service performance – resource load, memory – number of items to be processed, reliability / quality metrics.

Note that (2) and (3) may be expressed as attributes of (1). However one might also wish to independently discover entities of a particular type.

Remote Service Administration / Remote Services

Distributed Service discovery is covered within the OSGi Alliance's Remote Service Administration specification.

Service endpoints with associated properties (service.properties) may be advertised via a pluggable discovery mechanism.

However, the specification is a little vague as to the nature of advertised service properties. Being in effect the local service.properties registered in the local service registry – are Service endpoint properties assumed to be almost immutable; i.e. infrequently changing. Current RSA implementations work this way – in that – if Service properties change – the 'old' service is removed and the 'new' service discovered.

If static / immutable properties change - then by definition - the new entity is no-longer the same as the one initially discovered. However, properties that relate to Services might be highly volatile; perhaps local

performance statistics that we use to priority select / or balance across / service instances. If tracking a remote service and the properties change, we don't want to have to handle the service apparently disappearing and reappearing, especially if this happens frequently e.g. for properties added by the middleware such as "current load".

Should these be treated as property 'updates' to those initial advertised? So under the 'discovery' umbrella. Or treated as a separate monitoring concerns? Perhaps –

- advertised properties used for discovery are immutable – but included in this static advertisement is the information required to subsequently subscribe to / receive volatile / dynamic property updates.
- Or, perhaps mutable / immutable properties remain collocated - defined in a more sophisticated service advertisement.s.

Note that while an underlying implementation is not defined - service 'discovery' is already event centric. Given this one might like to treat the processing of update / monitoring events differently to discovery events - but transported by the same distributed eventing mechanisms.

Contracts Based Interactions

It has already been suggested that, w.r.t. Interactions, we need to focus on the *contract*. Each participant has its own role with respect to the contract: i.e. what it is expected to provide and what it can expect from the other participants.

A contract is just the agreed set of interactions between modules. When we think about services we tend to think of a contract as a Java Interface... e.g. the CreditCheck service provides method calculateRating(). But that's a simplistic contract with one provider (i.e. the credit rating provider) and some consumers, and it doesn't *appear* to support asynchronous interaction because invocation always originates from a consumer.

But if you consider a group of interfaces then you have something more powerful, because each participant can provide some interfaces and consume some interfaces. For example a stock exchange: the exchange itself provides the OrderEntry, and other participants provide ExecutionListeners or MarketDataProviders or whatever. Hence, the 'contract' this not with a single Java interface but with a coherent collection of interfaces: in other words a package: the 'contract' concept spanning synchronous, asynchronous and event based interactions.

2.1 Terminology + Abbreviations

This document uses terms defined in OSGi RFP 133 Cloud Computing. The terms are based on the NIST definitions for Cloud Computing and common industry naming practice.

Additional terminology:

Cloud Ecosystem – a dynamic Cloud System.

3 Problem Description

Cloud-based applications are often composed of multiple components each running on one or more cloud nodes.

For instance the following is an example application architecture.

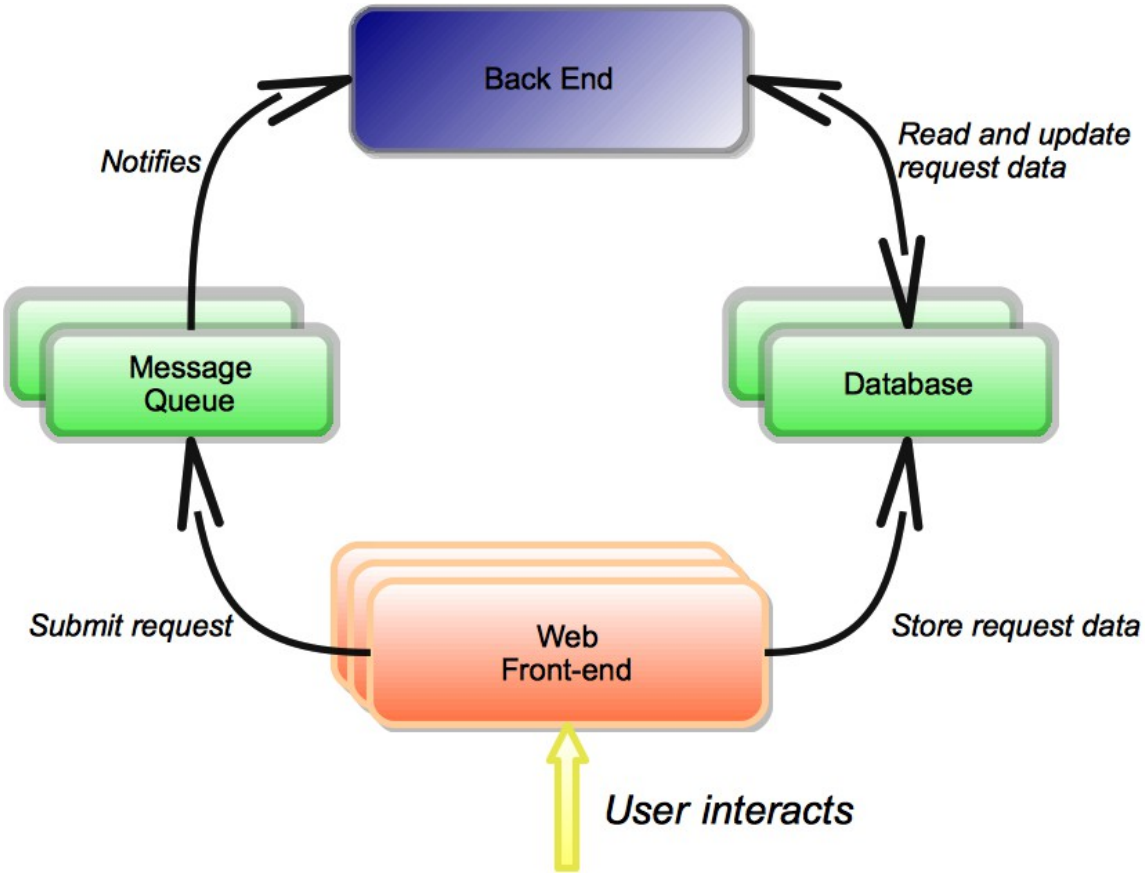


Illustration 1: A possible Application Architecture

This example e-commerce application has a web front-end, a database, a message queue and a back-end component. Each of these components is replicated on various nodes. In order to function the components of the application need to know where (e.g. on what IP) other components can be found. Components also need to be kept informed of the liveness of their component dependencies.

In traditional deployments this kind of information is often kept in static files, handled via a hardware load-balancer or through a proprietary HA solution.

In a cloud scenario a standards-based solution is needed to enable the discovery of application components in a this dynamic environment.

For more context please refer to the Problem Description section in RFP 133 [3].

4 Requirements

This RFC covers the following requirements listed in RFP 133 [3].

MAN0004 – The solution MUST provide APIs to discover available OSGi Frameworks in a Resource Domain.

MD0001 – The solution MUST define APIs that allow querying of capabilities and other metadata of OSGi Frameworks in the Cloud. This information SHOULD at least include the following:

- Framework GUID

MD0002 – An OSGi bundle MUST be able to add proprietary capabilities to the metadata exposed by its OSGi Framework in the Cloud.

MD0003 – The solution MUST provide information about the environment, system, and the capabilities and properties of the platform underlying an OSGi Framework in the Cloud. This information SHOULD include at least the following static capabilities:

- location
- IP address
- cpu architecture
- cpu capacity
- Total memory

And the following dynamic capabilities

- cpu load factor
- Available Memory

MD0004 – The solution MUST allow provider-specific capabilities to be added regarding the underlying platform.

MD0007 – The solution MUST allow querying the available OSGi Frameworks in a Cloud Domain based on the metadata and capabilities these OSGi Frameworks expose.

Additional requirements obtained during the EclipseCon 2012 Cloud Workshop:

CWS0010 – Make it possible to describe various Service unavailable States. A service may be unavailable in the cloud because of a variety of reasons.

- Maybe the amount of invocations available to you have exhausted for today.
- Maybe your credit card expired

- Maybe the node running the service crashed.

It should be possible to model these various failure states and it should also be possible to register 'callback' mechanisms that can deal with these states in whatever way is appropriate (blacklist the service, wait a while, send an email to the credit card holder, etc).

CWS0020 – Come up with a common and agreed architecture for Discovery. This should include consideration of Remote Services, Remote Events and Distributed Configuration Admin.

CWS0030 – Resource utilization. It should be possible to measure/report this for each cloud node. Number of threads available, amount of memory, power consumption etc…

CWS0040 – We need subsystems across frameworks. Possibly refer to them as 'Ecosystems'. These describe a number of subsystems deployed across a number of frameworks.

CWS0050 – It should be possible to look at the cloud system state:

- where am I (type of cloud, geographical location)?
- what nodes are there and what is their state?
- what frameworks are available in this cloud system?
- where's my service in the cloud~~OBR~~?
- what state am I in?
- what do I need here in order to operate?
- etc…

CWS0060 – Deployment - when deploying replicated nodes it should be possible to specify that the replica should *not* be deployed on certain nodes, to avoid that all the replicas are deployed on the same node.

5 Technical Solution

A complete solution can be broken into the following considerations:

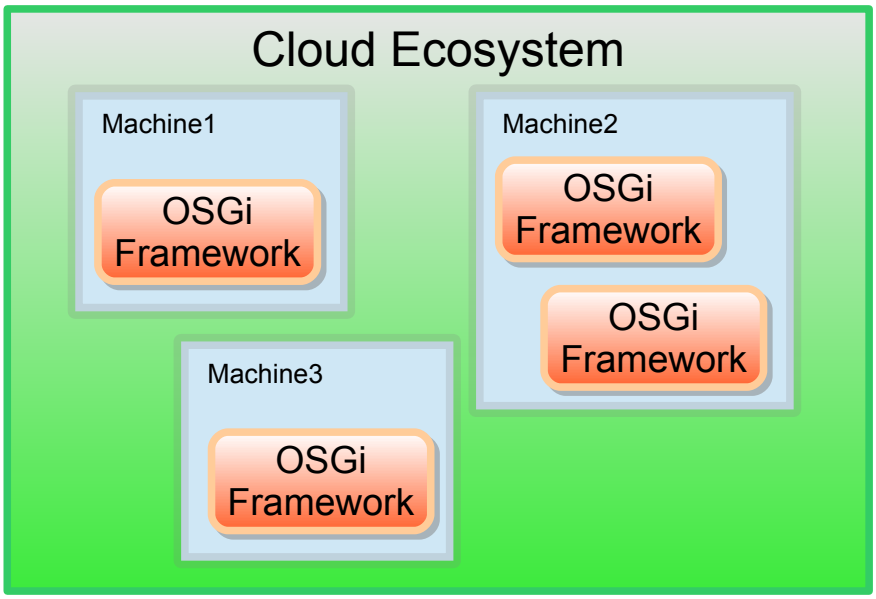
1. Definition of functional and deployment topologies – for System / EcoSystem
2. Discovery of Resource
3. Method of deployment – mapping required topology to available resource.

4. Subsequent re-discovery of deployed artifacts / available resource

5. Interaction models between deployed components.

This specification describes a platform where multiple OSGi Framework instances are running in different Java VMs and often on different actual or virtual computing nodes.

The platform provider provides the Ecosystem administrator with tools to create new computing nodes and to associate these nodes with an ecosystem. Each node is associated with at most one ecosystem, while a single ecosystem can be associated with many nodes. Note however that a cloud node may be virtual, so multiple nodes could potentially be hosted within the same platform or infrastructure through multi-tenancy.



This specification describes a means of discovering the topology of OSGi nodes in the ecosystem and provides primitives to provision these, which includes their initial provisioning as well as applying ongoing changes to the provisioning of the system during runtime, to react to changes in the ecosystem topology as well as changes in the runtime characteristics of the application, i.e. to scale up or scale down dynamically.

Therefore the scope of interest is 1, 2, 4 & 5 from the above list.

While concern 3 in the above list can be realized through the primitives in this specification, this document does not describe a format to declare a mapping from a topology to resources. It is expected that this will be done in a separate RFC.

5.1 Platform requirements

Platforms compliant with this specification provide the following components on each computing node.

1. An OSGi Core Framework as defined by the Core R6 specification.

2. A Remote Services Distribution Provider that understands the `osgi.configtype.ecosystem` Configuration Type.

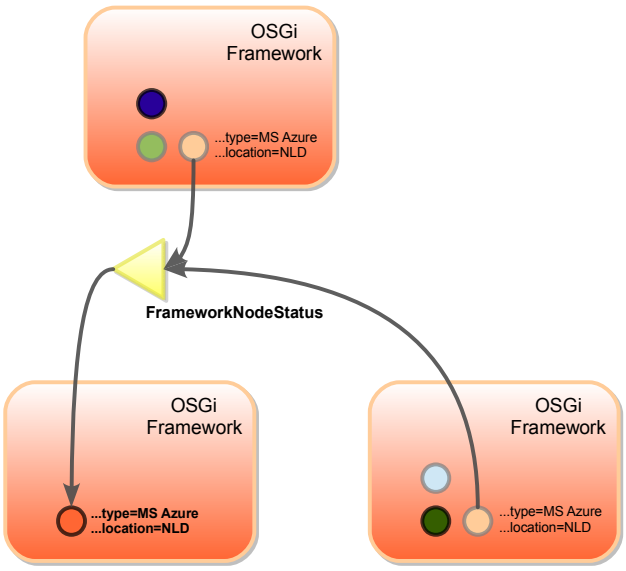
3. A discovery mechanism providing visibility to all remoted OSGi services with the `osgi.configtype.-ecosystem` to all other OSGi frameworks in the same ecosystem.

4. A component which registers an FrameworkNodeStatus service for each Framework running in the ecosystem. These FrameworkNodeStatus objects are registered as remote services with the `osgi.configtype.ecosystem` configuration type.

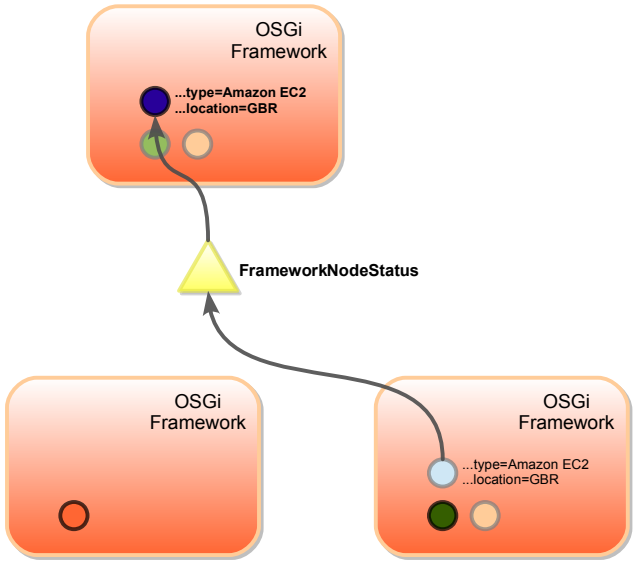
5. A remote management mechanism compliant with RFC 182. This mechanism should be accessible from within the ecosystem, but does not need to be accessible from the outside world.

5.2 Framework Node Status Service

The FrameworkNodeStatus service advertises the availability of an OSGi framework in the Ecosystem. The Framework ~~is~~can be exported with the Remote Services configuration type `osgi.configtype.ecosystem` so that other frameworks running in the same ecosystem have visibility of this Framework, however alternative mechanisms to distributed this service across the Ecosystem are also permitted. Furthermore, other frameworks can listen for services of this type to appear, disappear or change if they wish to be notified of changes occurring in the ecosystem. This is vital information for a Management agent or Provisioning component and can also be used to dynamically re-scale and re-purpose the deployments in the ecosystem.



FrameworkNodeStatus service accessible all interested nodes



Only nodes using the Service will get metadata mirrored

The FrameworkNodeStatus service provides metadata about the framework via Service Registration properties as well as via *variables* which can be obtained via the getVariable() API. Static or mostly static metadata is represented as Service Registration properties, dynamic metadata is represented as variables.

The FrameworkNodeStatus is available with the following Service Registration Properties.

key	***data type***	***description***
org.osgi.framework.uuid	String	The globally unique ID for this framework.
org.osgi.node.host~~ip~~	String+	The external host names or ip addresses~~IPs~~ for this OSGi Framework, if exists.
org.osgi.node.host~~ip~~.internal	String+	The internal host names or ip addresses~~IPs~~ for this OSGi Framework for access from inside the Ecosystem, if exists.
org.osgi.node.type	String	The name of the Cloud/Environment in which the Ecosystem operates.
org.osgi.node.version	String	The version of the Cloud/Environment in which the Ecosystem operates. The value follows the versioning scheme of the cloud provider and may therefore not comply with the OSGi versioning syntax.
org.osgi.node.country	String (3, optional)	ISO 3166-1 alpha-3 location where this Framework instance is running, if known.
org.osgi.node.location	String (optional)	ISO 3166-2 location where this framework instance is running, if known. This location is more detailed than the country code as it may contain province or territory.
org.osgi.node.region	String	Something smaller than a country and bigger than a location (e.g. us-east) ??? Jan Rellermeyer to provide more details...
org.osgi.node.rest.url	String+ (URL, optional)	The external URL of the framework management REST API, if available.
org.osgi.node.rest.url.internal	String+ (URL, optional)	The ecosystem-internal URL of the framework management API, if available.
org.osgi.framework.version	String	The value of the Framework properties as obtained via

key	*data type*	*description*
org.osgi.framework.processor org.osgi.framework.os.name org.osgi.framework.os. version		BundleContext.getProperty().
java.version, java.runtime.version, java.vm.vendor, java.vm.version, java.vm.name	String	The values of the corresponding Java system properties.
… additional properties …		Additional properties may appear, set by the framework, Remote Services implementation or other entity.
… custom properties …		See section .

The service registration properties and API only support the function of obtaining information about a (remote) OSGi Framework. They do not allow any modifications to the framework and/or its metadata to be made. The API of the FrameworkNodeStatus service is as follows.

```
public interface FrameworkNodeStatus {
	// Obtains dynamic framework metadata
	String[] listVariableNames();
	String getVariable(String name);
	Map<String, String> getVariables(String ... filter);
}
```

5.2.1 Obtaining dynamic framework metadata

While service registration properties are used to advertise and obtain static or mostly static metadata, dynamic metadata is available via the FrameworkNodeStatus.getVariable() API. Invoking this API will generally cause a remote service invocation to be performed in order to obtain the current value of the variable requested.

The following variable names are predefined, however none of these are required to be supported:

variable name	*data type*	*description*
available.memory	Long	The amount of memory available to the JVM in which the Framework runs in kilobytes.
available.diskspace	Long	The amount of disk space available to the ~~JVM~~bundles where the Framework runs in kilobytes. This could be temporary disk space that does not persist across node restarts.
processor.load	Integer [0..100]	The load of the machine. 0 means no load at all where 100 means operating at full capacity.
... custom variables ...		See section .

The getVariable() API returns all values as a String. The data type above provides guidance on the interpretation of the value.

Application-specific FrameworkNodeStatus metadata

Bundles can register a local `FrameworkNodeAddition` Service (via the Whiteboard pattern) that allows them to provide additional metadata associated with the `FrameworkNodeStatus` service. This allows the build-up of a catalog of capabilities, which is visible to remote frameworks but can also be of use in the local framework.

The `FrameworkNodeAddition` service is registered with the following properties:

property	*data type*	*description*
add.properties	String+	The names of additional framework properties. The specified properties will be copied from this service registration to the FrameworkNodeStatus service registration that is available to remote frameworks.
add.variables	String+	The names of additional framework variables. The variables can be accessed through the getVariable() API and are included in the listVariableNames() API.

When multiple FramworkNodeAddition services provide the same property or variable, the service with the highest service.ranking wins. In case of a tie, the service with the lowest service.id wins.

The service has the following API:

```
public interface FrameworkNodeAddition {
  String getVariable(ClientContext client, String name);
}
```

When clients invoke the FrameworkNodeStatus.getVariable() API for the added variables, the FrameworkNodeAddition service will receive a callback to handle the invocation. Note that the implementation is permitted to return different values for a given variable for different clients. For example clients with a different SLA can be provided with different values for a certain variable. The implementation can use the ClientContext passed in to distinguish between clients.

For example the following adds a service property `my-app.role=database` and a framework variable `network.load` to the FrameworkNodeStatus service:

```
Dictionary<String, Object> d = new Hashtable<String, Object>();
d.put("add.properties", new String [] {"my-app.role"});
d.put("my-app.role", "database");
d.put("add.variables", "network.load");
ctx.registerService(FrameworkNodeAddition.class.getName(),
  new MyAdditionImpl(), d);
```

The implementation of the FrameworkNodeAddition interface provides a callback mechanism to provide the network.load variable:

```
public class MyAdditionImpl implements FrameworkNodeAddition {
  @Override
  public String getFrameworkVariable(String name, ClientContext client) {
    if ("network.load".equals(name)) {
      return networkInfo.getLoad()
    }
    throw new IllegalArgumentException(name);
  }
}
```

5.2.2 Restrictions

Added property or variable names may not start with the following prefixes: “org.osgi.”, “osgi.”, “java.”, “objectClass” or “service.”.

5.2.3 Variable change notification

Consumers can subscribe to change notifications for Framework variables via the OSGi Event Admin Service. To obtain these notifications across VM boundaries a Distributed OSGi Event Admin can be used, see RFC XXX.

5.3 Service Distribution

OSGi Remote Services functionality is available throughout the ecosystem via the `osgi.configtype.ecosystem` configuration type. This configuration type can make OSGi services remotely available that restrict their API as follows:

- The services are registered under one *interface* only.
- The methods parameters and return types of the interface are restricted to basic datatypes as used for service properties: primitive types, their wrappers as well as arrays and basic collections (List, Set and Map).
- Additionally, method parameters and return types can be also be of interfaces composed using the previous restrictions.

Given any service that is implemented using these restrictions, registering the service with the interface specified in the service.exported.interfaces and the osgi.configtype.ecosystem as the service.exported.configs service registration property will make the service available to service consumers in all frameworks within the ecosystem.

5.4 Remote Services Admin extensions

The Remote Services Admin specification needs to be expanded.

- The EndpointListener interface in this specification was missing a modification callback. This needs to be added.
- The remote service provider side needs to gain more control over how remote clients invoke the service. For example it may need to do some bookkeeping and count every time a client invokes the service. Or it may decide to give every client a unique instance of the service, if this service holds state for that client.
- Remote services need to be able to provide metadata about themselves. This may include being able to communicate to clients whether the service is operating normally, whether there is maintenance being planned, whether payment or other quota-related issues cause disruption or otherwise.

5.4.1 Providing control over remote clients

To provide control over how clients can invoke the service, the service can be registered via a RemoteServiceInvocationHandler. The RemoteServiceInvocationHandler allows the implementor to:

- Identify the client. This is done via the ClientContext object.
- Count the client invocations.
- Block client invocations.
- Decide which backing object to use to handle the invocation. This allows pooling of backing objects in cases where each object can only be used by one client at a time. I also allows handing out of separate objects for each client or even separate objects for each invocation.

The interface of the RemoteServiceInvocationHandler is as follows:

```
/**
 * Provides control over services used by remote clients. When an instance of this
```

```
 * class is attached to the "service.exported.handler" property of the service
 * registration the handler will be called for each remote invocation instead of
 * the service object directly.
 */
public interface RemoteServiceInvocationHandler<T> {
    /**
     * Invoke the service on behalf of a remote client.
     * @param client Information in relation to the client.
     * @param reference The Service Reference representing the service being
     *   invoked.
     * @param method The method of the service being invoked.
     * @param args The method arguments.
     * @return Provide the return value to be returned to the remote client.
     * @throws Exception if the embedded invocation throws an exception.
     */
    public Object invoke(ClientContext client, ServiceReference<T> reference,
                         Method method, Object[] args) throws Exception;
}
```

The ClientContext interface provides information about the remote client which is making the invocation. It is defined as follows, some of the methods may return `null` as not all the information may be available all the time:

```
  public interface ClientContext {
    String getHostIPAddress();
    String getFrameworkUUID();

    Principal getPrincipal();
    Map<String, Object> getProperties();
  }
```

The following is an example implementation of a RemoteServiceInvocationHandler that disallows concurrent invocations by the same remote client:

```
  public class MyServiceInvocationHandler implements
                  RemoteServiceInvocationHandler<MyService> {
    ConcurrentMap<String, Object> activeClients =
      new ConcurrentHashMap<String, Object>();

    @Override
    public Object invoke(ClientContext client, ServiceReference reference,
                         Method method, Object[] args) {
      if (activeClients.putIfAbsent(client, Boolean.TRUE) != null)
        throw new IllegalStateException(
          "Only 1 concurrent invocation allowed per client.");
```

```
      try {
        return method.invoke(new MyServiceImpl(), args);
      } finally {
        activeClients.remove(client);
      }
    }
  }
```

The remote service MyService is then registered as follows:

```
    Hashtable<String, Object> props = new Hashtable<String, Object>();
    props.put("service.exported.interfaces", "*");
    props.put("service.exported.configs",
              new String [] {"osgi.configtype.ecosystem", "<<nodefault>>"});
    props.put("service.exported.handler", new MyServiceInvocationHandler());
  bundleContext.registerService(MyService.class, new MyService(), props);
```

5.4.2 Remote Service Metadata

Remote services can fail for reasons generally not applicable to local services. Sometimes the reasons for failure are non-technical. Maybe the credit card used for payment has expired, or maybe the service is down for maintenance.

5.4.2.1 Obtaining Remote Service Metadata

A Remote Service can provide metadata around its availability or other concern through the RemoteServiceMetadataProvider API. This metadata can be instrumental in deciding which service instance to invoke if more than one candidate is available. It can also be useful in preventing operational failure or in diagnosing failure scenarios. The RemoteServiceMetadataProvider has the following API:

```
public interface RemoteServiceMetadataProvider {
    String[] listVariablesNames();
    String getVariable(String name);
    Map<String, String> getVariables(String ... filter);
}
```

The metadata can be obtained by a consumer through the `"service.imported.metadata"` property of an imported remote service:

```
Object mdp = sref.getProperty("service.imported.metadata");
if (mdp instanceof RemoteServiceMetadataProvider) {
  RemoteServiceMetadataProvider provider =
    (RemoteServiceMetadataProvider) mdp;
  System.out.println(provider.getVariable("remaining.invocations"));
}
```

5.4.2.2 Providing Remote Service Metadata

Remote Service Metadata is always specific to the service and must therefore be plugged in by the service provider. When a remote service is registered a handler object can be provided. This handler object can provide the service metadata as well as handle the actual remote invocations (see section 5.4.1). Remote Service Metadata is provided through an instance of the RemoteServiceMetadataHandler interface:

```
public interface RemoteServiceMetadataHandler {
    String[] listVariableNames(ClientContext client);
    String getVariable(ClientContext client, String name);
}
```

An implementation of this interface can be registered with the "service.exported.handler" property alongside the other Remote Service related properties, for example:

```
  TestService ts = new TestServiceImpl();
  Dictionary<String, Object> props = new Hashtable<String, Object>();
  props.put("service.exported.interfaces", "*");
  props.put("service.exported.configs",
    new String [] {"osgi.configtype.ecosystem", "<<nodefault>>"});
  props.put("service.exported.handler", new TestServiceMetadataHandler());
  context.registerService(TestService.class.getName(), ts, props);
```

The Remote Service implementation ensures that the handler provided in such as way handles calls from remote clients on the object obtained from the "service.imported.metadata" property on the client-side service reference.

Note that to provide both an invocation handler as well as a metadata handler a single object implementing both the RemoteServiceInvocationHandler as well as the RemoteServiceMetadataHandler interface can be used.

5.4.2.3 Service variables

The following variable names are predefined but none of these are mandatory:

variable name	*data type*	*description*
service.status	String, from: "OK" "PAYMENT_NEEDED" "UNAUTHORIZED" "FORBIDDEN" "NOT_FOUND" "QUOTA_EXCEEDED" "SERVER_ERROR" "TEMPORARY_UNAVAILABLE" "WARNING"	Returns the status of the service. Return values that start with OK mean that the service can be invoked. Other return values indicate that service invocation will result in an error.
service.detail	"QUOTA_REMAINING:<int>{: <reset-date>}" "PAYMENT_NEEDED_AT:<date>" "TEMPORARY_UNAVAILABLE:<date>: <duration-mins>"	If the service.status has the value other than OK this variable can be used to obtain machine-readable details.
service.message{.<locale>}	String. The locale can be provided by appending it to the variable name. For example: service.message service.message.nl_BE service.message.de etc...	If the service.status has a value other than OK this variable can be used to provide a human-readable message with more information about the service.status/-service.-warning. Same locales can be supported as are described in Core spec 3.11. The supported localizations are advertised through the listVariableNames() API.
… custom …		Implementations may provide custom variables as appropriate.

5.4.3 EndpointListener enhancement

This section has moved to bug 164: https://www.osgi.org/bugzilla/show_bug.cgi?id=164

6 Data Transfer Objects

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

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

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

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

7 Considered Alternatives

7.1 Framework Discovery

It needs to be possible to discover the available OSGi frameworks in the cloud system. Along with the existence of the frameworks themselves it must be possible to discover metadata about the frameworks, such as machine characteristics, cloud provider and other cloud metadata and information such as the physical location of the machine (country) and the IP address of the machine.

This information can be used to make provisioning-based decisions but also to configure services available in the cloud system that are not directly represented as distributed OSGi service, for example a Messaging System which needs to be configured with an IP address of the broker.

Potential ways to realize this:

- Via an RSA-distributed service that represents an OSGi Framework.
  - This requires that RSA will be expanded to support update of Service properties, as framework characteristics change at runtime.
  - pros: strong support for services in the framework. Ability to reuse LDAP service queries to find matching frameworks.
  - cons: ?
- Via an eventing mechanism
  - Paremus has some experience with this – TODO need to elaborate the potential design.
- Through a Repository API that represents Frameworks in the System. In this case the contents of the repository is all the nodes in the Cloud System.
  - A Repository (-like) API can be used to select frameworks in the Cloud System.
  - pros: use Generic Capabilities and requirements to select matching frameworks.
  - cons: Generic Capabilities and requirements are less suitable for dynamically changing properties.

7.2 Service Discovery

Remoted OSGi Services must be discoverable in the Cloud System. RSA-based distribution is the most suitable realization for this, however it must be expanded to cover service property changes.

7.3 Configuration Changes

OSGi Configuration Admin is a highly versatile dynamic configuration system, which should be suitable for usage in a Cloud System. The Configuration Consumption API is already suitable for remote distribution but we should look into whether the Administration API needs to be enhanced. Is it necessary to be able to target a specific cloud node via Configuration Admin? And if so how can that be realized?

7.4 Application-specific Framework-related metadata

Introduce an OSGiFrameworkPublisher service that allows applications to add/remove/modify service registration properties on the (remoted) OSGiFramework service.

Note the properties may not start with org.osgi., java. or service.

TODO describe this service API

TODO the API should also allow for the registration of new framework and service variables

8 Security Considerations

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

9 Document Support

9.1 References

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

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

[3]. RFP 133 OSGi Cloud Computing, available via https://www.osgi.org/bugzilla/show_bug.cgi?id=114

9.2 Author's Address

Name	David Bosschaert
Company	Red Hat
Address	6700 Cork Airport Business Park Kinsale Road Cork Ireland
Voice	+353 21 230 3400
e-mail	david@redhat.com

Name	Richard Nicholson
Company	Paremus
Address	
Voice	
e-mail	

Name	Marc Schaaf
Company	
Address	
Voice	
e-mail	marc.schaaf@fhnw.ch

Name	Steffen Kächele
Company	University of Ulm
Address	
Voice	
e-mail	steffen.kaechele@uni-ulm.de

9.3 Acronyms and Abbreviations

9.4 End of Document