Buffering Issues in TCP-Based Services

John Leis
leis@usq.edu.au
http://www.usq.edu.au/users/leis/

October 10, 2002

Faculty of Engineering & Surveying Technical Reports
ISSN 1446-1846

Report TR-2002-01
ISBN 1 877078 01 8

Faculty of Engineering & Surveying
University of Southern Queensland
Toowoomba Qld 4350 Australia
http://www.usq.edu.au/

Purpose
The Faculty of Engineering and Surveying Technical Reports serve as a mechanism for disseminating results from certain of its research and development activities. The scope of reports in this series includes (but is not restricted to): literature reviews, designs, analyses, scientific and technical findings, commissioned research, and descriptions of software and hardware produced by staff of the Faculty of Engineering and Surveying.

Limitations of Use
The Council of the University of Southern Queensland, its Faculty of Engineering and surveying, and the staff of the University of Southern Queensland: (1) do not make any warranty or representation, express or implied, with respect to the accuracy, completeness, or usefulness of the information contained in these reports; or (2) do not assume any liability with respect to the use of, or for damages resulting from the use of, any information, data, method or process described in these reports.
Abstract

Client-server protocols such as HTTP over a TCP transport layer may be implemented in a relatively straightforward manner, as is often given in “textbook” examples. Many of these examples do not scale well to a full implementation across anything but a localhost connection with minimal data payload. Furthermore, such implementations are invariably less than optimal in terms of performance, robustness to differing remote client configurations, processor time utilization, and bandwidth usage. It is suggested that the examples often given in texts are overly simplified, to the point of being misleading. In this note, we examine some of these issues, resulting from the implementation of HTTP clients and servers using Unix (Berkeley) sockets, Windows NT (Winsock), and Java. Issues addressed include efficient send/receive buffering to minimizing kernel and user data copying, and robustness to variability in record delimiters.

1 Introduction

Client-server data transmission over IP normally uses the services provided by the TCP layer. The standard API for accessing TCP services is the so-called “socket” library, which is universally available on Unix-based Operating Systems (where it is termed “Berkeley Sockets”) and Windows (termed “Winsock”). The type of client-server software which originally motivated the development of this report is typified by the HTTP protocol, which is a simple synchronous request-response protocol.

Reliable operation across varying underlying networks is a goal which hardly needs to be stated. This ideal must extend to varying network conditions, such as congestion and delay (which will not be known in advance), and ideally would extend to various system platforms (which will presumably be known in advance). Because of the interaction between the various protocol stack layers, network peers and intermediate routers, fundamental performance problems are often difficult to determine, and may even go unnoticed. The remainder of this report discusses some common problems in network coding, and suggests some workable solutions. Code fragments are presented to illustrate the solution methods. The listings of code fragments in this report omit checking code for system call return values in the interests of clarity and focusing on the solution outline. System call return values should always be checked for return values; this is particularly so in network programming, where myriad errors on the local client, the remote system, or the network in between could potentially occur.

2 Generic Design Goals

The “Robustness Principle” as enunciated by Braden [1] is as follows:

At every layer of the protocols, there is a general rule whose application can lead to enormous benefits in robustness and interoperability:

“Be liberal in what you accept, and conservative in what you send”

Software should be written to deal with every conceivable error, no matter how unlikely; sooner or later a packet will come in with that particular combination of errors and attributes, and unless the software is prepared, chaos can ensue.

Good software design principles dictate that we should strive for a solution which has the following attributes (see [2]):

**Portable** Must be able to run on Windows, Unix or other platforms, and ideally be easily ported to other languages (C or Java). The solution must run across differing protocol stacks (Berkeley, Winsock).
Robust. Must be robust to differing network peers. The client and server libraries must correctly parse all headers in all cases, and must work in all possible connection scenarios.

Reliable. The system must use timeouts as appropriate, and gracefully handle failures.

Efficient. The system must minimize the number of system calls, the amount of copying to/from kernel memory, and the number of TCP protocol writes generated.

Correct. The system should not have any side-effects, such as stray characters left in the TCP input buffer.

3 Fundamental Design Issues

Figure 1 illustrates, in simple terms, the overall data flows in a peer-to-peer transaction in terms of buffers present in the sender or receiver protocol stack, and in transit within the network itself. Since TCP is a stream-based protocol, any record blocking — such as the header and data illustrated in the figure — must be performed by the application layer. The data itself may then be split into fragments for transmission across the network. This is governed by various factors such as the Maximum Transmission Unit (MTU) of the underlying network(s), the Maximum Receive Unit (MRU) of the receiver, and the advertised TCP receive window at the time. The receiver process must then read the data via read or recv. This is not synchronous with the sending process, and must not be assumed to occur atomically. In addition, the receiver must be cognizant of delays caused by the network or multitasking at either end. It must assume that the transmission may break at any stage, after any amount of data has been received. The TCP layer will do its best to guarantee delivery, but “best-effort delivery” does not imply “perfect delivery” [3].

![Diagram of data flow](image)

Figure 1: Flow of data from the sending application, through the network, and back to the peer application. The data may be blocked into arbitrarily-sized data segments at any stage; the diagram does not intend to imply that any particular blocking will occur with any predictability.

4 The Send Mechanism

A naïve approach would be to generate the data and build the corresponding header, and queue each of these components separately to the TCP layer via send or write. This masks a potential underlying inefficiency: if the header is queued for transmission separately to the data portion via separate calls to send, the underlying transport mechanism may in fact send the header in a separate packet before the data is sent. This results in an additional packet being introduced into the network, with corresponding overhead. For example, a HTTP response header could be as little as 50 bytes or less. Considering that the TCP and IP headers typically account for 20 bytes each, the efficiency (data payload out of entire transmission) is rather poor. The problem is exacerbated by the Nagle algorithm, which delays sending small packets in order to reduce congestion on the network [4]. One erroneous suggestion often put forward is to simply disable the Nagle algorithm using the TCP_NODELAY option to setsockopt. This should be avoided, as it simply masks
the efficiency problem — by giving apparently faster write operations — by a greater amount of network traffic. In a lightly-loaded network this may not be a problem, but if scalability is desired (as it should be), then this obvious solution turns out to be a poor one.

Thus, it is desirable that the data to be sent (header plus content) is coalesced into one (or very few) send operations. A straightforward solution would be to have one header buffer where the header is composed, a second buffer where the data portion is composed, and an application-layer send buffer, to which both header and data are copied. This buffer is then passed to the sockets layer, where it is likely copied again from user space into kernel space. At best, this results in copying the entire contents of the data packet at least once in the application. This copy operation may be dispensed with, by writing the header and data sequentially into the send buffer. This requires maintaining a pointer to the next byte available in the send buffer. One potential argument against this is that since the length of the send data is written into the HTTP header by means of the Content-Length element, it is impossible to know where the end of the header will be until the packet data has been assembled. However, if the contents is (for example) a file, then the file size may be easily determined using the stat library function.

If it is impractical to read an entire file into a memory buffer at once, a sensible compromise has to be reached. For example, a 1 M file, even if read into a malloc'd buffer, would not be sent as a single TCP segment. On the other hand, disk reads are often arranged to be sector-sized chunks (for example, 1024 bytes) for efficiency reasons. A data chunk of this size may be passed to the socket-layer send function without loss of efficiency. Listing 1 illustrates this.

Listing 1: Calling the WriteMessageBlock function.

```c
stat(imgFileName, &statbuf);
filesize = statbuf.st_size;
nleft = filesize;

strcpy(sendbuf, "HTTP 200 OK\r\n");
strcat(sendbuf, "Content-Type: text/html\r\n");
sprintf(tmpbuf, "Content-Length: %d\r\n", filesize);
strcat(sendbuf, tmpbuf);
strcat(sendbuf, "\r\n"); // blank line terminator

hdr = strlen(sendbuf);

imgFile = fopen(imgFileName, "rb");

read = fread(&sendbuf[nhdr], 1, MAXSENDBYTES - hdr, imgFile);

if( nread > 0 )
{
    nleft -= nread;
    nwrite = nread + hdr;
    WriteMessageBlock(ConnectionSocket, sendbuf, nwrite, WRITE_TIMEOUT);
}

while( nleft > 0 )
{
    nread = fread(sendbuf, 1, MAXSENDBYTES, imgFile);
    if( nread > 0 )
    {
        nleft -= nread;
        nwrite = nread;
        WriteMessageBlock(Socket, sendbuf, nwrite, WRITE_TIMEOUT);
    }
    else
        return;
}
fclose(imgFile);
```
4 The Send Mechanism

The function `WriteMessageBlock` encapsulates the functions of:

1. Making the socket non-blocking (`fcntl` or `ioctlsocket`).
2. Waiting for the socket to be ready for writing (`select`).
3. Actually writing the data to the send queue (`send`).

An outline of the code for `WriteMessageBlock` for the Berkeley sockets API is shown in Listing 2. For Winsock, the `fcntl` call is changed to `ioctlsocket`, and the processing pertaining to interrupted system calls (`EINTR`) is removed. Further details on the latter may be found in [3].

Note that the send operation may block due to various reasons, but typically blocking occurs due to full send buffers. In that case, the process should block on `send`. Since the application should fail gracefully, the `send` is preceded by a call to `select` to set a timeout — `select` either returns when the socket is ready for writing, or the timeout period expires. The read and write selectors passed to `select` indicate readiness for reading or writing, with `select` returning the number of sockets which satisfy the condition.

It should also be noted that a successful `send` call does not mean that the data was successfully delivered to the remote peer, or even that the data was sent at all; it simply means that it was queued for transmission by the TCP/IP stack.

Listing 2: The `WriteMessageBlock` function.

```c
fcntl(fd, F_SETFL, O_NONBLOCK);

writeTimeout.tv_sec = timeout;
writeTimeout.tv_usec = 0;
FD_ZERO(&replyFDSet);
FD_SET(fd, &replyFDSet);

// write buffer, one send() at a time
nleft = buflen;
pbuf = buf;
while( nleft > 0 )
{
    rc = select(FD_SETSIZE, (fd_set *)&null, &replyFDSet, (fd_set *)&null, &writeTimeout);
    if( rc <= 0 )
        return -1;

    // write as many bytes as possible, up to the remaining buffer size
    nwritten = send(fd, pbuf, nleft, 0);
    if( nwritten < 0 ) // send() error
        { 
            if( errno == EINTR ) // interrupted system call
                continue;
        }
    nleft -= nwritten;
pbuf += nwritten;
}
```
5 The Receive Mechanism

As with the send process, the receive process should not assume that data will be available in any particular block size, or that it will arrive at all. The only guarantee is that if data is available, it will be error-checked (to the extent possible by transport and link layers), and will be in the correct byte-order. The function `ReadMessageBlock` shown in Listing 3 encapsulates the functions of:

1. Making the socket non-blocking (`fcntl` or `ioctl`).
2. Waiting for the socket to be ready for reading (`select`).
3. Actually reading the data from the receive queue, into the application buffer (`recv`).

Listing 3: The `ReadMessageBlock` function.

```c
rc = fcntl(fd, F_SETFL, O_NONBLOCK);

requestTimeout.tv_sec = timeout;
requestTimeout.tv_nsec = 0;
FD_ZERO(&requestFDSets);
FD_SET(fd, &requestFDSets);

nleft = nbytes;
pbuf = buf;
while( nleft > 0 )
{
    rc = select(FD_SETSIZE, &requestFDSets, (fd_set *)NULL, (fd_set *)NULL, &requestTimeout);
    if( rc <= 0 )
        return -1;

    // read as many bytes as are available, up to the remaining buffer size
    nread = recv(fd, pbuf, nleft, 0);

    if( nread < 0 )  // recv() error
        continue;

    // check for interrupted system call
    if( errno == EINTR )
        continue;

    return -1;
}

if( nread == 0 )  // read() at eof
{
    return nbytes - nleft;  // return actual number of bytes in buffer
}

nleft -= nread;
pbuf += nread;
}
return nbytes;
```

Note in particular the method of updating the buffer read positions. Because this function reads, for efficiency, a fixed-size (presumably large) buffer, provision must be made for the case where the sending
peer only sends a short message, or indeed the last remaining bytes (less than one buffer-full) of a larger message. In this case, the TCP layer sends a FIN packet, and after the appropriate handshaking, the socket layer recv function returns zero. Note that a return of zero does not imply that bytes delayed in transit could still be read — the select call blocks until either data is available, a timeout occurs, or the peer has closed the connection. Checking the return codes from select and recv ensures that these conditions are properly dealt with. From the Winsock man page:

For connection-oriented sockets, readability can also indicate that a close request has been received from the peer. If the virtual circuit was closed gracefully, then a recv will return immediately with zero bytes read. If the virtual circuit was reset, then a recv will complete immediately with an error code, such as WSAECONNRESET.

The multiple-read approach is documented in various sources (for example, [3, 5, 6]).

The next problem is the correct determination of the extent of the header, in protocols such as HTTP [7] where the header may be of variable length. The HTTP header is delineated by a “line termination” code. Some operating systems (such as Macintosh) use a carriage return (CR, hex 0D) to signify this; some systems (Unix) use a linefeed (LF, hex 0A), and some (Windows) use CR-LF. Although the standard specifies CR-LF for end of line, and a pair of CR-LF characters (that is, CR-LF-CR-LF) to terminate the header, good practice indicates that we should accept either CR-CR, LF-LF, or CR-LF-CR-LF as a valid header termination.

This problem, if not dealt with correctly, may manifest itself in various ways which are difficult to trace. Supposedly ready-to-use libraries may not handle this correctly. For example, in the Java environment, the readLine methods do not correctly terminate if the client gives an end-of-line terminator different to linefeed, or what it was expecting (see for example [8], pages 107-109). The methods outlined here are robust to any terminator sequence: CR, LF or CR-LF.

Failure to correctly deal with header termination may lead to:

1. A client or server which waits for a header-terminating sequence which it never receives (sent CR, expecting LF or vice-versa); or
2. A client or server prematurely terminating the header.

Simply checking for two CR’s is not enough: the first case will be missed, and the second case will result in one LF byte remaining in the buffer.

One possible, though suboptimal, approach is to use the flags argument to the recv call. The flag argument may be set to “peek” at the message:

```
  nread = recv(socket, pbuffer, 1, MSG_PEEK);
```

It is tempting to use the “peek method” like this, to check for header termination. However, this is very inefficient, since each system call involves considerable overhead in terms of kernel context switching, and the copying to user space of one or two bytes. Not only is each byte copied, it is copied across system level boundaries (user to kernel space). Effectively, it means that each byte of the header is read separately, and thus the performance penalties in using such an approach could be severe.

The function ReadMessageWithHeader shown in Listing 4 accomplishes an efficient read, copy and check for a terminating sequence. Multiple TCP read operations are used to accumulate the receive block. Each read operation is called using the bytes remaining in the buffer, so as not to overflow the buffer. The scan for the termination sequence is done from the start of the new block, not from the beginning on each read. Of course, this must take into account the fact that a termination sequence may straddle a block boundary.
Listing 4: The ReadMessageHeader function.

```c
ndata = 0; // returned value

fcntl(fd, F_SETFL, O_NONBLOCK);

requestTimeout.tv_sec = timeout;
requestTimeout.tv_usec = 0;
FD_ZERO(&requestFDSet);
FD_SET(fd, &requestFDSet);

// accumulate buffer, one recv() at a time
nleft = buflen;
pbuf = buf;
dbuf = 0;
nCR = 0;
nLF = 0;

while( nleft > 0 )
{
    rc = select(FD_SETSIZE, &requestFDSet, (fdset*)NULL, (fdset*)NULL, &requestTimeout);
    if ( rc <= 0 )
        return -1;

    // read as many bytes as are available, up to the remaining buffer size
    nread = recv(fd, pbuf, nleft, 0);

    if ( nread < 0 ) // read() error
        {
            // check for interrupted system call
            if ( errno == EINTR )
                continue;
            return -1;
        }

    if ( nread == 0 ) // stream closed by peer
        return -1;

    n = nbuf; // start scanning at last read position
    nbuf += nread; // end of buffer so far

    while( n < nbuf )
    {
        if ( buf[n] == LF )
            { 
                if( ++nLF == 2 )
                    { // replace second LF with \0 (LF-LF or CR-LF CR-LF)
                        buf[n] = '\0';
                        nhdr = n+1;
                        ndata = nbuf - nhdr; // additional, non-header bytes read in
                    }
            }

    return nhdr; // return bytes in header, including last null
}
```
5 The Receive Mechanism

```c
if ( buf[n] == CR )
{
    if ( ++nCR == 2 )
    {
        if ( nLF == 0 )
        {
            // replace second CR with \0 (CR-CR)
            buf[n] = '\0';

            nhdr = n + 1;
            ndata = nbuf - nhdr; // additional, non-header bytes read in
            return nhdr; // return bytes in header, including last null
        }
    }
}
else if ( (buf[n] != CR) && (buf[n] != LF) )
{
    nLF = 0;
    nCR = 0;
}
}

nleft -= nread;
pbuf += nread;
```

Note that in reading a fixed-size block for efficiency, data beyond the end of the header may be stored in the supplied buffer. This state information is returned as:

1. The number of bytes in the header (returned value `nhdr`).
2. The length of any additional data beyond this (variable `ndata`).

Thus, a processing sequence such as shown in Listing 5 is required after the call to read the header. This holding of state information means that pointers must be passed in C so that a return value is saved. This is ideally suited to an object-oriented implementation, where the object can maintain the current state of the read operation within itself, hidden from the caller application.

Listing 5: Calling the ReadMessageHeader function.

```c
if ( nhdr > 0 )
{
    // check header
    ...
    if ( ndata > 0 )
    {
        // point to ndata bytes of data
        pdata = &buf[nhdr];
        ...
    }
}
```
6 Further Issues

Besides the fundamental operations of sending and receiving formatted data on a TCP virtual circuit, the implementation of client and server software requires attention to a number of other issues. In no particular order, these include:

Handling simultaneous connections using multiple sockets and multiple threads or processes.

Security aspects including reverse address resolution for servers, transaction and error logging in a failsafe manner, server filesystem protection, client/server authentication, and transaction encryption.

Server load balancing using pre-forked processes or pre-spawned threads on the server, and DNS rotary IP selection by the client.

7 Conclusion

This report has highlighted some fundamental problems in using application-layer protocols over TCP, and presented some solutions which have been found to work well in practice. These include efficient scheduling of the send operation in the presence of Nagle’s algorithm, and multiple non-overlapping read operations which, if required, can efficiently delineate the protocol header from subsequent binary or non-binary data. All of the above methods have been implemented in C on Unix and Windows NT platforms, and using Java on the Java Virtual Machine. It is noted that the Java implementation provides a particularly elegant way to maintain the current state of the transaction within an instance of an object (a MessageReader and a MessageWriter class), and that a C++ implementation of the C code currently used should be able to provide such advantages in tandem with the efficiency derived from compiled C++ code.

References


