Computer Nerworks. Unit 4 TCP

Notes of the subject *Xarxes de Computadors, Facultat Informàtica de Barcelona, FIB*

Llorenç Cerdà-Alabern

April 29, 2021

Contents

4 Unit 4: TCP

4.1 Transport layer: UDP/TCP

- UDP *User Datagram Protocol*:
  - Connectionless, no reliable
- TCP *Transmission Control Protocol*:
  - Connection oriented, reliable

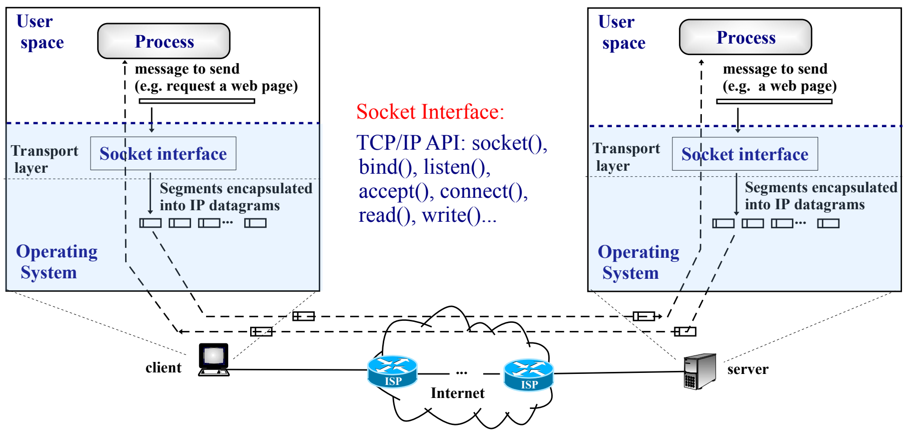

User space
Process
message to send (e.g. request a web page)
Transport layer
Socket interface
Segments encapsulated into IP datagrams
Operating System
Socket Interface:
TCP/IP API: socket(), bind(), listen(), accept(), connect(), read(), write()...
User space
Process
message to send (e.g. a web page)
Transport layer
Socket interface
Segments encapsulated into IP datagrams
Operating System
client
ISP
...
ISP
Internet
server

Figure 1: Client Server Paradigm: Sockets opened by the processes are identified by *port numbers*. Client-socket ports are assigned by the operating system. They are larger than 1023 and are called *ephemeral*. Server-socket ports are assigned by the application and are called *well-known* because they are standardized and identify the service. Source and destination port numbers are sent in the TCP/UDP headers.

4.2 UPD Protocol RFC768

- **Service**: same as IP:
  - Non reliable
  - No error recovery
  - No ack
  - Connectionless

- **Applications** that use UDP
  - short messages e.g. DHCP, DNS, RIP
  - Real time e.g. Voice over IP

4.2.1 UDP Header RFC768

- Fixed size of **8 bytes**
- **checksum**: computed using header, pseudo-header and payload
- Drawback of the pseudo-header: **NAT-PAT** must update the checksum

```
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|            Length             |           Checksum            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2: The UDP header has fixed size (8 bytes) and consists of the fields (bits): Source port(16), Destination port(16), Length(16) and checksum(16).

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                       Source Address                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Destination Address                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Zero      |   Protocol    |          UDP length           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 3: The UDP pseudo-header is added to the UDP heade to compute the checksum. Is consists of the fields: Source IP address, Destination IP address, one byte set to zero, the protocol field and the UDP length.

4.2.2 Practical example

Observe the traffic generated by writing 10 times in a UDP and a TCP sockets.

Writing 10 times in a UDP socket (ruby)

```ruby
#!/usr/bin/ruby -w

require 'socket'

server = UDPSocket.new
server.connect("192.168.11.36", 2000)
for i in 1..10 do
  puts "sending " + i.to_s
  server.send("Hello " + i.to_s, 0) # send
end
```

Minimal UDP server (ruby)

```ruby
#!/usr/bin/ruby

require 'socket'

server = UDPSocket.new # Server
server.bind("0.0.0.0", 2000) # bind to localhost port 2000
for i in 1..10 do
  puts server.recvfrom(1000)[0] # read (max 1000 bytes)
end
server.close # close socket
```

Writing 10 times in a TCP socket (ruby)

```ruby
#!/usr/bin/ruby -w

require 'socket'

server = TCPSocket.new('192.168.11.36', 2000)
for i in 1..10 do
  puts "sending " + i.to_s
  server.puts "Hello " + i.to_s # send
end
server.close # close socket
```

Minimal TCP server (ruby)

```ruby
#!/usr/bin/ruby -w

require 'socket'

server = TCPServer.new 2000 # bind to port 2000
client = server.accept # Wait for a client to connect
while line = client.gets # read message
  puts line
end
client.close # close socket
```

4.3 Automatic Repeat reQuest (ARQ) RFC3366

4.3.1 What is ARQ?

Communication channel between endpoints designed for **reliability** and **efficiency**. Typically involves:

- **Error detection**: detect corrupted or missing PDUs
- **Error recovery**: retransmit erroneous PDUs
- **Flow control**: the sender must not transmit faster than the receiver can read

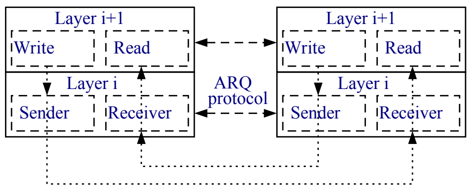


Figure 4: Generic OSI layers involved in an ARQ protocol.

4.3.2 ARQ Ingredients

- **Connection oriented**
- Tx/Rx (Transmission/Reception) **buffers**
- Acknowledgments (**ack**)
- Acks can be **piggybacked**
- Retransmission Timeout, **RTO**
- **Sequence Numbers**

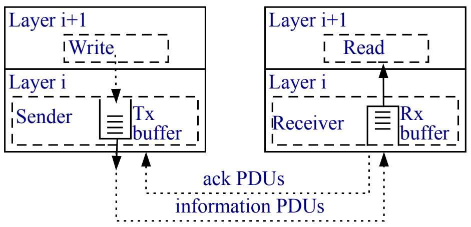


Figure 5: ARQ protocol flow of information and ack PDUs.

4.3.3 Basic ARQ Protocols

- Stop & Wait
- Go Back N
- Selective Retransmission

4.3.4 ARQ evaluation model

- evaluate **one direction**
- there is always information ready to send
- line of **distance** D [m] and bitrate v_t [bps]
- **propagation speed** of v_p [m/s]: **propagation delay** $t_p = D/v_p$
- Speed of light: $c \approx 3\ 10^8$ [m/s]
- Information PDUs (I_k) / ack PDUs (A_k)
- I_k, A_k of L_I, L_A bits
- Tx times $t_t = L_I/v_t$, $t_a = L_A/v_t$

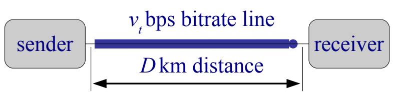


Figure 6: Model consisting of a single link for the ARQ evaluation.

4.3.5 Stop & Wait

1. When the **sender** is ready: (i) allows writing from upper layer, (ii) build I_k and pass it down for Tx
2. When I_k arrives to the **receiver**: (i) pass I_k to upper layer, (ii) generate A_k and pass it down for Tx
3. When A_k arrives to the **sender**, goto 1

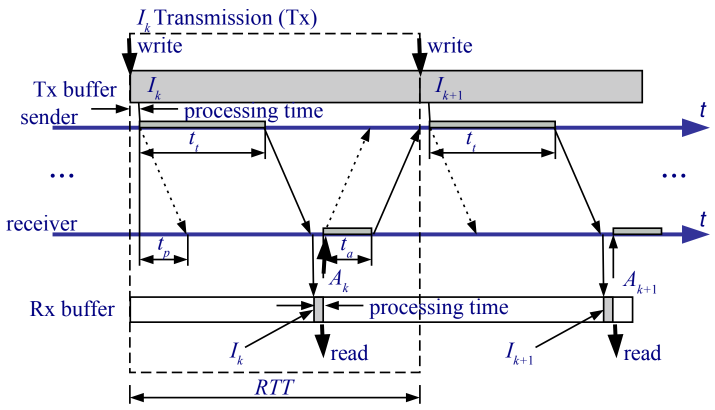


Figure 7: Detailed time diagram of stop-and wait.

Acronyms:
RTT: Round Trip Time

4.3.6 Stop & Wait simplified diagram

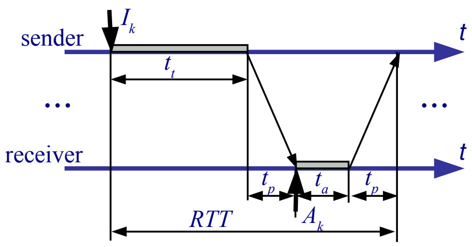


Figure 8: Simplified time diagram of stop-and wait. We only depict PDU's starting and end transmission times, and an arrow that shows the arrival time of the last PDU's bit after a transmission delay.

4.3.7 Stop & Wait Retransmission

- Retransmission timeout (**RTO**) is started upon each Tx
- If I_k does not arrive, or arrives with errors, **no ack** is sent
- When RTO expires, the sender **ReTx** (retransmits) I_k

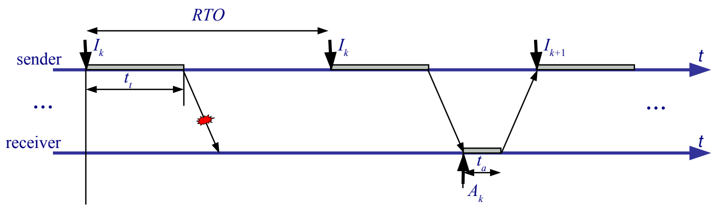


Figure 9: Time diagram of stop-and wait with errors. Upon expiring RTO I_k is retransmitted.

Acronyms:	
Tx:	Transmission
Rx:	Reception
ReTx:	Re-transmission

4.3.8 Why sequence numbers are needed?

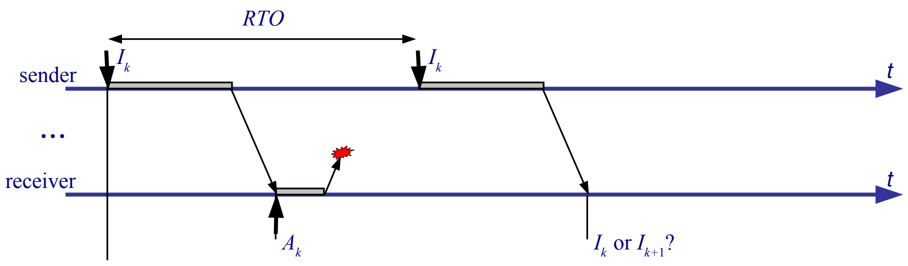


Figure 10: Time diagram showing the need of sequence numbers in I_k when an ack is lost.

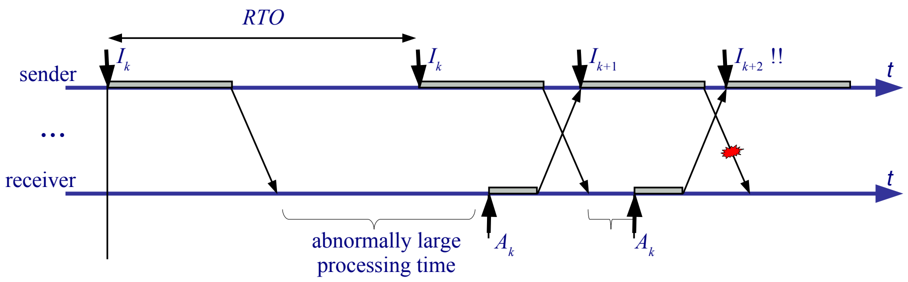


Figure 11: Time diagram showing the need of sequence numbers in A_k when an ack is retransmitted after a long delay, and duplicated ack are received.

Acronyms:	
PDU:	Protocol Data Unit
I_k:	Information PDU number k
A_k:	Ack PDU confirming I_k
RTO:	Retransmission Timeout

4.3.9 Evaluation

- Given a line with bitrate v_t [bps]:
- **Throughput** (*velocidad efectiva*)It is the average transmission rate of the user data in bps:

$$v_{ef}\,[\text{bps}] = \frac{\text{number of information bits}}{\text{observation time}, T}$$

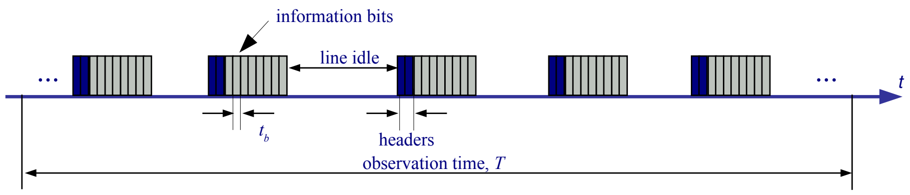


Figure 12: Definition of Throughput: information bits and observation time.

- **Efficiency** or channel utilization

$$E[\%] = \frac{v_{ef}}{v_t} \times 100$$

4.3.10 Efficiency in terms of time and bits

$$E = \frac{v_{ef}}{v_t} = \frac{\#\text{data bits}/T}{1/t_b} = \begin{cases} \dfrac{\#\text{data bits} \times t_b}{T} = \dfrac{\text{time Tx data}}{T} \\ \dfrac{\#\text{data bits}}{T/t_b} = \dfrac{\#\text{data bits}}{\#\text{bits at line bitrate}} \end{cases}$$

v_{ef}:	throughput
T:	Observation time
t_b:	bit Tx time
$v_t = \frac{1}{t_b}$:	line bitrate

4.3.11 Stop & Wait efficiency without Tx errors

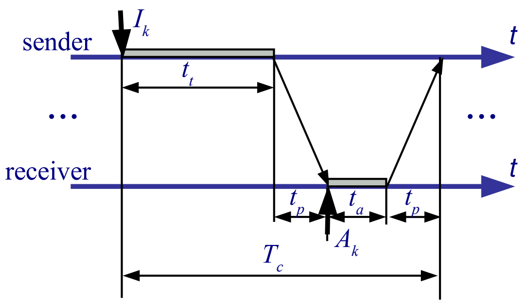


Figure 13: Time diagram of stop-and-wait without errors.

$$E_{protocol} = \frac{t_t}{RTT} = \frac{t_t}{t_t + t_a + 2\,t_p} \approx \frac{t_t}{t_t + 2\,t_p} = \frac{1}{1 + 2\,a},$$

where $a = t_p/t_t$

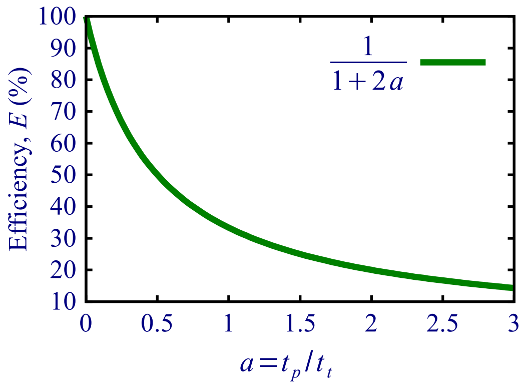


Figure 14: Graphic showing as stop-and-wait efficiency decreases as $1/(1 + 2\,a)$.

4.3.12 Continuous Tx Protocols

- Without errors: E = 100%

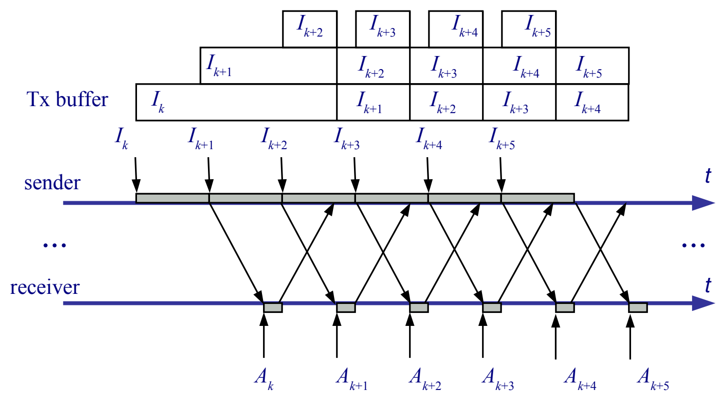


Figure 15: Time diagram of a continuous transmission protocol. PDUs are transmitted "back-to-back" (with no delay).

- In case of **errors**
  - Go Back N
  - Selective ReTx

4.3.13 Go Back N

- **Cumulative acks**: A_k confirm I_i, $i \le k$
- If error or out of order PDU: **Do not send acks**, discards all PDU until the expected PDU arrives. The receiver does not store out of order PDUs
- Upon **RTO**: go back and starts Tx from that PDU

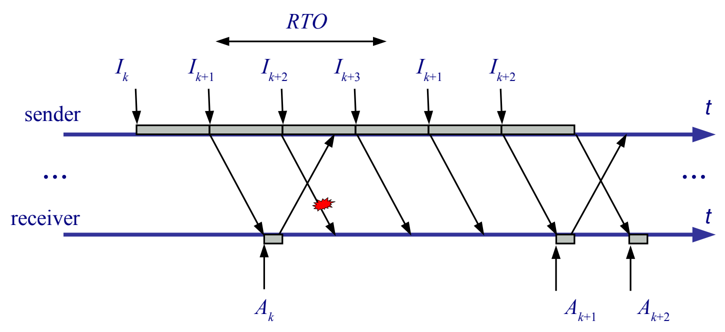


Figure 16: Time diagram of a Go-Back-N protocol with errors. When PDUs arrive out of order the received stops sending acks until the expected PDU is retransmitted.

4.3.14 Selective ReTx

- Same as Go Back N, but:
  - The sender only ReTx a PDU when a **RTO** occurs
  - The **receiver** stores out of order PDUs, and ack all stored PDUs when missing PDUs arrive

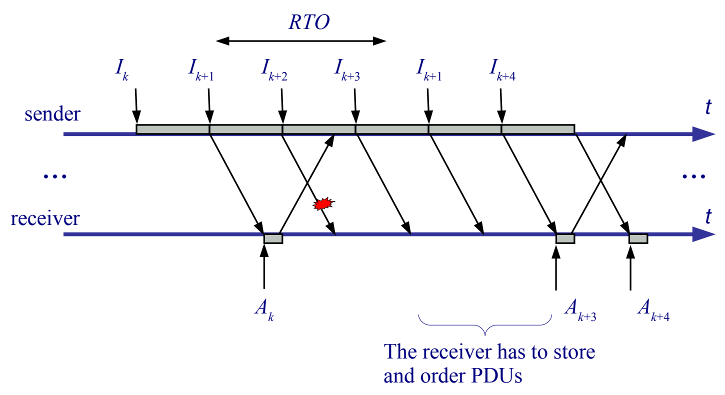


Figure 17: Time diagram of a ReTx protocol with errors. When PDUs arrive out of order the received stops sending acks until the expected PDU is retransmitted. Se sender does not ReTx PDUs for wich the RTO does not expires. The receiver stores out of order PDUs and sends accumulated acks.

4.3.15 Flow Control and Window Protocols

- **Flow control**: adapt Tx to Rx rate
- **Stop & Wait**: automatic Flow control
- **Continuous Tx** protocols: Use a **Tx window**
- **Tx window** maximum number of **non-ack PDUs that can be Tx**. If the Tx window is exhausted, the sender stales
- **Stop & Wait** is a window protocol with **Tx window = 1** PDU
- Tx window allows dimension the Tx and Rx buffers

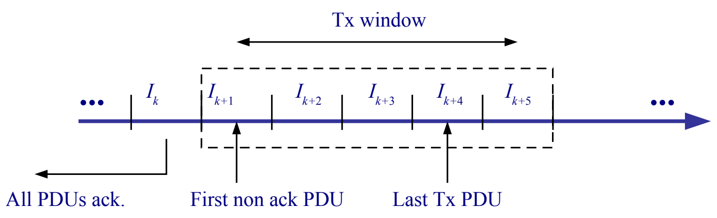


Figure 18: Window mechanism. When the number of unack PDUs reaches the Tx window, the sender stales.

4.3.16 Optimal Tx window

Optimal window: Minimum window that allows the maximum throughput

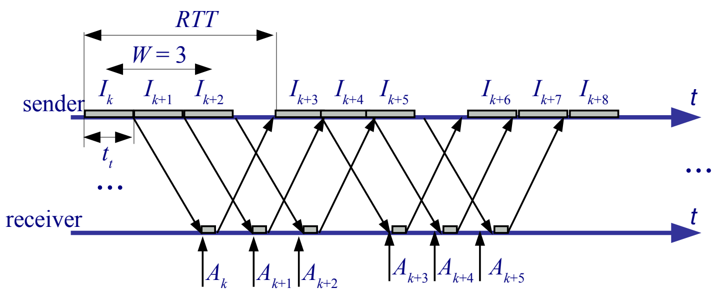


Figure 19: Time diagram of a suboptimal window. The sender stales waiting for acks.

W_{opt} is referred to as the **bandwidth delay product**:

$$W_{opt}[\text{PDU}] = \left\lceil \frac{\text{RTT}}{t_t} \right\rceil = \left\lceil v_{ef}^{max}[\text{PDU/s}] \times \text{RTT[s]} \right\rceil$$

In **bytes**:

$$W_{opt}[\text{bytes}] \approx v_{ef}^{max}[\text{bytes/s}] \times \text{RTT[s]} = \frac{v_{ef}^{max}[\text{bps}]}{8\ [\text{bits/byte}]} \times \text{RTT[s]}$$

Example:
for $v_{ef} = 4$ Mbps and RTT $= 200$ ms we need

$$W_{opt} = v_{ef} \times \text{RTT} = \frac{4 \times 10^6 \text{ bps}}{8\ [\text{bits/byte}]} \times 200 \times 10^{-3} \text{ s} = 100 \text{ kbyte}$$

4.4 TCP Protocol RFC793

4.4.1 TCP Service

- **Service**:
  - Reliable service (ARQ):
    - Connection oriented
    - Error recovery
    - Flow control: Adapt throughput to receiver
    - Congestion control: Adapt throughput to network
- **Usage**
  - Applications requiring reliability: Web, ftp, ssh, telnet, mail, ...

4.4.2 TCP Basis

- Segments of optimal size: Maximum Segment Size (**MSS**)
  - MSS adjusted using **MTU path discovery**
- **ARQ** window protocol, with **variable window**
- **TCP sender** immediately sends the segments allowed by the window
- Upon segment arrival **TCP receiver** immediately sends an **ack**
- acks are **cumulative** (ack all previous segments)
- In case of losses the TCP receiver send **duplicated acks**: with the same ack number (otherwise would ack the lost segment)

4.4.3 TCP window

- **wnd = min(awnd, cwnd)**
  - **awnd**, advertised window: used for **flow control** send by **TCP receiver** (TCP header). Set to the **free Rx buffer space** of the **TCP receiver** (see the figure).
  - **cwnd**, congestion window: used for **congestion control** computed by **TCP sender** (SS/CA algorithms)
- NOTE **wnd** and **awnd** are also known as **Tx** and **Rx window**, respectively

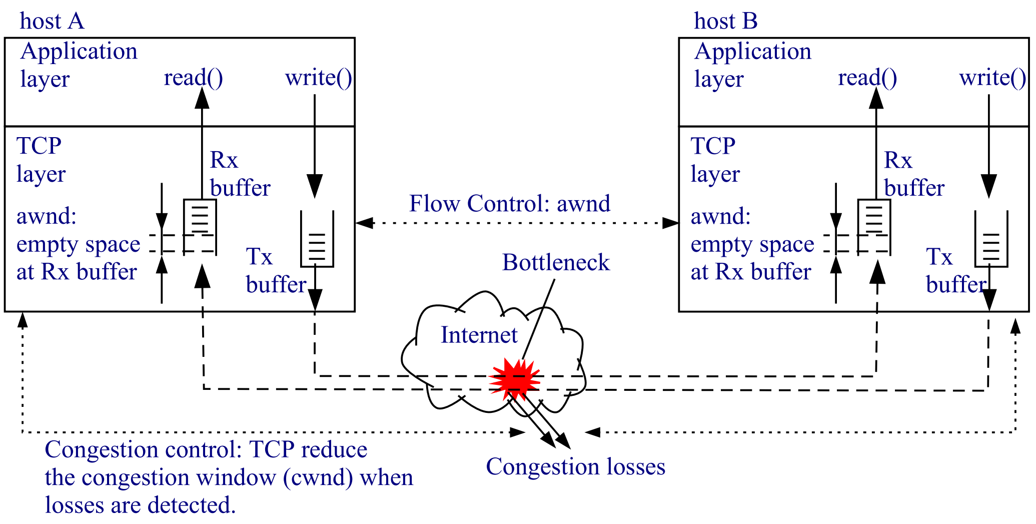


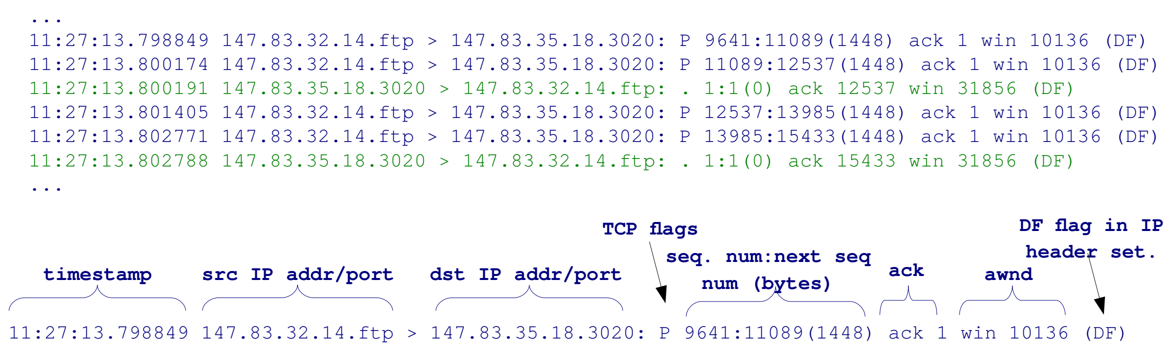
Figure 20: TCP window behavior. Upon sending an ack the awnd is set to the free Rx buffer size (flow control). When losses occur TCP reduces the cwnd (congestion control).

Acronyms:

SS:	Slow Start
CA:	Congestion Avoidance

4.4.4 Delayed acks RFC1122

- TCP connections can be classified as:
  - **Bulk**: (e.g. web, ftp) There are always bytes to send. TCP send MSS bytes in each segment.
  - **Interactive**: (eg. telnet, ssh) The user interacts with the remote host.
- In bulk connections **sending an ack every data segment** can unnecessarily send too many small segments. Solution:
- **Delayed ack**. It is used to reduce the amount of acks. Consists of sending **1 ack each 2 MSS segments**, or 200 ms. Acks are always sent in case of receiving out of order segments.
- **tcpdump example** (delayed acks, capture at the receiver side):



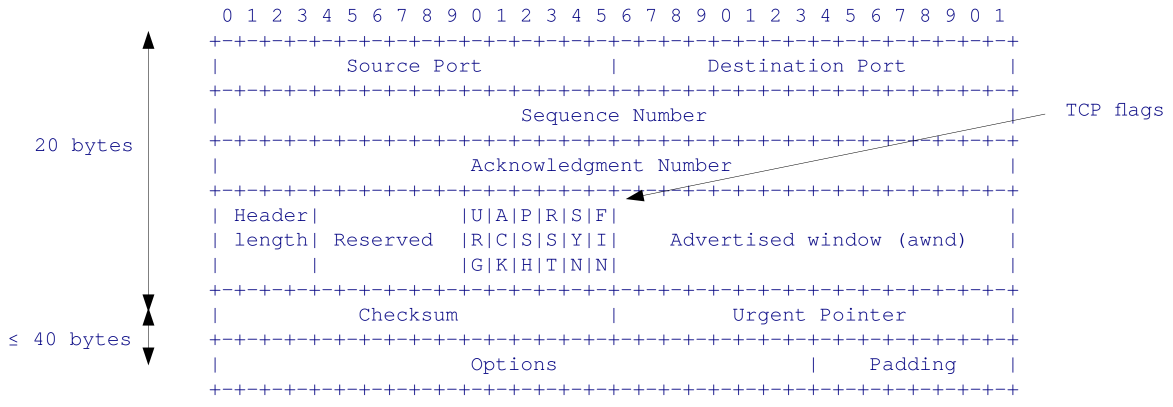
```
...
11:27:13.798849 147.83.32.14.ftp > 147.83.35.18.3020: P 9641:11089(1448) ack 1 win 10136 (DF)
11:27:13.800174 147.83.32.14.ftp > 147.83.35.18.3020: P 11089:12537(1448) ack 1 win 10136 (DF)
11:27:13.800191 147.83.35.18.3020 > 147.83.32.14.ftp: . 1:1(0) ack 12537 win 31856 (DF)
11:27:13.801405 147.83.32.14.ftp > 147.83.35.18.3020: P 12537:13985(1448) ack 1 win 10136 (DF)
11:27:13.802771 147.83.32.14.ftp > 147.83.35.18.3020: P 13985:15433(1448) ack 1 win 10136 (DF)
11:27:13.802788 147.83.35.18.3020 > 147.83.32.14.ftp: . 1:1(0) ack 15433 win 31856 (DF)
...
```

TCP flags
DF flag in IP header set.
seq. num:next seq num (bytes)
ack
awnd
timestamp
src IP addr/port
dst IP addr/port
11:27:13.798849 147.83.32.14.ftp > 147.83.35.18.3020: P 9641:11089(1448) ack 1 win 10136 (DF)

Figure 21: tcpdump capture showing delayed acks.

4.4.5 TCP header

- Fixed **20** bytes + **options** 15x4 = **60** bytes max
- Like UDP, the checksum is computed using header + **pseudo-header** + payload



```
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|          Source Port          |       Destination Port        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                        Sequence Number                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Acknowledgment Number                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Header|           |U|A|P|R|S|F|                               |
| length| Reserved  |R|C|S|S|Y|I|    Advertised window (awnd)   |
|       |           |G|K|H|T|N|N|                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           Checksum            |         Urgent Pointer        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Options                    |    Padding    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 22: TCP header.

4.4.6 TCP Flags

- **URG** (Urgent): Urgent Pointer points to the first urgent byte. Example: ˆC in a telnet session
- **ACK**: Always set except for the first segment
- **PSH** (Push): "push" all data to the receiving buffer
- **RST** (Reset): Abort the connection
- **SYN**: Used in the connection setup (**three-way-handshaking**)
- **FIN**: Used in the connection termination

4.4.7 Connection Setup and Termination

- The **client** always send the 1st segment
- **Three-way handshaking** segments have **payload = 0**
- **SYN** and **FIN** segments consume **1** sequence number
- **Initial** sequence number is **random** in each side

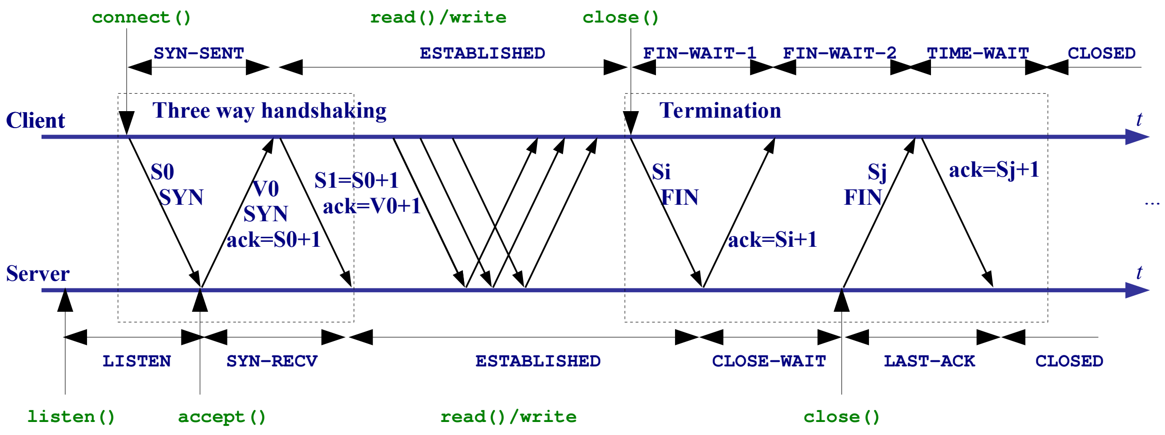


Figure 23: Time diagram with the connection establishment and termination of a TCP connection.

4.4.8 TCP Options

- **Maximum Segment Size** (MSS): Used in the **TWH**. **MSS=MTU-40** (IPv4+TCP headers without options=40 bytes)
- **Window Scale factor**: Used in the **TWH**. **awnd** is multiplied by $2^{\text{WindowScale}}$ (**WindowScale** = number of bits to left-shift awnd). Allows using awnd larger than $2^{16} = 65536$ bytes
- **Timestamp**: Used to compute the Round Trip Time (**RTT**). **10 bytes** option = TCP sender clock & echo of the timestamp of the segment being ack
- **SACK** (Selective ack): In case of errors, ack blocks of consecutive correctly received segments for Selective ReTx

Acronyms:

TWH:	Three-way handshaking

4.4.9 TCP Sequence Numbers

- **Sequence number** points the first payload byte
- **SYN** and **FIN** consume **1** sequence number
- **Ack number** points the next missing byte (all previous bytes are acknowledged)

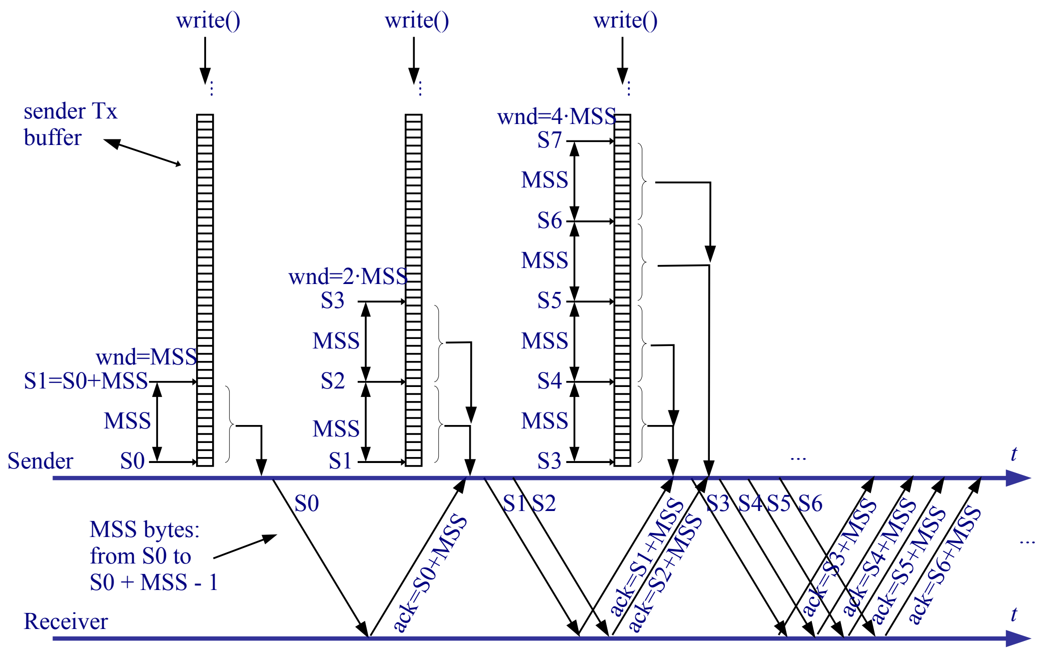


Figure 24: Time diagram showing the Tx of sequence numbers. ack sends a sequence number equal to the next expected byte.

Practical example

Capture a TCP connection with wireshark and observe the connection setup, options, termination and sequence numbers (bash)

```
1. wireshark
```

Minimal TCP server (ruby)

```
#!/usr/bin/ruby -w

require 'socket'

server = TCPServer.new 2000 # bind to port 2000
client = server.accept # Wait for a client to connect
while line = client.gets # read message
  puts line
end
client.close # close socket
```

Minimal TCP client (ruby)

```
#!/usr/bin/ruby -w

require 'socket'

server = TCPSocket.new('192.168.11.36', 2000)
for i in 1..10 do
  puts "sending " + i.to_s
  server.puts "Hello " + i.to_s # send
end
server.close # close socket
```

4.4.10 TCP Congestion Control RFC2581

- **wnd = min(awnd, cwnd)**
  - **awnd**, advertised window: used for **flow control**
  - **cwnd**, congestion window: used for **congestion control**
- TCP interprets losses as congestion
- Basic **Congestion Control Algorithm**:
  - **Slow Start / Congestion Avoidance** (SS/CA)

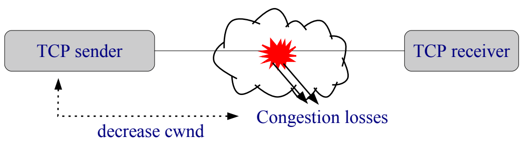


Figure 25: Congestion window cwnd is governed by SS/CA algorithms.

4.4.11 Slow Start / Congestion Avoidance (SS/CA) RFC2581

- **ssthresh**: Threshold between SS and CA

Slow Start / Congestion Avoidance (SS/CA) (c)

```
Initialization:
 cwnd = MSS ; /* NOTE: RFC2581 allows an initial window of 2 segments */
 ssthresh = infinity ;
/* Each time an ack confirming new data is received: */
 if(cwnd < ssthresh) { /* Slow Start */
   cwnd += MSS ;        /* add 1 segment */
 } else {               /* Congestion Avoidance */
   cwnd += MSS * MSS / cwnd ; /* add 1/cwnd segments */
  }
/* When RTO expires: */
 Retransmit first unack segment ;
 ssthresh = max(min(awnd, cwnd)/2, 2*MSS) ;
 cwnd = MSS ;
```

Figure 26: Pseudo-code of SS/CA algorithms.

This congestion control algorithm is referred to as **additive increase multiplicative decrease, AIMD**

Acronyms:

ssthresh:	Slow Start threshold
MSS:	Maximum Segment Size
cwnd:	Congestion Window
awnd:	Advertised Window
RTO:	Retransmission Timeout

- **SS** cwnd is rapidly increased to the "operational point"
- **CA** cwnd is slowly increased looking for more "bandwidth"

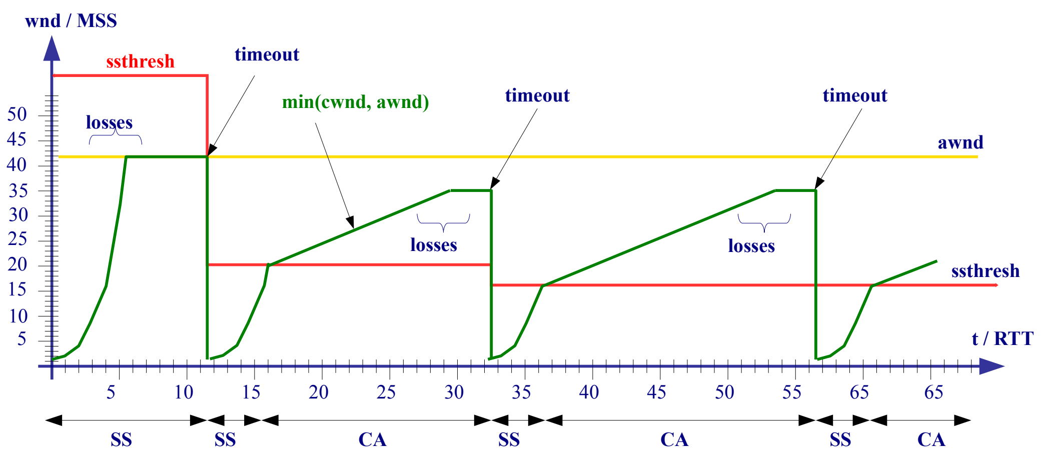


Figure 27: Typical saw shape evolution of cwnd when losses occur. During SS cwnd grows exponentially, during CA cwnd grows linearly.

4.4.12 Relation between throughput, mean W and mean RTT

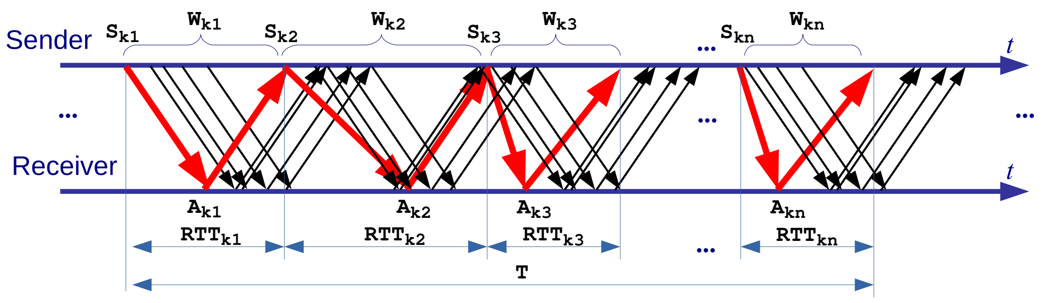


Figure 28: Time diagram showing the TCP window sent in consecutive RTTs.

Define:

- S_{k1} a segment and A_{k1} its ack received after RTT_{k1}
- During RTT_{k1} the sender is allowed to send W_{k1} bytes
- Upon receiving A_{k1} new data is ack, and the sender sends S_{k2}, etc.

Let $B(T)$ the number of bytes Tx during a time T. We have:

$$\boxed{v_{ef} = \lim_{T\to\infty} \frac{B(T)}{T} = \lim_{n\to\infty} \frac{\sum_{i=1}^{n} W_{ki}}{\sum_{i=1}^{n} RTT_{ki}} = \lim_{n\to\infty} \frac{\frac{\sum_{i=1}^{n} W_{ki}}{n}}{\frac{\sum_{i=1}^{n} RTT_{ki}}{n}} = \frac{W}{RTT}}$$

where W is the average window and RTT the average RTT. Beware of the **units**: if W is in bytes and RTT in seconds:

$$\boxed{v_{ef}[\text{bps}] = \frac{W[\text{bytes}]}{RTT[\text{s}]} \times \frac{8\ [\text{bits}]}{1[\text{byte}]}}$$

4.4.13 Evaluation Example Without Losses

Assume:

- propagation delays=0
- C1 and C2 **send** to S, $awnd = 64$ kB
- **Queue size of the router** $Q > 128$ kB

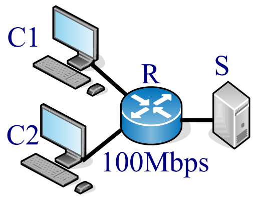


Figure 29: Two PCs and a server are connected to a router, all links are 100 Mpbs.

Compute the throughput and RTT

- The **bottleneck** is the link R-S
- For each connection $v_{ef} = 100/2 = 50$ Mbps
- In the **queue of the router** there will be 128 kB approx. (the 2 TCP windows)
- The **RTT** is the time in the queue of the router:

$$RTT = \frac{128 \text{ kB}}{100 \text{ Mbps}} = 10.24 \text{ ms}$$

- Check that

$$v_{ef} = \frac{W}{RTT} = \frac{64 \text{ kB}}{10.24 \text{ ms}} = 50 \text{ Mbps}$$

4.4.14 Evaluation Example With Losses

Assume:

- propagation delays=0
- C1 and C2 **send** to S, $awnd = 64$ kB
- **Queue size the router** $Q = 100$ kB

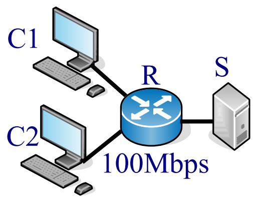


Figure 30: Two PCs and a server are connected to a router, all links are 100 Mpbs.

Compute the throughput and RTT

- **Losses** occur when both TCP windows ($W_1 + W_2$) add to 100 kB
- **Approximated** evolution of the **router queue**

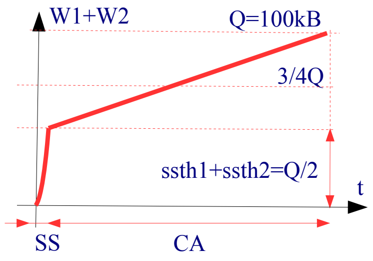


Figure 31: The buffer queue approximatelly follows the cwnd evolution of both PCs.

- The queue of the router will never be empty $\Rightarrow v_{ef} = 100/2 = 50$ Mbps
- Average **queue size**:

$$\bar{Q} = (Q/2 + Q)/2 = 3/4\,Q = 75 \text{ kB}$$

- Average **RTT**:

$$RTT = 75 \text{ kB}/100 \text{ Mbps} = 6 \text{ ms}$$

- Average **window** of each connection

$$\bar{W_1} = \bar{W_2} = 75 \text{ kB}/2 = 37.5 \text{ kB}$$

- Check that

$$v_{ef} = \frac{W}{RTT} = \frac{37.5 \text{ kB}}{6 \text{ ms}} = 50 \text{ Mbps}$$

4.4.15 Retransmission time-out (RTO) RFC2988

- Every TCP socket has **1 RTO timer**
- Activation:
  - Active whenever there are pending acks
  - Continuously decreased, **ReTx** occurs when RTO reaches zero
- Each time an ack **confirming new data** arrives:
  - RTO is computed
  - RTO is restarted if pending acks
- Computation:
  - TCP sender measures RTT mean (**srtt**) and variance (**rttvar**)
  - **RTO = srtt + 4 * rttvar**
  - RTO is **duplicated at each ReTx**
- **RTT** measurements:
  - Estimated by the OS using "slow-timer tics" (coarse)
  - Using the TCP **timestamp** option

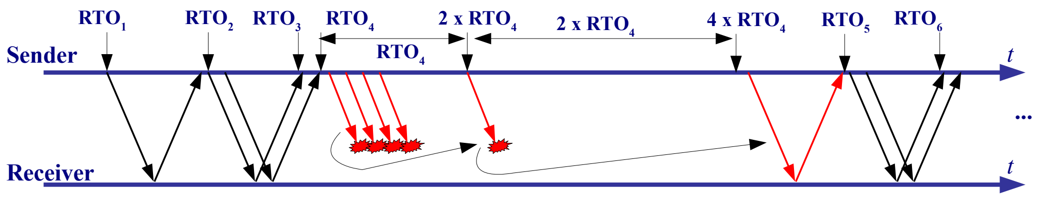


Figure 32: Time diagram of sucessive ReTx of the same segment. At each ReTx the RTO is duplicated.

4.4.16 TCP State diagram

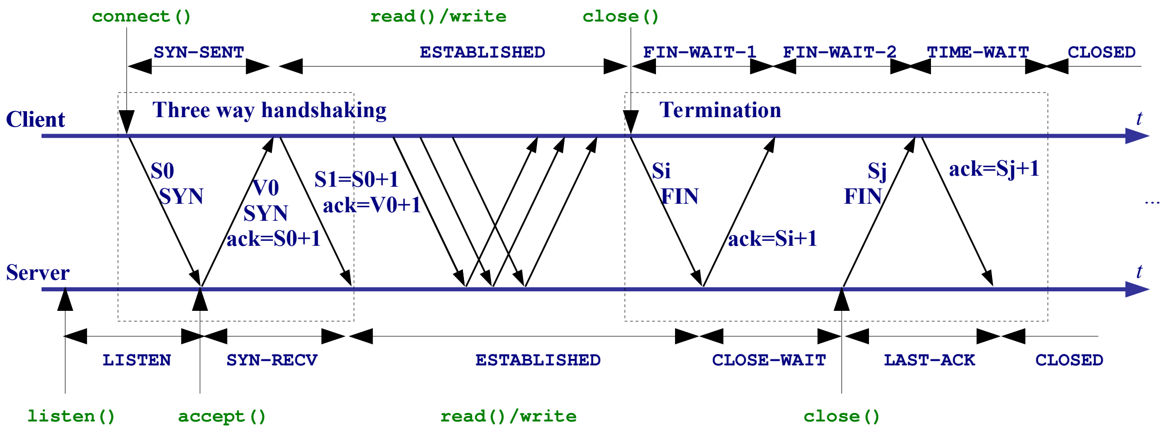


Figure 33: Time diagram with the connection establishment and termination of a TCP connection.

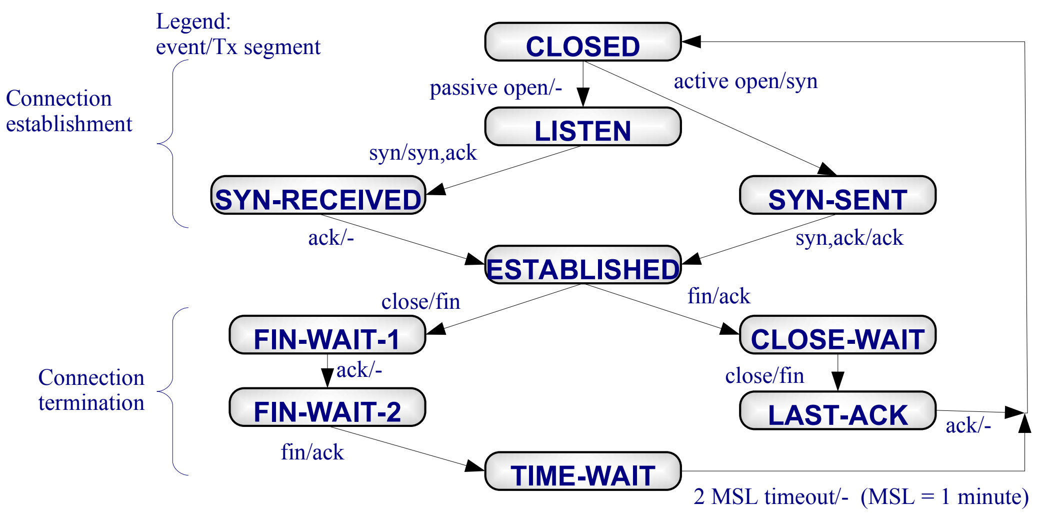


Figure 34: Diagram of TCP states followed during connection establishment and termination.

Acronyms:
MSL: Maximum Segment Live time

4.5 List of Acronyms

ack	Acknowledgment	ReTx	Re-transmission
API	Application Programming Interface	RFC	Request For Comments
ARQ	Automatic Repeat reQuest	Rx	Reception
A_k	Ack PDU confirming I_k	SS	Slow Start
I_k	Information PDU number k	TCP	Transmission Control Protocol
v_{ef}	Throughput	TWH	Three-way handshaking
CA	Congestion Avoidance	Tx	Transmission
MSL	Maximum Segment Live time	UDP	User Datagram Protocol
MSS	Maximum Segment Size	wnd	TCP Window
PDU	Protocol Data Unit	awnd	Advertised Window
RTO	Retransmission Timeout	cwnd	Congestion Window
RTT	Round Trip Time	ssthresh	Slow Start threshold