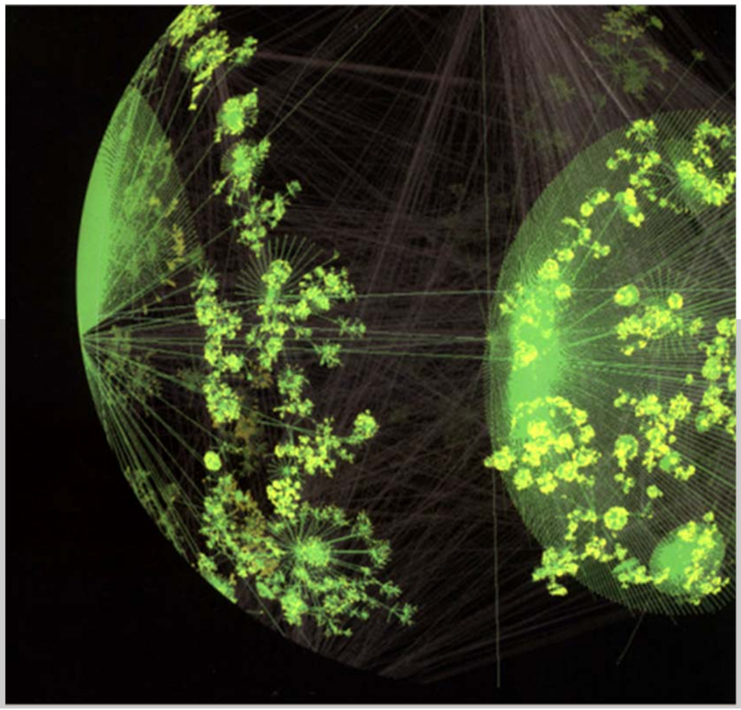
Chapter 6
TCP Study



TCP/IP Essentials

A Lab-Based Approach

Spring 2017

TCP Overview

- A transport layer protocol
- Provides connection-oriented, reliable service to applications, such as HTTP, email, FTP, telnet…
- Support unicast only
- Features
  - Error control
  - Flow control
  - Congestion control

TCP Connection

- Source and destination port numbers identify the sending and receiving application processes, respectively.
- Socket: the combination of and IP address and a port number.
- A TCP connection is uniquely identified by the two end sockets.

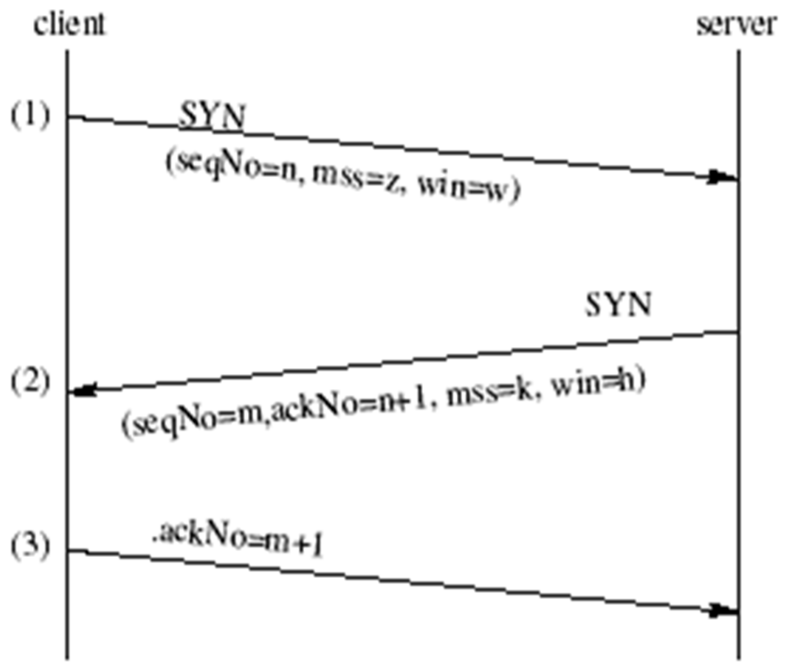
TCP Connection Management

- TCP connection establishment: two end TCP modules
  - allocate required resources for the connection, and
  - Negotiate the value of the parameter uses, such as
    - Maximum Segment Size (MSS) , given by the receiver side (a.k.a. destination)
    - Receiving buffer size (i.e. advertised window, WIN), given by the receiver
    - Initial sequence number (ISN), specified by the sender side (a.k.a. source)
- TCP connection termination

TCP Connection Establishment

Three-way Handshake

- An end host initiates a TCP connection by sending a packet with
  - ISN, say *n*, in the sequence number field,
  - An empty payload field,
  - MSS,
  - TCP receiving window size, and
  - SYN flag bit is set.
- The other end replies a SYN packet with
  - ACK=*n+1*
  - Its own ISN, say *m*
  - Its own MSS, and
  - Its ownTCP receiving window size
- The initiating host sends an acknowledgement: ACK=*m+1*



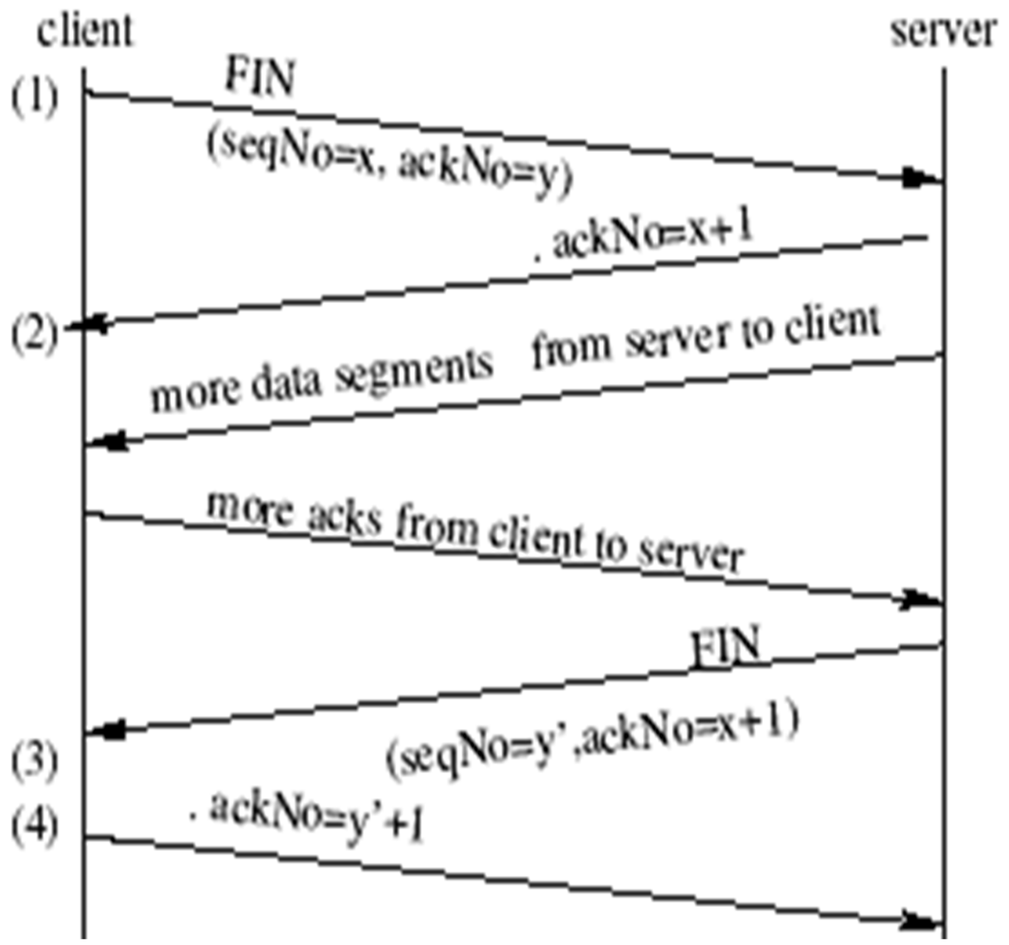
TCP Connection Termination

- A TCP connection is full duplex.
- Each end of the connection has to shut down its one-way data flow.
- After termination performed, the connection must stay in the TIME_WAIT state for twice the *Maximum Segment Life (MSL)* to wait for delayed segments.
- If an unrecoverable error is detected, either end can close the TCP connection by sending a RST segment.

TCP Connection Termination (cont'd)

Four-way handshake

- TCP Half-Close
  - One end TCP sends a packet with the FIN flag set.
  - The other end acknowledges the FIN segment.
  - The data flow in the opposite direction still works.
- Do Half-close in the opposite direction.



TCP Timers

TCP Connection Establishment Timer

- The maximum period of time TCP keeps on trying to build a connection before it gives up.

TCP Retransmission Timer

- Retransmit if no ACK is received for a TCP segment when this timer expires.

Delayed ACK Timer

- Used for delayed ACK in TCP interactive data flow.

TCP Keepalive Timer

- It reminds a station to check if the other end is still alive when a TCP connection has been idle for a long time.

TCP Timers (cont'd)

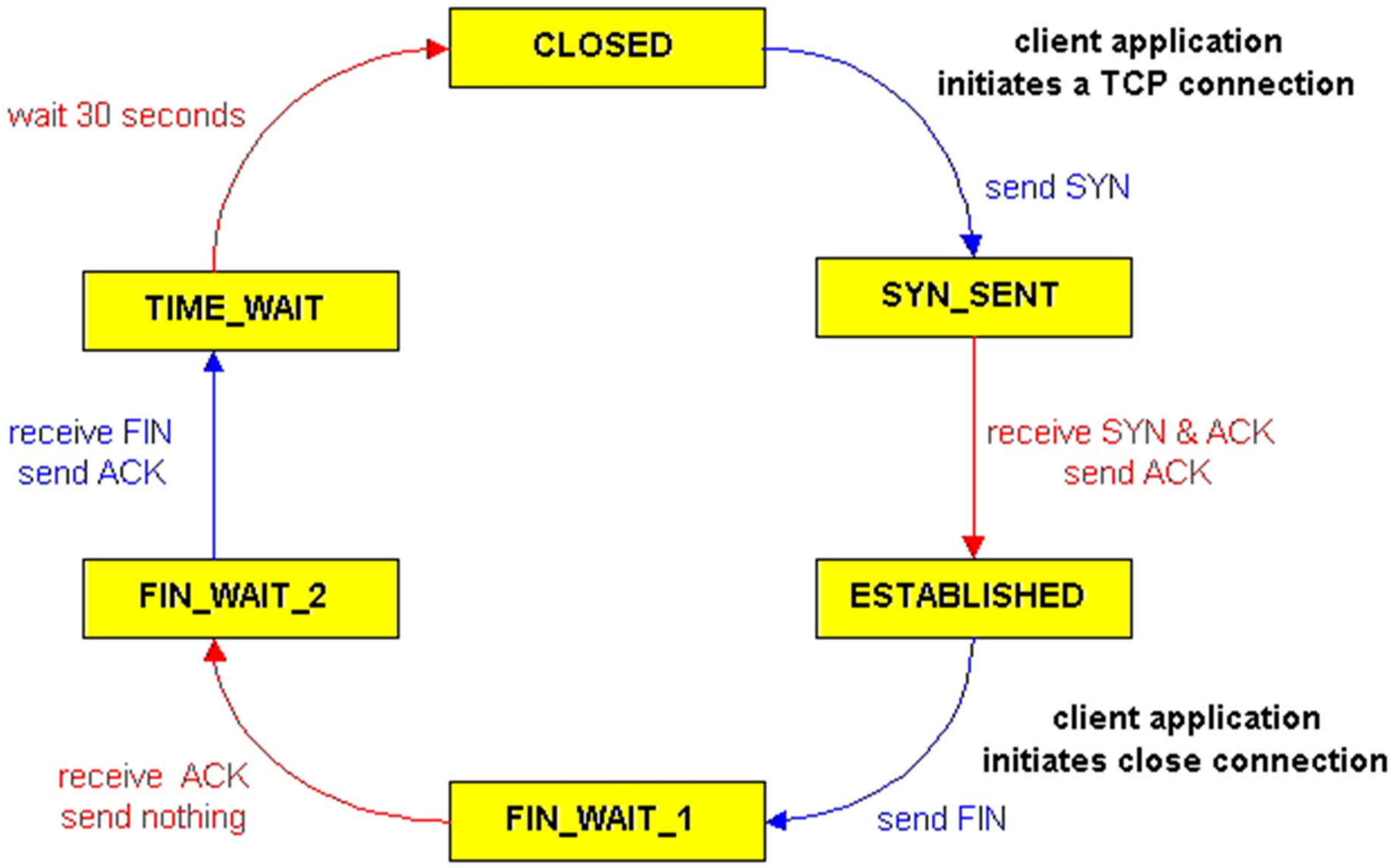
TCP Persist Timer

- Used in TCP flow control in the case of a fast transmitter and a slow receiver.
- While the advertised window size from the receiver is zero, the sender will probe the receiver for its window sizes when the timer times out.
- Uses the *Exponential Backoff* algorithm.

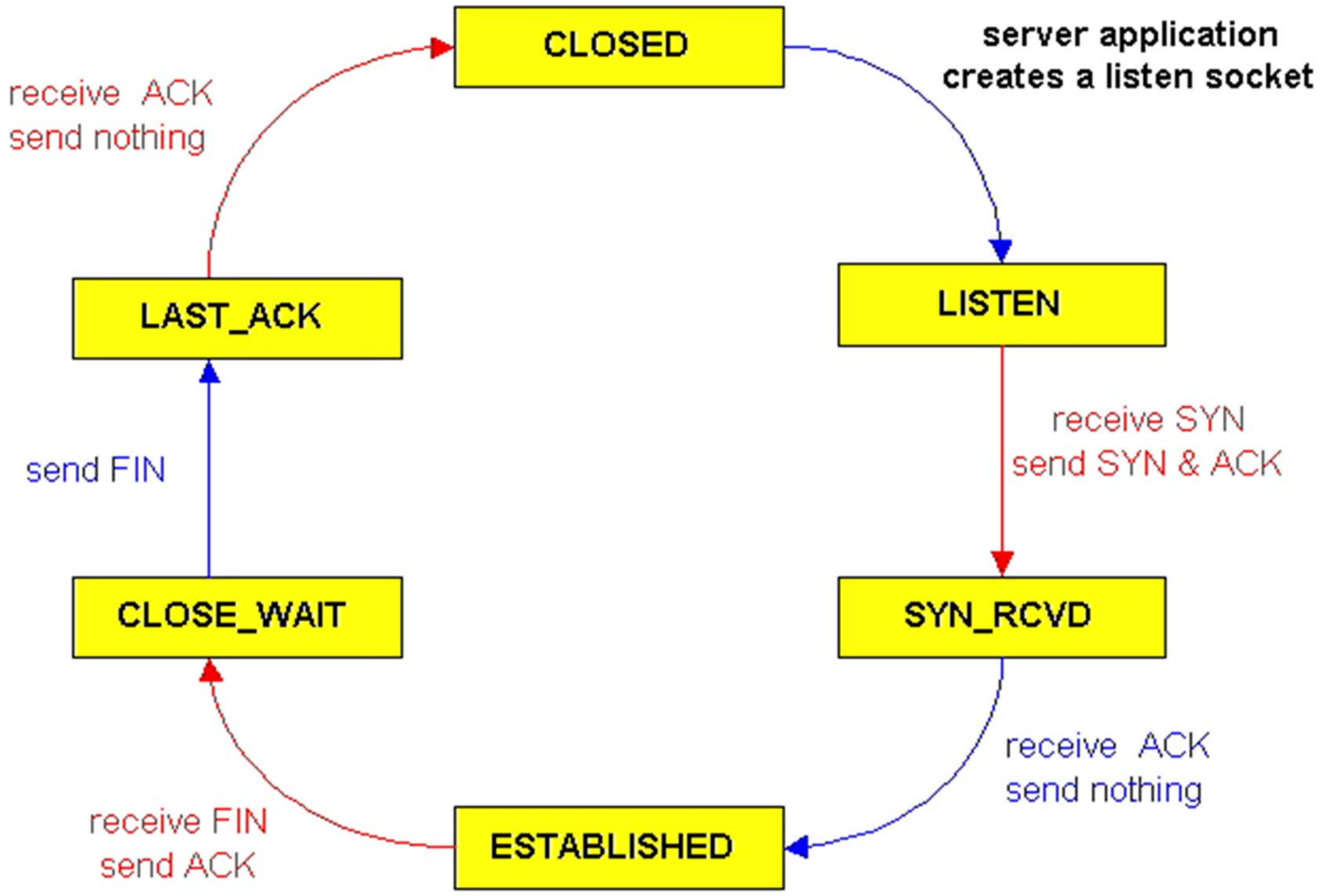
Two MSL Wait Timer

- Used in TCP connection termination.
- It is the period of time that a TCP connection keeps alive after the last ACK packet of the four-way handshake is sent.
- Prevents the delayed segments of a previous TCP connection from being interpreted as part of a new connection that uses the same sockets.

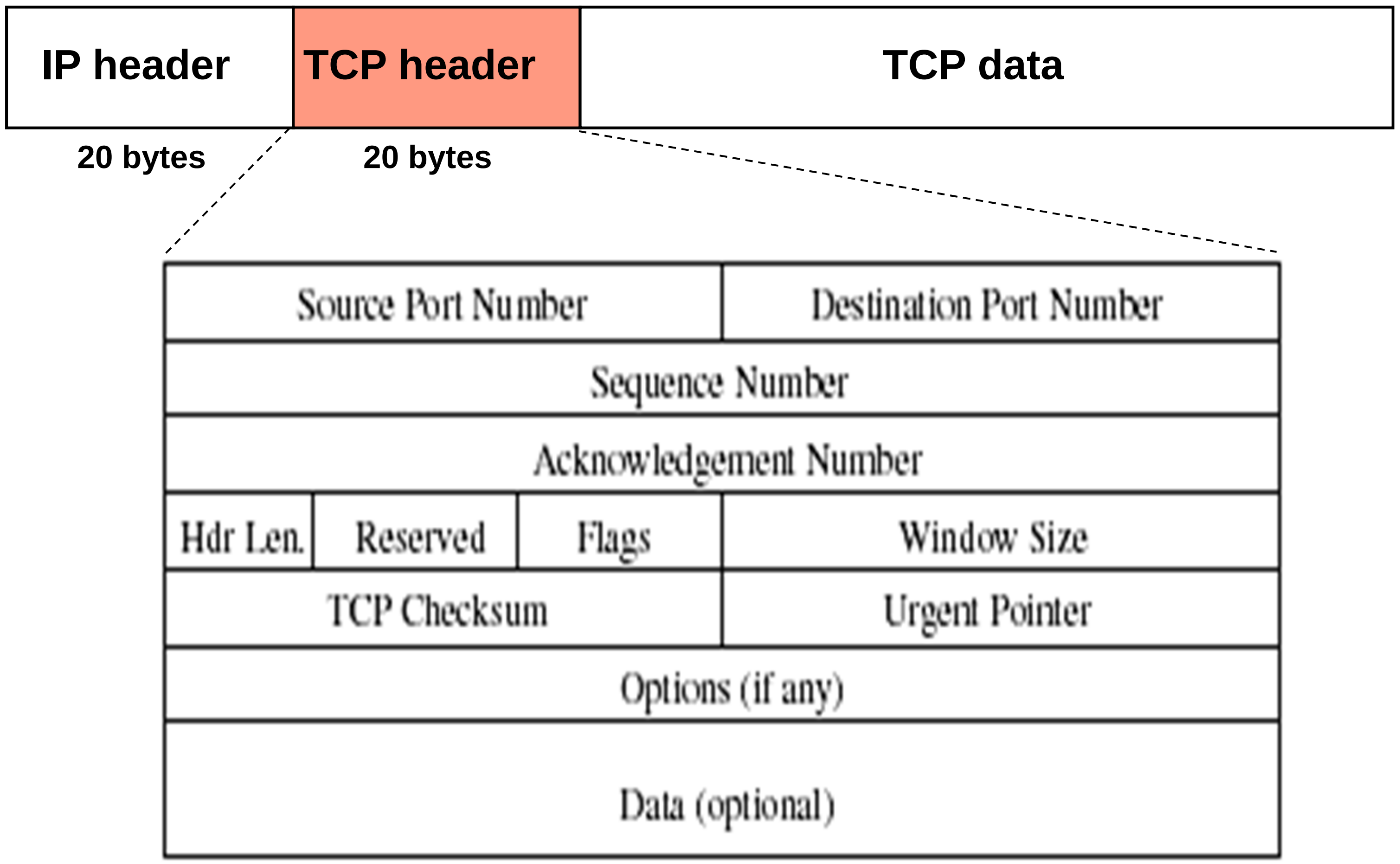
TCP Connection State Chart
- A Typical Sequence Visited by a Client



TCP Connection State Chart
- A Typical Sequence Visited by a Server



TCP Header Format



TCP Header Fields

- Source Port Number:
  - 16 bits
  - The port number of the source process
- Destination Port Number:
  - 16 bits
  - The port number of the destination process
- Sequence Number:
  - 32 bits
  - Identifies <u>the byte</u> in the stream of data from the sending TCP to the receiving TCP that the first byte of data in this segment represents
- Acknowledgement Number:
  - 32bits
  - The next sequence number that the host wants to receive
- Header Length, a.k.a. Data Offset
  - 4 bits
  - The length of the header <u>in 32-bit words</u>
- Reserved for future use: 6 bits

TCP Header Fields (cont'd)

- Window Size:
  - 16 bits
  - The maximum number of bytes that a receiver can accept
- TCP Checksum:
  - 16bits
  - Covers both the TCP header and TCP data
- Flags: 6 bits
  - URG: an urgent message is being carried.
  - ACK: the acknowledgment number is valid.
  - PSH: a notification from the sender to the receiver that it should pass all the data received to the application as soon as possible.
  - RST: signals a request to reset the TCP connection.
  - SYN: set when initiating a connection.
  - FIN: set to terminate a connection.
- Urgent Pointer
  - 16 bits
  - If the URG flag is set, the pointer points to the last byte of the urgent message in the TCP payload.

TCP Data Flow

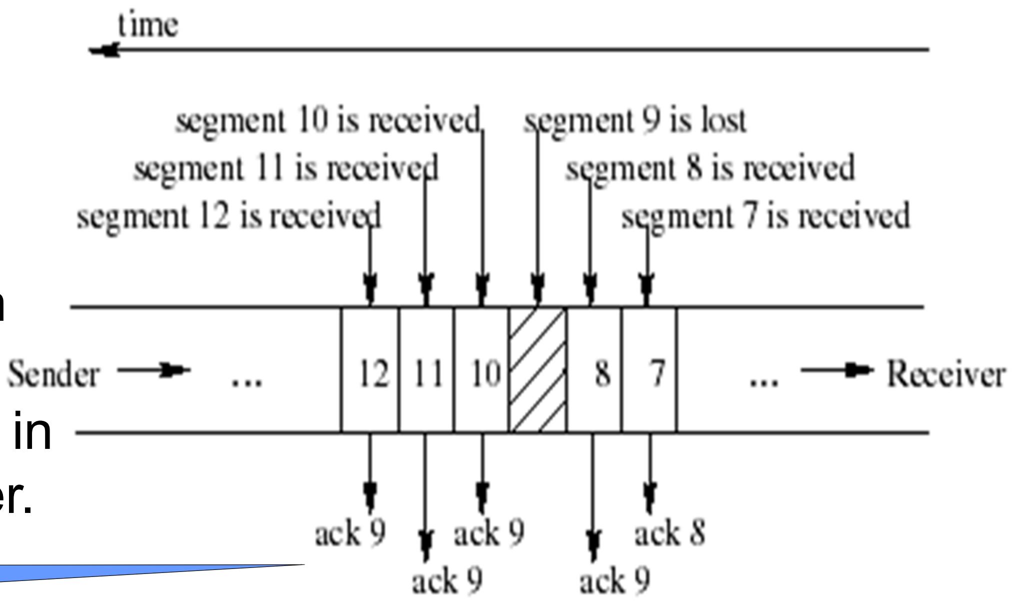
- TCP provides a byte-stream connection to the application layer.
- The sender TCP module
  - Receives a byte stream from the application and puts the bytes in a sending buffer.
  - Extracts the bytes from the sending buffer and sends to the lower network layer in blocks (TCP segments).
- The receiver TCP module
  - Uses a receiving buffer to store and reorder received TCP segments.
  - Restores a byte stream from the receiving buffer and sends to the application process.

TCP Error Control

- TCP segments may get lost in network or arrive at the destination out of order although TCP is a connection oriented transport protocol
  - TCP uses IP service.
  - IP is connectionless and unreliable.
- TCP provides error control for application data by retransmitting lost or errored segments.

Error Detection

- Each data byte is assigned a unique sequence number.
- TCP uses (positive) cumulative acknowledgments (by default) to inform the sender of the last correctly received byte in order.
- Error detection is performed in each layer of the TCP/IP stack by means of header checksums, and errored packets are dropped.
- If a segment is dropped, an acknowledgement will be sent to the sender for the 1st byte in this segment (expecting this byte).
- A gap in the received sequence numbers indicates a transmission loss or wrong order, and an acknowledgment for the first byte in the gap may be sent to the sender.



Assume each segment contains only one byte of data.

Error Detection Option – Selective Acknowledgement

- A window of TCP segments may be sent and received before an acknowledgment is received by the sender.
- Selective acknowledgment (SACK) is used to report multiple lost segments.
- The two ends use the TCP SACK-Permitted option (option 4 per RFC 2018) to negotiate if SACK is allowed while a TCP connection is being established.
- The receiver uses the TCP SACK option to acknowledge all segments that has been successfully received in the last window of segments, and the sender can retransmit more than one lost segment at a time.

TCP Retransmission

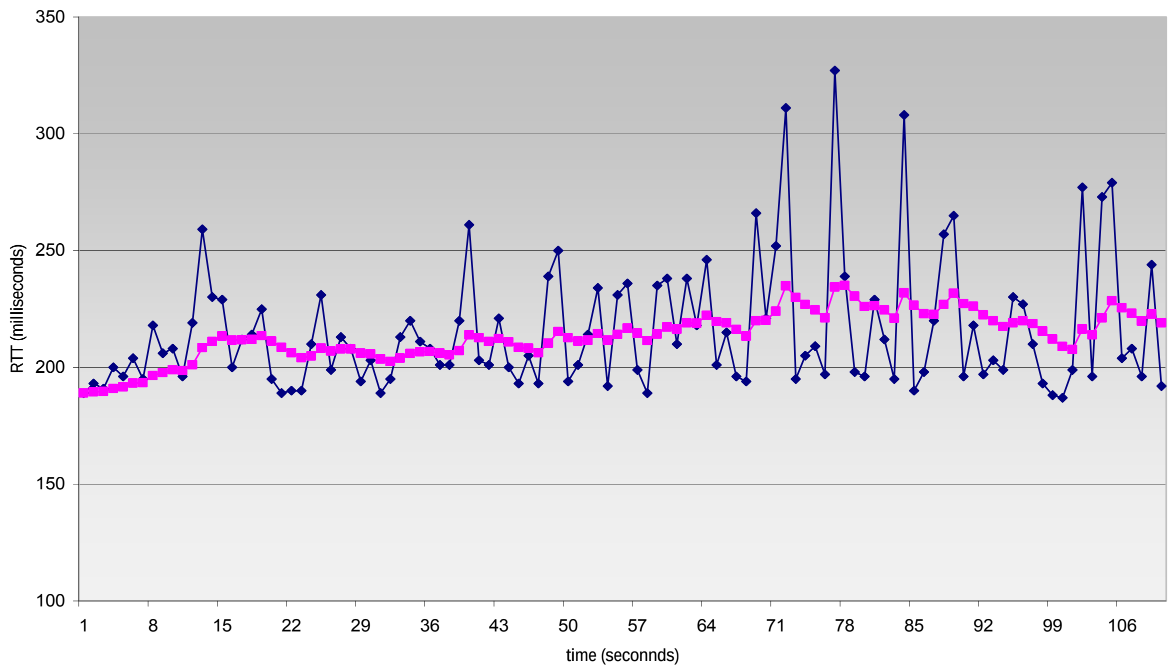
- A retransmission timer may be started by a sender when it sends out a TCP segment – referred as a target segment.
- If no ACK received when the timer expires, this segment is retransmitted.
- The value of the retransmission timer is critical to the TCP performance.
  - An overly small value causes frequent timeouts and unnecessary retransmissions.
  - A too large value causes a large delay when a segment gets lost.
  - The value should be larger than but of the same order of magnitude as a measurement of the Round Trip Time (RTT).
  - TCP continuously measure the RTT and updates the retransmission timer value, defined as Retransmission TimeOut (RTO), dynamically.

RTT Measurement

- The time difference between sending a target segment and receiving the ACK for the segment is measured.
- Each measured delay is call one RTT Measurement, denoted by *M*.
- Compute the RTO per RFC 2988:
  - RTT^s: smoothed RTT, set to the first measured RTT as $RTT^s_0 = M_0$.
  - RTT^d: smoothed RTT mean deviation, set initially as $RTT_0^d = M_0/2$
  - The initial value, $RTO_0 = RTT^s_0 + max\{G, 4 \times RTT^d_0\}$, where G is the timeout interval of the base timer.
  - For the i^{th} measured RTT value M_i:
    - $RTT^s_i = (1 - \alpha) \times RTT^s_{i-1} + \alpha \times M_i$,
    - $RTT^d_i = (1 - \beta) \times RTT^d_{i-1} + \beta \times |M_i - RTT^s_{i-1}|$,
    - $RTO_i = RTT^s_i + max\ \{G, 4 \times RTT^d_i\}$, where $\alpha=1/8$, $\beta=1/4$.
  - If RTO is less than 1 sec, round up to 1 sec. RTO may be capped (at least 60 sec.)

RTT Measurement (cont'd)

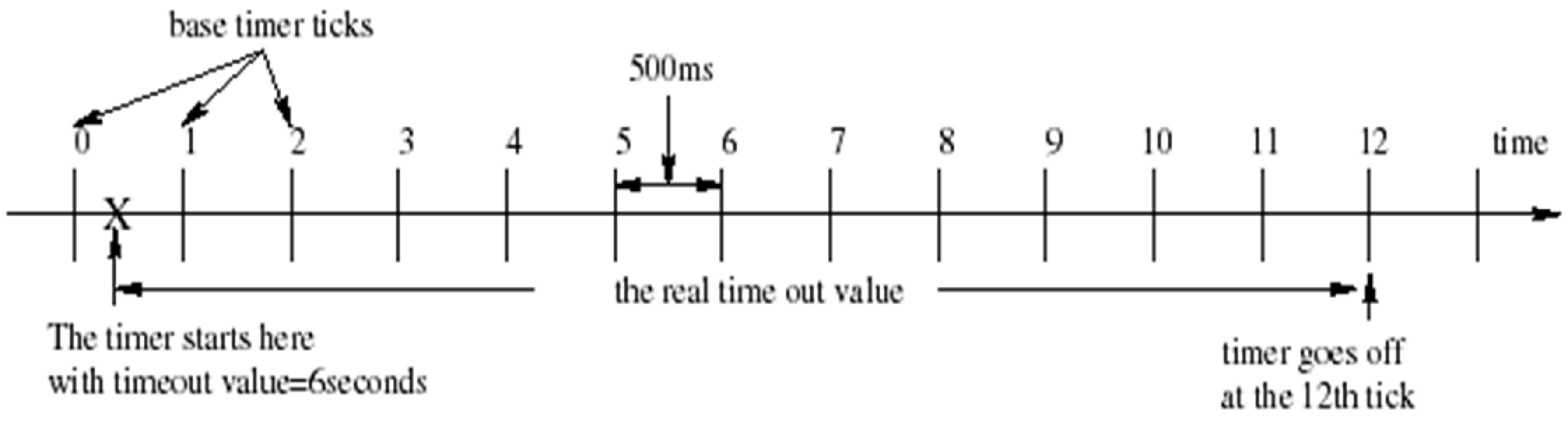
- RTT measurement is performed at the both ends of a TCP connection
  - −Each end may run the measurement only on one target segment at any time.



- RTT measurement is not performed for a retransmitted TCP segment.
  - −To avoid confused measurement: which segment does an ACK is referring to?
  - −Karn's Algorithm: RTTs and RTTd are not updated based on retransmission.

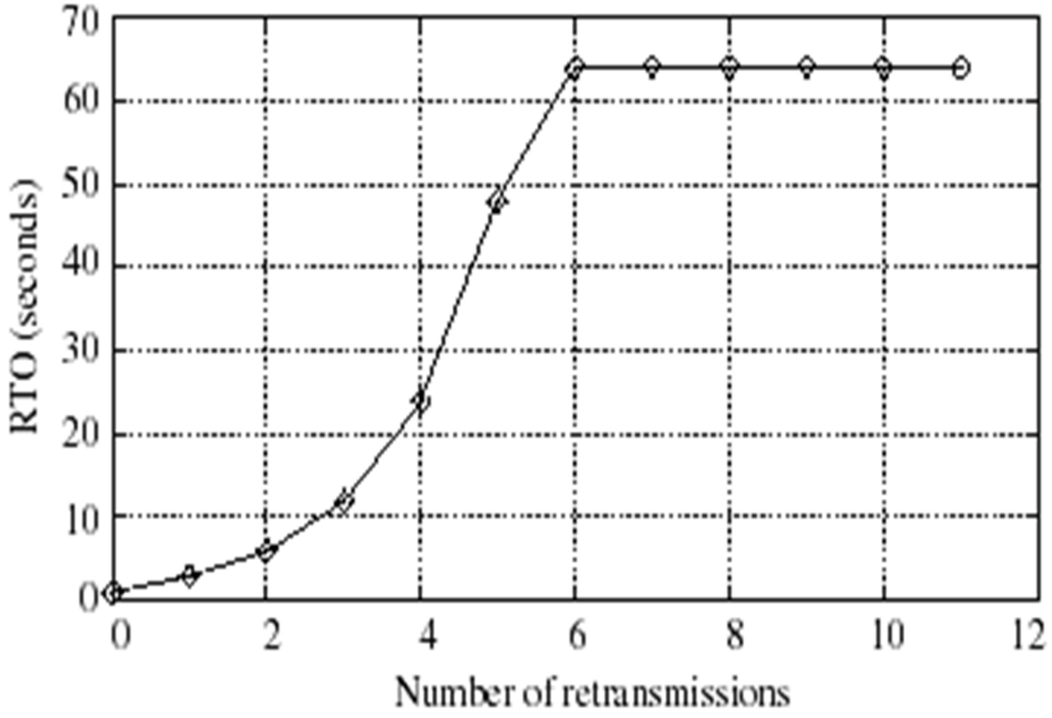
Retransmission Timer

- In some systems, a base timer that goes off every, e.g., 500ms, is used for both RTT measurements and RTO timeout counts.
- The measured RTT is $M = t \times 500ms$ if there are t base timer ticks during a measurement.
- All RTO timeouts occur at the base timer ticks.
  - In the example below, RTO = 6 sec that is counted by 12 base timer ticks.



RTO Exponential Backoff

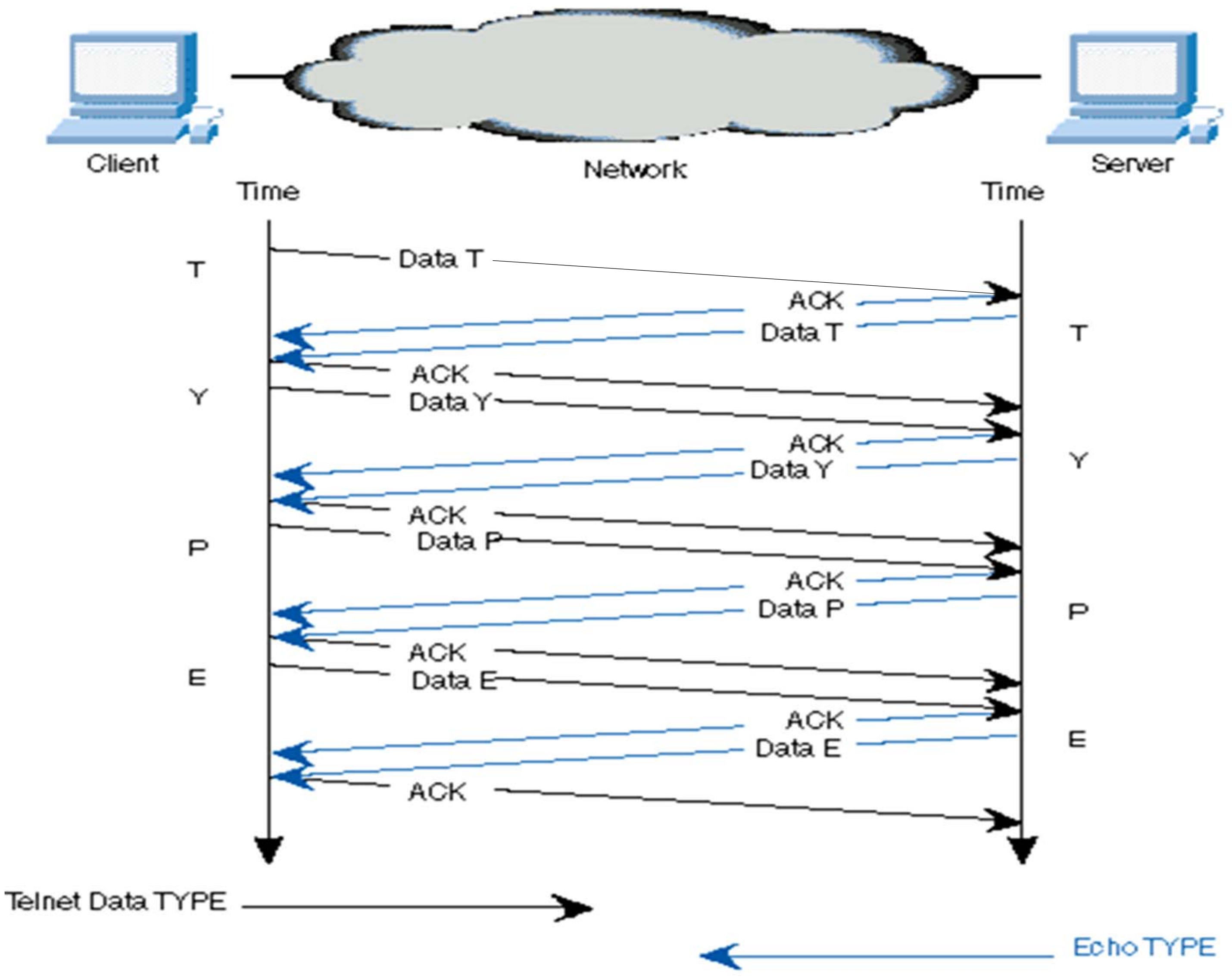
- Exponential Backoff algorithm is used to update RTO when the retransmission time expires for a retransmitted segment (no RTT measurement available).
  - RTO is doubled for each retransmission, but with a maximum value of 64 sec.



TCP Interactive Data Flow

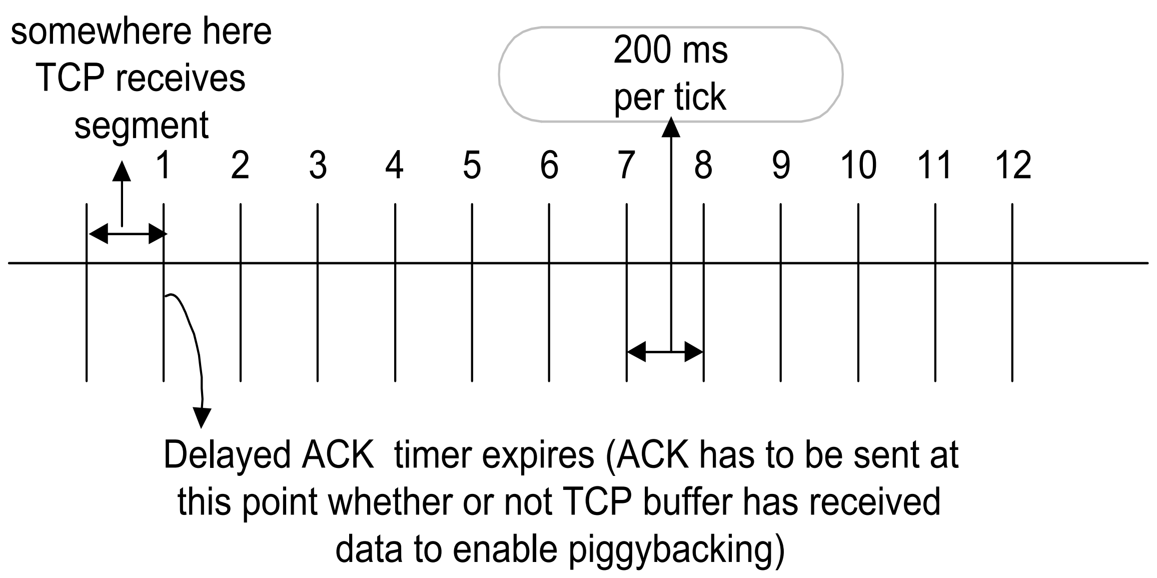
- TCP supports interactive data flow for interactive user applications
  - telnet
  - ssh
- To reduce the delay experienced by the user
  - A user keystroke is first sent from the user to the server.
  - The server echoes the key back to the user and piggybacks the acknowledgment for the key stroke.
  - The user sends an acknowledgment to the server for the received echo segment, and displays the echoed key on the screen.
- To reduce the number of small segments to be more efficient:
  - Delayed Acknowledgment
  - Nagle Algorithm

Delayed Acknowledgment – telnet example



Delayed Acknowledgment – for TCP Interactive Data Flow

- Delay acknowledgment timer goes off every K ms (e.g. 50ms).
- TCP delays sending the ACK for a data segment until the next tick of the delayed acknowledgment timer,
  - If there is new data to send during this period, the ACK can be piggybacked with the data segment.
  - Otherwise, an ACK segment is sent till K ms.
- An ACK may be delayed from 0 ms up to K ms.


somewhere here TCP receives segment
200 ms per tick
1
2
3
4
5
6
7
8
9
10
11
12
Delayed ACK timer expires (ACK has to be sent at this point whether or not TCP buffer has received data to enable piggybacking)

Nagle algorithm – for TCP Interactive Data Flow

- Each TCP connection can have only one small segment (say one byte) outstanding (not been acknowledges).
- TCP sends one segment and buffers all subsequent bytes until an ACK for the first segment is received.
- All buffered bytes are sent in a single segment.

```
if there is new data to send
  if the window size >= MSS and
        available data is >= MSS
    send complete MSS segment now
  else
    if there is unconfirmed data still in
        the pipe
      enqueue data in the buffer until an
            acknowledge is received
    else
      send data immediately
    end if
  end if
end if
```

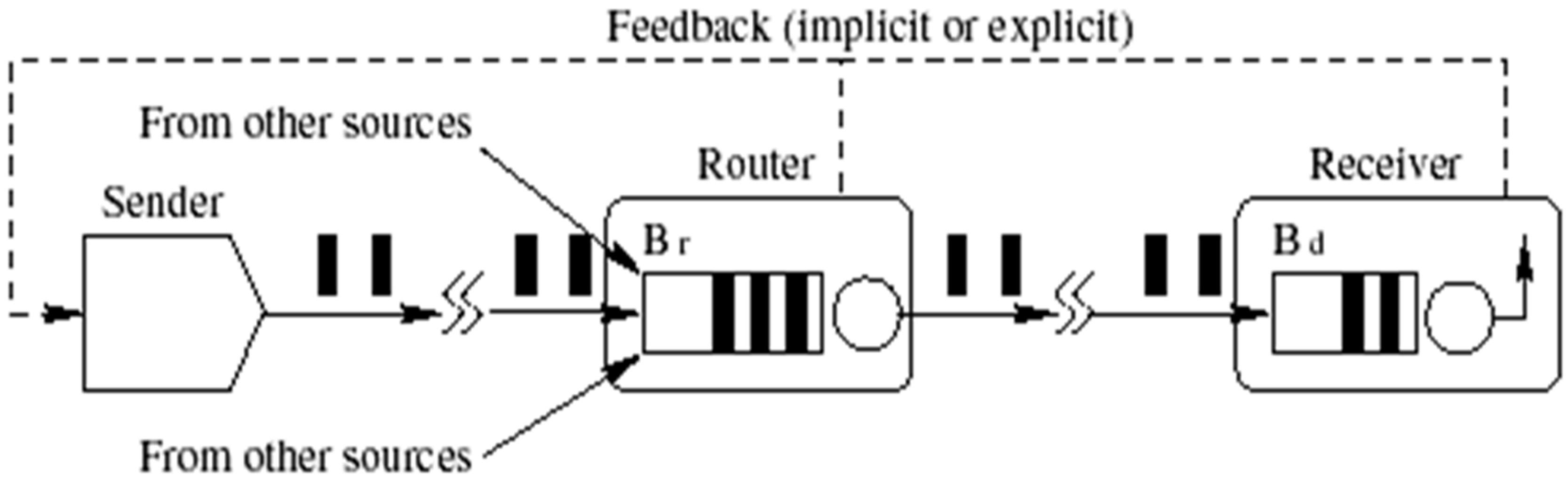
- More efficient than sending multiple segments, each with one byte of data, at the cost of increased delay for the user.

TCP Bulk Data Flow

- TCP supports bulk data flows, where a large number of bytes are sent through the TCP connection.
- Applications: email, FTP, HTTP, …
- Congestion may occur in an IP network, or even in the case of a fast transmitter and a slow receiver, and packets will be dropped when the receiver buffer is full.
  - The source always wants to increase sending rate to achieve high throughput.
  - The source rate should be bounded by the maximum rate that can be allowed without causing network congestion or receiver buffer overflow for a low packet loss rate.

Congestion Control and Flow Control

- Congestion control and flow control are used to cope with congestion problems.
- Let the source be adaptive to the buffer occupancies in the routers and the receiver.
- TCP used Slow Start and Congestion Avoidance to react to congestion in routers and to avoid receiver buffer overflow.



TCP Sliding Window Flow Control

- The receiver advertises the maximum amount of data it can receive (the Advertised Window, or *awnd*).
- The sender is not allowed to send more data than the advertised window.
- The sending rate is effectively determined by
  - the advertised window, and
  - how quickly a segment is acknowledged (to slide the window forward).
- Congestion can still occur.

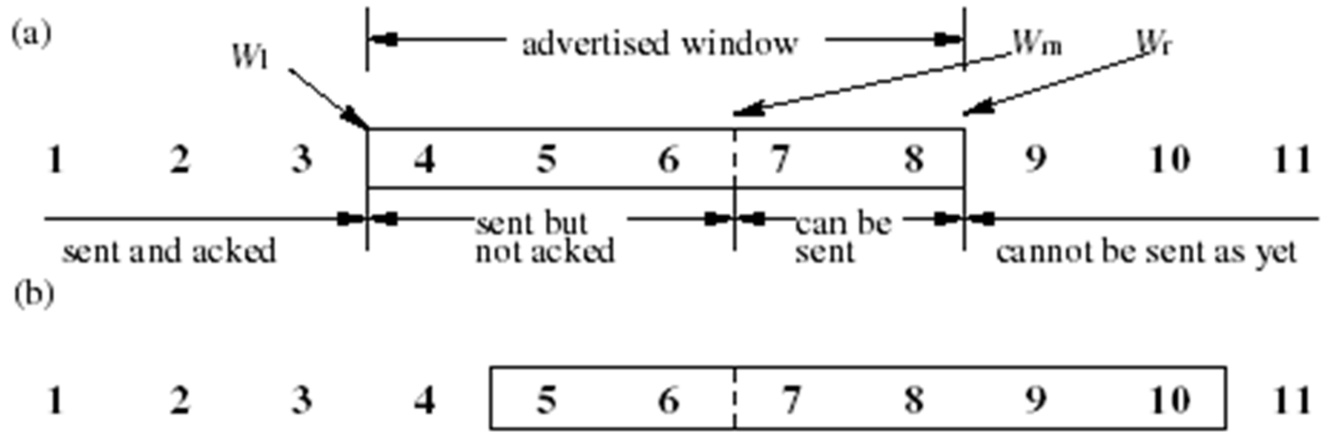
TCP Sliding Window Flow Control

The receiver notifies the sender

- the next segment it expects to receive (Acknowledgement Number), and
- the amount of data it can receive (Window Size)

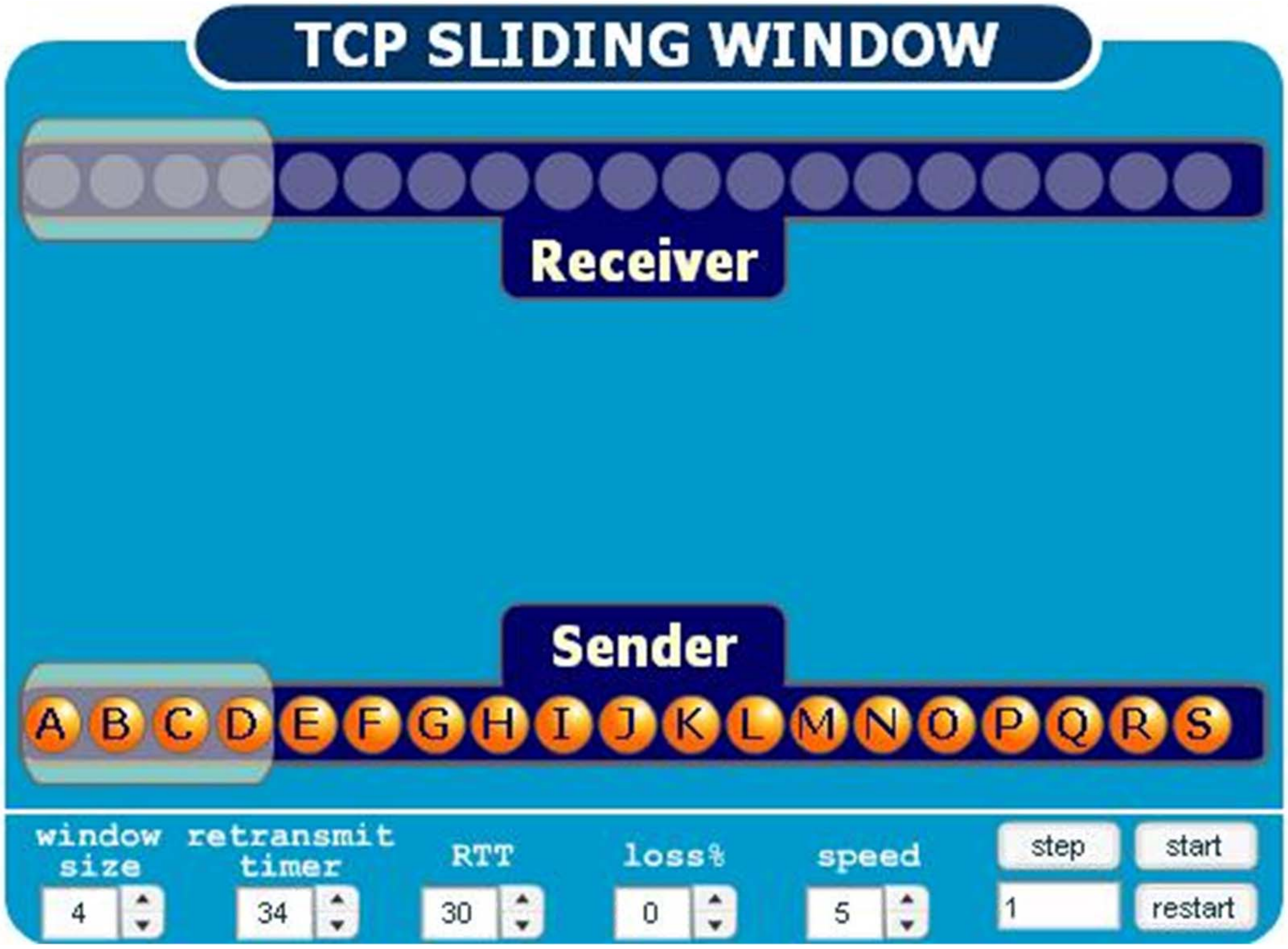
The sliding window

- W_l moves forward (to the right) when a new segment is acknowledged.
- W_m moves forward when each new segment is sent.
- W_r moves along with W_l, or
  - forward (to the right) when a larger window is advertised by the receiver or when new segments are acknowledged,
  - Backward (to the left) when a smaller window is advertised.



TCP Sliding Window Demo

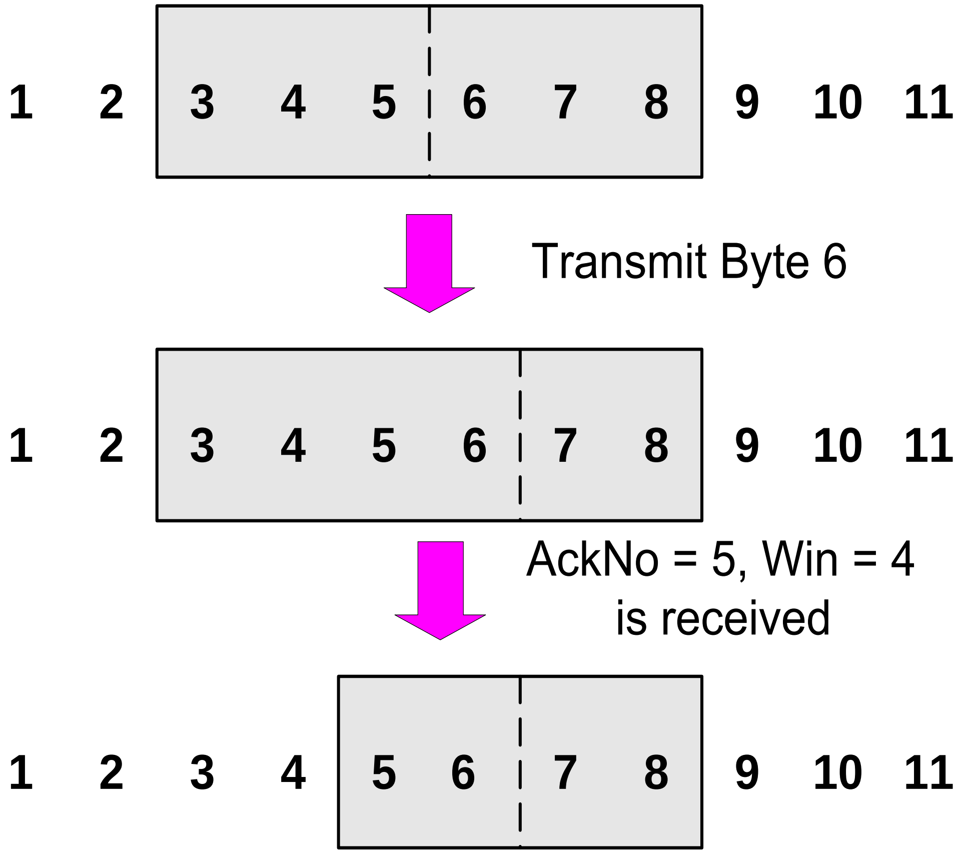
http://www2.rad.com/networks/2004/sliding_window/demo.html



window size: 4
retransmit timer: 34
RTT: 30
loss%: 0
speed: 5
1

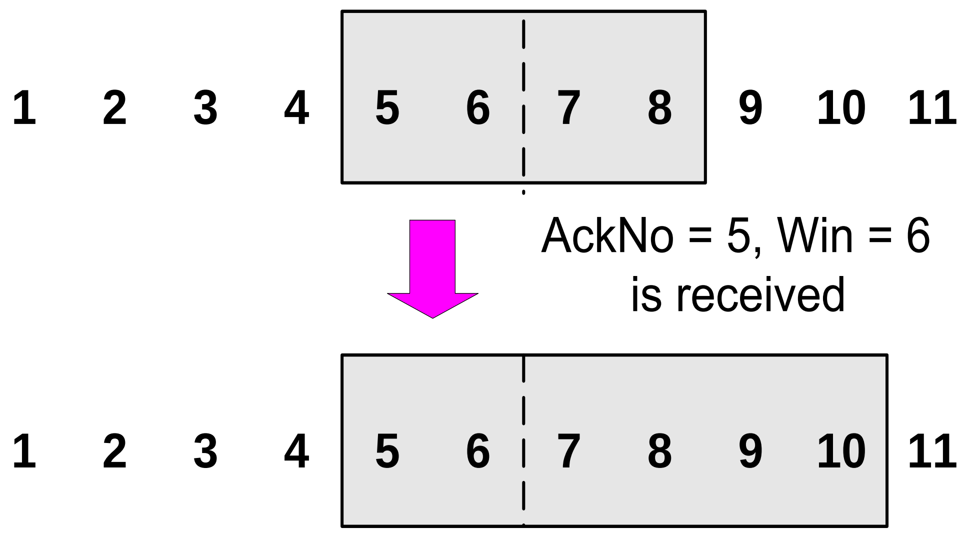
Sliding Window: "Window Closes"

- Transmission of a single byte (with SeqNo = 6) and acknowledgement is received (AckNo = 5, Win=4):



Sliding Window: “Window Opens”

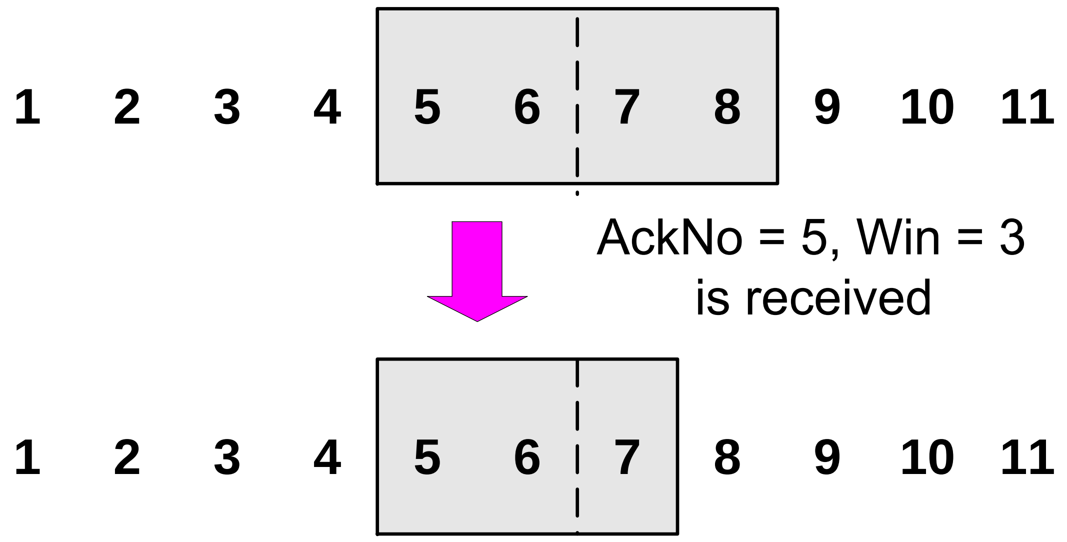
- Acknowledgement is received that enlarges the window to the right (AckNo = 5, Win=6):



- A receiver opens a window when TCP buffer empties (meaning that data is delivered to the application).

Sliding Window: “Window Shrinks”

- Acknowledgement is received that reduces the window from the right (AckNo = 5, Win=3):



- Shrinking a window should not be used – Host requirements RFC strongly discourages this
  - How to avoid window shrinking?

TCP Congestion Control

- TCP uses a control scheme to adapt to network congestion and achieve a high throughput.
- Usually the buffer in a router is shared by many TCP connections and other non-TCP data flows.
- TCP needs to adjust its sending rate in reaction to the rate fluctuations of other flows sharing the same buffer.
  - A new TCP connection should increase its rate as quickly as possible to take all the available bandwidth.
  - TCP should slow down its rate increase when the sending rate is higher than some threshold.
- The sender can infer congestion when a retransmission timer goes off.
- The receiver reports congestion implicitly by sending duplicate acknowledgements.

TCP Congestion Control (2/5) – Parameters

- The receiver provides two variables to influence senders transmission rate:
  - advertised Window size (*awnd*)
  - Maximum Segment Size (*MSS*)
- The sender maintains two variables for congestion control:
  - congestion window size (*cwnd*): to upper bound the sender rate.
  - slow start threshold (*ssthresh*)
- The sender uses Allowed Window = min (cwnd, awnd) as the size of the sliding window.

TCP Congestion Control (3/5) – Slow Start & Congestion Avoidance

- Slow Start and Congestion Avoidance

```
1) if $cwnd \le ssthresh$ then                    /* Slow Start Phase */
       each time an ACK is received:
           $cwnd = cwnd + segsize$ (= $MMS$)
   else (i.e. $cwnd > ssthresh$)                  /* Congestion Avoidance Phase */
       each time an ACK is received:
           $cwnd = cwnd + segsize \times segsize / cwnd + segsize / 8$
   end
2) when a congestion occurs (indicated by retransmission timeout), reset
           $ssthresh = \max[\,2\ segsize, \min(cwnd, awnd)/2\,]$
           $cwnd = 1\ segsize$                    /* into Slow Start Phase */
```

- **Note:**
  - Set *cwnd* = 1 *segsize* (= 1 *MMS* bytes) whenever starting traffic on a new connection, or whenever increasing traffic after congestion was experienced.
  - *ssthresh* takes an initial value of 65535 bytes, and changes only when a congestion occurs

TCP Congestion Control (4/5)
– Fast Retransmit & Fast Recovery

Fast Retransmit

- After receiving three duplicate acknowledgments, the sender retransmits the segments without waiting for the retransmission timer to expire.
- After the retransmission, congestion avoidance is performed,

Fast Recovery – used when three or more duplicated ACKs are received

1) after the third duplicate ACK is received:

ssthresh = max [2 segsize, min (cwnd, awnd)/2]

retransmit the missing segment, and then

cwnd = ssthresh + 3 segment /* in Congestion Avoidance */

2) for each additional duplicate acknowledgement received:

cwnd = cwnd + segsize /* in Congestion Avoidance */

transmit a segment if allowed by the window size

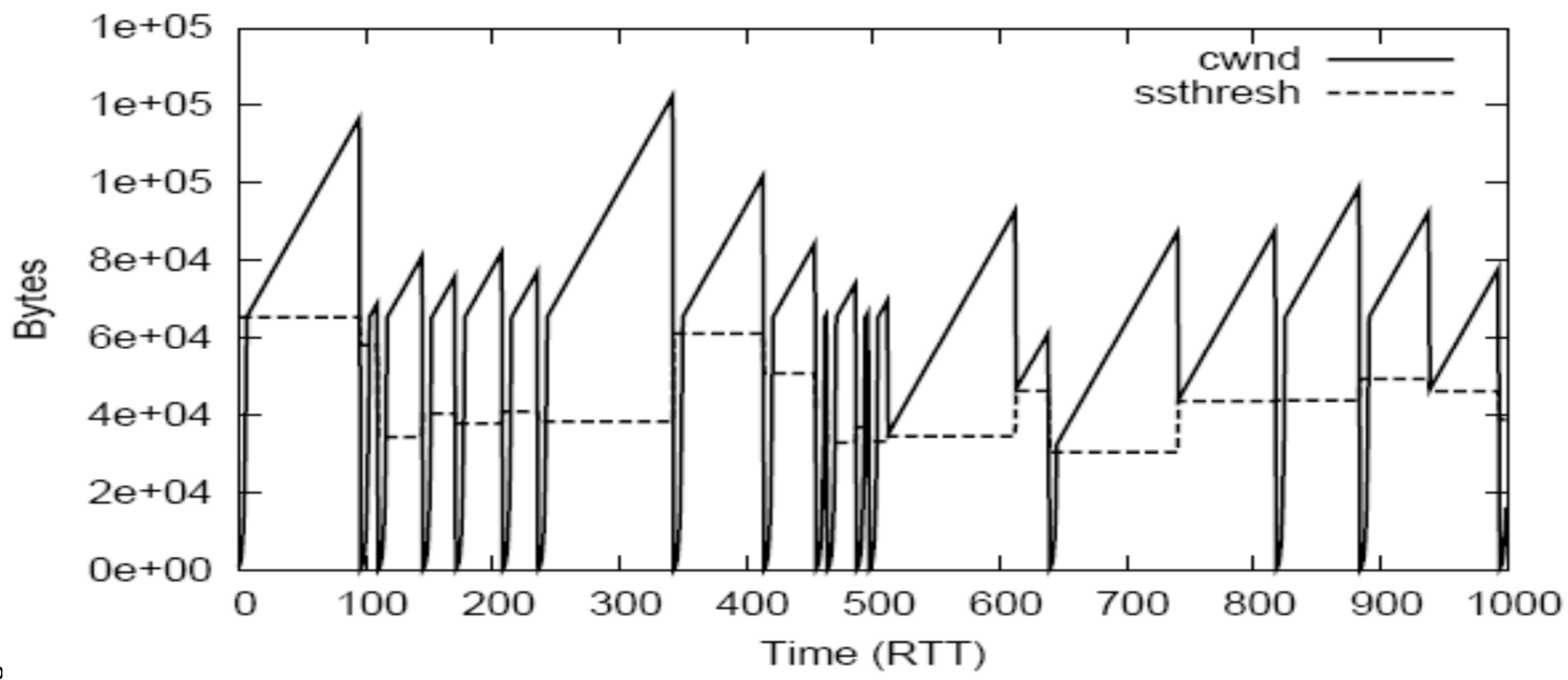
3) when the acknowledgement for the retransmitted segment arrives (new ACK):

cwnd = ssthresh + segsize /* in Congestion Avoidance */

TCP Congestion Control (5/5)

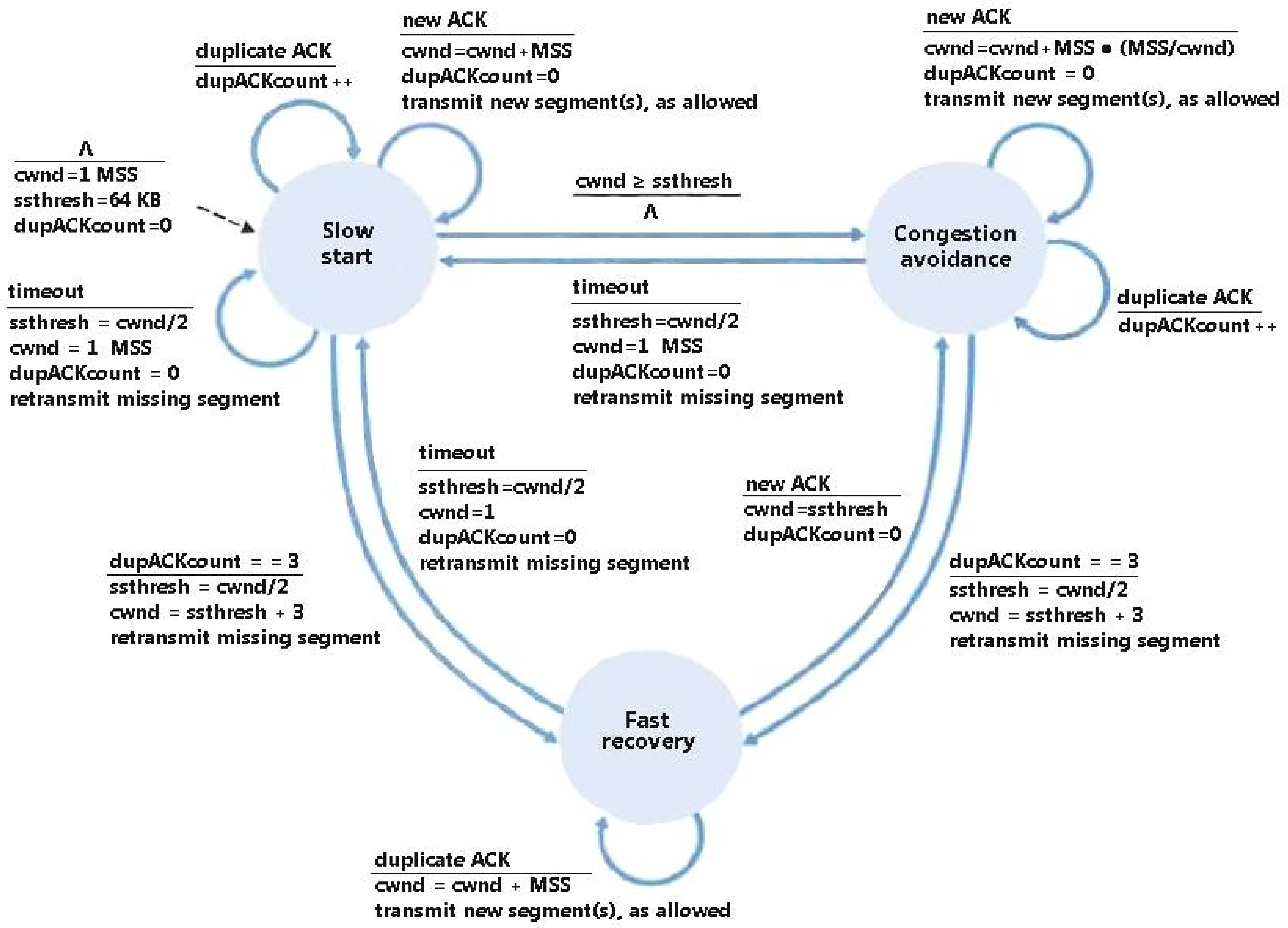
The evolution of cwnd and ssthresh for a TCP connection, including

- Slow start and Congestion avoidance
  - *cwnd* has two phases: an exponential increase phase and a linear increase phase.
  - *cwnd* drops drastically when there is a packet loss.
- Fast retransmit and fast recovery, occur at time around 610, 740, 950.



TCP ACK generation [RFC 1122, RFC 2581]

event at receiver	*TCP receiver action*
arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	delayed ACK. Wait up to 500 ms for next segment. If no next segment, send ACK
arrival of in-order segment with expected seq #. One other segment has ACK pending	immediately send single cumulative ACK, ACKing both in-order segments
arrival of out-of-order segment higher-than-expect seq. # . Gap detected	immediately send *duplicate ACK,* indicating seq. # of next expected byte
arrival of segment that partially or completely fills gap	immediate send ACK, provided that segment starts at lower end of gap


Λ
cwnd=1 MSS
ssthresh=64 KB
dupACKcount=0
duplicate ACK
dupACKcount++
new ACK
cwnd=cwnd+MSS
dupACKcount=0
transmit new segment(s), as allowed
Slow start
cwnd ≥ ssthresh
Λ
Congestion avoidance
new ACK
cwnd=cwnd+MSS • (MSS/cwnd)
dupACKcount = 0
transmit new segment(s), as allowed
duplicate ACK
dupACKcount++
timeout
ssthresh = cwnd/2
cwnd = 1 MSS
dupACKcount = 0
retransmit missing segment
timeout
ssthresh=cwnd/2
cwnd=1 MSS
dupACKcount=0
retransmit missing segment
timeout
ssthresh=cwnd/2
cwnd=1
dupACKcount=0
retransmit missing segment
new ACK
cwnd=ssthresh
dupACKcount=0
dupACKcount = = 3
ssthresh = cwnd/2
cwnd = ssthresh + 3
retransmit missing segment
dupACKcount = = 3
ssthresh = cwnd/2
cwnd = ssthresh + 3
retransmit missing segment
Fast recovery
duplicate ACK
cwnd = cwnd + MSS
transmit new segment(s), as allowed

Tuning the TCP/IP Kernel

- TCP/IP parameters
  - A set of default values may not be optimal for all applications.
  - The network administrator may wish to turn on or off some TCP/IP functions for performance or security considerations.
- Many Unix and Linux systems provide some flexibility in tuning the TCP/IP kernel.

Tuning the TCP/IP Kernel in Red Hat Linux

- /sbin/sysctl is used to configure the Linux kernel parameters at runtime.
  - Default kernel configuration file is /sbin/sysctl.conf.
  - Frequently used sysctl options:
    - sysctl –a or sysctl –A: list all current values.
    - sysctl –p *file_name*: load the sysctl setting from a configuration file.
    - sysctl –w *variable=value*: change the value of the parameter
- TCP/IP related kernel parameters are stored in /proc/sys/net/ipv4/. Files can be modified directly to change setting.

TCP Diagnostic Tools

1. The Distributed Benchmark System (DBS)
2. NIST Net
3. tcpdump output of TCP packets

The Distributed Benchmark System (DBS)

- A benchmark for TCP performance evaluation.
- Can be used to run tests with multiple TCP connections or UDP flows.
- Three tools:
  - dbsc: the DBS test controller
  - dbsd: the DBS daemon, running on each host participating in the test
  - dbs_view: a *Perl* script file, used to plot the experiment results

The Distributed Benchmark System (DBS)

DBS uses a command file to describe the test setting, which specifies

- How many TCP or UDP flows to generate
- The sender and receiver for each flow
- The traffic pattern and duration of each flow
- Which statistics to collect.

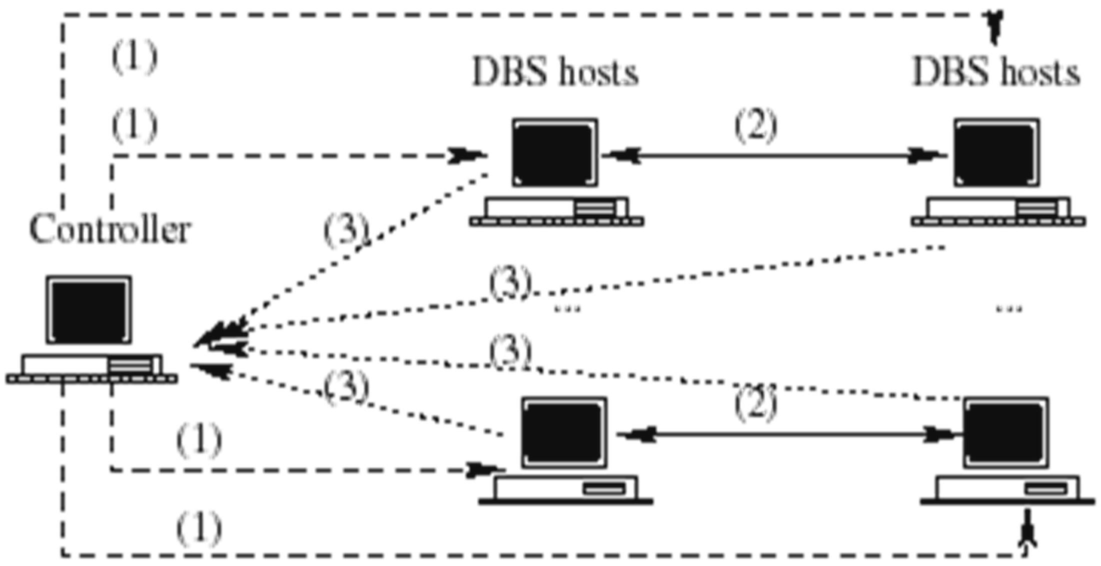
One host serves as the controller, running dbsc, and all other hosts are DBS hosts, running dbsd.

The controller reads the command file and send instructions to all DBS hosts.

TCP connections will be set up between the hosts and traffic is transmitted on them.

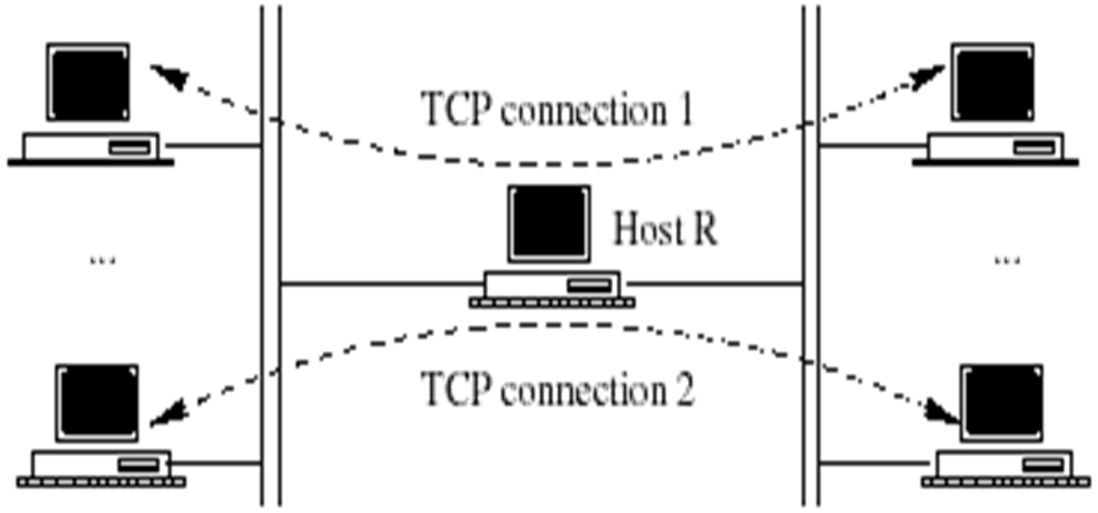
When the data transmissions are over, the dbsc collects statistics.

Using dbs_view to plot the collected statistics.



NIST Net

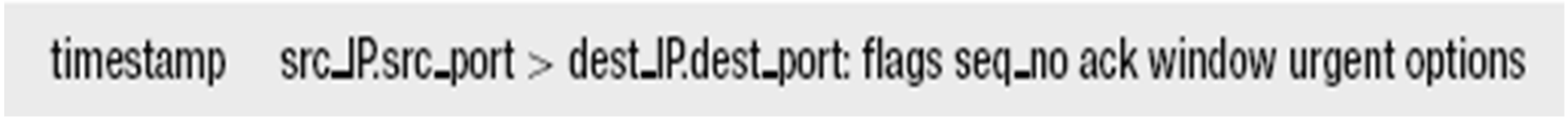
- A Linux-based network emulator that allows a single Linux PC set as a router to emulate various network conditions:
  - Packet loss, duplication, delay and jitter, bandwidth limitations, network congestion



- The host running NIST Net serves as a router between two subnets.
- NIST Net behaviors like a firewall.
- A user can specify a connection and enforce a policy on it.

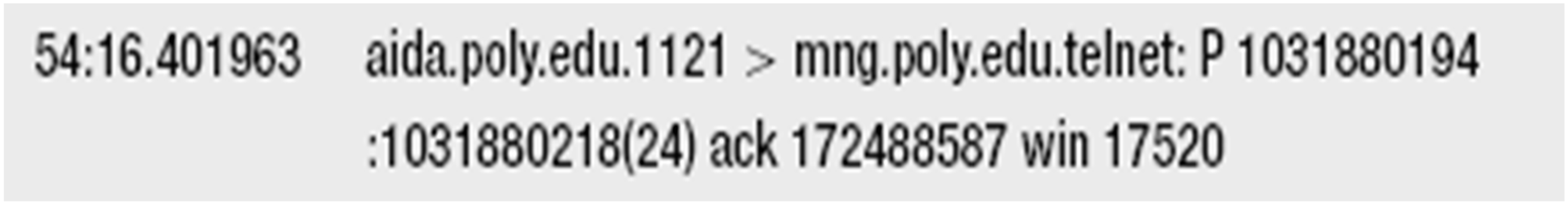
Tcpdump Output of TCP Packets

General format



```
timestamp    src_IP.src_port > dest_IP.dest_port: flags seq_no ack window urgent options
```

An example

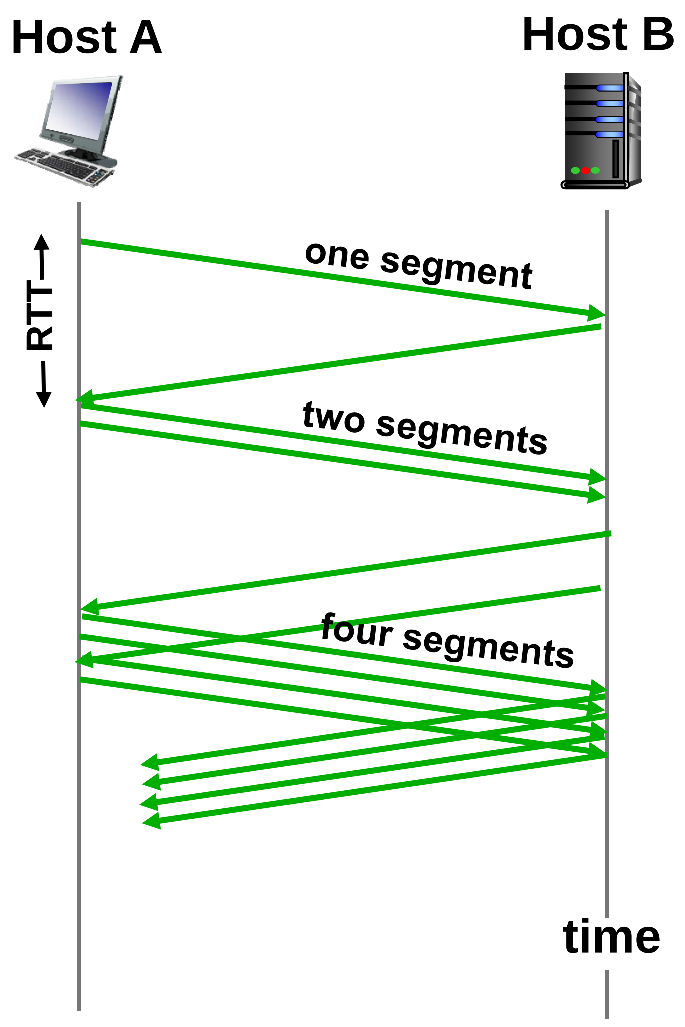


```
54:16.401963    aida.poly.edu.1121 > mng.poly.edu.telnet: P 1031880194
                :1031880218(24) ack 172488587 win 17520
```

Backups

TCP Slow Start

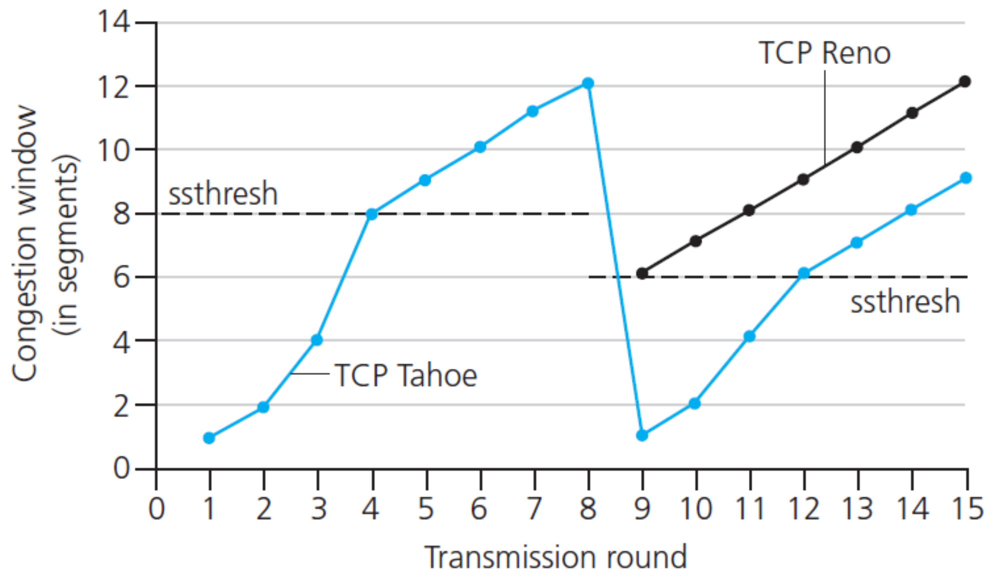
- ❖when connection begins, increase rate exponentially until first loss event:
  - ▪initially cwnd = 1 MSS
  - ▪double cwnd every RTT
  - ▪done by incrementing cwnd for every ACK received
- ❖*summary:* initial rate is slow but ramps up exponentially fast



TCP: switching from slow start to CA

Q: when should the exponential increase switch to linear?

A: when cwnd gets to 1/2 of its value before timeout.

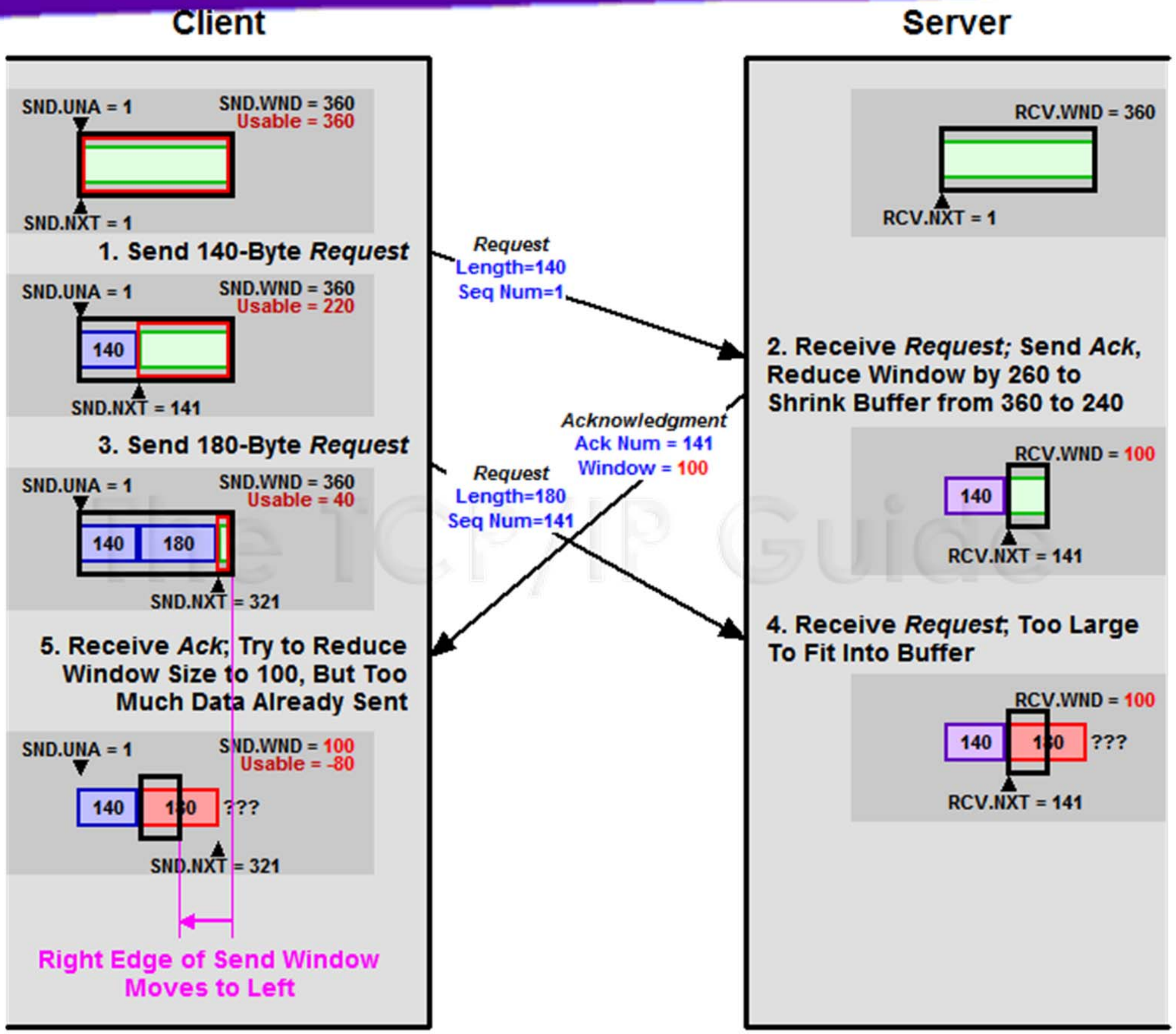


Implementation:

variable `ssthresh`

on loss event, `ssthresh` is set to 1/2 of cwnd just before loss event

TCP Window Shrinking


Client
Server
SND.UNA = 1
SND.WND = 360
Usable = 360
SND.NXT = 1
RCV.WND = 360
RCV.NXT = 1
1. Send 140-Byte Request
Request
Length=140
Seq Num=1
SND.UNA = 1
SND.WND = 360
Usable = 220
140
SND.NXT = 141
2. Receive Request; Send Ack, Reduce Window by 260 to Shrink Buffer from 360 to 240
Acknowledgment
Ack Num = 141
Window = 100
3. Send 180-Byte Request
Request
Length=180
Seq Num=141
SND.UNA = 1
SND.WND = 360
Usable = 40
140
180
SND.NXT = 321
RCV.WND = 100
140
RCV.NXT = 141
4. Receive Request; Too Large To Fit Into Buffer
RCV.WND = 100
140
180
???
RCV.NXT = 141
5. Receive Ack; Try to Reduce Window Size to 100, But Too Much Data Already Sent
SND.UNA = 1
SND.WND = 100
Usable = -80
140
180
???
SND.NXT = 321
Right Edge of Send Window Moves to Left