Computer Nerworks. Unit 3: TCP

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

Llorenç Cerdà-Alabern

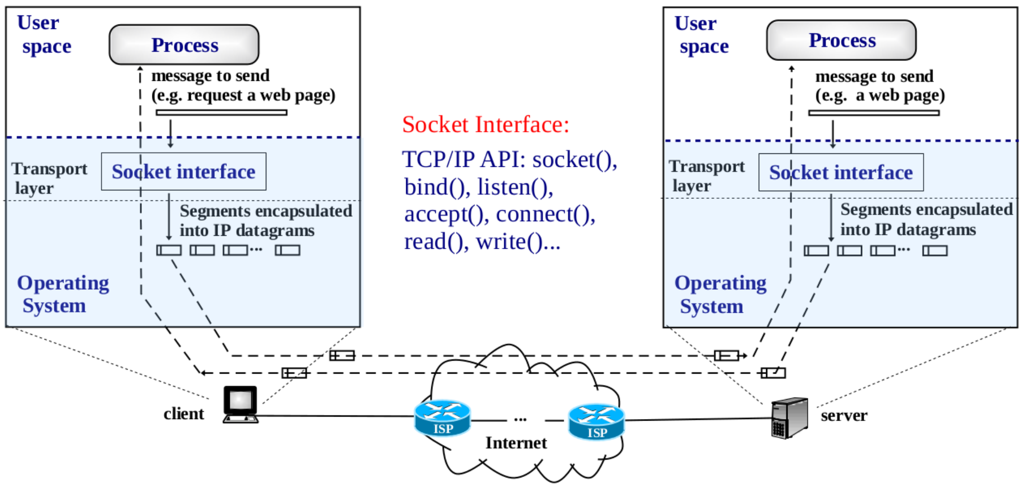
April 11, 2019

Contents

3 Unit 3: TCP

3.1 Transport layer: UDP/TCP

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



3.2 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

3.2.1 UDP Header RFC768

- Fixed size of **8 bytes**
- **checksum**: computed using header, pseudo-header, payload
- Drawback: **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            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

UDP datagram header

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

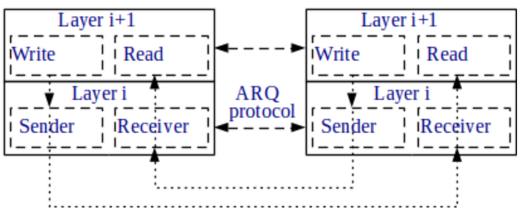
UDP pseudo-header

3.3 Automatic Repeat reQuest (ARQ) RFC3366

3.3.1 What is ARQ?

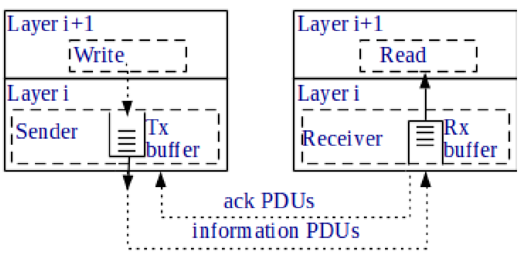
Communication channel between endpoints designed for **reliability** and **efficiency**. Typically involves:

- **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



3.3.2 ARQ Ingredients

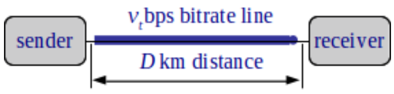
- **Connection oriented**
- Tx/Rx (Transmission/Reception) **buffers**
- Acknowledgments (**ack**)
- Acks can be piggybacked
- Retransmission Timeout, **RTO**
- **Sequence Numbers**



ARQ Protocol Implementation (one way)

3.3.3 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 $t_t = L_I/vt$, $t_a = L_A/vt$

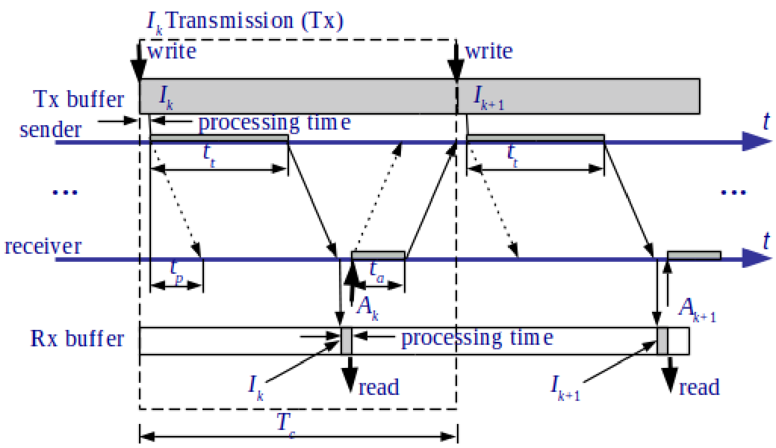


3.3.4 Basic ARQ Protocols

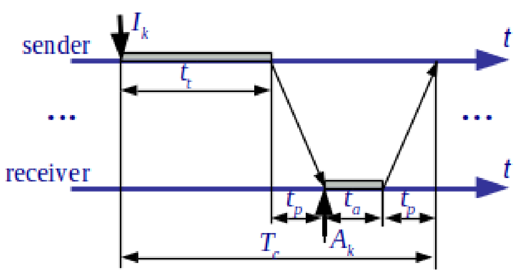
- Stop & Wait
- Go Back N
- Selective Retransmission

3.3.5 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

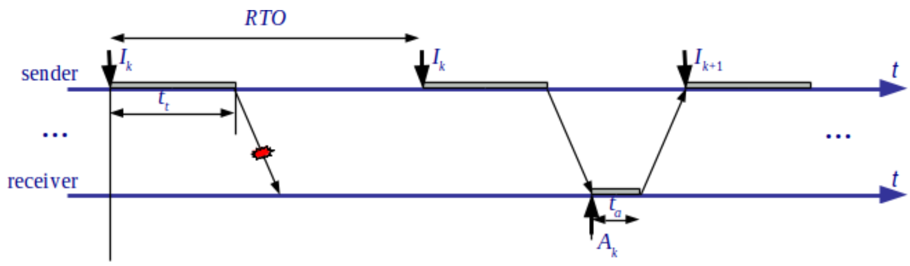


3.3.6 Stop & Wait simplified diagram



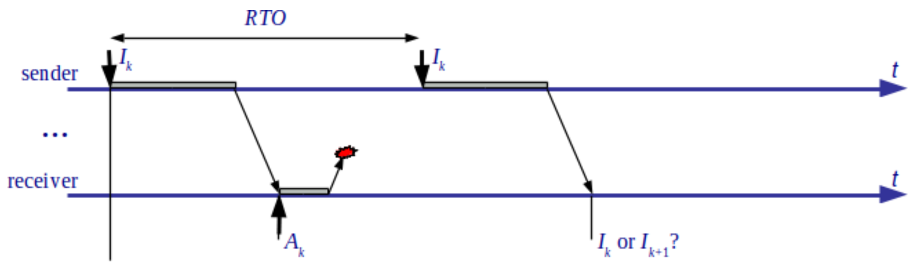
3.3.7 Stop & Wait Retransmission

- 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

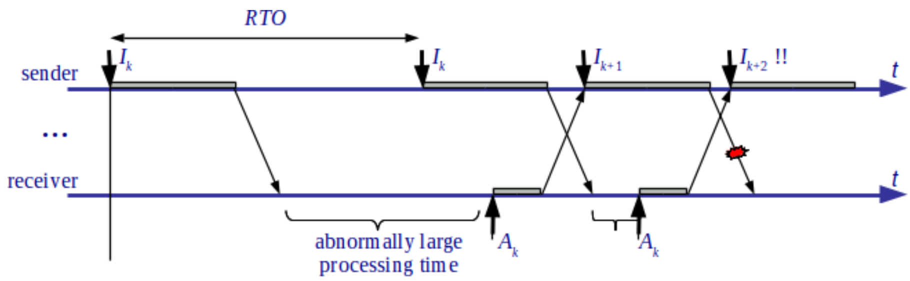


Tx: Transmission
Rx: Reception
ReTx: Re-transmission

3.3.8 Why sequence numbers are needed?



Need to number information PDUs



Need to number ack PDUs

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

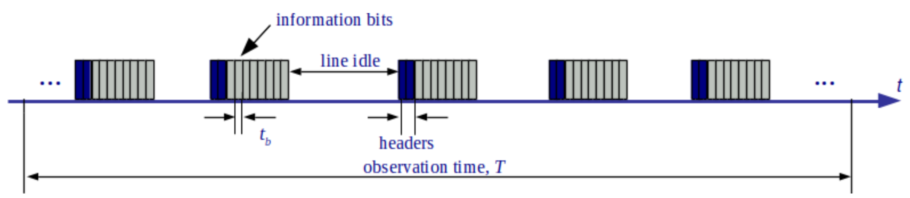
3.3.9 Evaluation

- Given a line with bitrate v_t [bps]:
- **Throughput** (*velocidad efectiva*)

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

- **Efficiency** or channel utilization

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



Practical example: throughput with speedtest

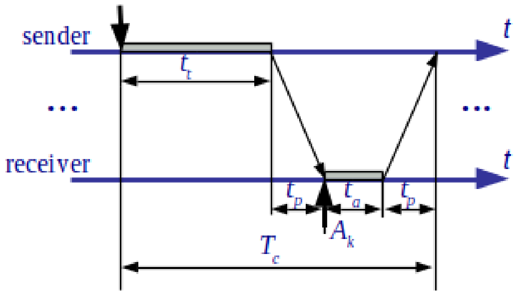
```
tcpdump -ni wlan0 tcp
```

3.3.10 Efficiency in terms of time and bits

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

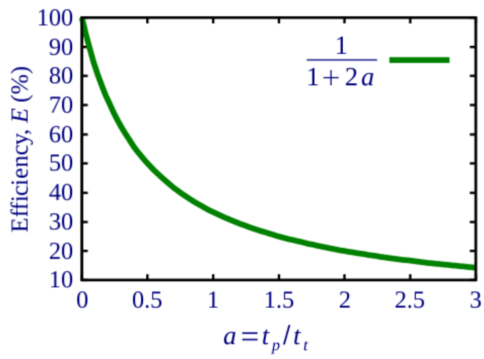
v_{ef} : throughput
T: Observation time
t_b: bit Tx time
$v_t = \frac{1}{t_b}$: line bitrate

3.3.11 Stop & Wait efficiency without Tx errors



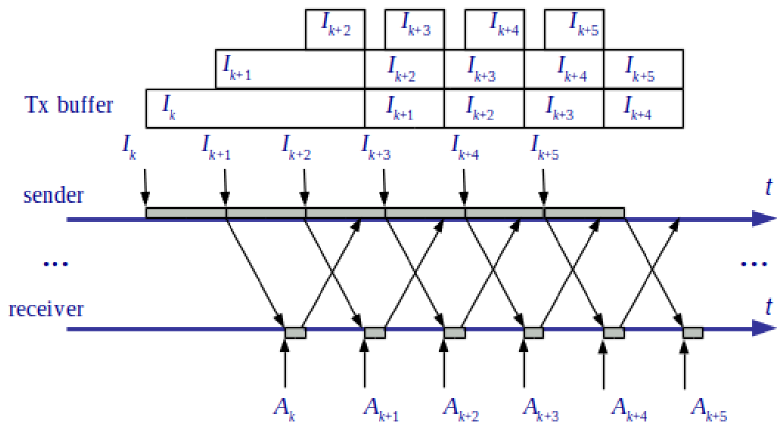
$$E_{protocol} = \frac{t_t}{T_C} = \frac{t_t}{t_t + t_a + 2\,t_p} \approx \frac{t_t}{t_t + 2\,t_p} = \frac{1}{1 + 2\,a},$$

where $a = t_p/t_t$



3.3.12 Continuous Tx Protocols

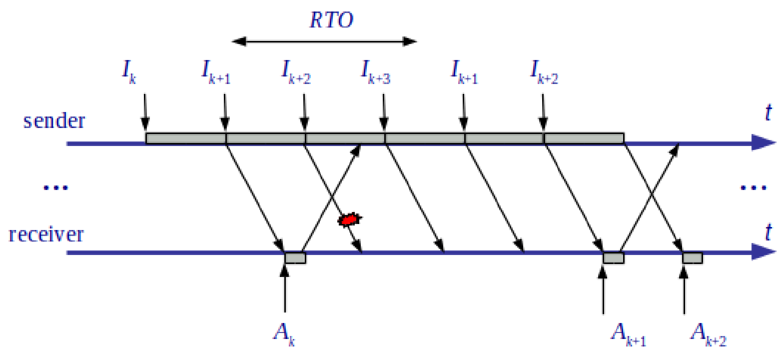
- Without errors: E = 100%



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

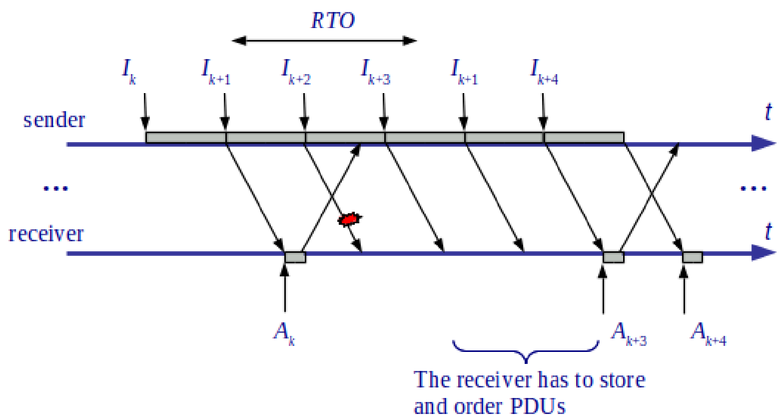
3.3.13 Go Back N

- **Cumulative acks**: A_k confirm I_i, $i \leq k$
- 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



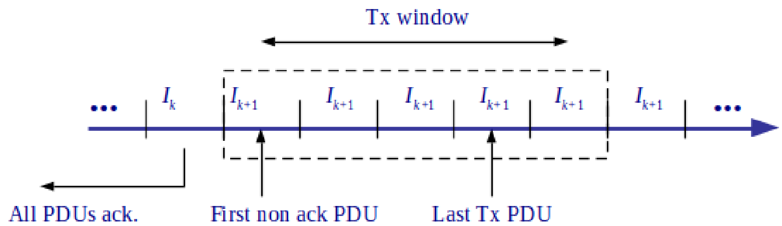
3.3.14 Selective ReTx

- 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



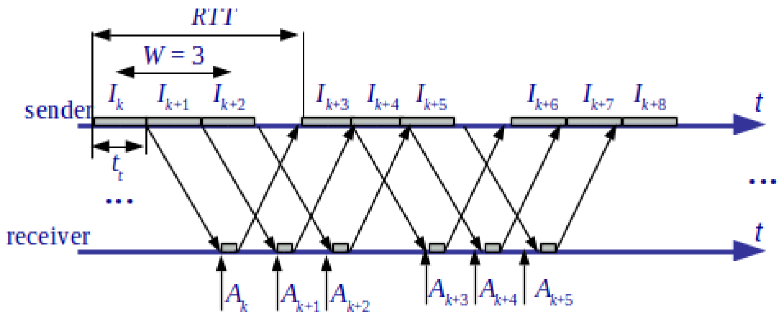
3.3.15 Flow Control and Window Protocols

- **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



3.3.16 Optimal Tx window

Optimal window: Minimum window that allows the maximum throughput



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

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

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

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

3.4 TCP Protocol RFC793

3.4.1 TCP Service

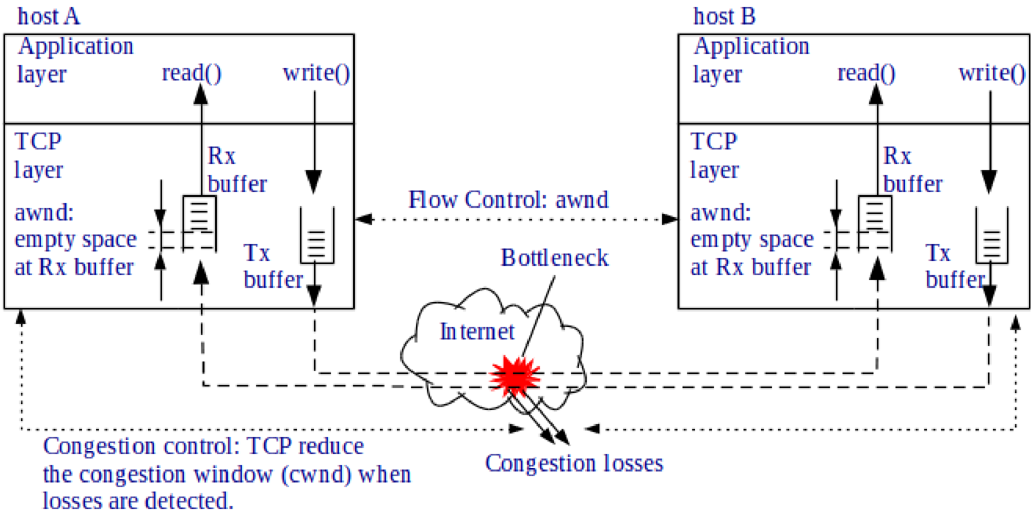
- **Service**:
  - Reliable service (ARQ):
    * Connection oriented
    * Error recovery
    * Congestion control: Adapt throughput to network
    * Flow control: Adapt throughput to receiver
- **Usage**
  - Applications requiring reliability: Web, ftp, ssh, telnet, mail, ...

3.4.2 TCP Basis

- Segments of optimal size: Maximum Segment Size (**MSS**)
  - MSS adjusted using **MTU path discovery**
- **ARQ** window protocol, with **variable window**
- Upon segment arrival TCP immediately sends an **ack**

3.4.3 TCP window

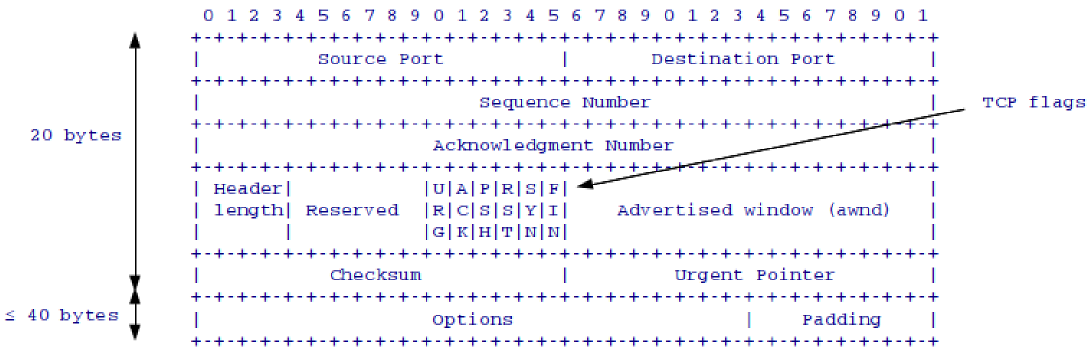
- **wnd = min(awnd, cwnd)**
  - **awnd**, advertised window: used for **flow control** send by **TCP receiver** (TCP header). Set to the **free Rx buffer space** of the **TCP receiver** (see the figure).
  - **cwnd**, congestion window: used for **congestion control** computed by **TCP sender** (SS/CA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



SS: Slow Start
CA: Congestion Avoidance

3.4.4 TCP header

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



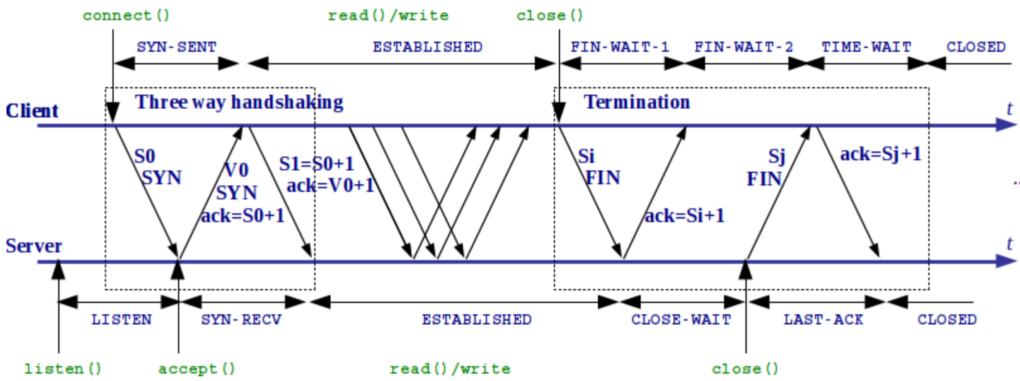
TCP segment header

3.4.5 TCP Flags

- **URG** (Urgent): Urgent Pointer points to the first urgent byte. Example: ^C 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

3.4.6 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**



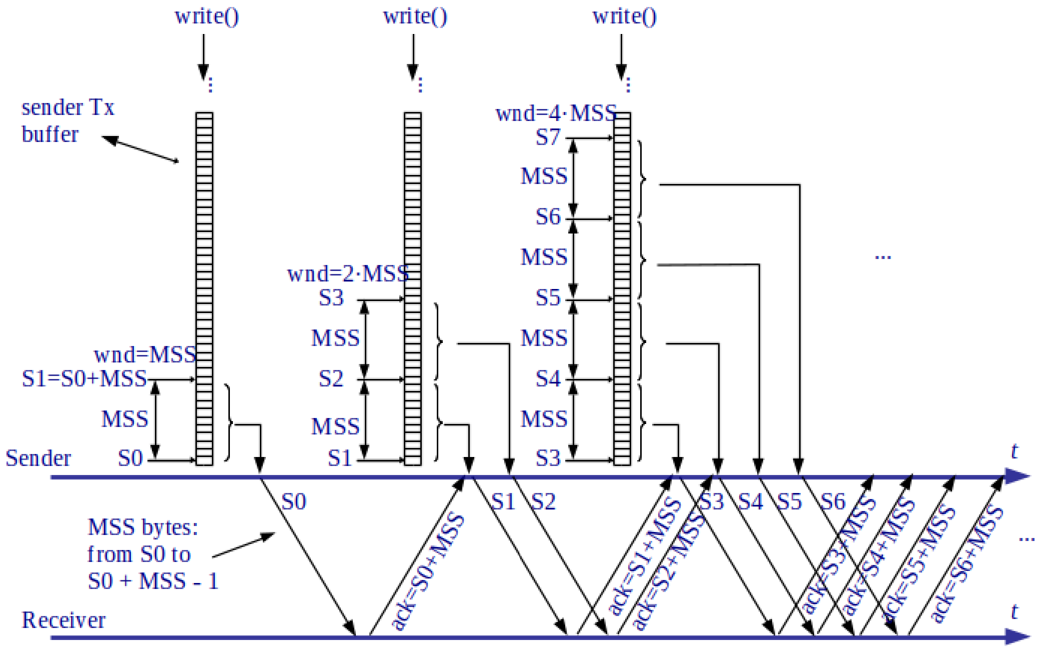
3.4.7 TCP Options

- **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

TWH: Three-way handshaking

3.4.8 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)



Practical example

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

```
1. change the loopback MTU:
   sudo ifconfig lo mtu 1500
2. wireshark
```

Minimal TCP server (perl)

```perl
#!/usr/bin/perl -w
use IO::Socket::INET; use Term::ANSIColor;

print "Sart TCP server.\n" ;
my $s_sock = IO::Socket::INET->new(
     LocalPort => 5000,
     Proto     => 'tcp',
     Listen    => 5
) or die "Could not create socket!\n";

while(1) {
  my $c_sock = $s_sock->accept() ;
  printf colored("Accepted: ", 'green')."%s, %s\n",
    $c_sock->peerhost(), $c_sock->peerport() ;
  while(<$c_sock>) {
    print "Received from Client: $_";
  }
  print colored("Connection closed", 'red')."\n" ;
}
```

Minimal TCP client (perl)

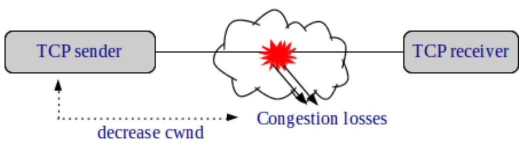
```perl
#!/usr/bin/perl -w
use IO::Socket::INET;

print "Sart TCP client.\n" ;
my $socket = IO::Socket::INET->new(
     PeerHost => '127.0.0.1',
     PeerPort  => 5000,
     Proto     => 'tcp'
) or die "Could not create socket: $!\n";

print "TCP Connected.\n" ;
while (<>) {
  print "sending $_" ;
  $socket->send($_);
}
```

3.4.9 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)



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

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

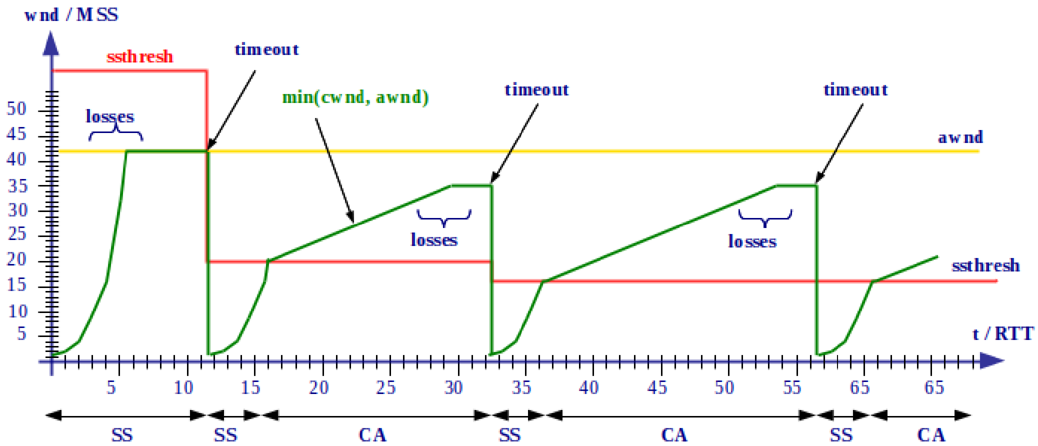
Slow Start / Congestion Avoidance (SS/CA) (c)

```
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 ;
```

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

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

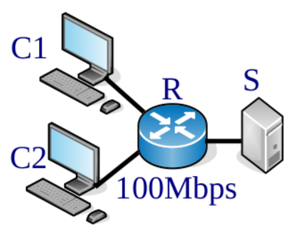
- **SS** cwnd is rapidly increased to the "operational point"
- **CA** cwnd is slowly increased looking for more "bandwidth"



3.4.11 Evaluation Example Without Losses

Assume:

- propagation delays=0
- C1 and C2 **send** to S, $awnd = 64$ kB



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:

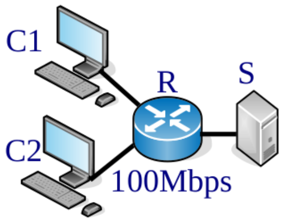
$$RTT = \frac{128 \text{ kB}}{100 \text{ Mbps}} = 10.24 \text{ ms}$$

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

3.4.12 Evaluation Example With Losses

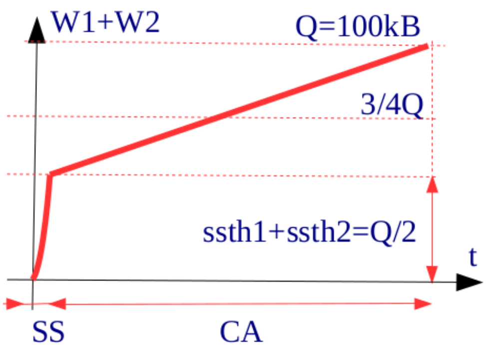
Assume:

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



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**



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

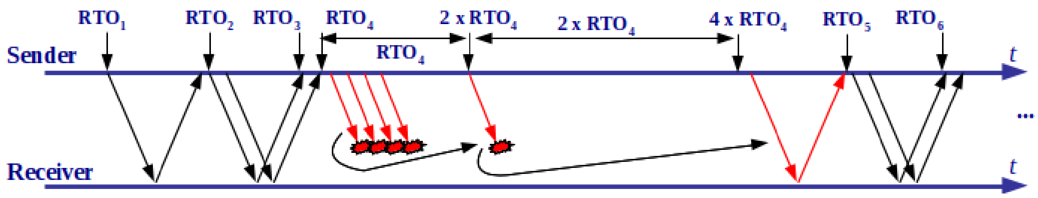
3.4.13 Retransmission time-out (RTO) RFC2988

- Activation:
  - Active whenever there are pending acks
  - Continuously decreased, **ReTx** occurs when RTO reaches zero
- Each time an ack **confirming new data** arrives:

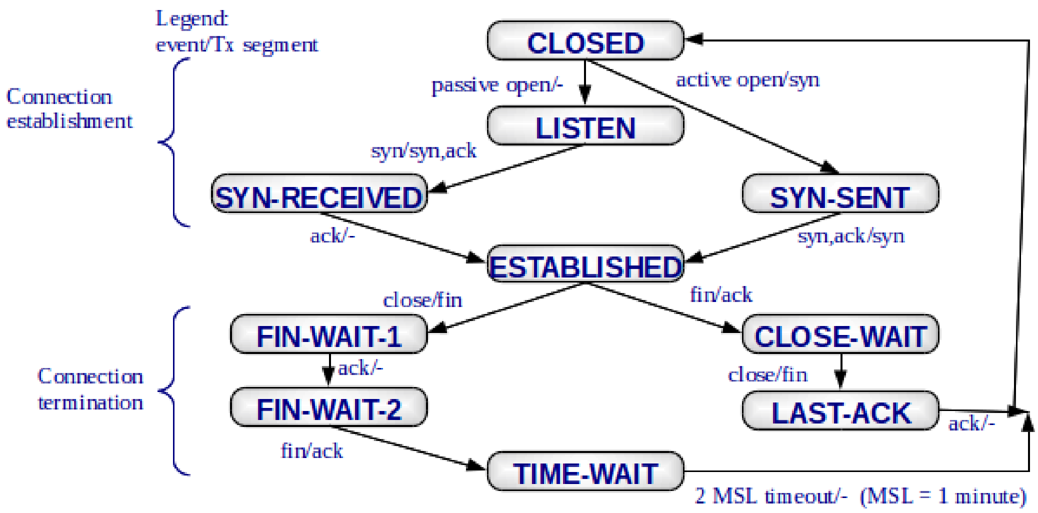
- RTO is computed
- RTO is restarted if pending acks

- Computation:
  - TCP sender measures RTT mean (**srtt**) and variance (**rttvar**)
  - **RTO = srtt + 4 * rttvar**
  - RTO is duplicated each retransmitted segment

- **RTT** measurements:
  - Using "slow-timer tics" (coarse)
  - Using the TCP **timestamp** option



3.4.14 TCP State diagram



Practical example

capture a TCP connection with tcpdump and observe the connection states (bash)

```
wireshark
netstat -nat
```