Transport Layer

establish virtual connections between applications

main protocols: TCP and UDP

Transport Layer

our goals:

- understand principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- learn about Internet transport layer protocols:
  - UDP: connectionless transport
  - TCP: connection-oriented reliable transport
  - TCP congestion control

Outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

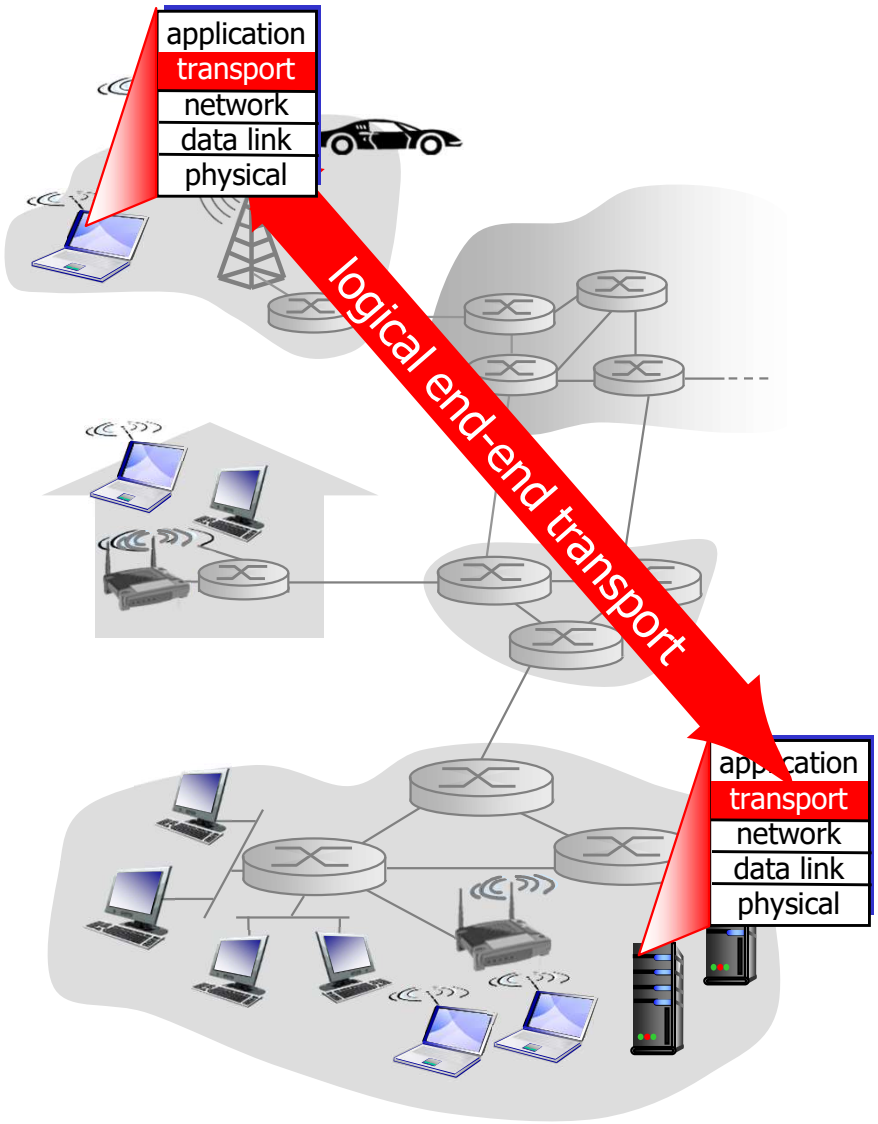
3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP

- segment structure
- reliable data transfer
- connection management

Transport services and protocols

- provide *logical communication* between app processes running on different hosts
- transport protocols run in end systems
  - send side: breaks app messages into *segments*, passes to network layer
  - receiving side side: reassembles segments into messages, passes to app layer
- more than one transport protocol available to apps
  - Internet: TCP and UDP



Transport vs. network layer

- *network layer:* logical communication between hosts
- *transport layer:* logical communication between processes
  - relies on, enhances, network layer services

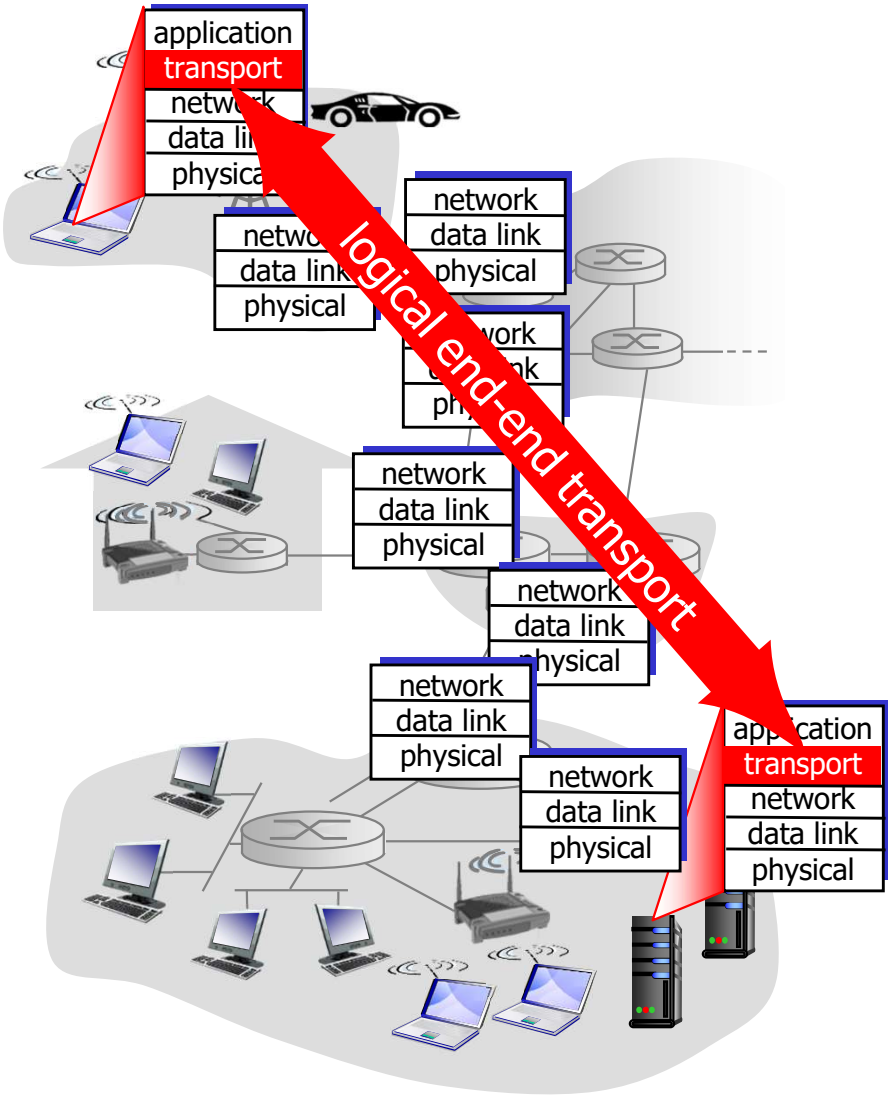
household analogy:

12 kids in Ann's house sending letters to 12 kids in Bill's house:

- hosts = houses
- processes = kids
- app messages = letters in envelopes
- transport protocol = Ann and Bill who demux to in-house siblings
- network-layer protocol = postal service

Internet transport-layer protocols

- reliable, in-order delivery (TCP)
  - congestion control
  - flow control
  - connection setup
- unreliable, unordered delivery: UDP
  - no-frills extension of "best-effort" IP
- services not available:
  - delay guarantees
  - bandwidth guarantees



Outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP

- segment structure
- reliable data transfer
- connection management

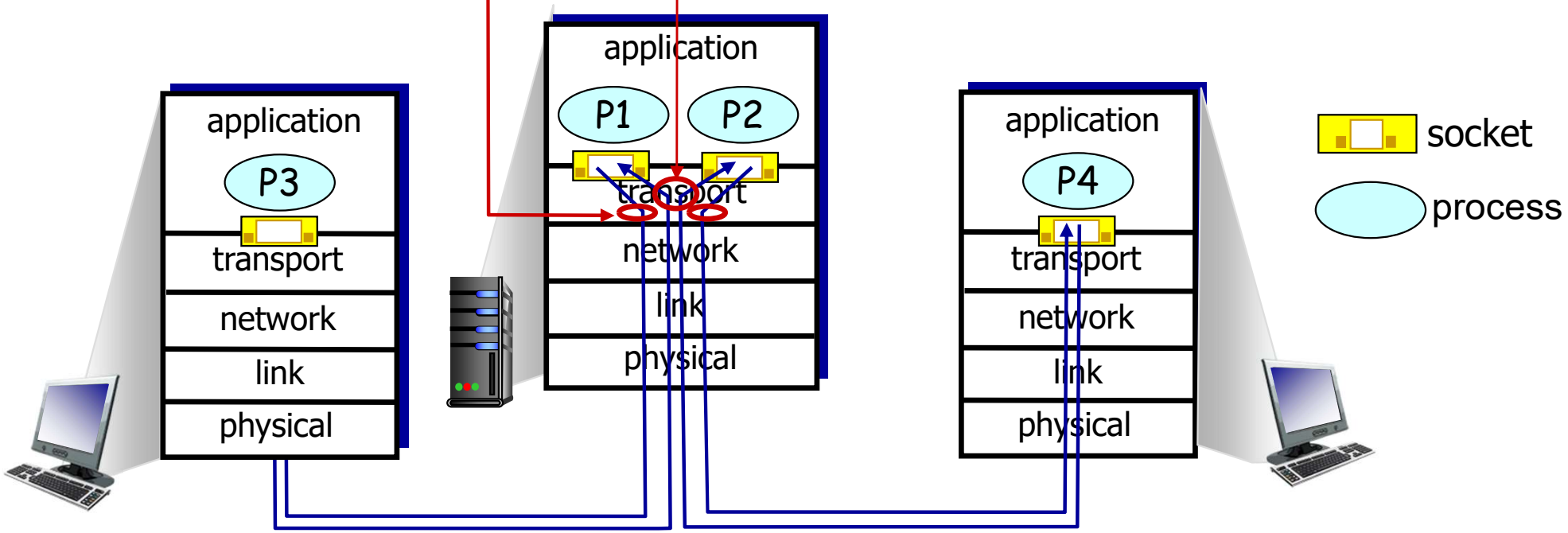
Multiplexing/demultiplexing

multiplexing at sender:

handle data from multiple sockets, add transport header (later used for demultiplexing)

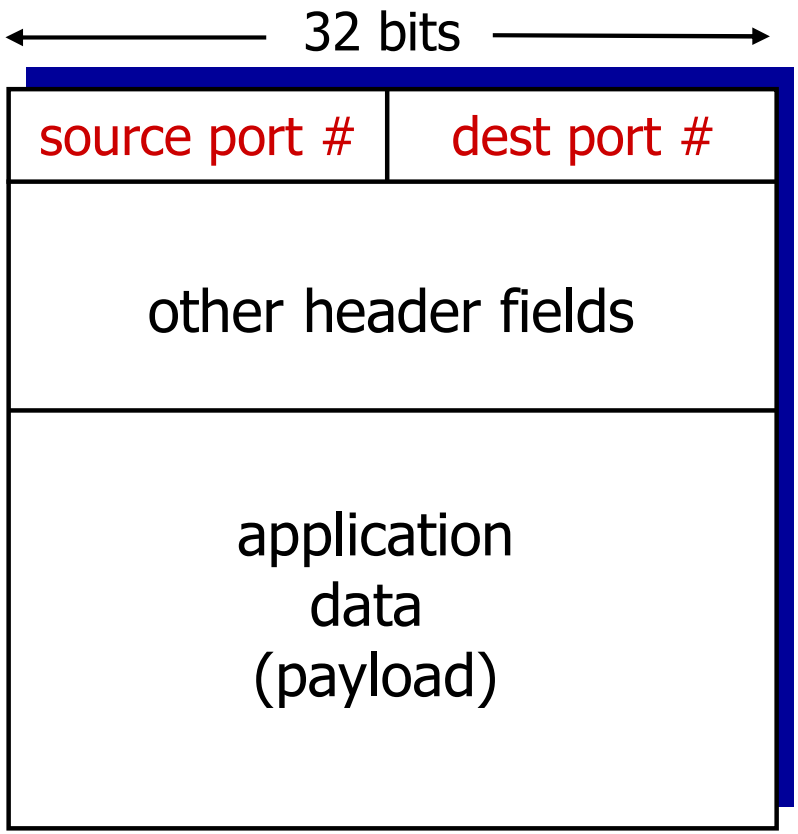
demultiplexing at receiver:

use header info to deliver received segments to correct socket



How demultiplexing works

- host receives IP datagrams
  - each datagram has source IP address, destination IP address
  - each datagram carries one transport-layer segment
  - each segment has source, destination port number
- host uses *IP addresses & port numbers* to direct segment to appropriate socket



TCP/UDP segment format

Connectionless demultiplexing

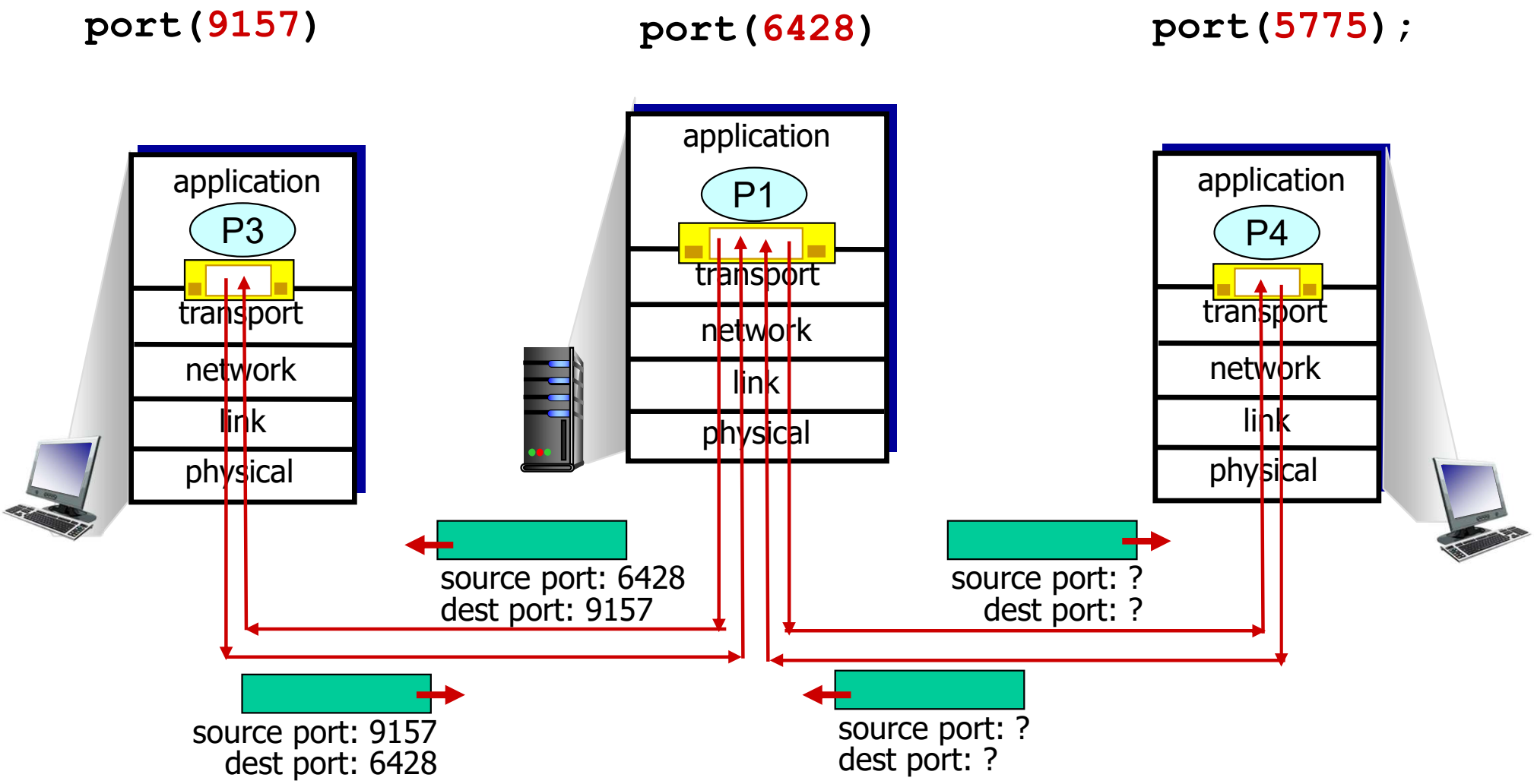
- *recall:* created socket has host-local port #:

```
clientSocket = socket(AF_INET, SOCK_DGRAM)
clientSocket.bind(('', 19157)) # to assign specific port
```

- when host receives UDP segment:
  - checks destination port # in segment
  - directs UDP segment to socket with that port #

IP datagrams with *same dest. port #,* but different source IP addresses and/or source port numbers will be directed to *same socket* at dest

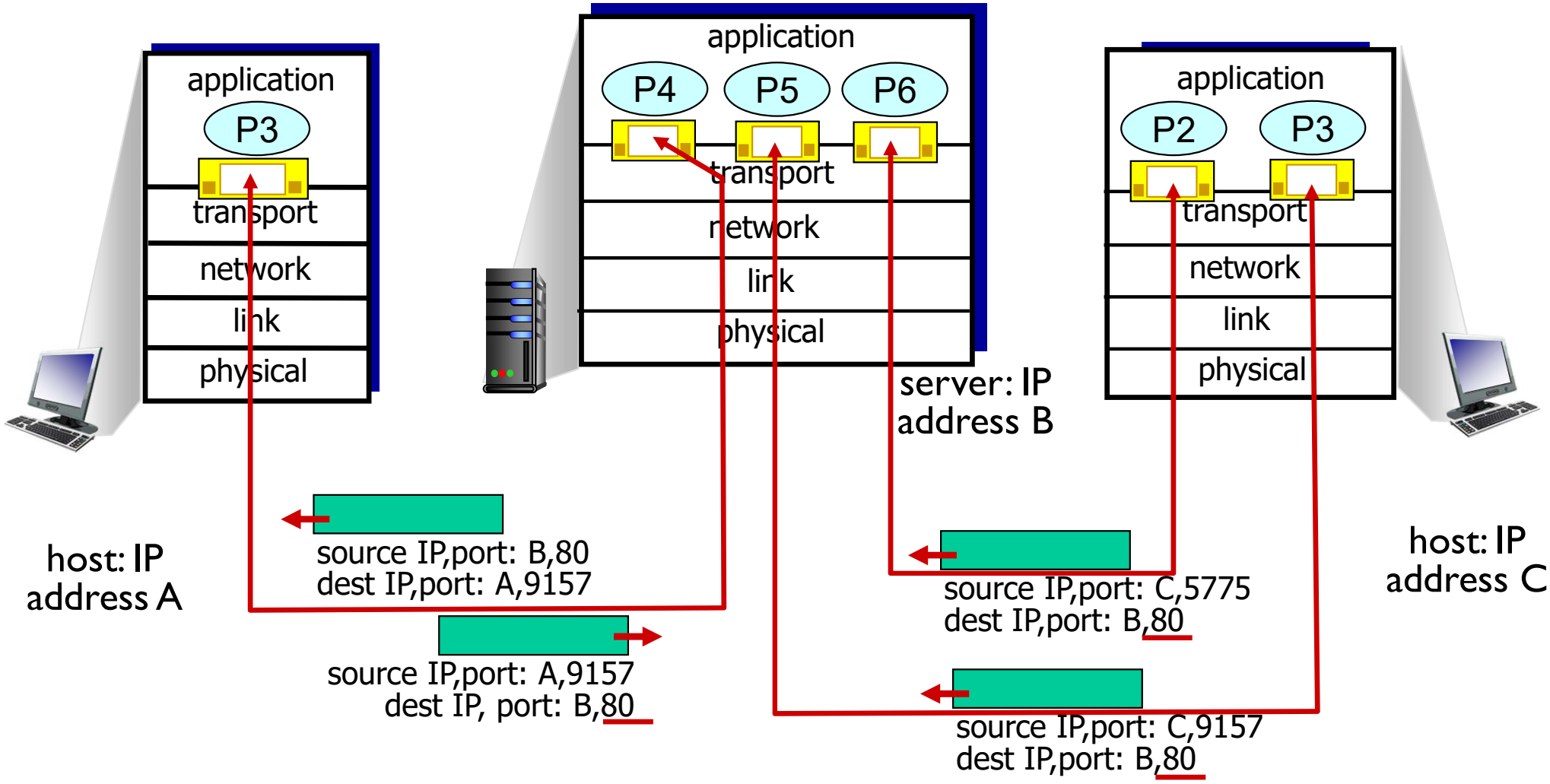
Connectionless demux: example



Connection-oriented demux

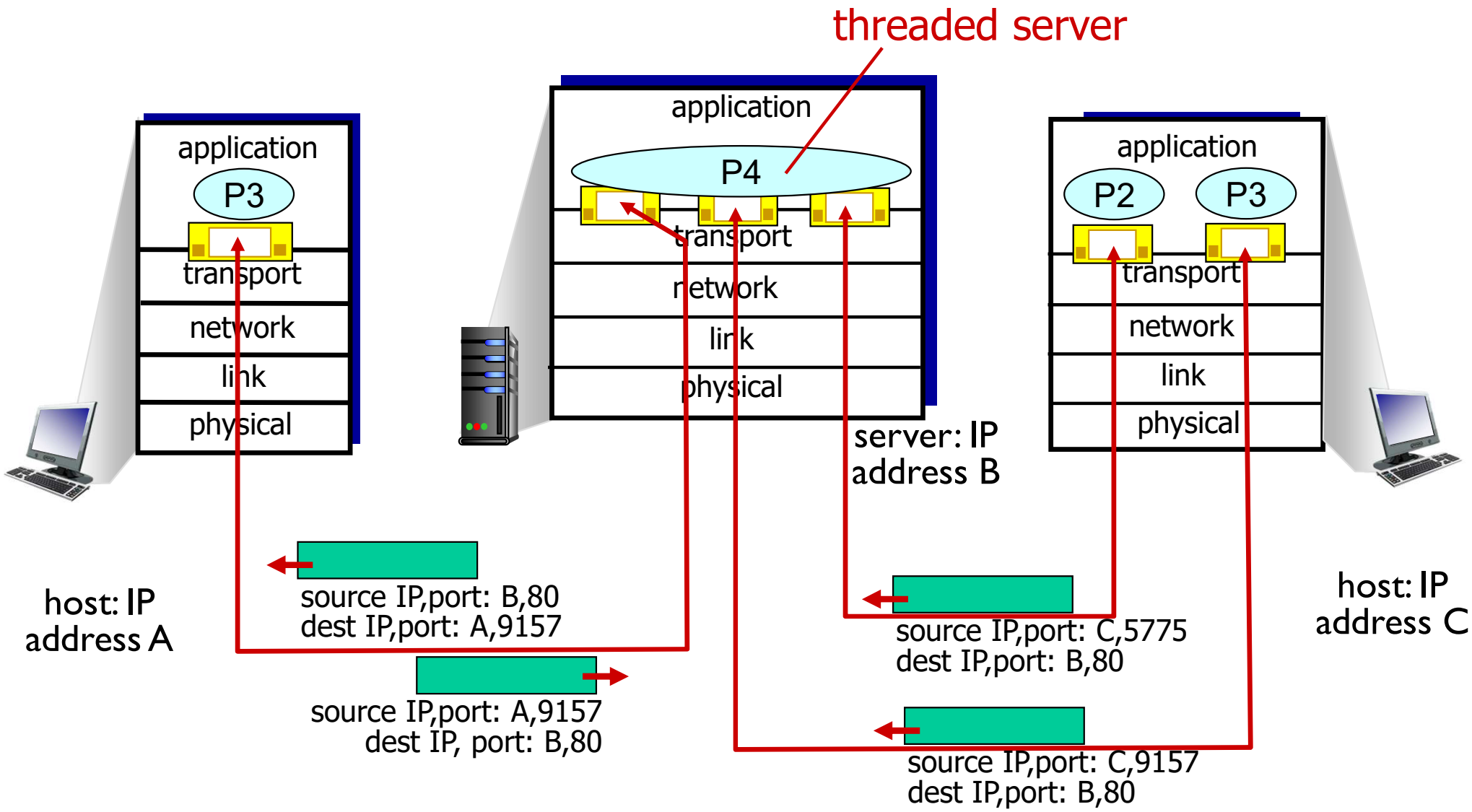
- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- demux: receiver uses all four values to direct segment to appropriate socket
- server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- web servers have different sockets for each connecting client
  - non-persistent HTTP will have different socket for each request

Connection-oriented demux: example



three segments, all destined to IP address: B, dest port: 80 are demultiplexed to *different* sockets

Connection-oriented demux: example



Outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP

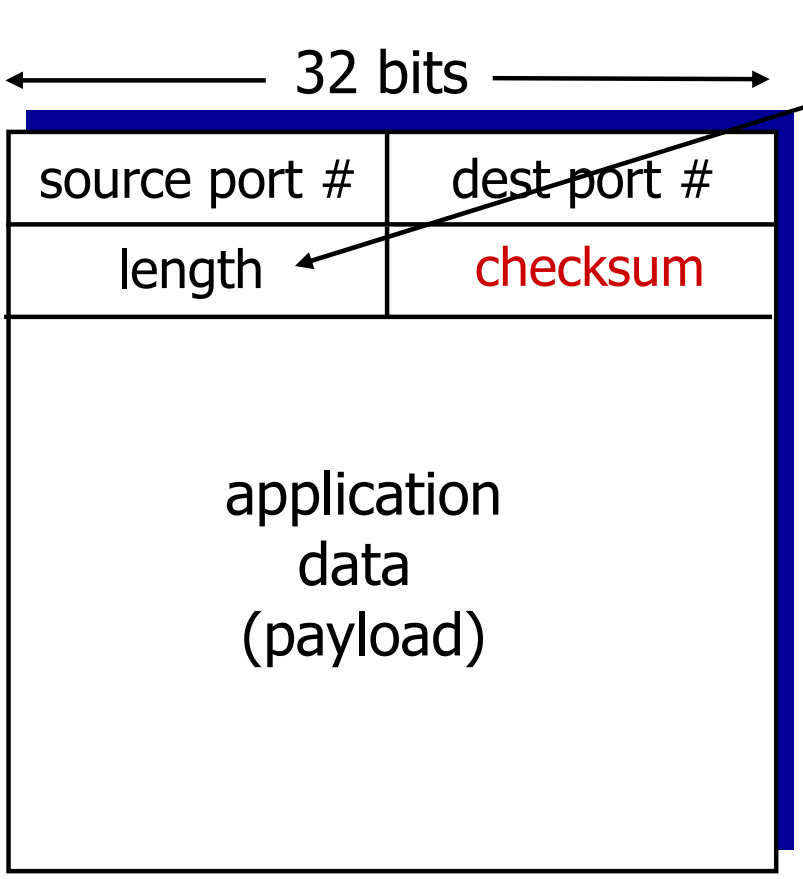
- segment structure
- reliable data transfer
- connection management

UDP: User Datagram Protocol [RFC 768]

- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out-of-order to app
- *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

- UDP use:
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
  - SNMP
- reliable transfer over UDP:
  - add reliability at application layer
  - application-specific error recovery!

UDP: segment header



UDP segment format

length, in bytes of UDP segment, including header

why is there a UDP?

- no connection establishment (which can add delay)
- simple: no connection state at sender, receiver
- small header size
- no congestion control: UDP can blast away as fast as desired

Applications and transport protocols

Application	Application-Layer Protocol	Underlying Transport Protocol
Electronic mail	SMTP	TCP
Remote terminal access	Telnet	TCP
Web	HTTP	TCP
File transfer	FTP	TCP
Remote file server	NFS	Typically UDP
Streaming multimedia	typically proprietary	UDP or TCP
Internet telephony	typically proprietary	UDP or TCP
Network management	SNMP	Typically UDP
Name translation	DNS	Typically UDP

must add 2 bytes by 2 bytes or word by word

UDP checksum

Goal: detect "errors" (e.g., flipped bits) in transmitted segment

sender:

- treat segment contents, including header fields, as sequence of 16-bit integers
- checksum: addition (one's complement sum) of segment contents
- sender puts checksum value into UDP checksum field

receiver:

- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected. *But maybe errors nonetheless?* More later ….

Internet checksum: example

example: add two 16-bit integers: E666 D555

2 bytes

	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
wraparound	(1)	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1
sum		1	0	1	1	1	0	1	1	1	0	1	1	1	1	0
checksum		0	1	0	0	0	1	0	0	0	1	0	0	0	0	1

one's complemnet addition/submission

e.g. add 2 bytes at a time and wraparound carryover

then take complement of submission

Notes:

- when adding numbers, a carryout from the most significant bit needs to be added to the result
- At the receiver, all three 16-bit words are added, including the checksum. If no errors, then the sum at the receiver will be 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

common mistake is addition byte by byte instead of word by word

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

Outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP

- segment structure
- reliable data transfer
- connection management

Principles of reliable data transfer

- One of the basic principles for reliable data transfer is flow control
- Flow control is a technique for assuring that a transmitting entity does not overwhelm a receiving entity with data
- The receiving entity typically allocates a data buffer of some maximum length for a transfer.
- When data are received, the receiver must do a certain amount of processing before passing the data to the higher-level software
- In the absence of flow control, the receiver's buffer may fill up and overflow while it is processing old data

Stop-and-Wait Flow Control

- The simplest form of flow control and it works as follows:
  - A source entity transmits a frame
  - After the destination entity receives the frame, it indicates its willingness to accept another frame by sending back an acknowledgment to the frame just received
  - The source must wait until it receives the acknowledgement before sending the next frame
  - The destination can thus stop the flow of data simply by withholding acknowledgment

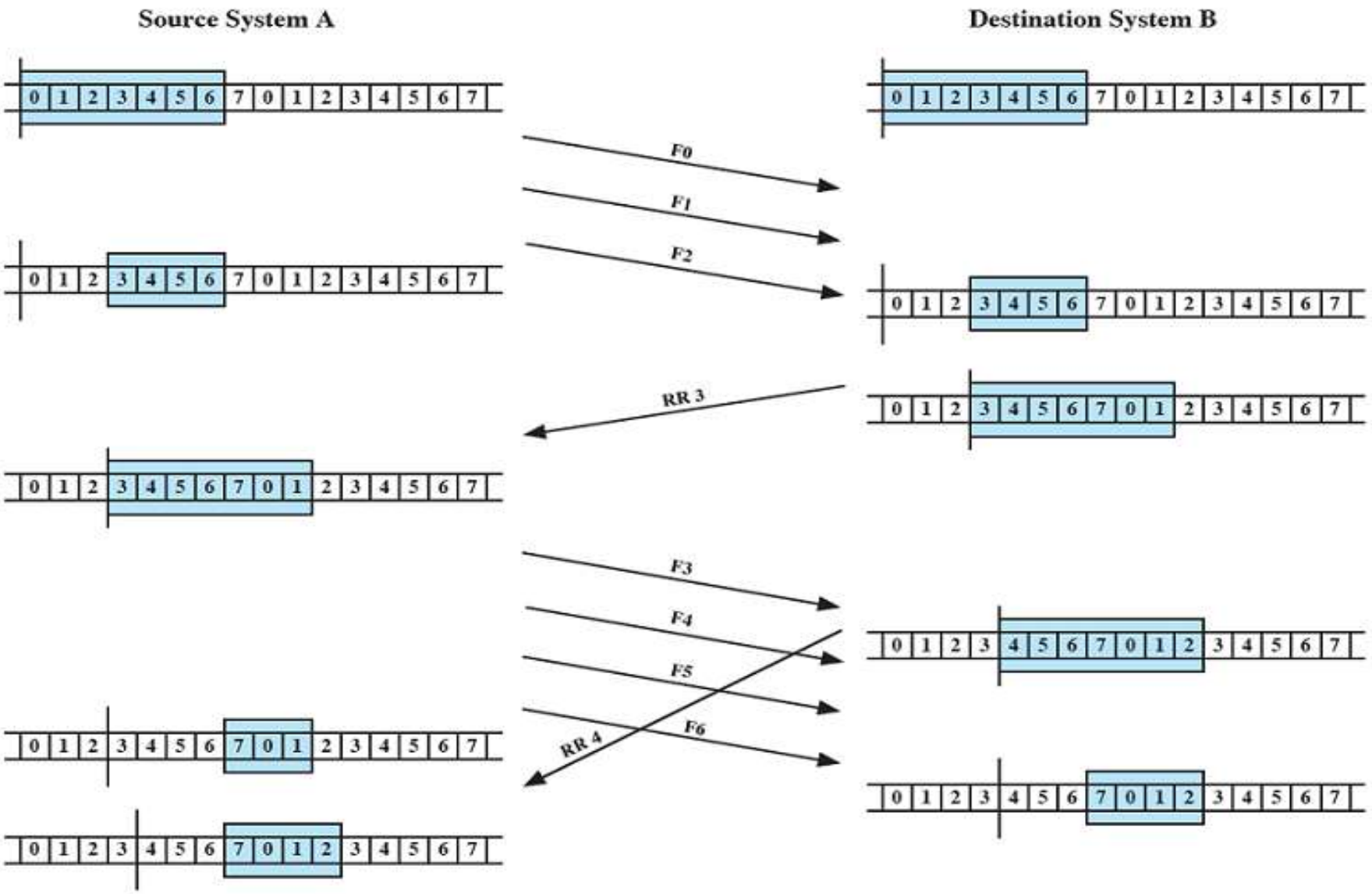
Stop-and-Wait Flow Control

- The procedure of stop-and-wait flow control works fine when a message is sent in a few large frames
- However, it is often the case that a source will break up a large block of data into smaller blocks and transmit the data in many frames. This is done for the following reasons:
  - The buffer size of the receiver may be limited
  - The longer the transmission, the more likely that there will be an error, necessitating retransmission of the entire frame
  - On a shared medium, it is usually desirable NOT to permit one station to occupy the medium for an extended period.
- With the use of multiple frames for a single message, the stop-and-wait procedure may be inadequate. The essence of the problem is that only one frame at a time can be in transit.

Sliding-Window Flow Control

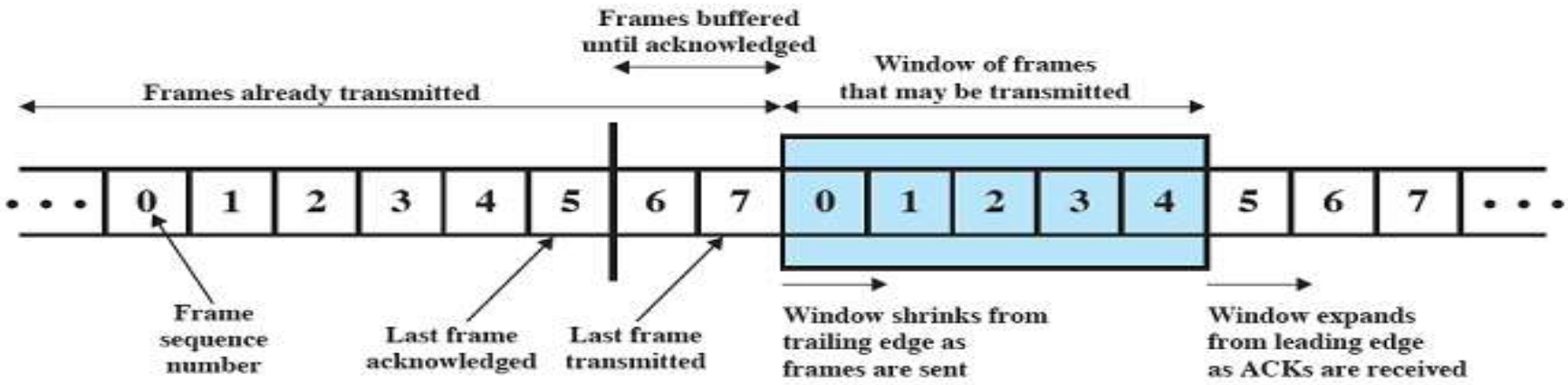
- It allows multiple frames to be in transit at the same time
- Receiver has a buffer space for ***W*** frames
- Transmitter can send **up to** ***W*** frames without acknowledgment (ACK)
- Each frame is numbered
- ACK includes number of next frame expected
- ACK implicitly announces that the receiver is prepared to receive the **next W frames**, beginning with the number specified with the ACK

Sliding-Window Flow Control

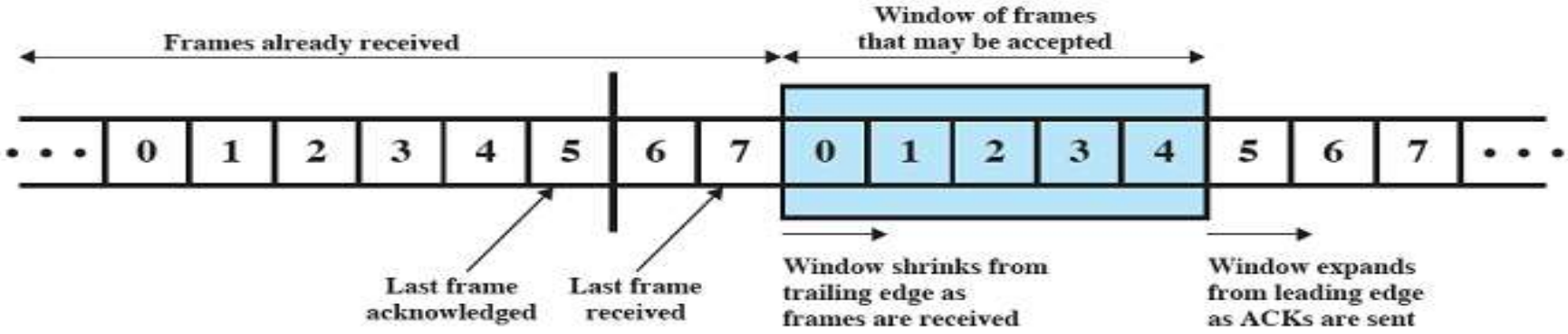


Example of a Sliding-Window Protocol

Sliding-Window Flow Control



(a) Sender's perspective



(b) Receiver's perspective

Sliding-Window Depiction

Sliding-Window Flow Control

- For example, a receiver could receive frames 2, 3, and 4 but withhold acknowledgement until frame 4 has arrived. By returning ACK with sequence number 5, the receiver is acknowledging frames 2,3, and 4. Also, it indicates that the receiver is ready to accept **W** frames beginning with frame number 5
- The sender maintains a list of sequence numbers that is allowed to send, and the receiver maintains a list of sequence numbers that it is prepared to receive. Each of these lists can be thought of as a *window* of frames
- Sequence number is bounded by size of field. For a ***k***-bit, sequence number is 0 through $2^k - 1$. Frames are numbered modulo 2^k . Example, for 3 bits, the sequence number ranges from 0 to 7.

Sliding-Window Flow Control

- Each time a frame is sent, the shaded window shrinks (from the left side)
- Each time an acknowledgement is received, the shaded window grows (to the right side)
- Frames between the vertical bar and the shaded window have been sent but not yet acknowledged. The sender must buffer these frames in case they need to be retransmitted
- The window size need *NOT* to be the maximum possible size (more on that later)

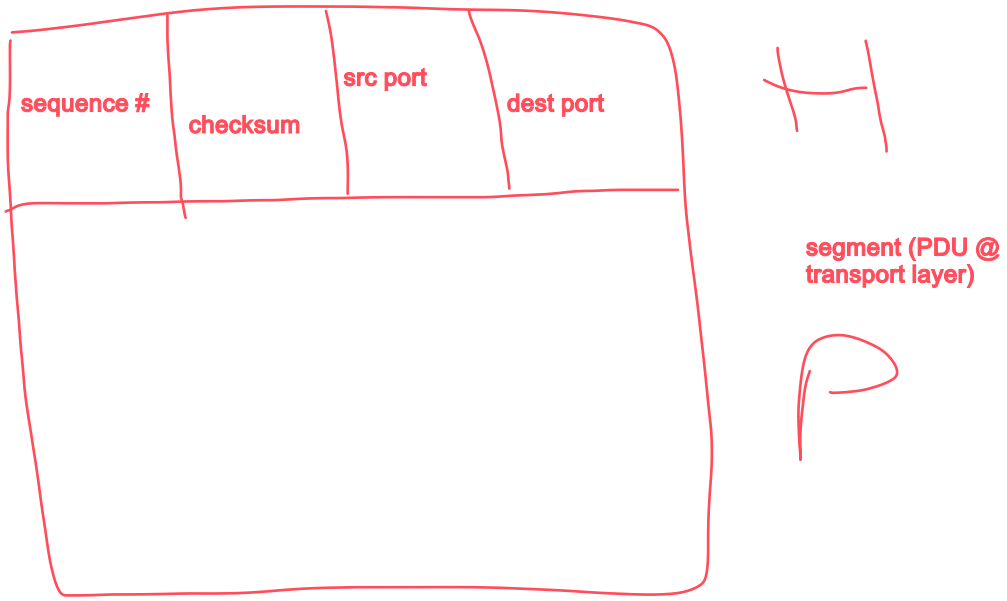
Sliding-Window Flow Control

- Receiver can acknowledge frames without permitting further transmission (Receive Not Ready RNR)
- Thus, RNR 5 means "I have received all frames up to number 4 but am unable to accept any more at this time"
- Later, the receiver must send a normal acknowledgement to resume transmission

sequence # is the frame #

Frame # can go up to 2k-1, k being bits of sequence #

is 4-bits, so can only go up to 15, 0000 -> 1111



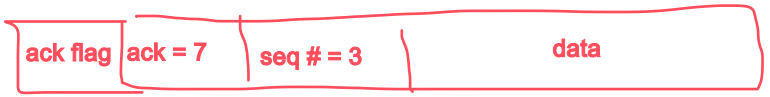
window size is negotiated during connection request

max W = 2k -1, if you use 2k then ambiguity issue occurs. k is bits of sequence #

Sliding-Window Flow Control

- If two stations exchange data (full-duplex), each needs to maintain two windows, one for transmit and one for receive. Also, each side needs to send the data and acknowledgements to the other.
- For efficiency, a feature known as piggybacking if provided.
  - Each data frame includes a field that holds the sequence number of that frame plus a field that holds the sequence number used for acknowledgement.
  - One frame can be used to send both data and acknowledgment, saving communication capacity
  - If a station has an acknowledgment but no data, it sends a separate acknowledgment such as RR or RNR
  - If a station has data to send but no *new* acknowledgment, it must repeat the last acknowledgment sequence number that it sent

acknowledge flag true means ack # is valid if false then its just a filler



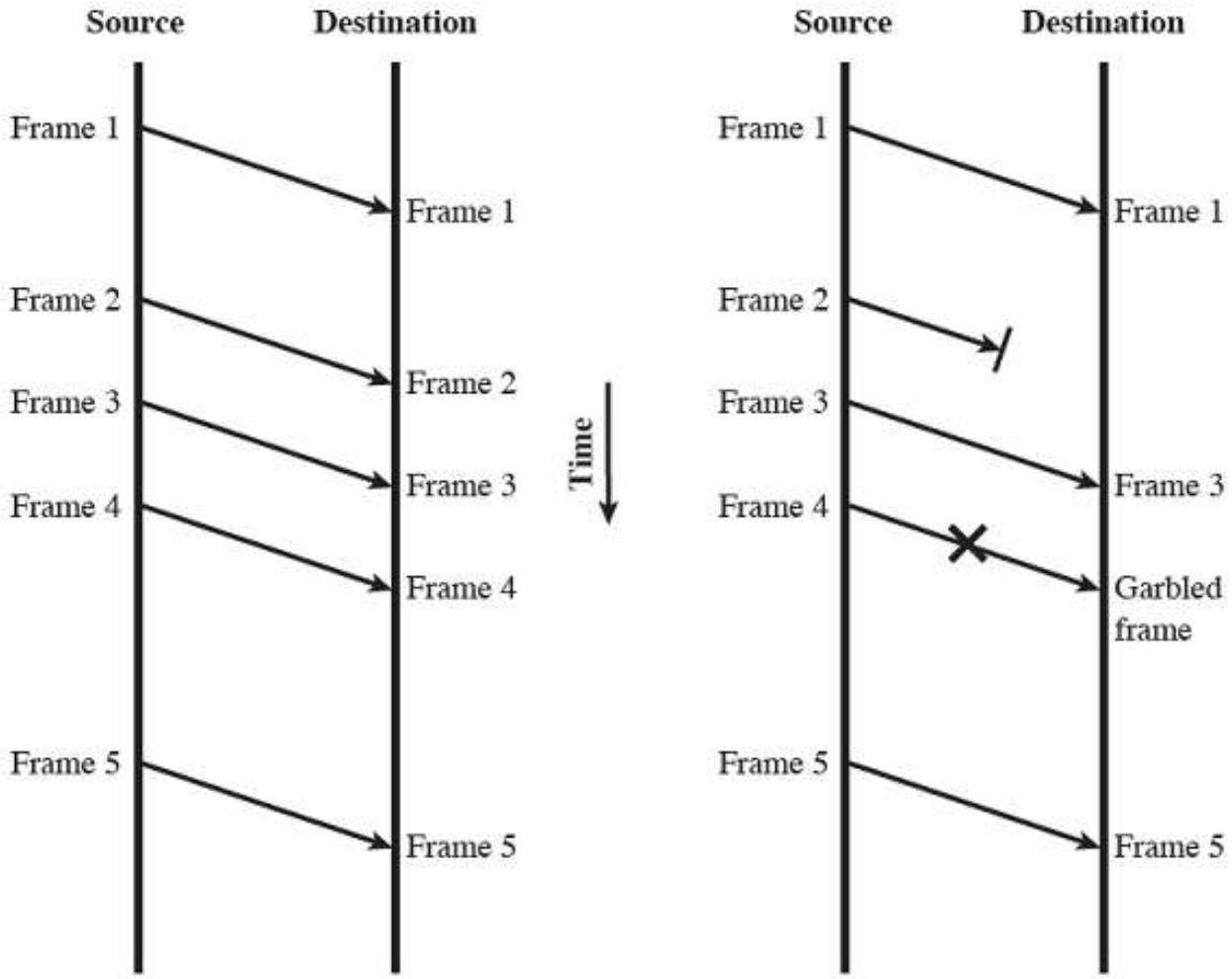
Sliding-Window Flow Control

- Sliding-window flow control is potentially much more efficient than stop-and-wait flow control. With sliding-window flow control, the transmission link is treated as a *pipeline* that may be filled with frames in transit. With stop-and-wait flow control, only one frame may be in the pipe at a time
- Assume 3-bit sequence number (0 to 7), and suppose a transmitter sends frame 0 and gets RR 1. Then, the transmitter sends frames 1, 2, 3, 4, 5, 6, 7, 0, and gets another RR 1. This could mean ALL 8 frames are received with no error or ALL of them are lost and the receiver is repeating the previous RR 1.
  - To avoid such situation, the maximum window size is limited to 7 (2^3-1). In general the maximum window size is 2^k-1

Error Control

- Error control is a basic principle for reliable data transfer
  - Error control refers to mechanisms to detect and correct errors *in the transmission of frames*.
- Two types of errors
  - Lost frame. A frame fails to arrive at the other side
  - Damaged frame. A frame does arrive, but some of bits are in error

Error Control



(a) Error-free transmission

(b) Transmission with losses and errors

Model of Frame Transmission

Error Control

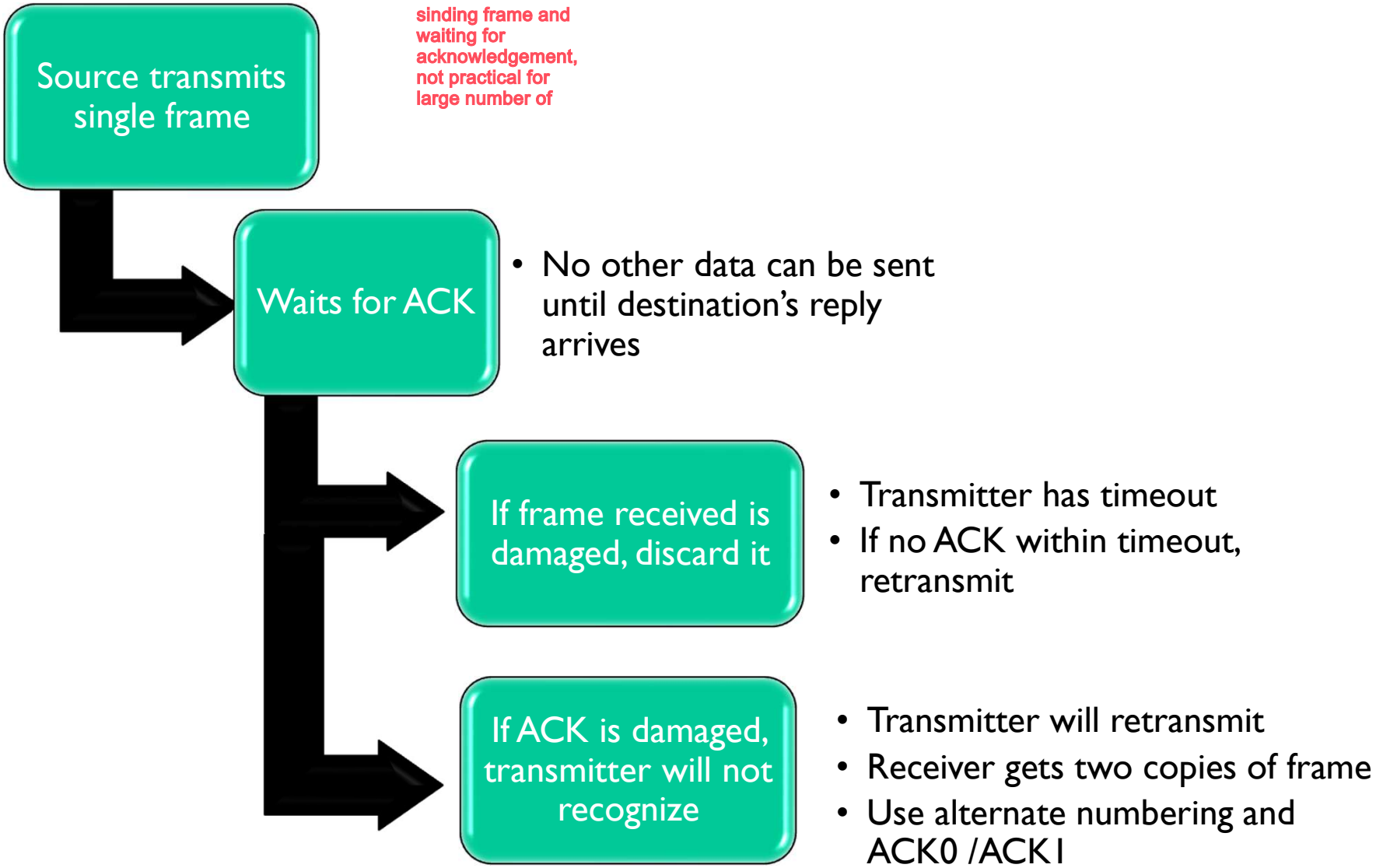
- The most common techniques for error controls are based on some or all of the following ingredients:
  - Error detection. Ignore frame with error
  - Positive acknowledgement. ACK for receiving error-free frame
  - Retransmission after timeout. No ACK within predetermined time, so retransmit
  - Negative acknowledgment and retransmission. ACK for detecting errors in a frame, and source retransmits such a frame

Error Control

Automatic Repeat Request (ARQ)

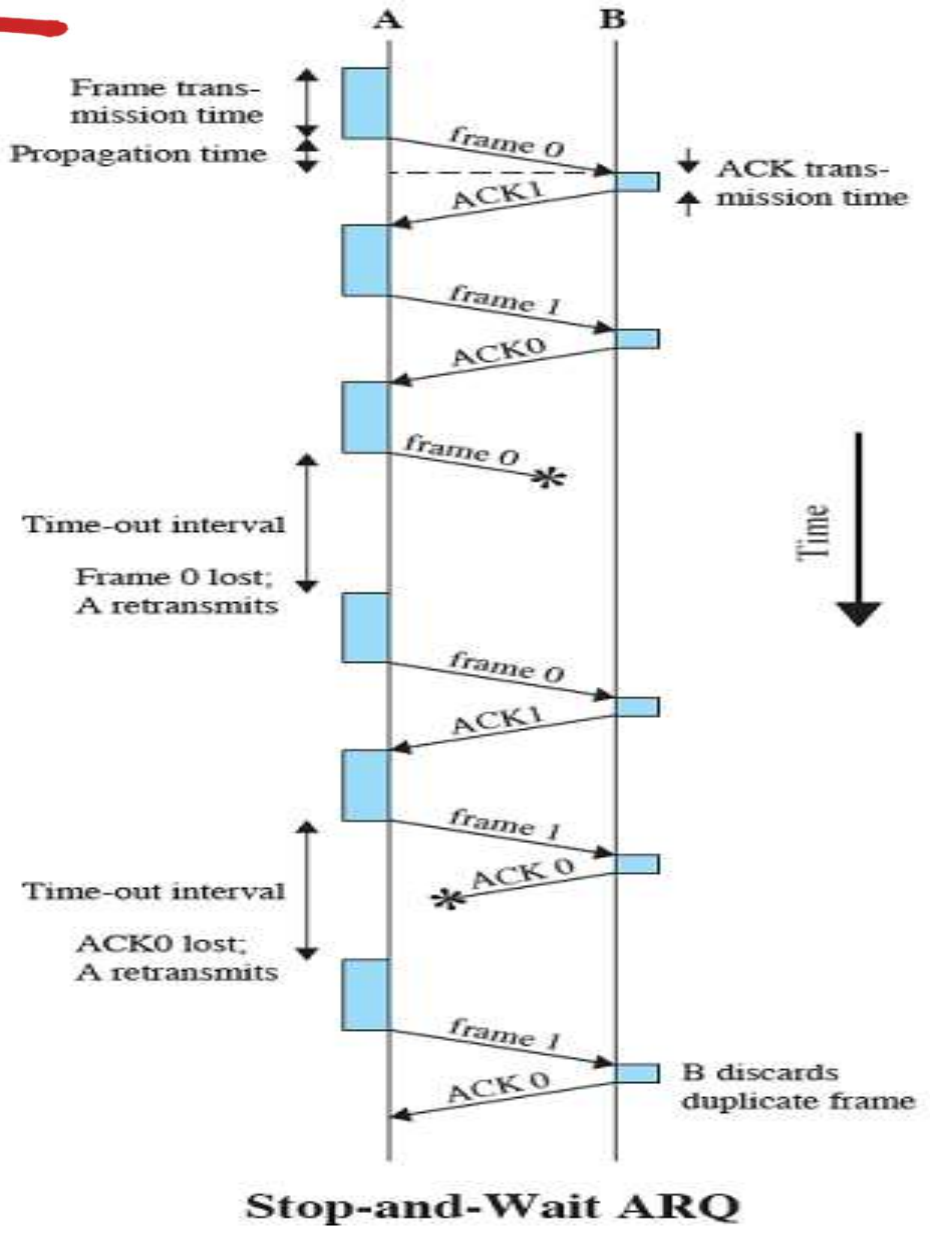
- Collective name for error control mechanisms
- Effect of ARQ is to turn an unreliable data link into a reliable one
- Three versions of ARQ have been standardized
  - Stop-and-wait ARQ
  - Go-back-N ARQ
  - Selective-reject ARQ

Stop-and-wait ARQ



Stop-and-wait ARQ

- The principal advantage of stop-and-wait ARQ is its simplicity
- The principal disadvantage, it is inefficient mechanism



Stop-and-Wait ARQ

send multiple then acknowledgement (sliding window like)

Go-Back-N ARQ

- Most commonly used error control (*why?*)
- Based on sliding-window flow control
- Use window size to control number of outstanding frames
- While no errors occur, the destination will acknowledge incoming frames as usual
  - RR = Receive Ready, or piggybacked acknowledgment
- If the destination detects an error in a frame, it may send a negative acknowledgment
  - REJ = Reject reject has a number so sender can resend that frame and al lsubsequent frames
  - Destination will discard that frame and *all future frames* until the frame in error is received correctly
  - Transmitter must go back and retransmit that frame and *all subsequent frames*

Go-Back-N ARQ

- Case 1: Damaged Frame
  - Receiver detects error in frame ***i***
  - Receiver sends REJ-***i***
  - Transmitter retransmits frame ***i*** and all subsequent frames
- Case 2: Lost Frame
  - Frame ***i*** is lost
  - Transmitter sends ***i*** + 1
  - Receiver gets frame ***i*** + 1 out of sequence
  - Receiver sends reject ***i***
  - Transmitter goes back to frame ***i*** and retransmits

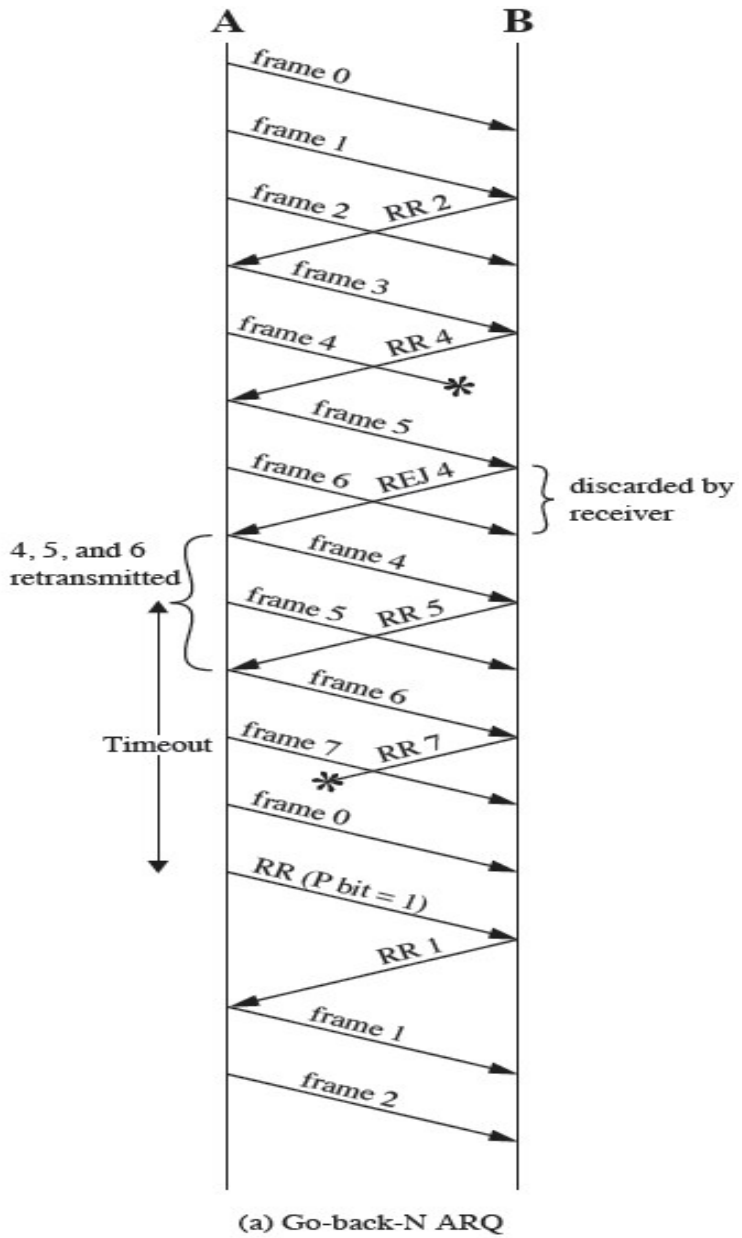
Go-Back-N ARQ

- Case 3: Lost Frame: Frame ***i*** is lost and no additional frame sent
  - Receiver gets nothing and returns neither acknowledgement nor rejection
  - Transmitter times out and sends acknowledgement frame with ***P*** bit set to 1
  - Receiver interprets this as a command which it acknowledges with the number of the next frame it expects (frame ***i***)
  - Transmitter then retransmits frame ***i***

Go-Back-N ARQ

- Case 4: Damaged Acknowledgement RR
  - Receiver gets frame ***i*** and send acknowledgement (***i***+1) which is lost
  - Acknowledgements are cumulative, so next acknowledgement (***i*** + *n*) may arrive before transmitter times out on frame ***i***
  - If transmitter times out, it sends acknowledgement with **P** bit set as before
  - This can be repeated a number of times before a reset procedure is initiated
- Case 5: Damaged Rejection REJ. As for Frame ***i*** is lost and no additional frame sent

Go-Back-N ARQ



(a) Go-back-N ARQ

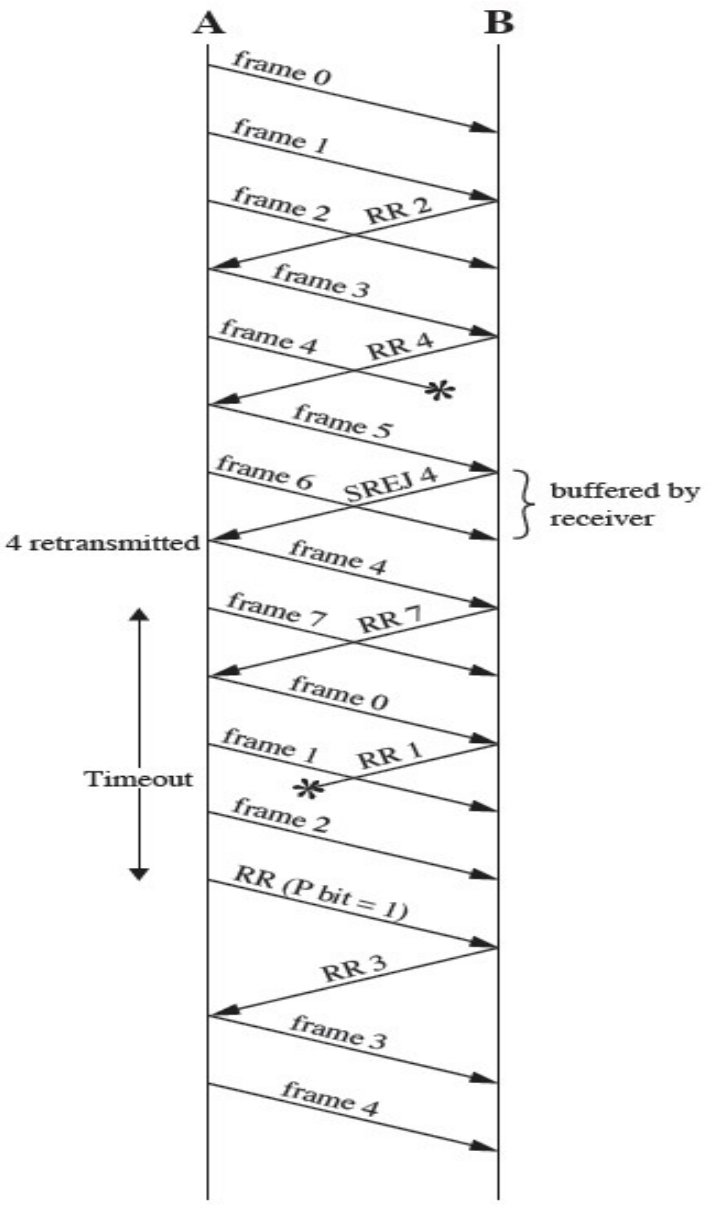
Selective-Reject ARQ

good for very high propogation delay scenarios and transmission delay scenarios

bc you dont have to go through that delay N times as in go-back-N

- Only rejected frames are retransmitted (frames that receive **SREJ**)
- Subsequent frames are accepted by the receiver and buffered
- Minimizes retransmission
- Receiver must maintain large enough buffer
- Receiver must contain logic for inserting frames in proper sequence
- Transmitter requires more complex logic to be able to send a frame out of sequence

Selective-Reject ARQ



(b) Selective-reject ARQ

Outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP

- segment structure
- reliable data transfer
- connection management

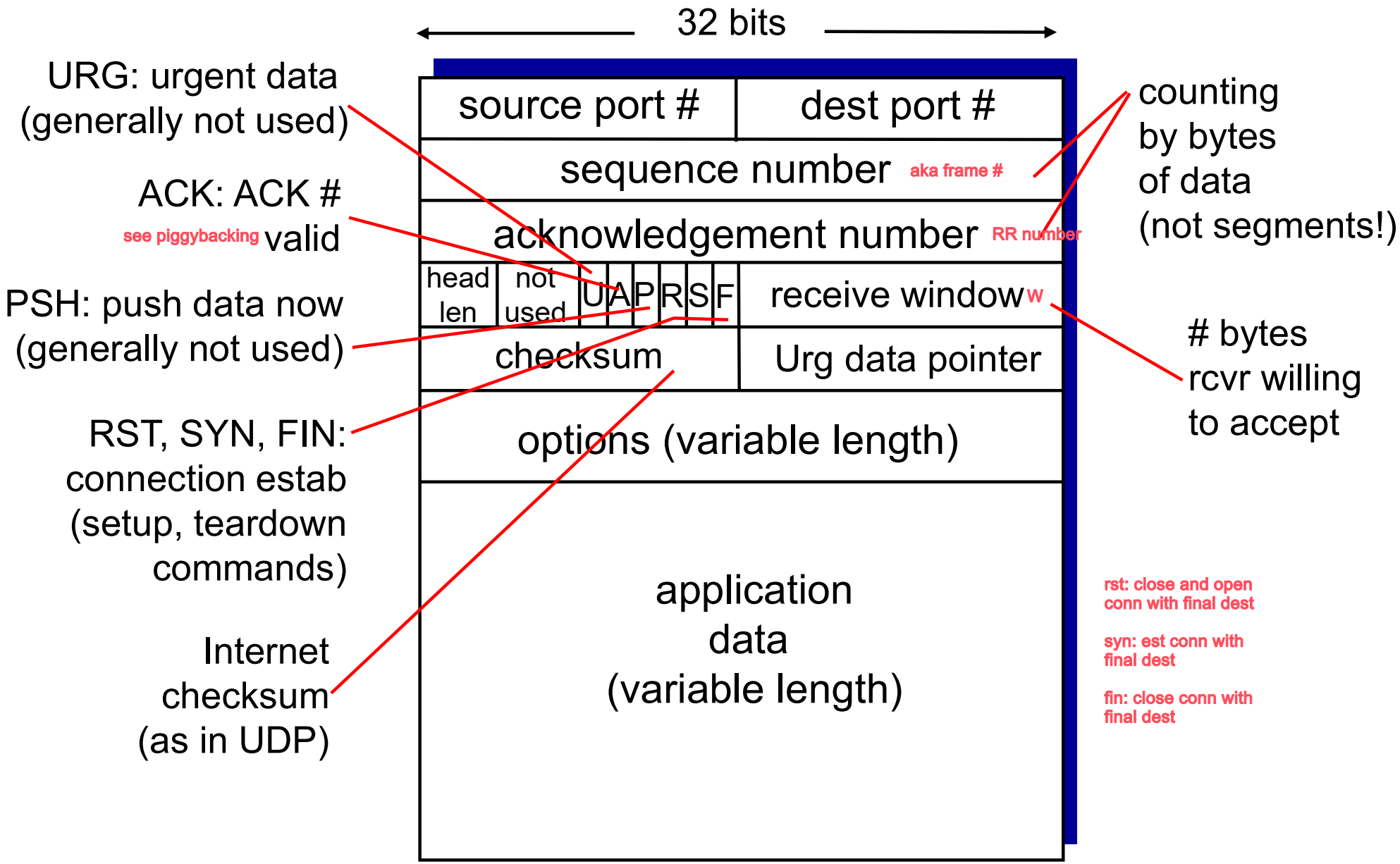
udp header: dest port, src port, checksum

tcp header: below, more attributes so more reliable but again slower

TCP: Overview RFCs: 793,1122,1323, 2018, 2581

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte stream:*
  - no "message boundaries"
- pipelined:
  - TCP congestion and flow control set window size
- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- connection-oriented:
  - handshaking (exchange of control msgs) inits sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

TCP segment structure



TCP seq. numbers, ACKs

index or first byte of frame

sequence numbers:

- byte stream "number" of first byte in segment's data

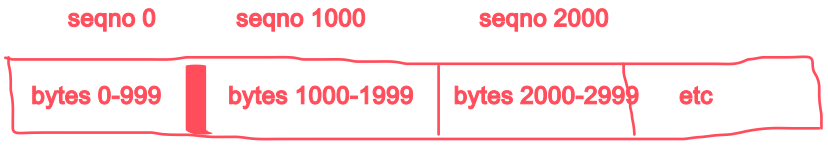
acknowledgements:

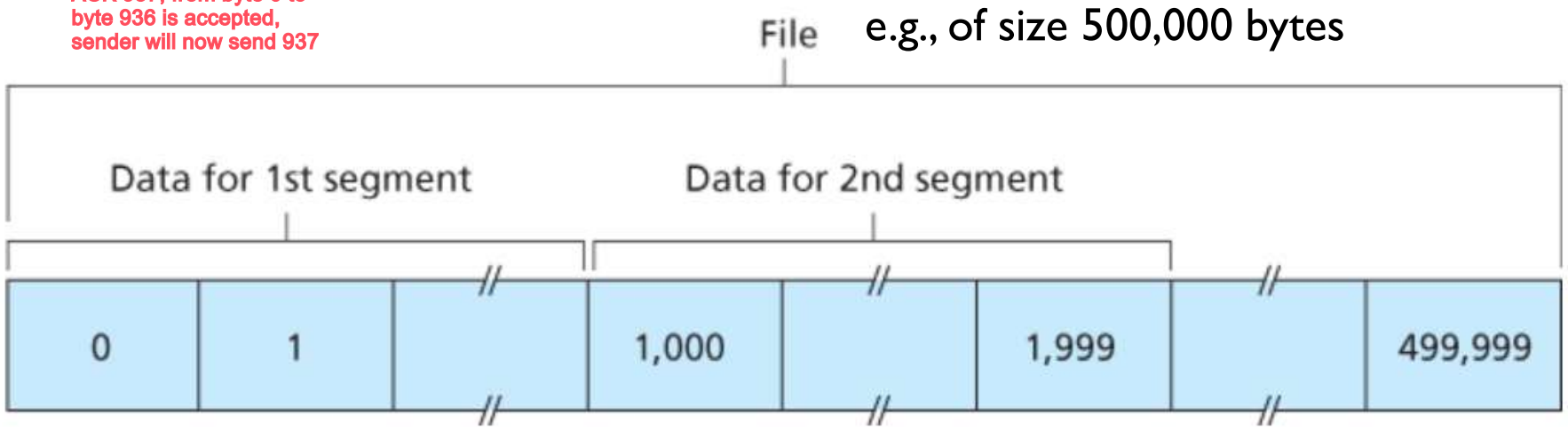
- seq # of next byte expected from other side
- cumulative ACK

ACK 937, from byte 0 to byte 936 is accepted, sender will now send 937

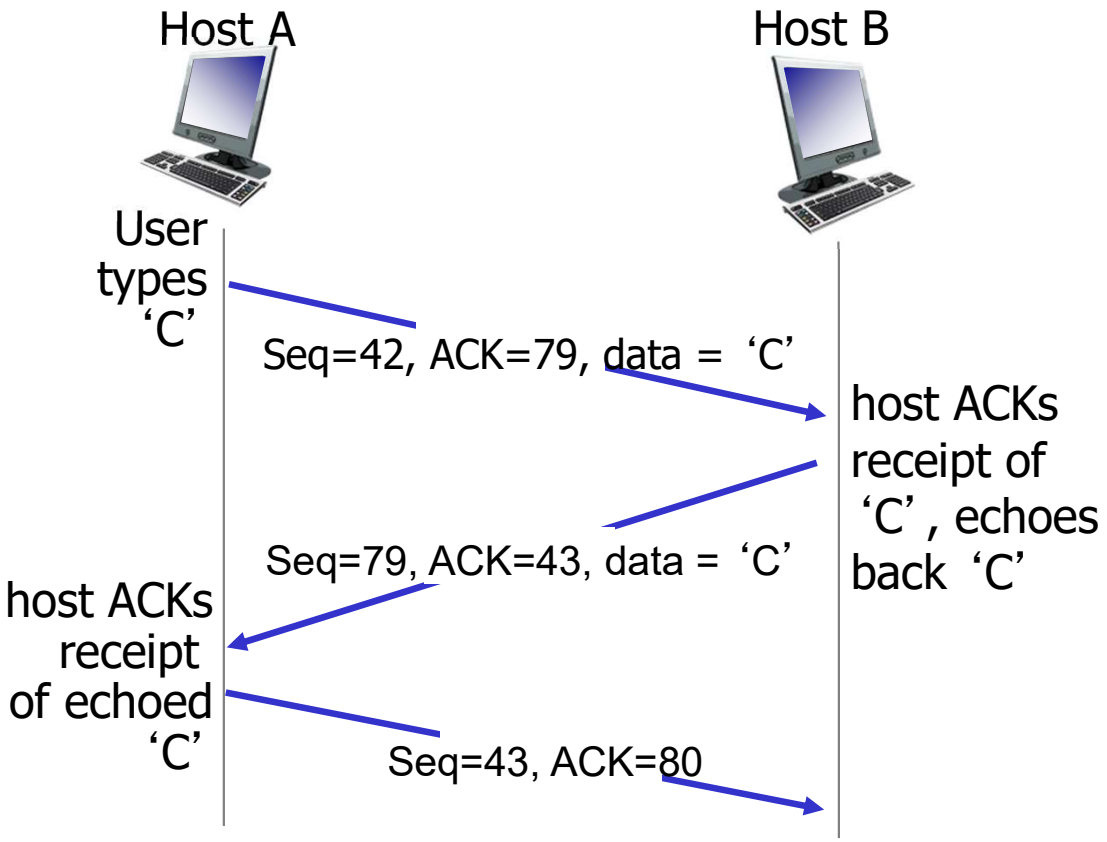
Q: how receiver handles out-of-order segments

- A: TCP spec doesn't say, - up to implementer





TCP seq. numbers, ACKs



simple telnet scenario

Outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

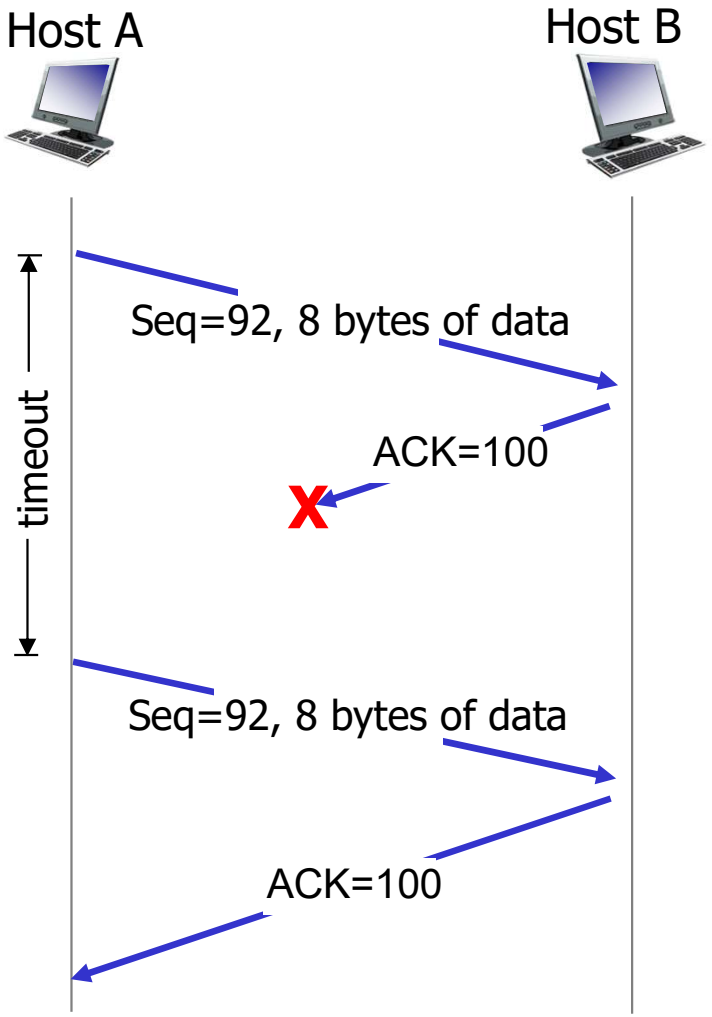
3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

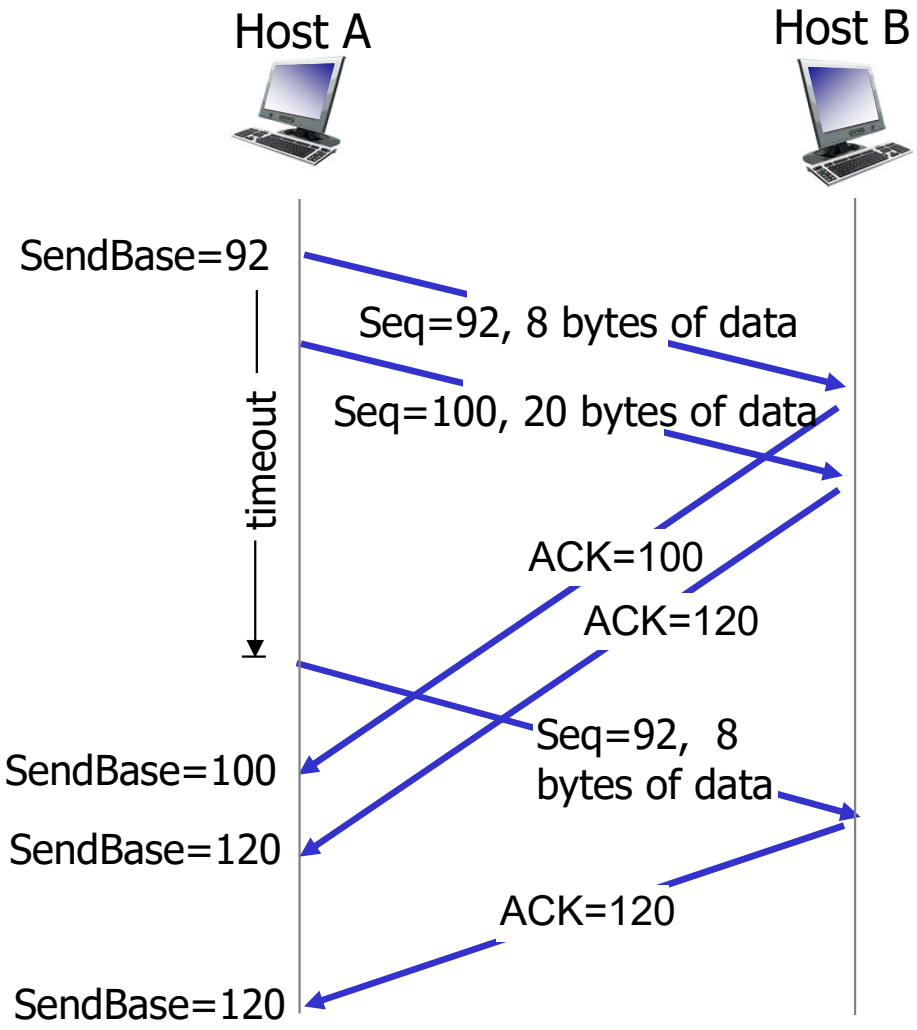
3.5 connection-oriented transport: TCP

- segment structure
- reliable data transfer
- connection management

TCP: retransmission scenarios

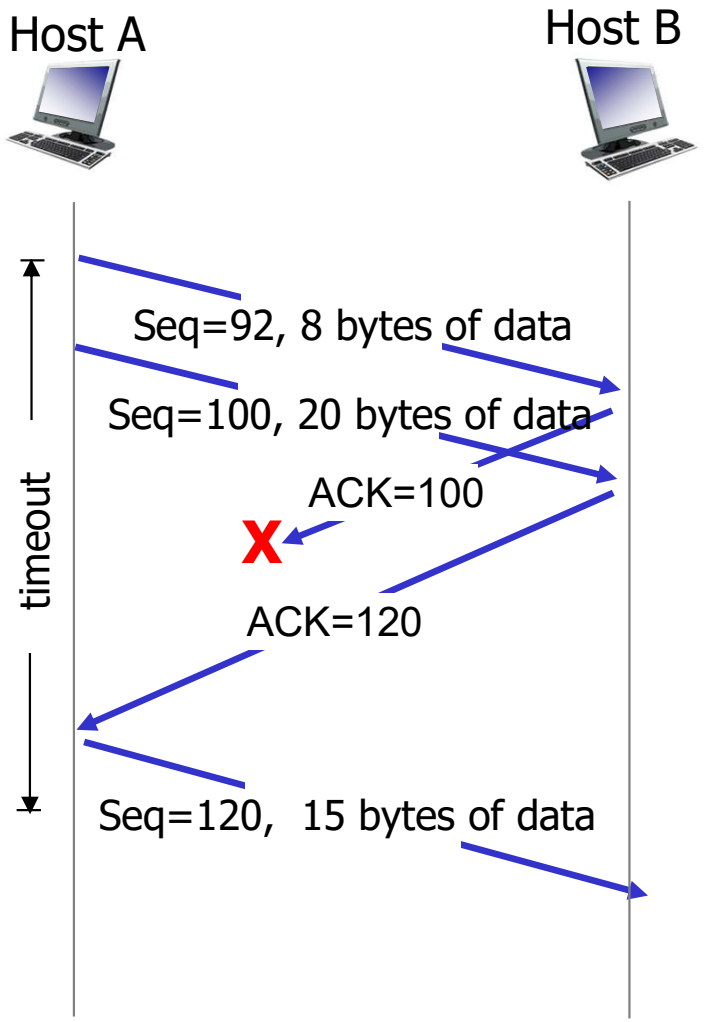


lost ACK scenario



premature timeout

TCP: retransmission scenarios



cumulative ACK

TCP fast retransmit

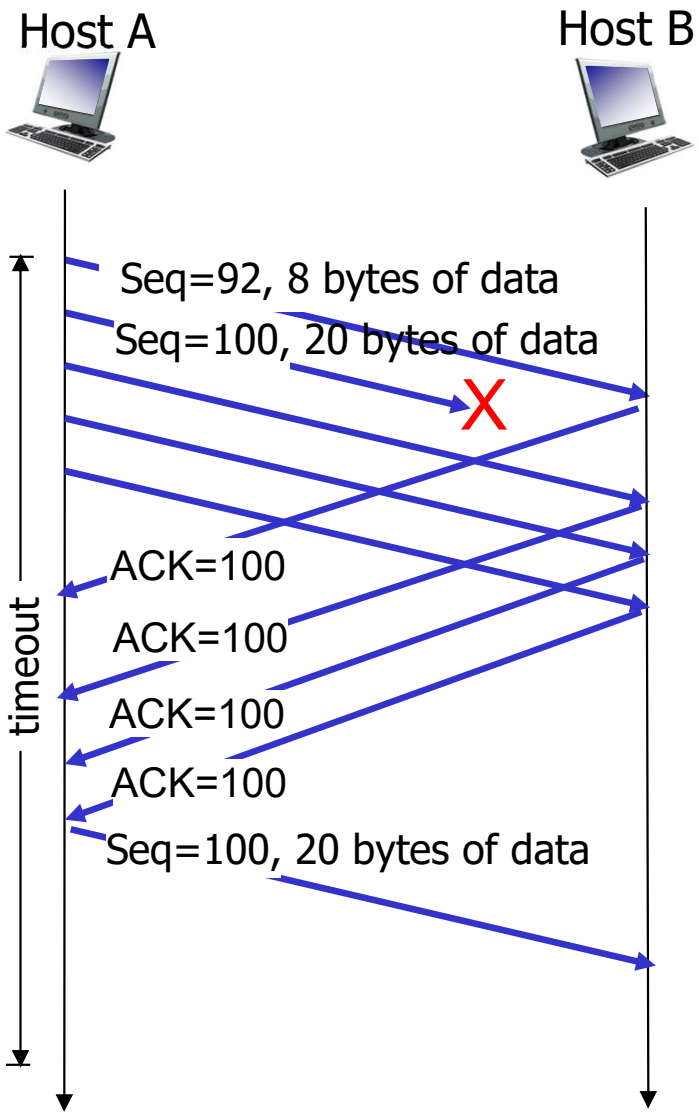
- time-out period often relatively long:
  - long delay before resending lost packet
- detect lost segments via duplicate ACKs.
  - sender often sends many segments back-to-back
  - if segment is lost, there will likely be many duplicate ACKs.

TCP fast retransmit

if sender receives 3 ACKs for same data ("triple duplicate ACKs"), resend unacked segment with smallest seq #

- likely that unacked segment lost, so don't wait for timeout

TCP fast retransmit



frame lost or corrupted will do the ack ack ack to ask for another. meaning REJ is implemented as ack

fast retransmit after sender receipt of triple duplicate ACK

Outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

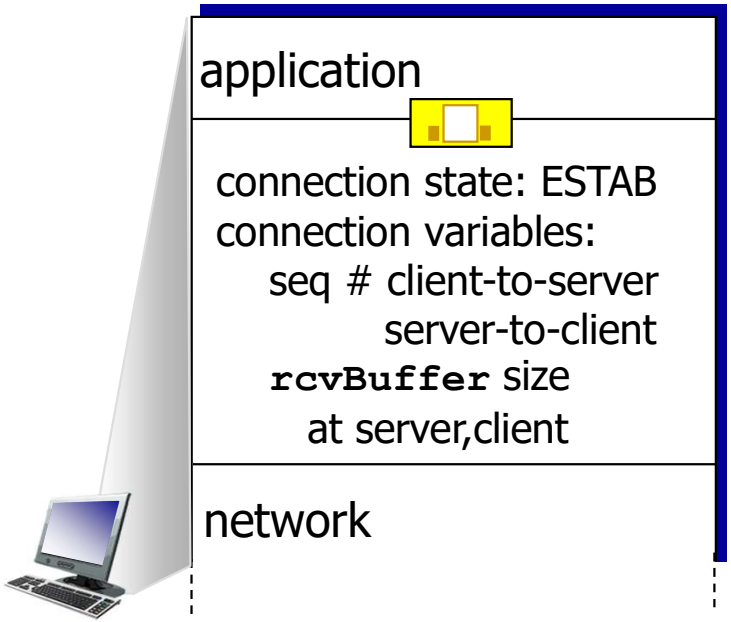
3.5 connection-oriented transport: TCP

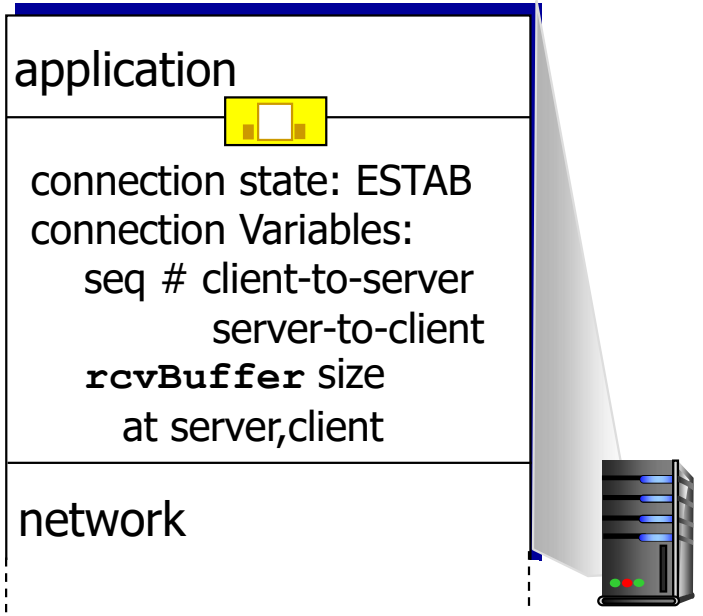
- segment structure
- reliable data transfer
- connection management

Connection Management

before exchanging data, sender/receiver "handshake":

- agree to establish connection (each knowing the other willing to establish connection)
- agree on connection parameters

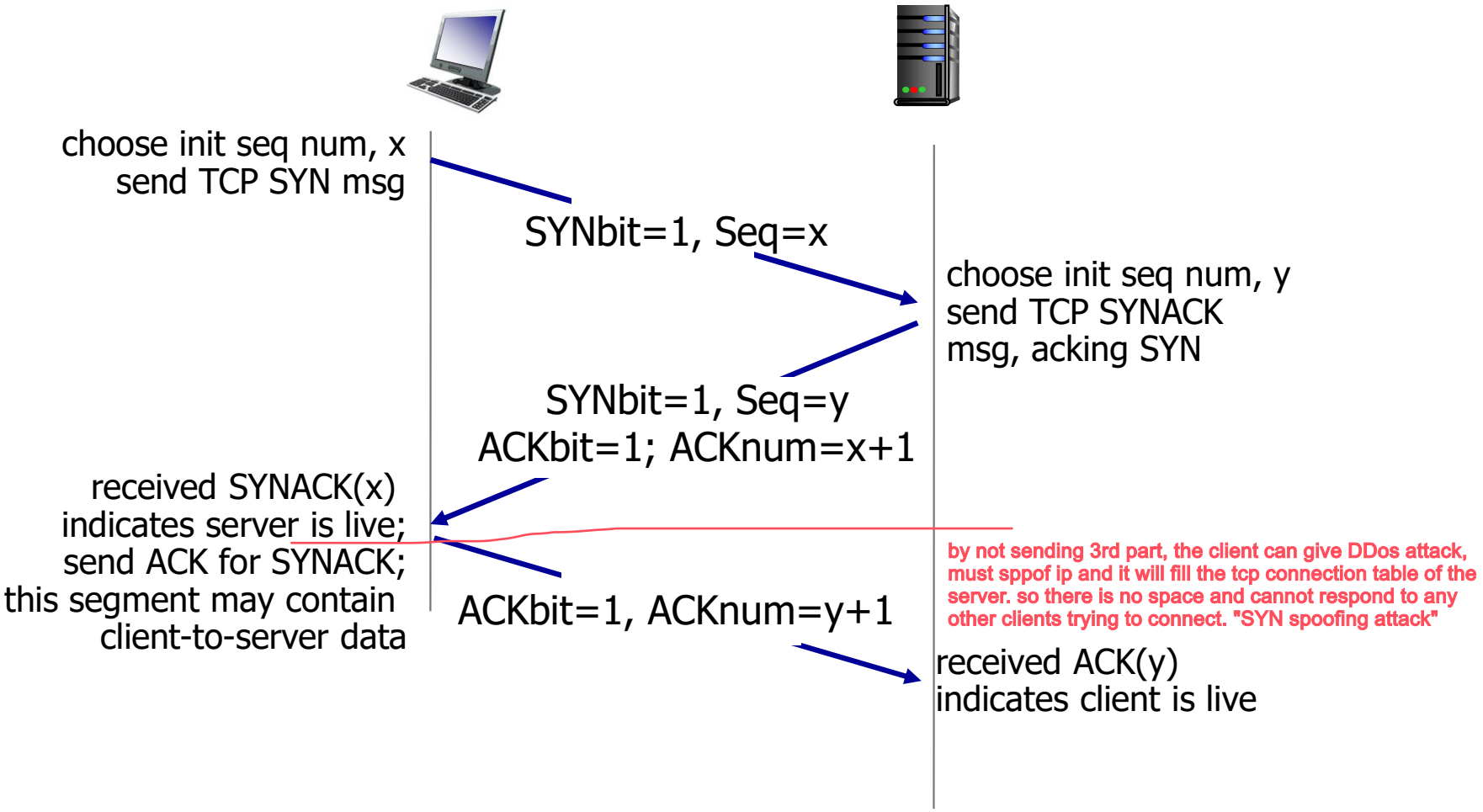





```
Socket clientSocket =
  newSocket("hostname","port
   number");
```

```
Socket connectionSocket =
   welcomeSocket.accept();
```

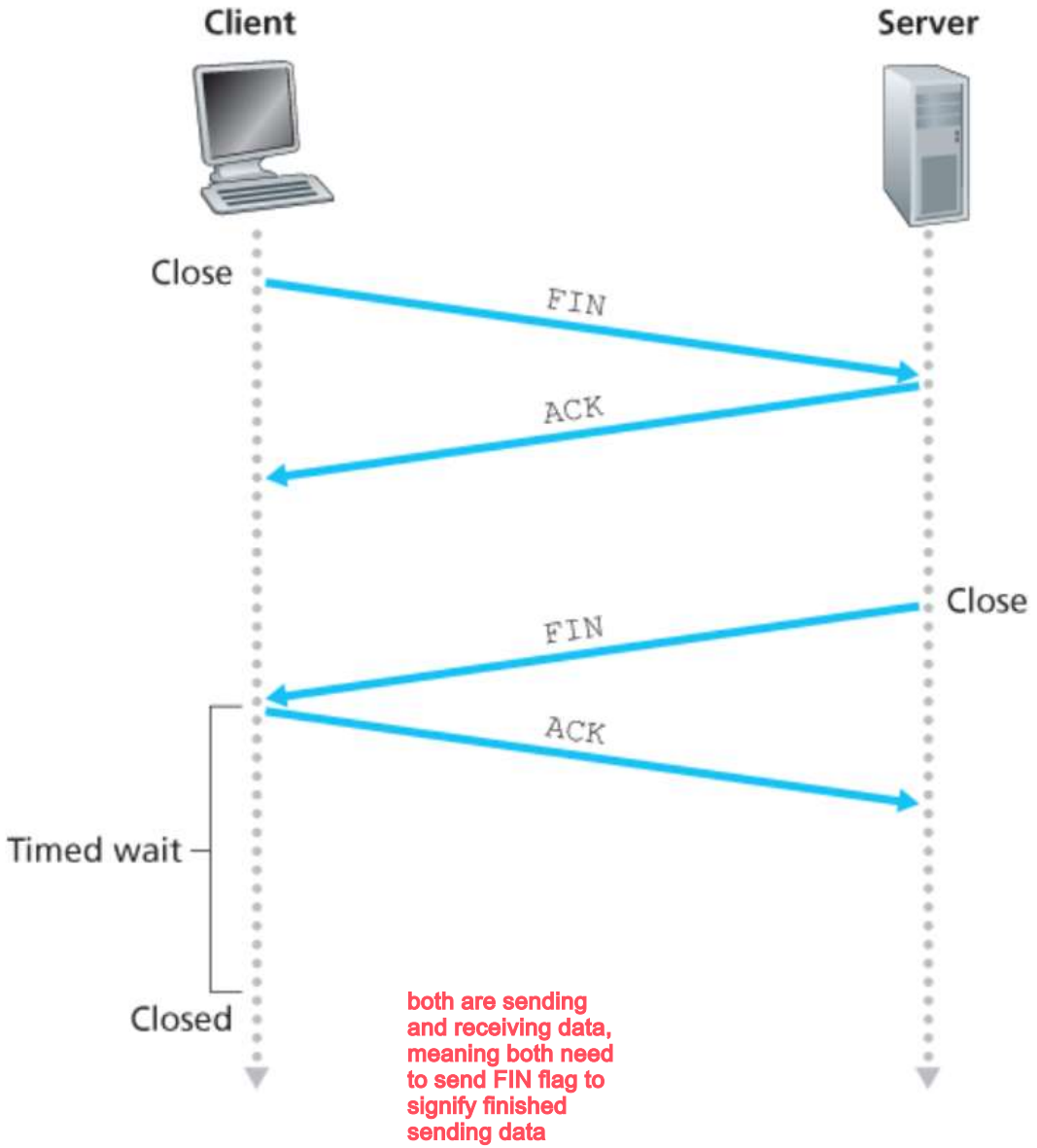
TCP 3-way handshake



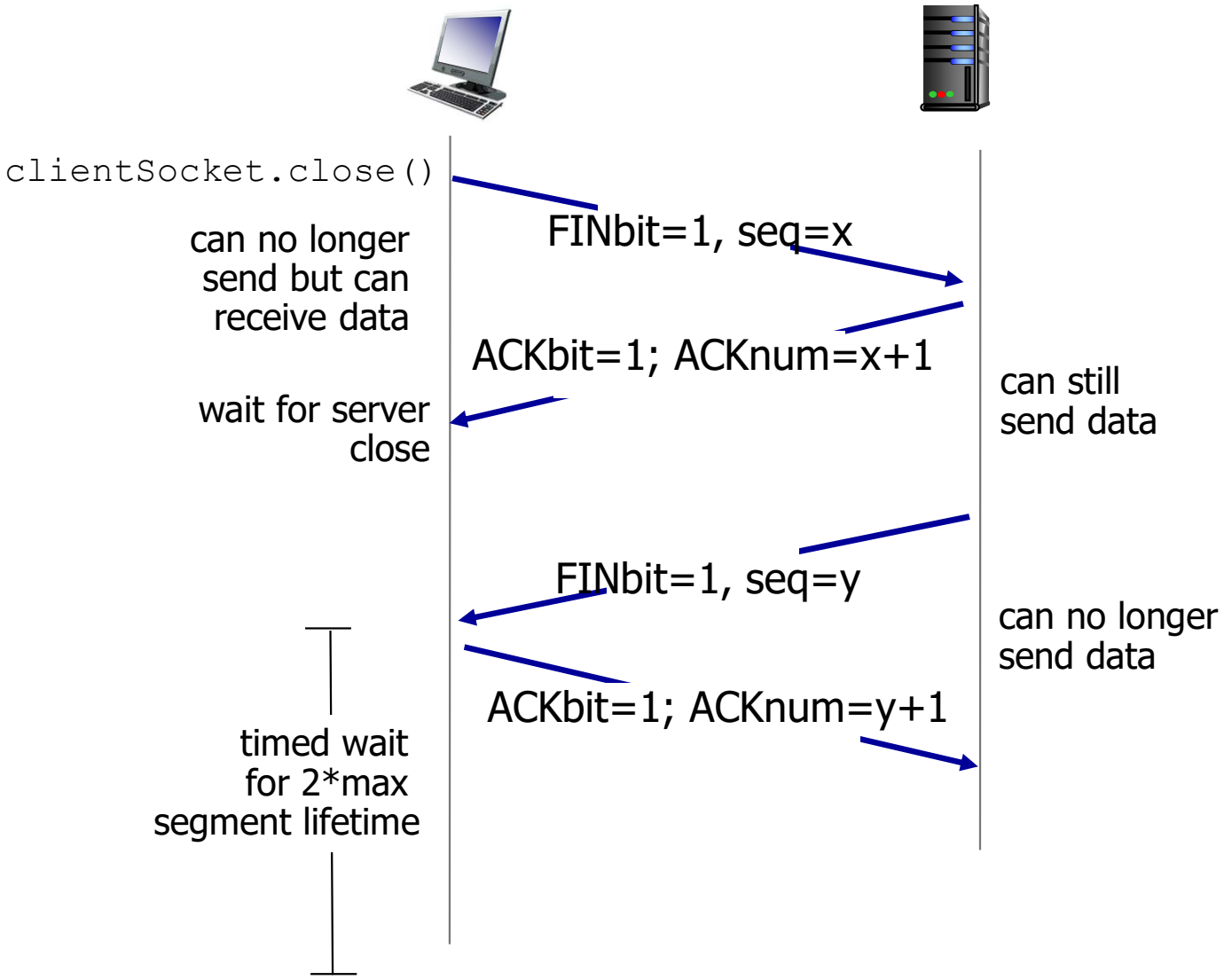
TCP: closing a connection

- client, server each close their side of connection
  - send TCP segment with FIN bit = 1
- respond to received FIN with ACK
  - on receiving FIN, ACK can be combined with own FIN
- simultaneous FIN exchanges can be handled

TCP: closing a connection



TCP: closing a connection



Set 3: summary

- principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- instantiation, implementation in the Internet
  - UDP
  - TCP

next:

- leaving the network "edge" (application, transport layers)
- into the network "core"
- two network layer chapters:
  - data plane
  - control plane