Computer Networks 2 and Introduction to Cybersecurity

Marco Sgobino

March 7, 2022

Contents

Part I

Transmission Control Protocol internals

Chapter 1

Basics

1.1 Brief recap of TCP

TCP is a procotol that has the following properties,

- allows *connection between processes*;
- is *connection-oriented*: before transmitting data, a connection must be established;
- is *reliable*: it assures all segments are correctly delivered through use of `ACK` mechanism, and **at most once**;
- offers a *sliding window* mechanism for congestion control and stream control. This assures read and send buffers are well-optimized in both sender and receiver;
- is *byte-oriented*: the byte stream is fragmented into multiple segments, and composed again after getting to destination.

The logical structure is the following one. There are client and server. The client first authenticates to the server, after that the server opens the connection and client executes send-receive loop. Both server and client create a *socket* s, and the client connect s to IP-srv, port-srv. The communication takes place on s by means of application protocol. It is *reliable*: no losses, no packet loss, packet arrive in the same order as they are sent.

A very simplified pseudo-code for TCP is as following,

```
 int s; s := socket(...);
connect(s, IP-srv, port-srv,...);
...
send (s, msgl, ...);
...
msg2 := receive(s,...);
...
```

The logical structure at server side is quite different. A server creates socket $s1$, chooses a port number to bind to that socket, then it declares willingness to accept connections on $s1$, and finally it awaits for connection requests on $s1$. Server remains on *sleep* until a connection is requested.

```
s1 := socket(...);
bind(s1, portsrvm ...);
listen(s1,...);
s2 := accept(s1,...); // another socket
...
msg1 := receive(s2,...);
...
send(s2,msg2,...);
...
```

In TCP, communication is *bidirectional*, with a pattern that depends on the application protocol.

The send-receive patterns depend on the application itself – browser send-receive sequences are very different from, let's say, an e-mail client send-receive sequence.

1.1.1 TCP Implementation

IP operates between *nodes*. It is *connectionless*, **unreliable**, and is *message-oriented*. The Maximum Transmission Unit size of an IP packet is $MTU = 64KB$. TCP lies on top of IP: to overcome the unreliable aspect of IP, countermeasures should be adopted.

TCP layers communicate between themselves in terms of *segments*. A segment is a *message between TCP layers*, and contains a *TCP header* and – eventually – data *payload*. Payload can either be 0 byte or carry some information useful for application layers. An important property is that *it must be small enough to fit in a single IP packet*, hence IP header + TCP segment size should be no greater than $64KB$.

A segment is thus composed by a IP header, whose payload is a TCP segment. The TCP segment is composed by a TCP header, followed by eventual application data. Usually, IP header size is usually 20 bytes, as well as TCP header that is 20 bytes. The IP datagram can be greater up to $64KB$, with the first $40, 50$ bytes reserved to headers.

Segments can carry portions of data (for instance, in a video stream many segments should be sent to client in order to carry enough information and let application layer reconstruct the video correctly).

In application layer, one application message could correspond to *many segments* in TCP layer, in **both** directions. In fact, at TCP level multiple segments are usually required in order to send a single application-level message.

Each TCP layer represents a connection as (<id>, <state>). The <id> is the <IP-local, port-local, IP-remote, port-remote>, while the <state> refers to the state of the TCP connection. Conceptually there is a single table storing both <id> along with connection <state>.

IP addresses are extracted from the IP header, while port numbers are extracted from TCP header. Packets are thus sorted accordingly. The connection <state> includes information on the *Maximum Segment Size* (MSS), which is the maximum size of the *data part* of a segment that the other part is willing to achieve. The MSS is negotiated upon connection opening. This value is, in practice, identical in both direction and is not arbitrary. In most cases, there are only 2 possible values for historical reasons:

- on different networks (through internet), MSS is 536 bytes (MTU=576), that is the maximum segment size that can fit in the smallest possible packet;
- on same network (ethernet), MSS is 1460 bytes (MTU=1500), which corresponds to ethernet MTU minus the IP header and TCP header.

The core idea is that each segment must be sufficiently small to fit in one packet along the full path, in order to prevent fragmentation.

TODO Add figure that recaps IP header + TCP header.

1.2 Establishing a TCP Connection

At the beginning of a TCP connection, 3 segments are needed, while 2 segments are needed to close it.

DNS -> To forge it, must change IP address to response AS WELL AS copying Transaction ID of request

1.3 TCP Architecture

TCP has many different implementations, depending mostly on chosen OS. Several variants of its components are written, with many of them largely optional.

TCP works in segments. Suppose to be at the application level. Execution flow at application level works independently and unpredictably with respect to the TCP-level flow. When an application sends something, multiple TCP packages must be exchanged. The sequence of bytes will be copied to a buffer (sliding window) and the `send()` function is, for example, invoked – time in which a segment is sent is *unpredictable*, *unrepeatable*.

Many events can provoke a transmission:

- application invokes `send()`;
- application invokes `receive()`;
- TCP layer *receives a segment*;
- a *timeout* occurs;

Each of the above will trigger a transmission either immediately or *withing a maximum predefined time* (in the case of a timeout, for instance). When a TCP layer is touched from above or below, **it reacts by transmitting a segment**.

Transmission may occur *even if there is no useful data to transmit* (e.g. the transmission buffer is empty) - in that case, a segment will only carry the header.

Transmission may transmit a varying number of bytes, ranging from empty up to bytes number *larger than Maximum Segment Size* (536 in Internet network, 1460 for same Ethernet network). In that case, the payload must be *fragmented* before delivery, and multiple segments are delivered in sequence. It happens that 2 or 3 segments are initially delivered before the acknowledgement.

CPU Cycle	$0.3ns$
Main Memory Access (DRAM)	$120ns$
SSD	50-150 μs
HHD	$10ms$
Internet SF to NY	$40ms$

Table 1.1: Some interesting metrics.

TCP starts sending slowly, increasing the exchange speed. Acceleration depends on the timing of *received* packets, by looking at the metrics of confirmation packets from receiver.

1.3.1 The `send()` system call

`send()` is the system call that processes call to send data through the network. The `send` function passes a memory buffer contained in application space (address, legth), copies bytes from transmission memory buffer (TX-buffer) in application space to memory byffer in TCP layer.

```
public void write(byte[] b)
  throws IOException
```

Send is first invoked by application level. Buffer at application layer is then copied to the TCP transmission buffer, to be sent immediately or later. New invokations of `send()` will copy data in TCP buffer **after** the data that is already present.

1.3.2 The `receive()` system call

When data reaches the receiver, the data is copied into a receiving buffer (RX-buffer). The receiver buffer is flushed only when the application invokes `receive()`. The function `receive()` copies receiver buffer to application buffer, receive copies without exceeding the size of the buffer (it returns how many bytes are copied). Receive takes as argument also the number of bytes to get from the TCP receiving buffer.

There are three possible cases:

- if the receive buffer is empty, the application is suspended and the process is put to sleep;
- if more than `length` bytes are available, a `length` number of bytes is fetched;
- if less than `length` bytes are available, all available bytes are copied.

1.3.3 Sending N bytes

Suppose to send N bytes with K consecutive `send()` invocations. How many segments will be exchanged? How many transmission events?

As a first approximation, the number of transmitted segments will roughly be

$$num = \frac{N}{MSS} + 1,$$

with the last segment +1 smaller than the previous ones. Things, however, can be much more complex due to packet loss and retransmissions.

Recall that the number per se is not predictable and not repeatable (TCP is byte-oriented).

1.4 Sequence numbers

Since IP is *unreliable* (packets can be lost, duplicated, or delivered in different order from which they were sent), each data byte is implicitly identified by a 32 bit **sequence number**. The association is implicit – the sender applies a sequence number to a segment, and the receiver uses it to reconstruct the actual order of segments.

There are several sequence number. `snd.User` is the variable carrying the value of the next byte the **application** will send. `snd.Next` is the variable carrying the value of the next byte that the **TCP layer** will transmit – its value is contained in the TCP header (initial byte of the sequence, of course).

The sequence number of application level must be computed from other information.

In short,

- `snd.Next` is the boundary between transmitted data and yet-to-transmit data;
- `snd.User` is the boundary between in TX-buffer data (data sent by application) and not-yet-associated bytes.

From the receiver's point of view, there is a variable, `rcv.Next`, that is the boundary between content of RX-buffer and not-yet-received data (right boundary of the data currently in buffer). Only packets having expected sequence number are collected and put in the buffer – however, if some packet has new parts of information and sequence numbers not collected, they will be collected and duplicated bytes are thrown away. There may be two reasons for packet duplications: IP duplication, and TCP sender retransmission because it thought it was lost. TCP stores packets in buffer until acknowledgement has been received, since they could be retransmitted in the immediate future.

- `rcv.Next` points at the next byte that is not yet being received;
- `rcv.User` points to the next byte to be received by the application. After `receive()` invokation by the application, all delivered to the application bytes can now be deleted, since there are no retransmission needs.

An important detail is that **sequence numbers are 32-bit integers**. Therefore, there may be a *wrap-around* (overflow-like behavior). TCP should handle these comparisons accordingly.

1.5 Handling duplicates and loss

IP is an unreliable protocol. This means that *packets can be loss*. Necessary mechanisms are

- **retransmission**;
- **acknowledgement**, which is a kind of *notification of receipt*, in order to be sure that the receiver has received all the data we sent them.

Acknowledgements is a mechanism that assures receipt of a message by notifications. Every header of a segment contains the sequence number of `snd.Next`. Every segment also *carries an information regarding the bytes that are received*: the **acknowledgement number**, which tracks the state of the RX-buffer by the pointer `rcv.Next`. This way, having both sequence number and acknowledgement number, one can successfully track the state of a TCP connection. When sending a TCP segment, both sequence number and acknowledgement number are sent, so that the receiver can reconstruct the state of the sender RX-buffer.

A fifth variable is needed: `snd.Ack`, which *points to the byte in TX-buffer that are both transmitted and acknowledged*. This variable is only increased upon receiving data (for instance, upon receiving a sequence with a greater ACK number from the sender). Data before `snd.Ack` pointer can safely be discarded. Acknowledgements are crucial to a TCP connection, in order to guarantee **reliability** of a connection (TCP is connection-oriented). Therefore, transmission is always necessary even if TX-buffer is empty. That case, no payload will be transmitted, only information in header is sent (increasing acknowledgement number).

Data bytes between pointers `snd.Ack` and `snd.Next` is said to be *in flight* data. These bytes have been transmitted but not yet acknowledged.

1.6 Delayed Acknowledgement

Delayed Acknowledgement is a famous TCP algorithm. It is pretty straightforward:

> Upon receiving a segment, if delayed-ack timer T has previously been set, transmit immediately. Else, set the delayed-ack timer T to a value.

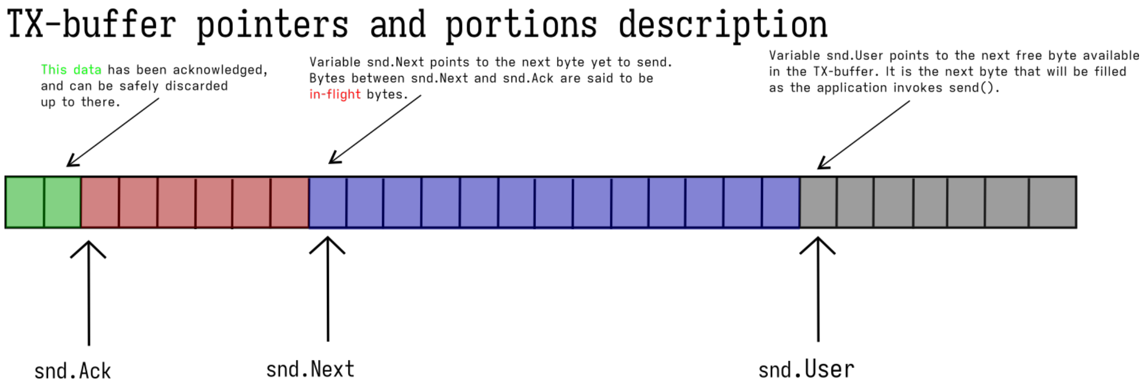

Figure 1.1:

Delayed-ack timer value depends on the operating system:

- RFC suggests $T = 500ms$;
- Windows has $T = 200ms$;
- Linux in general has $T = 40ms$;
- RHEL sets $T = 4ms$.

If there is not much data to transmit, it will be likely that the timer T will expire. If the other part is sending a lot of data, the contrary will be more likely to occur, breaking the awaiting.

The core idea is to minimize the number of segments to be sent. In fact, a delay time T assured to save sending some segments, a feature that historically was of a crucial importance.

1.7 Retransmissions

The connection state includes three more variables:

- a **retransmission timer**;
- a *variable that describes the duration of retransmission timer*, the **RTO**;
- a **retransmission counter**.

The algorithm is as follows:

> Upon transmitting segment S, the counter is cleared and set to 0, with the timer set to the RTO value. Upon receiving an ACK, `snd.Ack` is set to the maximum value between `snd.Ack` and `segment.Ack`. If `snd.Ack == snd.Next` the timer is switched off.
>
> When timer expires, the counter is incremented and if the counter has not yet reached a `MAX_COUNT` value, the segment is retransmitted. However, this time the timer value is set to RTO but `RTO = 2 * RTO`. Only in-flight data should be retransmitted (and all of it after timer expires). Basically, data for which we are sure that it has been received, should not be retransmitted in case the timer expires.
>
> When counter reached `MAX_COUNT` value, connection is closed.
>
> A particular condition is when an ACK arrives for a portion of the data that has been sent. In that case, the timer resets (the receiver has responded) and the sender awaits ACK for missing segments.

Windows closes connection after 5 failed attempts, Linux after 15.

It is simple to realise that there may be a lot of unnecessary retransmissions. The timer can, for instance, run out too soon for the acknowledgement to reach the sender. Unnecessary retransmissions are a waste of resources, a fundamental problem. The reason could be one of those:

- segments are lost;
- segments wich ACK are lost;
- RTO was set to a too small value.

Thus, RTO should be set to an appropriate value, since a too short value leads to high overhead and possibly many unnecessary retransmissions, while a too long value results in high latency and possibly a slow connection. RTO should be set **dinamically**. The RTO should be slightly greater than the *RTT* (round-trip-time), and it is an idea from Jacobson algorithm. This is of a crucial importance for TCP to work. Initial RTO value is heuristical, and varies from one OS to another. Linux and Window start from same value, macOS use a different value, and so on.

1.8 Multiple default gateways

More default gateways could be added to a single node. Reasons to add more than one default gateway all boil up to failure avoidance. To know whether a gateway has stopped working, a heuristic TCP algorithm tries gateway failure detection:

> if the number of retransmissions is greater than a `MAX_COUNT` divided by 2 number, the *connection* changes its default gateway. Moreover, if the number of connection that changed default gateway is greater than the number of open connections divided by 4, the *IP layer* changes the default gateway. This last feature speeds up reconfiguration of early connections that still have to make some retransmission attemps.

Basically, each connection can autonomously choose its own gateway, but the IP layer can force any – new or already present – connection to use a different default gateway.

1.9 Gaps in RX-buffer

Suppose that 4 segments are sent, but the second one has been lost. In this case, the receiver has got all segments except the second one, however it has no method to inform the sender to retransmit only the second segment. Sending ACK for only the first segment would result in unnecessary retransmission of segments 3 and 4. The receive buffer could end up having some **gaps**, missing bytes that are supposed to be received. The solutions are *Selective Acknowledgements* (SACK), a special kind of acknowledgements that carry both *left and right sequence number edges* of each out-of-order **block** in RX-buffer, this way avoiding unnecessary retransmissions. SACK protocol must be supported by both members of a connection.

SACK is very convenient, since many segments can be lost when a sender tries to deliver dozens of segments at once. This way, TCP can achieve *efficient handling* of the connection.

Of course, out-of-order segments could still lead to gaps in RX-buffer. In this case, unnecessary retransmissions are unavoidable when an out-of-order segment reaches the receiver too late (however, SACK reduces a lot the burden to the sender since only missing segments should selectively be sent).

1.10 Operating System TCP interrupts

Upon packet arrival, a *system interrupt* is sent.

Chapter 2

Flow and Congestion Control

Whenever TCP decides to transmit, there are three possible cases:

- an empty segment is sent when there is no payload;
- transmits a single segment if payload data is smaller than MSS;
- transmits multiple segments if paylod size exceeds MSS.

TCP initially starts slowly, then *increases its transfer speed* according to the rate in which acknowledgements are received. The overall interaction is bidirectional and quite complicated, since TCP implementation at sender's side tries to adapt to both connection properties and receiver's side properties. As a general rule, the number of in-flight bytes is always lower than an upper bound that is dynamically updated – this way, TCP can *adapt* to the peculiar characteristics of the connection and receiver's speed. Naturally speaking, a sender should not send more segments than how many can be managed by the receiver.

There are two different bounds and algorithms, the **flow control** and **congestion control**. The first algorithm constructs a bound in such a way that the capacity of the receiver is always respected. Let an extremely fast computer send data faster than a receiving, slow, computer. Slower computer has not enough speed to collect all data that has been sent – the flow control algorithm lets the sender speed adapt to the receiver speed by means of a *send window*. The second algorithm, the congestion control, constructs a state variable called *congestion window* that . The number of in-flight bytes *must be slower than both send window and congestion window multiplied by MSS*, so that

$$\text{in-flight} \leq min(\text{sndWin}, \text{congWin} \cdot \text{MSS}).$$

Basically, if the transmission buffer is full of data, a number of in-flight bytes equal to the minumum of both quantities should be sent, otherwise just send `snd.User - snd.Ack` bytes (those still to send).

2.1 Managing memory buffers

Buffers cannot increase indefinitely: boundaries must be set in order to assure system stability. In Linux kernel, buffer size are set upon compilation. Application should be suspended in case data cannot be pushed to TX-buffer (the case when the buffer is full). When the RX-buffer is full, packets have to be discarded since they cannot be collected. Flow control will act to prevent this situation by letting the sender know how much free space is available to the RX-buffer.

2.2 Flow control

Flow control is keeping a fast transmitter from overrunning a slower receiver.

Since application sends much data and faster than ACK arrive, the TX-buffer could end up being filled up. At receiver's side, application invokes `receive()` much slower than incoming packages speed. This way, the RX-buffer could fill up as well. To solve this issues, application invokes `send()` only when TX-buffer is full, hence it is put to sleep (blocked) until TCP gets proper ACK and advances `snd.Ack` (when it has more free space). TCP sender keeps track of free space in other end's RX-buffer by looking at a variable in header that informs it of how much free space is available, so that it can predict how much free space there is. When it guesses that receiver's RX-buffer could be full and have no space available, it stops transmission and suspends the application. If the maximum window corresponds to the size of a single segment, the protocol is called **stop-and-wait**. Larger windows enable pipelining of multiple segments in a row, enabling far more efficient usage of the connection.

Flow control algorithm adopts the concept of **sliding window**. Each segment header has a `WindowSize` field in the header that contains *how many free bytes are in the RX-buffer*. WindowSize field basically is the amount of free space available in RX-buffer. At sending side, `snd.winSize` contains the *number of free bytes in RX-buffer of the other side*. Initially it is set to the receiver's buffer size, and it is updated dynamically upon sending a segment. If `S.Ack` is greater or equal to `snd.Ack`, then the variable `snd.winSize` is set to `S.windowSize`. Basically, the number of in-flight bytes must never exceed the number of bytes in `snd.winSize` that are available in the RX-buffer. The goal of TCP is to reach a number of in-flight bytes that is as close as possible to the `snd.winSize` number of bytes, in order to optimize the connection efficiently.

2.2.1 Example: M * MSS window size and no transmission errors during flow