Fear of Attachments

Steve Loughran
2003-02-03

1.1 Introduction: SOAP and its attachments

One issue which has shown up with SOAP from the outset is "how do you send binary data" in a SOAP message. There are plenty of reasons for this. You may want to send binary data. You may want to send lots of binary data. Or you may want to send XML in a different encoding or schema/DTD from the SOAP message, and don't want to get into all the problems of nested XML.

The simplest solution as (and is) Base-64 encoded data. Encode the data, string it in a string and send it over the wire inside the SOAP message. This works across all possible transports, is known to interoperate well across all implementations, and is pretty much guaranteed to work with small amounts data. Also, it's very easy to code, provided the underlying library does the encoding and decoding for you.

Yet base-64 has its limits. Its inefficient, using only 6 bits of each byte, it adds about a third to the length of a message, plus perhaps a bit of padding to set the length. Note that there is a proposed base-85 encoding which uses less space, but is not integrated with any SOAP libraries. When sending large files, the issue becomes not just the expansion of the payload, but the actual process for encoding it and sending it over the wire. A 2KB file is easy to encode and ship; a 3MB file consumes memory while it is encoded and takes 4MB worth of uplink channel. It then causes no end of havoc at the far end, depending upon the XML parser. XML parsers are not usually engineered to have strings that large inside them. A 200MB file, or a 1GB file doesn't bear thinking about.

Clearly it is better to send large binary files in different ways. One getout clause is to use an intermediary, a file store or a web server, and send a URL to the content. This has some advantages. You can upload the files using an efficient protocol such as FTP, maybe even serve them directly from your file system, and the recipient can fetch the file at its leisure. If the server queues requests for processing, not having to retrieve the binary files until processing begins can be beneficial, and if the server implements caching, repeated requests using the same files can avoid repeated retransmissions of the data.

Before rushing to implement such a system, consider the drawbacks. You need to decide how long to hold content that is being served, and address firewall and security issues. To clean up files you effectively need a state machine, not deleting files till you know the message is processed, or have a housekeeping system to delete files after a certain time. Firewalls can only be addressed by using servers visible to all involved parties. Finally, the server side processing can be complex: you need to add retrieval and potentially caching to the server. That is simply too much work unless you need to add something fancy such as chunked file upload over multiple requests.

1.2 The relevant standards

1.2.1 SOAP

1.2.2 SwA: SOAP with Attachments

This is the first attachment proposal, a joint effort of Microsoft and John Barton of HP Labs. It extends the existing MIME standard to cover sending a SOAP message as the first attachment in a set of attachments sent to a Web Service endpoint.

1.2.3 DIME

This is the Microsoft format. They think it is better than SwA, and to make sure you use it, it is the only format that .NET framework can currently support –even then .NET1.0 does not support it out the box.

1.2.4 JAX-RPC and SAAJ

1.2.5 WS-I Basic Standard

This specification says 'use DIME'. No discussion, not explanation of the arguments, just 'use DIME', pure and simple. This means that basic JAX-RPC implementations are automatically outside the standard.

1.3 Apache Axis

Axis supports both Soap with Attachment (SwA) and DIME messages. When sending a message you need to decide which transport to use; to receive it you need do nothing. An Axis endpoint can handle both SwA and DIME endpoints.

1.3.1 Requirements

To send and receive attachments in Axis, you need two files, `activation.jar` and `mail.jar`, along with the core Axis libraries. These jars give Java MIME support, and are built in to J2EE, but not any of the current Java client runtimes. If these files are missing, Axis works, but attachments do not.

1.3.2 Client

Here is a fragment of a class to send a Mime or DIME request to a server.

```
public void submitJob(String name, File file, boolean dime)
             throws RemoteException {
    if (!isBound()) {
        throw new AxisFault("Not connected");
    }
    ZooBindingStub server = null;
    try {
        server = (ZooBindingStub) locator.getzooPort(_endpointURL);
    } catch (ServiceException e) {
        throw AxisFault.makeFault(e);
```

```
        }

        //turn dime on
        if (dime) {
            server._setProperty(Call.ATTACHMENT_ENCAPSULATION_FORMAT,
                    Call.ATTACHMENT_ENCAPSULATION_FORMAT_DIME);
        }
        //add our attachments
        DataHandler buildFile =
                 new DataHandler(new FileDataSource(file));
        server.addAttachment(buildFile);

        StringArray filenames = new StringArray();
        filenames.setElt(new String[]{file.getName()});
        server.submitSimpleJob(name, filenames);
}
```

Notice how the client is not only sending the file; it is adding the name of the file to a string array parameter in the request. This is because the server wants to know the extension, and there is no other way to do it. In Mime mode, we could add a Mime header, but not when the dime switch is set

What we are not showing here is how long it takes to upload large files. What may take a few seconds on a 100 MBit LAN against an unloaded development system will take much longer over a slower, busier network against a busy production machine. The tcpmon monitoring application in Axis1.1 can simulate delays to demonstrate the problems that slow links have. To address the delays, you need to be doing the upload outside of the main GUI thread, in a Swing app, as otherwise the app will lock up for an extended period.

1.3.3 Server

Server side is complex, depending on how you want to get at the attachments. The simplest approach is to not add attachments to the method signature, and just pull it out by hand. This means that any auto-generated WSDL lacks any information about the attachments: you need to document this in the text that accompanies the WSDL. Moreover, you need to test for those attachments later.

Manual attachment extraction

To extract the messages manually, you just get the attachment parts from the current request, then make sure that everything is valid, which means the number of attachments is in the valid range.

```
   public void submitSimpleJob(String jobName,
             StringArray filenames) throws RemoteException {
        log.info("SOAP::submitSimpleJob " jobName);
        AttachmentPart[] attachments = getMessageAttachments();
        if (log.isDebugEnabled()) {
            log.debug("received message with "
                 + attachments.length + " attachments");
        }
        if (attachments.length == 0) {
            throw new AxisFault("No job submitted with request");
        }
```

```
        if (attachments.length > 1) {
            throw new AxisFault("too many attachments! "+
               "One only please");
        }
        // validate everything
        if (filenames.getElt().length != attachments.length) {
            throw new AxisFault("Every attachment needs a matching "
                   +"filename entry, and vice versa");
        }
      //now work with the attachments....
}
```

The secret to making this work is that the helper method, `getMessageAttachments()`, that extracts an array of attachment entries from the current message.

```
    /**
     * extract attachments from the current request
     * @return a list of attachmentparts or
     * an empty array for no attachments support in this axis
     * buid/runtime
     */
    private AttachmentPart[] getMessageAttachments() throws AxisFault {
        MessageContext msgContext = MessageContext.getCurrentContext();
        Message reqMsg = msgContext.getRequestMessage();
        Attachments messageAttachments = reqMsg.getAttachmentsImpl();
        if (null == messageAttachments) {
            log.debug("no attachment support");
            return new AttachmentPart[0];
        }
        int attachmentCount= messageAttachments.getAttachmentCount();
        AttachmentPart attachments[] =
            new AttachmentPart[attachmentCount];
        Iterator it = messageAttachments.getAttachments().iterator();
        int count = 0;
        while (it.hasNext()) {
            AttachmentPart part = (AttachmentPart) it.next();
            attachments[count++] = part;
        }
        return attachments;
    }
```

Many people get confused by the fact that `MessageContext.getCurrentContext()` is a static method, and hence Axis cannot support more than one message at a time. To these people I say firstly, look up `ThreadLocal` in the JDK documentation, and secondly, if ever you have questions about Axis, look at the source.

This helper method always returns an empty array if there is no attachment support. We could throw an exception, or log the state at a higher level of importance. The latter makes sense if the system should always have attachments installed, and missing support means that something has gone wrong with the installation. One would expect the post-installation unit tests to try and submit jobs and so fail anyway, but it is always better for operations to have a clear message 'You

need to add the attachment jars' rather than 'testCanSubmitJob is failing saying we didn't send any job with the request'. One they can deal with, the other they page the dev team for. Unless you like being woken up, write error messages that make sense to operations.

Getting at the contents

You can get at the attachment by asking for the input stream:

```
attachment.getInputStream()
```

Simple, eh?

File Management

Attachments are saved to memory if they are small (under 16KB or so), and then to disk if they get larger. Axis1.1 automatically cleans up these files when the attachments are finalized, but not before. Forcing a gc on the server sporadically helps keep disk usage down and stops leakage when the server crashes.

Axis 1.0 did no cleanup whatsoever, and there was no way of getting at the files to delete by hand, other than having a housecleaner thread purge the attachment directory of old files

Another nice feature of Axis1.1 is that you can ask for the filename (if it exists), and tell the `AttachmentPart` not to delete the file when it is finalized, by *detaching* it.

```
//may be null
String savedFile = attachmentPart.getAttachmentFile();
attachmentPart.detachAttachmentFile();
attachmentPart=null
```

Once you have moved a file by hand, the `AttachmentPart` should not be used again, because it is in an incomplete state. You cannot ask it for its contents as the file containing those contents has been moved.

Accessing the file contents is therefore an advanced use, but one useful when you start receiving big files: renaming a file is a lot faster than copying a 30MB file, even across the same local hard disk.