Typical Internet Applications

A typical Internet application is structured as a distributed client-server system. Servers wait for clients to connect. A client – under the direction of some external user – connects to a specific server and sends a request or command to the server. The server processes the request, and sends a response to the client.

In some protocols, this completes the interaction and the connection would be closed following the message exchange. In others, a series of request-response message pairs are exchanged before the connection is closed. [insert diagram of client-server loop] In either case, the application is built on the sending and receiving specific messages. As a point of terminology, these messages may be called commands, requests, responses, or Protocol Data Units (PDUs). Instead of settling on a cumbersome standard name, we'll interchange these terms to best fit the context.

Motivation for Using TCP

What services would a client-server application like the network to provide? Here are some key requirements:

- Reliability: Assuming all the messages are important a client-server application would like the network to reliably deliver all the messages it sends.

- Flow control: We need to ensure we don't send more data than a receiver can handle at one time.

- Congestion control: a client-server application would like the network to share its capacity fairly so that all applications experience adequate performance on the shared
network.

- **Message packetization:** Since a sender may send multiple messages on the same TCP connection, a client-server application would like the network to clearly identify the ends of messages to the recipient.

TCP is a reliable, stream-oriented transport protocol that provides congestion control. It fulfills all the requirements above – except “message packetization.” As a stream-oriented protocol, TCP simply enables a sender to write an unbounded stream of bytes to a receiver. It doesn't provide any support for breaking up the stream of bytes into discrete messages. As this chapter will show, however, a network application can easily provide this functionality itself.

## How to read TCP data

### Binary Bytes or Characters?

Internet applications categorize almost all message content as either binary or text. Binary content stores information in all 8 bits of each byte, and typically represents arbitrary data, such as images, numerical values and compressed data. Traditionally, text content has followed the ASCII standard, storing information in only the lower 7 bits of each byte, and typically represents textual information such as message headers, protocol commands, email messages and HTML pages.

<table>
<thead>
<tr>
<th>H</th>
<th>T</th>
<th>M</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

*Mapping of characters to bits using the ASCII format.*

As computer usage has become increasingly internationalized, multiple bytes are now widely used to represent characters in both English and other languages, primarily using the Unicode standard. We will discuss Unicode further in Section [xref here]. (Note that binary data is sometimes encoded as text, for example, by MIME [xref here].)

### Delimiting Messages

Internet application programmers want to process application layer messages. However, as presented above, TCP transports streams of bytes. That is, TCP does not provide message delimiters – thus an Internet application must delimit messages itself. In fact, a lot of the content in RFCs focuses on how to delimit application layer messages. Internet applications employ three primary methods to delimit messages:

- **Boundary:** The sender appends a special byte sequence to the end of every message. The receiver searches for the byte sequence in the data, and ends the message when it is found. Some protocols use a standard sequence to end all messages. For example,
the email transport protocol SMTP employs “CRLF.CRLF”. In other protocols the
sender creates a sequence that it sends at the beginning of the message.

A protocol also must define a mechanism that expands the special sequence into
something else when it appears in raw data, so that a message is not terminated
prematurely. This mechanism is called \textit{byte stuffing}. A sender preforms the expansion,
and a receiver reverses the process.

We note that we obtained the name \textit{boundary} from the Multipurpose Internet Mail
Extensions (MIME) standard, RFC 2045, which calls its section separator a boundary.

- \textit{Byte count}: A sender precedes data with the number of bytes in the message. The
receiver reads the byte count and then reads the number of bytes indicated.

- \textit{End-of-file}: A sender closes the TCP connection after it sends a message. The receiver
reads the data until it reads the special EOF byte inserted by its local TCP stack. (In a
sense, in this case TCP does delimit the message.)

We illustrate these with examples.

- \textit{Boundary}: Many protocols – including SMTP, HTTP, IMAP, and POP3 – terminate
header lines with the byte sequence CRLF [does a space belong, does this need more
explanation i.e., \texttt{\textbackslash r \textbackslash n} or byte values?].

- \textit{Byte count}: HTTP 1.0 and later employ a 'Content-Length' header that indicates the
number of bytes in the body of some HTTP messages.

- \textit{End-of-file}: By contrast, HTTP 0.9 (which was never standardized in an RFC)
delimited the end of the body of HTTP Response messages by closing the connection.
HTTP 1.0 maintains this functionality for backward compatibility with old clients.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|l|}
\hline
\textbf{Delimiting strategy} & \textbf{Representation} & \textbf{Binary} \\
\hline
\textbf{Boundary} & Header line / CRLF & None that we can think of. \\
& SMTP message body / CRLF.CRLF & \\
& MIME encapsulation / string created dynamically, and sent at the start of the message & \\
\hline
\textbf{Byte count} & HTTP body Content-Length & HTTP body Content-Length \\
\hline
\textbf{End-of-file} & HTTP 0.9 & FTP stream data \\
\hline
\end{tabular}
\end{table}

Some examples of application protocol strategies for delimiting messages.

A single Internet protocol often utilizes more than one of these techniques to delimit
different kinds of messages, depending on the type of data being transferred. For example, FTP terminates commands with CRLF, but opens a separate connection for transferring data, and indicates the end of the data by closing the connection.

Each of these message delimiting techniques offers certain advantages and drawbacks. Let's consider each one individually.

**Boundary Sequence**

A protocol identifies a special byte sequence that marks the end of a message. Each call to read() will return an array of bytes that may include the special boundary character. The reader must scan through this array after every read to look for the boundary. This requires computational effort linear in the size of the message—a comparison must be done approximately once on every byte which is received. Because the receiver cannot predict in advance exactly how many times it must call read() before it can parse a complete message, it must take care in how memory is allocated for reading a temporary message. A naïve implementation might allocate a new array for every call to read(), and collect these arrays until they can be parsed. Typically, protocols specify a reasonable maximum message length, so that a memory-efficient structure such as a ring buffer [xref] can be used. A further limitation of using a special byte sequence to delimit messages is that the sender must take care never to accidentally send the boundary sequence because it happens to appear in the data of a message. Protocols typically employ a method known as byte stuffing to protect the data from being misinterpreted. If the boundary sequence happens to occur in a message, the sender will replace the bytes with an escape sequence that clearly identifies them to the receiver as data instead of the end of message marker. The boundary sequence may be agreed upon in advance, as in SMTP, or it may be specified during each session, potentially to minimize the amount of byte stuffing that needs to occur, as in MIME.

**Byte Count**

This method is desirable when maximum speed is needed. It avoids a sender's need to examine data for performing byte stuffing, and it avoids a receiver's need to scan data to unstuff bytes.

A potential drawback of this method is that it can be prone to errors in computing the count. Complications such as unexpected multi-byte characters in the data, or assembling data from multiple sources can make programmer errors more likely. Some protocols [example?] accompany the use of byte counts with a timeout, to prevent receivers from waiting forever if the sender has mistakenly sent fewer bytes than they initially declared.

One drawback of the byte count method arises when a message is being created dynamically in that the beginning of the message cannot be transmitted until the entire message has been created and its length has been counted. For example, suppose a
message is generated by a program which doesn't know the size of the message until it finishes creating it. The bigger the message, the more serious this drawback. Look at problem [xref] to see HTTP's clever approach (in Section of ) to dealing with this issue.

**End-of-file**

A protocol that needs to send only one message over a connection can simply close the connection when a complete message has been sent. The TCP stack on the receiver's side reports this condition as an end-of-file when reading from the stream. This can be a simple but effective method for indicating the end of a message. However, one drawback of this approach is that a connection may also be closed by unrelated network or system problems. Unless there is some other way of determining the size of a message, a receiver cannot properly determine whether it receives an entire message in this situation.

In practice, protocols employ this technique because it is extremely simple, and the message itself may not contain any information that could be used to verify its integrity. The receiver is left with a message that may have been truncated. If the message is a command to a server, the server should still execute it, although it may not be what the client intended. Here, we see an example of the Internet's “Best Effort” principles in action—if you don't get the response you expect, just reconnect and try again!

**InputStreams**

In the rest of this chapter we develop designs and implementations for parsing messages from TCP.

At the lowest level Java sockets pass TCP data to a receiver in an InputStream. The receiver can read directly from the InputStream or from some of its subclasses that provide additional functionality. In this section we design and develop code that uses InputStreams and its subclasses to parse messages from TCP streams.

Let's begin with some crucial observations about TCP[this list may be too much for a reader to digest at once]:

1. TCP transports 8-bit bytes (called OCTETs by the RFCs). Consequently, a byte array often provides the most convenient representation of TCP data after it has been read.

2. A TCP connection can close at any time. Closes happen for many reasons. The network application at the other end may deliberately close the connection or it may crash which causes its OS to close the connection, or the local host may lose network connectivity which causes its OS to close the connection. Thus a read() operation that obtains data from TCP must always be prepared to handle an IOException or an end-of-file (EOF) indication (typically a -1).

3. Some parts of a message may be binary, while other parts may be just text (more precisely, as section 2.2 of RFC 2822 says, header fields contain characters in the US-
ASCII range of 1 through 127).

4. TCP can break up the byte stream at any point. (The breaks may be used for TCP segmentation or for buffering in the TCP/IP stack.) In general, a read() operation can return any prefix of the bytes that have already been sent (but not already read, of course) on the TCP connection. Stated more simply, the last byte in a read() operation can be any byte in the stream. In particular, the last byte in a read() operation may not be the last byte of any of the write() operations that sent the data.

Converting these observations into requirements for the code that reads TCP data, we obtain:

- read bytes from an InputStream or one of its subclasses
- handle an IOException or an EOF at any time
- read the right things, that is, binary bytes when binary arrives, and characters when text arrives
- make certain that enough data has been read before trying to parse structures
- there are two cases, with these additional requirements, when reading headers:
  1. reading headers
  2. obtain lines terminated by CRLF
  3. reject header fields containing the values 0 or in the range 128 to 255

**Reading lines**

Many Internet protocols – including HTTP, SMTP, IMAP, POP3, NNTP, etc. – send some data structured as lines of text. Often they use the definition of a line provided in Section 2.1 of RFC 2822, *Internet Message Format*:

Messages are divided into lines of characters. A line is a series of characters that is delimited with the two characters carriage-return and line-feed; that is, the carriage return (CR) character (ASCII value 13) followed immediately by the line feed (LF) character (ASCII value 10). (The carriage-return/line-feed pair is usually written in this document as "CRLF".)

**readLine()**

A Java method called 'readLine()' sounds like an ideal library method for reading lines. However, java.io.DataInputStream.readLine() and java.io.BufferedReader.readLine() do not work reliably because readLine() returns lines terminated by CR, LF or CRLF. (In fact, they fail so subtly that we think the Java API documentation should contain the warning “This method should not be used to read from TCP.”) While they will work...
when TCP's timing is lucky [i want to phrase this better], they can fail in several ways:

1. If the data is binary any pattern can arrive, including CR, LF or CRLF. Since 
   readLine() does not indicate what terminated the line, it is impossible to 
   reconstruct the data sent by TCP. That is, the code using readLine() cannot know 
   how to reconstruct the line that readLine() read. Which character(s) should it 
   append to the line readLine() returns? [move this 'graph, since it does not address 
   the topic of reading lines. Where does it go?]

2. Suppose we try to use readLine() to read header lines. Shouldn't that work because 
   each header line is terminated by CRLF? Actually no. Two problems can arise:
   a. If the header is terminated by just CR or just LF it would be nice to catch 
      and, perhaps, report the error, but readLine() makes this impossible.
   b. Suppose the sender terminates lines with CRLF. Then, timing can cause 
      the following behavior: [it would be nice to illustrate this]
      i. TCP buffers a header line terminated by CR; readLine() returns the 
         line.
      ii. TCP now buffers some text starting with LF; readLine() returns a 
          blank line that it should not return!

For example, this code does not work properly:

```java
import java.io.InputStream;

/**
 * Example of reading incorrectly with readLine().
 */

public class BuggyServer implements Runnable {
    private Socket s;
    private BufferedReader in;
    private BufferedWriter out;

    BuggyServer(Socket s) {
        /*
         * Warning: constructing the InputStreamReader with a "US-ASCII"
         * character set (charset) will convert bytes to characters 
         * correctly for many Internet protocols, but may NOT convert 
         * correctly for some protocols. See [xref here] for more details.
         */
        in = new BufferedReader(
            new InputStreamReader(s.getInputStream(), "US-ASCII"));
        out = new BufferedWriter(
            new OutputStreamWriter(s.getOutputStream(), "US-ASCII"));
    }

    public void run() {
        try {
            while (true) {
                /*
                 * Warning: bad bug: readLine() may not terminate lines
                 * properly. See the text for more details.
                 */
```
String request = in.readLine();
if (request == null)
    break; // EOF was read - nothing else to do!
String response = process(request);
out.write(response);
out.flush();
}
s.close();
}
catch (IOException e) {
    System.err.println("IOException on socket: " + e.getMessage());
}
catch (InterruptedException e) {
    System.err.println("InterruptedException caught.");
}
}

private String process (String request) {
    /*
     * Stub method. Need to parse the request, determine if it is
     * valid, execute it, and generate a response.
     * We'll return an obviously bad response, so we don't assume this
     * method is working correctly.
     *
     * return "STUB RESPONSE";
     */
}

The above example is a minimal handler class that could be used by a simple TCP server. The code contained in the `public void run()` method would be executed in a new thread, after the main server accepts a connection from a client and starts up a handler like this one. The handler simply takes the following actions:

1. Read from the socket, until CR, LR, or CRLF is detected.
2. If the line returned indicates that the connection should be closed, quit.
3. Otherwise, process the request and generate a response.
4. Send the response to the client.
5. Repeat

As this overview should make clear, this code simply doesn't do the right thing, if messages are supposed to be delimited by the CRLF sequence.

What we really need is a replacement for `readLine()`. Suppose a server follows exactly the same sequence of actions, except that line 1 read as follows:

1. Read from the socket, until CRLF is detected.

The following code is a replacement for the `readLine` facility in the `BufferedReader` class. Like `BufferedReader`, you construct it by giving it an `InputStream` to wrap. You can then read lines delimited by CRLF by calling `readALine()`, instead of `readLine()`. It differs
from BufferedReader in that it does not bridge from bytes to characters, but returns an array of bytes. The application is free to convert the bytes into characters as desired (An easy way is to use the String(byte[] bytes, String charsetName) constructor: http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html#String(byte[],%20java.lang.String) [xref].)

The example below, including definition of the exceptions it throws and its unit tests are available at our web site.

```java
import java.io.BufferedInputStream;
import java.io.EOFException;
import java.io.IOException;
import java.io.InputStream;

/**
 * Provides support for reading RFC 2822 header lines from an InputStream,
 * although does not support 'folded' lines.
 * Typically, the InputStream will be connected to a socket.
 *
 * Created on Mar 22, 2005
 * TODO What's the JavaDoc operator, if it exists, for 'Created'? It doesn't
 * exist, unless you create or find a custom tag
 * @author Arthur.Goldberg
 */
public class ReadFromInputStream {
    private BufferedInputStream bin;

    ReadFromInputStream(InputStream theInputStream) {
        bin = new BufferedInputStream(theInputStream);
    }

    /**
     * Reads a line terminated by CRLF, CR or LF. Does not return the
     * terminator.
     * If the line is terminated by CRLF, return the line. If the line is
     * terminated
     * by CR or LF alone, throws an exception containing the line.
     * Performance consideration: Every call to this method allocates a new
     * buffer of size
     * Constants.maxLineLength, and copies at most that many bytes at a time
     * into it.
     * @return the line, without the terminating CRLF
     * @throws IOException
     * when the inputStream throws IOException
     * @throws IllegalCharacterException
     * when the inputStream contains a 0 or a byte in the range 128 to 255
     * @throws EOFException
     * when the inputStream closes
     * @throws OversizedLineException
     * when a line contains more than Constants.maxLineLength; also
     */
```
* discards the bytes up through the byte that did not fit in
* the buffer
* TODO: decide whether this is the best policy for oversized lines
*/

`byte[] readALine() throws IOException, IllegalCharacterException,
EOFException, OversizedLineException, LineTerminationException {`
    `int numBytes = 0;`
    `byte[] DataBuffer = new byte[Constants.maxLineLength];`
    `boolean foundCR = false;`
    `int b;`

    while (true) {
        `b = bin.read();`
        `if (Constants.EOF == b) {`
            `throw new EOFException();`
        }`
        `if (illegalByte(b)) {`
            `throw new IllegalCharacterException((byte) b);`
        }`
        `if (DataBuffer.length <= numBytes) {`
            `throw new OversizedLineException();`
        }`
        `if (foundCR) {`
            `if (b == Constants.LF) {`
                `return (copyDataBuffer(DataBuffer, numBytes));`
            } else {`
                // found CR but no LF: an error
                // 'push back' the byte following the CR -- b -- into bin
                // so that it is read on the next access.
                `bin.reset();`
                // return the line
                `throw new LineTerminationException(copyDataBuffer(
                    DataBuffer, numBytes));`
            }
        }`

        `if (b == Constants.CR) {`
            `foundCR = true;`
            // if LF is not the next char, we want to be able to
            // reread the next char on the next call to readALine()
            `bin.mark(1);`
            `continue;`
        }

        // found LF but no CR: an error
        `if (b == Constants.LF) {`
            // return the line
            `throw new LineTerminationException(copyDataBuffer(DataBuffer,
                numBytes));`
        }`

        `DataBuffer[numBytes++] = (byte) b;`
    }
}

/**
* copies the DataBuffer into a new byte array
* *
* @param DataBuffer
* the array of bytes to copy
* @param numBytes
*/
* the number of bytes in the array to copy
* @return copy of the DataBuffer
*/
static final private byte[] copyDataBuffer(byte[] DataBuffer, int numBytes) {
    byte[] lineBuffer = new byte[numBytes];
    // TODO: perhaps this copying could be avoided by clever use
    // of the BufferedInputStream
    System.arraycopy(DataBuffer, 0, lineBuffer, 0, numBytes);
    return lineBuffer;
}

/**
* return true if the value of the byte c is illegal
* @return  boolean true if the value of the byte c is illegal
*/
static final private boolean illegalByte(int c) {
    if ((c < Constants.FirstLegalChar) || (Constants.LastLegalChar < c)) {
        return true;
    }
    return false;
}

[reading a body; how is the end determined? I'm not sure what's meant by 'body' here.]

Reading binary data

Protocols which transfer binary data require some interpretation of the bits other than simply translating them into characters. For example, many protocols will define a header for each message consisting of binary data. By referring to the protocol's specification, implementors can understand how to interpret the bits they receive. We'll demonstrate with an example using the Domain Name System (DNS) protocol [xref RFC 1035?].

Diagrams like the following are used to indicate the content of binary data:

```
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| 1 | 2 |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| 3 | 4 |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| 5 | 6 |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

This diagram identifies six octets; that is, six eight-bit bytes that would be transferred over the internet in the order that they are numbered. The two lines at the top of the diagram indicate the bit position within each pair of octets. They're numbered from bit 00 to bit 15. The lowest common denominator on almost all machines these days is an eight-bit or some multiple of eight-bit word. Thus, binary protocols typically are defined in terms of eight-bit words, called octets in the standard language of RFCs.

The interpretation of the octets varies depending on the protocol. There is no particular
assumption we can make about how to group and transform these bits into numbers, characters, and command. We'll take a look at the binary header format used in DNS. It has the following structure:

```
+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+
|                      ID                       |
+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+
|QR|   Opcode  |AA|TC|RD|RA|   Z    |   RCODE   |
+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+
|                    QDCOUNT                    |
+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+
|                    ANCOUNT                    |
+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+
|                    NSCOUNT                    |
+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+
|                    ARCOUNT                    |
+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+-----------+
```

This specifies a 12-octet header, consisting of 13 fields. The fields can be classified into the following types:

- flags
- numbers
- opaque data
- fixed

A flag is binary field that can be on or off. In this example, QR, AA, TC, RD, and RA are flags. They specify conditions about the following message, such as whether it is a query or a response (QR).

A number represents a numeric value. In this example, Opcode, RCODE, QDCOUNT, ANCOUNT, NSCOUNT, and ARCOUNT are all numbers. There are many different numerical encodings a protocol might specify: ones complement, twos complement, IEEE 754 floating-point. However, almost all IETF protocols use a big-endian format to encode unsigned integers. That is, the left-most bit is the high-order or most significant bit. When a number spans multiple octets, the left-most bit still represents the most significant bit, and is transmitted first. Here's a diagram of the integer 170, encoded as an unsigned big-endian integer in a single octet:

```
0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+-+-
|1 0 1 0 1 0 1 0|
+-+-+-+-+-+-+-+-+-+-
```

Here's an example of the integer 256, which requires two octets:

```
0 1
0 1 0 1 0 1 0 1
```
Some data, such as the ID field in the DNS header, is opaque. That is, the protocol does not specify any interpretation at all for the meaning of these bits. In the case of DNS, a client constructing a query generates some 16-bit value that will help it identify each query uniquely. A server responding to a query is expected to exactly copy those bits back to a response it sends to a client. The server has no way to know what the ID bits represent.

Finally, some fields contain fixed data as specified by the protocol. The Z field is reserved for future use, and must contain all bits zero in queries and responses.

[present brief overview of DNS?]
[talk about 'endian'ness.][example here probably, see Section 4.1 of the TLS Protocol RFC, 2246]
[java example of reading a DNS packet?]

**Reading mixed data**

In some protocols, it is necessary to switch between the reading of text data and binary data coming from the same stream. For example, the IMAP protocol uses text-only commands, delimited by CRLF for features such as logging in and selecting a particular mailbox, but allows mail messages to be transferred in binary mode, using a byte count to indicate when the reader should switch back to text mode. For example, here's an interchange between an IMAP client and server in the middle of a connected session:

```
C:    a004 fetch 12 body[header]
S:    * 12 FETCH (BODY[HEADER] {342}
S:    Date: Wed, 17 Jul 1996 02:23:25 -0700 (PDT)
S:    From: Terry Gray <gray@cac.washington.edu>
S:    Subject: IMAP4rev1 WG mtg summary and minutes
S:    To: imap@cac.washington.edu
S:    cc: minutes@CNRI.Reston.VA.US, John Klensin <KLENSIN@MIT.EDU>
S:    Message-Id: <B27397-0100000@cac.washington.edu>
S:    MIME-Version: 1.0
S:    Content-Type: TEXT/PLAIN; CHARSET=US-ASCII
S:    )
S:    a004 OK FETCH completed
```
The client wants to read the header of message 12 in the currently selected mailbox. Each request is tagged with a client-specified ID, as in DNS. In the above example, the ID is the string “a004,” which is copied back into the server's ultimate response in the last line. After sending a header response that identifies the following data as the header of message 12, the server sends the string “{342}CRLF,” indicating that exactly 342 binary bytes of data will follow. The client is expected to read exactly 342 bytes without interpreting them as IMAP control data, but the 343rd byte will be part of a regular IMAP server response. In this example, it begins the server's OK response, indicating that the FETCH was successfully executed.

We can readily extend a stream wrapper class such as ReadFromInputStream to allow for the reading of mixed data.

[Need to mention that due to the use of a BufferedInputStream in the ReadFromInputStream class, we must not try to read from the InputStream using any other method, or we risk missing bytes that have been buffered by the BufferedInputStream. Alternate method: ring buffers—avoids allocation performance issue.]

[example here probably]

Homework

1. Write a small client and a small server that sometimes demonstrates the observation in the text that “the last byte in read() operations may not be the last byte of the write() operations that sent the data”. Knowing the behavior of TCP, what tricks can you use to create this behavior? Hints: consider very long and short writes, and consider delaying some reads.

2. The boundary method employs a special sequence to mark the end of a message. If the sequence appears in raw data then byte stuffing expands the sequence into a special expanded sequence. A receiver would then reverse the process so the data would not be changed. Wouldn't it be better to just change the sequence, rather than expand it so the sender does not need to transmit more data? Is this possible? Also what happens if the expanded sequence appears in the raw data? Doesn't the receiver change it back too and mess up the data?

3. Write a method that unfolds long headers as described in Sections 2.2.3 of RFC 2822. Assume that the method's input is a byte array that contains a sequence of headers, some of which may be long headers. If a long header contains syntactic errors then the method should throw an exception. Otherwise, the method should return an array of byte arrays, each of which contains an unfolded header.

4. HTTP 1.1 employs a clever technique called 'chunking' for identifying the length of a message body when the sending application does not know the size of the body before the application begins transmitting the message. The sender breaks the body into
chunks. Each chunk contains a positive length in the first line immediately followed by that number of bytes of data. The last chunk is 0 bytes long. [a small example might be nice] Sections 3.6 and 19.4.6 of the HTTP 1.1 specification (RFC 2616) describe chunking. Write a module that parses chunked message bodies. (To simplify the problem, assume that the elements chunk-extension and trailer are empty.) The module will be given an InputStream containing TCP data which starts with the first byte of the message body. Return the entire message body in a byte array. Throw appropriate exceptions. [maybe could be 2 Hws, one on RFC, other writing code.]

Chapter Glossary