Tutorial 10 - Transport layer TCP and UDP

Gidon Rosalki

2026-01-08

Notice: If you find any mistakes, please open an issue at `https://github.com/robomarvin1501/notes_networking`

1 The Transport Layer

Recall that the data is encapsulated at each layer going downwards, where the application layer merely sends the data, the transport layer has provided it with an abstraction with which to do that, but adds the TCP/UDP header. Similarly, the network layer has provided the transport layer with its abstraction, and needs to add the IP header to the data, and then the data link layer adds the frame header, and trailer to all the data encapsulated in the previous layers.

We will consider two protocols, UDP, and TCP. These extend layer 3 delivery service between end hosts to a delivery service between application layer processes running on the end hosts. Recall that layer 3 (IP) only provides a "best effort" guarantee for data delivery, but data might arrived corrupted, and packets may not arrive, or arrive out of order.

To distinguish between different sockets created by application processes on our host we allocate them different addresses through the use of *logical ports*. There are 65536 (16 bits) different logical ports that may be provided to applications. In UDP, which has no consistent connection, a socket is therefore identified by the 2-tuple (destination IP, destination port), and in TCP by the 4-tuple (source IP, source port, destination IP, destination port).
When an application is ready to receive data, it allocates a logical port through a system called binding as learned in OS. Note that not all ports are available, 1024 are reserved for specific protocols, for example:

- HTTP - 80
- HTTPS - 443
- SSH - 22
- DNS - 53

2 UDP

UDP does not provide any guarantees to the user regarding the sending of data, aside from allowing multiplexing (different packets to different applications through port numbers), and packet integrity by adding a checksum to the packet. The packet will not be resent if it arrives corrupted, but the receiving application will not be sent it from the transport layer if it did arrive corrupted, it will just be dropped.

There is in fact a source port field in the UDP packet header, but it is optional. Many protocols do in fact add it to the packet, for example DNS.

3 TCP

TCP provides additional features, such as error detection and in order delivery. It makes use of checksums to detect corrupted data, and makes use of an acknowledgement system to request retransmissions in order to ensure reliable delivery. Sequence numbers are used to detect losses, and reorder transmitted data.
The main subject for this tutorial is the congestion control that TCP provides. It tries to ensure that congestion within the network is avoided if possible. Finally, it also provides flow control, to avoid overflowing the recipients buffer.

TCP makes use of sessions, one must explicitly open and close a TCP session. In this session, there will be the local sequence of ACKS. Note that this system provides no guarantees about bandwidth, or delay, since those depend on the network. Additionally, these ACKs are cumulative, similar to GBN as learnt previously.

3.1 TCP Segments

One way we could send data is byte by byte. We would need ACK for every byte sent, so this is inefficient – the overhead of headers is too high. We could instead bundle several bytes into a single segment, though this does leave us with the question about when to send the segment. There is a maximum size for a segment that we can support,

which is the number of bytes in a single layer 2 frame (recall that layer 3 also adds its own header to the data, and our header).

Let us define a couple of concepts:

- MSS - maximum segment size (bytes). This is the maximum amount of application-layer data in the segment.
- MTU - maximal transfer unit (bytes). This is the largest link-layer frame that can be sent in the network.

As stated above, there is a distance between these two sizes, since we need other things to add to our packets besides the TCP segment (i.e., headers). Therefore MSS $<$ MTU.

We have three requirements for reliable transfer:

- Validate that a packet was received as it was sent (e.g., checksum)
- Notify sender that you got its message (e.g., ACK/NACK)
- Deliver the data in-order for the upper layer

There are applications that would suffer from having strictly ordered data like this, like video calls, VOIP, streaming, and so on. If a packet is lost, and then all further receiving is buffered while it waits to receive that packet, then the entire stream/conversation will lag while it waits for that packet. It is better to simply lose the packet, have a drop in quality for a moment, and then continue as normal.

3.2 Connections

3.2.1 Establishing a connection

A client initiates a connection with a server, and once it is initialised, both the client, and the server may use the connection simultaneously (meaning it is in full duplex). One of the initialisation steps is agreeing on the sequence numbers, since in order for them to have meaning, the sequence scheme needs to be established first.

The connection is established through a **three way handshake**.

- The client begins by sending **SYN**, with the field $seq = x$
- The server responds with **SYN-ACK**, with the fields $ack = x + 1, seq = y$
- The client then responds with **ACK**, with the fields $ack = y+1, seq = x+1$
- The client may now begin sending data

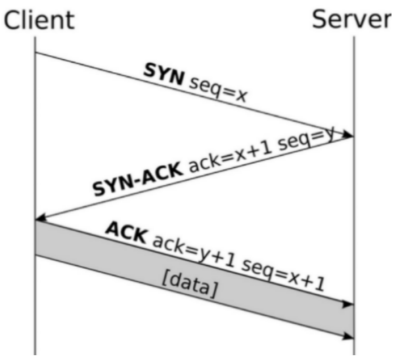

Figure 1: Three way handshake

If the SYN packets are lost, then eventually, no SYN-ACK will arrive, so the client sets a timer waiting for SYN-ACK. If it is not received, then the client will retransmit the SYN. This does leave us with the question of how long should the initial timeout be, since the client has no idea what the round trip time to the server is. Some TCPs use a default of 3 or 6 seconds, which is relatively long in comparison to the actual RTT value.

3.2.2 Closing a TCP connection

When the client wishes to close the connection:

- The client sends FIN to the server, closing its connection
- The server responds with ACK on the FIN
- The server sends a FIN message to the client, closing its side of the connection
- The client responds with ACK to the FIN

3.3 TCP sliding window

Just like in Go Back N, and Selective Repeat, TCP uses a sliding window scheme, where there will be at most N packets in flight. However, unlike GBN, and SR, the window size is **dynamic**, so we need to decide how to set it. We have two main parameters dictating the window size:

- The buffer size of the receiver (Flow control)
- The current "congestion" of the network (Congestion control)

3.3.1 Flow Control

We need to have some knowledge on the available buffer size of the receiver. The receiver sends the amount of buffer it can allocate for this connection in its ACKs, which is denoted as **rwnd** (receiver window). The sender will keep the invariant of

$$\text{Window size} = \text{LastByteSent} - \text{LastByteAcked} \leq rwnd$$

If rwnd = 0, then we want the sender to keep on sending datat to the receiver. These packets will be ACKed by the receiver, and eventually the receiver buffer will free up, and the sender could ramp up the amount of data it sends.

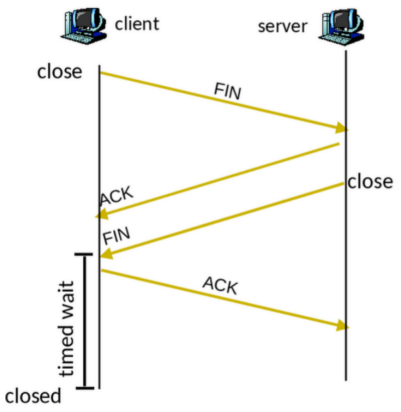

Figure 2: Close the TCP connection

3.3.2 Congestion control

the sender needs to approximate the load the network can sustain at any given moment. However, there is no feedback on congestion from the network. We may instead approximate this by the loss of ACKs, or expired timers. TCP needs to decide when to increase or decrease the congestion window size (ie, the number of packets we believe that the network can sustain).

In TCP Tahoe, and TCP Reno there are two states in which a TCP connection can be:

- Slow Start - Start sending a very small amount of data, and increase it over time
- Congestion Avoidance - Be prepared to handle congestion

There is also TCP Vegas, which is delay based, rather than loss based.

The sender starts with a pre set slow start threshold: ssthresh. This is the parameter that determines when to switch from Slow Start to Congestion Avoidance. The sender also needs to decide when to increase / decrease the congestion window size. SS works as follows:

- Start with a small window size (W = 1 MSS)
- Double the congestion window (cwnd) every RTT
- For every ACK: $W = W + MSS$
- Enter congestion avoidance when $W \geq ssthresh$

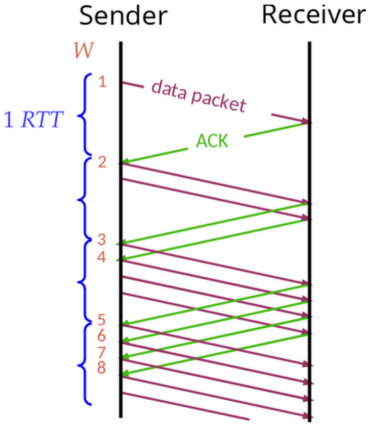

Figure 3: Slow start

Exponential increase stops either when some failure occurs, or when we are sending too much (even without failures), though this requires choosing a threshold.

When in congestion avoidance, for every N ACKs (ACKs for all packets in the window): $W := W + 1$. This results in a small additive increase – adds one MSS every RTT.

Packets should be resent after a timeout. However, timeouts can be relatively long, and waiting for that can increase latency. So instead, we can resend after getting duplicate ACKs. These are an indication that some packets failed, but afterwards some packets succeeded. In practice, we wait for 3 duplicate ACKs, and detecting them requires less time than waiting for the entire timeout. If packet 4 is lost, but 5, 6, 7 are all received, then packet 3 will be ACKed 3 times, and so we know that 4 was lost, but 5, 6, and 7 were all received, so 4 should be retransmitted.

Not every TCP variant implements this Fast Retransmission mechanism. In those that do, after receiving 3 duplicate ACKs, and retransmitting the relevant packet, depending on the TCP protocol (e.g. Reno / Tahoe), the sender may switch to SS, / CA, change the size of ssthresh, and so on. Note that those changes are not a part of fast retransmission, but a reaction to 3 duplicate ACKS.

On a packet loss, timeout, or 3 duplicate ACKs, both Tahoe, and Reno set $ssthresh = \frac{W}{2}$. Other changes depend on wheter a timeout, or 3 duplicate ACKs occurred:

	TCP Tahoe	TCP Reno
Timeout	Enter slow start, and set W = 1	Enter slow start, and set W = 1
3 Duplicate ACKs	Enter slow start, and set W = 1	Set $W = \frac{W}{2}$, and enter congestion avoidance

Table 1: Tahoe vs Reno

One may see an interactive example at this link.

So now we have the two parameters, the receiver buffer size (rwnd), and the congestion window (cwnd). TCP will select the minimal window of the two, meaning that the actual window size is:

$$\text{LastByteSentLastByteACKed} \leq \min\{rwnd, cwnd\}$$

4 Network Address Translation (NAT)

NAT is a mechanism that is optionally used by routers to reduce the number of devices using public IPv4 addresses, instead reusing a pool of private addresses. Recall that there are not many IPv4 addresses in the world, only around 4 billion, so we use NAT such that all the devices behind a single router have only the router's public IPv4 address, and it handles multiplexing the data to the relevant address within the network. This also has a possible privacy improvement.

Private IP addresses were thus constructed, these are reserved addresses that can only be used inside internal (private) LANs, they cannot be used in connections to the internet. For a device with a private IP address to communicate with devices outside of its LAN, its IP address needs to be translated to a public IP address. There are a few private ranges, such as 10.0.0.0 - 10.255.255.255, 172.16.0.0 - 172.31.255.255, and 192.168.0.0 - 192.168.255.255.

All devices in the LAN have private IP addresses. The router that has access to the internet is assigned a single public IP address, which will be used for the entire LAN. The router changes the Source IP and Source Port fields of outgoing packets and Dest IP, Dest Port of incoming packets. Note: this is different from "regular" routing, where routers do not change these fields. The router keeps a translation table, where it converts the translated port, and IP to the original port and IP. This is also matched based off protocol (TCP, UDP).

Consider a LAN with 3 computers, A, B, and C:

- A has the IP 192.168.0.1, and runs two TCP applications communicating with the internet of 1024, and 2000
- B has the IP 192.168.0.2, and runs two UDP applications communicating with the internet of 1000, and 2000
- B has the IP 192.168.0.3, and runs one TCP application communicating with the internet of 1024

The network's assigned public IP is 132.58.38.2.

Below is the NAT table, where we assume that the translated port numbers start at 5001 and increase by 1 when a new row is added to the table.

L4 Protocol	Outside network		Private Network	
	Translated L4 port	Translated IP	Original L4 port	Local IP
TCP	5001	132.58.38.2	1024	192.168.0.1
TCP	5002	132.58.38.2	2000	192.168.0.1
UDP	5003	132.58.38.2	1000	192.168.0.2
UDP	5004	132.58.38.2	2000	192.168.0.2
TCP	5005	132.58.38.2	1024	192.168.0.3

Table 2: NAT table