Introductory Notes on TCP Internet Transport

Daniel Díaz Sánchez
Florina Almenárez Mendoza
Andrés Marín López

Department of Telematics Engineering
Carlos III University of Madrid

Contains leveling reading (sections from 1 to 5) and introduction to advanced TCP concepts (sections from 6 to 9)

Contents

1 Introduction

Network applications require a layer that allows them to transport data over the underlying networks. This layer is called the transport layer, or simply the transport layer.

Applications demand transport layer services that allow:

- Indicating which application on which machine they want to send data to,
- Indicating from which applications on which machines they want to receive data,
- Allowing different user applications to send data to the same server, with the server knowing at all times the origin of the data and how to return the appropriate response,
- Providing a channel free from underlying network errors,
- Sending as much data as the application requires, beyond the limits imposed by the underlying networks,
- Adapting the speed of senders to that of receivers, so that a fast sender does not overwhelm a slow receiver,
- Sending urgent control data, prioritized over other "normal" data.

The transport layer, in turn, uses the services provided by the network layer. The network layer allows the interconnection of physical networks of different types, manufacturers, and organizations into a large logical network. This is the case of the IP protocol in the Internet.

The transport layer demands the following basic services from the network layer:

- Identification of a machine by its network address,
- Sending data from a source machine to a destination machine (identified by their network addresses).

The network layer is also responsible for adapting the data segments received from the transport layer to the APDU size of the traversed network (fragmentation) and, in particular, for preventing network congestion.

It is at the network layer where excess datagrams are discarded when the queues of routers exceed certain limits or overflow.

The transport layer, in turn, is responsible for providing its services while adapting to the state of the network. This means that if the network is congested, usually because there is an excessive volume of data traffic, greater than what the network can handle

2 Justification for the Transport Layer

Since the network layer is capable of sending data between any two machines, one might think that the transport layer is unnecessary, and that the network layer could offer the same services to applications.

However, networks are unreliable, and problems eventually occur. When we send data over a network, they can arrive out of order, duplicated, damaged, or even be lost.

These problems cannot be addressed by the user, as improving the quality of all network layer elements, such as routers, is beyond their control, since these network elements are owned by the network operator.

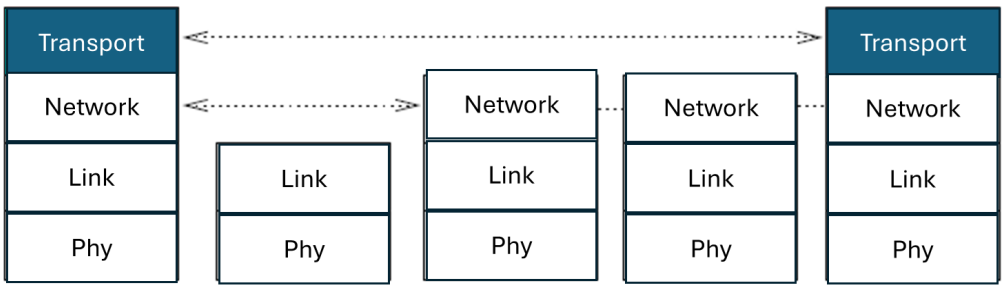

Figure 1: Representation of the link between two machines by stack levels.

While the network layer is present in all networks, the transport layer is only present in the source and destination machines, as shown in Figure 1.

Thus, when the transport layer detects problems in the network, it applies techniques such as retransmission of damaged or lost data, reordering of data, and, in case of severe problems, notifying the application to establish a new connection.

In summary, the transport layer offers users a more reliable data transmission service and is entirely under the control of end users, based on the services provided by the operators of the underlying networks.

3 Transport Addresses

A fundamental concept in communications is multiplexing. Multiplexing consists of sharing a resource to serve multiple independent communications simultaneously.

For example, a road connecting two towns allows several cars to use it simultaneously. Naturally, certain conditions are required: a safe distance (guard), speed limits, etc.

Similarly, the transport layer allows applications to share the network (or networks) to which their machine is directly connected. Each application requests a unique port from the transport layer. Once the transport layer grants the use of that port, all data received on that port will be passed to the application that owns the port.

3.1 Well-known Ports

Each port is linked to the network address of the machine on which it is located. The difference is that for each network address, there are 65,535 different ports. Those ranging from 1 to 1024 are typically used by servers to offer their services over the network; for this reason, they are also known as privileged ports.

In 1981, Jon Postel (`http://www.postel.org/postel.html`), one of the pioneers of the Internet, published RFC790, the first standard that established the list of existing services and the port assigned to each one.

Over time, the number of services grew, and it became necessary to publish frequent updates. The RFC3232 standard granted the standardization body IANA (`http://www.iana.org`) this task. It is currently accessible online from a database maintained by IANA at `http://www.iana.org/assignments/port-numbers`.

3.2 Sockets

The English word *socket* means plug, and in this context, it is used to denote a transport address. Transport addresses are specified by the network address and a port number.

When an application on one machine wants to send data to another, it must specify both parameters to the transport layer.

For example: when my HTML browser wants to connect to the web server of Carlos III University of Madrid, the address is (163.117.136.249,80). When I want to connect to the secure web service, the address is (163.117.136.249,443).

4 Transport Protocols on the Internet

The Internet offers two types of transport services: UDP and TCP. Each application can choose based on its needs the type of transport, and even some services are offered in both types, such as the name service, the clock synchronization service, etc.

The fundamental difference between TCP and UDP is that TCP is a reliable, connection-oriented service, while UDP is an unreliable and connectionless service. Below, we analyze them in more detail.

4.1 UDP: an unreliable and connectionless transport

What do we mean by unreliable in this context? It simply ensures that if the data reaches the other end, the transport layer of the destination machine will know if it arrived damaged or not. This is possible because when UDP constructs the data segments it sends to the other end, it calculates a code (*checksum*) that can be verified upon reception. The code is calculated over a pseudo-header, which is constructed by prefixing the UDP header with the source and destination IP addresses, 8 zero bits, the UDP protocol identifier, and the length of the UDP datagram. The length refers to both the header and the data (see Figure 2).

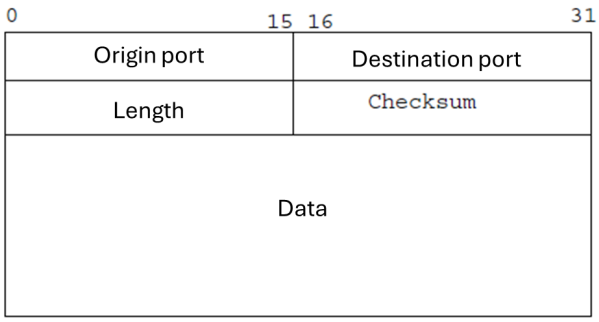

Figure 2: UDP Datagram. Header detail.

However, as we mentioned in Section 2, networks are prone to errors, and when using UDP, the following may occur:

- data may be lost,
- data may arrive out of order (out of sequence),
- duplicate data may arrive.

Since UDP is unreliable, how is it possible that some applications would want to use this transport?

4.1.1 Applications that use UDP

The applications that are interested in using UDP are those that require fast transport (with predictable delays), where occasional data loss is not a major issue, and that can use group transmissions (*multicast*).

Many multimedia applications that send real-time audio and video streams fall into this category. For these applications, occasional loss of data segments is not perceived as a significant degradation in quality by our senses. However, for these applications, it is crucial to have the data on time, meaning that the arrival delay of the data is constant or shows little variation.

These applications typically use a buffer, and when the buffer fills up to a certain point, the data starts being consumed. The buffer size accounts for delay calculation, so if the delay is large, it will take longer to fill the buffer than when the delay is small, but once the buffer is full, the multimedia content can be played.

When we see a satellite connection with a reporter on TV, the delay is very high, but this does not affect the quality. However, when the delay varies significantly (*jitter*), the buffer becomes ineffective. In some cases, the buffer empties frequently, and the application runs out of data, while in others, the buffer fills faster than the application can consume the data.

UDP is a very fast transport protocol since it does not need to check whether the data arrives or is lost, acknowledge data reception, or establish or release connections. For this reason, the delay is lower and more consistent than with TCP, making UDP the preferred transport for multimedia data stream applications.

4.2 TCP: A Reliable Connection-Oriented Transport

TCP is a connection-oriented transport. Connections are established using the datagram service provided by the network layer.

When an application uses TCP, it gets a reliable transport service, ensuring that the data sent is acknowledged by the other end:

- If the data does not reach the other end, TCP retransmits it.
- If the data arrives out of order, TCP reorders it.
- If duplicate data arrives (usually due to retransmissions), TCP discards it.

TCP is the preferred service for most applications that require reliable data transport, such as web (HTTP), email (SMTP, POP3, IMAP), file transfer (FTP, SCP), remote terminal (TELNET, SLOGIN), etc.

Similarly to UDP, TCP is encapsulated within IP datagrams, as shown in Figure 3.

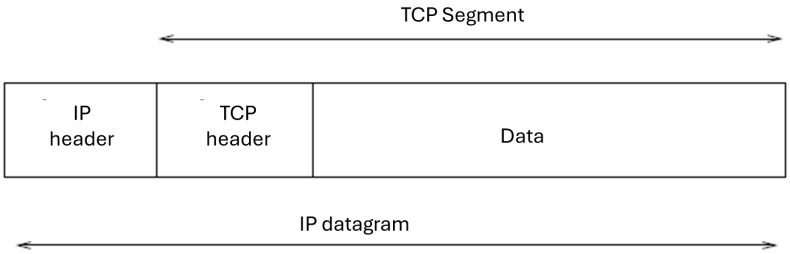

Figure 3: Encapsulation of the TCP segment in an IP datagram.

Figure 4 shows the header of TCP segments. The sequence and acknowledgment numbers are explained in Section 5.5. The "long" field is used to indicate the length of the TCP header, since the options field has a variable length. The one-bit fields following the reserved field are used as flags:

- **URG**: indicates that there is urgent data, with the end indicated by the urgent pointer.
- **ACK**: indicates that the acknowledgment number is valid.
- **PSH**: indicates that the receiver should pass the data to the application as soon as possible.
- **RST**: indicates that the connection should be reset.
- **SYN**: indicates a desire to initiate a connection.
- **FIN**: indicates a desire to close the connection (no more data to send).

Among TCP options, one of the most common is used to negotiate the maximum segment size (MSS) at the start of the connection. There are other options that can be used to: send timestamps, allow window scaling, enable selective acknowledgments (SACKs), etc.

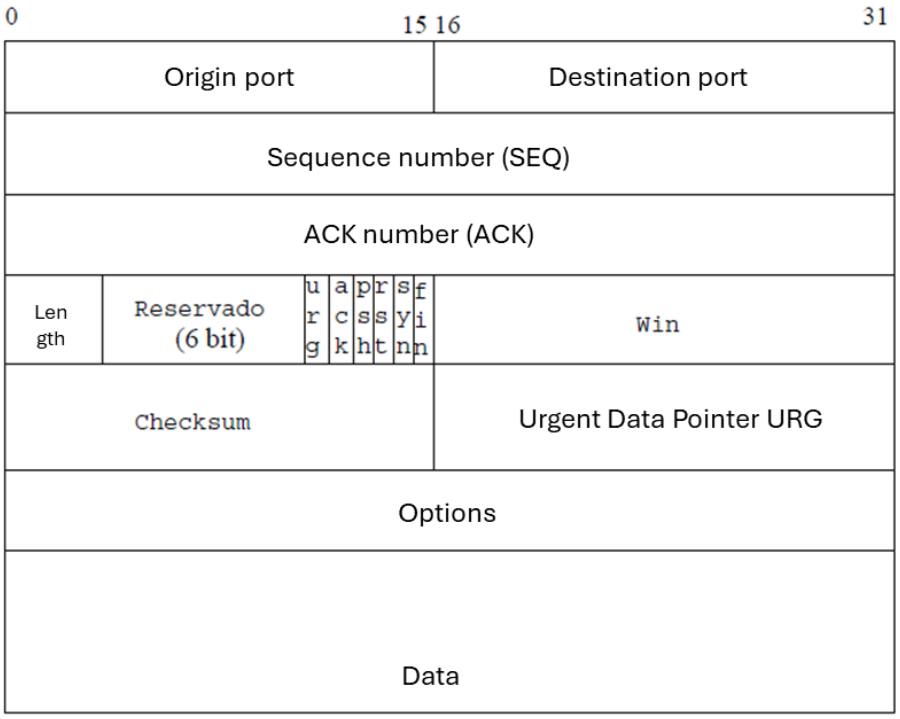

Figure 4: Detail of the TCP segment header.

5 TCP Connections

Every time an application wants to use TCP to send data to another machine, TCP opens a connection with that machine[1]. From then on, all data exchanged by both applications will be sent through the established connection.

TCP connections are established through a mechanism called the three-way handshake. In the first phase, the endpoint initiating the connection sends a connection initiation request. When this request is received, if there is an application listening at the destination address, another acknowledgment request is returned along with the acknowledgment of the received request. When the acknowledgment and the request reach the origin endpoint, it similarly returns the acknowledgment of the received request. The endpoint that initiated the connection is also called the active endpoint, and the other is called the passive endpoint.

[1]Remember that TCP addresses are formed by pairs of machine name and port, so a TCP connection is identified by four numbers: the local machine address, the local port, the destination machine address, and the destination port

In a TCP connection, each segment sends data to the other. These data are encapsulated in what is called segments. Each segment is a datagram composed of the network layer headers, the transport layer headers, and the application data, which are themselves encapsulated in the format required by the local area network link layer. When a TCP connection is established, a parameter called the maximum segment size (MSS) is negotiated. TCP segments are generally of maximum length unless there are not enough available data in the transmission buffer.

5.1 Byte Stream

TCP offers applications a reliable transport of byte streams. This means that if an application on one end of the connection writes 10 bytes, then 20, and then another 50 bytes, the application at the other end cannot determine the size of the original individual writes. It may read 80 bytes at once or four reads of 20 bytes each. The philosophy is that one end injects a stream of bytes, and the other consumes it.

TCP constructs segments from the bytes received from the application, and each time a segment is complete, it sends it. This behavior can be controlled through TCP options. We will see this in more detail in Section 6, dedicated to interactive traffic.

Additionally, these bytes are not interpreted, so they can be used for any purpose and encoding (binary data, ASCII, UTF, etc.).

5.2 Connection Establishment

Figure 5 shows the two modes defined in TCP for performing the three-way handshake. In normal opening mode, the part that sends the first segment with the **SYN** flag activated is called the active endpoint, usually played by the client. The endpoint that receives the **SYN** is the passive endpoint, the server. The client can start sending data as soon as it receives the **SYN** from the server with the correct **ACK**, and the server can send data after receiving the **ACK** from the client.

5.3 Connection Termination

The termination of a TCP connection can also be either normal or simultaneous. Generally, one endpoint, the active one, sends a segment with the **FIN** flag activated, indicating that it has finished sending data. The other endpoint, usually the server, signals this event to the application by sending an end-of-file (*EOF*) message. The passive endpoint can continue sending

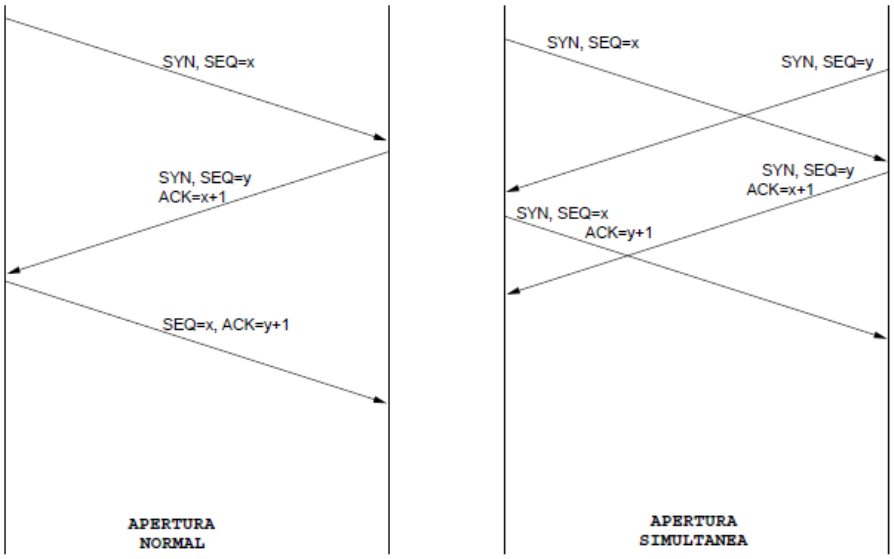

Figure 5: TCP connection establishment. Normal (client-server) and simultaneous mode.

data to the client; in fact, this TCP state is known as *half close*. The connection will not close until the passive side decides to send its corresponding **FIN** and receives the corresponding **ACK**. Active closure is also shown in Figure 6.

5.4 TCP State Diagram

The behavior of TCP connections can be expressed through a state diagram. The diagram considers all possible TCP states and transitions: normal and simultaneous openings and closings. In other words, it shows the states and transitions of TCP for both client and server behaviors.

The *TIME WAIT* state in the diagram is also known as the 2MSL wait state, which is twice the maximum time a segment can wander the network before being discarded. It has two effects:

- It prevents lost segments from being confused with future incarnations of the same connection, and
- If the **FIN** from the other endpoint is received again, the final **ACK** is retransmitted.

The *FIN WAIT 2* state is the state in which the endpoint that initiates the termination (usually the client) remains. In this state, it is mandatory to

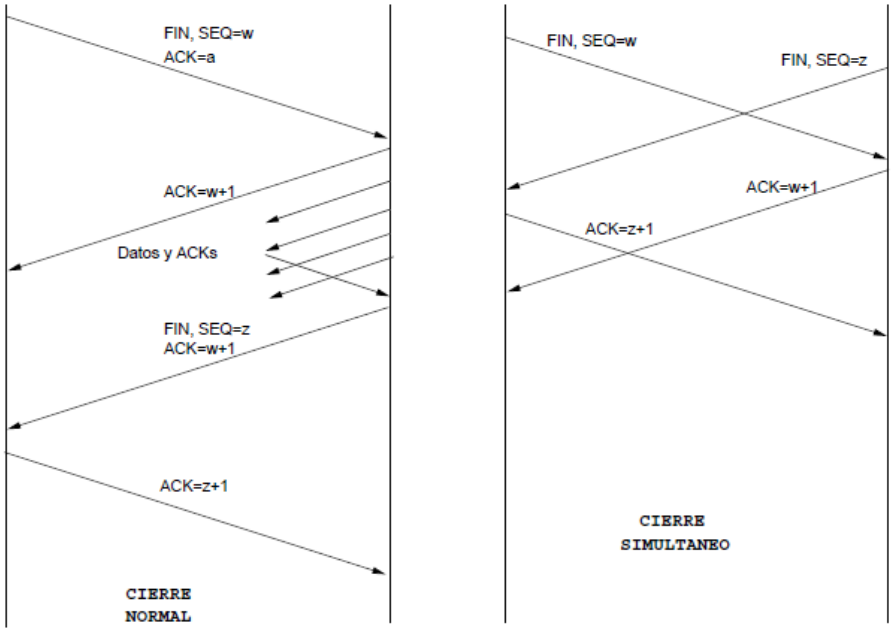

Figure 6: TCP connection termination. Normal (client-server) and simultaneous close.

wait to receive the **FIN** in order to send the final **ACK** and transition to the *TIME WAIT* state. If the **FIN** is not received, the connection could remain blocked, which is why many implementations include a timer to transition to the *CLOSED* state, contrary to the standard.

Practically in any state of the protocol, an RST can be received, which resets the connection to the LISTEN state in the case of a server or CLOSED in the case of a client. This reflects scenarios where a machine (with connections in any state) can be rebooted, and it allows TCP to recover from these situations at the cost of increased protocol complexity.

5.5 Sequence Numbers and Acknowledgment

Suppose the TCP layer of a machine receives two connection requests simultaneously. How can it distinguish between both connection requests? Through two parameters: the network address and source port, and the sequence number.

In each TCP connection, each byte is numbered using a sequence number. Sequence numbers are 32-bit numbers, meaning that up to 4 gigabytes of data could be sent in a connection without repeating a sequence number. In real connections, such large amounts of data are never transmitted.

The sequence number is used to identify each transmitted byte, so when an endpoint wants to acknowledge the correct reception of a data segment, it is sufficient to indicate that it expects to receive the next byte after the last received sequence number. The other endpoint will understand that all previous bytes have been correctly received.

For example, if a segment is received and the sequence number of the last byte is X, the acknowledgment sent to the other endpoint will carry the sequence number X+1.

5.6 Transmission and Reception Buffers

TCP has a buffer (storage memory) for data to be sent and another for received data (see Figure 7). Every time the application sends data to TCP, it places the data in the send buffer. When it can send data to the other endpoint, it sends the first available data from that buffer.

When TCP receives acknowledgment that a set of data (indicated by the ACK number) has successfully arrived at the other endpoint, TCP removes it from the send buffer.

Similarly, every time TCP receives data from the other endpoint, it places the data in the receive buffer. When the application asks TCP for data that has been received from the other endpoint, TCP delivers the data from the buffer that has the lowest sequence number (the data that was received first) and removes it from the receive buffer.

TCP is also responsible for sending acknowledgment to the other endpoint for the received data when it places them in the receive buffer.

If TCP receives data with a sequence number lower than expected, that data is duplicate data, and TCP discards it directly.

If TCP receives data with a sequence number higher than expected, it places it in the receive buffer but does not send acknowledgment[2]. Instead, it waits to receive the missing data, at which point it sends acknowledgment for all received data.

6 Interactive Traffic. Nagle's Algorithm

We can classify network applications based on the type of traffic they generate: those that generate very small segments (*tinygrams*) and those that generate maximum-sized segments. The former are interactive applications (such as *rlogin* or *telnet*), and the latter are bulk applications. The number of segments that travel across the network is roughly balanced, although in

[2]Unless the selective acknowledgment (SACK) option has been negotiated

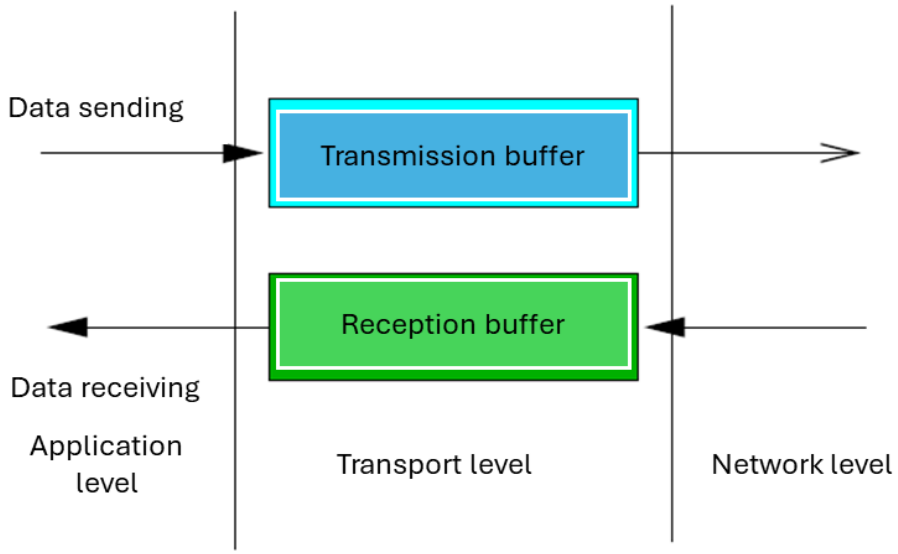

Figure 7: Transport layer buffers

terms of the number of bytes, bulk traffic applications are responsible for about 90% of the transmitted bytes.

In TCP, different treatment is applied depending on the type of traffic. For interactive traffic, two strategies are used: delayed *acks* and Nagle's algorithm.

Delayed *acks* minimize the sending of segments without data (acknowledgment segments). The technique involves using a timer (typically 200 ms) that starts upon receiving a segment. If the timer expires, the corresponding *ack* is sent, but if data becomes available for transmission before that, the timer is canceled, and the data is sent with the *ack* (known as *piggybacking*). Typical examples are character echoes sent by a server in an *rlogin* connection.

Nagle's algorithm also aims to reduce the number of *tinygrams* sent. The algorithm ensures that only one segment can be pending acknowledgment at any given time. If a segment is already pending acknowledgment and additional data is received from the application, the data is queued and sent together once the previous acknowledgment is received.

This algorithm is self-timed, meaning that if network conditions are good, data is sent as it is received from the application, and acknowledgments are received immediately with the character echo. If conditions worsen, transmissions are delayed, and several bytes are sent at once, resulting in delayed echo feedback. This causes the user to slow down their typing, as they see

the echo appear more slowly and are careful to avoid typing errors. As network conditions improve and bytes are sent without queuing, the user sees the echo more quickly and naturally increases their typing speed.

7 Bulk Traffic

Bulk traffic applications (FTP, HTTP, SMTP, etc.) benefit from TCP's flow control: a sliding window that allows them to send segments to the other end as long as the other end has space in the receive buffer. However, TCP combines flow control with congestion control, as explained in Section 7.2.

7.1 Flow Control in TCP: The Receive Window

TCP performs flow control, which is the ability to indicate to a fast sender that the receiver cannot keep up. This situation can occur whenever the receiving application does not read data at the rate it is arriving. It is important to highlight that flow control concerns the speed at which the receiving application consumes data, and while this often happens when one machine is slower than the other, it is not necessarily the case.

For example, it is enough for the application to run at a lower priority to be unable to keep up with the sender's imposed rate.

TCP provides a mechanism called the receive window. The receive window (*win*) informs the other end of how much free space is available to receive new data. This window, therefore, results from subtracting the data pending to be read by the receiving application from the size of the receive buffer.

win is one of the parameters present in the TCP header. That is, all TCP segments include information about the receive window. This way, whenever an endpoint receives a data segment, it knows how much space the other endpoint has left to store new data and takes this into account before sending. In fact, each endpoint must also consider the data it has already transmitted that has not yet been acknowledged when determining whether it can continue transmitting.

That is, before sending, TCP will determine how much data it can send by subtracting the unacknowledged data from *win*. This is how TCP ensures that a fast sender does not overwhelm a slow receiver but adjusts to its pace.

7.2 Congestion Control in TCP: The Congestion Window

Congestion can only be resolved by the network layer, which is the only one with complete visibility of the network. Routers, when congested, receive more traffic (more datagrams) in their input queues than they can process and start discarding datagrams.

TCP, in its design, also takes the network's congestion state into account. Since the network layer does not inform TCP of congestion, TCP can only detect congestion when it observes segments with duplicate acknowledgments or when an acknowledgment does not arrive for a segment, and a retransmission timer expires.

Once TCP detects that there may be congestion in the network, it decreases its transmission rate. The mechanism TCP uses is the congestion window (*cwnd*) and a threshold value for it (*ssthresh*). This congestion window is used in conjunction with the other endpoint's receive window (*win*) to decide if it can send more data segments.

TCP updates the congestion window by combining two algorithms: *slow start* and *congestion avoidance*.

7.3 The Slow Start Algorithm

When TCP establishes a connection, each endpoint initializes the congestion window (*cwnd*) to MSS bytes, where MSS is the maximum segment size negotiated during connection establishment (see Section 5). Every time an endpoint receives a new acknowledgment segment, it increases the value of *cwnd* by MSS bytes.

$$cwnd = MSS \quad \text{(initial)}$$

$$cwnd \leftarrow cwnd + MSS \quad \text{(new ACK received)}$$

When working with segments instead of bytes (common in problems), we will use:

$$cwnd = 1 \quad \text{(initial)}$$

$$cwnd \leftarrow cwnd + 1 \quad \text{(new ACK received)}$$

In this way, the congestion window increases exponentially, as each acknowledgment received for a segment allows the sending of two segments. Therefore, although the start is slow, TCP can quickly send more and more segments as new acknowledgments are received.

Of course, this growth has a limit: the one imposed by flow control (the receive window *win* announced by the other endpoint). In fact, TCP always applies the slow start algorithm unless a more restrictive one is applied, such as Nagle's algorithm used for interactive traffic, or the congestion avoidance algorithm, which we will see in the next section.

7.4 The Congestion Avoidance Algorithm

When TCP starts a connection, it initializes the slow start congestion window threshold (*ssthresh*) to 65,535 bytes. Each time TCP detects congestion, it updates the value of *ssthresh* to half of the minimum between *win* and *cwnd*. If the congestion indication occurred because a retransmission timeout (RTO) expired, the value of *cwnd* is initialized to MSS bytes. If, on the other hand, it occurred due to a duplicate acknowledgment, the value of *cwnd* is not updated.

Each time TCP is about to send a new data segment, it considers the value of *cwnd* similarly to the value of *win*: it can only send if there are fewer unacknowledged data than the minimum of *win* and *cwnd*.

When increasing the value of *cwnd* upon receiving a new acknowledgment, TCP considers whether the current value of *cwnd* is less than *ssthresh*. If so, it uses slow start; otherwise, it increases the value of *cwnd* linearly instead of exponentially.

If a timeout occurs (a segment is sent but its acknowledgment is not received) or a duplicate *ack* is received, TCP identifies this as a congestion problem. In both cases, the slow start threshold is reduced to half of the smaller value between the congestion window and the receive window, always at least 2MSS.

$$ssthresh \leftarrow \frac{1}{2}\min(cwnd, win) \geq 2MSS$$

If the congestion was caused by a timeout, the slow start window is reduced to the initial value (MSS).

When the slow start window exceeds the threshold, the growth becomes linear with each new acknowledgment received:

$$cwnd \leftarrow cwnd + \frac{1}{cwnd}$$

Expressed in bytes:

$$cwnd_B \leftarrow cwnd_B + \frac{MSS \times MSS}{cwnd_B}$$

The combination of the *slow start* and *congestion avoidance* algorithms allows TCP to adapt to network congestion conditions at all times: *slow start* dictates that *cwnd* begins to send data slowly but with exponential growth until it reaches the *ssthresh* threshold, at which point the growth becomes linear. In this way, if a network has several applications using different TCP connections, the network's transmission capacity is dynamically and fairly shared among all of them.

8 Fast Recovery Fast Retransmit

In TCP, the arrival of a duplicate *ACK* indicates network problems. It typically corresponds to a congestion situation where a segment has been lost, and another has arrived out of order. However, if we receive multiple duplicate *ACKs*, this suggests that there have been network issues but that they are no longer as severe (or have disappeared). In addition to the *congestion avoidance* algorithm, TCP uses the *fast recovery fast retransmit* algorithm to resolve these specific issues.

The algorithm works as follows: after receiving 3 duplicate *ACKs*, the lost segment is retransmitted, and congestion control is applied. The algorithm, expressed with parameters in bytes, is as follows:

- 3 duplicate *ACKs* are received.
- The lost segment is retransmitted.
- $ssthresh \leftarrow \frac{cwnd}{2}$, $cwnd \leftarrow ssthresh + 3 \times MSS$.
- If another duplicate *ACK* is received, $cwnd \leftarrow cwnd + MSS$ (and another segment is transmitted if *cwnd* and *win* allow it).
- When a non-duplicate *ACK* is received, $cwnd \leftarrow ssthresh$.

9 TCP Timers

TCP manages several timers:

- To delay the sending of *acks* (*delayed acks*) while waiting for data,
- To retransmit unacknowledged data segments,
- To receive window update indications from the receiver,
- To keep a connection alive in the absence of data.

In this section, we will cover them in more detail.

9.1 Retransmission

The retransmission timer allows a sender to resend data that has not been acknowledged by the receiver. The main challenge lies in estimating the retransmission time to avoid unnecessary retransmissions while also not delaying the resending of potentially lost data segments.

The way to estimate this time is based on measuring the round-trip time (RTT), which is the time from when a segment is sent to when its acknowledgment is received. If we could accurately know this parameter, it would be sufficient to retransmit the segments after this time has elapsed, as the absence of the acknowledgment would indicate that the data or the acknowledgment itself has been lost.

However, since network conditions are constantly changing — data segments may follow different paths, and waiting times in routers vary dynamically — it is necessary to make estimates to approximate the RTT value.

The initial approach proposed in RFC 793 was to measure the instantaneous RTT in a factor called M. Every time an *ack* for a data segment is received, the value of M is updated. The approach involves a low-pass filter:

$$R = \alpha R + (1 - \alpha)M$$

where α takes a value of 0.9. This corresponds to an estimated RTT based on 10

$$RTO = R\beta$$

where β is called the delay variance factor and its value is 2.

Later, in 1988, Jacobson introduced a better approximation for the RTT, allowing it to adapt more quickly to rapid fluctuations in delay. By considering not only the mean but also the delay variance, a much better response is achieved under changing network conditions. This is the mechanism TCP uses today to calculate the RTO. The algorithm is as follows:

$$\begin{aligned} \text{Err} &= M - A \\ A &\leftarrow A + g\text{Err} \\ D &\leftarrow D + h(|\text{Err}| - D) \\ RTO &= A + 4D \end{aligned}$$

In Jacobson's approximation, Err is the difference between the instantaneous value and the mean, A is the estimator of the mean, and D is the estimator of the delay variance. The initialization of the parameters is somewhat special:

Round	A	D	RTO
Initial	A=0	$D = 3$	$RTO = A + 2D$
First	$A = M + 0.5$	$D = A/2$	$RTO = A + 4D$

In the second and subsequent rounds, values for all parameters are available, and the previously explained formulas are applied.

Karn and Partridge improved Jacobson's algorithm for calculating the RTO by addressing the issue of duplicate *acks*. The Karn-Partridge algorithm states that the RTO can only be recalculated, and new measurements of M (instantaneous RTT) can only be taken when a non-duplicate *ack* is received, meaning an *ack* that acknowledges data that has not been previously acknowledged.

9.2 Persistence

In TCP, only segments carrying data are retransmitted, meaning those that consume sequence numbers. Suppose a receiver sends an *ack* segment indicating $win = 0$, and later, after the application has read the data, it sends a window update with a new *ack*. If this *ack*, which announces a new window value greater than 0, is lost, the sender would not be able to send anything, leading to a deadlock.

The persistence timer addresses the problem of lost *acks* with window updates. The timer allows the sender to send a special *ack* segment known as a *window probe*, to which the receiver would respond by resending the lost *ack* with the current window value *win*.

9.2.1 Silly Window Syndrome

Flow control mechanisms based on windows can suffer from what is known as Silly Window Syndrome. This occurs when the receiving application's consumption of data is significantly lower than MSS.

The syndrome leads to a situation where the sent segments contain very little data, adjusting to the slow reading rate of the receiver. The following measures are used to avoid this syndrome:

- The receiver cannot announce a window increase smaller than MSS or half of its buffer.
- The sender does not transmit unless:
  - It can send a full MSS.
  - It can send half of the receiver's buffer.

- It can send the entire sender's buffer, and one of the following two conditions is met:
  - It is not waiting for any *ack*.
  - Nagle's algorithm is disabled.

9.3 Keepalive

There are applications that want to maintain the connection even during long periods of inactivity. For example, imagine a newspaper connected to a news agency: even though no new news is arriving, the newspaper wants to stay connected and receive them as soon as they occur. TCP provides applications with a mechanism called *keepalive*, which allows one endpoint to check if the other endpoint is still connected even though neither has any data to send.