# WebServices - Axis

**1. Client-Side Axis**

## 1.1. Table of Contents



## 1.2. Introduction

This document looks at the issues related to developing the client side of a Web Service using Axis.

Axis supports SOAP, which is built on top of HTTP, a protocol built on TCP/IP. To understand what is going on, it is important to understand the levels underneath.

## 1.3. Core TCP/IP Concepts

We are not going to explain TCP/IP in any detail, as it is far too complex. Some of the concepts and features of the technology are worth covering.

TCP/IP builds a reliable channel between two computers, hosts. Every computer running TCP can receive messages coming in on any port, from 1-65535. That is, if a program on that machine has created a socket and is listening on that port. If not, you will see the message

connection refused.

Before a client connects to a host, it has to find its address. IPv4, the most widely supported version of TCP/IP uses 32 bit addresses, such as 127.0.0.1 (that's a special address, it means the local system). To connect to a host you need either the address, or a name of machine "www.w3.org" that can get mapped to an address. That mapping is provided by DNS, a hierarchical network that is so ubiquitous across the infrastructure that we usually take it for granted. Essentially, DNS servers take a hostname and return an IP address or an error message. How DNS servers work out the address is beyond the scope of this document; just assume that a local DNS server asks other ones if it thinks it needs to.

Not all systems have DNS support. A system can be configured to have its own host table; on Unix systems this lives in /etc/hosts, on windows in c:\windows\system32\drivers\etc\hosts. You can edit this file and bypass DNS completely, creating a maintenance nightmare in the process. If your users use static host tables rather than DNS, you can never change the network address of a host without serious grief.

Machines either have static addresses -the network managers assign them an address and they retain it over time, or dynamic addresses. The latter is common for client systems, especially laptops, desktop PCs and dial-up computers. Broadband computers often have static addresses, though it depends on the ISP.

After looking up the address, the client program creates a socket and tries to connect to the server. At this point the TCP protocol kicks in, the client initiating a conversation by sending a datagram to the destination. This commences the setup of the link, which takes a few packets (three). Once the connection is up and running, the server will be told of the caller (they can get its IP address), and they have a socket which is bound to the client for the duration of the connection. The client and server can send arbitrary binary data between each other, with the guarantees that (a) if the data arrives, it arrives in the order it was written to the socket, and (b) if it doesn't arrive, you will get an error message.

Some special things to know:

1. TCP does not (by default) send messages over an open connection to probe for a functioning link. You can turn this on by asking for keepalive packets on a connection, but it is wasteful of bandwidth.
2. TCP connections run with Nagle's algorithm enabled by default. This is a cooperative way of limiting bandwidth consumption, by having the sender only send at a rate the recipient can handle. The algorithm adapts, but it does ramp up slowly. Set TCP_NODELAY on a socket if you do not want it.

When communicating over a network, latency and bandwidth are the big constraints. Latency is how long it takes to communicate -all those bits of hardware like routers and firewalls add

to latency, as does the network wiring. Bandwidth is the measure of how much stuff you can send per second. A link with low latency (good) may still have low bandwidth (bad), while a high bandwidth connection might have a high latency, as the remote server is distant.

Firewalls are a critical feature of modern networks. A firewall blocks off ports to incoming calls. A stateful firewall examines every packet and only allows packets of the current TCP conversation in, and is even better. Firewalls are essential for security reasons, so that you can expose more services behind a network than to the outside world. As well as restricting incoming calls, they often restrict outbound connections. For example, outbound connections to port 80 (HTTP) may be blocked to encourage use of an HTTP proxy server (see below), outbound connections to port 143 (IMAP) blocked to stop people connecting to external mail servers. You need to assume that there will be firewalls in the network, between SOAP client and server, and so callbacks from server to client are not going to be possible.

## 1.4. Core HTTP Concepts

HTTP is at its heart, a very simple protocol. The client opens a TCP connection to a port on the remote system, usually port number 80. It then issues an HTTP request – a verb such as GET, POST, PUT or other standard request, a URL relative to the server and an HTTP version string. The client then sends zero or more headers – name:value pairs on individual lines, and then a blank line marking the end of headers. Those requests which involve data upload -such as POST- then continue with the upload of the data. Then the client waits for the server to respond.

The server can respond with an HTTP error code: a number, some headers of its own, and then usually the body of the request. Usually the headers include a MIME type declaration, and some others that are useful, such as "content-length" and "expires".

To get through firewalls, many organisations run a proxy server. This is a machine that has access to the outside network, while the rest of the Intranet does not. The clients must send a request to a proxy server, which then forward the request to the real server. A caching proxy server may cache request/response pairs, so popular requests do not consume bandwidth. This is very useful, but only works on requests that can be cached -historically only GET requests. Transparent Proxies are a special form of proxy -one that by virtue of the underlying network configuration route HTTP requests (especially those on port 80) through a proxy server -without any application configuration. Usually these are invisible until they go wrong, at which point your favourite web service appears as if is returning HTML destined for humans. This can lead to some interesting support calls.

SOAP over HTTP works above this underlying protocol. A SOAP request is a POST with an XML body; the response is an HTTP status code and ideally an XML message. As with normal HTTP, the 200 status code means all is well. The error code 500 may mean an

internal server error, or it can indicate that the SOAP stack and/or service threw a SOAPFault. A SOAPFault is a standardised XML message that contains information the recipients can parse. Although other HTTP response codes may be sent back in some circumstances, the WS-I organisation sets down the rules as to when and how these are allowed.

Because SOAP usually runs on top of HTTP, all the classic HTTP techniques for authentication and session management (i.e. cookies) apply. Note that at some point in the future alternate transports may become more popular, in which case the HTTP techniques will cease to work. This is why so many people are writing SOAP-based replacements, usually built using SOAP headers. Axis has some prototypes of alternate transports in its codebase, though none are (yet) production ready.

## 1.5. JAX-RPC

The JAX-RPC specification is the base specification that client-side Axis is built upon. If you are writing a client, read it.

There are essentially two ways to use JAX-RPC to invoke a SOAP endpoint -a URL at a server that processes SOAP messages. First, you can use the javax.xml classes to build a SOAP call by hand, and invoke a remote server. This is ugly but gives you an idea of what is going on behind the scenes: an XML message is being built up that is then sent to the remote server, whose response is parsed and deconstructed. Client code written at this level should run against any JAX-RPC implementation.

The other way is to have Axis hide the details of the call and generate a wrapper class for a web service. This is done by taking the WSDL description of the service and generating Java classes that make the low level calls appropriate to building the SOAP requests for each operation, then post-processing the results into the declared return values. Axis also takes note of any URL of the service included in the WSDL and compiles this in to the classes. Thus the client will automatically bind to the URL that the WSDL talks about -which is often the URL of the (development) server that the WSDL was retrieved from.

This automatic generation of proxy classes is convenient, as it makes calling a remote Web Service look almost like calling a local object. However, it has some disadvantages that developers need to be aware of:

- These generated classes are only compatible with Axis. This is allowed by the JAX-RPC specification, which has a notion of compile time compatiblity but not run-time compatibility. If you want stub classes that work with Sun's or BEA's SOAP implementation, you would need to generate stub classes from the WSDL using their platform's tools. The stub classes should all have the same names and methods, so the rest

of the code should not change.
- The JAX-RPC standard defines the translation of the service's operation and parameter names into valid Java method and variable names -the result may not be what you expect.
- Compile time is too early to bind to a Web Service URL -you need to add some configuration or dynamic binding routine.
- Remote Web Services are not the same as local objects. Pretending that they are is going to lead you astray. In particular, a method call to a local object often takes a few microseconds, while a call to a remote service can take tens of seconds, and fail with an obscure network error in the process, leaving the caller unsure if the call was successful or not. Making blocking calls to a Web Service from a web service will lead to a very unhappy end user experience.
- You have a more complex build process, as you need the WSDL before compiling the client, which may involve deploying the service.

Based on personal experience, dynamically generating stub classes is very useful, as it simplifies client side code and helps the client source recognise when a service has changed is operations' signatures a way that is incompatible. If the parameters of an operation changes, the Java method's parameters change, and hence the application no longer builds.

However, it is absolutely critical to always remember: web services are not local objects. The proxy class may appear local, but the server can be a long way away, over a narrow connection.

Never make a blocking call to a Web Service from the GUI thread.

## 1.6. Testing

If you want to test an Axis service, Wsdl2Java can be told to create a stub JUnit test class, containing a test case for every single operation that the remote service implements. These stub test cases need to be filled in with valid test data, followed by the relevant assertions to validate the results.

The generated test cases can then be run from the IDE or from an Ant- or Maven-based build process.

When testing the client -or the test cases- experiment with the special failure modes that distributed applications can experience. Unplug the network connector at different points in the program. Set the service up to connect to an invalid URL on the same host, or to a host that doesn't exist. Try going through a proxy server. Try using a slow connection -the TCP Monitor program can simulate this for you.

There are also third party applications that help you test Web Services -by providing SOAP monitors and by giving you forms-based construction of SOAP requests. These are

convenient to have, though you do have to pay for them.

Anteater, on sourceforge, is an Ant-based way of testing SOAP calls. You provide the payloads and then use xpath paths to verify the results. This may seem somewhat low-level but, it is very powerful.

## 1.7. Configuring Client-side handlers

TODO

Axis supports both client side JAX-RPC and Axis handlers. These handlers get called before a message is sent, and after it is received, just as for server-side handlers.

## 1.8. Redistribution

To redistribute an application running Axis, you need to redistribute

- axis.jar
- commons-logging.jar
- A logging implementation compatible with commons-logging. As Java1.4's intrinsic logging facility is compatible, you do not need to include a logging JAR for Java1.4. Otherwise the log4j.jar is a good one to use
- A logging configuration file for your chosen logger.
- An XML Parser. Java1.4 ships with crimson, although the axis team strongly recommend xerces over crimson.
- commons-discovery.jar

The Axis JAR is not signed, and so can not be used for auto-download from the Web Start facility in Java.

You do not currently need to include wsdl4j.jar, as the wsdl is not processed at run time. Note that this may change at some point in the future, as more knowledge about the structure of the SOAP message is needed to support doc/lit messages, which means limited runtime processing of WSDL files, or other metadata generated from the WSDL files during compilation.

## 1.9. Dynamically Discovering and Binding to a Web Service

When Axis generates client proxy classes code from WSDL, it binds the code to the endpoint URL specified in the WSDL -this is usually a URL generated from the URL of the inbound request. Using a http://localhost URL to fetch a WSDL page will result in client code also bound to a service served up on the localhost, which is not what you want in a redistributable. Similarly, even if you use the hostname when fetching the WSDL, you need

the fully qualified domain name, not any short name -http://s1.example.org/ and not http://s1/ -otherwise only callers in your own domain or subnet will be able to find the server. Hand-written WSDL does not exhibit this problem; the endpoint in the WSDL is the one the author typed in.

It is almost essential that you provide some way to update the URL on the clients. The simplest is some command line override option, as used in the Axis command line tools. More advanced is a dialog box for entering URLs, and more advanced yet is some automated discovery mechanism.

Axis does not provide any discovery mechanism in the JAR. There is a sibling project, jUDDI, that provides access to UDDI registries. There is also a multicast discovery jar that works with Axis in the Axis CVS tree; this is a proof-of-concept mechanism that uses XML messages but is not compatible with any existing standard. It works OK over LAN networks, but is not designed to be used over wider area.

TODO: how to set the URL in a service

## 1.10. Call configuration

The Call object can be configured before a call can be made. The `org.apache.axis.client.Call` is Axis's implementation of the `javax.xml.rpc.Call` interface. The JAX-RPC standard interface defines a `setProperty()` method that lets the caller set properties; there are both JAX-RPC standard properties and Axis's own properties that you can set.

All properties have a string name, a name defined in a public static final declaration in the class.

### 1.10.1. Standard Properties

### 1.10.2. Axis Properties

You can still set these Axis-specific properties in a portable client -other JAX-RPC implementations will not act on the options, of course.

## 1.11. Network configuration

Axis runs in a JVM, and JVM parameters control the client's behaviour. Here are JVM configuration options that are used.

Most of these options control proxy server settings; if they are missing and a proxy server is

needed for internet access -the client will get connectivity errors of some form or another. If they are present and wrong, the client will also get connectivity errors. Note that even though users can configure the proxy settings for the JVM when hosting Applets, these settings do not propagate to applications -this is one of the many mysteries of Java networking.

Every Web Service client application needs to provide some way to configure the proxy server settings. It is also useful to display these somewhere for support call diagnostics.

The final two properties are troublesome, all the more so because they are so little known about. To find out about them, look at "Address Caching" under java.net.InetAddress. Or just switch your DNS server off for a few minutes and observe how client applications not only fail to connect when the server is missing -servers stay unreachable when the DNS server is turned back on.

What is happening is that the runtime caches the IP addresses of hostnames it resolves using a DNS query. By default, these are cached forever, so that a long running Java application will break if the IP address of the remote server ever changed during the life of the client. Similarly, the runtime caches those hostnames that do not resolve to addresses. On Java1.3, these failed lookups are cached indefinitely -if DNS is down or a laptop off the net, the client will never be able to find them again.

Clearly it is essential for any non-trivial application to set the cache options to give a sensible lifetime for cached hostnames. These values are (in Java1.4) Java Security Properties; you set them using java.security.Security.setProperty(). In Java1.3 and earlier there was some alternate mechanism that mandated properties that could only be done on the command line. We would tell you what the properties are, except we have forgotten.

## 1.12. Troubleshooting Network Problems

The classic definition of a distributed system:

“You know you have one when the crash of a computer you’ve never heard of stops you from getting any work done.” Leslie Lamport

This may seem funny, but it is a depressingly accurate model of the state of distributed systems. Everyone knows that web sites are sometimes off-line, pages sometimes get served up incomplete or with some error trace instead of the results.

Web Services are similar, except instead of a human reading a web-browser-displayed error page, the client software receives the error and has to handle it or report it.

When the Axis client code receives an error, it throws an exception, specifically a subclass of java.rmi.RemoteException. This can be an AxisFault, or it can be something else. Either

way, it means trouble. Usually the fault string of the exception provides some error text which is meaningful to the experienced application developer -though less meaningful to either the end user or the support team.

Here is a list of network-related error responses that can be received by a client application. As Axis' adminclient application is a SOAP client, it can see these responses too. The Sitefinder comments are specific only if VeriSign SiteFinder or a successor is subverting the normal behaviour of DNS for their own goals, an action which complicates the normal failure modes of web services.

| | |
|---|---|
| Connection refused | The host exists, nothing is listening for connections on that port. Alternatively, a firewall is blocking that port.<br>Site Finder: the URL is using a port other than 80, and the .com or .net address is invalid |
| Unknown host | The hostname component of the URL is invalid, or the client is off-line. |
| 404: Not Found | There is a web server there, but nothing at the exact URL. Proxy servers can also generate 404 pages for unknown hosts. |
| 302: Moved | The content at the end of the URL has moved, and the client application does not follow the links.<br>Site Finder: the .com or .net address is invalid, the port is explicitly -or defaulting to- port 80 |
| Other 3xx response | The content at the end of the URL has moved, and the client application does not follow the links. |
| Wrong content type/MIME type | The URL may be incorrect, or the server application is not returning XML.<br>Site Finder: a 302 response is being returned as the host is unknown |
| XML parser error | This can be caused when the content is not XML, but the client application assumes it is.<br>Site Finder: this may be the body of a 302 response due to an unknown host, the client application should check return codes and the Content-Type header |
| 500: Internal Error | SOAP uses this as a cue that a SOAPFault has been |

| | |
|---|---|
| | returned, but it can also mean 'the server is not working through some internal fault' |
| Connection Timed out/ NoRouteToHost | The hostname can be resolved, but not reached. Either the host is missing (potentially a transient fault), or network/firewall issues are preventing access. The client may need to be configured for its proxy server. This can also crop up if the caller is completely off-line. |
| GUI hangs/ long pauses | Client application may be timing out on lookups/connects |

The support line's initial response to such messages should all be the same:

When a connectivity problem is suspected, get the URL that is at fault; the caller to view it in their web browser and see if you can view it yourself.

This is where you can take advantage of the fact that Web Service protocols -REST, XML-RPC and SOAP, are all built on top of HTTP, and use the common underlying notion of URLs defining services. Provided those same URLs generate some human-readable content -even if that is an XML message- then the end user and support contact can both bring it up in their web browser. This action is the core technique for diagnosing connectivity problems, primarily because the HTTP infrastructure -servers, proxies and clients- is designed to support this diagnosis process.

Web Service providers can simplify the process by:

- Having human readable content at every URL used in the Service. Specifically, you should support GET requests, even if it is only to return a message such as "There is a SOAP endpoint here".
- Using URLs that are human readable -short and describable over the telephone being the ideal.
- Having support-accessible logging to provide an escalation path should the problem turn out to be server side.

Another useful technique is for the service to implement the "Ping" design pattern. The service needs to support a simple "ping" operation, that immediately returns. This operation can be used by clients to probe for the presence of the service, without any other side effects or even placing much load on the server. Client applications should initiate communications with a server -uploads, complex requests, etc- by pinging it first. This detects failure early on, hopefully at a lower cost.

### 1.12.1. What can the developer of a Web Service client application do?

Networks are fundamentally unreliable; laptops move around and go offline, services get switched off.

Your application need to handle the connectivity problems and fail in a way that allows the problem to be diagnosed and corrected. Axis does not do this by itself, you need to help it.

1. It is good to translate framework errors/exceptions into error messages that are comprehensible by end users. XML parser errors, HTTP error codes and complaints about MIME types are not suitable for average end users, though the support organization may need these.
2. The target URL that failed needs to be disclosed to the end user, so that they can test it by hand.
3. For any error, the response body needs to be preserved for the benefit of support.
4. The fault diagnosis matrix listed above needs to be adapted to the client, and included in the documentation.
5. If the service implements a Ping operation, use it to probe for service existence, preferably in a background thread or asynchronous call, so that the GUI does not block.
6. Clients need to be tested over slow and unreliable networks. The Axis tcpmon SOAP monitor/HTTP proxy can be used to simulate slow HTTP connections.
7. Always verify that the MIME type of received content is exactly that documented.
8. Test the client's handling of HTTP response codes, and of HTML responses when XML is expected.
9. Look in the Java documents at "Address Caching" under java.io.InetAddress. Applications need to be configured to only cache DNS lookups, successful and unsuccessful, for a short period of time.