CSCI 460
Networks and Communications

Transport Layer

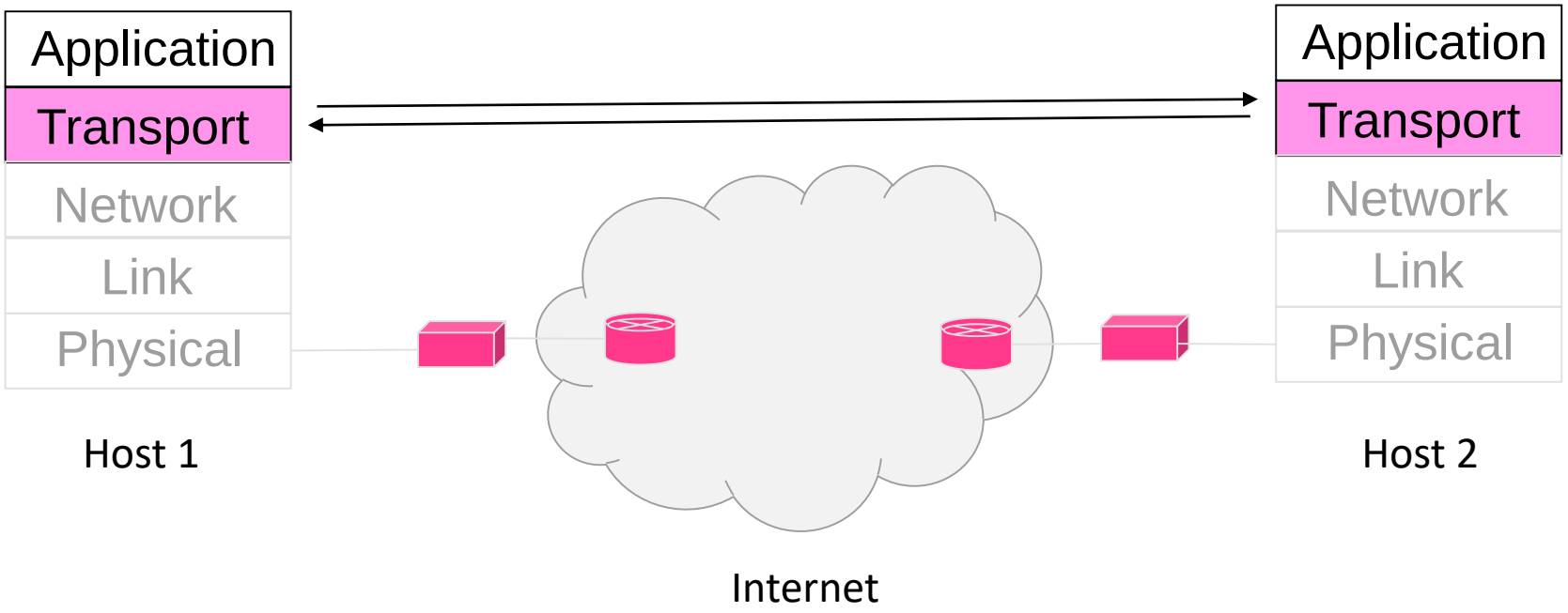
Humayun Kabir
Professor, CS, Vancouver Island University, BC, Canada

Outline

- User Datagram Protocol (UDP)
- Transport Control Protocol (TCP)
  - TCP Segment Header
  - TCP Connection
  - TCP Flow Control
  - TCP Congestion Control
  - TCP Retransmission Timer

The Transport Layer

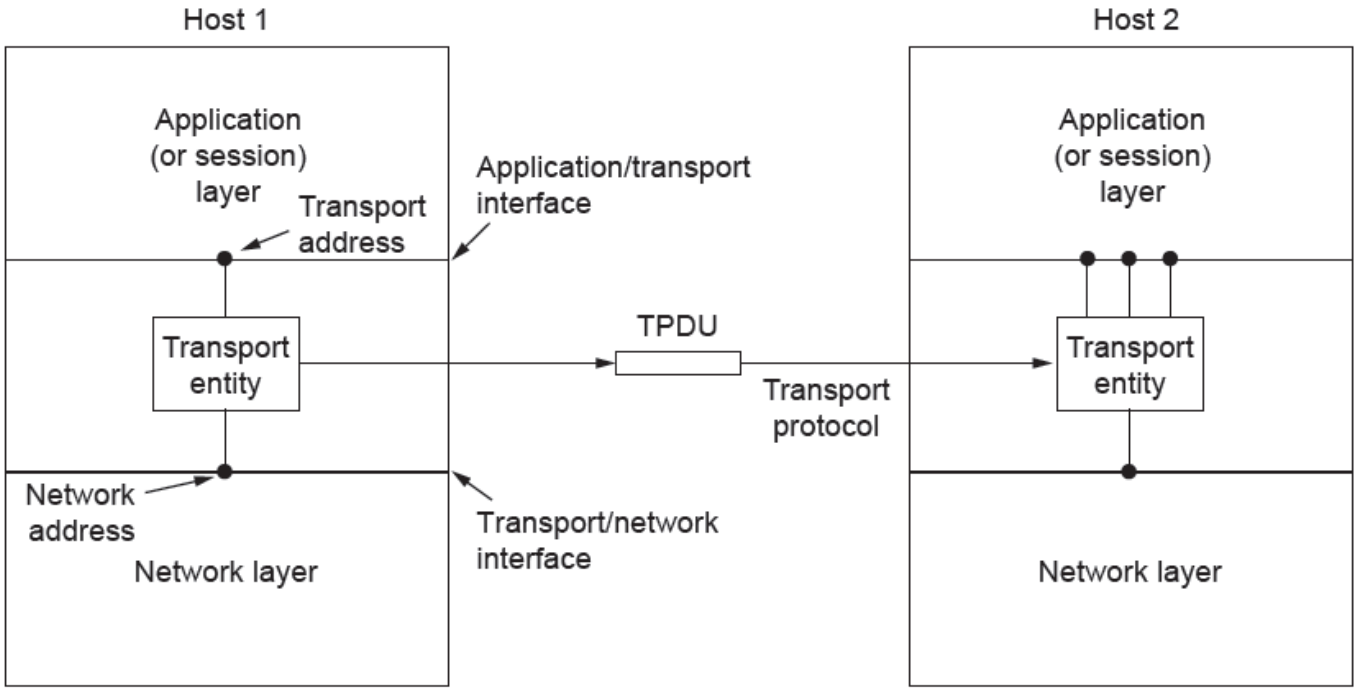
Responsible for delivering data from source to destination nodes (across networks) with the desired reliability or quality



Services Provided to Application Layer

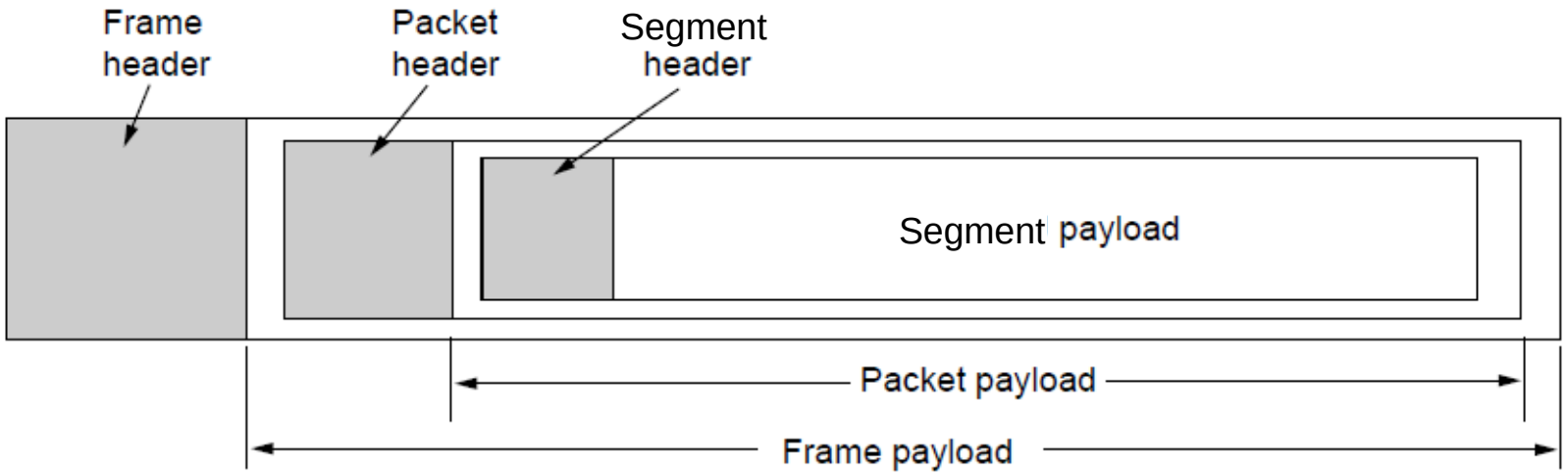
Transport layer adds reliability to the network layer

- Offers connectionless (e.g., UDP) and connection-oriented (e.g, TCP) service to applications



Services Provided to Application Layer

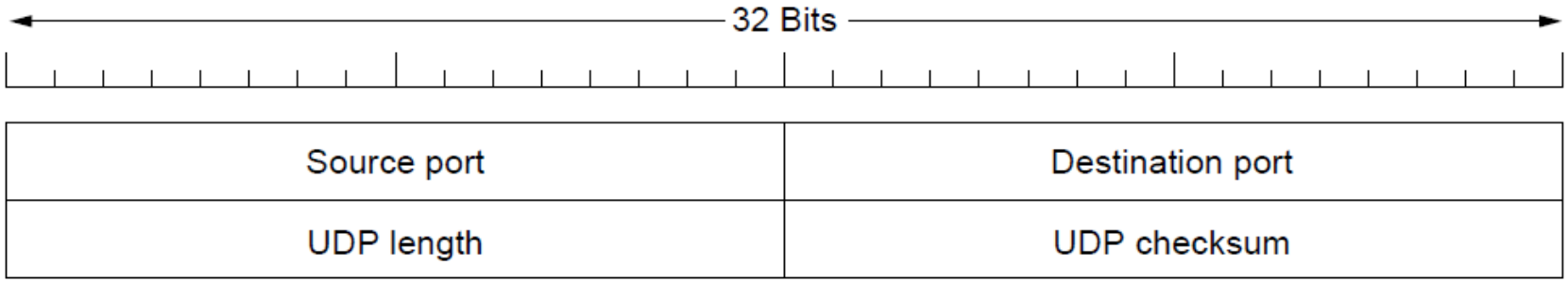
Transport layer sends transport segments inside network packets (inside datalink frames)



UDP in TCP/IP Protocol Stack

UDP (User Datagram Protocol) is a shim over IP

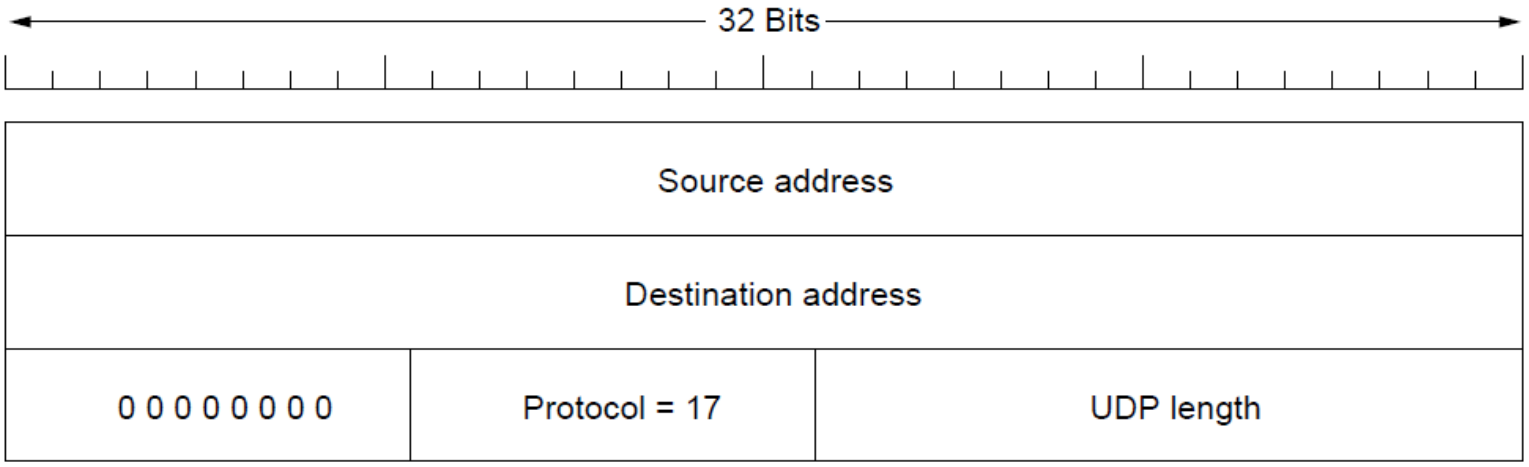
- Header has **ports** (TSAPs), **length** and **checksum**.



- **UDP ports** indicate the end points at the source and destination
- **UDP length** counts both UDP header and payload in bytes
- **UDP checksum** is optional and computed on IP pseudo header, UDP header, and UDP payload using modulo 2^{16} sum and its one's complement.

UDP in TCP/IP Protocol Stack

IP pseudo header for UDP checksum



- IP fields that change in the network are zeroed out, e.g., TTL

UDP checksum provides an end-to-end error check

TCP in TCP/IP Protocol Stack

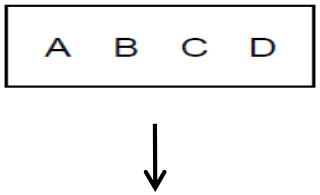
TCP provides applications with a reliable byte stream between processes; it is the workhorse of the Internet

- Many popular servers run on well-known ports using TCP

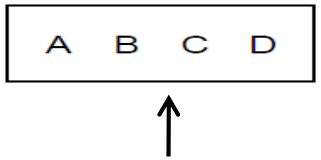
Port	Protocol	Use
20, 21	FTP	File transfer
22	SSH	Remote login, replacement for Telnet
25	SMTP	Email
80	HTTP	World Wide Web
110	POP-3	Remote email access
143	IMAP	Remote email access
443	HTTPS	Secure Web (HTTP over SSL/TLS)
543	RTSP	Media player control
631	IPP	Printer sharing

TCP in TCP/IP Protocol Stack

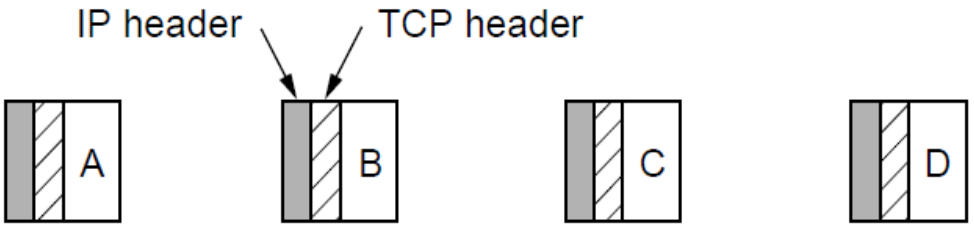
Applications using TCP see only the byte stream and not the segments sent as separate IP packets



2048 bytes of data handed by application



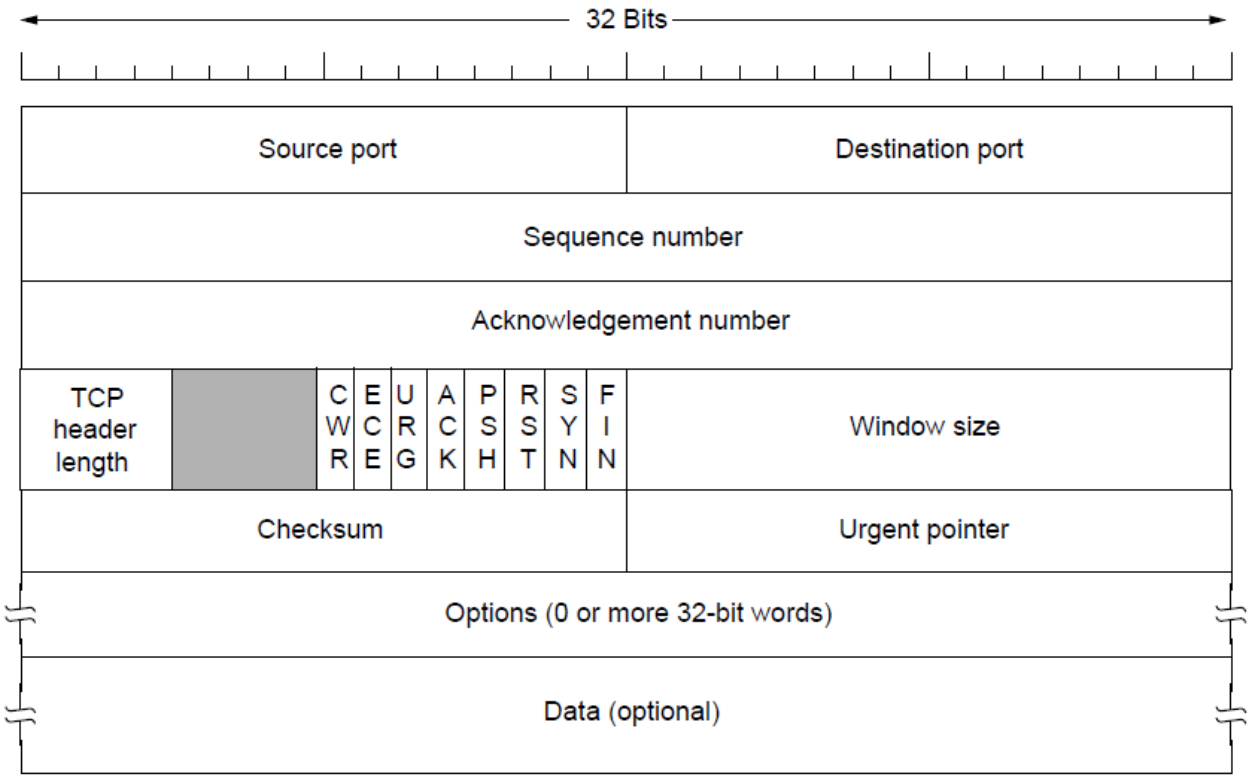
2048 bytes of data delivered to application



TCP default data or payload size is 536 bytes. 4 TCP segments, each with 536 bytes of data in first 3 segments and 440 bytes data in the 4th segment carried in 4 IP packets

TCP Segment Header

TCP header includes addressing (ports), sliding window (seq. / ack. number), flow control (window), error control (checksum) and more.



TCP Segment Header

- **TCP ports** indicate the end points at the source and destination
- **TCP sequence number** indicates the byte address of the first byte of TCP segment in the byte stream.
- **TCP acknowledge number** indicates which byte in the byte stream the receiver is expecting next and a cumulative acknowledgement of all the bytes before it.
- **TCP window size** indicates the size of the buffer available at the receiver.
- **TCP checksum** is computed like UDP checksum to provide end-to-end error control. Unlike UDP, it is not optional.

TCP Segment Header

- **TCP header length** counts the number of 32-bit words in TCP header.
- **TCP ACK flag** indicates that the segment's is also an acknowledgement segment and its **acknowledgement number** is a valid acknowledgment number.
- **TCP SYN flag** indicates that the segment is a SYN segment.
- **TCP FIN flag** indicates that the segment is a FIN segment.
- **TCP RST flag** indicates that the segment is a RST segment.

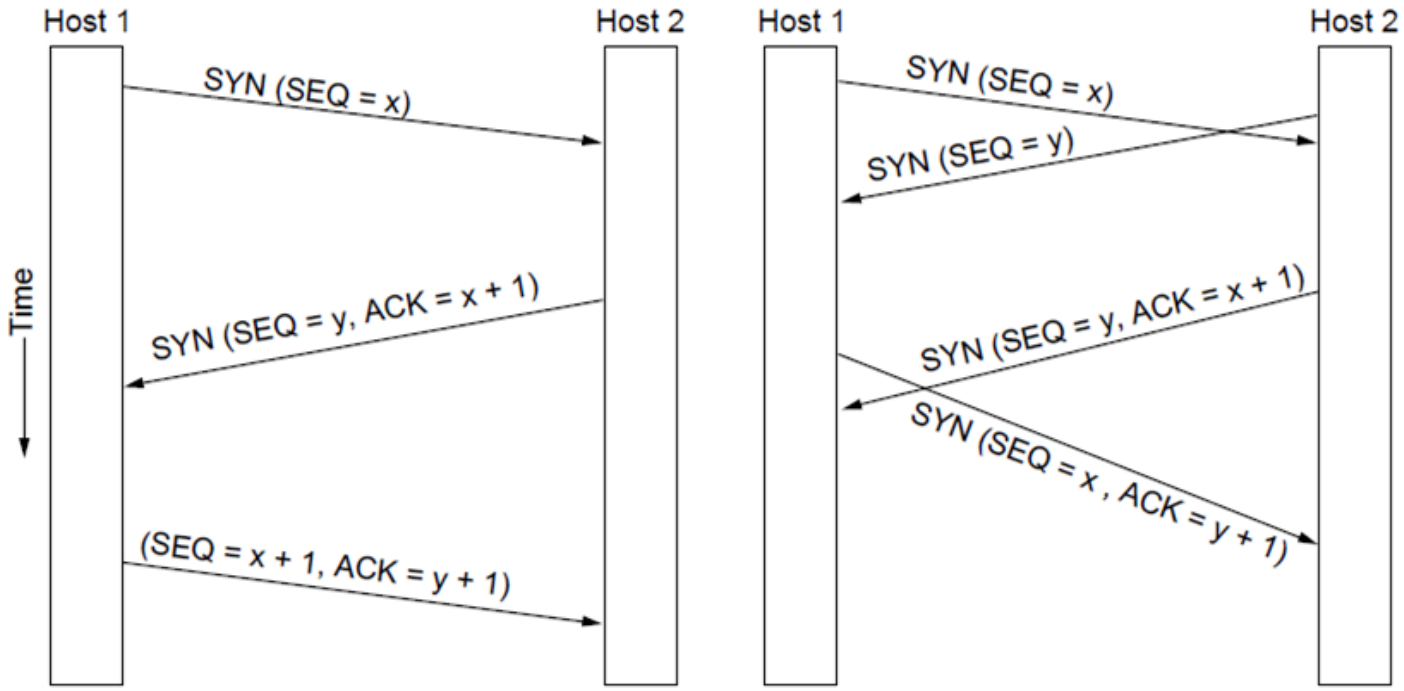
TCP Segment Header

- **TCP PUSH flag** signals the receiver not to buffer the payload, rather pass it to the upper layer.
- **TCP URG flag** signals that the segment has **urgent data** to be processed immediately and the urgent data ends at **urgent pointer**.
- **TCP ECE flag** is set by the receiver to request the sender to reduce its congestion window. Receiver repeats ECE until the sender reduces (50%) the congestion window and reports a **CWR** to the receiver.

TCP Connection Establishment

TCP sets up connections with the **three-way handshake**

- Release is symmetric, also as described before

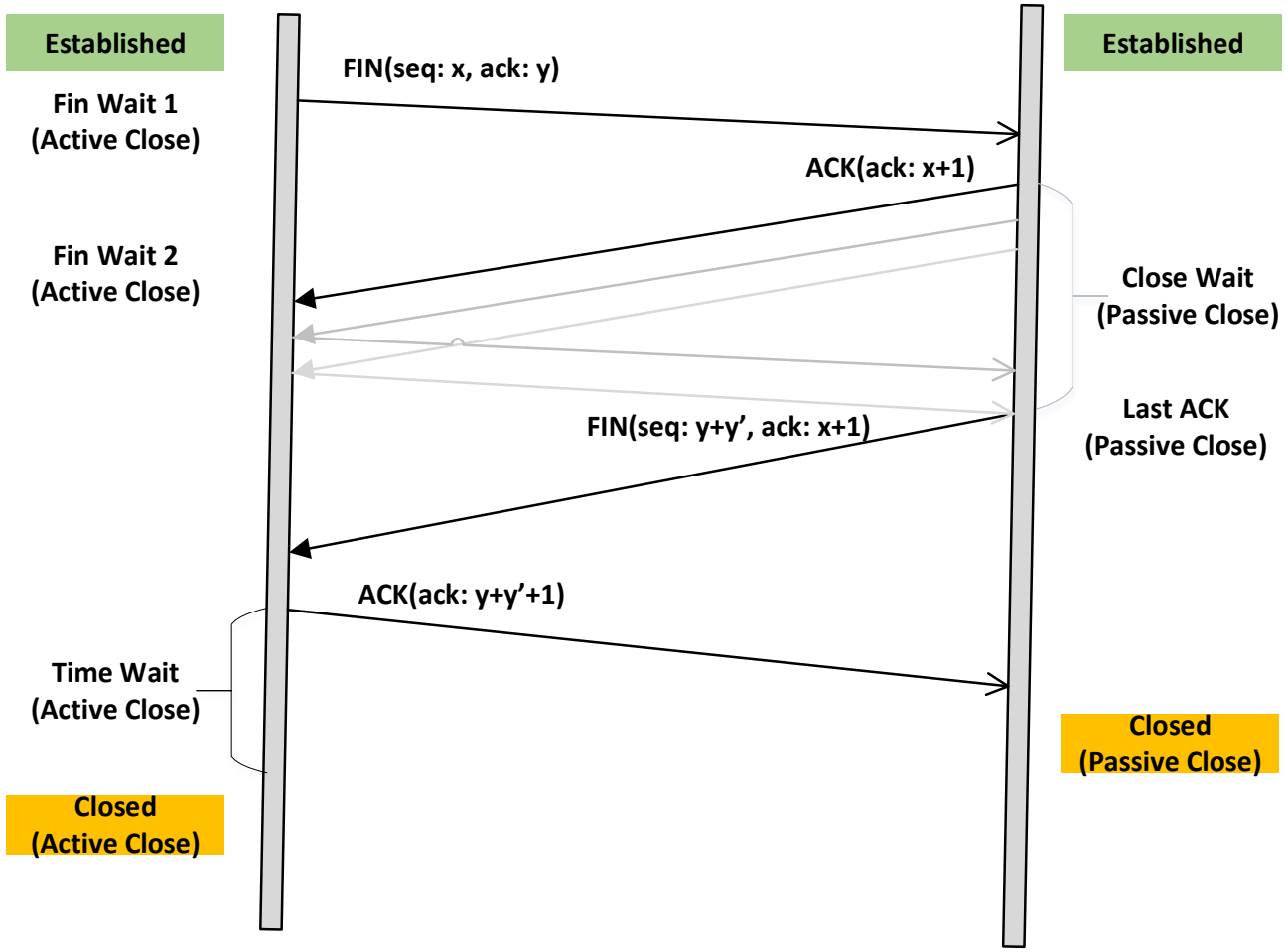


Normal case

Simultaneous connect

TCP Connection Release

TCP release connections with the **three-way handshake**.



TCP Connection State Modeling

The TCP connection finite state machine has more states than our simple example from earlier.

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

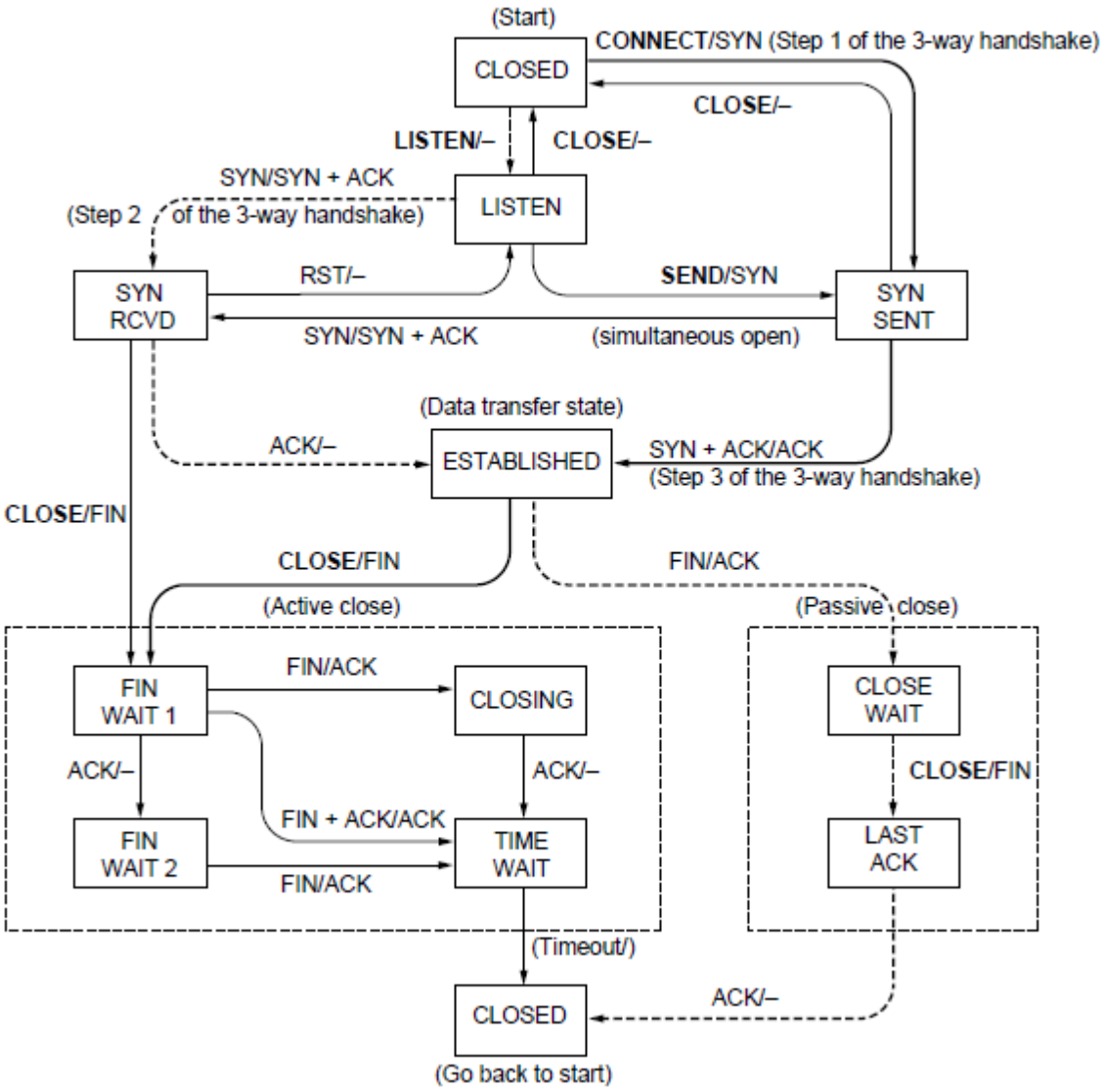
TCP Connection State Modeling

Solid line is the normal path for a client.

Dashed line is the normal path for a server.

Light lines are unusual events.

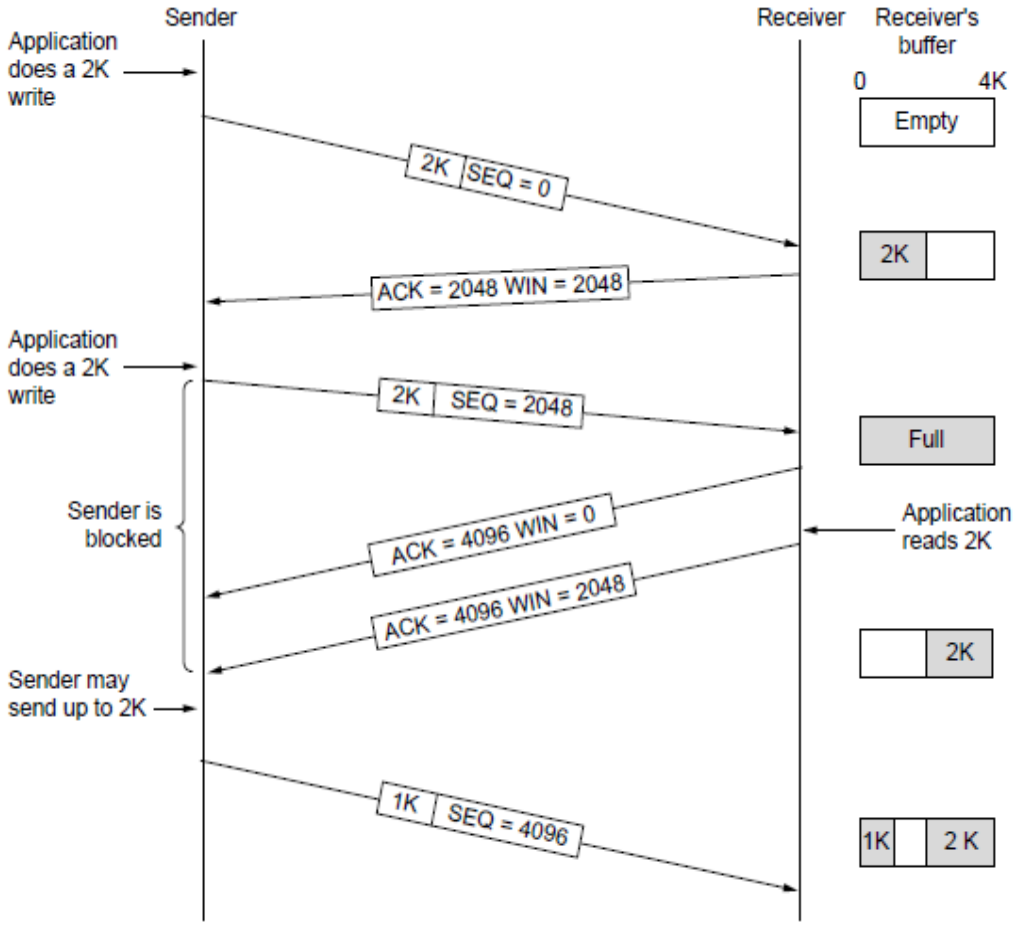
Transitions are labeled by the cause and action, separated by a slash.



TCP Sliding Window (Flow Control)

TCP adds flow control to the sliding window as before

- ACK + WIN is the sender's limit



TCP Sliding Window (Flow Control)

- **Delayed Acknowledgements**
  - TCP receiver delays the acknowledgments for 500 msec with the hope to acquire enough data from the sender.
  - **Minimizes** the number of **TCP ACK** segments over the network.
  - Longer the response time to interactive applications.

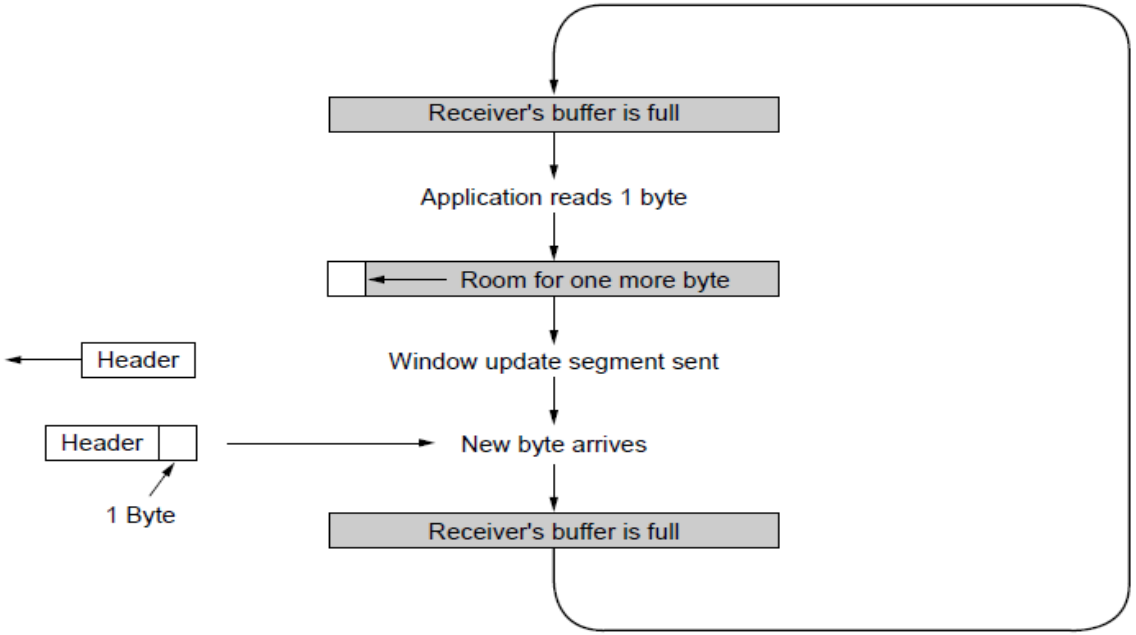
TCP Sliding Window (Flow Control)

- **Nagle's Algorithm**
  - When data from the applications comes at TCP sender in small pieces send the first piece and buffer the rest until the first acknowledgement is returned.
  - **Minimizes** the number of **TCP Data** segments over the network.
  - Longer the response time to interactive applications, it can be deactivated using TCP_NODELAY option.

TCP Sliding Window (Flow Control)

Need to add special cases to avoid unwanted behavior

- E.g., silly window syndrome [below]



Receiver application reads single bytes, so sender always sends one byte segments

TCP Sliding Window (Flow Control)

- **Cark Solution to Silly Window Syndrome**
  - TCP receiver **delays** the **acknowledgments** until either the half or an MSS equivalent of the receiver buffer is empty.
  - TCP sender instead of sending tiny segments it **sends segments** whose size is at least one MSS or half of the receiver window size.
  - **Minimizes** the number of both **data** and **acknowledgment segments**.
  - Longer the response time to interactive applications.

TCP Congestion Control: Congestion Window

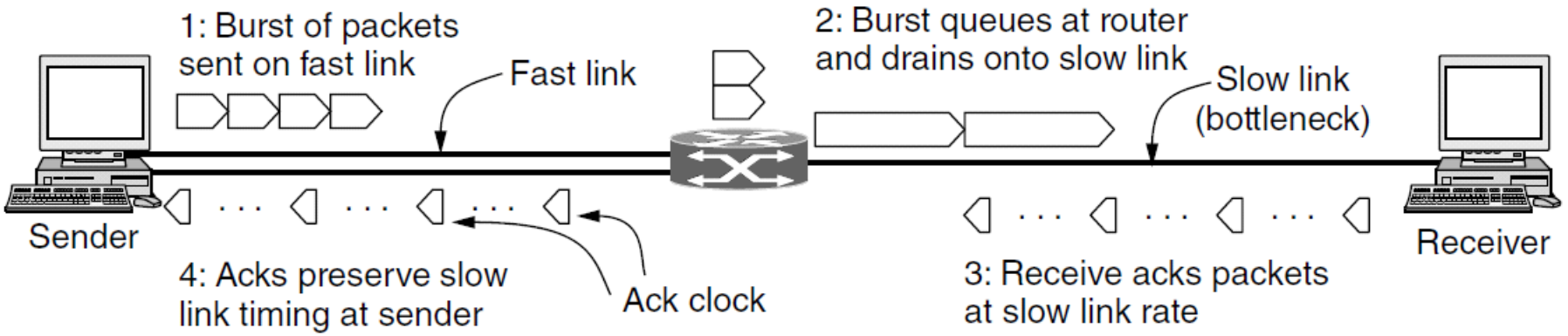
TCP uses **Additive Increase** and **Multiplicative Decrease** (AIMD) with loss signal to control congestion

- Implemented as a **congestion window** (**cwnd**) for the **number** of **unacknowledged segments** that may be in the network.
- A sender **sends** at most **cwnd** number of segments. i.e., **cwnd** controls the **sending rate**, **cwnd/RTT**.
- **cwnd** is **updated** at the reception of every **acknowledgment** segment.
- Uses several mechanisms that work together.

TCP Congestion Control

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

TCP Congestion Control: Ack Clock



ACKs pace new segments into the network and smooth bursts

- **ACK Clock** : The rate at which TCP sender receives the acknowledgements, reflects the rate of the slowest link in the network. Paces traffic and smooths out sender bursts.
- TCP **congestion window (cwnd)** is regulated on **ACK Clock**.

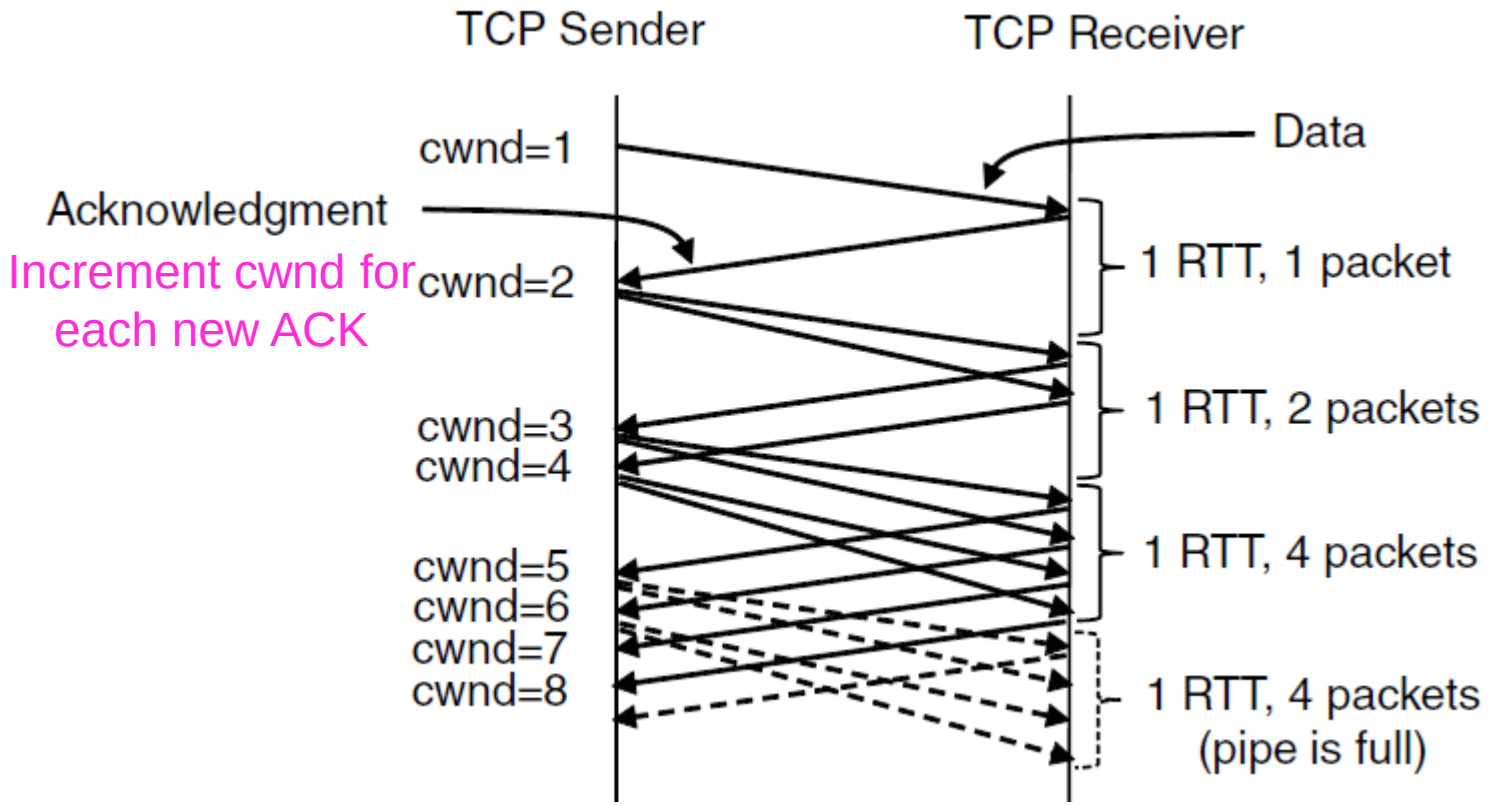
TCP Congestion Control: Slow Start Mode

- Congestion window (**cwnd**) starts with 1 MSS (at most 4 MSS).
- At the reception of each 1 MSS acknowledgement sender increments the cwnd 1 MSS and sends 2 MSS, i.e., the congestion window is doubled in every **round-trip-time** (RTT).
- This **exponential growth** continues until it reaches a threshold called **slow-start-threshold**.
- Initial slow-start-threshold is set to receiver window size and the sender enters into **additive increase** mode when **cwnd** reaches to **slow-start-threshold**.

TCP Congestion Control: Slow Start Mode

Slow start grows congestion window exponentially

- **Doubles** cwnd every RTT while keeping ACK clock going



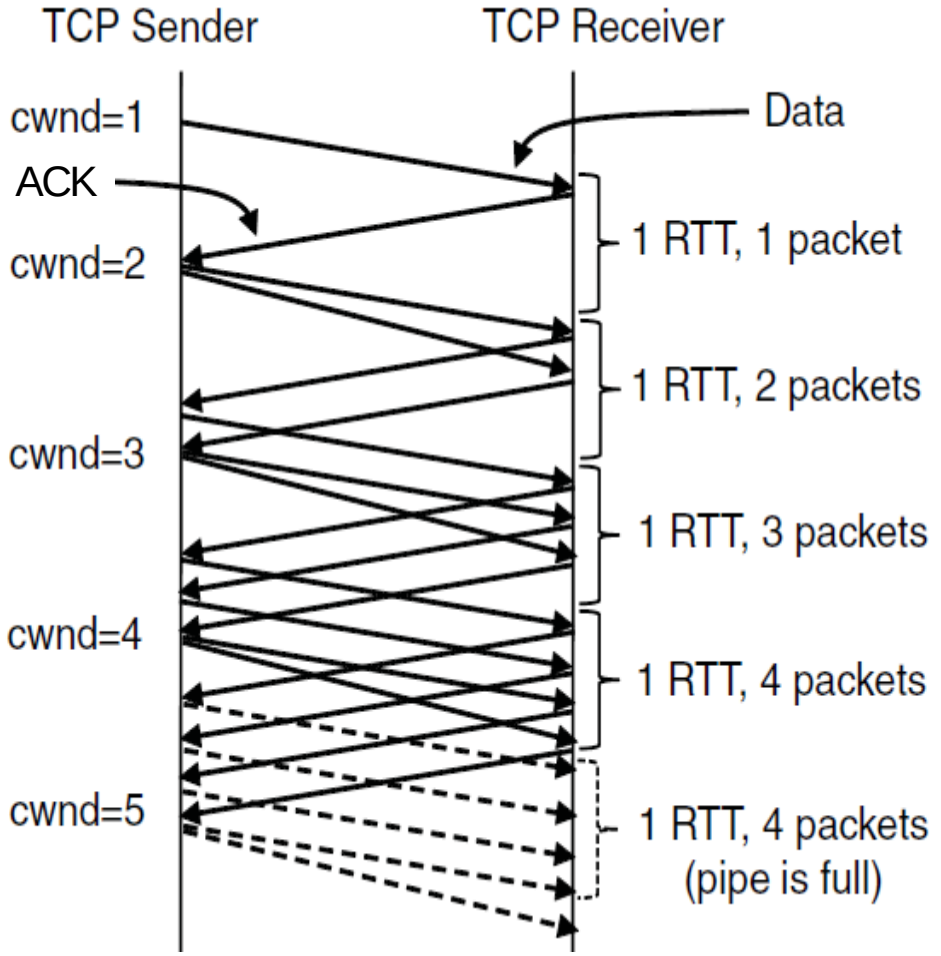
TCP Congestion Control: Additive Increase Mode

- Congestion window grows linearly instead of exponentially. Congestion window is incremented 1 MSS in every RTT.
- At the reception of each 1 MSS acknowledgement sender sends either 2 MSS or 1 MSS depending on its current congestion window credit and increments the congestion window by (1 **MSS x 1 MSS)/cwnd ≈ 1/cwnd**.

TCP Congestion Control: Additive Increase Mode

Additive increase grows cwnd slowly

- Adds 1 every RTT
- Keeps ACK clock



TCP Congestion Control: Segment Loss and Retransmission

- When a segment is lost, i.e., the **retransmission timer** times out, sender's **cwnd** undergoes a multiplicative decrease by resetting the **slow-start-threshold** to the **half** of its current **cwnd** and **cwnd** to **1 MSS**.
- Sender restarts the **slow-start mode**, irrespective of its current mode (slow-start or additive increase), after a segment loss.

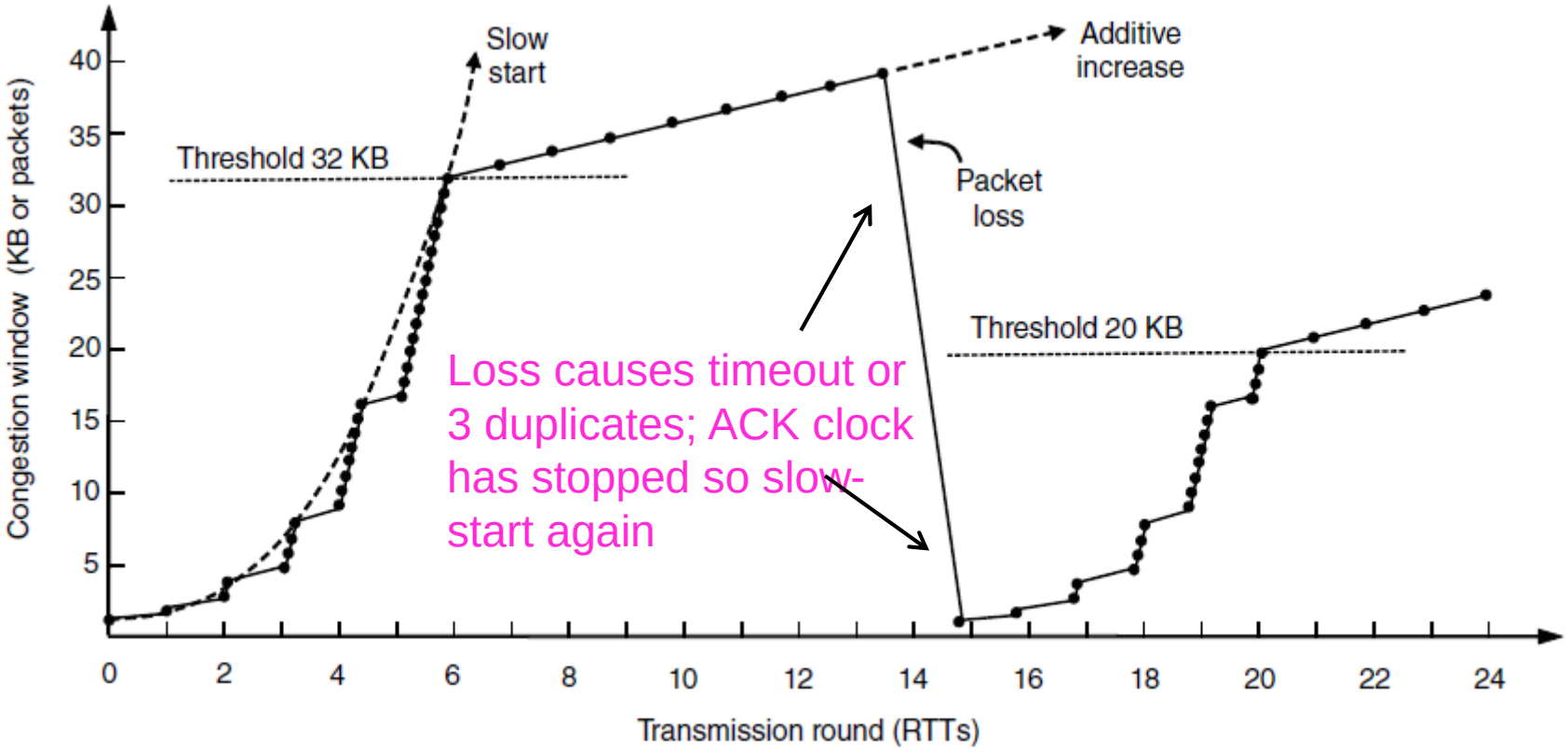
TCP Congestion Control: Fast Retransmission

- Sender assumes **segment loss** after receiving **3 duplicate acknowledgements**.
- Retransmits the lost segment, resets the slow-start-threshold to the half of the current congestion window, and repeats the slow start mode.
- Segment lost detection does not wait for the **retransmission timer to** time out, i.e., recovers from the loss quicker.

TCP Congestion Control: Fast Retransmission

Slow start followed by additive increase (TCP Tahoe)

- Threshold is half of previous loss cwnd



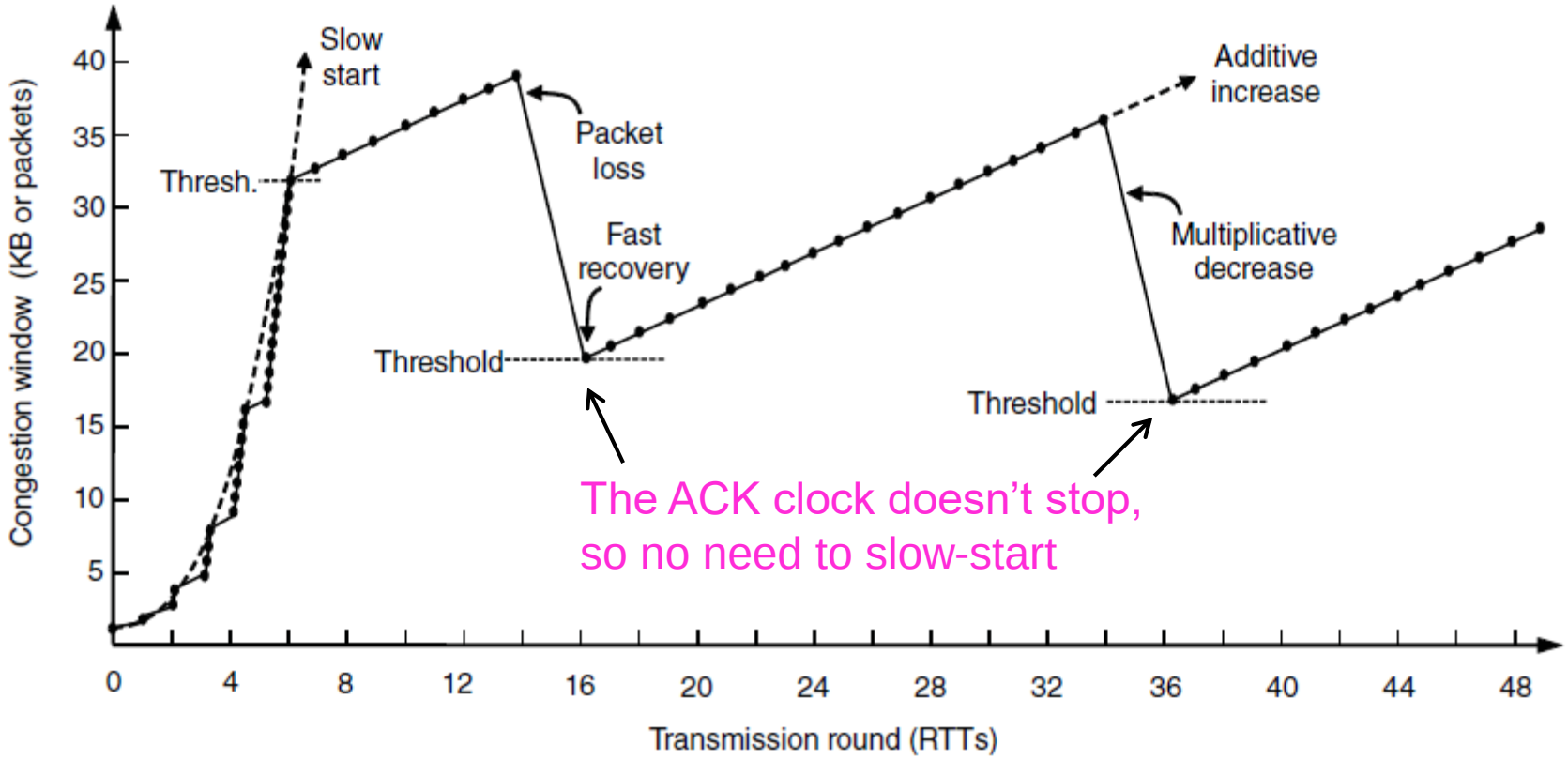
TCP Congestion Control: Fast Recovery

- After receiving **3 duplicate acknowledgements** retransmits the lost segment, resets both the congestion window and the slow-start-threshold to the half of the current congestion window, and enters into **fast recovery** mode for a short period of time.
- Counts all the duplicate acknowledgments, including the first 3 duplicates, and transmits a new segment against each duplicate acknowledgement.
- Exits from the fast recovery mode when duplicate acknowledgements ceases and repeats **additive mode**.

TCP Congestion Control: Fast Recovery

With fast recovery, we get the classic sawtooth (TCP Reno)

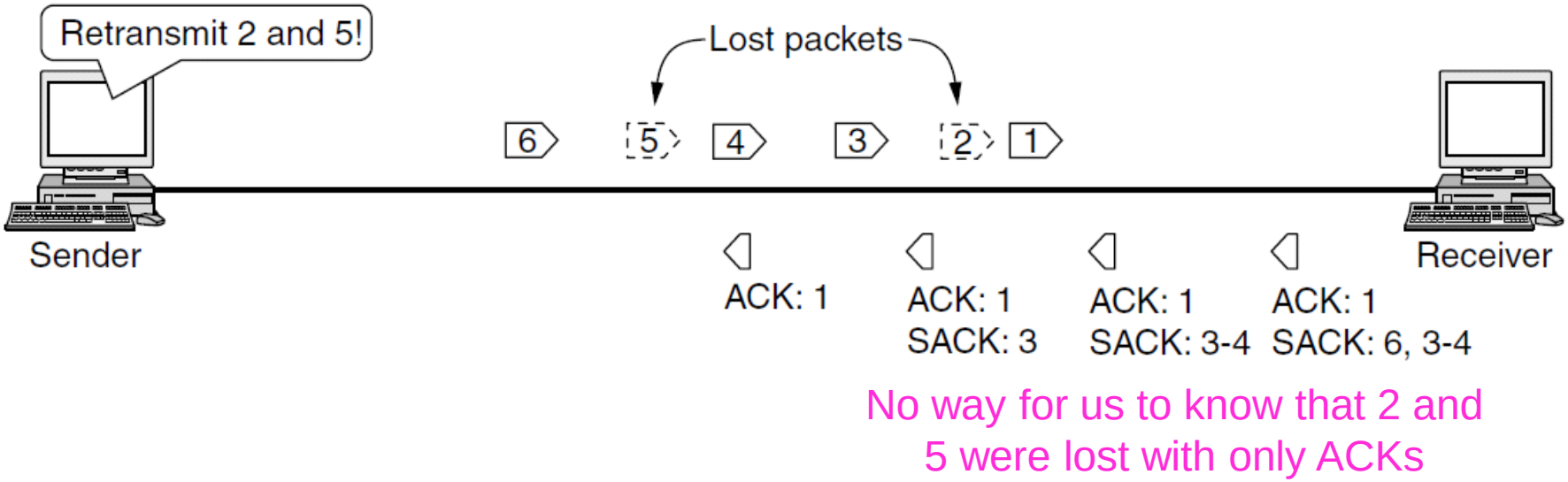
- Retransmit lost packet after 3 duplicate ACKs
- New packet for each dup. ACK until loss is repaired



TCP Congestion Control: Selective ACK

SACK (Selective ACKs) extend ACKs with a vector to describe received segments and hence losses

- Allows for more accurate retransmissions / recovery



No way for us to know that 2 and 5 were lost with only ACKs

Explicit Congestion Notification (ECN)

- Both TCP and IP layers work in synergy.
- **ECN** is **negotiated** between TCP sender and receiver while establishing TCP connection.
- ECN negotiated sender marks the outgoing IP packets with **ECN Capable Transport** (either 01 or 10).
- If a router on the path supports ECN and experiences congestion, it changes ECN marker of the IP packets to **Congestion Experienced** (11).

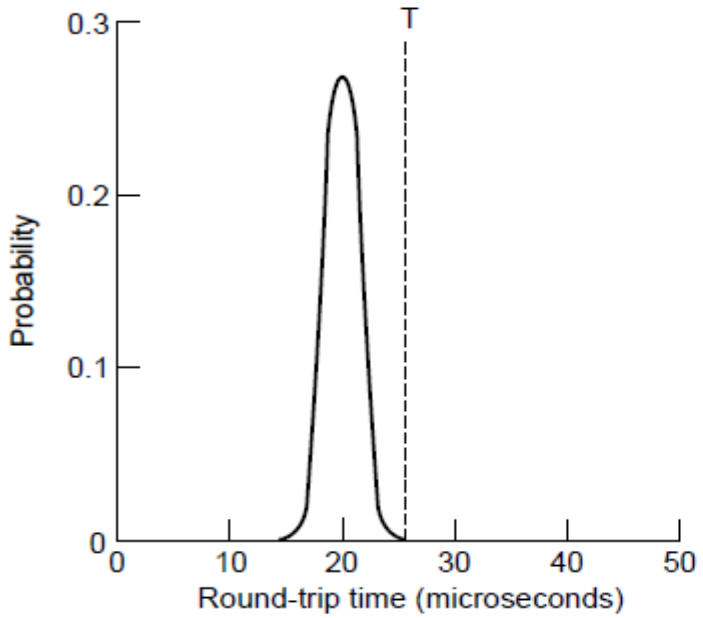
Explicit Congestion Notification (ECN)

- ECN negotiated TCP receiver keeps replying with **ECE (ECN-Echo)** bit set TCP segments until it receives a TCP segment with **CWR (Congestion Window Reduced)** bit set.
- Upon receiving a TCP segment with ECE bit set, TCP sender reduces the **congestion window** as for a segment drop and sends a TCP segment with CWR bit set.

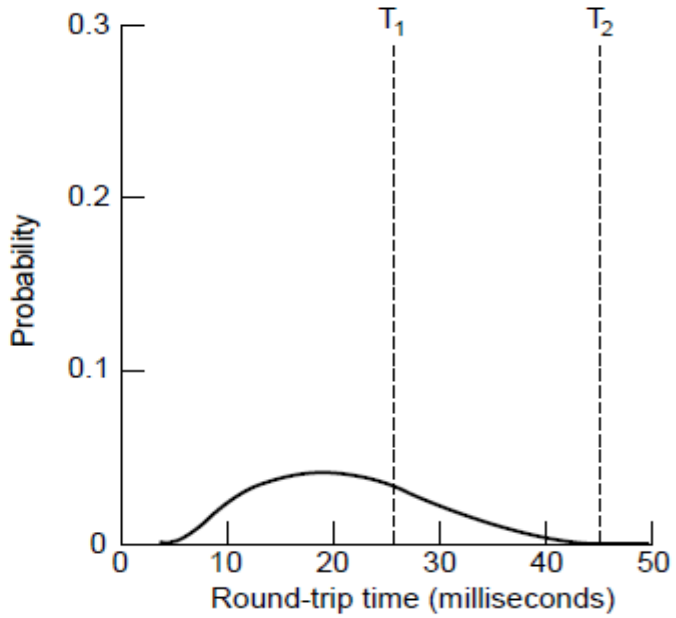
TCP Timer Management

TCP estimates retransmit timer from segment RTTs

- Tracks both average and variance (for Internet case)
- Timeout is set to average plus 4 x variance



LAN case – small, regular RTT



Internet case – large, varied RTT

TCP Timer Management

$R_0, R_1, R_2, R_3, \ldots. R_n$

$SRTT_0 = R_0$

$SRTTVAR_0 = R_0/2$

$\alpha = 0.0 \ldots 1.0$

$\beta = 0.0 \ldots 1.0$

$SRTT_n = \alpha * SRTT_{n-1} + (1-\alpha) * R_n$

$SRTTVAR_n = \beta * SRTTVAR_{n-1} + (1-\beta) * |(SRTT_n - R_n)|$

$RTT = SRTT_n + 4 * SRTTVAR_n$

Summary

- User Datagram Protocol (UDP)
- Transport Control Protocol (TCP)
  - TCP Segment Header
  - TCP Connection
  - TCP Flow Control
  - TCP Congestion Control
  - TCP Retransmission Timer

Next

Application Layer

- DNS
  - Name Space
  - Resource Record
  - DNS Server
- HTTP
  - URL
  - HTML
  - HTTP Methods
  - HTTP Headers
- FTP
  - Control and Data Connections
  - Commands and Replies