TCP/IP not reliable

Dipl. Ing. (FH) Klaus Rock

Bonhoefferstrasse 37
D - 73432 Aalen

Germany

E-Mail: : k.rock@rock-technologies.com

1 WHY IS TCP NOT RELIABLE?

This post is about an obscure corner of TCP network programming, a corner where almost everybody doesn't quite get what is going on. I used to think I understood it, but found out last week that I didn't.

So I decided to trawl the web and consult the experts, promising them to write up their wisdom once and for all, in hopes that this subject can be put to rest.

The experts (H. Willstrand, Evgeniy Polyakov, Bill Fink, Ilpo Jarvinen, and Herbert Xu) responded, and here is my write-up.

Even though I refer a lot to the Linux TCP implementation, the issue described is not Linux-specific, and can occur on any operating system.

1.1 WHAT IS THE ISSUE?

Sometimes, we have to send an unknown amount of data from one location to another. TCP, the reliable Transmission Control Protocol, sounds like it is exactly what we need. From the Linux tcp(7) manpage:

> *"TCP provides a reliable, stream-oriented, full-duplex connection between two sockets on top of ip(7), for both v4 and v6 versions. TCP guarantees that the data arrives in order and retransmits lost packets. It generates and checks a per-packet checksum to catch transmission errors."*

However, when we naively use TCP to just send the data we need to transmit, it often fails to do what we want - with the final kilobytes or sometimes megabytes of data transmitted never arriving.

Let's say we run the following two programs on two POSIX compliant operating systems, with the intention of sending 1 million bytes from program A to program B

A:

```
sock = socket(AF_INET, SOCK_STREAM, 0);
connect(sock, &remote, sizeof(remote));
write(sock, buffer, 1000000);             // returns 1000000
close(sock);
```

B:

```
int sock = socket(AF_INET, SOCK_STREAM, 0);
bind(sock, &local, sizeof(local));
listen(sock, 128);
int client=accept(sock, &local, locallen);
write(client, "220 Welcome\r\n", 13);

int bytesRead=0, res;
    for(;;) {
        res = read(client, buffer, 4096);
        if(res < 0)  {
            perror("read");
            exit(1);
        }
        if(!res)
            break;
        bytesRead += res;
 }
 printf("%d\n", bytesRead);
```

Quiz question - what will program B print on completion?

A) 1000000

B) something less than 1000000

C) it will exit reporting an error

D) could be any of the above

The right answer, sadly, is 'D'. But how could this happen? Program A reported that all data had been sent correctly!

1.2 WHAT IS GOING ON

Sending data over a TCP socket really does not offer the same 'it hit the disk' semantics as writing to a normal file does (if you remember to call fsync()).

In fact, all a successful write() in the TCP world means is that the kernel has accepted your data, and will now try to transmit it in its own sweet time. Even when the kernel feels that the packets carrying your data have been sent, in reality, they've only been handed off to the network adapter, which might actually even send the packets when it feels like it.

From that point on, the data will traverse many such adapters and queues over the network, until it arrives at the remote host. The kernel there will acknowledge the data on receipt, and if the process that owns the socket is actually paying attention and trying to read from it, the data will finally have arrived at the application, and in filesystem speak, 'hit the disk'.

Note that the acknowledgment sent out only means the kernel saw the data - it does not mean the application did!

1.3 But why didn't all data arrive in the example above?

When we issue a close() on a TCP/IP socket, depending on the circumstances, the kernel may do exactly that: close down the socket, and with it the TCP/IP connection that goes with it.

And this does in fact happen - even though some of your data was still waiting to be sent, or had been sent but not acknowledged: the kernel can close the whole connection.

This issue has led to a large number of postings on mailing lists, Usenet and fora, and these all quickly zero in on the SO_LINGER socket option, which appears to have been written with just this issue in mind:

> *"When enabled, a close(2) or shutdown(2) will not return until all queued messages for the socket have been successfully sent or the linger timeout has been reached. Otherwise, the call returns immediately and the closing is done in the background. When the socket is closed as part of exit(2), it always lingers in the background."*

So, we set this option, rerun our program. And it still does not work, not all our million bytes arrive.

1.4 How come?

It turns out that in this case, section 4.2.2.13 of RFC 1122 tells us that a close() with any pending readable data could lead to an immediate reset being sent.

> *"A host MAY implement a 'half-duplex' TCP close sequence, so that an application that has called CLOSE cannot continue to read data from the connection. If such a host issues a CLOSE call while received data is still pending in TCP, or if new data is received after CLOSE is called, its TCP SHOULD send a RST to show that data was lost."*

And in our case, we have such data pending: the "220 Welcome\r\n" we transmitted in program B, but never read in program A!

If that line has not been sent by program B, it is most likely that all our data would have arrived correctly.

1.5 Are we good to go?

Not really. The close() call really does not convey what we are trying to tell the kernel: please close the connection after sending all the data I submitted through write().

Luckily, the system call shutdown() is available, which tells the kernel exactly this. However, it alone is not enough. When shutdown() returns, we still have no indication that everything was received by program B.

What we can do however is issue a shutdown(), which will lead to a FIN packet being sent to program B. Program B in turn will close down its socket, and we can detect this from program A: a subsequent read() will return 0.

Program **A** now becomes:

```
sock = socket(AF_INET, SOCK_STREAM, 0);
    connect(sock, &remote, sizeof(remote));
    write(sock, buffer, 1000000);             // returns 1000000
    shutdown(sock, SHUT_WR);
    for(;;) {
        res=read(sock, buffer, 4000);
        if(res < 0) {
            perror("reading");
            exit(1);
        }
        if(!res)
            break;
 }
 close(sock);
```

1.6 So is this perfection?

Well.. If we look at the HTTP protocol, there data is usually sent with length information included, either at the beginning of an HTTP response, or in the course of transmitting information (so called ‘chunked’ mode).

And they do this for a reason. Only in this way can the receiving end be sure it received all information that it was sent.

Using the shutdown() technique above really only tells us that the remote closed the connection. It does not actually guarantee that all data was received correctly by program B.

The best advice is to send length information, and to have the remote program actively acknowledge that all data was received.

This only works if you have the ability to choose your own protocol, of course.

1.7 What else can be done?

If you need to deliver streaming data to a ‘stupid TCP/IP hole in the wall’, as I’ve had to do a number of times, it may be impossible to follow the sage advice above about sending length information, and getting acknowledgments.

In such cases, it may not be good enough to accept the closing of the receiving side of the socket as an indication that everything arrived.

Luckily, it turns out that Linux keeps track of the amount of unacknowledged data, which can be queried using the SIOCOUTQ ioctl(). Once we see this number hit 0, we can be reasonably sure our data reached at least the remote operating system.

Unlike the shutdown() technique described above, SIOCOUTQ appears to be Linux-specific. Updates for other operating systems are welcome.

1.8 BUT HOW COME IT 'JUST WORKED' LOTS OF TIMES!

As long as you have no unread pending data, the star and moon are aligned correctly, your operating system is of a certain version, you may remain blissfully unimpacted by the story above, and things will quite often 'just work'. But don't count on it.

1.9 SOME NOTES ON NON-BLOCKING SOCKETS

Volumes of communications have been devoted the the intricacies of SO_LINGER versus non-blocking (O_NONBLOCK) sockets. From what I can tell, the final word is: don't do it. Rely on the shutdown()-followed-by-read()-eof technique instead. Using the appropriate calls to poll/epoll/select(), of course.

1.10 A FEW WORDS ON THE LINUX SENDFILE() AND SPLICE() SYSTEM CALLS

It should also be noted that the Linux system calls sendfile() and splice() hit a spot in between - these usually manage to deliver the contents of the file to be sent, even if you immediately call close() after they return.

This has to do with the fact that splice() (on which sendfile() is based) can only safely return after all packets have hit the TCP stack since it is zero copy, and can't very well change its behaviour if you modify a file after the call returns!

Please note that the functions do not wait until all the data has been acknowledged, it only waits until it has been sent.

1.11 SOURCES

http://blog.netherlabs.nl/articles/2009/01/18/the-ultimate-so_linger-page-or-why-is-my-tcp-not-reliable