Configuration Description, Deployment, and Lifecycle Management

A Service API for Deployment

Draft 2005-01-24

This is an interim working draft for con

Status of this Memo

This document provides information to the community regarding the specification of the Configuration Description, Deployment, and Lifecycle Management (CDDLM) Language. Distribution of this document is unlimited. This is a DRAFT document and continues to be revised.

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.

GLOBAL GRID FORUM

office@ggf.org
www.ggf.org

1 Table of Contents

2 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. It is targeted at implementors and users of the API.

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 [RFC2119]

3 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 defines the WS-Resource Framework-based deployment API for performing such tasks. A CDDLM deployment infrastructure must implement this service in order for remote callers to create applications on the infrastructure.

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.

4 Purpose of the Deployment API

The deployment API is the SOAP/WS-ResourceFramework (WS-RF) API for deploying applications to one or more target computers, physical or virtual.

The API is written assuming that the end user is deploying through a console program, a portal UI 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 the relevant CDDLM services.

4.1 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 the JSDL language [JSDL] or equivalent) 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.

4.2 Fault Tolerance

he architecture is intended to support fault tolerant implementations, to the extent that *a failure of the deployment endpoint may not terminate the application, and may not render the application unreachable.*

To achieve this goal, any set of nodes onto which a system is deployed, must be visible to and manageable by more than one deployment endpoint. Furthermore, if the failure of this endpoint is not to prevent access, any SOAP endpoints that provide direct access to the system, must be hosted on the system nodes themselves.

5 Architecture

5.1 Core Architecture

The API comprises a model for deployment, and a WS-ResourceFramework [WS-RF] based means of interacting with this model.

A *deployment client* is an application that wishes to use the deployment API to deploy to one or more hosts that have been pre-allocated using a resource allocation system. A *deployment portal* is 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) [WS-A]. This specific EPR is referred to as the *portal EPR*.

To deploy, the client first issues a request to the *portal EPR* to create a system. This request includes a deployment descriptor in one of the CDDLM supported languages and potentially other information that describes and configures the application. This creation request returns a new EPR, which provides access to the state and operations of the system, the *system EPR*.

The system EPR can be bound to any node that the portal EPR chooses; there is no requirement that it is bound to the same portal node. For maximum availability, hosting the system EPR on the same node of the system may be the best approach. An example of this is shown in figure 1.

The caller can then make a request to the *system EPR* to *initialize* the system. If successful, the application asynchronously enters the next state in its lifecycle, *initialized*. Once a system has been initialized, it can be moved through other stages of its lifecycle. The complete lifecycle is defined in section 5.2.

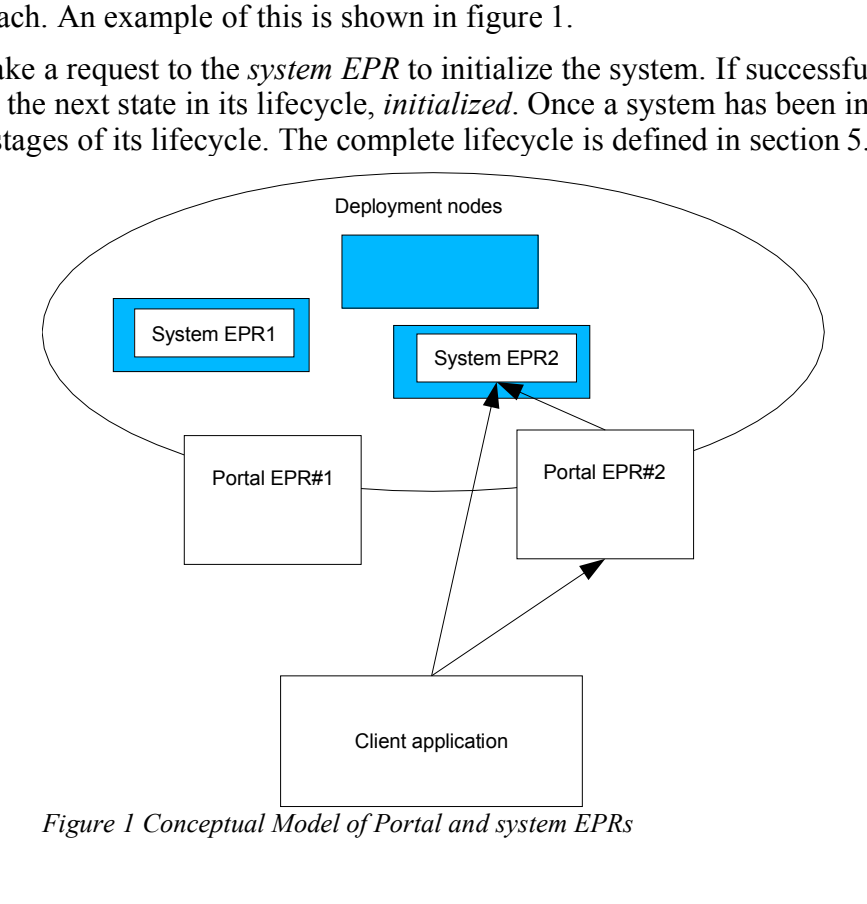

Figure 1 Conceptual Model of Portal and system EPRs

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

The portal EPR supports other properties and operations. The list of currently deployed systems can be determined, along with their system EPRs. There are also static information and dynamic information about the portal which can be retrieved from the server; again this information is represented as properties following the WS-Resource Properties specification.

The portal EPR must be able to raise events when new systems are created, using the WS-Notification protocol at a minimum.

5.2 Lifecycle

CDDLM components have a uniform lifecycle, one that is normatively described in the component model specification [Schaeffer05]. The lifecycle of a deployment matches the lifecycle of the components within. This is essential to permit aggregation of systems.

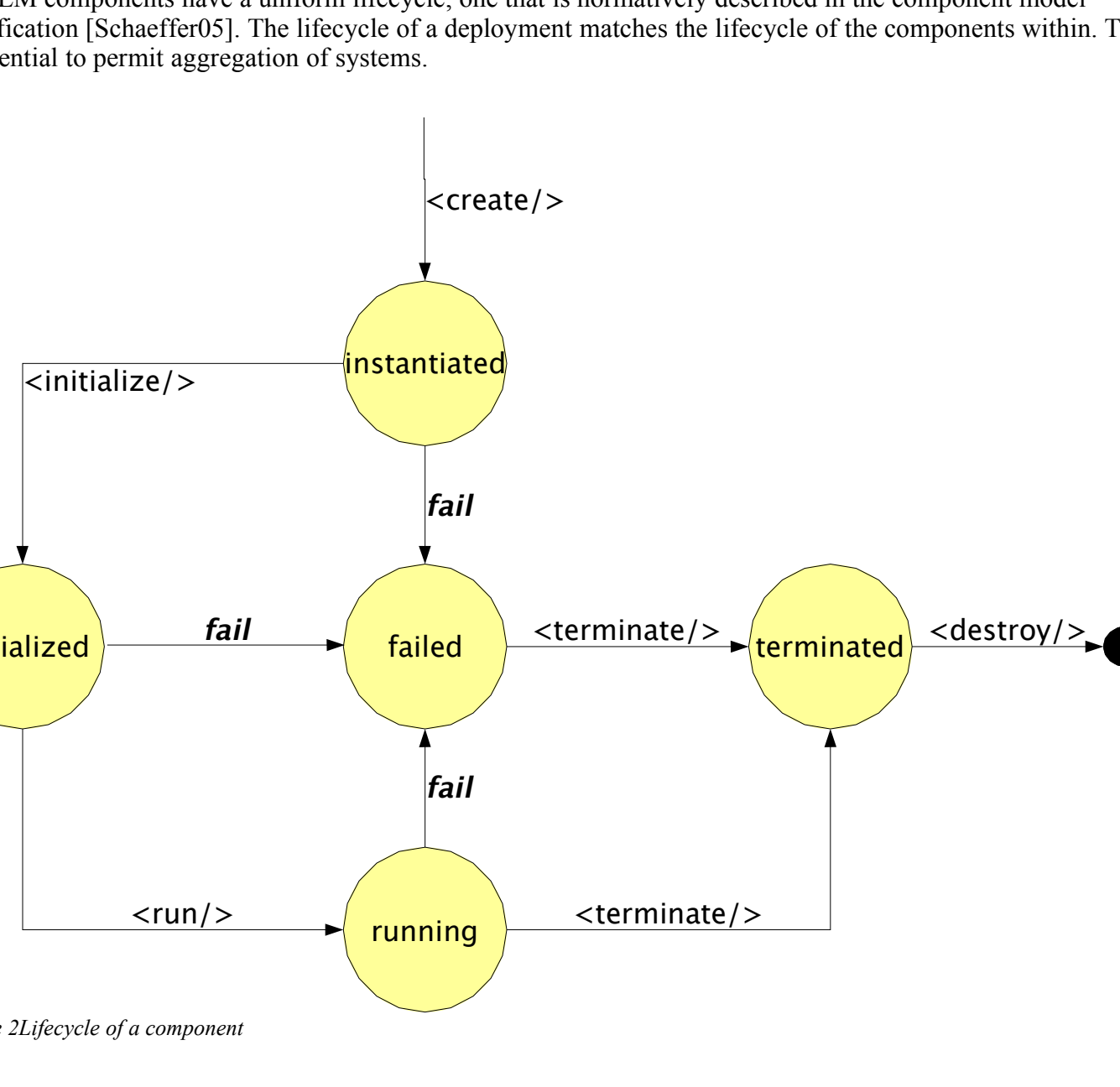

Figure 2Lifecycle of a component

The states of an system are as follows:

instantiated	The system has just been instantiated.
initialized	The system has been initialized.
running	The system is running
failed	The system has failed
terminated	The system has terminated
destroyed	The system is destroyed.

The normative definition of this lifecycle is the component model [CITE]. Instantiation and initialization represent the creation and configuration of a component, and when it is moved into *running* then it is actually functional. The *failed* state is entered automatically when a failure is detected; termination is the only exit condition; *terminated* is the end state of a component and and can be entered through a termination request.

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

5.3 Fault Tolerance Support

As stated, the architecture is must enable fault tolerant implementations. Here is how this is enabled:

- Multiple Portal EPRs can provide access to the same set of nodes.
- The failure of a portal does not imply the failure of a system.
- The failure of a node hosting a system EPR will result in the destruction of that system.
- Issuing a <destroy/> request to a system EPR *will destroy the system.*
- Every system instance must have a WS-RF property "ID" of type xsd:URI property that must be unique; this can be used for equality tests through simple string comparison.
- 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].
- Implementations may implement fault tolerant Portal EPRs through the use of a dynamic DNS service, one in which the CNAME entry for the hostname(s) of the portal are updated as portal instances appear and disappear. Client systems should be written with the knowledge that the IP addresses of an EPR may change, and not to cache the resolved IP address indefinitely[2].

5.4 Other Architectural Features.

5.4.1 Named systems

Callers may provide a string name for a system. This system name, if provided, must be unique amongst all systems that a portal EPR can manage.

The system name must begin with one of the characters in the set [A-Za-z_] and continue with characters in the range [A-Za-z0-9_.-]. This is a proper subset of the XSD type NCName element names, and is also a subset of the valid characters in a URL. This is intentional, and while the specification does not take advantage of either fact, implementations may.

5.4.2 Deployment Language Agnostic

The deployment API is agnostic as to which particular language, or version thereof, is used for a deployment descriptor. When a remote deployment is requested, the language and version of the descriptor must be supplied. The sole requirement of the language is that it can either be inserted inside an XML document, or that a URL to the descriptor is correctly accessible to the destination. In the case of a URL, the URL to the descriptor must provide which information.

Every language is identified by a unique URI. This language URI must be supplied with the deployment descriptor or URL.

5.4.3 Job Language Agnostic

Just as the API allows implementations to support deployment languages/versions, the API also permits multiple Job specification languages. That is, alongside JSDL, an implementation may support the Globus Resource Specification Language [GlobusRSL].

5.4.4 Deploy-time properties in the language and service API

Consider a deployment descriptor that wants to control onto which machine that it wants deployment can be deployed onto. When the descriptor is written, the actual hostnames are unknown. 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 substitution information to be passed alongside the descriptor.

The SmartFrog language [Goldsack04] supports this with the PROPERTY and IPROPERTY keywords, which bind keys in a Java System.Properties hashtable to string and integer values. For example, a deployment descriptor could be bound to three properties:

```
database extends Database {
    dbHost:host PROPERTY host.database;
    password PROPERTY database.password;
    localhost  LAZY PROPERTY local.hostname;
}
```

At deployment time, each property string is looked up and assigned to the attribute, or a fault is raised. The LAZY keyword indicates that the evaluation must not take place in the context of the process interpreting the deployment descriptor, but instead by the host actually hosting it. The XML language does not explicitly contain such a feature [XML-CDL], a standardized component could be designed to extract the values from the name/value pair.

To enable this functionality within the Service interface, one of the deployment options declares a set of name/value pairs. How these tuples are exposed to a deployment language/framework is a language-specific feature.

5.4.5 Extensibility

The deployment API is designed to support extensible implementations, and future enhancements to the API over time.

Extra Operations

A service implementation may offer extra operations at any EPR. Such extensions must not add new elements to the XML namespaces used in this document: they must be in their own, private, namespace. Implementations should document these operations and provide updated WSDL descriptions.

There is no requirement for the extra operations supported by an EPR to remain constant over any period of time.

Extra WS-Resource Properties

A service implementation may offer extra WS-Resource properties at any EPR. Again, they must be in their own, private, namespace. Implementations should document these properties and provide updated WSDL descriptions.

Extra deployment options

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

This is integrated through an <options> element in the <initialize> message. This (optional) element contains a list of zero or more deployment options. Every option is named by a URI, and contains either a boolean, string, integer value, or XML data; the XML is a single element. Options also have a mustUnderstand attribute which is used to indicate whether or not an option must be understood.

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 interoperability between different implementations. To manage this, the deployment API has the following requirements on all optional metadata parameters:

- All options must be that optional. It must not be an error to deploy with no options declared.
- Every option is named by a URI.
- All URIs that begin with http://gridforum.org/cddlm/ are reserved for the CDDLM working group.
- Options must contain either string, integer Boolean or arbitrary XML content. String and integer values are supported via attributes; XML is supported as nested data.
- An options must contain only one value type. Implementations must raise a fault if multiple are nested or attribute values are declared on the same option.
- All options that are implemented must be enumerated in the server information property of the portal EPR.
- It is an error to include multiple options of the same URI in the same request. Implementations must raise a fault when this occurs.
- Options may be processed in any order. Options must not require a specific order of processing.
- Service implementations must ignore any options that they do not recognize, if mustUnderstand="false" for that option.
- Service implementations must understand all options with mustUnderstand="true" for that option. If any such option is not understood, a fault must be raised.

The processing rules for deployment are as follows:

1. Option processing must take place before the system is started.
2. An implementation must be able to create a system even when the options portion of the request is empty or absent.
3. To be "understood", an option must be processed in accordance with the specification of that option.
4. Any option that is marked mustUnderstand="true" MUST be understood. If not, the Fault "not-understood" must be raised, with its URI in the extraData field of the fault.
5. Implementations must not raise this fault when they do not understand any options that are marked mustUnderstand="false", for which there is no mustUnderstand attribute.
6. Duplicate options of the same URI must be rejected with a bad-argument fault, identifying the particular option by its URI in the extraData field of the fault.

6 Deployment API Overview

The service API consists of two component types, portal endpoints, addressed by portal EPRs, and system endpoints, addressed by system EPRs. Portal EPRs return system EPRs to callers, either in response to lookup/mapping messages, or when a system is successfully created.

The two endpoint types are *Resources* within the terminology of the WS-Resource Framework specifications.

In this section of the document, the following listed prefixes refer to the stated namespaces

prefix	*URI*	*description*
xsd	http://www.w3.org/2001/XMLSchema	XML Schema Types
wsa	http://schemas.xmlsoap.org/ws/2003/03/addressing	WS-Addressing Types
api	http://www.gridforum.org/cddlm/serviceAPI/api/2004/10/11/	Deployment API
cddlm-types	http://www.gridforum.org/cddlm/serviceAPI/types/2004/10/11/	API types
cddlm-faults	http://www.gridforum.org/cddlm/serviceAPI/faults/2004/10/11/	API Faults
cdl	http://www.gridforum.org/2004/12/CDDLM/XML-CDL/1.0	XML CDL
cmp	http://www.gridforum.org/cddlm/components/2004/11/06	Component Model
wsrf-bf	http://www.ibm.com/xmlns/stdwip/web-services/WS-BaseFaults	WS-BaseFaults
wsrf-rl	http://www.ibm.com/xmlns/stdwip/web-services/WS-ResourceLifetime	WS-Resource Framework
wsrf-rp	http://www.ibm.com/xmlns/stdwip/web-services/WS-ResourceProperties	WS-Resource Properties
wsnt	http://www.ibm.com/xmlns/stdwip/web-services/WS-Notification	WS-Notification
wsrf-sg	http://www.ibm.com/xmlns/stdwip/web-services/WS-ServiceGroup	
env	http://www.w3.org/2003/05/soap-envelope	SOAP1.2 Envelope
xml	http://www.w3.org/XML/1998/namespace	XML attributes

6.1 Portal Endpoint

The portal endpoint is the endpoint which the caller initially locates and communicates with. It can be used to create new system instances from deployment descriptor packages, and it can be used to locate existing systems.

6.1.1 Portal EPR Properties

Name	*Type*	*Meaning*
staticPortalStatusProperty	cddlm-api:staticPortalStatusType	static server info; constant for the lifetime of the portal itself
dynamicPortalStatusProperty	cddlm-api:dynamicPortalStatusType	dynamic server info; may be different on every read
systemListProperty	cddlm-api:systemReferenceListType	List of system EPRs
WS-Resource Lifetime Properties		Properties required for WSRL

6.1.2 Portal EPR Operations

Name	*In*	*Out*	*Meaning*
create	name: xsd: string jsdl: xsd:any descriptor: xsd:any	wsa:EPR	Create a system.
lookupSystem	xsd:string	wsa:EPR	Map from system name to a system EPR
WS-Resource Lifetime Operations			Operations used by the WS-RL specification to manage the lifetime of EPR-referenced entities
WS-Notification Operations			Operations used by the WS-Notification specification to enable callers to subscribe to supported topics.

6.2 System Endpoint

This represents a system that has been deployed. System EPRs are obtainable by creating one at the portal EPR, or through lookup operation offers by a portal.

6.2.1 System EPR Properties

Name	*Type*	*Meaning*
systemNameProperty	xsd:string	user-defined name (optional)
systemIdentifierProperty	xsd:anyURI	unique identifier
systemStateProperty	cddlm-api:lifecycleStateEnum	current system state
stateInfo	xsd:string	Text state info
systemExtendedStateProperty	cddlm-api:unboundedXMLAnyNamespace	Component state
systemCreatedProperty	xsd:dateTime	Time system was created
systemStartedProperty	xsd:dateTime	Time system was terminated
systemTerminatedProperty	xsd:dateTime	end time (not present until system is terminated)
systemTerminationRecordProperty	cddlm-api:terminationRecordType	termination record
WS-Resource Lifetime Properties		Properties required for WSRL

6.2.2 System EPR Operations

Name	*In*	*Out*	*Meaning*
init	job cddlm-api:jobDescriptorType descriptor cddlm-api:deploymentDescriptorType	cddlm-api:void	Initialize a system; pass in the job and component descriptors and build up the component graph.
addFile	uri xsd:anyURI mimetype xsd:string data xsd:base64Binary	xsd:integer	Add a file to this document so that it is accessible by a URI from within the deployment descriptor.
run	cddlm-api:void	cddlm-api:void	Start running an initialized system
ping	cddlm-api:void	cddlm-api:status	Probe a system's health.
terminate	xsd:string Message	cddlm-api:void	Terminate a system; pass in a message
resolve	xsd:string path	xsd:any	Resolve a reference relative to this system. Can return EPRs to components; string or other data
WS-Resource Lifetime Operations			Operations used by the WS-RL specification to manage the lifetime of EPR-referenced entities
WS-Notification Operations			Operations used by the WS-Notification specification to enable callers to subscribe to supported topics.

7 Notification

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

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 property this can be used by callers to choose which mechanism is appropriate.

7.1 Notification Policy

- Implementations MUST support WS-Notification.
- Implementations MAY support alternate notification mechanisms.
- Implementations MUST list all supported notification mechanisms in the staticInfo information.
- Implementations 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 system lifecycle events.
- There will be one notification for the portal EPRs, which is raised when a system 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.

7.2 WS-Notification Support

As stated above, implementations MUST support WS-Notification; this does not prevent them also implementing supplementary mechanisms. There are specific topic spaces [WS-Topics] define:

- Portal EPRs must support a WS-TopicSpace that contains one topic: system addition events.
- System EPRs must support a WS-TopicSpace that contains one topic: lifecycle events

7.3 Fault-Tolerant Notification

Implementations are not required to provide fault-tolerant notification. The failure of portal may result in the loss of portal event subscriptions, and the failure of a system may result in the loss of system event subscriptions.

8 Fault Policy

Faults are based upon the WS-BaseFault model [WS-BF], taking on some of the lessons of [Loughran02], namely that extra information such as hostname and process is essential for locating which process among many has failed on a clustered system.

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. They can be returned to callers in response to a an operation on an endpoint, or sent as part of a notification event.

All faults that will be explicitly sent are derived from WS-BaseFault faults. Service implementations may implicitly raise SOAPFault faults, as that is inherent in most implementations.

8.1 Fault Categories

8.1.1 Service Faults

These are the faults that are raised by the service. They are grouped into a hierarchy of WS-BaseFault faults. There is a base fault class DeploymentFault, from which all others are derived.

All Service interfaces must declare that they raise these DeploymentFault instances, rather than list the specific faults. This is to provide forward extensibility.

The API lists specific subclassed faults of DeploymentFault that may be generated by a service or received by a client. These faults represent some of the faults that a service implementation may send.

If an implementation has a fault state whose meaning matches that of the predefined fault, the predefined fault must be thrown. If this predefined fault has standard elements for embedded fault information, the implementation should fill them in. The implementation may add implementation-specific data within the extra-data element of the fault, to supplement this information. This extra data must not add new types to the XML namespaces of this deployment data. The XML schema and semantics of this extra data should be documented.

If an existing fault type is not suitable, implementations may create new fault types.

If an implementation creates new fault types, these must extend the existing fault types which operations are declared as throwing, which effectively means that they must extend DeploymentFault. These new faults must not change the XML schemas of the deployment API, and they must be in a new namespace. The new faults and XML content should be publicly documented.

If an implementation adds new operations or properties at the existing endpoints, these new operations may raise whatever faults they see fit, within the constraints of the WS-BaseFaults specification. Again, the implementation must not add new types to the deployment API namespace.

8.1.2 Transport faults

Transport 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. Microsoft .NET WSE has a similar fault class.

8.1.3 Relayed Faults

Relayed faults are those received by the far end and passed on. They may be WS-BaseFault 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 WrappedSOAPFault. This type is actually an extension of DeploymentFault. This fault can nest any received SOAPFault, with an element containing the received XML data. Well-known elements in this fault data (such as the Apache Axis stack trace and HTTP fault code) should be copied into any fields in the main fault which fill the same role.

8.2 Fault Security

Sites offering deployment services, may, for security reasons, wish to strip out some information, such as stack trace data. Implementations should provide a means to enable such an action prior to transmitting faults to callers.

Host name and process information may be viewed as sensitive, yet again, this is exceedingly useful to operations. Implementations may provide a means to disguise this information, so that it does not describe the real hostname or process ID of a process, but instead pseudonyms that can still be used in communications with any operations team.

8.3 Internationalisation

The WS-BaseFault specification makes no statement upon which language error descriptions are in.

If an implementation can return descriptions in one language, it must use xml:lang attributes to indicate the language of a description. Multiple descriptions, in different languages may be included. The client application should extract the description(s) whose language is the nearest match to that of the client.

8.4 Fault Type Declarations

The fault hierarchy is shown in Figure 3.

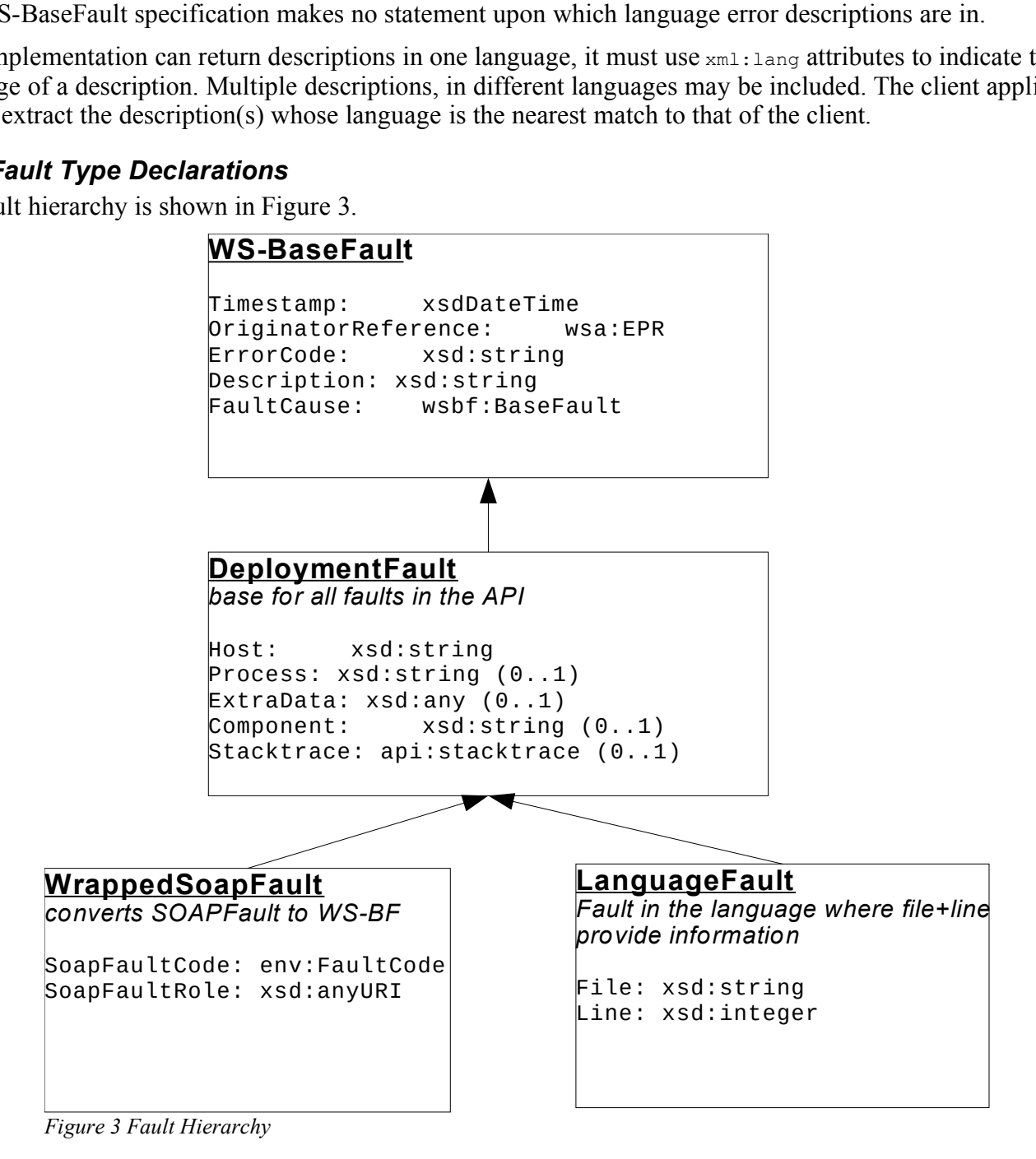

Figure 3 Fault Hierarchy

8.4.1 DeploymentFault

This type represents any fault thrown during deployment. All endpoint operations must declare that they throw this fault, and must not declare that they throw any derivative fault.

Element	*Type*	*Meaning*
Host	xsd:string	Hostname or pseudonym
Process	xsd:string	Any process identifier suitable for diagnostics
ExtraData	xsd:any	Extra fault data
Component	xsd:string	Path to component raising the fault
Stacktrace	api:stacktrace	Stack trace of fault

Implementations must include a component reference if it is known. Implementations should include hostname and process information. Process information may be a low level identifier (such as an operating system process ID), or it may be some application specific identifier. Its role is merely to distinguish which process amongst many in a load-balanced implementation raised the fault.

8.4.2 LanguageFault

A language fault represents any fault in language processing for which a file and line number are relevant.

Element	*Type*	*Meaning*
File	xsd:string	Filename/URI of file at fault
Line	xsd:integer	Line number within the file

This information must be included.

If the error is in the inline deployment descriptor, the File element must be empty. Furthermore, the Line element must be relative not to the deployment request, but to the inline descriptor. Recipients of faults can then infer from the empty file element that the

Note that a consequence of this design is that implementations must preserve white space in the deployment descriptor when saving them to file.

8.4.3 WrappedSOAPFault

This type represents a mapping of a classic W3C SOAPFault [SOAP1.2] to a WS-BaseFault, as an extension of DeploymentFault. It adds two new elements to contain data unique to SOAPFaults.

Element	*Type*	*Meaning*
SoapFaultCode	env:FaultCode	Fault code information
SoapFaultRole	xsd:anyURI	Role of sender

The normative mapping of SOAPFault elements to WrappedSOAPFault elements is as follows:

SOAP1.2	*WrappedSOAPFault*
/env:Code	/api:SoapFaultCode
/env:Role	/api:SoapFaultRole
/env:Detail	/api:ExtraData
/env:Reason/env:text	/wsrf-bf:Description

Any text elements under env:Reason must be converted into separate description elements in the fault; all xml:lang attribute must be preserved.

Detail from SOAP stacks with well-known fault fields, such as the Apache Axis stack trace, may be imported into appropriate fields in the DeploymentFault.

8.5 Fault Error Codes

Specific fault error codes, and their meaning, are covered in a separate document.

Every unique fault will be described by its own fault code. Deployment faults that are part of the API specification will all be in the namespace http://X/Y/Z with their code value described in the CDDLM Fault Specification.

9 Security

The deployment requests must only be granted by suitably authorized individuals, or their suitably authorized agents.

For deployment to a Grid infrastructure, that means that the standardized security model of the infrastructure must be used to authenticate callers. Only callers with the relevant rights may deploy systems.

When delegating deployments across nodes, the node issuing the deployments needs to have the rights to do so, and the deployment itself still needs to be authenticated as a legitimate request of the sender.

Along with deployment, the ability of a caller to list and manipulate running systems, introduces another security issue: that of who has access to the set of deployed systems.

10Editor Information

Steve Loughran, HP Laboratories

11References

[Axis] Apache Software Foundation, *Apache Axis*,
[Foster04] Foster et al., *Modeling Stateful Resources with Web Services*, 2004.
[Goldsack04] Goldsack, *SmartFrog Language*, 2004
[GlobusRSL] Globus, Resource Specification Language, 2004
[JSDL] *Job Service Description Language*, 2004.
[Loughran02] Loughran, *Making Web Services that Work*, HP Laboratories, TR-HPL-2002-274, 2002.
[Parastatidis03] Parastatidis et al., *A Grid Application Framework based on Web Services Specifications and Practises*, University of Newcastle, 2003.
[RFC2119] S. Bradner, *RFC 2119 - Key words for use in RFCs to Indicate Requirement Levels*, 1997
[Schaeffer05] Schaeffer., *CDDLM Component Model Specification*, 2005
[SOAP1.2] W3C, *SOAP Version 1.2*, 2003.
[XML-CDL] *CDDLM XML Configuration Description Language Specification version 1.0 draft 2004-12-10*.
[WS-A] Gudgin, M. and Hadley S., *Web Services Addressing -Core*, 2004.
[WS-BF] Tuecke et al., *Web Services Base Faults (WS-BaseFaults)*, 2004.
[WS-BaseNotification] Graham et al., *Web Services Base Notification 1.0 (WS-BaseNotification)*, 2004.
[WS-BrokeredNotification] Graham et al., *Web Services Brokered Notification 1.0 (WS-BrokeredNotification)*, 2004.
[WS-Policy] Schlimmer et al., *Web Services Policy Framework (WS-Policy)*, 2004
[WS-ResourceLifetime] Frey et al., *Web Services ResourceLifetime 1.1 (WS-ResourceLifetime)*, 2004.
[WS-RF] Tuecke et al., *Web Services Resource Framework (WS-RF)*, 2004.
[WS-ResourceProperties] Graham et al., *Web Services Resource Properties 1.1 (WS-ResourceProperties)*, 2004.
[WS-ServiceGroups] Graham et al., *Web Services Service Group Specfication 1.0 (WS-ServiceGroups)*, 2004.
[WS-Topics] Graham et al., *Web Services Topics (WS-Topics)*, 2004.

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

[2] This is of particular relevance to Java applications, where the default behavior is to cache the resolved address of a hostname for the duration of the application.