RFC 183 – Cloud Ecosystems

Draft

23 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

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
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

Revision	Date	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.
0.15	August, 2013	David Bosschaert, revamp the Remote Service Metadata section.
0.16	August, 2014	David Bosschaert, update to FrameworkNodeStatus, prepare for Madrid F2F
0.17	October, 2014	David Bosschaert, process feedback from the Madrid F2F.
0.18	September, 2015	David Bosschaert, feedback from the Cologne F2F.

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.

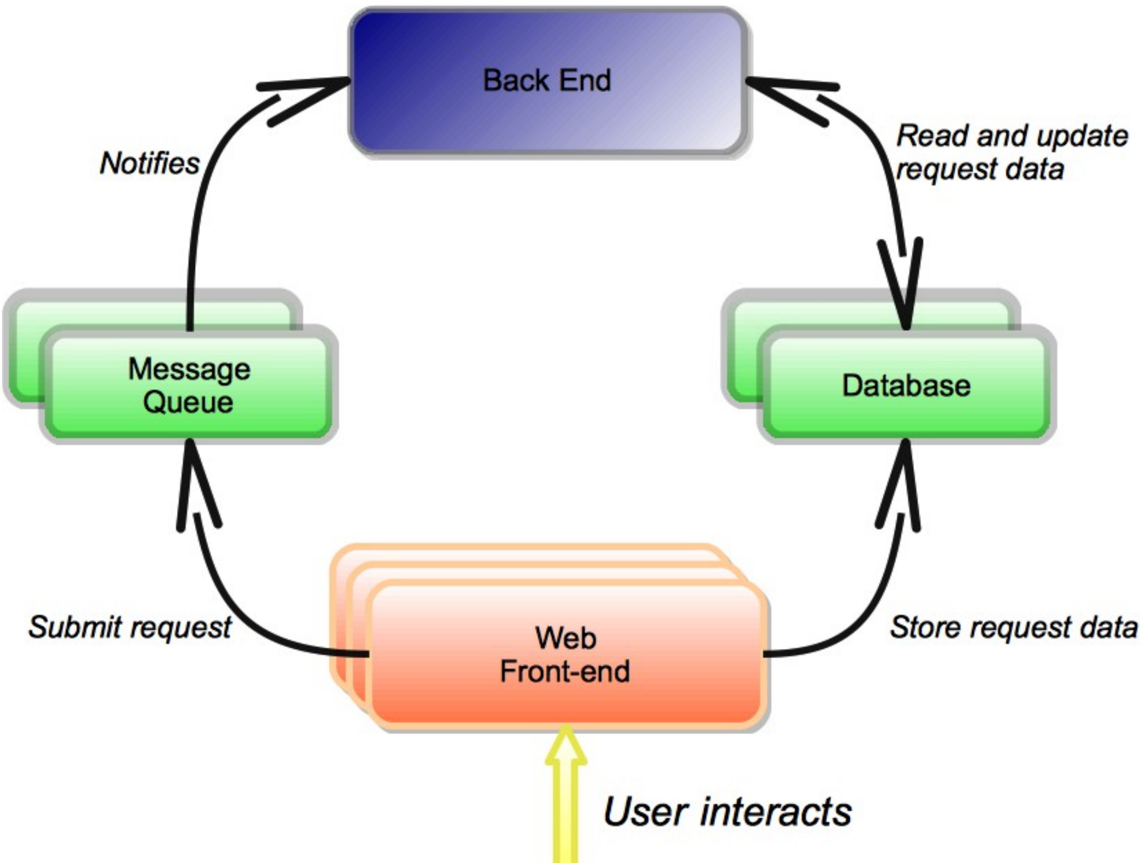


Illustration 1: A possible Application Architecture

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?
- 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:

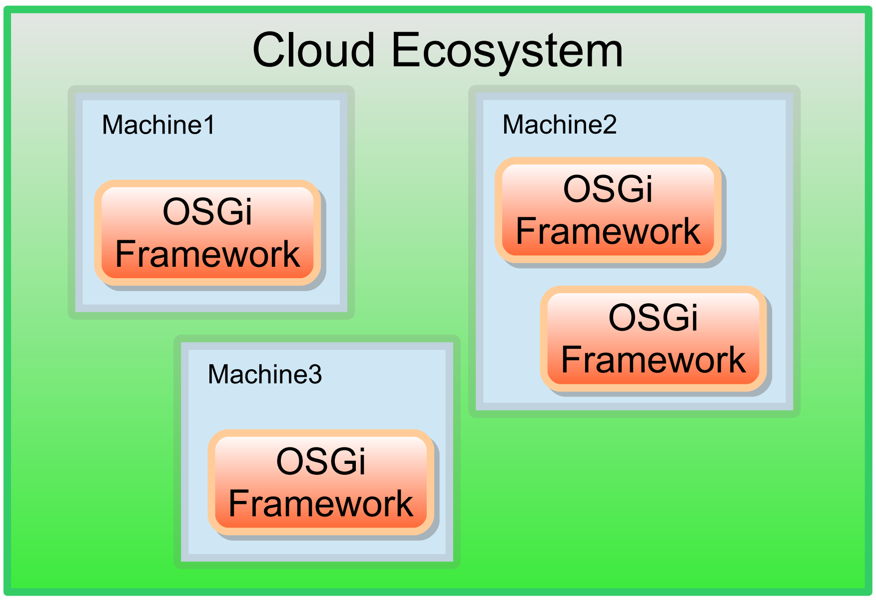
1. Definition of functional and deployment topologies – for System / EcoSystem
2. Discovery of Resource
3. Method of deployment – mapping required topology to available resource.

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 does not describe the how the compute nodes are created and associated with the ecosystem. Platform providers can provide proprietary solutions for this which may be realized through a web-based admin console, a set of command-line utilities, a REST-based API or otherwise.

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.

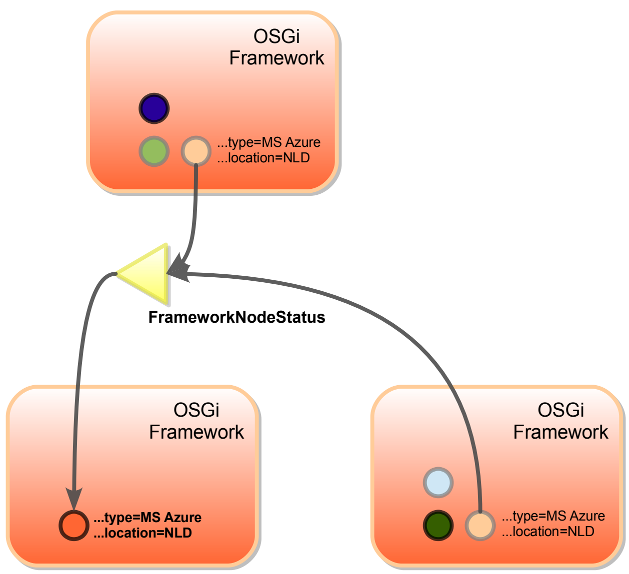
3. A discovery mechanism providing visibility to all remoted OSGi services with the `osgi.configtype.ecosystem` to all other OSGi frameworks in the same ecosystem.

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.

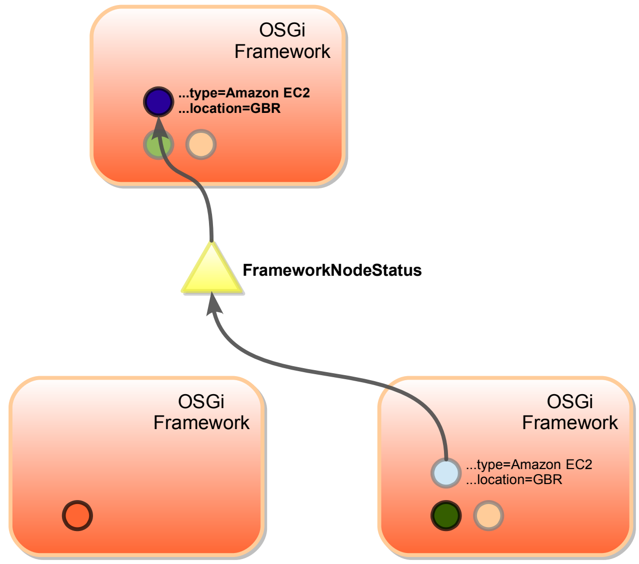
5. A remote management mechanism compliant with RFC 182. This mechanism should be accessible from within the ecosystem, but does not need to be accessible from the outside world.

5.2 Framework Node Status Service

The FrameworkNodeStatus service advertises the availability of an OSGi framework in the Ecosystem. The Framework 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 distribute 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.

TODO do we need to limit these if they are available via capabilities/DTOs?

key	***data type***	***description***
org.osgi.framework.uuid	String	The globally unique ID for this framework.
org.osgi.node.ecosystem	String	The name of the Ecosystem this node belongs to. Multiple ecosystems can co-exist within a single cloud system.
org.osgi.node.host	String+	The external host names or ip addresses for this OSGi Framework, if exists.
org.osgi.node.host.internal	String+	The internal host names or ip addresses for this OSGi Framework for access from inside the Ecosystem, if exists.
org.osgi.node.vendor~~type~~	String	The vendor 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-1) ~~??? Jan Rellermeyer to provide more details...~~
org.osgi.node.rest.url	String+	The external URL of the framework management REST API, if

key	data type	description
	(URL, optional)	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 org.osgi.framework.processor org.osgi.framework.os.name org.osgi.framework.os. version	String	The value of the Framework properties as obtained via 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 5.2.3.

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.

5.2.1.1 Capabilities

The following capabilities are available via the FrameworkNodeStatus service.

osgi.node.identity

attribute	*data type*	*description*
org.osgi.framework.uuid	String	The globally unique ID for this framework.
org.osgi.node.ecosystem	String	The name of the Ecosystem this node belongs to. Multiple ecosystems can co-exist within a single cloud system.
org.osgi.node.vendor	String	The vendor 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.

osgi.node.location

attribute	*data type*	*description*
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-1)
org.osgi.node.region.zone	String	Zone within a region, e.g. us-east-1a, us-east-1b etc...

osgi.node.framework

attribute	*data type*	*description*
org.osgi.framework.version	String	The value of the Framework property as obtained via BundleContext.getProperty().
org.osgi.framework.processor	String	The value of the Framework property as obtained via BundleContext.getProperty().
org.osgi.framework.os.name	String	The value of the Framework property as obtained via BundleContext.getProperty().
org.osgi.framework.os. version	String	The value of the Framework property as obtained via BundleContext.getProperty().

osgi.ee

The Framework Node Status service will also provide an **osgi.ee** capability.

Custom capabilities
Custom capabilities can also be provided. TODO need to define plug-in mechanism.

5.2.1.2 DTOs

This section contains a number of pre-defined DTOs available via the FrameworkNodeStatus service.

NetworkAccessDTO

field	*description*
String[] hosts	The external host names or ip addresses for this OSGi Framework, or an empty array if no external hosts names exists.
String[] internalHosts	The internal host names or ip addresses for this OSGi Framework, or an empty array if no internal hosts names exists.
URI[] restEndpoints	The external URL(s) of the framework management REST API, or an empty array if not available.
URI[] internalRestEndpoints	The ecosystem-internal URL(s) of the framework management API, or an empty array if not available.

CPULoadDTO

field	*description*
int processors	The number of processors available to the process.
float processCPULoad	The CPU load for the current process reported as percentage.
long processCPUTime	The amount of nanoseconds of cpu time used by this process.
float systemCPULoad	The CPU load for the entire reported as percentage.

AvailableMemoryDTO

field	*description*
long totalMemory	The total amount of memory available to the process.
long freeMemory	The amount of free memory for the process.

Custom DTOs
Custom DTOs can also be provided via getDTO(MyCustomDTO.class)
TODO need to define plugin mechanism.

5.2.1.3 Framework Node Status Service API

```
public interface FrameworkNodeStatus {
	// Obtains dynamic framework metadata
	String[] listVariableNames();
	String getVariable(String name);
	Map<String, String> getVariables(String ... filter);
	// Capabilities for static metadata
	Collection<Capability> getCapabilities(String namespace);
	boolean matches(Requirement ... requirements);

	// DTOs for dynamic metadata
	<T> T getDTO(Class<T> dtoClass);
}
```

Discussion:

Service properties:

- Benefits:
  - Can use ordinary Remote Service mechanics to communicate these across.
  - Can use service matching mechanics to find the right framework.
- Disadvantages:
  - ?

Capabilities/DTOS:

- Benefits:
  - More modern?
- Disadvantages:
  - Ordinary Remote Service does not work. Need a special client-side proxy to handle the requests.
  - Potentially more chatty over the network.

5.2.2 Obtaining dynamic framework metadata

TODO possibly remove in favor of DTOs

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 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 5.2.3.

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

5.2.3 Application-specific FrameworkNodeStatus metadata

TODO possibly remove in favor of DTOs

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);
  }
}
```

5.2.4 Restrictions

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

5.2.5 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 types.

`osgi.configtype.ecosystem.interface`: 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 that conform to the OSGi DTO rules are supported.

`osgi.configtype.ecosystem.private`: registering the service with the interface specified in the service.exported.interfaces and the `osgi.configtype.ecosystem.private` as the service.exported.configs service registration property will make the service available to service consumers in all frameworks within the ecosystem, but not outside.

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