Configuration Description, Deployment, and Lifecycle Management

CDDLM Deployment API

Draft 2005-02-08

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.

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 those who implement either end 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]

2.1 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.

3 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.

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 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.

3.2 Fault Tolerance

The 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 be 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.

4 Architecture

4.1 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 ore 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.

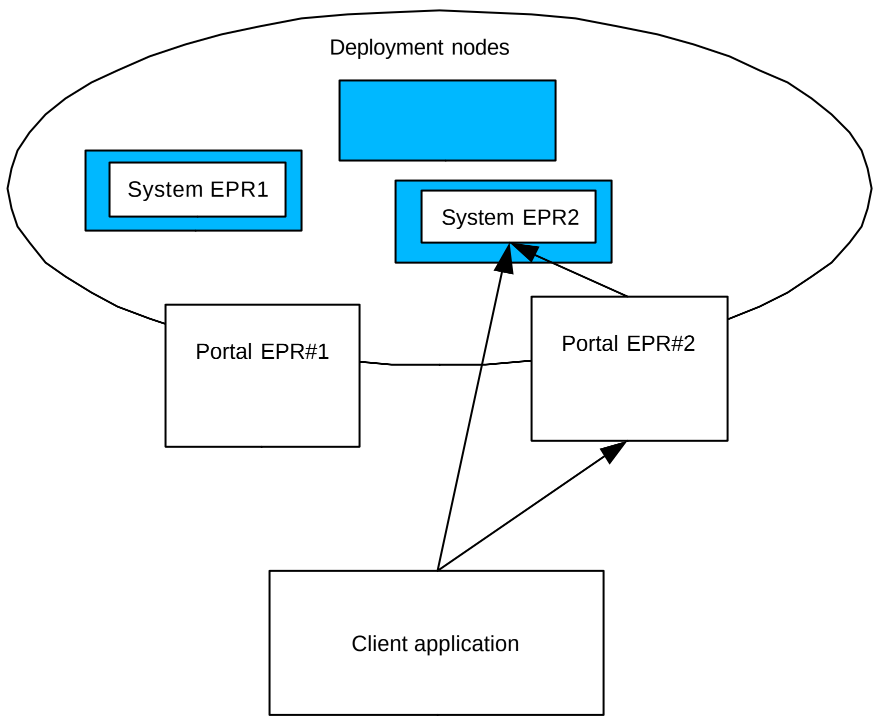


Figure 1 . Model of deployment and EPRs. Multiple Portal EPRs can manage the same set of deployment nodes.

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 4.2, and illustrated in figure 2.

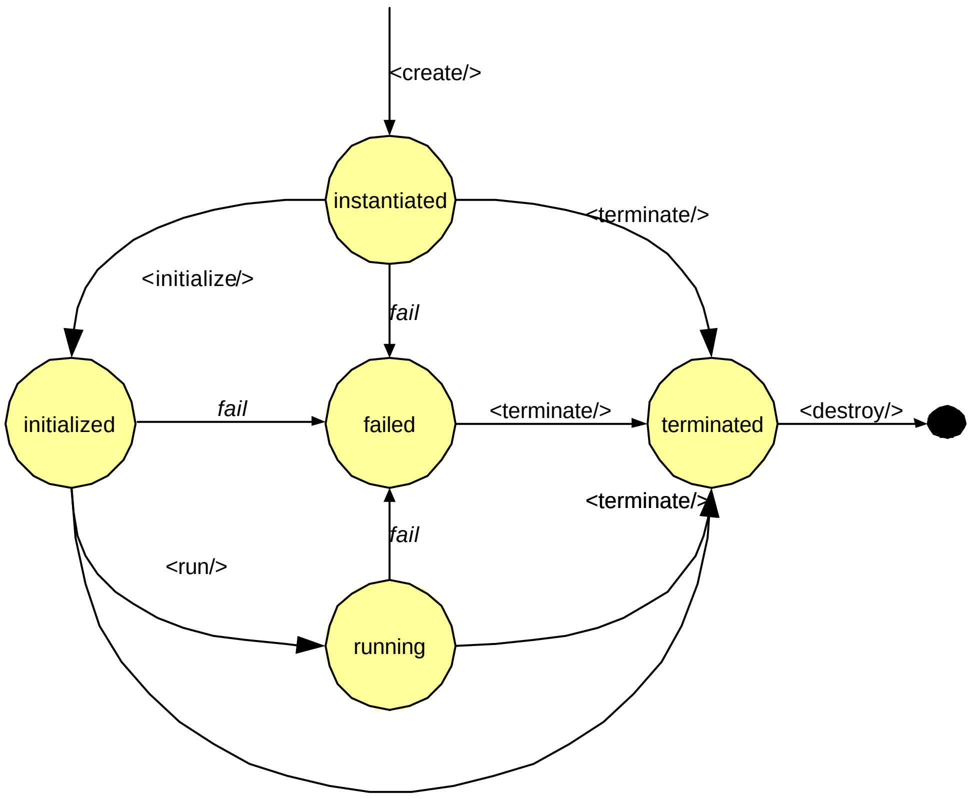


Figure 2 . The lifecycle of a deployed application

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-Resource Properties [WS-ResourceProperties] specification. There is also a synchronous, blocking call to probe the health of a 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 documents which can be retrieved from the server; again these are represented as properties following the WS-Resource Properties specification.

The portal EPR can raise events when new systems are created, using the WS-Notification protocol.

4.2 Lifecycle

CDDLM components have a uniform lifec ycle, 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. The main difference is the notion of a *destroyed* component. When a system is destroyed, all record of it is lost. A terminated system, may still have state that is remotely accessible.

The states of a 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.

Instantiation and initialization represent the creation and configuration of a component, and when it is moved into *running* then it is actually functional, 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.

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

4.3 Fault Tolerance

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

- 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 `<wsrl: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 EPRs through the use of a dynamic DNS service, one in which the DNS entries for the hostname(s) of the portal are updated as portal instances appear and disappear. Client systems should to be

written with the knowledge that the IP addresses of an EPR may change, and not to cache resolved IP addresses indefinitely.

4.4 Other Architectural Features.

4.4.1 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..z09_.]`. 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 itself take advantage of the fact, languages may choose to do so.

4.4.2 Deployment Language Agnostic

The deployment API is agnostic as to whic h particular language, or version thereof, is used for a deployment descriptor. When a remote deployment is created, the language and version of the descriptor must be supplied. The sole requirement of a language is that it can either be nested inside an XML document, or that a URL to the descriptor is remotely accessible to the destination. In the case of the latter, the URL to the descriptor must be provide when initializing the system.

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

4.4.3 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].

4.4.4 Deploy-time properties in the language and service API

Consider a deployment descriptor that wants to control onto which machine that it wants different components deployed onto. When the descriptor is written, the actual hosts 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 subsidiary 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 `java.System.Properties` hashtable to string and integer values. 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, 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 the system 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 list.

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.

4.4.5 Extensibility

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

4.4.5.1 Extra Operations

A service implementation may offer extra operations at any EPR. Such extensions must not add new declarations 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.

4.4.5.2 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.

4.4.5.3 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 `<initialize>` message. This (optional) element contains a list of zero or more deployment options. These are extra parameters to the deployment request. Every option is named with a URI, and can have a string or integer attribute value, or contain nested XML. A `mustUnderstand` attribute 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 consistency between different implementations. To manage this risk, the deployment API has the following requirements on optional metadata parameters:

- All options must be that: optional. It must not be an error to deploy a system with no options declared.
- 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 must contain either string, integer, Boolean or arbitrary XML values. String and integer values are supported via attributes; XML is supported as nested data.

- An option must contain only one value type. Implementations must raise a fault if multiple nested or attribute values are declared on the same option.
- All options that an implementation supports 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 a descriptor. Implementations must raise a fault when this occurs.
- Options may be processed in any order. Options must not require a specific order of processing.
- Servic e implementations must ignore any options that they do not recognize, if `mustUnderstand="false"` for that option.
- Service implementations must understand all options which are supplied with `mustUnderstand="true"` for that option. If any such option is not un derstood, a fault must be raised.

The processing rules for deployment are as follows:

1 Option processing must take place before the system is moved to the running state.

2 An implementation must be able to deploy a system when the entire options portion of the request is empty or omitted.

3 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 `extraData` field of the fault.

4 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. These must be ignored.

5 Duplicate options must cause the operation to be rejected with a `bad-argument` fault, identifying the particular option by its URI in the `extraData` field of the fault.

5 Deployment API Overview

The service API consists of two endpoint 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 doc ument, the following listed prefixes refer to the stated namespaces:

prefix	*URI*	*description*
`xsd`	`http://www.w3.org/2000/10/XMLSchema`	XML Schema Types
`wsa`	`http://schemas.xmlsoap.org/ws/2003/03/addressing`	WS-Addressing types
`api`	`http://www.gridforum.org/cddlm/serviceAPI/2004/10/11`	Deployment API
`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://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-BaseFaults-1.2-draft-01.xsd`	WS-BaseFault s
`wsrf-rl`	`http://www.ibm.com/xmlns/stdwip/web-services/WSResourceLifetime`	WS-Resource Framework
`wsrf-rp`	`http://www.ibm.com/xmlns/stdwip/web-services/WSResourceProperties`	WS Resource Properties
`wsrf-nt`	`http://www.ibm.com/xmlns/stdwip/web-services/WSBaseNotification`	WS-Notification
`wstop`	`http://www.ibm.com/xmlns/stdwip/web-services/WSTopics`	WS-Topics
`s12`	`http://www.w3.org/2003/05/soap-envelope`	SOAP1.2 Envelope
`xml`	`http://www.w3.org/XML/1998/namespace`	XML attributes

Unprefixed types in the document and accompanying schema are in the `api` namespace.

5.1 Portal Endpoint

The portal endpoint is the endpoint that the caller initially locates and communicates with. It can be used to create a new system within the set of nodes that it manages, it can be used to locate an existing system, and it can be used as a source of system creation events.

5.1.1 Portal EPR Properties

Name	*Type*	*Meaning*
`StaticPortalStatus`	`StaticPortalStatusType`	Static portal information; constant for the lifetime of the portal itself

Name	*Type*	*Meaning*
`DynamicPortalStatus`	`DynamicPortalStatusType`	Dynamic server information; may be different on every read
`DeployedSystems`	`SystemReferenceListType`	List of system EPRs
`Topics`	`wsrf-nt:TopicExpressionType`	List of topics
`FixedTopicSet`	`xsd:boolean`	flag to indicate whether topic set is fixed
`TopicExpressionDialects`	`xsd:anyURI`	Dialect of topicset

5.1.2 Portal EPR Operations

Name	*In*	*Out*
`Create`	`hostname: xsd: string`	`wsa:EPR`
	Create a system; hostname is optional	
`LookupSystem`	`xsd:string`	`wsa:EPR`
	Map from system name to a system EPR	
`wsrf-rp:GetResourceProperties`	`wsrf-rp: GetResourcePropertyRequest`	`wsrf-rp: GetResourcePropertyResponse`
	Get the value of a resource	
`wsrf-rp:GetMultipleResourceProperties`	`wsrf-rp: GetMultipleResourcePropertiesRequest`	`wsrf-rp: GetMultipleResourcePropertiesResponse`
	Read multiple resources	
`wsrf-nt:Subscribe`	`wsrf-nt:Subscribe`	`wsnt:SubscribeResponse`
	Subscribe to events	

If a portal has a managed lifetime, then it may also implement WS-ResourceLifetime properties and operations

5.2 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.

5.2.1 System EPR Properties

Name	*Type*	*Meaning*
`SystemName`	`xsd:string`	user-defined name

Name	*Type*	*Meaning*
`SystemIdentifier`	`xsd:anyUri`	unique identifier
`SystemState`	`cmp:LifecycleStateEnum`	current system state
`StateInfo`	`xsd:string`	Text state info
`SystemExtendedState`	`UnboundedXMLAnyNamespace`	Component state
`CreatedTime`	`xsd:dateTime`	Time system was created
`StartedTime`	`xsd:dateTime`	Time system was terminated
`TerminatedTime`	`xsd:dateTime`	end time (not present until system is terminated)
`TerminationRecord`	`TerminationRecordType`	termination record (present after termination)
`Topics`	`wsrf-nt:TopicExpressionType`	List of notification topics
`FixedTopicSet`	`xsd:boolean`	flag to indicate whether topic set is fixed
`TopicExpressionDialects`	`xsd:anyURI`	Dialect of topicset

5.2.2 System EPR Operations

Name	*In*	*Out*
`Initialize`	`job JobDescriptorType` `descriptor` `DeploymentDescriptorType`	`void`
	Initialize a system; pass in the job and component descriptors and build up the component graph.	
`addFile`	`mimetype xsd:string` `data xsd:base64Binary`	`xsd:anyURI`
	Add a file to this document so that it is accessible by a URI from within the deployment descriptor.	
`Run`	`void`	`void`
	Start running an initialized system	
`Ping`	`void`	`StatusType`
	Probe a system's health.	
`Resolve`	`xsd:string path`	`xsd:any`
	Resolve a reference relative to this system. Can return EPRs to	

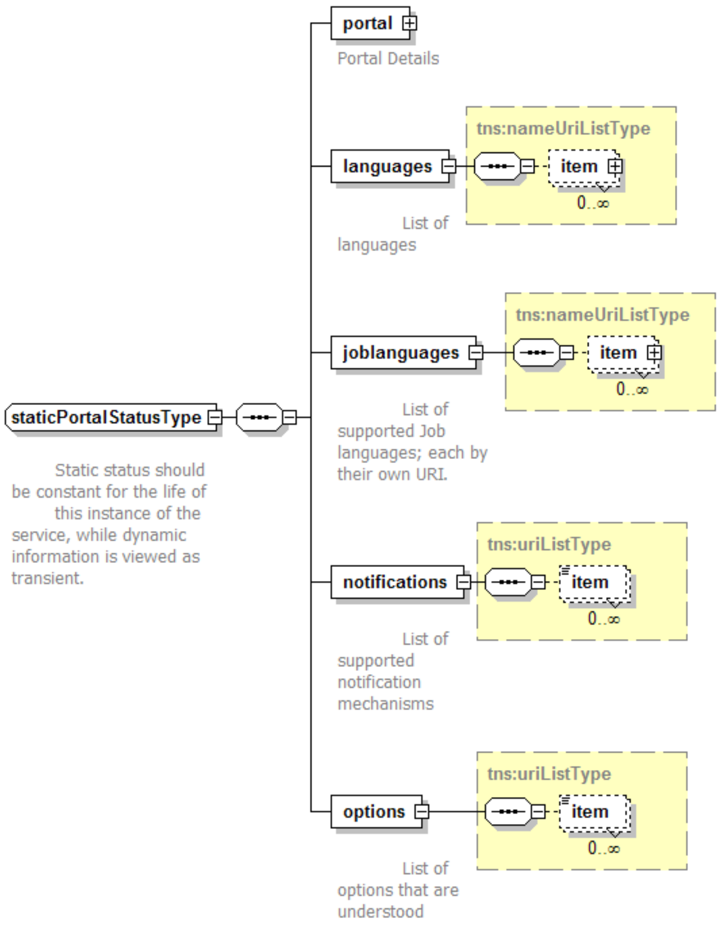
Name	*In*	*Out*
	components; string or other data	
`Terminate`	`xsd:string Message`	`void`
	Terminate a system; pass in a message	
`wsrf-rp:Destroy`		
	Destroy the System EPR, terminating the System if it is not yet terminated	
`wsrf-rp:GetResourceProperties`	`wsrf-rp:GetResourcePropertyRequest`	`wsrf-rp:GetResourcePropertyResponse`
	`Get the value of a resource`	
`wsrf-rp:GetMultipleResourceProperties`	`wsrf-rp:GetMultipleResourcePropertiesRequest`	`wsrf-rp:GetMultipleResourcePropertiesResponse`
	Read multiple resources	
`wsrf-nt:Subscribe`	`wsrf-nt:Subscribe`	`wsnt:SubscribeResponse`
	Subscribe to events	

6 Portal

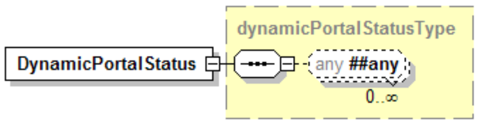
6.1 Portal Properties

6.1.1 StaticPortalStatus:

This property contains static portal information; information constant for the lifetime of the portal instance. The portal elements details contains static diagnostics information, such as product name and timezone portal. The information lists are all lists of URIs that can be used to determine features.


portal
Portal Details
tns:nameUriListType
languages
item
0..∞
List of languages
tns:nameUriListType
joblanguages
item
0..∞
List of supported Job languages; each by their own URI.
staticPortalStatusType
Static status should be constant for the life of this instance of the service, while dynamic information is viewed as transient.
tns:uriListType
notifications
item
0..∞
List of supported notification mechanisms
tns:uriListType
options
item
0..∞
List of options that are understood

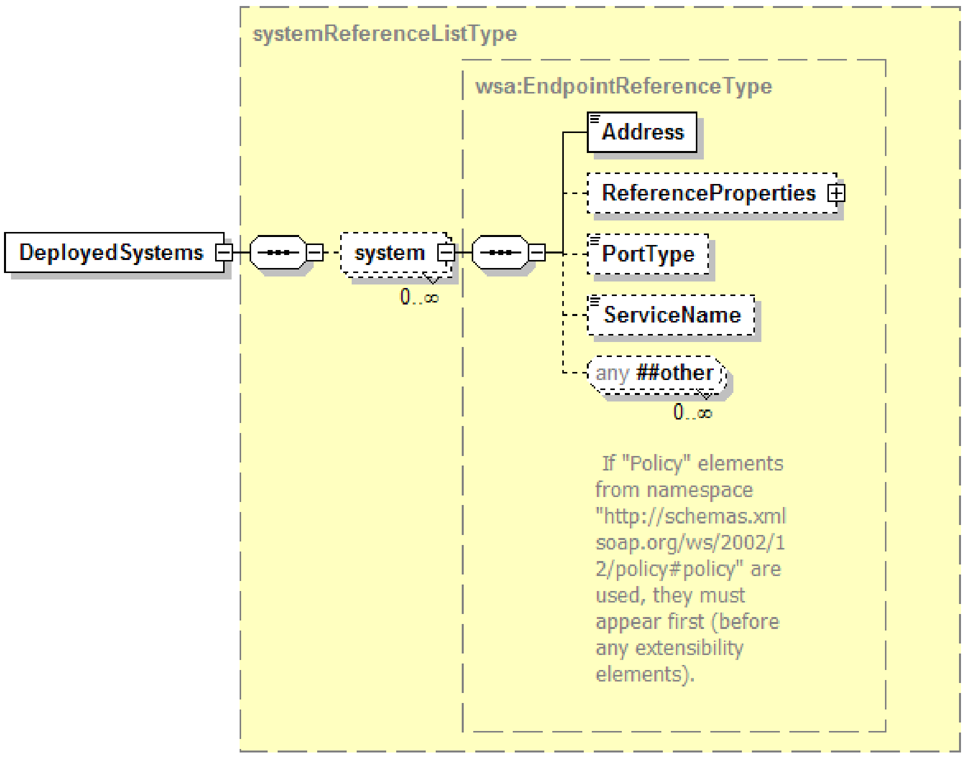
6.1.2 DynamicPortalStatus



This is any dynamic status information.

6.1.3 DeployedSystems

This is a list of deployed systems which the Portal is aware of. This may include systems in the portal which the portal did not deploy, but which a peer portal deployed. It may also be restricted to those systems to which the caller has access rights. Network partitioning and other events may cause systems to be temporarily invisible to this list, and return later.


systemReferenceListType
wsa:EndpointReferenceType
Address
ReferenceProperties
DeployedSystems
system
0..∞
PortType
ServiceName
any ##other
0..∞
If "Policy" elements from namespace "http://schemas.xmlsoap.org/ws/2002/12/policy#policy" are used, they must appear first (before any extensibility elements).

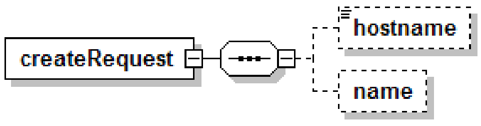
6.2 Operations

6.2.1 Portal::Create(hostname)

This requests the portal implementation to create a new system, ready for deployment.

The hostname element specifies an optional hostname. If set, it nominates a host onto which the port should instantiate the System, and hence the system EPR. If unset, or if the identified host is deemed unsuitable/unavailable, the portal can instantiate the system on a host of its choosing. Thus the hostname is merely a hint, a hint to improve availability and performance.

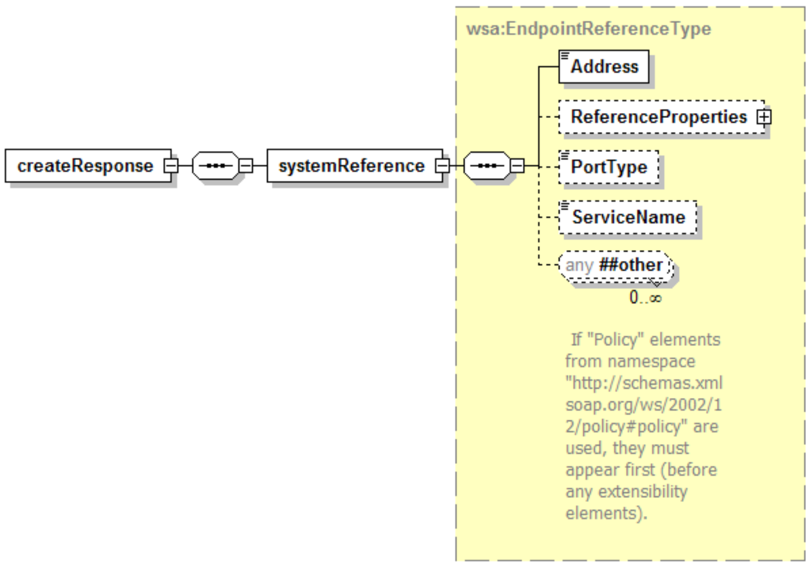
The name is an optional name of the system. One will be generated if none is supplied.



System names are constrained strings:

```
    <xsd:simpleType name="systemNameType">
        <xsd:annotation>
            <xsd:documentation>
        This is the policy for the naming of systems
      </xsd:documentation>
        </xsd:annotation>
        <xsd:restriction base="xsd:NCName">
            <xsd:pattern value="[a-zA-Z_\-\.][a-zA-Z_\-\.\P{Nd}]*"/>
        </xsd:restriction>
    </xsd:simpleType>
```

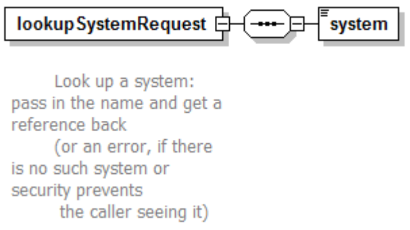
The response is an EPR to the instantiated system, an EPR which can be immediately used for direct communications. Creation of a system EPR is therefore a synchronous operation.

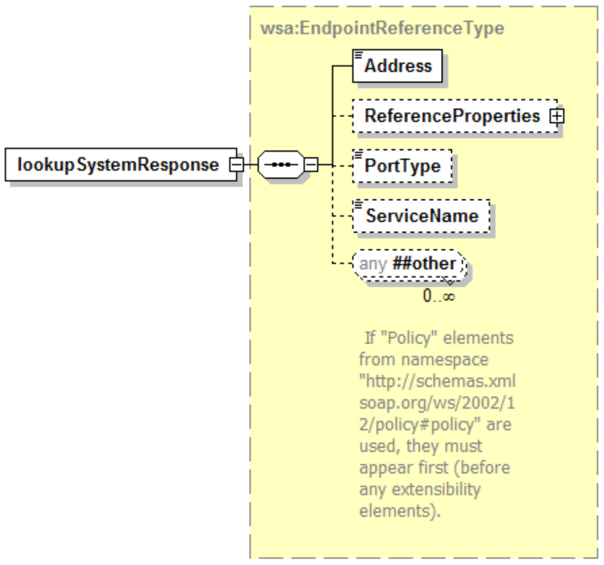

wsa:EndpointReferenceType
createResponse
systemReference
Address
ReferenceProperties
PortType
ServiceName
any ##other
0..∞
If "Policy" elements from namespace "http://schemas.xmlsoap.org/ws/2002/12/policy#policy" are used, they must appear first (before any extensibility elements).

If an entity is registered with the portal for creation events, then the portal must send notification to that entity that new system has been created. The notification must not be sent until the system is ready for direct communication. There is no specification of the ordering of returning from the `create` operation and the sending of any notification mechanism. If there are multiple portals supporting deployment to a cluster of nodes, notification events *may* be sent to listeners on one portal, even if the deployment was requested on the other.

6.2.2 LookupSystem(name)

This maps from an system name to an EPR (or a fault)

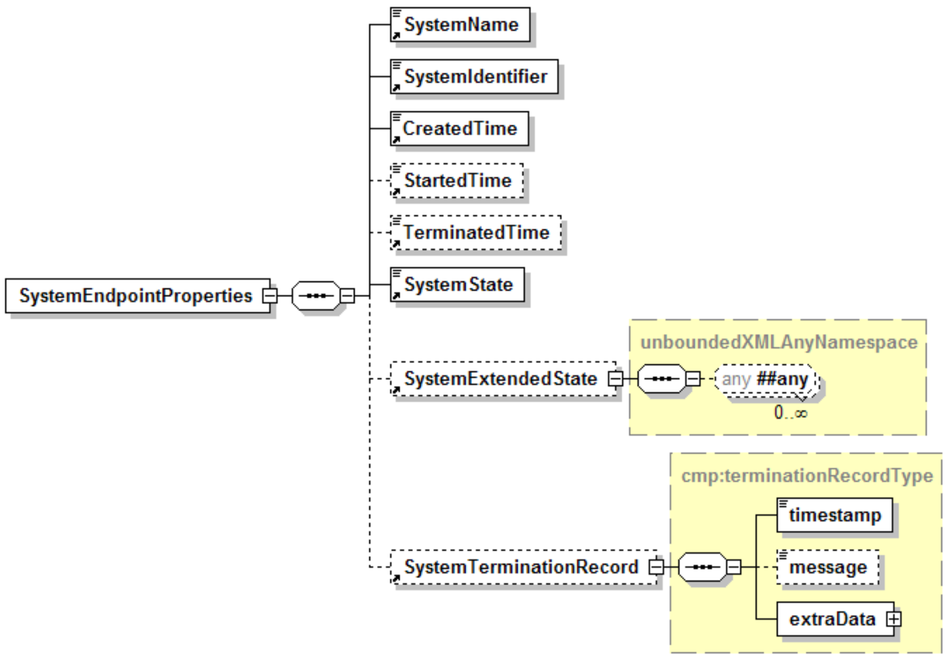

lookupSystemRequest
system
Look up a system:
pass in the name and get a reference back
(or an error, if there is no such system or security prevents
the caller seeing it)


wsa:EndpointReferenceType
Address
ReferenceProperties
lookupSystemResponse
PortType
ServiceName
any ##other
0..∞
If "Policy" elements from namespace "http://schemas.xmlsoap.org/ws/2002/12/policy#policy" are used, they must appear first (before any extensibility elements).

7 System

The System EPR represents the deployed system. After creation, it is still undefined, and must be configured before it can be moved to a running state.

7.1 System Properties



7.1.1 Syst emName

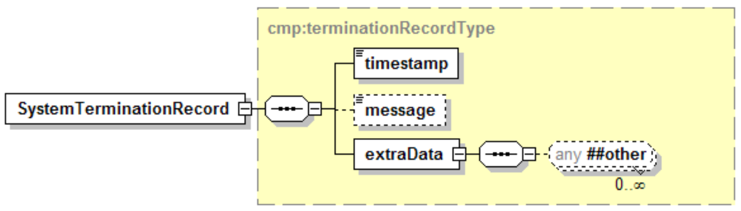
This is the name of the system.

7.1.2 CreatedTime/StartedTime/TerminatedTime

These are all `xsd:dateTime` timestamps of when a system entered a particular state.

7.1.3 SystemTerminationRecord

This contains a `cmp:` type, `terminationRecordType`



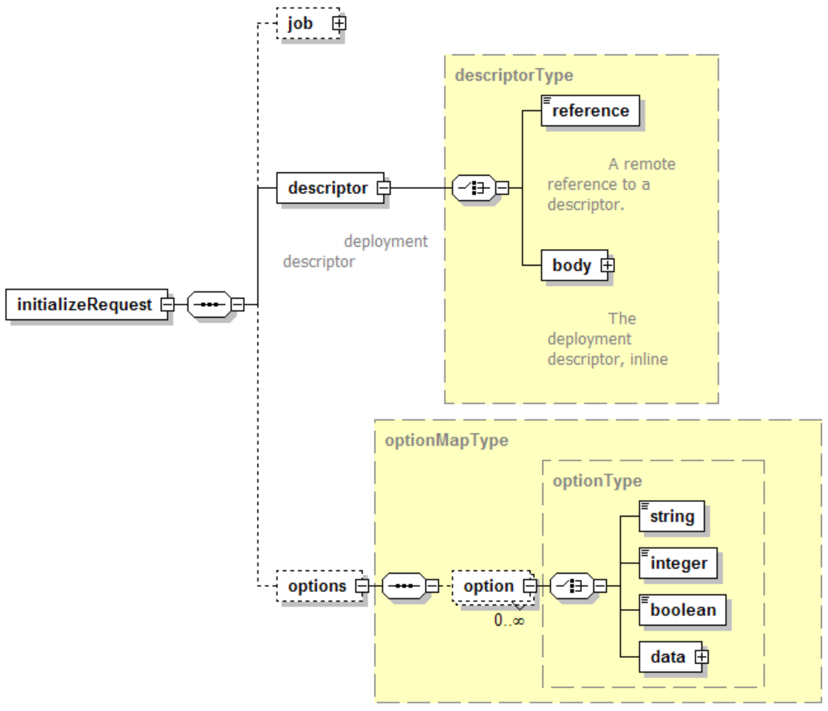
It contains information about the reason for the system's termination. It is only present after a system has been terminated.

7.2 System Operations

7.2.1 System::Initialize

This is a complex request, as it configures the system and moves it into the *initialized* state.

A deployment descriptor must be supplied; it consists of a language URI, and either an inline deployment descriptor or a URL to a location where the descriptor can be located.

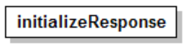


The optional `<jsdl>` element contains the job description that was used when submitting the job to the front-end portal. As with the `<descriptor>`, it is of type `descriptorType`; it must have a language URI and either an inline body or a URL to the descriptor. The interpretation of this data by the service implementation is undefined.

The optional `<options>` element contains a list of zero or more configuration options. These are late-binding parameters to the deployment request, or to the deployment runtime.

When the request message is received, the system EPR must validate it (synchronously) and initialize the system. For CDDLM implementations, initialization implies that the and deployment descriptor and JSDL descriptor may be retrieved (if needed) and parsed. The application is configured, entering the *initialized* state. This can be a time consuming process, so must be an asynchronous operation.

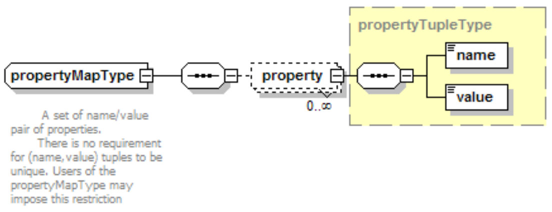
The response to a successful request is an empty response, `<initializeReponse>`:



It's presence implies that the initial validation was successful, and that initialization has begun, or has at least been scheduled.

7.2.1.1 *The propertyMap schema type*

To aid those options that take a map of name/value pairs, there is a predefined XML Schema type that can represent the construct:

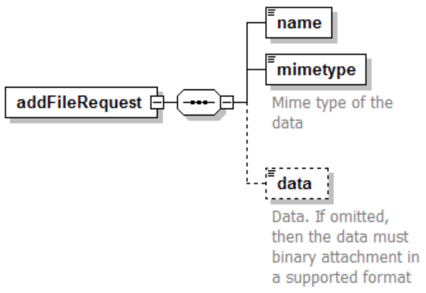

propertyTupleType
propertyMapType
property
0..∞
name
value
A set of name/value pair of properties.
There is no requirement for (name,value) tuples to be unique. Users of the propertyMapType may impose this restriction

`propertyMap` elements can be placed into the `<xml>` child element of an option. Both the name and value of a `propertyTuple` within a `propertyMap` element are of type `xsd:string`; individual options are free to declare extra restrictions on the value of properties, restrictions which can be validated when processing the option.

There is no requirement that the name/value pairs are unique within a `propertyMap` element; that is also a restriction that can be declared in a specification of a particular option.

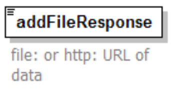
7.2.2 System::addFile(file)

This request uploads a file to the infrastructure, such that it is visible by deployed programs, and by the System EPR itself.


name
mimetype
Mime type of the data
addFileRequest
data
Data. If omitted, then the data must binary attachment in a supported format

The response returns a URI to the uploaded file, a URI either of type file: or http:

Comment [slo1]: Do clients want the right to ask for a particular type? They may get turned down, but sometimes you may want file: URLsfor legacy apps.



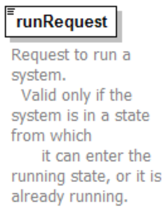
The file must be visible to programs deployed by this descriptor. They may be visible to other programs running with the same credentials, but this can not be guarantees. If exposed as a file: URL, the file should be read-only.

The lifespan of the uploaded file is bound to that of the created system; when the System EPR is destroyed, all uploaded files are destroyed.

There is no guarantee of high-availability in deployment; failure of a single node may render the URL unreachable.

7.2.3 System::Run

This request runs a system. This triggers an asynchronous action, as it may take some time to enter the running state. It is only valid from a state in which the lifecycle permits running to be reached; *initialized* and, implicitly, *running*. In the case of the latter, the operation is a no-op. If the system is initializing itself, as a result of an `Initialize` request, the request should be queued for processing after the state transition is completed.


runRequest
Request to run a system.
Valid only if the system is in a state from which
it can enter the running state, or it is already running.

The response is an empty element:

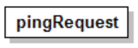
runResponse

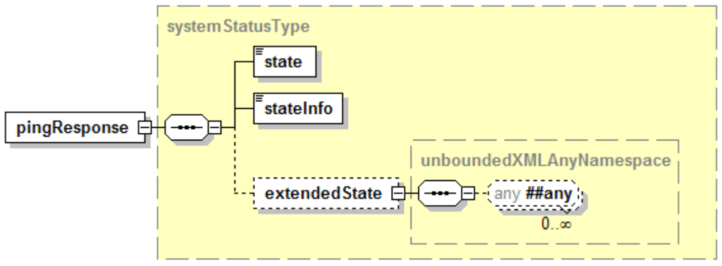
A response means that the system has been queued to enter the running state asynchronously, or that it now is in that state.

7.2.4 System::Ping

This is a synchronous request to the system, to query its health.

Comment [slo2]: May want to move types to cmp:



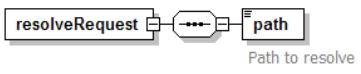


If the system is not running, the System EPR must return with the current state. If the system is running, the request must be forwarded to the application, which can return any extended state information.

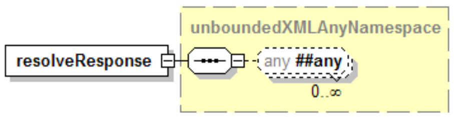
This effectively acts as a liveness test upon the application.

7.2.5 System::Resolve

This operation resolves a path and returns its value or an error. It must be a valid operation when a system is initialized or running. It may be valid in a failed or terminated system.

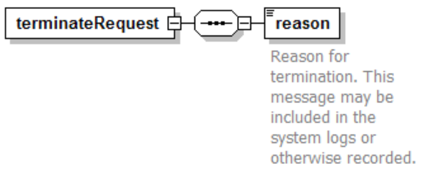


The response is arbitrary XML data, the contents of which depend upon what the path resolved to.



7.2.6 System::Terminate

This request terminates the system. To be idempotent, this call does not raise a fault when the system is already terminated.


terminateRequest
reason
Reason for termination. This message may be included in the system logs or otherwise recorded.

Upon receipt, system termination should commence. Termination is asynchronous.

The response is an empty element.

7.2.7 <wsrf-rp:Destroy/>

The `<wsrf-rp:Destroy/>` operation destroys the System EPR itself. All files uploaded are destroyed, and the system is terminated if it is not already terminated.

After sending this message and receiving a response, service consumers should not make calls of the EPR, as it may not be valid.

Implementations may continue to export System EPR valid until the system is terminated. If this is the case, receipt of a multiple Destroy request should not be an error. However, receipt of all other requests on the endpoint from external callers may be treated as faulting.

Comment [slo3]: Not sure about this behaviour.

8 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.

8.1 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.

8.2 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] defined:

- Portal EPRs must support a WS-TopicSpace that contains one topic: system creation events. This notifies callers that a new system has been created.
- System EPRs must support a WS-TopicSpace that contains one topic: lifecycle events. This notifies callers of changes in a system's lifecycle state.

8.3 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.

9 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.

9.1 Fault Categories

9.1.1 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-BaseFault specification. Again, the implementation must not add new types to the deployment API namespace.

9.1.2 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.

9.1.3 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 that fill the same role.

9.2 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.

9.3 Internationalization

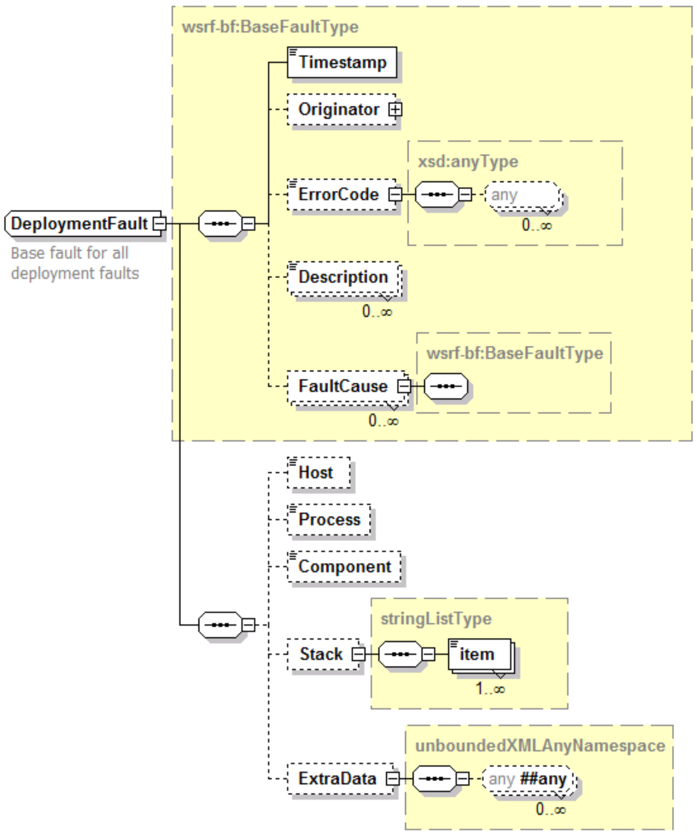
The WS-BaseFault specification makes no statement upon which language error descriptions are in.

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.

9.4 Faults

9.4.1 DeploymentFault

This type represents any fault thrown by the deployment infrastructure. All endpoint operations must declare that they throw this fault, and must not explicitly declare any derivative faults that they may throw.

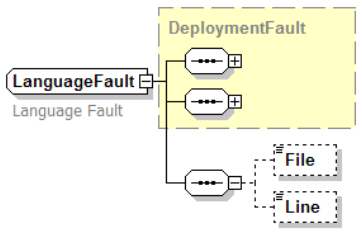


Element	*Type*	*Meaning*
`Host`	`xsd:string`	Hostname or pseudonym
`Process`	`xsd:string`	Any process identifier suitable for diagnostics
`ExtraData`	`unboundedXMLAnyNamespace`	Extra fault data
`Component`	`xsd:string`	Path to component raising the fault
`Stack`	`stringListType`	Optional stack trace

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.

9.4.2 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

If the error is in the inline deployment descriptor, the `File` element must be empty "" or omitted. 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/absent file element that the fault was in the inline request.

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

9.4.3 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*
`SoapFault`	`s12:Fault`	Fault code information

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

SOAP1.2	*WrappedSOAPFault*
`/s12:Fault`	`WrappedSOAPFault/api:SoapFault`
`SOAP endpoint`	`WrappedSOAPFault/wsrf-bf:originator`

The SOAP endpoint must be translated into a `wsa:EndpointReference` if it is a simple URL/SOAPAction tuple.

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`.

9.5 Fault Error Codes

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

10 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.

Files uploaded via `System::addFile` must only be visible to the deployed application, and potentially other applications deployed under the same credentials.

11 Editor Information

Steve Loughran, HP Laboratories

steve_loughran@hpl.hp.com

12 References

[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.