XML Watch: Worm's-eye BEEP

Part 2 of an introduction to the Blocks Extensible Exchange Protocol standard

Edd Dumbill (edd@xml.com)
Editor and publisher
xmlhack.com

01 March 2002

In this second article examining BEEP -- Blocks Extensible Exchange Protocol -- Edd builds on the broad principles of BEEP outlined in his previous article, explaining how the protocol is implemented, and providing an example of how it is used in Java.

View more content in this series

BEEP is a peer-to-peer protocol framework. It isn't as much a ready-made protocol in itself, like HTTP, but a framework on which such protocols can be built. It takes care of many of the features of such protocols so that they don't need to be reinvented. The main areas of functionality are:

- Separating one message from the next
- Encoding messages
- Multiple channels of communication over a single connection
- Reporting errors
- Negotiating encryption
- Negotiating authentication

In the previous article, I explained that communication in a BEEP session takes place over one or more channels, which are multiplexed over the transport protocol. (I will assume that, per RFC 3081, we're using BEEP over TCP/IP. This need not be the case: A mapping of BEEP could be made onto a different connection-oriented transport protocol.)

The first channel, channel 0, has a special role and is used for session management. All communication over a channel is defined by a profile, which is basically a description of permissible interactions. For XML-based protocols, a profile could be based around a DTD or schema that specifies the syntax for messages. (Obviously, more than just syntax specification is required to completely define the profile.)

When peers connect using BEEP, they request profiles of each other in order to structure their communication. Profiles fall into two categories: tuning and data exchange. Tuning profiles affect the whole session and are typically responsible for security and authentication. A data exchange profile, as indicated above, defines the exchanges on a particular channel.

From the worm's eye

Now that you have a high-level idea of what BEEP does, let's swoop down to the other extreme and investigate how the fundamentals of the protocol are implemented. It is not necessary to have a complete understanding of what goes on, but it is helpful in grasping what methods of exchange are available for a profile to use.

BEEP incorporates the concept of a *message* as a useful unit of communication. Such messages sent as part of an application protocol interaction might be "my CPU temperature is 80 degrees," or "here's a JPEG image." MIME is used for message envelopes, so messages can be of any content type. There is also no real limit on how big or small a message should be; it's an application-specific decision.

For this reason BEEP implements a unit of communication smaller than the message, and of which messages are then composed: the frame. While a message might often reasonably be sent all in one frame, it might need to be split up over a number of frames. A frame contains a header that identifies which channel and message it belongs to, and its sequence in the message. We'll leave the topic of frames for now, but keep in mind that a message might be composed of multiple frames.

In order to support its basic features as outlined above, BEEP provides three styles of interaction:

- **`MSG/RPY`**: The client sends a `MSG` message, typically requiring the server to perform a task. The server does this and sends a `RPY` message, indicating success and possibly conveying more information.
- **`MSG/ERR`**: This is similar to `XSG/RPY`, except that an error is encountered by the server. Instead of doing what the client asks, it returns an `ERR` message indicating failure and possibly describing the problem.
- **`MSG/ANS`**: The client sends a request `MSG`, to which the server replies with any number (including zero) of `ANS` messages. When the server has completed its task, it sends a `NUL` message to indicate that its reply is now complete.

An example interaction

To demonstrate these exchange types, we will consider an example of a highly simplified addressbook store. The simple operations we require are store, retrieve, and query. The peer that hosts the addressbook is indicated as `S:` and the client as `C:`. The actual XML used to define the protocol is of my own invention for the sake of the example, although inspiration is taken from my investigations of the Evolution e-mail client. Assume that the actual connection, tuning, and channel establishment have already taken place.

Example 1: Storing an entry

```
C: MSG <person>
         <name>Edd</name>
         <tel>1-234-567-7890</tel>
       </person>
S: RPY <person uid="edd_0"/>
```

The client sends a complete addressbook entry to the server. The server, seeing that there is no ID specified, assumes it is a new entry, stores it, and returns the ID of the new entry to the client.

Example 2: Retrieving an entry

```
C: MSG <person uid="joe_2"/>
S: RPY <person uid="joe_2">
         <name>Joe</name>
         <tel>1-634-222-2333</tel>
       </person>
```

The client wishes to retrieve full details for the entry with ID "joe_2". It sends as much as it knows to the server, which fills in the details and sends them back to the client.

Example 3: Querying

```
C: MSG <query>
         <match><tel>1-*</tel></match>
         <return><name /></return>
       </query>
S: ANS <person uid="edd_0">
         <name>Edd</name>
       </person>
S: ANS <person uid="joe_2">
         <name>Joe</name>
       </person>
C: NUL
```

The client wishes to find out the names of people who have telephone numbers in the US/Canada. It sends a query and the server starts searching its database -- an operation that could take some time. In order to give a timely response, it sends each entry back to the client as it finds it. When the query is done, it sends a `NUL`.

Example 4: An error occurs

```
C: MSG <person uid="joe_2"/>
S: ERR <error code="403">
          You are not allowed to access this information.
       </error>
```

The client wishes to retrieve an item, but the server has been programmed not to allow access to this item for this client, and returns an error condition.

Bidirectionality

One of BEEP's unique features is its bidirectionality. This can be illustrated in the context of our example above. For example, consider that the two machines communicating are not strictly client and server, but two servers. Our addressbook protocol could be used to enable them to keep their data in sync. In the above examples, imagine that M1 played the role of `S` throughout and M2 played that of `C`. M2 has connected to M1 and requested the addressbook protocol profile. If both machines supported our addressbook protocol profile, M1 could request a channel to M2 using the addressbook profile, and the roles would be reversed. The two connections could be open simultaneously: Each machine could query the other for a list of addressbook entries and retrieve the ones it doesn't have.

Writing some code

Happily, in order to use BEEP we don't need to start implementing the protocol stack from scratch. Over at beepcore.org (see Resources), there are implementations in Java, C, and Tcl, and there are implementations being developed for Python and Ruby. We'll use Java for our example.

First, let's look at how BEEP's concepts map to the Java BEEP core API.

- **`Session`**: One instance of this class is required per peer-to-peer connection. The session object is used to start any tuning profiles (such as encryption), and also used to start channels. A session that intends to support the server side of an interaction has an associated `ProfileRegistry`, which maps supported profile URIs to their corresponding profile "start channel" listeners (see the `Profile` description below).
- **`Channel`**: A channel object is returned as the result of calling a session's `startChannel()` method. It can send `MSG` messages via a `sendMSG()` method.
- **`Profile`**: The profile interface defines only one method that returns a listener for "start channel" events for a particular profile. All implementations of profiles implement this interface. The "start channel" listener receives a `Channel` object and registers a listener for messages (or frames) on the channel.
- **`Message`**: This class encapsulates the `MSG`, `RPY`, `ANS`, `ERR`, and `NULL` message types. It can be interrogated for its content, and has methods `sendRPY()`, `sendANS()`, `sendERR()`, and `sendNUL()`, which are used to respond to a message.
- **`Reply`**: A `Reply` object is used to handle the asynchronous replies from a peer. Its most useful method is `getNextReply()`, which returns a Message.

The classes are used differently depending on whether you are at the server or client side of an interaction. Servers register listeners for `MSG` messages on a channel, whereas clients will specify a `ReplyListener` (of which the `Reply` class is one) when sending a `MSG`. If your application is symmetric (i.e., both peers can initiate and listen for messages), then both peers will implement both message and reply listeners.

In order to show the code, we'll start to prototype the calendar protocol from the above examples.

The client side

We'll implement the majority of the client side in the `main()` method of a Java application. First, we'll establish a session with the server `calhost` running on port 6000:

```
import org.beepcore.beep.core.*;
import org.beepcore.lib.Reply;
import org.beepcore.transport.tcp.*;

   ...

Session session;
try {
    session =
       AutomatedTCPSessionCreator.initiate("calhost", 6000,
          new ProfileRegistry());
} catch (BEEPException e) {
   // error handling
}
```

We use one of the Java API's helper classes, `AutomatedTCPSessionCreator`, for creating the session. Note we pass an empty `ProfileRegistry:` -- this peer only intends to act the client role, so it makes sure not to advertise its support for any profiles. We now try to start the channel:

```
Channel channel;
try {
    channel = session.startChannel(
        "http://example.org/profiles/ADDRESSBOOK");
} catch (BEEPError e) {
  // catch and deal with errors based on
  // e.getCode()
} catch (BEEPException e) {
  // catch lower level errors
}
```

If everything has gone well, we now have a `channel` object that supports communication using the addressbook profile. We'll now send an addressbook entry for the server to store.

```
   String message = "<person><name>Edd</name>" +
       "<tel>1-234-567-7890</tel></person>;

   // for the sake of example, we've decided we'll
   // send all messages encoded in UTF-16
   // using org.beepcore.beep.core.ByteDataStream
   // we could use StringDataStream if we wanted UTF-8
   ByteDataStream msgst=new ByteDataStream(
      request.getByes("utf-16"));
   msgst.setContentType("text/xml");
   msgst.setTransferEncoding("utf-16");

   Reply reply = new Reply();
   channel.sendMSG(msgst, reply);
```

So far we've created and encoded the message, set up the MIME type of our message payload correctly, and sent the message to the server. We can now expect a reply from the server giving us the ID of the new entry we sent.

```
   Message repmsg = reply.getNextReply();
   DataStream ds = repmsg.getDataStream();
   InputStream is = ds.getInputStream();
   // the reply is a stream, not a string

   byte inputbuf[]=new byte[256];
   int inputlen;
   String reptxt;
   while (ds.isComplete() == false ||
           is.available() >0) {
       inputlen=is.read(inputbuf);
       if (inputlen>0) {
           reptxt=reptxt.concat(new String(inputbuf, 0,
               inputlen, "utf-16");
       }
   }
   System.out.println("got back:" +reptxt);
   Thread.currentThread.sleep(6000);
} while (true);
```

We make the assumption that nothing bad has happened, or if it did, we don't care. We then just dump back out to the console whatever we got back. The process of decoding the message may look a little complicated, but bear in mind that BEEP is a general purpose framework and makes no assumptions about the nature of the messages. You would have to do the same work of encoding if you were using HTTP, for instance. In practice, I've found myself writing utility classes to handle both the composition and decoding of messages for particular profiles.

The server side

We'll now skim through the essentials of implementing the server side of the interaction. In particular, we'll focus on the channel and message listeners. At the end of the article, I'll provide references to the beepcore-java project, where you can find a full example of setting up the server side.

```
public class AddressbookProfile implements
     Profile, StartChannelListener, MessageListener
{
    // we're rolled all the interfaces into one class
    // for convenience
    public StartChannelListener init(String uri,
        ProfileConfiguration config) throws BEEPException
    {
        return this;
    }

    public void startChannel(Channel channel, String encoding,
        String data) throws StartChannelException
    {
        // just register ourself as interested in incoming
        // messages
        channel.setDataListener(this);
    }

    public void closeChannel(Channel channel)
        throws CloseChannelException
    {
        // say we're not interested any more
        channel.setDataListener(null);
    }
```

So far, we've set up the housekeeping. For simplicity, we're only creating one instance of our message listener per session, so the same object gets used no matter how many channels are open.

```
    public void receiveMSG(Message message)
        throws BEEPError, AbortChannelException
    {
        DataStream ds=message.getDataStream();
        AddressbookEntry newentry;

        // we then do the same processing as we did
        // in the client example to get some data out
        // of the stream.  imagine that we then store
        // the new addressbook entry in some database
        // somewhere and fill the "newentry" with it

        // it's now time to reply with an "ok"
        String replytxt="<person uid=\"" +
            String.valueOf(newentry.getUID()) + \""/>";
        DataStream reply=new ByteDataStream(
            replytxt.getBytes("utf-16"));
        reply.setContentType("text/xml");
        try {
            message.sendRPY(reply);
        } catch (BEEPException e) {
            // handle the error
        }
    }
```

Again, I've omitted error handling, including the code to call `sendERR()` if an incorrect message was received, or some other error happened. As you can see, it's important to consider MIME types, character encodings, and transfer encodings. In many settings, you will not have control over the software the other peer is running, so use of Internet standards is important to ensure interoperability -- even if the only extra thing you do is refuse configurations that you don't support explicitly with an `ERR` message.

Digging further

The best place to start is by downloading the Java BEEP core implementation. The example included there will give you a start; it implements a simple "ping" protocol over BEEP, and demonstrates how to set up encryption on the connection.

From a practical point of view, I've found the Java BEEP core API to be just that, a core. Once you build up a few helper classes to handle the common work of encoding and decoding your profile payloads, the BEEP-specific code will become a "given" and you can concentrate on the real logic of your application -- which is the whole point of using BEEP in the first place.

Conclusion

BEEP offers the Internet protocols world, which includes the growing area of Web services, a plausible alternative to the continued overloading of HTTP. Its flexible nature and deep-rooted support for MIME means it should adapt well to connection-oriented protocol needs. The number of freely available implementations of BEEP is steadily increasing, as is the BEEP user community. Developers creating new application protocols should seriously consider using BEEP as the substrate for their work.

Resources

- beepcore.org is the home of the BEEP specifications and tools on the Web.
- The mapping of BEEP onto TCP is specified in RFC 3081.
- The author's previous article on BEEP gives a high-level overview of the protocol framework.
- beepcore-java is a Java implementation of RFC 3080 and 3081.
- Implementations of BEEP in Tcl and C are also available: beepcore-c and beepcore-tcl.
- Implementations for Python and Ruby are in progress.
- Rational Application Developer for WebSphere Software helps Java™ developers rapidly design, develop, assemble, test, profile and deploy high quality Java/J2EE, Portal, Web, Web services and SOA applications.
- Find other articles in Edd Dumbill's *XML Watch* column.

About the author

Edd Dumbill

Edd Dumbill is managing editor of XML.com and the editor and publisher of the XML developer news site XMLhack. He is co-author of O'Reilly's Programming Web Services with XML-RPC, and co-founder and adviser to the Pharmalicensing life sciences intellectual property exchange. Edd is also program chair of the XML Europe 2002 conference. You can contact Edd at edd@xml.com.