TCP

TCP, the Transmission Control Protocol, has a primary role of adding reliability to the unreliable lower layers. There are a lot of moving parts to it, and a lot of subtle details. The full data transfer is broken up into smaller units called *segments*. Reliability is achieved by assigning each packet a *sequence number*. As we'll see, sequence numbers are used to help each side know what data has and hasn't arrived and to help keep things in order since sometimes packets can take different routes through the internet and hence take varying amounts of time to arrive. We will start with how a TCP connection is established.

3-way handshake

We will refer to the two parties establishing the connection as *client* and *server*. Things start off with a *passive open*, which is where someone starts up a program on the server and tells it to listen for traffic at a specific port number. When the client wants to connect, they send a message to the server. This is called an *active open*. That first message contains a few things. The first is that the SYN flag in the TCP header is set to true. Here SYN is short for "synchronize sequence numbers". In addition to setting the SYN flag, the client also lets the server know what its initial sequence number is. For reasons we'll talk about later, the initial sequence number is not usually 0 or 1.

The server then sends a TCP message with the SYN and ACK flags set, along with its own initial sequence number. In TCP, each side can send data to the other, and each uses separate sequence numbers for their own data. The ACK flag is short for "acknowledgement". The ACK is the server acknowledging that it got the client's first message. The last part of the handshake is an ACK from the client, acknowledging that they got the message from the server. Now the connection is established, and data transfer can start.

Just remember it as three parts: (1) SYN, (2) SYN-ACK, (3) ACK. It's possible to mathematically prove that a 3-way handshake of this sort is needed in order for both sides to be confident that they are connected. For instance, if we omit the final ACK, there would be no way for the server to be sure if the client is receiving what it sends. As we'll see later, there's a little more that happens in the 3-way handshake, namely that each side can exchange information about their window size and certain TCP options.

Closing a connection

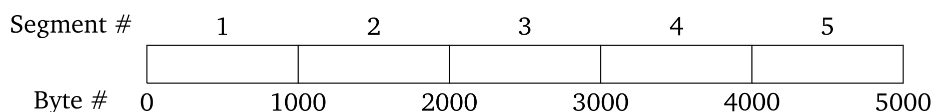
Closing a connection is somewhat similar to the 3-way handshake. Closing starts when one side decides it wants to end the connection. It sends a message with the FIN (finish) flag set. The ACK flag will also be set (it's actually set in every TCP message except in the initial SYN of a 3-way handshake). The other side will send back an ACK to acknowledge receipt. Then it will notify the application layer process that is using the connection, telling it that the other side wants to close the connection. This gives the process time to finish what it is doing. When it's done, the process notifies TCP and then a FIN-ACK is sent to the other side. That side sends one last ACK, and then the connection is closed.

Well, it's almost closed. There is usually a delay of a couple of minutes before reusing the same connection. If the same connection is reused too soon, it's possible that late-arriving packets from the previous connection could cause some confusion in the new connection.

How TCP achieves reliability

As mentioned earlier, TCP breaks data into chunks called segments. In most internet communication, these segments won't exceed 1500 bytes. This limit is due to ethernet (at the Data Link layer), which has a limitation of 1500-byte packets. Some of the 1500 bytes is taken up by headers, so a typical TCP segment size is around 1460 bytes. The limiting size is called the *maximum segment size* or MSS. In the examples below, we will use an MSS of 1000 bytes to keep the math simple.

Below is a picture of 5000 bytes of data, broken into five segments of 1000 bytes each.



Each segment is given a sequence number, which is the *number of the first byte of the segment*. For example, we have the following based on the picture above:

Segment 1 contains bytes 0 to 999. Its sequence number is 0.
Segment 2 contains bytes 1000 to 1999. Its sequence number is 1000.
Segment 3 contains bytes 2000 to 2999. Its sequence number is 2000.
Segment 4 contains bytes 3000 to 3999. Its sequence number is 3000.
Segment 5 contains bytes 4000 to 4999. Its sequence number is 4000.

When a receiver gets data, they send acknowledgements (ACKs) back to let the sender know they got the data. An acknowledgement is indicated by setting the ACK flag to true and by supplying an acknowledgement number. That acknowledgement number is *the number of the next byte the receiver is expecting*. This is a little subtle, and it takes a little time to understand the consequences. In particular, if a receiver sends an acknowledgement number of 3000, that means that they have received all the bytes from 0 to 2999 and the next one they are expecting to get is byte #3000. ACKs are sometimes said to be *cumulative* in that an ACK automatically acknowledges every segment that came before it. Here are a few scenarios using the figure above. The sender and receiver are called Alice and Bob.

1. Suppose Alice sends Segment 1 to Bob. It has sequence number 0 and contains bytes 0 to 999. Bob sends an ACK with acknowledgement number 1000 to acknowledge getting Segment 1. The number 1000 tells Alice that Bob has received bytes 0 to 999 of the overall transmission and is expecting to get byte 1000 next.

2. Suppose Alice next Segment 2 to Bob. It has sequence number 1000 and contains bytes 1000 to 1999. Bob sends an ACK with acknowledgement number 2000 to acknowledge getting Segment 2. The number 2000 tells Alice that Bob has received bytes 0 to 1999 of the overall transmission and is expecting to get byte 2000 next.

3. Suppose Alice sends Segments 1, 2, and 3 to Bob. Though Bob could send ACKs for each one separately, he can also send just a single ACK with number 3000. Since ACKs are cumulative, this is Bob telling Alice that he has received all the bytes from 0 to 2999 and the next byte he is expecting is 3000.

4. Suppose Alice sends Segments 1, 2, 3, 4, but 3 is lost in transmission. Even though Bob got Segment 4, he can't send an ACK for that. The ACK number for it would be 4000, which would be telling Alice that he got bytes 0-3999, which isn't the case. The best Bob can do is to send an ACK with number 2000, indicating that he got the first two segments.

A sender recognizes that a segment is lost in one of two ways:

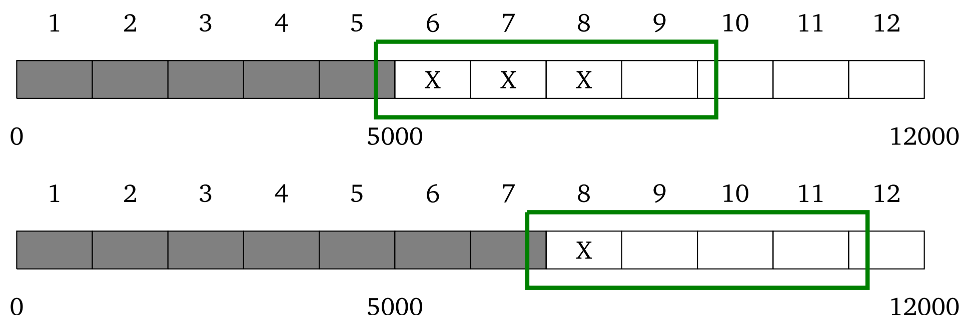
1. Timeout: If an acknowledgement doesn't come back after a certain amount of time, then the sender assumes the segment was lost and resends it. How long should the sender wait? The sender times packet transfers and keeps a running average of how long a round trip takes to the receiver and back. If an ACK doesn't come back within some multiple of the average, say twice the average, then that triggers a timeout.

2. Triple Duplicate ACK/Fast Retransmit: Suppose the sender sends Segments 1, 2, 3, and 4, and 2 is lost. The receiver can let the sender know that 2 was lost by sending four consecutive ACKs, all for the first segment. That's a special cue in TCP that a receiver can use to let the sender know to resend Segment 2. This process is actually faster than waiting for a timeout, which is where the *fast* in the name comes from. The *triple duplicate* name comes from the fact that three of the four ACKs sent are duplicates of the first one.

3. Selective Acknowledgement option: This is an option in TCP. More on this later.

Flow control

Besides reliability, TCP also provides *flow control*, which is a way for a receiver to keep from being overwhelmed by data from a sender. It allows the receiver to slow down or possibly speed up a sender. This is achieved via something called the *sliding window*. It is shown in the figure below.



The figure shows a 12,000-byte data transfer, broken into 12 segments of 1000 bytes each. The gray shaded cells indicate segments that have been sent and ACKed. The cells marked with an X indicated data that has been sent, but not yet ACKed. The unshaded cells indicate data that hasn't yet been sent. The green box around a group of cells indicates the sliding window.

In the top figure, the sliding window is 4000 bytes in size, currently starting at byte 5000, encompassing Segments 6 to 9. That 4000-byte window indicates the range in which the sender is allowed to send data. Currently, they have used up 3000 of the 4000 bytes. They are only allowed to send 1000 bytes more.

Let's suppose that the sender gets an ACK with number 7000 indicating that Segments 6 and 7 were received. This frees up 2000 bytes in the window, and the whole window slides 2 segments (2000 bytes) to the right. Now, 3000 bytes of the 4000-byte window are okay to send, specifically the data from bytes 8000 to 10999 (Segments 9, 10, and 11).

If the receiver gets bogged down processing the data being sent or if they are bogged down with other stuff, they can decrease the window size. They can also increase the window size if they can handle data at a faster rate. Every TCP message has a window size entry in the header, where the a receiver can specify their window size. Window sizes typically range from a few kilobytes up through a few megabytes on ordinary internet connections.

Silly window syndrome Sometimes a receiver can get so bogged down that they end up using really small window sizes. Once a window size gets small enough, sending packets at that size is really inefficient. The reason is that every packet has around 40 bytes of header information, and if the window size is small enough, then those 40 bytes of overhead take up a significant portion of the total packet. For instance, if there were 40 bytes of header and 10 bytes of data, then 80% of the network usage goes to overhead. It's a little like sending money through postal mail: If you're sending $1000, then the 50-cent cost of a stamp only adds a little overhead. But if you instead send a thousand $1-dollar checks, the 50-cent stamp on each is a lot of overhead.

A solution to this is something called *Nagle's Algorithm*. The basic idea of it is the following sequence of if statements.

```
if the sender has at least MSS bytes of data ready and the window size is >= MSS:
    send a full segment
else if there is some currently unACKed data:
    wait for an ACK before sending anything
else:
    send whatever you have, even if it's not a full segment.
```

Nagle's algorithm helps with the silly window syndrome, but sometimes it messes with programs that need to send small, frequent updates. In that case, you can disable it in your OS's settings.

An example of TCP data flow

Suppose Alice and Bob are in the middle of a TCP connection. Bob has already sent bytes 0 through 11999 to Alice, and Alice has already sent bytes 0 to 999 to Bob. Remember that each has their own set of data they are sending to the other, so Alice's data and sequence numbers are separate from Bob's. Let's say the next several steps are shown below.

1. Alice sends bytes 1000–1499 to Bob.
2. Bob sends bytes 12000–12999 to Alice.
3. Alice sends bytes 1500–1999 to Bob.
4. Bob sends bytes 13000–13999 to Alice.
5. Bob sends bytes 14000–14999 to Alice.
6. Alice ACKs Bob's recent transmissions but has no new data to send.
7. Bob sends bytes 15000–15999 to Alice.
8. Bob sends bytes 16000–16999 to Alice.
9. Alice ACKs Bob's recent transmissions but has no new data to send.

Here is what the sequence and ACK numbers would look like. Alice's data is indicated in red to help you keep things separate.

	Alice				Bob
1.	(sending bytes 1000–1499)	⟶	SEQ 1000	ACK 12000	
2.		⟵	SEQ 12000	ACK 1500	(sending bytes 12000-12999)
3.	(sending bytes 1500–1999)	⟶	SEQ 1500	ACK 13000	
4.		⟵	SEQ 13000	ACK 2000	(sending bytes 13000-13999)
5.		⟵	SEQ 14000	ACK 2000	(sending bytes 14000-14999)
6.	(sending no data)	⟶	SEQ 2000	ACK 15000	
7.		⟵	SEQ 15000	ACK 2000	(sending bytes 15000-15999)
8.		⟵	SEQ 16000	ACK 2000	(sending bytes 16000-16999)
9.	(sending no data)	⟶	SEQ 2000	ACK 17000	

In the first step, Alice sends bytes 1000–1499. The sequence number always matches the first byte of the data, so it's 1000. The ACK number is the next byte Alice is expecting from Bob. This is 12000 since we said that Bob already sent bytes 0 to 11999.

In the second step, Bob sends bytes (from his data) 12000–12999. The sequence number is 12000 (first byte of the data), and since he just got Alice's 1000–1499, he sends an ACK with the next number after that, which is 1500.

Skipping down a bit, notice that all of Bob's ACKs from Step 4 on are 2000. This is because Alice doesn't send any more new data after Step 3. So the next byte Bob is expecting stays at 2000.

Below is a screenshot from Wireshark showing a real data transfer when I downloaded a web page.

Length	Info
66	3079 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SA
68	80 → 3079 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=146
54	3079 → 80 [ACK] Seq=1 Ack=1 Win=65536 Len=0
494	GET /specs-grading-iteration-winner/ HTTP/1.1
56	80 → 3079 [ACK] Seq=1 Ack=441 Win=30336 Len=0
1514	80 → 3079 [ACK] Seq=1 Ack=441 Win=30336 Len=1460 [TCP se
1514	80 → 3079 [ACK] Seq=1461 Ack=441 Win=30336 Len=1460 [TCP
1514	80 → 3079 [ACK] Seq=2921 Ack=441 Win=30336 Len=1460 [TCP
54	3079 → 80 [ACK] Seq=441 Ack=4381 Win=65536 Len=0
1514	80 → 3079 [ACK] Seq=4381 Ack=441 Win=30336 Len=1460 [TCP
1514	80 → 3079 [ACK] Seq=5841 Ack=441 Win=30336 Len=1460 [TCP
1018	80 → 3079 [PSH, ACK] Seq=7301 Ack=441 Win=30336 Len=964
1514	80 → 3079 [ACK] Seq=8265 Ack=441 Win=30336 Len=1460 [TCP
1514	80 → 3079 [ACK] Seq=9725 Ack=441 Win=30336 Len=1460 [TCP
1514	80 → 3079 [ACK] Seq=11185 Ack=441 Win=30336 Len=1460 [TC
54	3079 → 80 [ACK] Seq=441 Ack=12645 Win=65536 Len=0

The first three segments are the handshake. Next comes the HTTP request that I made. That was 494 bytes, with 440 bytes of it being data and the rest being the headers from Layers 2 to 4. That was the last time I sent data to the server. Notice that all the remaining segments from me (port 3079) to the server (port 80) all have sequence number 441. This is me sending ACKs to the server. All of the server's ACK numbers are 441, indicating that the next byte they are expecting from me would be 441.

Notice also that the server's sequence numbers are mostly going up by 1460 (from 1 to 1461 to 2921 to 4381, etc.). They are sending me 1460 bytes of data. The 1514 in the length field indicates the total packet size including headers.

TCP Congestion control

Congestion in a network is similar to congestion on roads and highways. It happens when there is a lot of network traffic, with packets taking longer than usual to arrive or possibly getting dropped entirely.

In the 1980s, a phenomenon called *congestion collapse* happened to the internet. Speeds dropped from 32,000 bits per second all the way down to 40 bits per second. The reason for this has to do with how TCP deals with lost and slowly arriving packets. Suppose we start with a network that is pretty busy. This means packets arrive slowly or possibly not at all. This causes timeouts, which means TCP senders will resend the lost packets. This adds more traffic to the network, slowing it down even further. This, of course, triggers more retransmissions, which slows things down further still. This vicious cycle continues until network speeds slow to a crawl.

People had to come up with a solution to this. We will cover the simplest solution here. Over the years, this solution has been modified somewhat, but the basic idea is still the same. The original version was called TCP Tahoe, and the improved versions have names like TCP Reno and TCP Cubic.

Initially, senders have a limit of 1 segment that they are allowed to send. Sometimes this limit might be a little higher, like 5 or 10, but we'll go with 1 in this example. When they send the segment and get an ACK for it, they are allowed to increase their limit to 2 segments.

In fact, the general rule is this: *each ACK received increases the limit by 1.*

Here is how things will evolve:

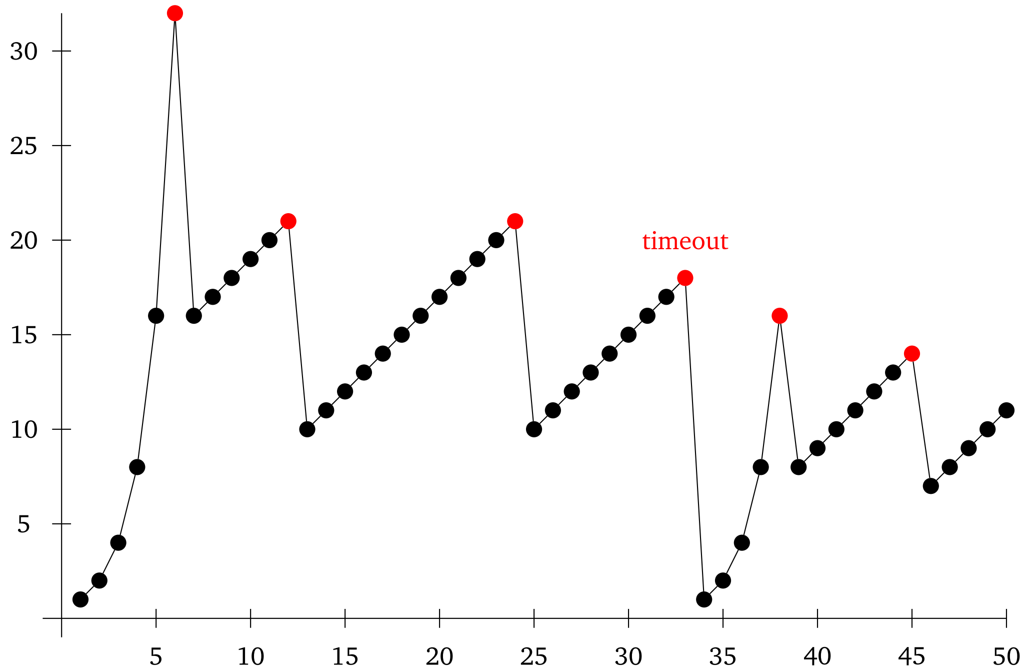
1. Initially, the limit is 1. The sender sends a segment, get an ACK back, and the limit increases to 2.
2. Now that the limit is 2, they send 2 segments. If everything is okay, they get back 2 ACKs, which increases their limit by 2. So the limit is now $2+2=4$.
3. The limit is now 4, and they send 4 segments. If things are still fine, they will get back 4 ACKs, which increases their limit by 4 to $4+4=8$.

4. The limit is 8, they send 8 segments, get back 8 ACKs, and increase their limit to $8 + 8 = 16$.

From here, we can see what happens: The limit increases exponentially as 1, 2, 4, 8, 16, 32, 64, ..., doubling every time. Of course, this can't continue forever. After 20 steps, we would be at a limit of over 1 million segments. So at some point, we will end up overwhelming the network and segments will be dropped.

That segment can be lost in one of two ways: a timeout or a triple duplicate ACK. Of these, a timeout is more severe. That indicates network congestion. If that happens, we drop back down to a limit of 1 and repeat the process. However, if we get a triple duplicate ACK, that indicates we have probably just sent a bit more stuff than an upstream router can handle. In this case we cut the limit in half, and enter a new phase where we increase the limit by a constant amount (let's say by 1) with each round trip time.

Below is a graph showing what this all looks like. The x-axis is time and the y-axis is the number of (full-sized) segments. The dots shown in red indicate packet losses. These are all indicated by triple duplicate ACKs except the one at time 33 which was due to a timeout.



We start out at a limit of 1 and go through the exponential doubling process until we get to 32 segments. At that point, we exceeded the limit of what the network could handle, and we lost a packet. This was detected by a triple duplicate ACK. So we drop down to half, which is 16 segments, and start increasing by 1 segment every RTT (round-trip time). Eventually, we hit the limit again and drop back to half. We then start the additive increase portion again until we hit the limit again.

We then drop down to half again and proceed to increase by 1. However, at time 33, we encountered a packet loss due to timeout. Packet losses indicated by timeouts are more severe than packet losses indicated by triple duplicate ACKs, so we drop all the way back to a limit of 1 and restart the doubling process. This time, we only get up to 16 before a packet loss occurs. The reason this is different from the earlier 32 is that the threshold for packet loss varies depending on how busy the network is. After time 30 or so, the network must have gotten very busy.

This loss was detected by a triple duplicate ACK, and we cut our limit down by half and enter the additive increase phase. We get one more packet loss at time 44 (a triple duplicate ACK) and cut back to half the limit and start increasing by 1 again.

The basic idea of the process is that the exponential doubling phase very quickly finds where the limit is. Once we find it, we drop back to the additive increase phase, which slowly approaches the limit. As mentioned, that limit will vary over the length of the connection due to changing amounts of traffic on the network.

Vocabulary The part of the process where we start at 1 and continuously double the limit is called *slow start*. It seems like a silly name since it grows so quickly, but the slow part of things is that we start at a very low limit. The linear part that follows a triple duplicate ACK is called *congestion avoidance*.

The thing we've been calling the limit is actually properly called the *congestion window*. In TCP there are thus two windows: the congestion window and the sliding window. The congestion window is about protecting the network from congestion. The sliding window is about flow control, protecting a receiver from being overwhelmed with incoming data. Overall, a sender must look at both window sizes and never send more data than is allowed by either. For instance, if the congestion window size is 4000 bytes and the sliding window is 7000 bytes, then 4000 is the most that can be sent.

One important consequence of all of this is that starting a new connection is slow. First, there's the 3-way handshake that involves some back and forth, and then, because of slow start, it takes a few more steps before we can be sending a decent amount of data. In the original version of HTTP, each resource downloaded from a new webpage required a new TCP connection. A big speedup was realized in the next version (HTTP/1.1) where connections could be reused for multiple resources.

The TCP header

Below is a diagram of the TCP header. This is the "envelope" that goes around the data. The data in the header is used for reliability, flow control, and more.

The width of the header in the figure below is 32 bits. In the first line, the source port runs from bits 0 to 15, and the destination port runs from bits 16 to 31. Then in the second line, the sequence number runs from bits 32 to 63. The header will always have the first five rows, giving it a minimum total length of $5 \times 32 = 160$ bits, which is 20 bytes. Options can take up further space.

<table>
<tr><td colspan="3">source port</td><td>destination port</td></tr>
<tr><td colspan="4">sequence number</td></tr>
<tr><td colspan="4">acknowledgement number</td></tr>
<tr><td>hlen</td><td>reserved</td><td>flags</td><td>window size</td></tr>
<tr><td colspan="3">checksum</td><td>urgent pointer</td></tr>
<tr><td colspan="4">options</td></tr>
</table>

Source and destination ports The destination port is used to by the receiving side to know which application to send the data to. The source port is important keeping track of connections, especially if there are multiple connections open with the same host. The port numbers can be used to distinguish different connections with the same host.

The source and destination ports are both 16 bits, which means port numbers can run from 0 to $2^{16}-1$, which is 0 to 65,535.

Sequence and acknowledgement numbers These, as we have seen, are used to keep the data in order as well as to know when segments have been lost. Sequence and ACK numbers are 32 bits, meaning they can run from 0 to $2^{32}-1$, which is from 0 to a little over 4 billion.

Once we get to a sequence number at the max, the numbers will wrap back around to 0. This is not a problem except on very fast (gigabit) connections. We'll talk more about this below.

The initial sequence number is not typically 0 or 1, but rather it's a random value. This is for security, namely to make it harder for a remote attacker to inject data into a TCP connection. In order to inject data, an attacker needs to send data with a sequence number that falls inside the sliding window. Anything else will be ignored. If sequence numbers predictably started at 0, this would be considerably easier. Further, starting at 0 could mean that late-arriving packets from old connections could more easily interfere with the current connection.

If you use Wireshark, it might seem like sequence numbers are starting at 0, but Wireshark by default shows relative sequence numbers. A segment might in actuality have sequence number 2230398882, but Wireshark will show it as 2000 if it's the 2000th byte of the connection. There is an option to show the true sequence number, but it's usually easier to work with the relative number.

hlen This is short for "header length". You may also see it called "data offset". This tells how long the header is, specifically how many rows of 32 bits there are. The purpose is so the receiver knows where the header ends and the data begins, as the options area can take up varying amounts of space. Remember that TCP segments are read and interpreted by computer programs, and those programs need some help in identifying where the header ends and the data begins. The hlen field is 4 bits in size, meaning the header can be up to 15 rows long.

Reserved This portion of the TCP header was set aside by the designers of TCP for future use. The idea is that when you design a protocol, there might be something important that gets left out or that might be needed by people in the future. This field was originally 6 bits, and over the years people did decide to use some of the reserved bits to add some flags, leaving only 3 bits for future use.

Flags Each flag is one bit of data, indicating a true or false value. When the value is 1 (true), we say the flag is set. Here are the original six flags.

- **SYN —** This is used in the famous 3-way handshake in starting a connection. It is short for "synchronize sequence numbers". It is never set except in the first two segments of the handshake.
- **ACK —** This flag is set in every TCP segment except for the initial SYN that starts a connection. Every other TCP segment will have an acknowledgement telling the other side what byte it's expecting next.
- **FIN —** This is used to end a connection. It's one side's way of telling the other that it is done sending data. It's only use is in closing a connection.
- **RST —** RST is short for "reset". The RST flag is often set to indicate some sort of error. If a host receives a TCP segment that it wasn't expecting, it will often send a segment with RST set. For instance, if a web server receives a segment with the ACK flag set without there having first been a connection established via the 3-way handshake, then it will send back an RST. This tells the other side to stop.

  RST's can also be use to abruptly end a connection. The proper way to end a connection is usually with FIN, but RST can be used as well. However, this can cause data loss as it means an immediate end to the connection. The Linux utility `tcpkill` uses RSTs to stop TCP connections.
- **PSH —** This is short for "push". TCP connections are used by application layer programs. TCP, at the transport layer, handles the data transfer and sends the data up to the application layer. However, it doesn't send every byte up to the application as soon as it arrives. That wouldn't be very efficient. Instead it maintains a buffer (storage area) and sends the data up to the application once there's enough to be worth sending.

  However, sometimes the application should get the data right away. That's what the PSH flag is used for, to tell the other side to push the data up to the application. If it's set, it tells the other side to send whatever is in the buffer immediately up to the application.

  A nice example of this are programs like Telnet, SSH, and remote desktop. With these utilities, you are performing operations on a remote computer. You expect your keypresses to show up immediately on screen and in a remote desktop connection, you want the mouse to be responsive. However, these keypresses and mouse updates generate very small segments that might be stored for a long time in the other side's TCP buffer. This would lead to a very laggy user experience. The PSH flag can be used to fix this.
- **URG —** This is short for "urgent". It is very rarely used anymore. In fact, many firewalls automatically zero out the flag if it's set. It was originally meant to be used with the urgent pointer to indicate that a particular part of the data transmission contained urgent data. If the flag was set, the recipient was supposed to use the urgent pointer go to a particular part of the data and do something about it.

The flags above are the original six in TCP. All but the last one are in very common use. Since the publication of the TCP specification in 1981, three more flags have been added: NS, CWR, and ECE, all of which are used for something called explicit congestion notification, which we won't get into here other than to say that it's used in concert with the network layer for routers to indicate to the endpoints of the connection that there is network congestion. Without it, the routers would typically drop the packets when they are congested.

Window size This is the sliding window size used for flow control. It is a 16-bit value, leading to a maximum window size of 65,535 bytes. This is not enough for the modern internet, and there is a TCP option, covered below, that addresses it.

Checksum and urgent pointer The last line of the regular TCP header has a checksum, which is the same as the UDP checksum, and the little-used urgent pointer.

TCP Options Options are, as you might guess, optional extensions to TCP. We will cover a few the most useful ones here.

- **Window Scaling —** As mentioned above, window sizes are limited to 65,535 bytes. If one side wants to use larger window sizes, they can use this option. It sets a scaling factor that tells what to multiply the window size value by. For instance, if the window size is 40,000 and the scaling factor is set to 8, then the actual window size will be 40,000·8 =320,000 bytes. The scaling value is always a power of 2 from 0 to 2^{14}.
- **Maximum Segment Size (MSS) –** This allows one side to tell the other what the largest segment it can handle is. This tells the other side not to send it anything larger. Each side can determine the MSS via a network layer technique called *path MTU discovery* that we will cover later. Because much of the internet runs over ethernet, which has a 1500-byte size limitation, the MSS is usually a number in the 1400s, to leave space for headers, with 1460 being pretty common.
- **SACK —** SACK is short for Selective Acknowledgements. In standard TCP, if a sender sends segments 1, 2, 3, 4, 5, and Segment 3 is lost, the best the receiver can do is acknowledge Segments 1 and 2, even though they also got 4 and 5. The SACK option provides a way to acknowledge multiple separate ranges. The receiver could tell the sender that they got 1 and 2 as well as 4 and 5.
- **Timestamp —** A timestamp is a string representing a date and time. An example is `2020-09-20-18:15:24.948002`. Timestamps are useful in something called PAWS, Protection Against Wrapped Sequence Numbers.

  Recall that once the sequence numbers reach the limit of $2^{32}-1$ (around 4 billion), they wrap back around to 0. This could be a problem on really fast networks. If we're on a multi-gigabit connection, we will actually run through all 4 billion sequence numbers in a few seconds or possibly even a fraction of a second. Typical packet travel times from source to destination can be around 100 ms, so it's quite possible on such a fast connection that two different segments with the same segment number could get to the receiver at around the same time. It wouldn't have any way of knowing which is the first one and which is the second. However, timestamps can be used to tell the two apart.
- **NOP —** This short for "no operation". It's not really an option. Instead it's used for padding, specifically so the options line up nicely in the 32-bit rows of the TCP header.