1. Intro

For this project, I explored how **TCP (Transmission Control Protocol)** is crucial communication protocol within the **Transport Layer** of the hourglass model of the Internet which works closely with **Internet Protocol (IP)** to provide the following services to the application layer above it:

1. **Connection Setup:** Establishes a reliable channel between the client and server using the three-way handshake.
2. **Reliable transmission:** Guarantees that all bytes sent are received correctly through acknowledgments and retransmissions.
    a. **In-order delivery:** Ensures that packets arrive at the receiver in the same order they were sent.
    b. **Flow control:** Prevents a fast sender from overwhelming a slower receiver using a sliding-window mechanism.
3. **Congestion control:** Dynamically adjusts the sending rate to avoid overloading the network.
4. **Connection termination:** Gracefully closes the session so that both ends finish sending data cleanly.

To illustrate how these services function, I collected a Trace using **WireShark for a Youtube video** to show how each of these services work. Some important context:

- I used primarily stream 11 to capture most of the services, with the exception of congestion control where stream 8 was used).
- For the sake of discussion, I will refer to myself as the client, and Youtube as the server.

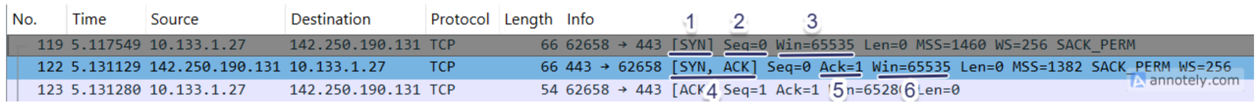
  Note: Since Youtube is owned by Google, it uses QUIC (over UDP), a different protocol. To ensure that my trace captured a TCP connection, I temporarily disabled QUIC in Google Chrome by typing `chrome://flags/#enable-quic` in the address bar and setting the flag Experimental QUIC protocol to Disabled. This forces Chrome to fall back to TCP.

Key Terms

TCP (Transmission Control Protocol): Reliable, connection-oriented transport protocol used for end-to-end data delivery.
IP Internet Protocol: Provides best-effort packet delivery
SYN (Synchronize Control flag): used to initiate a TCP connection (part of the 3-way handshake).
ACK (Acknowledgment Control flag): used to confirm receipt of data or connection setup messages.
FIN (Finish Control): flag used to signal the end of data transmission (connection teardown).
RST (Reset): abruptly terminated a connection.
RTT (Round-Trip Time): time between sending a packet and receiving its acknowledgment; used for timeout estimation.
AIMD (Additive Increase/Multiplicative Decrease): core congestion control algorithm that increases send rate gradually and decreases rapidly upon loss.

1.1 Connection Setup

Figure 1 shows the first core feature that TCP provides, which is setting up the connection between the user and server, known as a 3-way handshake.



```
No.  Time     Source          Destination     Protocol Length Info
 119 5.117549 10.133.1.27     142.250.190.131 TCP      66 62658 → 443 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=256 SACK_PERM
 122 5.131129 142.250.190.131 10.133.1.27     TCP      66 443 → 62658 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1382 SACK_PERM WS=256
 123 5.131280 10.133.1.27     142.250.190.131 TCP      54 62658 → 443 [ACK Seq=1 Ack=1 n=6528 Len=0
```

Figure 1: Shows packets involved in the initial 3-way handshake. Labels 1-6 reference bullets numbered 1-6.

- **Packet #119** shows me (**IP Address 10.113.1.27**) sending a SYN to **port 443**, which reaches the Youtube server (**IP Address 142.250.190.131**) and signals the start of the connection with a **SYN**: a control bit used to indicate the start of a connection.
    1. Additionally, the packet includes an **initial sequence number (ISN)** to note the starting point of the data stream.

2. The client also advertises its **window size** of 65,535 bytes, which indicates how much data it can have in its buffer.

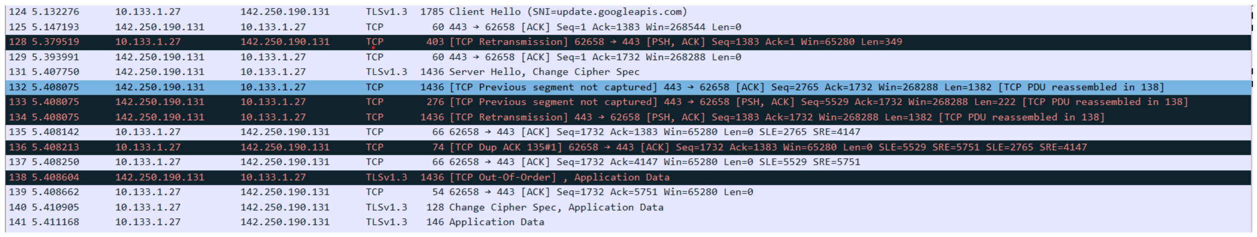
- **Packet #122**, about 13ms later, the server replies with the **SYN-ACK** flag.
  4. This is the second part of the handshake where the server acknowledges the user's SYN (this is what the ACK flag is for).
  5. The ACK number is now 1, which the server has received the client's ACK and now expects the next byte (byte 1).
  6. The server also advertises its initial window size of 65,535 bytes.

- **In packet #123,** the client confirms the SYN-ACK flag of the user by sending an ACK. and the connection enters the established state.

Overall, this process establishes the connection and ensures both the client and the server agree on sequence numbers, window size and are synchronized.

1.2 Reliable Transmission and Recovery

A key job for TCP is to ensure data arrives, even when packets are lost, delayed, or out of order.



```
124 5.132276   10.133.1.27      142.250.190.131  TLSv1.3  1785 Client Hello (SNI=update.googleapis.com)
125 5.147193   142.250.190.131  10.133.1.27      TCP        60 443 → 62658 [ACK] Seq=1 Ack=1383 Win=268544 Len=0
128 5.379519   10.133.1.27      142.250.190.131  TCP       403 [TCP Retransmission] 62658 → 443 [PSH, ACK] Seq=1383 Ack=1 Win=65280 Len=349
129 5.393991   142.250.190.131  10.133.1.27      TCP        60 443 → 62658 [ACK] Seq=1 Ack=1732 Win=268288 Len=0
131 5.407750   142.250.190.131  10.133.1.27      TLSv1.3  1436 Server Hello, Change Cipher Spec
132 5.408075   142.250.190.131  10.133.1.27      TCP      1436 [TCP Previous segment not captured] 443 → 62658 [ACK] Seq=2765 Ack=1732 Win=268288 Len=1382 [TCP PDU reassembled in 138]
133 5.408075   142.250.190.131  10.133.1.27      TCP       276 [TCP Previous segment not captured] 443 → 62658 [PSH, ACK] Seq=5529 Ack=1732 Win=268288 Len=222 [TCP PDU reassembled in 138]
134 5.408075   142.250.190.131  10.133.1.27      TCP      1436 [TCP Retransmission] 443 → 62658 [PSH, ACK] Seq=1383 Ack=1732 Win=268288 Len=1382 [TCP PDU reassembled in 138]
135 5.408142   10.133.1.27      142.250.190.131  TCP        66 62658 → 443 [ACK] Seq=1732 Ack=1383 Win=65280 Len=0 SLE=2765 SRE=4147
136 5.408213   10.133.1.27      142.250.190.131  TCP        74 [TCP Dup ACK 135#1] 62658 → 443 [ACK] Seq=1732 Ack=1383 Win=65280 Len=0 SLE=5529 SRE=5751 SLE=2765 SRE=4147
137 5.408250   10.133.1.27      142.250.190.131  TCP        66 62658 → 443 [ACK] Seq=1732 Ack=4147 Win=65280 Len=0 SLE=5529 SRE=5751
138 5.408604   142.250.190.131  10.133.1.27      TLSv1.3  1436 [TCP Out-Of-Order] , Application Data
139 5.408662   10.133.1.27      142.250.190.131  TCP        54 62658 → 443 [ACK] Seq=1732 Ack=5751 Win=65280 Len=0
140 5.410905   10.133.1.27      142.250.190.131  TLSv1.3   128 Change Cipher Spec, Application Data
141 5.411168   10.133.1.27      142.250.190.131  TLSv1.3   146 Application Data
```

Figure 2: Shows packets involved in data transmission and recovery.

Fig. 2 highlights how TCP ensures reliability in making sure the data is received and by extension, how it recovers if it hasn't.

After the connection was established from the previous step, the client sends a "Hello" in **packet #132**, and it gets acknowledged by the server (this is a part of HTTPS encryption and we are not really concerned with it for this project). At this point, the connection is functioning normally: each packet sent -> an ACK received for it.

Packet #128 is where things get complicated.

- This packet is TCP retransmission because something the client sent previously was not acknowledged by the server (Fig. 3):

403 [TCP Retransmission]

Figure 3: shows a TCP retransmission.

- Here, we see a **PUSH** AND an ACK flag: the push flag is used to indicate to TCP to send whatever data is in its buffer to the application immediately.
- The reason the retransmission occurs is because the client sent some data, and normally, the server would increment its ACK up to the sequence number for that data, but here it did not. This is selective ACKing, because the server was basically sending the same ACK numbers up to the data it received, meaning there is a gap in the stream.

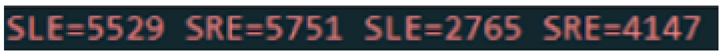
    Timeouts and RTT:
    Note: TCP uses a timeout to estimate how long to wait before resending data, so if it elapses, data is resent (we don't want to send data too often and clog the line). To keep the discussion brief, TCP changes the timeout according to how the RTTS changes (learn more here:)

Selective Acknowledgment and Out-of-Order delivery:

- In **packet #129**, after the previous transmission, this time, the server's ACK number goes up, meaning it has now received everything up to byte 1732.
- Now, after **packet #131**, Wireshark flags "previous segment not captured". This means that the data was out of order or missing.
- In **packet #134**, the server sends some data that was not previously acknowledged in starting at seq = 1383.
- In **packet #135,** this data is ACKed by the client (the ACK is 1383)!

> **Note:** SACK is a part of a TCP extension that allows it to augment its cumulative acknowledgment with selective acknowledgments of any additional segments that have been received but aren't contiguous with all previously received segments. This is the selective acknowledgment, or SACK, option. When the SACK option is used, the receiver continues to acknowledge segments normally, the meaning of the Acknowledge field does not change, but it also uses optional fields in the header to acknowledge any additional blocks of received data. This allows the sender to retransmit just the segments that are missing according to the selective acknowledgment.

- **Packet #136** is a duplicate ACK (since the ACK number hasn't advanced), and demonstrates out-of-order delivery. This is accompanied by **SACKs** (**Selective Acknowledgement**) in the header (**SRE AND SLE**), which mark the start and end of packets of data that have arrived out of order (Fig. 4)



```
SLE=5529 SRE=5751 SLE=2765 SRE=4147
```

Figure 4: Shows SACK headers.

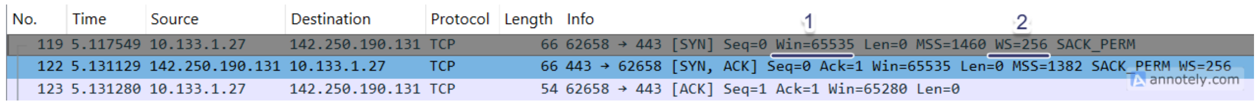
- In **packet #137**, finally, the missing data (seq 1383 - 2764) has arrived, so the client increments its ACK number to ACK=4147 (remember, it already had out of order packets 2765-4147 from before). This crucially highlights how TCP ensures data is in order.

After this, the client is all caught up and data transmission proceeds as normal. Overall, this section highlights how when packets are delayed (packet 128), TCP retransmits the segment to ensure reliability.

1.3 Flow Control and Sliding Window

The sliding window supplements TCP's reliable data transfer, in-order delivery, and flow control via window advertisements, ACKs, and retransmissions.

How the Sliding Window Works



```
No.   Time      Source          Destination     Protocol Length Info
  119 5.117549 10.133.1.27     142.250.190.131 TCP       66 62658 → 443 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=256 SACK_PERM
  122 5.131129 142.250.190.131 10.133.1.27     TCP       66 443 → 62658 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1382 SACK_PERM WS=256
  123 5.131280 10.133.1.27     142.250.190.131 TCP       54 62658 → 443 [ACK] Seq=1 Ack=1 Win=65280 Len=0
```

Figure 5: Shows the sliding window flags in the header. Labels 1-2 reference bullets numbered 1-2 below.

1. The diagram above shows the same connection setup from before, but this time lets focus on the **Advertised Window (Win field)** that highlights the advertised window. This is essentially the amount of data each side can hold in their buffer.
- You have a sender buffer (Youtube in this case).
- A receiver buffer (for the client).

2. Packet **#119** shows that the client's window size is 65,535 bytes, and the **scaling factor** is 256, for a total of about 16 MB (the sender's buffer size is the same in this case).

   Note: TCP connections often use the Window Scaling Option to support larger buffers.

Now, the available window size course changes as the connection proceeds and both sides receive data, hence the sliding window. Moreover, TCP only advances the sliding window if an ACK is received.

Interestingly, throughout the trace the advertised window remains pretty big, meaning both the TCP buffers are being effectively drained by both sides.

1.4 Congestion Control:

The sliding window makes sure the recipient of the data isn't overwhelmed, but **Congestion Control** is the TCP service that protects the network itself from being overwhelmed.So effectively, the sender is allowed to only send at most the minimum between the **Congestion Window** and **Advertised Window**. This becomes the Effective Window that the sender must adhere to.

For this discussion, the TCP trace graph illustrates the various congestion control measures.

Note: Youtube establishes multiple TCP connections with the client to send various data. Each TCP segment includes a **source port and a destination port, together with the IP addresses forming a unique 4-tuple: (Source IP , Source Port , Destination IP , Destination Port).** This combination uniquely identifies a single TCP connection. The graph in fig. 6 corresponds to stream 8 in my trace where the bulk of the data was transferred.

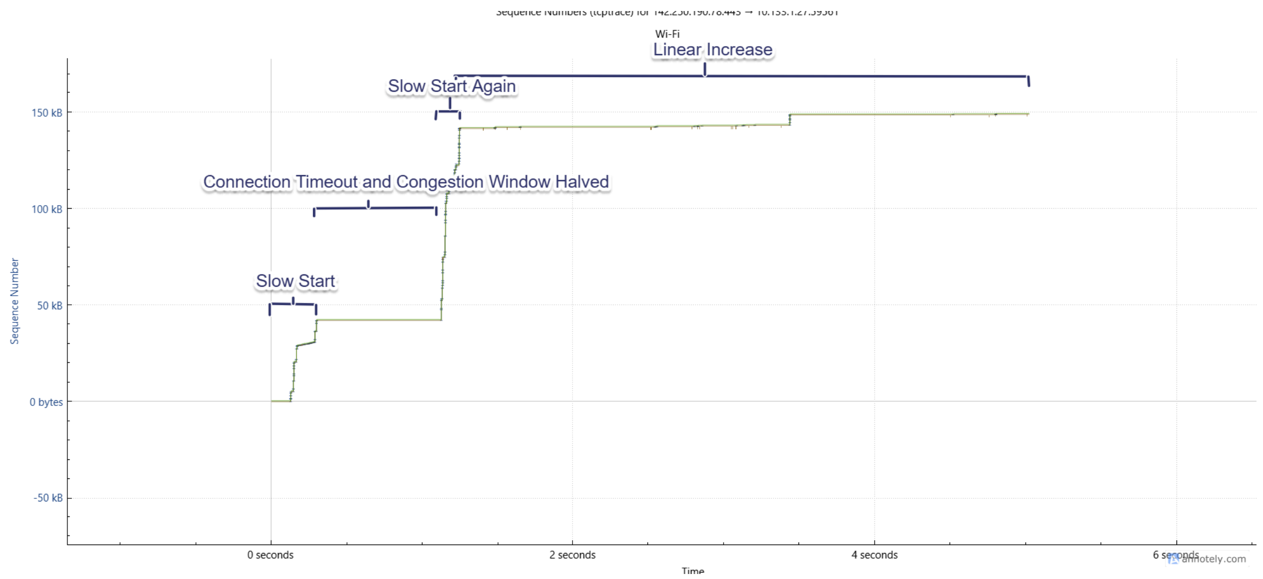


Figure 6: The y-axis represents the sequence number in kB, and the x axis represents time in seconds.

At the beginning, TCP does not have an estimate of the bandwidth (unlike flow control's advertised window which is within a packet header). Thus, TCP initiates an additive increase/multiplicative decrease (AIMD) to perceive the congestion in the network.

Slow Start

In fig. 6, from 0 to about 1.5 seconds, you can observe what is known as "Slow Start" in TCP where the number of packets sent doubles each RTT. Each time the sender receives an ACK, it knows it can send more time, so it exponentially increases the amount sent. In my graph, you can see the doubling from 0 to about 0.4 seconds.

> **Note**: Despite the name, Slow Start is actually fast. It's only slow compared to the original TCP behavior, which sent a full advertised window immediately.

Multiplicative Decrease

At about 1.5 seconds, TCP realizes there is congestion, and the line flattens to a plateau, this is likely because there was a complete timeout and no packets were in transit. This is a severe collapse so TCP resets the congestion threshold to half its value.

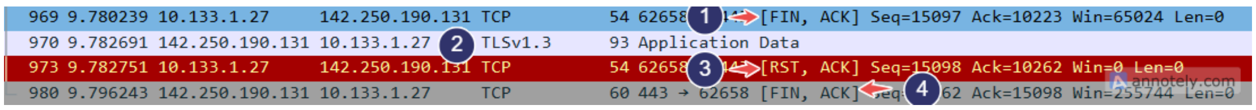
Recovery and Slow Start Re-entry
To make sure it can send data again, TCP probes with single packets and waits for the ACKs. At around 1.1 seconds, again, TCP starts to liberally pick up the connection with slow start once more. This time, it goes till the previous threshold, known as the **Congestion Threshold**, is reached (TCP knows at least that much data can be in transit).

Slow start continues until this threshold is reached, at which point a much more conservative and linear increase begins from about 1.2 to 5 seconds when the session ends.

Overall, this highlights how TCP ensures the network is not flooded through exponential growth to recover, then linear growth to stabilize.

> **Note**: TCP also utilizes Fast Recovery, which is a mechanism that works in conjunction with Fast Retransmit (triggered by three duplicate ACKs) to avoid dropping the Congestion Window all the way to 1 and restarting Slow Start. Instead, Fast Recovery halves the congestion window (Multiplicative Decrease) and immediately jumps back into the Congestion Avoidance (linear increase) phase, using the remaining ACKs to "clock" new transmissions and quickly restore throughput after a non-severe loss event. This phenomenon was not captured in any of my traces.

1.5 Connection Breakdown



```
969 9.780239 10.133.1.27      142.250.190.131 TCP      54 62658 1 44→[FIN, ACK] Seq=15097 Ack=10223 Win=65024 Len=0
970 9.782691 142.250.190.131 10.133.1.27  2 TLSv1.3  93 Application Data
973 9.782751 10.133.1.27      142.250.190.131 TCP      54 62658 3 44→[RST, ACK] Seq=15098 Ack=10262 Win=0 Len=0
980 9.796243 142.250.190.131 10.133.1.27      TCP      60 443 → 62658 [FIN, ACK]←Seq 4 62 Ack=15098 Win=255744 Len=0
```

Figure 7: shows the connection breakdown. Labels 1-4 reference bullets numbered 1-4 referenced below.

Just like the connection was established, it must be terminated gracefully too. Each side needs to close its side independently, ensuring all data has been received before shut down.

The TCP state-transition diagram (very complicatedly) shows the transitions that take place.

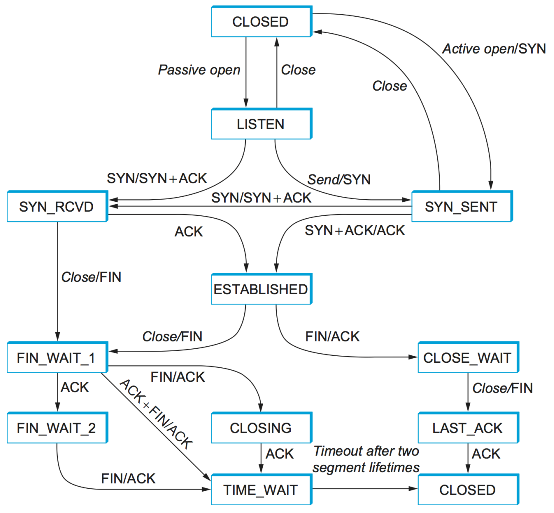


Figure 7.

1. In **packet #969** I as the client closed the youtube tab, so my computer sent the **FIN** flag, signaling that I had no more data to send. According to the TCP transition diagram, the connection moves from the **ESTABLISHED** to the **FIN_WAIT_1** state.
2. This particular stream perfectly highlighted the half-closure process, as even though I ended the connection, the server still sent a **TLS** data packet. However, the server has still not sent its own FIN to close the connection (this is a **half-close**).
3. Since I closed the tab in this case, a **RST (Reset)** flag was sent to the server due to abrupt closure and the connection was torn down. This meant no graceful teardown.
4. Finally, we also see a **FIN, ACK** sent by the server indicating that it is done sending. The ACK also confirms it has received the client's FIN.

What's interesting to note here is that since a RST was already sent, an ACK is not sent by the client to acknowledge the server's FIN-ACK as would typically happen with a TLS termination.

Overall, this trace illustrates how TCP layers multiple mechanisms flow control, congestion control, and reliability to transform the Internet's best-effort delivery into a dependable communication channel. Even when packets were lost or delayed, TCP adapted seamlessly, maintaining both stability and performance.