CEN445 – Network Protocols and Algorithms

Chapter 6 – Transport Layer

6.4 Internet Transport Protocols: TCP

Dr. Mostafa Hassan Dahshan
Department of Computer Engineering
College of Computer and Information Sciences
King Saud University
mdahshan@ksu.edu.sa
http://faculty.ksu.edu.sa/mdahshan

Introduction to TCP

- Designed to provide reliable end-to-end byte stream over unreliable internetwork
- Internetwork different from single network
  - topologies, bandwidths, delays, packet sizes, …
- TCP was designed to
  - dynamically adapts to properties of internet
  - be robust in face of many types of failures
- Def in RFC 793, many improv were added

Introduction to TCP

- Supported by many operating systems as
  - library proc, user process, part of kernel
- Breaks data stream into **segments**
- Maximum segment size is 64 KB
- Often 1460 to fit in single IP Ethernet frame
- Handles IP unreliability
  - timeout and retransmission
  - reassemble packets arriving out of order

TCP Service Model

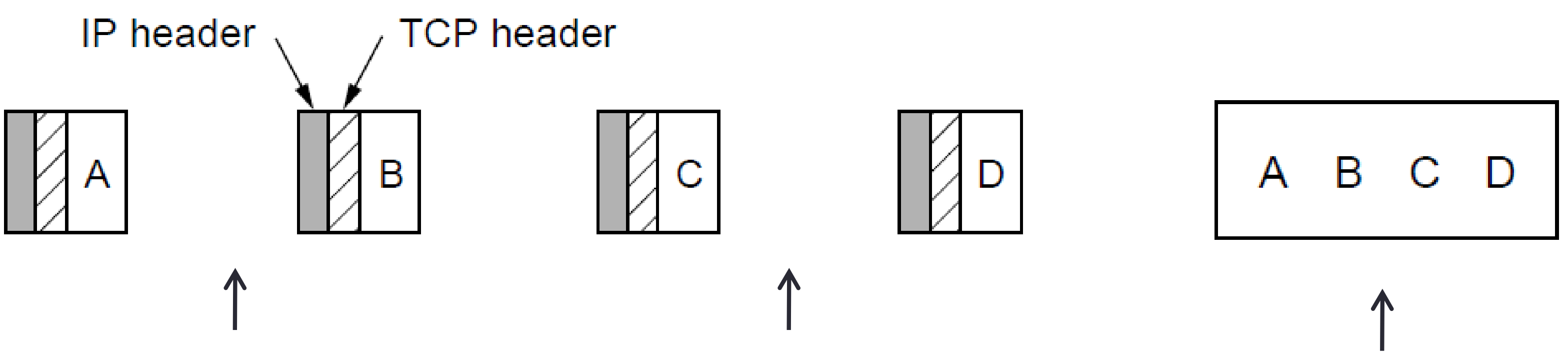
- Sender, receiver create end points: **sockets**
- Socket number: IP address, **port**: 16-bit #
- Socket can be used for multiple connections
- < 1024, **well known ports**, for std services
- 1024 .. 49151 can be registered with IANA
- Apps can choose their own ports
- Each server can listen @ its port at boot
- Better: single daemon @ all ports: inetd

TCP Service Model

Port	Protocol	Use
20, 21	FTP	File transfer
22	SSH	Remote login, replacement for Telnet
25	SMTP	Email
80	HTTP	World Wide Web
110	POP-3	Remote email access
143	IMAP	Remote email access
443	HTTPS	Secure Web (HTTP over SSL/TLS)
543	RTSP	Media player control
631	IPP	Printer sharing

TCP Service Model

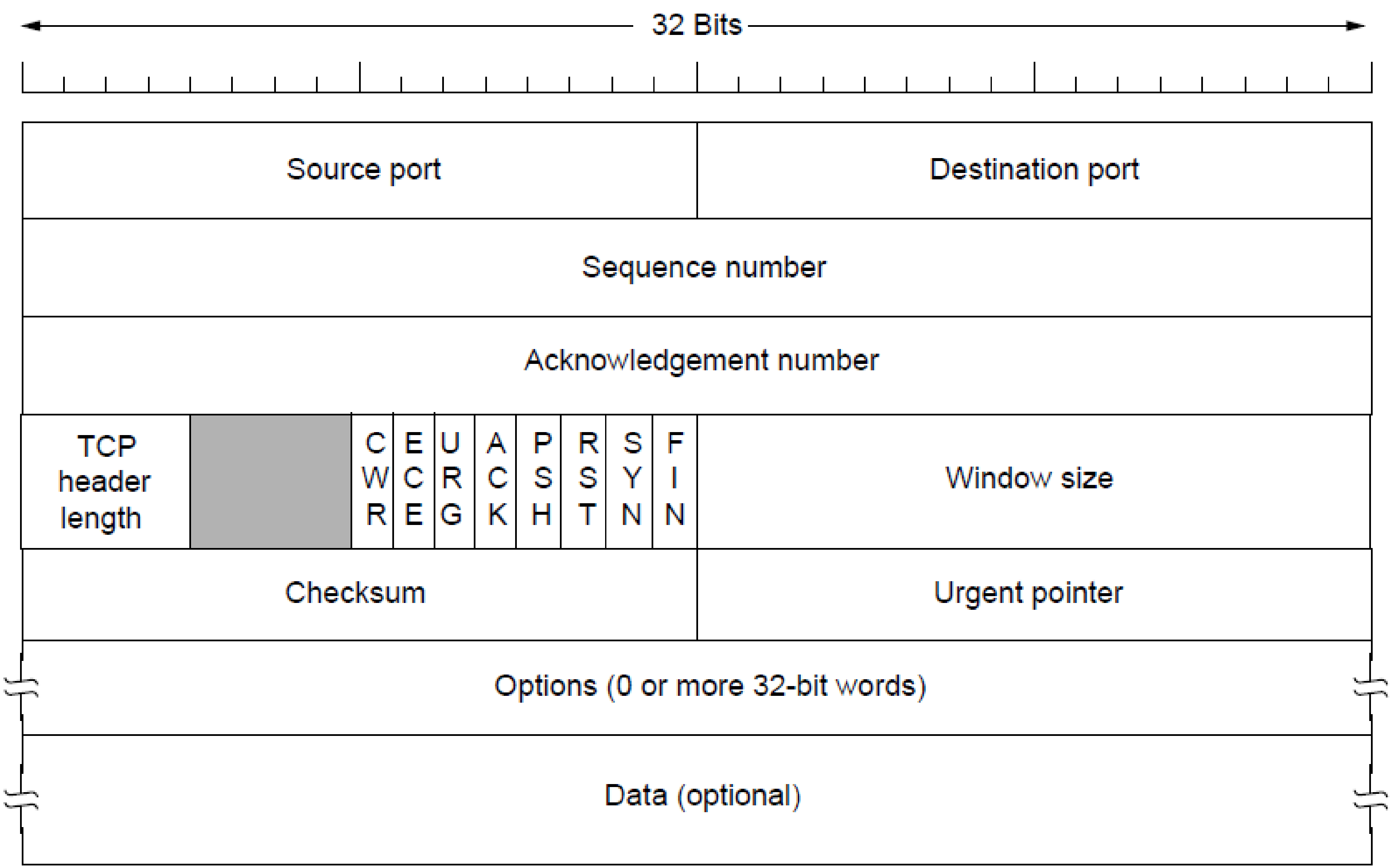
- Connection is full duplex, byte stream
- Message boundaries not reserved
- Send 1024, may receive 512, 512 or vice vera



Four segments, each with 512 bytes of data and carried in an IP packet

2048 bytes of data delivered to application in a single READ call

TCP Segment Header



TCP Segment Header

- Fixed 20 bytes, options: 0-40 bytes
- Data: 65535-20-20 = 65495 B may follow
- Src, dst ports: 16 bits ident conn endpoints
- Connection identifier: 5-tuple
  - protocol (TCP)
  - source and destination IP addresses
  - source and destination port numbers

TCP Segment Header

- Sequence number: # of 1^{st} byte in segment
- Ack number: next in-order byte expected
- Header length: 32-bit words (options var)
- ECE: tell sender to slow down
- CWR: tell receiver Congestion Win Reduced
- URG: set to 1 when Urgent Pointer is used
- Urgent pointer: offset of which urgent data start from current sequence number

TCP Segment Header

- ACK: indicate Ack number is valid
- PSH: request rcvr to push data, not buffer
- RST: reset connection
  - host crash, invalid segment, refuse connection
- SYN: used in establishing connection
- FIN: release connection, no more data
- Window size:
  - how many bytes may be sent after ack number
  - 0: ack#-1 rcvd, no more data please

TCP Segment Header

- Checksum: same as UDP
- Options: add extra facilities
  - MSS: max segment size willing to accept
  - window scale: win size factor
    - shift win size up to 14 bits
    - allow windows up to 2^{30} bytes (2^{14+16})
  - timestamp: sent by sender, echoed by receiver
    - used compute round-trip, estimate lost packet
    - extend seq # in fast links may wrap
    - PAWS: Protect Against Wrapped Seq numbers
  - SACK: Selective ACK, ranges to retransmit

Example – Window Scale

- OC-12 link (600 Mbps), 50 ms prop delay
  - What is the link utilization?
  - Answer:
    - time to transmit 64 KB = $\frac{64\times8\times2^{10}}{600\times2^{20}} = 0.83 \approx 1\text{ms}$
    - ACK arrives after 50 ms
    - total time = 50 + 1 = 51 ms
    - link idle for $\frac{prop}{prop+trans} = \frac{50}{50+1} \approx 98\%$ of the time
    - utilization $= \frac{trans}{prop+trans} = \frac{1}{50+1} \approx 2\%$ of the time

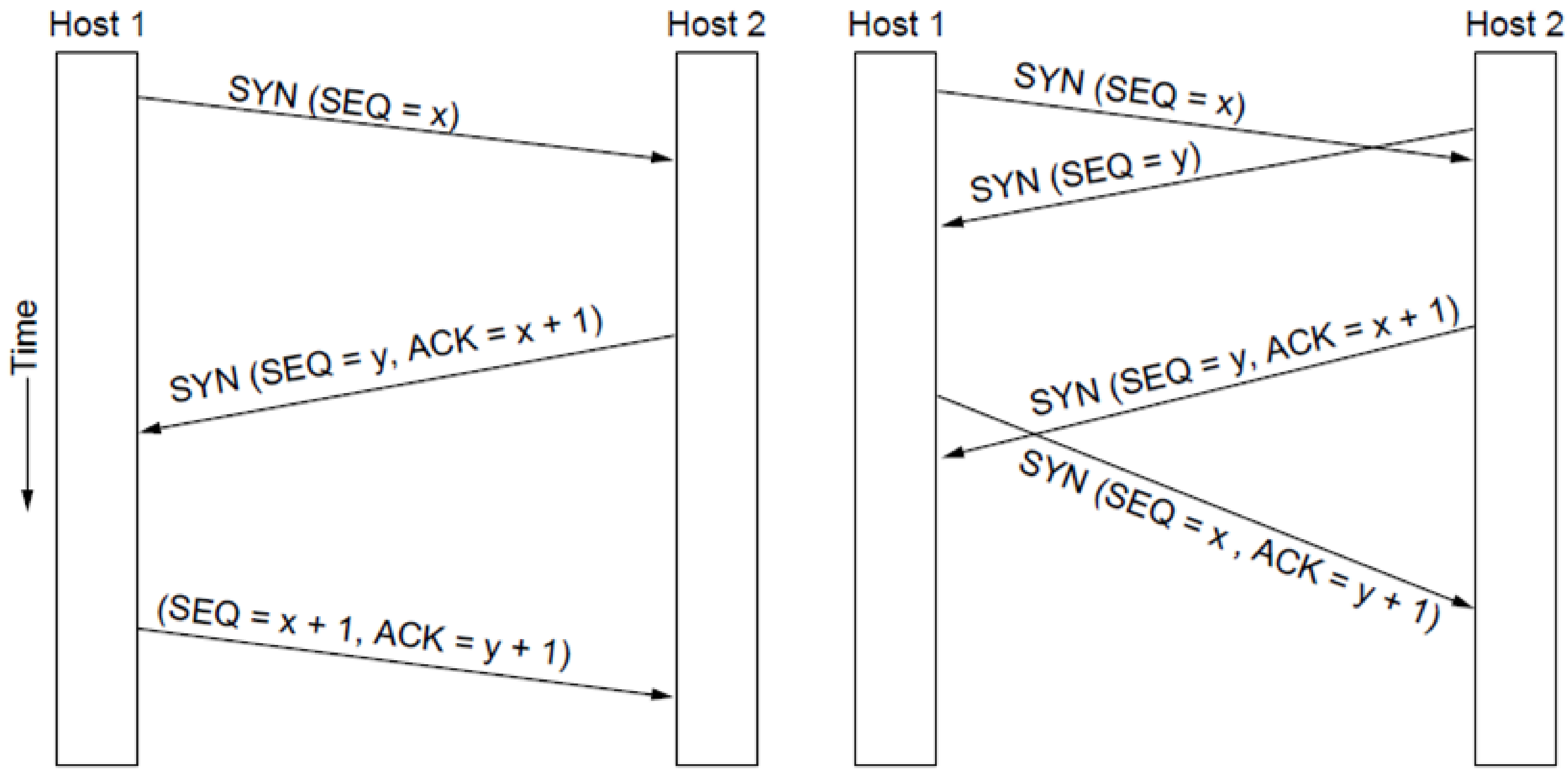
Example – Window Scale

- OC-12 link (600 Mbps), 50 ms prop delay
  - What value of window scale to allow 10% utilization?
  - Answer:
    - utilization $= \frac{trans}{prop+trans} \rightarrow 0.1 = \frac{trans}{50+trans}$
    - $trans = 5.56$ ms
    - window size $= \frac{5.56}{1000} \times \frac{600}{8} \times 2^{20}$ =437257 bytes
    - bits required = $\lceil \log_2(437257) \rceil$ =19
    - Window scale (shifted bits) = 19 – 16 = 3

TCP Connection Establishment

- Use three-way handshake
- Server executes LISTEN, ACCEPT
- Client exec CONNECT, sends [SYN=1, ACK=0]
- Client specifies IP, port, max segment size
- Server: checks for process listening on port
  - no? reply with [RST=1]
  - yes? reply with [SYN=1, ACK=1]
- Initial seq # cycle slowly, not start @ 0
  - protect against delayed duplicate, clock based

TCP Connection Establishment



Normal case

Simultaneous connect

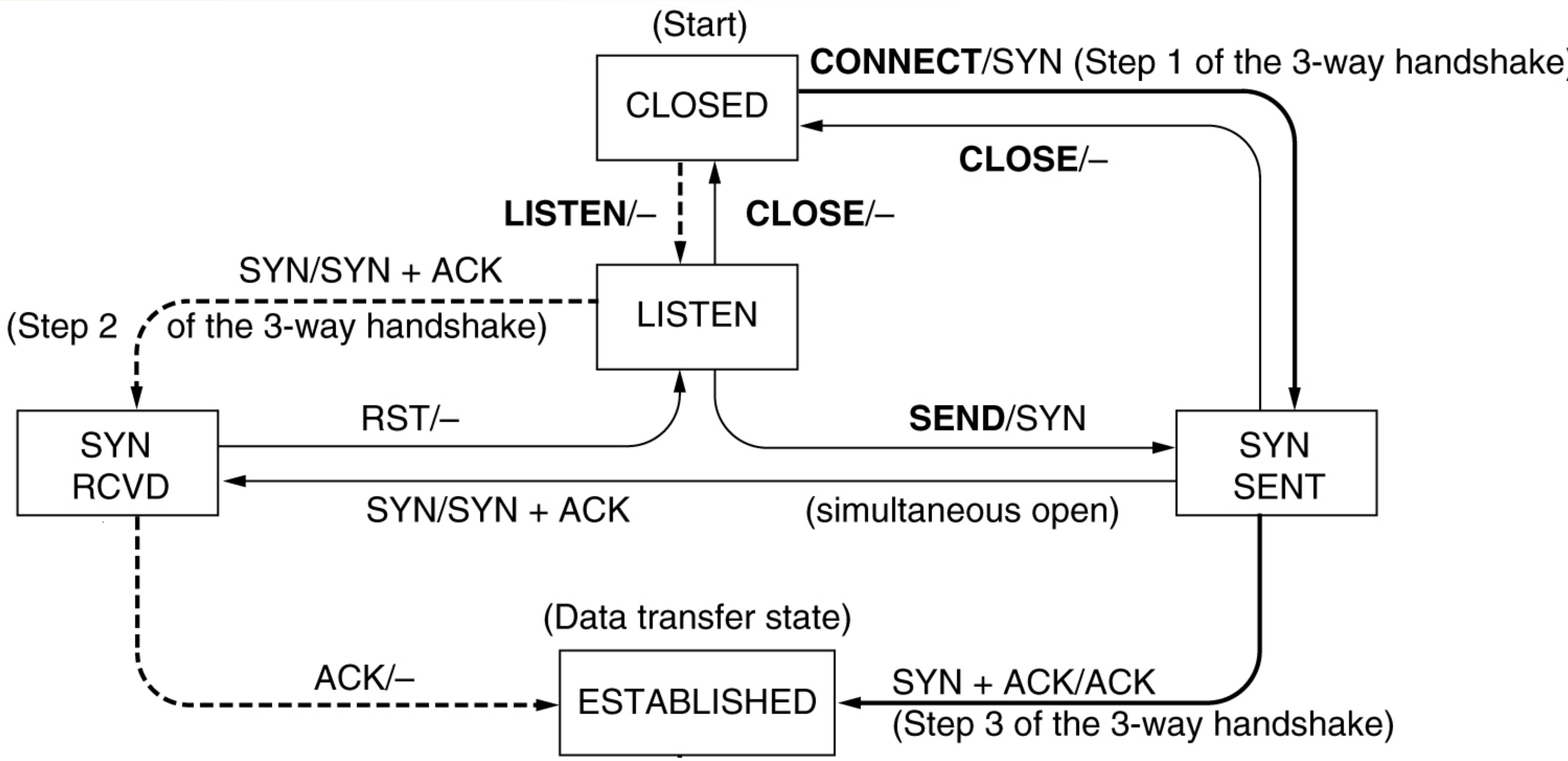
TCP Connection Release

- Each direction is released independently
- Send TCP segment with [FIN=1]
- When ack received, that direction shutdown
- Other side can sends ACK
- Possible to send FIN with ACK in 1 segment
- To avoid two-army problem
  - if no ACK received? sender times out, release
  - other side eventually time out as well

TCP Connection Management Modeling

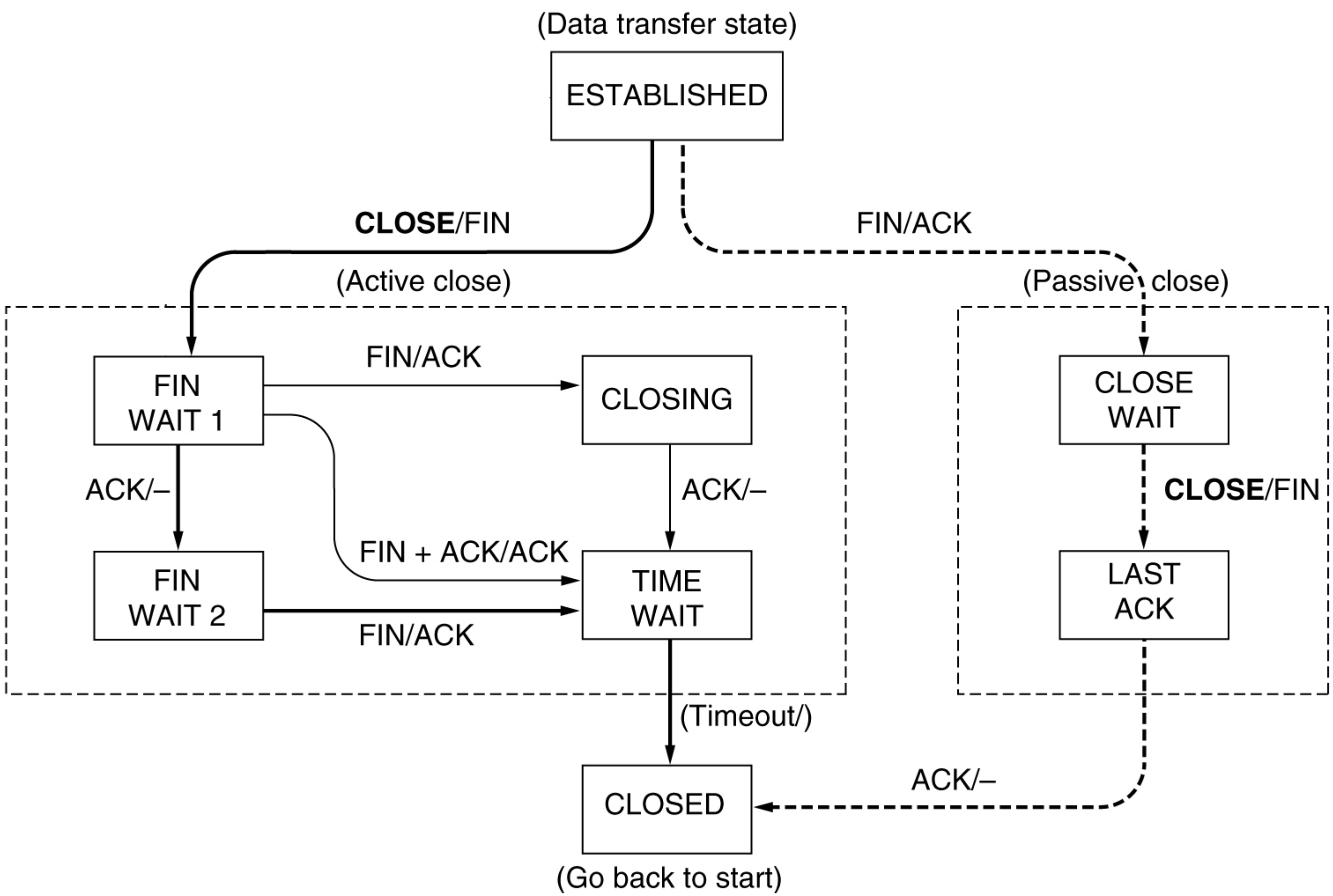
State	Description
CLOSED	No connection is active or pending
LISTEN	The server is waiting for an incoming call
SYN RCVD	A connection request has arrived; wait for ACK
SYN SENT	The application has started to open a connection
ESTABLISHED	The normal data transfer state
FIN WAIT 1	The application has said it is finished
FIN WAIT 2	The other side has agreed to release
TIME WAIT	Wait for all packets to die off
CLOSING	Both sides have tried to close simultaneously
CLOSE WAIT	The other side has initiated a release
LAST ACK	Wait for all packets to die off

TCP Connection Management Modeling



- Heavy solid line: normal path for a client
- Heavy dashed line: normal path for a server
- Light lines: unusual events
- Transition labeled: causing event/ resulting event

TCP Connection Management Modeling



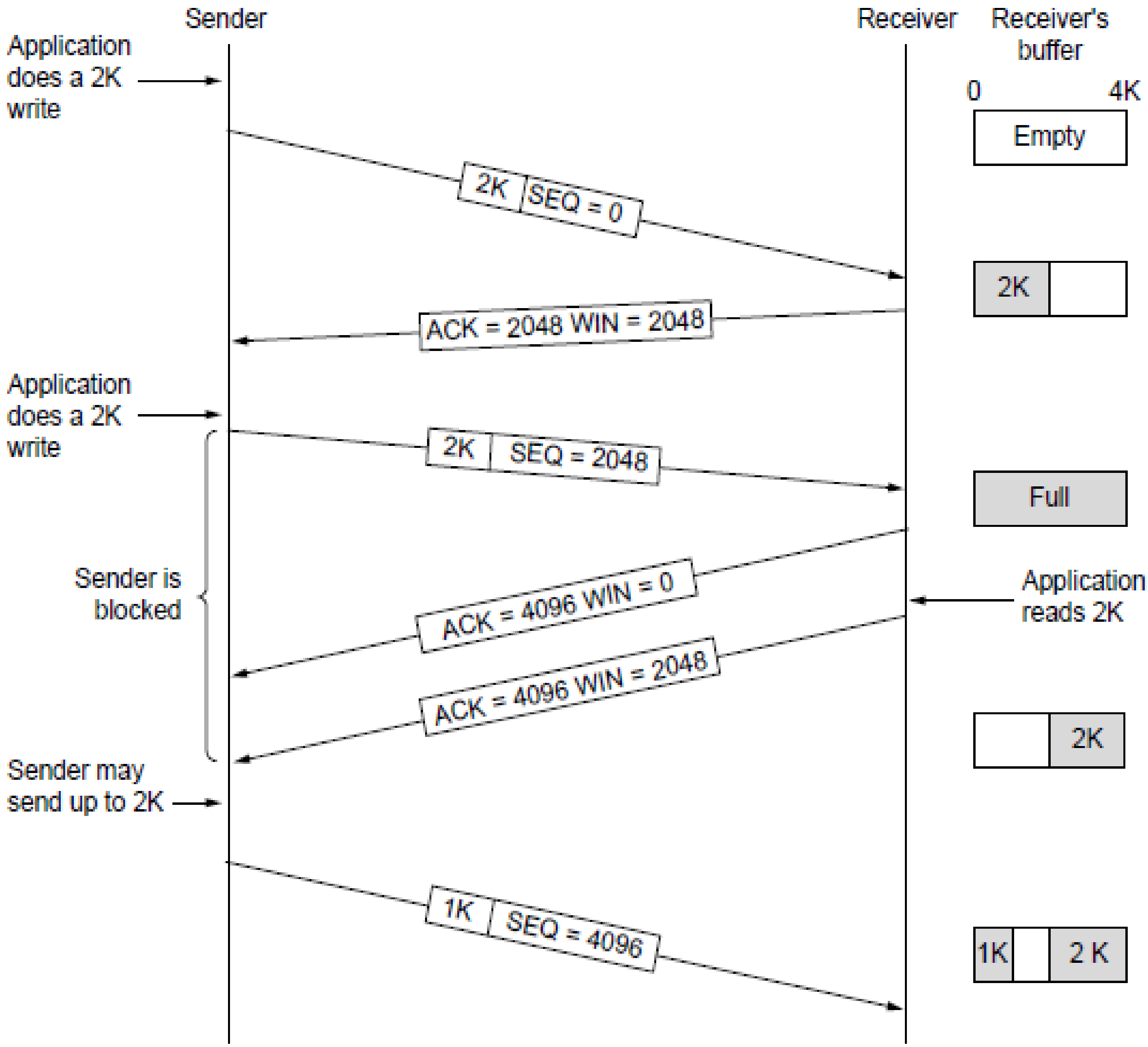
TCP Sliding Window

- TCP uses credit-based flow control
- Window management not tied to ack
- Remember, transport layer buffers data
- Receiver changes window according to buffer consumption by application

TCP Sliding Window

Example

- Receiver has a 4096-byte buffer
- Sender transmit 2048-b seg; correctly rcvd
- Receiver acknowledges segment
- But, app hasn't removed data from buffer
- Receiver advertise window 2048
- Starting at next byte expected
- Sender tr 2048 B; but receiver adv win=0



ACK + WIN is the sender's limit

TCP Sliding Window

- When win=0, sender can't send, except:
  - urgent data, to allow kill process remotely
  - 1-byte seg to force receiver re-announce win size (window probe), to prevent deadlock
- Sender can buffer data before sending
- Receiver can wait before acknowledging
- Can use flexibility to improve performance

TCP Sliding Window

Example – interactive telnet/SSH application

- User A types 1 character
- TCP A sends 41-byte segment
- TCP B sends 40-byte ACK
- Editor B echoes 1 character
- TCP B sends 41-byte segment
- TCP A sends 40-byte ACK
- Total of 162 bytes used for each char typed!

TCP Sliding Window

- To optimize, delayed acknowledgements
- Delay ACK, Win update for up to 500 ms
- More data may arrive
- If terminal echoes within 500 ms,
  - only 41-byte (ACK + data) are sent
  - total bytes 82; half bandwidth is saved

Think of similar examples

TCP Sliding Window

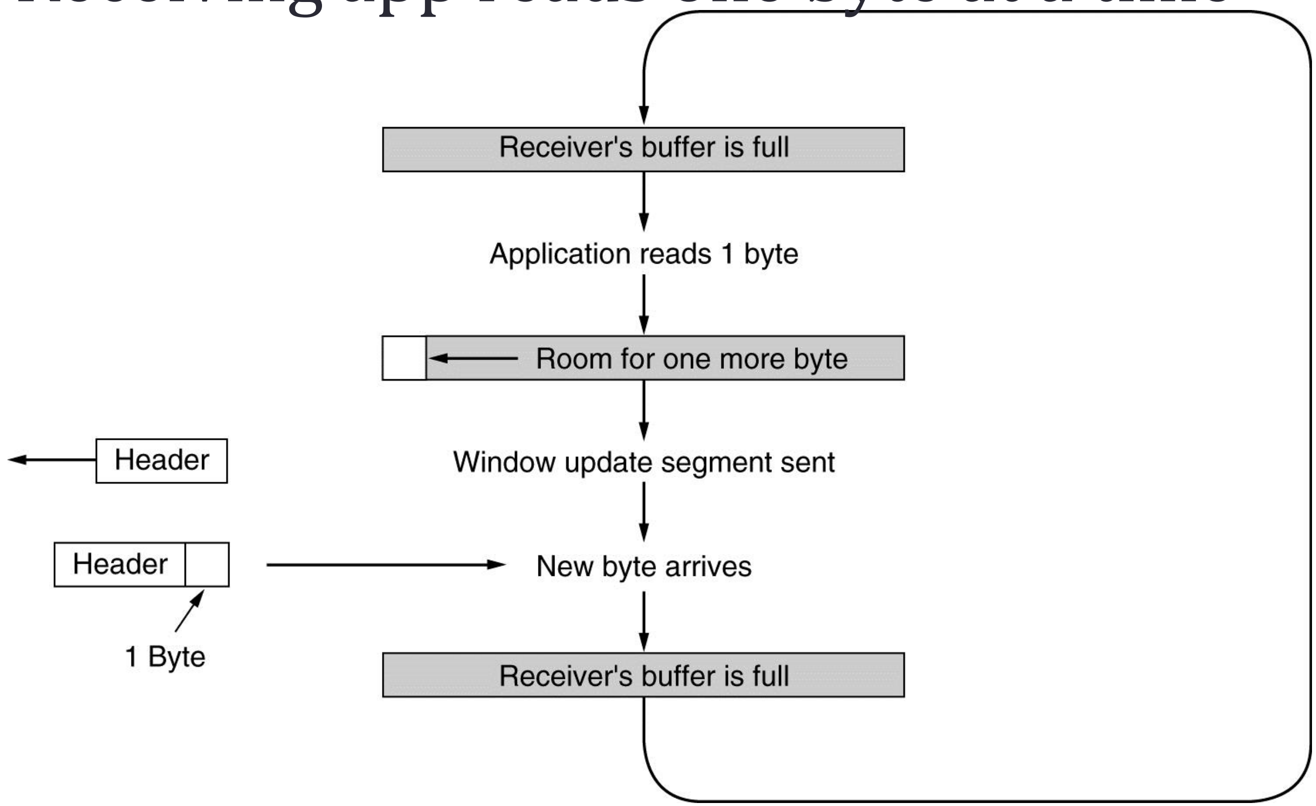
Nagle's Algorithm

- Send one byte; Buffer the rest, wait for ACK
- Send remaining bytes in one segment
- Buffer until ACK is received
- Good for interactive typing on a terminal
- Problems
  - not good for interactive games
  - can cause deadlock: app waiting for data

TCP Sliding Window

Silly window syndrome

- Sending TCP sends large blocks
- Receiving app reads one byte at a time



TCP Sliding Window

- Clark's solution to silly window syndrome
  - receiver should wait before sending updates
  - wait until more window space is available
  - sender should not send tiny segments
  - at least half receiver's buffer size
- Nagle, Clark solutions complement
- Segment can arrive out of order
- Can discard, but waste bandwidth
- ACK not sent until up to ACKed byte arrives

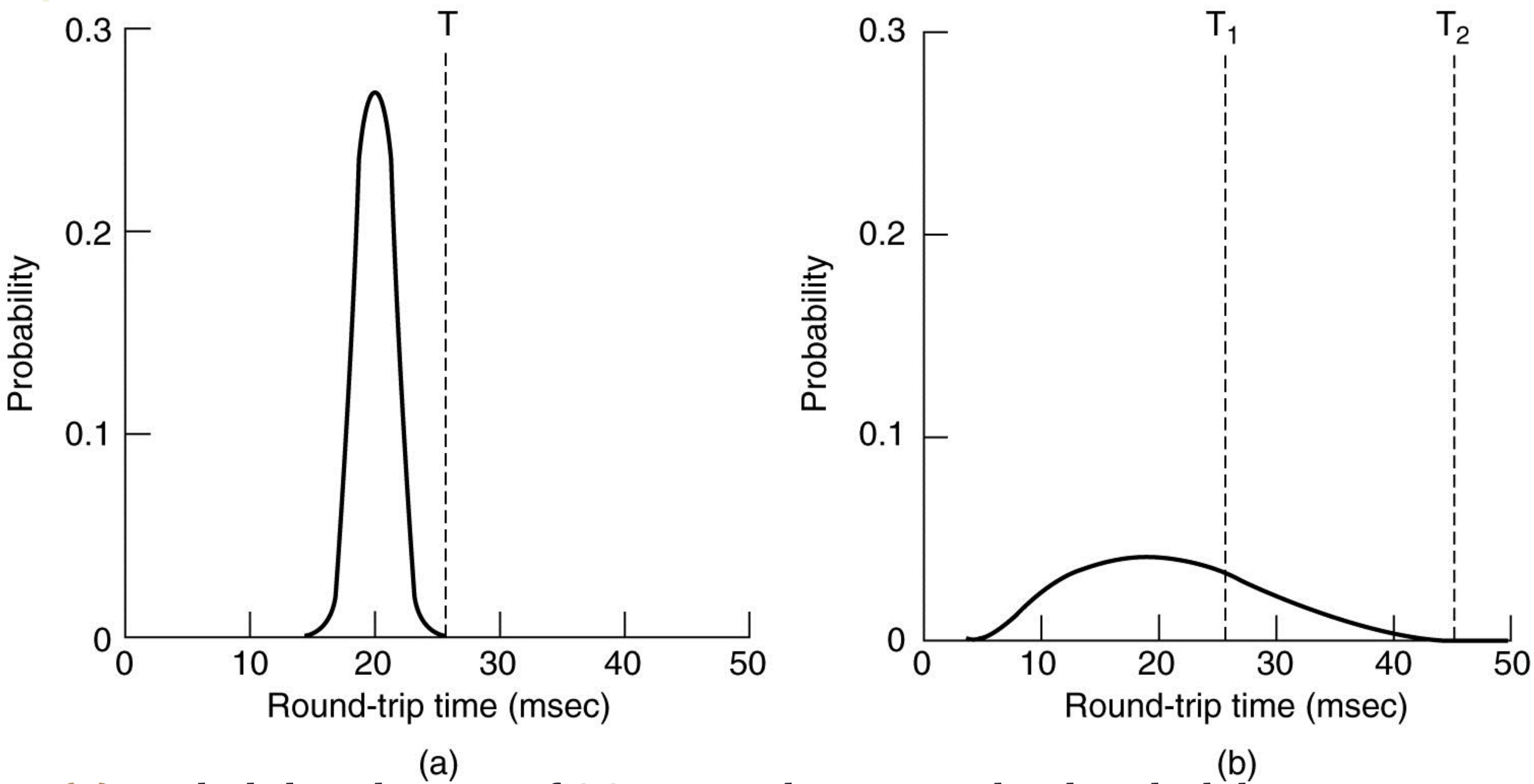
TCP Timer Management

- TCP uses multiple timers

Most imp: RTO (**R**etransmission **T**ime**O**ut)

  - start when segment is sent
  - stopped when ACK is received
- Determining timeout interval is more difficult in TCP than in data link layer
  - too small, unnecessary retransmissions
  - too long, long delay, performance suffers
- Solution: dynamic timeout

TCP Timer Management



(a) Probability density of ACK arrival times in the data link layer

(b) Probability density of ACK arrival times for TCP

TCP Timer Management

Jacobson's Algorithm

- Dynamically adjust timeout interval
- Maintain RTT for each connection
- Best current estimate for round-trip time
- If ACK takes R sec < timer expiration,

  $$SRTT = \alpha\ SRTT + (1- \alpha)\ R$$

- α is a smoothing factor
- Typically $\alpha = 7/8$

Example

If the TCP round-trip time, *RTT*, is currently 30 msec and the following acknowledgements come in after 26, 32, and 24 msec, respectively, what is the new *RTT* estimate using the Jacobson algorithm? Use $\alpha = 0.9$.

Solution

$$SRTT = \alpha\, SRTT + (1 - \alpha)\, R$$

$$\mathrm{SRTT}_1 = 0.9 \times 30 + (1 - 0.9) \times 26 = 29.6$$

$$\mathrm{SRTT}_2 = 0.9 \times 29.6 + (1 - 0.9) \times 32 = 29.84$$

$$\mathrm{SRTT}_3 = 0.9 \times 29.84 + (1 - 0.9) \times 24 = 29.256$$

TCP Timer Management

- Even w good $SRTT$, difficult to choose RTO
- Initial implementation used $2xRTT$
- Inflexible to response to large variance
- Delay becomes large when load $\approx$ capacity
- May retransmit when packet still in transit
- To fix this, make RTO sensitive to variance
  - $RTTVAR = \beta RTTVAR + (1 - \beta)|SRTT - R|$
  - $RTO = SRTT + 4 \times RTTVAR$
- Typically $\beta = 3/4$

TCP Timer Management

- When segment retransmitted, ACK arrives
- Is ACK for first or second segment?
- Wrong guess contaminate *SRTT*

Karn's Algorithm

- When packet is retransmitted,
  - don't update *SRTT*
  - double timeout value on each retransmission
- Used in most TCP implementation

TCP Timer Management

Persistence timer

- Prevent deadlock
  - receiver sends ACK, WIN=0, tell sender to wait
  - later receiver updates WIN, but update is lost
  - sender, receiver waiting for each other
- When timer goes off
  - sender transmits probe to receiver
  - receiver responds with window size
  - if WIN still 0, timer is set again and cycle repeat

TCP Timer Management

Keepalive timer

- When connection is idle for long time
- Check whether other side is still there
- No response? connection terminated

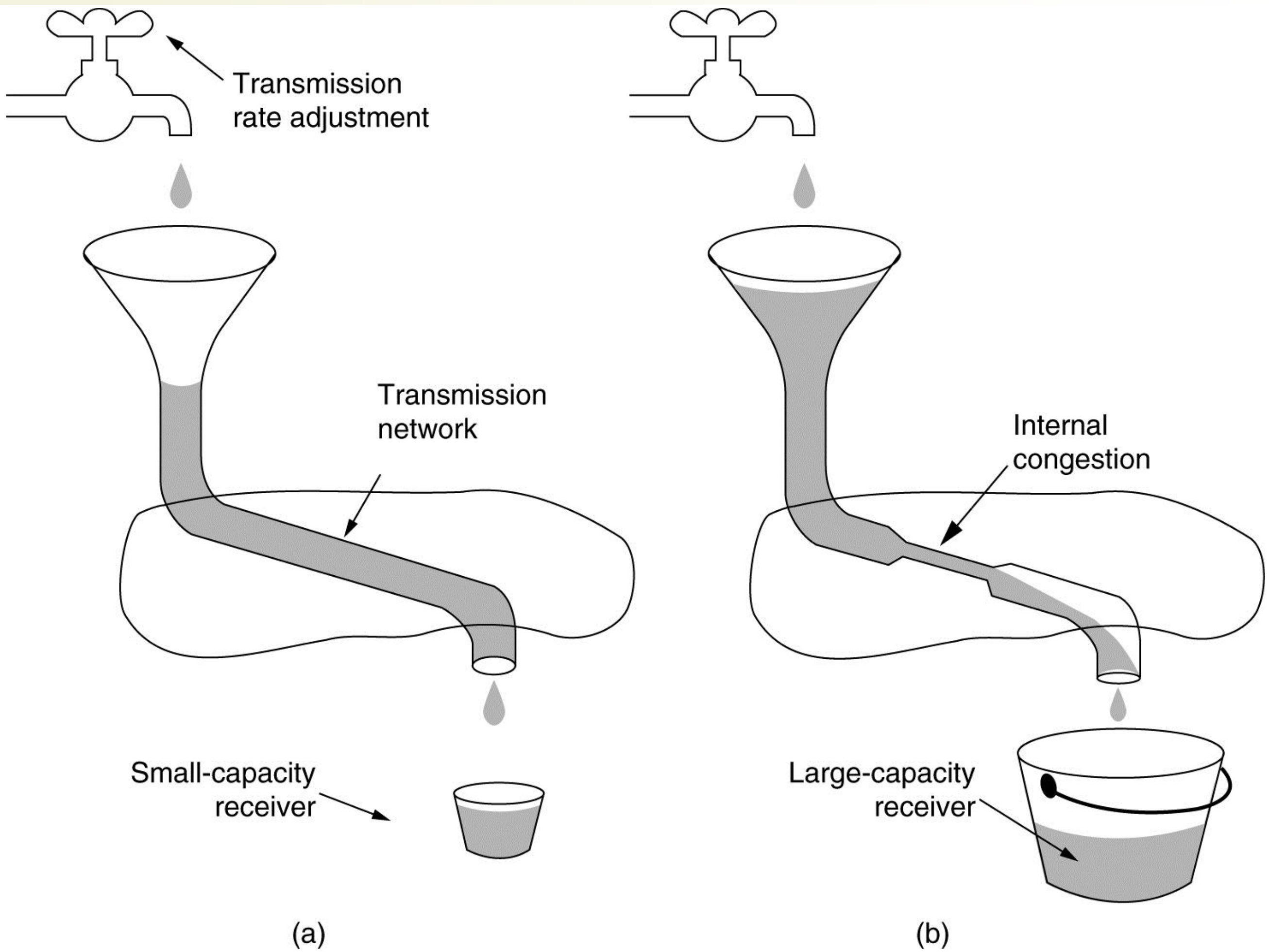
TIME WAIT state

- when closing connection
- wait twice max packet lifetime
  - make sure that when connection closed,
  - all packets created by it have died off

TCP Congestion Control

- Congestion control is a key function of TCP
- Congestion: offered load > network ability
- NL tries to manage; if it can't, drop
- TL receives feedback, slow down: TCP
- Additive Increase Multiplicative Decrease
- TCP implements AMID using: window, loss
- Maintains congestion window, in addition to flow control window

TCP Congestion Control



(a) A fast network feeding a low capacity receiver

(b) A slow network feeding a high-capacity receiver

TCP Congestion Control

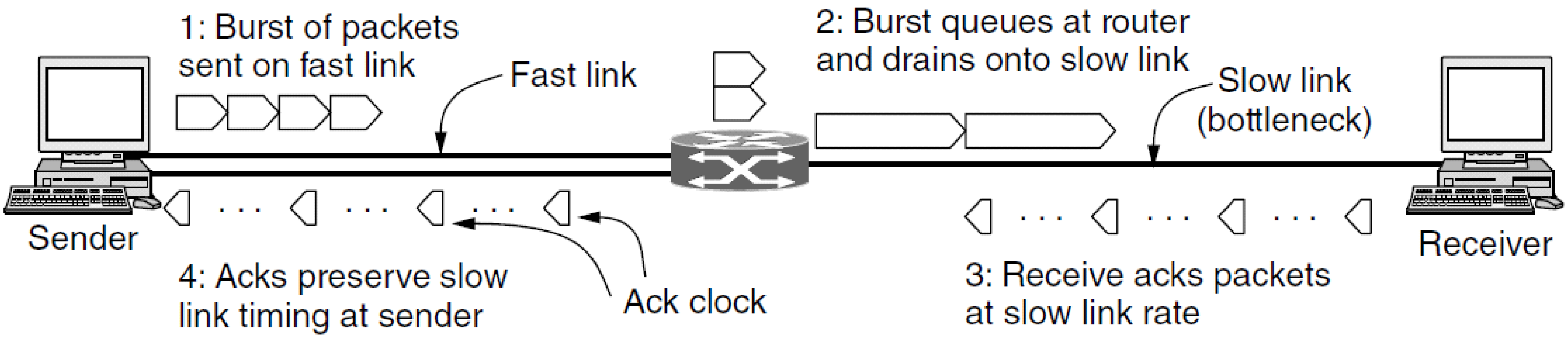
- Two windows maintained in parallel
  - flow control window
  - congestion window
- Effective windows is the smaller of the two
- Example
  - receiver says send 64 KB
  - sender knows > 32 KB can cause congestion
  - sender will send only 32 KB

TCP Congestion Control

- Congestion, noise can cause packet loss
- Loss due to noise is rare in wired medium
- Not the case in wireless links: 802.11
- Wireless include own retrans mechanisms;
- TCP always assume loss is due to congestion

TCP Congestion Control

- Acks return at rate = slowest link along path
- Called: ack clock; used by TCP to smooth traffic
- Sending at this rate avoid unnecessary queues

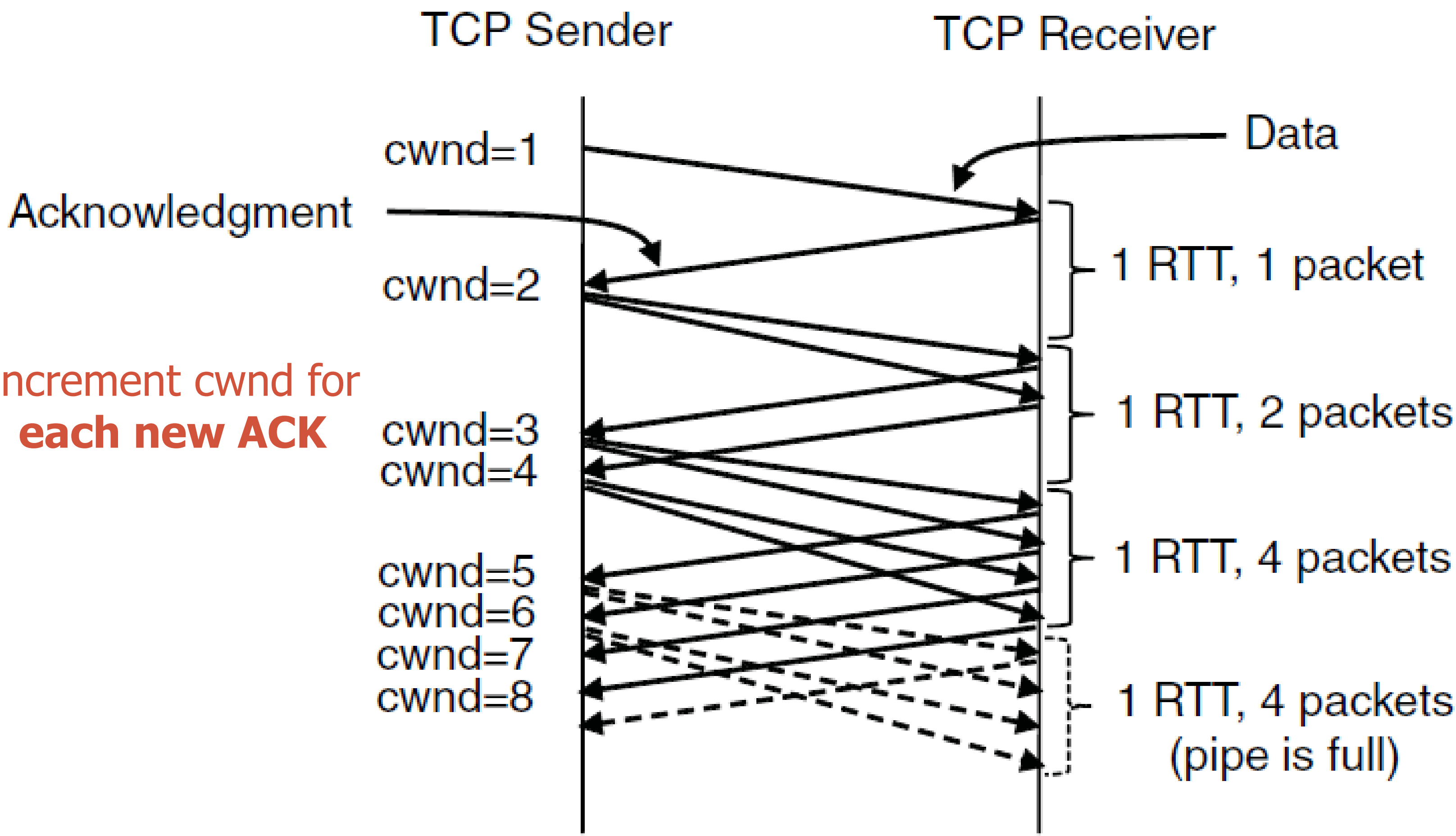


TCP Congestion Control

- AMID take long time to reach good point
- Start w max window? too large for slow link
- Jacobson sol: mix additive, multiplicative inc
- Called slow start (compared to max win)
- First time, send 1 packet (max segment size)
- For each segment ack'ed, send 2 segments
- Congestion window doubles every RTT
- Not slow at all; exponential growth

TCP Congestion Control

Slow start grows congestion window exponentially



Increment cwnd for
each new ACK

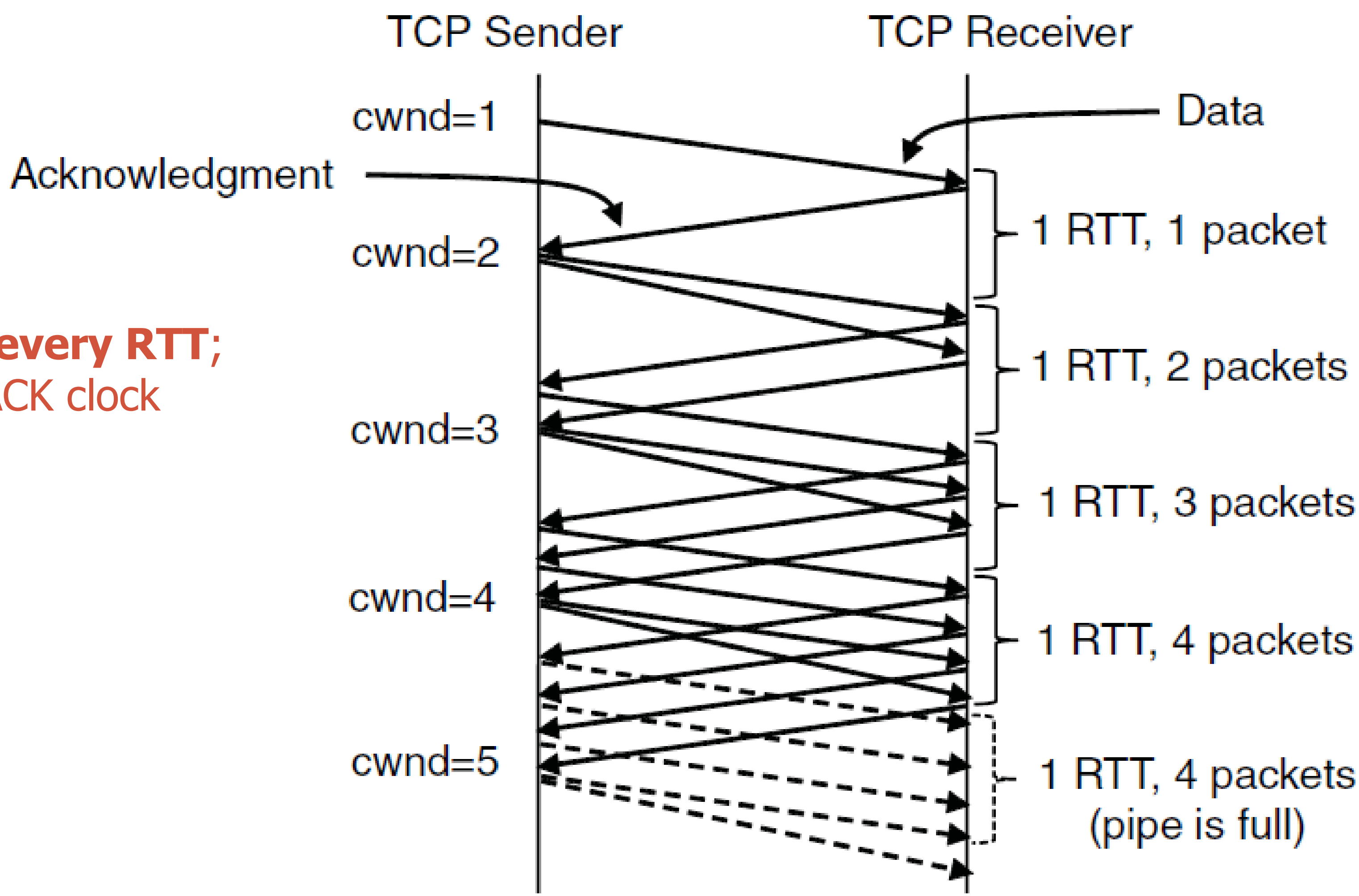
TCP Congestion Control

- To keep under control, use threshold
- Initially set to flow control window
- Congestion window keeps increasing until
  - timeout occur (packet is lost)
  - congestion window exceeds threshold
  - receiver's window is filled
- If packet loss happens
  - set threshold = ½ previous loss cwnd
  - set cwnd to its initial value
  - restart the slow start process

TCP Congestion Control

- When slow start threshold crossed;
- TCP sw from slow start to additive increase
- cwnd increased by 1 segment every RTT
- As in slow start, usually every ack, not RTT

TCP Congestion Control



Adds 1 **every RTT**;
Keeps ACK clock

TCP Congestion Control

- Defect: waiting for timeout; relatively long
- After packet lost
  - receiver can’t ack past it
  - ack number is fixed
  - sender can’t send new packets
  - long.. until timer fires, lost packet retransmitted
- Quick way to recognize loss: duplicate ack
  - same ack #, likely other pkt arrived, orig lost
  - may have taken diff path: out of order; unlikely

TCP Congestion Control

Fast retransmission

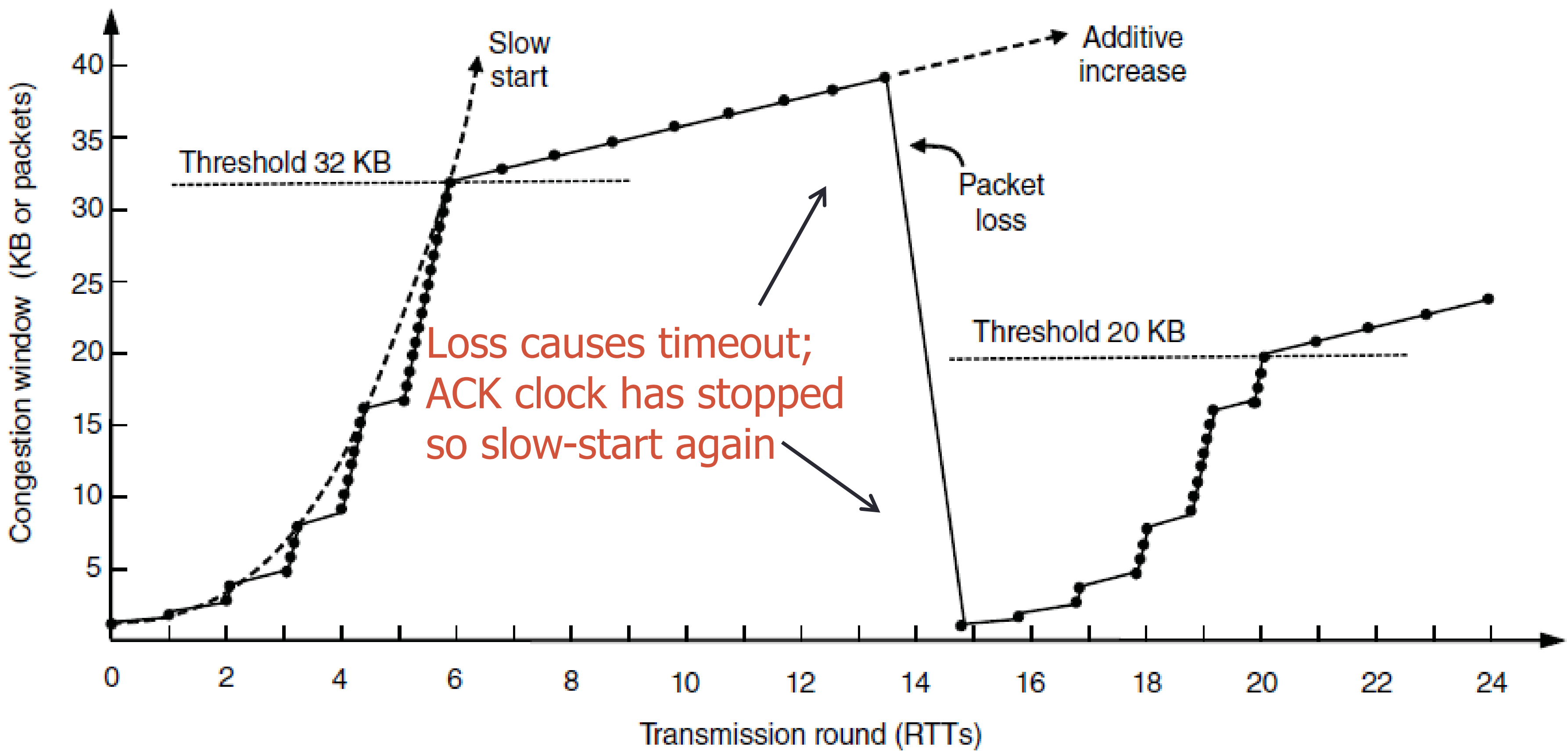
- Assume duplicate ack = packet loss
- After 3 duplicate acks, retrans lost packet
- Set threshold = ½ cwnd; same as w timeout
- Set cwnd = 1 segment
- Send new packet after ack of retransmitted

- Added in TCP Tahoe

TCP Congestion Control

TCP Tahoe

- Threshold is half of previous loss cwnd
- cwnd set to 1 segment



TCP Congestion Control

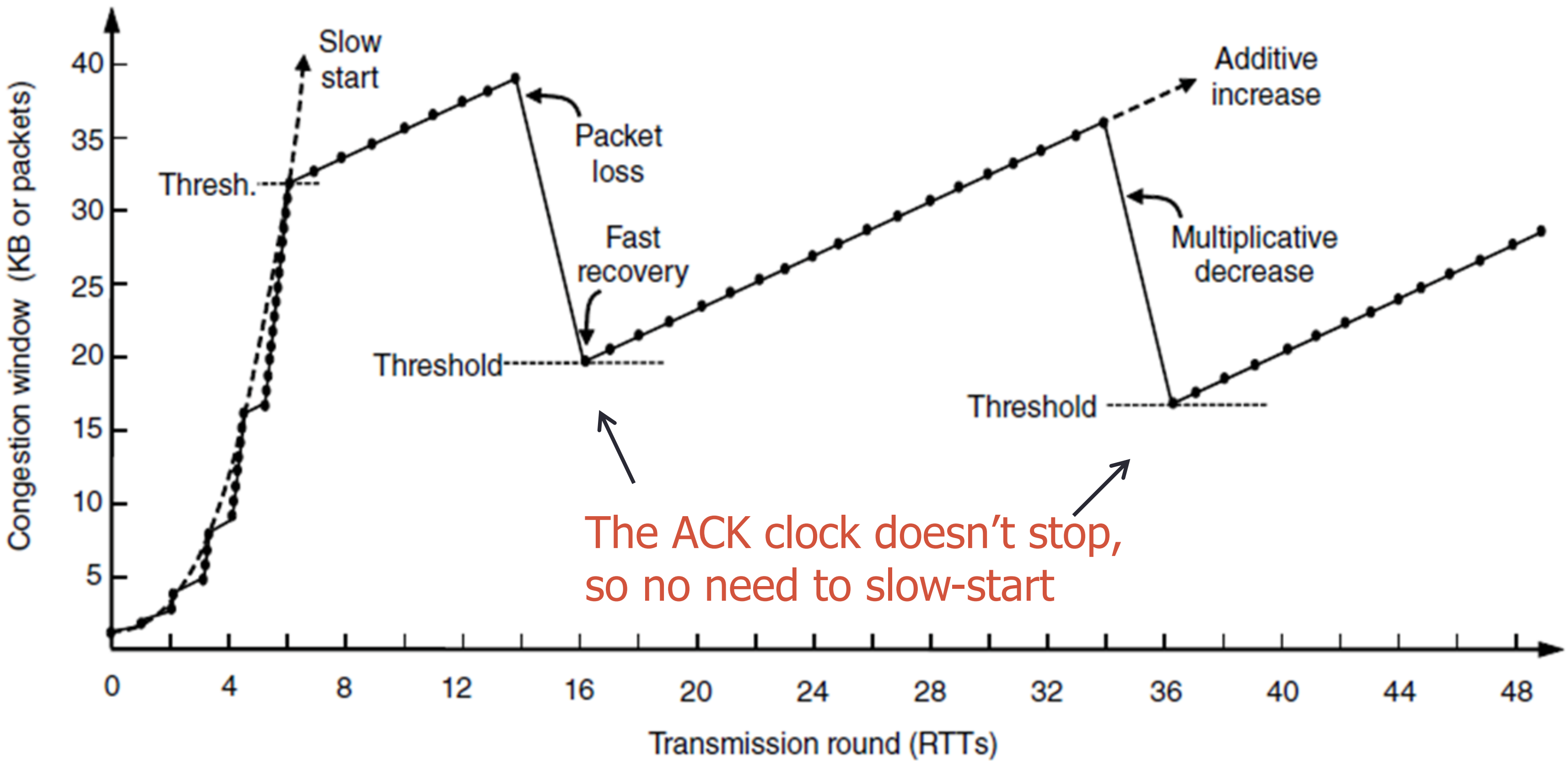
Fast recovery

- Temp mode maintain ack clock
- After fast retrans (3rd duplicate ACKs)
- After 1 RTT, lost packt ACK'ed, FR exited
- Set cwnd = ½ SS threshold (not 1 seg)
- cwnd continue linear increase

TCP Congestion Control

TCP Reno

- SS threshold is half of previous loss cwnd
- cwnd set to threshold



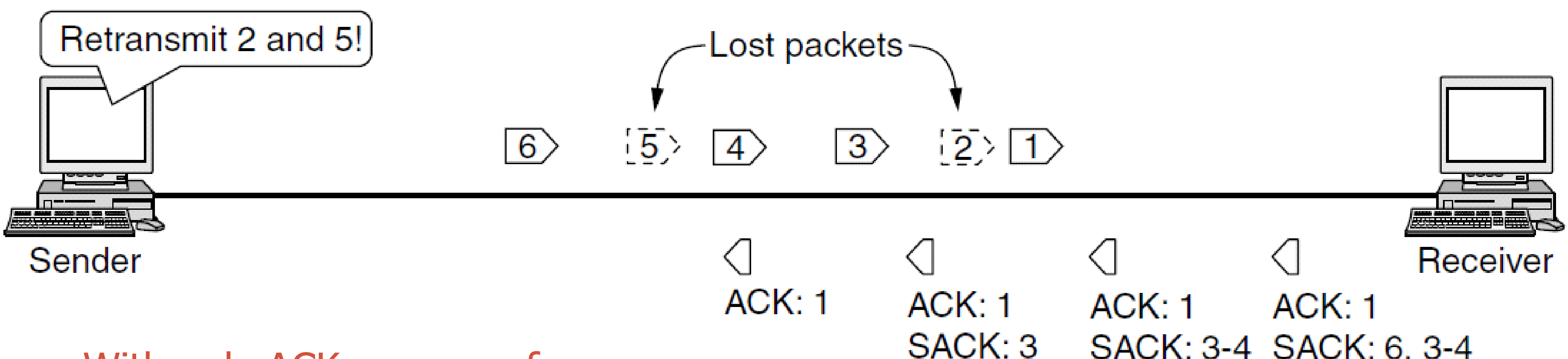
TCP Congestion Control

Name	Mechanism	Purpose
ACK clock	Congestion window (cwnd)	Smooth out packet bursts
Slow-start	Double cwnd each RTT	Rapidly increase send rate to reach roughly the right level
Additive Increase	Increase cwnd by 1 packet each RTT	Slowly increase send rate to probe at about the right level
Fast retransmit / recovery	Resend lost packet after 3 duplicate ACKs; send new packet for each new ACK	Recover from a lost packet without stopping ACK clock

- Other mechanisms
  - selective acknowledgements (SACK)
  - explicit congestion notification (ECN)

TCP Congestion Control

- Cumulative ack doesn't tell which seg lost
- Fix: use selective ack (SACK) option
- Lists up to 3 ranges of bytes received
- More accurate retransmissions



With only ACKs, no way for us to know that 2 and 5 were lost

TCP Congestion Control

- ECN: IP mech to notify hosts of congestion
- During TCP connection establishment, sender, receiver set ECE, CWR bits
- TCP packet flagged in IP header: ECN
- Routers set ECN signal when cong approach
- Instead of dropping packet after congestion
- Receiver sets ECE, sender responds w CWR
- Sender reacts same as packet loss

Example

Consider the effect of using slow start on a line with a 10-msec round-trip time and no congestion. The receive window is 24 KB and the maximum segment size is 2 KB.
How long does it take before the first full window can be sent?

Solution

Starting with window size = 1 segment = 2 KB
At RTT 1 (10 ms), cwnd = 4 KB
At RTT 2 (20 ms), cwnd = 8 KB
At RTT 3 (30 ms), cwnd = 16 KB
At RTT 4 (40 ms), cwnd = 24 (limited by receiver window)

Answer = 40 ms

Example

Consider a TCP connection with 10 ms round-trip time, max segment size = 2 KB, receiver window = 64 KB. Suppose packet #4 timed out and all other transmissions were successful. RTO value is 50 ms. Calculate the time required to reach the receiver window size. Assume RTT is fixed.

Solution

Initially, cwnd = 2KB,
ssthresh = 64 KB

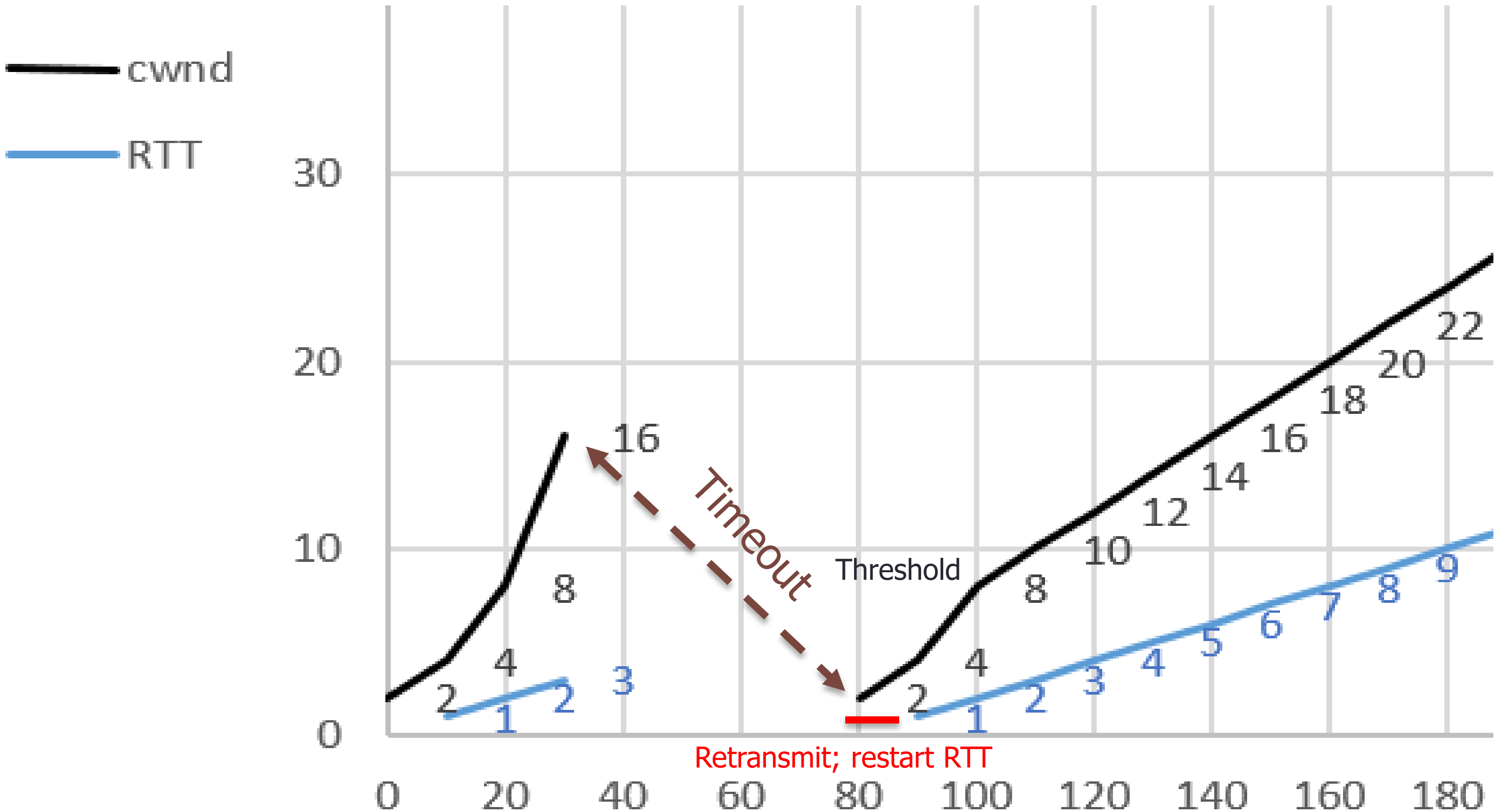
ACK #1 at RTT 1, cwnd = 2*2 = 4 KB,
ACK #2 at RTT 2, cwnd = 4*2 = 8 KB,
ACK #3 at RTT 3, cwnd = 8*2 = 16 KB
#4 is lost at RTT3+RTO = 30 + 50 =80 ms
ssthresh = 16/2 = 8 KB, cwnd = 2 KB
retransmit #4
ack clock restart
ACK #4 at RTT 1, cwnd = 2*2 = 4 KB
ACK #5 at RTT 2, cwnd = 4*2 = 8 KB

Since we reached sstresh, switch to additive increase:

ACK #6 at RTT 3, cwnd = 8+2 = 10 KB
ACK #7 at RTT 4, cwnd = 10+2 =12 KB
ACK #8 at RTT 5, cwnd = 11+2 = 14 KB
..
ACK #x+3 at RTT x, cwnd = (2x+4) KB
..
ACK #33 at RTT 30, cwnd = 64 KB

RTT 30 = 300 + 80 = 380 ms

Example


cwnd
RTT
30
20
10
0
0
20
40
60
80
100
120
140
160
180
16
8
4
2
Timeout
Threshold
2
4
8
10
12
14
16
18
20
22
1
2
3
1
2
3
4
5
6
7
8
9
Retransmit; restart RTT

Example

Suppose that the TCP congestion window is set to 18 KB and a timeout occurs. Assume that the maximum segment size is 1 KB.

a. How big will the window be if the next four transmission bursts are all successful?
b. After another four transmission bursts, how big will the window be?

Solution

a. The next transmission will be 1 maximum segment size. Then 2, 4, and 8. So after four successes, it will be 8 KB.

We started at 1 MSS because this is the first after timeout

b. Note that ssthresh = 18/2 = 9 KB.
After next transmission, cwnd = 8+1 = 9 KB
Then, 10, 11, 12.
So after 4 more bursts, it will be 12 KB.

ssthresh (slow-start threshold) = ½ of the threshold value before the timeout

External References

- TCP Congestion Control, http://web.cs.wpi.edu/~cs3516/b09/slides/tcp-cong-control.ppt