Network Working Group
Request for Comments: 3117
Category: Informational

M. Rose
Dover Beach Consulting, Inc.

November 2001

On the Design of Application Protocols

Status of this Memo

This memo provides information for the Internet community. It does not specify an Internet standard of any kind. Distribution of this memo is unlimited.

Copyright Notice

Abstract

This memo describes the design principles for the Blocks eXtensible eXchange Protocol (BXXP). BXXP is a generic application protocol framework for connection-oriented, asynchronous interactions. The framework permits simultaneous and independent exchanges within the context of a single application user-identity, supporting both textual and binary messages.

Table of Contents

1. A Problem 19 Years in the Making

SMTP [1] is close to being the perfect application protocol: it solves a large, important problem in a minimalist way. It's simple enough for an entry-level implementation to fit on one or two screens of code, and flexible enough to form the basis of very powerful product offerings in a robust and competitive market. Modulo a few oddities (e.g., SAML), the design is well conceived and the resulting specification is well-written and largely self-contained. There is very little about good application protocol design that you can't learn by reading the SMTP specification.

Unfortunately, there's one little problem: SMTP was originally published in 1981 and since that time, a lot of application protocols have been designed for the Internet, but there hasn't been a lot of reuse going on. You might expect this if the application protocols were all radically different, but this isn't the case: most are surprisingly similar in their functional behavior, even though the actual details vary considerably.

In late 1998, as Carl Malamud and I were sitting down to review the Blocks architecture, we realized that we needed to have a protocol for exchanging Blocks. The conventional wisdom is that when you need an application protocol, there are four ways to proceed:

1. find an existing exchange protocol that (more or less) does what you want;

2. define an exchange model on top of the world-wide web infrastructure that (more or less) does what you want;

3. define an exchange model on top of the electronic mail infrastructure that (more or less) does what you want; or,

4. define a new protocol from scratch that does exactly what you want.

An engineer can make reasoned arguments about the merits of each of the these approaches. Here's the process we followed...

The most appealing option is to find an existing protocol and use that. (In other words, we'd rather "buy" than "make".) So, we did a survey of many existing application protocols and found that none of them were a good match for the semantics of the protocol we needed.

For example, most application protocols are oriented toward client/server behavior, and emphasize the client pulling data from the server; in contrast with Blocks, a client usually pulls data from

the server, but it also may request the server to asynchronously push (new) data to it. Clearly, we could mutate a protocol such as FTP [2] or SMTP into what we wanted, but by the time we did all that, the base protocol and our protocol would have more differences than similarities. In other words, the cost of modifying an off-the-shelf implementation becomes comparable with starting from scratch.

Another approach is to use HTTP [3] as the exchange protocol and define the rules for data exchange over that. For example, IPP [4] (the Internet Printing Protocol) uses this approach. The basic idea is that HTTP defines the rules for exchanging data and then you define the data's syntax and semantics. Because you inherit the entire HTTP infrastructure (e.g., HTTP's authentication mechanisms, caching proxies, and so on), there's less for you to have to invent (and code!). Or, conversely, you might view the HTTP infrastructure as too helpful. As an added bonus, if you decide that your protocol runs over port 80, you may be able to sneak your traffic past older firewalls, at the cost of port 80 saturation.

HTTP has many strengths: it's ubiquitous, it's familiar, and there are a lot of tools available for developing HTTP-based systems. Another good thing about HTTP is that it uses MIME [5] for encoding data.

Unfortunately for us, even with HTTP 1.1 [6], there still wasn't a good fit. As a consequence of the highly-desirable goal of maintaining compatibility with the original HTTP, HTTP's framing mechanism isn't flexible enough to support server-side asynchronous behavior and its authentication model isn't similar to other Internet applications.

Mapping IPP onto HTTP 1.1 illustrates the former issue. For example, the IPP server is supposed to signal its client when a job completes. Since the HTTP client must originate all requests and since the decision to close a persistent connection in HTTP is unilateral, the best that the IPP specification can do is specify this functionality in a non-deterministic fashion.

Further, the IPP mapping onto HTTP shows that even subtle shifts in behavior have unintended consequences. For example, requests in IPP are typically much larger than those seen by many HTTP server implementations -- resulting in oddities in many HTTP servers (e.g., requests are sometimes silently truncated). The lesson is that HTTP's framing mechanism is very rigid with respect to its view of the request/response model.

Lastly, given our belief that the port field of the TCP header isn't a constant 80, we were immune to the seductive allure of wanting to sneak our traffic past unwary site administrators.

The third choice, layering the protocol on top of email, was attractive. Unfortunately, the nature of our application includes a lot of interactivity with relatively small response times. So, this left us the final alternative: defining a protocol from scratch.

To begin, we figured that our requirements, while a little more stringent than most, could fit inside a framework suitable for a large number of future application protocols. The trick is to avoid the kitchen-sink approach. (Dave Clark has a saying: "One of the roles of architecture is to tell you what you can't do.")

2. You can Solve Any Problem...

...if you're willing to make the problem small enough.

Our most important step is to limit the problem to application protocols that exhibit certain features:

- they are connection-oriented;
- they use requests and responses to exchange messages; and,
- they allow for asynchronous message exchange.

Let's look at each, in turn.

First, we're only going to consider connection-oriented application protocols (e.g., those that work on top of TCP [7]). Another branch in the taxonomy, connectionless, consists of those that don't want the delay or overhead of establishing and maintaining a reliable stream. For example, most DNS [8] traffic is characterized by a single request and response, both of which fit within a single IP datagram. In this case, it makes sense to implement a basic reliability service above the transport layer in the application protocol itself.

Second, we're only going to consider message-oriented application protocols. A "message" -- in our lexicon -- is simply structured data exchanged between loosely-coupled systems. Another branch in the taxonomy, tightly-coupled systems, uses remote procedure calls as the exchange paradigm. Unlike the connection-oriented/connectionless dichotomy, the issue of loosely- or tightly-coupled systems is similar to a continuous spectrum. Fortunately, the edges are fairly sharp.

For example, NFS [9] is a tightly-coupled system using RPCs. When running in a properly-configured LAN, a remote disk accessible via NFS is virtually indistinguishable from a local disk. To achieve this, tightly-coupled systems are highly concerned with issues of latency. Hence, most (but not all) tightly-coupled systems use connection-less RPC mechanisms; further, most tend to be implemented as operating system functions rather than user-level programs. (In some environments, the tightly-coupled systems are implemented as single-purpose servers, on hardware specifically optimized for that one function.)

Finally, we're going to consider the needs of application protocols that exchange messages asynchronously. The classic client/server model is that the client sends a request and the server sends a

response. If you think of requests as "questions" and responses as "answers", then the server answers only those questions that it's asked and it never asks any questions of its own. We'll need to support a more general model, peer-to-peer. In this model, for a given transaction one peer might be the "client" and the other the "server", but for the next transaction, the two peers might switch roles.

It turns out that the client/server model is a proper subset of the peer-to-peer model: it's acceptable for a particular application protocol to dictate that the peer that establishes the connection always acts as the client (initiates requests), and that the peer that listens for incoming connections always acts as the server (issuing responses to requests).

There are quite a few existing application domains that don't fit our requirements, e.g., nameservice (via the DNS), fileservice (via NFS), multicast-enabled applications such as distributed video conferencing, and so on. However, there are a lot of application domains that do fit these requirements, e.g., electronic mail, file transfer, remote shell, and the world-wide web. So, the bet we are placing in going forward is that there will continue to be reasons for defining protocols that fit within our framework.

3. Protocol Mechanisms

The next step is to look at the tasks that an application protocol must perform and how it goes about performing them. Although an exhaustive exposition might identify a dozen (or so) areas, the ones we're interested in 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 independent exchanges are handled;
- authentication, which tells how the peers at each end of the connection are identified and verified; and,
- privacy, which tells how the exchanges are protected against third-party interception or modification.

A notable absence in this list is naming -- we'll explain why later on.

3.1 Framing

There are three commonly used approaches to delimiting messages: octet-stuffing, octet-counting, and connection-blasting.

An example of a protocol that uses octet-stuffing is SMTP. Commands in SMTP are line-oriented (each command ends in a CR-LF pair). When an SMTP peer sends a message, it first transmits the "DATA" command, then it transmits the message, then it transmits a "." (dot) followed by a CR-LF. If the message contains any lines that begin with a dot, the sending SMTP peer sends two dots; similarly, when the other SMTP peer receives a line that begins with a dot, it discards the dot, and, if the line is empty, then it knows it's received the entire message. Octet-stuffing has the property that you don't need the entire message in front of you before you start sending it. Unfortunately, it's slow because both the sender and receiver must scan each line of the message to see if they need to transform it.

An example of a protocol that uses octet-counting is HTTP. Commands in HTTP consist of a request line followed by headers and a body. The headers contain an octet count indicating how large the body is. The properties of octet-counting are the inverse of octet-stuffing:

before you can start sending a message you need to know the length of the whole message, but you don't need to look at the content of the message once you start sending or receiving.

An example of a protocol that uses connection-blasting is FTP. Commands in FTP are line-oriented, and when it's time to exchange a message, a new TCP connection is established to transmit the message. Both octet-counting and connection-blasting have the property that the messages can be arbitrary binary data; however, the drawback of the connection-blasting approach is that the peers need to communicate IP addresses and TCP port numbers, which may be "transparently" altered by NATS [10] and network bugs. In addition, if the messages being exchanged are small (say less than 32k), then the overhead of establishing a connection for each message contributes significant latency during data exchange.

3.2 Encoding

There are many schemes used for encoding data (and many more encoding schemes have been proposed than are actually in use). Fortunately, only a few are burning brightly on the radar.

The messages exchanged using SMTP are encoded using the 822-style [11]. The 822-style divides a message into textual headers and an unstructured body. Each header consists of a name and a value and is terminated with a CR-LF pair. An additional CR-LF separates the headers from the body.

It is this structure that HTTP uses to indicate the length of the body for framing purposes. More formally, HTTP uses MIME, an application of the 822-style to encode both the data itself (the body) and information about the data (the headers). That is, although HTTP is commonly viewed as a retrieval mechanism for HTML [12], it is really a retrieval mechanism for objects encoded using MIME, most of which are either HTML pages or referenced objects such as GIFs.

3.3 Reporting

An application protocol needs a mechanism for conveying error information between peers. The first formal method for doing this was defined by SMTP's "theory of reply codes". The basic idea is that an error is identified by a three-digit string, with each position having a different significance:

the first digit: indicating success or failure, either permanent or transient;

the second digit: indicating the part of the system reporting the situation (e.g., the syntax analyzer); and,

the third digit: identifying the actual situation.

Operational experience with SMTP suggests that the range of error conditions is larger than can be comfortably encoded using a three-digit string (i.e., you can report on only 10 different things going wrong for any given part of the system). So, [13] provides a convenient mechanism for extending the number of values that can occur in the second and third positions.

Virtually all of the application protocols we've discussed thus far use the three-digit reply codes, although there is less coordination between the designers of different application protocols than most would care to admit. (A variation on the theory of reply codes is employed by IMAP [14] which provides the same information using a different syntax.)

In addition to conveying a reply code, most application protocols also send a textual diagnostic suitable for human, not machine, consumption. (More accurately, the textual diagnostic is suitable for people who can read a widely used variant of the English language.) Since reply codes reflect both positive and negative outcomes, there have been some innovative uses made for the text accompanying positive responses, e.g., prayer wheels [39]. Regardless, some of the more modern application protocols include a language localization parameter for the diagnostic text.

Finally, since the introduction of reply codes in 1981, two unresolved criticisms have been raised:

- a reply code is used both to signal the outcome of an operation and a change in the application protocol's state; and,

- a reply code doesn't specify whether the associated textual diagnostic is destined for the end-user, administrator, or programmer.

3.4 Asynchrony

Few application protocols today allow independent exchanges over the same connection. In fact, the more widely implemented approach is to allow pipelining, e.g., command pipelining [15] in SMTP or persistent connections in HTTP 1.1. Pipelining allows a client to make multiple requests of a server, but requires the requests to be processed serially. (Note that a protocol needs to explicitly provide support for pipelining, since, without explicit guidance, many implementors

produce systems that don't handle pipelining properly; typically, an error in a request causes subsequent requests in the pipeline to be discarded).

Pipelining is a powerful method for reducing network latency. For example, without persistent connections, HTTP's framing mechanism is really closer to connection-blasting than octet-counting, and it enjoys the same latency and efficiency problems.

In addition to reducing network latency (the pipelining effect), asynchrony also reduces server latency by allowing multiple requests to be processed by multi-threaded implementations. Note that if you allow any form of asynchronous exchange, then support for parallelism is also required, because exchanges aren't necessarily occurring under the synchronous direction of a single peer.

Unfortunately, when you allow parallelism, you also need a flow control mechanism to avoid starvation and deadlock. Otherwise, a single set of exchanges can monopolize the bandwidth provided by the transport layer. Further, if a peer is resource-starved, then it may not have enough buffers to receive a message and deadlock results.

Flow control is typically implemented at the transport layer. For example, TCP uses sequence numbers and a sliding window: each receiver manages a sliding window that indicates the number of data octets that may be transmitted before receiving further permission. However, it's now time for the second shoe to drop: segmentation. If you do flow control then you also need a segmentation mechanism to fragment messages into smaller pieces before sending and then re-assemble them as they're received.

Both flow control and segmentation have an impact on how the protocol does framing. Before we defined framing as "how to tell the beginning and end of each message" -- in addition, we need to be able to identify independent messages, send messages only when flow control allows us to, and segment them if they're larger than the available window (or too large for comfort).

Segmentation impacts framing in another way -- it relaxes the octet-counting requirement that you need to know the length of the whole message before sending it. With segmentation, you can start sending segments before the whole message is available. In HTTP 1.1 you can "chunk" (segment) data to get this advantage.

3.5 Authentication

Perhaps for historical (or hysterical) reasons, most application protocols don't do authentication. That is, they don't authenticate the identity of the peers on the connection or the authenticity of the messages being exchanged. Or, if authentication is done, it is domain-specific for each protocol. For example, FTP and HTTP use entirely different models and mechanisms for authenticating the initiator of a connection. (Independent of mainstream HTTP, there is a little-used variant [16] that authenticates the messages it exchanges.)

A large part of the problem is that different security mechanisms optimize for strength, scalability, or ease of deployment. So, a few years ago, SASL [17] (the Simple Authentication and Security Layer) was developed to provide a framework for authenticating protocol peers. SASL let's you describe how an authentication mechanism works, e.g., an OTP [18] (One-Time Password) exchange. It's then up to each protocol designer to specify how SASL exchanges are generically conveyed by the protocol. For example, [19] explains how SASL works with SMTP.

A notable exception to the SASL bandwagon is HTTP, which defines its own authentication mechanisms [20]. There is little reason why SASL couldn't be introduced to HTTP, although to avoid certain race-conditions, the persistent connection mechanism of HTTP 1.1 must be used.

SASL has an interesting feature in that in addition to explicit protocol exchanges to authenticate identity, it can also use implicit information provided from the layer below. For example, if the connection is running over IPsec [21], then the credentials of each peer are known and verified when the TCP connection is established.

Finally, as its name implies, SASL can do more than authentication -- depending on which SASL mechanism is in use, message integrity or privacy services may also be provided.

3.6 Privacy

HTTP is the first widely used protocol to make use of a transport security protocol to encrypt the data sent on the connection. The current version of this mechanism, TLS [22], is available to all application protocols, e.g., SMTP and ACAP [23] (the Application Configuration Access Protocol).

The key difference between the original mechanism and TLS, is one of provisioning not technology. In the original approach to provisioning, a world-wide web server listens on two ports (one for plaintext traffic and the other for secured traffic); in contrast, by today's conventions, a server implementing an application protocol that is specified as TLS-enabled (e.g., [24] and [25]) listens on a single port for plaintext traffic, and, once a connection is established, the use of TLS on that connection is negotiable.

Finally, note that both SASL and TLS are about "transport security" not "object security". What this means is that they focus on providing security properties for the actual communication, they don't provide any security properties for the data exchanged independent of the communication.

3.7 Let's Recap

Let's briefly compare the properties of the three main connection-oriented application protocols in use today:

Mechanism	ESMTP	FTP	HTTP1.1
Framing	stuffing	blasting	counting
Encoding	822-style	binary	MIME
Reporting	3-digit	3-digit	3-digit
Asynchrony	pipelining	none	pipelining and chunking
Authentication	SASL	user/pass	user/pass
Privacy	SASL or TLS	none	TLS (nee SSL)

Note that the username/password mechanisms used by FTP and HTTP are entirely different with one exception: both can be termed a "username/password" mechanism.

These three choices are broadly representative: as more protocols are considered, the patterns are reinforced. For example, POP [26] uses octet-stuffing, but IMAP uses octet-counting, and so on.

4. Protocol Properties

When we design an application protocol, there are a few properties that we should keep an eye on.

4.1 Scalability

A well-designed protocol is scalable.

Because few application protocols support asynchrony, a common trick is for a program to open multiple simultaneous connections to a single destination. The theory is that this reduces latency and increases throughput. The reality is that both the transport layer and the server view each connection as an independent instance of the application protocol, and this causes problems.

In terms of the transport layer, TCP uses adaptive algorithms to efficiently transmit data as networks conditions change. But what TCP learns is limited to each connection. So, if you have multiple TCP connections, you have to go through the same learning process multiple times -- even if you're going to the same host. Not only does this introduce unnecessary traffic spikes into the network, because TCP uses a slow-start algorithm when establishing a connection, the program still sees additional latency. To deal with the fact that a lack of asynchrony in application protocols causes implementors to make sloppy use of the transport layer, network protocols are now provisioned with increasing sophistication, e.g., RED [27]. Further, suggestions are also being considered for modification of TCP implementations to reduce concurrent learning, e.g., [28].

In terms of the server, each incoming connection must be dispatched and (probably) authenticated against the same resources. Consequently, server overhead increases based on the number of connections established, rather than the number of remote users. The same issues of fairness arise: it's much harder for servers to allocate resources on a per-user basis, when a user can cause an arbitrary number of connections to pound on the server.

Another important aspect of scalability to consider is the relative numbers of clients and servers. (This is true even in the peer-to-peer model, where a peer can act both in the client and server role.) Typically, there are many more client peers than server peers. In this case, functional requirements should be shifted from the servers onto the clients. The reason is that a server is likely to be interacting with multiple clients and this functional shift makes it easier to scale.

4.2 Efficiency

A well-designed protocol is efficient.

For example, although a compelling argument can be made than octet-stuffing leads to more elegant implementations than octet-counting, experience shows that octet-counting consumes far fewer cycles.

Regrettably, we sometimes have to compromise efficiency in order to satisfy other properties. For example, 822 (and MIME) use textual headers. We could certainly define a more efficient representation for the headers if we were willing to limit the header names and values that could be used. In this case, extensibility is viewed as more important than efficiency. Of course, if we were designing a network protocol instead of an application protocol, then we'd make the trade-offs using a razor with a different edge.

4.3 Simplicity

A well-designed protocol is simple.

Here's a good rule of thumb: a poorly-designed application protocol is one in which it is equally as "challenging" to do something basic as it is to do something complex. Easy things should be easy to do and hard things should be harder to do. The reason is simple: the pain should be proportional to the gain.

Another rule of thumb is that if an application protocol has two ways of doing the exact same thing, then there's a problem somewhere in the architecture underlying the design of the application protocol.

Hopefully, simple doesn't mean simple-minded: something that's well-designed accommodates everything in the problem domain, even the troublesome things at the edges. What makes the design simple is that it does this in a consistent fashion. Typically, this leads to an elegant design.

4.4 Extensibility

A well-designed protocol is extensible.

As clever as application protocol designers are, there are likely to be unforeseen problems that the application protocol will be asked to solve. So, it's important to provide the hooks that can be used to add functionality or customize behavior. This means that the protocol is evolutionary, and there must be a way for implementations reflecting different steps in the evolutionary path to negotiate which extensions will be used.

But, it's important to avoid falling into the extensibility trap: the hooks provided should not be targeted at half-baked future requirements. Above all, the hooks should be simple.

Of course good design goes a long way towards minimizing the need for extensibility. For example, although SMTP initially didn't have an extension framework, it was only after ten years of experience that its excellent design was altered. In contrast, a poorly-designed protocol such as Telnet [29] can't function without being built around the notion of extensions.

4.5 Robustness

A well-designed protocol is robust.

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.

Counter-intuitively, Postel's robustness principle ("be conservative in what you send, liberal in what you accept") often leads to deployment problems. Why? When a new implementation is initially fielded, it is likely that it will encounter only a subset of existing implementations. If those implementations follow the robustness principle, then errors in the new implementation will likely go undetected. The new implementation then sees some, but not widespread deployment. This process repeats for several new implementations. Eventually, the not-quite-correct implementations run into other implementations that are less liberal than the initial set of implementations. The reader should be able to figure out what happens next.

Accordingly, explicit consistency checks in a protocol are very useful, even if they impose implementation overhead.

5. The BXXP Framework

Finally, we get to the money shot: here's what we did.

We defined an application protocol framework called BXXP (the Blocks eXtensible eXchange Protocol). The reason it's a "framework" instead of an application protocol is that we provide all the mechanisms discussed earlier without actually specifying the kind of messages that get exchanged. So, when someone else needs an application protocol that requires connection-oriented, asynchronous interactions, they can start with BXXP. It's then their responsibility to define the last 10% of the application protocol, the part that does, as we say, "the useful work".

So, what does BXXP look like?

Mechanism	BXXP
Framing	counting, with a trailer
Encoding	MIME, defaulting to text/xml
Reporting	3-digit and localized textual diagnostic
Asynchrony	channels
Authentication	SASL
Privacy	SASL or TLS

5.1 Framing and Encoding

Framing in BXXP looks a lot like SMTP or HTTP: there's a command line that identifies the beginning of the frame, then there's a MIME object (headers and body). Unlike SMTP, BXXP uses octet-counting, but unlike HTTP, the command line is where you find the size of the payload. Finally, there's a trailer after the MIME object to aid in detecting framing errors.

Actually, the command line for BXXP has a lot of information, it tells you:

- what kind of message is in this frame;
- whether there's more to the message than just what's in this frame (a continuation flag);

- how to distinguish the message contained in this frame from other messages (a message number);

- where the payload occurs in the sliding window (a sequence number) along with how many octets are in the payload of this frame; and,

- which part of the application should get the message (a channel number).

  (The command line is textual and ends in a CR-LF pair, and the arguments are separated by a space.)

Since you need to know all this stuff to process a frame, we put it all in one easy to parse location. You could probably devise a more efficient encoding, but the command line is a very small part of the frame, so you wouldn't get much bounce from optimizing it. Further, because framing is at the heart of BXXP, the frame format has several consistency checks that catch the majority of programming errors. (The combination of a sequence number, an octet count, and a trailer allows for very robust error detection.)

Another trick is in the headers: because the command line contains all the framing information, the headers may contain minimal MIME information (such as Content-Type). Usually, however, the headers are empty. That's because the BXXP default payload is XML [30]. (Actually, a "Content-Type: text/xml" with binary transfer encoding).

We chose XML as the default because it provides a simple mechanism for nested, textual representations. (Alas, the 822-style encoding doesn't easily support nesting.) By design, XML's nature isn't optimized for compact representations. That's okay because we're focusing on loosely-coupled systems and besides there are efficient XML parsers available. Further, there's a fair amount of anecdotal experience -- and we'll stress the word "anecdotal" -- that if you have any kind of compression (either at the link-layer or during encryption), then XML encodings squeeze down nicely.

Even so, use of XML is probably the most controversial part of BXXP. After all, there are more efficient representations around. We agree, but the real issue isn't efficiency, it's ease of use: there are a lot of people who grok the XML thing and there are a lot of XML tools out there. The pain of recreating this social infrastructure far outweighs any benefits of devising a new representation. So, if the "make" option is too expensive, is there something else we can "buy" besides XML? Well, there's ASN.1/BER (just kidding).

In the early days of the SNMP [31], which does use ASN.1, the same issues arose. In the end, the working group agreed that the use of ASN.1 for SNMP was axiomatic, but not because anyone thought that ASN.1 was the most efficient, or the easiest to explain, or even well liked. ASN.1 was given axiomatic status because the working group decided it was not going to spend the next three years explaining an alternative encoding scheme to the developer community.

So -- and we apologize for appealing to dogma -- use of XML as the favored encoding scheme in BXXP is axiomatic.

5.2 Reporting

We use 3-digit error codes, with a localized textual diagnostic. (Each peer specifies a preferred ordering of languages.)

In addition, the reply to a message is flagged as either positive or negative. This makes it easy to signal success or failure and allow the receiving peer some freedom in the amount of parsing it wants to do on failure.

5.3 Asynchrony

Despite the lessons of SMTP and HTTP, there isn't a lot of field experience to rely on when designing the asynchrony features of BXXP. (Actually, there were several efforts in 1998 related to application layer framing, e.g., [32], but none appear to have achieved orbit.)

So, here's what we did: frames are exchanged in the context of a "channel". Each channel has an associated "profile" that defines the syntax and semantics of the messages exchanged over a channel.

Channels provide both an extensibility mechanism for BXXP and the basis for parallelism. Remember the last parameter in the command line of a BXXP frame? The "part of the application" that gets the message is identified by a channel number.

A profile is defined according to a "Profile Registration" template. The template defines how the profile is identified (using a URI [33]), what kind of messages get exchanged, along with the syntax and semantics of those messages. When you create a channel, you identify a profile and maybe piggyback your first message. If the channel is successfully created, you get back a positive response; otherwise, you get back a negative response explaining why.

Perhaps the easiest way to see how channels provide an extensibility mechanism is to consider what happens when a session is established. Each BXXP peer immediately sends a greeting on channel zero

identifying the profiles that each support. (Channel 0 is used for channel management -- it's automatically created when a session is opened.) If you want transport security, the very first thing you do is to create a channel that negotiates transport security, and, once the channel is created, you tell it to do its thing. Next, if you want to authenticate, you create a channel that performs user authentication, and, once the channel is created, you tell it to get busy. At this point, you create one or more channels for data exchange. This process is called "tuning"; once you've tuned the session, you start using the data exchange channels to do "the useful work".

The first channel that's successfully started has a trick associated with it: when you ask to start the channel, you're allowed to specify a "service name" that goes with it. This allows a server with multiple configurations to select one based on the client's suggestion. (A useful analogy is HTTP 1.1's "Host:" header.) If the server accepts the "service name", then this configuration is used for the rest of the session.

To allow parallelism, BXXP allows you to use multiple channels simultaneously. Each channel processes messages serially, but there are no constraints on the processing order for different channels. So, in a multi-threaded implementation, each channel maps to its own thread.

This is the most general case, of course. For one reason or another, an implementor may not be able to support this. So, BXXP allows for both positive and negative replies when a message is sent. So, if you want the classic client/server model, the client program should simply reject any new message sent by the server. This effectively throttles any asynchronous messages from the server.

Of course, we now need to provide mechanisms for segmentation and flow control. For the former, we just put a "continuation" or "more to come" flag in the command line for the frame. For the latter, we introduced the notion of a "transport mapping".

What this means is that BXXP doesn't directly define how it sits of top of TCP. Instead, it lists a bunch of requirements for how a transport service needs to support a BXXP session. Then, in a separate document, we defined how you can use TCP to meet these requirements.

This second document pretty much says "use TCP directly", except that it introduces a flow control mechanism for multiplexing channels over a single TCP connection. The mechanism we use is the same one used

by TCP (sequence numbers and a sliding window). It's proven, and can be trivially implemented by a minimal implementation of BXXP.

The introduction of flow control is a burden from an implementation perspective -- although TCP's mechanism is conceptually simple, an implementor must take great care. For example, issues such as priorities, queue management, and the like should be addressed. Regardless, we feel that the benefits of allowing parallelism for intra-application streams is worth it. (Besides, our belief is that few application implementors will actually code the BXXP framework directly -- rather, we expect them to use third-party packages that implement BXXP.)

5.4 Authentication

We use SASL. If you successfully authenticate using a channel, then there is a single user identity for each peer on that session (i.e., authentication is per-session, not per-channel). This design decision mandates that each session correspond to a single user regardless of how many channels are open on that session. One reason why this is important is that it allows service provisioning, such as quality of service (e.g., as in [34]) to be done on a per-user granularity.

5.5 Privacy

We use SASL and TLS. If you successfully complete a transport security negotiation using a channel, then all traffic on that session is secured (i.e., confidentiality is per-session, not per-channel, just like authentication).

We defined a BXXP profile that's used to start the TLS engine.

5.6 Things We Left Out

We purposefully excluded two things that are common to most application protocols: naming and authorization.

Naming was excluded from the framework because, outside of URIs, there isn't a commonly accepted framework for naming things. To our view, this remains a domain-specific problem for each application protocol. Maybe URIs are appropriate in the context of a particularly problem domain, maybe not. So, when an application protocol designer defines their own profile to do "the useful work", they'll have to deal with naming issues themselves. BXXP provides a mechanism for identifying profiles and binding them to channels. It's up to you to define the profile and use the channel.

Similarly, authorization was explicitly excluded from the framework. Every approach to authorization we've seen uses names to identify principals (i.e., targets and subjects), so if a framework doesn't include naming, it can't very well include authorization.

Of course, application protocols do have to deal with naming and authorization -- those are two of the issues addressed by the applications protocol designer when defining a profile for use with BXXP.

5.7 From Framework to Protocol

So, how do you go about using BXXP? To begin, call it "BEEP", not "BXXP" (we'll explain why momentarily).

First, get the BEEP core specification [35] and read it. Next, define your own profile. Finally, get one of the open source SDKs (in C, Java, or Tcl) and start coding.

The BEEP specification defines several profiles itself: a channel management profile, a family of profiles for SASL, and a transport security profile. In addition, there's a second specification [36] that explains how a BEEP session maps onto a single TCP connection.

For a complete example of an application protocol defined using BEEP, look at reliable syslog [37]. This document exemplifies the formula:

```
application protocol = BEEP + 1 or more profiles
                     + authorization policies
                     + provisioning rules (e.g., use of SRV RRs [38])
```

6. BXXP is now BEEP

We started work on BXXP in the fall of 1998. The IETF formed a working group on BXXP in the summer of 2000. Although the working group made some enhancements to BXXP, three are the most notable:

- The payload default is "application/octet-stream". This is primarily for wire-efficiency -- if you care about wire-efficiency, then you probably wouldn't be using "text/xml"...
- One-to-many exchanges are supported (the client sends one message and the server sends back many replies).
- BXXP is now called BEEP (more comic possibilities).

7. Security Considerations

Consult Section [35]'s Section 8 for a discussion of BEEP-related security issues.

References

[1] Postel, J., "Simple Mail Transfer Protocol", STD 10, RFC 821, August 1982.

[2] Postel, J. and J. Reynolds, "File Transfer Protocol", STD 9, RFC 959, October 1985.

[3] Berners-Lee, T., Fielding, R. and H. Nielsen, "Hypertext Transfer Protocol -- HTTP/1.0", RFC 1945, May 1996.

[4] Herriot, R., "Internet Printing Protocol/1.0: Encoding and Transport", RFC 2565, April 1999.

[5] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies", RFC 2045, November 1996.

[6] Fielding, R., Gettys, J., Mogul, J., Nielsen, H., Masinter, L., Leach, P. and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.

[7] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981.

[8] Mockapetris, P., "Domain names - concepts and facilities", STD 13, RFC 1034, November 1987.

[9] Microsystems, Sun., "NFS: Network File System Protocol specification", RFC 1094, March 1989.

[10] Srisuresh, P. and M. Holdrege, "IP Network Address Translator (NAT) Terminology and Considerations", RFC 2663, August 1999.

[11] Crocker, D., "Standard for the format of ARPA Internet text messages", STD 11, RFC 822, August 1982.

[12] Berners-Lee, T. and D. Connolly, "Hypertext Markup Language - 2.0", RFC 1866, November 1995.

[13] Freed, N., "SMTP Service Extension for Returning Enhanced Error Codes", RFC 2034, October 1996.

[14] Myers, J., "IMAP4 Authentication Mechanisms", RFC 1731, December 1994.

[15] Freed, N., "SMTP Service Extension for Command Pipelining", RFC 2197, September 1997.

[16] Rescorla, E. and A. Schiffman, "The Secure HyperText Transfer Protocol", RFC 2660, August 1999.

[17] Myers, J., "Simple Authentication and Security Layer (SASL)", RFC 2222, October 1997.

[18] Newman, C., "The One-Time-Password SASL Mechanism", RFC 2444, October 1998.

[19] Myers, J., "SMTP Service Extension for Authentication", RFC 2554, March 1999.

[20] Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A. and L. Stewart, "HTTP Authentication: Basic and Digest Access Authentication", RFC 2617, June 1999.

[21] Kent, S. and R. Atkinson, "Security Architecture for the Internet Protocol", RFC 2401, November 1998.

[22] Dierks, T. and C. Allen, "The TLS Protocol Version 1.0", RFC 2246, January 1999.

[23] Newman, C. and J. Myers, "ACAP -- Application Configuration Access Protocol", RFC 2244, November 1997.

[24] Hoffman, P., "SMTP Service Extension for Secure SMTP over TLS", RFC 2487, January 1999.

[25] Newman, C., "Using TLS with IMAP, POP3 and ACAP", RFC 2595, June 1999.

[26] Myers, J. and M. Rose, "Post Office Protocol - Version 3", STD 53, RFC 1939, May 1996.

[27] Braden, B., Clark, D., Crowcroft, J., Davie, B., Deering, S., Estrin, D., Floyd, S., Jacobson, V., Minshall, G., Partridge, C., Peterson, L., Ramakrishnan, K., Shenker, S., Wroclawski, J. and L. Zhang, "Recommendations on Queue Management and Congestion Avoidance in the Internet", RFC 2309, April 1998.

[28] Touch, J., "TCP Control Block Interdependence", RFC 2140, April 1997.

[29] Postel, J. and J. Reynolds, "Telnet Protocol Specification", STD 8, RFC 854, May 1983.

[30] World Wide Web Consortium, "Extensible Markup Language (XML) 1.0", W3C XML, February 1998, <http://www.w3.org/TR/1998/REC-xml-19980210>.

[31] Case, J., Fedor, M., Schoffstall, M. and C. Davin, "Simple Network Management Protocol (SNMP)", STD 15, RFC 1157, May 1990.

[32] World Wide Web Consortium, "SMUX Protocol Specification", Working Draft, July 1998, <http://www.w3.org/TR/1998/WD-mux-19980710>.

[33] Berners-Lee, T., Fielding, R. and L. Masinter, "Uniform Resource Identifiers (URI): Generic Syntax", RFC 2396, August 1998.

[34] Waitzman, D., "IP over Avian Carriers with Quality of Service", RFC 2549, April 1999.

[35] Rose, M., "The Blocks Extensible Exchange Protocol Core", RFC 3080, March 2001.

[36] Rose, M., "Mapping the BEEP Core onto TCP", RFC 3081, March 2001.

[37] New, D. and M. Rose, "Reliable Delivery for syslog", RFC 3195, November 2001.

[38] Gulbrandsen, A., Vixie, P. and L. Esibov, "A DNS RR for specifying the location of services (DNS SRV)", RFC 2782, February 2000.

[39] <http://mappa.mundi.net/cartography/Wheel/>

Author's Address

Marshall T. Rose
Dover Beach Consulting, Inc.
POB 255268
Sacramento, CA 95865-5268
US

Phone: +1 916 483 8878
EMail: mrose@dbc.mtview.ca.us

Full Copyright Statement

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.