A service API for Deployment

Article

1 author:

Steve Loughran
Apache Software Foundation
20 PUBLICATIONS **220** CITATIONS

SEE PROFILE

A service API for Deployment

Steve Loughran

steve_loughran@hpl.hp.com
October 18, 2004

Abstract

This paper discusses a prototype SOAP API for deploying software described in configuration languages -languages such as XML-CDL and SmartFrog.

The endpoint supports only a few operations: *deploy*, *undeploy*, *serverStatus*, *applicationStatus*, *listApplications, setCallback* and *lookup*. The key operation is *deploy*; the message used to trigger this operation is designed to be extensible, deployment language independent and easy to generate by a front-end application or portal with which the user can interact. Key to this are support of a JSDL job descriptor, arbitrary XML for the deployment descriptor, and a mechanism for adding optional XML fragments to the message, replicating aspects of the SOAP Header design.

The service makes minimal use of WS-RF facilities. The rationale for not doing so was to make the service broadly available to all callers. We will recommend improvements to the WS-BaseFault fault proposal.

Finally, we will explore implementation details: how to effectively implement the API, and how to deploy and test it using the underlying deployment framework. As testing distributed systems is always a challenge, it is interesting to explore how a deployment framework can improve the test process.

Introduction

The CDDLM working group has been addressing the problem of deployment, the act of instantiating running code on remote systems. For this purpose, we have been both standardising an existing language, SmartFrog [smartfrog], and developing an XML descendant (XML-CDL) [xmlcdl]. These languages describe a deployment as a declarative graph of *components*, each of which knows how to configure, start and stop an individual part of the greater system. How these components interact is being defined in the ongoing Component Model, a design which will use WS-RF to couple the component instances [wsrf]. The design of the component model is still ongoing, as it is a complex problem.

For deployment to be possible, one must not only have a representation of a system, and a runtime to parse and instantiate that representation, one needs a way of sending that representation to the deployment runtime. This is the role of a deployment service API.

This paper describes the initial service API for CDDLM deployment. It is very much an exploratory design, to see what works and what does not in terms of a public API for CDDLM-managed deployment.

Design Goals

The deployment API is intended to be broadly accessible to anything that wishes to to deploy. We expect a portal application using JSDL submissions [jsdl] to be the front end for

job submissions. This portal would retrieve the code and data files of the deployment, and -after negotiating with the resource management framework- send a deployment request to the CDDLM runtime. For the lifespan of the deployment, this front end would be expected to be sent notifications of changes in deployment status, be able to poll for the current status of the deployment, and to terminate the deployed application.

1. Provide a simple SOAP1.1 service endpoint for deploying, undeploying, and pinging applications.
2. Be agnostic as to the language of the deployment.
3. Offer a notification mechanism for lifecycle events.
4. Be flexible and extensible by implementations.

Service API

There is a single endpoint for the deployment service[1]. It offers seven operations.

Operation	*meaning*
`deploy`	Deploy an application described by the nested descriptor
`undeploy`	Undeploy an application
`setCallback`	Set or clear the lifecycle event callback for an application
`applicationStatus`	Check the current status of an application
`listApplications`	List all running applications
`serverStatus`	Get information about the server
`lookupApplication`	Map from an application name to a URI

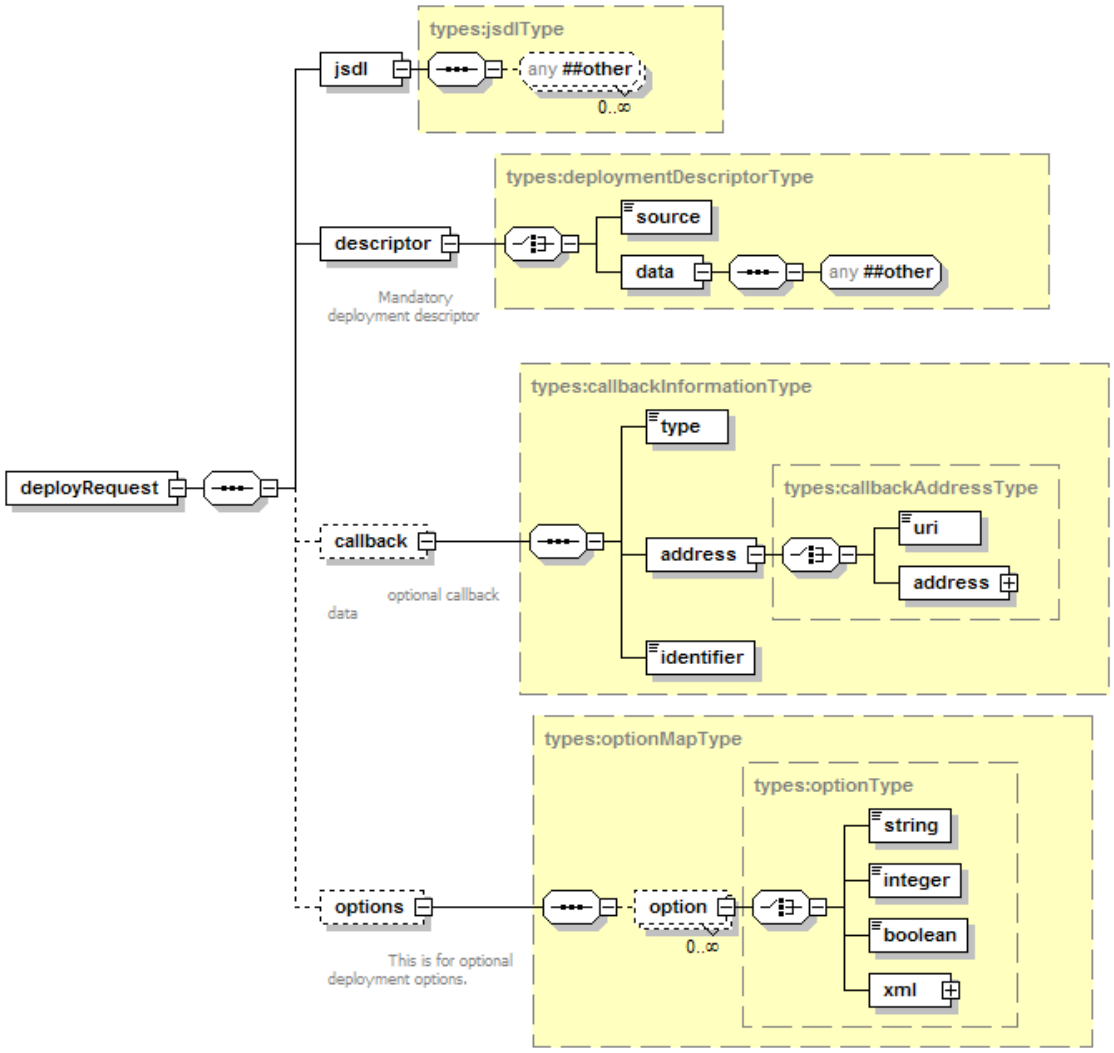
Deployed applications are identified by URIs. These are only guaranteed to be unique to a service instance, and do not have to resolve to any valid URL.

deploy

Deployment is the whole rationale for this service, so it is unsurprising that this is the most complex request in XML terms. It has four main features:

- A JSDL job descriptor [jsdl]
- A deployment descriptor. The namespace of this descriptor is used to determine the language of the deployment. That is, there is no explicit declaration of the language, it is implicit from the XML itself.
- The optional declaration of a callback to receive lifecycle events. This is the exact same information as in a `setCallback` request, except it is attached to the deployed application from the outset.
- A list of options.

1 This does not preclude a WS-RF based design for direct access to individual components.



The option list is a very powerful aspect of the deployment 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.

1. All options must be that: optional. A service implementation must be able to deploy an application when the entire options/ portion of the request is empty.
2. Every option configures a deployment feature of the remote system.
3. Every option is named by a URI
4. All URIs which begin `http://gridforum.org/cddlm/` are reserved for the CDDLM standards.
5. Options can contain string, integer, boolean or arbitrary XML values.
6. All options that a service implementation supports must be enumerated in `serverStatus` requests.

undeploy

This operation undeploys a named application. To be idempotent, this call does not raise a fault when an application is already terminated, or when the application is undefined. That is because there may be no record of the now-terminated application.

setCallback

This operation sets a lifecycle event callback for an application, or clears an existing application. Callbacks will be covered in more detail below.

applicationStatus

This operation probes the health of an application. The response -for a healthy application- is a simple message containing status information:-

```
<applicationStatusResponse >
 <reference>http://localhost/cddlm/1095458549045/job1</reference>
 <name>job1</name>
 <state>running</state>
 <callback>
  <type>cddlm-prototype</type>
  <address>
   <uri>http://localhost:5051/client/callbacks</uri>
  </address>
  <identifier>1564</identifier>
 </callback>
 <stateInfo xsi:nil="true"/>
</applicationStatusResponse>
```

The operation probes the health of the root component of the deployment hierarchy, which return build up a model of the health of subsidiary components. The draft component model [component-model] specification covers this. The call blocks until the health check is complete.

ServerStatus

This operation returns the status of the server. The status is composed of a static section and a dynamic section. The static section includes server build information,

```
<serverStatusResponse
xmlns="http://gridforum.org/cddlm/serviceAPI/types/2004/07/30">
 <static>
  <server>
   <name>SmartFrog implementation</name>
   <build>$Id: Constants.java,v 1.12 2004/09/15 12:59:13 steve_l Exp $</build>
   <home>http://smartfrog.org/</home>
   <location>unknown</location>
   <timezoneUTCOffset>0</timezoneUTCOffset>
  </server>
  <languages>
   <language>
    <name>SmartFrog</name>
    <version>1.0</version>
    <namespace>http://gridforge.org/cddlm/smartfrog/2004/07/30</namespace>
   </language>
   <language>
    <name>XML-CDL</name>
    <version>0.1</version>
    <namespace>http://gridforge.org/cddlm/xml/2004/07/30/</namespace>
```

```
   </language>
  </languages>
  <callbacks>
   <item>cddlm-prototype</item>
  </callbacks>
  <options>
  <item>http://gridforum.org/cddlm/serviceAPI/options/name/2004/07/30</item>
  <item>http://gridforum.org/cddlm/serviceAPI/options/propertyMap/2004/07/30
  </item>
  <item>http://gridforum.org/cddlm/serviceAPI/options/validateOnly/2004/07/30
  </item>
  </options>
 </static>
 <dynamic/>
</serverStatusResponse>
```

In this example, the server lists support for two languages, one prototype option and three deployment options; there is no dynamic status

listApplications

This operation returns a list of URIs of deployed application.

lookupApplication

This operation maps from a predefined application name to the current URI of an application deployed under that name. This is a vestigial operation that is likely to be removed in future.

Design Issues

Language Support

The protocol is independent of any particular configuration language; the sole requirement is that it must be embeddable inside the `xsd:any` section of the deployment descriptor, and that the namespace qualified name of the language must be unique enough to distinguish the language.

For the XML-CDL language, these attributes are implicit. To deploy SmartFrog content, we defined a minimal XML encoding, in which the entire document is encoded as text inside a suitably qualified element.

One interesting exercise would be to implement Apache Ant support [ant]. This may pose a challenge, as the core language is namespace-free XML; namespace support is a recent edition primarily to address task declaration clashes. As with SmartFrog, we may need to provide a wrapper element that declares an appropriate default namespace.

The alternate strategy for multi-language support would be to require explicit declaration of the language, and potentially the version of that language. This was not adopted as we felt that the XML namespace should provide all such information implicitly. It would be relatively easy to migrate to an explicit language declaration model. It may also improve readability for people viewing the wire format of the messages.

Options

The option mechanism offered in the deploy request provides a powerful, structured means of customising the deployment, either with specific options offered by a single

implementation of the service API, or through widely-recognised options formally defined in group specifications.

As stated above, having optional aspects to a protocol with multiple implementations is potentially dangerous. If a program deploys expecting a particular option being available, and that option turns out to be absent, then things will not work as expected. The `mustUnderstand` attribute of each option lets the caller mandate which options must be understood, and the ability to enumerate available options in a `serverStatus` call provides the ability to enumerate the set of supported options without even attempting to deploy an application. These features , and the requirement that an option-free deployment must be supported reduces the risk of the option feature.

The next way to reduce risk is to standardise the names and semantics of common options. The ones we propose are

- `validateOnly`: a flag to indicate that the deployment descriptor should be validated to the best of the ability of the runtime, without being actually deployed.
- `name`: this is covered below; it specifies an explicit name for the application within the runtime, which can be entirely unrelated to the URI used in the service API.
- `properties`: a set of name-value pairs to define properties for examination by the deployment descriptor. These properties can be used for deployment-time customisation of a descriptor, without rewriting the descriptor itself. For example, properties could define the final locations of uploaded code files, mapped into the file system of the target machine.
- `environment`: A set of name-value pairs defining the environment variables of the deployed program. This is subtly different from the `properties` option, as the environment is part of the operating system or shell, and so effects the system more dramatically. The JSDL specification includes an environment block; this would be a mapping of the same block into the options section of the deployment request.

Naming and identification

The original design of the service API had included a mandatory name attribute when deploying an application. This was derived from experience in SmartFrog, where the name of the deployed component graph could be used for cross-referencing components in parallel deployments. To fulfil such a role, the name had to be unique across all currently deployed applications.

It soon became clear -during testing- that this requirement was misguided, as it required the service caller to know that there was no other application with the same name already deployed. The service API was updated to remove the requirement to name any deployment, although it is retained as an option. That is, a deployment request can include a name option in its option map, and indicate whether the runtime is expected to understand that option or not.

One vestigial aspect of the original design is that we retain an operation `lookup(name):URI`, which looks up an application by name. With unique names still being an option, there is some value in retaining this operation. Yet there are other ways of binding running components together, and naming seems both crude and brittle.

We propose that the future model should be:

1. Every deployment is given a URI reference by the runtime
2. These URIs are only guaranteed to be unique on a single runtime, not across machines.
3. URIs should be unique across restarts of the runtime. That is, a single host should never deploy two applications which hold the same URI.
4. To locate applications or components in an application by well known names, other discovery and binding mechanisms should be used, such as UDDI, or multicast DNS.

The prototype achieves requirements (1,2,3) by creating a new URI of the form `http://localhost/${time}/job/${counter}` where `${time}` is the startup time of the runtime, and `${counter}` is a counter that is incremented for every URI generated.

Callbacks

Callbacks are an outstanding issue with Web Services. Although an integral part of distributed systems such as CORBA, DCOM and RMI, they are currently absent from the SOAP stack. The reason for this was that SOAP, being an evolution of XML-RPC messages sent to web sites via HTTP POST requests, was designed with the assumption of a firewall between the client and the (HTTP) server).

There is still no widely supported standard for callbacks, though WS-Eventing and WS-Notification are emerging as the two competing options -perhaps even two converging options. Both standards are still unstable, with recent update to WS-Eventing narrowing the difference between the two [wsevt]. It is also worth noting that the Jabber XMPP protocol has been used with some success in XML messaging projects, because of its ability to tunnel through firewalls via open port 80 connections to public jabber servers [xmpp].

With all the many emerging callback options, and with no one ready for use, the deployment API opted to support multiple callback possibilities.

1. The server status request includes a list of which callback mechanisms a server supports.
2. The deployment request includes an identification of the desired callback mechanism, and any XML data related to that specific callback option.
3. If the client wants to receive lifecycle events it must probe the server for a supported callback option and use that at deployment time.
4. For the prototype, we added a callback option, that of direct dispatch of a WSDL described lifecycle event message to a nominated HTTP URL.

This is too much flexibility to be useful, as clients do not have any guarantee that their supported callback mechanism will be supported in the server. A client would need to support all well-known callback options to be sure of receiving notifications.

The other callback issue is that one needs a means of subscribing to an existing deployment, to receive notifications, or to unsubscribe a callback registration. WS-Notification covers such options, if we expose our deployments as WS-Resource entities. As this an intended goal of the component model, WS-Notification is likely to become integral to the runtimes, and so supported by all deployment servers. If we were to mandate that this was required, then clients would only need to implement a single callback mechanism to receive

notifications.

Over time, WS-Notification will itself evolve, and we will be left with the problem of which version of WS-Notification to support. It may be prudent to retain the enumeration of callback types, and the ability to select a specific callback protocol, in order to adapt to this future need.

A further callback-related issue is how to support deployment from systems with no callback mechanism at all. One option that was added to the deployment request is the ability to request a blocking deploy, so that any deployment failure was detected and returned to the caller. This is only viable for deployments that start up immediately -anything with a slow startup time would start to trigger timeouts over the network.

Faults

We looked long and hard at the WS-Faults component of the WS-RF family of specifications. The fact that it has a formal model for nested faults does appeal, as it makes it possible to nest faults inside other faults in a structured manner.

However, one problem with the WS-Faults mechanism is that it seems to require the endpoint to declare every fault -and every fault element/attribute. To be so explicit in one's fault declaration seems to be opposed to the notion of future extensibility.

We need to experiment further with WS-RF to see if this is really the case, or it has been misunderstood.

For now, we are using standard SOAP faults. A normative XML file accompanies the XSD and WSDL descriptions of the service, to provide URIs, names and descriptions of every standard fault in the service API.

When compared to WS-Faults, this approach is over-flexible. An Axis hosted implementation can add arbitrary fields (stack trace, hostname), but other SOAP runtimes do not. This inconsistency in return data is a danger all of its own -it is too easy to code against one implementation, on one runtime, and so fail to correctly handle faults raised by other implementations, or even the same implementation on a different JAX-RPC-compliant runtime.

If we remain with "classic" SOAP faults, we must require that any of the service API faults must include our own typed elements inside for structured analysis of the contents. This would imply defining a `DeploymentFault` that extended the WS-BaseFault, and yet which included all the dynamic extensibility we desire.

Future Work

Security

The current prototype does not have any security; it grants unlimited rights to callers. There is only the option of using HTTP Basic Authentication for minimal security of the channel. Clearly this is inadequate.

Logging

We have neglected the entire process of capturing output from the application. A deployment

descriptor is free to configure its own logging, but there is no integration of that with the callback mechanism.

A first step would be to capture all output to files and provide a means of accessing the output remotely -even from running applications. Presumably the information would also need to be retained for some time after the application is terminated, so that it could all be retrieved.

JSDL Integration

Deployment requests take a JSDL descriptor, inside the request.

The prototype does not incorporate any of the information contained in this descriptor into its deployment. There are a number of ways in which the information in the JSDL descriptor could be used

1. The `JobName` and `JobAnnotation` text could be extracted for job information.
2. The environment variables can be extracted and used to set the environment for execution. This could also be done by a front end application, assuming that the service supported the proposed environment option.
3. Any explicit limits on jobs (such as `JobVirtualMemoryLimit` or `CPUTimeLimit`) should be used to place constraints on job execution.
4. The `stdIn` and `stdOut` elements could be mapped to data files.

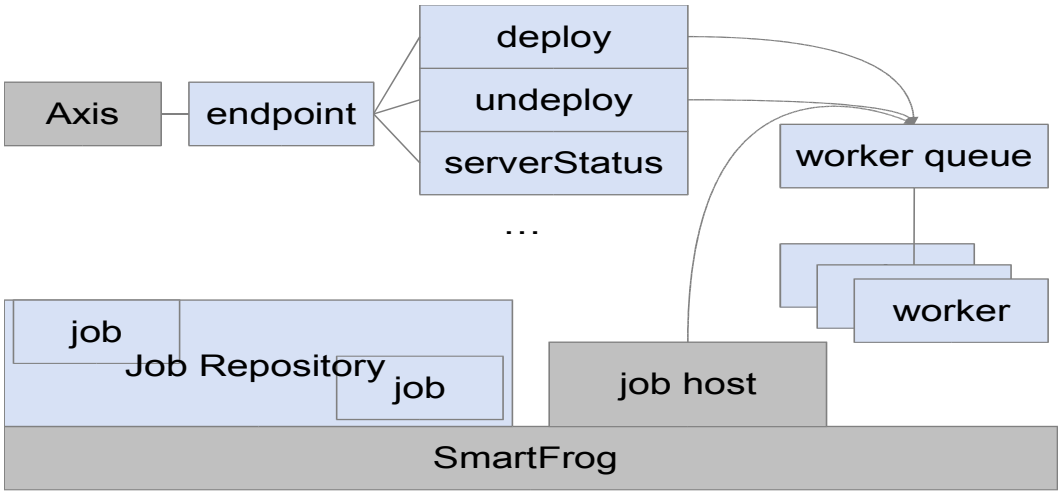
Overall, the JSDL specification is best-suited towards the queuing and execution of batch jobs. Adapting it to a model in which a 'job' is now a deployment of a complex set of interconnected sub-applications is possible, but it will take effort particularly if those applications are distributed across multiple machines.

Integration between a JSDL-centric front-end would be a good approach towards integrating with JSDL and to providing an end-user friendly front end to the deployment framework itself. The WS-JDML system is designed to support custom back ends; CDDLM support would be part of another such module [wsjdml]. The portal would be able to take on the task of retrieving the submitted files and creating a deployment descriptor combining the user's own submission with extra deployment information.

Implementation

Server

We have an initial implementation, written in Java and built on top of Apache Axis 1.2beta and the SmartFrog 3.0.04 runtime [axis]. Axis and the SOAP service that it hosts are actually deployed within the SmartFrog JVM; we wrote a SmartFrog component to configure and deploy Axis, then extended this with our CDDLM deployment service.



The endpoint retains its own repository of deployed jobs, each job instance storing all the extracted information from a deployment request -JSDL document, the deployment descriptor- and all the binding information to a deployed application in the SmartFrog runtime.

Slow operations are processed by creating Action instances; these actions are queued for execution in separate worker threads. The deploy and undeploy actions and all callbacks are processed in this way. This can lead to actions and messages being processed after they are no longer relevant -for example a series of lifecycle events could be queued in succession, or multiple undeploy actions for the same job being in the queue. A refined system would be more aware of the state of the queue, and merge duplicate or inconsistent actions.

Client

The initial deployment client consists of a set of command line based console applications -one per operation. These applications are easy to fully test with JUnit -the command line applications are effectively the core of the functional tests against the server. This is significant as the tools for automated GUI testing are still significantly immature.

We have some experimental support in SmartFrog itself for deploying JUnit test suites across multiple machines and correlating the results. This would make it possible to perform a full functional test in which multiple remote clients made deployment requests of a single server. This will be an interesting project in distributed testing.

Issues with JAX-RPC

Apache Axis is the most widely used open source implementation of the JAX-RPC API, which is the official Java API for SOAP [jaxrpc]. Axis provided both the client and server SOAP engines, and generated all the serialization classes that represent elements within the SOAP messages.

The design of this service API was done using what is considered the strictest and purest SOAP development process -we wrote the XML Schema, then the WSDL file, and used these to generate Java classes to represent the XML data in the messages. We also had a

design which was explicitly designed to support arbitrary XML in a number of places, such as the deployment descriptor itself, the options map, and in the server and application status messages.

It was somewhat unfortunate, therefore, that the JAX-RPC API, which is the standard Java API for working with SOAP messages, proved fundamentally unsuited towards supporting arbitrary XML within a request. All `xsd:any` declarations in the schema were mapped into `MessageElement` instances, a class that -mostly- resembles a W3C DOM. Yet the standard way of extracting the XML from the `MessageElement` classes is to marshal the XML into a string, and then parse it again. Turning arbitrary XML into a `MessageElement` tree is an even more complex operation. There is a fundamental mismatch here between what is consider best practise for XML Schema -adding structured extension points for future extensibility, and what JAX-RPC supports.

We also found that the object serialization generated for us was inadequate or inappropriate. For example, all enumeration types (`xsd:enumeration)` were mapped to Java1.4 classes with a string for each enumeration value, strings named `value1`, `value2`, `value3`, rather than with useful names. A change in the ordering of the enumeration values in the schema would therefore stop the code working.

A further issue is that as there is no real schema validation of incoming messages, it is left to the endpoint to validate the presence or absence of elements, as `minOccurs` and `maxOccurs` declarations in the schema have no effect. This increases the complexity of the endpoint -and increases the probability that other implementations will not be as robust.

There has to be a better way of allowing SOAP endpoints to process arbitrary incoming messages, a way that is resilient to change and works at the XML level, rather than that of badly-serialized Java datatypes. If such a technology is not created, then SOAP users will be left with the worst of both words -a painful interface to a wire format optimised for interoperability problems.

Conclusions

We have described an XML-based API for deploying services, one that is compatible with SOAP1.1 engines. The API is independent of any particular deployment language; the prototype has complete support for the SmartFrog syntax and rudimentary handling of XML-CDL content.

A key to future flexibility of the system is that a deployment request can contain an arbitrary set of options, any of which can be marked as an options which the runtime is required to be able to understand. This, combined with a group design of common options, should provide a structured extension mechanism for the API.

Many open issues have been raised by the prototype, such as naming, fault representation and callbacks. Ultimately, a single callback mechanism for SOAP-based applications will emerge. Until then, SOAP users are left to suffer. The multiple-callback tactic adopted by this API is crude and potentially likely to lead to incompatibilities between caller and service, unless every service implementation is guaranteed to support all common callback options.

Finally, the prototype implementation raises concerns about the suitability of Java's JAX-

RPC SOAP API for handling arbitrary XML. Many of the 'convenience' mappings of XML to Java turned out to be less than useful. We believe that if XML-centric processing of inbound or outbound SOAP messages is considered a good SOAP development practise, then something significantly more usable than JAX-RPC is required.

Bibliography

[ant] Duncan-Davidson et al., *Apache Ant 1.6*, http://ant.apache.org/

[axis] *Apache Axis*, http://ws.apache.org/axis

[jaxrpc] Chinnici et al, *Java API for XML-RPC JAX-RPC 1.1*, Sun Microsystems. 2003, http://java.sun.com/xml/jaxrpc/

[jdml]McGough, S. *The WS-JDML: A Web Service Interface for Job Submission and Monitoring* http://www.nesc.ac.uk/talks/415/02.pdf

[jsdl] *The JSDL Specification*, draft 0.5 http://forge.gridforum.org/projects/jsdl-wg/document/draft-ggf-jsdl-spec/en/9

[smartfrog] Goldsack, P. *SmartFrog Language* , 2004 http://forge.gridforum.org/projects/cddlm-wg/document/SmartFrogLanguage/en/1

[sreq] Loughran, S. *Requirements of a Service API for CDDLM* http://forge.gridforum.org/projects/cddlm-wg/document/Deployment_API_Requirements/en/1

[wsevt] *Web Services Eventing (WS-Eventing)* http://ftpna2.bea.com/pub/downloads/WS-Eventing.pdf

[wsrf] *The WS-Resource Framework*, http://www.globus.org/wsrf/

[xmlcdl] Tatemura, J. *XML-CDL*, 2004, http://forge.gridforum.org/projects/cddlm-wg/document/XML_CDL/en/1

[xmpp] *Jabber::Protocol* http://www.jabber.org/protocol/