Transport Layer

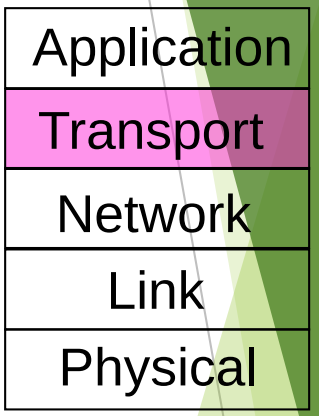
Chapter 6

- Transport Service
- Elements of Transport Protocols
- Congestion Control
- Internet Protocols – UDP
- Internet Protocols – TCP
- Performance Issues
- Delay-Tolerant Networking

Revised: August 2011

The Transport Layer

Responsible for delivering data across networks with the desired reliability or quality



Transport Service

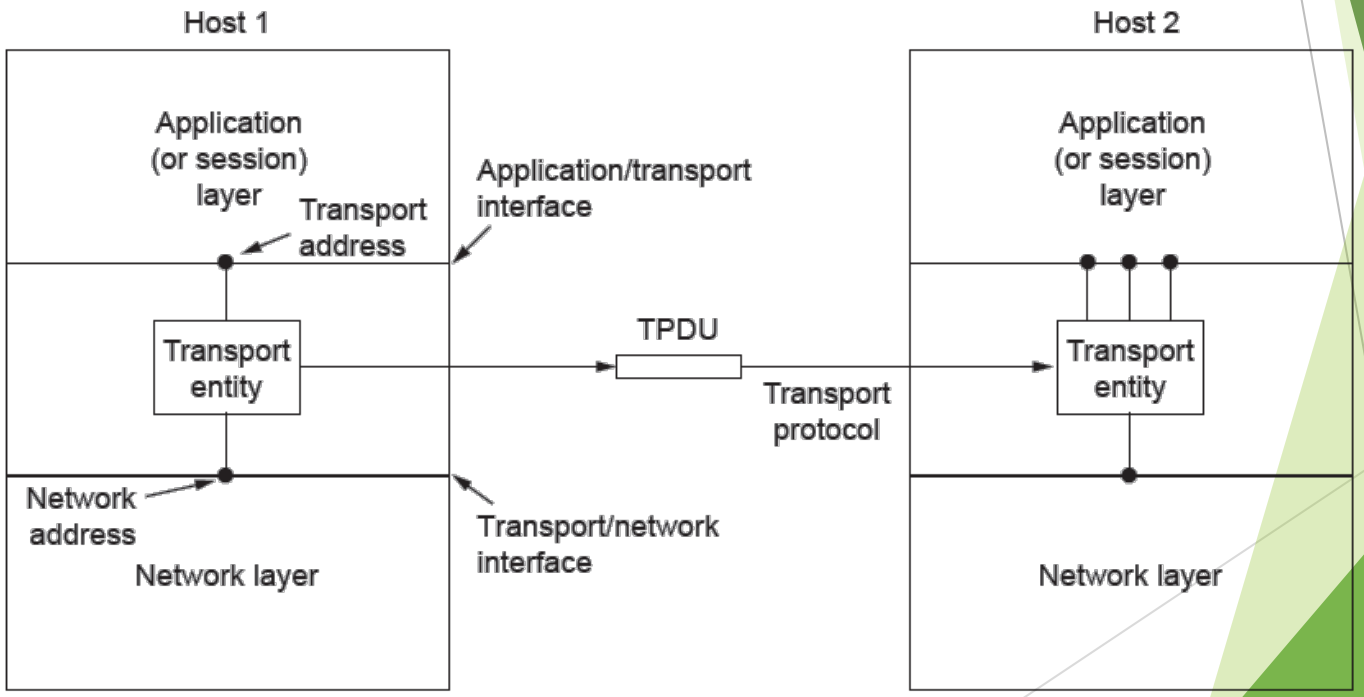
- Services Provided to the Upper Layer »
- Transport Service Primitives »
- Berkeley Sockets »
- Socket Example: Internet File Server »

Transport entity: software o hardware che implementa il transport layer, solitamente all'interno del kernel del sistema operativo.Cosí come nel network layer, es

Services Provided to the Upper Layers (1)

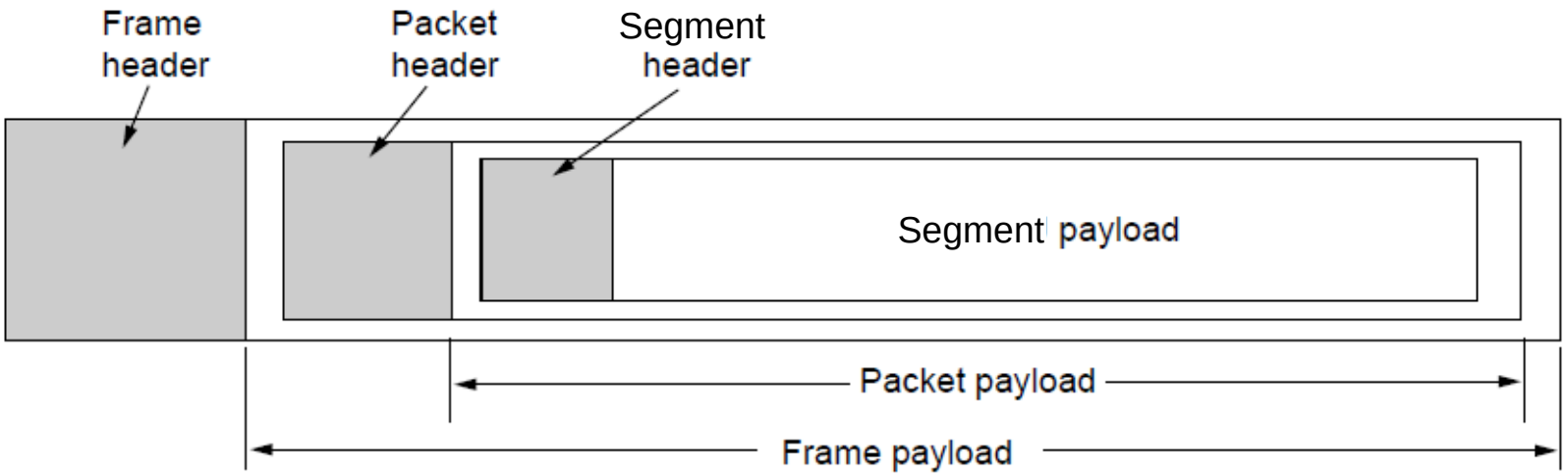
Transport layer adds reliability to the network layer

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



Services Provided to the Upper Layers (2)

Transport layer sends segments in packets (in frames)



Le reti reali, sono prone ad errori ed é compito del transport layer di renderle piú affidabili.I messaggi inoltrati da transport entity a transport entity sono chiamati segmetoi (se

Transport Service Primitives (1)

Primitives that applications might call to transport data for a simple connection-oriented service:

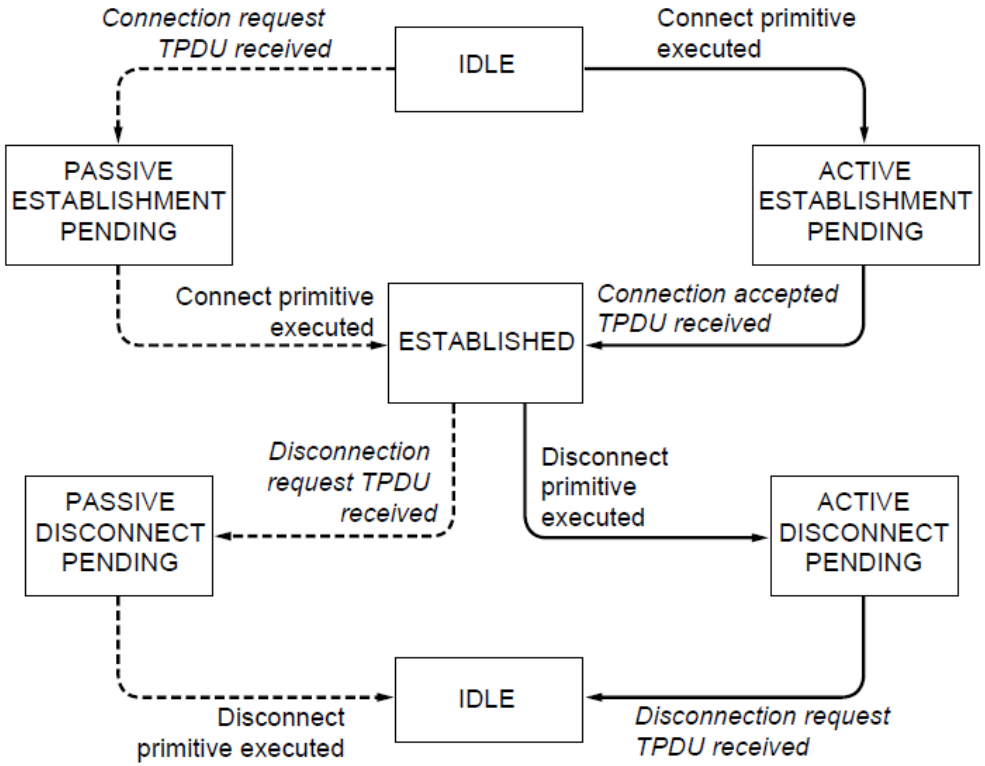
- Client calls CONNECT, SEND, RECEIVE, DISCONNECT
- Server calls LISTEN, RECEIVE, SEND, DISCONNECT

Le primitive minime ed essenziali sono: LISTEN, CONNECT, SEND, RECIVE e DISCONNECT. Tutti i protocolli di transport layer hanno queste primitieve in una forma o nell'

Primitive	Segment sent	Meaning
LISTEN	(none)	Block until some process tries to connect
CONNECT	CONNECTION REQ.	Actively attempt to establish a connection
SEND	DATA	Send information
RECEIVE	(none)	Block until a DATA packet arrives
DISCONNECT	DISCONNECTION REQ.	This side wants to release the connection

Transport Service Primitives (2)

State diagram for a simple connection-oriented service



Solid lines (right) show client state sequence

Dashed lines (left) show server state sequence

Transitions in italics are due to segment arrivals.

Berkeley Sockets

Very widely used primitives started with TCP on UNIX

- Notion of "sockets" as transport endpoints
- Like simple set plus SOCKET, BIND, and ACCEPT

Sia il server side che il client side, le primitive sono eseguite in quest'ordine.

	Primitive	Meaning
Server side	SOCKET	Create a new communication end point
	BIND	Associate a local address with a socket
	LISTEN	Announce willingness to accept connections; give queue size
	ACCEPT	Passively establish an incoming connection
Client Side	CONNECT	Actively attempt to establish a connection
	SEND	Send some data over the connection
	RECEIVE	Receive some data from the connection
	CLOSE	Release the connection

Socket Example – Internet File Server (1)

Client code

```
. . .

if (argc != 3) fatal("Usage: client server-name file-name");
h = gethostbyname(argv[1]);
if (!h) fatal("gethostbyname failed");

s = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
if (s <0) fatal("socket");
memset(&channel, 0, sizeof(channel));
channel.sin_family= AF_INET;
memcpy(&channel.sin_addr.s_addr, h->h_addr, h->h_length);
channel.sin_port= htons(SERVER_PORT);

c = connect(s, (struct sockaddr *) &channel, sizeof(channel));
if (c < 0) fatal("connect failed");

. . .
```

Get server's IP address

Make a socket

Try to connect

Socket Example – Internet File Server (2)

Client code (cont.)

. . .

```
write(s, argv[2], strlen(argv[2])+1);

while (1) {
    bytes = read(s, buf, BUF_SIZE);
    if (bytes <= 0) exit(0);
    write(1, buf, bytes);
  }
}
```

Write data (equivalent to send)

Loop reading (equivalent to receive) until no more data; exit implicitly calls close

Socket Example – Internet File Server (3)

Server code

```
. . .

memset(&channel, 0, sizeof(channel));
channel.sin_family = AF_INET;
channel.sin_addr.s_addr = htonl(INADDR_ANY);
channel.sin_port = htons(SERVER_PORT);


s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
if (s < 0) fatal("socket failed");
setsockopt(s, SOL_SOCKET, SO_REUSEADDR, (char *) &on, sizeof(on));

b = bind(s, (struct sockaddr *) &channel, sizeof(channel));
if (b < 0) fatal("bind failed");

l = listen(s, QUEUE_SIZE);
if (l < 0) fatal("listen failed");

. . .
```

Make a socket

Assign address

Prepare for incoming connections

Socket Example – Internet File Server (4)

Server code

```
. . .
while (1) {
    sa = accept(s, 0, 0);
    if (sa < 0) fatal("accept failed");

    read(sa, buf, BUF_SIZE);

    /* Get and return the file. */
    fd = open(buf, O_RDONLY);
    if (fd < 0) fatal("open failed");

    while (1) {
        bytes = read(fd, buf, BUF_SIZE);
        if (bytes <= 0) break;
        write(sa, buf, bytes);
    }
    close(fd);
    close(sa);
}
}
```

Block waiting for the next connection

Read (receive) request and treat as file name

Write (send) all file data

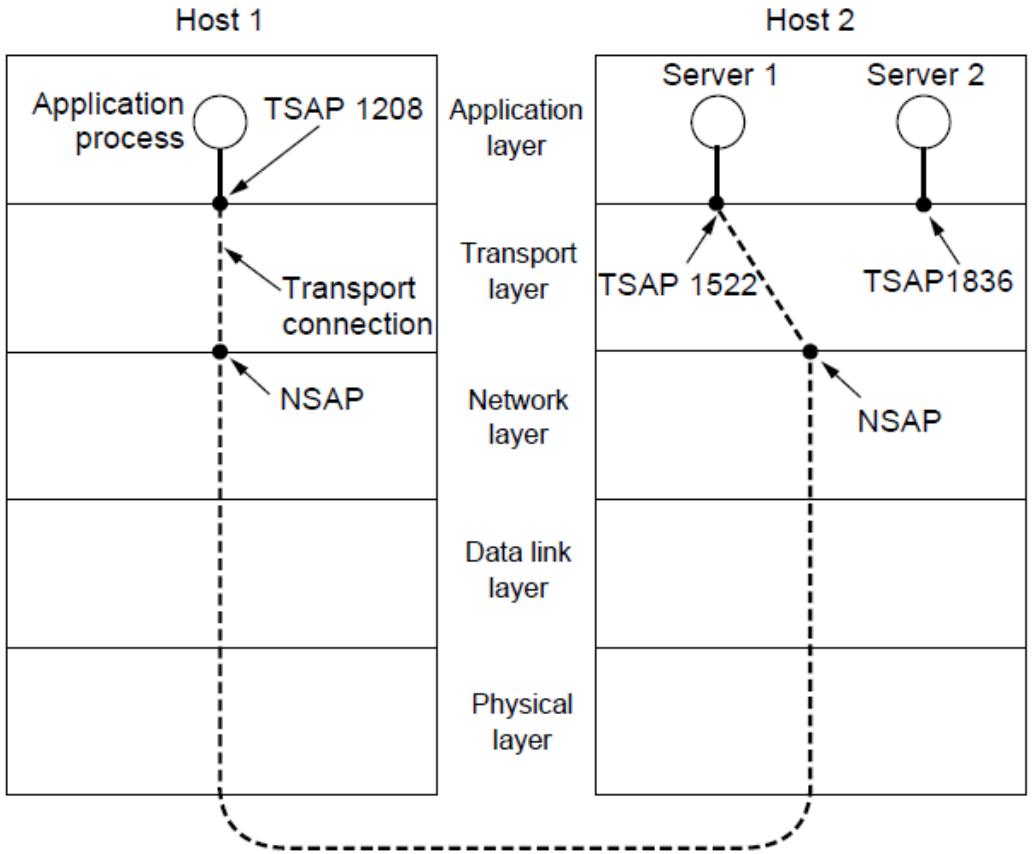
Done, so close this connection

Elements of Transport Protocols

- Addressing »
- Connection establishment »
- Connection release »
- Error control and flow control »
- Multiplexing »
- Crash recovery »

Addressing

- Transport layer adds TSAPs
- Multiple clients and servers can run on a host with a single network (IP) address
- TSAPs are ports for TCP/UDP



Connection Establishment (1)

Key problem is to ensure reliability even though packets may be lost, corrupted, delayed, and duplicated

- Don't treat an old or duplicate packet as new
- (Use ARQ and checksums for loss/corruption)

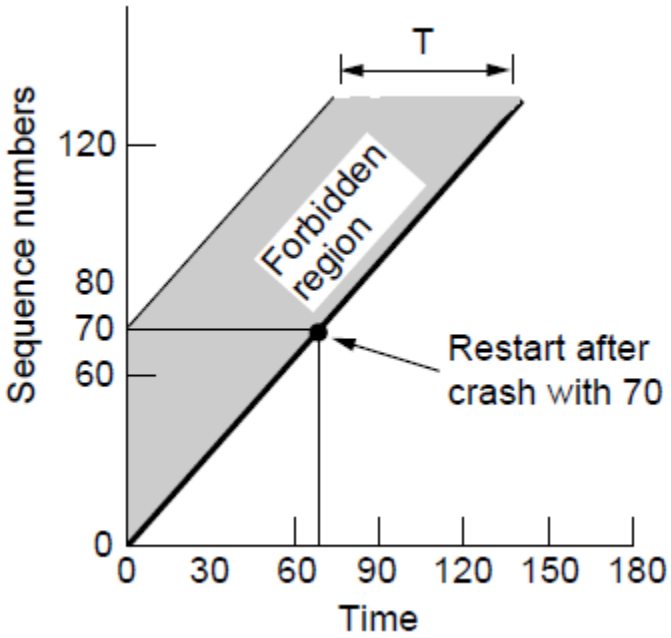
Approach:

- Don't reuse sequence numbers within twice the MSL (Maximum Segment Lifetime) of 2T=240 secs
- Three-way handshake for establishing connection

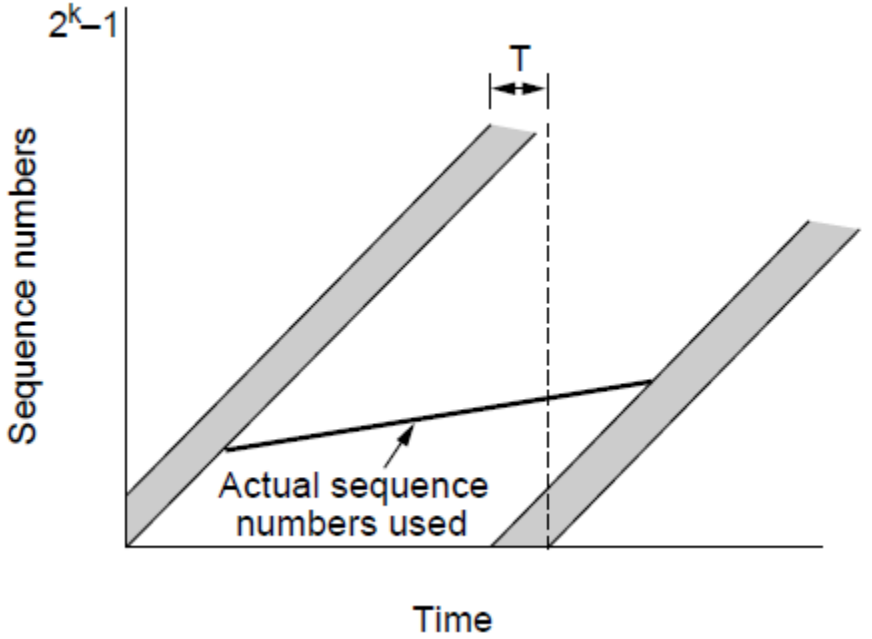
Connection Establishment (2)

Use a sequence number space large enough that it will not wrap, even when sending at full rate

- Clock (high bits) advances & keeps state over crash



Need seq. number not to wrap within T seconds

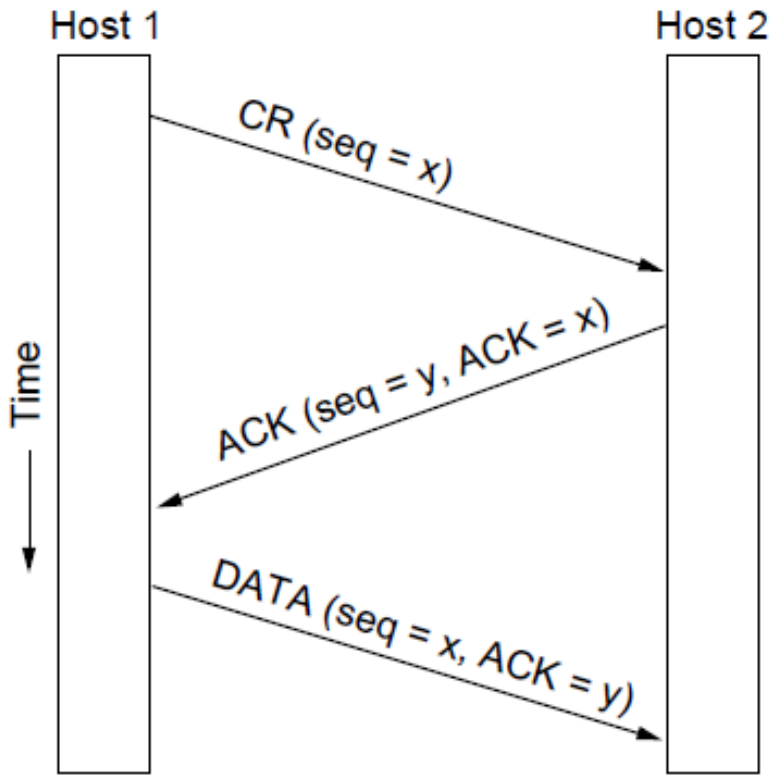


Need seq. number not to climb too slowly for too long

Connection Establishment (3)

Three-way handshake used for initial packet

- Since no state from previous connection
- Both hosts contribute fresh seq. numbers
- CR = Connect Request

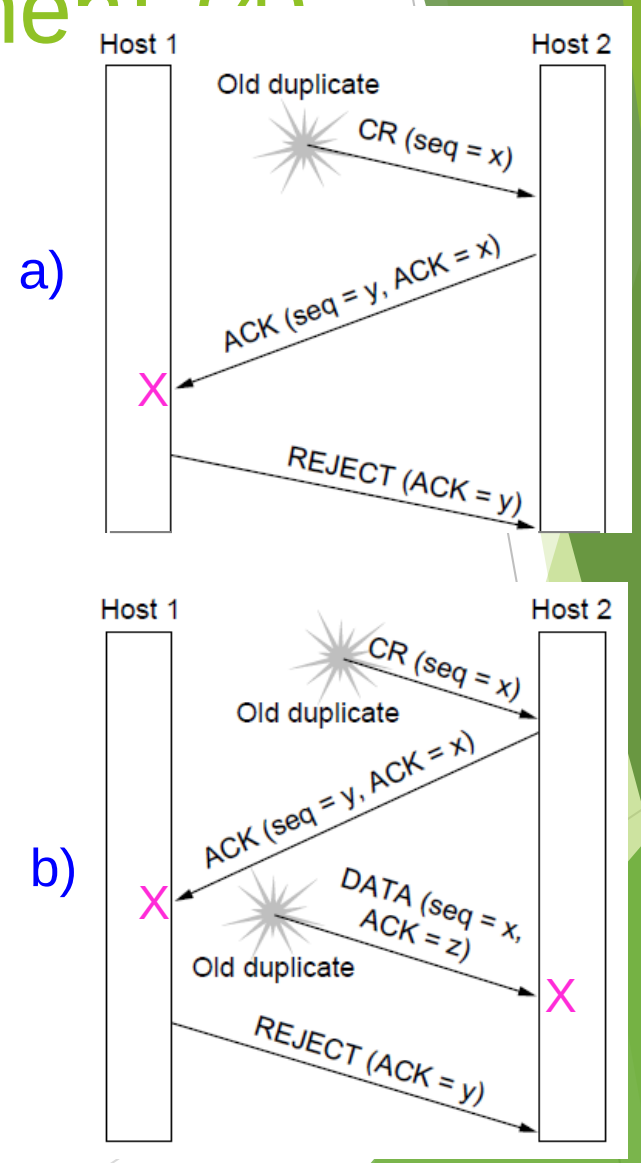


Connection Establishment (4)

Three-way handshake protects against odd cases:

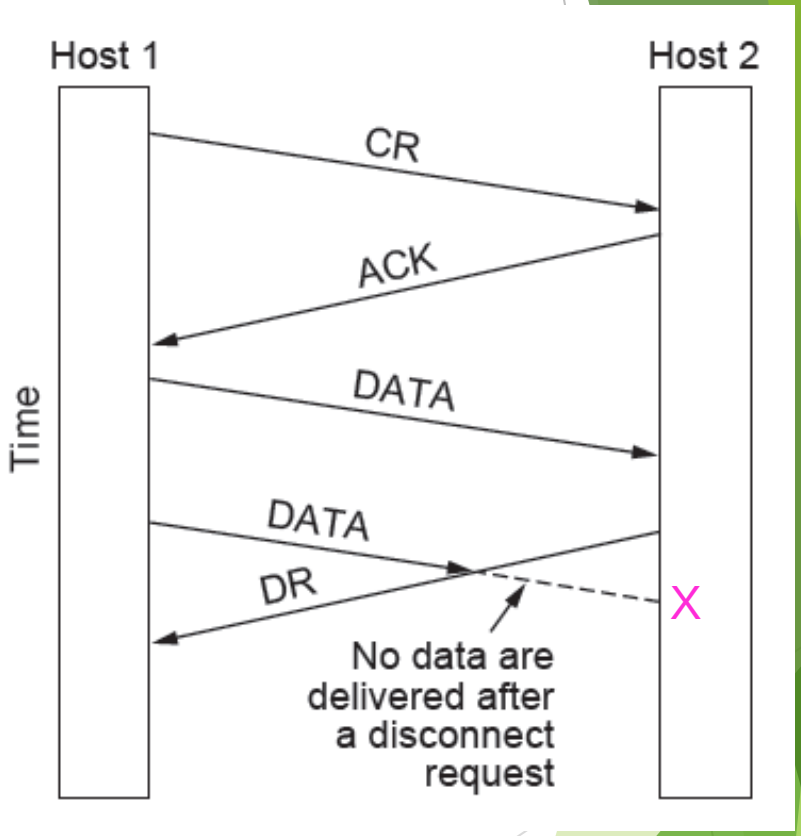
a) Duplicate CR. Spurious ACK does not connect

b) Duplicate CR and DATA. Same plus DATA will be rejected (wrong ACK).



Connection Release (1)

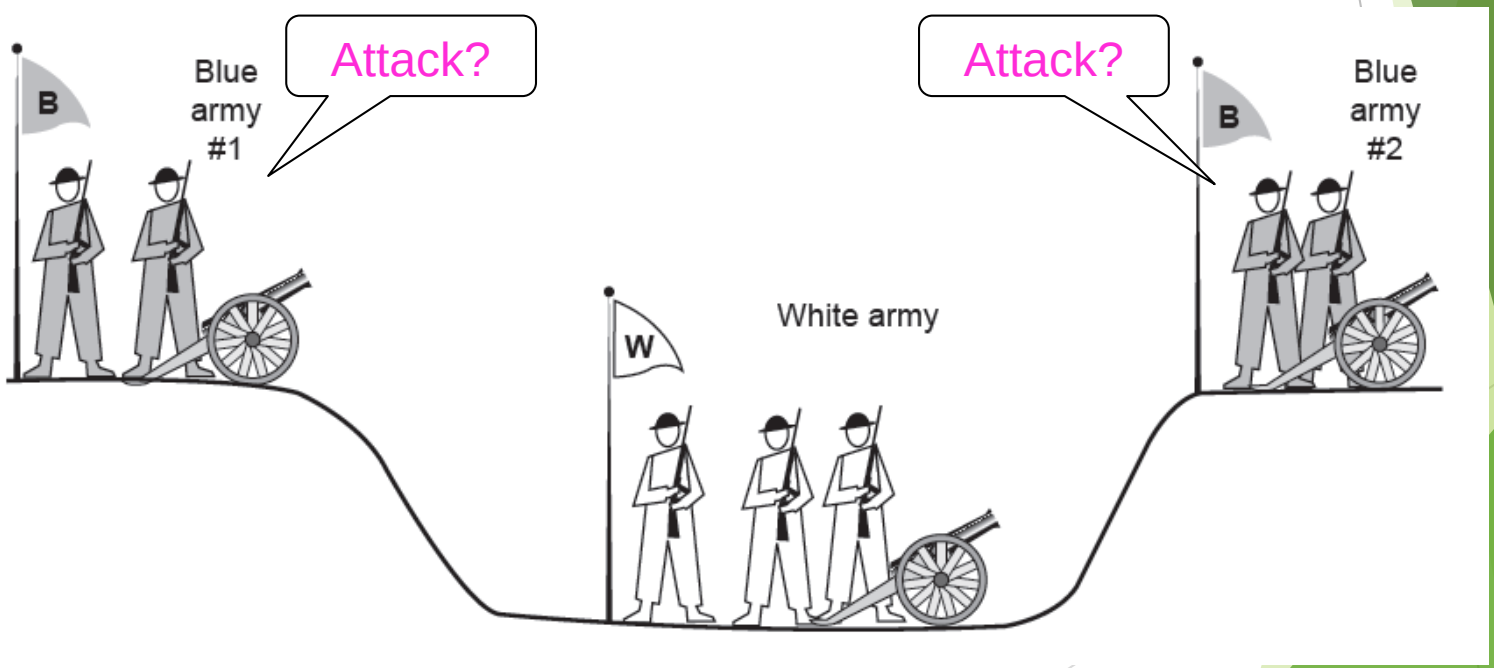
- Key problem is to ensure reliability while releasing
- Asymmetric release (when one side breaks connection) is abrupt and may lose data



Connection Release (2)

Symmetric release (both sides agree to release) can't be handled solely by the transport layer

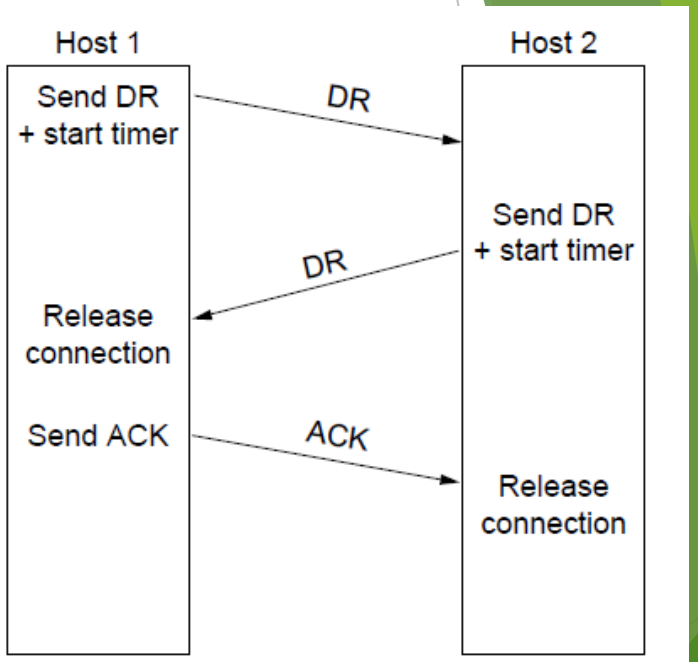
- Two-army problem shows pitfall of agreement



Connection Release (3)

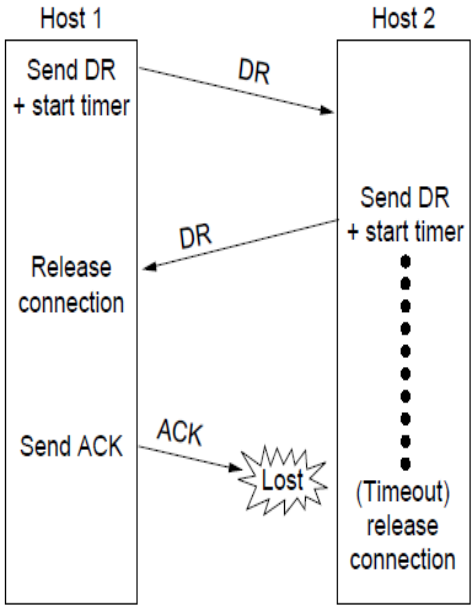
Normal release sequence, initiated by transport user on Host 1

- DR=Disconnect Request
- Both DRs are ACKed by the other side

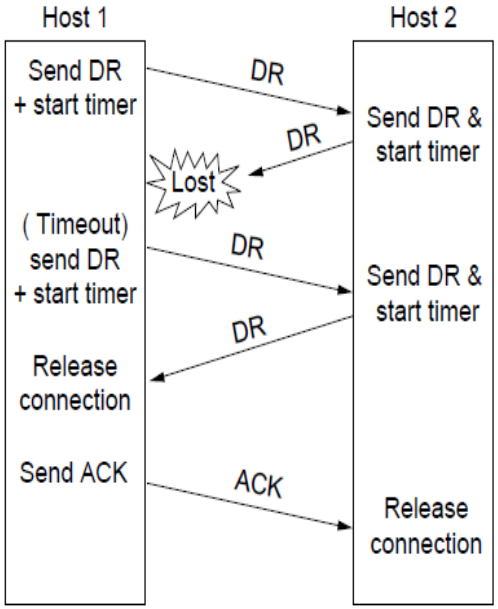


Connection Release (4)

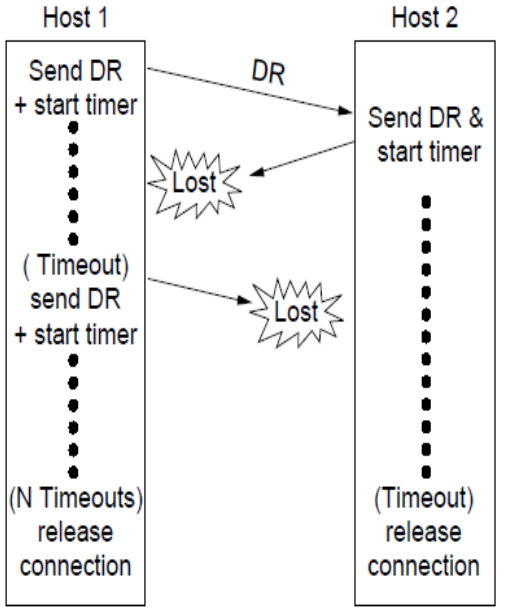
Error cases are handled with timer and retransmission



Final ACK lost, Host 2 times out



Lost DR causes retransmissions



Extreme: Many lost DRs cause both hosts to timeout

Error Control and Flow Control (1)

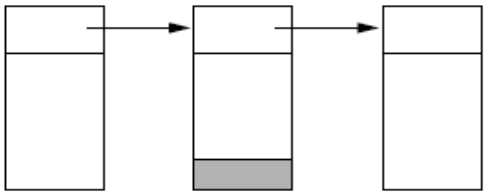
Foundation for error control is a sliding window (from Link layer) with checksums and retransmissions

Flow control manages buffering at sender/receiver

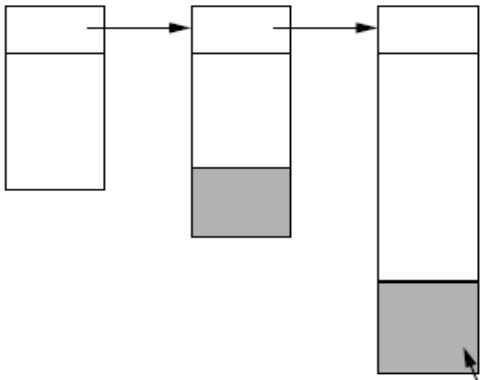
- Issue is that data goes to/from the network and applications at different times
- Window tells sender available buffering at receiver
- Makes a variable-size sliding window

Error Control and Flow Control (2)

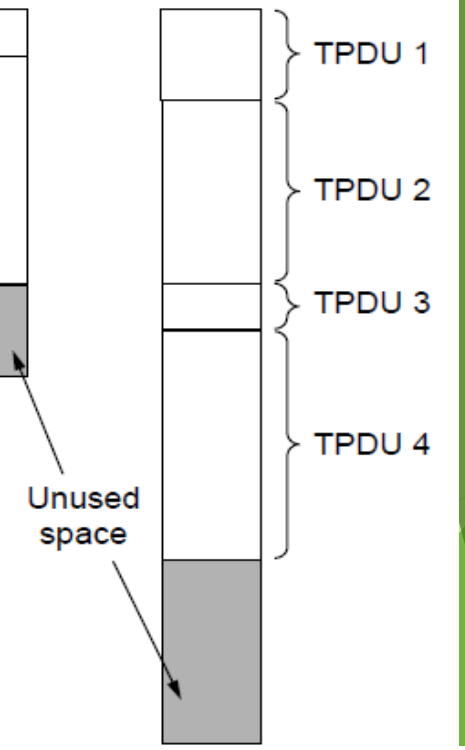
- Different buffer strategies trade efficiency / complexity



a) Chained fixed-size buffers



b) Chained variable-size buffers



c) One large circular buffer

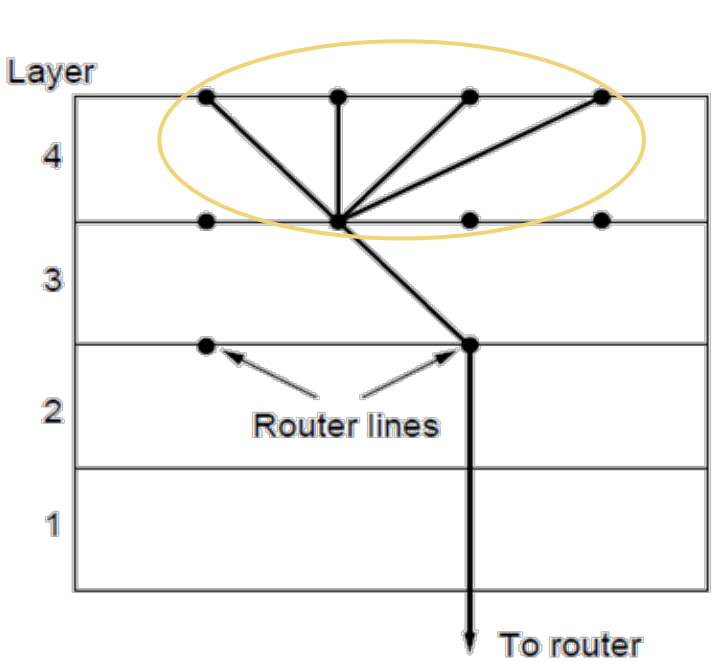
Error Control and Flow Control (3)

- Flow control example: A's data is limited by B's buffer

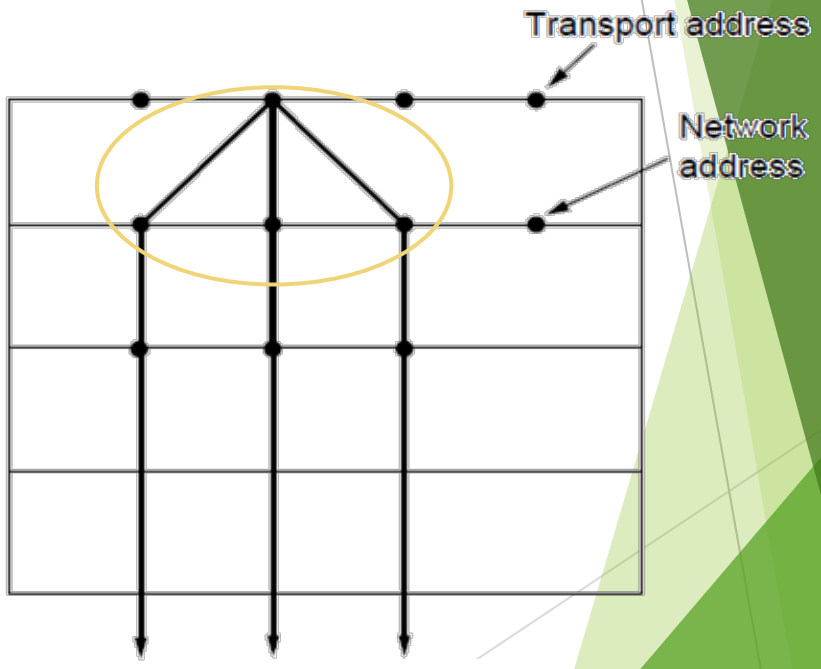
	A	Message	B	B's Buffer	Comments
1	→	< request 8 buffers>	→		A wants 8 buffers
2	←	<ack = 15, buf = 4>	←	0 1 2 3	B grants messages 0-3 only
3	→	<seq = 0, data = m0>	→	0 1 2 3	A has 3 buffers left now
4	→	<seq = 1, data = m1>	→	0 1 2 3	A has 2 buffers left now
5	→	<seq = 2, data = m2>	• • •	0 1 2 3	Message lost but A thinks it has 1 left
6	←	<ack = 1, buf = 3>	←	1 2 3 4	B acknowledges 0 and 1, permits 2-4
7	→	<seq = 3, data = m3>	→	1 2 3 4	A has 1 buffer left
8	→	<seq = 4, data = m4>	→	1 2 3 4	A has 0 buffers left, and must stop
9	→	<seq = 2, data = m2>	→	1 2 3 4	A times out and retransmits
10	←	<ack = 4, buf = 0>	←	1 2 3 4	Everything acknowledged, but A still blocked
11	←	<ack = 4, buf = 1>	←	2 3 4 5	A may now send 5
12	←	<ack = 4, buf = 2>	←	3 4 5 6	B found a new buffer somewhere
13	→	<seq = 5, data = m5>	→	3 4 5 6	A has 1 buffer left
14	→	<seq = 6, data = m6>	→	3 4 5 6	A is now blocked again
15	←	<ack = 6, buf = 0>	←	3 4 5 6	A is still blocked
16	• • •	<ack = 6, buf = 4>	←	7 8 9 10	Potential deadlock

Multiplexing

- Kinds of transport / network sharing that can occur:
  - Multiplexing: connections share a network address
  - Inverse multiplexing: addresses share a connection



Multiplexing



Inverse Multiplexing

Crash Recovery

Application needs to help recovering from a crash

- Transport can fail since A(ck) / W(rite) not atomic

Strategy used by sending host	Strategy used by receiving host: First ACK, then write			First write, then ACK		
	AC(W)	AWC	C(AW)	C(WA)	W AC	WC(A)
Always retransmit	OK	DUP	OK	OK	DUP	DUP
Never retransmit	LOST	OK	LOST	LOST	OK	OK
Retransmit in S0	OK	DUP	LOST	LOST	DUP	OK
Retransmit in S1	LOST	OK	OK	OK	OK	DUP

OK = Protocol functions correctly
DUP = Protocol generates a duplicate message
LOST = Protocol loses a message

Congestion Control

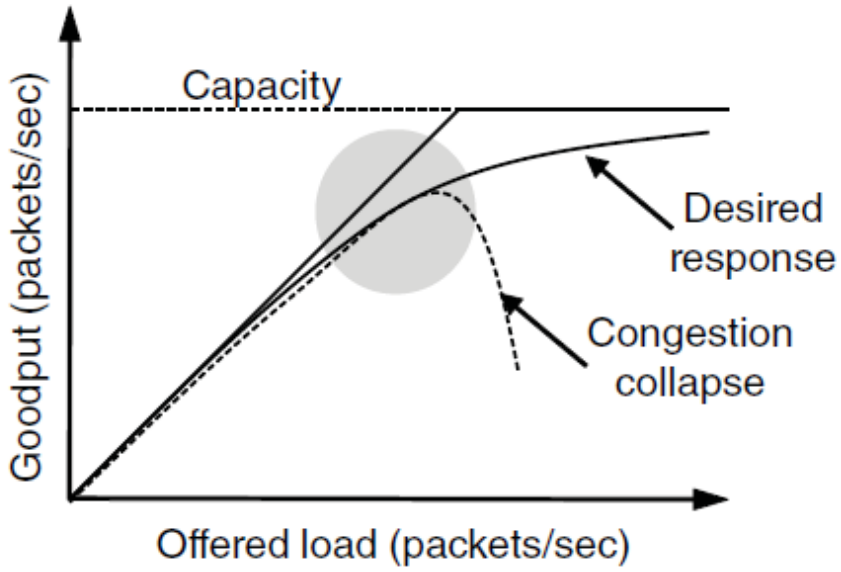
Two layers are responsible for congestion control:

- Transport layer, controls the offered load [here]
- Network layer, experiences congestion [previous]

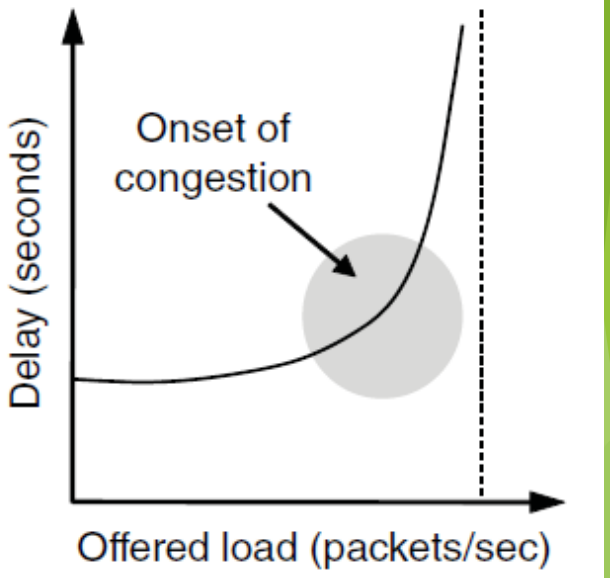
- Desirable bandwidth allocation »
- Regulating the sending rate »
- Wireless issues »

Desirable Bandwidth Allocation (1)

Efficient use of bandwidth gives high goodput, low delay



Goodput rises more slowly than load when congestion sets in

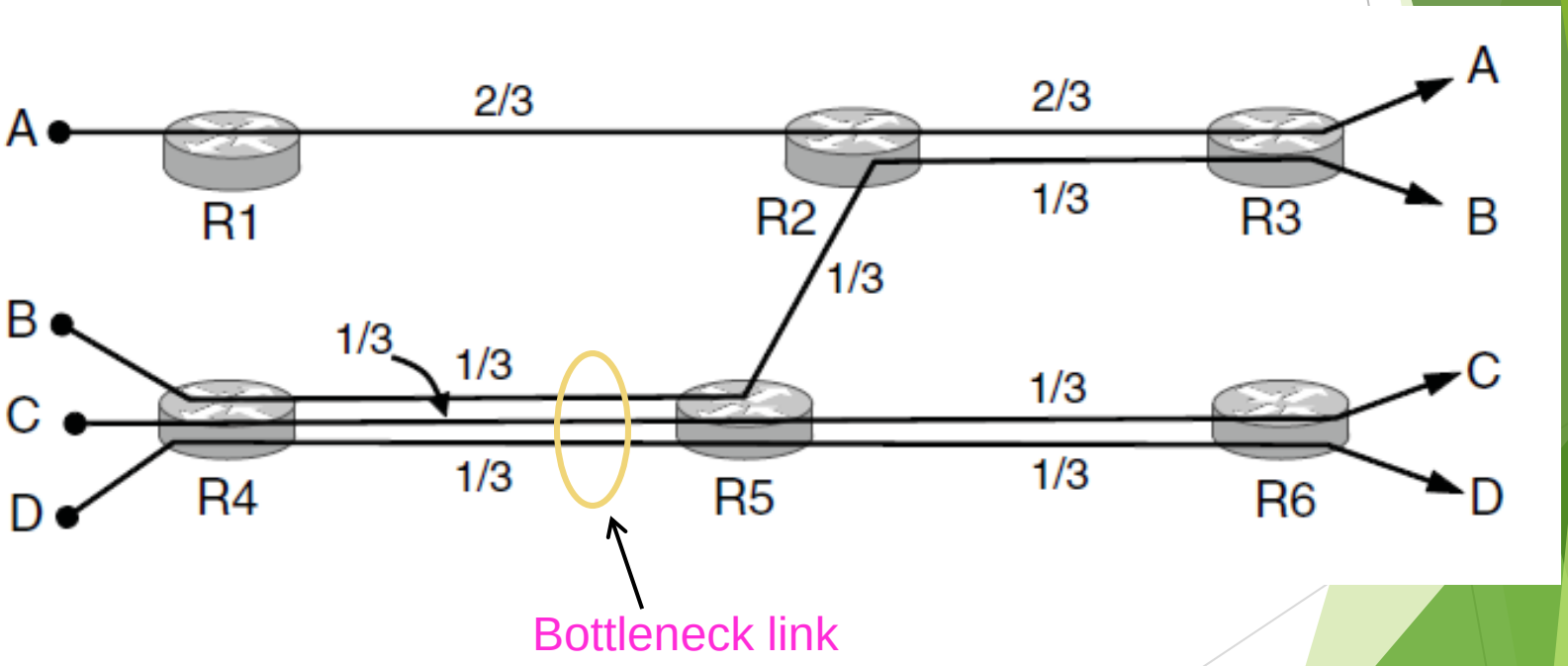


Delay begins to rise sharply when congestion sets in

Desirable Bandwidth Allocation (2)

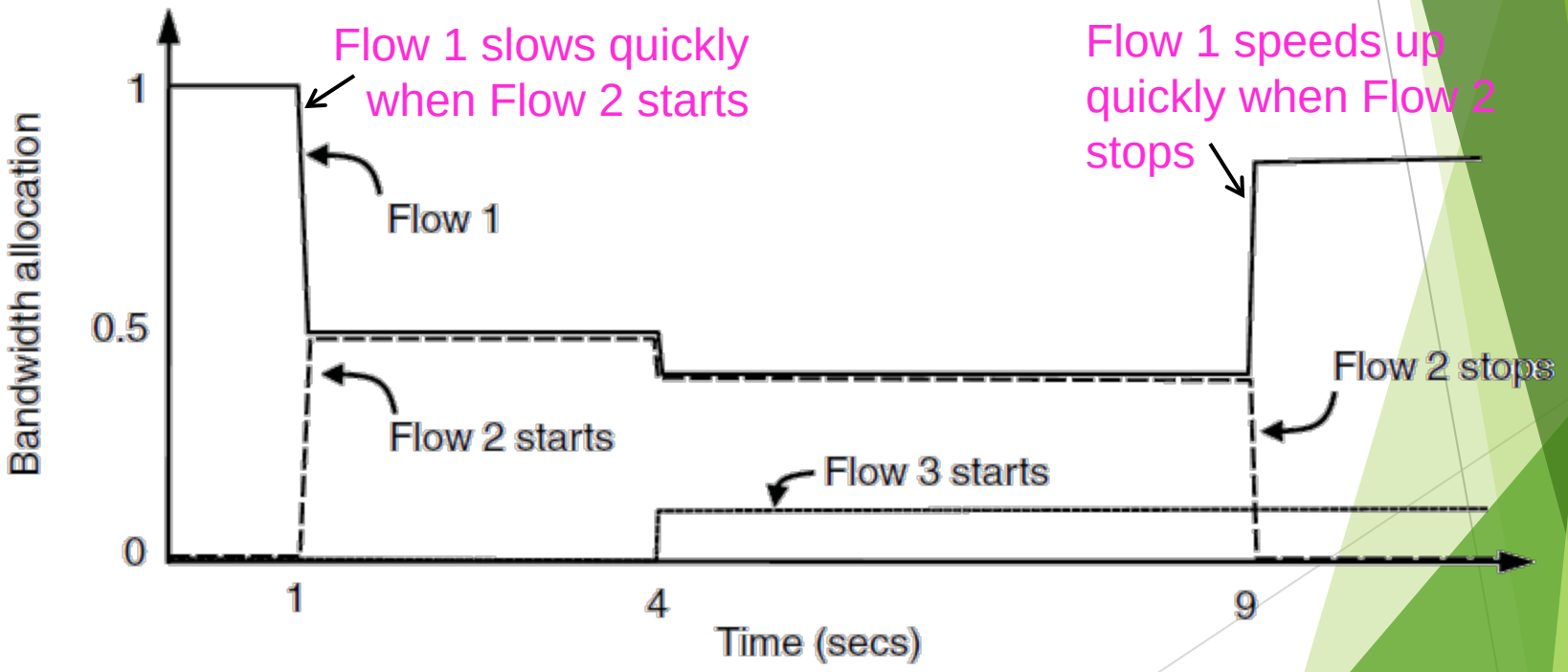
Fair use gives bandwidth to all flows (no starvation)

- Max-min fairness gives equal shares of bottleneck



Desirable Bandwidth Allocation (3)

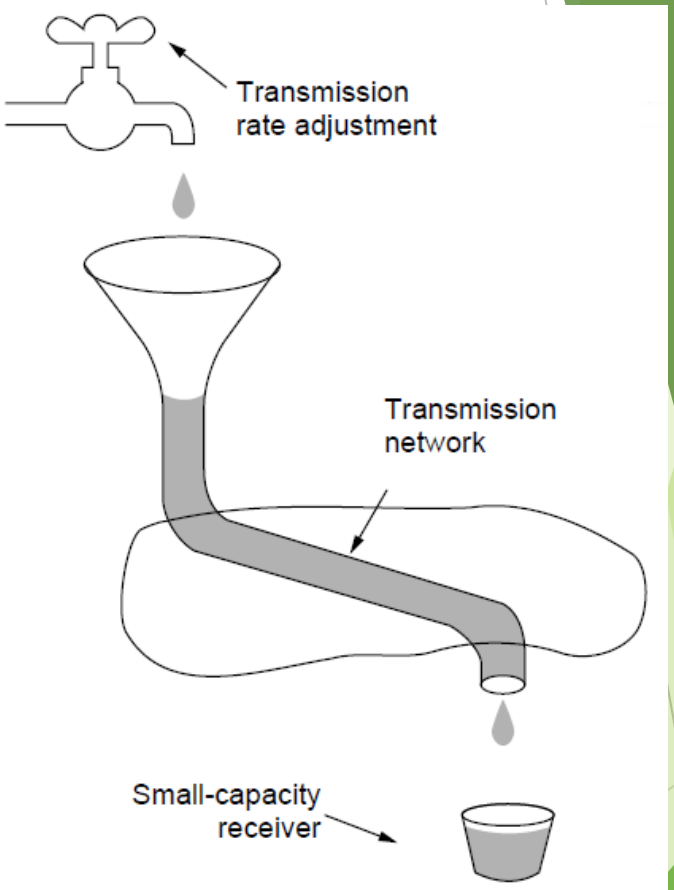
We want bandwidth levels to converge quickly when traffic patterns change



Regulating the Sending Rate (1)

Sender may need to slow down for different reasons:

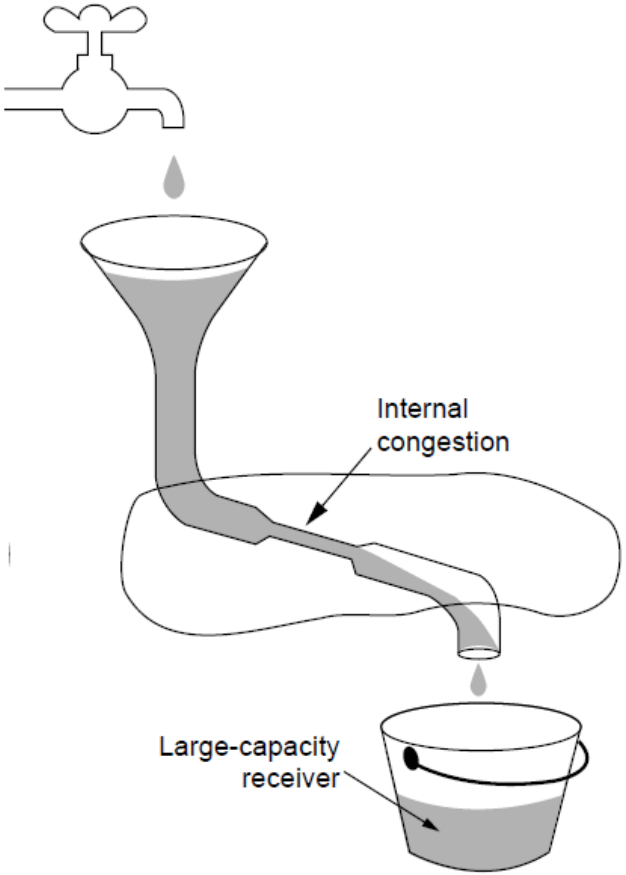
- Flow control, when the receiver is not fast enough [right]
- Congestion, when the network is not fast enough [over]



A fast network feeding a low-capacity receiver → flow control is needed

Regulating the Sending Rate (2)

Our focus is dealing with this problem – congestion



A slow network feeding a high-capacity receiver
→ congestion control is needed

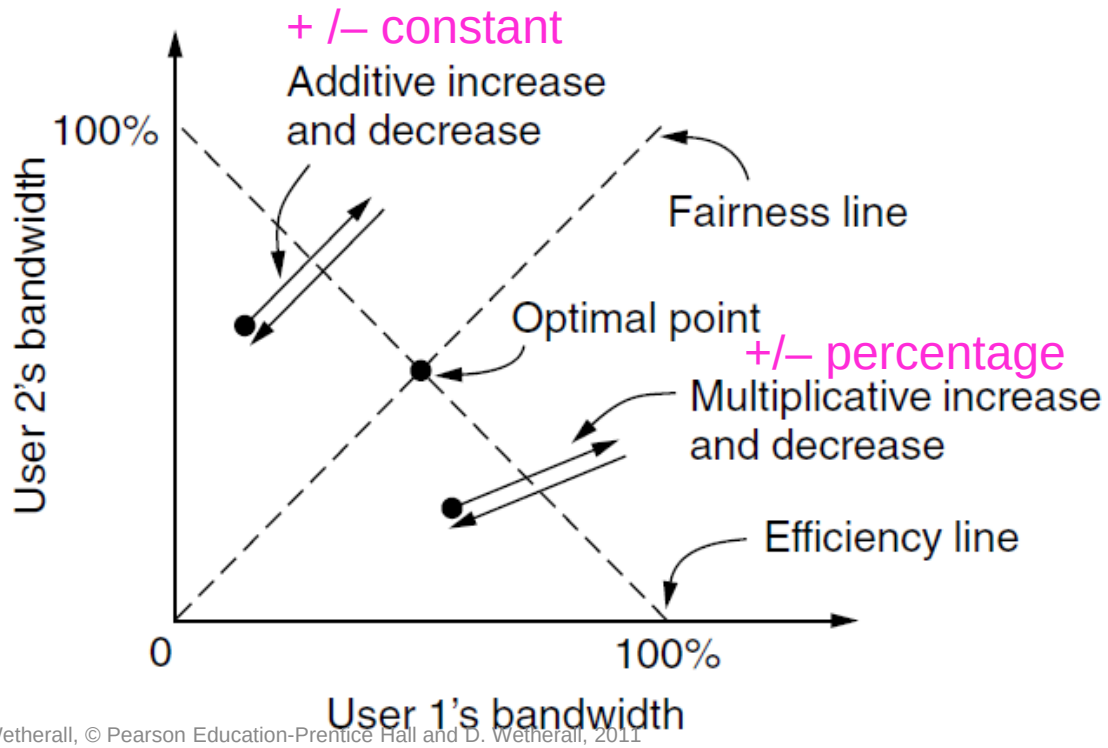
Regulating the Sending Rate (3)

- Different congestion signals the network may use to tell the transport endpoint to slow down (or speed up)

Protocol	Signal	Explicit?	Precise?
XCP	Rate to use	Yes	Yes
TCP with ECN	Congestion warning	Yes	No
FAST TCP	End-to-end delay	No	Yes
CUBIC TCP	Packet loss	No	No
TCP	Packet loss	No	No

Regulating the Sending Rate (3)

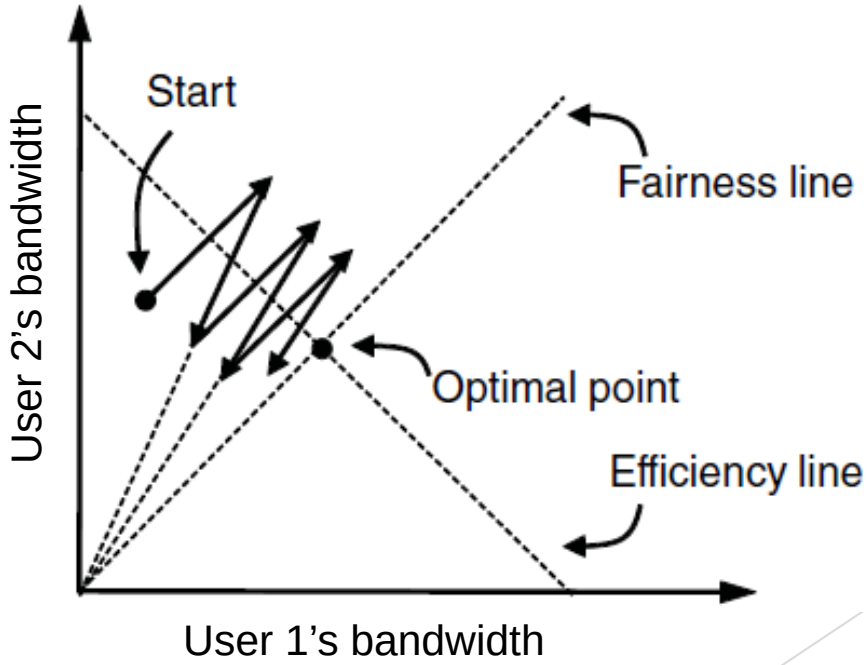
If two flows increase/decrease their bandwidth in the same way when the network signals free/busy they will not converge to a fair allocation



Regulating the Sending Rate (4)

The AIMD (Additive Increase Multiplicative Decrease) control law does converge to a fair and efficient point!

- TCP uses AIMD for this reason



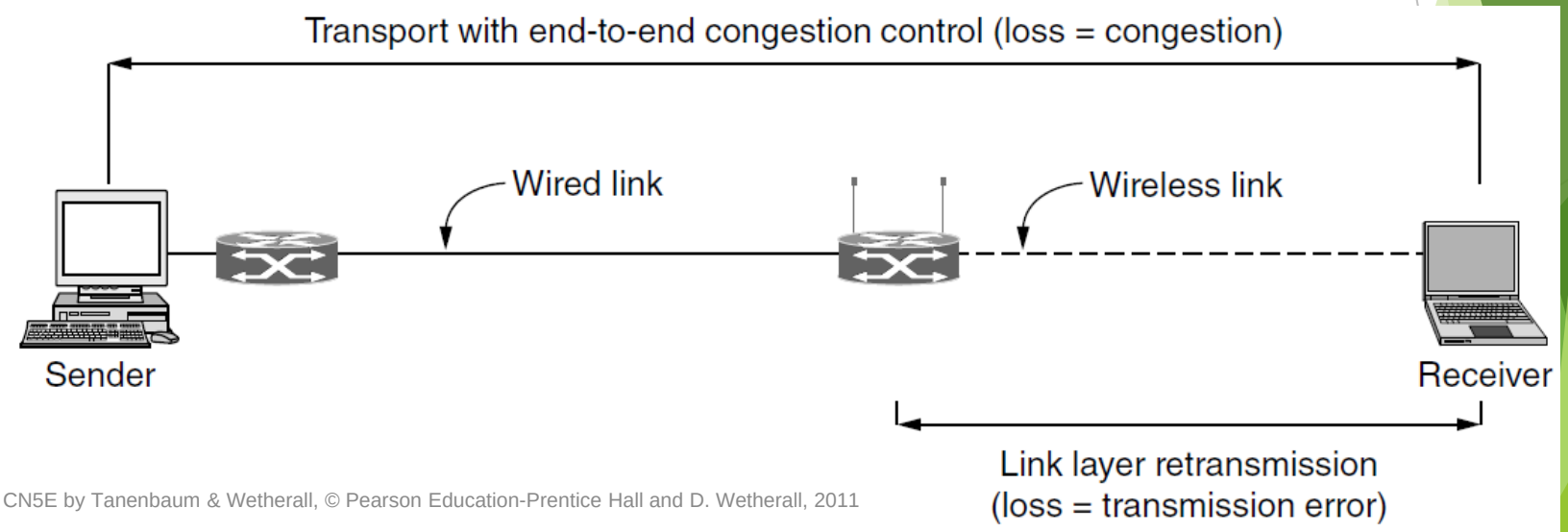
Wireless Issues

Wireless links lose packets due to transmission errors

- Do not want to confuse this loss with congestion
- Or connection will run slowly over wireless links!

Strategy:

- Wireless links use ARQ, which masks errors



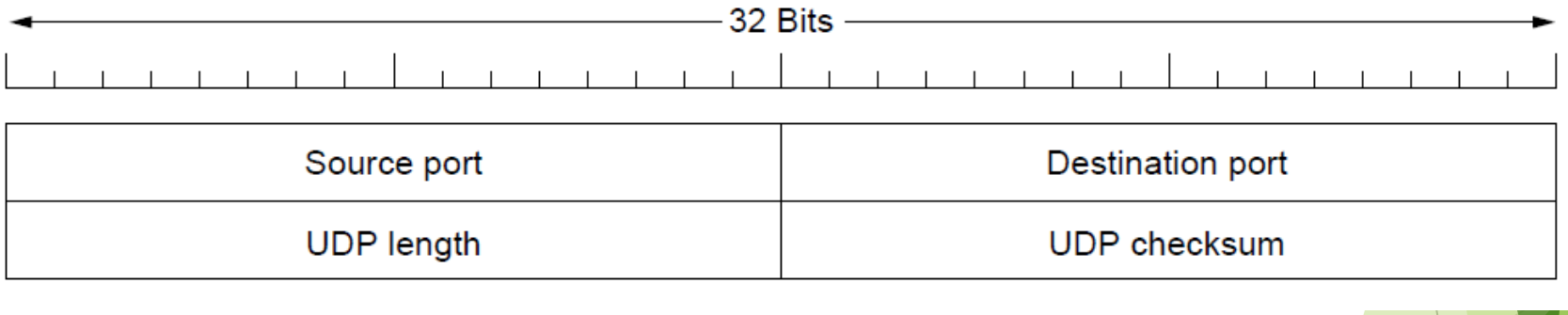
Internet Protocols – UDP

- Introduction to UDP »
- Remote Procedure Call »
- Real-Time Transport »

Introduction to UDP (1)

UDP (User Datagram Protocol) is a shim over IP

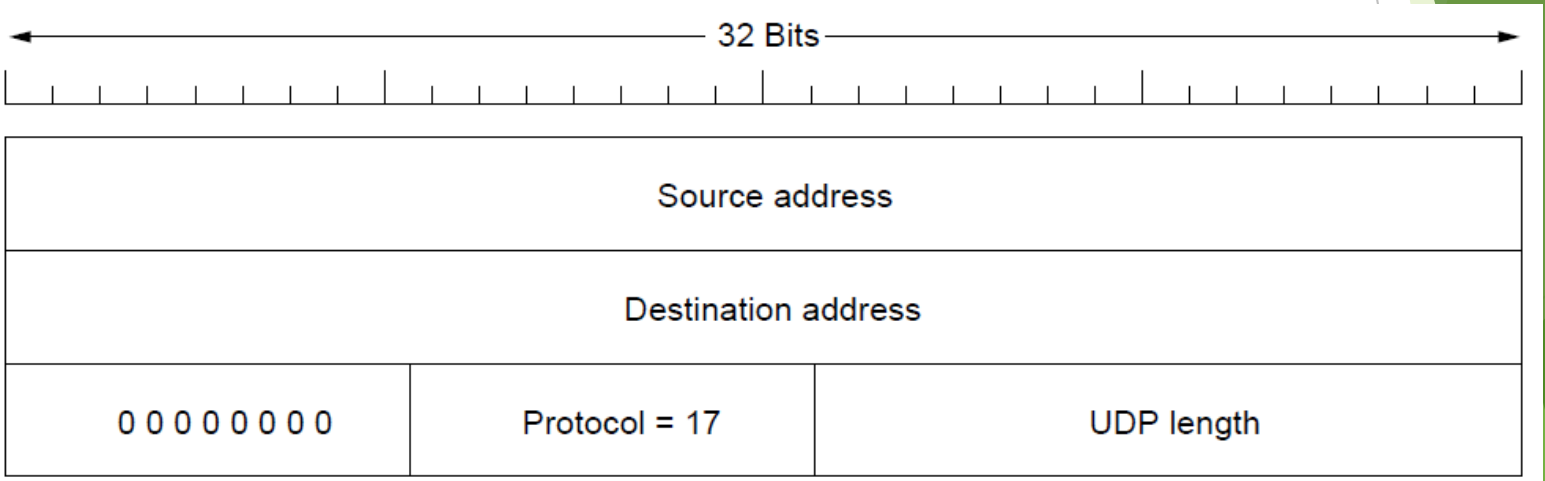
- Header has ports (TSAPs), length and checksum.



Introduction to UDP (2)

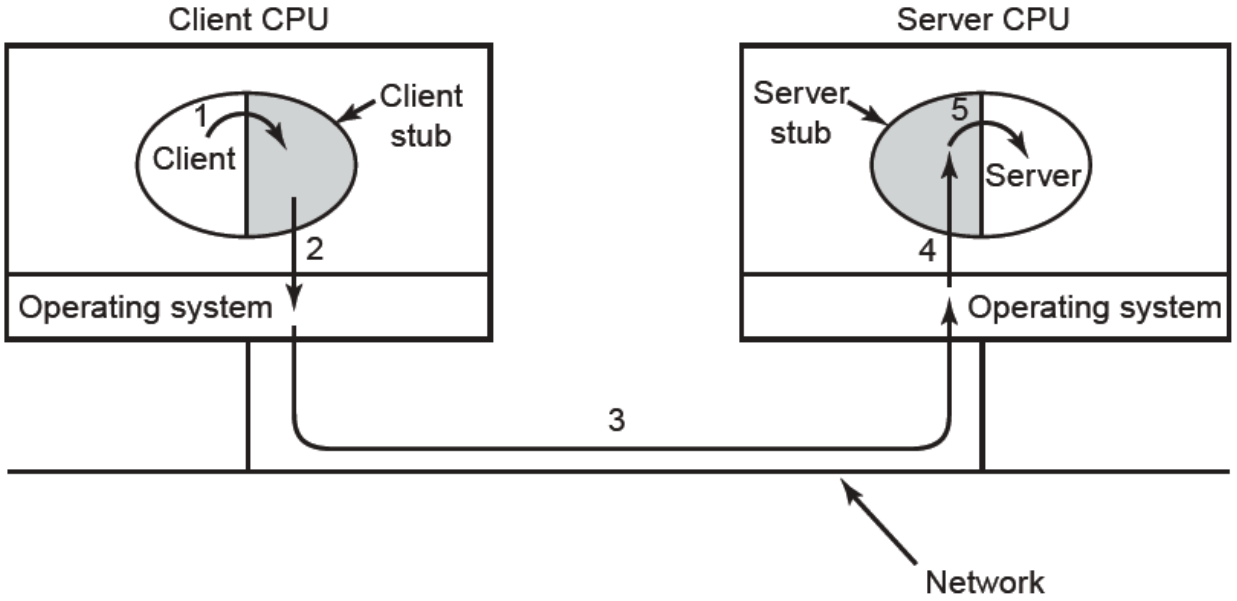
Checksum covers UDP segment and IP pseudoheader

- Fields that change in the network are zeroed out
- Provides an end-to-end delivery check



RPC (Remote Procedure Call)

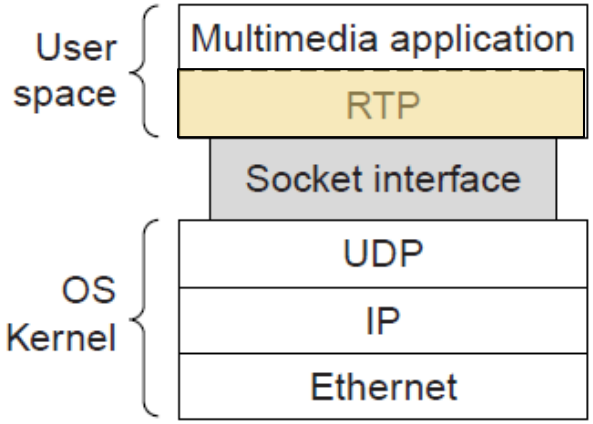
- RPC connects applications over the network with the familiar abstraction of procedure calls
  - Stubs package parameters/results into a message
  - UDP with retransmissions is a low-latency transport

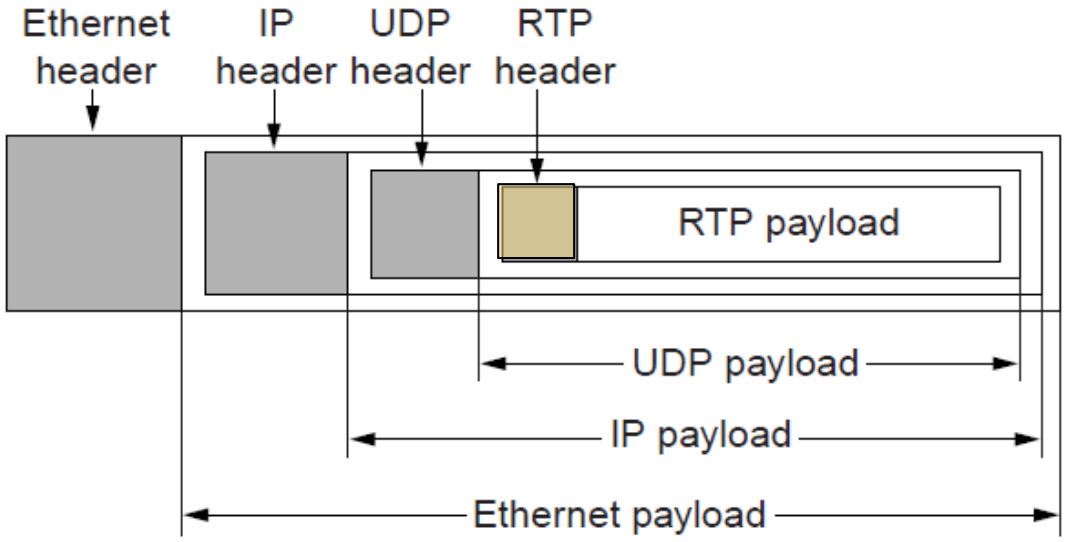


Real-Time Transport (1)

RTP (Real-time Transport Protocol) provides support for sending real-time media over UDP

- Often implemented as part of the application

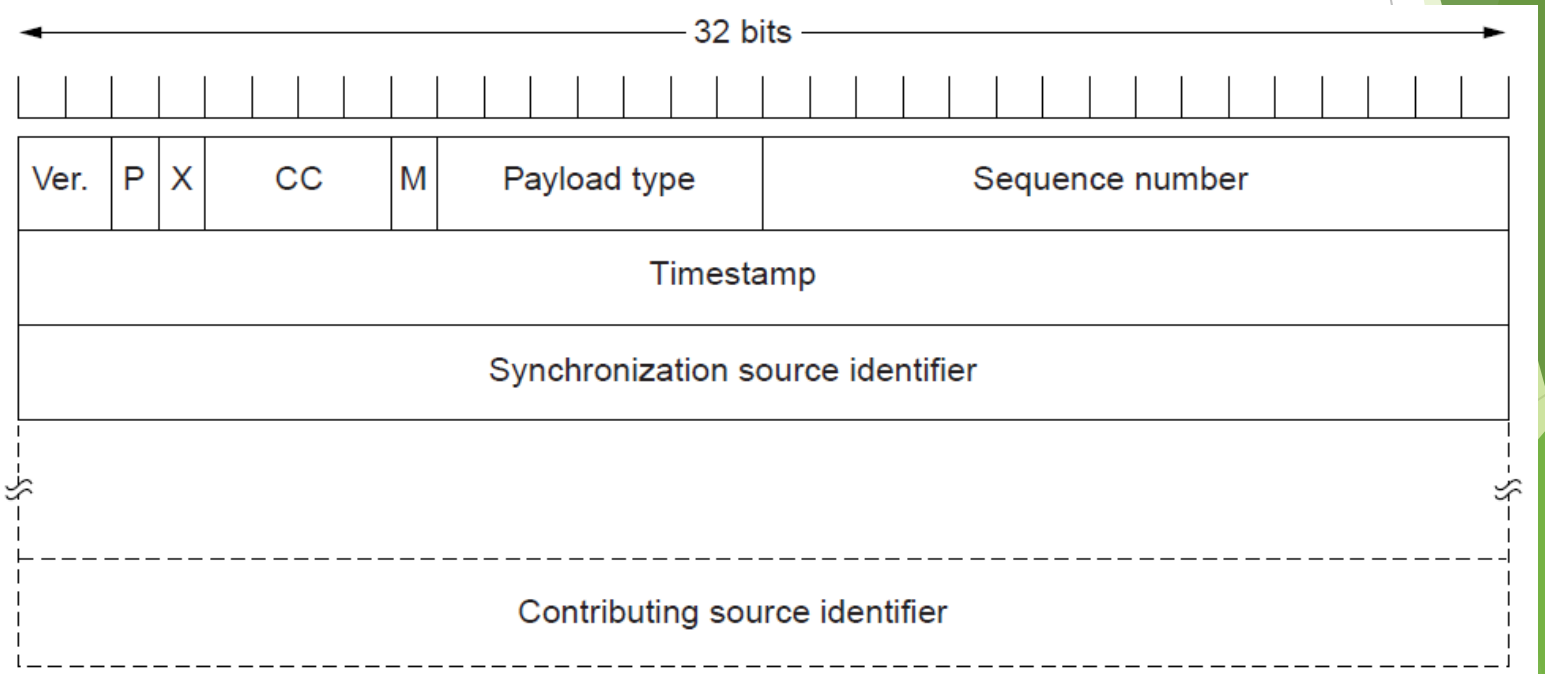




Real-Time Transport (2)

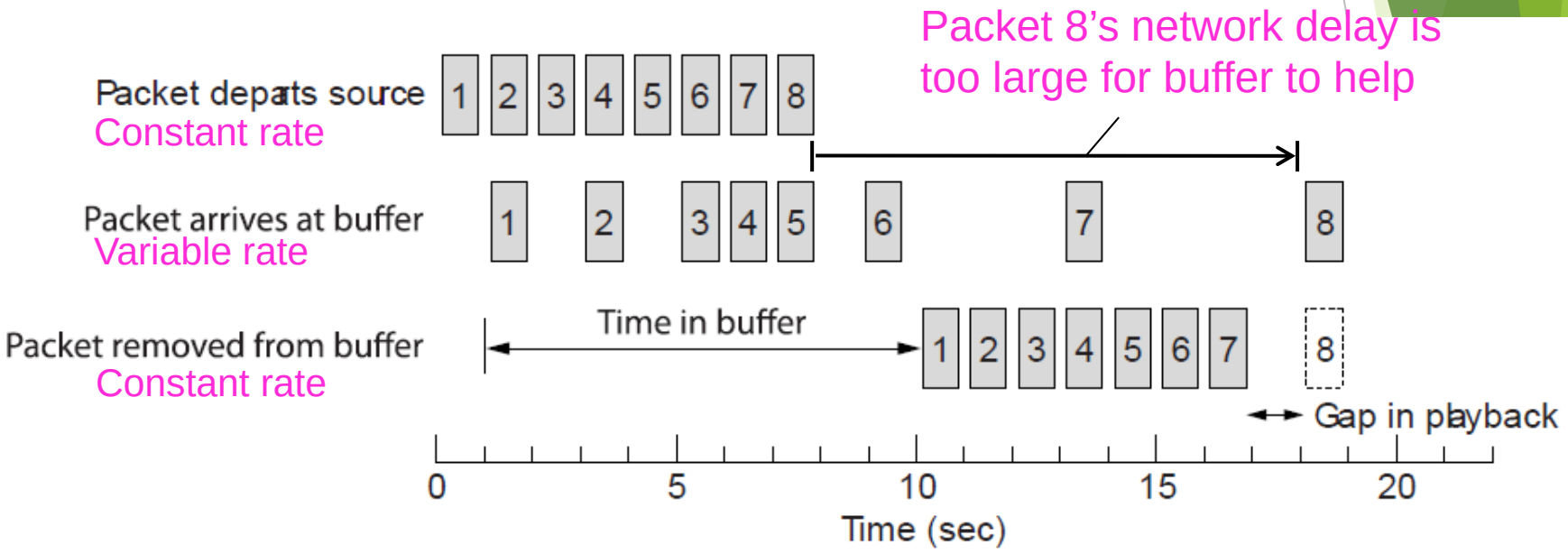
RTP header contains fields to describe the type of media and synchronize it across multiple streams

- RTCP sister protocol helps with management tasks



Real-Time Transport (3)

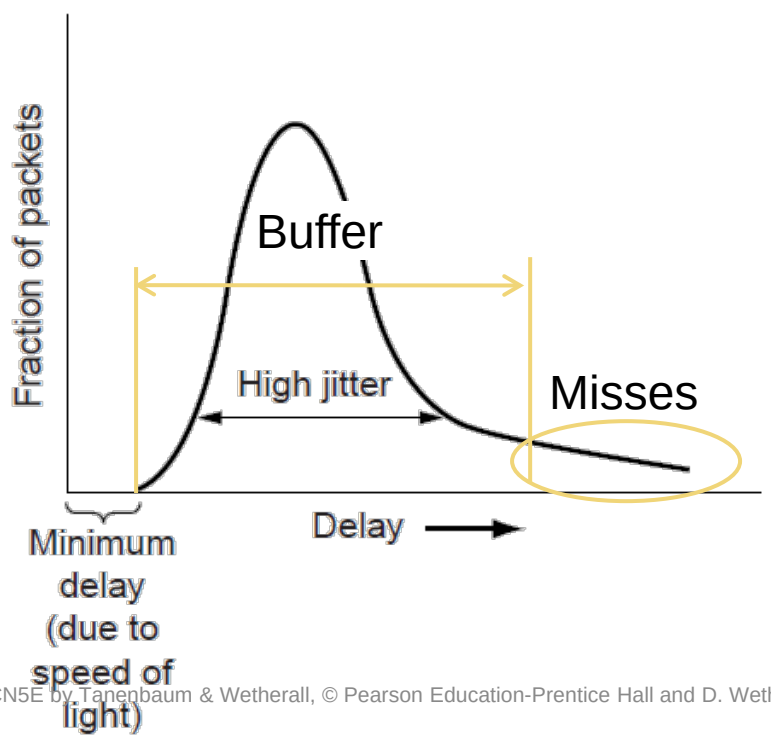
Buffer at receiver is used to delay packets and absorb jitter so that streaming media is played out smoothly

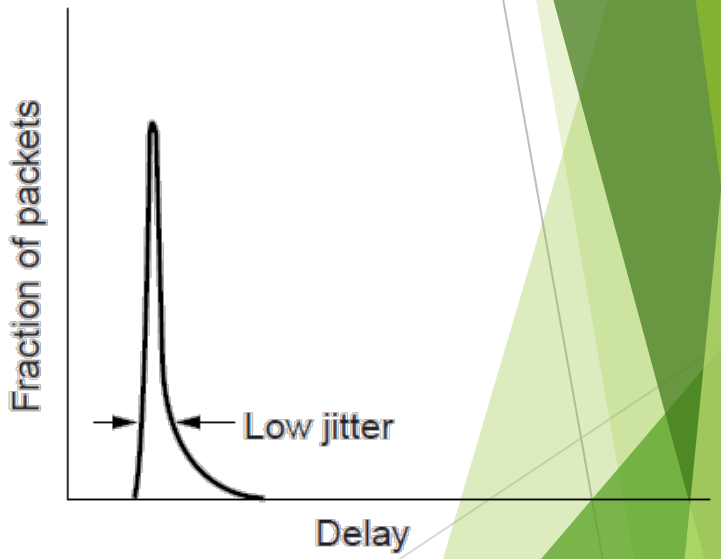


Real-Time Transport (3)

High jitter, or more variation in delay, requires a larger playout buffer to avoid playout misses

- Propagation delay does not affect buffer size





Internet Protocols – TCP

- The TCP service model »
- The TCP segment header »
- TCP connection establishment »
- TCP connection state modeling »
- TCP sliding window »
- TCP timer management »
- TCP congestion control »

The TCP Service Model (1)

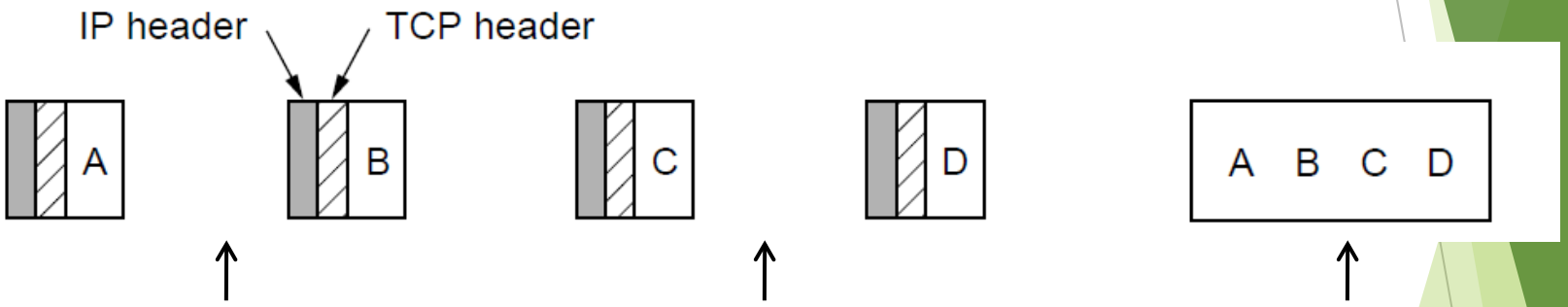
TCP provides applications with a reliable byte stream between processes; it is the workhorse of the Internet

- Popular servers run on well-known ports

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

The TCP Service Model (2)

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

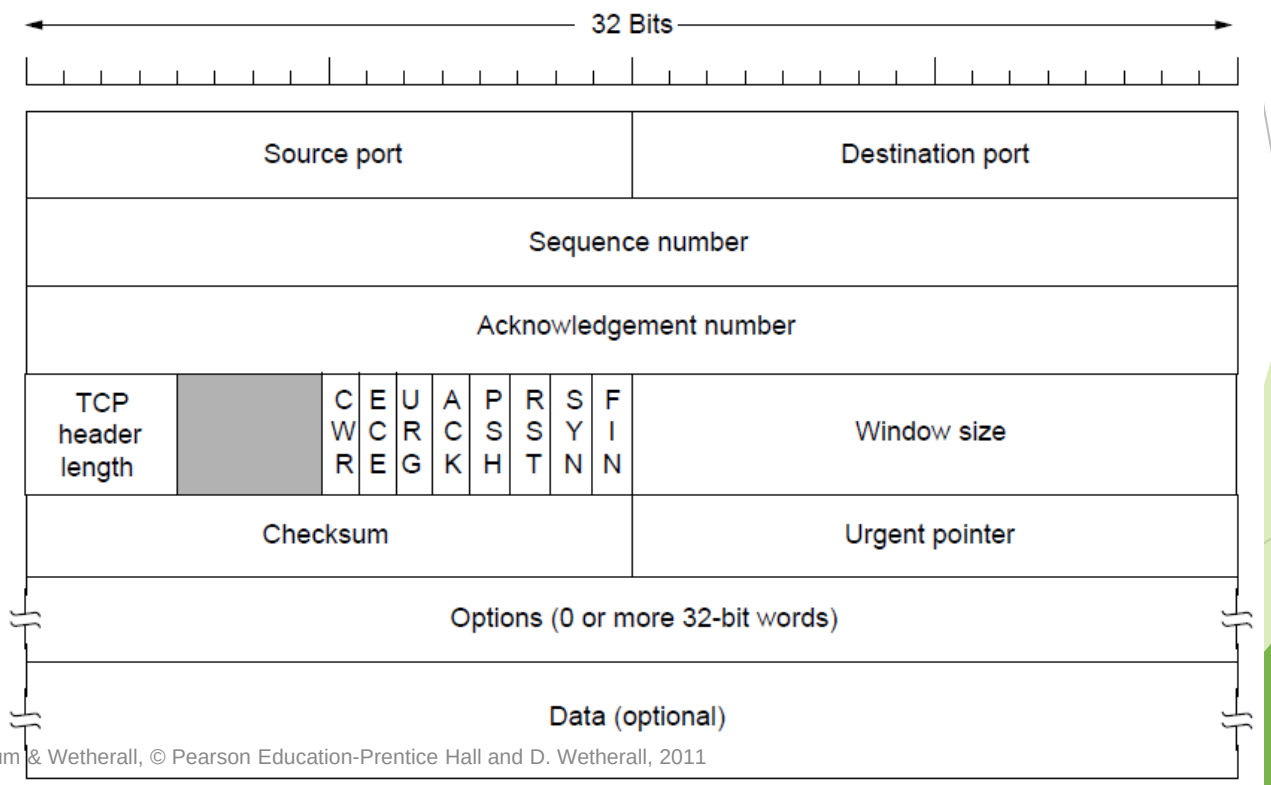


Four segments, each with 512 bytes of data and carried in an IP packet

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

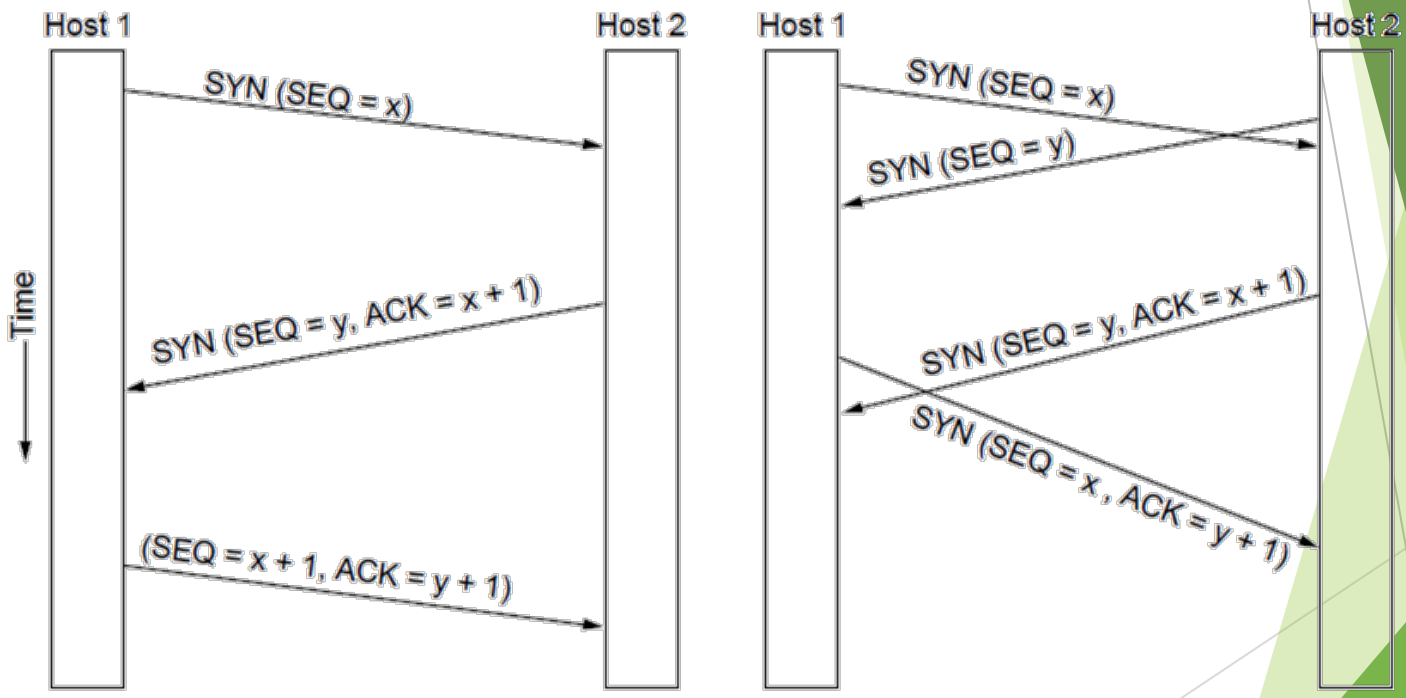
The TCP Segment Header

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



TCP Connection Establishment

- TCP sets up connections with the three-way handshake
  - Release is symmetric, also as described before



Normal case

Simultaneous connect

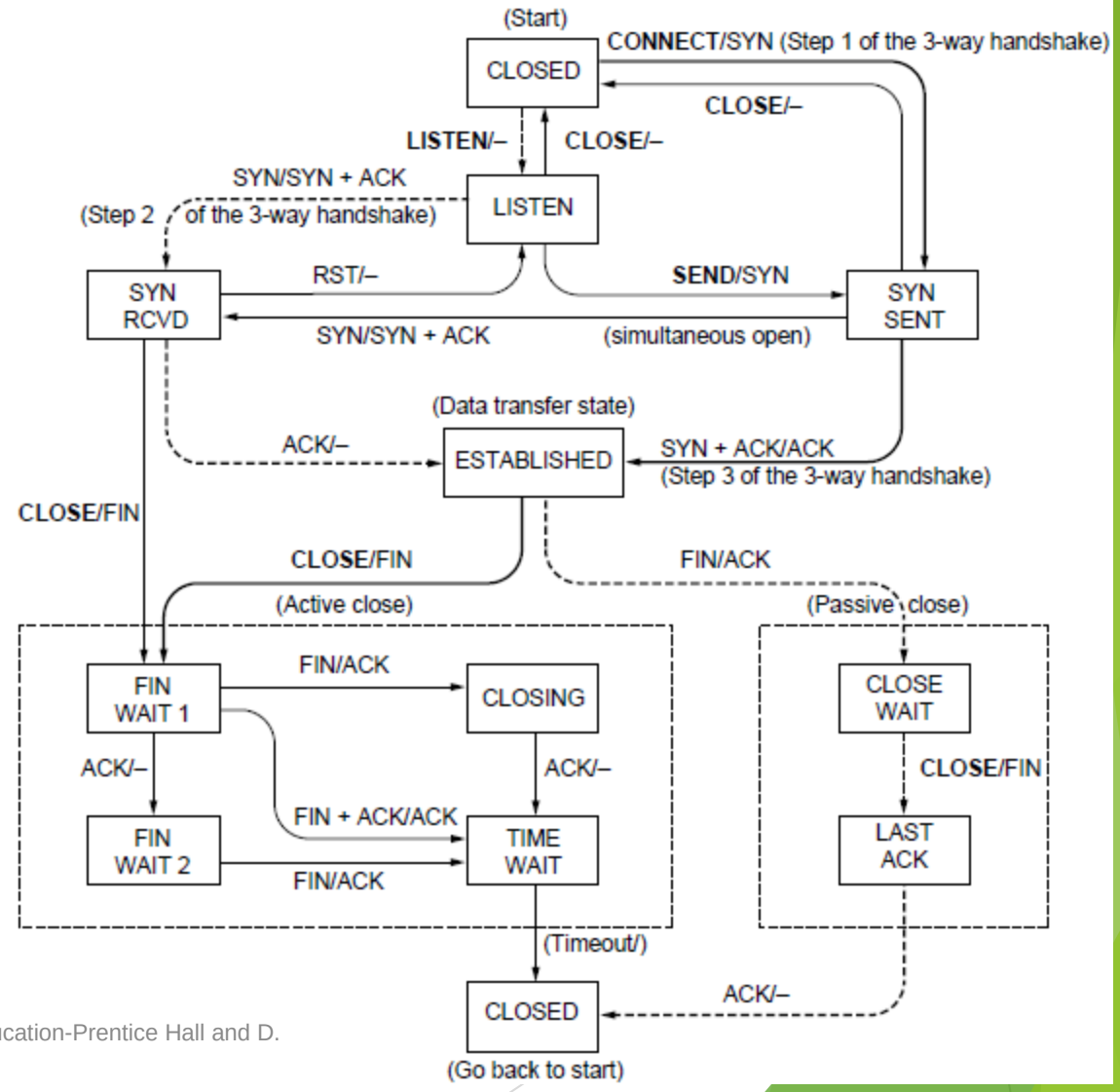
TCP Connection State Modeling (1)

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 (2)

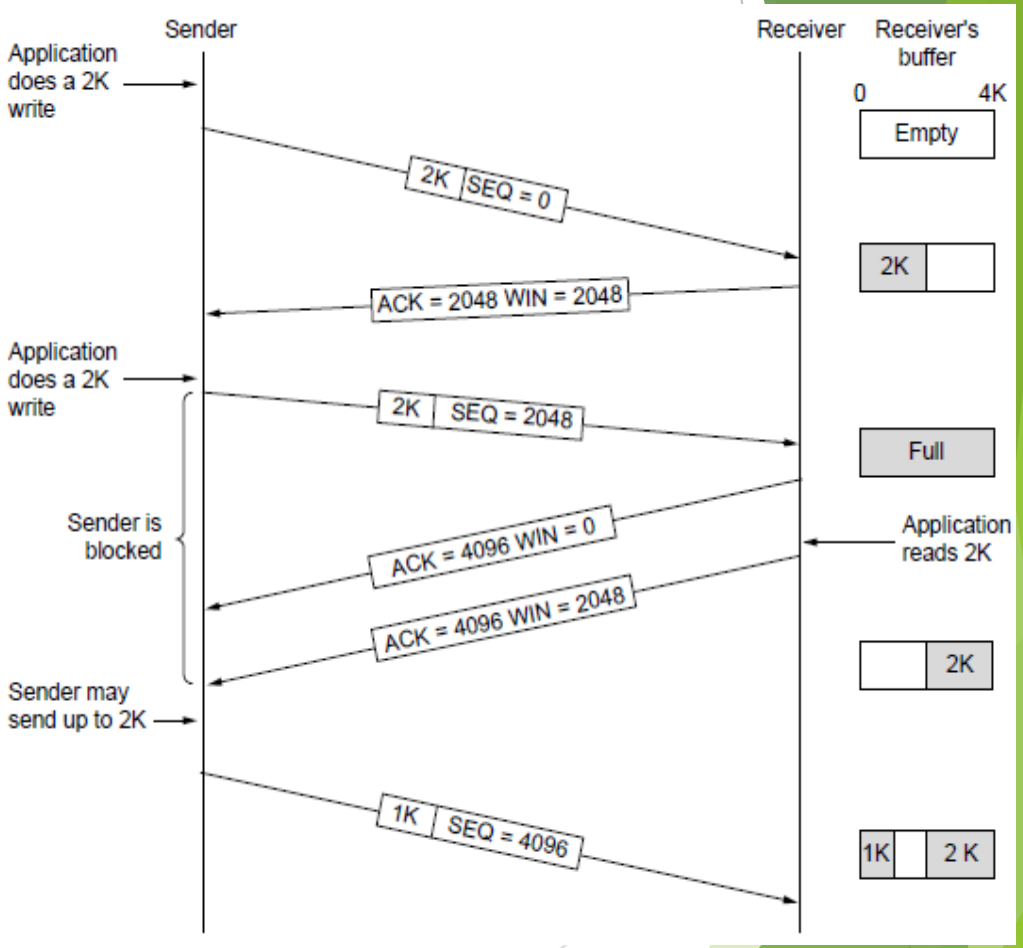
- Solid line is the normal path for a client.
- Dashed line is the normal path for a server.
- Light lines are unusual events.
- Transitions are labeled by the cause and action, separated by a slash.



TCP Sliding Window (1)

TCP adds flow control to the sliding window as before

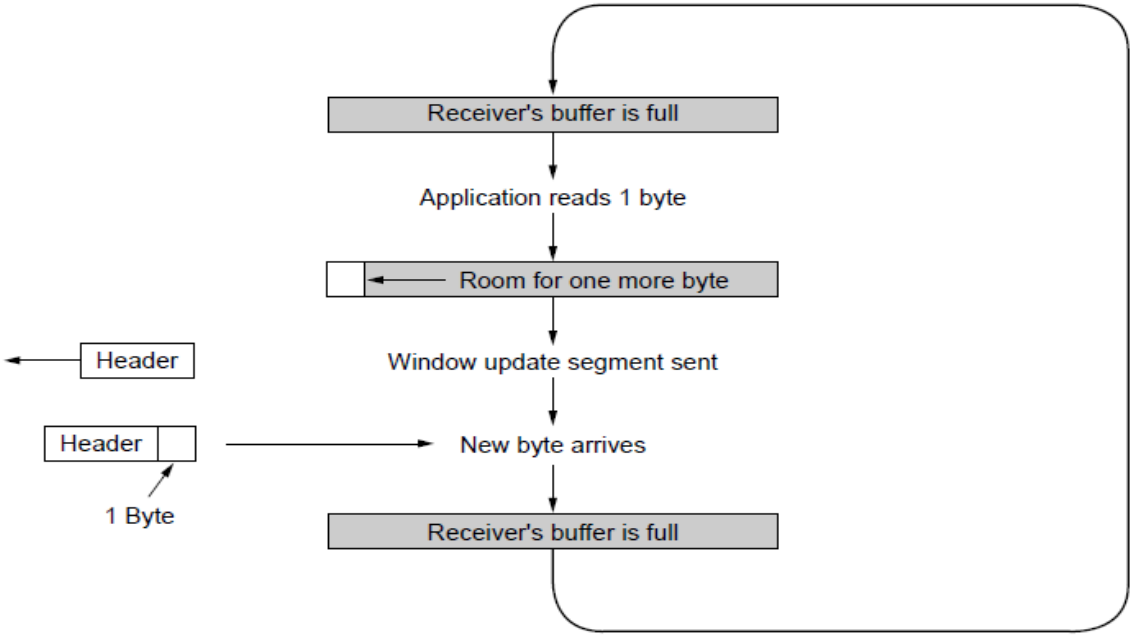
- ACK + WIN is the sender's limit



TCP Sliding Window (2)

Need to add special cases to avoid unwanted behavior

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

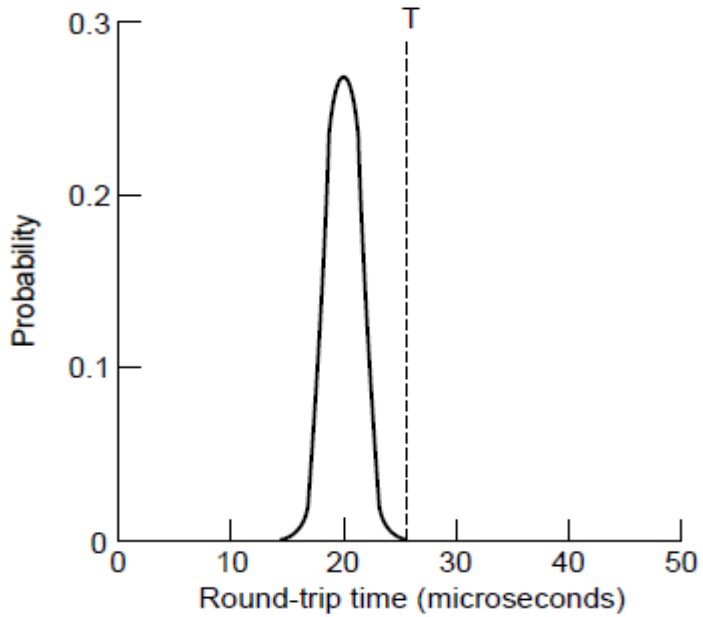


Receiver application reads single bytes, so sender always sends one byte segments

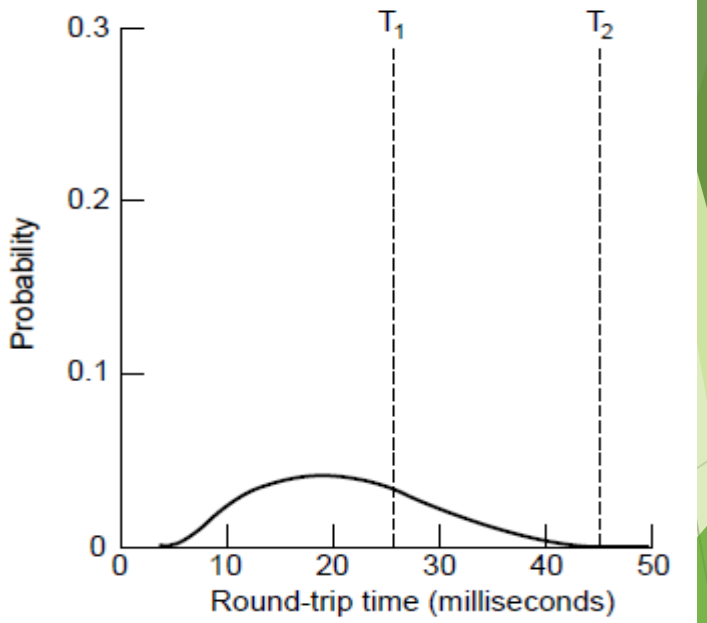
TCP Timer Management

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 Congestion Control (1)

TCP uses AIMD with loss signal to control congestion

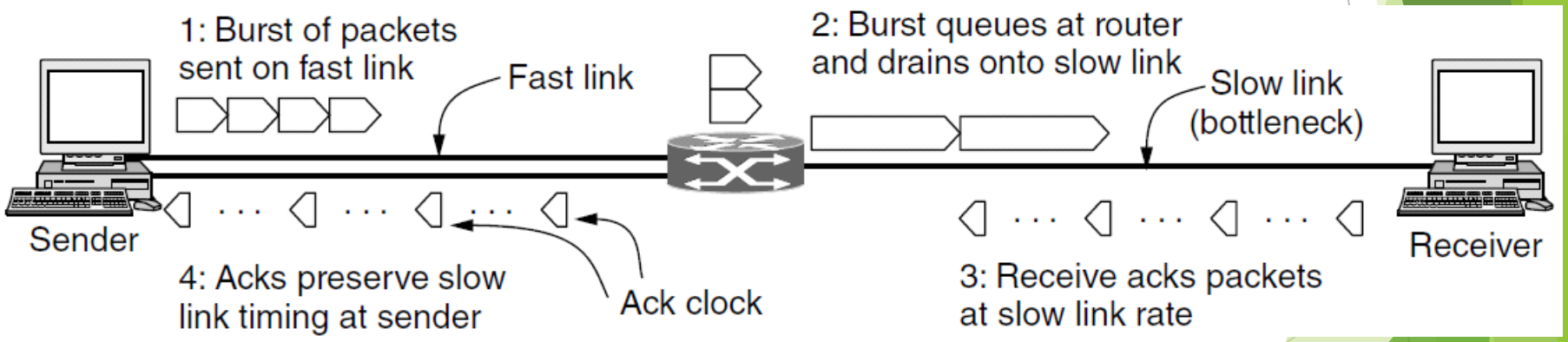
- Implemented as a congestion window (cwnd) for the number of segments that may be in the network
- Uses several mechanisms that work together

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

TCP Congestion Control (2)

Congestion window controls the sending rate

- Rate is cwnd / RTT; window can stop sender quickly
- ACK clock (regular receipt of ACKs) paces traffic and smoothes out sender bursts

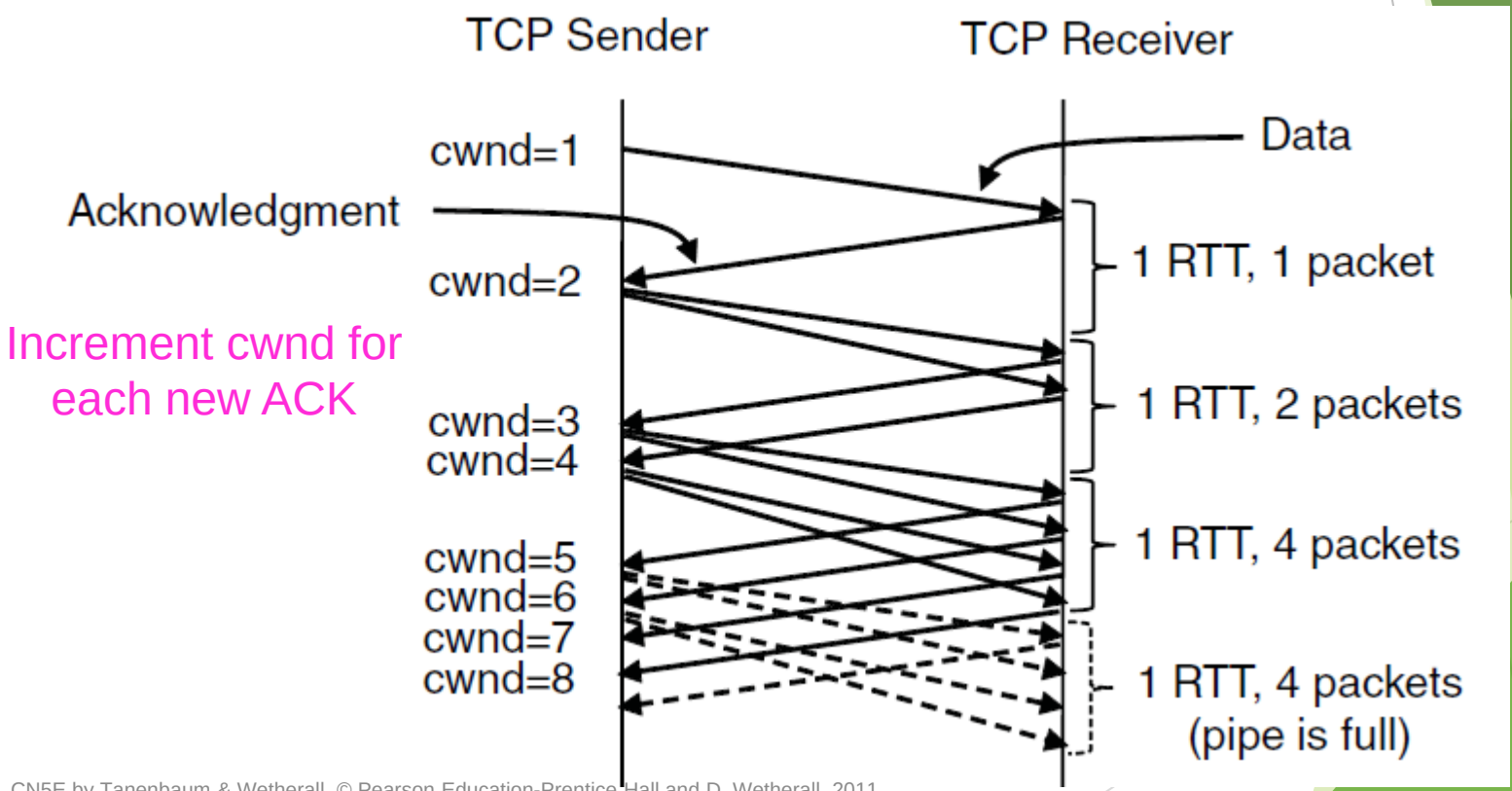


ACKs pace new segments into the network and smooth bursts

TCP Congestion Control (3)

Slow start grows congestion window exponentially

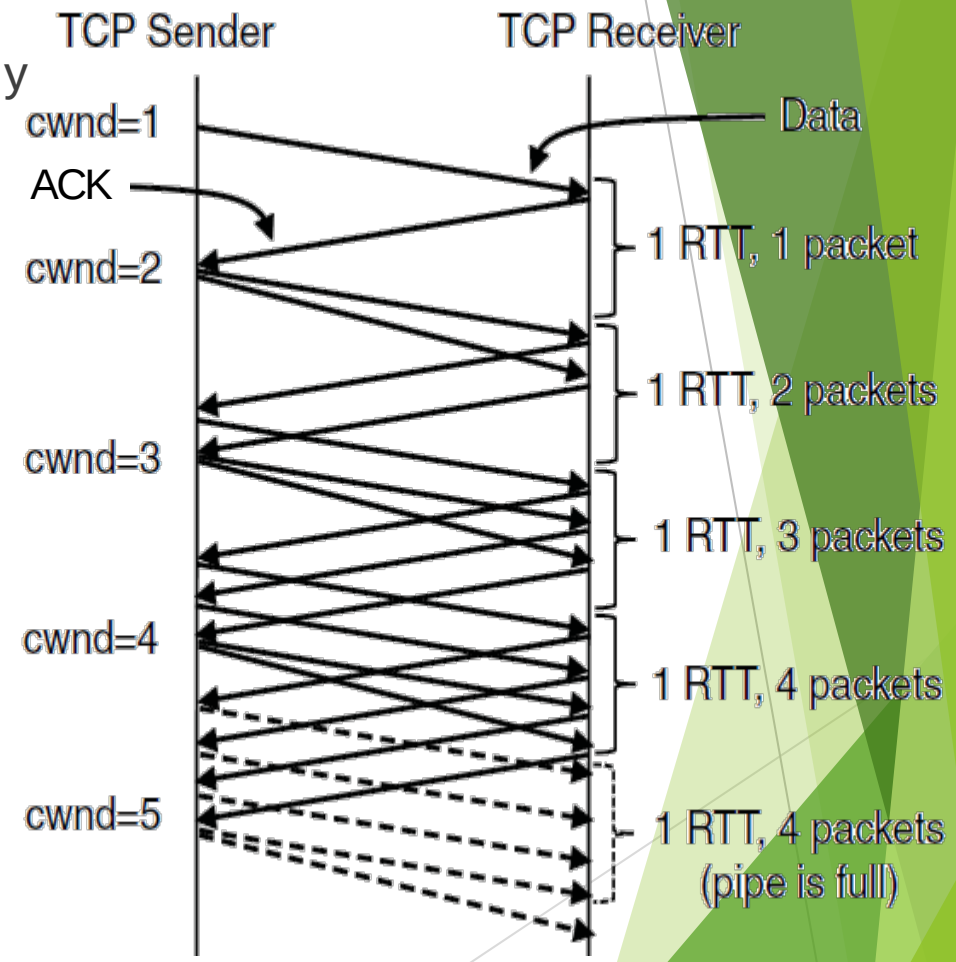
- Doubles every RTT while keeping ACK clock going



TCP Congestion Control (4)

Additive increase grows cwnd slowly

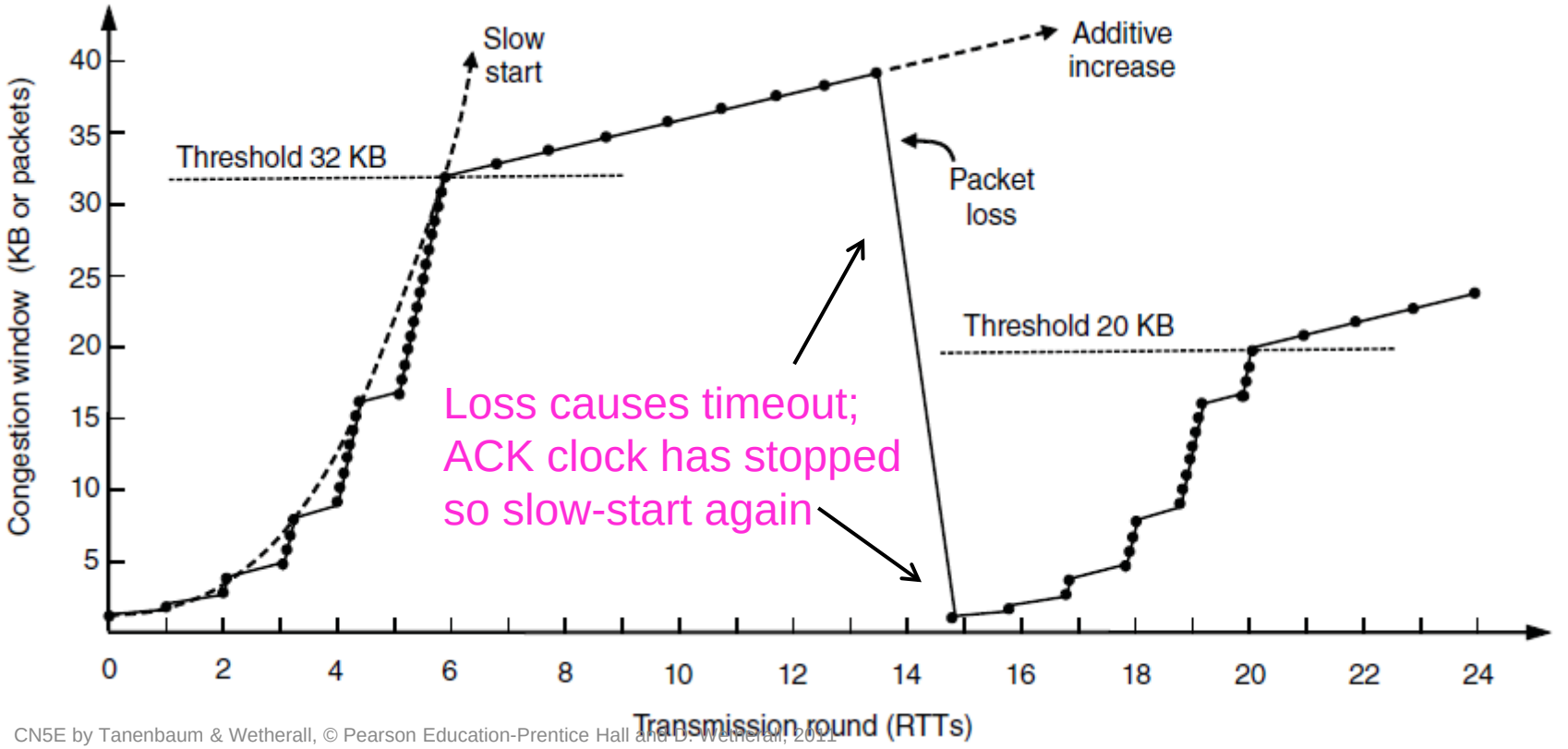
- Adds 1 every RTT
- Keeps ACK clock



TCP Congestion Control (5)

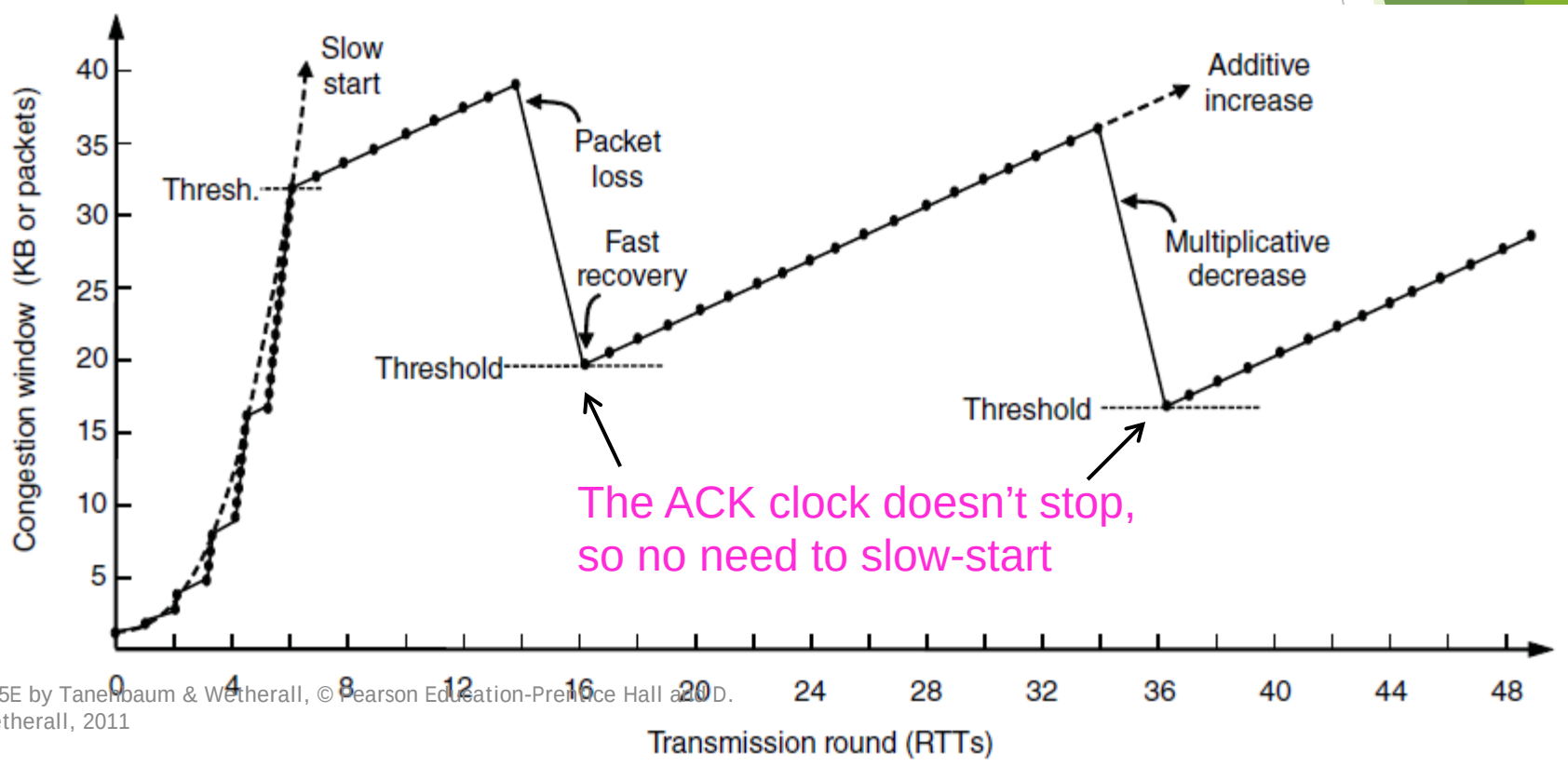
Slow start followed by additive increase (TCP Tahoe)

- Threshold is half of previous loss cwnd



TCP Congestion Control (6)

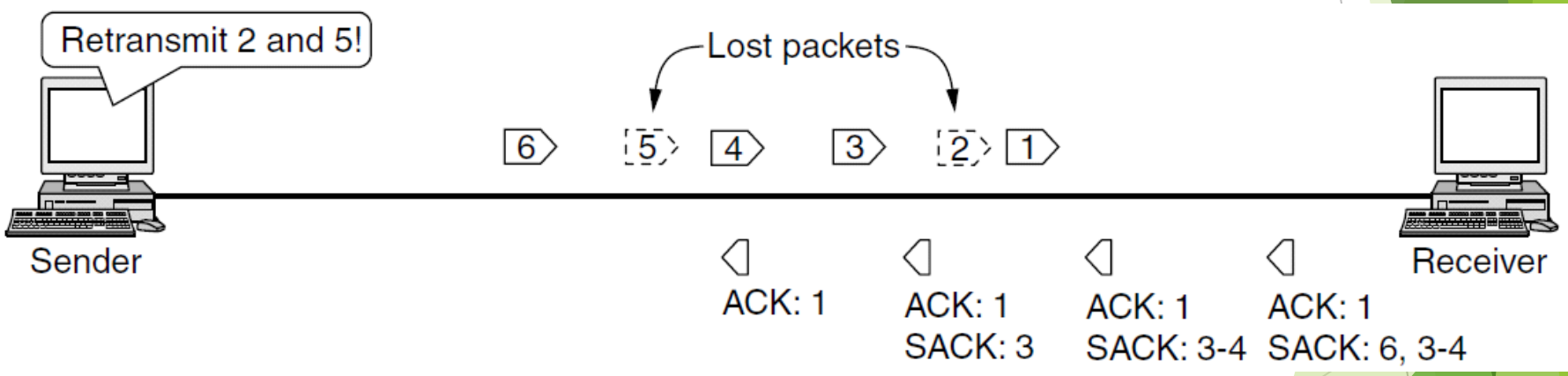
- With fast recovery, we get the classic sawtooth (TCP Reno)
  - Retransmit lost packet after 3 duplicate ACKs
  - New packet for each dup. ACK until loss is repaired



TCP Congestion Control (7)

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

Performance Issues

Many strategies for getting good performance have been learned over time

- Performance problems »
- Measuring network performance »
- Host design for fast networks »
- Fast segment processing »
- Header compression »
- Protocols for “long fat” networks »

Performance Problems

Unexpected loads often interact with protocols to cause performance problems

- Need to find the situations and improve the protocols

Examples:

- Broadcast storm: one broadcast triggers another
- Synchronization: a building of computers all contact the DHCP server together after a power failure
- Tiny packets: some situations can cause TCP to send many small packets instead of few large ones

Measuring Network Performance

Measurement is the key to understanding performance – but has its own pitfalls.

Example pitfalls:

- Caching: fetching Web pages will give surprisingly fast results if they are unexpectedly cached
- Timing: clocks may over/underestimate fast events
- Interference: there may be competing workloads

Host Design for Fast Networks

Poor host software can greatly slow down networks.

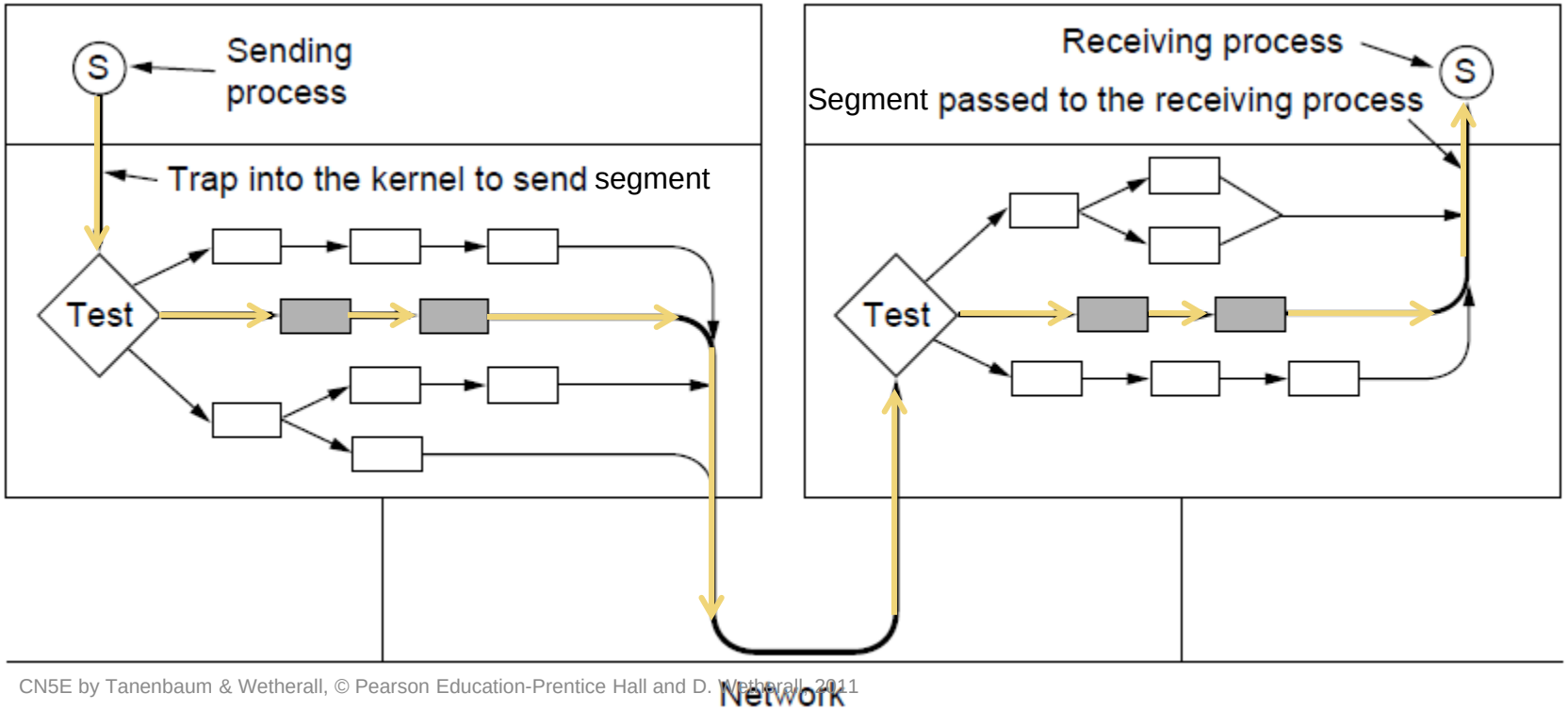
Rules of thumb for fast host software:

- Host speed more important than network speed
- Reduce packet count to reduce overhead
- Minimize data touching
- Minimize context switches
- Avoiding congestion is better than recovering from it
- Avoid timeouts

Fast Segment Processing (1)

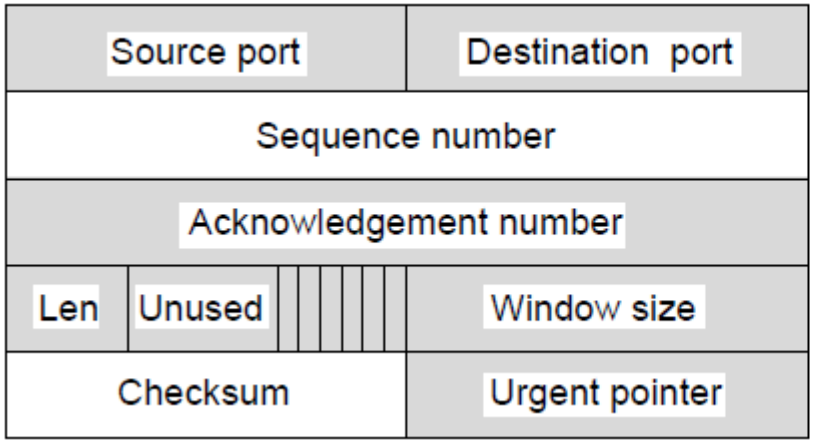
Speed up the common case with a fast path [pink]

- Handles packets with expected header; OK for others to run slowly

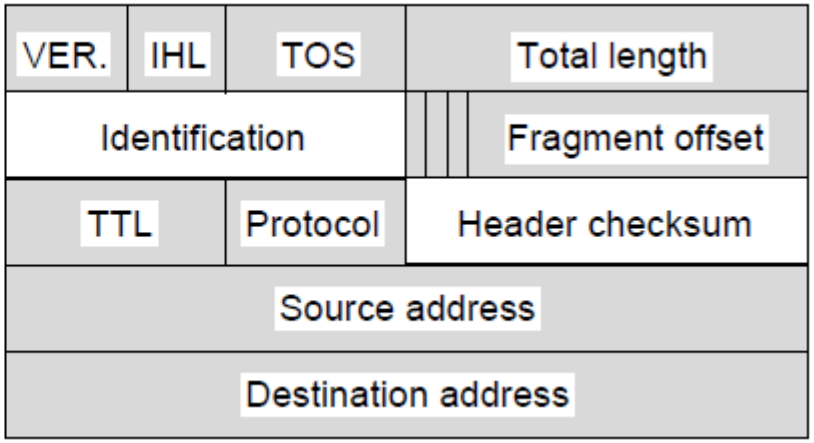


Fast Segment Processing (2)

Header fields are often the same from one packet to the next for a flow; copy/check them to speed up processing



TCP header fields that stay the same for a one-way flow (shaded)



IP header fields that are often the same for a one-way flow (shaded)

Header Compression

Overhead can be very large for small packets

- 40 bytes of header for RTP/UDP/IP VoIP packet
- Problematic for slow links, especially wireless

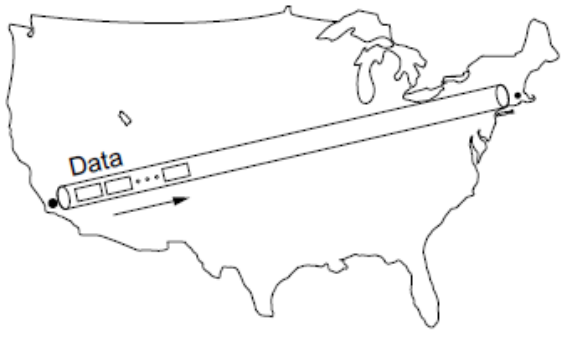
Header compression mitigates this problem

- Runs between Link and Network layer
- Omits fields that don't change or change predictably
  - 40 byte TCP/IP header → 3 bytes of information
- Gives simple high-layer headers and efficient links

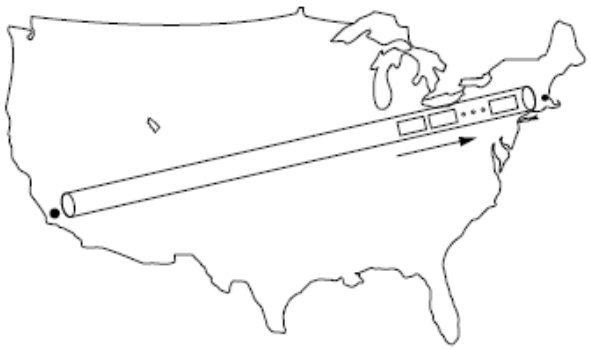
Protocols for "Long Fat" Networks (1)

Networks with high bandwidth ("Fat") and high delay ("Long") can store much information inside the network

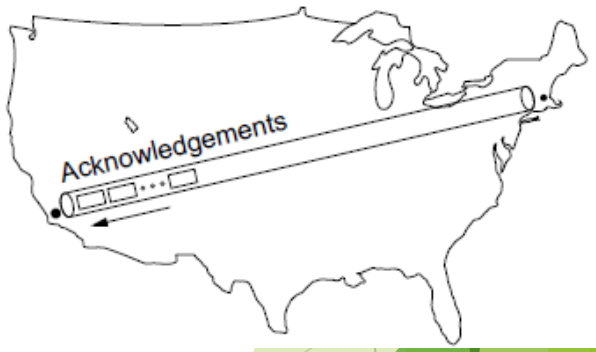
- Requires protocols with ample buffering and few RTTs, rather than reducing the bits on the wire



Starting to send 1 Mbit
San Diego → Boston



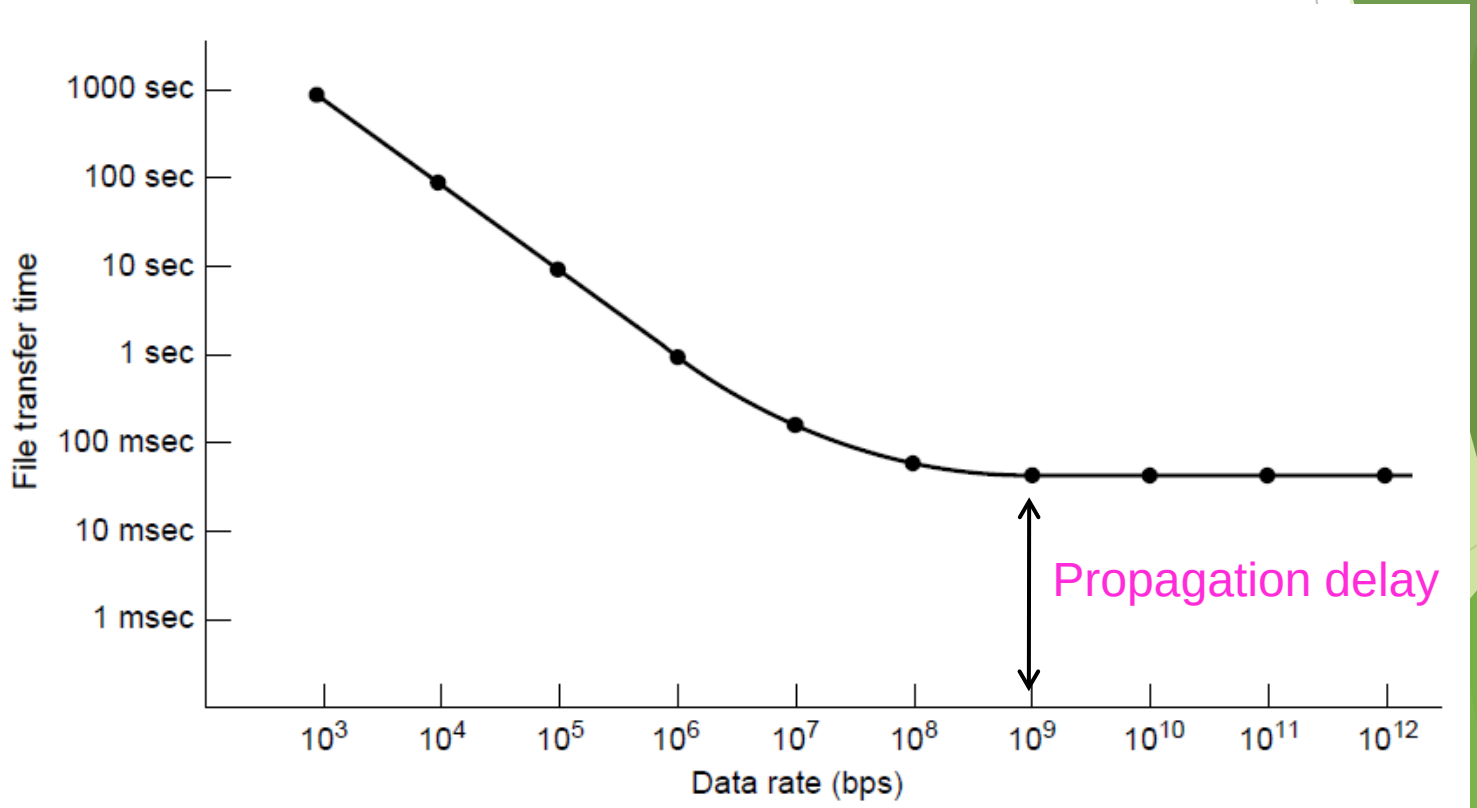
20ms after start



40ms after start

Protocols for "Long Fat" Networks (2)

- You can buy more bandwidth but not lower delay
  - Need to shift ends (e.g., into cloud) to lower further



Minimum time to send and ACK a 1-Mbit file over a 4000-km line

Delay Tolerant Networking

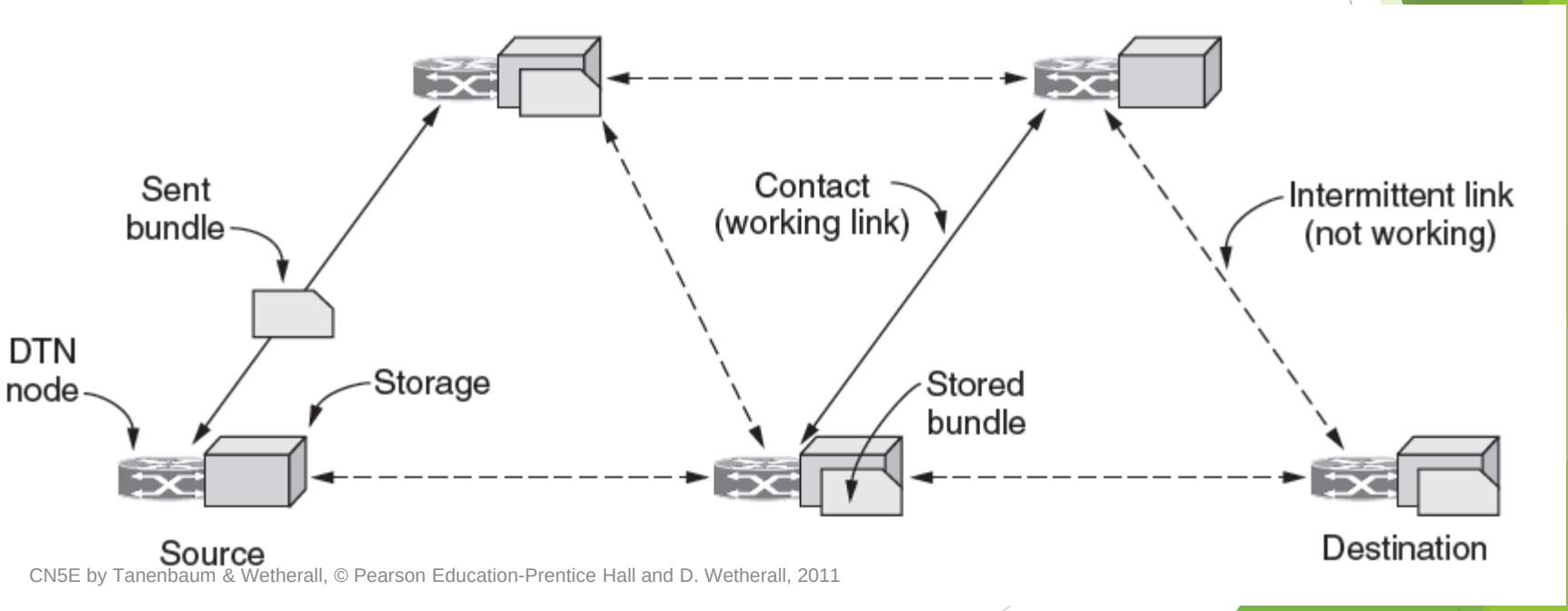
DTNs (Delay Tolerant Networks) store messages inside the network until they can be delivered

- DTN Architecture »
- Bundle Protocol »

DTN Architecture (1)

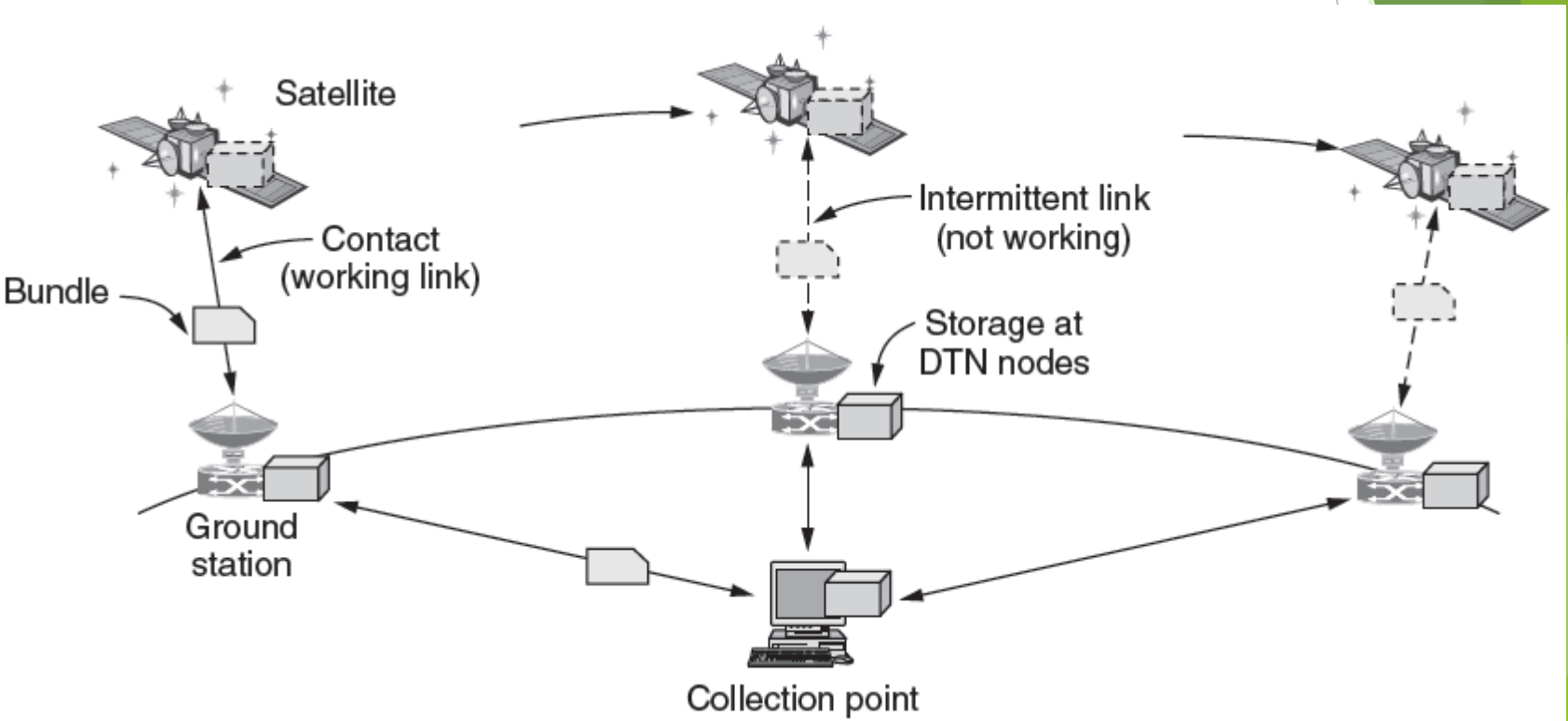
Messages called bundles are stored at DTN nodes while waiting for an intermittent link to become a contact

- Bundles might wait hours, not milliseconds in routers
- May be no working end-to-end path at any time



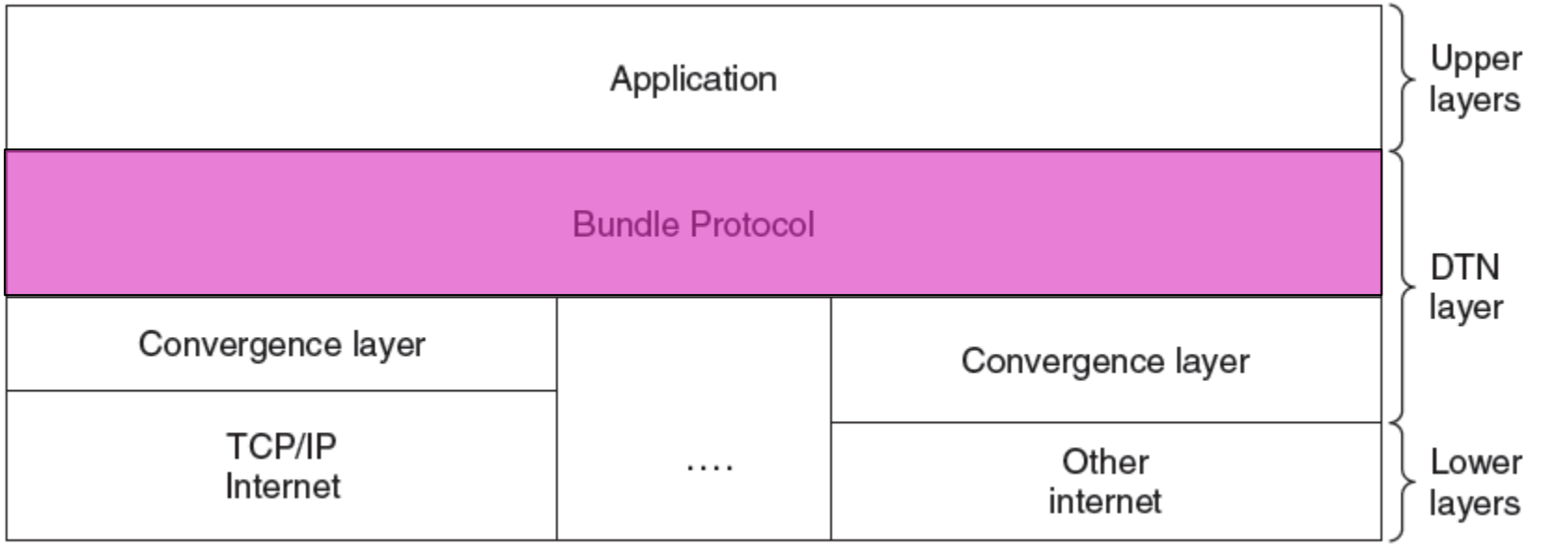
DTN Architecture (2)

Example DTN connecting a satellite to a collection point



Bundle Protocol (1)

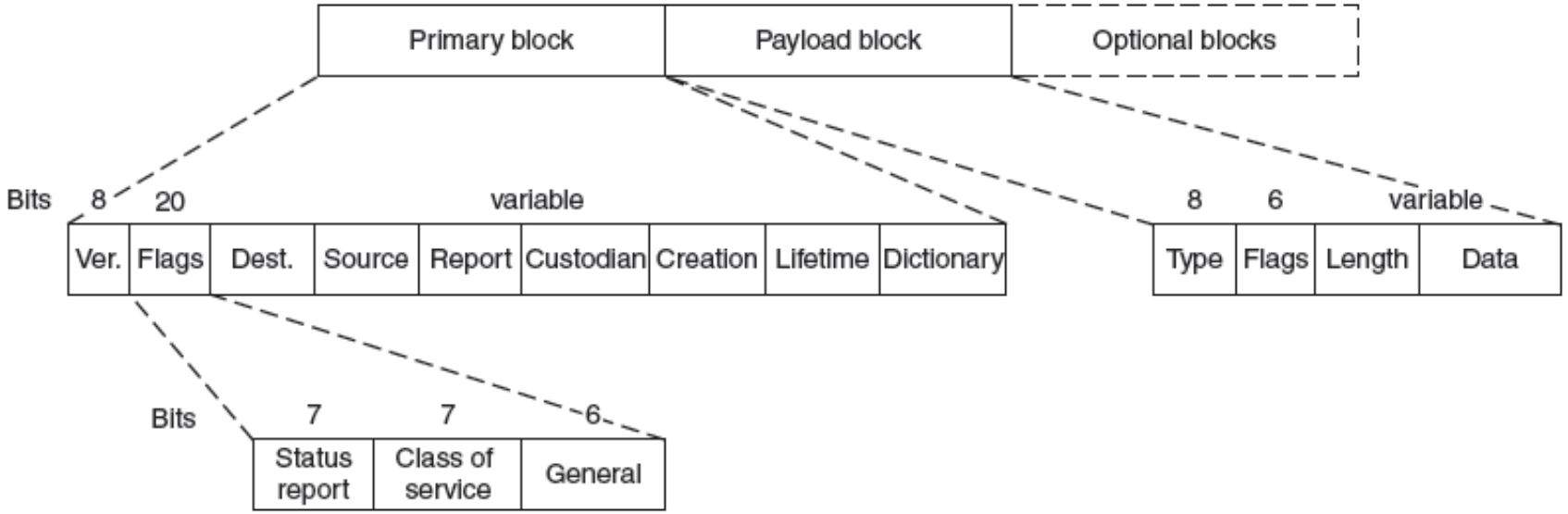
The Bundle protocol uses TCP or other transports and provides a DTN service to applications



Bundle Protocol (2)

Features of the bundle message format:

- Dest./source add high-level addresses (not port/IP)
- Custody transfer shifts delivery responsibility
- Dictionary provides compression for efficiency



End

Chapter 6