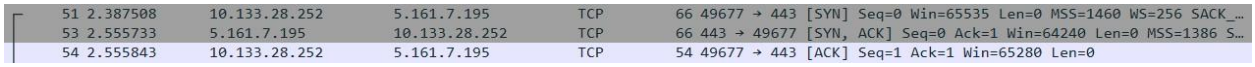
The Life, Death, and Miraculous Recovery of a TCP Connection

Alden Harcourt

The TCP Handshake

Before a computer downloads a single byte of data from a server, it must first have an introductory conversation consisting of 3 parts, known as the TCP handshake. This is more than just a simple hello, it is a sophisticated conversation that determines the rules of play for the upcoming data transfer between the client and server. I used Wireshark to document this process during a conversation in which my machine (10.133.28.252) attempted to download a 1GB file from a server (5.161.7.195). The screenshot below shows the first 3 packets of this conversation, which constitute the TCP handshake.



Let's start at a high level and then we'll delve into the fun and interesting details. Packet 51 is where our conversation begins. This is the client, my machine, sending a SYN (synchronize) packet to the server. As the name suggests, this packet starts the process of synchronization between the client and server. But what does that even mean!? My machine is telling the server "Hi, I'd like to start a connection. Here are some variables that represent how I want to talk to you". Let's zoom into what exactly those variables are.

```
49677 → 443 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=256 SACK_PERM
443 → 49677 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1386 SACK_PER…
```

As you can see above, the SYN packet includes a set of flags and data:

- Seq=0: This is the **Relative Sequence Number** for the packet. Think of this as the byte counter for the whole connection. This initial packet sets that counter to 0. In reality, the sequence number starts as a huge random number but for usability purposes, Wireshark converts sequence numbers to a 0 indexed value.

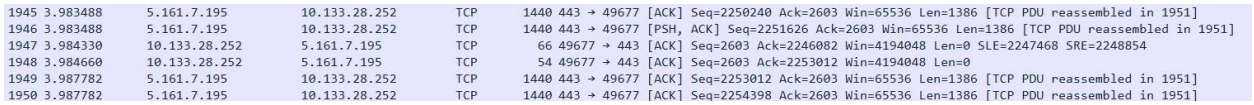
- Win=65535: Here, our client is saying “I have a 65kb mailbox (buffer), don’t send me more data than that at once, or my mailbox will overflow and I’ll have to drop your letters.” This is the key player in **Flow Control**.
- WS=256: You might be thinking, “A 65kb mailbox isn’t very big for a 1GB file!” You’re right. When TCP was introduced, buffers were typically small and the largest buffer that could be advertised was 65535 bytes. The **Window Scaling** factor allows a machine to advertise its true buffer size. Multiplying the Win by the WS gives the effective window, the amount of data that a party can buffer.
- Len=0: This stands for TCP Segment Length. It tells you the size, in bytes, of the actual data payload. You might notice that this value is 0. This is completely normal for a SYN packet. The only job of a SYN packet is to negotiate the connection, they aren’t carrying any application data.
- MSS=1460: Here, the client is saying “I propose a Maximum Segment Size of 1460 bytes.” This tells the server “Don’t send me data segments larger than 1460 bytes.” This is done to avoid packets fragmenting into multiple, which is slow and inefficient.
- SACK_PERM: Here, I’m saying that “I support Selective Acknowledgement.” This is a critical feature of what’s called **Fast Recovery**. It’s the client promising “If you send me packets 1, 2, 3, 5, I can tell you I’m missing 4 *and* I have 5.

Wow, that was a lot. Now let’s move on to the second packet out of 888,000: packet 53. This packet has both the SYN and ACK flags set. Much of it is the same: Seq, Len, and SACK_PERM all appear the same and the server is advertising its own MSS and WIN. However, this packet includes an ACK, something we will see repeatedly moving forward. This is the server acknowledging the SYN packet sent to it. ACK=1 (Client Seq Number + 1) confirms to the client that the server received packet 51.

Now for the final part of the TCP handshake. Packet 54 is simply my machine acknowledging that I received the packet containing the server’s proposals of conversation, the SYN part of packet 53, and the server’s acknowledgement of the first packet I sent, the ACK part. Now, the handshake is complete, the client and server are synchronized, and data can now flow.

Data Flow and Flow Control

Now that we have a connection established, the download begins. Hundreds of thousands of data packets flow through the connection as information is downloaded onto my computer. For the most part, this is a simple, repetitive loop: server sends data, client acknowledges it, server sends more data... But, the details of how this loop functions provide some interesting insights. Here is an example of this loop and some interesting phenomena that arise:



1945	3.983488	5.161.7.195	10.133.28.252	TCP	1440 443 → 49677 [ACK] Seq=2250240 Ack=2603 Win=65536 Len=1386 [TCP PDU reassembled in 1951]
1946	3.983488	5.161.7.195	10.133.28.252	TCP	1440 443 → 49677 [PSH, ACK] Seq=2251626 Ack=2603 Win=65536 Len=1386 [TCP PDU reassembled in 1951]
1947	3.984330	10.133.28.252	5.161.7.195	TCP	66 49677 → 443 [ACK] Seq=2603 Ack=2246082 Win=4194048 Len=0 SLE=2247468 SRE=2248854
1948	3.984660	10.133.28.252	5.161.7.195	TCP	54 49677 → 443 [ACK] Seq=2603 Ack=2253012 Win=4194048 Len=0
1949	3.987782	5.161.7.195	10.133.28.252	TCP	1440 443 → 49677 [ACK] Seq=2253012 Ack=2603 Win=65536 Len=1386 [TCP PDU reassembled in 1951]
1950	3.987782	5.161.7.195	10.133.28.252	TCP	1440 443 → 49677 [ACK] Seq=2254398 Ack=2603 Win=65536 Len=1386 [TCP PDU reassembled in 1951]

First, let's look at packet 1946. This is one of the many chunks of the 1GB file that the server is sending to my machine. 1946 has the ACK flag, acknowledging the last received packet, just like in the earlier packets we have discussed. It also has the PSH flag, short for push. This flag is a signal to our computer's OS saying, "Don't just buffer this data. Push it up to the application (in our case the browser I am downloading this file on) right away."

Packet 1946 also contains information that builds on our understanding of TCP flags such as Seq and Len. You can see Seq=2251626 and Len=1386. The server is sending 1386 bytes of data, which starts at byte #2,251,626 in the stream.

Now let's jump to Packet 1948. This one is sent from client to server, it is our reply. Notice its Len=0; it's not sending data, just acknowledging the reception of a packet. We can see that it has Ack=2253012, this is the most important part! Let's do a little bit of math to see why this value makes sense.

(Packet 1946's Seq) + (Packet 1946's Len) = (Packet 1948's ACK)

2251626 + 1386 = 2253012

By sending Ack=2253012, our client is saying, "I have successfully received all data you have sent me up to byte 2,253,012. I am expecting you to send me a packet that starts with byte #2,253,012." This Seq/ACK dance is what ensures the reliability of transmission through TCP.

But wait, there's more! This packet also shows Win=4194048. This is our flow control signal. We are advertising a buffer size of 4.19MB, saying "I have 4.19MB of space in my buffer, you can send me up to 4.19MB of data *without* waiting for another ACK, but no more than that." This is the key to

ensuring the stream of data doesn't overflow the capacity of the network while maximizing our resources.

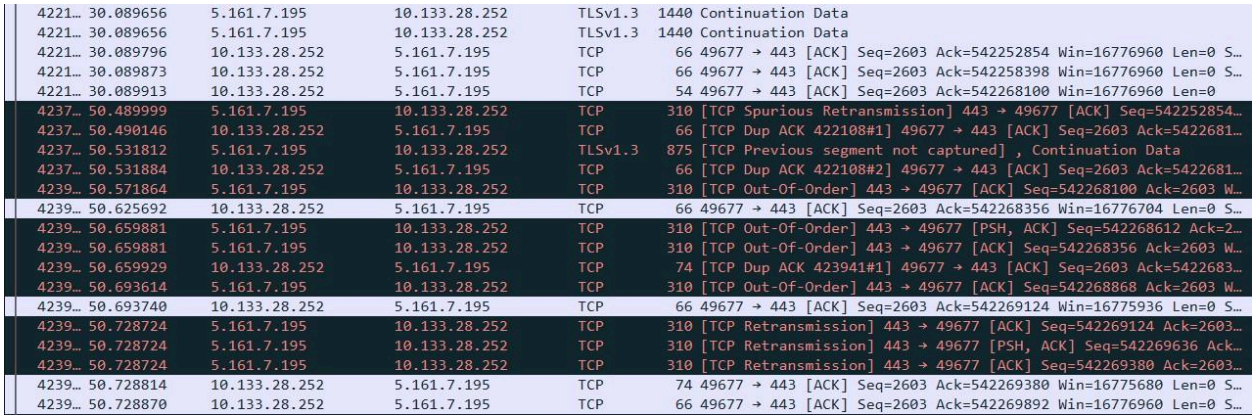
You may be thinking, "But Alden, packet 1946 sent you data, packet 1948 acknowledged that data, but what about packet 1947 which also seems to be acknowledging data?" 1947 is an ACK after all, and it comes right after 1946. This packet stuck in between is an example of Selective Acknowledgement at work. This is the payoff for the SACK_PERM that we negotiated during the TCP handshake. Let's zoom into this packet's details.

```
[ACK] Seq=2603 Ack=2246082 Win=4194048 Len=0 SLE=2247468 SRE=2248854
```

Let's start with Ack=2246082, SLE=2247468 (Start of Left Edge), and SRE=2248854 (Start of Right Edge). Here's the translation: Packet 1947 is our client telling the server, "Hey server, I'm still stuck waiting on the data that starts at byte 2246082. However, you don't need to resend me everything that comes after that because I did receive a chunk of data from byte 2247468 to byte 2248854. Just send me the data between bytes 2246082 and 2247467 "

This is super efficient. It means that just because a packet of data was received out of order, the client can still hang on to it and the server won't need to resend it Even the simple cycle of data transfer and acknowledgement that repeats thousands and thousands of times across this trace holds interesting insights into how TCP controls the flow of data and ensures a reliable connection between endpoints.

The Blackout



4221…	30.089656	5.161.7.195	10.133.28.252	TLSv1.3	1440 Continuation Data
4221…	30.089656	5.161.7.195	10.133.28.252	TLSv1.3	1440 Continuation Data
4221…	30.089796	10.133.28.252	5.161.7.195	TCP	66 49677 → 443 [ACK] Seq=2603 Ack=542252854 Win=16776960 Len=0 S…
4221…	30.089873	10.133.28.252	5.161.7.195	TCP	66 49677 → 443 [ACK] Seq=2603 Ack=542258398 Win=16776960 Len=0 S…
4221…	30.089913	10.133.28.252	5.161.7.195	TCP	54 49677 → 443 [ACK] Seq=2603 Ack=542268100 Win=16776960 Len=0
4237…	50.489999	5.161.7.195	10.133.28.252	TCP	310 [TCP Spurious Retransmission] 443 → 49677 [ACK] Seq=542252854…
4237…	50.490146	10.133.28.252	5.161.7.195	TCP	66 [TCP Dup ACK 422108#1] 49677 → 443 [ACK] Seq=2603 Ack=5422681…
4237…	50.531812	5.161.7.195	10.133.28.252	TLSv1.3	875 [TCP Previous segment not captured] , Continuation Data
4237…	50.531884	10.133.28.252	5.161.7.195	TCP	66 [TCP Dup ACK 422108#2] 49677 → 443 [ACK] Seq=2603 Ack=5422681…
4239…	50.571864	5.161.7.195	10.133.28.252	TCP	310 [TCP Out-Of-Order] 443 → 49677 [ACK] Seq=542268100 Ack=2603 W…
4239…	50.625692	10.133.28.252	5.161.7.195	TCP	66 49677 → 443 [ACK] Seq=2603 Ack=542268356 Win=16776704 Len=0 S…
4239…	50.659881	5.161.7.195	10.133.28.252	TCP	310 [TCP Out-Of-Order] 443 → 49677 [PSH, ACK] Seq=542268612 Ack=2…
4239…	50.659881	5.161.7.195	10.133.28.252	TCP	310 [TCP Out-Of-Order] 443 → 49677 [ACK] Seq=542268356 Ack=2603 W…
4239…	50.659929	10.133.28.252	5.161.7.195	TCP	74 [TCP Dup ACK 423941#1] 49677 → 443 [ACK] Seq=2603 Ack=5422683…
4239…	50.693614	5.161.7.195	10.133.28.252	TCP	310 [TCP Out-Of-Order] 443 → 49677 [ACK] Seq=542268868 Ack=2603 W…
4239…	50.693740	10.133.28.252	5.161.7.195	TCP	66 49677 → 443 [ACK] Seq=2603 Ack=542269124 Win=16775936 Len=0 S…
4239…	50.728724	5.161.7.195	10.133.28.252	TCP	310 [TCP Retransmission] 443 → 49677 [ACK] Seq=542269124 Ack=2603…
4239…	50.728724	5.161.7.195	10.133.28.252	TCP	310 [TCP Retransmission] 443 → 49677 [PSH, ACK] Seq=542269636 Ack…
4239…	50.728724	5.161.7.195	10.133.28.252	TCP	310 [TCP Retransmission] 443 → 49677 [ACK] Seq=542269380 Ack=2603…
4239…	50.728814	10.133.28.252	5.161.7.195	TCP	74 49677 → 443 [ACK] Seq=2603 Ack=542269380 Win=16775680 Len=0 S…
4239…	50.728870	10.133.28.252	5.161.7.195	TCP	66 49677 → 443 [ACK] Seq=2603 Ack=542269892 Win=16776960 Len=0 S…

Suddenly, 30 seconds into the trace, I turned off my computer's Wi-Fi connection. This ceased all internet traffic going to and from my computer. I

turned my computer's Wi-Fi connection capabilities back on after 20 seconds and a flood of reliability mechanisms fired at once, attempting to recover from this catastrophic loss of connection.

The first packet that appears is a TCP Spurious Retransmission, sent by the server. This is a timer-based recovery. Whenever a packet is sent, the sender has a Retransmission Timeout (RTO) corresponding to that packet. This is a timer that upon its expiration signals the necessity to retransmit a packet. The assumption is that for whatever reason, the packet originally sent may not have reached its intended destination. The RTO expired because the server got total silence from us and no acknowledgement of a previously sent packet. So, it re-sent the packet just in case. Wireshark labels this packet 'spurious' because it knows this data (starting at Seq=542252854) had already been successfully acknowledged by our client *before* the blackout (in Packet 422108, which acknowledged all data up to Ack=542268100).

Further down a number of packets are labeled as TCP Out-of-Order packets. Let's say we were waiting for packet 100, but we got 102, 103, and 104. These packets are flagged as "Out-of-Order" by wireshark. But remember, because we agreed to SACK_PERM, the client does not discard these packets. Instead it holds onto them as pieces of data in its buffer and reminds the server to resend the missing packets through another mechanism.

Mixed in with the TCP Out-of-Order packets are a number of [TCP Dup ACK] packets. This is our client's response to receiving all of those "Out-of-Order" packets.

And, as we saw earlier, because we negotiated SACK_PERM at the start, these aren't just *simple* Dup ACKs. They are SACK-enhanced Dup ACKs.

Let's look at packet 423779 to see this powerful feature in action during this massive failure.

```
[TCP Dup ACK 422108#2] 49677 → 443 [ACK] Seq=2603 Ack=542268100 Win=16776960 Len=0 SLE=545840948 SRE=545841769
```

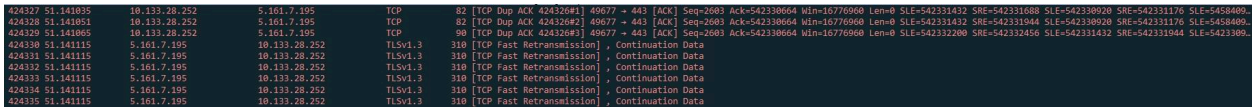
As we learned before, the packet is a [TCP Dup ACK] (the vehicle) that also contains SACK data (the cargo).

- The Vehicle (The Dup ACK): The packet is labeled [TCP Dup ACK 422108#2] and its acknowledgment number is Ack=542268100. This is the client yelling that it is still stuck waiting for the data that starts at byte 542,268,100.

- The Cargo (The SACK): Inside this same packet, we see the SACK options: SLE=545840948 SRE=545841769. This is the "P.S." message we learned about, telling the server, "...but I *did* receive the chunk of data from byte 545,840,948 to 545,841,769!"

This gives the server a perfect map of our buffer, setting the stage for the Fast Retransmission that's about to occur.

The Recovery



In the last section, we saw the client yelling at the receiver about the missing data using Dup ACKs with the SACK enhancement. Now we see the server's smart response. This is called **Fast Recovery**.

Look at packets 424327, 424328, and 424329. Wireshark has conveniently labeled these as [TCP Dup ACK #1], [#2], and [#3]. The third Dup ACK is the standard TCP signal that a packet is most likely lost. We wait for three Dup ACKs to give some extra time for a packet to arrive, in case it is just held up somewhere in the network, to avoid resending a packet that will reach its destination soon anyways.

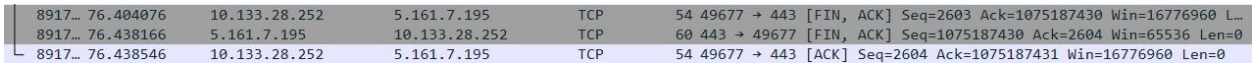
This triggers a **Fast Retransmission**. The server immediately responds to the third Dup ACK. Look at packet 424330, it's labeled [TCP Fast Retransmission]. This retransmission is fast because it didn't wait for a slow RTO timer as we've seen before. Instead, it avoids waiting for this timer and is able to quickly resend the missing packets before the RTO timer sets off.

If we take an even closer look, we see that there were actually six Fast Retransmission packets sent in response to a single trio of Dup ACKs. This is because of the detailed SACK information in these Dup ACKs. As you can see, there is a set of four SLE and SREs in the Dup ACKs. This details to the server the specific gaps of data my computer is missing, so the server knows all six packets it must retransmit to fill in those gaps.

A Polite Goodbye

Thus far we have seen the birth of a connection, healthy transmission, a catastrophic loss of connectivity, and a chaotic recovery. Thankfully, the recovery was successful and I was able to download the 1GB file. All good things must come to an end, so now we look at the teardown of a connection. This is the polite goodbye of TCP.

Because TCP is a **full-duplex** protocol, meaning that data can flow in both directions, one side can't just hang up on the conversation. Each side must shut down their flow of data, without leaving any data in limbo.



We'll start with Packet 891710 from the figure above. Here, my machine sends a packet with the flags, [FIN, ACK]. The FIN is our client saying, "The download is complete, there is no more information I need to send you, goodbye". The accompanying ACK is there to keep things tidy, acknowledging the last bit of data that was received from the server.

Packet 891711 is the server's response to the proposed teardown. It replies with a [FIN, ACK] of its own. The ACK acknowledges our client's FIN. The server's FIN functions the same as the client's FIN from earlier. The server is also saying "There is no more data I need to send you either, goodbye".

Finally, packet 891712 ends our nearly 1 million packet conversation. Our client is simply acknowledging the server's FIN. And with that, the connection is fully closed. Both sides have confirmed that both themselves and the other are finished, allowing all resources for this connection to be safely freed up by both operating systems.

A Quick Summary

As you now understand, TCP is not just a protocol, but an intelligent and adaptive system. Throughout this tutorial, we have followed the entire 76 second life of a single TCP connection. From its birth in a handshake to its polite death in a teardown. Along the way, we've dissected some of the core services that TCP provides. The simple and constant Seq/Ack dance throughout a healthy data transfer to ensure reliability, the efficient and

proactive negotiation of rules such as SACK_PERM to synchronize sender and receiver, TCPs ability to control the flow of data through the Window Size, the protocol's impressive ability to recover from a loss of connection, and so much more. Ultimately, this trace shows how these complex features all work together in fractions of a second to make the modern high-speed internet possible.

Sources/Resources:

https://ash-speed.hetzner.com/ (For the download)

https://www.youtube.com/watch?v=2PJVHvthrNU (Info on window scaling)

https://www.chappell-university.com/post/spurious-retransmissions-a-concern (Spurious retransmissions)