the globus toolkit

GT4.0 C WS Design

Design of Web Services in C for the Globus Toolkit

1 Table Of Contents

1 Table Of Contents 1
2 Introduction 1
3 Service Engine 1
  3.1 State Diagram 2
  3.2 Service Engine API 5
  3.3 Service Container 5
  3.4 Application-Embedded Services 5
4 Service Modules 6
  4.1 Operation Dispatching 6
5 Operation Providers 7
  5.1 Implementation 7
6 Handlers 8
7 Wildcards 8
  7.1 Deserialization 10
  7.2 Type Registry 11
  7.3 Serialization 11

2 Introduction

This document provides a description of the design and organization of the C web services components of the Globus Toolkit. The primary motivation for these components is the emergence of the Web Services Resource Framework, and the need for a stable, high performance C implentation of the Web Services protocols on which WSRF is based. The main components discussed here are the service engine, which manages service invocations and routes the invocations to the appropriate services; operation providers which allow generic implementations to be defined and added to many services at runtime; message handlers which allow dynamic control of how message headers are serialized and deserialized; and some of the details of XSD wildcard handling in C.

3 Service Engine

The entire invocation made by a service request passes through a number of basic steps. Figure 1 depicts each of these steps:

- Transport – This component is responsible for the I/O (listening, reading, writing) and specific transport protocol handling (http, httpg, etc.), and from this, producing or consuming generic SOAP messages.
- Handler Chain – The handlers perform service generic pre processing of the SOAP request message before it gets de-serialized by the service module, and generic post-processing after the response message gets serialized by the service module.
- Service Module – This provides the (de)serialization of the SOAP message body, dispatching of the appropriate operation, and finally calling the actual service implementation

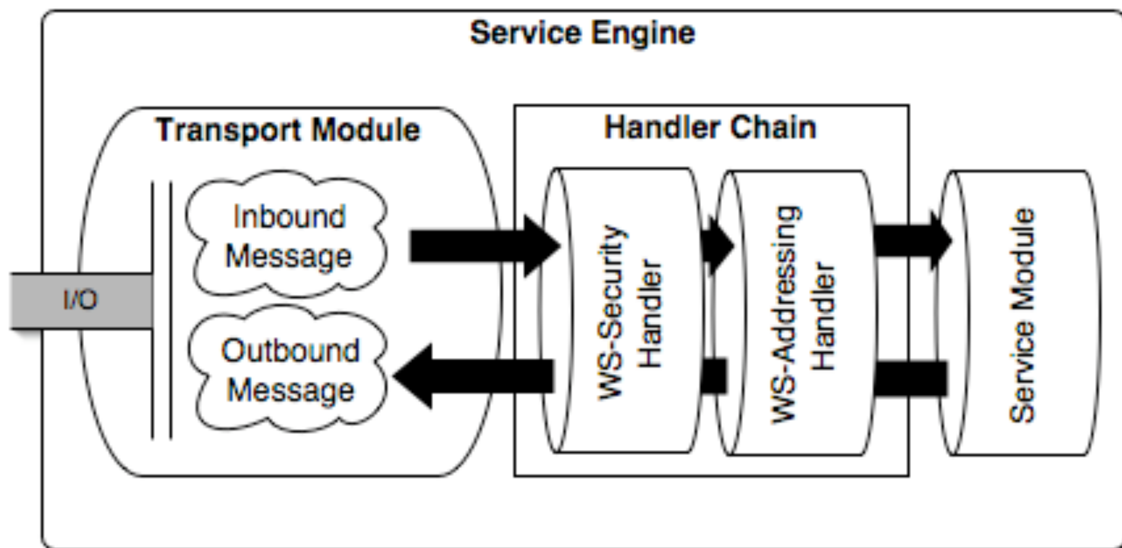

Figure 1: Basic Flow Diagram of the Service Engine

3.1 State Diagram

The service engine passes through a number of states from accepting a new request, to sending the response and closing the connection. The following state diagrams provide the different states that are taking place from starting the service engine, to stopping it. The service engine state machine is separated into two diagrams, one for the service engine accepting new sessions and one for processing a session.

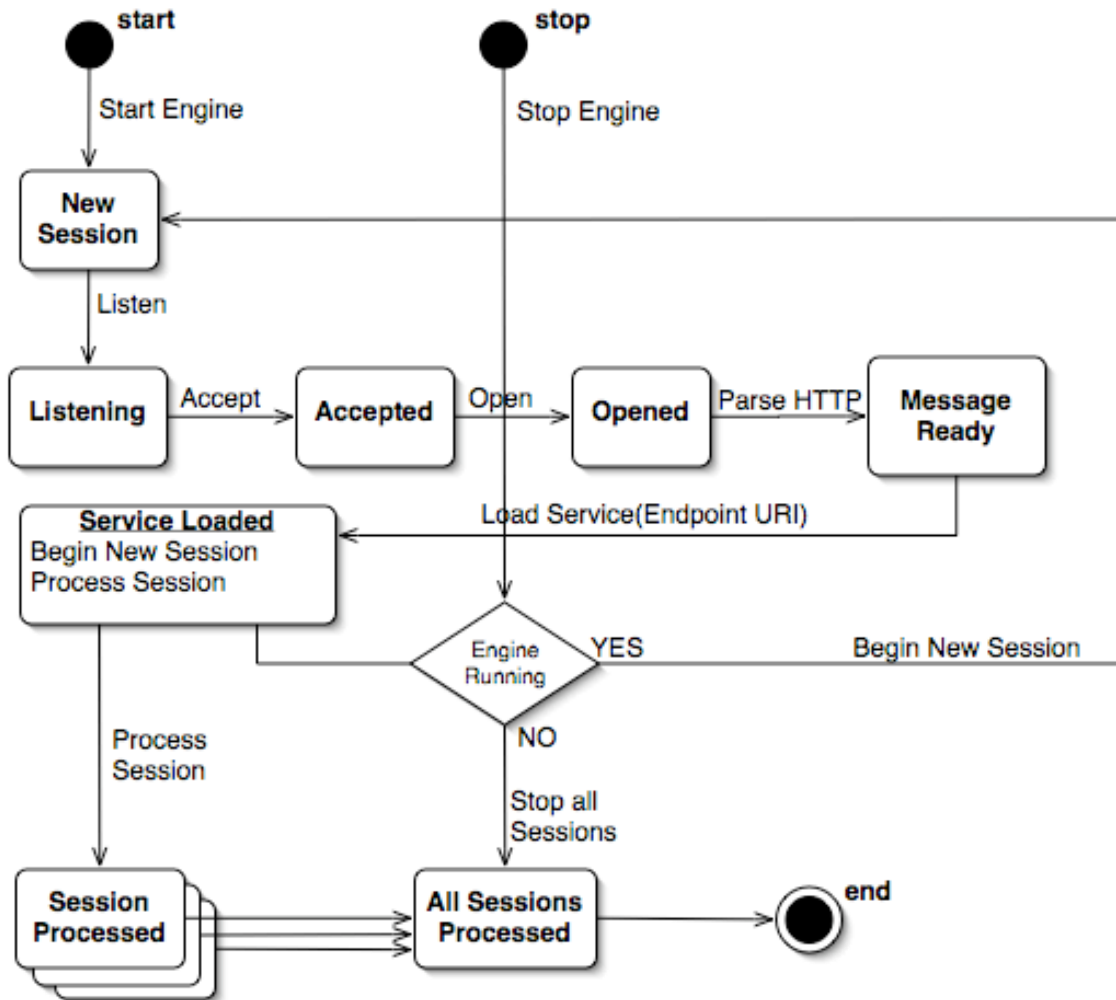

Figure 2: The state machine for new service sessions

Notice that the state machine registers for a new session before processing the current one. This is done using the globus asynchronous event handling API. This allows for a new session to be accepted (and eventually processed) even while the current session is being processed.

The next diagram demonstrates the entire session processing state machine. Its an expansion of the 'Session Processed' node in the above diagram.

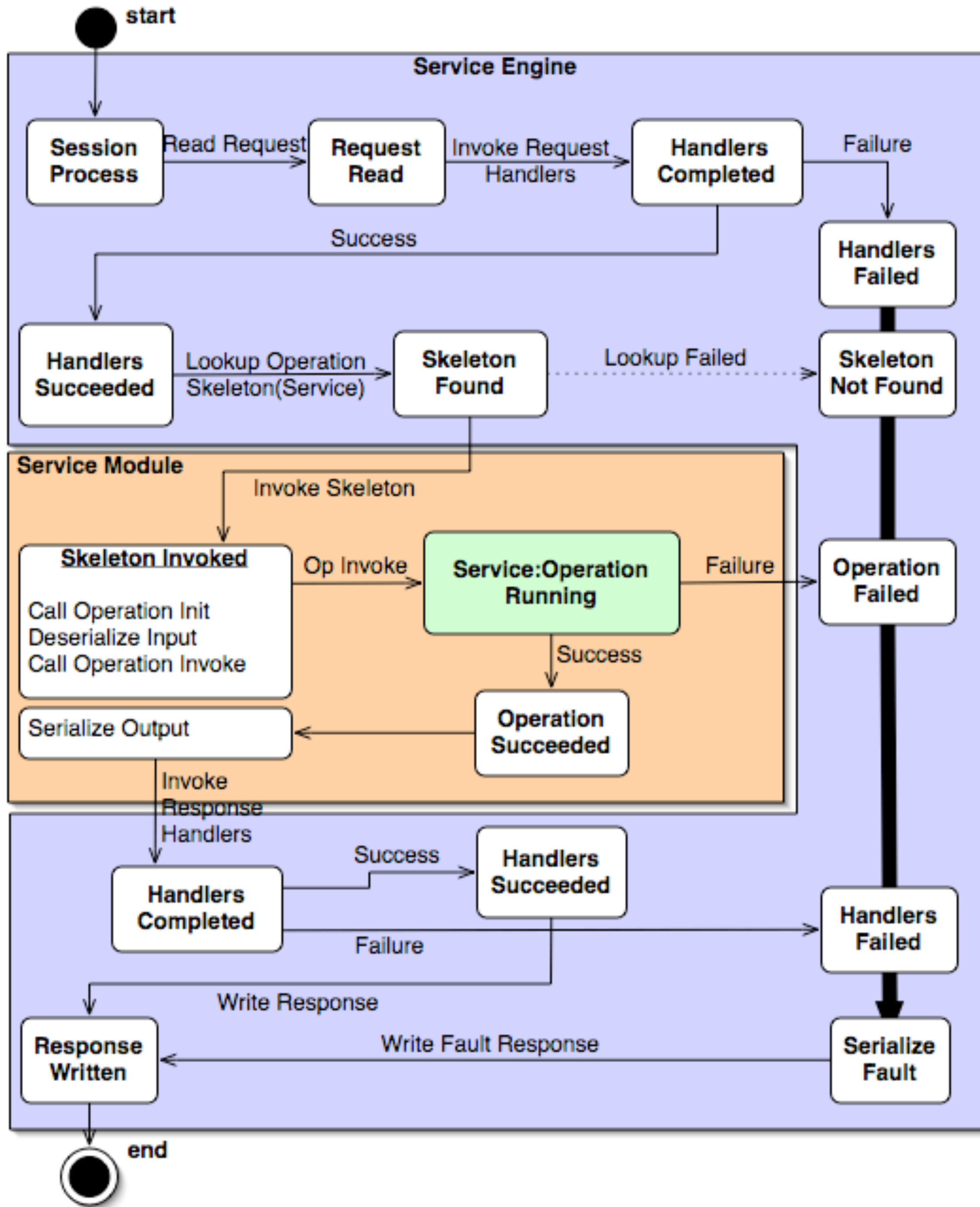

Figure 3: State machine for session processing

The diagram in figure 3 shows the state machine used for processing a service session. This consists of parsing the SOAP message, invoking the appropriate handlers, finding the operation specified in the message, and invoking it. Once the invocation completes, the response handlers are invoked and the response message is sent. The colored boxes separate the service engine from the actual service module containing the service

skeletons and service implementation. All errors are handled by converting the error to a fault of some form, serializing that fault and writing it as the response.

3.2 Service Engine API

The service engine is represented to the user as an API that provides an abstraction of all the different components that it includes. The API provides the following features:

- Provides a simple abstraction of the underlying transport protocols, marshalling, and handler message processing, so that the engine API is clean and simple.
- The service engine is built on Globus event handling API to allow asynchronous handling of each session (with or without threads) so that implementers (such as for stand-alone services) can perform other processing tasks without waiting for a particular operation's session to finish.
- A convenient start/stop API for the service engine with asynchronous handling of sessions. This allows embedded containers to be run in the background of a process.

The service engine API provides the level of abstraction that enables different usage scenarios. The two primary uses of the API in GT 4.0 are the stand-alone service container and the application-embeddable service engine. We describe these in the next sections.

3.3 Service Container

The service container is a process that accepts incoming invocations and routes the requests to the appropriate service skeletons. It allows multiple asynchronous service requests for different services to be processed, or just a single service, based on the installed service modules available. The service container can be built threaded, which will allow multiple services to run simultaneously. This may increase the responsiveness of the service container for services that perform time consuming computations, essentially blocking other service calls from being executed. If a service implementer does need to block in their implementation, this is best handled using calls to `globus_poll_periodic()`.

3.4 Application-Embedded Services

Application-embedded services are easy to setup with the C WS Core. `globus_service_engine_t` handle, and call the `globus_service_engine_register_start` function to begin processing service requests and a call to the `globus_service_engine_register_stop` function to stop inbound requests from being processed. New sessions are driven with the globus event polling functions, such as `globus_cond_wait` or `globus_poll`. Just as with the service container, the service invocations of an embedded service engine will be processed asynchronously. If the application is built with a threaded flavor, the service invocations will be be processed simultaneously.

4 SERVICE MODULES

In C, a web service is represented on the server side as a dynamically loadable module (AKA: shared library) that gets loaded by the container (the umbrella process) as needed. The module provides operation dispatching, de-marshalling and marshalling of the service's messages (generated binding code from a WSDL schema), and the implementation of the service itself (added by the service implementer). The service module is loaded based on the URL passed in from the transport. It gets invoked from the container via a standard interface that must be part of the service module (its implementation is generated for each service by the WSDL to C tool). Pushing all the code for a specific service into a separate module allows the services to be added to a container dynamically without recompiling. It also allows services to be managed by the container through dynamic loading, which allows the container to remain small.

The service module manages operations for a service, performing operation dispatching once a service is invoked. It also keeps track of XSD types defined for that service in a registry, which allows xsd:any types to be handled appropriately.

4.1 Operation Dispatching

Operations dispatching is implemented within the service module. The following requirements for operation dispatching in the GT WS architecture exist:

1. Operation dispatching from the service must conform to the WS-I Basic Profile, which requires that the child element of the SOAP body for a message be unique for a document/literal service, so that it can be used to map from element name to operation.
2. References to service implementation functions must be dynamic, in order to provide operation provider functionality.

The first of these requirements is handled by providing a mapping from element QName to operation name. This mapping is generated at service binding generation.
In order to support the second requirement, instead of hardcoding the operations as the service bindings are generated, a table of operation names to function pointers (actual service functions) is used. This works by providing a hashtable to each generated service module that keeps the operation name to implementation mapping. The table gets initialized and filled (for the operations defined in the schema) as the service module is initialized. The table is able to maintain all the function pointers by storing void pointers and casting the void pointer back to the operation's defined interface once the operation is ready to be dispatched.

The steps of operation dispatching are:

1. Map child element QName of SOAP Body to operation skeleton

2. Call de-marshalling function for operation – This step initializes the parameters that are passed to the service implementation of the operation.
3. Map the operation name to the function pointer that points to the service implementation of the operation
4. Call the function pointer with the parameters from 2.
5. Perform any necessary destruction of parameters created in 2.

5 OPERATION PROVIDERS

Operation providers are implementations of a service operation that provide common functionality (operations) to a service. Such implementations can be useful for two services which require similar/same functionality, but don't want to duplicate each other's work. The motivation for operation providers is based on the Web Services Resource Framework (WSRF), which (among many other things) provides a base set of operations for managing stateful resources. For example, the WS-ResourceProperty.wsdl schema defines the GetResourceProperty operation in the GetResourceProperty portType. Someone implementing a WSRF enabled web service would write a schema that defines portTypes that implement (in the WSRF sense) the GetResourceProperty portType. The service implementation must then somehow provide an implementation of the GetResourceProperty operation, but since this operation provides the same functionality to any web service, its implementation can be reused.

5.1 Implementation

The implementation of operation providers in C is built on the Globus C extension framework, which allows a module to be placed in a registry for later access.
An operation provider is added to a service module by including the header of the operation provider and adding a dependency to the operation provider package. This must be done by the service implementer. The operation provider does not provide any (de)marshalling or dispatching for its operations, it contains only the service implementation(s) of the operation(s) it provides. The (de)marshalling and dispatching mechanisms of the service provider are expected to be included in the generated bindings for the service module.

A service implementer must specify the operation providers they want programmatically. The C architecture does not allow an operation provider to be added to a service at run time through dynamic deployment. The service implementer must take the following steps to use an operation provider:

- Add the operation provider's header to the service implementation source file
- Add a dependency on the operation provider package for the service module package
- Use the following function to specify which operation providers should be used:

```
globus_result_t
globus_service_set_operation_provider(
      const char *                       operation_name,
      void *                             operation_impl);
```

This function can be called anywhere from within the service implementation, but in general it should be called during service activation so that the operations an operation provider implements will be available to service requestors.

6 HANDLERS

Normally, a request message is received from the transport layer and passed directly to the service. This limits the functionality of the service container. For example, developers may want to do pre-processing on the request message or post-processing on the response message at a global level (for all services in the container). Also, such generic processing of messages may need to be specified for some services and not others. Handlers are a way of adding this functionality to the service engine.

From an implementation standpoint, a handler is simply a function that implements a standard interface accepting a message context (an abstract representation of the message) as input, and returning one as output. Handlers are called before de-marshalling for inbound messages, and after marshalling for outbound messages, so that they can modify the message right before being sent, or directly after being received. Each inbound and outbound message has an associated handler chain, which contains a user-defined set of handlers in a specified order. As the message is processed, the handlers in the chain are invoked in turn.

Handlers can also be implemented as dynamic modules, loadable at runtime. The module includes the function for the handler, as well as code for loading and managing the module. The global handlers and service specific handlers added to a chain for a given message are defined programmatically through the stub interfaces.

7 WILDCARDS

XML Schema allows the xsd:any type to represent an arbitrary XML blob as a typed element or other XML content. This is useful where an element must be represented, but the exact type of the element is unknown in advance (at compile time). In order for the C SOAP engine to be able to handle this xsd:any type, the following structure is used:

```
struct globus_xsd_type_info_s
{
    xsd_QName *                         type;
    globus_xsd_serialize_func_t         serialize;
    globus_xsd_deserialize_func_t       deserialize;
    globus_xsd_init_func_t              initialize;
    globus_xsd_destroy_func_t           destroy;
    globus_xsd_copy_func_t              copy;
    globus_xsd_init_contents_func_t     initialize_contents;
    globus_xsd_destroy_contents_func_t  destroy_contents;
    globus_xsd_copy_contents_func_t     copy_contents;
    size_t                              type_size;
    globus_xsd_array_push_func_t        push;
    globus_xsd_type_info_t              contents_info;
    globus_xsd_type_info_t              array_info;
};
```

The first field of this struct points to another global variable generated for the given type and statically initialized. This is the QName of the given type, and is used as the key entry into the type registry. The other fields consist primarily of function pointers that define how the type is to be serialized and deserialized, as well as initialized, copied and destroyed. The any type is represented in C as the following structure:

```
typedef struct xsd_any_s
{
    globus_xsd_type_registry_t          registry;
    globus_xsd_type_info_t              any_info;
    xsd_QName *                         element;
    void *                              value;
} xsd_any;
```

This type allows us to represent the C form of a type that is unknown until runtime, using the void pointer `value` field. This acts as a placeholder for any other variable that can be cast to a void pointer (all other pointers). Each of the fields of this type are defined here:

- `registry` – a reference to a type registry used only to deserialize this particular any variable.
- `any_info` – a reference to the type information used to deserialize this type
- `element` – the outermost element used for serializing this variable. If the variable is expected to be serialized, the element must be set by the user. If the variable is a going to be filled in by deserialization, the element will be set by the deserialization code.
- `value` – this is the void pointer holding the actual deserialized content of this any variable.

Notice the `value` member of `globus_xsd_any_s` struct is a void pointer. Using void pointers allows us to represent any possible pointer type, as pointers can be cast to void pointers and back to their original type. The different function pointers provide information about how to handle the any type.

7.1 Deserialization

For both services and clients, an xsd:any type in an inbound message must be deserialized appropriately. This is done using a type registry, which maps type QNames and element QNames to type info structures. Once an element is reached during deserialization which is known from schema to be a wildcard, the `xsd_any_deserialize` function is called with an instance of the `xsd_any` type. The deserialize function performs a search for a valid type info structure to deserialize the any variable. The search is done as follows:

1) **Instance Type Info** - The `any_info` field of the `xsd_any` instance is used if it gets initialized (non-null) before the deserialize function is called.
2) **Element QName** - The element QName is used first as the key to lookup the type info in a registry. If the first registry lookup fails, the next registry in the search is used. The order of the registries used to perform the lookups are as follows:
    a) **Instance Registry** – If the `xsd_any` instance's registry is non-null, this registry is used to lookup the type info. If the lookup with the element QName succeeds in finding a valid type info from this registry, that type info is used, and the search is aborted.
    b) **Message Registry** – The message handle passed to each deserialize function also maintains its own reference to a type registry. If this registry is non-null and the lookup with the element QName succeeds, the found type info is used, and the search is aborted.
    c) **Global Registry** – The global registry is the last registry used to lookup the type info. If the element QName is a valid key in the registry, the found type info is used and the search is aborted.
3) **xsi:type Attribute** – The element may contain an xsi:type attribute which defines the QName to use for deserialization. If such an attribute exists, the QName value is used as the key for each of the lookups in the following type registries:
    a) **Instance Registry** – If the `xsd_any` instance's registry is non-null, this registry is used to lookup the type info. If the lookup with the value of the xsi:type attribute succeeds in finding a valid type info from this registry, that type info is used, and the search is aborted.
    b) **Message Registry** – The message handle passed to each deserialize function also maintains its own reference to a type registry. If this registry is non-null and the lookup with the value of the xsi:type attribute succeeds, the found type info is used, and the search is aborted.
    c) **Global Registry** – The global registry is the last registry used to lookup the type info. If the xsi:type attribute is a valid key in the registry, the found type info is used and the search is aborted.
4) If no valid type info has been found in any of the previous searches, the type info is set to the `globus_xml_buffer_info`, which essentially places the wildcard element and all its children into a buffer. This allows the deserialization of the wildcard to be handled by the user.

7.2 Type Registry

A type registry maintains a mapping of QNames for types and elements to type info structures, allowing deserialization of wildcards to take place at runtime. There are different type registries that can be used in different contexts. We describe each of them here:

- **Global Registry** – A global registry for the entire process, contains mappings for every linked in element and type QName. So all the top level element and type definitions defined in the XML Schemas that are referenced by a service definition will have entries in this registry. The element and type definitions are added at runtime during module activation, so when the client module is activated, or the service module is loaded, the registry will be populated.
- **Message Registry** – A per-message registry that allows the message handlers and operation init functions to modify the behavior of deserialization parameters for an individual message. Entries found in the message registry will override entries in the global registry. By default this registry is empty. In order to setup a special entry in this registry for a particular element or type, the globus_soap_message_handle_set_registry
- **Instance Registry** – A registry contained within the wildcard instance, to allow the deserialization behavior of each wildcard to be controlled.

7.3 Serialization

At serialization, the `xsd_any` structure must contain a valid `any_info` field which points to the `globus_xsd_any_info_t` containing the serialization functions to use for this particular wildcard. The `value` and `element` fields must also contain valid values for the content to be serialized and the outer element to use for serialization. Once these parameters are set the serialization functions take care of the rest.