CDDLM Deployment API

Steve Loughran

steve_loughran@hpl.hp.com

Abstract

Successful realization of the Grid vision of a broadly applicable and adopted framework for distributed system integration, virtualization, and management requires the support for configuring Grid services, their deployment, and managing their lifecycle. A major part of this framework is a language in which to describe the components and systems that are required. This document, produced by the CDDLM working group within the Global Grid Forum (GGF), provides a definition of the service API whereby a Grid Resource is configured, instantiated, and destroyed.

1. Introduction

The CDDLM framework needs to provide a deployment API for programs submitting jobs into the system for deployment, terminating existing jobs, and probing the state of the system.

This document defines the WS-Resource Framework-based deployment API for performing such tasks.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [12].

2. CDDLM-WG and the Purpose of this Document

The CDDLM WG addresses how to: describe configuration of services; deploy

them on the Grid; and manage their deployment lifecycle (instantiate, initiate, start, stop, restart, etc.). The intent of the WG is to gather researchers, developers, practitioners, and theoreticians in the areas of services and application configuration, deployment, and deployment life-cycle management and to explore the community need for a broader effort in this area. The target of the CDDLM WG is to come up with the specifications for CDDML a) language, b) component model, and c) basic services.

This document describes the SOAP API which a CDDLM service must export, in order for remote callers to create applications on hosts managed by the service.

This document is accompanied by an XML Schema (XSD) file and a WSDL service declaration. The latter two documents are to be viewed as the normative definitions of message elements and service operations. This document is the normative definition of the semantics of the operations themselves.

3. Purpose of the Deployment API

The deployment API describes the SOAP/WS-RF endpoints and attributes and operations thereof, of a means of deploying applications to one or more host computers.

The API is written assuming that the end user is deploying through a console program, a web site or some automated process. This program will be something written by a third party to facilitate deployment onto a grid fabric or other network infrastructure which is running an implementation of the CDDLM infrastructure. That is, the API is not intended for direct invocation by end-user applications, but by front-end applications that provide the interface to the grid fabric.

3.1. Use Cases

There are three different use cases that it is designed to support:

1. The deployment target is an OGSA-compliant Grid Fabric. Resource allocation and Job submission (using JSDL) is part of the deployment process. In this use case, the deployment API must integrate with the negotiation, and deploy a CDDLM-language described system over the machines allocated by the resource manager.
2. The deployment target is a pre-allocated cluster set of machines. The resource allocation process is bypassed -it can be presumed to have happened out of band.

3. One instance of a CDDLM runtime is delegating part of a deployment to another host. There is no guarantee that the two runtimes are the same implementation of CDDLM, or, if they are, that they are the same version.

In all these use cases, there is no expectation that the application will be deployed on the same host that is proving the portal EPR.

3.2. Fault Tolerance

Another aspect of the architecture is that it is intended to be fault tolerant, at least to the extent that *a failure of the portal should not terminate the application, and may not render the application unreachable*.

This is the goal. To be achievable, any set of nodes onto which an application is deployed, must be visible to and manageable by more than one deployment portal. This is similar to how a high-availability web service traditionally requires multiple hosts with a load-balancing front end. The specification avoids depending upon such a facility by noting that WS-ResourceLifetime [bib:ws-rf] permits EPR renewal through WS-Policy metadata contained within the EPR.

A mechanism whereby WS-Policy is used to generate a new application EPR from an EPR that was routed through a failed host is not in this specification. What is included in the specification is a design that is resilient to such an event occurring.

4. Architecture

4.1. Core Architecture

The API comprises a model for deployment, and a WS-ResourceFramework (WSRF) based means of doing so.

A deployment client is an application that wishes to use the deployment API to deploy to a set of machines that have been pre-allocated using a resource allocation system. A deployment portal is required to be operating at or near these machines. This portal exports a WS-RF service endpoint that the deployment client communicates to, in order to deploy applications, and endpoint addressed via a WS-Addressing Endpoint Reference (EPR). This specific EPR will be referred to as the deployment EPR through the remainder of the document.

To deploy, the caller first issues a request to the deployment EPR, to create an application. This creation request returns a new EPR, which provides access to the state and operations of the application, the application EPR. .

With the application EPR, the caller can make a request to initialize the application. This request includes a deployment descriptor and other information that describes and configures the application. The result will be that the application enters the next state in its lifecycle, initialized. This state transition will be asynchronous, as it may take some time to enter the new state.

Once an application has been initialized, it can be started, and later, terminated. An application can also fail, which means that a component in the application has failed. A failed application can only be terminated. The complete lifecycle is defined in Section 4.2

As an application moves through its stages of the lifecycle, it can send lifecycle event notification messages to a registered listener, using WS-Notification. The lifecycle state of the application can also be determined by querying the appropriate property of the application, using the mechanism described in WS-Resource Properties. There is also a synchronous, blocking call to probe the health of an application; this must be routed to the application itself, so that it can determine its own health. This will return its current state, and any custom status information the application chooses to return. If the application has failed, or terminated after a failure, the status information will include the fault information.

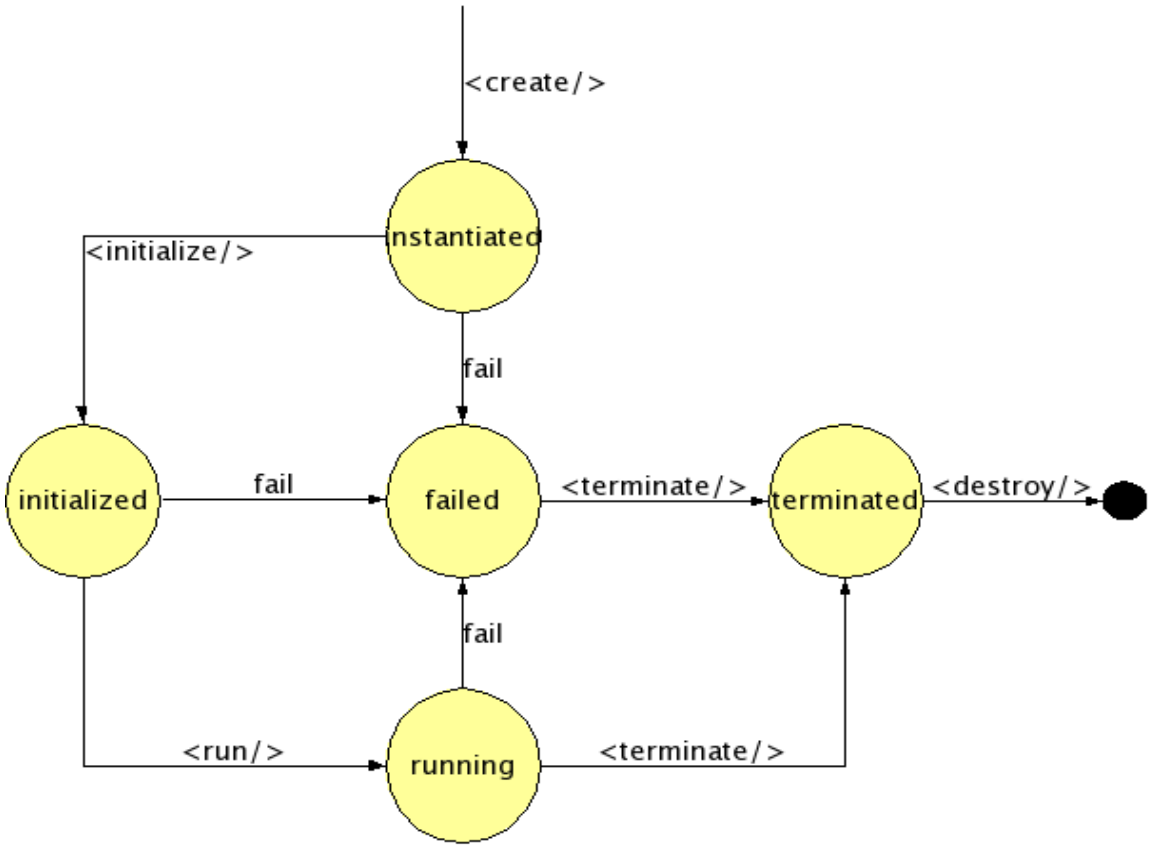
The portal EPR offers some properties and operations alongside the create request. The list of currently deployed applications can be determined, along with their application EPRs. There are also static information and dynamic information documents which can be retrieved from the server; again these are represented as properties following the WS-Resource Properties specification.

The portal EPR also supports WS-Notification events when new applications are created.

4.2. Lifecycle

CDDLM components have a well-defined lifecycle, one that is covered in the component model specification [12]. The lifecycle of a deployment matches the lifecycle of the components within. This is essential to permit delegation, in which a component can be built to manage the entire deployment of an application to a remote system using the deployment API.

Figure 1. The lifecycle of a deployed application



The lifecycle states are listed in Table 1

Table 1. States of an application

State	Meaning
instantiated	the application has just been created
initialized	The application has been initialized
running	The application is running
failed	The application has failed
terminated	The application has terminated

The normative definition of this lifecycle is the component model.

Instantiation and initialization cover the creation and configuration of a component, and when it is moved into *running* then it is actually functioning, The state *failed* is entered automatically when a failure is detected; termination is the only exit condition; `terminated` is the end state of a component and can be entered through a termination request. From terminated, a *destroy* operation will remove all record of the application itself.

The lifecycle is exposed through the operations [1] of the service. The *create* operation is will create and instantiate an application. The run operation will move the application to the running state, and terminate will move it to the terminated state.

4.3. Implementing Fault Tolerance

The design that is intended to enable fault It has the following implications:

- Application EPRs resolve to WS-RF resources that represent views of the application, not the application itself.
- Multiple, different EPRs can map to the same application. For example, EPRs that routed through different portal nodes would be different.
- Application EPR comparison is not appropriate for comparing applications for equality. Multiple EPRs may refer to the same application.
- Every application must have an ID property that must be unique; this can be used for equality tests through simple string comparision
- Issuing a **<wsrl:Destroy/>** request to an application EPR *does not destroy the application*. All that does is destroy that particular view of the application.

- A portal EPR must provide a means of mapping from an invalid application EPR (from a now-defunct portal) to a valid one hosted out of that EPR. This is the underpinning of the renewal mechanism.
- Portal EPRs servicing a set of nodes should be discoverable by a client in some manner. Registration in a service group is one option [WS-ServiceGroup]

4.4. Other Aspects

These are other features of the architecture

4.4.1. Language Agnostic

The deployment API is agnostic as to which particular language is being used, or which version of that language. When a remote deployment is created, the language and version of the descriptor must be supplied. The sole requirement is that it can either be nested inside a SOAP request, or that a URL to the descriptor is remotely accessible to the destination.

Every language is identified by a unique URI. In a create operation, the URI of the language is included, along with the deployment descriptor itself.

4.4.2. Deploy-time properties

Consider a deployment descriptor that declares the host that it wants different components to deploy to. When the descriptor is written, the actual hosts are unknown, so placeholders have to be used. It is only during deployment that the mapping becomes apparent. Either the descriptor is rewritten with the fixed values, or we provide a way for subsidiary information to be passed alongside the descriptor.

The SmartFrog language supports this with the `PROPERTY` and `IPROPERTY` keywords, which bind keys in the Java virtual machine's `java.System.Properties` global HashTable to string and integer values [12]. For example, a deployment descriptor could be bound to three properties:

```
database extends Database {
  sfHostname PROPERTY hosts.database;
  password   PROPERTY database.password;
  localhost  LAZY PROPERTY local.hostname
}
```

At deployment time, the value of the property string is extracted and assigned to the attribute, or a resolution fault is raised. The `LAZY` keyword indicates that the

evaluation is not to take place in the context of the process interpreting the deployment descriptor, but instead the system actually hosting it. As Java System properties are extractable, this can make a significant difference. Although the XML language does not explicitly contain such a feature, a standardized component could be designed to extract the values from the name/value list.

To enable this functionality within the Service interface a set of name/value pairs is one of the options that be specified when creating a deployed application.

4.5. Extensibility

It is inevitable that different implementations of the framework will, over time, have different features that can be used to control and configure a deployment. While the deployment API does not address the need to offer new operations by the service endpoints, it does allow the operation

4.5.1. Extra operations

A service implementation may offer extra operations at the same endpoint as an portal EPR an application EPR. Such extensions must not add new declarations to the XML namespaces used in this document.

4.5.2. Extra WS-Resource Properties

A service implementation may offer extra WS-Resource properties at an EPR.

4.5.3. Extra deployment options

It is possible that extra deployment options will be desired on different implementations or over time. The core of such customization should be in deployment descriptors themselves, yet there may be a need to provide extra deployment metadata.

This is implemented through an **<options>** element in the message that instantiates an application. This (optional) element contains a list of zero or more creation options. These are extra parameters to the deployment request.

The option list is a very powerful aspect of the API, but potentially dangerous. Any protocol standard which has optional aspects is harder to write clients against than one which does not, as there is likely to be less consistency between different implementations.

- All options must be that: optional.
- Every option provides metadata to the deployment infrastructure.
- Every option is named by a URI.
- All URIs that begin with `http://gridforum.org/cddlm/` are reserved for options defined by the CDDLM working group.
- Options can contain string, integer, Boolean or arbitrary XML values.
- All options that an implementation supports must be enumerated in the static server info message
- It is an error to include multiple options of the same URI in a descriptor
- There are no rules as to which order options may be processed. Options must not be designed so that they should be processed in a particular order.

Here are the processing rules for options

1. Option processing must take place before the application is instantiated.
2. An implementation must be able to create an application when the entire options portion of the request is empty.
3. To be "understood", an option must be processed in accordance with the specification of that option. That is, it is not sufficient to recognise an option; it must be acted upon.
4. Any option that is marked `mustUnderstand="true"` MUST be understood. If not, the Fault *"not-understood"* must be raised, identifying the particular option by its URI in the body of the fault.
5. Implementations must not raise this fault when they do not understand any options that are marked `mustUnderstand="false"`, or for which there is no `mustUnderstand` attribute.
6. Duplicate options must cause the operation to be rejected with a `bad-argument` fault.

5. Service Endpoints

These are the endpoints of the service

5.1. Portal Endpoint

The portal endpoint is what the caller initially locates and communicates with. It can be used to create a new application within the set of nodes that it manages, or it can be used to locate an existing application.

Table 2. Properties

Name	Type	Meaning
`staticInfo`		static server info; constant for the lifetime of the portal itself
dynamicInfo		dynamic server info; may be different on every read
`applications`	`xsd:list`	List of application EPRs

Table 3. Properties

Name	In	Out	Meaning
`create`	`TODO`	`wsa:EPR`	Create an application
`lookup`	`xsd:string`	`wsa:EPR`	Map from application na application
`rebind`	`wsa:EPR`	`wsa:EPR`	Take an existi application EP from any porta serving the sa set of nodes, a return an EPR bound to this portal. If the application EP already bound this node, this no-op.

Name	In	Out	Meaning
WS-Resource Lifetime Operations			Operations us by the WS-RL specification t manage the lif of EPR-refere entities
WS-Notification Operations			Operations us by the WS-Notificati specification t enable callers subscribe to supported topi

5.2. Application Endpoint

This represents an application that has been created.

It has an ID property that MUST be strongly unique. That is, it SHOULD be unique for a single deployment of a single application, MUST NOT be re-used, and SHOULD be unique even between different deployment installations. The recommended approach is to use a guid: URI with a properly generated GUID.

Table 4. Properties

Name	Type	Meaning
`name`	`xsd:string`	user-defined name (optional)
`id`	`xsd:uri`	unique name
`deploymentInfo`	`xsd:any`	deployment data
`state`	`xsd:enum state`	current application state
`stateInfo`	`(xsd:any)`	most recent extra state info
`terminationInfo`	`(message, fault, xsd:any)`	termination info
`started`	`xsd:dateTime`	started time
`terminated`	`xsd:dateTime`	end time (not present until application is terminated)

Name	Type	Meaning
`WS-Resource Lifetime Properties`		Properties requires for WSRL

It may also be convenient to formalise timestamp recording as a list of state transitions and the time those transitions occurred.

Table 5. Properties

Name	In	Out	Meaning
`initialize`	`xsd:any`		Initialize an application; pa the componen descriptor and build up the component gra
`run`	`xsd:string Message`		Start running initialized application
`ping`			
`resolve`	`xsd:string path`	`xsd:any`	Resolve a reference relat to this applica Can return EP components; s or other data
WS-Resource Lifetime Operations			Operations us by the WS-RL specification t manage the lif of EPR-refere entities
WS-Notification Operations			Operations us by the WS-Notificati specification t enable callers subscribe to supported topi

6. Notification

Notification enables front-end applications to receive notification when an application finishes. It also enables management tools to track the number of running applications.

All implementations of the deployment API must support WS-Notification (WS-N), as specified in the document. The implementations are free to implement alternate mechanisms; that is beyond the scope of this document. What is covered, however, is a means of listing all notification mechanisms supported by an implementation. Every server instance is required to enumerate all supported mechanisms in a list included in its static server information document; this can be used by callers to choose which mechanism is appropriate.

6.1. Notification Policy

- Implementations MUST support WS-Notification.
- Implementations MAY support alternate notification mechanims.
- Implementations MUST list all supported notification mechanisms in the `staticInfo` information.
- Implementation must support the topics defined below, on the relevant EPR types.
- Implementations MAY also support Terminate notification events of WS-ResourceLifetime, which are raised after an EPR is destroyed.
- There will be one notification for lifecycle events of applications
- There will be one notification for the portal EPRs, which is raised when an application is created.
- There is no guarantee of fault tolerant subscriptions. Implementations MAY include WS-Policy metadata that informs callers how to renew subscriptions in the event of system failure.

6.2. WS-Notification Support

As stated above, implementations MUST support WS-Notification; this does not prevent them also implementing alternate mechanism.

Application EPRs support a WS-TopicSpace that contains one topic: lifecycle events

Portal EPRs support a A WS-TopicSpace that contains one topic: application addition events.

6.3. Fault-Tolerant Notification

Fault tolerance is an extra complication; because the origin EPR is included in each notification, the sender must know its EPR. If an app is visible through more than one portal, it must know the EPR used by a subscriber, and return that EPR with the request. If the application/portal EPR is changed when the subscriber switches to a different portal, then the EPRs in the notifications must also be updated.

The easy way to do this is to have the portal manage the notifications, and do not require subscription EPRs to be fault tolerant. If a portal fails, the subscriptions are lost. As they are renewable anyway, this is no real hardship. The biggest risk that a subscriber does not know that a subscription has been lost, and so an event is missed.

An alternative is for the application to retain all knowledge about subscribers on the node on which it runs, including the EPR by which each subscriber knows it. This adds an interesting feature: the ability for an application to send a prescriptive 'EPR changed' notification for the benefit of all relevant subscribers, when the EPR in use was no longer available.

7. Fault Policy

7.1. Origin of Faults

Faults are raised in response to errors either at the remote endpoint, in the local framework, or between the remote endpoint and other parts of the distributed system.

7.1.1. Local faults

Local faults will inevitably be raised as the appropriate fault for the system. For example, the Apache Axis SOAP client raises `AxisFault` faults for all SOAP events, wrapping stack trace and even HTTP Fault data within the fault as DOM elements.

7.1.2. Service Faults

These are the faults that are raised by the service. They are categorized into a hierarchy of WS-BaseFault faults.

7.1.3. Relayed Faults

Relayed faults are those received by the far end and passed on. They may be WS-BaseFault (WS-BF) Faults; HTTP error codes, SOAP faults, native language faults wrapped as SOAPFaults, or predefined deployment faults.

WS-BaseFault uses fault nesting for relaying faults; however, all faults must be a derivative of WS-BaseFault. This is addressed by defining a new WS-BaseFault derivative, a wrapped SOAPFault.

7.2. Fault List

The normative list of faults is described in the XML Schema documents that accompany this document. *Appendix B* is generated from an XSLT transformation of this fault list.

8. Out of scope

The following activities are not covered by the service API.

8.1. Resource Allocation

The API assumes that the appropriate resources for an application have been allocated by the underlying infrastructure. Furthermore, information provided in or alongside the deployment infrastructure provides explicit information mapping the components of the application to specific resources.

That is: it is not the role of the deployment API to decide where application components should go, merely to enact decisions already made.

8.2. File upload

Uploading files is not addressed. We are assuming that other mechanisms -specifically a JSDL-based front end to the system- will address that problem by

retrieving the files and offering them somewhere that deployment hosts can retrieve by means of a URL.

We do need a way of knowing the mapping from the identification of files in the job submission. This can be either by having the front end generate a new deployment descriptor which includes this information, or by providing a name/value mapping list alongside the descriptor.

8.3. Discovery and binding

We assume that registration of, discovery of and binding to instances of the deployment portal is handled by external discovery and binding mechanisms.

8.4. EPR Replacement

We do not cover how EPRs held against a failed server are renewed.

9. Compliance

This section covers requirements and non-requirements for compliance with the specification.

9.1. Non-Requirements

These are things which are explicitly not required for compliance. That does not mean that implementations MUST not support them, but that they MAY support them if desired. There will be no tests for these features in any compliance test suite.

- There is no requirement to support WS-MetaData Exchange
- There is no requirement to implement get/set multiple properties, and if implemented, no requirement for atomicity.
- There is no requirement to implement QueryResourceProperties
- If a component supports WS-MetaDataExchange, there is no requirement for the list of resources and operations above and beyond the set we define to remain constant for any period of time. That means an application EPR can add and remove attributes, and the metadata could dynamically list those attributes, but they could be removed at any time.

Bibliography

WS-Resource Framework, 2004.

Notes

1. In this document, operations, are taken to mean message exchanges between caller and the relevant WS-Addressing EPR-referenced endpoint.