Computer Networks - *Xarxes de Computadors*

Outline

- Course Syllabus
- Unit 1: Introduction
- Unit 2. IP Networks
- Unit 3. LANs
- **Unit 4. TCP**
- Unit 5. Network applications

Based on: https://studies.ac.upc.edu/FIB/grau/XC/#slides

Unit 4. TCP

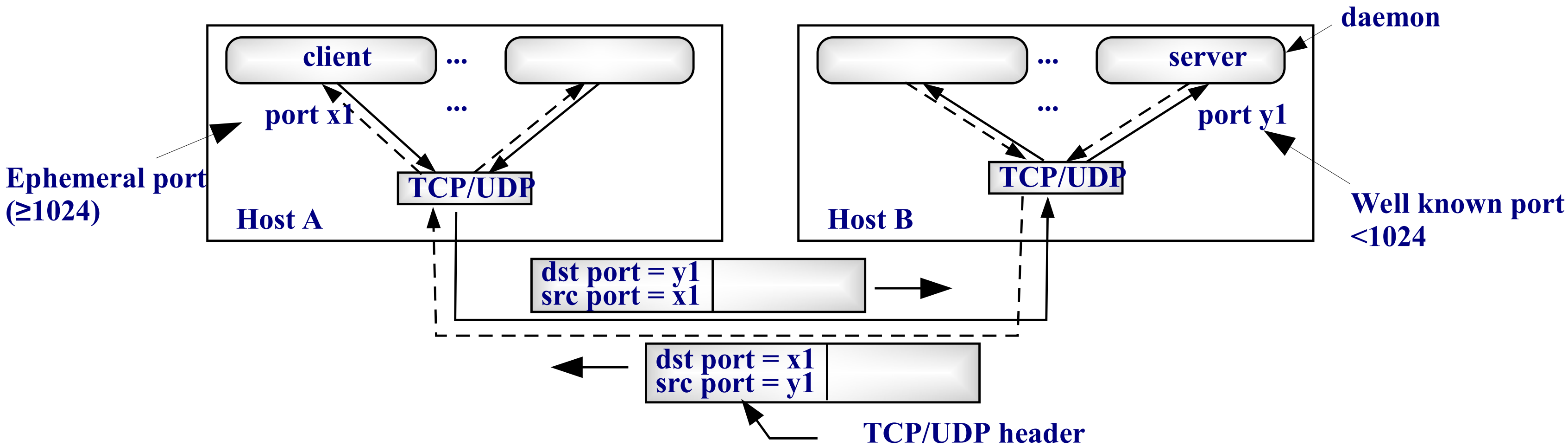
Outline

- **UDP Protocol**
- ARQ Protocols
- TCP Protocol

Unit 4. TCP

UPD Protocol – Introduction: The Internet Transport Layer

- Two protocols are used at the TCP/IP transport layer: User Datagram Protocol (UDP) and Transmission Control Protocol (TCP).
- UDP offers a *datagram service* (non reliable).
- TCP offers a reliable service.
- Transport layer offers a communication channel between applications.
- Transport layer access points (applications) are identified by a 16 bits port numbers.
- TCP/UDP use the client/server paradigm:



Unit 4. TCP

UPD Protocol – Description (RFC 768)

- Datagram service: same as IP.
  - Non reliable
  - No error recovery
  - No ack
  - Connectionless
  - No flow control
- UDP PDU is referred to as UDP datagram.
- UDP does not have a Tx buffer: each *write* operation at application level generates a UDP datagram.
- UDP is typically used:
  - Applications where short messages are exchanged: e.g. DHCP, DNS, RIP.
  - Real time applications: e.g. Voice over IP, videoconferencing, stream audio/video. These applications does not tolerate large delay variations (which would occur using an ARQ).

Unit 4. TCP

UPD Protocol – UDP Header

```
 0                   1                   2                   3
 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

- Fixed size of 8 bytes.
- The source port can be set to 0 (zero) if the sending process does not expect messages in response.
- The length is in bytes of the UDP header and UDP data. The minimum length is 8 bytes (the length of the header).
- The checksum is computed including the UPD header without Checksum, the payload and an IP pseudo-header (srcIP, dstIP, protocol). Not mandatory in IPv4 (mandatory in IPv6).

0x11 (Unit 2)

Unit 4. TCP

UPD Protocol – UDP Header – Checksum

IPv4 pseudo header format

Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	0	Source IPv4 Address																															
4	32	Destination IPv4 Address																															
8	64	Zeroes								Protocol								UDP Length															
12	96	Source Port																Destination Port															
16	128	Length																Checksum															
20	160+	Data																															

Source https://en.wikipedia.org/wiki/User_Datagram_Protocol#IPv4_pseudo_header

Same value

- Zeros: one byte set to zero.
- Protocol: 0x11 (see Unit 2).
- UDP Length: the length of the UDP header and data in bytes (including the 2B of the checksum).
- Checksum: ‘The 16-bit one's complement of the one's complement sum of a pseudo header of information from the IP header, the UDP header, and the data, padded with zero octets at the end (if necessary) to make a multiple of two octets.’ (RFC768)

Unit 4. TCP

Outline

- UDP Protocol
- **ARQ Protocols**
- TCP Protocol

Unit 4. TCP

ARQ protocols - Introduction

- Automatic Repeat reQuest (ARQ) protocols build a communication channel between endpoints, adding functionalities of the type:
  - Error detection
  - Error recovery
  - Flow control

Recall: IP does not provide a reliable communication channel

- Basic ARQ Protocols:
  - Stop & Wait
  - Go Back N
  - Selective Retransmission (Selective Repeat)
- Used in:
  - Transport layer: TCP
  - Data link layer: wireless (IEEE 802.11)
- TCP uses a variant of Go-Back-N (with SACK -see TCP slides- it uses Selective Repeat)

Unit 4. TCP

ARQ protocols - Introduction

ARQ Ingredients

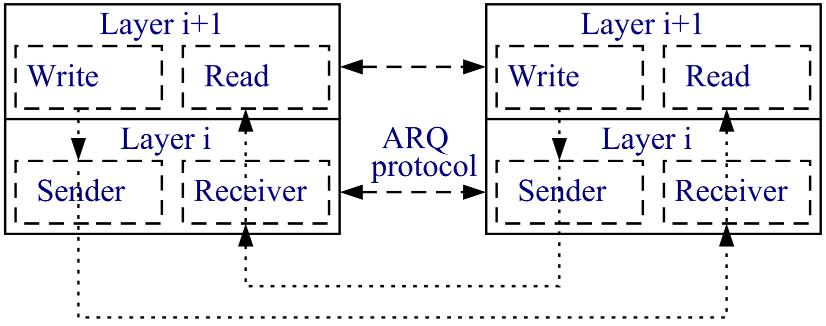
- Connection oriented
- Tx/Rx buffers
- Acknowledgments (ack)
- Acks can be *piggybacked* in information PDUs sent in the opposite direction
- Retransmission Timeout (RTO)
- Sequence Numbers

Recall: connection oriented protocol vs. connectionless protocol

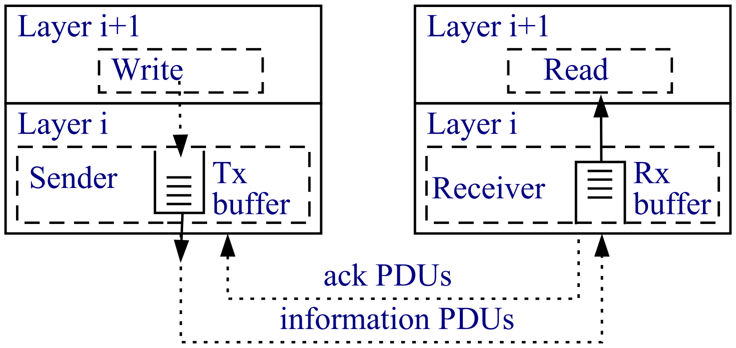
Connection oriented:
Phase 1: connection establishment
Phase 2: data stream
Phase 3: connection termination

NACK (negative ACK) to indicate some kind of error

Send ACK in data packages sent in the opposite direction



ARQ Protocol Architecture

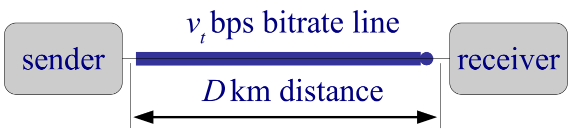


ARQ Protocol Implementation (one way)

Unit 4. TCP

ARQ Protocols - Assumptions

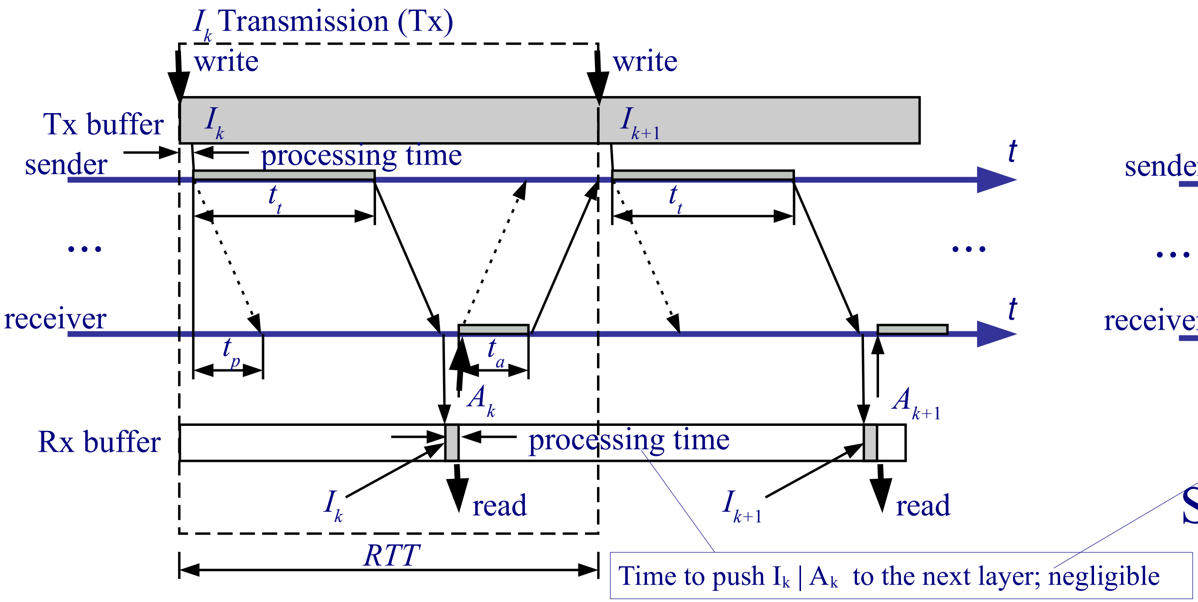
- We shall focus on the the transmission in one direction.
- We shall assume a saturated source: There is always information ready to send.
- We shall assume full duplex links.
- Protocol over a line of D m distance and v_t [bps] bitrate.
- Propagation speed of v_p [m/s], thus, propagation delay of D/v_p [s].
- We shall refer to a generic layer, where the sender sends Information PDUs (I_k) and the receiver sends ack PDUs (A_k).
- Frames carrying I_k and A_k, are Tx using L_I and L_A bits, thus the Tx times are respectively: $t_t = L_I/v_t$ and $t_a = L_A/v_t$ s.



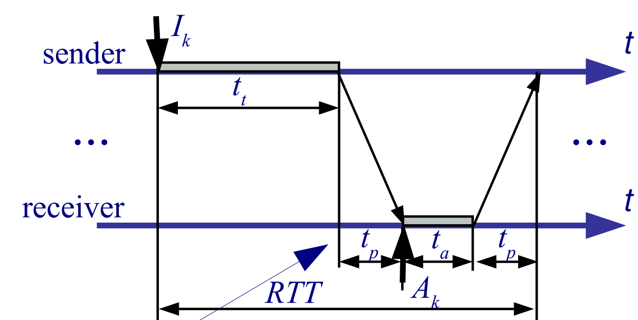
Unit 4. TCP

ARQ Protocols - Stop & Wait

1. When the sender is ready: (i) allows writing from upper layer, (ii) builds I_k, (iii) I_k goes down to data-link layer and Tx starts.
2. When I_k completely arrives to the receiver: (i) it is read by the upper layer, (ii) A_k is generated, A_k goes down to data-link layer and Tx starts.
3. When A_k completely arrives to the sender, goto 1.



Time diagram

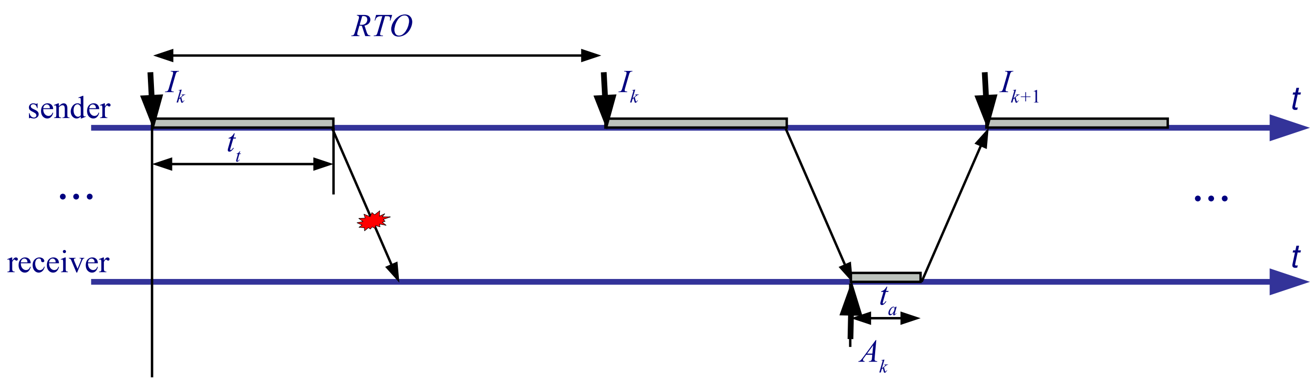


Simplified time diagram

Unit 4. TCP

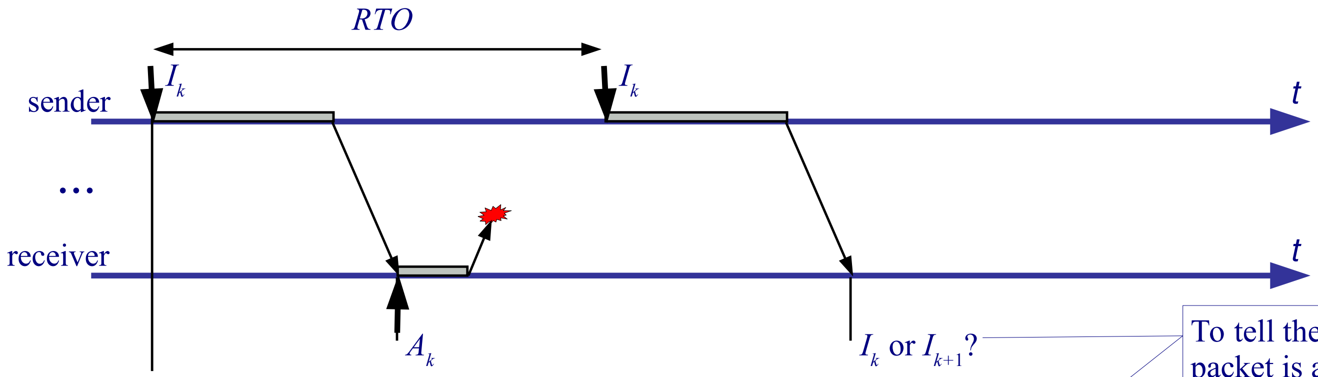
ARQ Protocols - Stop & Wait Retransmission

- Each time the sender Tx a PDU, a RTO is started.
- If the information PDU do not arrives, or arrives with errors, no ack is sent.
- When RTO expires, the sender ReTx (retransmit) the PDU.



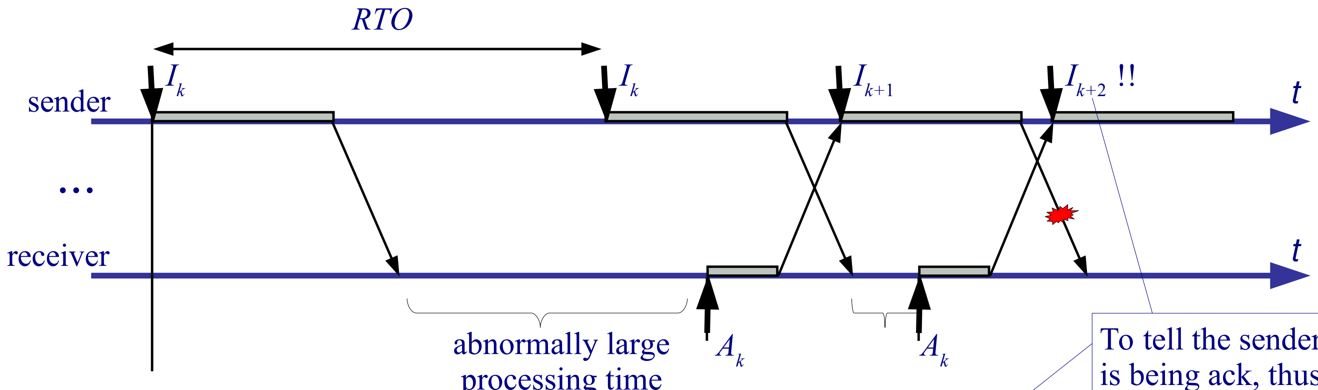
Unit 4. TCP

ARQ Protocols – Why sequence numbers are needed?



To tell the receiver that the next packet is a retransmission

Need to number information PDUs



To tell the sender which packet is being ack, thus, to avoid unnecessary retransmissions

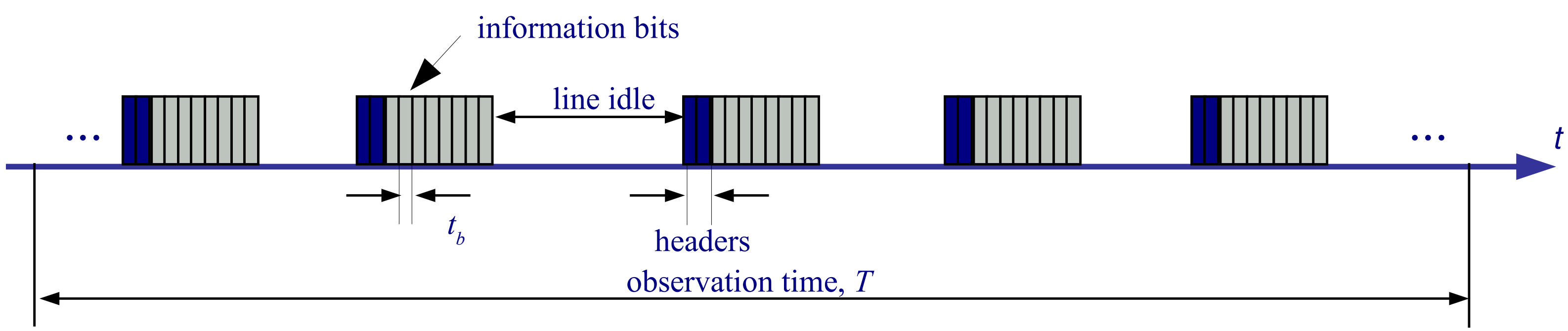
Need to number ack PDUs

Unit 4. TCP

ARQ Protocols – Notes on computing the efficiency (channel utilization)

- Line bitrate (*velocitat de transmissió de la línia*): $v_t = 1/t_b$, [bps]
- Throughput (*velocitat efectiva*) v_{ef} = number of inf. bits / obs. time, [bps]
- Efficiency or channel utilization $E = v_{ef} / v_t$ (times 100, in percentage)

The average transmission rate of the user data in bps

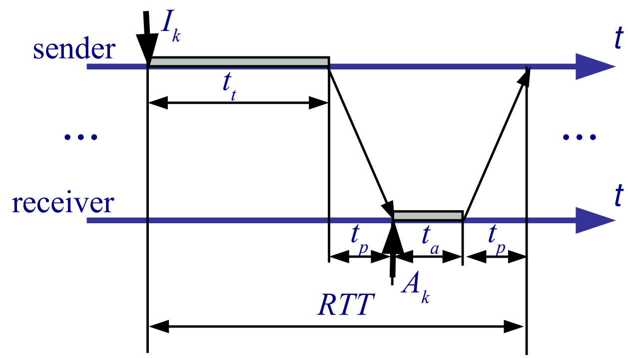


$$E = \frac{v_{ef}}{v_t} = \frac{\text{\#info bits}/T}{1/t_b} = \begin{cases} \dfrac{\text{\#info bits} \times t_b}{T} = \dfrac{\text{time Tx information}}{T} \\[2ex] \dfrac{\text{\#info bits}}{T/t_b} = \dfrac{\text{\#info bits}}{\text{\#bits at line bitrate}} \end{cases}$$

Unit 4. TCP

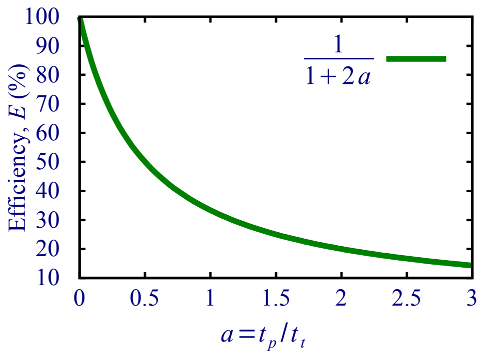
ARQ Protocols – Stop & Wait efficiency

- Assuming no errors (maximum efficiency), the Tx is periodic, with period *RTT*.
- $E_{protocol}$: We do not take into account headers.



$$E_{protocol} = \frac{t_t}{T_C} = \frac{t_t}{t_t + t_a + 2t_p} =$$

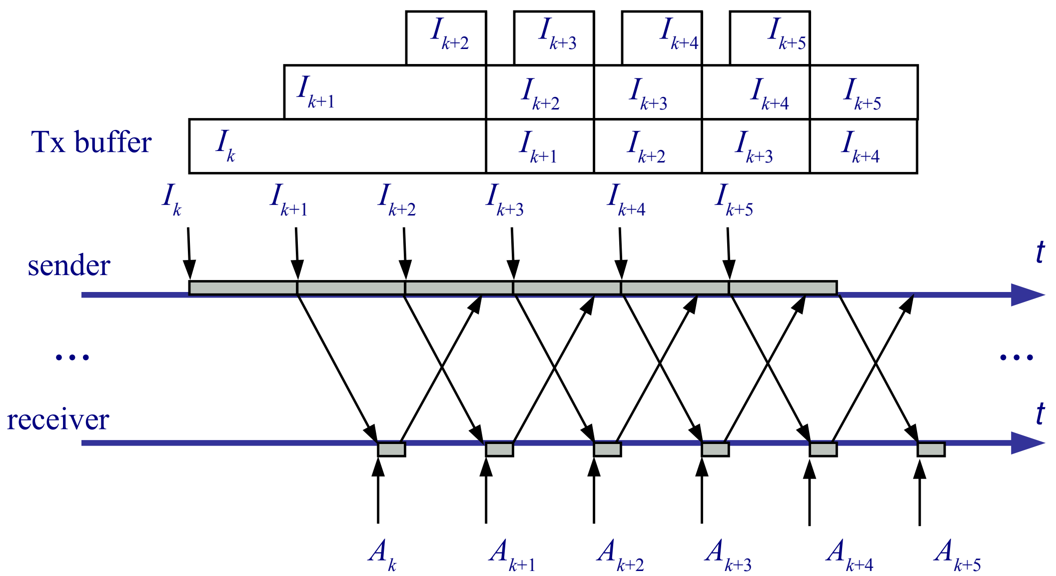
$$\frac{t_t}{t_t + 2t_p} \simeq \frac{1}{1+2a}, \text{ where } a = \frac{t_p}{t_t}$$



Unit 4. TCP

ARQ Protocols – Continuous Tx Protocols

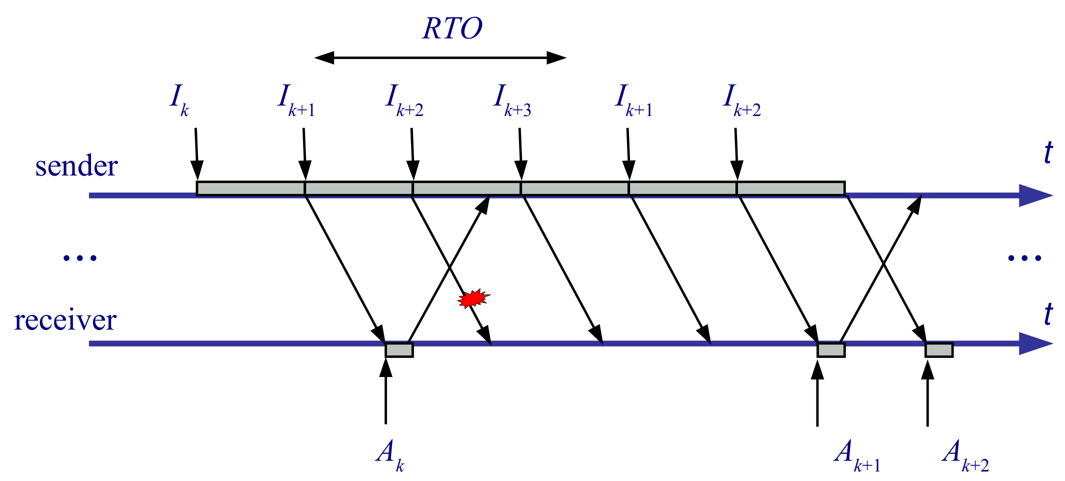
- Goal: Allow high efficiency independently of propagation delay.
- Without errors: E = 100 %
- PDUs transmitted “back-to-back” (i.e. without dealy)



Unit 4. TCP

ARQ Protocols – Go Back N

- Cumulative acks: A_k confirm I_i, $i \le k$ — i.e. A_k confirms all PDUs until k
- If the sender receives an error or out of order PDU: Do not send acks, discards all PDU until the expected PDU arrives. Thus, the receiver does not store out of order PDUs.
- When a RTO occurs, the sender *go back* and starts Tx from that PDU.

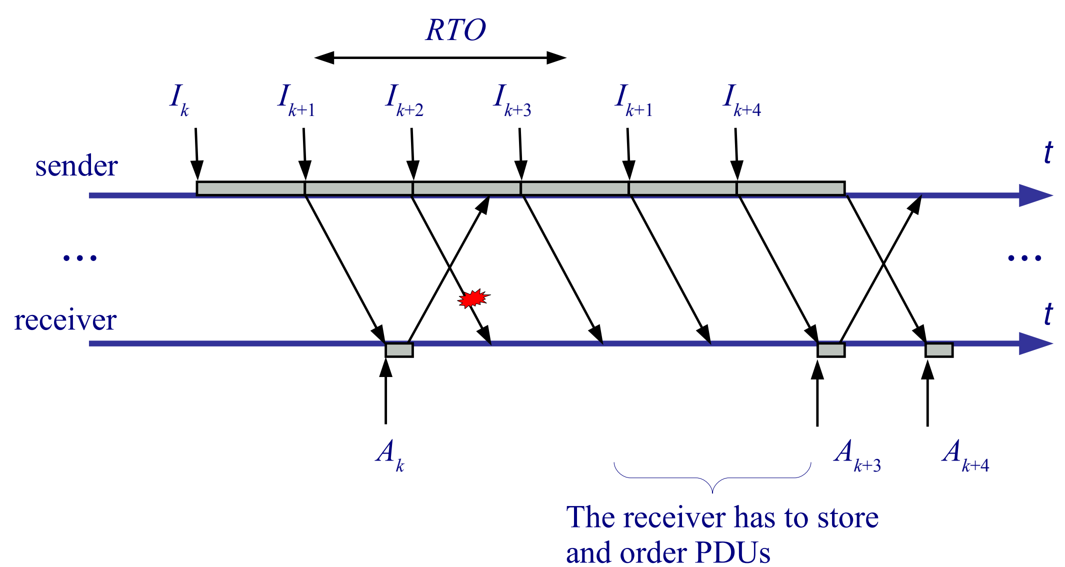


Unit 4. TCP

ARQ Protocols – Selective Retransmission (selective repeat)

- The 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.

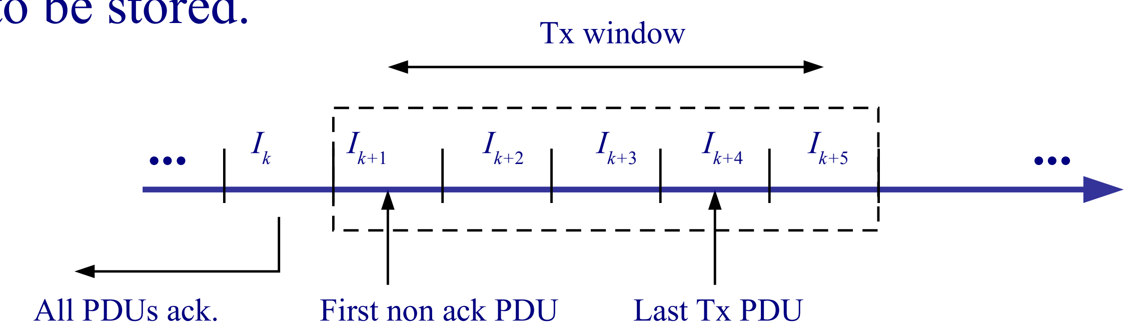
The implementation of the receiver is more complex because PDUs must be reordered



Unit 4. TCP

ARQ Protocols – Flow Control and Window Protocols

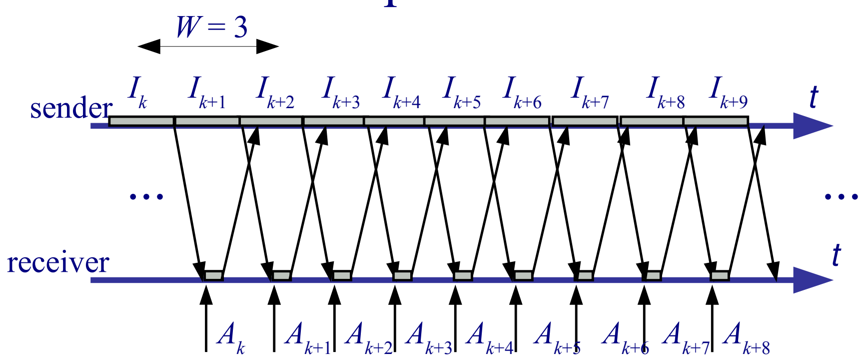
- ARQ are also used for flow control. Flow control consists on avoiding the sender to Tx at higher PDU rate than can be consumed by the receiver.
- With Stop & Wait, if the receiver is slower, acks are delayed and the sender reduces the throughput.
- With continuous Tx protocols: A *Tx window* is used. The window is the 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.
- Furthermore, the Tx window allows dimensioning the Tx buffer, and the Rx buffer for Selective Retransmission: No more the Tx window PDUs need to be stored.



Unit 4. TCP

ARQ Protocols – Optimal Tx window

- Optimal window: Minimum window that allows the maximum throughput.
- Optimal window example:

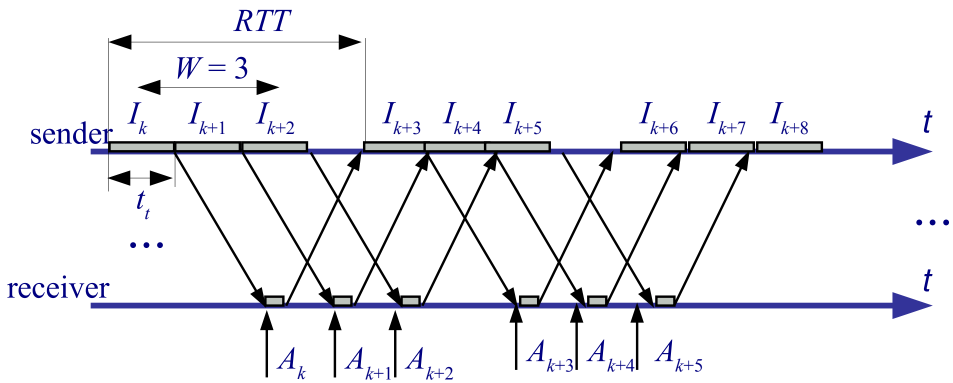


- Formula:

$$W_{opt}[PDU] = \left\lceil \frac{RTT}{t_t} \right\rceil$$

⌈ ⌉ stands for ceiling function

- Non optimal window example:



Unit 4. TCP

ARQ Protocols – Optimal Tx window

Recall: W_{opt} is the min. win. that allows max thtroughput, i.e. v_{ef}

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

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

Thus, we need a window $\geq$ 100 kbyte to reach 4Mbps with an RTT=200ms.

Unit 4. TCP

Outline

- UDP Protocol
- ARQ Protocols
- **TCP Protocol**

Unit 4. TCP

TCP Protocol – Description (RFC 793)

- Reliable service (ARQ with variable window)
  - Error recovery
  - Connection oriented
  - Flow control: Adapt throughput to the receiver
  - Congestion control: Adapt throughput to network
- Segments of optimal size: Maximum Segment Size (MSS).
  - MSS adjusted using MTU path discovery: send datagrams with Don't Fragment bit set, and reduce MSS upon receiving ICMP error messages.
- TCP PDU is referred to as TCP segment.
- TCP is typically used:
  - Applications requiring reliability: Web, ftp, ssh, telnet, mail, ...

Unit 4. TCP

TCP Protocol – Basic operation

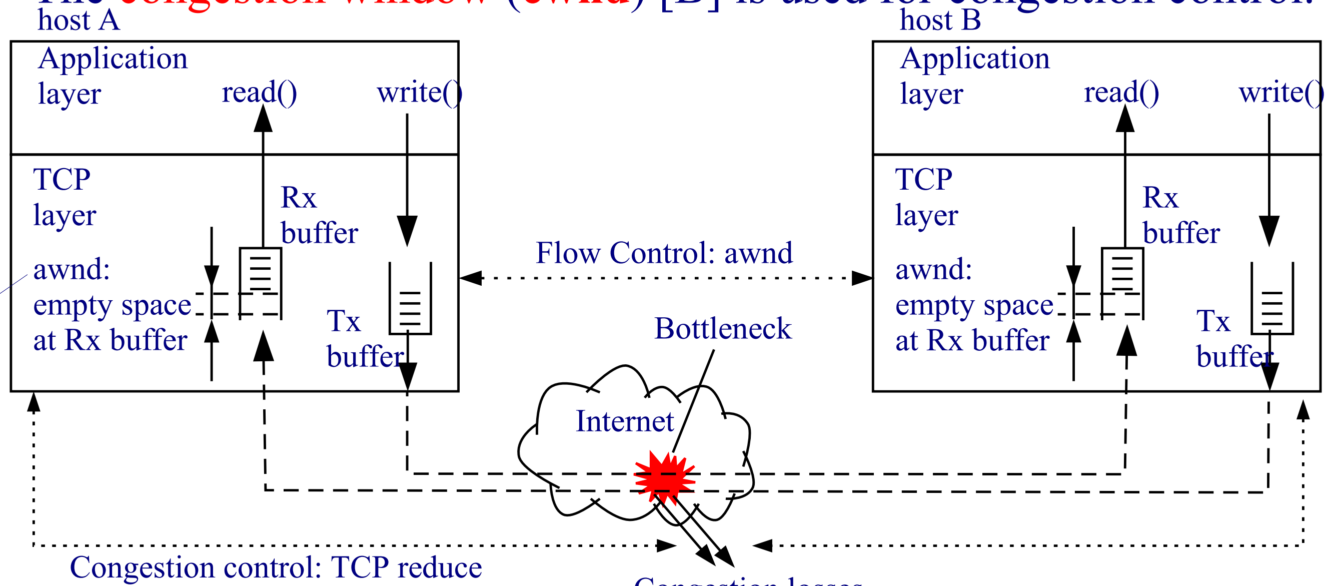
- TCP sender immediately sends the segments allowed by the window
- Upon segment arrival TCP receiver immediately sends sends an ack (unless delayed ack is used) without waiting for the upper layer to read the data.
  - Ack are cumulative
- ARQ window protocol, with **variable window**: [B] **wnd = min(**awnd, cwnd**)**
  - The advertised window (**awnd**) [B] is used for flow control.
  - The congestion window (**cwnd**) [B] is used for congestion control.

The Tx window

The awnd from the receiver

awnd is set by the receiver. E.g. to slow down the tx rate the receiver shrinks the awnd. The sender stores the receiver's awnd as the max number of bytes sent without confirmation

awnd is also known as Rx window



The awnd announced. Part of the TCP header, (thus, updated in every packet) because it may change frequently

Congestion control: TCP reduce the congestion window (cwnd) when losses are detected.

Unit 4. TCP

TCP Protocol – Delayed acks

- TCP connections can be classified as:
  - Bulk: (e.g. web, ftp) There are always bytes to send. TCP send MSS bytes.
  - 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 acks.

Delayed ack. It is used to reduce the amount of acks. Consists of sending 1 ack each 2 MSS segments whenever possible under the time limit of 200 ms. Acks are always sent in case of receiving out of order segments.

Remark: After SYN tcpdump extracts the initial sequence number from the sequence numbers

tcpdump example (bulk transfer):

It's a download

```
...
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)
...
```

Lenght (11089 - 9641 = 1448)

TCP flags

DF flag in IP header set.

timestamp | **src IP addr/port** | **dst IP addr/port** | **seq. num:next seq num (bytes)** | **ack** | **awnd**

```
11:27:13.798849 147.83.32.14.ftp > 147.83.35.18.3020: P 9641:11089(1448) ack 1 win 10136 (DF)
```

Unit 4. TCP

TCP Protocol – TCP Header

```
              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                        |
             +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  20 bytes   |                    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        |
             +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 ≤ 40 bytes  |                    Options                    |    Padding    |
             +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
                                                     TCP flags
```

Note: The length of the data section is not specified because it is not required at implementation level (the TCP stack just does a sizeof(buf) to find out how much data a segment has). However, it can be calculated by subtracting the combined length of the segment header and IP header from the total IP packet length specified in the IP header.

Conversely, the UDP includes the length because this data is needed at implementation level (obtaining it from the IP header is not acceptable at implementation level because it implies to mix data from different layers and because it cannot be assumed that it runs over IP).

- Variable size: Fixed fields of 20 bytes + options (15x4 = 60 bytes max.).
- The header length is in 32-bit words. Between 5 and 15).
- Reserved for future protocol extensions (currently 3bits already in use).
- Like in UDP, the checksum is computed using an IP pseudo-header:

In TCP it is mandatory

```
 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 Address                        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                      Destination Address                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Zero      |   Protocol    |          TCP length           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Unit 4. TCP

TCP Protocol – TCP Flags

- URG (Urgent): The Urgent Pointer is used. It points to the first urgent byte. Rarely used. Example: ^C in a telnet session.
- ACK: The ack field is used. Always set except for the first segment sent by the client.
- PSH (Push): The sender indicates to "push" all buffered data to the receiving application. Most BSD derived TCPs set the PSH flag when the send buffer is emptied.
- RST (Reset): Abort the connection.
- SYN: Used in the connection setup (*three-way-handshaking, TWH*). Only the first packet sent from each end should have this flag set (bcs it synchronizes sequence numbers). Some other flags and fields change meaning based on this flag.
- FIN: Used in the connection termination. Only set in the last packet from the sender.

If set (1), this is the initial sequence number, thus, the receiver has to acknowledge this sequence number plus one.
Otherwise (0), this is the accumulated seq. num. of the first data byte of this segment for the current session.

Unit 4. TCP

TCP Protocol – TCP Flags

- tcpdump example:

Note: After SYN tcpdump extracts the initial sequence number from the sequence numbers

TCP flags
S: SYN
P: PUSH
.: No flag (except ack) is set

3624662632+1

```
09:33:02.556785 IP 147.83.34.125.24374 > 147.83.194.21.80: S 3624662632:3624662632(0) win 5840
                <mss 1460,sackOK,timestamp 531419155 0,nop,wscale 7>
09:33:02.558054 IP 147.83.194.21.80 > 147.83.34.125.24374: S 2204366975:2204366975(0) ack
                3624662633 win 5792 <mss 1460,sackOK,timestamp 3872304344 531419155,nop,wscale 2>
09:33:02.558081 IP 147.83.34.125.24374 > 147.83.194.21.80: . ack 1 win 46 <nop,nop,timestamp
                531419156 3872304344>
09:33:02.558437 IP 147.83.34.125.24374 > 147.83.194.21.80: P 1:627(626) ack 1 win 46
                <nop,nop,timestamp 531419156 3872304344>
09:33:02.559146 IP 147.83.194.21.80 > 147.83.34.125.24374: . ack 627 win 1761 <nop,nop,timestamp
                3872304345 531419156>
09:33:02.559507 IP 147.83.194.21.80 > 147.83.34.125.24374: P 1:271(270) ack 627 win 1761
                <nop,nop,timestamp 3872304345 531419156>
09:33:02.559519 IP 147.83.34.125.24374 > 147.83.194.21.80: . ack 271 win 54 <nop,nop,timestamp
                531419156 3872304345>
09:33:02.560154 IP 147.83.194.21.80 > 147.83.34.125.24374: . 271:1719(1448) ack 627 win 1761
                <nop,nop,timestamp 3872304345 531419156>
09:33:02.560167 IP 147.83.34.125.24374 > 147.83.194.21.80: . ack 1719 win 77 <nop,nop,timestamp
                531419156 3872304345>
09:33:02.560256 IP 147.83.194.21.80 > 147.83.34.125.24374: . 1719:3167(1448) ack 627 win 1761
                <nop,nop,timestamp 3872304345 531419156>
09:33:02.560261 IP 147.83.34.125.24374 > 147.83.194.21.80: . ack 3167 win 100 <nop,nop,timestamp
                531419156 3872304345>
...
```

Unit 4. TCP

TCP Protocol – TCP Options

"suggest" to the other end

- Maximum Segment Size (MSS): Used in the TWH to initialize the MSS. Only with SYN set.
  - Common values in IPv4:
    - **MSS 1460**: 1500-40 (MTU [1500B] - (IPv4 header [20B] + TCP header without options [20B]))
    - **MSS 1448**: 1500-52 (MTU [1500B] - (IPv4 header [20B] + TCP [20B] with timestamp options [10B] and 2 nops [1B])). Currently the most common value.
- Window Scale factor: Used in the TWH. The awnd is multiplied by $2^{\text{Window Scale}}$ (i.e. the window scale indicates the number of bits to left-shift awnd). It allows using awnd larger than 2^{16} bytes. Only with SYN set.
- Timestamp: Used to compute the Round Trip Time (RTT). Is a 10 bytes option, with the timestamp clock of the TCP sender, and an echo of the timestamp of the TCP segment being ack. Used to set RTO.
- SACK: In case of errors, indicate blocks of consecutive correctly received segments for Selective Retransmission. Only with SYN set.
- NOP: Used for padding (1B).
- Etc. (~35 in total; may be extended in the future)

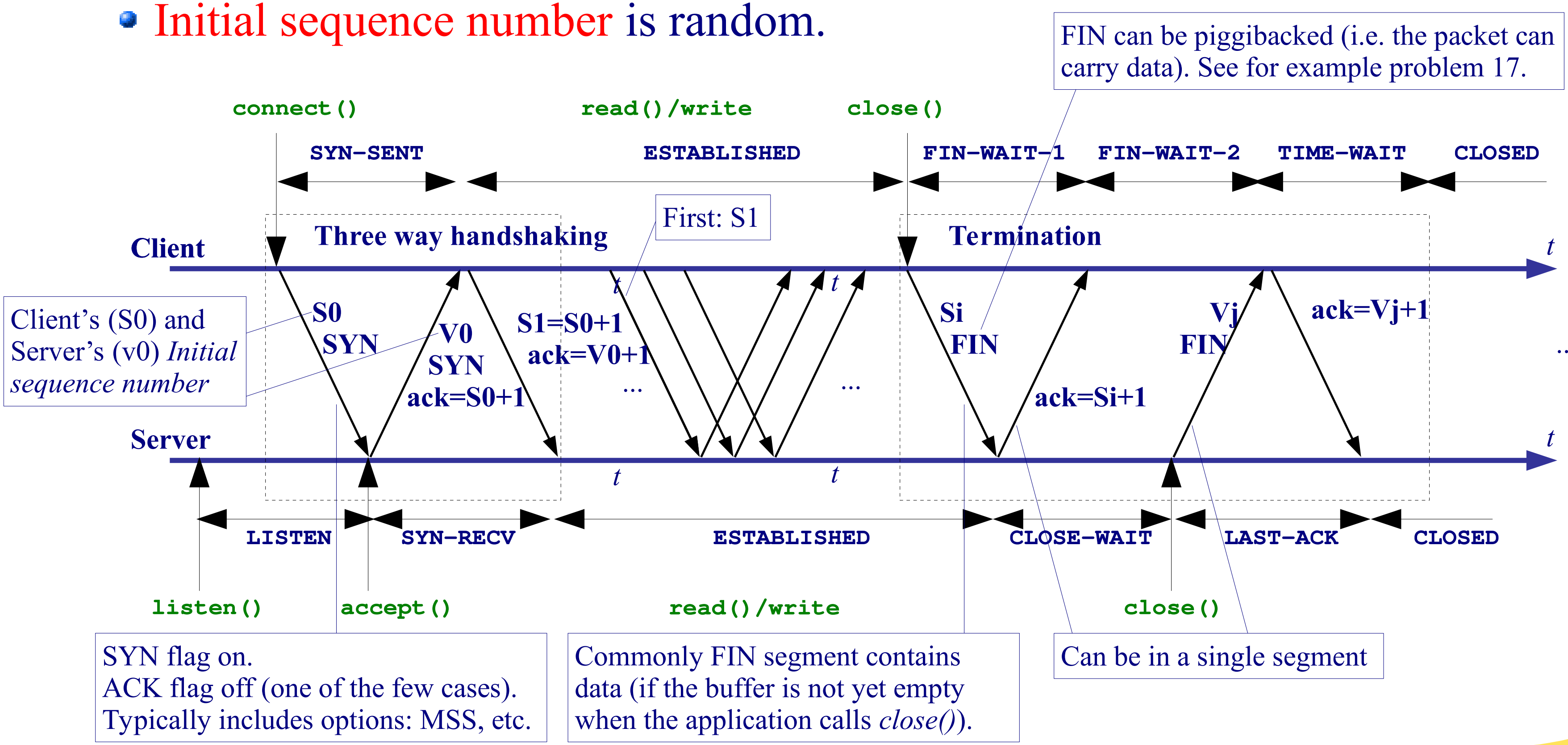
Remark: In TCP the options are **frequently** used (contrary to IP)

Unit 4. TCP

TCP Protocol – Connection Setup and Termination

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

Conversely, the FIN might be sent by the server first



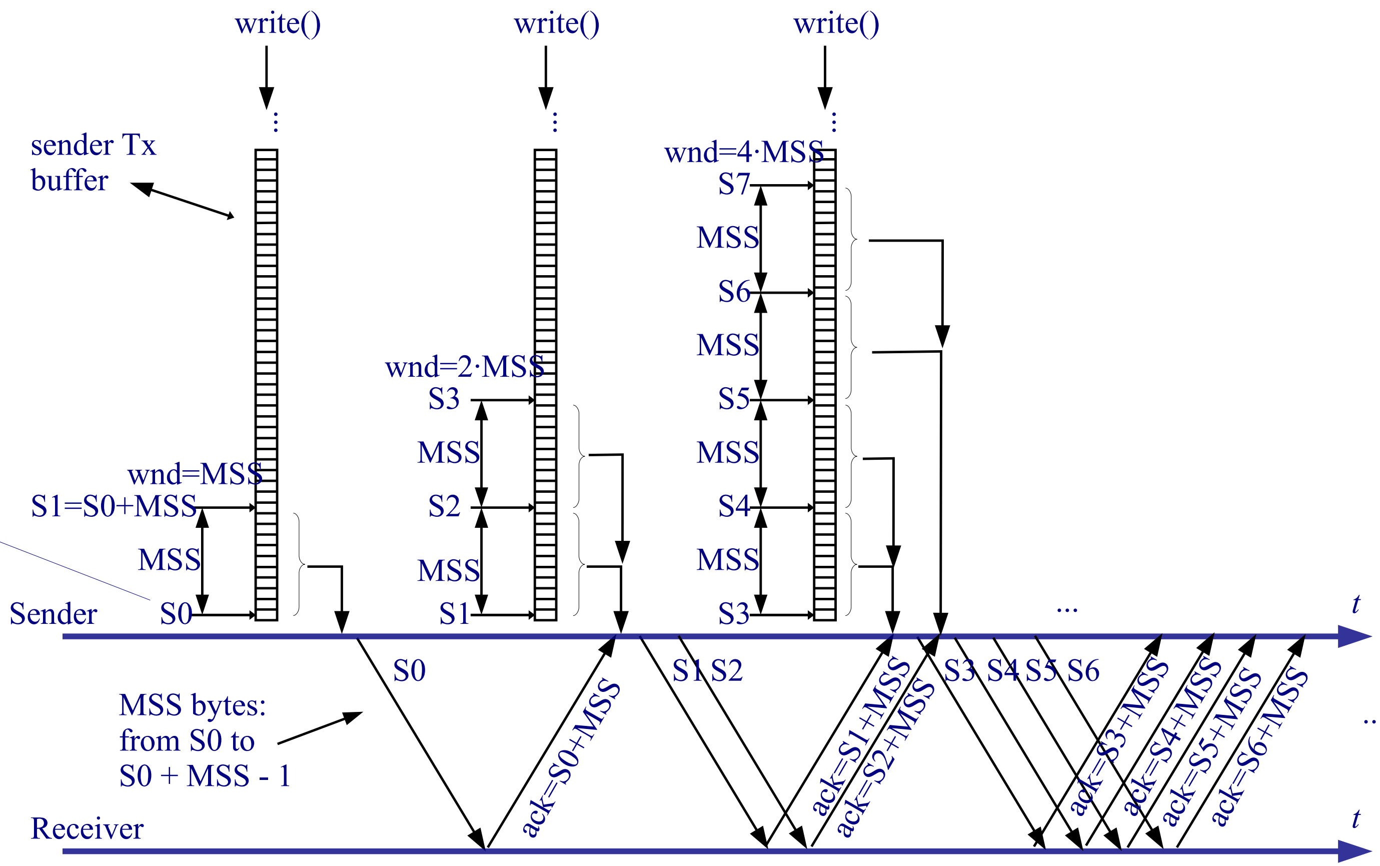
Unit 4. TCP

TCP Protocol – TCP Sequence Numbers [byte]

- The sequence number identifies the first payload byte.
- The ack number identifies the next byte the receiver is waiting for.

Initial sequence number (ISN) are arbitrarily chosen (e.g. hashing IP+ports+etc.) as a measure a to mitigate the TCP sequence prediction attack

If the segment follows right after the SYN/SYN-ACK/ACK sequence seen in the previous slide and sender is the client, the "S0" of this slide equals to "S1" of the previous one (i.e. S0+1)



Unit 4. TCP

TCP Protocol – tcpdump example (web page download)

win is recommended to be a multiple of mss for optimisation reasons, but it is just a recommendation.

TWH

Data stream

Termination

```
1  12:30:37.069541 IP 147.83.34.125.17788 > 147.83.32.82.80: S 3473661146:3473661146(0) win 5840 <mss 1460,sackOK,timestamp
                        296476754 0,nop,wscale 7>
2  12:30:37.070021 IP 147.83.32.82.80 > 147.83.34.125.17788: S 544373216:544373216(0) ack 3473661147 win 5792 <mss
                        1460,sackOK,timestamp 1824770623 296476754,nop,wscale 2>
3  12:30:37.070038 IP 147.83.34.125.17788 > 147.83.32.82.80: . ack 1 win 46 <nop,nop,timestamp 296476754 1824770623>
4  12:30:37.072763 IP 147.83.34.125.17788 > 147.83.32.82.80: P 1:602(601) ack 1 win 46 <nop,nop,timestamp 296476754 1824770623>
5  12:30:37.073546 IP 147.83.32.82.80 > 147.83.34.125.17788: . ack 602 win 1749 <nop,nop,timestamp 1824770627 296476754>
6  12:30:37.075932 IP 147.83.32.82.80 > 147.83.34.125.17788: P 1:526(525) ack 602 win 1749 <nop,nop,timestamp 1824770629 296476754>
7  12:30:37.075948 IP 147.83.34.125.17788 > 147.83.32.82.80: . ack 526 win 54 <nop,nop,timestamp 296476755 1824770629>
8  12:30:53.880704 IP 147.83.32.82.80 > 147.83.34.125.17788: F 526:526(0) ack 602 win 1749 <nop,nop,timestamp 1824787435 296476755>
0  12:30:53.920354 IP 147.83.34.125.17788 > 147.83.32.82.80: . ack 527 win 54 <nop,nop,timestamp 296480966 1824787435>
10 12:30:56.070200 IP 147.83.34.125.17788 > 147.83.32.82.80: F 602:602(0) ack 527 win 54 <nop,nop,timestamp 296481504 1824787435>
11 12:30:56.070486 IP 147.83.32.82.80 > 147.83.34.125.17788: . ack 603 win 1749 <nop,nop,timestamp 1824789625 296481504>
```

1, 2, 3: Three way-handshake
4: The client (147.83.34.125) requests a web page (e.g. index.html)
5: The server (147.83.32.82) ACKs the client's request
6: The server sends the web page in a single segment
7: The client ACKs the segment containing the web page
8: The server starts the connection termination
9: The client ACKs server's termination
10: The client starts the connection termination
11: The server ACKs client's termination

Recall: Flags
S SYN
· ACK only
P Push
F FIN

TCP flags

timestamp | src IP addr/port | dst IP addr/port | seq. num:next seq num (bytes) | awnd

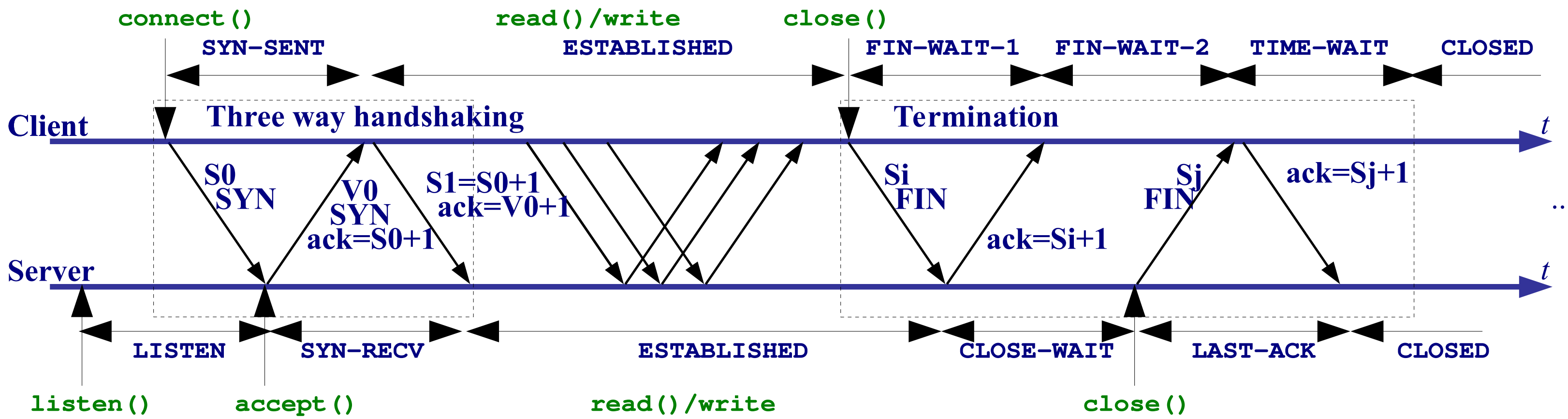
```
12:30:37.069541 IP 147.83.34.125.17788 > 147.83.32.82.80: S 3473661146:3473661146(0) win 5840
          <mss 1460,sackOK,timestamp 296476754 0,nop,wscale 7>
```

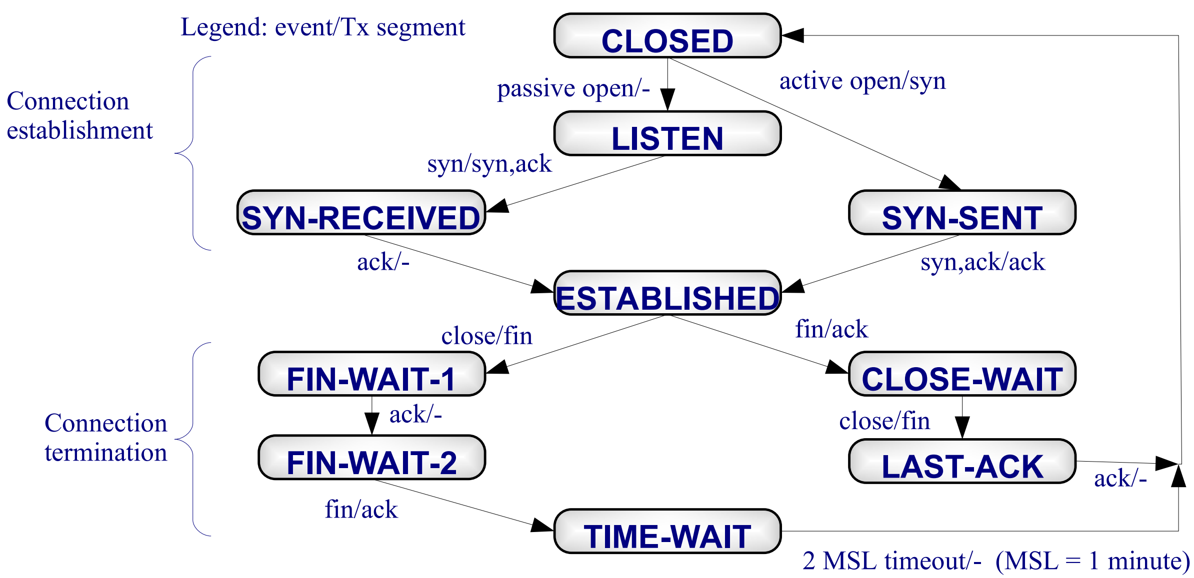
MSS | SACK | timestamp | window scale

padding

Unit 4. TCP

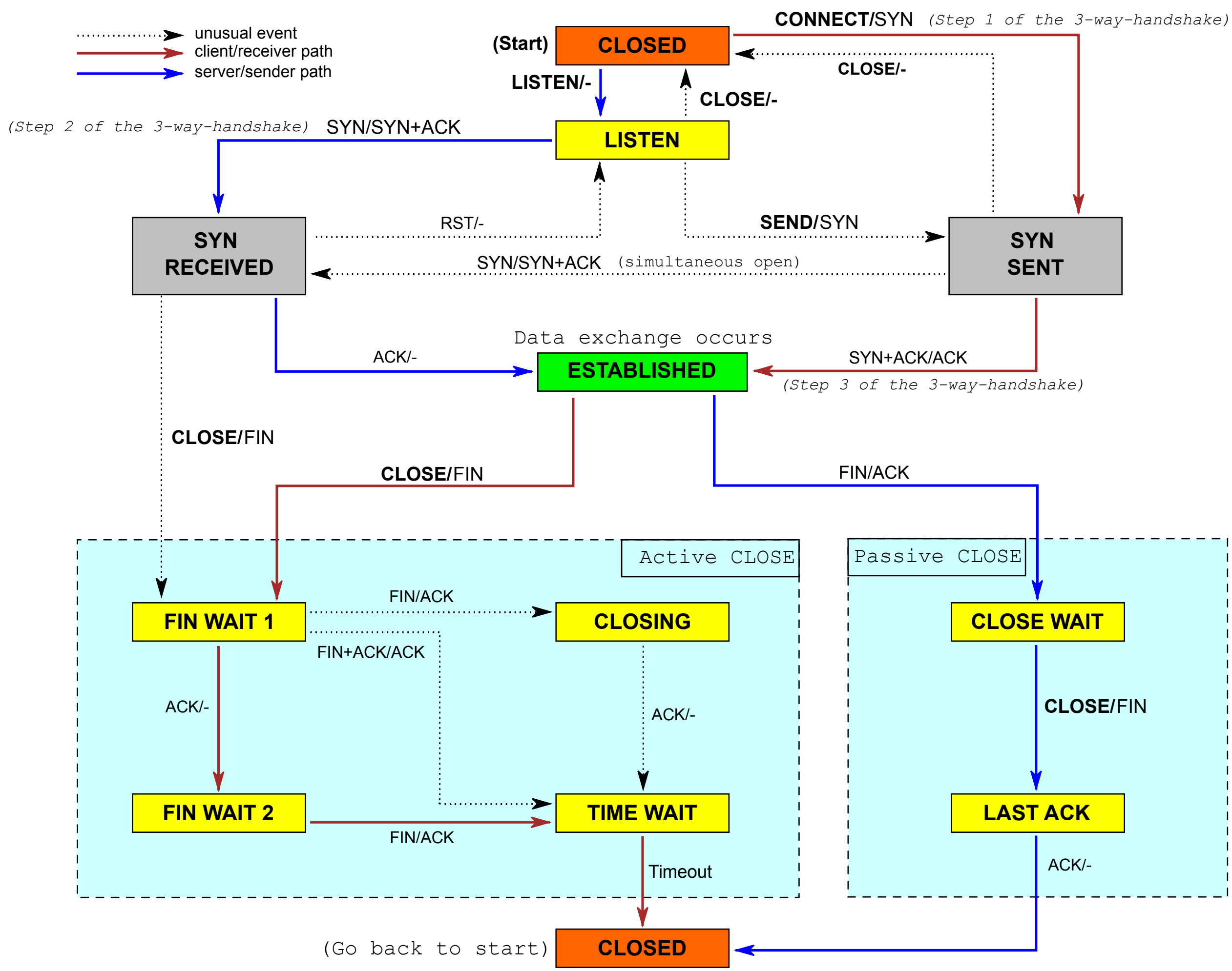
TCP Protocol – State diagram (simplified)





Unit 4. TCP

TCP Protocol – State diagram (simplified)



Source https://upload.wikimedia.org/wikipedia/commons/f/f6/Tcp_state_diagram_fixed_new.svg

Unit 4. TCP

TCP Protocol – netstat dump

- Option -t shows tcp sockets.

```
linux# netstat -nt
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address          Foreign Address        State
tcp        0   1286 192.168.0.128:29537    199.181.77.52:80       ESTABLISHED
tcp        0      0 192.168.0.128:13690    67.19.9.2:80           TIME_WAIT
tcp        0      1 192.168.0.128:12339    64.154.80.132:80       FIN_WAIT1
tcp        0      1 192.168.0.128:29529    199.181.77.52:80       SYN_SENT
tcp        1      0 192.168.0.128:17722    66.98.194.91:80        CLOSE_WAIT
tcp        0      0 192.168.0.128:14875    210.201.136.36:80      ESTABLISHED
tcp        0      0 192.168.0.128:12804    67.18.114.62:80        ESTABLISHED
tcp        0      1 192.168.0.128:25232    66.150.87.2:80         LAST_ACK
tcp        0      0 192.168.0.128:29820    66.102.9.147:80        ESTABLISHED
tcp        0      0 192.168.0.128:29821    66.102.9.147:80        ESTABLISHED
tcp        1      0 127.0.0.1:25911        127.0.0.1:80           CLOSE_WAIT
tcp        0      0 127.0.0.1:25912        127.0.0.1:80           ESTABLISHED
tcp        0      0 127.0.0.1:80           127.0.0.1:25911        FIN_WAIT2
tcp        0      0 127.0.0.1:80           127.0.0.1:25912        ESTABLISHED
```

man netstat

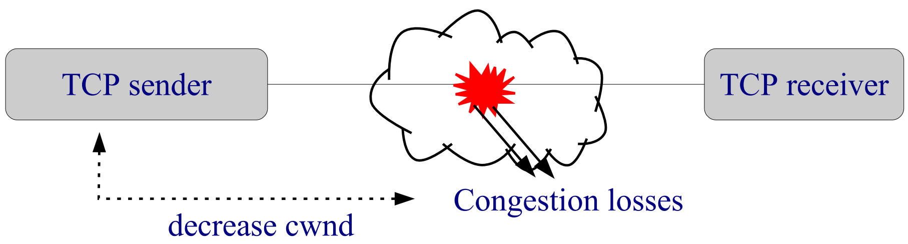
```
The count of bytes not acknowledged by the remote host.

The count of bytes not copied by the user program connected to this
socket.
```

The command `lsof` provides similar info.
These commands an be conveniently combined with `watch` e.g.:
`watch -d lsof -n -i @127.0.1.1`

Unit 4. TCP

TCP Protocol – Congestion Control (RFC 2581)

- **wnd = min(awnd, cwnd)**
  - The advertised window (awnd) is used for flow control.
  - The congestion window (cwnd) is used for congestion control.
- TCP interprets losses as congestion:



- Basic Congestion Control Algorithm:
  - Slow Start / Congestion Avoidance (SS/CA)

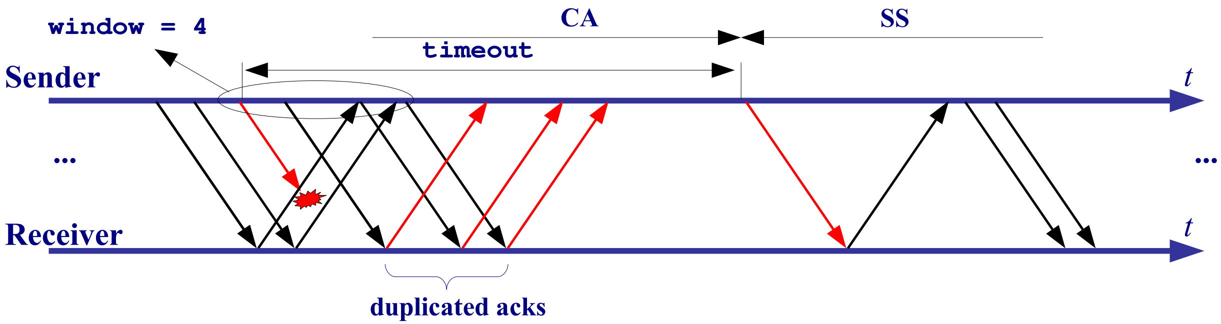
Unit 4. TCP

TCP Protocol – Slow Start / Congestion Avoidance (SS/CA)

- Variables:
  - snd_una: First non ack segment (head of the TCP transmission queue).
  - ssthresh: Threshold between SS and CA.

```
Initialization:
        cwnd = MSS ; NOTE: RFC 2581 allows an initial window of 2 segments.
        ssthresh = infinity ;

Each time an ack confirming new data is received:
        if(cwnd < ssthresh) {
                cwnd += MSS ;                /* Slow Start */
        } else {
                cwnd += MSS * MSS / cwnd ; /* Congestion Avoidance */
        }

When there is a time-out:
        Retransmit snd_una ;
        ssthresh = max(min(awnd, cwnd) / 2, 2 MSS) ;
        cwnd = MSS ;
```

Time-out Example:

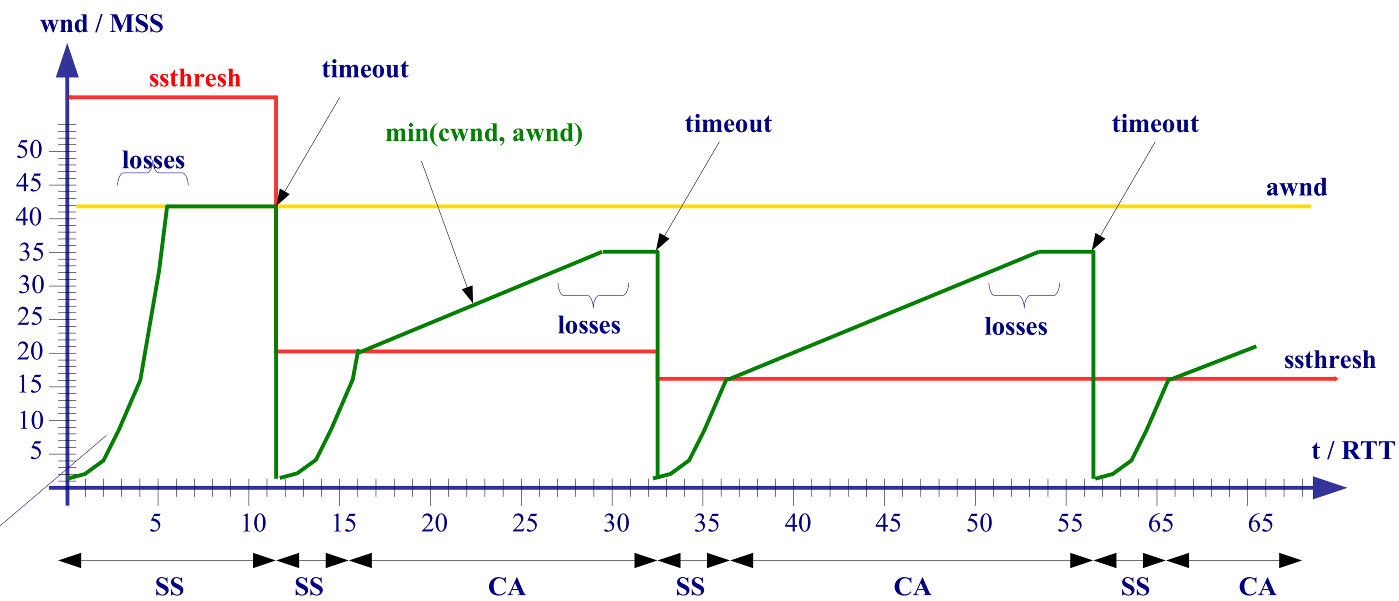


Unit 4. TCP

TCP Protocol – Slow Start / Congestion Avoidance (SS/CA)

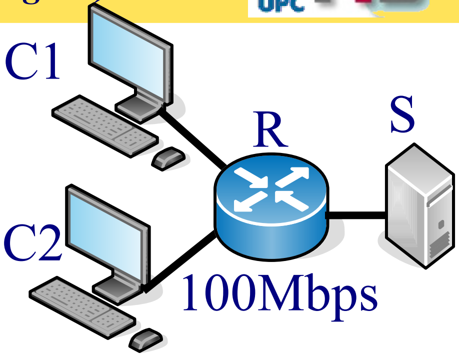
- During SS cwnd is rapidly increased to the "operational point".
- During CA cwnd is slowly increased looking for more available "bandwidth".

```
Initialization:
        cwnd = MSS ;
        ssthresh = infinit ;

Each time an ack confirming new data is received:
        if(cwnd < ssthresh) {
                cwnd += MSS ;                  /* SS */
        } else {
                cwnd += MSS * MSS / cwnd ; /* CA */
        }

When there is a time-out:
        Retransmit snd_una ;
        ssthresh = max(min(awnd, cwnd) / 2, 2 MSS) ;
        cwnd = MSS ;
```



"slow start" is somehow contradictory given the exponential shape :p

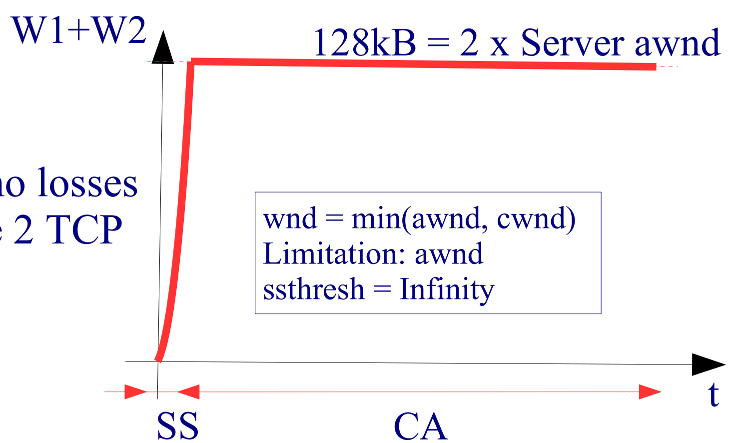
Unit 4. TCP



TCP Protocol – Evaluation without losses

- Preliminaries:
  - TCP sends the entire window, W (in several segments)
  - The segments accumulate in the queues of the interfaces where there are bottlenecks
  - Steady state: the TCP connection started time ago
  - In general, we can assume that, on the average, is fulfilled $\boxed{vef = W / RTT}$
  - If there are no losses, W will be awnd, otherwise W follows a "saw tooth"

- Example without losses: C1 and C2 send to S, each with a TCP connection. Server awnd = 64kB. Router queues ≧ 128kB.
  - The bottleneck is the link R-S
  - For each connection vef = 100/2 = 50 Mbps
  - If the propagation delays in the links are negligible and no losses occur in the queue of the router there will be 128 kB (the 2 TCP windows)
    - The RTT is the time in the queue of the router:
      - RTT=128 kB/100 Mbps = 10,24 ms
    - Check that vef=W/RTT = 64 kB/10,24 ms = 50 Mbps



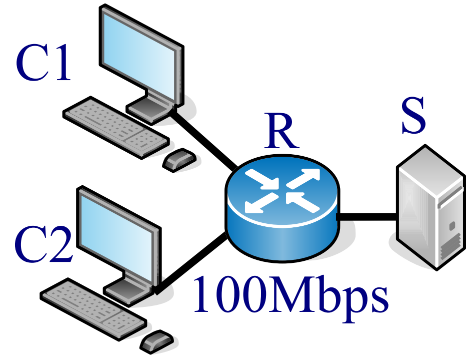
Unit 4. TCP

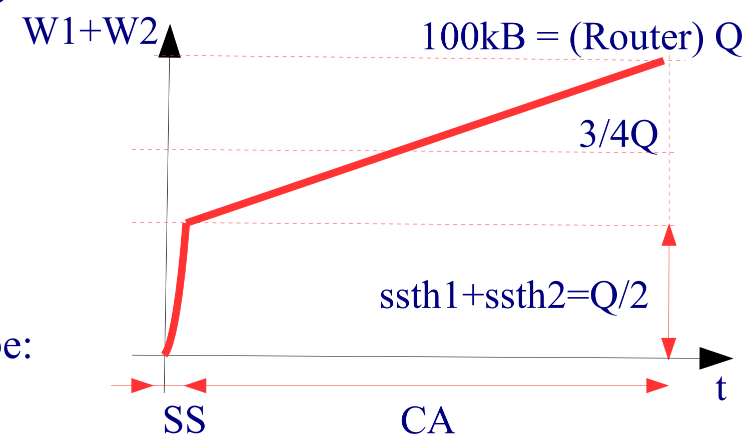
TCP Protocol – Evaluation with losses

- Example with losses: C1 and C2 send to S, each with a TCP connection. Server awnd = 64kB. Assume now that the interface queue of the router is limited to Q = 100 kB
  - The bottleneck is the link R-S
  - For each connection vef = 100/2 = 50 Mbps
  - There will be losses, because when both TCP windows add to 100kB, there will be no space left in the router queue.
  - The figure shows a possible evolution of the queue in the router, which stores the window of both connections: W1+W2. When the queue is full, both connections have losses and reduce the ssth to the half. Therefore, the average queue size in the router will be, approximately:

    (Q/2+Q)/2=3/4Q=75 kB
  - Thus, the average RTT will be:
    - $\overline{\text{RTT}}$=75 kB/100 Mbps = 6 ms
  - Note that the average window of each connection will be:

    $\overline{\text{W1}}$=$\overline{\text{W2}}$=75 kB/2=37,5 kB
  - Check that vef=$\overline{\text{W}}$/$\overline{\text{RTT}}$ = 37,5 kB/6 ms = 50 Mbps





wnd = min(awnd, cwnd)
Limitation: cwnd
ssthresh = Q/2

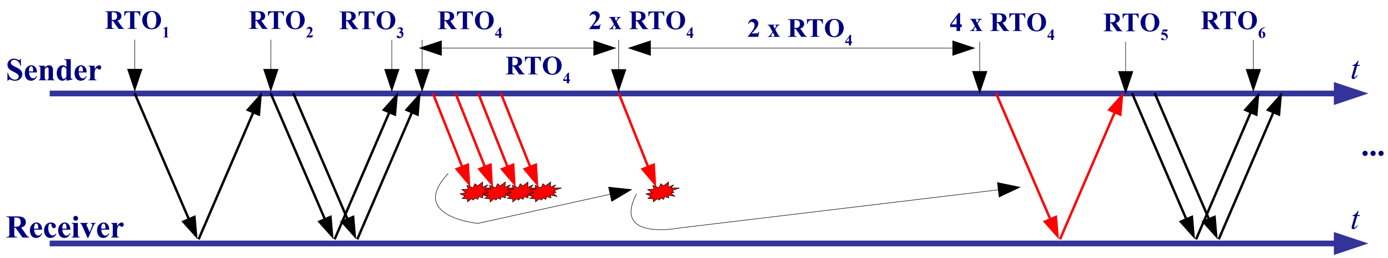
Unit 4. TCP

TCP Protocol – Retransmission time-out (RTO)

- Activation:
  - RTO is active whenever there are pending acks.
  - When RTO is active, it is continuously decreased, and a ReTx occurs when RTO reaches zero.
  - Each time an ack confirming new data arrives:
    - RTO is computed.
    - RTO is restarted if there are pending acks, otherwise, RTO is stopped.
- Computation:
  - The TCP sender measures the RTT mean (srtt) and variance (rttvar).
  - The retransmission time-out is given by: RTO = srtt + 4 x rttvar.
  - RTO is duplicated each retransmitted segment (exponential backoff).
- RTT measurements:
  - Using "slow-timer tics" (coarse).
  - Using the TCP timestamp option.

Unit 4. TCP

TCP Protocol – Retransmission time-out (RTO)



Unit 4. TCP

TCP Protocol – Example: TCP echo server

- Code extracted from https://docs.python.org/3/library/socket.html
- Run the example in 3 terminals (you can also run it step-by-step in 2 python consoles):
  1) `wireshark -n -i any -k -f "host 127.0.1.1"`
  2) `python3 serverTCP.py`
  3) `python3 clientTCP.py`

Attention!!!

serverTCP.py

```
import socket
HOST = ''      # Symbolic name meaning all available interfaces
PORT = 50007   # Arbitrary non-privileged port

MAX_DATA = 128*1448  # Must be larger than the data sent by
                     # the client (RST otherwise)
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.setsockopt(socket.IPPROTO_TCP, socket.TCP_MAXSEG, 1460)
    s.bind((HOST, PORT))
    s.listen(1)
    conn, addr = s.accept()
    with conn:
        print('Connected by', addr)
        while True:
            data = conn.recv(MAX_DATA)
            if not data: break
            conn.sendall(data)
```

clientTCP.py

```
# WARNING: GNU/Linux - To make the OS stick to MSS run:
#    sudo ethtool -K lo tso off

import socket

HOST = '127.0.1.1'     # Server IP
PORT = 50007           # Server port
MSS = 1448             # TCP MSS

#MSG = s.sendall(b'Hello, world')
MSG = 2*MSS*b'a'

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.setsockopt(socket.IPPROTO_TCP, socket.TCP_MAXSEG, MSS)
    s.connect((HOST, PORT))
    s.sendall(MSG)
    data = s.recv(len(MSG))
```

generic-receive-offload: off

print('Received', repr(data))

Unit 4. TCP

TCP Protocol – Example: TCP echo server

Time (format as specified)

Time	No.	Source	Destination	Protoco	Length	Info
0.000000000	1	127.0.0.1	127.0.1.1	TCP	76	35964 → 50007 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=1 TSval=919802755 TSecr=0 WS=128
[illegible]	2	127.0.1.1	127.0.0.1	TCP	76	50007 → 35964 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM=1 TSval=4252976948 TSecr=919802755 WS=128
0.000029126	3	127.0.0.1	127.0.1.1	TCP	68	35964 → 50007 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=919802755 TSecr=4252976948
0.000060134	4	127.0.0.1	127.0.1.1	TCP	80	35964 → 50007 [PSH, ACK] Seq=1 Ack=1 Win=65536 Len=12 TSval=919802755 TSecr=4252976948
0.000064953	5	127.0.1.1	127.0.0.1	TCP	68	50007 → 35964 [ACK] Seq=1 Ack=13 Win=65536 Len=0 TSval=4252976948 TSecr=919802755
0.000213641	6	127.0.1.1	127.0.0.1	TCP	80	50007 → 35964 [PSH, ACK] Seq=1 Ack=13 Win=65536 Len=12 TSval=4252976948 TSecr=919802755
0.000220747	7	127.0.0.1	127.0.1.1	TCP	68	35964 → 50007 [ACK] Seq=13 Ack=13 Win=65536 Len=0 TSval=919802755 TSecr=4252976948
0.000289663	8	127.0.0.1	127.0.1.1	TCP	68	35964 → 50007 [FIN, ACK] Seq=13 Ack=13 Win=65536 Len=0 TSval=919802755 TSecr=4252976948
0.000358987	9	127.0.1.1	127.0.0.1	TCP	68	50007 → 35964 [FIN, ACK] Seq=13 Ack=14 Win=65536 Len=0 TSval=4252976948 TSecr=919802755
0.000374648	10	127.0.0.1	127.0.1.1	TCP	68	35964 → 50007 [ACK] Seq=14 Ack=14 Win=65536 Len=0 TSval=919802755 TSecr=4252976948
650.667033500	11	127.0.0.1	127.0.1.1	TCP	76	47628 → 50007 [SYN] Seq=0 Win=65495 Len=0 MSS=65495 SACK_PERM=1 TSval=920453422 TSecr=0 WS=128
650.667051418	12	127.0.1.1	127.0.0.1	TCP	76	50007 → 47628 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM=1 TSval=4253627615 TSecr=920453422 WS=128
650.667062793	13	127.0.0.1	127.0.1.1	TCP	68	47628 → 50007 [ACK] Seq=1 Ack=1 Win=65536 Len=0 TSval=920453422 TSecr=4253627615
650.667087914	14	127.0.0.1	127.0.1.1	TCP	80	47628 → 50007 [PSH, ACK] Seq=1 Ack=1 Win=65536 Len=12 TSval=920453422 TSecr=4253627615
650.667092691	15	127.0.1.1	127.0.0.1	TCP	68	50007 → 47628 [ACK] Seq=1 Ack=13 Win=65536 Len=0 TSval=4253627615 TSecr=920453422
650.667105953	16	127.0.0.1	127.0.1.1	TCP	68	47628 → 50007 [FIN, ACK] Seq=13 Ack=1 Win=65536 Len=0 TSval=920453422 TSecr=4253627615
650.667268260	17	127.0.1.1	127.0.0.1	TCP	80	50007 → 47628 [PSH, ACK] Seq=1 Ack=14 Win=65536 Len=12 TSval=4253627615 TSecr=920453422
650.667283291	18	127.0.0.1	127.0.1.1	TCP	56	47628 → 50007 [RST] Seq=14 Win=0 Len=0

Lines 1-10: generated with the code of the previous slide

Lines 11-18: generated by removing the lines ‘`data = s.recv(1024)`’ and ‘`print('Received', repr(data))`’ from the client’s code of the previous slide