XML Watch: Bird's-eye BEEP

Part 1 of an introduction to the Blocks Extensible Exchange Protocol standard of the IETF

Edd Dumbill

01 December 2001

While debate continues on reusing HTTP as a convenient way to connect applications, a new protocol called BEEP -- Blocks Extensible Exchange Protocol -- has been standardized by the Internet Engineering Task Force (IETF). Making use of XML itself, BEEP does for Internet protocols what XML has done for documents and data. In his first column for developerWorks, seasoned XML observer Edd Dumbill explains how BEEP provides a framework that allows developers to focus on the important aspects of their applications rather than wasting time with the detail of establishing communication channels.

View more content in this series

Welcome to the first article in a series that will examine the practicalities of using new XML-based technologies. In these columns, I'll take a look at an XML technology, and at attempts to deploy it in a practical system. In addition to reporting on the deployment experience, I expect to have some fun along the way too. I won't expect too much prior knowledge from the reader, but a grounding in basic Web standards such as XML and HTTP will help.

Introducing BEEP

For starters, the next few columns will look at BEEP, one of the many acronyms lurking in the alphabet soup of Web services. The purpose of this article is to introduce the BEEP protocol framework and to suggest where it may be appropriately used.

BEEP stands for Blocks Extensible Exchange Protocol, an expansion which makes almost as little sense as just saying *BEEP*, and frankly is far less entertaining. Nevertheless, XML users will likely find themselves drawn to the word *extensible*, and indeed it's extensibility that makes BEEP worth looking at in the first place. More of that later; first let's look at the problem that BEEP solves.

You're writing a networked application, and you want instances of your programs to be able to communicate via TCP/IP. Before you can even get around to the logic of your application itself, you need to figure out how your programs are going to connect, authenticate themselves, send

messages, receive messages, and report errors. The cumulative time you'll spend on this may well outweigh the effort needed for the application logic itself. In a nutshell, this is the problem BEEP solves. It implements all the "hygiene factors" of creating a new network protocol so you don't have to worry about them.

At this point you may well be wondering, and not without justification, why we need another type of distributed computing protocol to add to CORBA/IIOP, SOAP, XML-RPC, and friends. In answer, you need to recognize that BEEP sits at a different level. It's a framework. SOAP can happily be implemented on top of BEEP. BEEP takes care of the connections, the authentication, and the packaging up at the TCP/IP level of the messages -- matters that SOAP deliberately avoids. BEEP really competes on the same level as HTTP.

Reuse and refactoring

Designers of recent application protocols have looked upon HTTP, and seen that it was good. Well, good enough. So WebDAV, the protocol underlying the "net folders" feature in Windows, added a few verbs to HTTP in order to allow distributed authoring. The Internet Printing Protocol invented some HTTP headers in order to use HTTP/1.1 to do its work, and the binding of SOAP to HTTP has done a similar thing (see Resources for background on these three uses of the HTTP protocol).

In principle, the right thing has been done. A ubiquitous and widely implemented protocol, HTTP, has been reused in an efficient way. There are some unfortunate consequences: the first of these is the resultant overloading of port 80. Since not just Web page requests, but potentially security-critical business requests are passing through port 80 now, increased vigilance is required. The many interactions with Web caches and other devices which affect port 80 must be taken into account. These issues have been rehearsed extensively elsewhere (see Resources), so I won't go into detail here.

The second consequence of reusing HTTP is that you're tied to using its model of interaction. HTTP is a stateless request-response-oriented protocol. There can be no requests without a response, and there can be no response without a request. Additionally, no state is preserved between requests. Unfortunately, this isn't good enough for many interaction schemes, as it precludes things like asynchrony, stateful interaction, and peer-to-peer communication. These problems can and have been circumvented by layering on top of HTTP, but most of these solutions feel awkward at best.

It is at this point that the seasoned programmer would tell you it's time to *refactor*, that is, to place the responsibilities of a system at their correct level and abstract out common functionality. This is the best way to look at BEEP: it is essentially a refactoring of an overloaded HTTP to support the common requirements of today's Internet application protocols.

So what can it do?

Enough scene setting, it's time to look at what BEEP can and cannot do, so you can get an idea of why you might want to use it.

In a presentation given by Marshall Rose (see Resources), the author of the BEEP specification, BEEP's "target market" of application is described in the following terms:

- **Connection-oriented**: Applications passing data using BEEP are expected to connect, do their business, and then disconnect. This gives rise to the characteristics of communication being ordered, reliable, and congestion sensitive. (Paralleling at the IP level shares many of the same characteristics of using TCP rather than UDP.)
- **Message-oriented**: Applications passing data using BEEP are expected to communicate using defined bundles of structured data. This means that the communicating applications are loosely coupled and don't require extensive knowledge of each others' interfaces.
- **Asynchronous**: Unlike HTTP, BEEP is not restricted to a particular ordering of requests and responses. Asynchronicity allows for peer-to-peer style communication, but it doesn't rule out conventional client/server communication either.

While these characteristics encompass a large number of potential applications (for instance, they would happily permit the re-implementation of HTTP, FTP, SMTP, and various instant-messaging protocols), a number of applications fall outside of BEEP's scope. These include one-shot exchanges such as DNS lookup, where the cost introduced by BEEP would be disproportionate, or tightly coupled RPC protocols like NFS.

Given that an application falls into the target market, what can BEEP offer? Its main areas of functionality are:

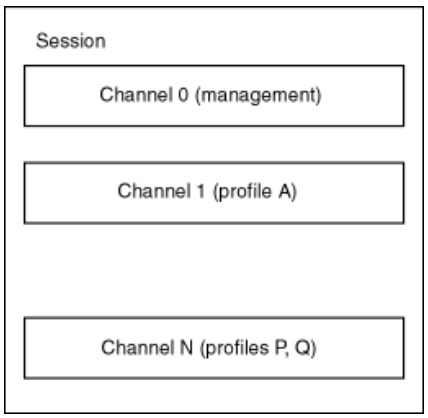
- Separating one message from the next (framing)
- Encoding of messages
- Allowing multiple asynchronous requests
- Reporting errors
- Negotiating encryption
- Negotiating authentication

The fact you don't have to worry about these things leaves you free to add the other ingredients to your networked application. You can start thinking about message types and structures, for instance.

BEEP concepts

BEEP is a peer-to-peer protocol, which means that it has no notion of client or server, unlike HTTP. However, as with arguments and romance, somebody has to make the first move. So for convenience I'll refer to the peer that starts a connection as the *initiator*, and the peer accepting the connection as the *listener*. When a connection is established between the two, a BEEP session is created.

Figure 1. BEEP sessions, channels, and profiles



Channels

All communication in a session happens within one or more *channels*, as illustrated in Figure 1. The peers require only one IP connection, which is then multiplexed to create channels. The nature of communication possible within that channel is determined by the *profiles* it supports (each channel may have one or more.)

The first channel, channel 0, has a special purpose. It supports the BEEP management profile, which is used to negotiate the setup of further channels. The supported profiles determine the precise interaction between the peers in a particular channel. Defining a protocol using BEEP comes down to the definition of profiles.

Two types of profiles

After the establishment of a session, the initiator asks to start a channel for the particular profile or set of profiles it wishes to use. If the listener supports the profile(s), the channel will be created. Profiles themselves take one of two forms: those for initial tuning, and those for data exchange.

Tuning profiles, set up at the start of communication, affect the rest of the session in some way. For instance, requesting the TLS profile ensures that channels are encrypted using Transport Layer Security. Other tuning profiles perform steps such as authentication.

Data-exchange profiles set expectations between the two peers as to what sort of exchanges will be allowed in a channel, similar to the way Java interfaces set expectations between interacting objects as to what methods are available. As with XML namespaces, a profile is identified by a URI. For instance, the example "Echo" profile from the BEEP Java tools has the URI http://xml.resource.org/profiles/NULL/ECHO.

Types of data

BEEP puts no limits on the kind of data a channel can carry. BEEP uses the MIME standard to support payloads of arbitrary type. This approach neatly sidesteps the sorts of issues raised by SOAP of how to send an XML document or a binary file inside a SOAP message.

The XML connection

At the beginning of this article I promised you that BEEP made use of XML, and by this point you'd be forgiven for wondering where. In fact, the BEEP management profile, responsible for channel initiation, is defined as an XML DTD (see Resources for a pointer to the management profile definition). This is why XML and BEEP fit so well together: as BEEP takes care of protocol infrastructure, XML takes care of data structuring. Hence XML is a natural choice in which to define the syntax of messages in BEEP profiles (although, as noted above, profiles can use any MIME type).

Aside from the channel-management profile, many emerging BEEP application profiles have used XML as an encoding for their messages. This is a boon, as it means that any existing messaging standard defined in terms of XML documents has a reasonably straightforward mapping into a BEEP profile.

Wrapping it up

In this article I've explained the rationale for using BEEP and outlined its target application areas. I've given a very high-level overview of how BEEP interactions take place. The next column will go into more detail on how communication is achieved through channels and profiles, with an example implementation in Java.