December 2011
Geoff Huston
Emile Aben

The Curious Case of the Crooked TCP Handshake

In this article we will be delving into the behaviour of the Linux implementation of TCP, and looking at the way in which TCP establishes a connection. There are socket options in Linux that cause the TCP handshake to behave in a rather curious way.

In examining dual stack behavioural patterns with various combinations of browsers and client operating systems (http://www.potaroo.net/ispcol/2011-12/esotropia.html) I used a Linux server running an Apache HTTP daemon (this is a relatively commonplace configuration for delivering web content). There was one curious transaction in the opening TCP handshake:

```
 Time  Activity

       → SYN to 88.198.69.81
  296    ← SYN+ACK from 88.198.69.81
    0  → ACK to 88.198.69.81
 3200    ← SYN+ACK from 88.198.69.81
    0  → ACK to 88.198.69.81
```

The measurement was performed at the client side of the connection, and the times are the inter-packet times, measured in milliseconds.

That second SYN+ACK 3 seconds after the initial SYN+ACK packet is the curious part of this transaction. If the server has received the ACK packet then it should've opened the TCP connection and should now be waiting for a data packet which is the HTTP request.

If this had happened once then packet loss would be a highly likely explanation, but in this case the behaviour is consistently repeatable. If a client connects to this server and simply waits (as happens with some browsers that open up multiple parallel sessions as a means of speeding up the retrieval of compound web pages) then three seconds after receiving the original SYN+ACK it will receive a duplicate SYN+ACK.

Is this a bug in Linux? What's going on?

A simple server on Linux will not show this behaviour. What exposes this duplicate SYN_+ACK behaviour is setting the TCP connection with the socket option TCP_DEFER_ACCEPT.

This option takes a integer parameter, which is documented as a time (in seconds) that is related to the maximum number of attempts TCP will make to complete the connection.

Here's the extract from the man page on "tcp" on a Linux host:

```
$ man tcp
 [...]
 Socket Options
   To set or get a TCP socket option, call getsockopt(2) to read or
   setsockopt(2) to write the option with the option level argument
   set to IPPROTO_TCP.  In addition, most IPPROTO_IP socket options
    are  valid on TCP sockets.  For more information see ip(7).
    [...]
   TCP_DEFER_ACCEPT (since Linux 2.4)
      Allow  a listener to be awakened only when data arrives on the
      socket.  Takes an integer value (seconds), this can bound the
      maximum number of attempts TCP will make to complete the
      connection.  This option should not be used in code intended
      to be portable.
```

In a conventional TCP server implementation the server sets up a listening process associated with a port (or a "socket"). This process then awaits an incoming connection. This connection is made by the TCP three-way handshake, where the server sees an incoming SYN packet, responds with a SYN+ACK packet and awaits an ACK from the remote client. When this TCP connection completes the TCP code in the kernel signals the server process with an incoming connection. At this point the server process typically forks off a dedicated child process to handle the connection, and the application session is underway. Often the first call the child process will make is a read on the newly established connection, to read the application level request.

At the other end the client initiates the connection by sending the initial SYN packet, and then awaits the SYN+ACK response. When the client receives the SYN+ACK packet, tt will then send an empty ACK packet to complete the TCP connection, and the kernel TCP connection process returns success to the client application. If the application level protocol is one which is initiated by the client (such as HTTP), the client will immediately follow this connection with a write to the newly opened network socket, which becomes a TCP data packet that contains new data, and also repeats the ACK information contained in the final packet of the initial TCP three-way connection.

TCP-DEFER_ACCEPT Behaviour

How does the TCP_DEFER_ACCEPT option alter this behaviour?

If the server sets this option on the listener socket then the server will not complete the initial handshake when it receives the empty ACK packet that is intended to complete the connection. Instead the server will discard this packet and will await the following data+ACK packet. When it received this data+ACK packet it will awaken the listener process with a new connection request, and the data is already available for the server process to read.

The server will only wait a certain time for this data+ACK packet, where the time is set (indirectly) by the server as a parameter to the TCP_DEFER_ACCEPT option. If the timer expires before receiving the data+ACK packet the server will lift its ACK filter and resend the SYN+ACK packet. It will now complete the connection process when it receives an ACK with or without a data payload, at which point it will awaken the listener process to start the session.

Our experiment used the Apache HTTP daemon, which sets the TCP_DEFER_ACCEPT timer to 3 seconds on Linux hosts, and because we were tracking a connection being made by the client browser in "standby mode" then there was no data to send when the connection was made, so after three seconds the server's TCP_DEFER_ACCEPT timer expired and the second SYN+ACK packet was sent.

Lets look at the TCP_DEFER_ACCEPT option in a bit more detail.

The code to set this socket option in the server is

```
      if (setsockopt(sockfd, IPPROTO_TCP, TCP_DEFER_ACCEPT, &defer_accept_value,
           sizeof(int)) == -1) {
```

The documentation suggests that value of this socket option has some relationship with the "maximum number of attempts TCP will make to complete the connection." In practice this is not the case, and what appears to be going on is that this value has something to do with an ACK timer. When the listener sends the SYN+ACK packet it starts a timer and awaits an ACK packet with a data payload, discarding all intervening ACKs with an empty payload. If the timer expires without receiving such an ACK with a data payload it resends the SYN+ACK and then will be prepared to complete the connection with any received ACK packet.

But the value provided to the setsocket option is not exactly the value of the ACK timer used by the server. I've performed an experiment with setting the server's TCP_DEFER_ACCEPT value from 1 to 12 and recorded the time between successive received SYN+ACK packets at the client end. The values recorded by the server are shown in the table below.

TCP_DEFER_ACCEPT Value	Delay Timer (secs)
1	3
2	3
3	3
4	9
5	9
6	9
7	9
8	9
9	9
10	22
11	22
12	22

The successive differences in the delay timer values here are 3, 6 and 12 seconds. That 3, 6, 12 sequence is familiar, as it is the SYN timeout sequence used to govern SYN retransmits when attempting to connect to an unresponsive address.

Perhaps a glance at the source code of the TCP implementation in LINUX would help:

```
    In tcp.c there is code to perform the setsockopt call:

         case TCP_DEFER_ACCEPT:
           /* Translate value in seconds to number of retransmits */
           icsk->icsk_accept_queue.rskq_defer_accept =
                secs_to_retrans(val, TCP_TIMEOUT_INIT / HZ, TCP_RTO_MAX / HZ);
           break;

     and the secs_to-retrans() function is:

       /* Convert seconds to retransmits based on initial and max timeout */
     static u8 secs_to_retrans(int seconds, int timeout, int rto_max)
     {
       u8 res = 0;
         if (seconds > 0) {
           int period = timeout;
           res = 1;
           while (seconds > period && res < 255) {
             res++;
             timeout <<= 1;
             if (timeout > rto_max)
```

```
                timeout = rto_max;
            period += timeout;
            }
        }
      return res;
    }
```

If TCP_TIMEOUT_INIT is defined as 3,and HZ as 1 then the discrete TCP_DEFER_ACCEPT timeout values are 3, 9, 22, 46 and 94 seconds, which is what we've observed.

What use is this option?

If the server uses this option, then instead of completing the connection and invoking a dedicated child process to handle the connection upon reception of the ACK packet that would normally complete the connection, the server will await the first ACK packet with a data payload, and then return a connected status to the server process with data available in the socket's read buffer.
But is this any faster? Well, no. If we take the initial protocol transaction as the five packet exchange:

```
1              SYN →
2                      ← SYN+ACK
3              ACK →
4         ACK+data →
5                      ← ACK
```

then the elapsed time to complete this initial transaction is 2 Round Trip Times (RTT). The server would normally invoke the child process upon receipt of packet 3, and with the TCP_DEFER_ACCEPT socket option turned on it will defer this to receipt on packet 4. In this case the number of packets being sent is unaltered, and the transaction takes precisely the same amount of time.

For the server, setting this option appears to achieve nothing in isolation.

What about setting TCP_DEFER_ACCEPT for the client?

It appears that in this case the connect function will return back to the client process without sending the final ACK of the handshake, and will instead fire up a 200ms timer. If the client performs a socket write call in this period the timer is cancelled and this data packet is sent to both complete the connection handshake and pass data in to the read buffer at the server side. I'm unsure what point setting a timer value is on the client side, as this 200ms timer appears to be invariant for all TCP_DEFER_ACCEPT values set in the `setsockopt` call.

If the client uses this option then the initial connection transaction can be combined with the first data packet from the client to the server and a five packet transaction can be dropped to four packets.

```
1              SYN →
2                      ← SYN+ACK
3         ACK+data →
4                       ← ACK
```

In this case it does not appear to matter whether the server sets the TCP_DEFER_ACCEPT or not. Servers on systems that do not support the TCP_DEFER_ACCEPT option are still prepared to accept ACK packets that contain a data payload to complete the TCP connection, and pass the data to the established connection.

But, frankly I can't see the point of this - the total time taken in either case is the same for both client and server. It takes 2 RTT intervals to undertake the connection and complete the first data transaction in either case. Yes, one empty TCP packet is not sent, but this makes no difference to the speed of the connection.

Bug or Feature?

It's hard to make a convincing case that this option is a useful feature.

Setting this option on the server, the client, or at both ends of the connection, makes no difference to the elapsed time to set up a connection. In the case of a server with this option set, then if the client also has the option set then certain client behaviours can reduce the packet count by one empty TCP packet. While other client behaviours (not immediately sending data after establishing the connection) can increase the total packet count by one. Considering that the packet saved in one case, or additional packets sent in another case, are null payload packets the impact of setting (or not setting) this option is slight. There is no saving in elapsed time in any combination of server and client setting this option, and it bears all the appearance of an added item of detail without any real performance benefit.

So if its not a feature, then is it a bug?

As far as I can tell the option itself is mostly harmless. Setting this option at the client or the server does not appear to cause the remote client or server to hang. The one slight caveat is that setting this on the client side requires the server to correctly handle the data payload contained in the closing ACK of the TCP handshake.

What about at the server? Setting this at the server side appears to be largely cosmetic. The client will still send the empty ACK, and the only change is that the server will not immediately fire off a child process to manage the connection, but will wait in connecting state for the first data packet to be received. Again this appears to be a change in behaviour that is mostly harmless. (I say "mostly" harmless, because some configurations that involve server farms and front end load balancers assume that there is a clean separation on the initial TCP handshake and the subsequent transaction, and there are complex failure modes that arise when this option is used in such a case.)

However, there is still something that looks to me a lot like an implementation bug. When the server's TCP_DEFER_ACCEPT timer fires off it resends that last SYN+ACK packet. I think this behaviour is an implementation bug. The server code should not deliberately forget the fact that it has seen a ACK packet during the DEFER time period. Instead, it should set a state flag when it receives such a packet during the DEFER wait time within the connection setup. If the timer expires without receiving an ACK packet with a data payload, then this state flag would allow the server to transition to the CONNECTED state silently, thereby avoiding resending the SYN+ACK packet.

Resources and Further Reading

Test Code to exercise this option:

client.c -- http://www.potaroo.net/ispcol/2011-12/client.c
server.c -- http://www.potaroo.net/ispcol/2011-12/server.c

Other references to this TCP option:

http://www.techrepublic.com/article/take-advantage-of-tcpip-options-to-optimize-data-transmission/1050771
https://bugs.launchpad.net/ubuntu/+source/apache2/+bug/134274

http://marc.info/?l=linux-netdev&m=116753348815044&w=2

Disclaimer

Authors

Geoff Huston B.Sc., M.Sc., is the Chief Scientist at APNIC, the Regional Internet Registry serving the Asia Pacific region. He has been closely involved with the development of the Internet for many years, particularly within Australia, where he was responsible for the initial build of the Internet within the Australian academic and research sector. He is author of a number of Internet-related books, and was a member of the Internet Architecture Board from 1999 until 2005, and served on the Board of Trustees of the Internet Society from 1992 until 2001.
www.potaroo.net

Emile Aben B.Sc, M.Sc., is a System Architect for the Research and Development department at the RIPE NCC, the Regional Internet Registry serving Europe and the Middle East. Prior to that role, he worked in the RIPE NCC Science Group as a research engineer since 2009. In the 10 years before that he has worked as a web developer, sysadmin, security consultant and researcher. He is interested in technology changes like IPv6 and DNSSEC deployment. Emile has a MSc. degree in Chemistry from the University of Nijmegen, NL.