Basic Execution Management Services (BES): Rendering

Karl Czajkowski, Ian Foster and Steve Tuecke for the Globus Alliance

karlcz@univa.com, foster@mcs.anl.gov, tuecke@univa.com

Draft of 9 March 2005

This document builds on the companion document "BES Approach" to describe in more detail the issues related to protocol rendering for BES that are of concern to the Globus Alliance.

Pattern: We assume the main BES interface is a message interchange that represents the creation pattern wherein the Actor Instance Descriptor (the job description) is input and a response is either rejection (a fault) or the name/address of the newly created Actor Proxy (management interface) representing the Actor.

The GT 4.0 GRAM creation pattern is a proprietary factory interface[1] that takes our own job definition XML document[2] and returns the EPR of a "ManagedJob" WS-Resource to which job-specific operations can be directed.

A WS-Agreement[3] rendering of EMS would use the AgreementFactory for this creation step, with the successful result being the EPR of the resulting Agreement. The execution management state can be represented in three different ways:

1) the Agreement WS-Resource may be extended with additional execution-specific operations in its WSDL to form an Actor Proxy;
2) the Agreement WS-Resource can reference an external (newly created) Actor Proxy WS-Resource by holding its EPR in a resource property; or
3) the Agreement WS-Resource can expose some other arbitrary Abstract Name which is used in other domain-specific service interfaces.

Input: We believe that the creation pattern should use a declarative syntax in which the Actor Instance Descriptor is expressed completely in the input document.

We believe that the GGF JSDL work[4] is a good starting point to capture typical execution descriptions. We can examine in detail differences between JSDL and other candidate languages such as GT 4.0 GRAM job descriptions. As indicated in the overview document, there are issues with capturing the precise agreement semantics for a particular BES Container, e.g., to denote the kind of obligations the Container is accepting when responding positively to a creation request. We suspect that this can be addressed through

[1] http://www-unix.globus.org/toolkit/docs/development/4.0-drafts/execution/wsgram/WS_GRAM_Public_Interfaces.html
[2] http://www-unix.globus.org/toolkit/docs/development/4.0-drafts/execution/wsgram/schemas/mjs_job_description.html
[3] http://www.gridforum.org/Meetings/GGF11/Documents/draft-ggf-graap-agreement.pdf
[4] http://www.epcc.ed.ac.uk/~ali/WORK/GGF/JSDL-WG/

either envelope or “extension terms” in JSDL to essentially annotate *how* the JSDL expression is to be understood.

Reliability: One issue that arises is the potentially high stakes of errors in the creation pattern. Specifically, while a client may well be expected to tolerate rejection and even Actor failures, it is undesirable to have Actor “disappear” or get duplicated due to message-layer, client, or provider failures.

We assert that a simple creation pattern via one request-response exchange is sufficient because there are multiple binding options available to get reliability. One approach is to have transactional bindings which use commit and rollback to make sure a creation occurs exactly once or not at all.

However, not all Internet deployments will have transactional messaging, so another approach is to use WS-Addressing MessageID header to get idempotent invocation. This mechanism allows the client to resend an application message in case it missed the response message, and the provider must resend the response without duplicating actions such as Actor instantiation. This “at most once” semantics is sufficient for many real world BES scenarios, as the client will eventually learn whether the Actor was accepted or not.

GT 4.0 GRAM optionally uses a proprietary message-level concept that is equivalent to the WS-Addressing MessageID, both to work across any client tooling and because it was designed before the WS-Addressing mechanism was fully clarified by its authors. In this variant, the idempotent ID is sent as an extra field in the input message and interpreted early in the request processing to avoid duplication.

The idempotent operation style optimizes the success case by not requiring additional message exchanges unless there is an error condition or timeout. In contrast, a transactional approach requires more exchanges to setup and commit (or cancel) the invocation.

Asynchrony: Another concern is how much delay may be encountered in the creation pattern. The Web services architecture[5] makes no statements about the relative duration of an “in-out” message exchange, as that is essentially a binding issue. However, two camps seem to dislike long message delays for different reasons which depend on a number of "practical" assumptions not present in the base architecture.

First, one camp confuses the WSDL message protocol with an API specification, so they believe that a WSDL “in out” message must mean a blocking procedure call in their client bindings. They are uncomfortable with the implication that a long-delayed operation cannot be invoked asynchronously by their client application. We believe this is a mistaken viewpoint to take when designing protocols, because the asynchrony of the client can be addressed simply by using an appropriate tooling strategy. For example, GT4.0’s C language Web services tooling generates synchronous and asynchronous stubs for each WSDL operation, so our GT 4.0 GRAM client tool is able to perform the

[5] http://www.w3.org/TR/ws-arch/

creation message exchange using asynchronous “post message” and “response callback” idioms in the C programming language. Similarly, current revisions of the JaxRPC model provide improved asynchronous API bindings.

The second dissenting camp is concerned that long response delays will be fragile because some bindings cannot tolerate the delay. For example, a SOAP over HTTP binding may not be able to wait long enough for a response before the TCP connection is lost. Because an “in out” message pattern is correlated by POST and response in SOAP over HTTP, it is not as durable as an explicit peer-to-peer message exchange using “in only” messages sent to explicit endpoints at both peer sites.

Unfortunately, the "in only" style is also problematic in constrained binding environments because SOAP over HTTP is often valued specifically for being asymmetric and for allowing simple NAT/firewall traversal from “anonymous” clients to well-known providers. If we render a peer-to-peer interface model in order to side-step fragile binding support for "in out" exchanges, we create obstacles for these other common deployment environments.

A third solution which happens to address both camps is to render explicit “post” and “poll” interfaces to initiate the logical operation and then hold the response content at the provider until the client can reconnect and retrieve the result. This supports fragile bindings in NAT/firewall environments and also yields a somewhat asynchronous interface, even over naive tooling that generates synchronous stubs for “in out” message exchanges. However, it complicates the application-level modeling and lifts transport-level message buffering and correlation into the application-level service implementation.

We argue that a simple “in out” message exchange in combination with idempotent identifier mechanisms can often satisfy the fragile bindings and NAT/firewall asymmetry concerns. This approach still retains state at the provider, but rather than adding post/poll operations to the WSDL it simply uses message send for “post” and message resend for “poll.” This approach also means that the application logic can be written using the more natural “in out” pattern and a simple buffering layer at (or slightly above) the binding code can handle the resends at the provider.

We believe that in the long-term, OGSA-WG and GGF in general should be advocating the development and wide deployment of more suitable message binding mechanisms for Web services in a Grid environment, rather than compromising service models in order to cope with immature, first-generation Web service bindings.