Artix™

Developing Artix Plug-Ins with C++

Version 3.0, June 2005

Making Software Work Together™

Updated: 30-Oct-2005

Contents

List of Figures

Preface

What is Covered in this Book

Artix is built on top of IONA's ART (Adaptive Runtime Technology), which uses dynamic linking to load Artix plug-ins at runtime. This book explains how to write your own plug-ins for the ART framework. Two major areas are covered: implementing Artix interceptors, which enables you to access request and reply messages as they pass through the stack; and implementing Artix transports, which enables you to implement custom transport protocols.

Who Should Read this Book

This book is aimed at experienced Artix developers who need to customize the behavior of their Artix applications using advanced APIs.

If you would like to know more about WSDL concepts, see the Introduction to WSDL in Getting Started with Artix.

Finding Your Way Around the Library

The Artix library contains several books that provide assistance for any of the tasks you are trying to perform. The Artix library is listed here, with a short description of each book.

If you are new to Artix

You may be interested in reading:

- Release Notes contains release-specific information about Artix.
- Installation Guide describes the prerequisites for installing Artix and the procedures for installing Artix on supported systems.
- Getting Started with Artix describes basic Artix and WSDL concepts.

To design and develop Artix solutions

Read one or more of the following:

- Designing Artix Solutions provides detailed information about describing services in Artix contracts and using Artix services to solve problems.
- Developing Artix Applications in C++ discusses the technical aspects of programming applications using the C++ API.
- Developing Artix Plug-ins with C++ discusses the technical aspects of implementing plug-ins to the Artix bus using the C++ API.
- Developing Artix Applications in Java discusses the technical aspects of programming applications using the Java API.
- Artix for CORBA provides detailed information on using Artix in a CORBA environment.
- Artix for J2EE provides detailed information on using Artix to integrate with J2EE applications.
- Artix Technical Use Cases provides a number of step-by-step examples of building common Artix solutions.

To configure and manage your Artix solution

Read one or more of the following:

- Deploying and Managing Artix Solutions describes how to deploy Artix-enabled systems, and provides detailed examples for a number of typical use cases.
- Artix Configuration Guide explains how to configure your Artix environment. It also provides reference information on Artix configuration variables.
- IONA Tivoli Integration Guide explains how to integrate Artix with IBM Tivoli.
- IONA BMC Patrol Integration Guide explains how to integrate Artix with BMC Patrol.
- Artix Security Guide provides detailed information about using the security features of Artix.

Reference material

In addition to the technical guides, the Artix library includes the following reference manuals:

- Artix Command Line Reference
- Artix C++ API Reference
- Artix Java API Reference

Have you got the latest version?

The latest updates to the Artix documentation can be found at http://www.iona.com/support/docs.

Compare the version dates on the web page for your product version with the date printed on the copyright page of the PDF edition of the book you are reading.

Searching the Artix Library

You can search the online documentation by using the **Search** box at the top right of the documentation home page:

http://www.iona.com/support/docs

To search a particular library version, browse to the required index page, and use the **Search** box at the top right. For example:

http://www.iona.com/support/docs/artix/3.0/index.xml

You can also search within a particular book. To search within an HTML version of a book, use the **Search** box at the top left of the page. To search within a PDF version of a book, in Adobe Acrobat, select **Edit**|**Find**, and enter your search text.

Online Help

Artix Designer includes comprehensive online help, providing:

- Detailed step-by-step instructions on how to perform important tasks.
- A description of each screen.
- A comprehensive index, and glossary.
- A full search feature.
- Context-sensitive help.

There are two ways that you can access the online help:

- Click the Help button on the Artix Designer panel, or
- Select **Contents** from the Help menu

Additional Resources

The IONA Knowledge Base (http://www.iona.com/support/knowledge_base/index.xml) contains helpful articles written by IONA experts about Artix and other products.

The IONA Update Center (http://www.iona.com/support/updates/index.xml) contains the latest releases and patches for IONA products.

If you need help with this or any other IONA product, go to IONA Online Support (http://www.iona.com/support/index.xml).

Comments, corrections, and suggestions on IONA documentation can be sent to docs-support@iona.com .

Document Conventions

Typographical conventions

This book uses the following typographical conventions:

`Fixed width`	Fixed width (courier font) in normal text represents portions of code and literal names of items such as classes, functions, variables, and data structures. For example, text might refer to the `IT_Bus::AnyType` class.
	Constant width paragraphs represent code examples or information a system displays on the screen. For example:
	`#include <stdio.h>`
`Fixed width italic`	Fixed width italic words or characters in code and commands represent variable values you must supply, such as arguments to commands or path names for your particular system. For example:
	`% cd /users/`*`YourUserName`*
Italic	Italic words in normal text represent *emphasis* and introduce *new terms*.
Bold	Bold words in normal text represent graphical user interface components such as menu commands and dialog boxes. For example: the **User Preferences** dialog.

Keying Conventions

This book uses the following keying conventions:

No prompt	When a command's format is the same for multiple platforms, the command prompt is not shown.
%	A percent sign represents the UNIX command shell prompt for a command that does not require root privileges.
#	A number sign represents the UNIX command shell prompt for a command that requires root privileges.
>	The notation > represents the MS-DOS or Windows command prompt.
... . . .	Horizontal or vertical ellipses in format and syntax descriptions indicate that material has been eliminated to simplify a discussion.
[]	Brackets enclose optional items in format and syntax descriptions.
{}	Braces enclose a list from which you must choose an item in format and syntax descriptions.
|	In format and syntax descriptions, a vertical bar separates items in a list of choices enclosed in {} (braces). In graphical user interface descriptions, a vertical bar separates menu commands (for example, select **File|Open**).

Basic Plug-In Implementation

This chapter describes how to implement the core classes of an Artix plug-in, IT_Bus::BusPlugInFactory and IT_Bus::BusPlugIn.

In this chapter

This chapter discusses the following topics:

Overview of a Basic Artix Plug-In

Overview

This section describes the basic features of an Artix plug-in:

- ORB plug-ins.
- Artix plug-ins.
- Plug-in packaging.
- Configuration.
- Loading the plug-in.
- Initializing the plug-in.
- BusPlugInFactory object.
- BusPlugIn object.

ORB plug-ins

An *ORB plug-in* is a well-defined component that can be independently loaded into an application. The term, ORB plug-in, betrays the CORBA origins of the Artix plug-in framework. In practice, however, ORB plug-ins are *not* specific to CORBA.

Artix plug-ins

An *Artix plug-in* is just a special case of an ORB plug-in. Artix defines a platform-independent framework for loading plug-ins dynamically, based on the dynamic linking capabilities of modern operating systems (that is, using shared libraries or DLLs).

Plug-in packaging

Plug-ins are packaged in a form that is compatible with the dynamic linking capabilities of the particular platform on which they are deployed: a shared library, a DLL, or a JAR file.

For example, version 5 of a `tunnel` plug-in implemented in C++ for the Visual C++ 6.0 compiler on the Windows platform would be packaged as a `.dll` file and a `.dps` file (ART-specific dependencies file), as follows:

```
it_tunnel5_vc60.dll
it_tunnel5_vc60.dps
```

Configuration

The plug-ins that an application should load are specified by the `orb_plugins` configuration variable, which contains a list of plug-in names.

In addition, for each plug-in that is to be loaded, you need to identify the whereabouts of the plug-in. For C++ applications, you specify the root name of the corresponding shared library using the `plugins:`*`<plugin_name>`*`:shlib_name` configuration variable.

For example, the following extract shows how to configure an application, whose ORB name is `plugin_example`, to load a single plug-in, `sample_artix_interceptor`.

```
# Artix domain configuration file
...
plugin_example {
    orb_plugins = ["sample_artix_interceptor"];

    plugins:sample_artix_interceptor:shlib_name =
   "it_sample_artix_interceptor";
};
```

Loading the plug-in

Figure 1 show how a plug-in is loaded by an application as the application starts up. The steps to load the plug-in are as follows:

1. The user launches the application, `app`, specifying the ORB name as `plugin_example` at the command line.
2. As the application starts up, it scans the Artix configuration file to determine which plug-ins to load. Priority is given to the configuration settings in the `plugin_example` configuration scope (that is, the ORB name determines which configuration scopes to search).
3. The Artix core loads the plug-ins specified by the application's configuration.

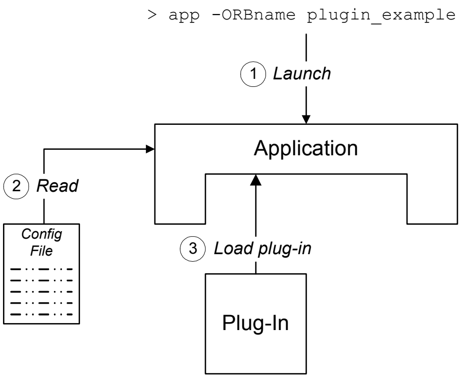

Figure 1: *Loading a Plug-In*

Initializing the plug-in

Plug-ins are usually initialized when the main application code calls `IT_Bus::init()`. Figure 2 shows the plug-in initialization sequence, which proceeds as follows:

1. The main application code calls `IT_Bus::init()`.
2. The Artix core iterates over all of the plug-ins in the `orb_plugins` list, calling `IT_Bus::BusPlugInFactory::create_bus_plugin()` on each one.
3. The `BusPlugInFactory` object creates an `IT_Bus::BusPlugIn` object, which initializes the state of the plug-in for the current ORB instance.
4. After all of the `BusPlugIn` objects have been created, the Artix core calls `bus_init()` on each `BusPlugIn` object.

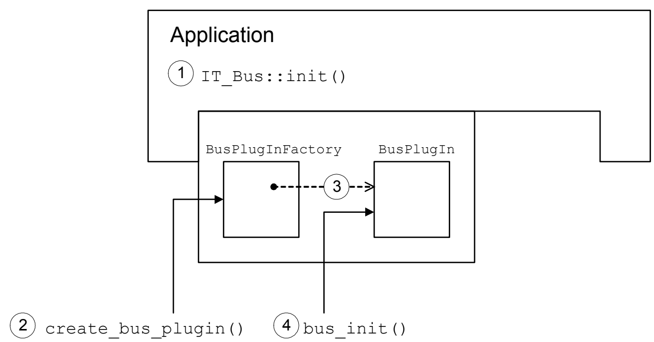

Figure 2: *Initializing a Plug-In*

BusPlugInFactory object

A `BusPlugInFactory` object provides the basic hook for initializing an Artix plug-in. A single static instance of the `BusPlugInFactory` object is created when the plug-in is loaded into an application. See "Implementing a BusPlugInFactory Class" on page 8 for more details.

BusPlugIn object

A `BusPlugIn` object caches the state of the plug-in for the current Bus instance (an application can create multiple Bus instances). Typically, the `BusPlugIn` object is responsible for performing most of the plug-in initialization and shutdown tasks.

Developing an Artix Plug-In

Overview

This section describes how to develop the basic classes for the `sample_artix_interceptor` plug-in. The objects described here, of `IT_Bus::BusPlugInFactory` and `IT_Bus::BusPlugIn` type, are the basic objects needed by every Artix plug-in, enabling a plug-in to initialize and register with the Artix core.

In this section

This section contains the following subsections:

Development Steps

How to implement

To implement an Artix plug-in, perform the following steps:

<table>
<tr><th>Step</th><th>Action</th></tr>
<tr><td>1</td><td>Implement a class that inherits from the <code>IT_Bus::BusPlugInFactory</code> base class. This class should:
Implement <code>create_bus_plugin()</code> to return a new <code>IT_Bus::BusPlugIn</code> object.
Implement <code>destroy_bus_plugin()</code> to clean up the allocated <code>BusPlugIn</code> object at shutdown time.</td></tr>
<tr><td>2</td><td>Implement a class that inherits from the <code>IT_Bus::BusPlugIn</code> base class. This class should:
Implement <code>bus_init()</code> to perform various actions at initialization time.
Implement <code>bus_shutdown()</code> to perform various actions at shutdown time.</td></tr>
<tr><td>3</td><td>Create the following static instances:
A static instance of the newly implemented <code>BusPlugInFactory</code> class.
Either of the following static instances:
A static instance of the <code>IT_Bus::BusORBPlugIn</code> class (for plug-ins packaged as a shared library), or
A static instance of the <code>IT_Bus::GlobalBusORBPlugIn</code> class (for plug-ins linked directly to the application).
The static instances are created when the library containing the plug-in is loaded.</td></tr>
</table>

Implementing a BusPlugInFactory Class

Overview

This section describes how to implement a `BusPlugInFactory` class for the `sample_artix_interceptor` plug-in.

An `BusPlugInFactory` object is the most fundamental constituent of a plug-in and is responsible for bootstrapping the rest of the plug-in functionality. A typical `BusPlugInFactory` implementation does not do very much. Usually it just creates a new `BusPlugIn` object in response to an invocation of the `create_bus_plugin()` operation.

C++ BusPlugInFactory header

Example 1 shows the C++ header for the `SampleBusPlugInFactory` class, which is an example of an `IT_Bus::BusPlugInFactory` class.

Example 1: *C++ Header for the BusPlugInFactory Class*

```
// C++
#include <it_bus/bus.h>
#include <it_bus/exception.h>

1 #include <it_bus_pdk/bus_plugin_factory.h>

// In namespace, IT_SampleArtixInterceptor
2 class SampleBusPlugInFactory :
    public IT_Bus::BusPlugInFactory
{
  public:
    SampleBusPlugInFactory();
    virtual ~SampleBusPlugInFactory();

    virtual IT_Bus::BusPlugIn*
    create_bus_plugin(
        IT_Bus::Bus_ptr bus
    ) IT_THROW_DECL((IT_Bus::Exception));

    virtual void
    destroy_bus_plugin(
        IT_Bus::BusPlugIn* bus_plugin
    );

  private:
    SampleBusPlugInFactory(const SampleBusPlugInFactory&);
```

Example 1: *C++ Header for the BusPlugInFactory Class*

```
    SampleBusPlugInFactory&
    operator=(const SampleBusPlugInFactory&);
};
```

The preceding header file can be described as follows:

1. Include `it_bus_pdk/bus_plugin_factory.h`, which is the header file for the `IT_Bus::BusPlugInFactory` class.
2. The plug-in factory class, `SampleBusPlugInFactory`, inherits from `IT_Bus::BusPlugInFactory`, which is the base class for all plug-in factories.

C++ SampleBusPlugInFactory implementation

Example 2 shows the C++ implementation of the `SampleBusPlugInFactory` class, which is an example of an `IT_Bus::BusPlugInFactory` class.

Example 2: *C++ Implementation of the SampleBusPlugInFactory Class*

```
// C++

//   SampleBusPlugInFactory
//

SampleBusPlugInFactory::SampleBusPlugInFactory()
{
    // complete
}

SampleBusPlugInFactory::~SampleBusPlugInFactory()
{
    // complete
}

IT_Bus::BusPlugIn*
1 SampleBusPlugInFactory::create_bus_plugin(
    IT_Bus::Bus* bus
) IT_THROW_DECL((IT_Bus::Exception))
{
    return new SampleBusPlugIn(bus);
}
```

Example 2: *C++ Implementation of the SampleBusPlugInFactory Class*

```
void
2 SampleBusPlugInFactory::destroy_bus_plugin(
    IT_Bus::BusPlugIn* bus_plugin
)
{
    delete bus_plugin;
}
```

The preceding implementation can be described as follows:

1. The `SampleBusPlugInFactory::create_bus_plugin()` creates an instance of an `IT_Bus::BusPlugIn` object.

   The `create_bus_plugin()` operation is automatically called whenever a new Bus instance is created (for example, whenever you call `IT_Bus::init()`). Because you are allowed to create more than one Bus instance, the plug-in must keep track of its state for each Bus—hence the need for a separate `BusPlugIn` object.

2. The `SampleBusPlugInFactory::destroy_bus_plugin()` cleans up Bus plug-in objects at shutdown time.

Implementing a BusPlugIn Class

Overview

This section describes how to implement a `BusPlugIn` class for the `sample_artix_interceptor` plug-in.

`BusPlugIn` objects are typically responsible for the following tasks:

- Registering factory objects that extend Artix functionality.
- Coordinating the plug-in's initialization and shutdown tasks.
- Caching the plug-in's per-Bus data and object references.

C++ BusPlugIn header

Example 3 shows the C++ header for the `SampleBusPlugIn` class, which is an example of an `IT_Bus::BusPlugIn` class.

Example 3: *C++ Header for the BusPlugIn Class*

```
  // C++
  #include <it_bus/bus.h>
  #include <it_bus/exception.h>

1 #include <it_bus_pdk/bus_plugin.h>

  // In namespace IT_SampleArtixInterceptor

2 class SampleBusPlugIn :
          public IT_Bus::BusPlugIn,
          public IT_Bus::InterceptorFactory
  {
    public:
      // IT_Bus::BusPlugIn
      //
      IT_EXPLICIT
      SampleBusPlugIn(
          IT_Bus::Bus_ptr bus
      ) IT_THROW_DECL((IT_Bus::Exception));

      virtual ~SampleBusPlugIn();

      virtual void
      bus_init() IT_THROW_DECL((IT_Bus::Exception));

      virtual void
      bus_shutdown() IT_THROW_DECL((IT_Bus::Exception));
```

Example 3: *C++ Header for the BusPlugIn Class*

```
    // IT_Bus::InterceptorFactory
    //
    ...  // (not shown)

  private:
    SampleBusPlugIn(const SampleBusPlugIn&);

    SampleBusPlugIn&
    operator=(const SampleBusPlugIn&);

    IT_Bus::String m_name;
};
```

The preceding C++ header can be described as follows:

1. Include `it_bus_pdk/bus_plugin.h`, which is the header file for the `IT_Bus::BusPlugIn` class.
2. The plug-in class, `SampleBusPlugIn`, inherits from two base classes:
   - `IT_Bus::BusPlugIn`—the base class for all plug-in classes.
   - `IT_Bus::InterceptorFactory`—the base class for an interceptor factory. You only need this class, if you are implementing Artix interceptors (the code here is taken from an Artix interceptor demonstration).

C++ BusPlugIn implementation

Example 4 shows the C++ implementation of the `SampleBusPlugIn` class, which is an example of an `IT_Bus::BusPlugIn` class.

Example 4: *C++ Implementation of the BusPlugIn Class*

```
// C++

// In namespace IT_SampleArtixInterceptor

1 SampleBusPlugIn::SampleBusPlugIn(
    IT_Bus::Bus_ptr bus
) IT_THROW_DECL((IT_Bus::Exception))
  :
2   BusPlugIn(bus),
3   m_name("artix_interceptor")
{
```

Example 4: *C++ Implementation of the BusPlugIn Class*

```
    assert(bus != 0);
}

SampleBusPlugIn::~SampleBusPlugIn()
{
    // complete
}

void
4 SampleBusPlugIn::bus_init(
) IT_THROW_DECL((IT_Bus::Exception))
{
5     IT_Bus::Bus_ptr bus = get_bus();

    InterceptorFactoryManager& factory_manager =
        bus->get_pdk_bus()->get_interceptor_factory_manager();

6     factory_manager.register_interceptor_factory(
        m_name,
        this
    );
}

void
7 SampleBusPlugIn::bus_shutdown(
) IT_THROW_DECL((IT_Bus::Exception))
{
    IT_Bus::Bus_ptr bus = get_bus();
    assert(bus != 0);

    InterceptorFactoryManager& factory_manager =
        bus->get_pdk_bus()->get_interceptor_factory_manager();

8     factory_manager.unregister_interceptor_factory(
        this
    );
}
```

The preceding C++ implementation can be described as follows:

1. The `BusPlugIn` constructor typically does not do much, apart from initializing a couple of member variables.
2. You must always pass the `bus` instance to the base constructor, `IT_Bus::BusPlugIn()`, which caches the reference and makes it available through the `IT_Bus::BusPlugIn::get_bus()` accessor.
3. The `m_name` member variable caches the name of the interceptor factory for later use. The interceptor name is used in the following contexts:
    - When registering the interceptor factory with the bus.
    - To enable the interceptor, by adding the interceptor name to the relevant lists of interceptors in the `artix.cfg` file.
4. Artix calls `bus_init()` after all of the plug-ins have been created by calls to `create_bus_plugin()`. The `bus_init()` function is where most of the plug-in initialization actually occurs. Typical tasks performed in `bus_init()` include:
    - Reading configuration information from the `artix.cfg` configuration file.
    - Registering special kinds of objects, such as interceptor factories, transport factories, binding factories, and so on.
    - Logging.
5. The `BusPlugIn::get_bus()` function accesses the Bus reference that was cached by the `BusPlugIn` base class constructor.
6. Because this code is from an interceptor demonstration, the `bus_init()` implementation registers an interceptor factory. The register function takes the interceptor name, `m_name`, and the interceptor factory instance, `this`, as arguments.
7. Artix calls `bus_shutdown()` as the Bus is being shut down. This is a the place to clean up any resources used by the plug-in implementation. Typically, you would also unregister objects that were registered in `bus_init()`.
8. Because this code is from an interceptor demonstration, unregister the interceptor factory.

Creating Static Instances

Overview

The mechanism for bootstrapping a plug-in is based on declaring two static objects, as follows:

- A static instance of the plug-in factory (a subtype of `IT_Bus::BusPlugInFactory`).
- Either of the following static instances:
  - BusORBPlugIn static instance.
  - GlobalBusORBPlugIn static instance.

BusORBPlugIn static instance

Create a static instance of `IT_Bus::BusORBPlugIn` type, if you intend to package your plug-in as a shared library. The `BusORBPlugIn` constructor has the following characteristics:

- The constructor registers the Bus plug-in factory with the Bus core.
- The constructor does *not* call `create_bus_plugin()` on the factory.

If a plug-in is packaged as a shared library, you must list the plug-in name in the `orb_plugins` list in the Artix configuration file. For each of the plug-ins listed in `orb_plugins`, Artix does the following:

- Artix attempts to load the relevant shared library (dynamic loading).
- Artix calls `create_bus_plugin()` on the factory.

GlobalBusORBPlugIn static instance

Create a static instance of `IT_Bus::GlobalBusORBPlugIn` type, if you intend to link the plug-in code directly into your application. The `GlobalBusORBPlugIn` constructor has the following characteristics:

- The constructor registers the Bus plug-in factory with the Bus core.
- The constructor calls `create_bus_plugin()` on the factory.

A side effect of using `GlobalBusORBPlugIn` is that you can have only one `IT_Bus::BusPlugIn` object for each application (instead of one `IT_Bus::BusPlugIn` object for each Bus object).

If a plug-in is linked directly with your application, there is no need to add the plug-in name to the `orb_plugins` list in the Artix configuration.

C++ static instances

Static instances, of `SampleBusPlugInFactory` and `IT_Bus::BusORBPlugIn` type, are created by the following lines of code.

Example 5: *Creating Static Objects for a Plug-In*

```
// C++
namespace IT_SampleArtixInterceptor
{
1    const char* const und_sample_plugin_name =
    "sample_artix_interceptor";

2    SampleBusPlugInFactory und_sample_plugin_factory;

3    IT_Bus::BusORBPlugIn und_sample_interceptor_plugin(
        und_sample_plugin_name,
        und_sample_plugin_factory
    );
}
```

The preceding code can be explained as follows:

1. Define the plug-in name to be `sample_artix_interceptor`. This is the name that must be added to the `orb_plugins` list in the `artix.cfg` file in order to load the plug-in.
2. Create a static `SampleBusPlugInFactory` instance, `und_sample_plugin_factory`. This static instance is created automatically, as soon as the `sample_artix_interceptor` plug-in is loaded.
3. Create a static `IT_Bus::BusORBPlugIn` instance, `und_sample_interceptor_plugin`, taking the plug-in name, `und_sample_plugin_name`, and the plug-in factory, `und_sample_plugin_factory`, as arguments.

   This line is of critical importance because it bootstraps the entire plug-in functionality. When the static `BusORBPlugIn` constructor is called, it automatically registers the plug-in factory with the Bus.

Request Interceptors

Artix request interceptors enable you to intercept operation requests and replies, where the request and reply data are accessible in a high-level format. This chapter describes how to access and modify header data and parameter data from within a request interceptor.

In this chapter

This chapter discusses the following topics:

Overview of Request Interceptors

Overview

This section provides a high-level overview of the architecture of request interceptors in Artix.

In this section

This section contains the following subsections:

Client Request Interceptors

Overview

Client request interceptors are used to intercept requests (and replies) on the client side, between the proxy object and the binding. Figure 3 shows the architecture of a client request interceptor chain.

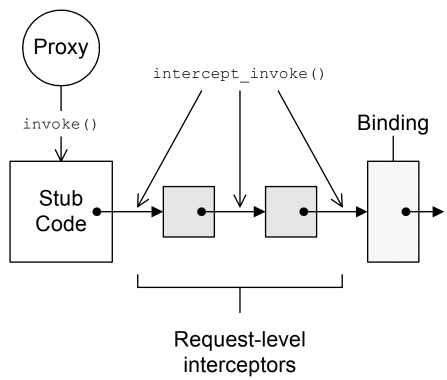

Figure 3: *A Client Request Interceptor Chain*

Interceptor chaining

A client request interceptor chain is arranged as a singly-linked list: each interceptor in the chain stores a pointer to the next and the chain is terminated by a binding object.

Client request interceptor chains are created dynamically. The Artix core reads the relevant configuration variables as it starts up and initializes a chain of interceptors that link together in the specified order.

ClientRequestInterceptor class

A client request interceptor is represented by an instance of `IT_Bus::ClientRequestInterceptor` type. The `ClientRequestInterceptor` class has the following members:

- `m_next_interceptor` member variable.

  Stores the pointer to the next `ClientRequestInterceptor` in the chain. The `m_next_interceptor` variable is automatically initialized by Artix when it constructs the chain.

- `intercept_invoke()` member function.

  This is the main interceptor function. You implement this function to implement new features with interceptors.

---

intercept_invoke() function

Example 6 shows the basic outline of how to implement the `intercept_invoke()` function.

Example 6: *Outline of intercept_invoke() Function*

```
// C++
using namespace IT_Bus;

void
CustomCltReqInterceptor::intercept_invoke(ClientOperation& data)
{
    // PRE-INVOKE processing
    // ...

     m_next_interceptor->intercept_invoke(data);

    // POST-INVOKE processing
    // ...
}
```

The typical implementation of `intercept_invoke()` has three main parts:

- *Pre-invoke processing*—put any code here that you would want to execute *before* the request is dispatched to the remote server. At this point, the input parts are already initialized. You can examine or replace input parts.
- *Call the next interceptor in the chain*—you must always call `intercept_invoke()` on the next interceptor, as shown here.
- *Post-invoke processing*—put any code here that you would want to execute *after* the reply is received from the remote server. At this point, both the input and output parts are initialized. You can examine or modify the output parts. Replacing parts has no effect.

ClientOperation class

The `data` object that passes along the client request interceptor chain is an instance of the `IT_Bus::ClientOperation` class. The `ClientOperation` class encapsulates all of the request and reply data.

The most important member functions of the `ClientOperation` class are as follows:

- `get_name()`

  Returns an `IT_Bus::String` that holds the name of the operation that is being invoked.
- `get_input_message()`

  Returns an `IT_Bus::WritableMessage` object that contains the input parts. The simplest way to obtain the input parts list is to call `get_input_message().get_parts()`.
- `get_output_message()`

  Returns an `IT_Bus::ReadableMessage` object that contains the output parts. The simplest way to obtain the output parts list is to call `get_output_message().get_parts()`.
- `request_contexts()`

  Returns an `IT_Bus::ContextContainer` object that provides access to request contexts. You can use this object to write or read headers in the request message.
- `reply_contexts()`

  Returns an `IT_Bus::ContextContainer` object that provides access to reply contexts. You can use this object to write or read headers in the reply message.

Configuring a client request interceptor

To configure Artix to use a client request interceptor, you must update the client request interceptor list in the Artix configuration file. The client request interceptor list consists of a list of alternative chain configurations, as follows:

```
binding:artix:client_request_interceptor_list = ["Chain01",
   "Chain02", "Chain03", ...];
```

The Artix core first attempts to construct an interceptor chain according to pattern in *`Chain01`*. If this attempt fails (for example, if one of the interceptors in the chain is unavailable) Artix attempts to use the next chain configuration, *`Chain02`*, instead.

Each chain configuration is specified in the following format:

```
"InterceptorA+InterceptorB+..."
```

Where *`InterceptorA`* is the name of interceptor A and *`InterceptorB`* is the name of interceptor B and so on. An *interceptor name* is the name under which the interceptor factory is registered with the `IT_Bus::InterceptorFactoryManager`.

Configuring an interceptor in an Artix router

If an interceptor is meant to be used within an Artix router process, you might need to configure the router to ensure the interceptor is not bypassed. Specifically, if you configure a route that maps messages between two bindings of the same type (for example, CORBA-to-CORBA), the router bypasses interceptors by default. This is often a useful optimization, but is unsuitable for some applications.

To force all routed messages to pass through the interceptors in the router, you should add the following line to the router's configuration:

```
plugins:routing:use_pass_through = "false";
```

Server Request Interceptors

Overview

Server request interceptors are used to intercept requests (and replies) on the server side, between the binding and the servant object. Figure 4 shows the architecture of a server request interceptor chain.

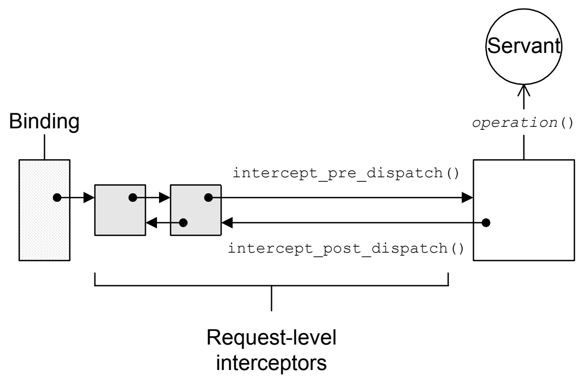

Figure 4: *Server Request Interceptor Chain*

Interceptor chaining

A server request interceptor chain is arranged as a doubly-linked list: each interceptor in the chain stores pointers to the next one and the previous one.

Server request interceptor chains are created dynamically. The Artix core reads the relevant configuration variables as it starts up and initializes a chain of interceptors that link together in the specified order.

Alternative interceptor model

Server request interceptors support an alternative interceptor model, which requires you to implement a single interceptor function, `intercept_around_dispatch()`, as shown in Figure 5.

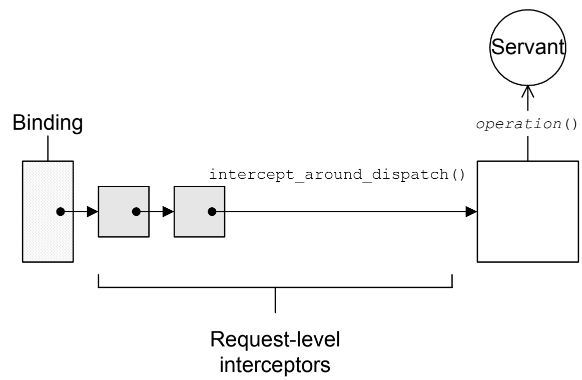

Figure 5: *Server Request Interceptors Using intercept_around_dispatch()*

The `intercept_around_dispatch()` is called at the very start of the dispatch process (before `intercept_pre_dispatch()`) and returns at the very end of the dispatch process (after `interceptor_post_dispatch()`).

ServerRequestInterceptor class

A server request interceptor is represented by an instance of `IT_Bus::ServerRequestInterceptor` type. The `ServerRequestInterceptor` class has the following members:

- `m_next_interceptor` member variable.

  Stores the pointer to the next `ServerRequestInterceptor` in the chain. The `m_next_interceptor` variable is automatically initialized by Artix.
- `m_prev_interceptor` member variable.

  Stores the pointer to the preceding `ServerRequestInterceptor` in the chain. The `m_prev_interceptor` variable is automatically initialized by Artix.
- `intercept_around_dispatch()` member function.

  An intercept point that is called at the very start of the dispatch process (before the input parts have been unmarshalled); and returns at the very end of the dispatch process (after the output parts have been marshalled).

If you don't want to implement this function, you can inherit the default implementation from `IT_Bus::ServerRequestInterceptor`, which simply calls the next interceptor in the chain.

- `intercept_pre_dispatch()` member function.

  Called after the input parts have been unmarshalled, but before dispatching to the servant.

  If you don't want to implement this function, you can inherit the default implementation from `IT_Bus::ServerRequestInterceptor`, which simply calls the next interceptor in the chain.

- `intercept_post_dispatch()` member function.

  Called after dispatching to the servant, but before marshalling the output parts.

  If you don't want to implement this function, you can inherit the default implementation from `IT_Bus::ServerRequestInterceptor`, which simply calls the next interceptor in the chain.

Combining the interceptor models

If necessary, you can combine the two interceptor models by implementing all of the intercept functions from the `ServerRequestInterceptor` class. In this case, the sequence of interceptor calls is as follows:

1. Artix calls `intercept_around_dispatch()` on the first interceptor, which calls `intercept_around_dispatch()` on the second interceptor, and so on to the end of the chain.
2. Inside the call to `intercept_around_dispatch()`, Artix calls the first interceptor's `intercept_pre_dispatch()` function, which calls the second interceptor's `intercept_pre_dispatch()` function, and so on to the end of the chain. The last interceptor returns, then the next-to-last interceptor, and then all the way back to the first interceptor.
3. Artix calls the application code.
4. Artix calls the last interceptor's `intercept_post_dispatch()` function, which calls the next-to-last interceptor's `intercept_post_dispatch()` function and so on. The first interceptor returns all the way back to the last.

5. The last interceptor's call to `intercept_around_dispatch()` returns, all the way back to the first interceptor.

Sample call sequence

To illustrate the sequence of calls that results when the intercept functions are all used together, consider the chain of three interceptors, `A`, `B`, and `C`, where `A` is the first interceptor in the chain, and `C` is the last. Example 7 shows the sequence of events, where `>>` denotes entering a function and `<<` denotes leaving a function.

Example 7: *Sample Server Interceptor Call Sequence*

```
A >> interceptor_around_dispatch()
  B >> interceptor_around_dispatch()
    C >> interceptor_around_dispatch()
      A >> interceptor_pre_dispatch()
        B >> interceptor_pre_dispatch()
          C >> interceptor_pre_dispatch()
          C << interceptor_pre_dispatch()
        B << interceptor_pre_dispatch()
      A << interceptor_pre_dispatch()
      Application >> invoke()
      Application << invoke()
      C >> interceptor_post_dispatch()
        B >> interceptor_post_dispatch()
          A >> interceptor_post_dispatch()
          A << interceptor_post_dispatch()
        B << interceptor_post_dispatch()
      C << interceptor_post_dispatch()
    C << interceptor_around_dispatch()
  B << interceptor_around_dispatch()
A << interceptor_around_dispatch()
```

intercept_around_dispatch() function

Example 8 shows the basic outline of how to implement the `intercept_around_dispatch()` function.

Example 8: *Outline of intercept_around_dispatch() Function*

```
// C++
using namespace IT_Bus;

void
```

Example 8: *Outline of intercept_around_dispatch() Function*

```
CustomSrvrReqInterceptor::intercept_around_dispatch(
    ServerOperation& data
)
{
    // PRE-UNMARSHAL processing
    // ...

    if (m_next_interceptor != 0) {
        m_next_interceptor->intercept_around_dispatch(data);
    }

    // POST-MARSHAL processing
    // ...
}
```

The typical implementation of `intercept_around_dispatch()` has three main parts:

- *Pre-unmarshal processing*—put any code here that you would want to execute *before* the request is dispatched to the servant object. At this point, the input parts are not yet unmarshalled. Therefore, you cannot access the input parts.
- *Call the next interceptor in the chain*—you must always call `intercept_around_dispatch()` on the next interceptor, as shown here.
- *Post-marshal processing*—put any code here that you would want to execute *after* the servant code has executed. At this point, both the input and output parts are available. You can examine or modify the output parts. Replacing parts has no effect.

---

intercept_pre_dispatch() function

Example 9 shows the basic outline of how to implement the `intercept_pre_dispatch()` function.

Example 9: *Outline of intercept_pre_dispatch() Function*

```
// C++
using namespace IT_Bus;

void
CustomSrvrReqInterceptor::intercept_pre_dispatch(
    ServerOperation& data
)
```

Example 9: *Outline of intercept_pre_dispatch() Function*

```
{
    // PRE-DISPATCH processing
    // ...

    if (m_next_interceptor != 0) {
        m_next_interceptor->intercept_pre_dispatch(data);
    }
}
```

The typical implementation of `intercept_pre_dispatch()` has two main parts:

- *Pre-dispatch processing*—put any code here that you would want to execute *before* the request is dispatched to the servant object. At this point, the input parts are unmarshalled. You can access or modify (but not replace) the input parts.
- *Call the next interceptor in the chain*—you must always call `intercept_pre_dispatch()` on the next interceptor, as shown here.

---

intercept_post_dispatch() function

Example 10 shows the basic outline of how to implement the `intercept_post_dispatch()` function.

Example 10: *Outline of intercept_post_dispatch() Function*

```
// C++
using namespace IT_Bus;

void
CustomSrvrReqInterceptor::intercept_post_dispatch(
    ServerOperation& data
)
{
    // POST-DISPATCH processing
    // ...

    if (m_prev_interceptor != 0) {
        m_prev_interceptor->intercept_post_dispatch(data);
    }
}
```

The typical implementation of `intercept_post_dispatch()` has two main parts:

- *Post-dispatch processing*—put any code here that you would want to execute *after* the request is dispatched to the servant object. At this point, the output parts are initialized. You can access or replace the output parts.
- *Call the previous interceptor in the chain*—you must always call `intercept_post_dispatch()` on the previous interceptor, as shown here.

ServerOperation class

The `data` object that passes along the server request interceptor chain is an instance of the `IT_Bus::ServerOperation` class. The `ServerOperation` class encapsulates the request and reply data.

The most important member functions of the `ServerOperation` class are as follows:

- `get_name()`
  Returns an `IT_Bus::String` that holds the name of the operation that is being dispatched.
- `get_input_message()`
  Returns an `IT_Bus::ReadableMessage` object that contains the input parts. The simplest way to obtain the input parts list is to call `get_input_message().get_parts()`.
- `get_output_message()`
  Returns an `IT_Bus::WritableMessage` object that contains the output parts. The simplest way to obtain the output parts list is to call `get_output_message().get_parts()`.
- `request_contexts()`
  Returns an `IT_Bus::ContextContainer` object that provides access to request contexts. You can use this object to write or read headers in the request message.
- `reply_contexts()`
  Returns an `IT_Bus::ContextContainer` object that provides access to reply contexts. You can use this object to write or read headers in the reply message.

Configuring a server request interceptor

To configure Artix to use a server request interceptor, you must update the server request interceptor list in the Artix configuration file. The server request interceptor list consists of a list of alternative chain configurations, as follows:

```
binding:artix:server_request_interceptor_list = ["Chain01",
    "Chain02", "Chain03", ...];
```

The Artix core first attempts to construct an interceptor chain according to pattern in `Chain01`. If this attempt fails (for example, if one of the interceptors in the chain is unavailable) Artix attempts to use the next chain configuration, `Chain02`, instead.

Each chain configuration is specified in the following format:

```
"InterceptorA+InterceptorB+..."
```

Where `InterceptorA` is the name of interceptor A and `InterceptorB` is the name of interceptor B and so on. An interceptor name is the name under which the interceptor factory is registered with the `IT_Bus::InterceptorFactoryManager`.

Configuring an interceptor in an Artix router

If an interceptor is meant to be used within an Artix router process, you might need to configure the router to ensure the interceptor is not bypassed. Specifically, if you configure a route that maps messages between two bindings of the same type (for example, CORBA-to-CORBA), the router bypasses interceptors by default. This is often a useful optimization, but is unsuitable for some applications.

To force all routed messages to pass through the interceptors in the router, you should add the following line to the router's configuration:

```
plugins:routing:use_pass_through = "false";
```

Sending and Receiving Header Contexts

Overview

You can use Artix interceptors to send and receive header contexts to transmit with operation request and replies. While it is also possible to program header contexts at the application level, there are significant advantages to writing this code at the interceptor level. Header contexts are typically used to send security credentials and other out-of-band data that are not specific to any port type. By putting this common code into an interceptor, you can avoid cluttering your servant code and client code.

In this section

This section contains the following subsections:

SOAP Header Context Example

Overview

The examples in this section are based on the shared library demonstration, which is located in the following Artix directory:

ArtixInstallDir/artix/*Version*/demos/advanced/shared_library

Figure 6 shows an overview of the shared library demonstration, showing how the client piggybacks context data along with an invocation request that is invoked on the `sayHi` operation.

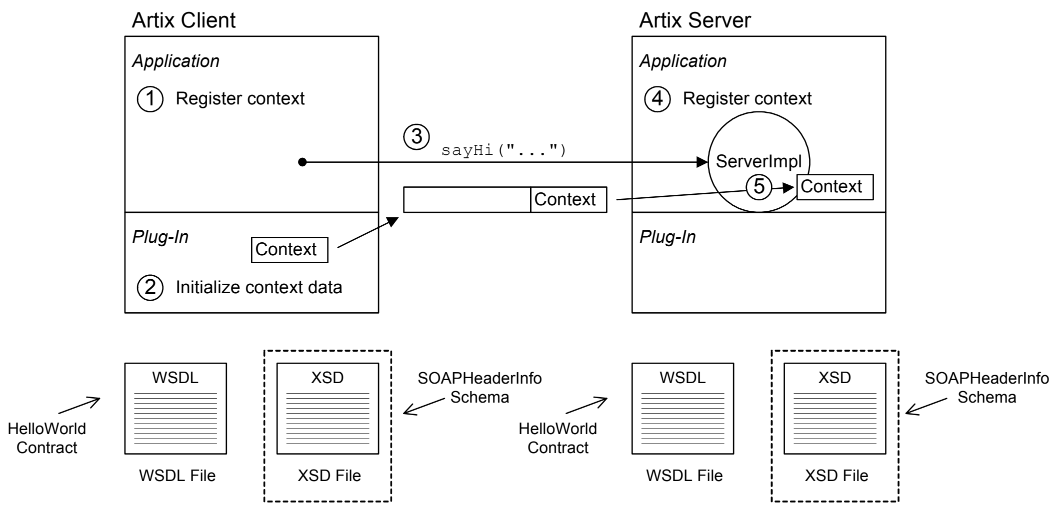

Figure 6: *Overview of the Custom SOAP Header Demonstration*

Transmission of context data

As illustrated in Figure 6, SOAP context data is transmitted as follows:

1. The client registers the context type, `SOAPHeaderInfo`, with the Bus.
2. The client interceptor initializes the context data instance.
3. The client invokes the `sayHi()` operation on the server.
4. As the server starts up, it registers the `SOAPHeaderInfo` context type with the Bus.
5. When the `sayHi()` operation request arrives on the server side, the `sayHi()` operation implementation extracts the context data from the request.

HelloWorld WSDL contract

The HelloWorld WSDL contract defines the contract implemented by the server in this demonstration. In particular, the HelloWorld contract defines the `Greeter` port type containing the `sayHi` WSDL operation.

SOAPHeaderInfo schema

The `SOAPHeaderInfo` schema (in the `demos/advanced/shared_library/etc/contextTypes.xsd` file) defines the custom data type used as the context data type. This schema is specific to the shared library demonstration.

Sample Context Schema

Overview

This subsection describes how to define an XML schema for a context type. In this example, the `SOAPHeaderInfo` type is declared in an XML schema. The `SOAPHeaderInfo` type is then used by the shared library demonstration to send custom data in a SOAP header.

SOAPHeaderInfo XML declaration

Example 11 shows the schema for the `SOAPHeaderInfo` type, which is defined specifically for the shared library demonstration to carry some sample data in a SOAP header. Note that Example 11 is a pure schema declaration, *not* a WSDL declaration.

Example 11: *XML Schema for the SOAPHeaderInfo Context Type*

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://schemas.iona.com/types/context"
    elementFormDefault="qualified"
   attributeFormDefault="unqualified">
    <xs:complexType name="SOAPHeaderInfo">
        <xs:annotation>
            <xs:documentation>
                Content to be added to a SOAP header
            </xs:documentation>
        </xs:annotation>
        <xs:sequence>
            <xs:element name="originator" type="xs:string"/>
            <xs:element name="message" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```

The `SOAPHeaderInfo` complex type defines two member elements, as follows:

- `originator`—holds an arbitrary client identifier.
- `message`—holds an arbitrary example message.

Target namespace

You can use any target namespace for a context schema (as long as it does not clash with an existing namespace). This demonstration uses the following target namespace:

```
http://schemas.iona.com/types/context
```

Compiling the SOAPHeaderInfo schema

To compile the `SOAPHeaderInfo` schema, invoke the `wsdltocpp` compiler utility at the command line, as follows:

```
wsdltocpp -n custom_interceptor contextTypes.xsd
```

Where `contextTypes.xsd` is a file containing the XML schema from Example 11. This command generates the following C++ stub files:

```
contextTypes_xsdTypes.h
contextTypes_xsdTypesFactory.h
contextTypes_xsdTypes.cxx
contextTypes_xsdTypesFactory.cxx
```

SOAPHeaderInfo C++ mapping

Example 12 shows how the schema from Example 11 on page 34 maps to C++, to give the `custom_interceptor::SOAPHeaderInfo` C++ class.

Example 12: *C++ Mapping of the SOAPHeaderInfo Context Type*

```
// C++
...
namespace custom_interceptor
{
    ...
    class SOAPHeaderInfo : public IT_Bus::SequenceComplexType
    {
      public:
        static const IT_Bus::QName type_name;

        SOAPHeaderInfo();
        SOAPHeaderInfo(const SOAPHeaderInfo & copy);
        virtual ~SOAPHeaderInfo();
        ...
        IT_Bus::String &       getoriginator();
        const IT_Bus::String & getoriginator() const;
        void setoriginator(const IT_Bus::String & val);

        IT_Bus::String &       getmessage();
        const IT_Bus::String & getmessage() const;
        void setmessage(const IT_Bus::String & val);
        ...
```

Example 12: *C++ Mapping of the SOAPHeaderInfo Context Type*

```
    };
    ...
}
```

Implementation of the Client Request Interceptor

Overview

A client request interceptor performs processing on the client operation object which passes through the client interceptor chain. You implement the `intercept_invoke()` operation (called by the preceding interceptor in the chain) to perform request processing.

The ClientRequestInterceptor base class

Example 13 shows the declarations of the `IT_Bus::Interceptor` class and the `IT_Bus::ClientRequestInterceptor` class, which is the base class for a client request interceptor. The member functions that must be implemented by derived classes are highlighted in bold font.

Example 13: *The IT_Bus::ClientRequestInterceptor Class*

```
// C++
// In file: it_bus_pdk/interceptor.h
...
namespace IT_Bus {
    enum InterceptorType
    {
        CPP_INTERCEPTOR,
        JAVA_INTERCEPTOR
    };

1   class IT_BUS_API Interceptor
    {
      public:
        Interceptor();
        Interceptor(InterceptorFactory* factory);
        virtual ~Interceptor();

        virtual InterceptorFactory* get_factory();
        virtual InterceptorType get_type();

      private:
        InterceptorFactory* m_factory;
    };
```

Example 13: *The IT_Bus::ClientRequestInterceptor Class*

2
```
    class IT_BUS_API ClientRequestInterceptor
      : public Interceptor
    {
    public:
        ClientRequestInterceptor();
        ClientRequestInterceptor(InterceptorFactory* factory);
        virtual ~ClientRequestInterceptor();

        virtual void
        chain_assembled(ClientRequestInterceptorChain& chain);

        virtual void
        chain_finalized(
            ClientRequestInterceptor* next_interceptor
        );

        virtual void
        intercept_invoke(ClientOperation& data);

    protected:
        ClientRequestInterceptor* m_next_interceptor;
    };
};
```

The preceding code can be explained as follows:

1. The `IT_Bus::Interceptor` class is the common base class for all interceptor types.
2. The `IT_Bus::ClientRequestInterceptor` class, which inherits from `IT_Bus::Interceptor`, is the base class for client request interceptors.

C++ client request interceptor header

Example 14 shows the declaration of the `IT_SampleArtixInterceptor::ClientInterceptor` class, which is derived from the `IT_Bus::ClientRequestInterceptor` class.

Example 14: *Sample Client Request Interceptor Header File*

```
// C++
// In file: demos/advanced/shared_library/
//                                  cxx/plugin/client_interceptor.h

#include <it_bus/qname.h>
#include <it_bus/bus.h>
```

Example 14: *Sample Client Request Interceptor Header File*

```
#include <it_bus_pdk/interceptor.h>
#include <it_cal/cal.h>

namespace IT_SampleArtixInterceptor
{
1       class ClientInterceptor :
            public virtual IT_Bus::ClientRequestInterceptor
        {
          public:
            ClientInterceptor(
                IT_Bus::Bus_ptr      bus
            );

            virtual ~ClientInterceptor();

            virtual void
            intercept_invoke(IT_Bus::ClientOperation& data);

          private:
            ClientInterceptor&
            operator = (const ClientInterceptor& rhs);

            ClientInterceptor(const ClientInterceptor& rhs);

2           IT_Bus::Bus_ptr m_bus;
        };
};
```

The preceding code can be explained as follows:

1. The `ClientInterceptor` implementation class inherits from the `IT_Bus::ClientRequestInterceptor` base class.
2. The `m_bus` member variable stores a reference to the Bus object.

C++ client request interceptor implementation

Example 15 shows the implementation of the `IT_SampleArtixInterceptor::ClientInterceptor` class.

Example 15: *Sample Client Request Interceptor Implementation*

```
// C++
// In file: demos/advanced/shared_library/
//                              cxx/plugin/client_interceptor.cxx
```

Example 15: *Sample Client Request Interceptor Implementation*

```
// Include header files related to the soap context
#include <it_bus/operation.h>
#include <it_bus_pdk/context.h>

// Include header files representing the soap header content
#include "../types/contextTypes_xsdTypes.h"
#include "../types/contextTypes_xsdTypesFactory.h"

#include "client_interceptor.h"

IT_USING_NAMESPACE_STD
using namespace custom_interceptor;

using namespace IT_Bus;
using namespace IT_WSDL;
using namespace IT_SampleArtixInterceptor;

1   ClientInterceptor::ClientInterceptor(
        Bus_ptr      bus
    )
        : m_bus(bus)
    {
    }

    ClientInterceptor::~ClientInterceptor()  { }

    void
2   ClientInterceptor::intercept_invoke(ClientOperation& data)
    {
        cout << "\tClient interceptor intercept_invoke method"
             << "\tOperation called: " << data.get_name()
             << endl;

3       // -----> PRE-INVOKE processing comes here <-----
        // For the sayHi operation, change the originator and message
4       if (data.get_name() == "sayHi")
        {
            // Obtain a pointer to the bus
            Bus_var bus = Bus::create_reference();

            // Use the bus to obtain a pointer to the ContextRegistry
            //   created by the soap plugin
            ContextRegistry* context_registry =
                bus->get_context_registry();
```

Example 15: *Sample Client Request Interceptor Implementation*

```
        // Create QName objects needed to define a context
        const QName principal_ctx_name(
            "",
            "SOAPHeaderInfo",
            ""
        );

        // Obtain a pointer to the RequestContextContainer
5       ContextContainer* context_container =
            data.request_contexts();

        // Obtain a reference to the context
6       AnyType* info = context_container->get_context(
            principal_ctx_name,
            true
        );

        if (0 == info)
        {
            throw Exception("Could not access Context");
        }

        // Cast the context into a SOAPHeaderInfo object
7       SOAPHeaderInfo* header_info =
            dynamic_cast<SOAPHeaderInfo*> (info);

        if (0 == header_info)
        {
            throw Exception("Could not cast Context");
        }

        // Create the content to be added to the header
        const String originator("Artix Engineering");
        const String message("We are Great!");

        // Add the header content
        cout << "\tSetting SOAP header with originator: "
           << originator << " and message: " << message << endl;
8       header_info->setoriginator(originator);
        header_info->setmessage(message);
    }

    if (ClientRequestInterceptor::m_next_interceptor != 0)
    {
```

Example 15: *Sample Client Request Interceptor Implementation*

```
9
    ClientRequestInterceptor::m_next_interceptor->intercept_invok
    e(data);
     }
10   // -----> POST-INVOKE processing comes here <-----
}
```

The preceding code can be explained as follows:

1. The `ClientInterceptor` constructor is called by the interceptor factory at the time the interceptor chain is constructed (see “Implementation of the Interceptor Factory” on page 50). Here you should initialize a local reference to the Bus, `m_bus`, and the interceptor name, `m_name`.
2. The `intercept_invoke()` function is the key function in the client request interceptor. This is the point at which you can intercept and affect an operation invocation.
3. At this point (prior to invoking `intercept_invoke()` on the next interceptor), you can add in any processing that needs to complete *before* invoking the WSDL operation.
4. The interceptor modifies the context only for the `sayHi` operation from the `Greeter` port type.
5. The interceptor obtains a reference to the context container for outgoing requests.
6. Get a pointer to the context identified by the `SOAPHeaderInfo` QName. If an instance of this context does not already exist, the `get_context()` function creates a new one (indicated by setting the second parameter to `true`).
7. Cast the `IT_Bus::AnyType*` variable from the previous step, `info`, to the `SOAPHeaderInfo*` variable, `header_info`.
8. Set the originator and message attributes on the `SOAPHeaderInfo` instance, `header_info`.
9. Invoke `intercept_invoke()` on the next interceptor in the chain. This step is mandatory for almost all interceptors (a possible exception being a security interceptor that decides to prevent an invocation from proceeding).

10. At this point (after invoking `intercept_invoke()` on the next interceptor), you can add in any processing that needs to occur *after* invoking the WSDL operation.

Implementation of the Server Request Interceptor

Overview

A server request interceptor performs processing on the server operation object which passes through the server interceptor chain. You must implement the following functions to intercept incoming requests:

- `intercept_pre_dispatch()`
- `intercept_post_dispatch()`

The ServerRequestInterceptor base class

Example 16 shows the declarations of the `IT_Bus::Interceptor` class and the `IT_Bus::ServerRequestInterceptor` class, which is the base class for a server request interceptor. The member functions that must be implemented by derived classes are highlighted in bold font.

Example 16: *The IT_Bus::ServerRequestInterceptor Class*

```
// C++
// In file: it_bus_pdk/interceptor.h
...
namespace IT_Bus {
    enum InterceptorType
    {
        CPP_INTERCEPTOR,
        JAVA_INTERCEPTOR
    };

1   class IT_BUS_API Interceptor
    {
      public:
        Interceptor();
        Interceptor(InterceptorFactory* factory);
        virtual ~Interceptor();

        virtual InterceptorFactory* get_factory();
        virtual InterceptorType get_type();

      private:
        InterceptorFactory* m_factory;
    };

2   class IT_BUS_API ServerRequestInterceptor
      : public Interceptor
```

Example 16: *The IT_Bus::ServerRequestInterceptor Class*

```
    {
    public:
        ServerRequestInterceptor();
        ServerRequestInterceptor(InterceptorFactory* factory);
        virtual ~ServerRequestInterceptor();

        virtual void
        chain_assembled(ServerRequestInterceptorChain& chain);

        virtual void
        chain_finalized(
            ServerRequestInterceptor* next_interceptor
        );

        virtual void
        intercept_pre_dispatch(ServerOperation& data);

        virtual void
        intercept_post_dispatch(ServerOperation& data);

        virtual void
        intercept_around_dispatch(ServerOperation& data);

    protected:
3       ServerRequestInterceptor* m_next_interceptor;
        ServerRequestInterceptor* m_prev_interceptor;
    };
};
```

The preceding code can be explained as follows:

1. The `IT_Bus::Interceptor` class is the common base class for all interceptor types.
2. The `IT_Bus::ServerRequestInterceptor` class, which inherits from `IT_Bus::Interceptor`, is the base class for server request interceptors.
3. The server request interceptor stores references both to the next interceptor and the previous interceptor in the chain. A server request interceptor chain is thus a doubly linked list.

C++ server request interceptor header

Example 17 shows the declaration of the `IT_SampleArtixInterceptor::ServerInterceptor` class, which is derived from the `IT_Bus::ServerRequestInterceptor` class.

Example 17: *Sample Server Request Interceptor Header File*

```
// C++
// In file: demos/advanced/shared_library/
//                                cxx/plugin/server_interceptor.h

#include <it_bus/qname.h>
#include <it_bus/bus.h>
#include <it_bus_pdk/interceptor.h>

namespace IT_SampleArtixInterceptor
{
1       class ServerInterceptor :
            public virtual IT_Bus::ServerRequestInterceptor
    {
      public:
        ServerInterceptor(
            IT_Bus::Bus_ptr      bus
        );

        virtual ~ServerInterceptor();

        virtual void
        intercept_pre_dispatch(IT_Bus::ServerOperation& data);

        virtual void
        intercept_post_dispatch(IT_Bus::ServerOperation& data);

      private:
        ServerInterceptor&
        operator = (const ServerInterceptor& rhs);

        ServerInterceptor(const ServerInterceptor& rhs);

2           IT_Bus::Bus_ptr                          m_bus;
    };
};
```

The preceding code can be explained as follows:

1. The `ServerInterceptor` implementation class inherits from the `IT_Bus::ServerRequestInterceptor` base class.
2. The `m_bus` member variable stores a reference to the Bus object.

C++ server request interceptor implementation

Example 18 shows the implementation of the `IT_SampleArtixInterceptor::ServerInterceptor` class.

Example 18: *Sample Server Request Interceptor Implementation*

```
  // C++
  // In file: demos/advanced/custom_interceptor/
  //
     cxx/plugin/server_interceptor.cxx
  #include "server_interceptor.h"

  using namespace IT_Bus;
  using namespace IT_WSDL;
  using namespace IT_SampleArtixInterceptor;

  IT_USING_NAMESPACE_STD

1 ServerInterceptor::ServerInterceptor(
      Bus_ptr      bus
  )
      : m_bus(bus)
  {
  }

  ServerInterceptor::~ServerInterceptor()  { }

  void
2 ServerInterceptor::intercept_pre_dispatch(
      IT_Bus::ServerOperation& data
  )
  {
3     cout << "\tServer interceptor intercept_pre_dispatch invoked"
           << "\tOperation called: " << data.get_name() << endl;
4     // -----> PRE-INVOKE processing comes here <-----

      if (ServerRequestInterceptor::m_next_interceptor != 0)
      {
5 ServerRequestInterceptor::m_next_interceptor->intercept_pre_disp
     atch(data);
```

Example 18: *Sample Server Request Interceptor Implementation*

```
        }
    }

    void
6   ServerInterceptor::intercept_post_dispatch(
        IT_Bus::ServerOperation& data
    )
    {
        cout << "\tServer interceptor intercept_post_dispatch "
             << "invoked \tReturn from operation: "
             << data.get_name() << endl;
7       // -----> POST-INVOKE processing comes here <-----

        if (ServerRequestInterceptor::m_prev_interceptor != 0)
        {
8   ServerRequestInterceptor::m_prev_interceptor->intercept_post_dis
       patch(data);

        }
    }
```

The preceding code can be explained as follows:

1. The `ServerInterceptor` constructor is called by the interceptor factory at the time the interceptor chain is constructed (see "Implementation of the Interceptor Factory" on page 50). Here you should initialize a local reference to the Bus, `m_bus`, and the interceptor name, `m_name`.
2. The `intercept_pre_dispatch()` function is called before the incoming request has been dispatched to the service endpoint. This key function gives you a chance to access the request before it is executed on the server side.
3. Print the name of the invoked WSDL operation to standard output. For simplicity, in this demonstration the operation name is printed using `cout`. In general, however, it is better practice to use the Artix logging feature.
4. At this point (prior to invoking `intercept_pre_dispatch()` on the next interceptor), you can add any processing that needs to complete *before* invoking the WSDL operation.

5. Invoke `intercept_pre_dispatch()` on the next interceptor in the chain. This step is mandatory for almost all interceptors (a possible exception being a security interceptor that decides to prevent an invocation from proceeding).
6. The `intercept_post_dispatch()` function is called after the incoming request has been dispatched to the service endpoint, but before the output parts have been marshalled.
7. The post-invoke processing should *precede* the call on the next interceptor in the chain.
8. Invoke `intercept_post_dispatch()` on the previous interceptor in the chain. This step is mandatory.

Implementation of the Interceptor Factory

Overview

Artix uses a factory pattern to manage the lifecycle of interceptor objects. To install a set of interceptors, you must implement an interceptor factory and register an instance of this factory with the interceptor factory manager object. The interceptor factory exposes functions that the Artix runtime can then call to create new interceptor instances.

Request interceptors are created by the following functions:

- `get_client_request_interceptor()`
- `get_server_request_interceptor()`

Message interceptors are created by the following functions:

- `get_client_message_interceptor()`
- `get_server_message_interceptor()`

If a particular kind of interceptor is not implemented, you can indicate this with a return value of `0`. The interceptor is then omitted from the chain.

The InterceptorFactory base class

Example 19 shows the declarations of the `IT_Bus::InterceptorFactory` class, which is the base class for an interceptor factory.

Example 19: *The IT_Bus::InterceptorFactory Class*

```
// C++
// In file: it_bus_pdk/interceptor.h
...
namespace IT_Bus {
    class IT_BUS_API InterceptorFactory
    {
      public:
        virtual ClientMessageInterceptor *
        get_client_message_interceptor(
            const IT_WSDL::WSDLNode* const wsdl_node = 0
        );

        virtual void destroy_client_message_interceptor(
            ClientMessageInterceptor * message_interceptor
        );
```

Example 19: *The IT_Bus::InterceptorFactory Class*

```
        virtual ClientRequestInterceptor *
        get_client_request_interceptor(
            const IT_WSDL::WSDLNode* const wsdl_node = 0
        );

        virtual void destroy_client_request_interceptor(
            ClientRequestInterceptor * request_interceptor
        );

        virtual ServerMessageInterceptor*
        get_server_message_interceptor(
            const IT_WSDL::WSDLNode* const wsdl_node = 0
        );

        virtual void destroy_server_message_interceptor(
            ServerMessageInterceptor* message_interceptor
        );

        virtual ServerRequestInterceptor*
        get_server_request_interceptor(
            const IT_WSDL::WSDLNode* const wsdl_node = 0
        );

        virtual void destroy_server_request_interceptor(
            ServerRequestInterceptor* request_interceptor
        );

        virtual const String& name() = 0;

      protected:
        ...
    };
};
```

C++ interceptor factory header

Example 20 shows the declaration of the `IT_SampleArtixInterceptor::SampleBusPlugIn` class, which implements the `IT_Bus::InterceptorFactory` class.

Example 20: *Sample Interceptor Factory Header File*

```
// C++
// In file: demos/advanced/shared_library/
//                                 cxx/plugin/plugin.cxx
```

Example 20: *Sample Interceptor Factory Header File*

```
...
namespace IT_SampleArtixInterceptor
{
1       class SampleBusPlugIn :
            public IT_Bus::BusPlugIn,
            public IT_Bus::InterceptorFactory
        {
          public:
            IT_EXPLICIT
            SampleBusPlugIn(
                IT_Bus::Bus_ptr bus
            ) IT_THROW_DECL((IT_Bus::Exception));

            virtual ~SampleBusPlugIn();

2           // IT_Bus::BusPlugIn
            //
            ...    // Not shown.

3           //IT_Bus::InterceptorFactory
            //
            virtual IT_Bus::ClientMessageInterceptor *
            get_client_message_interceptor(
                const IT_WSDL::WSDLNode* const wsdl_node = 0
            );

            virtual void destroy_client_message_interceptor(
               IT_Bus::ClientMessageInterceptor* message_interceptor
            );

            virtual IT_Bus::ClientRequestInterceptor *
            get_client_request_interceptor(
                const IT_WSDL::WSDLNode* const wsdl_node = 0
            );

            virtual void destroy_client_request_interceptor(
              IT_Bus::ClientRequestInterceptor * request_interceptor
            ) ;

            virtual IT_Bus::ServerMessageInterceptor*
            get_server_message_interceptor(
                const IT_WSDL::WSDLNode* const wsdl_node = 0
            );
```

Example 20: *Sample Interceptor Factory Header File*

```
        virtual void destroy_server_message_interceptor(
            IT_Bus::ServerMessageInterceptor* message_interceptor
        );

        virtual IT_Bus::ServerRequestInterceptor*
        get_server_request_interceptor(
            const IT_WSDL::WSDLNode* const wsdl_node = 0
        );

        virtual void destroy_server_request_interceptor(
            IT_Bus::ServerRequestInterceptor* request_interceptor
        );

        virtual const IT_Bus::QName& name();

      private:
        SampleBusPlugIn(const SampleBusPlugIn&);

        SampleBusPlugIn&
        operator=(const SampleBusPlugIn&);

4       IT_Bus::String m_name;
    };
};
```

The preceding code can be explained as follows:

1. In this example, the `IT_Bus::InterceptorFactory` base class is implemented by the plug-in class, `SampleBusPlugIn`. If you prefer, you could implement `IT_Bus::InterceptorFactory` using a separate class instead.
2. The implementation of the functions inherited from the `IT_Bus::BusPlugIn` base class is discussed in another chapter—see “Basic Plug-In Implementation” on page 1.
3. From this point on, all of the functions shown are inherited from `IT_Bus::InterceptorFactory`.
4. The `m_name` variable is used to store the interceptor name.

C++ interceptor factory implementation

Example 21 shows the implementation of the `IT_SampleArtixInterceptor::SampleBusPlugIn` class.

Example 21: *Sample Interceptor Factory Implementation*

```
// C++

using namespace IT_Bus;
using namespace IT_WSDL;
using namespace IT_SampleArtixInterceptor;

//  SampleBusPlugIn
//

SampleBusPlugIn:: SampleBusPlugIn(
    IT_Bus::Bus_ptr bus
) IT_THROW_DECL((IT_Bus::Exception))
  :
    BusPlugIn(bus),
    m_name("artix_shlib_interceptor")
{
    assert(bus != 0);
}

SampleBusPlugIn::~SampleBusPlugIn()  { }

// IT_Bus::BusPlugIn functions
//
void
SampleBusPlugIn::bus_init(
) IT_THROW_DECL((IT_Bus::Exception))
{
    IT_Bus::Bus_ptr bus = get_bus();
    assert(bus != 0);

1   InterceptorFactoryManager& factory_manager =
        bus->get_pdk_bus()->get_interceptor_factory_manager();

2   factory_manager.register_interceptor_factory(
        m_name,
        this
    );
}
```

Example 21: *Sample Interceptor Factory Implementation*

```
void
SampleBusPlugIn::bus_shutdown(
) IT_THROW_DECL((IT_Bus::Exception))
{
    IT_Bus::Bus_ptr bus = get_bus();
    assert(bus != 0);

    InterceptorFactoryManager& factory_manager =
        bus->get_pdk_bus()->get_interceptor_factory_manager();

3       factory_manager.unregister_interceptor_factory(
        this
    );
}

// IT_Bus::InterceptorFactory functions
//
ClientMessageInterceptor *
4 SampleBusPlugIn::get_client_message_interceptor(
    const WSDLNode* const
)
{
    return 0;
}

void
5 SampleBusPlugIn::destroy_client_message_interceptor(
    ClientMessageInterceptor* message_interceptor
)
{
    delete message_interceptor;
}

ClientRequestInterceptor *
6 SampleBusPlugIn::get_client_request_interceptor(
    const WSDLNode* const
)
{
    return new ClientInterceptor(get_bus());
}

void
7 SampleBusPlugIn::destroy_client_request_interceptor(
    ClientRequestInterceptor * request_interceptor
)
```

Example 21: *Sample Interceptor Factory Implementation*

```
{
    delete request_interceptor;
}

ServerMessageInterceptor*
SampleBusPlugIn::get_server_message_interceptor(
    const WSDLNode* const
)
{
    return 0;
}

void
SampleBusPlugIn::destroy_server_message_interceptor(
    ServerMessageInterceptor* message_interceptor
)
{
    delete message_interceptor;
}

ServerRequestInterceptor*
8 SampleBusPlugIn::get_server_request_interceptor(
    const WSDLNode* const
)
{
    return new ServerInterceptor(get_bus());
}

void
9 SampleBusPlugIn::destroy_server_request_interceptor(
    ServerRequestInterceptor* request_interceptor
)
{
    delete request_interceptor;
}

const String&
10 SampleBusPlugIn::name()
{
    return m_name;
}
```

The preceding code can be explained as follows:

1. The `IT_Bus::InterceptorFactoryManager` object stores a list of all interceptor factories. It is implemented by the Artix runtime.
2. You must register the interceptor factory instance with the interceptor factory manager, as shown here. The register function takes the interceptor name, `m_name`, and the interceptor factory instance, `this`, as arguments.
3. You usually unregister the interceptor factory in the body of the `IT_Bus::BusPlugIn::bus_shutdown()` function to ensure a clean shutdown of the Artix Bus.
4. You would implement the `get_client_message_interceptor()` function to install a client message interceptor. In this example, the function returns 0 to indicate that a client message interceptor is not available.
5. The `destroy_client_message_interceptor()` function would be called by the Artix runtime to clean up resources associated with the client message interceptor.
6. The Artix runtime calls `get_client_request_interceptor()` in the course of constructing a new interceptor chain to obtain a client request interceptor instance.

   The `get_client_request_interceptor()` function takes the following arguments:

   - `wsdl_node`—(defaults to 0).

   In this example, the implementation of `get_client_request_interceptor()` simply returns a new client interceptor object.
7. The `destroy_client_request_interceptor()` function is called by the Artix runtime to clean up resources associated with the client request interceptor.
8. The Artix runtime calls `get_server_request_interceptor()` in the course of constructing a new interceptor chain to obtain a server request interceptor instance.

The `get_server_request_interceptor()` function takes the following arguments:

- `wsdl_node`—(defaults to 0).

In this example, the implementation of `get_server_request_interceptor()` simply returns a new server interceptor object.

9. The `destroy_server_request_interceptor()` function is called by the Artix runtime to clean up resources associated with the server request interceptor.
10. The `name()` function returns the interceptor name.

Accessing and Modifying Parameters

Overview

Artix interceptors enable you to access and modify both input and output parameters, as a message passes back and forth along the interceptor chain. On the client side, the input and output parameters are accessible from the `IT_Bus::ClientOperation` object. On the server side, the input and output parameters are accessible from the `IT_Bus::ServerOperation` object.

In this section

This section contains the following subsections:

Reflection Example

Overview

In order to access and modify operation parameters from within an interceptor, it is essential to use the Artix reflection API. In contrast to code written at the application level, an interceptor must typically be able to process any port type or operation. Hence, an interceptor implementation must be able to parse any parameter type; this capability is provided by the Artix reflection API.

To access operation parameters from within an interceptor, you would typically need to use the following APIs:

- Part list type.
- Reflection API.

Part list type

Given either an `IT_Bus::ClientRequestInterceptor` instance or an `IT_Bus::ServerRequestInterceptor` instance, `data`, you can access the input parts and the output parts as follows:

- To obtain a reference to the *input* part list, call:

  `data.get_input_message().get_parts()`
- To obtain a reference to the *output* part list, call:

  `data.get_output_message().get_parts()`

The returned part list (of `IT_Bus::PartList&` type) is essentially a vector of `(IT_Bus::QName, IT_Bus::AnyType*)` pairs.

Reflection API

The reflection API enables you to parse any Artix data type and to process the data without any advance knowledge of its type. For the example described in this section, you need only the following classes:

- `IT_Reflect::Reflection` class—the base class for all reflection types.
- `IT_Reflect::Value<IT_Bus::String>` class—the reflection type that represents a string.
- `IT_Bus::Var<T>` template—a smart pointer template type that ensures that the referenced data is not leaked.

Reflection interceptor demonstration

The sample code in this section is taken from the following Artix demonstration:

ArtixInstallDir/artix/*Version*/demos/reflection/interceptor

Example 22 shows the WSDL definition of the `Greeter` port type that is used in this demonstration.

Example 22: *The Greeter Port Type*

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
    name="HelloWorld"
    targetNamespace="http://www.iona.com/reflect_interceptor"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://www.iona.com/reflect_interceptor"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema" ...>
    <wsdl:types>
        <schema
   targetNamespace="http://www.iona.com/reflect_interceptor"
            xmlns="http://www.w3.org/2001/XMLSchema">
            <element name="responseType" type="xsd:string"/>
            <element name="requestType" type="xsd:string"/>
        </schema>
    </wsdl:types>

    <wsdl:message name="sayHiRequest"/>
    <wsdl:message name="sayHiResponse">
        <wsdl:part element="tns:responseType"
   name="theResponse"/>
    </wsdl:message>
    <wsdl:message name="greetMeRequest">
        <wsdl:part element="tns:requestType" name="me"/>
    </wsdl:message>
    <wsdl:message name="greetMeResponse">
        <wsdl:part element="tns:responseType"
   name="theResponse"/>
    </wsdl:message>

    <wsdl:portType name="Greeter">
        <wsdl:operation name="sayHi">
            <wsdl:input message="tns:sayHiRequest"
   name="sayHiRequest"/>
            <wsdl:output message="tns:sayHiResponse"
   name="sayHiResponse"/>
        </wsdl:operation>
```

Example 22: *The Greeter Port Type*

```
        <wsdl:operation name="greetMe">
            <wsdl:input message="tns:greetMeRequest"
   name="greetMeRequest"/>
            <wsdl:output message="tns:greetMeResponse"
   name="greetMeResponse"/>
        </wsdl:operation>
    </wsdl:portType>
    ...
</wsdl:definitions>
```

Implementation of the Client Request Interceptor

Overview

This subsection describes how to implement a client request interceptor that uses reflection to modify an operation's input and output parameters.

> **Note:** This example is only intended to be used in conjunction with the `Greeter` port type, as defined in Example 22 on page 61.

C++ client request interceptor header

Example 23 shows the header for the `ClientInterceptor` class, which is derived from the `IT_Bus::ClientRequestInterceptor` base class.

Example 23: *Client Interceptor Header for Reflection Example*

```
// C++
#include <it_bus/bus.h>
#include <it_bus/qname.h>
#include <it_bus_pdk/interceptor.h>

class ClientInterceptor :
    public virtual IT_Bus::ClientRequestInterceptor
{
  public:
    ClientInterceptor(
        IT_Bus::Bus_ptr      bus
    );

    virtual ~ClientInterceptor();

    virtual void
    intercept_invoke(
        IT_Bus::ClientOperation& data
    );

  private:
    IT_Bus::Bus_ptr m_bus;
};
```

C++ client request interceptor implementation

Example 24 shows the implementation of the `ClientInterceptor` class.

Example 24: *Client Interceptor Implementation for Reflection Example*

```
// C++
#include "client_interceptor.h"
#include <it_bus/operation.h>
#include <it_bus/part_list.h>
#include <it_bus/reflect/value.h>
#include <it_cal/iostream.h>

IT_USING_NAMESPACE_STD;
using namespace IT_Bus;

ClientInterceptor::ClientInterceptor(
    Bus_ptr      bus
)
    : m_bus(bus)
{
    // Complete
}

ClientInterceptor::~ClientInterceptor()
{
    // Complete
}

void
1 ClientInterceptor::intercept_invoke(
    ClientOperation& data
)
{
    // Get the value of the input part using reflection.
    // Client-side input parts are "serializable" that is they
    // will be serialized to the underlying transport.
    // Serializable parts are read-only.
    //
2    PartList& input_parts = data.get_input_message().get_parts();
3    if (input_parts.size() == 1)
    {
4        Var<const IT_Reflect::Reflection> r =
            input_parts[0].get_const_value().get_reflection();
5        Var<const IT_Reflect::Value<String> > input_reflection =
           dynamic_cast_var<const IT_Reflect::Value<String> >(r);
        assert(input_reflection.get());
```

Example 24: *Client Interceptor Implementation for Reflection Example*

```
        String input_string = input_reflection->get_value();

        // Print a message
        //
6       String replace_input = input_string + ",1";
        cout << "[Client pre-invoke intercepted: "
             << input_string << "]" << endl;
        cout << "[Replacing with " << replace_input << "]" <<
   endl;

        // Replace the part before calling next interceptor.
        //
7       set_const_value(input_parts[0], replace_input);
    }

    // Call the next interceptor
    //
8   m_next_interceptor->intercept_invoke(data);

    // Get the value of the output string using reflection.
    //
    PartList& output_parts =
   data.get_output_message().get_parts();
9   if (output_parts.size() == 1)
    {
        Var<IT_Reflect::Reflection> r2 =

   output_parts[0].get_modifiable_value().get_reflection();
        Var<IT_Reflect::Value<String> > output_reflection =
            dynamic_cast_var<IT_Reflect::Value<String> >(r2);
        assert(output_reflection.get());
        String output_string = output_reflection->get_value();

        // Print a messsage
        //
        String replace_output = output_string + ",4";
        cout << "[Client post-invoke intercepted: " <<
   output_string << "]"
             << endl;
        cout << "[Replacing with " << replace_output << "]" <<
   endl;

        // Modify the value of the output part. This directly
        // modifies the underlying application data value.
        //
```

Example 24: *Client Interceptor Implementation for Reflection Example*

```
            output_reflection->set_value(replace_output);
        }
}
```

The preceding interceptor implementation can be explained as follows:

1. This implementation of `intercept_invoke()` is designed to modify the parameters of the `sayHi` and `greetMe` WSDL operations by adding a short string to the input parameter and to the output parameter.
2. The returned part list, `input_parts`, contains all of the WSDL parts containing input parameters for the operation. A part list is essentially a vector of `(IT_Bus::QName, IT_Bus::AnyType*)` pairs. The `IT_Bus::AnyType` is the base type for all WSDL types in Artix.
3. The code in this `if`-block uses reflection to modify the first input part. This example is hard-coded to work *only* with the `sayHi` and `greetMe` operation from the `Greeter` port type. The example modifies the request message, only if it consists of a single part which is a string.
4. From the first (and only) pair in the part list, return the `const IT_Bus::AnyType` value (using `get_const_value()`) and convert it into a reflection object (using `get_reflection()`).
5. Assuming that the part contains a string, cast the reflection object to a string reflection.

   This step is only intended to work for the `Greeter` port type. In the general case, you would have to use the reflection interface to figure out the data type.
6. Define a modified string, `replace_input`, which adds `,1` to the original string.

7. Call `set_const_value()` to replace the sole input part in the request. The `set_const_value()` function is a convenience template, which is used only for simple types. It is defined in `it_bus/part.h` as follows:

```
// C++
namespace IT_Bus {
    template <class T>
    void set_const_value(
        Part&        part,
        T&           value
    )
    {
        part.set_const_value(
           new AnySimpleTypeT<T>(value), Part::AUTO_DELETE);
    }
}
```

   The `IT_Bus::Part::set_const_value()` function takes an `IT_Bus::AnyType` as its first parameter. Because simple atomic types, such as `IT_Bus::String`, do not inherit from `AnyType`, it is necessary to wrap them in an `IT_Bus::AnySimpleTypeT<T>` instance, which does inherit from `AnyType`.

   For user-defined types (and other types that inherit from `AnyType`), you can pass them directly to the `IT_Bus::Part::set_const_value()` function.

8. The obligatory call to delegate to the next interceptor in the chain.
9. In the reply message, modify the output, only if it consists of a single part containing a string (intended for the `Greeter` port type only).

Implementation of the Server Request Interceptor

Overview

This subsection describes how to implement a server request interceptor that uses reflection to modify an operation's input and output parameters.

Note: This example is only intended to be used in conjunction with the `Greeter` port type, as defined in Example 22 on page 61.

C++ server request interceptor header

Example 25 shows the header for the `ServerInterceptor` class, which is derived from the `IT_Bus::ServerRequestInterceptor` base class.

Example 25: *Server Interceptor Header for Reflection Example*

```
// C++
#include <it_bus/qname.h>
#include <it_bus/bus.h>
#include <it_bus_pdk/interceptor.h>

class ServerInterceptor :
    public virtual IT_Bus::ServerRequestInterceptor
{
  public:
    ServerInterceptor(
        IT_Bus::Bus_ptr      bus
    );

    virtual ~ServerInterceptor();

    virtual void
    intercept_pre_dispatch(
        IT_Bus::ServerOperation& data
    );

    virtual void
    intercept_post_dispatch(
        IT_Bus::ServerOperation& data
    );

  private:
    IT_Bus::Bus_ptr                          m_bus;
};
```

C++ server request interceptor implementation

Example 26 shows the implementation of the `ServerInterceptor` class.

Example 26: *Server Interceptor Implementation for Reflection Example*

```
// C++
#include <it_bus/operation.h>
#include <it_bus/reflect/value.h>
#include <it_bus/part_list.h>
#include "server_interceptor.h"

using namespace IT_Bus;
using namespace IT_WSDL;
IT_USING_NAMESPACE_STD

ServerInterceptor::ServerInterceptor(
    Bus_ptr      bus
)
    : m_bus(bus)
{
    // Complete.
}

ServerInterceptor::~ServerInterceptor()
{
    // Complete.
}

void
1 ServerInterceptor::intercept_pre_dispatch(
    IT_Bus::ServerOperation& data
)
{
    // Get the value of the input string using reflection.
    // The value points to the value unmarshalled from the wire.
    //
2   PartList& input_parts = data.get_input_message().get_parts();
3   if (input_parts.size() == 1)
    {
4       Var<IT_Reflect::Reflection> r =
          input_parts[0].get_modifiable_value().get_reflection();
5       Var<IT_Reflect::Value<String> > input_reflection =
            dynamic_cast_var<IT_Reflect::Value<String> >(r);
        assert(input_reflection.get());
        String input_string = input_reflection->get_value();
```

Example 26: *Server Interceptor Implementation for Reflection Example*

```
        // Print a messsage
        //
6       String replace_input = input_string + ",2";
        cout << "[Server pre-invoke intercepted: "
             << input_string << "]" << endl;
        cout << "[Replacing with " << replace_input << "]"
             << endl;

        // Modify the value of the input part before the server
        // sees it.
7       input_reflection->set_value(replace_input);
    }

    if (m_next_interceptor != 0)
    {
        m_next_interceptor->intercept_pre_dispatch(data);
    }
}

void
8 ServerInterceptor::intercept_post_dispatch(
    IT_Bus::ServerOperation& data
)
{
    // Get the value of the output part using reflection.
    //
    PartList& output_parts =
   data.get_output_message().get_parts();
9   if (output_parts.size() == 1)
    {
        Var<const IT_Reflect::Reflection> r =
            output_parts[0].get_const_value().get_reflection();
       Var<const IT_Reflect::Value<String> > output_reflection =
            dynamic_cast_var<const IT_Reflect::Value<String>
   >(r);
        assert(output_reflection.get());
        String output_string = output_reflection->get_value();

        // Print a messageppp
        //
        String replace_output = output_string + ",3";
        cout << "[Server post-invoke intercepted: "
             << output_string << "]" << endl;
        cout << "[Replacing with " << replace_output << "]" <<
   endl;
```

Example 26: *Server Interceptor Implementation for Reflection Example*

```
            // Replace the value before calling next interceptor.
            //
10          set_const_value(output_parts[0], replace_output);
        }

        if (m_prev_interceptor != 0)
        {
            m_prev_interceptor->intercept_post_dispatch(data);
        }
    }
```

The preceding interceptor implementation can be explained as follows:

1. The implementation of `intercept_pre_dispatch()` is designed to modify the input parameter of the `sayHi` and `greetMe` WSDL operations by appending a short string.
2. The returned part list, `input_parts`, contains all of the WSDL parts containing input parameters for the operation. A part list is essentially a vector of `(IT_Bus::QName, IT_Bus::AnyType*)` pairs. The `IT_Bus::AnyType` is the base type for all WSDL types in Artix.
3. The code in this `if`-block uses reflection to modify the first input part. This example is hard-coded to work *only* with the `sayHi` and `greetMe` operation from the `Greeter` port type. The example modifies the request message, only if it consists of a single part which is a string.
4. From the first (and only) pair in the part list, return the `IT_Bus::AnyType` value (using `get_modifiable_value()`) and convert it into a reflection object (using `get_reflection()`).
5. Assuming that the part contains a string, cast the reflection object to a string reflection.

   This step is only intended to work for the `Greeter` port type. In the general case, you would have to use the reflection interface to figure out the data type.
6. Define a modified string, `replace_input`, which adds `,2` to the original string.
7. Call `IT_Reflect::Value<String>::set_value()` to modify the input part in the request.

8. The implementation of `intercept_post_dispatch()` is designed to modify the output parameter of the `sayHi` and `greetMe` WSDL operations by appending a short string.
9. In the reply message, modify the output, only if it consists of a single part containing a string (intended for the `Greeter` port type only).
10. Call `set_const_value()` to replace the sole output part in the request. The `set_const_value()` function is a convenience template, which sets the part value to a simple type. It is defined in `it_bus/part.h` as follows:

```
// C++
namespace IT_Bus {
    template <class T>
    void set_const_value(
        Part&           part,
        T&              value
    )
    {
        part.set_const_value(
           new AnySimpleTypeT<T>(value), Part::AUTO_DELETE);
    }
}
```

The `IT_Bus::Part::set_const_value()` function takes an `IT_Bus::AnyType` as its first parameter. Because simple atomic types, such as `IT_Bus::String`, do not inherit from `AnyType`, it is necessary to wrap them in an `IT_Bus::AnySimpleTypeT<T>` instance, which does inherit from `AnyType`.

For user-defined types (and other types that inherit from `AnyType`), you can pass them directly to the `IT_Bus::Part::set_const_value()` function.

WSDL Extension Elements

If you implement your own transport or binding plug-in, you would typically configure it by defining a custom tag (or tags) in the WSDL contract. This chapter describes how to add a custom tag—that is, a WSDL extension element—to the Artix WSDL parser.

In this chapter

This chapter discusses the following topics:

WSDL Structure

Overview

This section describes some basic features of the WSDL language that are important for WSDL parsing. The following topics are discussed:

- WSDL Example.
- Standard elements.
- Extensibility/extension elements.

WSDL Example

Example 27 shows the outline of a typical WSDL file, including the important high-level elements that you would find in most WSDL files.

Example 27: *WSDL Contract with Extensibility Elements*

```
<wsdl:definitions name="nmtoken"? targetNamespace="uri"?>
    <wsdl:types> ?
        <xsd:schema .... />*
        <-- extensibility element --> *
    </wsdl:types>
    ....
    <wsdl:binding name="nmtoken" type="qname">*
        <-- extensibility element --> *
        <wsdl:operation .... />*
    </wsdl:binding>
    <wsdl:service name="nmtoken"> *
        <wsdl:port name="nmtoken" binding="qname"> *
           <-- extensibility element -->
        </wsdl:port>
        <-- extensibility element -->
    </wsdl:service>
    <-- extensibility element --> *
</wsdl:definitions>
```

Standard elements

The core of WSDL defines many standard XML elements (in Example 27 on page 74, these tags appear without any prefix before their names). For example, `portType`, `binding`, and `service`. These elements belong to the *base WSDL specification*.

Extensibility/extension elements

In addition to the standard elements, the WSDL standard allows you to extend the language by adding new WSDL elements known as *extensibility elements* or *extension elements*.

The WSDL standard does impose some restrictions, however, on where you can add these extension elements (see appendix 3 of the WSDL specification, http://www.w3.org/TR/wsdl).

WSDL Parse Tree

Overview

When an Artix application reads a WSDL file, the complete contents of the file are parsed and analyzed into a linked tree of objects, the *WSDL parse tree*. There are, in fact, two views of this tree, as follows:

- XML view—this view of the parse tree is provided by the `IT_Bus::XMLNode` base class. This view of the parse tree provides XML parsing support, but has no awareness of WSDL features.
- WSDL view—this view of the parse tree is provided by classes that inherit from `IT_WSDL::WSDLNode`. This view of the parse tree provides support for WSDL features.

This section focuses exclusively on the WSDL view of the parse tree. You should be aware, however, that you might also encounter the parse tree through the XML view. An `IT_Bus::XMLNode` object and an `IT_WSDL::WSDLNode` object can both refer to the same underlying node in the parse tree.

Parse tree classes

[Figure 7](#) shows part of the inheritance hierarchy for the classes in a WSDL parse tree. The WSDL nodes are classified into two main types:

- `IT_WSDL::WSDLExtensibleNode` nodes—base class for standard elements.
- `IT_WSDL::WSDLExtensionElement` nodes—base class for extension elements.

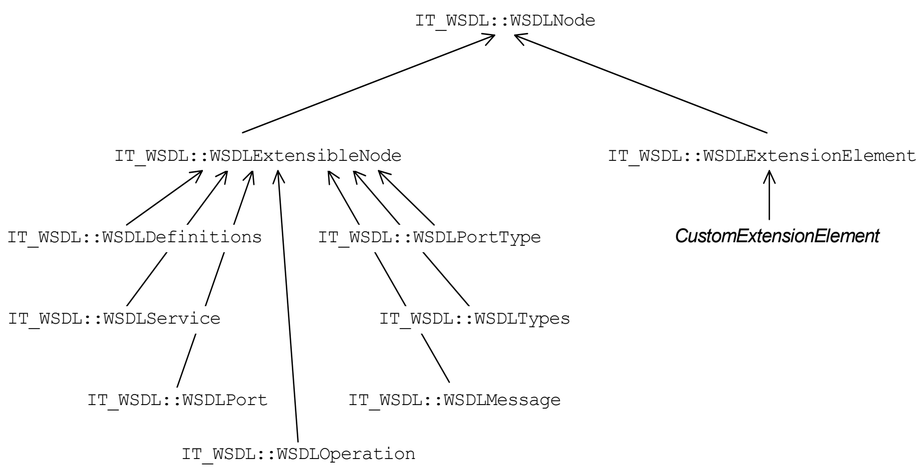

Figure 7: *WSDL Parse Tree Inheritance Hierarchy*

WSDLNode

The `IT_WSDL::WSDLNode` class is the base class for all nodes of the WSDL parse tree. It defines the following public member functions:

```
// C++
IT_WSDL::NodeType get_node_type();

// Get the QName of this element node
const IT_Bus::QName & get_element_name();

// Get the namespace URI for this element node
const IT_Bus::String & get_target_namespace();
```

WSDLExtensibleNode

The `IT_WSDL::WSDLExtensibleNode` class is used as the base class for the standard elements in WSDL. The nodes that inherit from `WSDLExtensibleNode` are extensible, in the sense that they may contain extension elements as sub-elements. In addition to the functions inherited from `IT_WSDL::WSDLNode`, the `WSDLExtensibleNode` base class defines the following public member functions:

```
// C++
IT_WSDL::WSDLExtensionElementList & get_extension_elements();

IT_WSDL::WSDLExtensionElement *
find_extension_element(
    const IT_Bus::QName &extension_element
);

IT_WSDL::WSDLExtensionElement *
create_extension_element(
    const IT_Bus::QName &extension_element
);

void
add_extension_element(
    IT_WSDL::WSDLExtensionElement *extension_element
);
```

WSDLPort

The `IT_WSDL::WSDLPort` extensible node represents the WSDL `port` element. This WSDL node type is important for Artix transports, because it encapsulates all of the information required either to open a connection (client side) or to listen for a connection (server side). The `WSDLPort` class defines the following member functions:

```
// C++
const IT_Bus::String &        get_name ()
const IT_WSDL::WSDLService &  get_service ()
const IT_WSDL::WSDLBinding *  get_binding ()
```

WSDLBinding

The `IT_WSDL::WSDLBinding` extensible node represents the WSDL `binding` element. This WSDL node type (together with a WSDL port) encapsulates the information that is needed to establish a WSDL binding. The `WSDLBinding` class defines the following member functions:

```
// C++
IT_WSDL::WSDLDefinitions &               get_definitions();
const IT_WSDL::WSDLDefinitions &         get_definitions();
```

```
const IT_WSDL::IT_Bus::QName &            get_name();
const IT_WSDL::WSDLBindingOperationMap &  get_operations();
IT_WSDL::WSDLBindingOperationMap &        get_operations();
const IT_WSDL::IT_Bus::QName &            get_port_type_name();
const IT_WSDL::WSDLPortType *             get_port_type();

const IT_WSDL::WSDLBindingOperation *
get_binding_operation (
    const IT_Bus::String &operation_name
);

const IT_Bus::String&  get_binding_namespace() const;
```

WSDLExtensionElement

The `IT_WSDL::WSDLExtensionElement` is the base class for custom extension elements. If you want to implement your own extension element class, you should make it inherit from `WSDLExtensionElement`. In your own extension element implementation, you must override the following member functions:

```
// C++
IT_WSDL::WSDLExtensionFactory &  get_extension_factory();

bool parse(
    const XMLIterator &port_type_iter,
    const IT_Bus::XMLNode &parent_node,
    IT_WSDL::WSDLErrorHandler &error_handler
);
```

How to Extend WSDL

Overview

This section provides a high-level overview of how you can extend the parsing capabilities of WSDL by adding extension elements.

Sample WSDL extensions

For example, consider the MessageQueue (MQ) plug-in for Artix, which introduces two new extension elements, `mq:client` and `mq:server`, to WSDL. These new extension elements belong to the `http://schemas.iona.com/transports/mq` namespace. Example 28 shows a WSDL extract with the MQ extension elements.

Example 28: *WSDL Extract with MQ Extension Elements*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="BaseService"
   targetNamespace="http://soapinterop.org/"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:mq="http://schemas.iona.com/transports/mq"
    ...
        >
    ...
    <service name="MQBaseService">
        <port ... >
            <mq:client ... />
            <mq:server ... />
        </port>
    </service>
</definitions>
```

Factory pattern

The scheme for extending the WSDL parser is based on a factory pattern. The programmer registers an extension factory, which is then responsible for creating instances of the extension elements on demand. Figure 8 illustrates the process of creating extension elements.

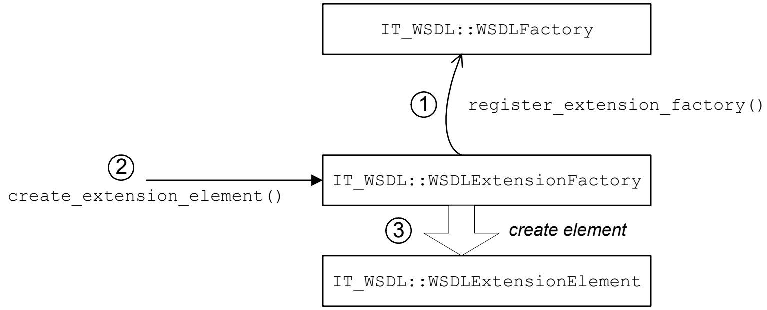

Figure 8: *Factory Pattern for WSDL Extension Elements*

Factory pattern stages

The factory pattern for creating extension elements, as shown in Figure 8 on page 81, operates as follows:

Stage	Description
1	The programmer registers a custom WSDL extension factory by calling `register_extension_factory()` on the `IT_WSDL::WSDLFactory` object. In this example, the extension factory is registered against the `http://schemas.iona.com/transports/mq` namespace URI.
2	Whenever the WSDL parser encounters an element belonging to the `http://schemas.iona.com/transports/mq` namespace, it calls `create_extension_element()` on the extension factory.
3	The extension factory figures out which type of extension element to create by examining the local part of the supplied QName and then returns a new instance of this extension element type.

Classes to implement

Figure 9 shows an outline of the inheritance hierarchy for the classes you would need to write in order to extend WSDL. There are typically three different kinds of class to implement:

- Extension factory—inherits from `IT_WSDL::WSDLExtensionFactory`.
- Extension element base class—inherits from `IT_WSDL::WSDLExtensionElement`.
- Extension elements (one or more of)—inherit from the extension element base class.

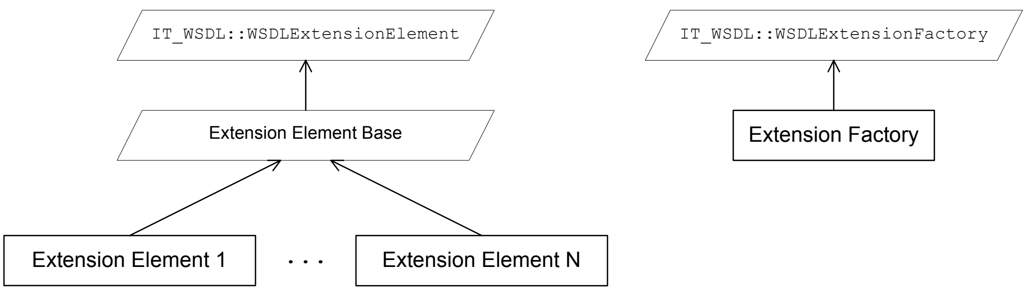

Figure 9: *Extension Element Classes*

Extension Elements for the Stub Plug-In

Overview

This section describes how to extend WSDL, by implementing an extension element class and an extension factory class for the stub plug-in. Although the particular example shown here is based on a transport plug-in, this section is relevant for binding plug-ins as well.

In this section

This section contains the following subsections:

Implementing an Extension Element Base Class

Overview

This subsection describes how to implement an extension element base class for the stub transport. Although it is not strictly necessary to define an extension element base class, if you have just one extension element, it is nevertheless good coding practice. Once you have defined a base class for your custom extension elements, it is relatively easy to add new extension elements as needed.

Extension element base header

Example 29 shows the header for the stub plug-in's extension element base class.

Example 29: *Header for the StubTransportWSDLExtensionElement Class*

```
   // C++
   #include <it_wsdl/wsdl_extension_element.h>
   #include <it_wsdl/wsdl_port.h>

   namespace IT_Transport_Stub
   {
1      class StubTransportWSDLExtensionElement :
           public IT_WSDL::WSDLExtensionElement,
           public IT_Bus::XMLNode
       {
         public:
           StubTransportWSDLExtensionElement(
               IT_WSDL::WSDLExtensibleNode* the_node
           );

           virtual const IT_Bus::QName &
2          get_element_name() const;

           virtual const IT_Bus::String &
           get_target_namespace() const;

           virtual
           IT_WSDL::WSDLExtensionFactory &
3          get_extension_factory();

           virtual ~StubTransportWSDLExtensionElement();

           virtual void
```

Example 29: *Header for the StubTransportWSDLExtensionElement Class*

```
        read(
            const IT_Bus::QName& name,
            IT_Bus::ComplexTypeReader & reader
        ) IT_THROW_DECL((IT_Bus::DeserializationException))
        {
            throw IT_Bus::IOException("Not Supported");
        }

        virtual void
        write(
            const IT_Bus::QName& element_name,
            IT_Bus::ComplexTypeWriter & writer
        ) const IT_THROW_DECL((IT_Bus::SerializationException))
        {
            // complete
        }

        virtual void
        write(
            IT_Bus::XMLOutputStream & stream
        ) const IT_THROW_DECL((IT_Bus::IOException))
        {
            // complete
        }

        virtual
        IT_Bus::AnyType&
        copy(
            const IT_Bus::AnyType & rhs
        )
        {
            return *this;
        }

      protected:
4       IT_WSDL::WSDLExtensibleNode *  m_wsdl_extensible_node;

      private:
        ...
    };
};
```

The preceding header file can be explained as follows:

1. The extension element base class must inherit from `IT_WSDL::WSDLExtensionElement` and `IT_Bus::XMLNode`.
2. The `get_element_name()` and `get_target_namespace()` functions are inherited from the `IT_WSDL::WSDLNode` base class, by way of the `IT_WSDL::WSDLExtensionElement` class.
3. The `get_extension_factory()` element is inherited from the `IT_WSDL::WSDLExtensionElement` class.
4. The `m_wsdl_extensible_node` is used to store a pointer to the parent node (that is, a pointer to the `WSDLExtensibleNode` instance that contains this node).

Extension element base implementation

Example 30 shows the implementation of the stub plug-in's extension element base class.

Example 30: *Implementation of StubTransportWSDLExtensionElement*

```
// C++
#include "stub_transport_wsdl_extension_element.h"
#include "stub_transport_wsdl_extension_factory.h"

using namespace IT_Bus;
using namespace IT_WSDL;
using namespace IT_Transport_Stub;

1  StubTransportWSDLExtensionElement::StubTransportWSDLExtensionEle
    ment(
     IT_WSDL::WSDLExtensibleNode* the_node
) : m_wsdl_extensible_node(the_node)
{
    // complete
}

StubTransportWSDLExtensionElement::~StubTransportWSDLExtensionEl
    ement()
{
    // complete
}

WSDLExtensionFactory &
2  StubTransportWSDLExtensionElement::get_extension_factory()
{
```

Example 30: *Implementation of StubTransportWSDLExtensionElement*

```
    return StubTransportWSDLExtensionFactory::get_instance();
}

const IT_Bus::QName &
3 StubTransportWSDLExtensionElement::get_element_name() const
{
    return get_tag_name();
}

const IT_Bus::String &
4 StubTransportWSDLExtensionElement::get_target_namespace() const
{
    return XMLNode::get_target_namespace();
}
```

The preceding implementation class can be described as follows:

1. The sole constructor argument, `the_node`, is a pointer to the parent extensible element node (an extensible element node is a node that can contain other element nodes).
2. The `get_extension_factory()` function returns a reference to the extension factory that is responsible for creating all of the WSDL extension elements that inherit from this extension element base class.
3. The implementation of `get_tag_name()` is inherited from the `IT_Bus::XMLNode` base class. It returns the `QName` of the current element.
4. The implementation of `get_target_namespace()` simply calls the implementation from the `IT_Bus::XMLNode` base class.

Implementing the Extension Element Classes

Overview

This subsection describes how to implement the stub extension element class (there is only one extension element in the stub transport plug-in). This class must be capable of parsing the stub extension element.

Stub extension element

The stub plug-in adds a single extension element to WSDL, as shown in Example 31. The stub extension element name is `NamespacePrefix:address`, with a single attribute, `location`. In Example 31, the `NamespacePrefix` is defined as `stub`.

Example 31: *Sample WSDL with Stub Extension Element*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions ...
    targetNamespace = ...
    xmlns       = "http://schemas.xmlsoap.org/wsdl/"
    xmlns:stub= "http://schemas.iona.com/transports/stub"
    ...
    >
    ...
    <service ... >
        <port ... >
            <stub:address
                location="local_0001"
            />
        </port>
    </service>
</definitions>
```

Extension element header

Example 32 shows the header file for the stub extension element class.

Example 32: *Header for the StubTransportWSDLAddress Class*

```
// C++
#include "stub_transport_wsdl_extension_element.h"

namespace IT_Transport_Stub
{
1       class StubTransportWSDLAddress :
            public StubTransportWSDLExtensionElement
```

Example 32: *Header for the StubTransportWSDLAddress Class*

```
{
  public:

    StubTransportWSDLAddress(
        IT_WSDL::WSDLExtensibleNode* the_node
    );
    StubTransportWSDLAddress();
    virtual ~StubTransportWSDLAddress();

    IT_WSDL::WSDLExtensionElement*
    clone() const;

    virtual bool
    parse(
        const IT_Bus::XMLIterator & element_iterator,
        const IT_Bus::XMLNode & element,
        IT_WSDL::WSDLErrorHandler & error_handler
    );

    const IT_Bus::String&
2   get_location() const;

    virtual void
    set_location(
        const IT_Bus::String & location
    );

    virtual
    IT_Bus::AnyType&
    operator=(
        const IT_Bus::AnyType & rhs
    )
    {
        return *this;
    }

3   static const IT_Bus::String ELEMENT_NAME;
    static const IT_Bus::String TYPE_ATTRIBUTE_NAME;

  private:

4   IT_Bus::String m_location;
    IT_Bus::String m_target_namespace;
    ...
};
```

Example 32: *Header for the StubTransportWSDLAddress Class*

```
};
```

The preceding header file can be described as follows:

1. The stub extension element inherits from the stub extension element base class, `StubTransportWSDLExtensionElement`.
2. The `get_location()` and `set_location()` functions are not inherited. They are specific to the `StubTransportWSDLAddress` class.
3. Two convenient constants are declared here: `ELEMENT_NAME` is the local part of the extension element QName, which is `address`; `TYPE_ATTRIBUTE_NAME` is the name of the attribute, `location`.
4. The `m_location` variable stores the value of the `location` attribute, (which is, essentially, all of the useful information that is contained in the `address` element).

Extension element implementation

Example 33 shows the implementation of the stub extension element class.

Example 33: *Implementation of the StubTransportWSDLAddress Class*

```
// C++
#include "stub_transport_wsdl_address.h"

#include "stub_transport_wsdl_extension_factory.h"

using namespace IT_Bus;
using namespace IT_WSDL;
using namespace IT_Transport_Stub;


1 const String StubTransportWSDLAddress::ELEMENT_NAME = "address";
const String StubTransportWSDLAddress::TYPE_ATTRIBUTE_NAME =
    "location";


2 StubTransportWSDLAddress::StubTransportWSDLAddress(
    IT_WSDL::WSDLExtensibleNode* the_node
)
 : StubTransportWSDLExtensionElement(the_node)
{
    // complete
```

Example 33: *Implementation of the StubTransportWSDLAddress Class*

```
}

3 StubTransportWSDLAddress::StubTransportWSDLAddress()
 : StubTransportWSDLExtensionElement(0)
{
    set_tag_name(
        StubTransportWSDLAddress::ELEMENT_NAME.c_str(),
        StubTransportWSDLExtensionFactory::SCHEMA_URL.c_str(),
        0
    );
}


StubTransportWSDLAddress::~StubTransportWSDLAddress()
{
    // complete
}

IT_WSDL::WSDLExtensionElement*
4 StubTransportWSDLAddress::clone() const
{
    StubTransportWSDLAddress* clone =
                           new StubTransportWSDLAddress();
    clone->set_location(this->get_location());
    return clone;
}

bool
5 StubTransportWSDLAddress::parse(
    const XMLIterator & element_iterator,
    const IT_Bus::XMLNode & element,
    IT_WSDL::WSDLErrorHandler & error_handler
)
{
6     XMLNode::operator =(element);
7     m_location = element_iterator.get_field_as_string(
                     TYPE_ATTRIBUTE_NAME
                 );
    return true;
}

const String&
8 StubTransportWSDLAddress::get_location() const
{
    return m_location;
```

Example 33: *Implementation of the StubTransportWSDLAddress Class*

```
}

void
StubTransportWSDLAddress::set_location(
    const String & location
)
{
    m_location = location;
}
```

The preceding class implementation can be explained as follows:

1. The `ELEMENT_NAME` and `TYPE_ATTRIBUTE_NAME` constants are defined here.
2. This form of the constructor takes a pointer to the parent extensible element. This is the form of constructor called by the stub plug-in's WSDL extension factory.
3. The default constructor sets the QName of this element by calling the `set_tag_name()` function, which is inherited from the `IT_Bus::XMLNode` class.
4. The `clone()` method makes a copy of the WSDL extension element.
5. The `parse()` function is automatically called by the Artix core as it constructs the in-memory WSDL model of the application's WSDL contract.
6. This call to `XMLNode::operator=()` copies the contents of the `element` parameter into the current element. The unusual syntax ensures that only the `XMLNode` version of the assignment operator is used (as opposed to an assignment operator defined lower down the inheritance hierarchy).
7. The call to `XMLIterator::get_field_as_string()` searches the node for the value of the `location` attribute (in this context, *field* means an attribute value).
8. The `get_location()` function can be called by other components of the stub plug-in to access the value of the `location` attribute from the `address` element.

Implementing the Extension Factory

Overview

This subsection describes how to write the stub extension factory class. An extension factory must be capable of creating *all* types of extension element that belong to a specific namespace (identified by a namespace URI).

In particular, the stub extension factory must be capable of creating all WSDL extension elements belonging to the `http://schemas.iona.com/transports/iiop_stub` namespace. There is, in fact, only one such extension element: *`stubPrefix`*`:address`.

Stub extension factory header

Example 34 shows the header file for the stub extension factory class.

Example 34: *Header for the StubTransportWSDLExtensionFactory Class*

```
// C++
#include <it_wsdl/wsdl_extension_factory.h>
#include <it_bus/bus.h>
#include "stub_transport_wsdl_extension_element.h"

namespace IT_Transport_Stub
{
1    class StubTransportWSDLExtensionFactory
      : public IT_WSDL::WSDLExtensionFactory
    {
      public:
        virtual
        IT_WSDL::WSDLExtensionElement *
        create_extension_element(
            IT_WSDL::WSDLExtensibleNode& parent,
            const IT_Bus::QName& extension_element
        ) const;

        virtual IT_Bus::AnyType *
        create_type(
            const IT_Bus::QName& extension_element
        ) const;

        virtual void
        destroy_type(
            IT_Bus::AnyType * element
        ) const;
```

Example 34: *Header for the StubTransportWSDLExtensionFactory Class*

```
        static StubTransportWSDLExtensionFactory &
        get_instance();

        static StubTransportWSDLExtensionElement*
2       get_extension_element(
            const IT_WSDL::WSDLPort& wsdl_port,
            const IT_Bus::String& element_name
        );

        StubTransportWSDLExtensionFactory();
        virtual ~StubTransportWSDLExtensionFactory();

3       static const IT_Bus::String SCHEMA_URL;

      private:
        ...
    };
};
```

The preceding header file can be explained as follows:

1. The extension factory must inherit from the `IT_WSDL::WSDLExtensionFactory` base class.
2. The `get_extension_element()` function is not inherited. It is specific to the stub WSDL extension factory.
3. The `SCHEMA_URL` is a convenient string constant that stores the namespace URI for this extension factory. It is initialized to be `http://schemas.iona.com/transports/stub`.

Stub extension factory implementation

Example 35 shows the implementation of the stub extension factory class.

Example 35: *Implementation of the StubTransportWSDLExtensionFactory*

```
// C++
#include "stub_transport_wsdl_address.h"
#include "stub_transport_wsdl_extension_factory.h"

using namespace IT_WSDL;
using namespace IT_Bus;
using namespace IT_Transport_Stub;
```

Example 35: *Implementation of the StubTransportWSDLExtensionFactory*

```
1 const String StubTransportWSDLExtensionFactory::SCHEMA_URL =
     "http://schemas.iona.com/transports/stub";

  StubTransportWSDLExtensionFactory::StubTransportWSDLExtensionFac
     tory()
  {
      // complete
  }

  StubTransportWSDLExtensionFactory::~StubTransportWSDLExtensionFa
     ctory()
  {
      // complete
  }

  IT_Bus::AnyType *
2 StubTransportWSDLExtensionFactory::create_type(
      const QName& extension_element
  ) const
  {
      return 0;
  }

  WSDLExtensionElement *
3 StubTransportWSDLExtensionFactory::create_extension_element(
      WSDLExtensibleNode& parent,
      const QName& extension_element
  ) const
  {
      String local_part = extension_element.get_local_part();

4     if (local_part == StubTransportWSDLAddress::ELEMENT_NAME)
      {
          return new StubTransportWSDLAddress(&parent);
      }

5     return 0;
  }

  void
  StubTransportWSDLExtensionFactory::destroy_type(
      IT_Bus::AnyType * element
  ) const
  {
      delete IT_DYNAMIC_CAST(
```

Example 35: *Implementation of the StubTransportWSDLExtensionFactory*

```
                StubTransportWSDLExtensionElement *,
                element
            );
}

6 StubTransportWSDLExtensionFactory
    it_glob_stub_transport_wsdl_extension_factory_instance;

StubTransportWSDLExtensionFactory &
StubTransportWSDLExtensionFactory::get_instance()
{
    return
    it_glob_stub_transport_wsdl_extension_factory_instance;
}

StubTransportWSDLExtensionElement*
7 StubTransportWSDLExtensionFactory::get_extension_element(
    const WSDLPort& wsdl_port,
    const String& element_name
)
{
    StubTransportWSDLExtensionElement* extension_element = 0;

8   const WSDLExtensionElementList & port_children_nodes =
        wsdl_port.get_extension_elements();

9   WSDLExtensionElementList::const_iterator node_iter =
        port_children_nodes.begin();

    QName element_qname("", element_name, SCHEMA_URL);

    while (node_iter != port_children_nodes.end())
    {
        const QName & curr_qname =
                         (*node_iter)->get_element_name();

        if (element_qname == curr_qname)
        {
            extension_element = IT_DYNAMIC_CAST(
                StubTransportWSDLExtensionElement *,
                (*node_iter)
            );
        }
        node_iter++;
    }
```

Example 35: *Implementation of the StubTransportWSDLExtensionFactory*

```
    return extension_element;
}
```

The preceding implementation class can be explained as follows:

1. This line sets the `SCHEMA_URL` to `http://schemas.iona.com/transports/stub`, which is the namespace URI that identifies this WSDL extension factory.
2. A WSDL extension factory can also be used to define new XML schema types, which can be instantiated using the `create_type()` function. Because the stub plug-in's schema does not define any new types, this function has a dummy implementation.
3. The `create_extension_element()` function is called by the Artix core while it is creating the in-memory WSDL parse tree. When the WSDL parser encounters an element that belongs to the stub plug-in's namespace URI, it delegates creation of the element to this extension factory. The `create_extension_element()` function is responsible for creating *all* of the different kinds of elements that belong to the `http://schemas.iona.com/transports/stub` namespace URI.
4. Because there is only one extension element defined by the stub plug-in (that is, `address`), it is only necessary to check if the local part of the QName equals `address` before creating a `StubTransportWSDLAddress` instance.

   In general, however, an implementation of `create_extension_element()` would typically have to compare the value of `local_part` with several different extension element names to select the right type of element.
5. A return value of `0` indicates that `create_extension_element()` could not create the requested element type.
6. This line creates a single global instance of the stub plug-in's WSDL extension factory.

   > **Note:** You do not necessarily have to create this factory as a global static object. Any variation of a singleton implementation pattern would do here.

7. The `get_extension_element()` function is specific to this extension factory implementation. It searches a WSDL port element, `wsdl_port`, for a sub-element with the given name, `element_name`. The transport code uses this function to extract configuration details from the WSDL port.
8. The `get_extension_elements()` function returns a list of all the sub-elements contained in the WSDL port.
9. The extension element list is modelled on the C++ Standard Template Library list type, `std::list`. Hence, you can use an iterator to search through the WSDL port's sub-elements.

Registering the Extension Factory

Overview

The final step is to register the stub extension factory, so that the extensions become available to the overall WSDL parse tree. Registration is performed by calling the `register_extension_factory()` function on the WSDL factory object.

WSDL factory

The *WSDL factory* is an object of `IT_WSDL::WSDLFactory` type that maintains a registry of all WSDL extension factory classes. The following `IT_WSDL::WSDLFactory` member functions manage the extension factory registry:

```
// C++
void register_extension_factory(
    const IT_Bus::String &extension_namespace,
    const WSDLExtensionFactory &factory
);

void deregister_extension_factory(
    const IT_Bus::String &extension_namespace
);
```

Namespace URI

Registration associates a specific namespace URI with an extension factory. While parsing a WSDL file, the WSDL factory will call on the extension factory whenever it encounters elements from this namespace.

In the case of the stub extension factory, the namespace URI is:

```
http://schemas.iona.com/transports/stub
```

Example

Example 36 shows how to register a stub extension factory with the `IT_WSDL::WSDLFactory` object. For the stub plug-in, registration is performed by the `TransportFactory` object—see "Implementing the Transport Factory" on page 152.

Example 36: *Registering a WSDL Extension Factory Instance*

```
// C++
...
using namespace IT_Bus;
using namespace IT_WSDL;
...
void
IT_Transport_Stub::StubTransportFactory::register_wsdl_extension
   _factories(
    IT_WSDL::WSDLFactory & factory
) const
{
    factory.register_extension_factory(
        "http://schemas.iona.com/transports/stub",
        it_glob_stub_transport_wsdl_extension_factory_instance
    );
}

void
IT_Transport_Stub::StubTransportFactory::deregister_wsdl_extensi
   on_factories(
    IT_WSDL::WSDLFactory & factory
) const
{
    factory.register_extension_factory(
        "http://schemas.iona.com/transports/stub",
        it_glob_stub_transport_wsdl_extension_factory_instance
    );
}
```

Artix Transport Plug-Ins

This chapter describes how to implement an Artix transport plug-in, which enables you to integrate Artix with any transport protocol.

In this chapter

This chapter discusses the following topics:

The Artix Transport Layer

Overview

This section provides an overview of the architecture and API for the Artix transport layer.

In this section

This section contains the following subsections:

Architecture Overview

Transport architecture

Figure 10 gives a high-level overview of the Artix transport architecture.

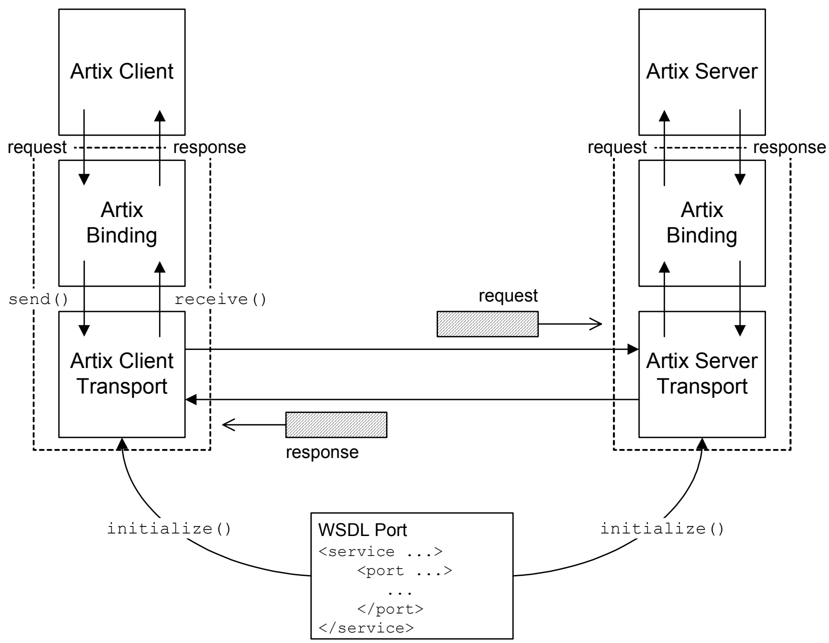

Figure 10: *Artix Transport Architecture*

WSDL port

The WSDL port, as shown in Figure 10, refers to the WSDL `port` element that specifies the connection parameters for this transport instance. For example, the WSDL port for a TCP/IP-based transport would specify values for the server's host and IP port.

In the general case, a WSDL port can specify connection parameters for both client and server.

Client transport

A client transport is an object of `IT_Bus::ClientTransport` type, which can be implemented by an Artix plug-in developer. The main functions supported by the client transport class are, as follows:

- `initialize()`—configure the client connection (usually based on the parameters read from the WSDL port).
- `connect()`/`disconnect()`—open/close a connection to the remote host.
- `invoke()`/`invoke_oneway()`—send and receive messages in raw binary format.

Server transport

A server transport is an object of `IT_Bus::ServerTransport` type, which can be implemented by an Artix plug-in developer. The main functions supported by the server transport class are, as follows:

- `activate()`—begin listening for client connection attempts and incoming request messages. Typically, the implementation of this function spawns a new thread to listen for incoming messages.
- `deactivate()`—stop listening for client connection attempts and incoming request messages.
- `get_configuration()`—return a reference to the WSDL extension element that configures this transport.
- `shutdown()`—notifies the server transport that the Bus is shutting down.
- `send()`—a callback to send reply messages back to the client. This function is called, only if you select an asynchronous style of message dispatch (which is indicated by enabling the requires stack unwind policy).
- `run()`—for a certain combination of policies, this function contains the code that listens for incoming requests. If you select the `MESSAGING_PORT_DRIVEN` threading resources policy in combination with the `MULTI_THREADED` messaging port threading policy, the `run()` function is called concurrently by multiple messaging port threads.

Artix Transport Classes

Overview

Figure 11 shows an overview of the main classes that are relevant to the implementation of an Artix transport. A brief description of each of these classes is provided in this subsection.

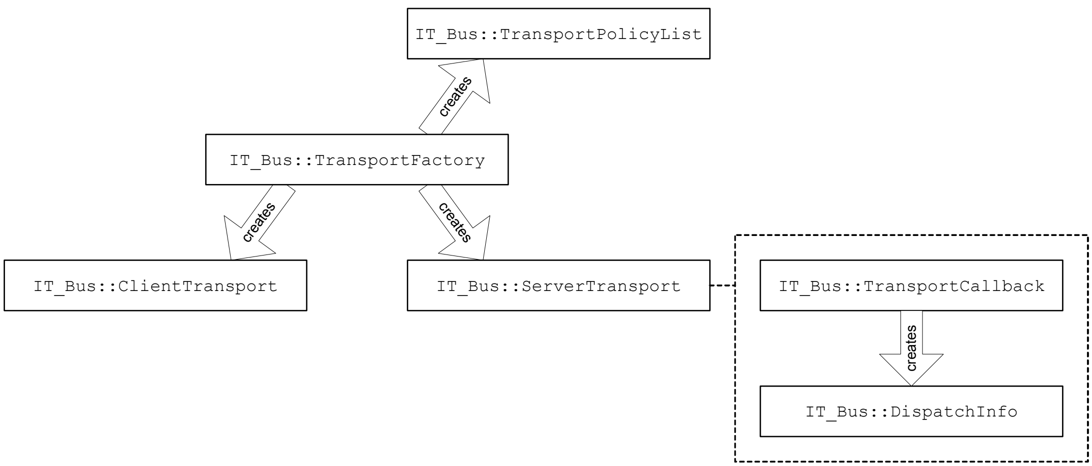

Figure 11: *Overview of the Artix Transport Classes*

TransportFactory Class

The `IT_Bus::TransportFactory` is responsible for creating the basic objects in a transport implementation.When implementing a transport, you must implement a class that derives from `TransportFactory` and then register an instance of the transport factory implementation with the Artix Bus.

ClientTransport Class

For the client side of a transport, you must define and implement a class that derives from the `IT_Bus::ClientTransport` class. The client transport must be capable of opening a connection to a remote service, as well as sending and receiving binary buffers through the transport.

ServerTransport Class

For the server side of a transport, you must define and implement a class that derives from the `IT_Bus::ServerTransport` class. The server transport implementation should be capable of listening for incoming request messages (in binary format) from the transport layer and dispatching these messages up the call stack.

Requests are dispatched by calling the `IT_Bus::TransportCallback::dispatch()` function.

TransportCallback Class

The `IT_Bus::TransportCallback` class is provided by the Artix runtime; you do *not* need to implement this class. The most important member of `TransportCallback` is the `dispatch()` function, which the server code uses to dispatch a request message up the call stack.

The `TransportCallback` class acts as an *observer* for the `ServerTransport` class. The `TransportCallback` functions must be called from within a `ServerTransport` object as follows:

- `TransportCallback::transport_activated()`—called from within `ServerTransport::activate()`, after the transport is activated.
- `TransportCallback::transport_deactivated()`—called from within `ServerTransport::deactivate()`, after the transport is deactivated.
- `TransportCallback::transport_shutdown()`—called from within `ServerTransport::shutdown()`, after the transport has been shut down.

DispatchInfo Class

The `IT_Bus::DispatchInfo` class is provided by the Artix runtime. You can obtain a `DispatchInfo` object by calling the `TransportCallback::get_dispatch_context()` function. On the server side, a `DispatchInfo` object is used to encapsulate additional information about the current message.

For example, the `DispatchInfo` object is used to hold incoming and outgoing context data. You can also use the `DispatchInfo::get_correlation_id()` function to obtain an ID that lets you match incoming requests to outgoing replies.

TransportPolicyList Class

The `IT_Bus::TransportPolicyList` holds a collection of policy options that affect the semantics of the server side of the transport. You can customize the interaction between the Artix runtime and the server transport by setting the appropriate policies on a `TransportPolicyList` instance and returning this instance from the `TransportFactory::get_policies()` function.

Transport Threading Models

Overview

Artix provides a variety of threading models for server transports. For a relatively simple server transport implementation, you can take advantage of the messaging port thread pool, which makes it unnecessary to write the threading code yourself. Alternatively, if you need more flexibility, you can use the externally driven threading model, which allows you to implement a custom threading model.

In this section

This section contains the following subsections:

Threading Introduction

Overview

The server transport threading model is selected by setting threading policies on an `IT_Bus::TransportPolicyList` object. This section provides a brief overview of the various threading policy combinations. The chosen threading policy combination affects the transport in two ways:

- It dictates a particular programming model for the server transport and
- It regulates the interaction between the Artix runtime and the server transport.

Threading resources policy

The threading resources policy is used to tell the Artix runtime where the server transport's threading resources must come from:

- `MESSAGING_PORT_DRIVEN` policy value—the threads used to read incoming request messages are supplied from the messaging port thread pool. This policy setting can be combined with one of the following messaging port threading policies:
  - `MULTI_INSTANCE`,
  - `MULTI_THREADED`,
  - `SINGLE_THREADED`.
- `EXTERNALLY_DRIVEN` policy value—the reader threads are either created by the server transport itself or provided from some other external source.

Messaging port threading model policy

If you have selected the `MESSAGING_PORT_DRIVEN` threading resources policy, you can combine it with a messaging port threading model policy. The following policy values are supported:

- `MULTI_INSTANCE` policy value—the Artix runtime creates multiple instances of the `ServerTransport` class and each instance consumes a single thread from the messaging port thread pool.
- `MULTI_THREADED` policy value—the Artix runtime creates a single instance of the `ServerTransport` class and this single instance consumes multiple threads from the messaging port thread pool.

- `SINGLE_THREADED` policy value—the Artix runtime creates a single instance of the `ServerTransport` class and this instance consumes a single thread from the messaging port thread pool.

Setting the server transport threading policies

To set the server threading policies, create an `IT_Bus::TransportPolicyList` instance, initialize it with the relevant policy values, and return the policy list from the `TransportFactory::get_policies()` function.

When the Artix runtime is about to activate a service, it calls the `get_policies()` function to discover what kind of policies should govern the server transport. This includes the settings for the threading model.

MESSAGING_PORT_DRIVEN and MULTI_INSTANCE

Overview

By combining the `MESSAGING_PORT_DRIVEN` and `MULTI_INSTANCE` policy values, you obtain the threading model shown in Figure 12. When the service is activated, Artix creates multiple `ServerTransport` instances to service the incoming requests. Each of the `ServerTransport` instances consumes a thread from the messaging port thread pool.

The implementation of the `activate()` function incorporates a while loop which continuously reads request messages from the transport layer and dispatches these requests to a `TransportCallback` object. It is this blocked `activate()` function which consumes a messaging port thread.

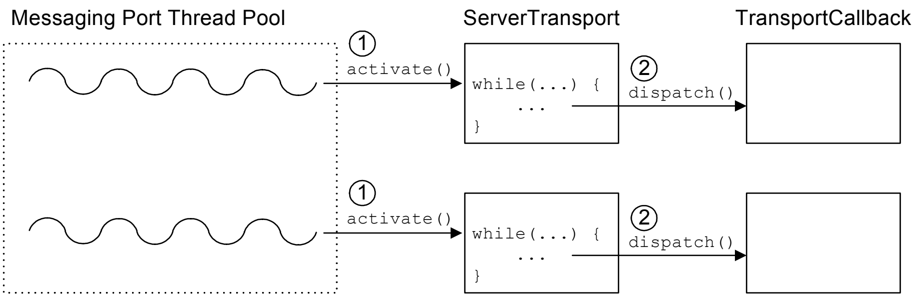

Figure 12: *MESSAGING_PORT_DRIVEN and MULTI_INSTANCE Threading Model*

How it works

The `MESSAGING_PORT_DRIVEN` and `MULTI_INSTANCE` threading model shown in Figure 12 works as follows

Stage	Description
1	Each of the threads in the messaging port thread pool calls `activate()` on a separate `IT_Bus::ServerTransport` instance. The `activate()` function remains blocked for as long as the service is active (the `activate()` implementation typically contains a while loop).

Stage	Description
2	Each of the `ServerTransport` objects calls `dispatch()` on a separate `IT_Bus::TransportCallback` instance.

Setting the policies

To set the server threading policies, create an `IT_Bus::TransportPolicyList` instance, initialize it with the relevant policy values, and return the policy list from the `TransportFactory::get_policies()` function.

Example 37 shows how to set the `MESSAGING_PORT_DRIVEN` and `MULTI_INSTANCE` policy values.

Example 37: *Setting Policies for MESSAGING_PORT_DRIVEN and MULTI_INSTANCE Threading Model*

```
// C++
void
TransportFactoryImpl::initialize(Bus_ptr bus)
{
   m_transport_policylist =
      bus->get_pdk_bus()->create_transport_policy_list();
   m_transport_policylist->set_policy_threading_resources(
      IT_Bus::MESSAGING_PORT_DRIVEN
   );
   m_transport_policylist->set_policy_messaging_port_threading(
      IT_Bus::MULTI_INSTANCE
   );
}

const TransportPolicyList*
TransportFactoryImpl::get_policies()
{
    return m_transport_policylist;
}
```

Configuring the thread pool

To configure the thread pool for a transport that uses a combination of the `MESSAGING_PORT_DRIVEN` and `MULTI_INSTANCE` policies, set the following variable in the Artix configuration file:

`policy:messaging_transport:min_threads`

This variable specifies the number of threads in the messaging port's thread pool, when the multi-instance policy is in effect. The default is `1`.

MESSAGING_PORT_DRIVEN and MULTI_THREADED

Overview

By combining the `MESSAGING_PORT_DRIVEN` and `MULTI_THREADED` policy values, you obtain the threading model shown in Figure 13. When the service is activated, Artix creates a *single* `ServerTransport` instance to service the incoming requests. The `activate()` function is responsible for initializing the transport and the `run()` function, which is called concurrently by multiple threads, is responsible for processing incoming requests.

The implementation of the `run()` function incorporates a while loop which continuously reads request messages from the transport layer and dispatches these requests to the `TransportCallback` object.

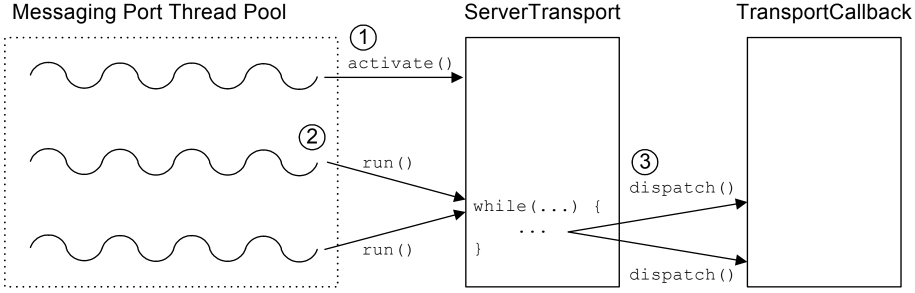

Figure 13: *MESSAGING_PORT_DRIVEN and MULTI_THREADED Threading Model*

How it works

The `MESSAGING_PORT_DRIVEN` and `MULTI_THREADED` threading model shown in Figure 13 works as follows

Stage	Description
1	A thread from the messaging port thread pool calls `activate()` on the sole `IT_Bus::ServerTransport` instance. The `activate()` function puts the transport layer into a state where it is ready to receive request messages, but the function does not process any messages and returns immediately.

Stage	Description
2	A number of threads from the thread pool call `run()` on the sole `IT_Bus::ServerTransport` instance. The `run()` function is responsible for reading request messages from the transport and dispatching them to the `TransportCallback` object. Hence, the calls to `run()` remain blocked for as long as the service is active.
3	Within each of the concurrent `run()` calls, the implementation code calls `dispatch()` on the `IT_Bus::TransportCallback` instance whenever a request message is received on the transport.

Setting the policies

To set the server threading policies, create an `IT_Bus::TransportPolicyList` instance, initialize it with the relevant policy values, and return the policy list from the `TransportFactory::get_policies()` function.

Example 38 shows how to set the `MESSAGING_PORT_DRIVEN` and `MULTI_THREEADED` policy values.

Example 38: *Setting Policies for MESSAGING_PORT_DRIVEN and MULTI_THREADED Threading Model*

```
// C++
void
TransportFactoryImpl::initialize(Bus_ptr bus)
{
   m_transport_policylist =
      bus->get_pdk_bus()->create_transport_policy_list();
   m_transport_policylist->set_policy_threading_resources(
      IT_Bus::MESSAGING_PORT_DRIVEN
   );
   m_transport_policylist->set_policy_messaging_port_threading(
      IT_Bus::MULTI_THREADED
   );
}

const TransportPolicyList*
TransportFactoryImpl::get_policies()
{
    return m_transport_policylist;
}
```

Thread safety

When you use the `MULTI_THREADED` policy value, there is only a single instance of the `ServerTransport`, but the instance's `run()` function is called concurrently from multiple threads. *It follows that you must take care to make the implementation of* `run()` *completely thread-safe.*

For example, member variables of the `ServerTransport` class must be protected by a mutex lock whenever they are accessed from within the `run()` function.

Configuring the thread pool

To configure the thread pool for a transport that uses a combination of the `MESSAGING_PORT_DRIVEN` and `MULTI_THREADED` policies, set the following variable in the Artix configuration file:

```
policy:messaging_transport:concurrency
```

This variable specifies the number of threads in the messaging port's thread pool, when the multi-threaded policy is in effect. The default is `1`.

MESSAGING_PORT_DRIVEN and SINGLE_THREADED

Overview

By combining the `MESSAGING_PORT_DRIVEN` and `SINGLE_THREADED` policy values, you obtain the threading model shown in Figure 14. When the service is activated, Artix creates a single `ServerTransport` instance to service the incoming requests. The `ServerTransport` instance consumes a single thread from the messaging port thread pool.

The implementation of the `activate()` function incorporates a while loop which continuously reads request messages from the transport layer and dispatches these requests to the `TransportCallback` object.

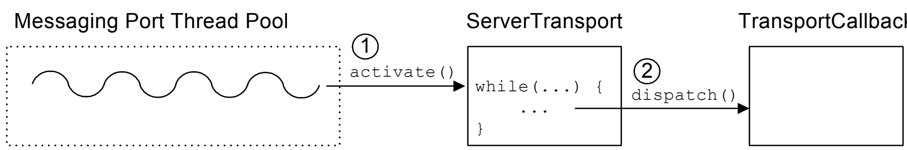

Figure 14: *MESSAGING_PORT_DRIVEN and SINGLE_THREADED Threading Model*

How it works

The `MESSAGING_PORT_DRIVEN` and `SINGLE_THREADED` threading model shown in Figure 14 works as follows

Stage	Description
1	A single thread in the messaging port thread pool calls `activate()` on a single `IT_Bus::ServerTransport` instance. The `activate()` function remains blocked for as long as the service is active (the `activate()` implementation typically contains a while loop).
2	The `ServerTransport` object calls `dispatch()` on the `IT_Bus::TransportCallback` instance.

Setting the policies

To set the server threading policies, create an `IT_Bus::TransportPolicyList` instance, initialize it with the relevant policy values, and return the policy list from the `TransportFactory::get_policies()` function.

Example 39 shows how to set the `MESSAGING_PORT_DRIVEN` and `SINGLE_THREADED` policy values.

Example 39: *Setting Policies for MESSAGING_PORT_DRIVEN and SINGLE_THREADED Threading Model*

```
// C++
void
TransportFactoryImpl::initialize(Bus_ptr bus)
{
   m_transport_policylist =
      bus->get_pdk_bus()->create_transport_policy_list();
   m_transport_policylist->set_policy_threading_resources(
      IT_Bus::MESSAGING_PORT_DRIVEN
   );
   m_transport_policylist->set_policy_messaging_port_threading(
      IT_Bus::SINGLE_THREADED
   );
}

const TransportPolicyList*
TransportFactoryImpl::get_policies()
{
    return m_transport_policylist;
}
```

EXTERNALLY_DRIVEN

Overview

By selecting the `EXTERNALLY_DRIVEN` policy value, you obtain the threading model shown in Figure 15. When the service is activated, Artix creates a single `ServerTransport` instance to service the incoming requests. The `ServerTransport` instance does *not* consume any threads from the messaging port thread pool. That is, the call to `activate()` must be non-blocking.

The essence of the `EXTERNALLY_DRIVEN` thread model is that it does not consume any messaging port threads. This model is useful if you use a transport library that has its own threading capabilities.

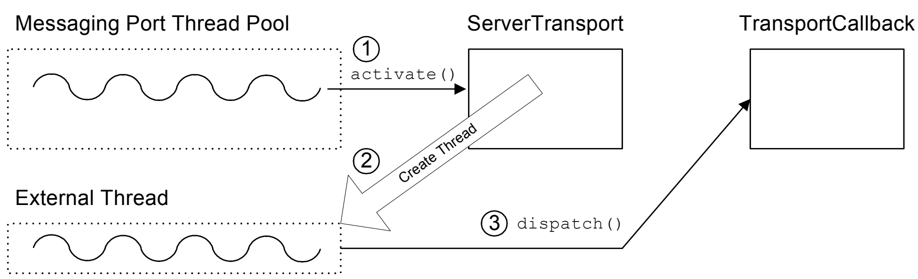

Figure 15: *EXTERNALLY_DRIVEN Threading Model*

How it works

The `EXTERNALLY_DRIVEN` threading model shown in Figure 15 works as follows

Stage	Description
1	A single thread in the messaging port thread pool calls `activate()` on an `IT_Bus::ServerTransport` instance. The `activate()` function puts the transport layer into a state where it is ready to receive request messages, but it does not process any messages.

Stage	Description
2	Before returning, the `activate()` function either obtains a thread from an external source or creates a new thread to process the incoming request messages. The request processing code could be put into a private member function of `ServerTransport` or it could belong to a different object altogether.
3	The request processing code, which is running in the external thread, calls `dispatch()` on the `IT_Bus::TransportCallback` instance.

Setting the policies

To set the server threading policies, create an `IT_Bus::TransportPolicyList` instance, initialize it with the relevant policy values, and return the policy list from the `TransportFactory::get_policies()` function.

Example 40 shows how to set the `EXTERNALLY_DRIVEN` policy value.

Example 40: *Setting Policies for EXTERNALLY_DRIVEN Threading Model*

```
// C++
void
TransportFactoryImpl::initialize(Bus_ptr bus)
{
   m_transport_policylist =
      bus->get_pdk_bus()->create_transport_policy_list();
   m_transport_policylist->set_policy_threading_resources(
      IT_Bus::EXTERNALLY_DRIVEN
   );
}

const TransportPolicyList*
TransportFactoryImpl::get_policies()
{
    return m_transport_policylist;
}
```

Dispatch Policies

Overview

Dispatching refers to the stage just after the server transport obtains the request message in the form of a raw buffer. The server transport calls the `dispatch()` function to pass the request message up to the next layer in the stack, where it is processed and ultimately routed to the appropriate servant object.

The dispatch policies enable you to control the degree to which dispatching is synchronized with the transport layer. Broadly speaking, the two main options are synchronous call semantics (RPC-style dispatch) or asynchronous call semantics (messaging-style dispatch).

In this section

This section contains the following subsections:

Dispatch Policy Overview

Overview

On the server side, the manner in which a request message is dispatched to the upper layers of an application can be influenced by a number of policies, as follows:

- Stack unwind policy.
- Asynchronous dispatch policy.

Stack unwind policy

The stack unwind policy can be set or read from a `TransportPolicyList` object using the following API functions:

```
// C++
namespace IT_Bus {
    class IT_BUS_API TransportPolicyList
    {
      public:
        ...
        virtual void
        set_policy_requires_stack_unwind(const bool policy) = 0;

        virtual const bool
        get_policy_requires_stack_unwind() const = 0;
};
```

The stack unwind policy selects between an RPC-style dispatch and a messaging-style dispatch.

If the stack unwind policy is `true`, you must call the `DispatchInfo::provide_response_buffer()` function to provide a reply buffer reference and the `TransportCallback::dispatch()` function blocks until the reply buffer is written.

If the stack unwind policy is `false`, you must call the `TransportCallback::dispatch()` function to dispatch a request buffer. The reply buffer is passed back to the `ServerTransport` through a callback on the `ServerTransport::send()` function. In this case also, the `dispatch()` function blocks until the reply buffer is written.

The default is `false`.

Asynchronous dispatch policy

The asynchronous dispatch policy can be set on a per-request basis and is set by passing a boolean value into the optional parameter of the `TransportCallback::dispatch()` function, which has the following signature:

```
// C++
namespace IT_Bus {
    class IT_BUS_API TransportCallback
    {
      public:
        ...
        virtual void
        dispatch(
            BinaryBuffer& request_message,
            DispatchInfo& dispatch_context,
            bool          dispatch_acynchronously_if_possible = 0
        ) = 0;
};
```

The asynchronous dispatch policy is an optimization that enables you to decouple the reader thread from the dispatch processing.

If the asynchronous dispatch policy is `true`, the `dispatch()` function returns immediately after adding the request message to a work queue.

If the asynchronous dispatch policy is `false`, the `dispatch()` function remains blocked until the dispatch processing is complete.

Note: As of Artix 3.0.2, the asynchronous dispatch policy has *not* yet been implemented. That is, the `dispatch()` function always blocks. The non-blocking functionality will be implemented in a later release.

RPC-Style Dispatch

Overview

Some implementations of a server transport could be layered over a Remote Procedure Call (RPC) transport infrastructure. For this kind of transport, it is more convenient if the upcall blocks until the reply buffer becomes available (synchronous invocation). Figure 16 shows an overview of an RPC-style dispatch call.

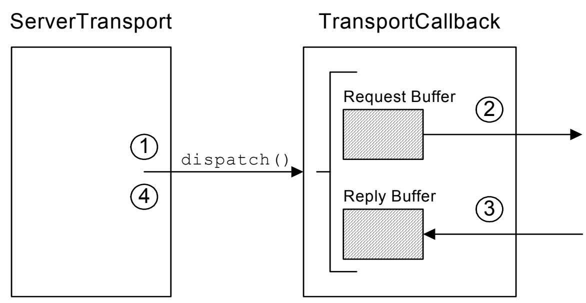

Figure 16: *Overview of RPC-Style Dispatch*

Dispatch steps

The stages shown in Figure 16 can be described as follows:

Stage	Description
1	The server transport code calls `dispatch()` on the `TransportCallback` object, passing in a reference to the request buffer.
2	The `TransportCallback` object processes the request message, resulting in an upcall to the relevant servant object.
3	After processing the request, the `TransportCallback` writes the reply data into the reply buffer. **Note:** The reply buffer must be supplied to the `TransportCallback` object in advance, using the `DispatchInfo::provide_response_buffer()` function. For details, see Example 42 on page 125.

Stage	Description
4	The `dispatch()` call remains blocked until the reply buffer is written. After `dispatch()` returns, therefore, the reply buffer is available and ready to be sent back to the client.

Setting the requisite policies

To set the transport policies, create an `IT_Bus::TransportPolicyList` instance, initialize it with the relevant policy values, and then return the policy list from the `TransportFactory::get_policies()` function. Example 41 shows how to implement a transport factory with the policies required for RPC-style dispatch.

Example 41: *Setting Policies for RPC-Style Dispatch*

```
// C++
void
TransportFactoryImpl::initialize(Bus_ptr bus)
{
   m_transport_policylist =
      bus->get_pdk_bus()->create_transport_policy_list();
   m_transport_policylist->set_policy_requires_stack_unwind(
      true
   );
}

const TransportPolicyList*
TransportFactoryImpl::get_policies()
{
    return m_transport_policylist;
}
```

Implementation example

The code fragment in Example 42 shows how to make an upcall into the Artix application using RPC-style dispatch. This code fragment could appear in the body of the `ServerTransport::activate()` function, in the body of the `ServerTransport::run()` function, or in a completely different object, depending on the type of threading model that is used (see “Transport Threading Models” on page 108).

Example 42: *Making an Upcall Using RPC-Style Dispatch*

```
// C++
DispatchInfo& dispatch_context =
   m_callback->get_dispatch_context();

dispatch_context.provide_response_buffer(
    vvReceiveBuffer
);

m_callback->dispatch(
    vvSendBuffer,
    dispatch_context
);

// At this point, vvReceiveBuffer contains the reply message.
```

Messaging-Style Dispatch

Overview

The default style of dispatching used by the Artix server transport is *messaging-style dispatch*, which is suitable for message-oriented transports such as the MQ-Series transport. For this kind of transport, the upcall returns as soon as it has dispatched the request buffer. The reply buffer is returned asynchronously, through a callback on the `ServerTransport::send()` function. Figure 17 shows an overview of a messaging-style dispatch call.

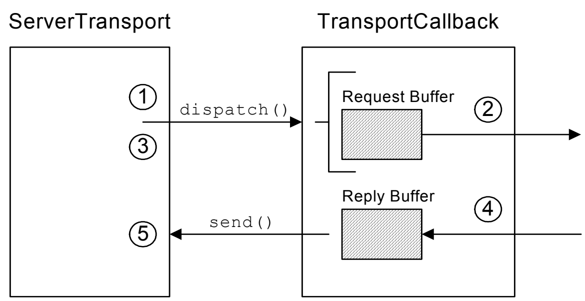

Figure 17: *Overview of Messaging-Style Dispatch*

Dispatch steps

The stages shown in Figure 17 can be described as follows:

Stage	Description
1	The server transport code calls `dispatch()` on the `TransportCallback` object, passing in a reference to the request buffer.
2	The `TransportCallback` object processes the request message, resulting in an upcall to the relevant servant object.
3	The `dispatch()` call returns directly after dispatching the request message.

Stage	Description
4	After processing the request, the `TransportCallback` writes the reply data into the reply buffer.
5	The Artix runtime calls `send()` on the `ServerTransport` object, passing in a reference to the reply buffer.

Setting the requisite policies

Normally, there is no need to set transport policies explicitly for messaging-style dispatch, because it is the default. If you do set some transport policies, however, you must be sure that the value of the *requires stack unwind policy* is `false` (the default).

Implementation example

The code fragment in Example 43 shows how to make an upcall into the Artix application using messaging-style dispatch. This code fragment could appear in the body of the `ServerTransport::activate()` function, in the body of the `ServerTransport::run()` function, or in a completely different object, depending on the type of threading model that is used (see "Transport Threading Models" on page 108).

Example 43: *Making an Upcall Using Messaging-Style Dispatch*

```
// C++
DispatchInfo& dispatch_context =
   m_callback->get_dispatch_context();

m_callback->dispatch(
    vvSendBuffer,
    dispatch_context
);

// At this point, vvReceiveBuffer contains the reply message.
```

In addition to dispatching the request buffer, you must implement the `ServerTransport::send()` function to receive the callback containing the reply buffer. Example 44 shows an outline implementation of the `send()` function, which is suitable for message-style dispatch.

Example 44: *Implementation of send() for Message-Style Dispatch*

```
// C++
void
ServerTransportImpl::send(
    BinaryBuffer& reply_message,
    DispatchInfo& dispatch_context
)
{
    // Send the reply_message over the transport layer
    // back to the client.
    ... // (transport-specific details)
}
```

Accessing Contexts

Overview

Contexts are an Artix mechanism that enables application code to communicate with plug-ins. Contexts are typically used by transports for the following purposes:

- Setting connection parameters (for example, timeouts).
- Sending data in message headers (either as part of a request message or a reply message).

This section describes how to access and use contexts from within a transport implementation.

> **Note:** Although Artix contexts are accessible from the transport, in many cases it is more appropriate to access contexts from within an interceptor. The use of interceptors makes your code more modular: you can load individual interceptors independently of the transport.

Accessing contexts on the client side

The following extract from the `IT_Bus::ClientTransport` class shows how you can access Artix contexts from the `connect()`, `invoke_oneway()`, and `invoke()` functions.

```
// C++
namespace IT_Bus
{
    class IT_BUS_API ClientTransport
    {
      public:
        virtual void
        connect(
            ContextContainer* out_context_container
        ) = 0;
        ...
        virtual void
        invoke_oneway(
            const IT_WSDL::WSDLOperation& wsdl_operation,
            const BinaryBuffer&           request_buffer,
            ContextContainer*             out_container,
            ContextContainer*             in_container
        ) = 0;
```

```
        virtual void
        invoke(
            const IT_WSDL::WSDLOperation& wsdl_operation,
            const BinaryBuffer&           request_buffer,
            BinaryBuffer&                 response_buffer,
            ContextContainer*             out_container,
            ContextContainer*             in_container
        ) = 0;
        ...
    };
};
```

In each of these functions, the contexts are used as follows:

- `connect()` function—the outgoing context container could contain settings that influence the transport connection (for example, connection timeouts). You can define your own context type specifically for this purpose.
- `invoke_oneway()` function—contexts can be used to send and receive header information across a transport protocol, as follows:
    - If there is outgoing data to send in a header, the transport implementation reads it from the relevant outgoing context (obtained from `out_container`) and inserts it into a request message header.
    - If there is incoming data to receive from a header, the transport implementation extracts it from the reply message and writes it into the relevant incoming context (obtained from `in_container`).

  Note: Incoming reply contexts (read from incoming reply messages) are supported, even though this is a oneway WSDL operation. Oneway operations are *not* necessarily implemented as oneways by the transport layer. Sometimes, it is necessary to extract context data from reply messages, even for oneway operations.

- `invoke()` function—both outgoing contexts and incoming contexts are available, just as for the `invoke_oneway()` function.

Accessing contexts with RPC-style dispatch

On the server side, incoming contexts and outgoing contexts are accessible through the current `IT_Bus::DispatchInfo` object. For example, the code for accessing contexts within an RPC-style dispatch would have the following general outline:

```
// C++
DispatchInfo& dispatch_context =
   m_callback->get_dispatch_context();

dispatch_context.provide_response_buffer(
    vvReceiveBuffer
);

ContextContainer& incoming_container =
   dispatch_context.get_incoming_context_container();

// Process each incoming context as follows:
// 1. Extract the relevant header data from the incoming request.
// 2. Obtain the relevant context instance from the
//    incoming_container.
// 3. Populate the context instance with the header data.

m_callback->dispatch(
    vvSendBuffer,
    dispatch_context
);

ContextContainer& outgoing_container =
   dispatch_context.get_outgoing_context_container();

// Process each outgoing context as follows:
// 1. Obtain the relevant context instance from the
//    outgoing_container.
// 1. Read the context data from the context instance.
// 3. Marshal the context data into an outgoing reply header.
```

Accessing contexts with messaging-style dispatch

With messaging-style dispatch, there are two different points in the code where you access contexts. Firstly, to access incoming contexts, you need to insert some code before the `TransportCallback::dispatch()` call, as follows:

```
// C++
DispatchInfo& dispatch_context =
   m_callback->get_dispatch_context();

dispatch_context.provide_response_buffer(
    vvReceiveBuffer
);

ContextContainer& incoming_container =
   dispatch_context.get_incoming_context_container();

// Process each incoming context as follows:
// 1. Extract the relevant header data from the incoming request.
// 2. Obtain the relevant context instance from the
//    incoming_container.
// 3. Populate the context instance with the header data.

m_callback->dispatch(
    vvSendBuffer,
    dispatch_context
);
```

Next, to access outgoing contexts, you need to insert some code into the `ServerTransport::send()` function, as follows:

```
// C++
void
ServerTransportImpl::send(
    BinaryBuffer& reply_message,
    DispatchInfo& dispatch_context
)
{
    ...
    ContextContainer& outgoing_container =
        dispatch_context.get_outgoing_context_container();

    // Process each outgoing context as follows:
    // 1. Obtain the relevant context instance from the
    //    outgoing_container.
    // 1. Read the context data from the context instance.
```

```
    // 3. Marshal the context data into an outgoing reply header.
    ...
}
```

Oneway Semantics

Overview

WSDL syntax allows you to define two different kinds of operations:

- *Normal operations*—which include one or more output messages.
- *Oneway operations*—which include *only* input messages.

In general, the remote invocation of a oneway operation can be optimized so that it consists only of a request message; there is no need to wait for a reply message, because no data is expected in the reply. This is a valuable optimization, which is supported by Artix.

Oneway semantics on the client side

When it comes to implementing oneway semantics on a specific transport, however, there can be a mismatch between the WSDL notion of a oneway and the semantics supported by the underlying transport protocol. For example, the HTTP protocol requires that you must always send an acknowledgment reply (HTTP 202 OK reply), even if there is no reply data.

To give you sufficient flexibility to implement oneways, therefore, the `ClientTransport` class requires you to implement separate functions for handling normal operations and oneway operations, as follows:

- `ClientTransport::invoke()` function—called when the WSDL operation includes one or more output messages.
- `ClientTransport::invoke_oneway()` function—called when the WSDL operation includes only input messages.

Oneway semantics with RPC-style dispatch

Within the section of code that implements an RPC-style dispatch on the server side, you can check whether a WSDL operation is oneway by calling the `DispatchInfo::is_oneway()` function. If the operation is oneway, you should handle it in the appropriate way for the particular transport protocol.

For example, the code for performing an RPC-style dispatch would have the following general outline:

```
// C++
DispatchInfo& dispatch_context =
   m_callback->get_dispatch_context();

dispatch_context.provide_response_buffer(
```

```
    vvReceiveBuffer
);

m_callback->dispatch(
    vvSendBuffer,
    dispatch_context
);

if (! dispatch_context.is_oneway() ) {
    // Normal (two-way) WSDL operation

    // Use transport to send vvReceiveBuffer reply to client.
}
else {
    // Oneway WSDL operation
    // (vvReceiveBuffer is empty in this case)

    // HTTP protocol example: send an acknowledgment.

    // MQ-Series example: do not send any reply.
}
```

Oneway semantics with messaging-style dispatch

Within the implementation of the `IT_Bus::ServerTransport::send()` function (which is responsible for sending replies back to the client), you can check whether a WSDL operation is oneway by calling the `DispatchInfo::is_oneway()` function. If the operation is oneway, you should handle it in the appropriate way for the particular transport protocol.

For example, an implementation of `ServerTransport::send()` would have the following general outline:

```
// C++
void
ServerTransportImpl::send(
    BinaryBuffer& reply_message,
    DispatchInfo& dispatch_context
)
{
    if (! dispatch_context.is_oneway()) {
        // Normal (two-way) WSDL operation

        // Use transport to send reply_message back to client.
    }
    else {
        // Oneway WSDL operation
```

```
        // HTTP protocol example: send an acknowledgment
        //                        before returning.

        // MQ-Series example: return immediately.
    }
}
```

Stub Transport Example

Overview

The stub transport is a very simple transport that facilitates communication between a client and a server that are colocated in the same process. The client transport object holds a pointer that points directly at the server transport object. When the client has a message to send to the server, it simply invokes a dispatch function directly on the server transport object.

For this transport to work, the client and server *must* be colocated. This transport is potentially useful as a diagnostic tool: it enables you to send messages through the binding layers, without doing any significant work at the transport layer.

In this section

This section contains the following subsections:

Implementing the Client Transport

Overview

This subsection describes how to make a custom implementation of the `IT_Bus::ClientTransport` class, using the stub client transport as an example. The purpose of the client transport class is to manage connections and send/receive messages in binary format.

Sequence of call

Artix calls back on the client transport functions in the following sequence:

1. `initialize()`—called once, to configure the port.
2. `connect()`—called once, to establish a connection to the remote host. The `connect()` function should be non-blocking.
3. `invoke()`/`invoke_oneway()`—called for each WSDL operation invocation, depending on whether it is a normal operation or a oneway operation.
4. `disconnect()`—called once, to close the connection to the remote host.

Client transport header

Example 45 shows the header file for the stub plug-in's client transport class.

Example 45: *Header for the StubClientTransport Class*

```
// C++
#include <it_bus_sys/bus_context.h>
#include <it_bus_pdk/messaging_transport.h>
#include "stub_transport_factory.h"
#include "stub_transport_wsdl_address.h"

namespace IT_Transport_Stub
{
1       class StubClientTransport : public IT_Bus::ClientTransport
        {
          public:
            StubClientTransport(
2                   ServerTransportMap & server_transport_map
            );
            virtual ~StubClientTransport();
```

Example 45: *Header for the StubClientTransport Class*

```
3       virtual void
        initialize(const IT_WSDL::WSDLPort& Configuration);

        virtual IT_WSDL::WSDLExtensionElement&
        get_configuration();

        virtual void
        connect(IT_Bus::ContextContainer* out_context_container);

        virtual void disconnect();

        virtual void
        invoke_oneway(
            const IT_WSDL::WSDLOperation& wsdl_operation,
            const IT_Bus::BinaryBuffer&   request_buffer,
            IT_Bus::ContextContainer*     out_container,
            IT_Bus::ContextContainer*     in_container
        );

        virtual void
        invoke(
            const IT_WSDL::WSDLOperation& wsdl_operation,
            const IT_Bus::BinaryBuffer&   request_buffer,
            IT_Bus::BinaryBuffer&         response_buffer,
            IT_Bus::ContextContainer*     out_container,
            IT_Bus::ContextContainer*     in_container
        );

      protected:
4       ServerTransportMap &       m_server_transport_map;
5       StubServerTransport *      m_server_transport;
6       StubTransportWSDLAddress * m_address_element;
7       IT_Bus::BinaryBuffer       m_received;

      private:
        virtual void send(
            const IT_WSDL::WSDLOperation& wsdl_operation,
            const IT_Bus::BinaryBuffer& vvSendBuffer,
            IT_Bus::ContextContainer*  out_context_container
        );

        virtual void receive(
            const IT_WSDL::WSDLOperation& wsdl_operation,
            IT_Bus::BinaryBuffer& vvReceiveBuffer,
            IT_Bus::ContextContainer*  in_context_container
```

Example 45: *Header for the StubClientTransport Class*

```
        );
    };
};
```

The preceding transport class header can be explained as follows:

1. The tunnel client transport class must inherit from `IT_Bus::ClientTransport`.
2. The `IT_Transport_Stub::ServerTransportMap` type is a typedef of `IT_Bus::StringMap<StubServerTransport *>`, defined in the stub plug-in's transport factory header. The `ServerTransportMap` class is a hash table that uses a string as the key to retrieve a server transport instance. This hash table is the discovery mechanism used by the stub plug-in to find a colocated server transport instance.
3. The following functions, `initialize()`, `get_configuration()`, `connect()`, `disconnect()`, `send()`, and `receive()`, are all inherited from the `IT_Bus::ClientTransport` base class.
4. The `m_server_transport_map` variable stores a reference to the `ServerTransportMap` instance passed into the constructor.
5. The `m_server_transport` variable stores a pointer to the target server transport instance.
6. The `m_address_element` variable stores a pointer to the `stub:address` WSDL element that defines the location of the server transport.
7. The `m_received` binary buffer is used to store received messages temporarily.

Client transport implementation

Example 46 shows the implementation of the client transport class.

Example 46: *Implementation of the StubClientTransport Class*

```
// C++
#include "stub_client_transport.h"
#include "stub_transport_wsdl_extension_factory.h"
#include "stub_server_transport.h"

using namespace IT_Bus;
using namespace IT_WSDL;
```

Example 46: *Implementation of the StubClientTransport Class*

```
IT_Transport_Stub::StubClientTransport::StubClientTransport(
    ServerTransportMap & server_transport_map
)
: m_server_transport_map(server_transport_map)
{
    m_server_transport = 0;
    m_address_element = 0;

}

IT_Transport_Stub::StubClientTransport::~StubClientTransport()
{
}

void
1 IT_Transport_Stub::StubClientTransport::initialize(
    const IT_WSDL::WSDLPort& wsdl_port
)
{
    // get address from the WSDL
    //
    String location;
    //address extensor
    WSDLExtensionElement* wsdl_element =
2       StubTransportWSDLExtensionFactory::get_extension_element(
            wsdl_port,
            StubTransportWSDLAddress::ELEMENT_NAME
        );

    m_address_element =
        IT_DYNAMIC_CAST(StubTransportWSDLAddress *,
   wsdl_element);

    if (m_address_element != 0)
    {
        location = m_address_element->get_location();
    }
}

IT_WSDL::WSDLExtensionElement&
3 IT_Transport_Stub::StubClientTransport::get_configuration()
{
    IT_WSDL::WSDLExtensionElement * elem = 0;
    return *elem;
```

Example 46: *Implementation of the StubClientTransport Class*

```
  }

  void
4 IT_Transport_Stub::StubClientTransport::connect(
      ContextContainer* out_context_container
  )
  {
5     String location = m_address_element->get_location();

6     ServerTransportMap::iterator iter =
          m_server_transport_map.find(location);

      if (iter == m_server_transport_map.end())
      {
          throw Exception(
              "Couldn't find server for stub transport address",
              location.c_str()
          );
      }

      m_server_transport = (*iter).second;
  }

  void
7 IT_Transport_Stub::StubClientTransport::disconnect()
  {
  }

  void
  IT_Transport_Stub::StubClientTransport::invoke_oneway(
      const WSDLOperation& wsdl_operation,
      const BinaryBuffer&  request_buffer,
      ContextContainer*    out_container,
      ContextContainer*    //in_container
  )
  {
      send(
          wsdl_operation,
          request_buffer,
          out_container
      );
  }
```

Example 46: *Implementation of the StubClientTransport Class*

```
void
IT_Transport_Stub::StubClientTransport::invoke(
    const WSDLOperation& wsdl_operation,
    const BinaryBuffer&  request_buffer,
    BinaryBuffer&        response_buffer,
    ContextContainer*    out_container,
    ContextContainer*    in_container
)
{
    send(
        wsdl_operation,
        request_buffer,
        out_container
    );

    receive(
        wsdl_operation,
        response_buffer,
        in_container
    );
}

void
8 IT_Transport_Stub::StubClientTransport::send(
    const IT_WSDL::WSDLOperation& wsdl_operation,
    const BinaryBuffer& vvSendBuffer,
    ContextContainer*  out_context_container
)
{
    BinaryBuffer send_buffer(vvSendBuffer);
9   m_server_transport->dispatch(send_buffer, m_received);
}

void
10 IT_Transport_Stub::StubClientTransport::receive(
    const IT_WSDL::WSDLOperation& wsdl_operation,
    BinaryBuffer& vvReceiveBuffer,
    ContextContainer*  in_context_container
)
{
    vvReceiveBuffer.attach(m_received);
}
```

The preceding client transport implementation can be explained as follows:

1. The main purpose of the `initialize()` function is to initialize the configuration of the client transport port. The `wsdl_port` parameter is an object of `IT_WSDL::WSDLPort` type, which is a parse-tree node containing the data from a WSDL `<port ... > </port>` element.
2. The `get_extension_element()` static function searches the WSDL port node to find a *`StubPrefix`*`:address` sub-element, which is then stored in `m_address_element`. See "Implementing the Extension Element Classes" on page 88 for details.
3. The `get_configuration()` function has a dummy implementation.
4. The `connect()` function is responsible for establishing a connection to a service endpoint. In the case of the stub transport, it attempts to find the colocated server transport instance identified by the `location` attribute from the `<`*`StubPrefix`*`:address>` tag.
5. The `get_location()` function returns the value of the `location` attribute from the `<`*`StubPrefix`*`:address>` tag.
6. Search the server transport map, using the location attribute as a key, in order to find a colocated `StubServerTransport` instance.

   The entries in the `ServerTransportMap` hash table are created by one or more colocated `StubServerTransport` instances.
7. The `disconnect()` function has a dummy implementation. No action is needed to disconnect from a stub server transport.
8. The `send()` function transmits a WSDL request in the form of a binary buffer, `request_buffer`.
9. For the stub transport, the implementation of `send()` is trivial: you invoke `dispatch()` directly on the colocated stub server transport instance.
10. The `receive()` function returns the binary buffer, `m_received`, that was stored from the previous call to `send()`.

Implementing the Server Transport

Overview

This subsection describes how to make a custom implementation of the `IT_Bus::ServerTransport` class, using the stub server transport as an example. The purpose of the server transport class is to listen for client connection attempts, listen for incoming messages and to dispatch incoming messages up to the Artix binding layer.

Server transport header

Example 47 shows the stub plug-in's server transport class:

Example 47: *Header for the StubServerTransport Class*

```
// C++
#include <it_bus_pdk/messaging_transport.h>
#include <it_bus_sys/bus_context.h>
#include "stub_transport_wsdl_address.h"
#include "stub_transport_factory.h"

namespace IT_Transport_Stub
{
1    class StubServerTransport : public IT_Bus::ServerTransport
    {
      public:
        StubServerTransport(
            ServerTransportMap & server_transport_map,
            const IT_WSDL::WSDLPort& wsdl_port
        );
        virtual ~StubServerTransport();

2       virtual void
        activate(
            IT_Bus::TransportCallback&  callback,
            IT_WorkQueue::WorkQueue_ptr work_queue = 0
        );

        virtual IT_WSDL::WSDLExtensionElement&
        get_configuration();

        virtual void deactivate();

        virtual void shutdown();

        virtual void
```

Example 47: *Header for the StubServerTransport Class*

```
        send(
            IT_Bus::BinaryBuffer& reply_message,
            IT_Bus::DispatchInfo& dispatch_context
        );

        void dispatch(
            IT_Bus::BinaryBuffer& vvSendBuffer,
            IT_Bus::BinaryBuffer& vvReceiveBuffer
        );

      protected:
3       StubTransportWSDLAddress *  m_address_element;
4       IT_Bus::TransportCallback * m_callback;
5       ServerTransportMap &        m_server_transport_map;
    };
};
```

The preceding server transport header can be described as follows:

1. The tunnel server transport class must inherit from `IT_Bus::ServerTransport`.
2. The following functions, `activate()`, `get_configuration()`, `deactivate()`, `shutdown()`, `send()`, and `dispatch()`, are all inherited from the `IT_Bus::ServerTransport` base class.
3. The `m_address_element` variable stores a pointer to the `<`*`StubPrefix`*`:address>` WSDL element that defines the location of the server transport.
4. The `m_callback` variable stores a pointer to the `TransportCalback` object, which is used to dispatch requests to the next layer on the server side.
5. The `m_server_transport_map` variable stores a reference to the `ServerTransportMap` instance, which holds a hash table consisting of pairs of location attribute string and pointer to `StubServerTransport`.

Server transport implementation

Example 48 shows the implementation of the server transport class.

Example 48: *Implementation of the StubServerTransport Class*

```
// C++
```

Example 48: *Implementation of the StubServerTransport Class*

```
#include "stub_server_transport.h"
#include "stub_transport_wsdl_extension_factory.h"

using namespace IT_Bus;
using namespace IT_WSDL;


1 IT_Transport_Stub::StubServerTransport::StubServerTransport(
    ServerTransportMap & server_transport_map,
    const WSDLPort& wsdl_port
)
: m_server_transport_map(server_transport_map)
{
    m_callback = 0;
    // get address from the WSDL
    //
    String location;
    //address extensor
    WSDLExtensionElement* wsdl_element =
2       StubTransportWSDLExtensionFactory::get_extension_element(
            wsdl_port,
            StubTransportWSDLAddress::ELEMENT_NAME
        );

    m_address_element =
      IT_DYNAMIC_CAST(StubTransportWSDLAddress *, wsdl_element);

    if (m_address_element != 0)
    {
        location = m_address_element->get_location();
    }
}

IT_Transport_Stub::StubServerTransport::~StubServerTransport()
{
}

void
3 IT_Transport_Stub::StubServerTransport::activate(
    IT_Bus::TransportCallback & callback,
    IT_WorkQueue::WorkQueue_ptr work_queue
)
{
    m_callback = &callback;
```

Example 48: *Implementation of the StubServerTransport Class*

```
4       m_server_transport_map.insert(
            ServerTransportMap::value_type(
                m_address_element->get_location(),
                this
            )
        );

5       m_callback->transport_activated();
    }

    IT_WSDL::WSDLExtensionElement&
6   IT_Transport_Stub::StubServerTransport::get_configuration()
    {
        IT_WSDL::WSDLExtensionElement * elem = 0;
        return *elem;
    }

    void
7   IT_Transport_Stub::StubServerTransport::deactivate()
    {
        // Note: It is impossible to deactivate the stub transport
        // m_callback->transport_deactivated();
    }

    void
8   IT_Transport_Stub::StubServerTransport::shutdown()
    {
        ServerTransportMap::iterator iter =
     m_server_transport_map.find(m_address_element->get_location());

        if (iter != m_server_transport_map.end())
        {
            m_server_transport_map.erase(iter);
        }

9       m_callback->transport_shutdown_complete();
    }

    void
10  IT_Transport_Stub::StubServerTransport::send(
        BinaryBuffer& reply_message,
        DispatchInfo& dispatch_context
    )
    {
        assert(0);
```

Example 48: *Implementation of the StubServerTransport Class*

```
}

void
11 IT_Transport_Stub::StubServerTransport::dispatch(
    BinaryBuffer& vvSendBuffer,
    BinaryBuffer& vvReceiveBuffer
)
{
    DispatchInfo& dispatch_context =
        m_callback->get_dispatch_context();

12  dispatch_context.provide_response_buffer(
        vvReceiveBuffer
    );

13  m_callback->dispatch(
        vvSendBuffer,
        dispatch_context
    );
}
```

The preceding server transport implementation can be described as follows:

1. The `StubServerTransport` constructor receives two parameters from the transport factory:
    - `server_transport_map`—a `String` to `StubServerTransport*` map, which is used to advertize the availability of stub server transports to stub client transports.
    - `wsdl_port`—an object of `IT_WSDL::WSDLPort` type, which is a parse-tree node containing the data from a WSDL `<port ... > </port>` element.
2. The `get_extension_element()` static function searches the WSDL port node to find a *`StubPrefix`*`:address` sub-element, which is then stored in `m_address_element`. See "Implementing the Extension Element Classes" on page 88 for details.
3. The `activate()` function is called by the Artix core to start up the server transport. It takes the following arguments:

- `callback`—the `TransportCallback` object is used to communicate with the Artix core. In particular, `TransportCallback::dispatch()` is used to dispatch requests up to the application code.
- `work_queue`—this is a NULL pointer, unless you choose the `BORROWS_WORKQUEUE_SELF_DRIVEN` threading resources policy.

The `deactivate()` and `activate()` functions can be used to pause and resume the server transport. The `activate()` function must be non-blocking.

4. Advertise this `StubServerTransport` instance by adding an entry to the server transport map. Because the colocated stub client transports have a reference to the same server transport map instance, they will be able to find the stub server transport by supplying the relevant `location` value as a key.
5. Before exiting the body of the `activate()` function, you must notify the Artix core of the current activation status by calling back on the `IT_Bus::TransportCallback` object. There are two alternatives:
   - `TransportCallback::transport_activated()`—call this, if the transport activation is successfull.
   - `TransportCallback::transport_activation_failed()`—call this, if the transport activation fails.
6. The `get_configuration()` function has a dummy implementation.
7. The `deactivate()` function is called in order to deactivate the server transport temporarily. It can be used in combination with `activate()` to pause and resume the server transport.

   Before exiting the body of the `deactivate()` function, you must notify the Artix core by calling `TransportCallback::transport_deactivated()`.

   Note: The stub server transport is a special case, however, because it is not possible to deactivate it. Strictly speaking, therefore, we ought *not* to include the `transport_deactivated()` call here.

8. The `shutdown()` function is called by the Artix core while the Bus shuts down. At this point, you should shut down the server transport and perform whatever cleanup is necessary.

9. Before exiting the body of the `shutdown()` function, you must notify the Artix core by calling `TransportCallback::transport_shutdown_complete()`.
10. The `send()` function is called, only if you have configured the server transport to use the asynchronous dispatch model. Because the stub transport uses the synchronous dispatch model, the `send()` function is left unimplemented.

    The choice between a synchronous or an asynchronous dispatch model is selected by the *requires stack unwind policy*. If the policy is `true`, the synchronous model is selected; if `false`, the asynchronous model is selected. For more details see "Implementing the Transport Factory" on page 152.
11. This `dispatch()` function is *not* inherited from `IT_Bus::ServerTransport`. It is specific to the stub transport. The `dispatch()` function represents a simple mechanism for stub client transports to send a request and receive a reply from the stub server transport: the client transport simply makes a colocated call on the `StubServerTransport::dispatch()` function.
12. Because this server transport uses the synchronous dispatch model, you must call `DispatchInfo::provide_response_buffer()` to provide a buffer into which the reply message will be written.
13. Call `TransportCallback::dispatch()` to dispatch the request message to the next stage. Because the transport uses the synchronous dispatch model, the reply message is available in the buffer, `vvReceiveBuffer`, as soon as the `TransportCallback::dispatch()` call returns.

Implementing the Transport Factory

Overview

You must implement a transport factory as part of the stub transport implementation. The Artix core calls functions on the transport factory to create `IT_Bus::ClientTransport` and `IT_Bus::ServerTransport` instances as needed.

Transport factory header

Example 49 shows the stub plug-in's transport factory header.

Example 49: *Header for the StubTransportFactory Class*

```
// C++
#include <it_bus/bus.h>
#include <it_bus_pdk/messaging_transport.h>
#include <it_bus/string_map.h>

namespace IT_Transport_Stub
{
    class StubServerTransport;

1   typedef IT_Bus::StringMap<StubServerTransport *>
                ServerTransportMap;

2   class StubTransportFactory : public IT_Bus::TransportFactory
    {
      public:
        StubTransportFactory();
        virtual ~StubTransportFactory();

        virtual IT_Bus::ClientTransport *
        create_client_transport();

        virtual void destroy_client_transport(
            IT_Bus::ClientTransport * transport
        );

        virtual IT_Bus::ServerTransport*
        create_server_transport(
            const IT_WSDL::WSDLPort& configuration
        );

        virtual void
        destroy_server_transport(
```

Example 49: *Header for the StubTransportFactory Class*

```
            IT_Bus::ServerTransport* transport
        );

        virtual IT_Bus::ThreadingModel
        get_client_threading_model();

        virtual void
        register_wsdl_extension_factories(
            IT_WSDL::WSDLFactory & factory
        ) const;

        virtual void
        deregister_wsdl_extension_factories(
            IT_WSDL::WSDLFactory & factory
        ) const;

        virtual const IT_Bus::TransportPolicyList*
        get_policies();

        void
        initialize(
            IT_Bus::Bus_ptr bus
        );

      protected:
        ...
3       ServerTransportMap           m_server_transport_map;
4       IT_Bus::TransportPolicyList* m_transport_policylist;
    };
};
```

The preceding header file can be explained as follows:

1. The `ServerTransportMap` type is defined to be a hash table that uses a string key to find pointers to `StubServerTransport` instances. The server transport map is the endpoint discovery mechanism for the stub transport.
2. The `StubTransportFactory` class inherits from the `IT_Bus::TransportFactory` base class.
3. The `m_server_transport_map` variable is the concrete server transport map instance, which is referenced by the client transport objects and the server transport objects.

4. The `m_transport_policylist` variable stores a pointer to an object that encapsulates the stub transport's threading policies.

Transport factory implementation

Example 50 shows the transport factory implementation.

Example 50: *Implementation of the StubTransportFactory Class*

```
// C++
#include <it_bus_pdk/pdk_bus.h>
#include "stub_transport_factory.h"
#include "stub_client_transport.h"
#include "stub_server_transport.h"

#include "stub_transport_wsdl_extension_factory.h"

using namespace IT_Bus;

IT_Transport_Stub::StubTransportFactory::StubTransportFactory()
{
}

IT_Transport_Stub::StubTransportFactory::~StubTransportFactory()
{
    delete m_transport_policylist;
}

IT_Bus::ClientTransport *
1 IT_Transport_Stub::StubTransportFactory::create_client_transport
    ()
{
    return new
    IT_Transport_Stub::StubClientTransport(m_server_transport_map
    );
}

void
2 IT_Transport_Stub::StubTransportFactory::destroy_client_transpor
    t(
    IT_Bus::ClientTransport * transport
)
{
    delete transport;
}

IT_Bus::ServerTransport*
```

Example 50: *Implementation of the StubTransportFactory Class*

```
3  IT_Transport_Stub::StubTransportFactory::create_server_transport
    (
    const IT_WSDL::WSDLPort& wsdl_port
)
{
    return new IT_Transport_Stub::StubServerTransport(
                    m_server_transport_map,
                    wsdl_port
                );
}

void
4  IT_Transport_Stub::StubTransportFactory::destroy_server_transpor
    t(
    IT_Bus::ServerTransport* transport
)
{
    delete transport;
}

IT_Bus::ThreadingModel
5  IT_Transport_Stub::StubTransportFactory::get_client_threading_mo
    del()
{
    return IT_Bus::MULTI_INSTANCE;
}

6  extern IT_Transport_Stub::StubTransportWSDLExtensionFactory
    it_glob_stub_transport_wsdl_extension_factory_instance;

void
7  IT_Transport_Stub::StubTransportFactory::register_wsdl_extension
    _factories(
    IT_WSDL::WSDLFactory & factory
) const
{
8      factory.register_extension_factory(
        "http://schemas.iona.com/transports/stub",
        it_glob_stub_transport_wsdl_extension_factory_instance
    );
}

void
9  IT_Transport_Stub::StubTransportFactory::deregister_wsdl_extensi
    on_factories(
```

Example 50: *Implementation of the StubTransportFactory Class*

```
    IT_WSDL::WSDLFactory & factory
) const
{
}

const TransportPolicyList*
10 IT_Transport_Stub::StubTransportFactory::get_policies()
{
    return m_transport_policylist;
}

void
11 IT_Transport_Stub::StubTransportFactory::initialize(
    Bus_ptr bus
)
{
    m_transport_policylist =
        bus->get_pdk_bus()->create_transport_policy_list();

12 m_transport_policylist->set_policy_threading_resources(EXTERNALL
    Y_DRIVEN);
13 m_transport_policylist->set_policy_requires_concurrent_dispatch(
    true);
14 m_transport_policylist->set_policy_requires_stack_unwind(true);
}
```

The preceding transport factory implementation can be explained as follows:

1. The `create_client_transport()` function is called by the Artix core whenever a new `StubClientTransport` instance is needed. The `StubClientTransport` constructor takes on parameter: a reference to the server transport map, which enables the stub client transport to discover stub service endpoints.
2. The `destroy_client_transport()` function is normally implemented exactly as shown here.
3. The `create_server_transport()` function is called by the Artix core whenever a new `StubServerTransport` instance is needed. The `StubServerTransport` constructor takes two parameters:
    - A reference to the server transport map, which enables the stub server transport to advertise its existence to colocated clients.

- A reference to the WSDL port that contains a description of this service endpoint.

4. The `destroy_server_transport()` function is normally implemented exactly as shown here.
5. The `get_client_threading_model()` is implemented to select the `MULTI_INSTANCE` client threading model.
6. This variable references a global static instance of the stub plug-in's WSDL extension factory.
7. The `register_wsdl_extension_factories()` function is called by the Artix core while the stub plug-in is initializing. It gives you an opportunity to register one or more WSDL extension factories with the Bus.
8. This line registers the stub plug-in's WSDL extension factory, associating it with the `http://schemas.iona.com/transports/stub` namespace URI. This is the namespace that can be associated with the *`StubPrefix`* to let you configure the *`StubPrefix`*`:address` element in your WSDL contract.
9. As the stub plug-in shuts down, it calls `deregister_wsdl_extension_factories()`.
10. As the stub plug-in starts up, the Artix core calls `get_policies()` to discover what policies are to be used with this transport plug-in (the policies are mostly concerned with server threading).
11. If you need to customize the transport policy list, you can do this in the body of the `initialize()` function.
12. Usually, when the server transport's threading policy is set to `EXTERNALLY_DRIVEN`, it would imply that the server transport code creates its own reader threads to process incoming requests. In this case, because the stub transport is a colocated transport, the situation is somewhat exceptional. The reader thread is actually provided by the client side—the client transport simply calls the server transport's `dispatch()` function directly.
13. The server's concurrent dispatch policy is set to `true`. This indicates to the Artix core that the stub server transport is liable to make concurrent dispatches to the server-side binding (by calling `TransportCallback::dispatch()` from multiple threads).

14. The requires stack unwind policy is set to `true`. This selects a synchronous approach to dispatching requests on the server side. If you enable the stack unwind policy, you must implement your server transport according to the following pattern:
    - Do not implement `ServerTransport::send()` (this function is only used to receive replies asynchronously).
    - In the implementation of `ServerTransport::dispatch()`, prior to calling `TransportCallback::dispatch()`, call `DispatchContext::provide_response_buffer()` to specify a buffer into which the result will be written.
    - As soon as `TransportCallback::dispatch()` returns, the response buffer contains the reply.

Registering and Packaging the Transport

Stub plug-in name

Example 51 shows how to register the stub transport plug-in by creating a static instance of `IT_Bus::BusORBPlugIn` type. The constructor registers the plug-in under the specified name, `stub_transport`.

Example 51: *Registering the Stub Transport Plug-In*

```
// C++
namespace IT_Bus {
    ...
    const char* const und_stub_transport_plugin_name =
   "stub_transport";

    StubTransportBusPlugInFactory
   und_stub_transport_plugin_factory;

    IT_Bus::BusORBPlugIn und_stub_transport_plugin(
        und_stub_transport_plugin_name,
        und_stub_transport_plugin_factory
    );
}
```

Registering the stub transport factory with the Bus

Example 52 shows how to register the stub transport factory with the Bus.

Example 52: *Registering the Stub Transport Factory*

```
// C++
void
StubTransportBusPlugIn::bus_init(
) IT_THROW_DECL((IT_Bus::Exception))
{
    IT_Bus::Bus_ptr bus = get_bus();
    assert(bus != 0);

    m_transport_factory.initialize(bus);
    bus->get_pdk_bus()->register_transport_factory(
        "http://schemas.iona.com/transports/stub",
        &m_transport_factory
    );
}
```

Example 52: *Registering the Stub Transport Factory*

```
void
StubTransportBusPlugIn::bus_shutdown(
) IT_THROW_DECL((IT_Bus::Exception))
{
    IT_Bus::Bus_ptr bus = get_bus();
    assert(bus != 0);

    bus->get_pdk_bus()->deregister_transport_factory(
        "http://schemas.iona.com/transports/stub"
    );
}
```

To register the transport factory, perform the following steps:

1. Call the `IT_Bus::TransportFactory::initialize()` function to initialize the transport factory.
2. Call the `IT_Bus::PDKBus::register_transport_factory()` factory to register the transport factory.

Configuring the stub transport plug-in

To configure an application to use the stub transport plug-in, you must add the plug-in name, `stub_transport`, to the `orb_plugins` list, as follows:

Example 53: *Configuring the Stub Transport Plug-In*

```
# Artix Configuration File

ApplicationScope {
    orb_plugins = [ ..., "stub_transport"];
    ...
};
```

Artix Logging Reference

This chapter explains how to use Artix TRACE macros, and explains the Artix logging APIs.

In this chapter

This chapter includes the following sections:

Using Artix TRACE Macros

Overview

This section describes how to use TRACE macros in your own code in order to send logging messages to the Artix event log. The output from this Artix logging mechanism can then be controlled using the configuration settings described in *Deploying and Managing Artix Solutions*.

This section describes the following aspects of using Artix TRACE macros:

- Header file.
- Initializing the Bus logger.
- Artix subsystem scope.
- Artix trace levels.
- Passing in arguments.
- Creating your own output.

Header file

To use the Artix TRACE macros, you must include the `it_bus/bus_logger.h` header as follows:

```
#include <it_bus/bus_logger.h>
```

Note: In versions prior to Artix 3.0.2, the `it_bus/logging_support.h` header was used instead. This header is now deprecated, but it can be used to support legacy logging code.

Initializing the Bus logger

In order to control logging independently for each Bus, it is necessary to initialize a Bus logger object and associate it with a particular Bus instance. The Bus logger must be initialized before you can perform any tracing.

The normal way to initialize a Bus logger instance is to define it as a member of the class you happen to be implementing. For example, you can define and initialize a Bus logger instance in a class, `MyClass`, as follows:

1. Declare a `BusLogger` pointer by inserting the `IT_DECLARE_BUS_LOGGER_MEM` macro as a protected member in the class header file:

```
// C++
class myClass {
    ...
  protected:
    IT_DECLARE_BUS_LOGGER_MEM
};
```

2. In the class constructor, call the `IT_INIT_BUS_LOGGER_MEM` macro to initialize the `BusLogger` instance, passing a valid Bus instance to the macro argument:

```
// C++
myClass::myClass(IT_Bus::Bus_ptr bus) : m_bus(bus)
{
   IT_INIT_BUS_LOGGER_MEM(m_bus)
}
```

3. In the class destructor, call the `IT_FINALISE_BUS_LOGGER_MEM` macro to clean up the `BusLogger` instance.

```
// C++
myClass::~myClass()
{
  IT_FINALISE_BUS_LOGGER_MEM(m_bus)
}
```

   The Bus pointer passed to the macro in the destructor must be the same as the one passed to the macro in the constructor.

Artix subsystem scope

Artix uses a hierarchy of subsystem scopes that enables you to filter the messages that go into the event log. Artix uses several different subsystem scopes internally, for example:

```
IT_BUS.CORE
IT_BUS.TRANSPORT.HTTP
IT_BUS.BINDING.SOAP
IT_BUS.BINDING.CORBA
IT_BUS.BINDING.CORBA.RUNTIME
```

You can then define an event log filter in the Artix configuration file to control the level of logging from each of the subsystems. For example:

```
# Artix Configuration File
event_log:filters=["IT_BUS=FATAL+ERROR",
                   "IT_BUS.BINDING.CORBA=WARN+FATAL+ERROR"];
```

The default subsystem scope for any TRACE macros in your code is `IT_BUS`. Instead of using the default, however, it is better to specify a subsystem scope explicitly by defining the `_IT_SUBSYSTEM_SCOPE` macro in your code.

For example, if you are generating logging messages from a custom transport, you could define the subsystem scope as follows:

```
// C++
// Class implementation file.

// Header files:
#include <it_bus/bus_logger.h>
...

// Define the _IT_SUBSYSTEM_SCOPE *after* including the headers.
#define _IT_SUBSYSTEM_SCOPE IT_BUS.TRANSPORT
```

You can define the subsystem scope to be any identifier consisting of alphanumerics and the . character. The . character is used as a delimiter to separate the subsystem levels.

Artix trace levels

When the event log filter and log stream are properly configured, the Artix logging output from the TRACE macros is sent to the event log.

When using TRACE macros, the most important concept is the trace level, which is an `enum` that lets you filter events. Trace levels are defined in the *InstallDir*/artix/*Version*/include/it_bus/logging_support.h file:

```
const IT_TraceLevel IT_TRACE_FATAL = 64;            //FATAL
const IT_TraceLevel IT_TRACE_ERROR = 32;            //ERROR
const IT_TraceLevel IT_TRACE_WARNING = 16;          //WARNING
const IT_TraceLevel IT_TRACE = 4;                   //INFO_HIGH
const IT_TraceLevel IT_TRACE_BUFFER = 2;            //INFO_MED
const IT_TraceLevel IT_TRACE_METHODS = 1;           //INFO_LOW
const IT_TraceLevel IT_TRACE_METHODS_INTERNAL = 1; //INFO_LOW
```

The simplest trace statement emits a constant string at level `IT_TRACE`. For example:

```
TRACELOG("Hello world");
```

Passing in arguments

Several versions of the macro allow using a C `printf` format string, and passing in some arguments. Because you cannot have variable argument lists for macros, there are several defined according to how many arguments are allowed:

```
TRACELOG1("My name is: %s", "Slim Shady");
TRACELOG2("At state number %d, this happened: %s", 44, "connection failure");
```

Both the zero argument and the multiple argument versions have a setting that allows a trace level to be passed in, instead of level `IT_TRACE`. For example:

```
TRACELOG_WITH_LEVEL(IT_METHODS, "MyClass::MyClass()");
TRACELOG_WITH_LEVEL1(IT_TRACE_METHODS_INTERNAL, "Value of my_name_field was %s", my_name_field);
```

Creating your own output

If you need to create your own output using `iostreams` or another expensive process that is not supported by the macro, use the trace guard block. This ensures that the trace level test prevents your trace creation code from running when it does not produce output. For example:

```
BEGIN_TRACE(IT_TRACE)
        String trace_message = "data elements: ";
        for(i = 0; i < data_count; i++)
        {
                trace_message = trace_message + data_item[i] + "
   ";
        }
        TRACELOG(trace_message.c_str());
END_TRACE
```

To create binary output (for instance, a hex dump of the buffer), use `TRACELOGBUFFER`. For example:

```
TRACELOGBUFFER(vvMQMessageData, vvMQMessageData.GetSize())
```

If the trace statement issues at a level less than or equal to the process trace level, the entry is written to disk. The default log file name is `it_bus.log`.

Index