Blocks Extensible Exchange Protocol (BEEP)

Seminar:
Datenkommunikation und verteilte Systeme
Wintersemester 2003/2004

Sandra Cüsters
Matrikelnummer: 223195

Betreuung: Torsten Dinsing
Ericsson Eurolab Deutschland GmbH, Herzogenrath
1. Introduction

Within networked applications instances of the programs should be able to communicate via transport protocols (e.g. TCP/IP). Before getting around to the logic of the applications itself, it is to be defined how the programs are going to connect, authenticate themselves, send messages and report errors. The cumulative time spending on this may well outweigh the effort needed for the application logic itself. This is the problem BEEP (Blocks Extensible Exchange Protocol) tries to solve. It implements all the “hygiene factors” of creating a new network protocol.

But why do we need another type of distributed computing protocol to add to CORBA, SOAP or XML-RPC. The answer is: BEEP sits at a different level. It is a generic application protocol framework. For example SOAP is a protocol for accessing a Web Service and can easily 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. BEEP really competes on the same level as HTTP.

<table>
<thead>
<tr>
<th>SOAP</th>
<th>CORBA</th>
<th>XML.RPC</th>
<th>.......</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP</td>
<td>BEEP</td>
<td>SMTP</td>
<td>FTP</td>
</tr>
<tr>
<td>TCP/UDP</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 1: three different layers

At first an overview on the design of application protocols, their mechanisms and properties will be given. This work will then introduce the BEEP framework and its mapping for TCP. Taking the example of SOAP, it will be illustrated how SOAP can be bound to HTTP on the one hand and to BEEP and on the other hand. The work will finally compare these two scenarios.
2. The Design of Application Protocols

In 1998, one of the most prolific creators of internet protocols, Dr. Marshall T. Rose, realized that the design of new internet protocols had reached a critical point. Marshall Rose teamed up with Carl Malamud to invent a new application protocol. Creating an application protocol, there are four possible ways to proceed:

- find an existing exchange protocol that (more or less) does what you want;
- define an exchange model on top of the world-wide web infrastructure that (more or less) does what you want;
- define an exchange model on top of the electronic mail infrastructure that (more or less) does what you want; or,
- define a new protocol from scratch that does exactly what you want.

The most appealing option is to find an existing protocol and use that. So, M. Rose and C. Malamud did a survey of many existing application protocols and found that none of them were a good match for the semantics of the protocol they needed. They decided to define a new protocol called BEEP from scratch.

2.1 Protocol Mechanisms

After selecting one of the four possible ways, the next step is to look at the tasks that an application protocol must perform and how it works. The most important items within such an application protocol are:

- framing, which tells how the beginning and ending of each message is delimited;
- encoding, which tells how a message is represented when exchanged;
- reporting, which tells how errors are described;
- asynchrony, which tells how the exchanges are handled;
- authentication, which tells how the peers at each end of the connection are identified and verified against third-party interception or modification

2.2 Protocol Properties

Designing an application protocol, there are a few properties one should keep an eye on.

- **Scalability**: Because few application protocols support asynchrony, a common trick is for a program to open multiple simultaneous connections to a single destination. In theory this reduces latency and increases throughput. In reality the
transport layer as well as the server are viewing each connection as an independent instance of the application protocol, and this can cause problems. Another important aspect of scalability to consider is the relative numbers of clients and servers. Typically, there are many more client peers than server peers. In this case functional requirements should be shifted from the servers to the clients.

- **Efficiency**: For example, although octet-stuffing leads to more elegant implementations, it should be replaced though octet-counting. Experiences show that octet-counting consumes far fewer cycles. But recognise that you sometimes have to compromise efficiency in order to satisfy other properties.

- **Simplicity**: A well-designed protocol should be as simple as possible. Simple does not mean simple-minded. Something well designed focused all problems, even the smallest problems, in a domain. Doing this in a consistent fashion makes the design simple. Typically, this leads to an elegant design.

- **Extensibility**: The protocol should be evolutionary, and there must be a way for implementations reflecting different steps in the evolutionary path to negotiate which extension will be used.

- **Robustness**: Robustness and efficiency are often at odds. For example, although defaults are useful to reduce packet sizes and processing time, they tend to encourage implementation errors.

### 2.3 Pipelining

Few application protocols today allow independent exchanges over the same connection. In fact, the more widely implemented approach is to allow pipelining. With pipelining a client can make multiple requests to a server within a single connection, but requires the requests to be processed serially. It is a powerful method for reducing network latency. In addition to reducing network latency, asynchrony also reduces server latency by allowing multiple requests to be processed by multi-threaded implementations. Using this kind of parallelism, a flow control mechanisms is needed to avoid starvation and deadlocks. Flow control is typically implemented at the transport layer. For example, TCP uses sequence numbers and a sliding window. Using flow control a segmentation mechanism is needed to fragment messages into smaller pieces before sending and then reassemble them when there are received. Section 4 will give more information about this behaviour. But first BEEP, an application protocol framework that allows pipelining, will be introduced.
3. BEEP

BEEP – The Blocks Extensible Exchange Protocol – is a new specification that provides a common set of services to help designers of application protocols. In the past, when designers were faced with the need of a new application protocol, they had either to face the daunting task of designing a complete new protocol – which is a lot of work, or somehow piggyback on top of an existing protocol – and live with all the advantages and disadvantages of the existing protocol. BEEP is a new approach that recognises that many application protocols are trying to solve the same set of issues again and again.

Marshall Rose, the author of the BEEP specification, explains that the “target market” of applications 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.

- **Message-oriented**: Application passing data using BEEP are expected to communicate using defined bundles of structured data. This means that the communicating applications are loosely coupled and do not require extensive knowledge of each others’ interface.

- **Asynchronous**: Unlike HTTP, BEEP is not restricted to a particular ordering of requests and responses. Asynchrony peer-to-peer style communication is allowed between the client and the server.

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
3.1 BEEP Concepts

BEEP is a peer-to-peer protocol, which means that it has no notion of client or server, unlike HTTP. The classic client/server model is that the client sends a request and the server sends a response. If thinking of requests as questions and responses as answers, then the server answers only those questions that it is asked and it never asks any questions of its own. In the peer-to-peer model, for a given transaction one peer might be the client and the other the server, but for the next transaction, the two peer might switch roles. The peer that establishes the connection always acts as the client (initiates request), and the peer that listens for incoming connections always acts as the server (issuing responses to requests). When a connection is established between the two, a BEEP session is created.

3.1.1 BEEP Sessions

All communication in a session happens within one or more channels, as illustrated in figure 2. 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.

![Figure 2: structure of a BEEP session](image)

After a session has been established, the initiator (client) asks to start a channel for a particular profile or a set of profiles it wishes to use. If the listener (server) 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 (see Section 5).
3.1.2 Types of Interaction

A BEEP message is the complete set of data to be exchanged. BEEP defines three styles of message exchange, distinguished by the type of reply the server returns:

- **MSG/RPY**: The client sends a “MSG” (message) message, typically requiring the server to perform a task. The server does this and sends a “RPY” (reply) message, indicating success and possibly conveying more information.

- **MSG/ERR**: This is similar to MSG/RPY, except that the server encounters an error. Instead of doing what the client asks, it returns an “ERR” (error) 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” (answer) messages. When the server has completed its task, it sends a “NUL” (terminator) message to indicate that its reply is now complete.

3.1.3 Framing of Messages

Messages are divided into frames (see figure 3). Based on BEEP specification (RFC3080) a message is normally sent in a single frame. Frames are containing of a header, a payload, and a trailer. The header and trailer are encoded in printable ASCII and terminated with a CRLF – in other words, typically IETF protocol style.

The payload is encoded in an arbitrary set of octets. Every payload octet sent in each direction on a channel has an associated sequence number. Numbering of payload octets within a frame is such that the first payload octet is the lowest numbered, and the following payload octets are numbered consecutively.

The frame trailer consists of “END” followed by a CRLF pair. When receiving a frame, if the characters immediately following the payload don’t correspond to a trailer, then the session will be terminated without generating a response.

The frame headers are almost identical and are structured as follows:

```
  type SP channel SP msgno SP more SP seqno SP size CR LF
```

SP, CR and LF stands for the ASCII space, carriage-return and linefeed characters respectively.

The type specifies the message type and is a three-byte strings – “MSG”, “RPY”, “ANS”, “ERR”, or “NUL” – which corresponding to the message types listed in the exchange patterns above.

The channel identifies the multiplex channel of the communication and is the printed form of a number between zero and \(2^{31} - 1\). BEEP reserves channel zero for management tasks, like creating new channels. So the simplest BEEP application will need two channels and is, therefore, likely to end up being multiplex-ready.
The **msgno** uniquely identifies the message. It's a number in the same format as channel and acts like a session identifier: a reply to a given message will have the same **msgno**. A **msgno** cannot be reused until the final response packet – “RPY”, “ERR” or “NUL” – has been received.

The **more** indicator is represented either by “*” or by “•”. The code “•” describes, that the current frame is the only or the last frame of a message. In comparison the code “*” means, that the current frame has one ore more following frames, which also belongs to the same message.

The **seqno** is an unsigned 32-bit number and specifies the offset of the first octet in the current payload. This is different from the conventional use of the term “sequence number”: perhaps “offset” might have been a better chose. In most cases, the **seqno** of frame n will be the sum of the payload lengths of the prior n – 1 packets. Some applications, however, might want to efficiently use BEEP to omit large sets of default values. For example, a distributed file system protocol could avoid sending a large number of all-zero-byte frames.

The final field is the **size**, which has the obvious meaning of specifying the number of bytes in the message payload. While BEEP is often used for XML or other textual data, payload can be arbitrary. Payloads are MIME objects, which may have MIME headers describing the type and encoding of the payload.

---

![Figure 3: structure of a BEEP message](image-url)
3.1.4 Message Semantics

After the client has established a connection, messages of various types will be exchanged between the client (C) and the server (S). There are five different types of messages relevant in a BEEP session, these are: “greeting”, “start”, “close”, “ok”, “error”. Depending on the different message types the client and the server can send one of the registered elements as shown in the following table.

<table>
<thead>
<tr>
<th>rule</th>
<th>MSG</th>
<th>RPY</th>
<th>ERR</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and S</td>
<td>greeting</td>
<td>error</td>
<td></td>
</tr>
<tr>
<td>C or S</td>
<td>start</td>
<td>profile</td>
<td>error</td>
</tr>
<tr>
<td>C or S</td>
<td>close</td>
<td>ok</td>
<td>error</td>
</tr>
</tbody>
</table>

Table 1: types and elements of messages

While creating a BEEP session the client sends a greeting message to channel zero of the listener to signify its availability. If the listener is not available, a negative answer will be sent and the session is closed, otherwise a positive answer with greeting element is sent back to go on with further communication. When a BEEP peer wants to close a channel, it sends a close element on channel zero. When the server agrees to close a channel, it sends an ok element in a positive reply and the connection is closed immediately.

Further on an example communication is shown. The first message of a BEEP session has the message number zero and includes a greeting element.

With this message the receiver is informed that the sender is ready to start the communication, e.g.:

C: RPY 0 0 . 0 110
C: Content-Type: application/beep+xml
C: <greeting>
C:     <profile uri='html://iana.org/beep/TLS' />
C: </greeting>
C: END

S: RPY 0 0 . 0 110
S: Content-Type: application/beep+xml
S: <greeting />
S: END

Shown here in this example two messages of the client and the server are send one after the other. The answer is related to the clients greeting.

If a peer intents to open an additional channel, a message with start element will be sent on channel 0. The start element has a number attribute, an optional serverName attribute and one or more profile elements. The number attribute indicates the channel number used to identify the channel in future messages. To avoid channel number conflicts, the client should use odd positive integer numbers and the server should use even positive integer numbers. The serverName attribute indicates the
desired server name for the BEEP session. Each profile element contained in the
start element has an URI attribute identifying the profile, e.g.:

C: MSG 0 1 . 30 120
C: Content-Type: application/beep+xml
C: <start number='1' serverName='example.com'>
  <profile uri='http://iana.org/beep/SASL/OTP' />
  <profile uri='http://ibr.cs.de/fangming/contact' />
C: </start>
C: END

If a peer intent to close a channel, a close element will be sent to channel 0. To
agree this, an ok element will be returned from the other peer, e.g.:

C: MSG 0 2 . 390 80
C: Content-Type: application/beep+xml
C: <close number='1' code='200' />
C: END

S: RPY 0 2 . 210 257
S: Content-Type: application/beep+xml
S: <ok />
S: END

The close element has a number attribute and a code attribute. The number attribute
indicates the channel number. The code attribute is a three-digit reply code
meaningful to programs.

If a peer is not able to solve the task of the other peer, e.g. it cannot open a new
channel with channel number 10 or may be could not close it, an error message will
be sent to the requesting peer, e.g.:

C: MSG 0 2 . 390 80
C: Content-Type: application/beep+xml
C: <close number='1' code='200' />
S: END

S: ERR 0 2 . 211 27
S: Content-Type: application/beep+xml
S: <error code='550'>still working</error>
S: END

The error element has a code attribute and an optional textual diagnostic. The code
attribute is meaningful to programs. The textual diagnostic, which gives hints for the
reason of the error, is meaningful to implementers or users, but never to programs.
3.2 Asynchrony

BEEP accommodates asynchronous interactions, both within a single channel and between separate channels. This feature allows pipelining and parallelism.

- **Within a single channel**: A client may send multiple MSG messages on the same channel without waiting to receive the corresponding replies. This provides pipelining within a single channel. A server must process all MSG messages for a given channel in the same order as they are received. As a consequence, the server must generate replies in the same order as the corresponding MSG messages are received on a given channel.

- **Between different channels**: A client may send multiple MSG messages on different channels without waiting to receive the corresponding replies. Channels operate independently in parallel. A server may process MSG messages received on different channels in any order it chooses. As a consequence, although the replies for a given channel appear to be generated in the same order in which the corresponding MSG messages are received, there is no ordering constraint for replies on different channels.

3.3 Transport Mappings

All transport interactions occur in the context of a session – a mapping onto a particular transport service. Any document describing how a particular transport service realizes a BEEP session must satisfy the following requirements.

**Session Management**

A BEEP session is connection-oriented. A mapping document must define:

- how a BEEP session is established;
- how a BEEP peer is identified as acting in the listening role;
- how a BEEP peer is identified as acting in the initiating role;
- how a BEEP session is terminated.

**Message Exchange**

A BEEP session is message-oriented. A mapping document must define:

- how messages are reliably sent and received;
- how messages on the same channel are received in the same order as they were sent;
- how messages on different channels are sent without ordering constraint.

To illustrate how a mapping for BEEP can look like BEEP mapping onto TCP is described in the next chapter.
4. Mapping BEEP onto TCP

The mapping of BEEP session management onto the TCP service is straight-forward. A BEEP session is established when a connection is established between two BEEP peers. In this connection the BEEP peer that issues a passive TCP OPEN call is termed the listener and the peer that issues an active TCP OPEN call is called the initiator.

If both BEEP peers agree to release a BEEP session, the peer will send the ok reply, immediately issues the TCP CLOSE call. Upon receiving the reply, the other peer immediately issues the TCP CLOSE call.

A BEEP session is terminated when one peer issues the TCP ABORT call, and the TCP connection is subsequently aborted.

4.1 Message Exchange

Messages are reliably sent and received using TCP’s SEND and RECEIVE calls. If multiple channels are simultaneously in use on a BEEP session, BEEP must provide a mechanism to avoid starvation and deadlock. To archive this, each channel has a sliding window that indicates the number of payload octets that a peer may transmit before receiving further permission.

4.2 Flow Control

Every payload sent on a channel has a sequence number (see section 3.1.3). When a channel is created, the sequence number associated with the first payload octet of the first data frame is 0, and the initial window size for that channel is 4096 octets. After channel creation, a BEEP peer may update the window size by sending a SEQ frame. A SEQ frame is structured as follows (see figure 3):

```
SEQ SP channel SP ackno SP window CR LF
```

The `channel` identifies the multiplex channel of the communication and is the printed form of a number between zero and $2^{31} - 1$.

The `ackno` indicates the value of the next sequence number that the sender is expecting to receive on this channel.

`Window` describes the window size, that indicates the number of payload octets beginning with the one indicated by the acknowledgement number (ackno) that the sender is expecting to receive on this channel.

When a BEEP peer is asked to create a channel and it is unable to allocate at least 4096 octets for that channel, it must decline creation of the channel. Similarly, during
establishment of the BEEP session, if the BEEP peer acting in the listening role is unable to allocate at least 4096 octets for channel 0, then it must return a negative reply.

Before a message is sent, the sending BEEP peer must ensure that the size of the payload is within the window advertised by the receiving BEEP peer. If not, it has three choices:

1. The BEEP peer may segment the message and start by sending a smaller data frame up to the size of the remaining window.

2. The BEEP peer may delay sending the message until the window becomes larger.

3. The BEEP peer may signal to its application that it is unable to send the message, allowing the application to try again at a later time.
5. BEEP Profiles

A profile explains which messages can or must be transmitted on a channel, e.g. transport security and authentication are described within their respective profiles. A profile is identified with a URI and it defines the syntax and semantic of registered messages. There are two different forms of profiles:

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

2. **Data-exchange profiles** set expectation between the two peers as to what sort of exchanges will be allowed in a channel. For instance, the “Echo” profile from the BEEP Java tools has the URI http://xml.resource.org/profiles/NULL/ECHO.

BEEP defines a number of profiles for channel management, TLS (can realise a user oriented profile based on practical needs.

The TLS transmission security profile is identified in the BEEP profile element while creating a channel as [http://iana.org/beep/TLS](http://iana.org/beep/TLS). The related profile element within the start element can contain a ready element. If the channel is established successfully the BEEP peer will send a proceed element within the related profile element, e.g.:

```
C: MSG 0 1 . 52 80
C: Content-Type: application/beep+xml
C:
C: <start number='1' >
C:   <profile uri='http://iana.org/beep/TLS'>
C:     <![CDATA[<ready />]]>
C: </profile>
C: </start>
C: END
```

```
S: RPY 0 2 . 121 273
S: Content-Type: application/beep+xml
S: <profile uri='http://iana.org/beep/TLS'>
S: <![CDATA[<proceed />]]>
S: </profile>
S: END
```

There can be an optional attribute version within the ready element. This version attribute defines the oldest version of TLS that allows the current interaction. The proceed element has no attribute, it is just the answer to the ready element.

The next section specifies how SOAP envelopes are transmitted using a BEEP profile.

SOAP stands for *Simple Object Access Protocol* or *Service Oriented Access Protocol*. It is a lightweight protocol for exchange of information in a decentralized, distributed environment. It is an XML based protocol that consists of three parts: an envelope that defines a framework for describing what is in a message and how to process it, a set of encoding rules for expressing instances of application-defined data types, and a convention for representing remote procedure calls and responses. SOAP can potentially be used in combination with a variety of other protocols; however, the only bindings described in this document show how to use SOAP in combination with either BEEP or HTTP.

6.1 Using SOAP in BEEP

The BEEP profile for SOAP is identified as: [http://iana.org/beep/soap](http://iana.org/beep/soap) in the BEEP profile element during channel creation.

There are two states in the BEEP profile for SOAP, boot and ready. In the boot state, the peer requesting the creation of the channel sends a “bootmsg”. If the other peer sends a “bootrpy”, then the ready state will be entered. Otherwise, the other peer sends an “error”. The boot message has a resource attribute and an optional feature attribute, which contains one or more feature tokens indicating an optional feature of the BEEP profile for SOAP. The boot message is used for two purposes:

1. Resource identification: Each channel bound to the BEEP profile for SOAP provides access to a single resource (e.g. a network data object or service).
2. Feature negotiation: If new features of SOAP emerge, their use can be negotiated.

In the ready state, one peer begins a SOAP message pattern by sending a “MSG” message containing an envelope. The other peer completes the message pattern either by sending back a “RPY” message containing an envelope or sending back zero or more "ANS" messages (each containing an envelope) following by a “NUL” message.
6.1.1 SOAP BEEP Example

Here the boot message and its response are exchanged during channel initialisation:

C:   <start number='1' >
C:     <profile uri='http://iana.org/beep/soap'>
C:       <![CDATA[<bootmsg resource='/StockQuote />
C:     </profile>
C:   </start>

S:   <profile uri='http://iana.org/beep/soap'>
S:     <![CDATA[<bootrpy />
S:   </profile>

The channel bound to the BEEP profile for SOAP is now in the ready state.

The BEEP profile for SOAP transmits envelopes encoded as UTF-8 using the media type “application/xml”, e.g.:

MSG 1 1 . 0 364
Content-Type: application/xml

<SOAP-ENV:Envelope ...>
   <SOAP-ENV:Body>
      ...
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>

More than one envelope can be transmitted at the same time on the same channel if the start element within a BEEP message contains more than one profiles. In comparison using SOAP in HTTP, this would not be possible. The next section describes how a SOAP envelope can be transmitted within a HTTP request or response.
6.2 Using SOAP in HTTP

This section describes how to use SOAP within HTTP. Carrying SOAP in HTTP does not mean that SOAP overrides existing semantics of HTTP but rather that the semantics of SOAP over HTTP maps naturally to HTTP semantics. SOAP naturally follows the HTTP request/response message model providing SOAP request parameters in a HTTP request and SOAP response parameters in a HTTP response. Note, however, that SOAP intermediaries are not the same as HTTP intermediaries. That is, an HTTP intermediary addressed with the HTTP connection header field cannot be expected to inspect or process the SOAP entity body carried in the HTTP request.

6.2.1 SOAP HTTP Request

Although SOAP might be used in combination with a variety of HTTP request methods, this binding only defines SOAP within HTTP POST. The HTTP POST request specifies at least three HTTP header fields.

1. **Soap Action**: The SOAP Action header field can be used to indicate the intent of the SOAP HTTP request. The value is a URI identifying the intent. SOAP places no restrictions on the format or specificity of the URI. An HTTP client must use this header field when issuing a SOAP HTTP Request. The presence and content of the SOAP Action header field can be used by servers such as firewalls to appropriately filter SOAP request messages in HTTP. The header field value of empty string (" ") means that the intent of the SOAP message is provided by the HTTP Request-URI. No value means that there is no indication of the intent of the message.

2. **Content-Type**: The Content-Type header field for a SOAP request and response defines the MIME type for the message and the character encoding used for the XML body of the request or response.

3. **Content-Length**: The Content-Length header field specifies the number of bytes in the body of the request or response.

6.2.2 SOAP HTTP Response

SOAP HTTP follows the semantics of the HTTP Status codes for communicating status information in HTTP. For example, a 2xx status code indicates that the clients request including the SOAP component was successfully received, understood and accepted etc.

In case of a SOAP error while processing the request, the SOAP HTTP server MUST issue an HTTP 500 "Internal Server Error" response and include a SOAP message in the response containing a SOAP fault element indicating the SOAP processing error.
6.2.3 SOAP HTTP Examples

Request:

POST /StockQuote HTTP/1.1
Content-Type: application/soap+xml; charset="utf-8"
Content-Length: 280
SOAPAction: "http://electrocommerce.org/abc#MyMessage"

<br/>&lt;SOAP-ENV:Envelope ...
 &nbsp; &nbsp; &lt;SOAP-ENV:Body>
 &nbsp; &nbsp; ...
 &nbsp; &nbsp; &lt;/SOAP-ENV:Body>
 &lt;/SOAP-ENV:Envelope>
<br/>

Response:

HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset="utf-8"
Content-Length: 280
<br/>

<br/>&lt;SOAP-ENV:Envelope ...
 &nbsp; &nbsp; &lt;SOAP-ENV:Body ....
 &nbsp; &nbsp; ...
 &nbsp; &nbsp; &lt;/SOAP-ENV:Body>
 &lt;/SOAP-ENV:Envelope>
<br/>

In comparison to BEEP this example shows, if more than one SOAP envelope should be exchanged over HTTP between two peers, it will be necessary to send a new request and get the corresponding response for each envelope. Each request or response contains only one SOAP envelope. But this is not the only disadvantage of HTTP in comparison to BEEP.
7. Comparison between BEEP and HTTP

BEEP is a peer-to-peer protocol. Naturally, one process must be “listening” while the other process is “initiating” a connection, but once the TCP connection is established, the two processes are completely symmetric in their use of the protocol. A peer-to-peer protocol like BEEP can be contrasted with the popular client-server protocol HTTP.

7.1 Asynchronous Notification

There is sometimes the need to have the server send a request to a client, in addition to the traditional responding to requests from the client. This is often the case when a client wishes to interact with programmatic components on the server via SOAP. But with a HTTP connection between a client and an origin server, the only way that a server can send information to a client is as part of a response to a previous request from a client. There are a number of available techniques allowing a server to send asynchronous notifications over HTTP to a client. But all this techniques based on HTTP involve compromise – as trying to use HTTP for a task it was not originally designed for. The solution for this problem is given by the completely different application protocol BEEP, but the problem is that this is not widely deployed. The reason HTTP remains an interesting basis for a solution is that support for it is widely deployed on clients and origin servers and it can travel through firewalls via HTTP proxy servers, which are also widely deployed. Many firewall administrators simply will not permit other application protocols through.

7.2 Multiple Requests

With HTTP, every request must perform a 3-way TCP handshake to establish the connection. Each request must include identify information and authentication information, must determine the optimal TCP window size and IP packet size and must renegotiate encryption. HTTP must use “session cookies” or similar technology for each request to route the request to the same machine and process that handled the previous request.

BEEP based applications can be sure that multiple requests are all being handled by the same peer, eliminating the need for “session cookies” and other such state-management information. Fetching a web page with multiple embedded images using BEEP instead of HTTP simply uses multiple requests over the same channel or multiple channels, if interleaving of downloads is desired. Since only one connection is established for the lifetime of the conversation, the overhead of 3-way handshakes, authentication, encryption negotiation, TCP window “slow start” and the bandwidth and processing power that goes with all of that, is conserved.
7.2.1 Features of HTTP/1.1 by Contrast with BEEP

HTTP opens and closes a new TCP connection for each operation. Since most Web objects are small, this practice means a high fraction of TCP control packets are used to open and close a connection. Furthermore, when a TCP connection is first opened, TCP employs an algorithm known as slow start. Slow start uses the first several data packets to probe the network to determine the optimal transmission rate. If Web objects are small, most objects will be transferred before their TCP connection completes the slow start algorithm. An unnecessary overhead will be the result.

HTTP/1.1 tries to reduce this overhead by leaving the TCP connection open between consecutive operations. This technique is called "persistent connections", which avoids the costs of multiple opens and closes and reduces the impact of slow start. Persistent connections are more efficient than the current practice of running multiple short TCP connections in parallel. By leaving the TCP connection open between requests, many packets can be avoided.

Persistent connections allow multiple requests to be sent without waiting for a response. Multiple requests and responses can be contained in a single TCP segment. This can be used to avoid many round trip delays, improving performance and reducing the number of packets further. This technique is called "pipelining" in HTTP. Using pipelining, the requestor waits for the responses to arrive in the order in which they were requested.

Pipelining can also dramatically reduce the number of TCP/IP packets. With a typical MSS (maximum segment size) of 512 bytes, it is possible to pack several HTTP requests into one TCP/IP packet.

Note that HTTP/1.1 does not allow multiplexing multiple transfers simultaneously in a single TCP connection. Multiple responses are always serialised.

BEEP allows a single application to operate in multiple roles and even multiple roles simultaneously, often by eliminating what would otherwise be a complex negotiation or an awkward tying of multiple transport connections. For example, even within a client-server type of application, BEEP allows the server to spontaneously notify the client of events, rather than requiring a client to poll periodically for changes.

Beep is similar to HTTP but more flexible. It allows bi-directional request and response and it can multiplex multiple channels onto the same connection. It’s designed to carry applications that otherwise are overloading the semantics of HTTP.
8. Summary

BEEP is a connection-oriented protocol that supports asynchronous interaction. The messages are divided into frames and are sent between two peers in different styles of message exchange. The data are represented by XML to offer flexible data exchange. On this base, profiles like those for SOAP were developed. A BEEP message can contain one or more profiles, which define the syntax and semantics of the messages exchanged. BEEP offers standard profiles which supports transport security and authentication.

BEEP solves once a number of problems common to most connection-oriented application protocols. It specifies a standard for framing messages to separate them from earlier and later messages. It specifies mechanisms for synchronising messages, for managing parallel operations, and for pipelining operations. Since every modern application should consider authentication and privacy, BEEP builds in extensible mechanisms for handling both of these without effort from the application programmer.

Of course, there are a number of situations in which the use of BEEP would be inappropriate. If a protocol already exists and is widely deployed, there is little sense in rewriting it to take advantage of BEEP. For example, it would be silly to specify email delivery to use BEEP when the entire world already uses SMTP.

In comparison to HTTP, BEEP has many advantages. But it is not to be neglected that HTTP is already deployed and many persons like developers or administrators can understand it.
9. Literature


(2) M. Rose, Mapping the BEEP Core onto TCP, RFC 3081, March 2001

(3) E. O’Tuathail and M. Rose, Using SOAP in BEEP, RFC 3288, June 2002


(6) E. Dumbill, Worm’s-eye BEEP, Part 2 of an introduction to the Blocks Extensible Exchange Protocol standard of the IETF, March 2002

(7) SOAP Tutorial: SOAP HTTP Binding, URL: http://www.w3schools.com/soap/soap-httpbinding.asp


(10) HTTP Asynchronous Client Notification, technical Whitepaper URL: http://clipcode.com

10. Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANS:</td>
<td>Answer</td>
</tr>
<tr>
<td>BEEP:</td>
<td>Blocks Extensible Exchange Protocol</td>
</tr>
<tr>
<td>CORBA:</td>
<td>Common Object Request Broker Architecture</td>
</tr>
<tr>
<td>ENV:</td>
<td>Envelope</td>
</tr>
<tr>
<td>ERR:</td>
<td>Error</td>
</tr>
<tr>
<td>FTP:</td>
<td>File Transfer Protocol</td>
</tr>
<tr>
<td>HTTP:</td>
<td>HyperText Transfer Protocol</td>
</tr>
<tr>
<td>MIME:</td>
<td>Multipurpose Internet Mail Extensions</td>
</tr>
<tr>
<td>MSG:</td>
<td>Message</td>
</tr>
<tr>
<td>MSS:</td>
<td>Maximum Segment Size</td>
</tr>
<tr>
<td>RFC:</td>
<td>Request For Comments</td>
</tr>
<tr>
<td>RPC:</td>
<td>Remote Procedure Calling Protocol</td>
</tr>
<tr>
<td>RPY:</td>
<td>Reply</td>
</tr>
<tr>
<td>SMTP:</td>
<td>Simple Mail Transfer Protocol</td>
</tr>
<tr>
<td>SOAP:</td>
<td>Simple Object Access Protocol</td>
</tr>
<tr>
<td>TCP:</td>
<td>Transmission Control Protocol</td>
</tr>
<tr>
<td>TLS:</td>
<td>Transport Layer Security</td>
</tr>
<tr>
<td>UDP:</td>
<td>User Datagram Protocol</td>
</tr>
<tr>
<td>URI:</td>
<td>Uniform Resource Identifier</td>
</tr>
<tr>
<td>XML:</td>
<td>Extensible Markup Language</td>
</tr>
</tbody>
</table>