GRID SERVICES AND CLIENT PROGRAMMING MODELS

A Simple Registry Utilizing the OGSI Service Group Concepts

Let us now explore how a simple private registry holds factories responsible for the creation of grid services in a container. This is a locally managed registry for the factories of the services running in a container. An example is simply constructed, based upon the OperatingSystemFactory service, which is responsible for OperatingSystem creation.
This registry implements the ServiceGroup portType. The meta-data in the registry comes from ogsi:Factory portType, where the service data specifies all the services that a factory can create.
The following discussion is focused on creating this grouping service. Since most of the meta-data for the factory registry is well defined and quite static in nature, the data can be defined in a GWSDL description as static service data values.

The important information we can derive from the above listing is:

- Since these services are managed locally, the serviceGroupEntryLocator is not utilized.
- The memberLocator rule provides the handle to the factory instance, where we have two factories registered with this registry.
- The content rule states what type of services each factory can create. These services must conform to the interfaces identified in the logic.

This registry, upon startup, loads this information and updates ServiceGroup's "membership-Rules" and "entry" service data fields.

The client can utilize the standard grid service calls to discover the factory instances that can be utilized for creating the "operating system" services. The client can query the registry for all the ServiceGroup entries using the "findServiceData" operation.

The clients can now run an XPath query on the above results to find out the factory that can be used to create an "OperatingSystem" service using the XPath expression in the following logic: //ogsi:EntryType[content/ogsi:CreateServiceExtensibilityType/createInstance="OperatingSystem"].

This results in a list of EntryTypes that will create an operating system service. Again, to obtain the specific handle to the factory execution, this XPath query is: //ogsi:handle.

The client can utilize this GSH to the factory, obtained by execution of the above step, in order to create the operating system service.

We can now reduce the number of client-side XPath operations if the service supports a query by the XPath "operationExtensibility" type. In these cases, the client can issue the "findServiceData" call with the XPath information, and can then directly obtain the factory handle.

Grid Services and Client Programming Models

As presented in previous discussions, the details surrounding the concepts of OGSI are rich and robust across several dimensions. Let us now focus on exactly how to define the client-side programming patterns that interact with an OGSI-defined grid service.

Earlier discussions were focused on how the OGSI is based upon Web services concepts, and how the core direction of this standardization process is on the interoperability between grid services and Web services. This interoperability can be achieved by using the OGSI standards-driven XML messages and the exchange patterns. The OGSI is not defining a new programming model, rather the OGSI is grounding itself upon the existing Web service programming model. At the same time, the OGSA provides some standard interaction pattern including conversion of the GSH to the GSR.

The grid services are uniquely identified by the GSH. A client should convert the GSH (i.e., a permanent network-wide pointer) to a GSR prior to accessing the services. The GSH does not convey tremendous amounts of information; instead, the GSH provides the identity of a service. Once the client retains a GSR, it can access the service.

There are two types of clients:

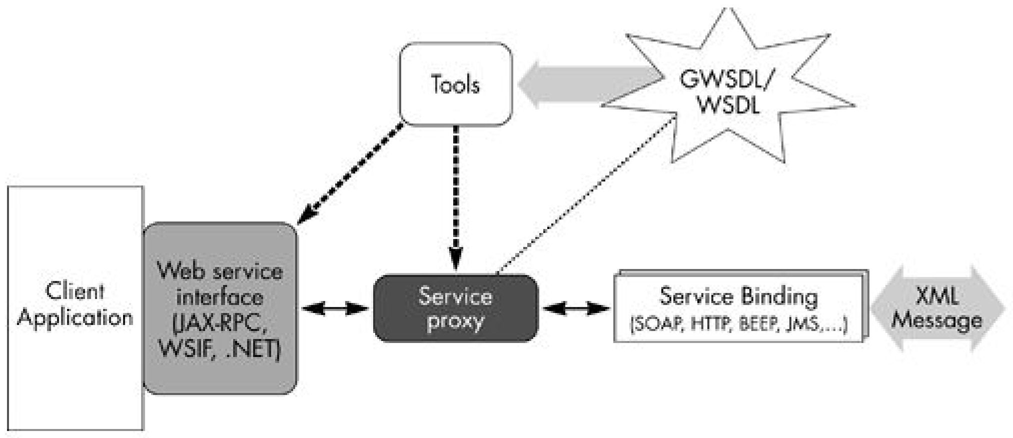
Static. These kinds of clients have plurality of a priori knowledge on the runtime binding information. This includes aspects of native host system message mapping capabilities, the language maps, types for marshalling/de-marshalling of messages, creation of service helpers, and proxies. Normally speaking, the client uses the WSDL/GWSDL information to create these artifacts. These kinds of clients are faster, yet less flexible in operations. Any change in service description requires the recreation of the above artifacts.

Dynamic. These types of clients are flexible and they are not bound to any predefined artifacts. They start from the service description discovery and runtime creation of interaction artifacts, including binding and type-mapping information. These types of clients are highly flexible, yet may perform with less efficiencies.

The locator helper class, as defined by OGSI, provides an abstraction for the service binding process and handle resolution process. Most of the grid toolkits today may generate this artifact to help the client deal with GSHs, GSRs, and interactions between the two entities.

The client framework is, in theory, simple, as illustrated in Figure 6.4

Figure 6.4. The client-side framework


Tools
GWSDL/ WSDL
Client Application
Web service interface (JAX-RPC, WSIF, .NET)
Service proxy
Service Binding (SOAP, HTTP, BEEP, JMS,...)
XML Message

The client always needs to be aware of the GWSDL extension, and should rely on the tools for the transformation of GWSDL portTypes to WSDL portTypes for simplified aspects involving interoperability with existing WSDL 1.1 tools and frameworks. In the next section, we introduce specific implementations of the client-side artifacts from the perspectives of different grid toolkits.

Source : http://elearningatria.files.wordpress.com/2013/10/ise-viii-grid-computing-06is845-notes.pdf