

Computer Networks 2 and Introduction to Cybersecurity

Marco Sgobino

March 3, 2022

Contents

I	Low-Level Network Protocols	2
1	TCP	3
1.1	Brief recap of TCP	3
1.1.1	TCP Implementation	4
1.2	Establishing a TCP Connection	5
1.3	TCP Architecture	6
1.3.1	The <code>send()</code> system call	7
1.3.2	The <code>receive()</code> system call	7
1.3.3	Sending N bytes	7
1.4	Sequence numbers	8

Part I

Low-Level Network Protocols

Chapter 1

TCP

1.1 Brief recap of TCP

TCP is a protocol 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, msg1, ...);
```

```
...
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, portsrv ...);
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 μ s
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, `length`), 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 invocations 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 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.