**Message Framing**

First, one must understand the abstraction of TCP/IP. From the application’s perspective, TCP operates on streams of data, never packets. Repeat this mantra three times: “TCP does not operate on packets of data. TCP operates on streams of data.”

There is no way to send a packet of data over TCP; that function call does not exist. Rather, there are two streams in a TCP connection: an incoming stream and an outgoing stream. One may read from the incoming stream by calling a “receive” method, and one may write to the outgoing stream by calling a “send” method. If one side calls “send” to send 5 bytes, and then calls “send” to send 5 more bytes, then there are 10 bytes that are placed in the outgoing stream. The receiving side may decide to read them one at a time from its receiving stream if it so wishes (calling “receive” 10 times), or it may wait for all 10 bytes to arrive and then read them all at once with a single call to “receive”.

Sending data to the TCP stream is rather easy; all one has to do is call “send”, and the appropriate bytes are queued to the outgoing stream. Receiving data from the TCP stream is a bit more tricky, because the “receive N bytes” operation will wait until at least one byte and at most N bytes arrive on the incoming stream before it returns. Note that the “receive N bytes” operation will complete even if it doesn’t read all N bytes, giving the application a chance to act on partial data while the rest of the data bytes are in transit. In the real world, very few programs can process partial receives; almost all programs need a buffer to store partial receives until they have enough data to do meaningful work.

To repeat: TCP operates on streams, not on packets. However, most application protocols are based on the idea of “messages”; for example, a client may send a “Lookup X” message to the server, and the server will respond with an “X Data” or “X Not Found” message. Since TCP operates on streams, one must design a “message framing” protocol that will wrap the messages sent back and forth.

There are two approaches commonly used for message framing: length prefixing and delimiters.

Length prefixing prepends each message with the length of that message. The format (and length) of the length prefix must be explicitly stated; “4-byte signed little-endian” (i.e., “int” in C#) is a common choice. To send a message, the sending side first converts the message to a byte array and then sends the length of the byte array followed by the byte array itself.

Receiving a length-prefixed message is harder, because of the possibility of partial receives. First, one must read the length of the message into a buffer until the buffer is full (e.g., if using “4-byte signed little-endian”, this buffer is 4 bytes). Then one allocates a second buffer and reads the data into that buffer. When the second buffer is full, then a single message has arrived, and one goes back to reading the length of the next message.

Delimiters are more complex to get right. When sending, any delimiter characters in the data must be replaced, usually with an escaping function. The receiving code cannot predict the incoming message size, so it must append all received data onto the end of a receiving buffer, growing the buffer as necessary. When a delimiter is found, the receiving side can apply an unescaping function to the receiving buffer to get the message. If the messages will never contain delimiters, then one may skip the escaping/unescaping functions.

Whether using length-prefixing or delimiters, one must include code to prevent denial of service attacks. Length-prefixed readers can be given a huge message size; delimiting readers can be given a huge amount of data without delimiters. Either of these may result in an OutOfMemoryException, so one must include a maximum message size “sanity check” in the socket reading code.