GT 4.0.x: C WS Core

GT 4.0.x: C WS Core

Table of Contents

1. Key Concepts 1
    1. Overview 1
    2. Conceptual Details 1
    3. Related Documents 6
2. System Administrator's Guide 7
    1. Introduction 7
    2. Building and Installing 7
    3. Configuring 8
    4. Deploying 8
    5. Testing 8
    6. Security Considerations 8
    7. Troubleshooting 9
    8. Usage statistics collection by the Globus Alliance 9
3. User's Guide 10
    1. Introduction 10
    2. Command-line tools 10
    3. Troubleshooting 10
    4. Usage statistics collection by the Globus Alliance 10
4. Developer's Guide 12
    1. Introduction 12
    2. Before you begin 12
    3. Architecture and design overview 14
    4. Public interface 14
    5. Usage scenarios 14
    6. Tutorials 18
    7. Debugging 35
    8. Troubleshooting 35
    9. Related Documentation 35
5. Fact Sheet 36
    1. Brief component overview 36
    2. Summary of features 36
    3. Usability summary 36
    4. Backward compatibility summary 36
    5. Technology dependencies 37
    6. Tested platforms 37
    7. Associated standards 37
    8. For More Information 38
6. Public Interface Guide 39
    1. Semantics and syntax of APIs 39
    2. Semantics and syntax of the WSDL 40
    3. Command-line tools 40
    4. Overview of Graphical User Interface 40
    5. Semantics and syntax of domain-specific interface 40
    6. Configuration interface 40
    7. Environment variable interface 41
7. Quality Profile 42
    1. Test coverage reports 42
    2. Code analysis reports 42
    3. Outstanding bugs 42
    4. Bug Fixes 42
    5. Performance reports 43

8. GT 4.0 Samples for C WS Core 44
1. Counter Client 44
9. Migrating Guide 45
1. Migrating from GT2 45
2. Migrating from GT3 45
I. GT 4.0: C WS Core Command Reference 46
globus-wsc-container 47
globus-wsrf-cgen 48
10. 4.0.8 Release Notes 50
1. Introduction 50
2. Changes Summary 50
3. Bug Fixes 50
4. Known Problems 50
5. For More Information 50
11. 4.0.7 Release Notes 51
1. Introduction 51
2. Changes Summary 51
3. Bug Fixes 51
4. Known Problems 51
5. For More Information 51
12. 4.0.6 Release Notes 52
1. Introduction 52
2. Changes Summary 52
3. Bug Fixes 52
4. Known Problems 52
5. For More Information 52
13. 4.0.5 Release Notes 53
1. Introduction 53
2. Changes Summary 53
3. Bug Fixes 53
4. Known Problems 53
5. For More Information 53
14. 4.0.4 Release Notes 54
1. Introduction 54
2. Changes Summary 54
3. Bug Fixes 54
4. Known Problems 54
5. For More Information 55
15. 4.0.3 Release Notes 56
1. Introduction 56
2. Changes Summary 56
3. Bug Fixes 56
4. Known Problems 56
5. For More Information 56
16. 4.0.2 Release Notes 57
1. Introduction 57
2. Changes Summary 57
3. Bug Fixes 57
4. Known Problems 58
5. For More Information 58
17. 4.0.1 Release Notes 59
1. Introduction 59
2. Changes Summary 59
3. Bug Fixes 59

4. Known Problems 60
5. For More Information 60
18. 4.0.0 Release Notes 61
1. Component Overview 61
2. Feature Summary 61
3. Bug Fixes 61
4. Known Problems 62
5. Technology Dependencies 63
6. Supported Platforms 63
7. Backward Compatibility Summary 63
8. For More Information 64

List of Tables

2.1. Building C WS Core from installer 7
2.2. Building C WS Core from CVS 8

Chapter 1. GT 4.0 Common Runtime Components: Key Concepts

1. Overview

The common runtime components provide GT4 web and pre-web services with a set of libraries and tools that allows these services to be platform independent, to build on various abstraction layers (threading, io) and to leverage functionality lower in the web services stack (WSRF, WSN, etc).

These components are architecturally diverse and it is thus hard to identify a overarching theme. Instead a few subthemes have been identified and elaborated on in the below.

2. Conceptual Details

2.1. Web Services

We introduce basic concepts relating to Web services and their use and implementation within GT4, in particular within the "WS Core" (Java & C) components.

2.1.1. GT4, Distributed Systems, and Web Services

GT4 is a set of software components for building distributed systems: systems in which diverse and discrete software agents interact via message exchanges over a network to perform some tasks. Distributed systems face particular challenges relating to sometimes high and unpredictable network latencies, the possibility of partial failure, and issues of concurrency. In addition, system components may be located within distinct administrative domains, thus introducing issues of decentralized control and negotiation.

GT4 is, more specifically, a set of software components that (with some exceptions) implement Web services mechanisms for building distributed systems. Web services provide a standard means of interoperating between different software applications running on a variety of platforms and/or frameworks.

A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.

Web services standardize the messages that entities in a distributed system must exchange in order to perform various operations. At the lowest level, this standardization concerns the protocol used to transport messages (typically HTTP), message encoding (SOAP), and interface description (WSDL). A client interacts with a Web service by sending it a SOAP message; the client may subsequently receive response message(s) in reply. At higher levels, other specifications define conventions for securing message exchanges (e.g., WS-Security), for management (e.g., WSDM), and for higher-level functions such as discovery and choreography. Figure 1 presents a view of these different component technologies; we discuss specific specifications below in Section 2.1.4, “Web Services Specifications”.

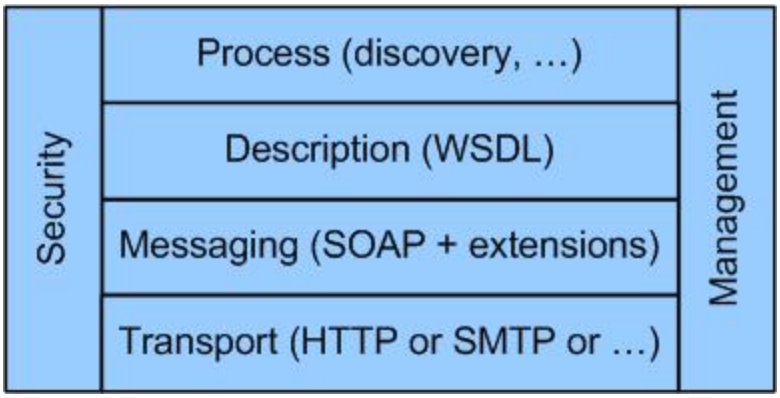

Figure 1: An abstract view of the various specifications that define the Web services architecture

2.1.2. Service Oriented Applications and Infrastructure

Web services technologies, and GT4 in particular, can be used to build both service-oriented applications and service-oriented infrastructure. Deferring discussion of the sometimes controversial term "service-oriented" to later in Section 2.1.9, “Service Oriented Architecture”, we note that a service-oriented application is constructed via the composition of components defined by service interfaces (in the current context, Web services): for example, a financial or biological database, an options pricing routine, or a biological sequence analyzer. Many descriptions of Web services and SOAP focus on the task of defining interfaces to such components, often illustrating their discussion with examples such as a "stock quote service" (the "hello world" of Web services).

Particularly when servicing many such requests from a distributed community, we face the related problem of orchestrating and managing numerous distributed hardware and software components. Web services can be used for this purpose also, and thus we introduce the term service-oriented infrastructure to denote the resource management and provisioning mechanisms used to meet quality of service goals for components and applications. Many GT4 features are concerned with enabling the construction of service-oriented infrastructure.

2.1.3. Web Services Implementation

From the client perspective, a Web service is simply a network-accessible entity that processes SOAP messages. Things are somewhat more complex under the covers. To simplify service implementation, it is common for a Web services implementation to distinguish between:

1. the hosting environment (or container), the (domain-independent) logic used to receive a SOAP message and identify and invoke the appropriate code to handle the message, and potentially also to provide related administration functions, and:

2. the Web service implementation, the (domain-specific) code that handles the message.

This separation of concerns means that the developer need only provide the domain-specific message handling code to implement a new service. It is also common to further partition the hosting environment logic into that concerned with transporting the SOAP message (typically via HTTP, thus an "HTTP engine" or "Web server"-sometimes termed an "application server") and that concerned with processing SOAP messages (the "SOAP engine" or "SOAP processor"). Figure 2 illustrates these various components.

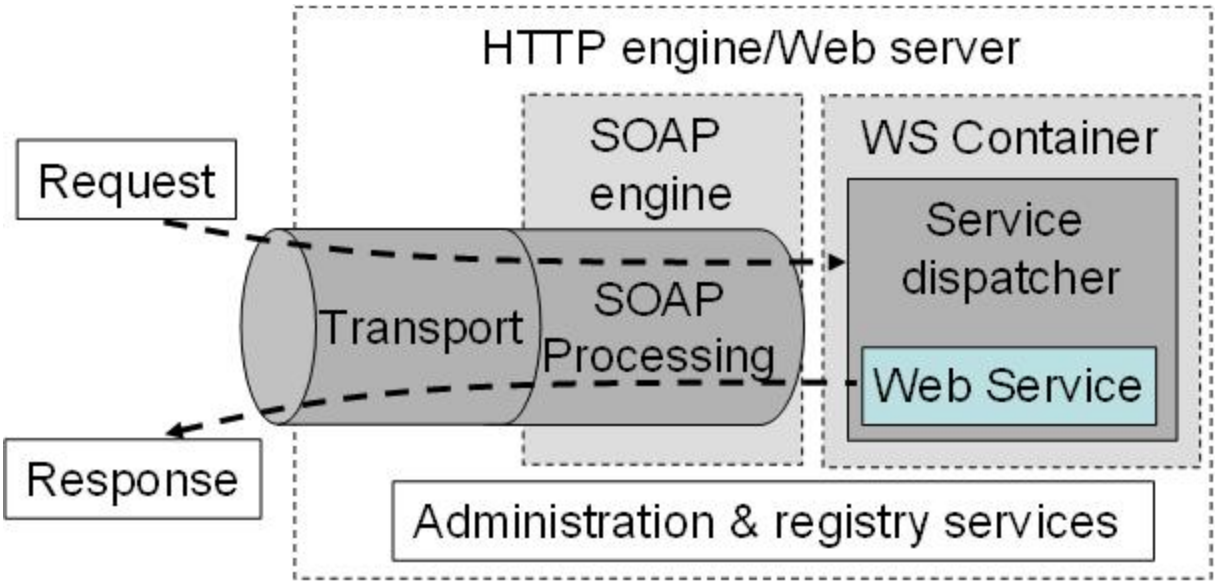

Figure 2: WS Container. High-level picture of functional components commonly encountered in Web service implementations, showing the path taken by requests and responses.

Many different containers exist, with different performance properties, supported Web services implementation languages, security support, and so forth. We mention below those used in GT4.

2.1.4. Web Services Specifications

We provide pointers to the Web services specifications that underlie GT4. These comprise the core specifications that define the Web services architecture (XML, SOAP, WSDL); WS-Security and other specifications relating to security; and the WS-Addressing, WSRF, and WS-Notification specifications used to define, name, and interact with stateful resources. We also speak briefly to emerging specifications that are likely to be important in future GT evolution. An important source of information on the requirements that motivate the use and development of these specifications is the Open Grid Services Architecture.

2.1.5. XML, SOAP, WSDL

XML is used extensively within Web services as a standard, flexible, and extensible data format. In addition to XML syntax, other important specifications are XML Schema and XML Namespaces. Note that while current Web services tools typically adopt a textual serialization, a binary encoding is also possible and may provide higher efficiency.

SOAP 1.2 provides a standard, extensible, composable framework for packaging and exchanging XML messages between a service provider and a service requester. SOAP is independent of the underlying transport protocol, but is most commonly carried on HTTP.

WSDL 1.1 is an XML document for describing Web services. Standardized binding conventions define how to use WSDL in conjunction with SOAP and other messaging substrates. WSDL interfaces can be compiled to generate proxy code that constructs messages and manages communications on behalf of the client application. The proxy automatically maps the XML message structures into native language objects that can be directly manipulated by the application. The proxy frees the developer from having to understand and manipulate XML.

2.1.6. WS-Security and Friends

The WS-Security family of specifications addresses a range of issues relating to authentication, authorization, policy representation, and trust negotiation in a Web services context. GT4 uses a number of these specifications plus other related specifications, notably Security Authorization Markup Language (SAML), to address message protection, authentication, delegation, and authorization, as follows:

- TLS (transport-level) or WS-Security and WS-SecureConversation (message level) are used as message protection mechanisms in combination with SOAP.
- X.509 End Entity Certificates or Username and Password are used as authentication credentials.
- X.509 Proxy Certificates and WS-Trust are used for delegation.
- SAML assertions are used for authorization.

2.1.7. WS-Addressing, WSRF, and WS-Notification

A number of related specifications provide functionality important for service oriented infrastructure in which we need to be able to represent and manipulate stateful entities such as physical resources of various kinds, logical components such as software licenses, and transient activities such as tasks and workflows.

The WS-Addressing specification defines transport-neutral mechanisms to address Web services and messages. Specifically, this specification defines XML elements to identify Web service endpoints and to secure end-to-end endpoint identification in messages.

The WS Resource Framework (WSRF) specifications define a generic and open framework for modeling and accessing stateful resources using Web services. This framework comprises mechanisms to describe views on the state (WS-ResourceProperties), to support management of the state through properties associated with the Web service (WS-ResourceLifetime), to describe how these mechanisms are extensible to groups of Web services (WS-ServiceGroup), and to deal with faults (WS-BaseFaults).

The WS-Notification family of specifications define a pattern-based approach to allowing Web services to disseminate information to one another. This framework comprises mechanisms for basic notification (WS-Notification), topic-based notification (WS-Topics), and brokered notification (WS-BrokeredNotification).

We note that the Web services standards space is in some turmoil due to competing proposed specifications. In particular, Microsoft and others recently proposed WS-Transfer, WS-Eventing, and WS-Management, which define similar functionality to WSRF, WS-Notification, and WSDM (discussed below), respectively, but using different syntax. We hope that these differences will be resolved in the future.

2.1.8. Other Relevant Specifications

The WS-Interoperability (WS-I) organization has produced a number of profiles that define ways in which existing Web services specifications can be used to promote interoperability among different implementations. The WS-I Basic Profile speaks to messaging and service description: primarily XML, SOAP, and WSDL. The WS-I Basic Security Profile speaks to basic security mechanisms. Other profiles are under development.

Web services distributed management (WSDM) specifications under development within OASIS are likely to play a role in future GT implementations as a means of managing GT components.

WS-CIM specifications under development within DMTF are likely to play a role in future GT implementations as a means of representing physical and virtual resources.

The Global Grid Forum's Open Grid Services Architecture (OGSA) working group has completed a document that provides a high-level description of the functionality required for future service-oriented infrastructure and applications, and a framework that suggests how this functionality can be factored into distinct specifications. The OGSA working group is now proceeding to define OGSA Profiles that, like WS-I profiles, will identify technical specifications that can be used to address specific Grid scenarios.

2.1.9. Service Oriented Architecture

We provide some additional discussion concerning the term service oriented architecture (SOA), which is used widely but not necessarily consistently within the Web services community. One common usage is simply to indicate the use of Web services technologies. However, the intention of those who coined the term seems to be rather to contrast two different styles of building distributed systems. Distributed object systems are distributed systems in which the semantics of object initialization and method invocation are exposed to remote systems by means of a proprietary or standardized mechanism to broker requests across system boundaries, marshal and unmarshal method argument data, etc. Distributed objects systems typically (albeit not necessarily) are characterized by objects maintaining a fairly complex internal state required to support their methods, a fine grained or "chatty" interaction between an object and a program using it, and a focus on a shared implementation type system and interface hierarchy between the object and the program that uses it.

In contrast, a Service Oriented Architecture (SOA) is a form of distributed systems architecture that is typically characterized by the following properties:

- Logical view: The service is an abstracted, logical view of actual programs, databases, business processes, etc., defined in terms of what it does, typically carrying out a business-level operation.

- Message orientation: The service is formally defined in terms of the messages exchanged between provider agents and requester agents, and not the properties of the agents themselves. The internal structure of an agent, including features such as its implementation language, process structure and even database structure, are deliberately abstracted away in the SOA: using the SOA discipline one does not and should not need to know how an agent implementing a service is constructed. A key benefit of this concerns so-called legacy systems. By avoiding any knowledge of the internal structure of an agent, one can incorporate any software component or application that can be "wrapped" in message handling code that allows it to adhere to the formal service definition.

- Description orientation: A service is described by machine-processable metadata. The description supports the public nature of the SOA: only those details that are exposed to the public and important for the use of the service should be included in the description. The semantics of a service should be documented, either directly or indirectly, by its description.

- Granularity: Services tend to use a small number of operations with relatively large and complex messages.

- Network orientation: Services tend to be oriented toward use over a network, though this is not an absolute requirement.

- Platform neutral: Messages are sent in a platform-neutral, standardized format delivered through the interfaces. XML is the most obvious format that meets this constraint.

It is argued that these features can allow service-oriented architectures to cope more effectively with issues that arise in distributed systems, such as problems introduced by latency and unreliability of the underlying transport, the lack of shared memory between the caller and object, problems introduced by partial failure scenarios, the challenges of concurrent access to remote resources, and the fragility of distributed systems if incompatible updates are introduced to any participant.

Web services technologies in general, and GT4 in particular, can be used to build both distributed object systems and service-oriented architectures. The specific design principles to be followed in a particular setting will depend on a variety of issues, including target environment, scale, platform heterogeneity, and expected future evolution.

3. Related Documents

3.1. Web Services

- Booth, D., Haas, H., McCabe, F., Newcomer, E., Champion, M., Ferris, C. and Orchard, D. Web Services Architecture. W3C, Working Draft[1]

[1] http://www.w3.org/TR/2003/WD-ws-arch-20030808/

Chapter 2. GT 4.0 C WS Core : System Administrator's Guide

1. Introduction

This guide contains advanced configuration information for system administrators working with C WS Core. It provides references to information on procedures typically performed by system administrators, including installation, configuring, deploying, and testing the installation.

Important

This information is in addition to the basic Globus Toolkit prerequisite, overview, installation, security configuration instructions in the GT 4.0 System Administrator's Guide[1]. Read through this guide before continuing!

2. Building and Installing

In order to build and install the C WS-Core component from an official release:

Table 2.1. Building C WS Core from installer

1	Obtain the latest GT 4.x.x release tarball installer from the Download webpage[2].
2	Untar the tarball: `tar xvfz gt<version>-all-source-installer.tar.gz`
3	Change to the installer directory: `cd gt<version>-all-source-installer`
4	Run: `./configure -prefix=<path to install>`
5	Run: `make wsc`
6	Run: `make install`

In order to build C WS Core from CVS:

[1] http://www.globus.org/toolkit/docs/4.0/admin/docbook/
[2] http://www.globus.org/toolkit/downloads/

Table 2.2. Building C WS Core from CVS

1	Obtain the source code for C WS Core from CVS:
	1. To get the latest source from CVS execute:
	`cvs -d :pserver:anonymous@cvs.globus.org:/home/globdev/CVS/globus-packages \ checkout packaging`
	2. Change into the `packaging` directory.
	`cd packaging`

2	Set the `GLOBUS_LOCATION` environment variable to the absolute path of the target directory of your installation.
	On Unix/Linux:
	`setenv GLOBUS_LOCATION /soft/gt4/`
	or
	`export GLOBUS_LOCATION=/soft/gt4/`

3	Run `make-packages.pl` `./make-packages.pl -bundles=gt4-c-ws-core -deps -install=$GLOBUS_LOCATION`

3. Configuring

3.1. Configuration overview

The C WS-Core component does not provide global configuration functionality.

4. Deploying

The C WS-Core does not require configuration/deployment steps. All parameter configuration is done programmatically at present.

5. Testing

The C WS-Core has a test suite that tests a number of different packages included in the component. The tests can be built using either the `wsctests` target to `make` in the installer of a release, or they can be built using the `gt4-c-ws-core-test` bundle in the `./make-packages.pl` command mentioned previously.

The tests are installed into $GLOBUS_LOCATION/test, and can be run from the appropriate sub-directories.

6. Security Considerations

C WS-Core supports secure transport (https) and secure message (just X509 signing, not encryption).

6.1. Secure Transport

With secure transport, the entire container must be run over an https transport. This is done by default for the C container. If the user does not want security in the container, or wants to use secure message instead of secure transport, they should use the `-nosec` argument to `globus-wsc-container`.

For clients, the secure transport is enabled if the contact URI contains the 'https' scheme instead of 'http', so the client doesn't have to enable or disable it explicitly.

7. Troubleshooting

This is a new component. If you are having trouble using it, please let us know!

8. Usage statistics collection by the Globus Alliance

The following usage statistics are sent by C WS Core by default in a UDP packet :

- Component identifier
- Usage data format identifier
- Time stamp
- Source IP address
- Source hostname (to differentiate between hosts with identical private IP addresses)

It sends it at container startup (globus-wsc-container) and receipt of that packet tells us that the container started.

If you wish to disable this feature, you can set the following environment variable before running the C container:

```
export GLOBUS_USAGE_OPTOUT=1
```

By default, these usage statistics UDP packets are sent to `usage-stats.globus.org:4180` but can be redirected to another host/port or multiple host/ports with the following environment variable:

```
export GLOBUS_USAGE_TARGETS="myhost.mydomain:12345 myhost2.mydomain:54321"
```

You can also dump the usage stats packets to stderr as they are sent (although most of the content is non-ascii). Use the following environment variable for that:

```
export GLOBUS_USAGE_DEBUG=MESSAGES
```

Also, please see our policy statement[3] on the collection of usage statistics.

[3] http://www.globus.org/toolkit/docs/4.0/Usage_Stats.html

Chapter 3. GT 4.0 C WS Core : User's Guide

1. Introduction

The C WS Core is an implementation of Web Services, WSRF, and WSN specifications in the C programming language. This means that a user can write their own Web Services and clients in C, and use the APIs and tools included in in the C WS Core to manage WS-Resources.

The C WS Core includes:

- A small container for services
- An embeddable service container API
- API for managing resources
- API for managing notification consumers
- A WSDL to C binding generator
- Security Support

It does not include support at this time for WSDL generation from C header files, or embedding services into 3rd party containers.

2. Command-line tools

Please see the C WS Core Command Reference.

3. Troubleshooting

This is a new component. If you are having trouble, please let us know!

4. Usage statistics collection by the Globus Alliance

The following usage statistics are sent by C WS Core by default in a UDP packet :

- Component identifier
- Usage data format identifier
- Time stamp
- Source IP address
- Source hostname (to differentiate between hosts with identical private IP addresses)

It sends it at container startup (globus-wsc-container) and receipt of that packet tells us that the container started.

If you wish to disable this feature, you can set the following environment variable before running the C container:

```
export GLOBUS_USAGE_OPTOUT=1
```

By default, these usage statistics UDP packets are sent to `usage-stats.globus.org:4180` but can be redirected to another host/port or multiple host/ports with the following environment variable:

```
export GLOBUS_USAGE_TARGETS="myhost.mydomain:12345 myhost2.mydomain:54321"
```

You can also dump the usage stats packets to stderr as they are sent (although most of the content is non-ascii). Use the following environment variable for that:

```
export GLOBUS_USAGE_DEBUG=MESSAGES
```

Also, please see our policy statement[1] on the collection of usage statistics.

[1] http://www.globus.org/toolkit/docs/4.0/Usage_Stats.html

Chapter 4. GT 4.0 C WS Core : Developer's Guide

1. Introduction

The C WS-Core developer's guide provides information related to writing and running web services and WSRF-enabled services in C. It includes tutorials walking the developer through creation of services, and clients to interact with services. It includes scenarios for possible configurations that the developer may want. It also provides references to APIs and their documentation.

2. Before you begin

2.1. Feature summary

Binding Generation:

- Binding Generation directly from WSDL schemas
  - ANSI-C stubs and skeletons
  - Non-blocking client stubs for writing event-driven code
  - EPR (EndpointReference) encapsulation
  - WSRF enabled client stubs and services
- HTTP/1.1 Support
- Embeddable Service API
- Standalone service container
- WSRF-enabled services

Deprecated Features

- Dynamic Deployment (WSDD) using AxisC++ was included in an early pre-release but is no longer supported.

2.2. Tested platforms

Tested Platforms for C WS Core

- IA32/Linux/gcc32
- IA64/Linux/gcc64
- x86_64/Linux/gcc64
- SPARC/Solaris 9/vendorcc32
- PowerPC/AIX 5.2/vendorcc32

- Mac/OS X/gcc32

2.3. Backward compatibility summary

Protocol changes since GT version 3.2

- SOAP messages conform to WSRF schemas instead of previous OGSI/OGSA schemas.
- WS-Addressing has been added to the list of supported standards, as defined by the WS-Resource Framework.
- HTTP/1.1 with 'chunked' transfer encoding is used by default.

API changes since GT version 3.2

- The 3.2 cbindings API is obsolete, with no overlap to the new API. Bindings APIs are now generated directly from WSDL.
- The underlying XML/SOAP messaging framework is also new, based on the libxml2 pull parser API.

Schema changes since GT version 3.2

- Schemas are completely new. The WS C Core implements the OASIS WSRF and WSN working drafts specifications (with minor fixes to the 1.2-draft-01 published schemas and with the March 2004 version of the WS-Addressing specification.)

2.4. Technology dependencies

C WS Core depends on the following GT components:

- C Common Libraries
- Pre-WS Authentication and Authorization (GSI)
- Globus XIO[1] (used by C WS core for efficient HTTP and TCP transport)

C WS Core depends on the following 3rd party software:

- Libxml2[2] (used by C WS Core for SOAP XML parsing and WSDL parsing)
- OpenSSL[3] (used by C WS Core for Security)
- JavaScript[4] (used by C WS Core as a template language to generate the C bindings from WSDL schemas)

2.5. Security considerations

C WS-Core supports secure transport (https) and secure message (just X509 signing, not encryption).

[1] http://www.globus.org/toolkit/docs/4.0/common/xio/
[2] http://www.xmlsoft.org/
[3] http://www.openssl.org
[4] http://www.mozilla.org

2.5.1. Secure Transport

With secure transport, the entire container must be run over an https transport. This is done by default for the C container. If the user does not want security in the container, or wants to use secure message instead of secure transport, they should use the `-nosec` argument to `globus-wsc-container`.

For clients, the secure transport is enabled if the contact URI contains the 'https' scheme instead of 'http', so the client doesn't have to enable or disable it explicitly.

3. Architecture and design overview

- Mapping WSDL to C Bindings[5] - describes how to use the C bindings generated from WSDL and XML schema.
- Design of Web Services Architecture in C[6]
- Design of WSRF in C[7]

4. Public interface

The semantics and syntax of the APIs and WSDL for the component, along with descriptions of domain-specific structured interface data, can be found in the Chapter 6, *GT 4.0 Component Guide to Public Interfaces: C WS Core*.

5. Usage scenarios

Here we provide some scenarios for using C WS-Core that aren't described in the tutorials.

5.1. Using Wildcards

Both clients and services may need to create or parse instances of `xsd_any` or `xsd_anyType` types. This is necessary when the XML schema defines a type that includes the `xsd_any` or `xsd_anyType` as a type for one of its elements, such as:

```
<xsd:complexType name="TemporalType">
  <xsd:sequence>
    <xsd:any minOccurs="1" maxOccurs="1" processContents="lax" />
  </xsd:sequence>
</xsd:complexType>

<xsd:element name="Temporal" type="tns:TemporalType"/>
```

The content of an instance of `TemporalType` is not restricted by the schema definition, and so must be handled specially at runtime. For serialization and deserialization of wildcard elements, a special global variable of type `globus_xsd_type_info_t` is associated with each type that can be set on the wildcard. For example, if a user wanted an instance of `TemporalType` to contain an instance of an `xsd:dateTime`, the `any` field must be filled in properly. The following bit of C code does this:

[5] WSDLtoCBindings.pdf
[6] C-GT4-WS-Design.pdf
[7] C-GT4-WSRF-Design.pdf

```
time_t current;
TemporalType temp;
xsd_QName * element;
xsd_dateTime * time; /* this is just a struct tm */

...

result = TemporalType_init_contents(&temp);
/* check result */

temp.any.any_info = &xsd_dateTime_info;

result = xsd_dateTime_init(&time);
/* check result */

current = time(NULL);

/* get the current time */
result = xsd_dateTime_copy_contents(
    time,
    (xsd_dateTime *)localtime(&current));
/* check result */

temp.any.value = (void *)time;

result  = xsd_QName_init(&element);
/* check result */

element->Namespace = globus_libc_strdup("http://temporal.com");
element->local = globus_libc_strdup("Time");

temp.any.element = element;

/* now we can serialize it */

result = TemporalType_serialize(
    &Temporal_qname,
    temp,
    handle,
    0);
/* check result */
```

This serializes the `TemporalType` to the contain the current timestamp. The resulting serialized elements would look like this:

```
<time:Temporal xmlns:time="http://temporal.com">
<time:Time>Mon Apr 17 10:14:22 CDT 2005</time:Time>
</time:Temporal>
```

If we want to serialize it to a string of the current day of the week, we would do this:

```
time_t current;
TemporalType temp;
xsd_QName * element;
xsd_string * day; /* this is just a pointer to char * */

...

result = TemporalType_init_contents(&temp);
/* check result */

temp.any.any_info = &xsd_string_info;

result = xsd_string_init_cstr(&day, "Monday");
/* check result */

temp.any.value = (void *)day;

result  = xsd_QName_init(&element);
/* check result */

element->Namespace = globus_libc_strdup("http://temporal.com");
element->local = globus_libc_strdup("Day");

temp.any.element = element;

/* now we can serialize it */

result = TemporalType_serialize(
    &Temporal_qname,
    temp,
    handle,
    0);
/* check result */
```

This allows us to serialize the temporal time element as the day of the week. The resulting serialized elements for this code would look like this:

```
<time:Temporal xmlns:time="http://temporal.com">
<time:Day>Monday</time:Day>
</time:Temporal>
```

So this allows us to inject types into wildcard elements at runtime, and demonstrates how to serialize those wildcards.

For deserialization of wildcard types, a registry is used to lookup the actual type of the element based on QName or the xsi:type attribute. The registry contains key/value pairs of QName to globus_xsd_type_info_t structures. These structures contain the appropriate information about deserializing the type.

5.2. Using Asynchronous client stubs

A client may wish to perform many invocations of resource property requests to different services (or the same service) at once, without waiting for the response from one request before starting a second request. The asynchronous client

stubs generated for each operation allow the client to do this. The example code below shows the implementation of the callback that gets called once the response from a resource property has been received for the CounterService.

```
typedef struct
{
    globus_cond_t cond;
    globus_mutex_t mutex;
} counter_monitor;

void
get_rp_counter_value_callback(
    CounterService_client_handle_t      handle,
    void *                              user_args,
    globus_result_t                     result,
    const wsrp_GetResourcePropertyResponseType *
                                        GetResourcePropertyResponse,
    CounterPortType_GetResourceProperty_fault_t
                                        fault_type,
    const xsd_any *                     fault)
{
    counter_monitor_t * monitor = (user_args);
    xsd_int * rp_value;

    if(GetResourcePropertyResponse->any.elements[0].any_info !=
       (&Value_rp_info))
    {
        /* error - expected Value as the first (and only) resource
         * property
         */
    }

    rp_value = (xsd_int *)GetResourcePropertyResponse->any.elements[0].value;

    globus_mutex_lock(&monitor->mutex);
    monitor->value = *rp_value;
    monitor->done = 1;
    globus_cond_signal(&monitor->cond);
    globus_mutex_unlock(&monitor->mutex);
}

...

counter_monitor_t * monitor;

monitor = globus_malloc(sizeof(counter_monitor_t));
/* check OOM */

globus_cond_init(&monitor->cond, NULL);
globus_mutex_init(&monitor->mutex, NULL);
monitor->done = 0;
monitor->value = 0;
```

```
result = CounterPortType_GetResourceProperty_epr_register(
    client_handle,
    createCounterResponse->EndpointReference,
    &Value_rp_qname,
    get_rp_counter_value_callback,
    prop_monitor);
if(result != GLOBUS_SUCCESS)
{
   ...
}

/* do other processing */

globus_mutex_lock(&monitor->mutex);
while(!monitor->done)
{
    globus_cond_wait(&monitor->cond, &monitor->mutex);

    /* do other processing */
}
globus_mutex_unlock(&monitor->mutex);
```

This allows us to do other processing while the GetResourceProperty operation is invoked, and the response is returned. For something as simple as the CounterService, the wait for the callback to be called will most likely be short (unless there is network delay). For more complex services, the delay may be longer, and the client may want to perform other processing instead of just waiting.

6. Tutorials

6.1. Writing Clients for the BlogService

6.1.1. Introduction: A Blog Service

The Globus Toolkit C WS-Core codebase provides tools and APIs for interacting with web services from a client written in C. It provides additional support for interacting with resource enabled (WSRF) web services. This tutorial provides a walkthrough of the steps to take to create such a C client.

The client we implement interacts with the BlogService, which is a simple example of a Blog web service. See the Wikipedia entry on Blogs[8] for more information on Blogging. In our simple example, the topic for each Blog is maintained as a *WS-Resource*. The primary ResourceProperty type associated with each Blog resource is an array of strings of all the entries made to that blog topic.

Clients can create new Blog resources with the *createBlogTopic* factory operation, and append their own entries to that resource with the *addEntry* operation. Because the blog stores the entries beyond the lifetime of a single web service invocation (such as *addEntry*), maintaining each blog topic as a resource is a natural use of the framework.

The public interface to a Blog's entry strings is through the *resource property* named `BlogEntry`, and the resource property operations (i.e. *GetResourceProperty*) that are inherited by the BlogService.

[8] http://en.wikipedia.org/w/wiki.phtml?title=weblog

The tutorial walks through creation of a blog resource, invoking the *addEntry* operation on that resource, accessing the blog's entries, and finally destroying the blog resource.

For this tutorial we provide the following:

- Complete source for the clients:
  - create_blog.c[9]
  - add_blog_entry.c[10]
  - get_blog_entries.c[11]
  - destroy_blog.c[12]
- WSDL schemas:
  - blog.wsdl[13] - Includes the input/output type definitions for the BlogService operations, the ResourceProperty definitions, and the portType definition.
  - blog_bindings.wsdl[14] - Includes the binding definition for the BlogService. The `blog.wsdl` schema file is imported.
  - blog_service.wsdl[15] - Includes the service definition. The `blog_binding.wsdl` schema file is imported.
- A GPT package blog_client_bindings-0.2.tar.gz[16] of the blog bindings source code. This is the package generated from the WSDL schemas using the `globus-wsrf-cgen` command.
- A tarball blog_client.tar.gz[17] of the counter client source and Makefiles described in this tutorial.

See: Section 6.2, “Implementing a Blog Service” for further information on the service side of the implementation. Here, we provide the steps for creating the C client:

6.1.2. Step 0: Acquire a WSDL schema

This is the first step to writing your own client. You must either obtain a pre-existing WSDL schema file (or files), or you must write your own. If you are just going to write a client that interacts with a pre-existing service, the WSDL schema for that service already exists, and you should be able to obtain it from the service author.

For the BlogService, we provide blog.wsdl[18] that defines the factory operation `createBlogTopic`, the append operation `addEntry`, and the `BlogEntry` resource property for each blog resource. The WSDL schema files should be installed somewhere in the $GLOBUS_LOCATION/share/schema tree. In the case of the blog WSDL, you need to install them into $GLOBUS_LOCATION/share/schema/tutorials/blog/. This allows the relative paths in the schema import declarations to work.

[9] developer/tutorials/blog/client/create_blog.c
[10] developer/tutorials/blog/client/add_blog_entry.c
[11] developer/tutorials/blog/client/get_blog_entries.c
[12] developer/tutorials/blog/client/destroy_blog.c
[13] developer/tutorials/blog/blog.wsdl
[14] developer/tutorials/blog/blog_bindings.wsdl
[15] developer/tutorials/blog/blog_service.wsdl
[16] developer/tutorials/blog/client/blog_client_bindings-0.2.tar.gz
[17] developer/tutorials/blog/client/blog_client.tar.gz
[18] developer/tutorials/blog/blog.wsdl

6.1.3. Step 1: Generate Client Bindings

Once you have the WSDL schema(s) for the service, you need to generate the client bindings from that schema. This will provide the C types and functions (bindings) you need to use to interact with the service. The command used to generate the bindings is `globus-wsrf-cgen`.

To run this command on the blog schema files, they must be placed in `$GLOBUS_LOCATION/share/schema/tutorials/blog/`, as described in Step 0. The command for generating the blog client bindings looks like this:

```
$GLOBUS_LOCATION/bin/globus-wsrf-cgen -no-service -s blog_client \
    -flavor <flavor> -d $PWD/bindings \
    $GLOBUS_LOCATION/share/schema/tutorials/blog/blog_service.wsdl
```

This command will generate the GPT package listed above. The package can be built and installed using the following command:

```
$GLOBUS_LOCATION/sbin/gpt-build bindings/blog_client_bindings-0.2.tar.gz <flavor>
```

6.1.4. Step 2: Write the Client

In order to write a C WS-Core client, the following steps should be followed in general:

6.1.4.1. Include the Client Header

The client bindings generated from Section 6.1.3, “Step 1: Generate Client Bindings” include a client header which provides the necessary function declarations to perform the client invocations we need to make. In the case of the BlogService, the BlogService_client.h[19] is the header we need, so it gets included at the top of the file:

```
#include "BlogService_client.h"
```

6.1.4.2. Module Activation

The first step of the client is to activate the module defined for the client. Module activation is a pattern used frequently in the Globus Toolkit. It provides initialization and setup for a particular library, and the libraries it depends on. In this case, the module we are activating is the `BLOGSERVICE_MODULE`, defined in `BlogService_client.h`[20], as follows:

```
globus_module_activate(BLOGSERVICE_MODULE);
```

6.1.4.3. Client Handle Init

Once the module is activated, the client handle must be initialized:

```
BlogService_client_handle_t  blog_handle;

...
```

[19] developer/tutorials/blog/client/BlogService_client.h
[20] developer/tutorials/blog/client/BlogService_client.h

```
result = BlogService_client_init(
    &blog_handle,
    NULL, NULL);
```

This handle provides abstraction for messaging and transport configuration parameters, and is used by all Blog service invocations. The second and third parameters are attrs and handler chains that determine how the message is serialized and transported. In this example, we use the default configuration, so the parameters are `NULL`.

In some scenarios, attrs and handlers will need to be setup explicitly by the user.

6.1.4.4. Creating a Resource

Once the client handle is initialized, the next step is to create the blog resource in the BlogService. The create_blog.c[21] performs resource creation by invoking the *createBlogTopic* factory operation. The bindings call from the client looks like this:

```
    createBlogTopicType                 createBlogTopic;
    createBlogTopicResponseType *       createBlogTopicResponse;
    Blog_createBlogTopic_fault_t        create_fault_type;
    xsd_any *                           fault;

    ...

    createBlogTopic.Topic = "Emacs vs. vi: Which is better?";
    createBlogTopic.Creator = "slang";

    result = Blog_createBlogTopic(
        blog_handle,
        "http://the.service.host:8080/wsrf/services/BlogService",
        &createBlogTopic,
        &createBlogTopicResponse,
        &create_fault_type,
        &fault);
```

This is a code of the createBlogTopic invocation, similar to what's in the create_blog.c[22] example. The `Blog_createBlogTopic` function is defined in BlogService_client.h[23]. The parameters are the initialized blog handle, the endpoint URI to the BlogService (i.e. *"http://the.service.host:8080/wsrf/services/BlogService"*), the input and output parameters, and the fault parameters. In this particular example, the `createBlogTopic` input parameter holds the topic for the blog, and the creator of the blog. The `createBlogTopicResponse` output parameter is filled in by the function call, with the EndpointReference of the resource created by the *createBlogTopic* invocation. In our example code, we export the EndpointFeference to a file, which allows us to access it after the createBlogTopic process has completed.

```
    globus_soap_message_handle_t        epr_out_handle;

...

    result = globus_soap_message_handle_init_to_file(
```

[21] developer/tutorials/blog/client/create_blog.c
[22] developer/tutorials/blog/client/create_blog.c
[23] developer/tutorials/blog/client/BlogService_client.h

```
        &epr_out_handle,
        "emacs_vi_epr.xml",
        GLOBUS_XIO_FILE_CREAT);

...

    result = wsa_EndpointReferenceType_serialize(
        &BlogEPR_qname,
        &createBlogTopicResponse->EndpointReference,
        epr_out_handle,
        0);
...

    globus_soap_message_handle_destroy(epr_out_handle);
```

Now we must destroy the response from *createBlogTopic* invocation:

```
createBlogTopicResponse_destroy(createBlogTopicResponse);
```

6.1.4.5. Invoking a Resource Operation

Once the EndpointReference has been written to file, we have a reference to the blog resource, so we can call the `addEntry` operation on that resource from another process. This is what the add_blog_entry.c[24] client example does. The EndpointReference for the blog resource is first imported from the file:

```
    globus_soap_message_handle_t        epr_in_handle;

...

    result = globus_soap_message_handle_init_from_file(
        &epr_in_handle,
        "emacs_vi_epr.xml");

...

    result = wsa_EndpointReferenceType_init(&blog_resource_reference);

...

    result = wsa_EndpointReferenceType_deserialize(
        &BlogEPR_qname,
        blog_resource_reference,
        epr_in_handle,
        0);

...

    globus_soap_message_handle_destroy(epr_in_handle);
```

[24] developer/tutorials/blog/client/add_blog_entry.c

Once the EndpointReference is imported, the addEntry operation is invoked as follows:

```
    addEntryType                        entry;
    addEntryResponseType *              blog_entries;

    Blog_addEntry_fault_t               add_fault_type;
    xsd_any *                             fault;

    entry.Comment = "What's vi??";
    entry.Author = "EmacsPowerUser";

    result = Blog_addEntry_epr(
        blog_handle,
        blog_resource_reference,
        &entry,
        &blog_entries,
        &add_fault_type,
        &fault);
```

For this invocation, we're using the `Blog_addEntry_epr` function (instead of `Blog_addEntry`). This allows us to pass in the EndpointReference of the resource directly as the second parameter (that's why the function ends in `_epr`). The first parameter is the client handle, The third and fourth parameters are the input and output parameters to the operation (the blog entry to add, and the resulting entries on the blog), followed by the fault parameters. Once this function call returns successfully, the `addEntryResponse` parameter will contain all the entries made to the blog. This call can be made subsequently and entries will continue to be appended to the resource. Once the response is no longer needed after a call to `Blog_addEntry_epr`, we must destroy it:

```
xsd_string_destroy(addEntryResponse);
```

The output of running `add-blog-entry` will look something like this:

```
./add-blog-entry emacs_vi_blog.xml "Emacs rocks!" anonymous

BLOG ENTRIES:

On Wed Dec 22 04:57:42 CST 2004, anonymous said: "Emacs rocks!"

On Tue Oct 26 01:01:11 CST 2004, wq said: "CTRL-ALT-SHIFT-X CTRL-C...I'm running out of fi

On Thu Aug 12 10:44:32 CST 2004, EmacsPowerUser said: "What's vi??"
```

6.1.4.6. Getting a Resource Property Value

The WSDL schema for the BlogService defines a Resource Property *BlogEntry* as part of the resource property document for the Blog port type. This resource property allows us to access the state of the resource (get the entries) with the `GetResourceProperty` operation defined in the WS-ResourceProperties schema and inherited by the Blog portType. The get_blog_entries.c[25] client example performs this operation on the Blog resource. The invocation is made as follows:

[25] developer/tutorials/blog/client/get_blog_entries.c

```
#include "BlogEntry.h"

...

    wsrp_GetResourcePropertyResponseType *      RPResponse;
    Blog_GetResourceProperty_fault_t            getrp_fault_type;
    xsd_any *                                   fault;

    ...

    result = Blog_GetResourceProperty_epr(
        blog_handle,
        blog_resource_reference,
        &BlogEntry_qname,
        &RPResponse,
        &getrp_fault_type,
        &fault);
```

In this function call, the client handle and endpoint reference are passed as the first two parameters. The third parameter (the operation input) is the qualified name of the Resource Property. In this case, the QName is declared in the generated header *BlogEntry.h* as the global variable `BlogEntry_qname`. The output parameter `RPResponse` is the response from the `GetResourceProperty` operation. On successful completion of the function, this response parameter will contain the value(s) of the ResourceProperty. Because resource properties can have any type, the response is deserialized as an array of `xsd_any *` instances. In order to access the actual value from this structure, the type of the `xsd_any *` instance must be verified to match the expected type:

```
    if(RPResponse->any.elements[i].any_info->type !=
       (&BlogEntry_qname) &&
       (RPResponse->any.elements[i].any_info->type !=
       (&Blog_BlogEntry_rp_qname))
    {
        /* error!  Unexpected type */
    }
```

What's happening here? The `wsrp_GetResourcePropertyResponseType` structure contains the field `any`, which is an `xsd_any_array`. This array is assumed to contain one element at index 0. In order to check that the element was deserialized as the appropriate element (i.e. *BlogEntry*), we must compare the `any_info` field against the reference to the global variable `BlogEntry_qname` declared in *BlogEntry.h*.

Once the type of the element in the response is verified, we can access the value contained in the `value` field of the `xsd_any`.

```
    blog_entry = *RPResponse->any.elements[i].value;

    printf("BLOG ENTRIES:\n\n%s\n", blog_comments);
```

After the value of the resource property has been accessed, we need to destroy the response instance created by the `Blog_GetResourceProperty_epr` function call:

```
wsrp_GetResourcePropertyResponseType_destroy(RPResponse);
```

The output of running `get-blog-entries` will look something like this:

```
./get-blog-entries emacs_vi_blog.xml

BLOG ENTRIES:

On Wed Dec 22 04:57:42 CST 2004, anonymous said: "Emacs rocks!"

On Tue Oct 26 01:01:11 CST 2004, wq said: "CTRL-ALT-SHIFT-X CTRL-C...I'm running out of fi

On Thu Aug 12 10:44:32 CST 2004, EmacsPowerUser said: "What's vi??"
```

6.1.4.7. Destroy the Resource

In order to destroy the resource we've created after all our invocations to it are complete, we use the `Destroy` operation defined in WS-ResourceLifetime schema and inherited by the Blog portType. The destroy_blog.c[26] client is an example of using this operation for the blog resource. The example imports the resource reference, calls the `Destroy` operation, and then removes the file that referenced the resource.

```
    wsrl_DestroyType                    Destroy;
    wsrl_DestroyResponseType *          DestroyResponse;
    Blog_Destroy_fault_t                destroy_fault_type;
    xsd_any *                           fault;

    result = globus_wsrf_core_import_endpoint_reference(
        "emacs_vi_blog.xml", &blog_resource_reference, NULL);

    ...

    result = Blog_Destroy_epr(
        blog_handle,
        blog_resource_reference,
        &Destroy,
        &DestroyResponse,
        &destroy_fault_type,
        &fault);
```

As with the previous EndpointReference invocations, the first two parameters passed to this function are the client handle and the endpoint reference to the resource. In the case of invoking the `Destroy` operation, the `Destroy` and `DestroyResponse` input and output parameters are just empty structures and don't contain any pertinent information. Nevertheless, the `DestroyResponse` variable should be cleaned up after the `Destroy` operation has completed:

```
    wsrl_DestroyResponse_destroy(DestroyResponse);
```

6.1.4.8. Cleanup

Once all the desired invocations have completed for a particular process, the client handle needs to be destroyed, and the module must be deactivated.

[26] developer/tutorials/blog/client/destroy_blog.c

```
    Blog_client_handle_destroy(blog_handle);

    globus_module_deactivate(BLOGSERVICE_MODULE);
```

These calls exist in each of the client examples.

6.1.5. Step 3: Build the Client

Now you've written an end-to-end C WS-Core WSRF-enabled client. In order to compile the client we demonstrate how to write a Makefile for it. First, the following command must be run:

```
$GLOBUS_LOCATION/bin/globus-makefile-header \
    --flavor=<flavor> <package> \
    > MyMakefile.include
```

Assuming you compiled the Globus Toolkit with a `gcc32dbg` flavor, and using the blog client bindings package from this tutorial, the command would be:

```
$GLOBUS_LOCATION/bin/globus-makefile-header \
    --flavor=gcc32dbg blog_client_bindings \
    > BlogClientMakefile.include
```

The resulting `BlogClientMakefile.include` [27] contains include and link definitions for our client. Now we just need to write a Makefile, using the variables defined in the output of the `globus-makefile-header` command. We've provided a blog client Makefile[28]. Once your Makefile is written, running `make` will generate the client executables. At this point you're not quite ready to run it. The client needs to have a service running somewhere to interact with. See Section 6.2, “Implementing a Blog Service” in order to create and run a BlogService that you can invoke with your new client.

6.2. Implementing a Blog Service

6.2.1. Introduction: A Blog Service

The Globus Toolkit's C WS-Core component provides tools and APIs for creating web services in C. It also provides additional support for creating web services which are WSRF-enabled, meaning the service can manage resources and the associated resource properties. This tutorial provides a walkthrough of the steps needed to create a WSRF-enabled service in C, from defining a WSDL schema for the service to actually running the service in the C service container.

The service we implement in this tutorial is the BlogService, which is a simple service that allows new Blog topics to be created as resources, and then allows comments to be added to a particular Blog topic. See the Blog Wikipedia entry[29] for more information on Blogs.

In our BlogService, the primary resource property is the `BlogEntry` element, which is an array of BlogEntryType instances containing the comment, author, and timestamp of each entry posted to the Blog topic. For the tutorial, we will demonstrate how to generate the service stubs and skeletons for the BlogService, and how to provide the service implementation, including creation of new Blog topics as resources, and adding new blog entries to the BlogEntry Resource Property. For the purposes of this tutorial, we provide the following:

[27] developer/tutorials/blog/client/BlogClientMakefile.include
[28] developer/tutorials/blog/client/Makefile.example
[29] http://en.wikipedia.org/w/wiki.phtml?title=Blog&redirect=no

- WSDL schema files for the BlogService:
  - blog.wsdl[30] - Includes the input/output type definitions for the BlogService operations, the ResourceProperty definitions, and the portType definition.
  - blog_bindings.wsdl[31] - Includes the binding definition for the BlogService. The blog.wsdl schema file is imported.
  - blog_service.wsdl[32] - Includes the service definition. The blog_binding.wsdl schema file is imported.
- Source file for the complete BlogService implementation:
  - BlogService_skeleton.c[33]
- A GPT package blog_service_bindings-0.2.tar.gz[34] that contains the complete BlogService implementation (includes the skeleton from the above bullet).

This tutorial defines 5 steps needed to create any WSRF-enabled service using C WS-Core, and then provides example walkthroughs of those steps with the BlogService.

6.2.2. Step 1: Acquiring a WSDL Schema

You must either obtain pre-existing WSDL schema files or write your own. The schema files must contain a service definition that defines the service. Please note that the C WS-Core only supports document/literal style WSDL schema files at present.

For the BlogService, we provide blog.wsdl[35] that defines the factory operation `createBlogTopic` and the append operation `addEntry`, as well as the `BlogEntry` resource property for each blog resource.

6.2.3. Step 2: Generating Service Bindings

Once you have the WSDL schema(s) for the service, you need to generate the service bindings from that schema. This will provide the C skeleton functions for the service implementation. The command used to generate the bindings is `globus-wsrf-cgen`.

To run this command on the Blog schema files, they must be placed in `$GLOBUS_LOCATION/share/schema/tutorials/blog/`, so that the relative import paths are correct. The command for generating the blog service bindings looks like this:

```
$GLOBUS_LOCATION/bin/globus-wsrf-cgen -no-client -s blog_service \
    -d $PWD/bindings -flavor <flavor> \
    $GLOBUS_LOCATION/share/schema/tutorials/blog/blog_service.wsdl
```

This command generates source and header files for the service, and as a final step, creates a GPT package (a `.tar.gz` file) that contains all the source, headers and necessary build files. Building this package is described in Section 6.2.5, "Step 4: Building/Installing the Service Package". The above command generates build files and type bindings files in the bindings directory as a sub-directory of the current directory. Service specific files are output to a sub-directory of bindings named <service name> ($PWD/bindings/<servicename>/). In this example the sub-directory is named `BlogService`.

[30] developer/tutorials/blog/blog.wsdl
[31] developer/tutorials/blog/blog_bindings.wsdl
[32] developer/tutorials/blog/blog_service.wsdl
[33] developer/tutorials/blog/service/BlogService_skeleton.c
[34] developer/tutorials/blog/service/blog_service_bindings-0.2.tar.gz
[35] developer/tutorials/blog/blog.wsdl

The `-d <dir>` argument outputs the generated files to `<dir>`. Use the `-help` argument to get further info.

6.2.4. Step 3: Writing the Service implementation

Once the service binding generation has completed, the service skeleton functions will reside in the <service name>_skeleton.c source file contained in the <service name> directory. This is the file with the operation functions that must be filled in to complete the implementation of the service. For this example, the file we must modify is `BlogService/BlogService_skeleton.c`. This source file includes skeleton functions for each of the operations defined in the blog.wsdl[36] schema file. The two operations that need to be implemented are `createBlogTopic` and `addEntry`. The associated functions in BlogService_skeleton.c[37] are `Blog_createBlogTopic_impl` and `Blog_addEntry_impl`.

6.2.4.1. Creating a Resource

In the WS-ResourceFramework, operations which create new resources and provide us with references to them are called *factories*. In the BlogService, the `createBlogTopic` operation is the factory that creates a new resource (a new Blog topic), and returns a reference to it (as an EnpointReference). This function creates the resource instance, fills in the EndpointReference to be returned, and creates a resource property `BlogEntry` on the resource.

6.2.4.2. The Resource ID

As the first step of creating a resource in our `Blog_createBlogTopic_impl` function, we must acquire a resource ID. The resource ID is an application specific object that acts as a unique identifier for the resource within the service, and gets embedded within the EndpointRerence for the new resource. For C WS-Core, the resource ID must be in the form of a string. In many services, the resource ID is a UUID string, generated by the `globus_uuid_create` function. See the UUID library documentation for further info.

In the case of the BlogService, we assume that no two Blogs created by the same person will have the same topic, so we can join the author and topic strings together as the resource ID for the new resource we are about to create.

```
globus_result_t
Blog_createBlogTopic_impl(
    globus_service_engine_t             engine,
    globus_soap_message_handle_t        message,
    globus_service_descriptor_t *       descriptor,
    createBlogTopicType * createBlogTopic,
    createBlogTopicResponseType * createBlogTopicResponse,
    const char ** fault_name,
    void ** fault)
{
    char *                              resource_id;
    globus_result_t                     result = GLOBUS_SUCCESS;

    GlobusFuncName(Blog_createBlogTopic_impl);
    BlogServiceDebugEnter();

    blog_id = globus_common_create_string(
        "%s#%s", createBlogTopic->Creator, createBlogTopic->Topic);
```

The `blog_id` is then passed to the `globus_resource_create` function, which will create a managed resource and return it in `blog_resource`.

[36] developer/tutorials/blog/blog.wsdl
[37] developer/tutorials/blog/service/BlogService_skeleton.c

```
result = globus_resource_create(
    blog_id,
    &blog_resource);

...

result = BlogServiceInitResource(blog_id);
```

The second call in the code listing above is the service's resource init function, which allows the operation providers to initialize the resource properties of the resource you've just created. For example, the WS-ResourceLifetime operation provider adds `CurrentTime` and `TerminationTime` resource properties to the resource.

The bindings for any service definition will include a `<service name>InitResource([resource id]);` macro which calls the resource initialization functions for each operation provider the service includes.

6.2.4.3. The EndpointReference (EPR)

Once the resource is created the EndpointReference must be created. The first step is to initialize a reference property of the EPR, which will contain the resource ID we just created. The reference property is a field in the `wsa_EndpointReferenceType` type. Since the property can be anything, it is typed to the XSD wildcard `xsd_any`, which we must create an instance of and initialize to contain the appropriate type and value for the reference property.

```
result = xsd_any_init(&reference_property);
reference_property->any_info = &xsd_string_info;

...

result = xsd_QName_init(reference_property->element);

...

reference_property->element->Namespace = globus_libc_strdup(
    BlogService_service_qname.Namespace);
reference_property->element->local = globus_libc_strdup("BlogID");

result = xsd_string_copy_cstr(
    (xsd_string **)&reference_property->value,
    blog_id);
```

The `xsd_any` type we initialize has 3 important fields. The `any_info` field contains the type information used by the marshalling engine to determine how to serialize the reference property. In this case the reference property is just a string, so we set the `any_info` field to the globally defined `xsd_string_info` variable. For more information on using wildcards in your service implementation, see Section 5, “Usage scenarios ”.

The `element` field in `xsd_any` is a *QName* of the element to define for serializing the type. In the BlogService case, we set the element to *http://globus.org/blog#BlogID*. The other field we need to set in the reference property is the `value`, which is a `(void *)`, set to the pointer of the instance of the resource id (in this case the blog id string). We use the `xsd_string_copy_cstr` function to actually copy the contents of the string to the `value` field.

Once the reference property has been initialized, we can create the EndpointReference. The `globus_wsrf_core_create_endpoint_reference` convenience function has been provided to create the endpoint reference.

```
result = globus_wsrf_core_create_endpoint_reference(
    engine,
    BLOGSERVICE_BASE_PATH,
    &reference_property,
    &createBlogTopicResponse->EndpointReference);
```

This call takes the `engine` passed into the skeleton function, the base path of the URI for the service (each service has a `<service name>_BASE_PATH` variable defined), and the reference property we just initialized. The resulting EndpointReference must be set to the `EndpointReference` field in the `createBlogTopicResponse` variable passed into the skeleton function.

6.2.4.4. The Resource Property

As the last step of the `Blog_createBlogTopic_impl` function, we set the BlogEntry resource property of the resource. Since the Blog initially doesn't contain any entries, we set the resource property to an empty array. We will add new entries to this resource property in the `Blog_addEntry_impl` skeleton function.

```
result = BlogEntryType_array_init(&blog_entries);

...

result = globus_resource_create_property(
    blog_resource,
    &Blog_BlogEntry_rp_qname,
    &BlogEntry_array_info,
    blog_entries);
```

The arguments passed to this function are the created resource, the QName of the resource property (in this case, *BlogEntry*), the info variable of the resource property type to create, and the empty blog array instance. See the Resource API[38] for further documentation.

6.2.4.5. Add an Entry to the Blog Topic

Once a resource has been created, clients will invoke the `addEntry` operation to add new entries to the blog. The implementation of the `Blog_addEntry_impl` adds the new entry to the blog topic.

6.2.4.6. Access the Resource

The resource is accessed through the EndpointReference contained in the message. The utility function `globus_wsrf_core_get_resource` is used to access the resource. The EndpointReference is accessed through the first parameter (`message`) passed to the function.

```
result = globus_wsrf_core_get_resource(
    message,
    descriptor,
    &blog_resource);
```

Information about how the resource ID is accessed from the EndpointReference is maintained by the service descriptor, so this gets passed in as the second parameter (`service`).

[38] http://www.globus.org/api/c-globus-4.0/globus_c_wsrf_resource/html/index.html

6.2.4.7. Get the Resource Property

Once we have the resource we can access the BlogEntry resource property using the `globus_resource_get_property` function.

```
result = globus_resource_get_property(
    resource,
    &Blog_BlogEntry_rp_qname,
    (void **)&blog_entries,
    NULL);
```

The first parameter is the blog resource we just accessed, the second parameter is the QName of the BlogEntry resource property. `Blog_BlogEntry_rp_qname` is a global variable declared in `BlogService.h`. Global QName variables exist for each resource property in a service. The third parameter is the array of blog entries we want to get. The last parameter is the type info structure of the resource property we're accessing. Since we know the type of the resource property, we can just set this to `NULL`.

6.2.4.8. Add the Blog Entry

Now that we have the array of blog entries, we need to add a new element to the end of it with the values of the entry. Each array type generated from an XML schema document has an associated `_array_push` function, which creates a new instance of the type and adds it to the end of the array, returning the new instance. In this case, we create a new entry at the end of the array with the `BlogEntryType_array_push` function.

```
new_entry = BlogEntryType_array_push(blog_entries);
```

Now we need to fill in this entry with the values passed into the skeleton function.

```
tstamp = time(NULL)
result = xsd_dateTime_copy_contents(
    &new_entry->Timestamp,
    (xsd_dateTime *)localtime(&tstamp));

...

result = xsd_string_copy_contents(
    &new_entry->Author,
    (xsd_string *)&addEntry->Author);

...

result = xsd_string_copy_contents(
    &new_entry->Comment,
    (xsd_string *)&addEntry->Comment);
```

These functions copy the entry's comment and author from the input parameter to the new entry instance we've created. The timestamp of the entry is set to the current local time. This completes the addition of a resource property value to the resource property maintained by the resource instance.

The `addEntry` operation expects as the response a list of the entries in the Blog. Since this is just the array of blog entries that we just added to, we can simply copy this array to the response output parameter:

```
result = BlogEntryType_array_copy_contents(
    &addEntryResponse->BlogEntries,
    blog_entries);
```

6.2.4.9. Resource Finish

As a last step of the `Blog_addEntry_impl` function, we need to release the blog resource we accessed in the first step. This allows the resource management computeroutput to handle locking and reference counting for the resource.

```
globus_resource_finish(blog_resource);
```

6.2.4.10. Other Issues

In this section we describe other parts of implementing the skeleton functions that might be of interest.

6.2.4.11. Service Initialization

Besides the skeleton functions defined for each operation in a service, `BlogService_skeleton.c` also contains functions for initializing and finalizing the BlogService. The `BlogService_init` function should contain any service specific computeroutput that needs to be run when the service is loaded, and the `BlogService_finalize` function should contain computeroutput that needs to be run when the service is unloaded (presumably cleanup from `BlogService_init`). These functions most likely can remain empty no-ops, but if for example you want a service to have persistent resources which exist throughout the lifetime of the service, they should be created in the service's `init` function and destroyed in the `finalize` function.

6.2.4.12. Error Handling

Almost all of the function calls in our BlogService return a `globus_result_t` type. The `globus_result_t` informs the caller of the success or failure of the function call, and is used to reference the error object created if an the function call failed. The standard practice in the Globus Toolkit for handling errors is to check the return value of the function:

```
if(result != GLOBUS_SUCCESS)
```

and if an error occurred, either chain the error or handle the error at that level (exit the process, print an error message, etc.). The skeleton functions we've implemented in this tutorial have a `globus_result_t` return value, so the skeleton function needs to create and return error values if and when they occur within the service implementation. The bindings generated for a service include macros for each operation in the service's header file that create `globus_result_t` error values to be returned by the skeleton function. For example, the signatures of the macros generated for the `addEntry` operation are:

```
globus_result_t
Blog_addEntry_error(const char *);

globus_result_t
Blog_addEntry_chain_error(globus_result_t, const char *);
```

In general, each operation will have an associated error create function that takes a string and returns a `globus_result_t` error as well as an error function that takes a base error `globus_result_t` and a string and returns a new `globus_result_t`.

The first function macro listed is useful for error cases where the error is the primary base cause, while the second function is useful when another globus function has been called and value which is not equal to `GLOBUS_SUCCESS`.

6.2.4.13. Operation Providers

For the operations inherited from the WSRF schemas (*GetResourceProperty*, *Destroy*, *SetTerminationTime*), their implementation has already been provided for us. This is achieved using operation providers, which replace the functions defined in the `BlogService_skeleton.c` source file with generic pre-defined versions of those functions when the service is loaded by the container. Even though the contents of those functions remain empty in the skeleton source file, they don't get used, so they can be safely ignored.

6.2.4.14. Service-Side Notifications

`BlogService_skeleton.c` also includes functions for the *Subscribe* and *GetCurrentMessage* operations that are part of the *WS-BaseN* schema (inherited by the BlogService), but the C WS-Core currently doesn't provide implementations of *NotificationProducer* or *SubscriptionManager* at present, so these skeleton functions can remain empty as well.

6.2.5. Step 4: Building/Installing the Service Package

6.2.5.1. Packaging

Once the service implementation is complete, the service package can be re-packaged (create the tarball) with the implemented computeroutput using `make`. Change the working directory to the directory the bindings were generated in, and run:

```
make dist
```

This will create (or re-create) the `blog_service_bindings-0.2.tar.gz` package in that directory with the new service implementation. This package can be distributed to any machine with a C WS-Core installation and installed there.

6.2.5.2. Building

To build the package you just created, run the following command:

```
$GPT_LOCATION/sbin/gpt-build blog_service_bindings-0.2.tar.gz <flavor>
```

This will compile the source files for the types and service and build them into a library module named `libblog_service_bindings.so` (the suffix of the library may differ depending on the platform). The header files are installed into `$GLOBUS_LOCATION/include/<flavor>` and the library is installed in `$GLOBUS_LOCATION/lib/<service base path>`.

6.2.6. Step 5: Running the Service Container

Once the BlogService library module has been installed, the service container can be run and the BlogService can be invoked, causing execution of the service implementation. The service container is run with the command:

```
$GLOBUS_LOCATION/bin/globus-wsc-container
```

6.2.7. Step 6: Debugging the Service Implementation

6.2.7.1. Adding Debug Statements

Each service module includes debugging macros that allow the service developer to print debug statements in a configurable way. The debug statements can have different levels of severity, and are controlled by environment variables. Debug statements are only printed when the service module is compiled with a debug flavor (such as gcc32dbg).

The macro declaration for printing a debug statement in the service skeleton is:

```
<service>DebugPrintf(LEVEL, MESSAGE);
```

Where `LEVEL` is one of:

- `<SERVICE>_INFO`
- `<SERVICE>_DEBUG`
- `<SERVICE>_TRACE`
- `<SERVICE>_WARN`
- `<SERVICE>_ERROR`

The `MESSAGE` parameter consists of the debug message to be printed. It must contain parentheses () around the actual message. Inside the parentheses can be a format string, and a variable number of arguments (like printf). For example, in the BlogService's `addEntry` skeleton implementation (`Blog_addEntry_impl`), the developer may want to see the entry to be added for debugging purposes. The following statement would print the debug message if the DEBUG level was turned on:

```
BlogServiceDebugPrintf(BLOGSERVICE_DEBUG,
                       ("ADD ENTRY:\n\tCOMMENT: %s\n\tAUTHOR: %s\n",
                        addEntry->Comment,
                        addEntry->Author));
```

6.2.7.2. Setting Debug Environment Variables

In order for debug statements to be printed to the terminal, the user must set the appropriate environment variable before running the service container. Each service has a separate debug environment variable that can be set to different debug levels. Optionally, the value of the variable can include a filename to write the debug output to as well.

The environment variable to set for service debugging is:

```
<SERVICE>_DEBUG=<DEBUG LEVEL>
```

This environment variable has five disjoint debug levels that can be set, and match the level definitions used for the debug statement in the previous section. The five levels are:

- `INFO` - general information useful to users of the service.
- `DEBUG` - debug output used by the service skeleton implementor to verify code works.
- `TRACE` - output the entry and exit points of each of the service skeleton functions.

- `WARN` - warn the user that something bad may be happening.
- `ERROR` - output an error for the user to see as it gets returned.

There is also a `ALL` level that will show the debug output for all the levels.

For our BlogService example, if we wanted to see the debug statements at the DEBUG level, then in bash we would set:

```
export BLOGSERVICE_DEBUG=DEBUG
```

If the user wants to see output from multiple debug levels, the levels can be joined together:

```
export BLOGSERVICE_DEBUG="DEBUG|TRACE"
```

7. Debugging

Besides the standard debugging tools available on your platform for C programs, the C WS-Core has a number of environment variables that can be set to debug or trace program execution of the service container. The useful environment variables that can be set are:

- `GLOBUS_SERVICE_ENGINE_DEBUG` - useful for tracing execution of the service engine. The possible values this variable can have are:
  - DEBUG - show debug messages about execution of the engine.
  - INFO - show information regarding service invocations.
  - TRACE - show entry and exit points of functions for the service engine.
  - ERROR - show error occurring during service invocation.
  - ALL - all the above levels joined together.

8. Troubleshooting

This is a new component. If you're having trouble, please let us know!

9. Related Documentation

None at present.

Chapter 5. GT 4.0 Component Fact Sheet: C Web Services Core (C WS Core)

1. Brief component overview

The C WS Core provides a basic toolset in C for creating WSRF-enabled web services and clients conforming to the WS-Resource and WS-Notification specifications.

2. Summary of features

Binding Generation:

- Binding Generation directly from WSDL schemas
  - ANSI-C stubs and skeletons
  - Non-blocking client stubs for writing event-driven code
  - EPR (EndpointReference) encapsulation
  - WSRF enabled client stubs and services
- HTTP/1.1 Support
- Embeddable Service API
- Standalone service container
- WSRF-enabled services

Deprecated Features

- Dynamic Deployment (WSDD) using AxisC++ was included in an early pre-release but is no longer supported.

3. Usability summary

Usability improvements for C WS Core:

C WS-Core is a new component and does not have usability improvements.

4. Backward compatibility summary

Protocol changes since GT version 3.2

- SOAP messages conform to WSRF schemas instead of previous OGSI/OGSA schemas.
- WS-Addressing has been added to the list of supported standards, as defined by the WS-Resource Framework.
- HTTP/1.1 with 'chunked' transfer encoding is used by default.

API changes since GT version 3.2

- The 3.2 cbindings API is obsolete, with no overlap to the new API. Bindings APIs are now generated directly from WSDL.
- The underlying XML/SOAP messaging framework is also new, based on the libxml2 pull parser API.

Schema changes since GT version 3.2

- Schemas are completely new. The WS C Core implements the OASIS WSRF and WSN working drafts specifications (with minor fixes to the 1.2-draft-01 published schemas and with the March 2004 version of the WS-Addressing specification.)

5. Technology dependencies

C WS Core depends on the following GT components:

- C Common Libraries
- Pre-WS Authentication and Authorization (GSI)
- Globus XIO[1] (used by C WS core for efficient HTTP and TCP transport)

C WS Core depends on the following 3rd party software:

- Libxml2[2] (used by C WS Core for SOAP XML parsing and WSDL parsing)
- OpenSSL[3] (used by C WS Core for Security)
- JavaScript[4] (used by C WS Core as a template language to generate the C bindings from WSDL schemas)

6. Tested platforms

Tested Platforms for C WS Core

- IA32/Linux/gcc32
- IA64/Linux/gcc64
- x86_64/Linux/gcc64
- SPARC/Solaris 9/vendorcc32
- PowerPC/AIX 5.2/vendorcc32
- Mac/OS X/gcc32

7. Associated standards

Associated standards for C WS Core:

[1] http://www.globus.org/toolkit/docs/4.0/common/xio/
[2] http://www.xmlsoft.org/
[3] http://www.openssl.org
[4] http://www.mozilla.org

- HTTP
- SOAP
- XML Schema
- WSDL
- WS Security
- WS-Addressing
- WS-Resource Framework
- WS-Notification

8. For More Information

Click here[5] for more information about this component.

[5] index.html

Chapter 6. GT 4.0 Component Guide to Public Interfaces: C WS Core

1. Semantics and syntax of APIs

1.1. Programming Model Overview

The C WS-Core provides interfaces for developers interested in writing web services and clients in C. The primary APIs available to the developer are the C stub bindings generated from WSDL and XSD. These APIs provide the structures and type definitions for each XML Schema type, client stub functions for invoking services, and service skeleton code that allows service writers to fill in the service implementation.

The client stub bindings provide the following:

- Portable ANSI-C API
- Control of message handling and configurable attributes through client handles
- Asynchronous stub functions for non-blocking requests
- EPR encapsulation for easy interaction with resources
- Convenient handling of XSD wildcards

For service writers, the C WS-Core provides service-side skeleton bindings that perform the necessary routing and marshalling for a service operation. The interface to the developer is through the service implementation functions that must be filled in. The service-side programming model includes the ability to load operation providers, which are generic operation implementations that exist over a set of services. This is useful with WSRF, where pseudo-operation inheritance exists. As well, message handling can be controlled at the service implementation level, providing flexibility and control to the service developer.

The C WS-Core provides resource management using the resource API. This is a C API that can be invoked from within C services for creation, access, and control of resources and resource properties.

1.2. Component API

- Resource and Resource Property API[1]: Useful for writing WSRF-enabled services. This API allows resources to be created, accessed, and modified from within a C Web Service implementation.
- Service Engine and Message Attributes[2]: The message attributes provides mechanisms for manipulating runtime parameters of messages. This includes security setup, specific HTTP and WS-Addressing configuration, among others.

  The service engine API is useful for embedding Web Services in C programs. This API allows an application to directly control service invocations and interact with services as they are being invoked. It also provides a convenient API for running a NotificationConsumer service (receiving notifications) from within a client application.

[1] http://www.globus.org/api/c-globus-4.0/globus_c_wsrf_resource/html/index.html
[2] http://www-unix.mcs.anl.gov/~slang/wsrf/c/message/source/doxygen/doc/html/index.html

- Notification Consumer API[3]: Allows creation of NotificationConsumer resource instances from a client API. This API can be used in combination with the Service Engine API to receive notifications.
- WSRF Core Bindings API[4]: These are the types generated from the set of core WSRF schemas. For example, the *wsa_EndpointReferenceType* passed to all EPR stub functions is a generated type from the WS-Addressing schema. The other schemas include:
  - WS-Addressing
  - WS-BaseFaults
  - WS-ResourceProperties
  - WS-ResourceLifetime
  - WS-BaseN
  - WS-ServiceGroup

2. Semantics and syntax of the WSDL

2.1. Protocol overview

The C WS-Core does not provide WSDL interfaces

3. Command-line tools

Please see the C WS Core Command Reference.

4. Overview of Graphical User Interface

There is no support for this type of interface for C WS Core.

5. Semantics and syntax of domain-specific interface

5.1. Interface introduction

The C WS-Core does not provide domain specific interfaces.

6. Configuration interface

6.1. Configuration overview

The C WS-Core component does not provide global configuration functionality.

[3] http://www.globus.org/api/c-globus-4.0/globus_notification_consumer/html/index.html
[4] http://www.globus.org/api/c-globus-4.0/globus_c_wsrf_core_bindings/html/index.html

7. Environment variable interface

The C WS-Core does not provide any user-level component specific environment variables.

Chapter 7. GT 4.0 C WS Core: Quality Profile

1. Test coverage reports

- Test Coverage Reports for C WS Core[1]

2. Code analysis reports

C WS-Core does not have any code analysis reports at this time.

3. Outstanding bugs

- Bug 2310: support for http get queries of WSDL schemas[2]
- Bug 2437: Faults returned from C container[3]
- Bug 2460: utility funcs[4]
- Bug 2911: globus_service_engine_stop hang[5]
- Bug 3018: globus_c_wsrf_core_bindings fails to build on AIX[6]
- Bug 3058: globus_soap_message_handle_init_from_dom() doesn't work[7]
- Bug 3208: C registryService bindings in 4.0[8]

4. Bug Fixes

- globus_wsrf_resource.h missing c++ guards[9]
- globus-wsrf-cgen segfaults on 64bit architectures[10]
- Current trunk AIX failure[11]
- globus_c_wsrf_core_bindings package layout problem[12]
- Rendezvous client bindings generation error on FC3 x86_64[13]

[1] http://www.mcs.anl.gov/~bester/c-ws-core/coverage/
[2] http://bugzilla.globus.org/globus/show_bug.cgi?id=2310
[3] http://bugzilla.globus.org/globus/show_bug.cgi?id=2437
[4] http://bugzilla.globus.org/globus/show_bug.cgi?id=2460
[5] http://bugzilla.globus.org/globus/show_bug.cgi?id=2911
[6] http://bugzilla.globus.org/globus/show_bug.cgi?id=3018
[7] http://bugzilla.globus.org/globus/show_bug.cgi?id=3058
[8] http://bugzilla.globus.org/globus/show_bug.cgi?id=3208
[9] http://bugzilla.globus.org/globus/show_bug.cgi?id=2834
[10] http://bugzilla.globus.org/globus/show_bug.cgi?id=2819
[11] http://bugzilla.globus.org/globus/show_bug.cgi?id=2915
[12] http://bugzilla.globus.org/globus/show_bug.cgi?id=2770
[13] http://bugzilla.globus.org/globus/show_bug.cgi?id=2763

- AIX problem with sed and long command line lengths[14]
- Building globus_c_gram_client_bindings on AIX fails[15]
- Parser Error Building 3.9.4 RC1 on FC2 x86_64[16]
- Core tools uses wrong path[17]
- globus_js does not pick up LDFLAGS[18]
- Core dump when submit job with globusrun-ws[19]
- Missing filelist entries[20]
- AIX build failure in C messaging tests[21]
- globus-wsc-container crash[22]
- Leaks in WS-C Core[23]
- missing error handling in soap read callbacks causes hang[24]
- parsing array with globus_xml_buffer consumes all memory[25]

5. Performance reports

C WS-Core does not have any performance reports at this time.

[14] http://bugzilla.globus.org/globus/show_bug.cgi?id=2928
[15] http://bugzilla.globus.org/globus/show_bug.cgi?id=2580
[16] http://bugzilla.globus.org/globus/show_bug.cgi?id=2446
[17] http://bugzilla.globus.org/globus/show_bug.cgi?id=3004
[18] http://bugzilla.globus.org/globus/show_bug.cgi?id=2543
[19] http://bugzilla.globus.org/globus/show_bug.cgi?id=2867
[20] http://bugzilla.globus.org/globus/show_bug.cgi?id=2617
[21] http://bugzilla.globus.org/globus/show_bug.cgi?id=2952
[22] http://bugzilla.globus.org/globus/show_bug.cgi?id=2610
[23] http://bugzilla.globus.org/globus/show_bug.cgi?id=2619
[24] http://bugzilla.globus.org/globus/show_bug.cgi?id=3153
[25] http://bugzilla.globus.org/globus/show_bug.cgi?id=3154

Chapter 8. GT 4.0 Samples for C WS Core

1. Counter Client

The Counter Client consists of a set of client programs that can be run to interact with the CounterService by creating new counter resources, calling add on those resources, and finally destroying those resources. The reference to each resource (the EPR) is stored in a file.

The sample is a good way to get going fast with C WS-Core client programming, as the user does not have to install/deploy the CounterService—it is installed by default in GT4 containers.

- counter_port_type.wsdl[1] - WSDL document containing the definition of the CounterService operations and types.
- create_count.c[2] - this program invokes the createCounter operation on the CounterService and stores the resulting EPR that points to the new counter resource in a file.
- add_count.c[3] - this program reads the EPR file and invokes the add operation on the resource (of the CounterService) pointed to by the EPR.
- destroy_count.c[4] - this program reads the EPR file and destroys the resource pointed to by the EPR. Once the resource is destroyed, the EPR is no longer valid, so the file is removed.
- Makefile[5] - a Makefile to use for building the counter samples.
- makefile_header[6] - the makefile header generated by globus-makefile-header. This gets included by the Makefile. The user should generate his own makefile_header with the globus-makefile-header command.

[1] developer/tutorials/counter/counter_flattened.wsdl
[2] developer/tutorials/counter/client/create_count.c
[3] developer/tutorials/counter/client/add_count.c
[4] developer/tutorials/counter/client/destroy_count.c
[5] developer/tutorials/counter/client/Makefile.example
[6] developer/tutorials/counter/client/makefile_header

Chapter 9. GT 4.0 Migrating Guide for C WS Core

The following provides available information about migrating from previous versions of the Globus Toolkit.

1. Migrating from GT2

The C WS-Core is a new component of GT4. No migration from GT2 exists.

2. Migrating from GT3

The C WS-Core is a new component of GT4. No migration from GT3 exists.

GT 4.0: C WS Core Command Reference

Name

globus-wsc-container -- Hosts C web services

```
globus-wsc-container
```

Tool description

This command starts the C WS container, allowing WS and WSRF-enabled services to be invoked. *globus-wsc-container* must be running to invoke services written using the C WS core.

Features

- The container can be run in the background with the *-bg* option, and *-pidfile* allows the pid of the process to written to a specified file. This is useful for scripting the command, especially when running tests, or when the container process is expected to have a short lifetime.
- Supports HTTPS by default. In order to turn off HTTPS, use the *-nosec* argument

Limitations

- The C container does not have a shutdown command (the Java container has *globus-stop-container*). To shutdown the C container, you can either CRTL-C the process, or kill the process with the process ID (use *-pidfile*)

Command syntax

Run: *globus-wsc-container -help*

```
Syntax: globus-wsc-container [-help][-max <max sessions>] ...

    Options
    -help, -usage             Displays usage
    -version                  Displays version
    -max      <max sessions>  Max sessions that can be started in parallel
    -port <port>              Set the port of the container
    -pidfile <path>           Write PID of container to this file
    -bg                       Run container in the background
    -nosec                    Don't use https
```

Name

globus-wsrf-cgen -- Generate Stubs/Skeletons in C

`globus-wsrf-cgen`

Tool description

This tool generates C bindings from a set of WSDL schema files. The tool is able to generate client bindings, service bindings, just types, or all three. The WSDL to C mapping document[1] gives more information on how WSDL is mapped to the C programming language.

Command syntax

Run: *globus-wsrf-cgen -help*

```
globus-wsrf-cgen [-help][-s <package name>][-f <flavor>] \
                 [-p <prefix file>][-P <prefix>][-r <relative path>] \
                 [-d <output dir>][-N <namespace>][-n <file>] \
                 [-G <namespace>][-g <file>][-np][-nk][-nf <function>][-ns] \
                 <wsdl schema>

    This command generates client side bindings in C from a WSDL schema file.

    Optional arguments:

     -help, -usage          : displays this message
     -s <package name>      : name used to create the package.  Defaults to
                              the service name from the WSDL schema. This
                              argument is required, unless -no-package is
                              specified.
     -flavor <flavor>
     -fl <flavor>           : Specifies build flavor for the bindings package
                              e.g. gcc32dbg.  This option is required, unless
                              -no-package or -no-tarball is specified.
     -p <file>              : location of the Namespace to Prefix mappings
     -P <prefix map>        : additional Namespace to Prefix mapping specified
                              on the command line.  This argument should be
                              be formatted as <prefix>=<namespace>.
     -N <namespace>         : Namespace to generate types for.  -N arguments
                              limit which types are generated.  Multiple
                              -N arguments can be combined with -n args.
                              With -N or -n arguments, -G and -g arguments
                              are ignored.
     -n <file>              : File with Namespaces to generate types for.
                              One namespace per line.  See -N for further info.
     -G <namespace>         : Namespace to NOT generate types for.  If -N
                              or -n are specified, -G arguments are ignored.
     -g <file>              : File with Namespaces to NOT generate types for.
```

[1] WSDLtoCBindings.pdf

```
                          If -N or -n are specified, -g arguments are
                          ignored.
    -r <relative path>    : the relative path where generated headers are
                          install into.  $GL/include/<flavor>/<relpath>
    -d <output dir>       : directory to put the generated files in
    -no-package
    -np                   : No package creation.  Just generate files.
    -no-tarball
    -nb                   : Package files are created, but no package
                          tarball is generated.  -np implies -nt
    -no-func <func>
    -nf <func>            : No generation of the function <func>.  e.g.
                          wsnt_TopicExpressionType_deserialize.  This
                          option is useful if you want to write your
                          own marshalling functions for a given type.
    -no-skel
    -nk                   : No skeleton source file generation
    -no-service
    -ns                   : No service.  Only generate client bindings
                          and types.  -ns implies -nk.
    -no-client
    -nc                   : No client.  Only generate service bindings
                          and types.
    -no-types
    -nt                   : No types.  Only generate client and
                          service bindings.

  Required Argument:
   <wsdl schema>          : the WSDL schema to generate client side bindings
```

Limitations

- Only generates bindings from document/literal style WSDL schemas. For more information on WSDL schema styles, go here[2].
- Only generates ANSI-C bindings. C++ bindings are not supported.

[2] http://www-106.ibm.com/developerworks/webservices/library/ws-whichwsdl/

Chapter 10. GT 4.0.8 Incremental Release Notes: C WS Core

1. Introduction

These release notes are for the incremental release 4.0.8. It includes a summary of changes since 4.0.7, bug fixes since 4.0.7 and any known problems that still exist at the time of the 4.0.8 release. This page is in addition to the top-level 4.0.8 release notes at http://www.globus.org/toolkit/releasenotes/4.0.8.

For release notes about 4.0 (including feature summary, technology dependencies, etc) go to the C Common Libraries 4.0 Release Notes[1].

2. Changes Summary

Aside from bug fixes, no changes were made to this component since 4.0.7.

3. Bug Fixes

- Bug 5486:[2] C-ws core related problem - Deserialization of Header failed

4. Known Problems

- Bug 3405:[3] default serialization does not order correctly with respect to unqualified attributes
- Bug 3806:[4] Generated services are not prefixed
- Bug 4694:[5] many memory leak in web service calling
- Bug 6035:[6] WS Client handle initialization fails

5. For More Information

Click here[7] for more information about this component.

[1] http://www.globus.org/toolkit/docs/4.0/common/ccommonlib/C_Common_Libraries_Release_Notes.html
[2] http://bugzilla.globus.org/bugzilla/show_bug.cgi?id=5486
[3] http://bugzilla.globus.org/bugzilla/show_bug.cgi?id=3405
[4] http://bugzilla.globus.org/bugzilla/show_bug.cgi?id=3806
[5] http://bugzilla.globus.org/bugzilla/show_bug.cgi?id=4694
[6] http://bugzilla.globus.org/bugzilla/show_bug.cgi?id=6035
[7] index.html

Chapter 11. GT 4.0.7 Incremental Release Notes: C WS Core

1. Introduction

These release notes are for the incremental release 4.0.7. It includes a summary of changes since 4.0.6, bug fixes since 4.0.6 and any known problems that still exist at the time of the 4.0.7 release. This page is in addition to the top-level 4.0.7 release notes at http://www.globus.org/toolkit/releasenotes/4.0.7.

For release notes about 4.0 (including feature summary, technology dependencies, etc) go to the C Common Libraries 4.0 Release Notes[1].

2. Changes Summary

Aside from bug fixes, no changes were made to this component since 4.0.6.

3. Bug Fixes

No bugs have been fixed since 4.0.6.

4. Known Problems

- Multiple schemas which use the same namespace prefixes can confuse the WSDL parser.
- Nillable elements are not serialized or deserialized correctly if the element does not contain the minOccurs="0" attribute
- The service engine and clients are not thread-safe

5. For More Information

Click here[2] for more information about this component.

[1] http://www.globus.org/toolkit/docs/4.0/common/ccommonlib/C_Common_Libraries_Release_Notes.html
[2] index.html

Chapter 12. GT 4.0.6 Incremental Release Notes: C WS Core

1. Introduction

These release notes are for the incremental release 4.0.6. It includes a summary of changes since 4.0.5, bug fixes since 4.0.5 and any known problems that still exist at the time of the 4.0.6 release. This page is in addition to the top-level 4.0.6 release notes at http://www.globus.org/toolkit/releasenotes/4.0.6.

For release notes about 4.0 (including feature summary, technology dependencies, etc) go to the C Common Libraries 4.0 Release Notes[1].

2. Changes Summary

Aside from bug fixes, no changes were made to this component since 4.0.5.

3. Bug Fixes

- Bug 5381:[2] Loss of precision for xsd:float and xsd:double
- Bug 5424:[3] Error generating bindings from wsdl with multiple port types
- Bug 5759:[4] globus-wsrf-cgen creates a bad wsa_Relationship.c

4. Known Problems

- Multiple schemas which use the same namespace prefixes can confuse the WSDL parser.
- Nillable elements are not serialized or deserialized correctly if the element does not contain the minOccurs="0" attribute
- The service engine and clients are not thread-safe

5. For More Information

Click here[5] for more information about this component.

[1] http://www.globus.org/toolkit/docs/4.0/common/ccommonlib/C_Common_Libraries_Release_Notes.html
[2] http://bugzilla.globus.org/bugzilla/show_bug.cgi?id=5381
[3] http://bugzilla.globus.org/bugzilla/show_bug.cgi?id=5424
[4] http://bugzilla.globus.org/bugzilla/show_bug.cgi?id=5759
[5] index.html

Chapter 13. GT 4.0.5 Incremental Release Notes: C WS Core

1. Introduction

These release notes are for the incremental release 4.0.5. It includes a summary of changes since 4.0.4, bug fixes since 4.0.4 and any known problems that still exist at the time of the 4.0.5 release. This page is in addition to the top-level 4.0.5 release notes at http://www.globus.org/toolkit/releasenotes/4.0.5.

For release notes about 4.0 (including feature summary, technology dependencies, etc) go to the C Common Libraries 4.0 Release Notes[1].

2. Changes Summary

No changes have been made since 4.0.4.

3. Bug Fixes

No bugs have been fixed since 4.0.4.

4. Known Problems

- Multiple schemas which use the same namespace prefixes can confuse the WSDL parser.
- Nillable elements are not serialized or deserialized correctly if the element does not contain the minOccurs="0" attribute
- The service engine and clients are not thread-safe

5. For More Information

Click here[2] for more information about this component.

[1] http://www.globus.org/toolkit/docs/4.0/common/ccommonlib/C_Common_Libraries_Release_Notes.html
[2] index.html

Chapter 14. GT 4.0.4 Incremental Release Notes: C WS Core

1. Introduction

These release notes are for the incremental release 4.0.4. It includes a summary of changes since 4.0.3, bug fixes since 4.0.3 and any known problems that still exist at the time of the 4.0.4 release. This page is in addition to the top-level 4.0.4 release notes at http://www.globus.org/toolkit/releasenotes/4.0.4.

For release notes about 4.0 (including feature summary, technology dependencies, etc) go to the C Common Libraries 4.0 Release Notes[1].

2. Changes Summary

- Upgraded SpiderMonkey version 1.60
- Ported to Mac OS X / Intel x86
- Improved bindings generation: better error messages and handling of more XML Schema constructs.

3. Bug Fixes

- Bug #4772:[2] globus-wsrf-cgen uses literal NULL type when certain schema errors occur
- Bug #4773:[3] Bad error messages from globus-wsrf-cgen
- Bug #4807:[4] Incorrect default SOAPAction
- Bug #4857[5] Bug 4857 - globus-wsrf-cgen treats all element declarations as global
- Bug #4921[6] Local non-namespaced attributes can conflict

4. Known Problems

- Multiple schemas which use the same namespace prefixes can confuse the WSDL parser.
- Nillable elements are not serialized or deserialized correctly if the element does not contain the minOccurs="0" attribute
- The service engine and clients are not thread-safe

[1] http://www.globus.org/toolkit/docs/4.0/common/ccommonlib/C_Common_Libraries_Release_Notes.html
[2] http://bugzilla.globus.org/globus/show_bug.cgi?id=4772
[3] http://bugzilla.globus.org/globus/show_bug.cgi?id=4773
[4] http://bugzilla.globus.org/globus/show_bug.cgi?id=4807
[5] http://bugzilla.globus.org/globus/show_bug.cgi?id=4857
[6] http://bugzilla.globus.org/globus/show_bug.cgi?id=4921

5. For More Information

Click here[7] for more information about this component.

[7] index.html

Chapter 15. GT 4.0.3 Incremental Release Notes: C WS Core

1. Introduction

These release notes are for the incremental release 4.0.3. It includes a summary of changes since 4.0.2, bug fixes since 4.0.2 and any known problems that still exist at the time of the 4.0.3 release. This page is in addition to the top-level 4.0.3 release notes at http://www.globus.org/toolkit/releasenotes/4.0.3.

For release notes about 4.0 (including feature summary, technology dependencies, etc) go to the C Common Libraries 4.0 Release Notes[1].

2. Changes Summary

C WS Core implemented a security fix (by improving /tmp file handling for C WS Core tests) and a bug fix since GT 4.0.2.

3. Bug Fixes

The following bugs have been fixed in the C WS Core since GT 4.0.2:

- Bug 4647:[2] Improved /tmp file handling for C WS Core tests
- Bug 4536:[3] Fix error message in ws-addressing implementation

4. Known Problems

The following problems are known to exist for C WS Core at the time of the 4.0.4 release:

- Multiple schemas which use the same namespace prefixes can confuse the WSDL parser.
- Nillable elements are not serialized or deserialized correctly if the element does not contain the minOccurs="0" attribute
- The service engine and clients are not thread-safe

5. For More Information

Click here[4] for more information about this component.

[1] http://www.globus.org/toolkit/docs/4.0/common/ccommonlib/C_Common_Libraries_Release_Notes.html
[2] http://bugzilla.globus.org/globus/show_bug.cgi?id=4647
[3] http://bugzilla.globus.org/globus/show_bug.cgi?id=4536
[4] index.html

Chapter 16. GT 4.0.2 Release Notes: C WS Core

1. Introduction

These release notes are for the incremental release 4.0.2. It includes a summary of changes since 4.0.1, bug fixes since 4.0.1 and any known problems that still exist at the time of the 4.0.2 release. This page is in addition to the top-level 4.0.2 release notes at http://www.globus.org/toolkit/releasenotes/4.0.2.

For release notes about 4.0 (including feature summary, technology dependencies, etc) go to the C WS Core 4.0 Release Notes[1].

2. Changes Summary

The following changes have occurred for C WS Core.

- Improved WSDL parsing and bindings package generation
- Improved serialization of complex types

3. Bug Fixes

The following bugs were fixed for C WS Core:

- Bug 3641:[2] C WS Security package dependency error.
- Bug 3727:[3] Invalid mutex unlock in client bindings.
- Bug 3805:[4] Bindings generator does not handle wsdl:documentation nodes
- Bug 3904:[5] Generated packages are not relocatable.
- Bug 4086:[6] Attributes of xsd:schema elements are not handled correctly when a schema file is included.
- Bug 4168:[7] Trailing slash in libdir paths causes build error on AIX
- Bug 4171:[8] Date serialization errors in WS C tests
- Bug 4252:[9] xsi:type attributes missing on complex types

[1] http://www.globus.org/toolkit/docs/4.0/common/cwscore/C_WS_Core_Release_Notes.html
[2] http://bugzilla.globus.org/globus/show_bug.cgi?id=3641
[3] http://bugzilla.globus.org/globus/show_bug.cgi?id=3727
[4] http://bugzilla.globus.org/globus/show_bug.cgi?id=3805
[5] http://bugzilla.globus.org/globus/show_bug.cgi?id=3904
[6] http://bugzilla.globus.org/globus/show_bug.cgi?id=4086
[7] http://bugzilla.globus.org/globus/show_bug.cgi?id=4168
[8] http://bugzilla.globus.org/globus/show_bug.cgi?id=4171
[9] http://bugzilla.globus.org/globus/show_bug.cgi?id=4252

4. Known Problems

The following problems are known to exist for C WS Core at the time of the 4.0.2 release:

- Multiple schemas which use the same namespace prefixes can confuse the WSDL parser.
- Nillable elements are not serialized or deserialized correctly if the element does not contain the minOccurs="0" attribute
- The service engine and clients are not thread-safe

5. For More Information

Click here[10] for more information about this component.

[10] index.html

Chapter 17. GT 4.0.1 Release Notes: C WS Core

1. Introduction

These release notes are for the incremental release 4.0.1. It includes a summary of changes since 4.0.0, bug fixes since 4.0.0 and any known problems that still exist at the time of the 4.0.1 release. This page is in addition to the top-level 4.0.1 release notes at http://www.globus.org/toolkit/releasenotes/4.0.1.

For release notes about 4.0 (including feature summary, technology dependencies, etc) go to the C WS Core 4.0 Release Notes[1].

2. Changes Summary

The following changes have occurred for C WS Core.

- New implementation of WS-Secure Messaging.
- Handler data structures changes slightly to accomodate WS-Secure Messaging implementation.
- Client bindings now include doxygen documentation blocks which can be used to generate API documentation
- Improved XML parsing, especially of floating-point numbers and arbitrary-precision integers
- New attribute to include verbose error messages from the XML parser.
- Improved handling of GPT metadata by the C bindings generator.

3. Bug Fixes

The following bugs were fixed for C WS Core:

- Bug 3339:[2] The globus-wsrf-cgen program in the globus_c_wsrf_cgen package creates an empty client bindings library when the -no-client option is passed to it. This results in a compile error on some architectures.
- Bug 3370:[3] The C WSRF Core perfomance test programs fails if a large number of iterations is used. The problem is an error in the test program.
- Bug 3395:[4] The globus_js package does not compile on Tru64.
- Bug 3433:[5] The tests in the globus_c_ws_messaging_test package fail to run when the user's PATH does not contain ".".
- Bug 3434:[6] The tests in the globus_c_wsrf_resource_test package fail to run when the user's PATH does not contain ".".

[1] http://www.globus.org/toolkit/docs/4.0/common/cwscore/C_WS_Core_Release_Notes.html
[2] http://bugzilla.globus.org/globus/show_bug.cgi?id=3339
[3] http://bugzilla.globus.org/globus/show_bug.cgi?id=3370
[4] http://bugzilla.globus.org/globus/show_bug.cgi?id=3395
[5] http://bugzilla.globus.org/globus/show_bug.cgi?id=3433
[6] http://bugzilla.globus.org/globus/show_bug.cgi?id=3433

- Bug 3318:[7] SOAP deserializer fails to deserialize xsd:dateTime when the local timezone offset is negative.
- Bug 2178:[8] SOAP headers used for dispatching operations are not signed when TLS is not used. GT 4.0.1 includes an implementation of WS-Secure message which signs WS-Addressing-related headers as well as SOAP bodies.
- Bug 2270:[9] Packages generated by globus-wsrf-cgen do not include dependencies from the patch-and-build package metadata.
- Bug 3275:[10] The generated serialize_contents bindings pass GLOBUS_XSD_ELEMENT_CONTENTS_ONLY to subelements, causing subelement markup to be discarded.
- Bug 3322:[11] The globus_wsrf_core_create_endpoint_reference() function partially initializes endpoint the endpoint. Applications which use this function may crash when accessing the EPR.
- Bug 3505:[12] Error compiling globus_c_ws_messaging package on AIX. Error serializing floating-point numbers in XML.
- Bug 3515:[13] Error compiling test_secure_message_counter.c test program on AIX.

4. Known Problems

The following problems are known to exist for C WS Core at the time of the 4.0.1 release:

- SOAP faults without header elements are not parsed correctly.
- Multiple schemas which use the same namespace prefixes can confuse the WSDL parser.
- Nillable elements are not serialized or deserialized correctly if the element does not contain the minOccurs="0" attribute
- Errors that occur when processing resource properties on systems which crash when the printf format %s is matched with a null parameter will crash the c service container.

5. For More Information

Click here[14] for more information about this component.

[7] http://bugzilla.globus.org/globus/show_bug.cgi?id=3318
[8] http://bugzilla.globus.org/globus/show_bug.cgi?id=2178
[9] http://bugzilla.globus.org/globus/show_bug.cgi?id=2270
[10] http://bugzilla.globus.org/globus/show_bug.cgi?id=3275
[11] http://bugzilla.globus.org/globus/show_bug.cgi?id=3322
[12] http://bugzilla.globus.org/globus/show_bug.cgi?id=3505
[13] http://bugzilla.globus.org/globus/show_bug.cgi?id=3515
[14] index.html

Chapter 18. GT 4.0 Release Notes: C WS Core

1. Component Overview

The C WS Core provides a basic toolset in C for creating WSRF-enabled web services and clients conforming to the WS-Resource and WS-Notification specifications.

2. Feature Summary

Binding Generation:

- Binding Generation directly from WSDL schemas
  - ANSI-C stubs and skeletons
  - Non-blocking client stubs for writing event-driven code
  - EPR (EndpointReference) encapsulation
  - WSRF enabled client stubs and services
- HTTP/1.1 Support
- Embeddable Service API
- Standalone service container
- WSRF-enabled services

Deprecated Features

- Dynamic Deployment (WSDD) using AxisC++ was included in an early pre-release but is no longer supported.

3. Bug Fixes

- globus_wsrf_resource.h missing c++ guards[1]
- globus-wsrf-cgen segfaults on 64bit architectures[2]
- Current trunk AIX failure[3]
- globus_c_wsrf_core_bindings package layout problem[4]
- Rendezvous client bindings generation error on FC3 x86_64[5]

[1] http://bugzilla.globus.org/globus/show_bug.cgi?id=2834
[2] http://bugzilla.globus.org/globus/show_bug.cgi?id=2819
[3] http://bugzilla.globus.org/globus/show_bug.cgi?id=2915
[4] http://bugzilla.globus.org/globus/show_bug.cgi?id=2770
[5] http://bugzilla.globus.org/globus/show_bug.cgi?id=2763

- AIX problem with sed and long command line lengths[6]
- Building globus_c_gram_client_bindings on AIX fails[7]
- Parser Error Building 3.9.4 RC1 on FC2 x86_64[8]
- Core tools uses wrong path[9]
- globus_js does not pick up LDFLAGS[10]
- Core dump when submit job with globusrun-ws[11]
- Missing filelist entries[12]
- AIX build failure in C messaging tests[13]
- globus-wsc-container crash[14]
- Leaks in WS-C Core[15]
- missing error handling in soap read callbacks causes hang[16]
- parsing array with globus_xml_buffer consumes all memory[17]

4. Known Problems

- Bug 2310: support for http get queries of WSDL schemas[18]
- Bug 2437: Faults returned from C container[19]
- Bug 2460: utility funcs[20]
- Bug 2911: globus_service_engine_stop hang[21]
- Bug 3018: globus_c_wsrf_core_bindings fails to build on AIX[22]
- Bug 3058: globus_soap_message_handle_init_from_dom() doesn't work[23]
- Bug 3208: C registryService bindings in 4.0[24]

[6] http://bugzilla.globus.org/globus/show_bug.cgi?id=2928
[7] http://bugzilla.globus.org/globus/show_bug.cgi?id=2580
[8] http://bugzilla.globus.org/globus/show_bug.cgi?id=2446
[9] http://bugzilla.globus.org/globus/show_bug.cgi?id=3004
[10] http://bugzilla.globus.org/globus/show_bug.cgi?id=2543
[11] http://bugzilla.globus.org/globus/show_bug.cgi?id=2867
[12] http://bugzilla.globus.org/globus/show_bug.cgi?id=2617
[13] http://bugzilla.globus.org/globus/show_bug.cgi?id=2952
[14] http://bugzilla.globus.org/globus/show_bug.cgi?id=2610
[15] http://bugzilla.globus.org/globus/show_bug.cgi?id=2619
[16] http://bugzilla.globus.org/globus/show_bug.cgi?id=3153
[17] http://bugzilla.globus.org/globus/show_bug.cgi?id=3154
[18] http://bugzilla.globus.org/globus/show_bug.cgi?id=2310
[19] http://bugzilla.globus.org/globus/show_bug.cgi?id=2437
[20] http://bugzilla.globus.org/globus/show_bug.cgi?id=2460
[21] http://bugzilla.globus.org/globus/show_bug.cgi?id=2911
[22] http://bugzilla.globus.org/globus/show_bug.cgi?id=3018
[23] http://bugzilla.globus.org/globus/show_bug.cgi?id=3058
[24] http://bugzilla.globus.org/globus/show_bug.cgi?id=3208

5. Technology Dependencies

C WS Core depends on the following GT components:

- C Common Libraries
- Pre-WS Authentication and Authorization (GSI)
- Globus XIO[25] (used by C WS core for efficient HTTP and TCP transport)

C WS Core depends on the following 3rd party software:

- Libxml2[26] (used by C WS Core for SOAP XML parsing and WSDL parsing)
- OpenSSL[27] (used by C WS Core for Security)
- JavaScript[28] (used by C WS Core as a template language to generate the C bindings from WSDL schemas)

6. Supported Platforms

Tested Platforms for C WS Core

- IA32/Linux/gcc32
- IA64/Linux/gcc64
- x86_64/Linux/gcc64
- SPARC/Solaris 9/vendorcc32
- PowerPC/AIX 5.2/vendorcc32
- Mac/OS X/gcc32

7. Backward Compatibility Summary

Protocol changes since GT version 3.2

- SOAP messages conform to WSRF schemas instead of previous OGSI/OGSA schemas.
- WS-Addressing has been added to the list of supported standards, as defined by the WS-Resource Framework.
- HTTP/1.1 with 'chunked' transfer encoding is used by default.

API changes since GT version 3.2

- The 3.2 cbindings API is obsolete, with no overlap to the new API. Bindings APIs are now generated directly from WSDL.
- The underlying XML/SOAP messaging framework is also new, based on the libxml2 pull parser API.

[25] http://www.globus.org/toolkit/docs/4.0/common/xio/
[26] http://www.xmlsoft.org/
[27] http://www.openssl.org
[28] http://www.mozilla.org

Schema changes since GT version 3.2

- Schemas are completely new. The WS C Core implements the OASIS WSRF and WSN working drafts specifications (with minor fixes to the 1.2-draft-01 published schemas and with the March 2004 version of the WS-Addressing specification.)

8. For More Information

Click here[29] for more information about this component.

[29] index.html