IK1203

Lecture 5: UDP and TCP

Literature:

Forouzan, TCP/IP Protocol Suite: Ch 11-12

UDP

User Datagram Protocol - RFC 768

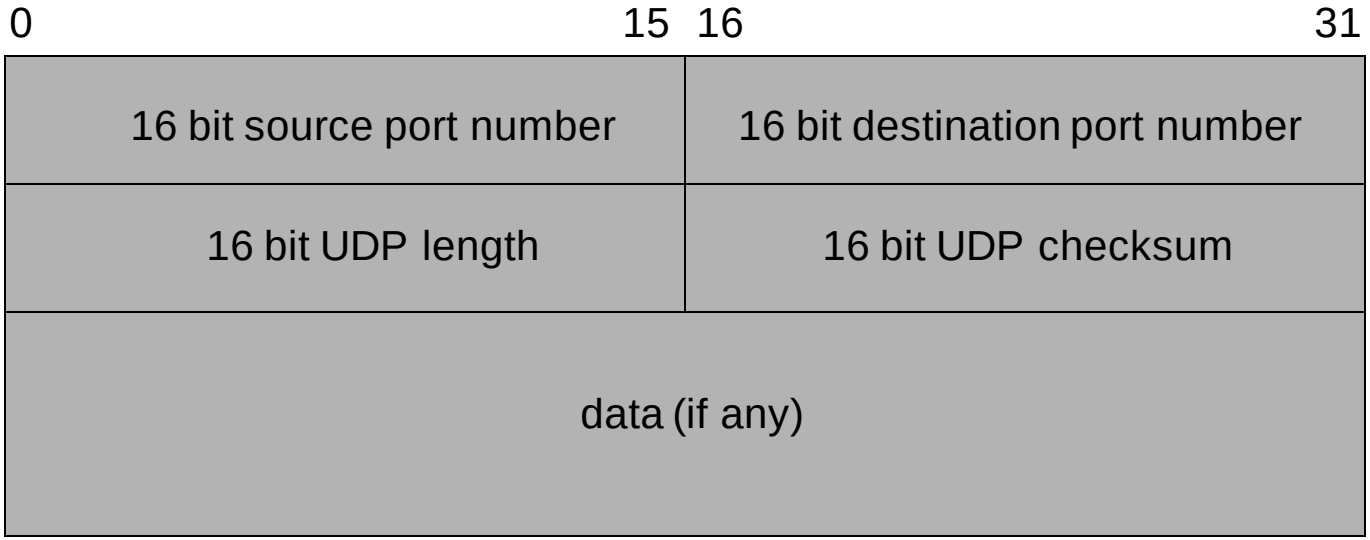
UDP

UDP – User Datagram Protocol

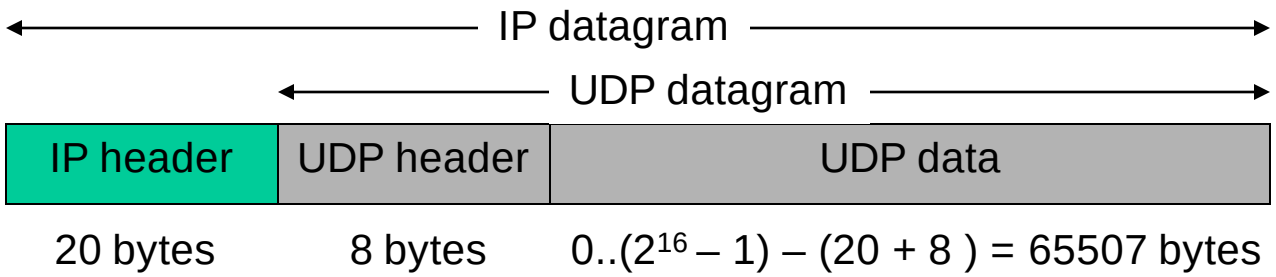
- Datagram-oriented transport layer protocol
- Provides connectionless unreliable service
- Provides optional end-to-end checksum covering header and data
- Provides no feedback to control data rate
- An UDP datagram is silently discarded if checksum errors
- UDP messages can be lost, duplicated, or arrive out of order
- Application programs using UDP must deal with reliability problems
  - DNS, DHCP, SNMP, NFS, VoIP, etc. use UDP
  - An advantage of UDP is that it is a base to build your own protocols on

UDP Message Format

8 byte header + possible data

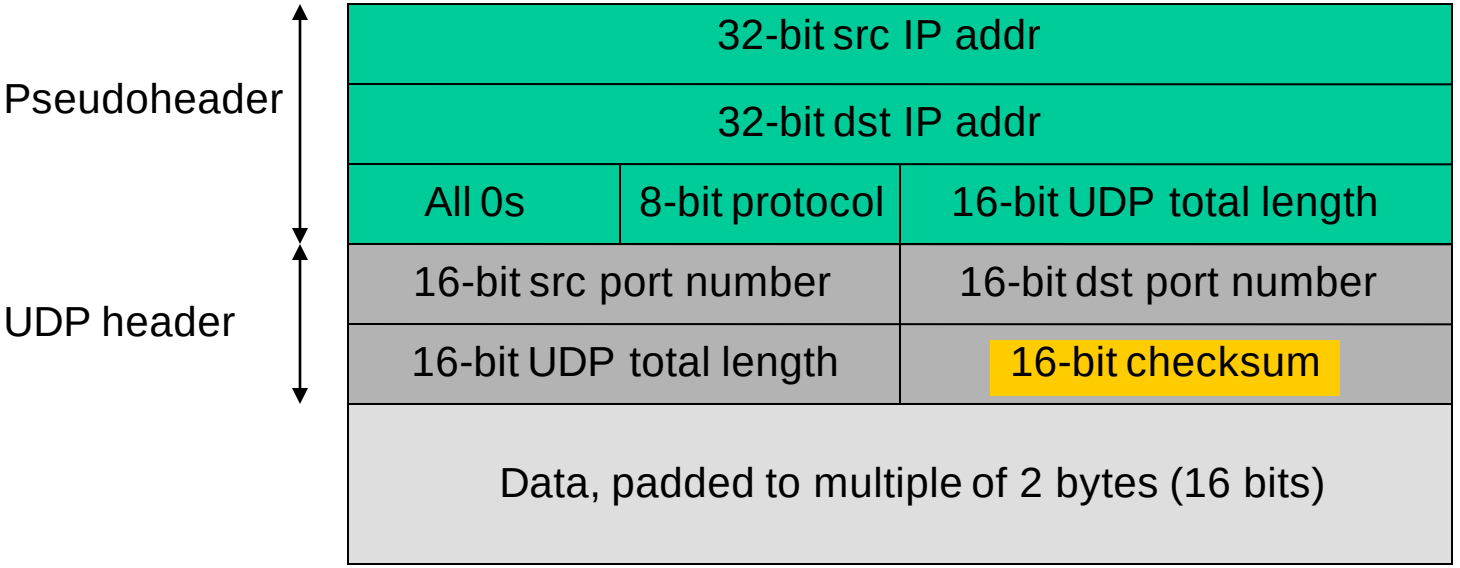


• UDP length field is redundant, since the IP software can pass this info to UDP



UDP Checksum and Pseudo-header

- UDP checksum covers
  - application data, UDP header,a pseudoheader, and pad byte (if needed)
- Purpose with pseudo-header:
  - double-check that packet arrived to correct destination
  - check that IP delivered the packet to the correct protocol (UDP/TCP)
- Pseudoheader and pad byte not transmitted, only used for computation



UDP Summary

- Transport Layer Basics
  - end-to-end delivery of messages
- UDP
  - a fairly simple connectionless protocol

TCP

Outline

- TCP – Transmission Control Protocol
  - RFC 793 (and several follow-ups)
  - Connection Management
  - Reliability
  - Flow control
  - Congestion control

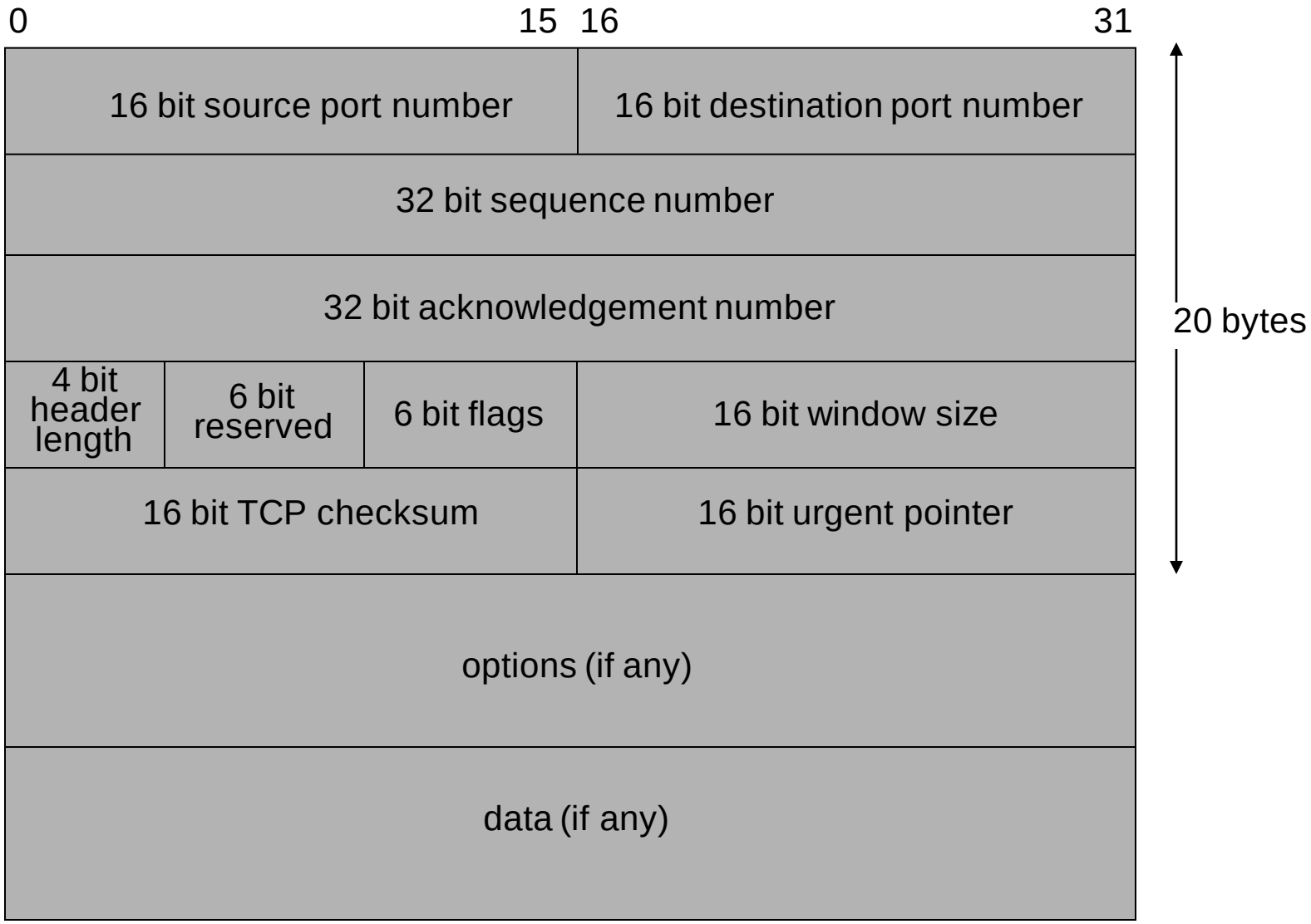
TCP

- TCP is a connection-oriented transport protocol
- A TCP connection is a full duplex connection between exactly two end-points
  - Broadcast and multicast are not applicable to TCP
- TCP provides a reliable byte stream service
  - A stream of 8-bit bytes is exchanged across the TCP connection
  - No record markers inserted by TCP
  - The receiving (reading) end cannot tell what sizes the individual writes were at the sending end
  - TCP decides how much data to send (not the application); each unit is a **segment**
- Lots of applications have been implemented on top of TCP
  - TELNET (virtual terminal), FTP (file transfers), SMTP (email), HTTP

How TCP Provides Reliability

- TCP breaks application data into best sized chunks (*segments*)
- TCP maintains a timer for each segment, waiting for acknowledgement
- TCP retransmits segment if ACK is not returned before timeout
- When TCP receives data it sends an acknowledgement back to sender
- TCP maintains end-to-end checksum on its header and data
- TCP resequences data at the receiving side, if necessary
- TCP discards duplicate data at the receiving side
- TCP provides flow control. Finite buffer at each end of the connection

TCP Segment


0
15
16
31
16 bit source port number
16 bit destination port number
32 bit sequence number
32 bit acknowledgement number
4 bit header length
6 bit reserved
6 bit flags
16 bit window size
16 bit TCP checksum
16 bit urgent pointer
options (if any)
data (if any)
20 bytes

TCP Header Fields

- *Src and Dst port numbers*: identify sending/receiving application.
  - These values along with src/dst IP addresses are called a *socket pair* and uniquely define the TCP connection.
- *Sequence number*: identifies the byte in the stream that the first byte of this segment represents. Wraps around at 2^{32}-1.
- *Acknowledgement number*: contains the next sequence number that the receiver (sender of the ACK) is ready to receive.
- *Header length*: gives length of the header in 32-bit words. Required since options field is variable. Max header length is 60 bytes ($4*(2^4 - 1) = 60$).
- *Reserved*: for future use.

TCP Header Fields cont'd

- *Flags*: 6 flags determining the purpose and contents of the segment.
- *Window size*: defines the number of bytes, starting with the one specified in the acknowledgement number, that the receiver is willing to accept. Used for *flow control* and *congestion control* (later slides).
- *TCP Checksum*: covers header and data. The calculation and verification is mandatory.
- *Urgent pointer*: provides a way for sender to transmit emergency data to receiver, e.g., end of session.
- *Options*: used by TCP to negotiate certain features between the end-points, e.g., maximum segment size (MSS).

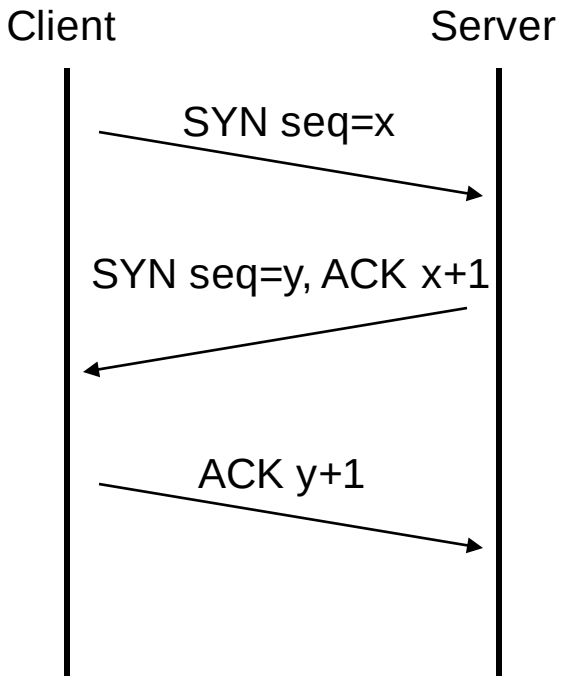
TCP Flags

The 6 flag bits in the TCP header are:

- ACK - acknowledgement number valid
- RST - reset the connection
- SYN - synchronize sequence numbers to initiate connection
- FIN - sender is finished sending data
- (URG - urgent pointer valid, set when sender wants the receiver to read a piece of data urgently and possibly out of order)
- (PSH - receiver should immediately pass the data to the application, buffers should be emptied, note difference compared to URG)

TCP Connection Establishment

TCP uses a three-way handshake to establish a connection



Normally, client initiates the connection

3-way handshake:

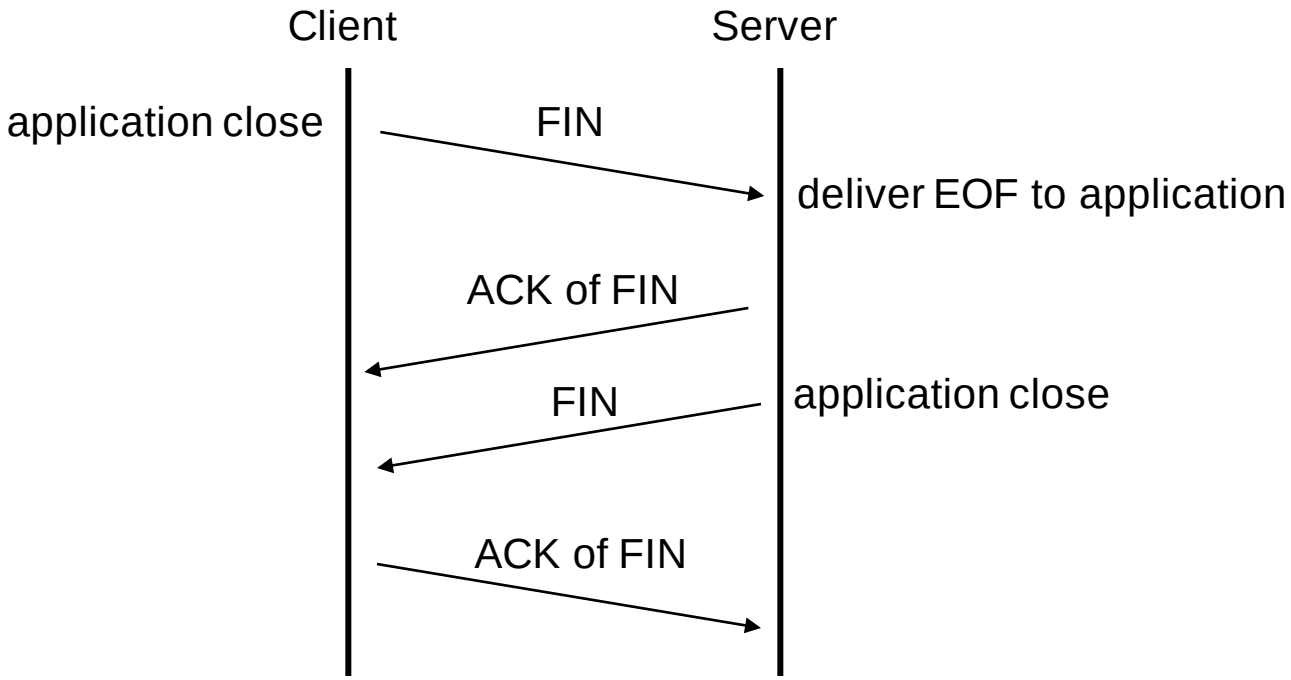
- Guarantees both sides ready to transfer data
- Allows both sides to agree on initial sequence numbers

Initial sequence number (ISN) must be chosen so that each incarnation of a specific TCP connection between two end-points has a different ISN**.

**) Prevent packets that get delayed in the network from being delivered later and misinterpreted as part of an existing connection (there is no CID).

TCP Connection Teardown

Takes 4 segments to terminate a connection.



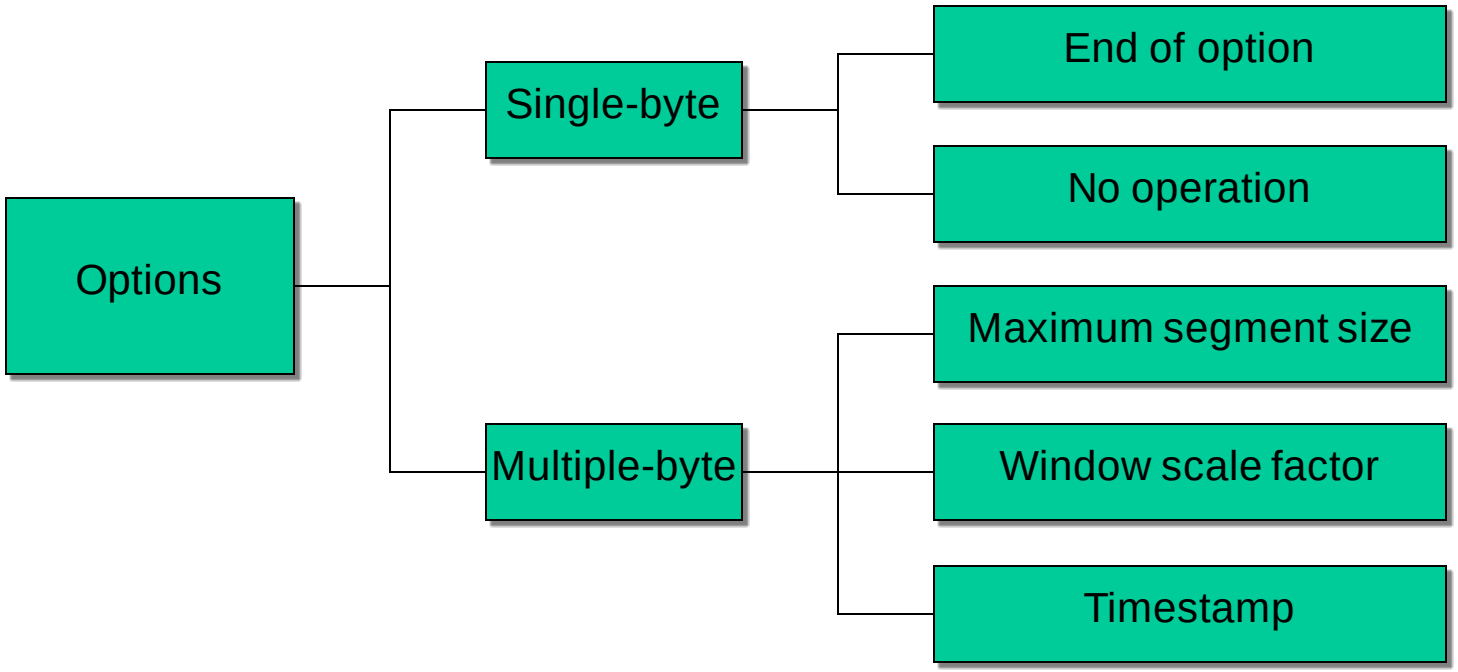
Normally, client performs *active close* and server performs *passive close*

Maximum Segment Size

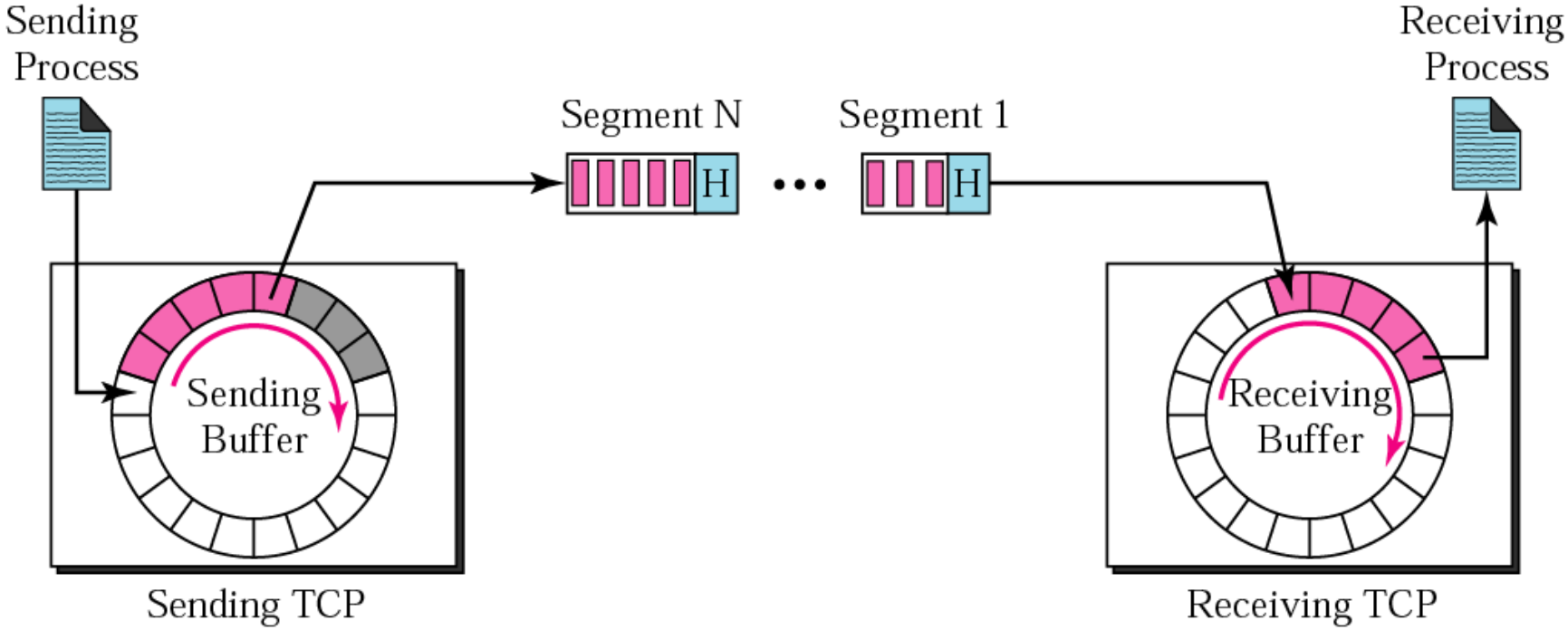
- MSS (Maximum Segment Size) is the largest chunk of data TCP will send to the other side
- MSS can be announced in the options field of the TCP header during connection establishment
- If MSS is not announced, a default value of 536 is assumed
  - 576 bytes is min MTU for IP networks, subtract 20 B IP hdr and 20 B TCP hdr → default MSS 536 to avoid IP fragmentation
- In general, the larger MSS the better until fragmentation occurs

TCP Options

- A TCP header can have up to 40 bytes of optional information
- Options are used to convey additional information to the destination or to align another options



Byte Streams, Buffers, and Segments



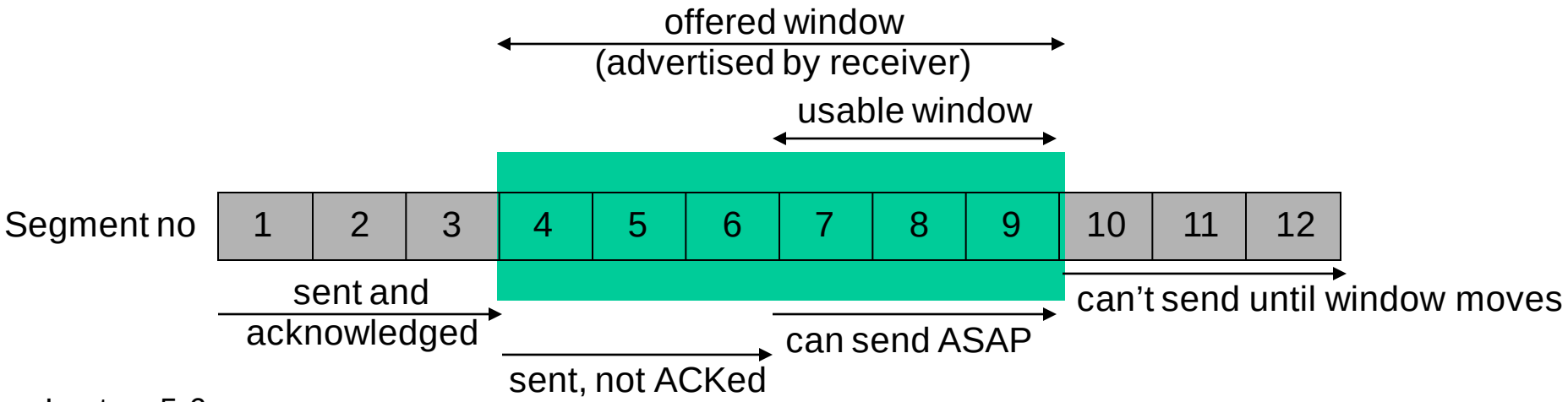
- White area: empty locations, ready to be filled
- Gray area: bytes that have been sent but not ACKed
- Colored area:
  - bytes to be sent by the sending TCP
  - bytes to be delivered by the receiving TCP

Flow Control

- *Flow control* defines the amount of data a source can send before receiving an acknowledgement from receiver
  - The flow control protocol must not be too slow (can't let sender send 1 byte and wait for acknowledgement)
  - The flow control protocol must make sure that receiver does not get overwhelmed with data (can't let sender send all of its data without worrying about acknowledgements)
- TCP uses a *sliding window* protocol to accomplish flow control
- For each TCP connection (always duplex), the sending and receiving TCP peer use this window to control the flow

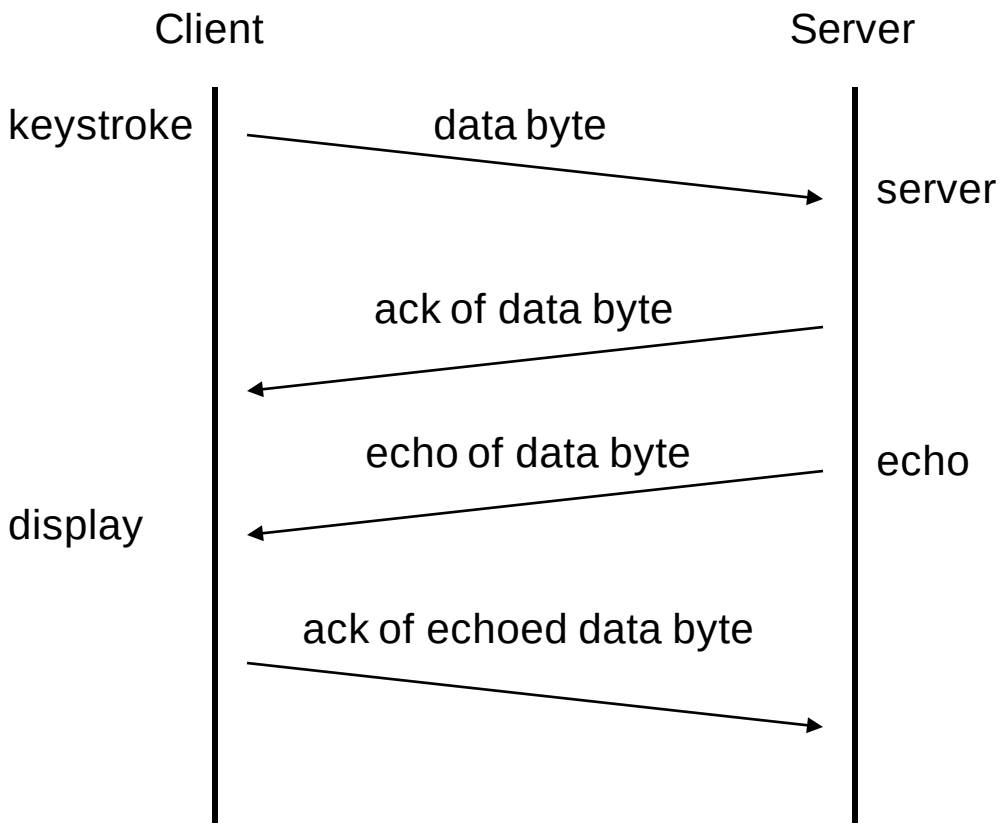
Sliding Windows

- Receiver: *offered window* – acknowledges data sent and what it is prepared to receive
  - receiver can send an ACK, but with an offered window of 0
  - later, the receiver sends a *window update* with a non-zero offered window size
- Sender: *usable window* - how much data it is prepared to send immediately



TCP Interactive Data Flow

In an rlogin session, each individual keystroke normally generates a packet



Silly Window Syndrome

Serious problems can arise in the sliding window operation when:

- Sending application creates data slowly, or
- Receiving applications consumes data slowly

(or both)

In the following we will look at how:

- sender initiated silly window syndrome is solved with *Nagle's algorithm*
- receiver initiated silly window syndrome is solved with *delayed ACKs*

Nagle's Algorithm

- telnet/rlogin/... generate a packet (41 bytes) for each 1 byte of user data
  - these small packets are called "tinygrams"
  - not a problem on LANs
  - adds to the congestion on WANs
- Nagle Algorithm
  - each TCP connection can have only one outstanding (i.e., unacknowledged) small segment (i.e., a tinygram)
  - while waiting - additional data is accumulated and sent as one segment when the ACK arrives, or when maximum segment size can be filled
  - self-clock: the faster ACKs come, the more often data is sent
  - thus automatically on slow WANs fewer segments are sent
- Round trip time (RTT) on a typical ethernet is ~16ms - thus to generate data faster than this would require typing faster than 60 characters per second! Thus rarely will Nagle be invoked on a LAN.

Delayed Acknowledgements

Suppose sender sends blocks of 1K, but receiving application consumes data 1 byte at a time. Receiving TCP buffer will get full, and ACKs will be sent for each byte read from the buffer.

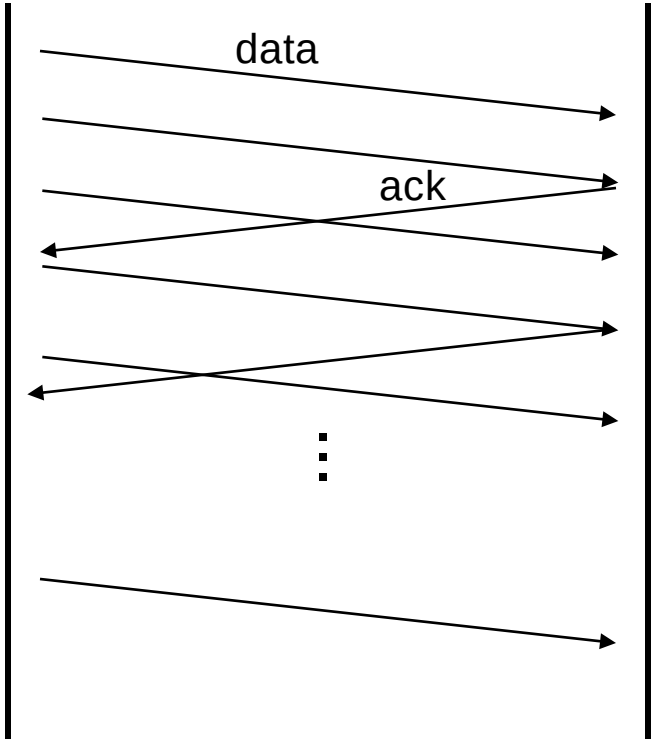
Solution:

- Rather than sending an ACK immediately, TCP waits up to ~200ms
- The delayed ACK will prevent sender from sliding its window
- Other advantages:
  - Traffic is reduced, increased chance that data can be piggy-backed on the ACK
- Host Requirements RFC states the delay must be less than 500ms

TCP Bulk Data Flow

Data transfer: full MSS.

Typically: ACK every other segment



Bandwidth-Delay Product

How large should the window be for optimal throughput?

Calculate the capacity of the ”pipe” as:

capacity(bits) = bandwidth(bits/sec) x RTT(sec)

This is the size the receiver advertised window should have for optimal throughput.

Example:

T1 connection across the US:

capacity = 1.544Mbit/s x 60ms = 11,580 bytes

Window size field is 16 bits → Max value is 65535 bytes

For ”Long Fat Pipes”, the window scale option can be used to allow larger window sizes

Regarding Flow Control

Some points about TCP's sliding windows

- The src does not have to send a full window's worth of data
- Window size can be increased/decreased by dst
- The dst can send an ACK at any time

In TCP, the sender window size is totally controlled by the receiver window value (the number of empty locations in the receiver buffer). However, the *actual* window size can be smaller if there is *congestion* in the network.

Typical sizes of the receiver buffer are 4K, 8K, or 16K. However, application can change the value through the programming interface (e.g., the *socket API*)

Congestion

- Previos section about flow control assumed the network could always deliver the data as fast as it was created by the sender

However,

- Each router along the way has buffers, that stores the incoming packets before they are processed and forwarded
- If packets are received faster than they can be processed, congestion might occur and packets could be dropped
- Lost packet means lost ACK, and sender retransmits
- Retransmission will add to congestion → network collapses

Thus, the *network* has to have a way to decrease window size

Congestion Control

- Congestion Window
  - Sender's window size is not only determined by the receiver, but also by the congestion in the network
- Sender maintains 2 window sizes:
  - Receiver-advertised window
  - Congestion window (CWND)
- Actual window size = min(rcv window, CWND)
- To deal with congestion, sender uses several strategies:
  - Slow start
  - Additive increase of CWND
  - Multiplicative decrease of CWND

Slow Start

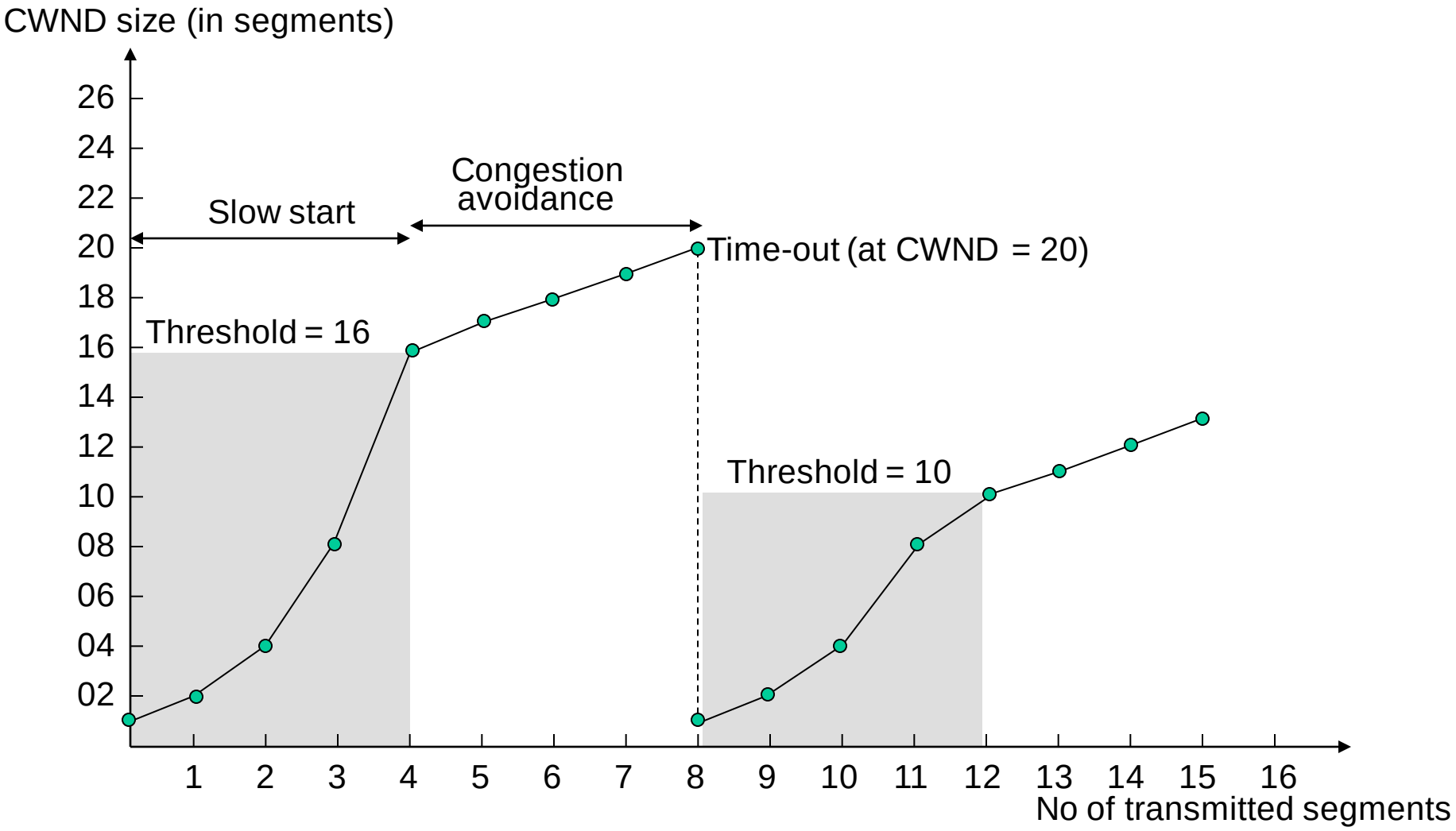
Slow start was introduced by Van Jacobson 1989, based on his analysis of actual traffic, and the application of control theory. All TCP implementations are now required to implement slow start.

- At beginning of connection, CWND=MSS
- For each ACKed segment, CWND is increased by one MSS
  - Until a threshold of half allowable window size is reached
- Process not slow at all – window size grows exponentially
  - CWND = 1
  - Send 1 segment
  - Receive 1 ACK, and increase CWND to 2
  - Send 2 segments
  - Receive ACK for 2 segments, and increase CWND to 4

Congestion Avoidance

- Slow start will increase CWND size exponentially until threshold value is reached
- To avoid congestion this exponential growth must slow down:
  - After the threshold is reached, CWND will be increased additively (one segment per ACK, even if several segments are ACKed)
  - This linear growth will continue until either:
    - Receiver-advertised window size is reached, or
    - ACK timeout occurs → TCP assumes congestion
- When congestion occurs:
  - Threshold value is set to ½ the last CWND
  - CWND restarts from one MSS again
- Initial threshold value is 65535 bytes, minimum value is 512 bytes

Congestion Avoidance cont'd



Fast Retransmit and Fast Recovery

Van Jacobson introduced this modification to the congestion avoidance algorithm in 1990.

TCP is required to generate an immediate ACK (a duplicate ACK) when an out-of-order segment is received, to inform sender what segment number that is expected.

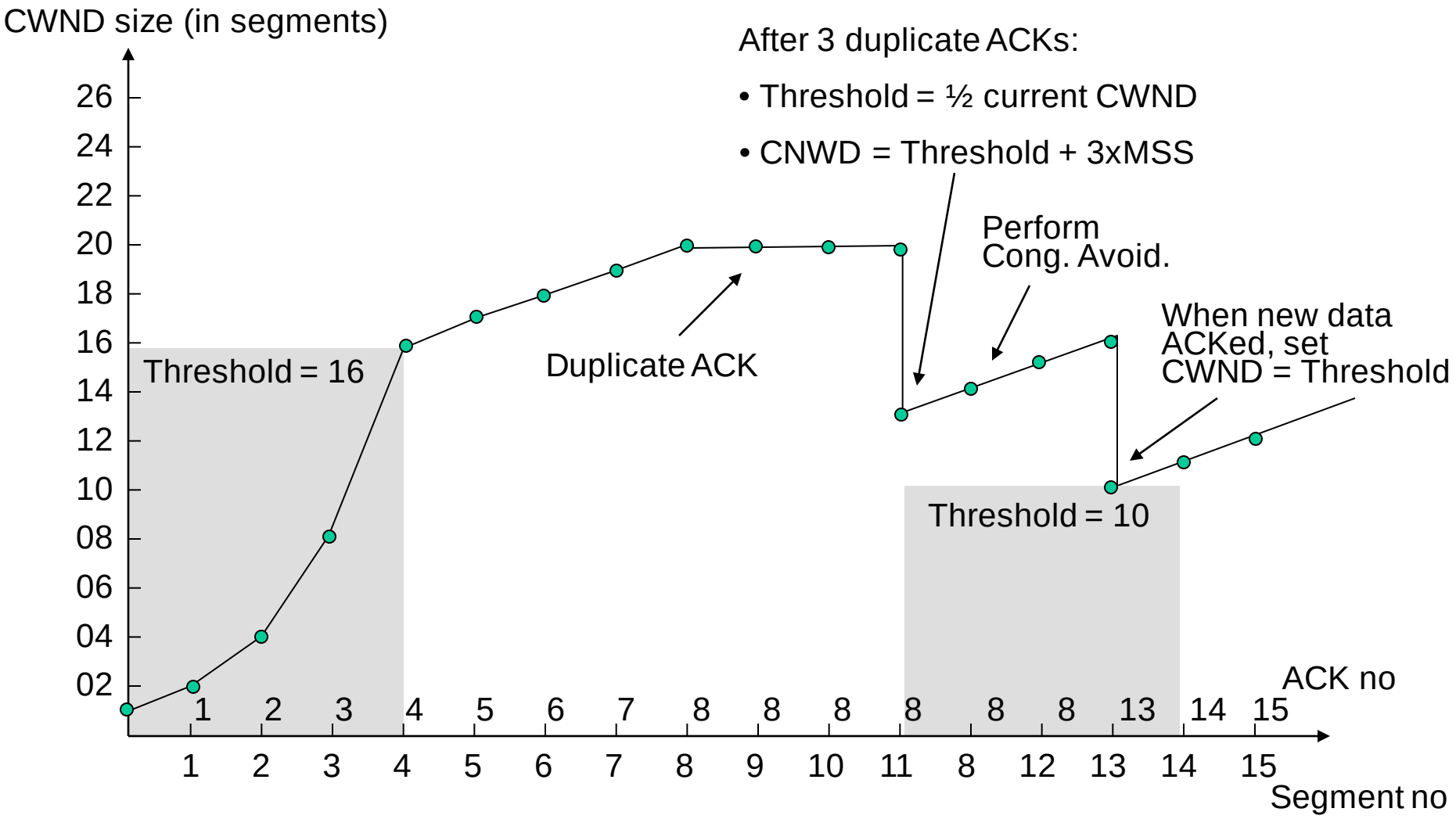
Cause of duplicate ACK could be

- lost segment, or
- reordering of segments
- but data is still flowing between the two ends!!

If 3 (or more) duplicate ACKs are received in a row:

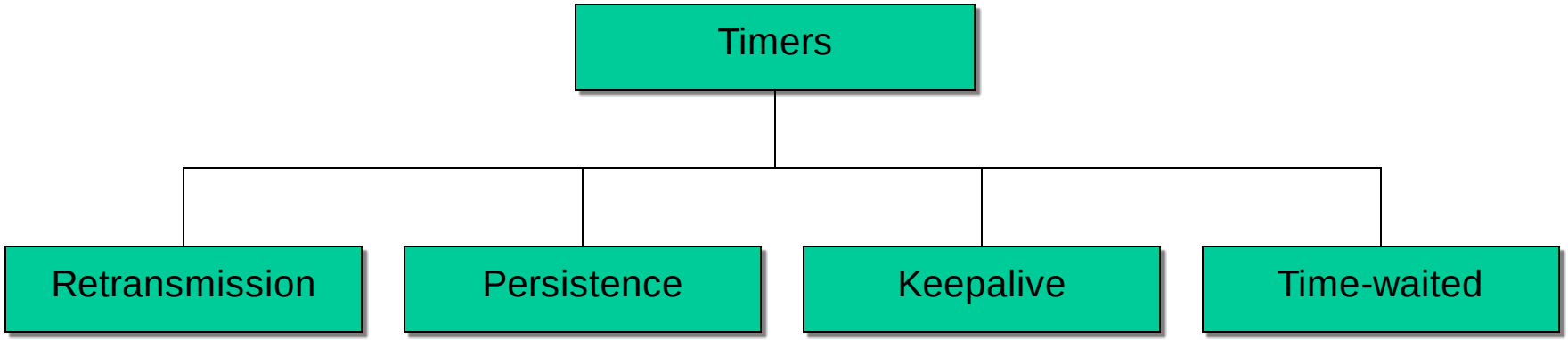
- Retransmit immediately (before time-out) - *Fast Retransmit*
- Do congestion avoidance (not slow start) - *Fast Recovery*

Fast Retransmit and Recovery cont'd



Timers

To perform its operation smoothly, TCP uses four timers



Retransmission Timer

- Retransmission time is the waiting time for an ACK of a segment (Retransmission Time-Out – RTO)
- 2 situations:
  1. If ACK is received before timer goes off, the timer is cancelled
  2. If timer goes off before ACK arrives, segment is retransmitted
- TCP cannot use same RTO for all connections
- TCP even adapts the RTO within a particular connection
- RTO can be made dynamic by basing it on the round-trip time (RTT)
- Exponential backoff: double the RTO for each retransmission

Calculation of RTT

Originally, TCP specified

- $RTT_{smooth} = a \times RTT_{smooth} + (1 - a) \times RTT_{measured}$
  - $RTT_{measured}$ is measured (send segment, measure time until ACK)
  - a (smoothing factor) is normally set to 0.875
  - $RTO = 2 \times RTT_{smooth}$ (one method)
- Another method
  - RTT variance:
    - $RTT_{variance} = (1 - b) \times RTT_{variance} + b \times | RTT_{measured} - RTT_{smooth} |$
  - $RTO = RTT_{smooth} + 4 \times RTT_{variance}$

Karn's Algorithm

Problem:

- Suppose a segment is not ACKed before RTO and thus retransmitted
- When this segment is ACKed the sender does not know if it is the original or retransmitted segment that is ACKed
- Thus, TCP can't get the measured value needed for RTT calculation

Karn found a simple solution:

- Don't consider the RTT of a retransmitted segment
- Reuse the backed off RTO for the next transmission
- Don't update RTT value until you get an ACK without need for retransmission

Persistence Timer

- If receiver announces a window size of 0, sender will stop sending and wait for an ACK with non-zero window size
- If this ACK gets lost, a deadlock occurs
  - Sender waits for ACK
  - Receiver waits for data
- To resolve the deadlock, sending TCP uses a persistence timer that cause the sender to periodically send *window probes* to find out if window size has increased
- Persistence timer ensures that window probes are sent at least every 60 seconds until window is reopened

Keepalive Timer

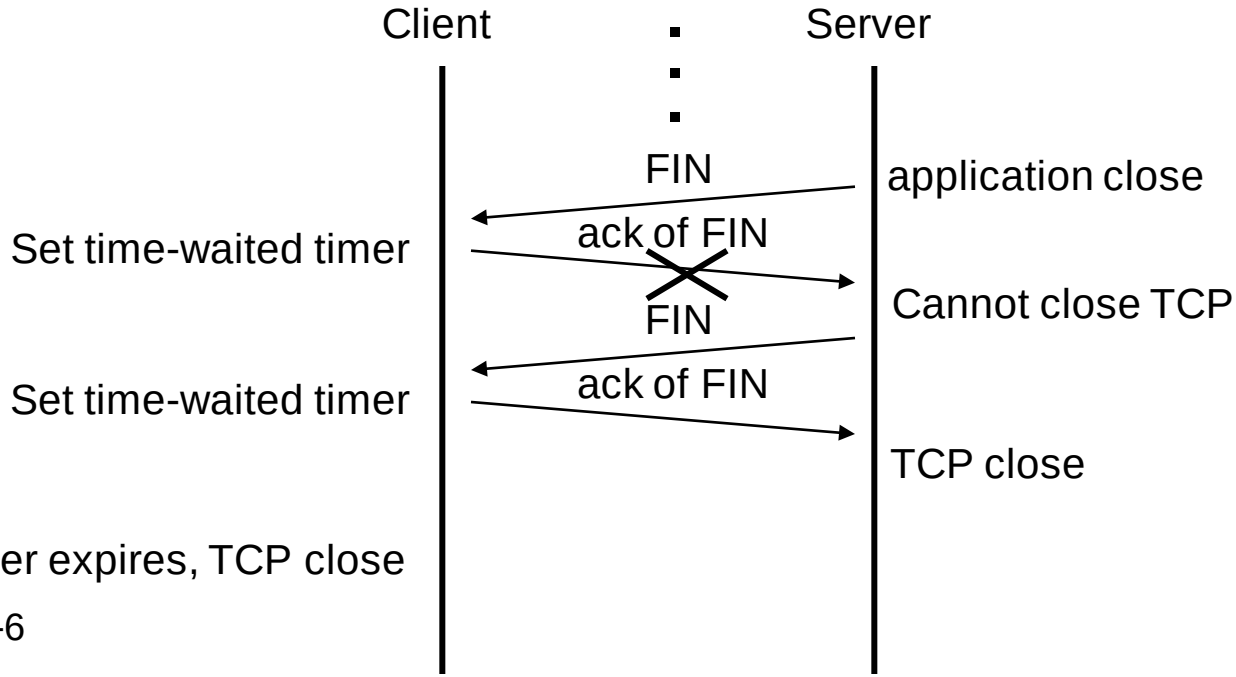
Keepalive Timer is used to prevent TCP connections to exist forever

- Imagine a client connects to a server, transmits some data, and then crashes
- TCP connection would continue to exist, but the keepalive timer ensures that connection gets terminated
- Normally, the server gets a keepalive time-out after 2 hours
- After keepalive time-out, server probes the client
- If client doesn’t respond, the connection is torn down

Time-Waited Timer

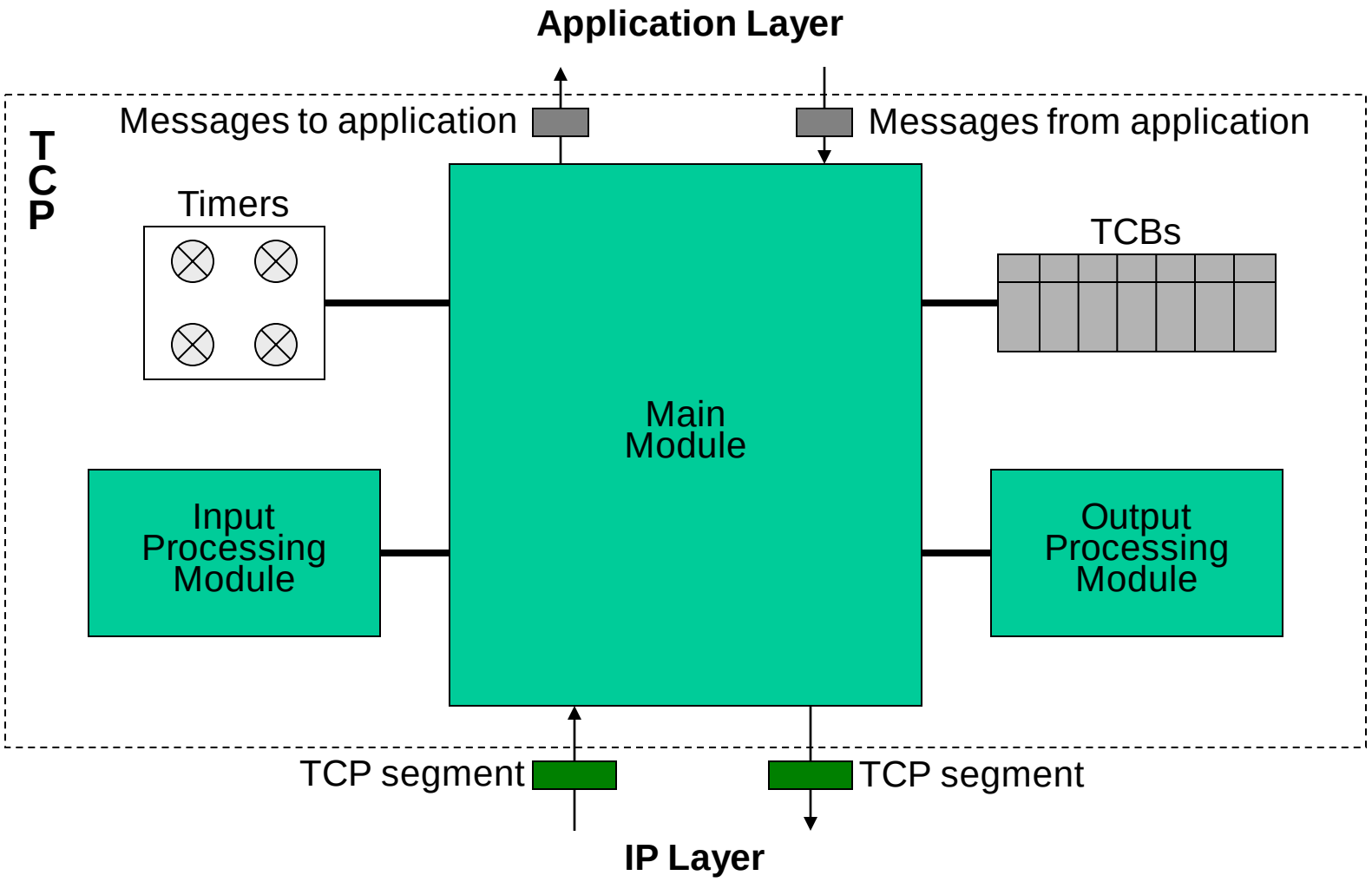
Time-waited timer is used during connection termination.

When TCP closes a connection, the connection is held in a limbo for a time-waited period allowing duplicate FIN segments to arrive at the destination. The value of the timer is usually twice the expected lifetime of a segment.
(2 MSL = 2 x Maximum Segment Lifetime)



TCP Package

TCP is a very complex protocol, tens of thousands lines of code!



TCP Flavors

- Tahoe: Van Jacobson, 1988
  - Congestion avoidance and control
- Reno: Van Jacobson, 1990
  - Modified congestion avoidance algorithm
  - Fast retransmit and fast recovery
  - Reactive algorithm
- Vegas: Brakmo, Peterson, 1994
  - New techniques for congestion detection and avoidance
  - Proactive algorithm
- Westwood: Casetti et. al, 2000
  - Adaptation to high-speed wired/wireless environment
  - Deals with large bandwidth-delay product and packet loss (leaky pipes)

TCP Summary

- TCP is a very complex protocol
  - Connection Management
  - Reliability
  - Flow control
  - Congestion control
  - Timers