RFC 183 – Cluster Information

Draft

24 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

0.2 Trademarks

0.3 Feedback

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

0.4 Table of Contents

0.5 Terminology and Document Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in 9.1.

`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
0.11	February, 2013	David Bosschaert, Richard Nicholson, cleanup for EA draft.

Revision	Date	Comments
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.
0.19	September/October 2015	David Bosschaert, feedback from the Turin F2F.
0.20.	November, 2015	David Bosschaert, process feedback from Tim Verbelen, prepare for Chicago F2F.
0.21	December, 2015	David Bosschaert, process feedback from Chicago F2F.
0.22	August, 2016	Tim Verbelen, process feedback from Darmstadt F2F rename to “Node Ecosystems”
0.23	October, 2016	Renamed spec

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. By using of OSGi software becomes adaptable, flexible and reconfigurable. 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.

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.

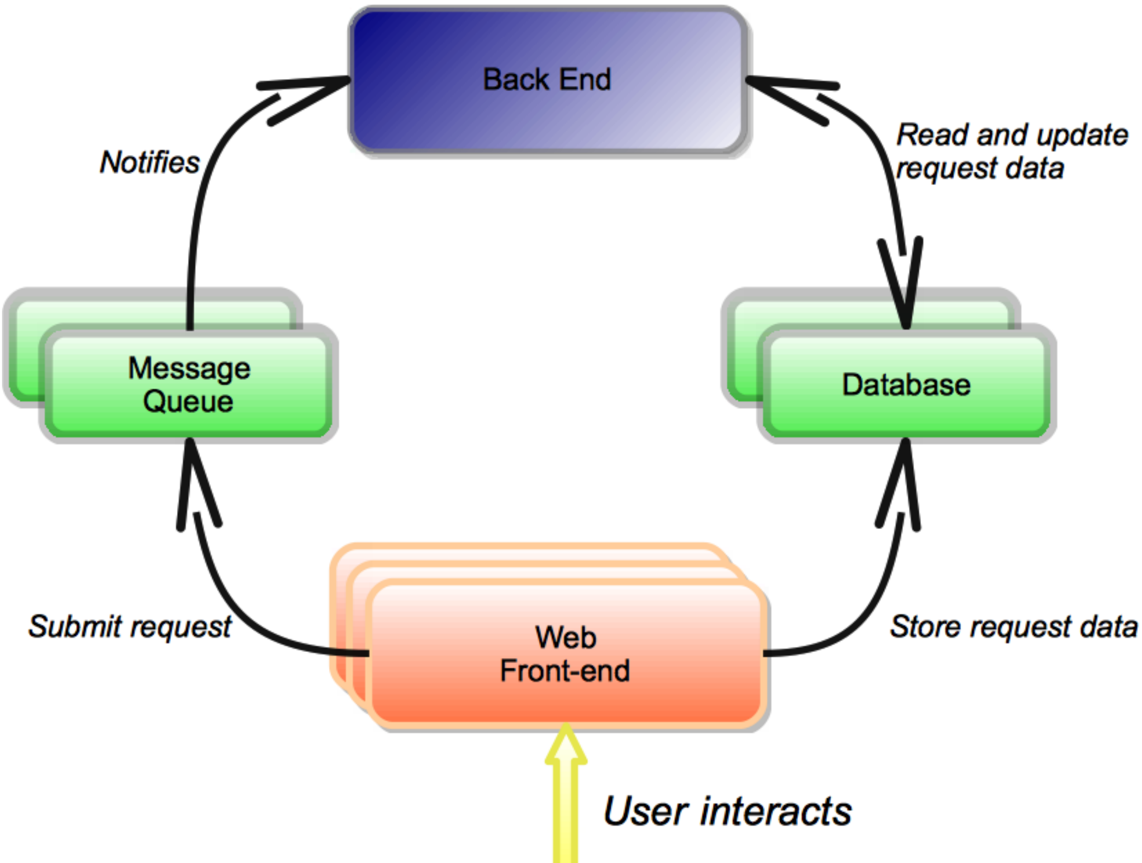


Illustration 1: A possible Application Architecture

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.

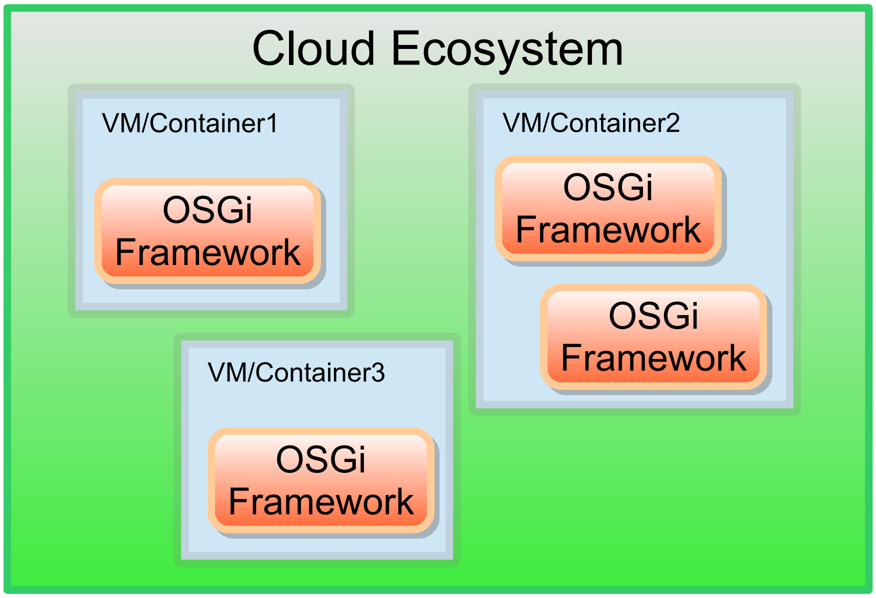
5 Technical Solution

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 separate Java VMs and often on different actual or virtual computing nodes or containers.

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

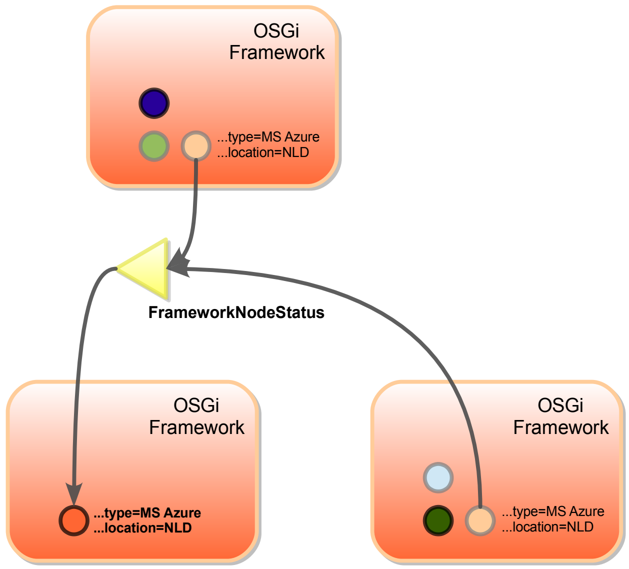
Platforms compliant with this specification provide the following components on each computing node.

1. An OSGi Core Framework as defined by the Core R6 specification.
2. A Remote Services Distribution Provider that supports 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 a `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. Non-framework entities in the system can be registered as a `NodeStatus` service.

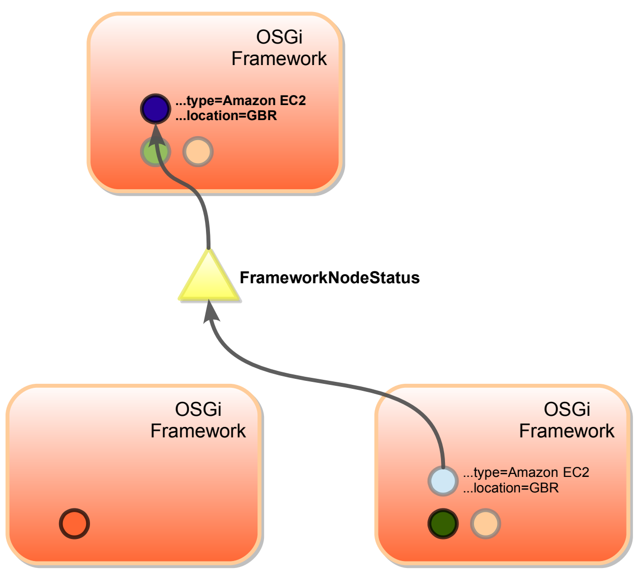
5.2 Presence services

This RFC defines two services that advertise the presence of entities in the Ecosystem: the FrameworkNodeStatus service and the NodeStatus 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 which can be obtained via the service API. Static or mostly static metadata is represented as Service Registration properties, dynamic metadata is represented as via a map. Both can be converted to type-safe DTOs via the object converter.

Additionally a NodeStatus service can be used to represent entities in the cloud ecosystem that are not running an OSGi framework, such as a Database or Load Balancer etc...

5.2.1.1

5.2.2 Node Status Service API

The node status service represents a node in the node ecosystem. This node could represent an entity available in the network that is not necessarily running an OSGi framework, such as a database or a load balancer.

```
public interface NodeStatus {

      Map<String, Object> getMetrics(String ... names)
}
```

Service Properties

key	***data type***	***description***
id	String (mandatory)	The globally unique ID for this node. For example the Docker ID if this note is running inside a Docker container.
parentid	String	Node id of the parent of this node. For example when a Docker container runs within a VM, the parent ID would be the ID of the VM NodeStatus service
ecosystem	String (mandatory)	The name of the Ecosystem this node belongs to. Multiple ecosystems can co-exist within a single cloud system.
endpoint	String+ (optional)	The endpoint(s) at which this node can be accessed from the viewpoint of the consumer of the service.
vendor	String (mandatory)	The vendor name of the Cloud/Environment in which the Ecosystem operates.
version	String (mandatory)	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.
country	String (3, optional)	ISO 3166-1 alpha-3 location where this Framework instance is running, if known.
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.
region	String (optional)	Something smaller than a country and bigger than a location (e.g. us-east-1 or other cloud-specific location)
zone	String (optional)	Regions are often subdivided in zones that represent different physical locations. The zone can be provided here.
tags	String+ (optional)	Tags associated with this node that can be contributed to by the provider and also by bundles.

These service properties can be converted to a NodeStatusDTO using the object convertor to have type-safe access to these properties. The DTO is presented in section 6

5.2.3 Metrics

The NodeStatus provides a getMetrics method to query dynamic properties from the node. Depending on the type of node, these will be other key-value pairs, for example for an OSGi framework this could be CPU and memory usage, for a database node this could be the number of database reads/writes, and for a VM these could be metrics made accessible by the Cloud provider. In this case the service implementator can provide you with the DTOs to have a type-safe way to access these metrics by converting the returned map to one of these DTOs. For example a FrameworkMetricsDTO is presented in Section 6.3.

5.2.4 Framework Node Status Service API

The FrameworkNodeStatus service is a Remote Service representing an OSGi framework running in the ecosystem. It extends NodeStatus and has the following interface.

```
public interface FrameworkNodeStatus extends NodeStatus
                                        implements FrameworkManager{
}
```

Service Properties

The FrameworkNodeStatus service supports all the service properties of the NodeStatus service. In this case the node ID is the framework UUID. In addition the following properties are supported.

key	***data type***	***description***
org.osgi.framework.version org.osgi.framework.processor org.osgi.framework.os.name org.osgi.framework.os.version	String (mandatory)	The value of the Framework properties as obtained via BundleContext.getProperty().
java.version, java.runtime.version, java.specification.version, java.vm.version	String (mandatory)	The values of the corresponding Java system properties.
Other properties set by Remote Services.		

Again, these service properties can also be converted into a FrameworkNodeStatusDTO presented in Section 6.2.

5.2.5 Management API

A Framework Node status service can also register itself under the FrameworkManager interface. This will allow it to remotely manage the framework. An implementation could use the OSGi REST management or another technology to realize this functionality.

The FrameworkManager interface provides a subset of the RestClient interface (the rest specific APIs have been removed). It could be inserted as a base interface of the RestClient interface in the REST spec.

```
public interface FrameworkManager {

      BundleDTO getBundle(long id) throws Exception;

      Map<String,String> getBundleHeaders(long id) throws Exception;
```



```
    Collection<BundleDTO> getBundles() throws Exception;

    BundleStartLevelDTO getBundleStartLevel(long id) throws Exception;

    int getBundleState(long id) throws Exception;

    FrameworkStartLevelDTO getFrameworkStartLevel() throws Exception;

    ServiceReferenceDTO getServiceReference(long id) throws Exception;

    Collection<ServiceReferenceDTO> getServiceReferences() throws Exception;

    Collection<ServiceReferenceDTO> getServiceReferences(String filter) throws
Exception;

    BundleDTO installBundle(String location) throws Exception;

    void setBundleStartLevel(long id, int startLevel) throws Exception;

    void setFrameworkStartLevel(FrameworkStartLevelDTO dto) throws Exception;

    void startBundle(long id) throws Exception;

    void startBundle(long id, int options) throws Exception;

    void stopBundle(long id) throws Exception;

    void stopBundle(long id, int options) throws Exception;

    BundleDTO uninstallBundle(long id) throws Exception;

    BundleDTO updateBundle(long id) throws Exception;

    BundleDTO updateBundle(long id, String url) throws Exception;

}
```

5.2.6 Application-specific NodeStatus metadata

The NodeStatus service provides a tags property. Here, application specific tags can be assigned to the NodeStatus services. For example, one could assign different roles to the nodes such as “worker”, “database”, “storage”, “gateway”, etc. These roles are application-specific and should be defined by the application developer.

5.3 Service Distribution

OSGi Remote Services functionality is available throughout the ecosystem via the `osgi.configtype.ecosystem.interface` 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 that conform to the OSGi DTO rules are supported.

In addition to the configuration type, the ecosystem can also define the intent `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.intents service registration property will make the service available to service consumers in all frameworks within the ecosystem, but not outside, e.g. by binding the endpoint to a private ip address not accessible from outside.

6 Data Transfer Objects

NodeStatusDTO

The service properties of the NodeStatus service can be converted to a NodeStatusDTO

```
public class NodeStatusDTO extends DTO {

	/**
	 * The globally unique ID for this node. For example the Docker
	 * ID if this node is a Docker container or the framework UUID in case of an
OSGi framework.
	 */
	public String id;

	/**
	 * The ID of the parent node if applicable. For example an OSGi framework
node
	 * running inside a Docker container could have the Docker ID as parent.
	 */
	public String parentid;

	/**
	 * The name of the Ecosystem this node belongs to. Multiple
	 * ecosystems can co-exist within a single cloud system.
	 */
	public String ecosystem;

	/**
	 * The endpoint(s) at which this node can be accessed from the
	 * viewpoint of the consumer of the service.
	 */
	public String[] endpoints;

	/**
```

```
     * The vendor name of the Cloud/Environment in which the
     * Ecosystem operates.
     */
    public String vendor;

    /**
     * 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.
     */
    public String version;

    /**
     * ISO 3166-1 alpha-3 location where this node instance is
     * running, if known.
     */
    public String country;

    /**
     * ISO 3166-2 location where this node instance is running,
     * if known. This location is more detailed than the country code
     * as it may contain province or territory.
     */
    public String location;

    /**
     * Something smaller than a country and bigger than a location
     * (e.g. us-east-1 or other cloud-specific location)
     */
    public String region;

    /**
     * Regions are often subdivided in zones that represent different
     * physical locations. The zone can be provided here.
     */
    public String zone;

    /**
     * Tags associated with this node that can be contributed to by
     * the provider and also by bundles.
     */
    public String[] tags;

}
```

6.1 FrameworkNodeStatusDTO

The FrameworkNodeStatus service properties can be converted to a FrameworkNodeStatusDTO:

```
public class FrameworkNodeStatusDTO extends NodeStatusDTO {

    public String frameworkVersion;
```



```
    public String frameworkProcessor;

    public String frameworkOsName;

    public String javaVersion;

    public String javaRuntimeVersion;

    public String javaSpecificationVersion;

    public String javaVmVersion;

}
```

6.2 MetricsDTOs

The metrics map returned by the getMetrics() of NodeStatus can be converted to a matching DTO. For example, a FrameworkNodeStatusService should at least provide these metrics that are available via JMX

```
public class FrameworkMetricsDTO extends DTO {

    /**
     * The number of processors available to the process.
     */
    public int processors;

    /**
     * The CPU load for the current process reported as percentage.
     */
    public float processCPULoad;

    /**
     * The amount of nanoseconds of cpu time used by this process.
     */
    public long processCPUTime;

    /**
     * The CPU load for the entire reported as percentage.
     */
    public float systemCPULoad;

    /**
     * The total amount of memory available to the process.
     */
    public long totalMemory;

    /**
     * The amount of free memory for the process.
     */
    public long freeMemory;

}
```

Other NodeStatus services can provide other metrics and therefore provide other DTO types to access these. For example, an Amazon VM instance NodeStatus could provide the following metrics available through Amazon CloudWatch:

```
public class AmazonMetricsDTO extends DTO {

      /**
       * The number of CPU credits consumed during the specified period.
       */
      public long cpuCreditUsage;

      /**
       * The number of CPU credits that an instance has accumulated.
       */
      public long cpuCreditBalance;

      /**
       * The percentage of allocated EC2 compute units that are currently in use on
the instance.
       */
      public float cpuUtilization;

      /**
       * Completed read operations from all instance store volumes available to the
instance
       * in a specified period of time.
       */
      public long diskReadOps;

      /**
       * Completed write operations to all instance store volumes available to the
instance
       * in a specified period of time.
       */
      public long diskWriteOps;

      /**
       * Bytes read from all instance store volumes available to the instance.
       */
      public long diskReadBytes;

      /**
       * Bytes written to all instance store volumes available to the instance.
       */
      public long diskWriteBytes;

      /**
       * The number of bytes received on all network interfaces by the instance.
       */
      public long networkIn;

      /**
       * The number of bytes sent out on all network interfaces by the instance.
       */
      public long networkOut;
```

```
    /**
     * The number of packets received on all network interfaces by the instance.
     */
    public long networkPacketsIn;

    /**
     * The number of packets sent out on all network interfaces by the instance.
     */
    public long networkPacketsOut;

}
```

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.

Monitor Admin Service Specification

The MonitorAdmin Sevice Specification could also be used to monitor dynamic properties of a node. However, in this spec the goal is not to gather a constant stream of monitoring information, rather get an instantaneous status of the dynamic properties. Also, the MonitorAdmin service uses the StatusVariable objects to pass monitoring information, which are non-DTOs and therefore not easily serializable to use with a remote service admin.

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

Name	Tim Verbelen
Company	iMinds – Ghent University
Address	
Voice	
e-mail	tim.verbelen@intec.ugent.be

9.3 Acronyms and Abbreviations

9.4 End of Document

