Telematic Applications

Universidad Carlos III de Madrid

Advanced Transmission Control Protocol

Aplicaciones Telemáticas (Telematic Applications)
Grado en Ingeniería Tecnologías de las Telecomunicaciones

Based on Celeste Campo and Calos García slides.
Modified by Daniel Díaz ,Andres Marín, Florina Almenarez.

Outlook

Outlook

1. Introduction
2. Connection establishment and termination
3. Interactive data flow
4. Bulk data flow
5. Timeout and retransmission
6. Persistent timer
7. Keepalive timer
8. Relation with lower layers

Bibliography

- Basic:
  - "TCP/IP Illustrated, Vol. 1 - The protocols", W. R. Stevens. Addison-Wesley 1994. (Chapter 17-24).
- Additional:
  - "Computer Networking: A Top-Down Approach Featuring the Internet", 3a Ed. J.F. Kurose and K. W. Ross. Addison-Wesley, 2005. (Chapter 3).
  - "Internetworking with TCP/IP Volume I. Principles, Protocols and Architecture", 5a Ed. D.E. Comer and D.L. Stevens, Prentice-Hall Int., 2006. (Chapter 12).
- RFCs:
  - RFC 793: Transmission Control Protocol. 1981.
  - RFC 2581: TCP Congestión Control. 1999.

Universidad Carlos III de Madrid

Introduction

1. Introduction
   - A. Definition
   - B. TCP segment format
   - C. TCP header fields

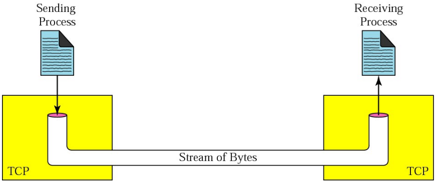
Lesson outlook

1. Introduction
2. Connection establishment and termination
3. Interactive data flow
4. Bulk data flow
5. Timeout and retransmission
6. Persistent timer
7. Keepalive timer
8. Relation with lower layers

Universidad Carlos III de Madrid

Introduction >> Definition

- TCP is a connection oriented protocol providing a reliable byte flow transport across applications:
  - **"Connection oriented"** ⇒ both parties have to set up a connection prior to data exchange
  - **"Reliable"** ⇒ TCP guarantees ordered delivery of the bytes exchanged between the communicating parties
    - Does IP can do that? Can UDP?
    - Why is that important?
  - **"Byte flow"** ⇒ through the connection a flow of bytes is transmitted
    - TCP organizes the byte flow in segments to deliver them to IP
    - No character oriented, works as a file
    - What happens with fractioning?



Introduction >> Definition

- Exercises
  - Mention protocols that are
    - Connection oriented
    - Datagram oriented
  - Which protocols are reliable and which not?

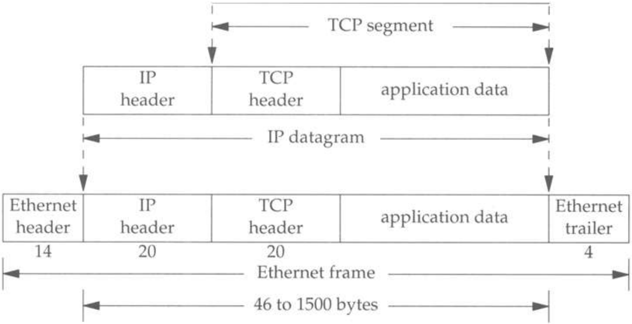
Universidad Carlos III de Madrid

Introduction >> Functionality

- TCP guarantees reliability over (unreliable) IP
  - error control using a checksum redundancy code over data and header
  - guarantees delivery of segments
    - when a segment is sent, a timer is scheduled (retransmission timer) waiting for segment reception acknowledgment
    - When a segment is received, an acknowledgment is returned
    - If the retransmission timer expires, the segment is retransmitted
  - Segments are reordered and duplicates discarded
- TCP provides
  - flow control
  - congestion control

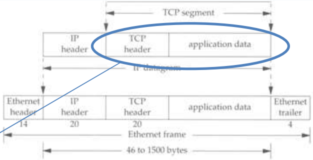
Introduction >> Segment format

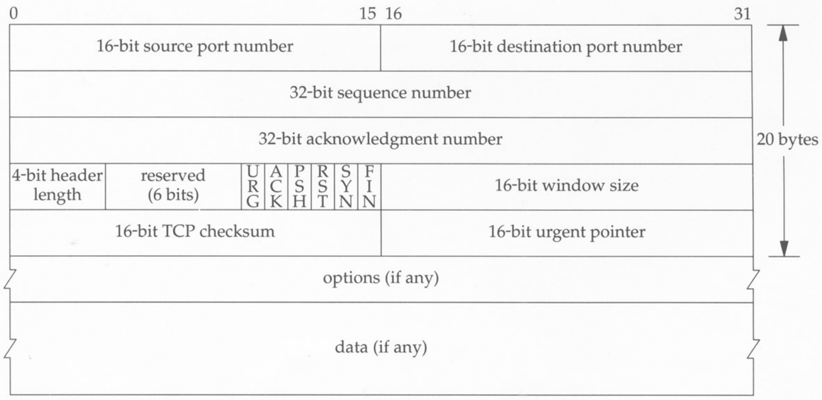
- TCP segments are encapsulated in IP datagrams (payload)



Introduction >> Segment format >> Details

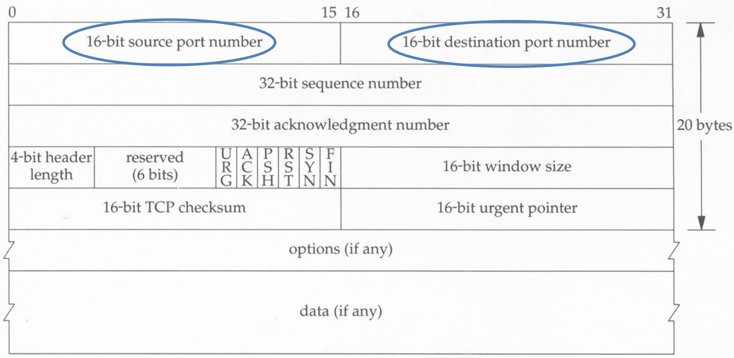
- TCP segments are
  - encapsulated in IP datagrams (payload)





Introduction >> Segment format >> Details >> Header fields

- Origin and destination ports: tracks the logic connection across the communicating applications (FTP, TELNET, SMTP, etc.)
  - Unique connection identifier:
    - Socket pair:
      - sockets (defined in RFC 793) = IP address + TCP port
    - (Source IP address, source port, destination IP address, destination port)

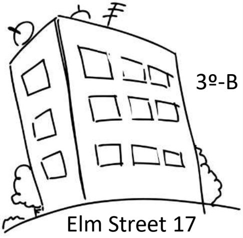


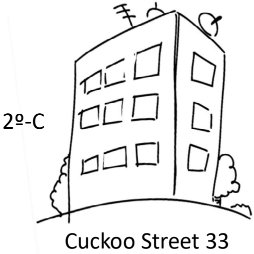
Introduction >> Segment format >> Details >> Header fields

- Exercises
  - How addressing happens in TCP/IP?
    - How many IP addresses can a host have?
    - What is the loopback interfaces (also known as localhost)?
  - How many ports can a host have?
  - How many services can use a port?

Introduction >> Segment format >> Details >> Header fields

- Exercises

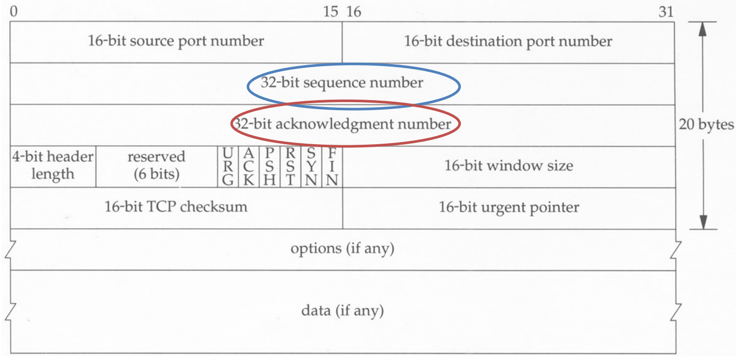




Using this methaphore… what protocols would handle in Internet the street and building number and with one the flat number?

Introduction >> Segment format >> Details >> Header fields

- Sequence number: position within the total byte flow of the first octet in the data field, i.e. every octet is numbered.
  - 32 bits field ⇒ range 0-232-1
  - if the SYN flag is set, it contains the initial sequence number (n) and the first data octet is n+1. The SYN segment consumes one sequence number.
  - "full-duplex" service model ⇒ each party maintains its own independent sequence number.
- ACK number: sequence number expected (next to the last received). Valid if ACK flag set

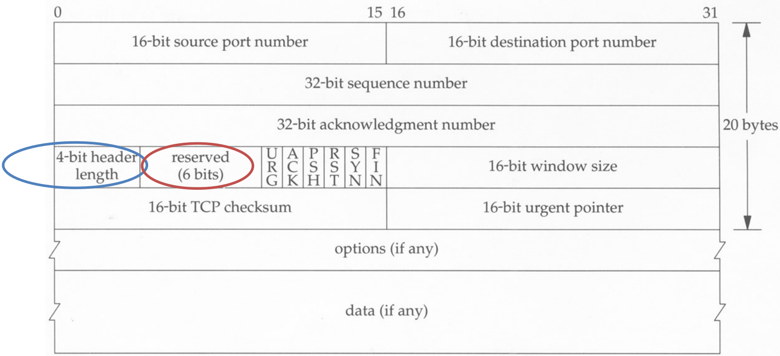


Introduction >> Segment format >> Details >> Header fields

- Sequence number are used to order packets and to detect looses and congestion.
  - Src -> Dst :: I send you packet 14 (n)
- ACK number: are used to tell the other side what packets have been correctly received
  - If all the packets until number 14 has been received…
  - Src <- Dst :: ACK 15 (n+1) (means ready to receive packet 15)

Introduction >> Segment format >> Details >> Header fields

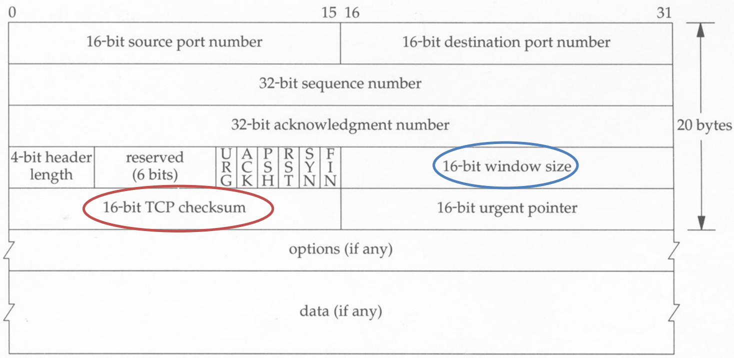
- Header length: number of 32 bit words of the TCP header
  - Needed because the header length is variable (options).
  - 4 bits field ⇒ TCP header limited to 60 octets.
  - Default value 20 (no options)
- Reserved: reserved for future use. Should be at 0 (unset).



Universidad Carlos III de Madrid

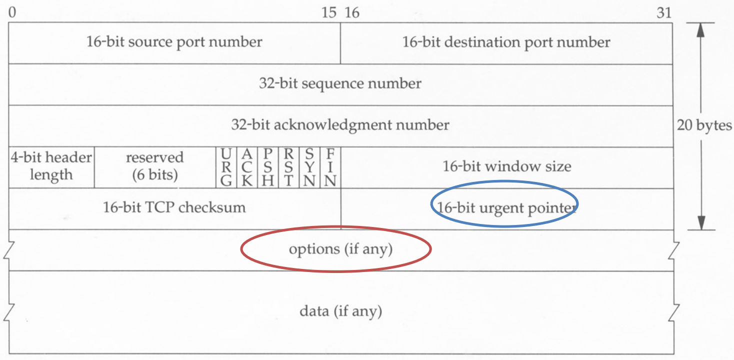
Introduction >> Segment format >> Details >> Header fields

- Window size: number of octets (besides the acknowledged) this party is able to buffer
  - Used for TCP flow control mechanism
  - Each party announces its own buffersize (we can limit the amount of information to receive)
  - 16 bits field ⇒ limited to 65535 octets (but window-scale option)
- Checksum: redundancy code computed by the origin and verified by the destination
  - Includes the whole segment (TCP header + data)

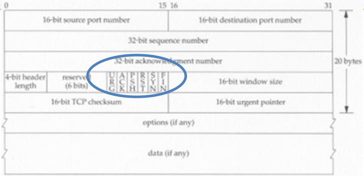


Introduction >> Segment format >> Details >> Header fields

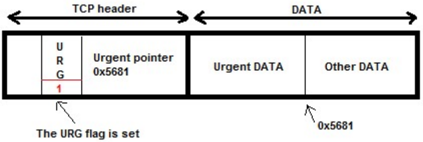
- Urgent data pointer: positive offset to the end of the urgent data (sequence number + offset = end urgent data)
  - Only valid if URG flag is set
- Options (will see few later)



Introduction >> Segment format >> Details >> Header fields >> Flags

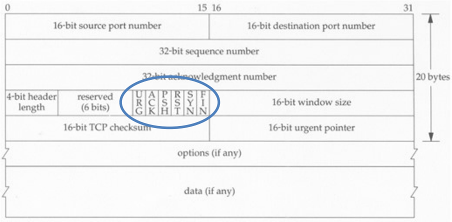


- Flags: are bits that can be set or not
  - There are many, used for controlling TCP
  - URG: the urgent data pointer has a valid value
    - We can deliver data into data field and place inside important or urgent data that should be processed as soon as possible
    - If URG is set, we will find the start of urgent data within the data field in the place pointed by the 16-bit urgent data pointer field



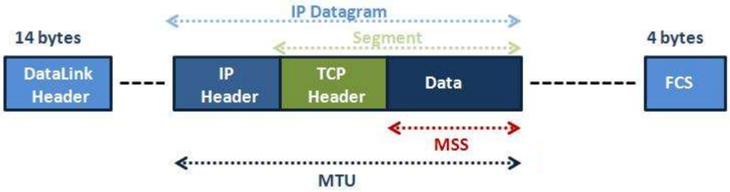
Introduction >> Segment format >> Details >> Header fields >> Flags

- Flags: are bits that can be set or not
  - ACK: the ack number field has a valid value
    - The receiver tells the other endpoint it has received up to some data sequence
  - PSH: the segment requires immediate send and delivery
    - Usually on when buffer is empty
  - RST: connection abort
    - After a problem (will see examples later)
  - SYN: connection establishment
    - To stablish connection and SYNchronize the sequence numbers to be used
    - IS confirmed with ACK
  - FIN: connection termination
    - IS confirmed with ACK



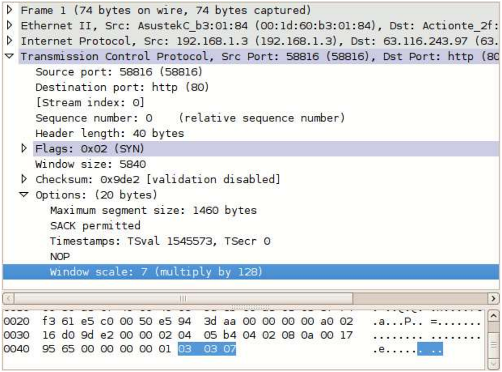
Introduction >> Segment format >> Details >> Options

- Options should fit in K * 4-byte sizes
- Noop or Nop: for padding
  - Means nothing, it is used to separate option fields (1 byte)
- Maximum segment size (MSS)
  - Used in connection establishment (only – cannot be changed after that)
  - Indicates maximum segment size one party can receive without fragmentation (usually computed in a local link MTU basis)



Introduction >> Segment format >> Details >> Options

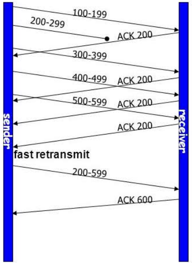
- Window scaling
  - Expands the size of receiving window
  - Remember window field in header is just 16bit, if you want bigger windows, you need to use this…
  - Allows larger windows sizes for connection with high RTTs, or very fast connections (delay x bandwidth)



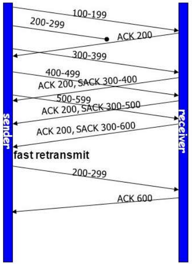
Introduction >> Segment format >> Details >> Options

- Timestamp
  - to accurately set the timer threshold value for a virtual circuit, it has to measure the round-trip delivery times for various segments. Finally, it has to monitor additional segments throughout the connection's lifetime to keep up with the changes in the network
- Selective Acknowledgements (SACK)
  - Plain ACKs allows for expressing "still waiting for segment 20"
  - Especially good for wireless networks that reports packet losses faster
    - The receiver can report sooner the loss
  - Unless instructed otherwise, we don't use this in calculations for problems

TCP without SACK



TCP with SACK



Universidad Carlos III de Madrid

Conn. establishment and termination

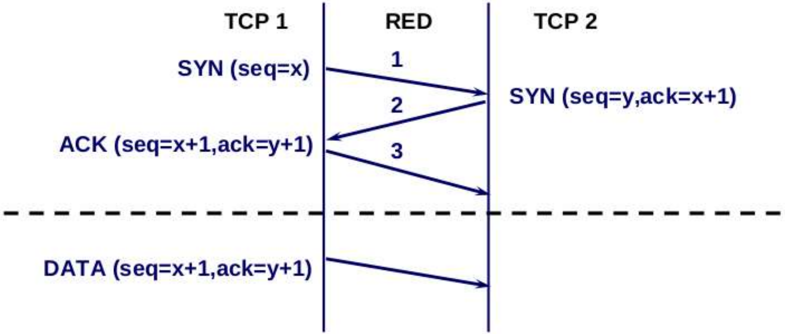
2. Connection establishment and termination
   A. Connection establishment ("three-way handshake").
   B. Connection termination
   C. TCP Half-close.
   D. MSS - Maximum Segment Size
   E. TCP states
   F. Simultaneous open
   G. Simultaneous close

Lesson outlook

1. Introduction
2. Connection establishment and termination
3. Interactive data flow
4. Bulk data flow
5. Timeout and retransmission
6. Persistent timer
7. Keepalive timer
8. Relation with lower layers

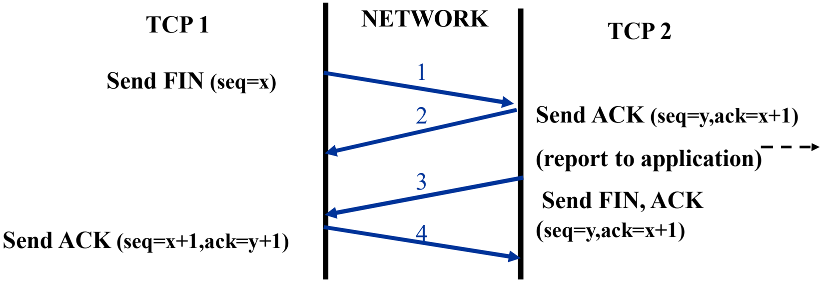
Conn. est. and ter. >> Conn. Establishment

- "Three way handshake".
  - Connection request: send a segment with SYN flag set
  - Wait for SYN+ACK response segment
  - Send confirming ACK



Conn. est. and ter. >> Connection close

- Connection is terminated independently by each party
  - A party willing to stop sending data send a segment with FIN flag set
  - The other party sends confirming ACK
- Connection is closed when both parties have received the ACK



Universidad Carlos III de Madrid

Conn. est. and ter. >> example (I)

- Take a linux terminal and use telnet to make a TCP connection:
  - If the telnet port (23) is not used, telnet command just opens a TCP connection
  - From machine called (host name) srv4 to machine called bsdi
  - Using port “discard” (well known ports can be named according to file /etc/services)
    - Discards service does nothing with the data (but TCP signaling flows)

```
tcpmux          1/tcp                           # TCP port service multiplexer
echo            7/tcp
echo            7/udp
discard         9/tcp           sink null
discard         9/udp           sink null
systat          11/tcp          users
daytime         13/tcp
```

- Type the following

```
srv4 % telnet bsdi discard
Trying 192.82.148.3...
Connected to it011.lab.it.uc3m.es.
Escape character is '^]'.
^]

telnet> quit
Connection closed
```

Conn. est. and ter. >> example (II)

- To observe the traffic will use tcpdump command that has the following notation

```
Timestamp          src > dst: flags first:last(nbytes) ack window urgent options
```

where flags are abbreviated this way:

Flag	Abbr.	Description
S	SYN	Synchronices sequence numbers
F	FIN	Sender has finished sending data
R	RST	Connection abort
P	PUSH	Segment requires immediate send and delivery
.		None of the above flags active

- Running the command and capturing traffic we see the establishment:
  - Destination port is “discard” (number 9) at bsdi (server) and origin port is random (1037) at srv4 (client)
  - Client (srv4) sends SYN and synchronizes sequence number (note data bytes are 0)
  - Server (bsdi) sends a SYN with its sequence numbers (no data)
  - Client sends an ACK

```
1  0.0                    svr4.1037 > bsdi.discard:  S 1415531521:1415531521(0)
                                                       win 4096 <mss 1024>

2  0.002402 (0.0024)      bsdi.discard > svr4.1037:  S 1823083521:1823083521(0)
                                                       ack 1415531522 win 4096
                                                       <mss 1024>

3  0.007224 (0.0048)      svr4.1037 > bsdi.discard:  . ack 1823083522 win 4096
```

Conn. est. and ter. >> example (III)

- Typing the escape character “^]” connection get closed and we can see closure in TCPDUMP:
  - Srv4 (client) sends FIN, bsdi (server) sends ACK
  - Then bsdi (server) sends FIN, Srv4 (client) sends ACK

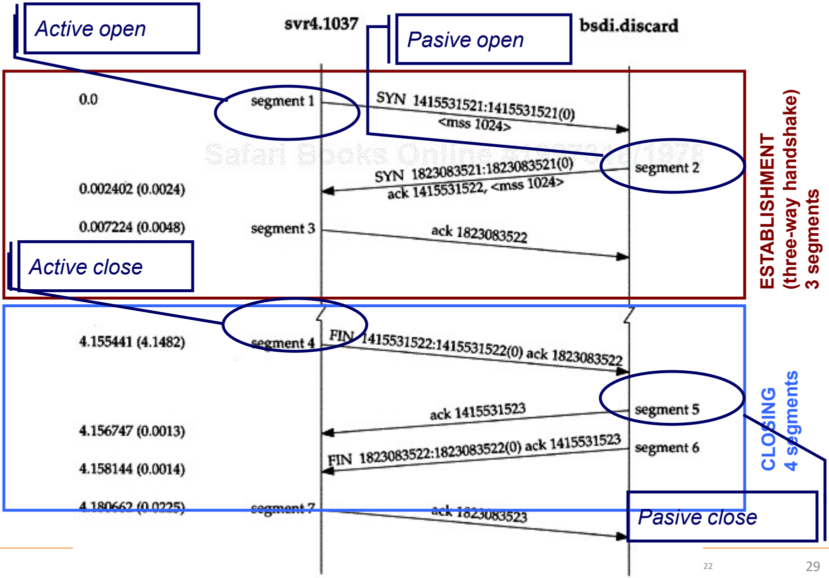
```
4  4.155441 (4.1482)   svr4.1037 > bsdi.discard:   F 1415531522:1415531522(0)
                                                     ack 1823083522 win 4096

5  4.156747 (0.0013)   bsdi.discard > svr4.1037:   . ack 1415531523 win 4096

6  4.158144 (0.0014)   bsdi.discard > svr4.1037:   F 1823083522:1823083522(0)
                                                     ack 1415531523 win 4096
7  4.180662 (0.0225)   svr4.1037 > bsdi.discard:   . ack 1823083523 win 4096
```

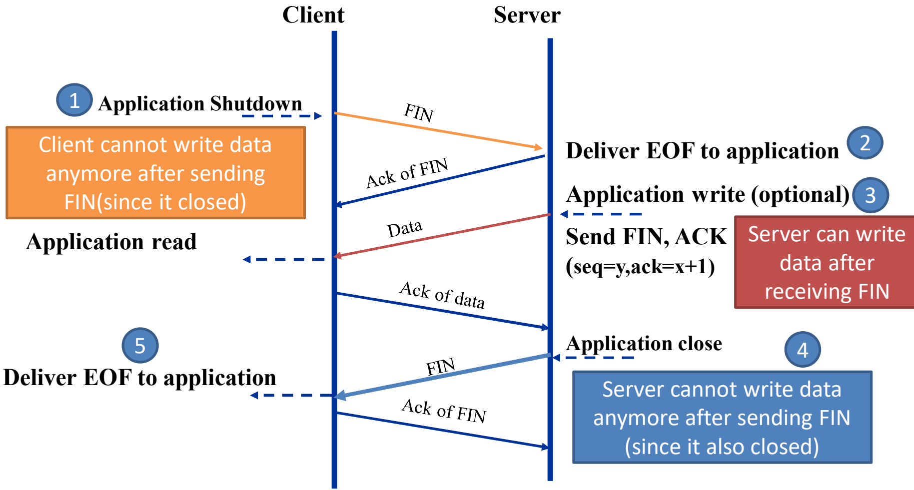
Conn. est. and ter. >> example (IV)

- Timeline view



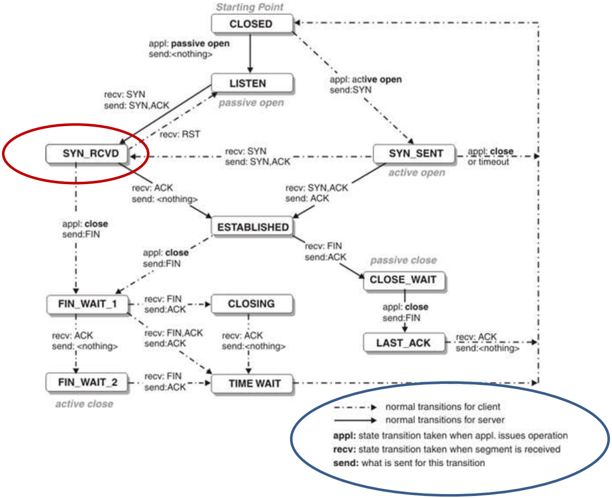
Conn. est. and ter. >> Especial Conn. Close >> Half close

- Full-duplex allows each party to send FIN (and finish sending data) independently



Conn. est. and ter. >> TCP States

- As TCP is a connection-oriented protocol
  - should keep the state of the connection with this diagram (states/transitions)



Universidad Carlos III de Madrid

Conn. est. and ter. >> TCP States >> Establishment (I)

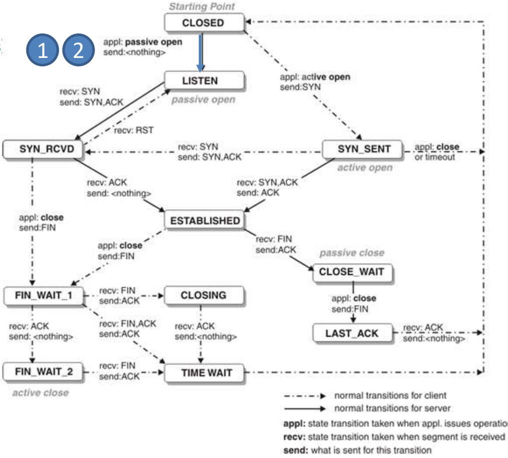
- Connection establishment
- The server has a "passive socket" (ps)
  - Cannot read or write (accepts new connections
  - Every new connection creates an active socket (as
    - Active sockets can read and write
    - But cannot accept connections

```
//pseudocode at server
ps = socket(information);   1
bind(ps);  //reserves port
// start listening for incoming conns.
listen(ps);  2

//since it should support multiple
clients

for(;;){
   as = accept(ps);
   //new active socket (as) at server
   //represents a client

}
```

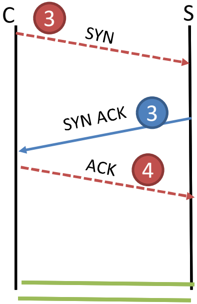


Universidad Carlos III de Madrid

Conn. est. and ter. >> TCP States >> Establishment (II)

- A client creates a connection
- A client has an "active socket"
  - Can read or write
  - But cannot accept connections

```
//pseudocode at client
as = socket(information);
connect(as);  //connect
```



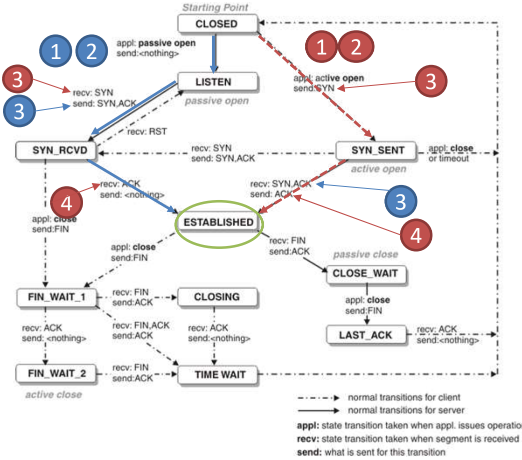
Client sends SYN
Client moves to SYN_SENT

Server receives SYN,
And sends SYN ACK
Server moves to SYN_RCVD

Client receives SYN ACK,
and sends ACK
Client moves to ESTABLISHED

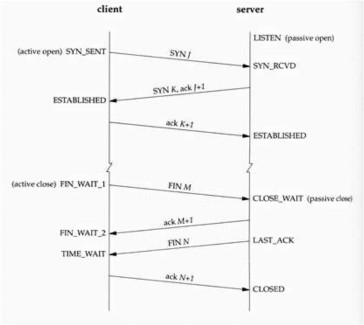
Server receives ACK,
and sends nothing
Server moves to ESTABLISHED

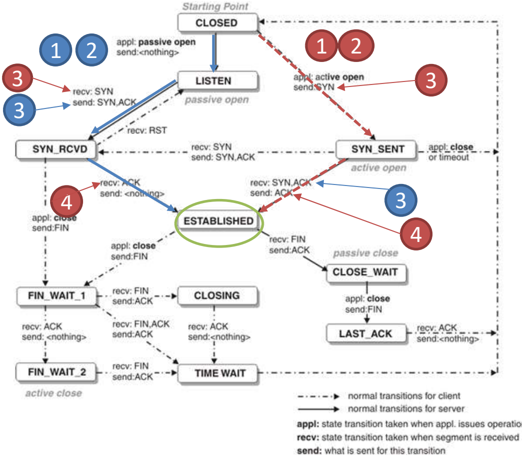
Connection established



Conn. est. and ter. >> TCP States >> Establishment (III)

- So, the server creates a passive socket
- The client creates an active socket
- When the client connects
  - Active open
- Server creates an active socket
  - One per client

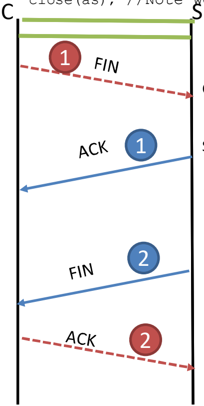




Conn. est. and ter. >> TCP States >> Con. Close (I)

- When connection is stablished
  - Can read/write
- Client closes the connection

```
//pseudocode at client
close(as); //Note we close the active…
```



Connection established

Client sends FIN
Client moves to FIN_WAIT_1

Server receives FIN and sends ACK
Server moves to CLOSE_WAIT

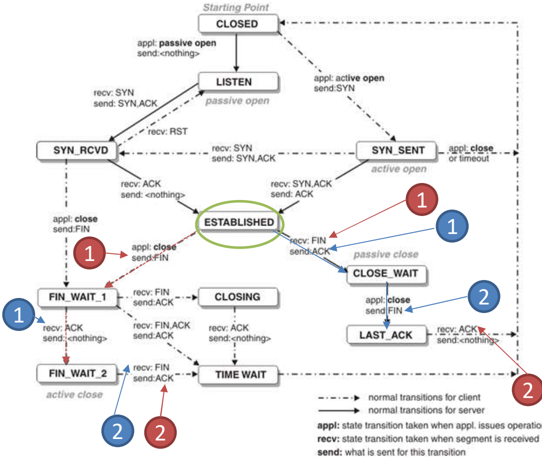
Client receives ACK,
Client moves to FIN_WAIT_2

Server application closes

Server sends FIN
Server moves to LAST_ACK

Client receives FIN and sends ACK
Client moves to TIME_WAIT

Server receives ACK
Server moves to CLOSED

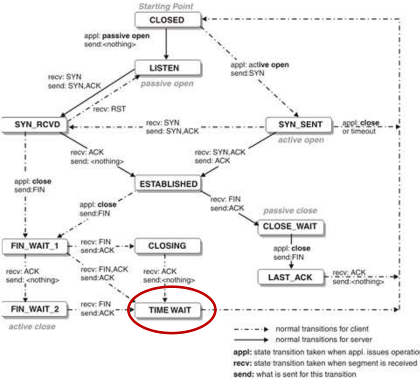


Universidad Carlos III de Madrid

Conn. est. and ter. >> TCP States >> Con. Close (II)

- Server goes to CLOSED
- But client stays in TIME_WAIT
  - How long should the client wait in TIME_WAIT?
  - 2MSL wait state = TIME_WAIT

- TCP has to wait for t=TIME_WAIT after performing "active" close and sending the ACK
- Reasons:
  - Allows TCP retransmit the ACK of the FIN (if a retransmitted FIN is received)
  - The socket pair is not reusable until 2MSL timer expires
- MSL (Maximum Segment Lifetime) value is left to implementors choice:
  - typically 2 minutes || 1 minute || 30 seconds

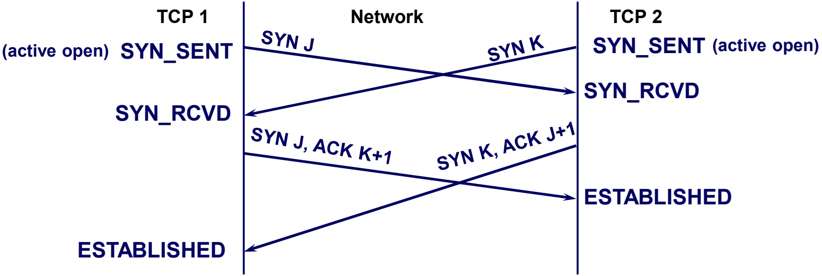


Conn. est. and ter. >> TCP States >> Connection Reset

- Indicated setting flag RST
- Typically generated by TCP stack (kernel)
- When is it set?
  - **Connection request incoming to a destination port not assigned to any application** (nobody listening to destination port)
  - Incoming TCP segment **not** corresponding to any **active connection**
  - **Aborting a connection at application level** (instead of sending FIN, a RST is sent):
    - Receiver discards EVERY byte of data pending of transmission
    - RST segment is not acknowdleged
    - Connection is thus terminated

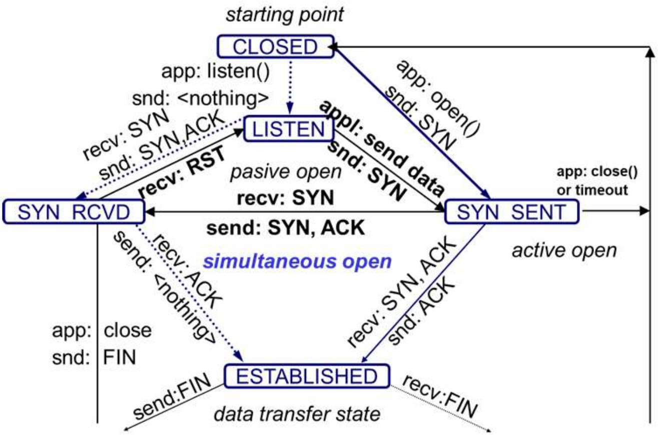
Conn. est. and ter. >> TCP States >> Simultaneous open

- Both parties perform an active open
- TCP is designed to ensure a single connection
  - other protocols create two connections
- Connection establishment requires 4 segments not 3



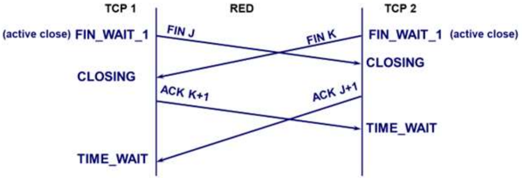
Conn. est. and ter. >> TCP States >> Simultaneous open

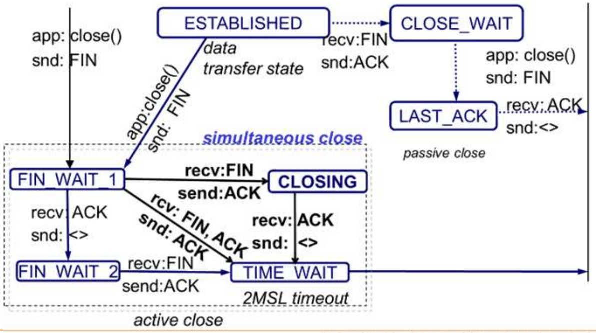
- Simultaneous open is supported by TCP states



Conn. est. and ter. >> TCP States >> Simultaneous close

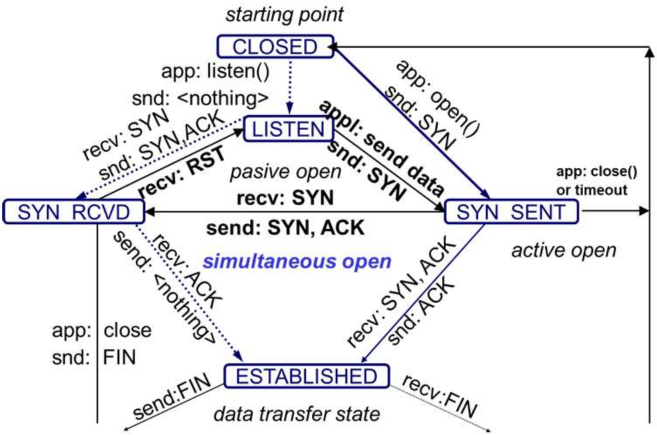
- Both parties perform a close…





Conn. est. and ter. >> TCP States >> Simultaneous close

- Simultaneous open is supported by TCP states



Universidad Carlos III de Madrid

Conn. est. and ter. >> Exercise

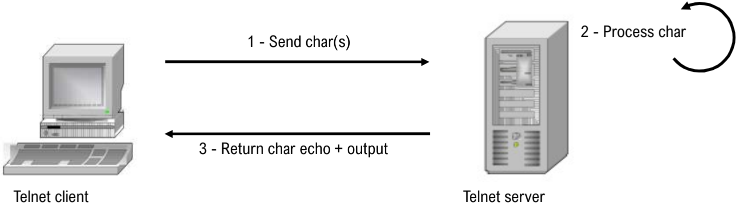
- Exercises
  - Regarding sockets API (you need lab), connection establishment and closing… please research and give an answer to the following…
    - What is a passive socket?
    - What is an active socket?
    - What is active open and passive open?
    - If we have a server with a single client connected to it… how many passive and active sockets are working? Where are those sockets (client app / server app)?

Interactive data flow

3. Interactive data flow
   A. Interactive input
   B. Delayed acknowledgements
   C. Nagle's algorithm

Interactive data flow >> Introduction

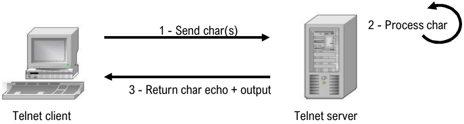
- Example of interactive input



- Remote terminal application send user typed chars to server. Server process char and returns echo and the response of the command
- Every char may require transmission of 3 TCP segments:
  - (1) Client ⇒ Server: char
  - (2) Server ⇒ Client: char echo + ack of (1)
  - (3) Client ⇒ Server: ack of (2)

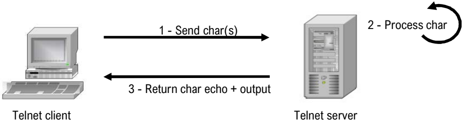
Interactive data flow >> Delayed ACKs

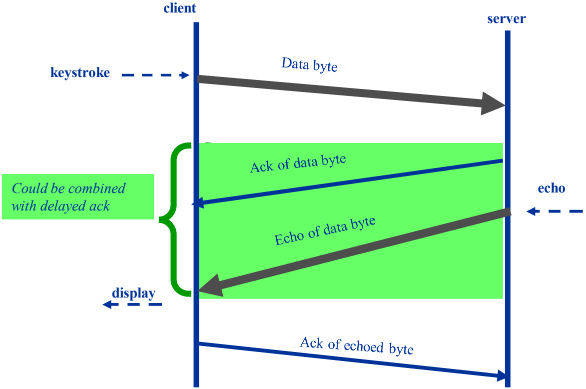
- ACKs are not usually sent immediately after receiving data.
  - In problems we can send them immediately
- An ACK can be delayed:
  - Until there is data to send to the other end
    - The ACK is included in the data segment (ACK **piggyback** with the data).
  - Up to a maximum of 200 ms. (usual value in deployments):
    - RFC indicates that a delay should be implemented in the sending of ACKs and that this delay should be less than 500 ms.



Interactive data flow >> Delayed ACKs

- Example of delayed ACKs





Interactive data flow >> Delayed ACKs >> Example

- 0.0 bsdi.1023 > svr4.login: P 0:1(1) ack 1 win 4096
- 0.016497 (0.0165) svr4.login > bsdi.1023: P 1:2(1) ack 1 win 4096

3 0.139955 (0.1235) bsdi.1023 > svr4.login: . ack 2 win 4096

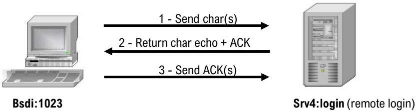
1

2. ECHO

2. “ACK” of 1

3. “ACK” of 2

```
bsdi:~> rlogin svr4
password: XXXX
date\n
Sat Feb 6 07:52:17 MST 1993
```



Universidad Carlos III de Madrid

Interactive data flow >> Delayed ACKs >> Example

d	1	0.0		bsdi.1023 > svr4.login:	P	0:1(1) ack 1 win 4096
	2	0.016497	(0.0165)	svr4.login > bsdi.1023:	P	1:2(1) ack 1 win 4096
	3	0.139955	(0.1235)	bsdi.1023 > svr4.login:	.	ack 2 win 4096
a	4	0.458037	(0.3181)	bsdi.1023 > svr4.login:	P	1:2(1) ack 2 win 4096
	5	0.474386	(0.0163)	svr4.login > bsdi.1023:	P	2:3(1) ack 2 win 4096
	6	0.539943	(0.0656)	bsdi.1023 > svr4.login:	.	ack 3 win 4096
t	7	0.814582	(0.2746)	bsdi.1023 > svr4.login:	P	2:3(1) ack 3 win 4096
	8	0.831108	(0.0165)	svr4.login > bsdi.1023:	P	3:4(1) ack 3 win 4096
	9	0.940112	(0.1090)	bsdi.1023 > svr4.login:	.	ack 4 win 4096
e	10	1.191287	(0.2512)	bsdi.1023 > svr4.login:	P	3:4(1) ack 4 win 4096
	11	1.207701	(0.0164)	svr4.login > bsdi.1023:	P	4:5(1) ack 4 win 4096
	12	1.339994	(0.1323)	bsdi.1023 > svr4.login:	.	ack 5 win 4096
\n	13	1.680646	(0.3407)	bsdi.1023 > svr4.login:	P	4:5(1) ack 5 win 4096
(Echo:CR/LF)	14	1.697977	(0.0173)	svr4.login > bsdi.1023:	P	5:7(**2**) ack 5 win 4096
	15	1.739974	(0.0420)	bsdi.1023 > svr4.login:	.	ack 7 win 4096

Interactive data flow >> Nagle Algorithm

- Nagle algorithm (RFC 896) proposes a solution to the following problem:
  - In some cases, few data octets are sent in each segment (tinygrams).
    - That increases overhead, due to headers.
  - It has little effect in LANs
    - but may contribute to WAN congestion
  - In the previous example, 41 octets (for sending 1 byte):
    - IP header (20 octets)
    - TCP header (20 octets)
    - Data (1 octet)

Interactive data flow >> Nagle Algorithm >> Definition

- A TCP connection can have only one outstanding small segment that has not yet been acknowledged
  - Just one tinygram pending of ACK
  - No additional small segments can be sent until ack is received

- Instead, small amounts of data are collected by TCP and sent in a single segment when ack is received
  - Keystrokes are saved for later

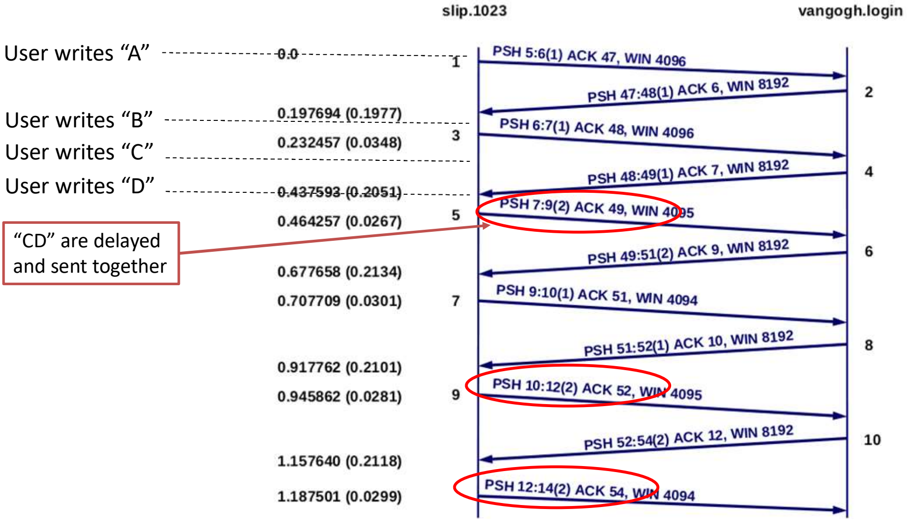
- Self-clocking property:
  - The faster ACKs come back, the faster data is sent
  - On a slow WAN, fewer tinygrams are sent
    - Sending two chars together is a 3x improvement!

Interactive data flow >> Nagle Algorithm >> Disable Nagle

- There are times when the Nagle algorithm needs to be turned off:
  - Window system server (remote desktop apps)
    - Mouse movements must be delivered without delay
  - Sending function keys (composed of several chars)
    - Example Ctrl-C
    - If these chars are not sent together the server is not able to generate echo,
    - so that ack is delayed for 200 ms,
    - and the interactive user experiences suffer noticeable delays

- TCP Implementations allow to disable this algorithm

Interactive data flow >> Nagle Algorithm >> Example



Universidad Carlos III de Madrid

Bulk data flow

3. Bulk data flow:
   - A. Normal data flow
   - B. Flow control mechanism
     - TCP sliding window
   - C. congestion control mechanism
     - TCP "Slow start"
   - D. PUSH flag
   - E. URG flag

Lesson outlook

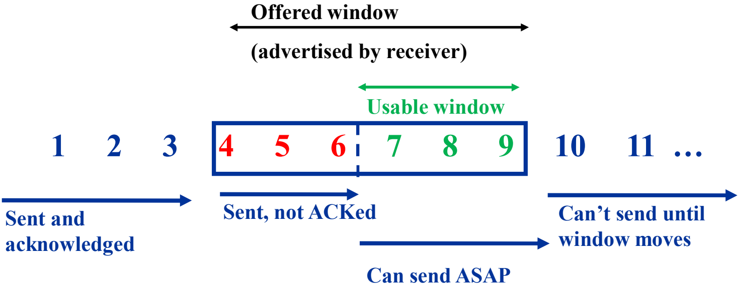
Bulk data flow >> Normal Data flow

- **Idea**: allow sending multiple segments before stop and wait for acknowledgements
  - Faster data transfers
- **Flow control** mechanisms prevent receiving nodes being overwhelmed with data from sending nodes:
  - TCP uses a sliding window for flow control: the receiving window
    - It allows sender to transmit several segments before stop and wait for ack without overwhelming the receiver
- **Congestion control** **prevents** sender to worsen a congested network:
  - TCP "slow-start":
    - Allows sending data at the rate acks are sent from the other end
  - More mechanisms with different timeouts (congestion avoidance, FR/FR…)

Universidad Carlos III de Madrid

Bulk data flow >> Flow Control >> TCP Sliding window

- Allows flow control, characteristics:
  - ACK sending is independent of receiver window size variation (win is only affected by segments containing)
  - ACKs do not automatically increase window size (win is only affected by data)
  - ACK are cumulative



Universidad Carlos III de Madrid

Bulk data flow >> Flow Control >> TCP Sliding window >> Example

- Sliding window example

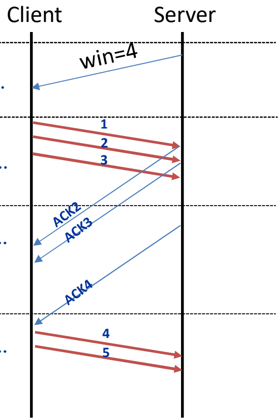
Segments to be delivered 1 2 3 4 5 6 7 8 9 10 11 …

win is set to 4 [1 2 3 4] 5 6 7 8 9 10 11 …

client delivers 3 segs.
1,2 and 3 wait for ACK
4 can be delivered
[1 2 3 ¦ 4] 5 6 7 8 9 10 11 …

client receives ack 2 and 3
1 and 2 are finished
3 pending; 4,5 and 6 can be delivered
1 2 [3 ¦ 4 5 6] 7 8 9 10 11 …

client sends 4 and 5 and receives ACK 4
3 is finished
4 and 5 are pending
6 and 7 can be delivered
1 2 3 [4 5 ¦ 6 7] 8 9 10 11 …



Bulk data flow >> Flow Control >> TCP Sliding window

- The window **closes** (left wall moves right)
  - Receiver advertised **win=6**. Segs 4, 5 and 6 are pending of ACK

    1 2 3 | 4 5 6 ¦ 7 8 9 | 10 11 …

  - Then **sender sends seg 7** (still no ACKs, so window starts closing…)

    1 2 3 | 4 5 6 7 ¦ 8 9 | 10 11 …

  - The situation go worse… one ACK is **received** (**ack 6**) but **window reduced** by receiver (**win=4**)

    1 2 3 4 5 | 6 7 ¦ 8 9 | 10 11 …

  - The **sender sends segs 8 and 9** so it cannot send anymore until ACKs arrive

    1 2 3 4 5 | 6 7 8 9 | 10 11 …

Bulk data flow >> Flow Control >> TCP Sliding window

- The window **opens** (right wall moves right)
  - Receiver advertised **win=6**. Segs 4, 5 and 6 are pending of ACK

1 2 3 4 5 6 7 8 9 10 11 …

  - Then **ack 6 received** (more segments can be sent)

1 2 3 4 5 6 7 8 9 10 11 12 13…

  - Then ack 7 **received**

1 2 3 4 5 6 7 8 9 10 11 12 13…

Bulk data flow >> Flow Control >> TCP Sliding window

- The window can also **shrink** (right wall moves left)
  - The RFC strongly recommends not to shrink the window
- Receiver advertised **win=6**. Segs 4, 5 and 6 are pending of ACK

1 2 3 4 5 6 7 8 9 10 11 …

  - Then **ack 5 received** but win changed to win=3

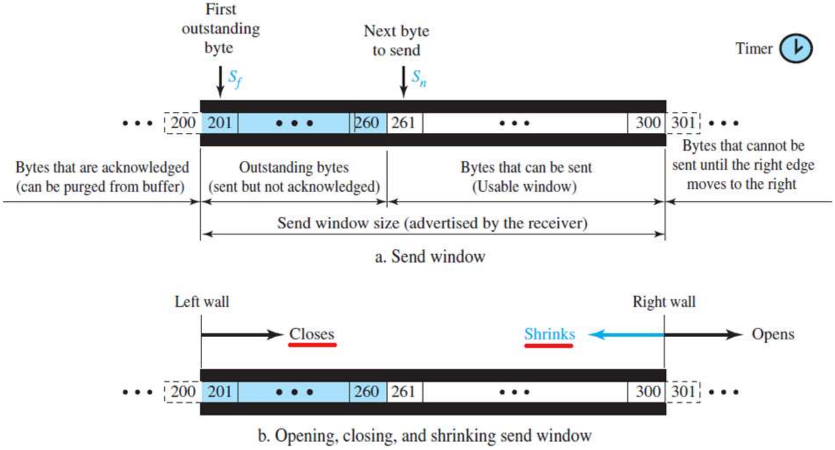
1 2 3 4 5 6 7 8 9 10 11 …

  - Despite the receiver can shrink the window, it is not recomended

Universidad Carlos III de Madrid

Bulk data flow >> Flow Control >> TCP Sliding window

- Window opens, closes and shrinks… moving walls



Images taken from Foruzan's "Data Communications and Networking | 5th Edition", 978-1259064753. Images belongs to their authors.

Bulk data flow >> Flow Control >> TCP Sliding win. >> Management

- Receiver sends back to sender two parameters:

ACK	WIN
32 bits	**16 bits**

- Interpreted as:
  - "I am ready to receive new data, from seq=ack to ack+win-1"

- Receiver may acknowledge data without opening its window
- Receiver may change its window size without acknowledging new data

.

Bulk data flow >> Flow Control >> TCP Sliding win. >> Example

- In the following example we send 8192 bytes (1024*8) from **srv4** (client) to **bsdi** (server) using the command "sock"
  - Server (bsdi in this case) sinks data from socket (-i), operate as server (-s) use port 7777 to listen for incoming connections
  - Client (srv4 in this case) source data to socket (-i), write 8 buffers of 1024 to the other endpoint (-n8), connect to bsdi on port 7777

On server side type

```
bsdi:~> sock -i -s 7777
```

On client side type

```
Srv4:~> sock -i -n8 bsdi 7777
```

Sock command may be downloaded from http://ttcplinux.sourceforge.net/tools/tools.html. To use it un the labs, please type:

wget http://ttcplinux.sourceforge.net/tools/sock

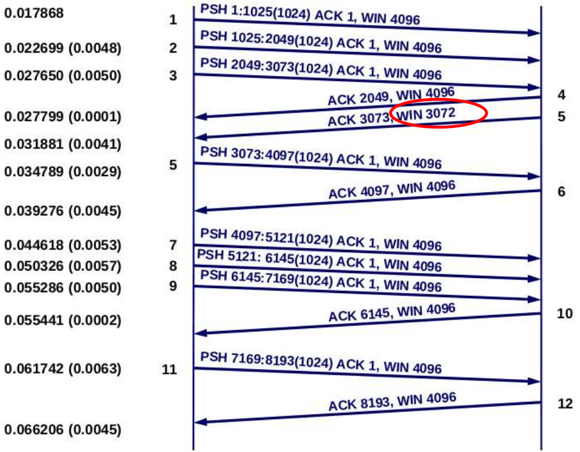
chmod u+x sock

Then you can use it from two different machines as ./sock <options>

Universidad Carlos III de Madrid

Bulk data flow >> Flow Control >> TCP Sliding win. >> Example

- At given time receiver advertises a smaller window



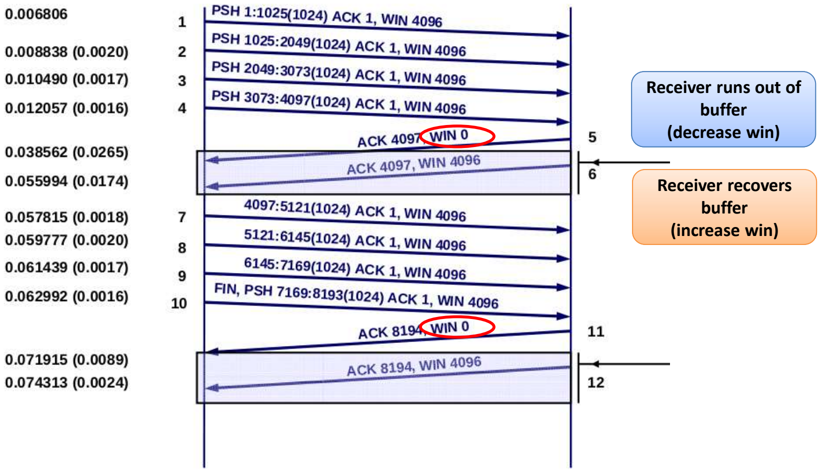
Use tcpdump to capture traffic (etho can be changed to the appropriate interface):

tcpdump -i etho port 7777

Universidad Carlos III de Madrid

Bulk data flow >> Flow Control >> TCP Sliding win. >> Example (II)

- An example of fast sender, slow receiver…



Bulk data flow >> Congestion Control

- TCP has several congestion control mechanisms
  - **Slow Start**
  - Congestion avoidance
  - Fast Recovery/Fast retransmit
- Now we will study Slow Start, later will introduce others in Timeouts and retransmission section

.

Bulk data flow >> Congestion Control >> Slow Start

- Algorithm for congestion control:
  - Easy selection of connection parameters in LANs
  - Difficulties in WANs as router queues get full
- Slow start algorithm:
  - **Rate of packet sending depends on rate of receiving acks**
  - A new window is introduced in the sender:
    - The **congestion window (cwnd)**
- Should be **applied in combination with Flow Control**
  - The maximum amount to be sent will be min(cwnd,win)

.

Bulk data flow >> Congestion Control >> Slow Start >> Definition

- At connection establishment:
  - Set cwnd to 1 segment
    - cwnd= 1 segment
    - cwnd= 1xMSS bytes
- Every time an ACK is received:
  - Increase cwnd in one segment:
    - cwnd+= 1 segment
    - cwnd+= 1xMSS bytes
  - **A received ACK always increases one segment** regardless the ACK acknowledges more or less than one segment
- Sender may send as much data as the min(cwnd, win)
- cwnd normally **grows exponentially** if every segment is acknowledged independently
  - Linear if cumulative acks are sent

.

Universidad Carlos III de Madrid

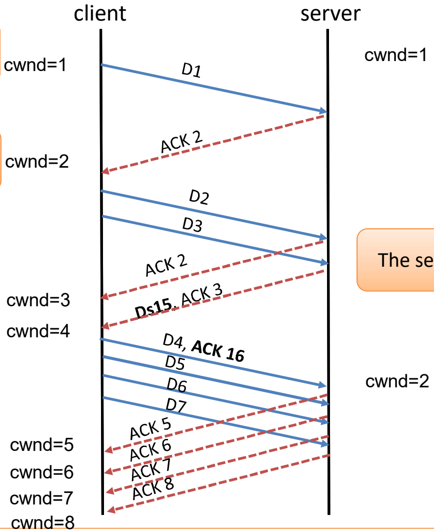
Bulk data flow >> Congestion Control >> Slow Start >> Definition

- Exponential grow (every segment is individually acknowledged)

Starts with cwnd = 1

ACK arrives, cwnd increaseses by one

Exponential grow



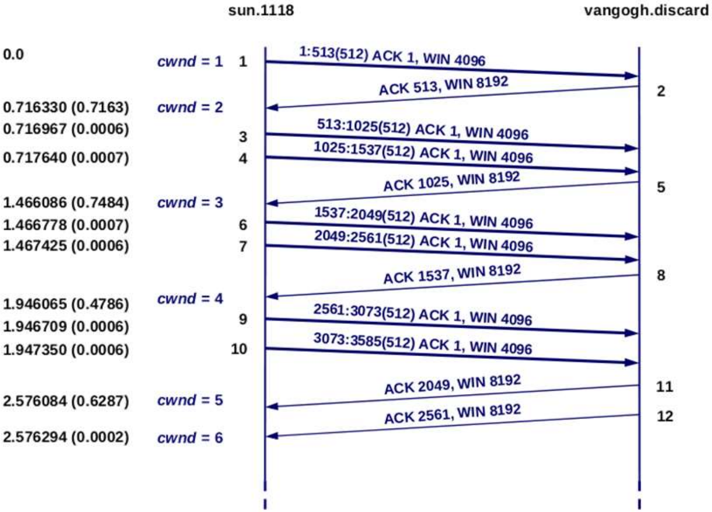
Starts with cwnd = 1 (server has its own cwnd)

The server also sends data Ds15

ACK arrives, cwnd increaseses by one

Bulk data flow >> Congestion Control >> Slow Start >> Definition

- Exponential grow (every segment is individually acknowledged)



Universidad Carlos III de Madrid

Bulk data flow >> Other aspects >> PUSH Flag

- The PUSH flag was defined as set when (PSH) **the segment requires immediate send and delivery**
  - We deferred the discussion
- It is a **notification from SENDER for the receiver** to immediately pass received data to receiving application
  - Applies to every byte contained in the segment with PUSH flag set
  - Previous data stored by TCP stack (kernel) should be also passed to the application
- Initially, applications indicated TCP to set the PUSH flag to immediately send data
  - **Most implementations APIs do NOT support this**
  - Actual implementations (Berkeley and derived):
    - Set the PUSH flag if the segment empties the transmission buffer
      - Could see this during lab sessions
    - Always deliver data to application immediately, thus ignoring the PUSH flag

.

Bulk data flow >> Other aspects >> URG Flag

- Sender notifies about urgent data in the byte stream:
  - Sets the URG flag
  - Value of urgent data pointer
    - Indicates where the end of urgent data is within the segment
    - It is a positive offset to the beginning of the segment (RFC 1122)
    - In several implementations it wrongly points to the byte following urgent data
- Receiver shall indicate urgent data arrival to receiving application
- Why is it used?
  - It allows sending a segment even if the receiver window is 0
- When is it used?
  - Control key for interrupting remote connection (rlogin, telnet), ftp transfer, etc.

.

Bulk data flow >> Other aspects >> URG Flag >> example

- sock program on server bsdi and **have it pause for 10 ms** after first read

  bsdi % sock -i -s -P10 5555

- Then start the client on sun telling it to use a send buffer of 8192 bytes (-S option) and perform **six** 1024-byte writes to the network (-n option).
  - We also specify to write **1 byte of urgent data** (-U5) before writing the fifth buffer to the network. We specify the verbose flag to see the order of the writes. It actually "enters urgent mode" so will send it many times until received (remember the server will sleep for 10ms)

```
sun % sock -v -i -n6 -S8192 -U5 bsdi 5555
connected on 140.252.13.33.1305 to 140.252.13.35.5555
SO_SNDBUF = 8192
TCP_MAXSEG = 1024
wrote 1024 bytes
wrote 1024 bytes
wrote 1024 bytes
wrote 1024 bytes
wrote 1 byte of urgent data
…
```

Bulk data flow >> Other aspects >> URG Flag >> example (I)

- Observe traffic with tcpdump

Client writes 4 times (out of 6) 1024 and TCP delivers that data

```
1    0.0                    sun.1305 > bsdi.5555:    P 1:1025(1024) ack 1 win 4096
2    0.073743 (0.0737)      sun.1305 > bsdi.5555:  P 1025:2049(1024) ack 1 win 4096
3    0.096969 (0.0232)      sun.1305 > bsdi.5555: P 2049:3073(1024) ack 1 win 4096
4    0.157514 (0.0605)      bsdi.5555 > sun.1305:  . ack 3073 win 1024
5    0.164267 (0.0068)      sun.1305 > bsdi.5555:  P 3073:4097(1024) ack 1 win 4096
```

Client writes 1 octet of urgent data (seq=4097) (before fifth time)

```
6    0.167961 (0.0037)      sun.1305 > bsdi.5555:    . ack 1 win 4096 urg 4098
```

Universidad Carlos III de Madrid

Bulk data flow >> Other aspects >> URG Flag >> example (II)

- Observe traffic with tcpdump

Client writes 1024 octects (note the TCP is not delivering that data – window full – server sleeps)

```
7.0.171969 (0.0040)       sun.1305 > bsdi.5555:    . ack 1 win 4096 urg 4098
```

Client writes 1024 octects (note the TCP is not delivering that data – window full – server sleeps)

```
8.0.176196 (0.0042)       sun.1305 > bsdi.5555:    . ack 1 win 4096 urg 4098
```

```
9. 0.180373 (0.0042)      sun.1305 > bsdi.5555:    . ack 1 win 4096 urg 4098
10. 0.180768 (0.0004)     sun.1305 > bsdi.5555:    . ack 1 win 4096 urg 4098
11. 0.367533 (0.1868)     bsdi.5555 > sun.1305:    . ack 4097 win 0
12. 0.368478 (0.0009)     sun.1305 > bsdi.5555:    . ack 1 win 4096 urg 4098
```

.

Universidad Carlos III de Madrid

Bulk data flow >> Other aspects >> URG Flag >> example (III)

- Observe traffic with tcpdump

Server opens window (was 0) since it wakes up

```
13   9.829712 (9.4612)     bsdi.5555 > sun.1305:     . ack 4097 win 2048
```

TCP stack on client side is NOW able to send the urgent data

```
14   9.831578 (0.0019)     sun.1305 > bsdi.5555:     . 4097:5121(1024) ack 1 win
     4096 urg 4098
```

TCP on client side disables urgent data as it has arrived destination

```
15   9.833303 (0.0017)     sun.1305 > bsdi.5555:     . 5121:6145(1024) ack 1 win
     4096
```

Server opens the window

```
16   9.835089 (0.0018)     bsdi.5555 > sun.1305:     . ack 4097 win 4096
```

.

Bulk data flow >> Other aspects >> URG Flag >> example (IV)

- Observe traffic with tcpdump

TCP stack sends the lates data octect (was missing as URG data stole window from normal data)
Starts closing the connection

```
16. 9.835913 (0.0008)    sun.1305 > bsdi.5555:    FP 6145:6146(1) ack 1 win 4096
17. 9.840264 (0.0044)    bsdi.5555 > sun.1305:    . ack 6147 win 2048
18. 9.842386 (0.0021)    bsdi.5555 > sun.1305:    . ack 6147 win 4096
19. 9.843622 (0.0012)    bsdi.5555 > sun.1305:    F 1:1(0) ack 6147 win 4096
20. 9.844320 (0.0007)    sun.1305 > bsdi.5555:    . ack 2 win 4096
```

.

Timeouts and retransmissions

5. Timeouts and retransmissions:
   - A. Timers in TCP
   - B. Retransmissions and retransmission timer in TCP
   - C. Karn/Partridge algorithm
   - D. **Congestion avoidance** algorithm
   - E. **Fast recovery/Fast retransmit algorithm**

Timeouts/retrans. >> Retransmissions

- Error control is needed because
  - Segments may get lost
  - Segment may be received damaged
- Error control techniques
  - Retransmission schema
  - Error detection schema - CRC
- TCP uses a version of Go-Back-N ARQ (“Automatic Repeat Request”).
  - More efficient than Stop-and-wait ARQ but may send segments multiple times
- If TCP assumes a segment is lost, it retransmits the segment
  - If ack is not received and retransmission timer expires
  - Multiple ack are received for the same segment (following segment is lost?)

Universidad Carlos III de Madrid

Timeouts/retrans. >> Exercise

– Research and find differences between sop-and-wait ARQ and Go-Back-N ARQ

Universidad Carlos III de Madrid

Timeouts/retrans. >> Retr. Timer in TCP

- TCP allocates **a retransmission timer to every connection**

- When the timer expires (timeout), **the first unacknowledged segment** is retransmitted:
  - RTO = “Retransmission Timeout”

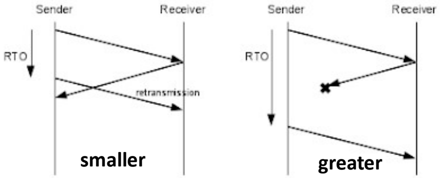
- Timer is deactivated when
  - All sent segments are acknowledged

- Timer activates when
  - Sending a segment and timer was deactivated
  - ACK of new data arrives, but it does not acknowledge all data sent
  - A segment is retransmitted

.

Timeouts/retrans. >> Retr. Timer in TCP >> RTO

- RTO value is critical in TCP performance.
- Assuming an optimal value, if we set RTO to a value
  - **smaller**, this may imply **unneeded retransmissions**
  - **greater**, this may imply **waiting too long for retransmission**



- Computing optimal value is not feasible, since network delays are not fixed
  - RTO value should be **computed dynamically** and adapted to network conditions.
  - Thus, **based in round-trip time** (RTT) measured by TCP

Image has been taken from http://sgros.blogspot.com/2012/02/calculating-tcp-rto.html.

Universidad Carlos III de Madrid

Timeouts/retrans. >> Retr. Timer in TCP >> RTO >> RTT Measurement

- RTO is based on RTT measurement
- TCP measures the time interval between segment transmission and corresponding ACK arrival.
- **A single measure is performed at a time**, i.e. there are no overlapping simultaneous measuring processes

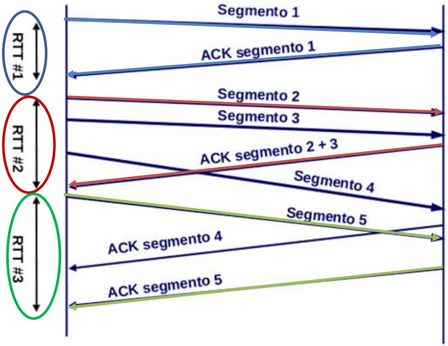


Image has been taken from http://sgros.blogspot.com/2012/02/calculating-tcp-rto.html.

Timeouts/retrans. >> Retr. Timer in TCP >> RTO comp. >> Measures

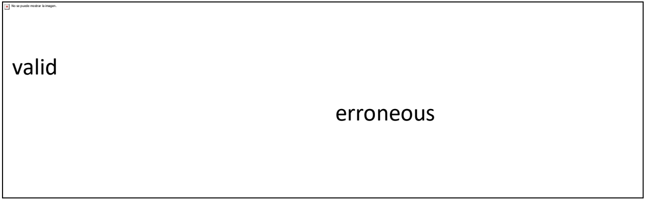
- The RTT measured that way is called RTT_M (RTT or Measured RTT)
  - A single timer for RTT_M that can experiment huge fluctuation
- Smoothed RTT (RTT_S)
  - Avoids big fluctuations of RTT
  - RTT_S (Smoothed Round-Trip Time) is the average mean between RTT_M and the latest calculated RTT_S:
    - First Measure: $RTT_S = RTT_M$
    - Subsequent $RTT_{S\,(n+1)} = \alpha\, RTT_{S(n)} + (1-\alpha)\, RTT_M$ where $\alpha=0.9$
- RTT deviation (RTT_D)
  - There is a huge fluctuation in RTT and thus forcing unnecessary retransmissions, the network load is incorporated (by Jacobson):
    - First Measure: $RTT_D = RTT_M/2$ or 3 seg
    - Subsequent $RTT_{D\,(n+1)} = (1-ß)\, RTT_{D\,(n)} + ß\, |\, RTT_M - RTT_{S(n)}\,|$ where $\beta = 0.25$

Timeouts/retrans. >> Retr. Timer in TCP >> RTO comp. >> Algorithms

- Jacobson algorithm
  - Considers RTT_S only
  - RTO is calculated after every RTT measure as
    - $RTT_{S\,(n+1)} = \alpha\, RTT_{S(n)} + (1-\alpha)\, RTT_M$ where α=0.9
    - $\mathbf{RTO}_{(n)} = ß\, RTT_{S(n)}$ where ß=2
- Jacobson/Karels algorithm
  - Combines RTT_S and RTT_D to overcome high fluctuation in RTT measurement, changes RTT_S calculation
  - RTO is calculated the following way
    - $RTT_{S\,(n+1)} = (1-\alpha)\, RTT_{S(n)} + \alpha\, RTT_M$ where α=0.125
    - $RTT_{D\,(n+1)} = (1-ß)\, RTT_{D\,(n)} + ß\, |\, RTT_M - RTT_{S(n)}\, |$ where β = 0.25
    - $\mathbf{RTO}_{(n)} = RTT_{S(n)} + 4\, RTT_{D\,(n)}$
    - Initial values RTO(0) = 6s

Timeouts/retrans. >> Retr. Timer in TCP >> RTO comp. >> Algorithms

- Problem of ambiguous ACKs: RTT_M measurement associated with retransmitted segmens can be erroneous



- **Karn/Partridge algorithm**:
  - Each time TCP retransmit a segment RTT measurement is stopped until an ACK of a non retransmitted segment is received
    - The last RTO is used
  - Every time a segment is retransmitted the RTO is set to the following (“exponential backoff”)
    - $RTO_{(n+1)} = \min(2RTO_{(n)}, 64)$ s

Timeouts/retrans. >> Congestion Avoidance

- Segment losses indicate there is network congestion
- Retransmission mechanism in TCP considering
  - RTO value as a RTT function
  - Exponential backoff
- There are specific algorithms to prevent congestion
  - Slow start [Jacobson 88]: already discuseed
  - Congestion avoidance [Jacobson 88].
  - Fast retransmit / fast recovery [Jacobson 90]

Timeouts/retrans. >> Congestion Avoidance

- When TCP considers a segment is lost (there is congestion)?
  - Retransmission timer expires
  - Duplicated acks are received (generated by segments out of sequence)
- Congestion is “avoided” (or prevented) combining “slow start” and “congestion avoidance” algorithms
  - cwnd must greatly decrease as congestion gets worse
  - cwnd must slowly increase when congestion gets better
- A new variable called “ssthresh” is introduced (Slow Start Threshold)
  - If cwnd <= ssthresh ⇒ TCP in slow start mode
  - If cwnd > ssthresh ⇒ TCP in congestion avoidance mode

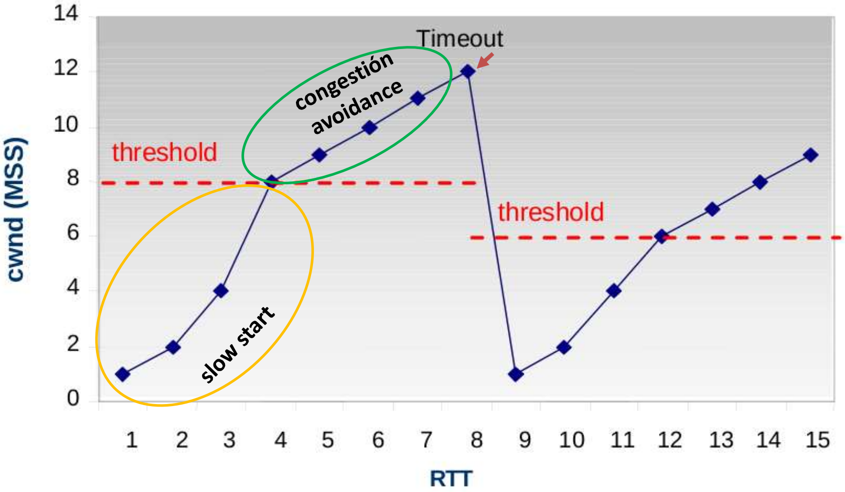
Timeouts/retrans. >> C. Avoid. >> Slow Start + congestion avoidance

1. Initial values
   - `cwnd=1` (segment) and `ssthresh=65535` octets
2. Sender may transmit `Vef=min(win,cwnd)` that is called effective window and is the minimum between win and the congestion window
3. If congestion is detected …
   - …because of duplicated ACK
     - `stthresh = max(2 segments, Vef/2)`
     - `cwnd(n+1) = cwnd(n)`
   - …because of a timeout
     - `stthresh = max(2 segments, Vef/2)`
     - `cwnd(n+1) = 1`
4. When new data is acknowledged
   - If TCP is working in slow start mode (`cwnd <= ssthresh`):
     - `cwnd = cwnd + 1 (segments)`
   - If TCP is working in congestion avoidance (`cwnd > ssthresh`):
     - `cwnd = cwnd + 1/cwnd (segments)`

Universidad Carlos III de Madrid

Timeouts/retrans. >> C. Avoid. >> Slow Start + congestion avoidance

- ssthresh and cwnd changes with congestion



Timeouts/retrans. >> Fast retransmit and fast recovery

- Fast retransmit is applied **when 3 or more duplicated ACK arrive**:
  - **Reason**: if TCP receives a segment out of order it must ACK without further delay (causing the duplicated acks)
  - Algorithm:
    - Immediately send a duplicate of the first unacknowledged segment (do not wait for timeout event)
    - Set `ssthresh=max(2 segments, Vef/2)`
    - Enters in slow start mode (`cwnd=1`)
- Fast recovery avoids slow start after a fast retransmit:
  - Reason: if acks are being received, data is flowing
- What kind of congestion fits into this schema? usually
  - Problems that affects one way (i.e. client -> server) but not the other one (ACKs still flow)
  - That are limited in time (the problem affects one or two segments)
- Note: in order to have, for instance, three duplicated ACK_5, you should see **4** ACK_5 (the original ACK_5 and three ACK_5 more)

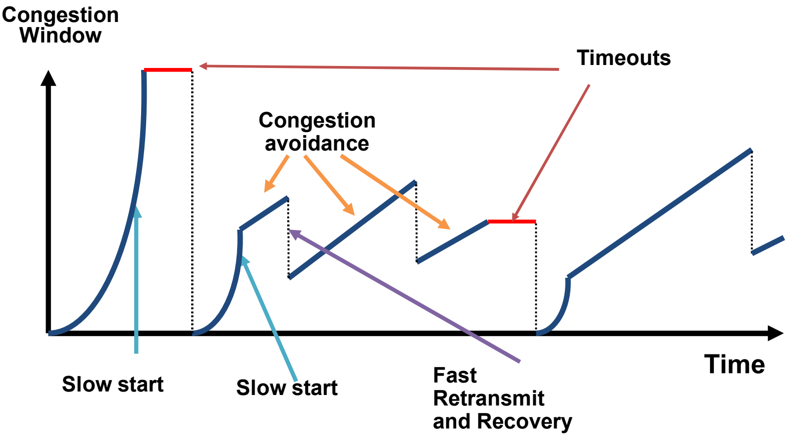
Timeouts/retrans. >> Fast retransmit and fast recovery

- In practice, both algorithms are implemented together according to the following:
- 1.) the third duplicated ack is received:
  - 1.1. Update ssthresh to `ssthresh=max(2 segments, Vef/2)`
  - 1.2. Retransmit lost segment immediately
  - 1.3. Update cwnd to `cwnd=(ssthresh+3)(segments)`
- 2.a) Another duplicate ACK is received:
  - Increase cwnd to `cwnd = cwnd+1`
  - Transmit new segments (if allowed by effective window)
- 2.b) An ACK of new data is received (non duplicated ack):
  - Update cwnd to `cwnd=ssthresh`
  - Ends FR/FR

Universidad Carlos III de Madrid

Timeouts/retrans. >> Fast retransmit and fast recovery

- Effect of congestion control algorithms



Persistent timer

6. Persistent timer
   A. Zero-window-deadlock
   B. Silly window syndrome

Persistent timer >> TCP flow control problems

- In TCP it may happen (either or both) that:
  - The sender sends data very slowly
  - The receiver reads data very slowly
- It can happen that (**Silly Window Syndrome**)
  - If the server reads very slowly, it closes the window
  - Until it is that small that is considered a “silly” value
- It may also happen that (**zero-window-size deadlock** situation):
  - Receiver closes window to zero size
  - Receiver sends ACK to open window, but it gets lost
  - Since ack segments are not acknowledged this may end in a deadlock situation

.

Persistent timer >> Zero-Window-size deadlock

- To overcome this zero-window-size deadlock TCP uses persistent timer
  - With WIN=0, the sender cannot send data
  - but CAN send periodic segments ACKs "window probes" to know if the win has opened
    - This segment contains only 1 byte of new data.
    - It has a sequence number, but its sequence number is never acknowledged; it is even ignored in calculating the sequence number for the rest of the data.
    - The probe causes the receiver to resend the latest acknowledgment which could has been lost.
  - TCP can send **one octet beyond offered window** (win)

- Persist timer establishes frequency of these segments
  - Depends on the implementation (can repeat 5 to 60 seconds)

.

Universidad Carlos III de Madrid

Persistent timer >> Zero-Window-size deadlock >> Example

- Svr4 sleeps for 100000 ms, so received data will not be read by the application: `svr4 % sock -i -s -P100000 5555`
- Client (bsdi) performs writes of 1024. After a time, TCP stack at srv4 closes the window (win=0)

```
0.0                bsdi.1027>svr4.5555: P 1:1025(1024) ack 1 win 4096
0.191961(0.1920) svr4.5555>bsdi.1027: . ack 1025 win 4096
0.196950(0.0050) bsdi.1027>svr4.5555: . 1025:2049(1024) ack 1 win 4096
0.200340(0.0034) bsdi.1027>svr4.5555: . 2049:3073(1024) ack 1 win 4096
0.207506(0.0072) svr4.5555>bsdi.1027: . ack 3073 win 4096
0.212676(0.0052) bsdi.1027>svr4.5555: . 3073:4097(1024) ack 1 win 4096
0.216113(0.0034) bsdi.1027>svr4.5555: P 4097:5121(1024) ack 1 win 4096
0.219997(0.0039) bsdi.1027>svr4.5555: P 5121:6145(1024) ack 1 win 4096
0.227882(0.0079) svr4.5555>bsdi.1027: . ack 5121 win 4096
0.233012(0.0051) bsdi.1027>svr4.5555: P 6145:7169(1024) ack 1 win 4096
0.237014(0.0040) bsdi.1027>svr4.5555: P 7169:8193(1024) ack 1 win 4096
0.240961(0.0039) bsdi.1027>svr4.5555: P 8193:9217(1024) ack 1 win 4096
0.402143(0.1612) svr4.5555>bsdi.1027: . ack 9217 win 0
```

NOTE: Before closing the window srv4 TCP implementation accepts 9216 octets due to a different implementation (it should have closed after receiving 4096)

Persistent timer >> Zero-Window-size deadlock >> Example

- Since win=0, TCP stack at bsdi will send probes…

Activates persist timer (send window probe ~5 s afterwards)

```
1    5.351561(4.9494)    bsdi.1027>svr4.5555:   . 9217:9218(1) ack 1 win 4096
2    5.355571(0.0040)    svr4.5555>bsdi.1027:   . ack 9217 win 0
```

send window probe ~5 s afterwards

```
1    10.351714(4.9961)   bsdi.1027>svr4.5555:   . 9217:9218(1) ack 1 win 4096
2    10.355670(0.0040)   svr4.5555>bsdi.1027:   . ack 9217 win 0
```

send window probe ~6 s afterwards

```
1    16.351881(5.9962)   bsdi.1027>svr4.5555:   . 9217:9218(1) ack 1 win 4096
2    16.355849(0.0040)   svr4.5555>bsdi.1027:   . ack 9217 win 0
```

send window probe ~12 s afterwards

```
1    28.352213(11.9964)  bsdi.1027>svr4.5555:   . 9217:9218(1) ack 1 win 4096
2    28.356178(0.0040)   svr4.5555>bsdi.1027:   . ack 9217 win 0
```

send window probe ~24 s afterwards

```
1    52.352874(23.9967)  bsdi.1027>svr4.5555:   . 9217:9218(1) ack 1 win 4096
2    52.356839(0.0040)   svr4.5555>bsdi.1027:   . ack 9217 win 0
```

send window probe ~48 s afterwards

```
1    100.354224(47.9974) bsdi.1027>svr4.5555:   . 9217:9218(1) ack 1 win 4096
2    100.358207(0.0040)  svr4.5555>bsdi.1027:   . ack 9217 win 0
```

send window probe ~60 s afterwards

```
1    160.355914(59.9977) bsdi.1027>svr4.5555:   . 9217:9218(1) ack 1 win 4096
```

Universidad Carlos III de Madrid

Persistent timer >> Silly window syndrome (SWS)

- Situation that may appear in protocols using sliding window mechanisms for flow control
  - Problem: small segments are exchanged, instead of full segments (MSS),
    - Receiver announces small window instead of waiting to have a larger one
    - Sender transmits small data segments, instead of waiting to be able to transmit larger amounts of data
- This situation **can only be solved if both parties follow rules** to avoid the silly window syndrome

.

Persistent timer >> Silly window syndrome (SWS) >> Rules

- Rules for avoiding SWS
- Receiver rules
  - should not announce windows of small size
  - do not announce an increase in window size until the window can fit a segment of MSS, half of the receiver buffer or `min(MSS,rcv_buffer/2)`
- Sender rules
  - Do not transmit until one of the following conditions hold:
    - A complete segment may be sent (MSS).
    - Half of the receiver buffer may be sent (estimated from maximum win announcement)
    - All buffered data may be sent providing that:
      - There is no outstanding segment of acknowledgment and Nagle is enabled, or
      - Nagle is disabled

.

Keepalive timer

7. Keepalive timer

Lesson outlook

Keepalive timer

- TCP does not close a connection when no data is being transmitted (no polling mechanisms)
- TCP introduces keepalive timer to test reachability of the other end during large inactivity periods
- Algorithm: if a connection is inactive longer than two hours, one end may send a probe segment to the other end, and:
  - The other end may answer normally ⇒ timer is set again (2h)
  - No answer is coming ⇒ up to 10 probe segments (75s apart) before informing the application that the connection is down
  - A RST is received ⇒ inform the application that the connection is down

.

Universidad Carlos III de Madrid

Keepalive timer

8. Relation with lower layers and others…

Lesson outlook

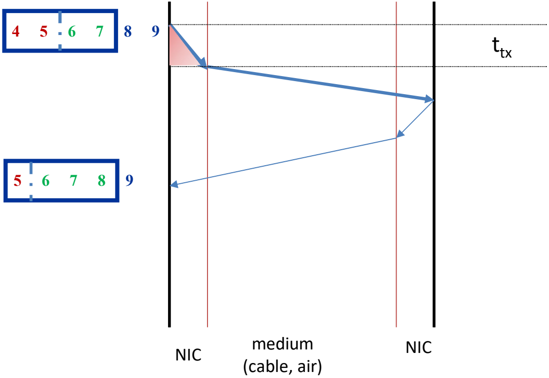
1. Introduction
2. Connection establishment and termination
3. Interactive data flow
4. Bulk data flow
5. Timeout and retransmission
6. Persistent timer
7. Keepalive timer
8. Relation with lower layers

Relation with lower layers... (I)

- What is MSS?
- MSS: maximum data length that will be sent in a connection
- Typically, IP datagram length = 40 + MSS
  - During session establishment, each side may announce its own MSS (default is 536 octets).
    - Option in SYN segment
    - MSS= MTU – IP header size – TCP header size
      - In a Ethernet announced MSS will be 1460
- MTU
  - Maximum transfer unit imposed by Link layer
- What is application bandwidth?
  - The amount of information the sender can send to the receiver during a given time
  - The link speed may also affect the bandwidth at application level

Relation with lower layers... > Continuous sending window

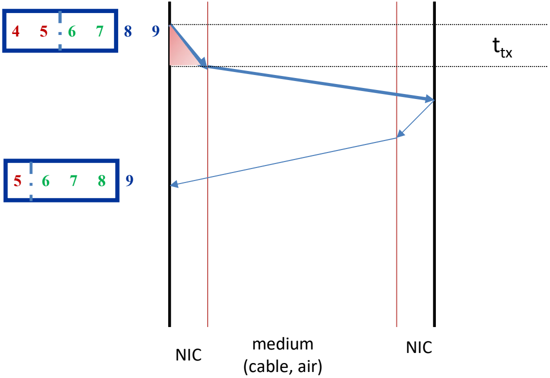
- Continuous sending window
  - Is the window that allows to fully use the link
  - To calculate it, let's introduce the time model



- The faster the link is, the smaller the transmisión time
- The smaller the transmisión time, the bigger could be the window
- The continuos sending window calculate the best effective window to squeeze the link the most

Relation with lower layers… > Continuous sending window

- Continuous sending window
  - Is the window that allows to fully use the link
  - To calculate it, let's introduce the time model

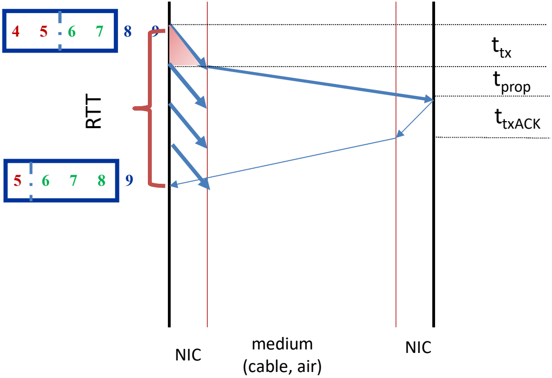


- During the t_{tx}, the link cannot be used for sending more data
- The time it takes sending a segment and receiving a response is RTT
- When an ACK is received, it adds room for other segments in sliding window

Universidad Carlos III de Madrid

Relation with lower layers… > Continuous sending window

- Continuous sending window
  - Calculated as how many transmisión times can fit into a RTT



- $RTT = t_{tx} + 2*t_{prop} + t_{txACK}$
- $V_{cs} = RTT / t_{tx}$

Relation with lower layers... > Continuous sending window

- Continuous sending window
- Please note
  - cwnd value can be higher that continuous send window
  - win value can be higher that continuous send window
  - Thus, Vef (effective window) can be higher than continuous send window
  - But there is no material way to send more data **during a RTT** than the continuous sending window

Relation with lower layers… > Continuous sending window

- Calculate Vcs (Vec in Spanish) for the following data
  - Link speed 1Gbps, MTU 1500Bytes, propagation time 2ms

  - Link speed 100Mbps, MTU 1500Bytes, propagation time 0,5ms

Relation with lower layers… (II)

- Simplifications
  - ACKs and sending pattern
  - RTT and windows
  - Congestion avoidance calculation

Relation with lower layers… > ACKs and sending pattern

- **Unless instructed otherwise** in the problem you can assume:
  - Every segment is individually acknowledged
    - This simplifies calculations
  - The sender always send if there are data to be sent
    - Whenever there is enough window and data pending to be sent
- Reason about differences in both cases

Universidad Carlos III de Madrid

Relation with lower layers… > RTT and windows

- Time a whole window takes to be sent and acknowledged
  - For the sake of simplicity sometimes problems are solved as step lock, so the first segment of the next window is sent only when the latest segment of the previous window is acknowledged
  - But in real scenarios TCP can send whenever it has available data and enough window
  - Draw the differences

Relation with lower layers… > Congestion avoidance/SS calculation

- If the sender sends the whole window during an RTT, how much will the window grow depending on the TCP mode? How can it be simplified?
  - Congestion avoidance