ELEC / COMP 177 – Fall 2012

Computer Networking
→ Transport Layer (TCP & UDP)

Some slides from Kurose and Ross, *Computer Networking*, 5th Edition

Schedule

- **Tuesday**
  - Transport Layer (UDP/TCP)
  - **Homework 2 assigned**
- **Thursday**
  - Transport Layer (UDP/TCP)
- **Next Tuesday**
  - **Homework 2 due**
  - C Programming tutorial
    + Virtual Machine debug day

Bring your laptop next Tuesday!

Virtual Machine Installation

- Have a Linux Virtual Machine or dual-boot setup already?
- **Yes?** Just re-use it for this class
- **No?** Follow instructions from ECPE 170:
  - http://ecs-network.serv.pacific.edu/ecpe-170
    - Under "Tutorials"-> "Virtual Machine Setup"

Do this independently by next Tuesday (Sept 25th)
Email me if you have problems!

Software Installation

- Packages to install
  - Eclipse IDE with C/C++ Development Tools (CDT)
  - Compiler / build tools

```
sudo apt-get install build-essential eclipse-cdt
```

Transport Layer

Recap – Network Model

Application Layer
Transport Layer
Network Layer
Link Layer
Physical Layer

Recap – Physical Layer

Application Layer

Transport Layer

Network Layer

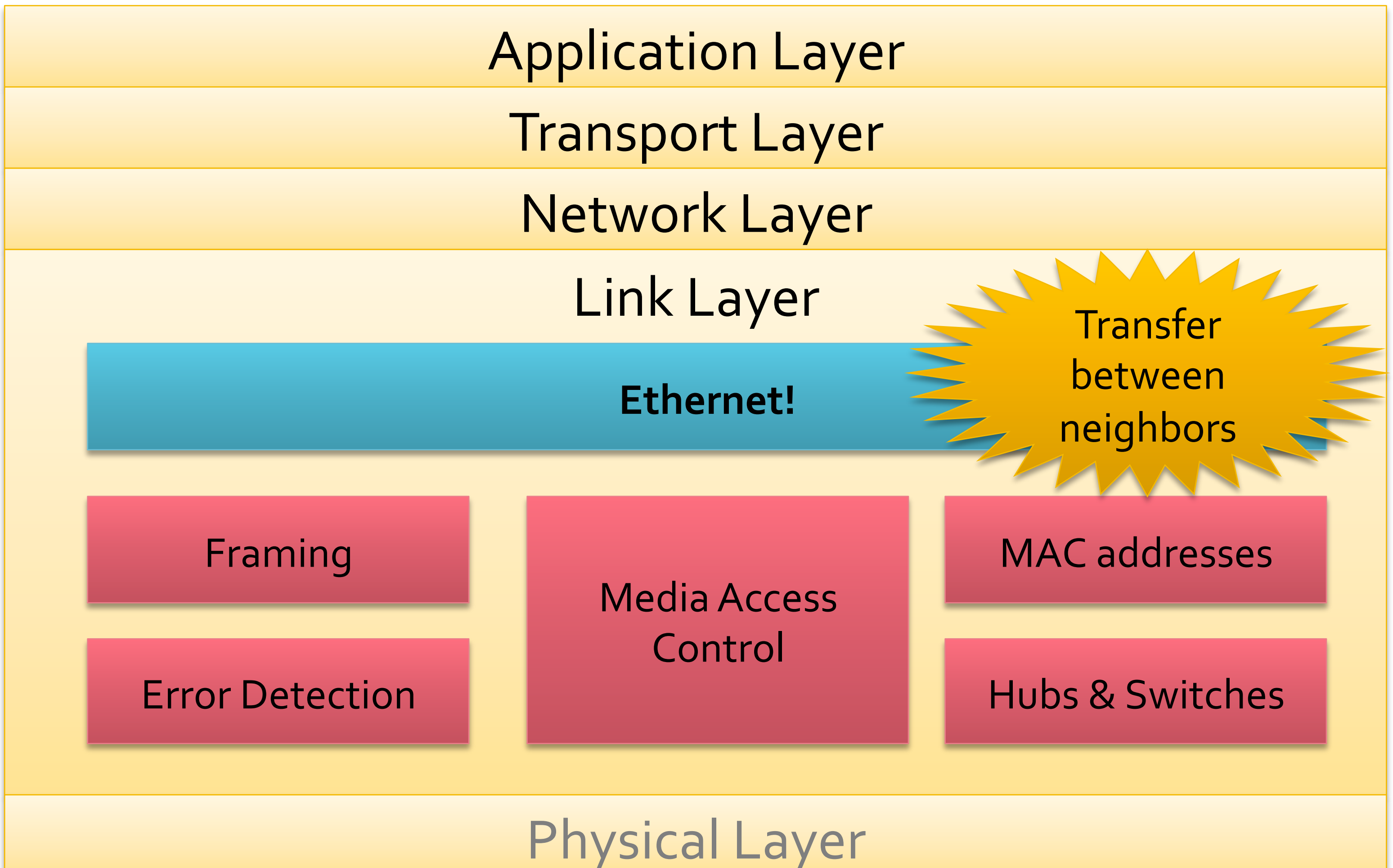
Link Layer

Physical Layer

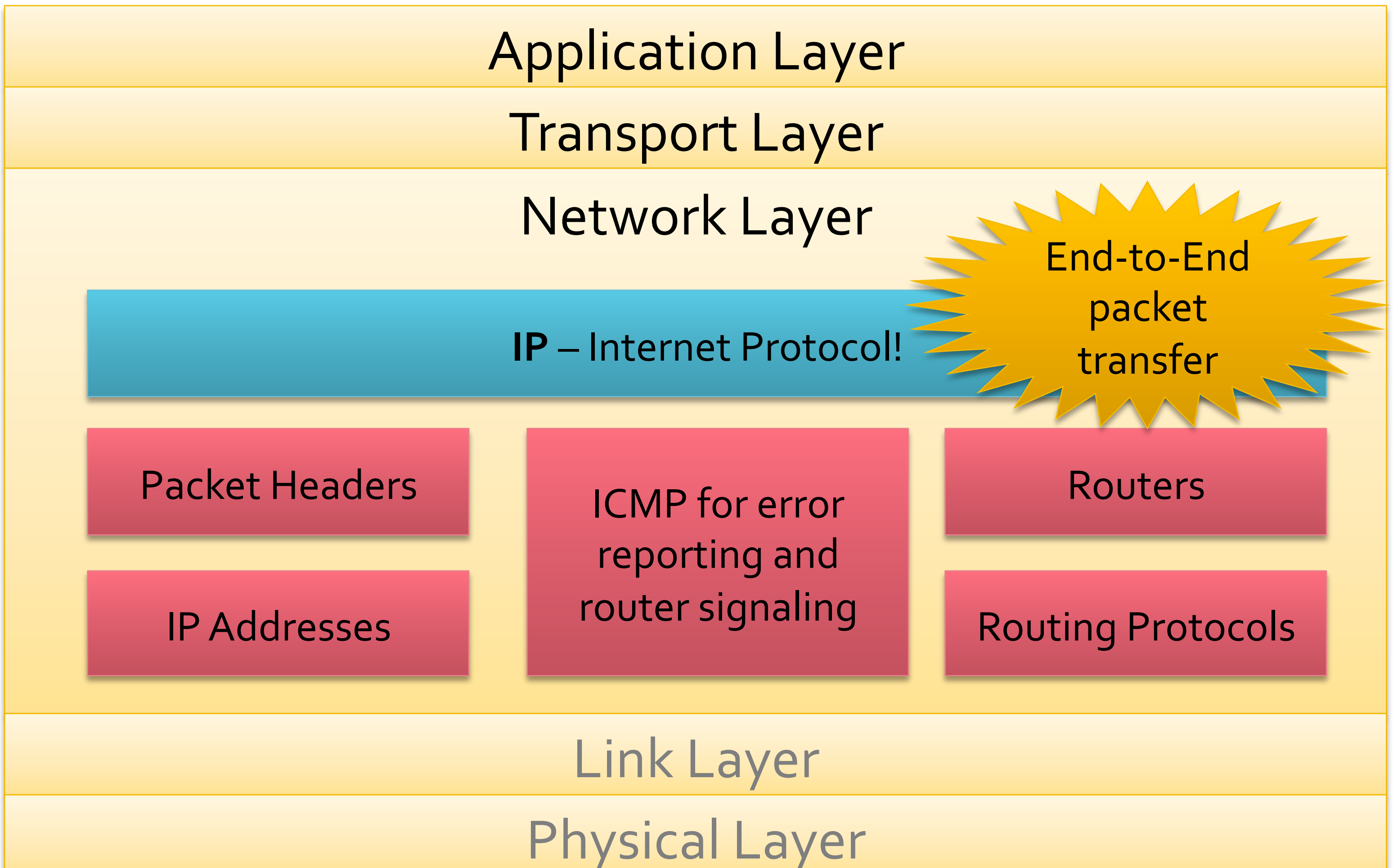
"Bits on a wire"

Encoding schemes fight:
attenuation
distortion
clock skew

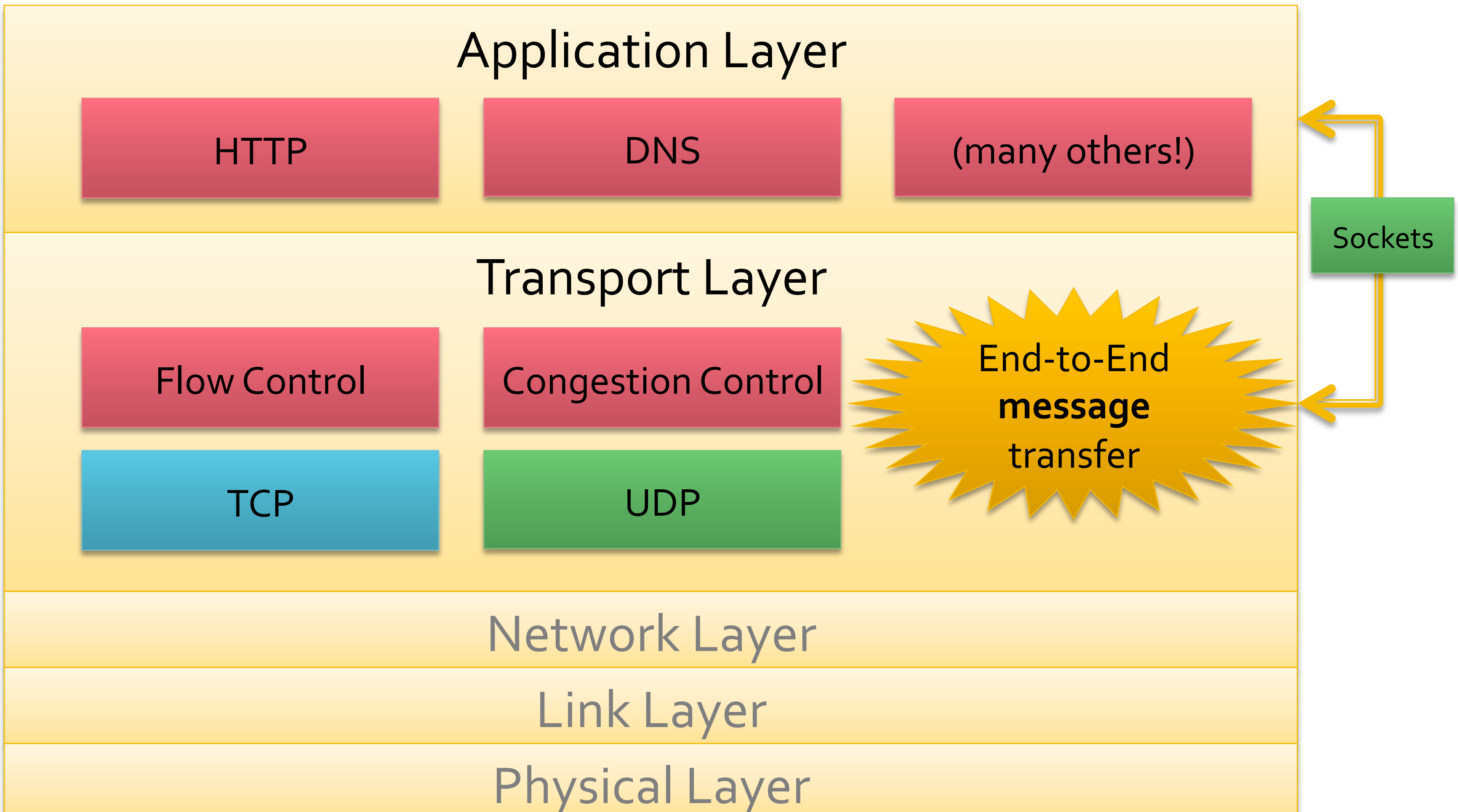
Recap – Link Layer



Recap – Network Layer



Introducing the Transport Layer

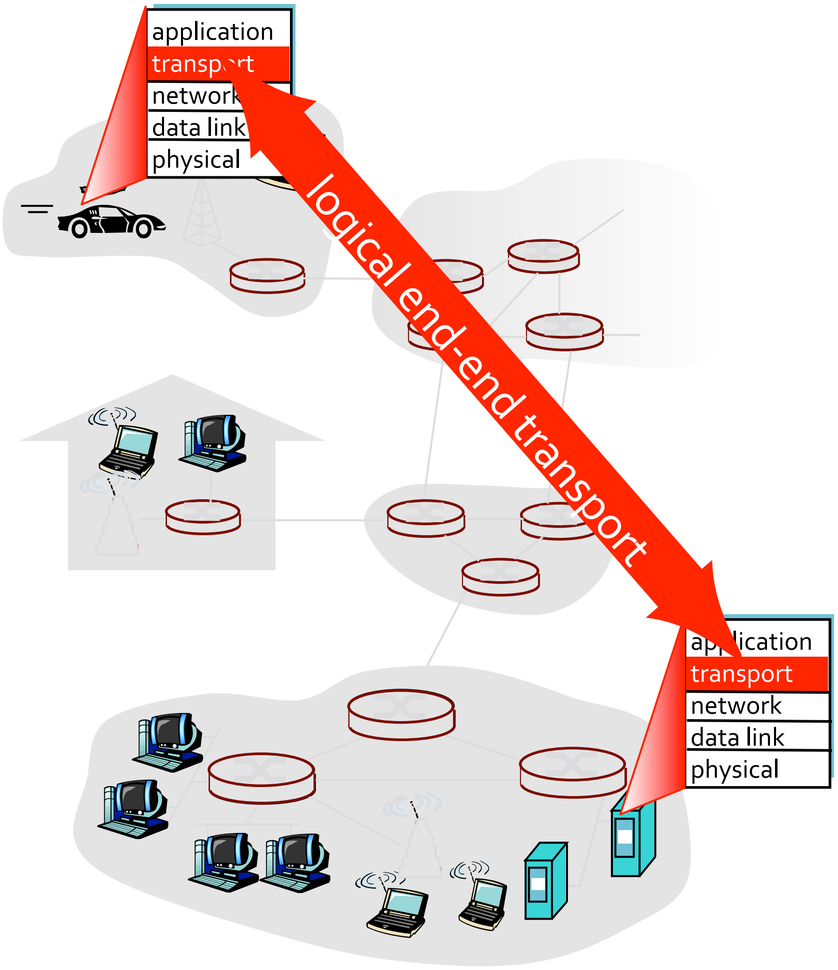


Goals this Week

- Understand principles behind transport layer services:
  - Multiplexing/demultiplexing
  - Reliable data transfer
  - Flow control
  - Congestion control
- Learn about transport layer protocols in the Internet:
  - UDP: connectionless transport
  - TCP: connection-oriented transport
    - Flow control + congestion control

Goal of Transport Layer

- Provide *logical communication* between application processes running on different hosts
- Transport protocols run in end systems
  - Send side: breaks app messages into segments, passes to network layer
  - Receive 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

- *Transport layer:* logical communication between **processes**
  - Relies on and enhances network layer services
- *Network layer:* logical communication between **hosts**

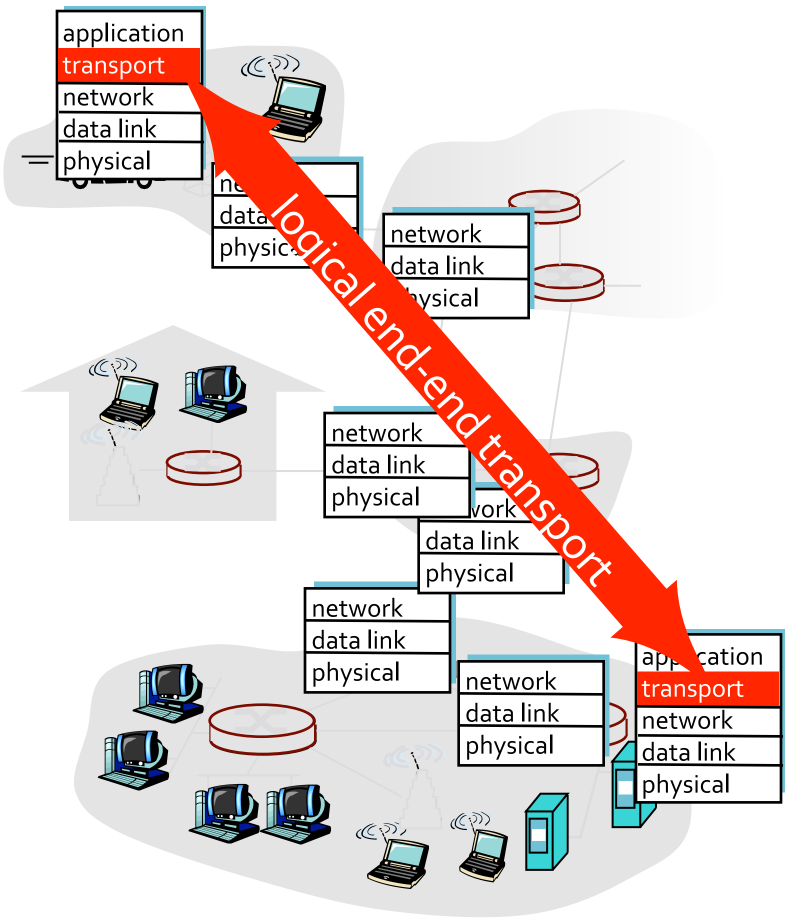
Household analogy:

12 kids sending letters to 12 kids

- Processes = kids
- App messages = letters in envelopes
- Hosts = houses
- Transport protocol = Parents
- Network-layer protocol = postal service

Internet Transport-layer Protocols

- Unreliable, unordered delivery (UDP)
  - No-frills extension of "best-effort" IP
- Reliable, in-order delivery (TCP)
  - Congestion control
  - Flow control
  - Connection setup
- Services not available:
  - Delay guarantees
  - Bandwidth guarantees



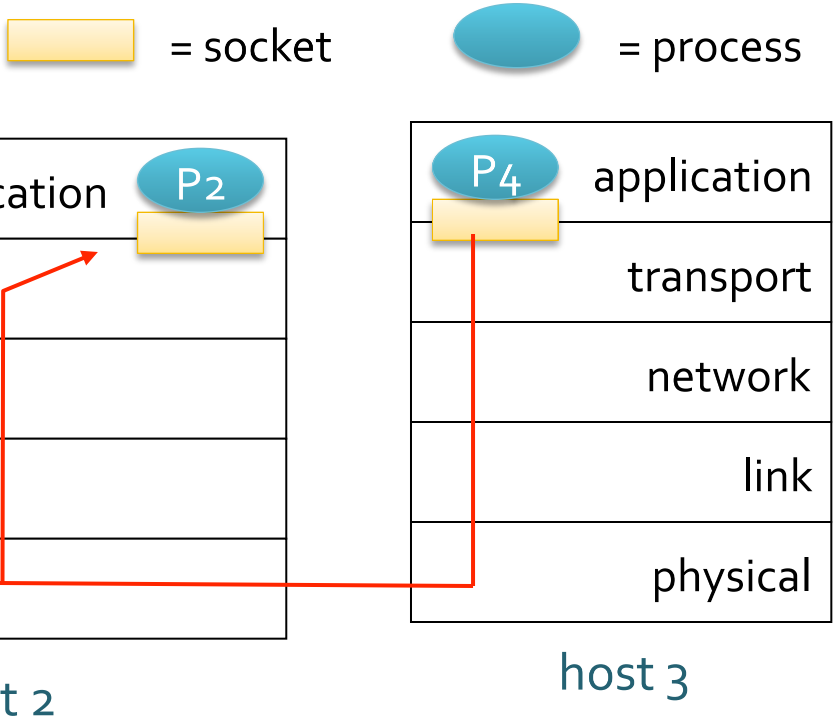
Multiplexing / Demultiplexing

Multiplexing at send host:

Gathering data from multiple sockets, enveloping data with header (later used for demultiplexing)

Demultiplexing at recv host:

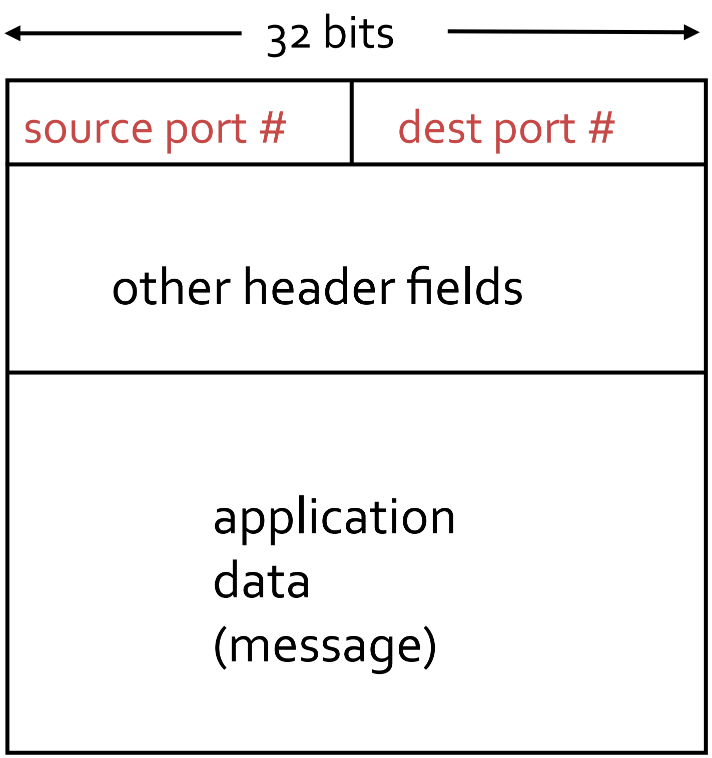
Delivering received segments to correct socket



Demultiplexing Operation

- Host receives **IP datagrams**
  - Each datagram has source and destination IP addresses
  - Each datagram carries 1 **transport-layer segment**
  - Each segment has source and destination **port number**
- Host uses IP addresses & port numbers to direct segment to appropriate socket

TCP/UDP segment format

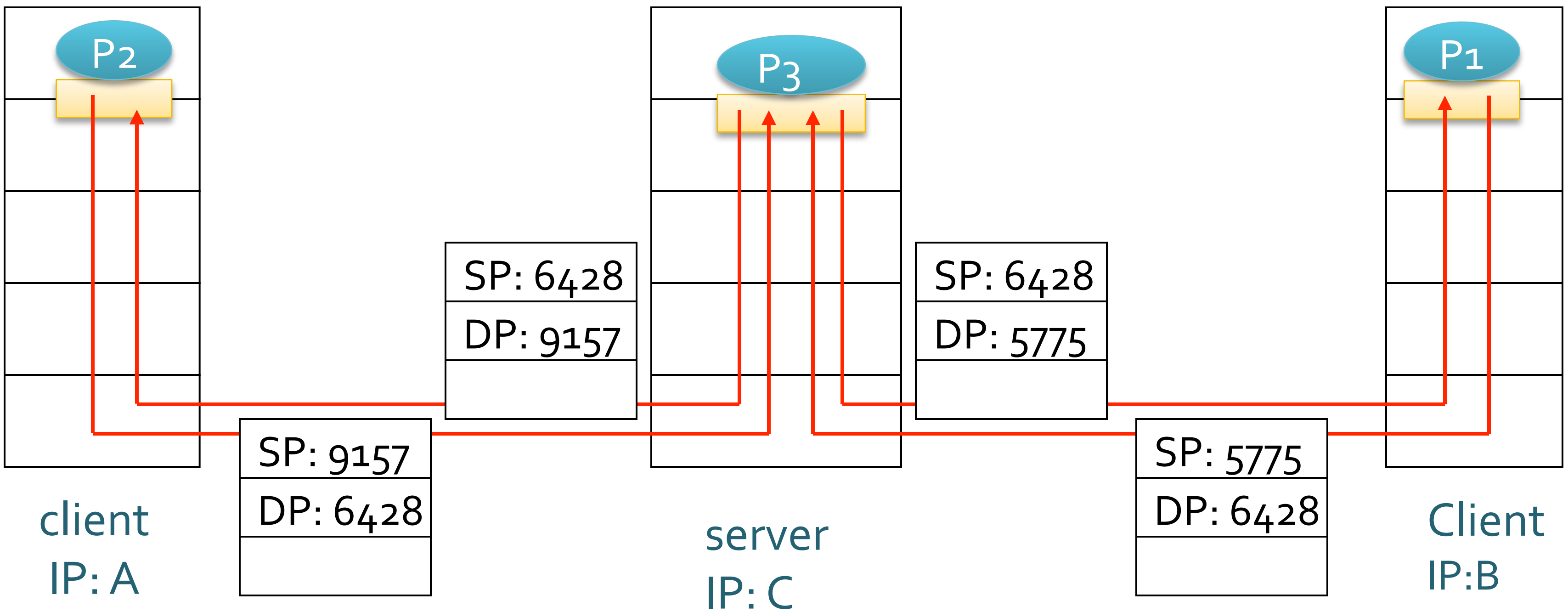


Connectionless Demultiplexing

- Create sockets with port numbers
- UDP socket identified by two keys:
  - **(dest IP address, dest port number)**
- When host receives UDP segment:
  - Check destination port number in segment
  - Direct UDP segment to socket with that port number
- IP datagrams with different source IP addresses and/or source port numbers are directed to same socket on receiver

Connectionless Demultiplexing

Server C is listening on port 6428

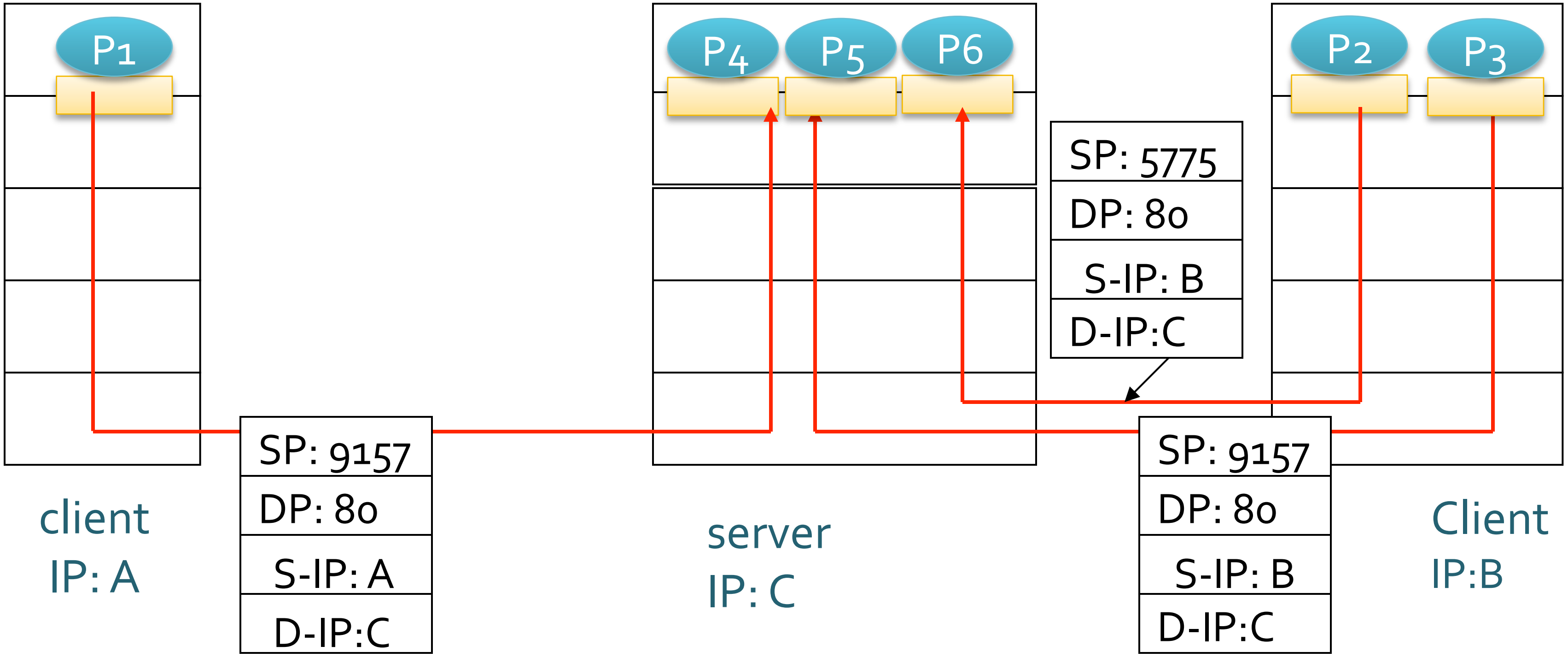


Source Port (SP) provides "return address"

Connection-Oriented Demux

- TCP socket identified by 4 keys:
  - Source IP address
  - Source port number
  - Dest IP address
  - Dest port number
- Receiving host uses all four values to direct segment to appropriate socket
- Server may support many simultaneous TCP sockets:
  - Each socket identified by its own 4 keys
- Web servers have different sockets for each connecting client
  - Non-persistent HTTP will have different socket for each request

Connection-Oriented Demux



UDP – User Datagram Protocol

Connectionless Transport

UDP: User Datagram Protocol [RFC 768]

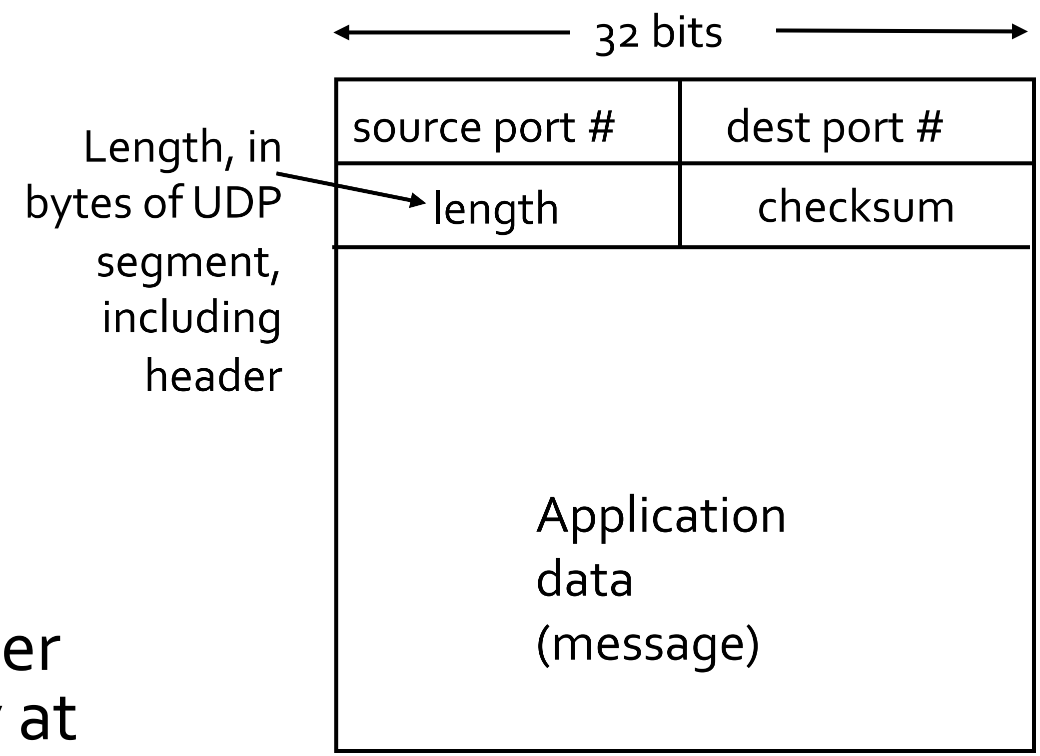
- “No frills, bare bones” Internet transport protocol
- “Best effort” service
- UDP segments may be:
  - Lost
  - Delivered out of order to app
- Connectionless
  - No handshaking between UDP sender, receiver
  - Each UDP segment handled independently of others

Why is there a UDP?

- No connection establishment (adds delay)
- Simple: no connection state at sender / receiver
- Small segment header
- No congestion control
  - UDP can blast away as fast as desired

UDP

- Often used for streaming multimedia apps
  - Loss tolerant
  - Rate sensitive
- Other UDP uses
  - DNS
  - SNMP
- Reliable transfer over UDP: add reliability at application layer
  - Application-specific error recovery!



UDP segment format

UDP Checksum

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

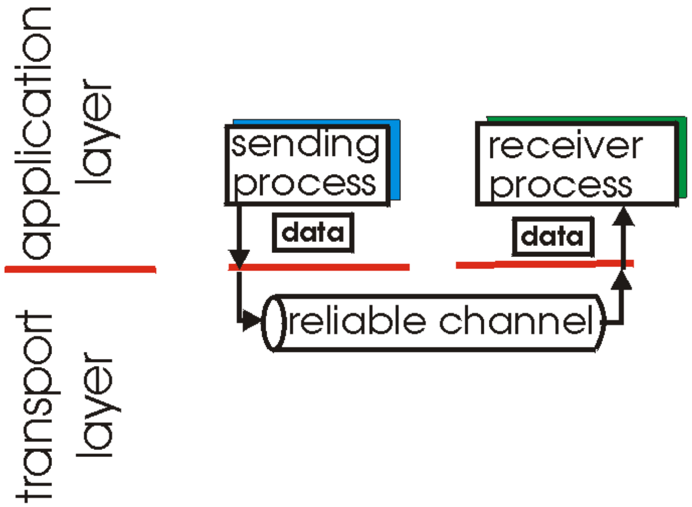
- **Sender**
  - Treat segment contents as sequence of 16-bit integers
  - Checksum: addition (1's complement sum) of segment contents
  - Sender puts checksum value into UDP checksum field
- **Receiver**
  - Compute checksum of received segment
  - Check if computed checksum equals checksum field value:
    - NO - error detected
    - YES - no error detected. But maybe errors nonetheless? *More later ….*

Reliable Data Transfer

Stepping through the design of TCP

Principles of Reliable data transfer

- Reliability is important in application, transport, and link layers

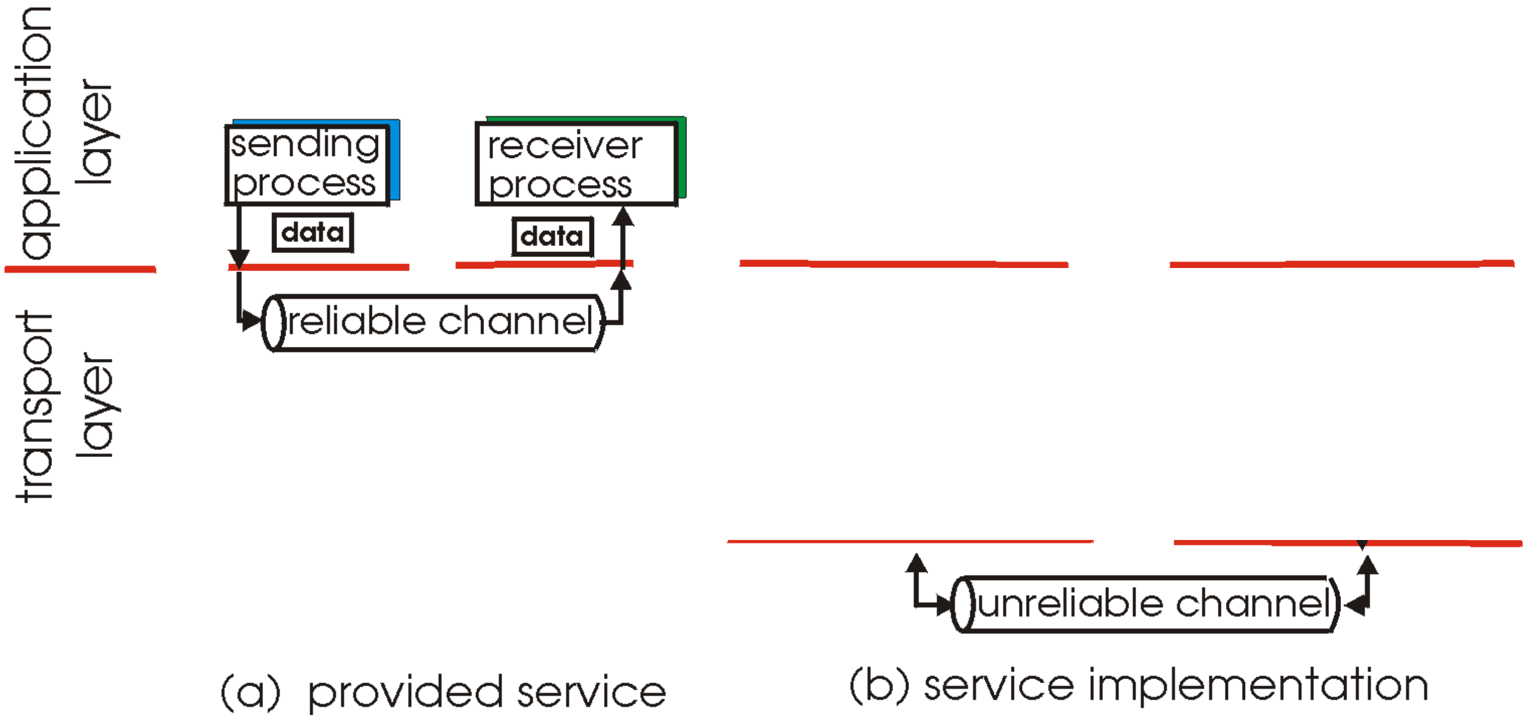


(a) provided service

- Characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Principles of Reliable data transfer

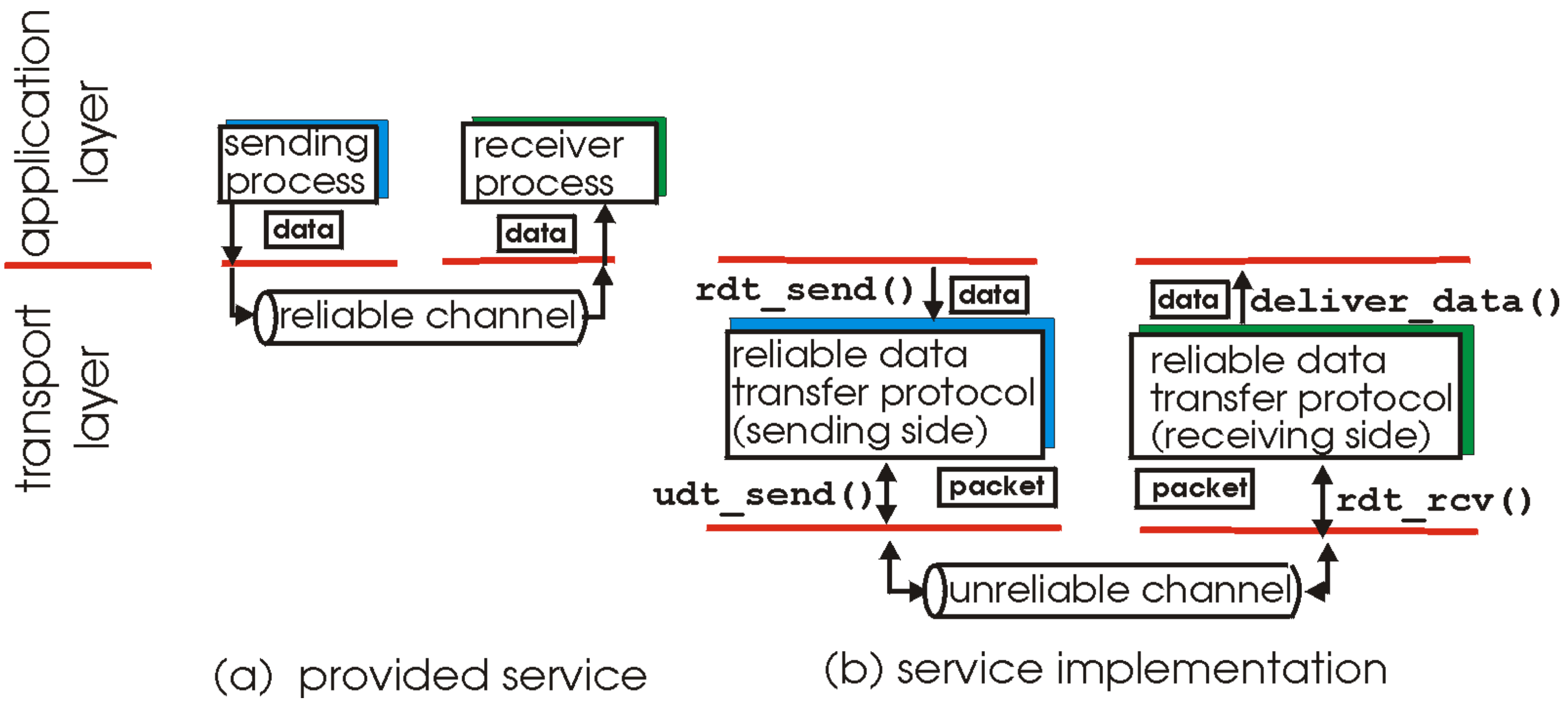
- Reliability is important in application, transport, and link layers



- Characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Principles of Reliable data transfer

- Reliability is important in application, transport, and link layers

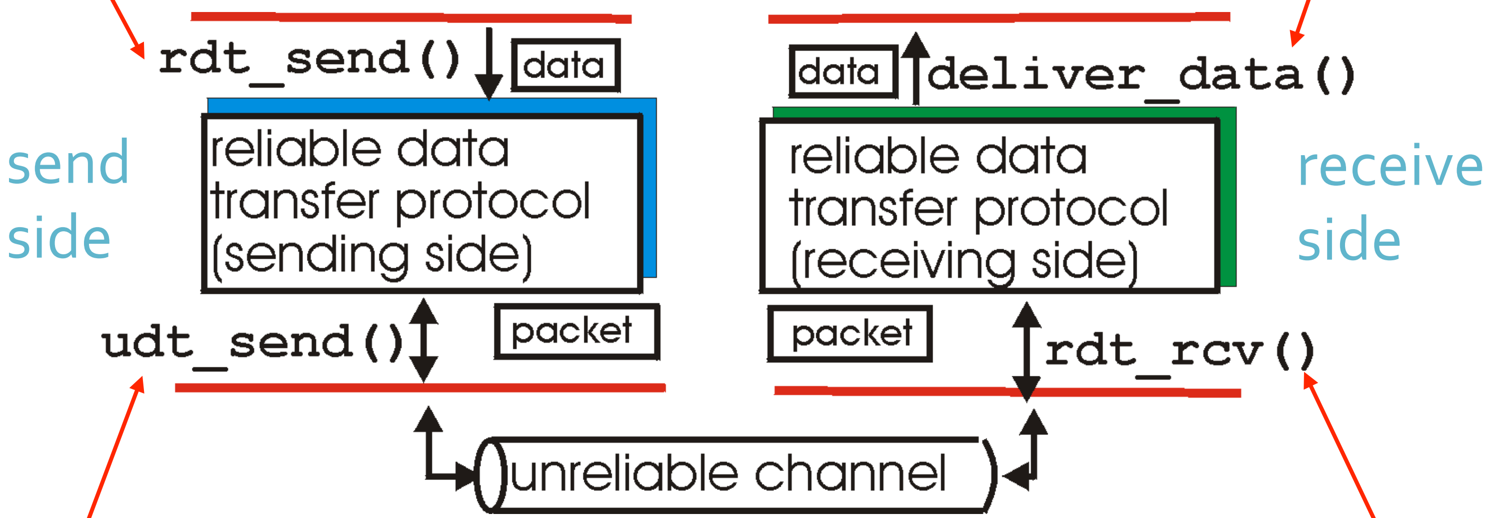


- Characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Reliable data transfer: getting started

rdt_send(): called from above, (e.g., by app.). Passed data to deliver to receiver upper layer

deliver_data(): called by **rdt** to deliver data to app.



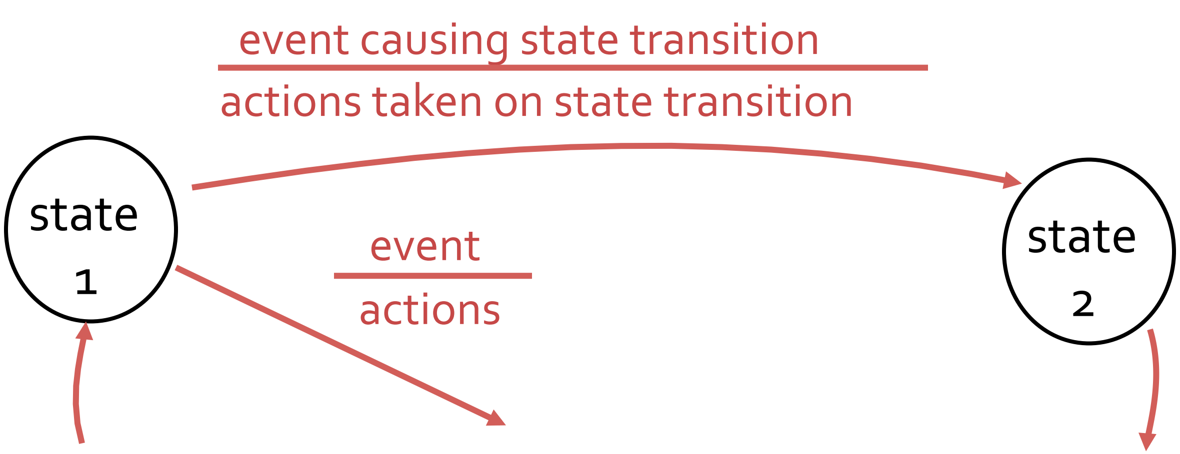
udt_send(): called by rdt, to transfer packet over unreliable channel to receiver

rdt_rcv(): called when packet arrives on rcv-side of channel

Intro to Reliable Data Transfer

- The plan: **Incrementally** develop sender / receiver sides of **reliable data transfer protocol (rdt), a fictional protocol**
  - **TCP is similar to RDT but too complex to describe all at once**
- Consider only unidirectional data transfer
  - but control info will flow on both directions!
- Use **finite state machines (FSM)** to specify sender, receiver

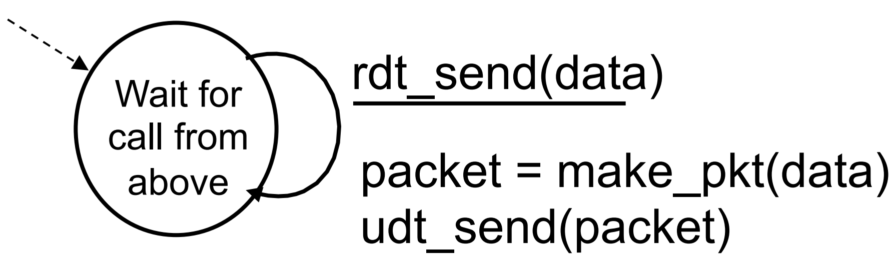
State: When in this "state", next state uniquely determined by next event



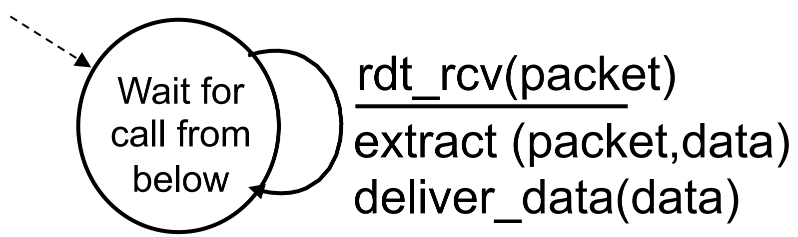
Λ (uppercase Lambda = empty set)

rdt1.0: Reliable Transfer Over a Reliable Channel

- Underlying channel perfectly reliable
  - No bit errors
  - No loss of packets
- Separate FSMs for sender, receiver:
  - Sender sends data into underlying channel
  - Receiver reads data from underlying channel



sender

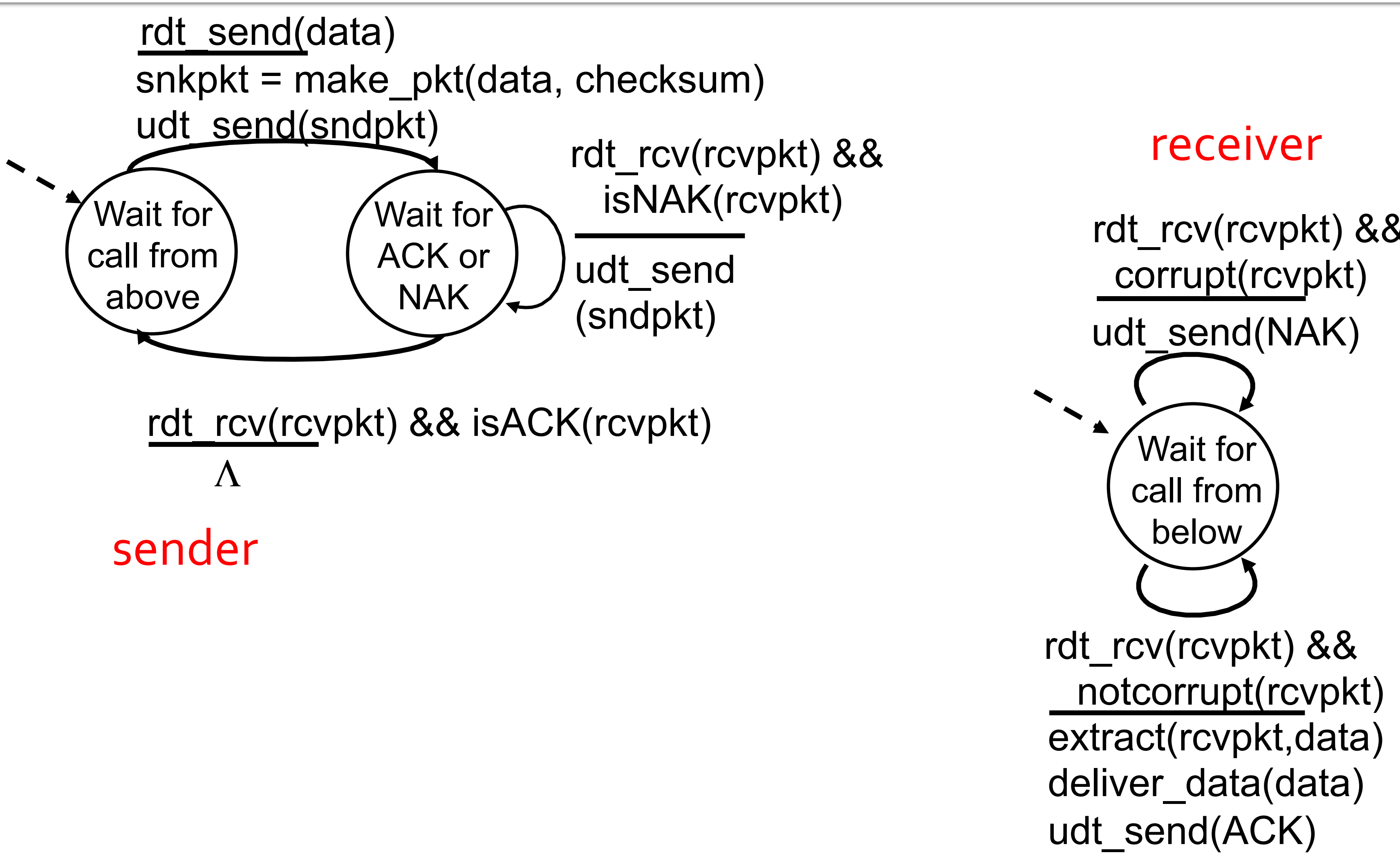


receiver

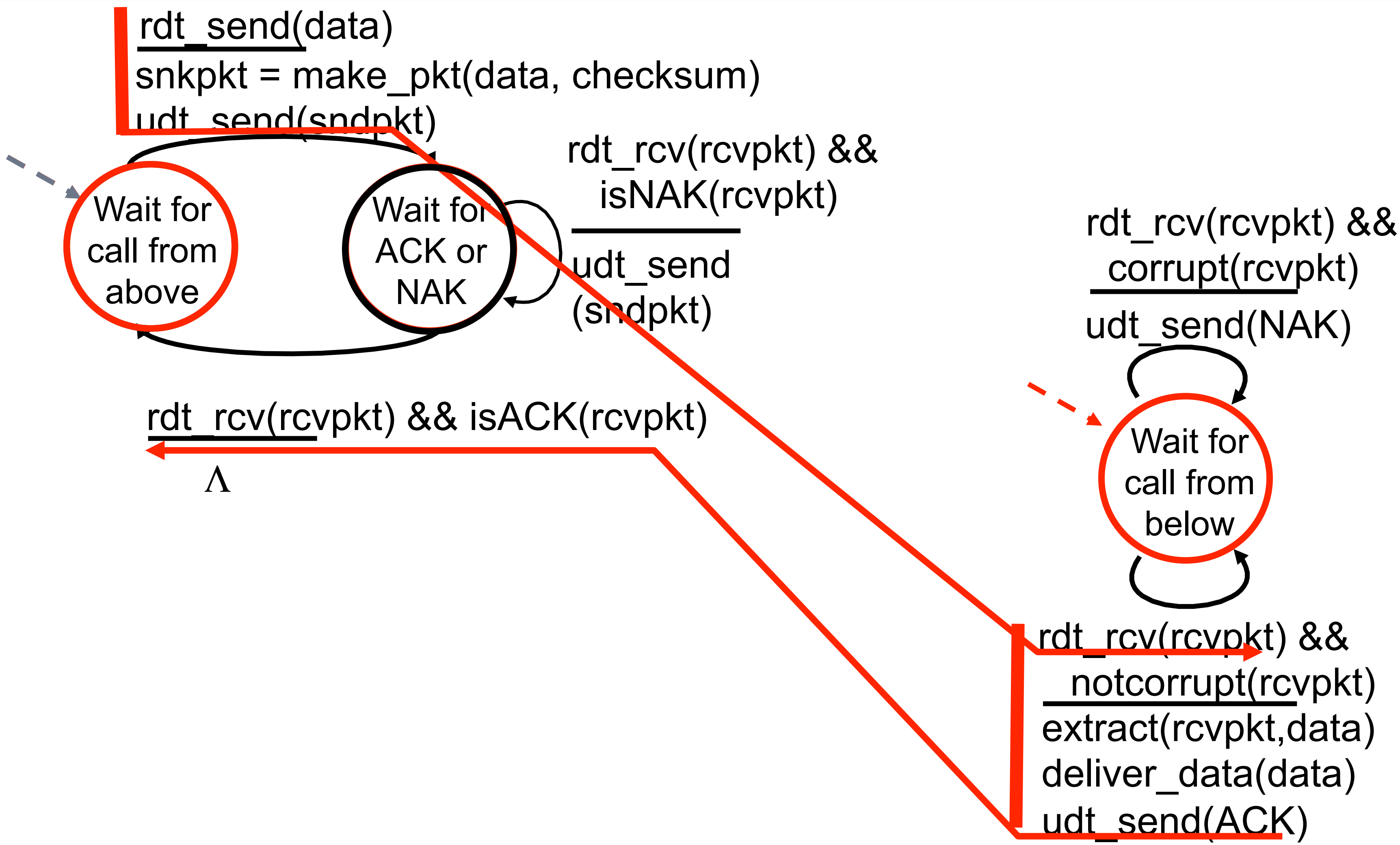
rdt2.0: Channel with Bit Errors

- Underlying channel may **flip bits** in packet
  - Checksum to detect bit errors
- But, how do we recover from errors?
  - **Acknowledgements (ACKs)**: receiver explicitly tells sender that packet received OK
  - **Negative acknowledgements (NAKs):** receiver explicitly tells sender that packet had errors
  - Sender retransmits packet on receipt of NAK
- New mechanisms in `rdt2.0` (beyond `rdt1.0`):
  - Error detection
  - Receiver feedback
    - Control msgs (ACK,NAK) go from receiver to sender

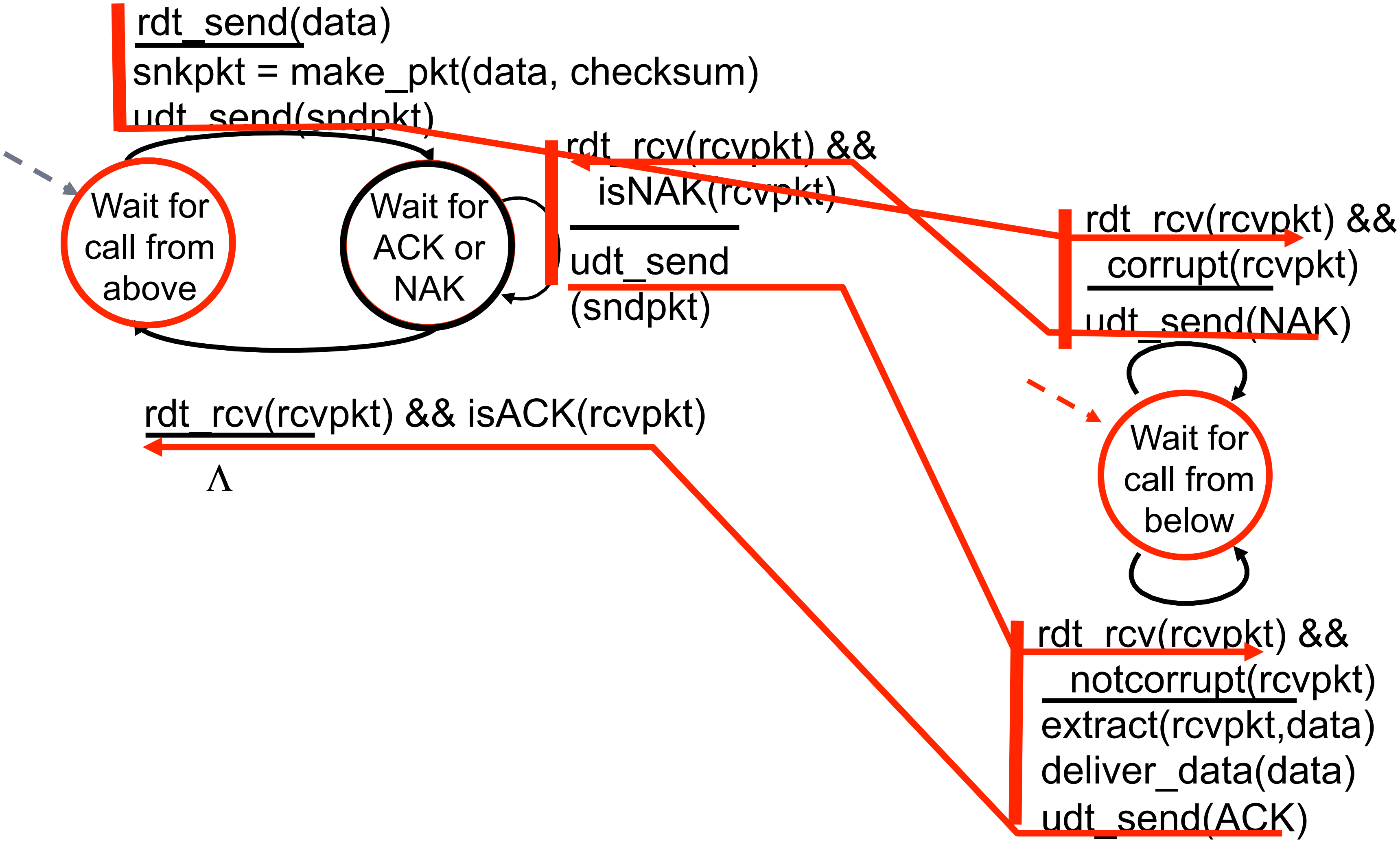
rdt2.0: FSM specification



rdt2.0: Operation with No Errors



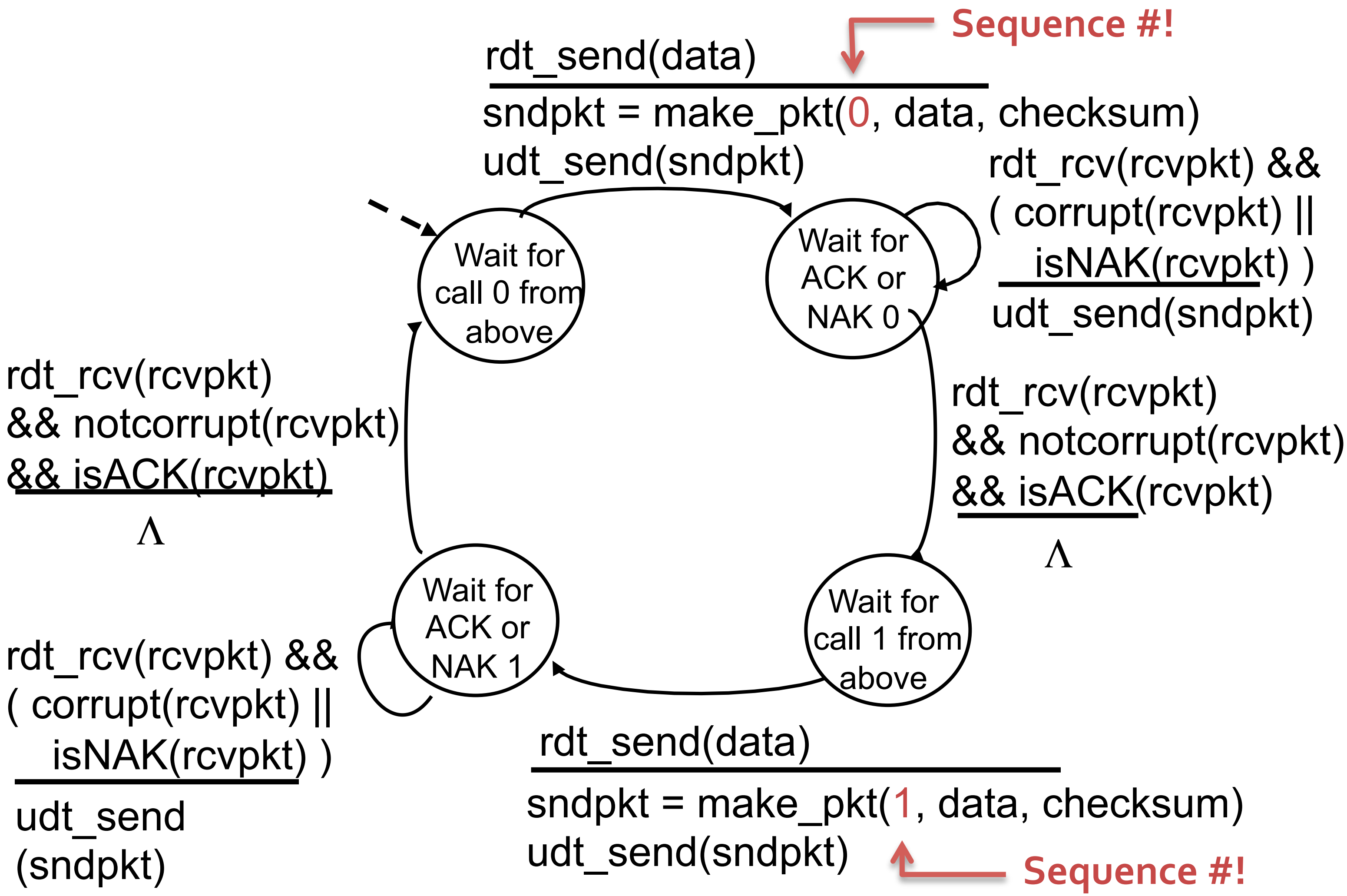
rdt2.0: Error Scenario



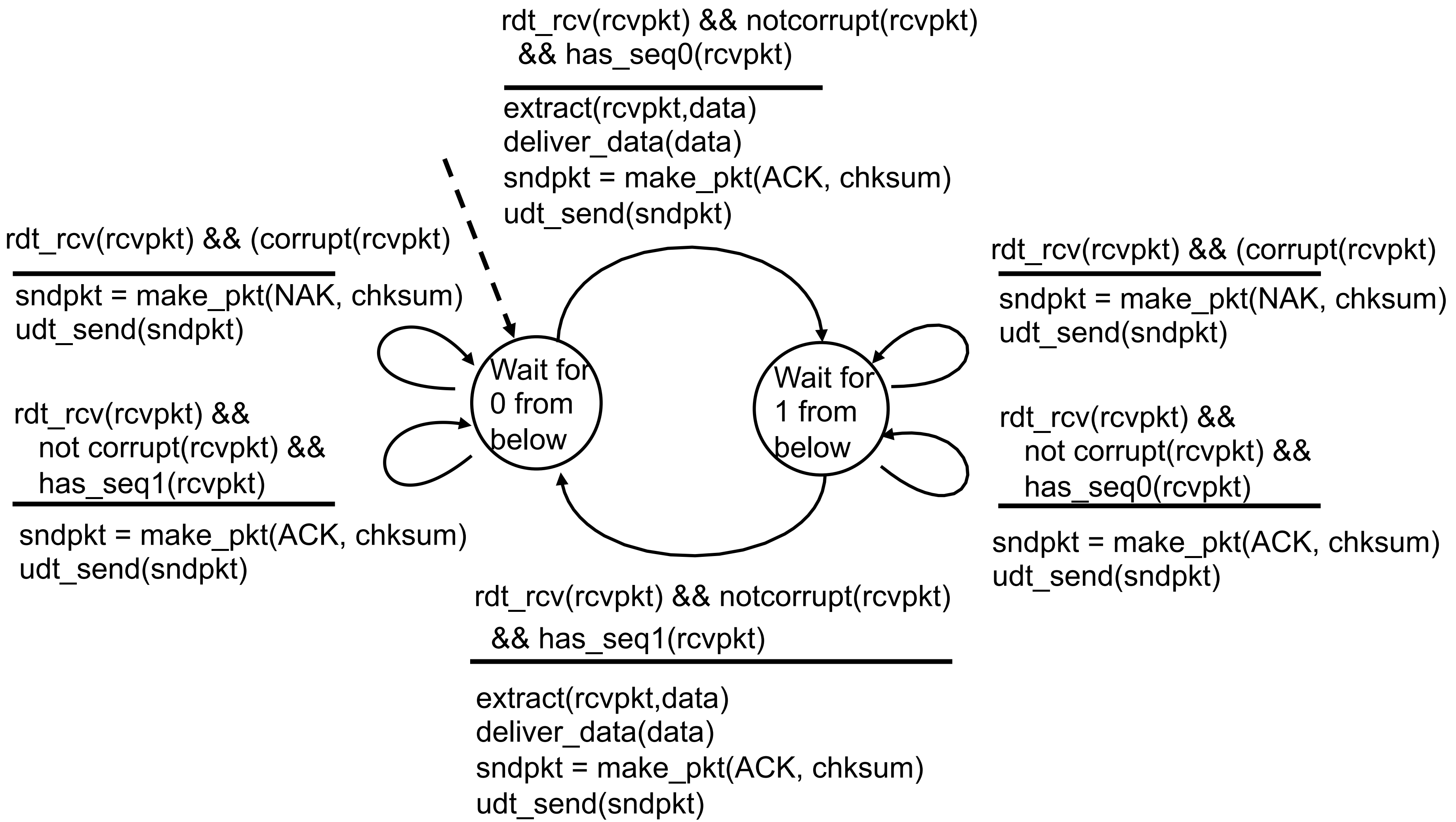
rdt2.0 has a Fatal Flaw!

- What happens if **ACK/NAK is corrupted**?
  - Sender doesn't know what happened at receiver!
- Can't just retransmit
  - Receiver might get duplicate data
- Handling duplicates:
  - Sender retransmits current packet if ACK/NAK garbled
  - Sender adds **sequence number** to each packet
  - Receiver discards (doesn't deliver) duplicate packet
- **Stop and wait design**
  - Sender sends 1 packet, then waits for receiver response

rdt2.1: Sender – Handles Garbled ACK/NAKs



rdt2.1: Receiver – Handles Garbled ACK/NAKs



rdt2.1: Discussion

Sender:

- Seq # added to pkt
- Two seq. #'s (0,1) will suffice. **Why?**
- Must check if received ACK/NAK corrupted
- Twice as many states
  - State must "remember" whether "current" packet has sequence number of 0 or 1

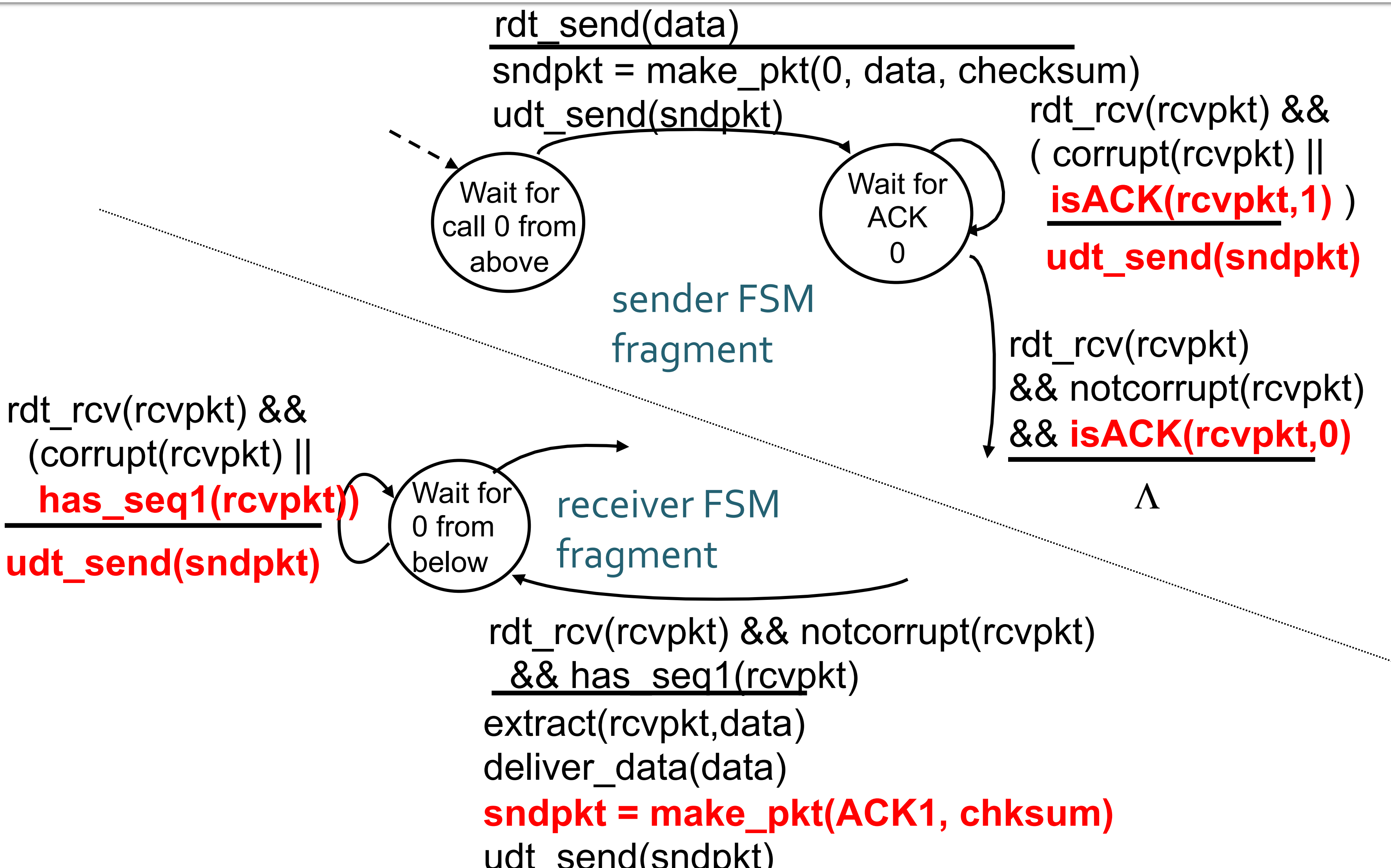
Receiver:

- Must check if received packet is duplicate
  - State indicates whether 0 or 1 is expected packet sequence number
- Receiver can *not* know if its last ACK/NAK received OK at sender
  - *Packet corruption can affect ACK/NAK packets…*

rdt2.2: a NAK-free protocol

- Same *functionality* as rdt2.1
- No NAKs!
  - Receiver instead sends ACK for last packet received OK
  - Receiver must explicitly include seq # of packet being ACKed
- Duplicate ACK at sender results in same action as NAK
  - Retransmit current packet

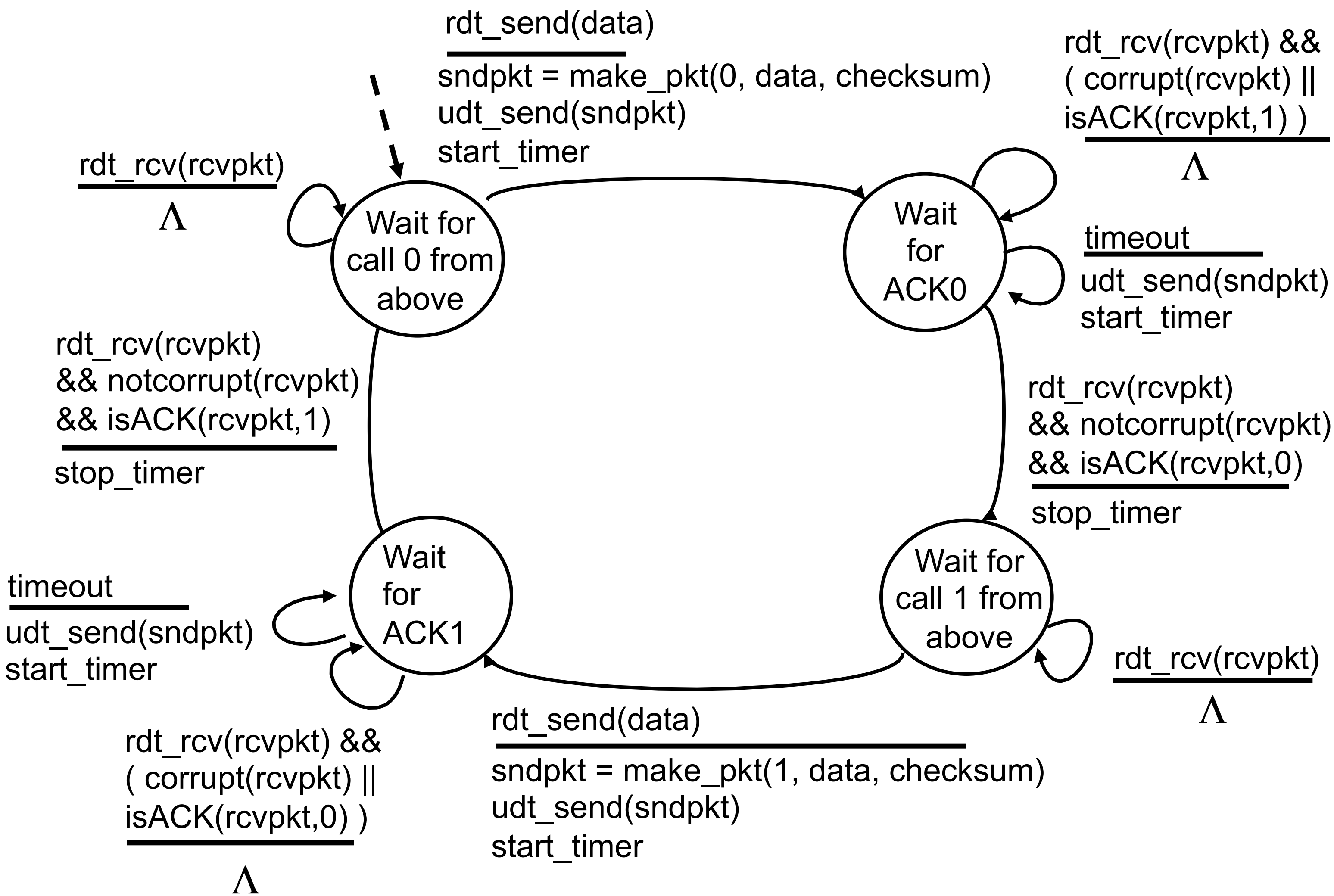
rdt2.2: Partial Sender and Receiver



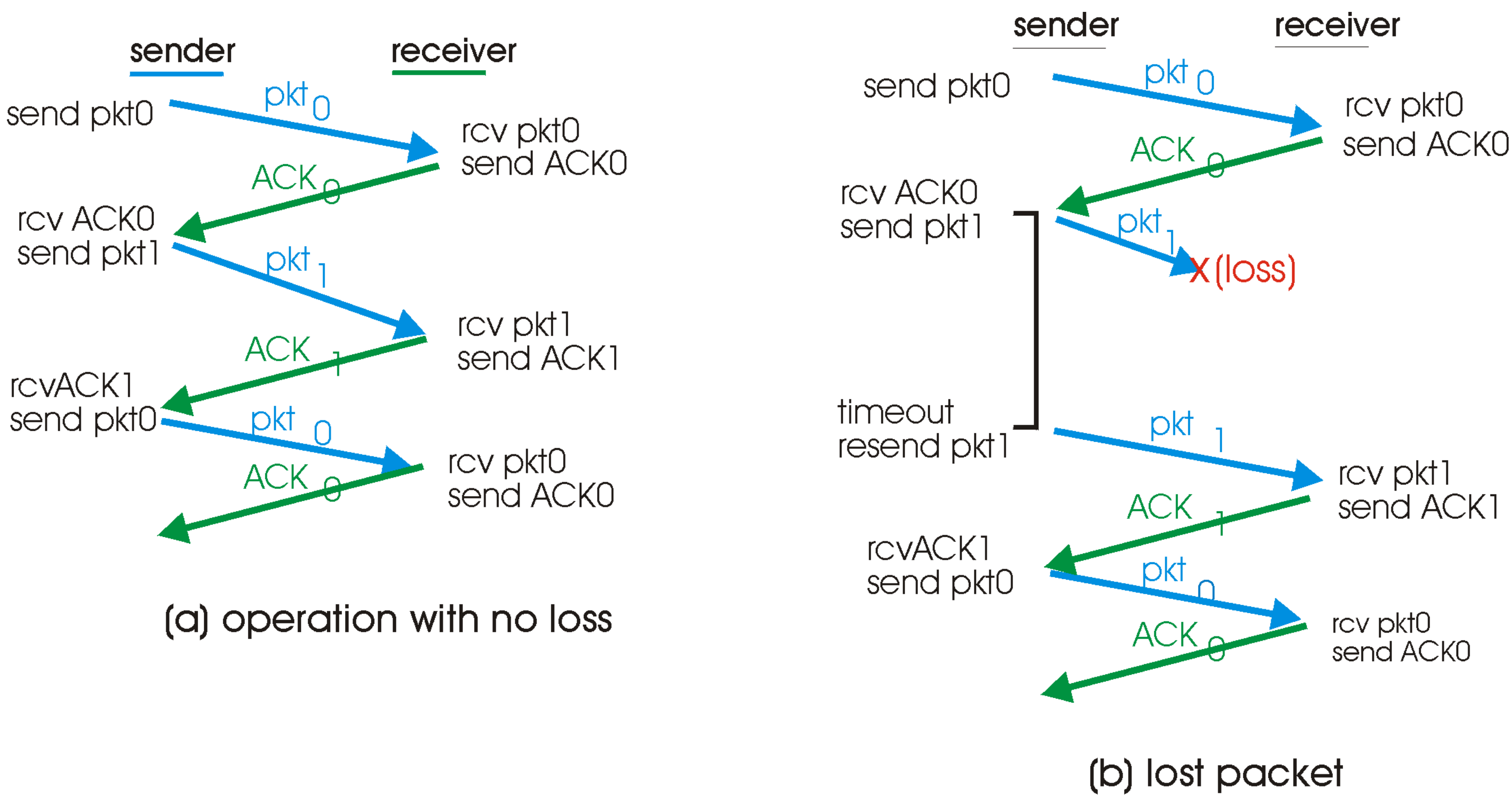
rdt3.0: Channels with Errors and Loss

- New assumption
  - Underlying channel can also **lose packets** (data **or** ACKs)
  - Checksum, seq. #, ACKs, and retransmissions will help but are not sufficient
- New approach
  - Sender waits "reasonable" amount of time for ACK
  - Retransmits if no ACK received in this time
  - If pkt (or ACK) is just delayed but not lost:
    - Retransmission will be duplicate, but seq. #'s solves this problem
    - Receiver must specify seq # of pkt being ACKed
  - Requires countdown timer

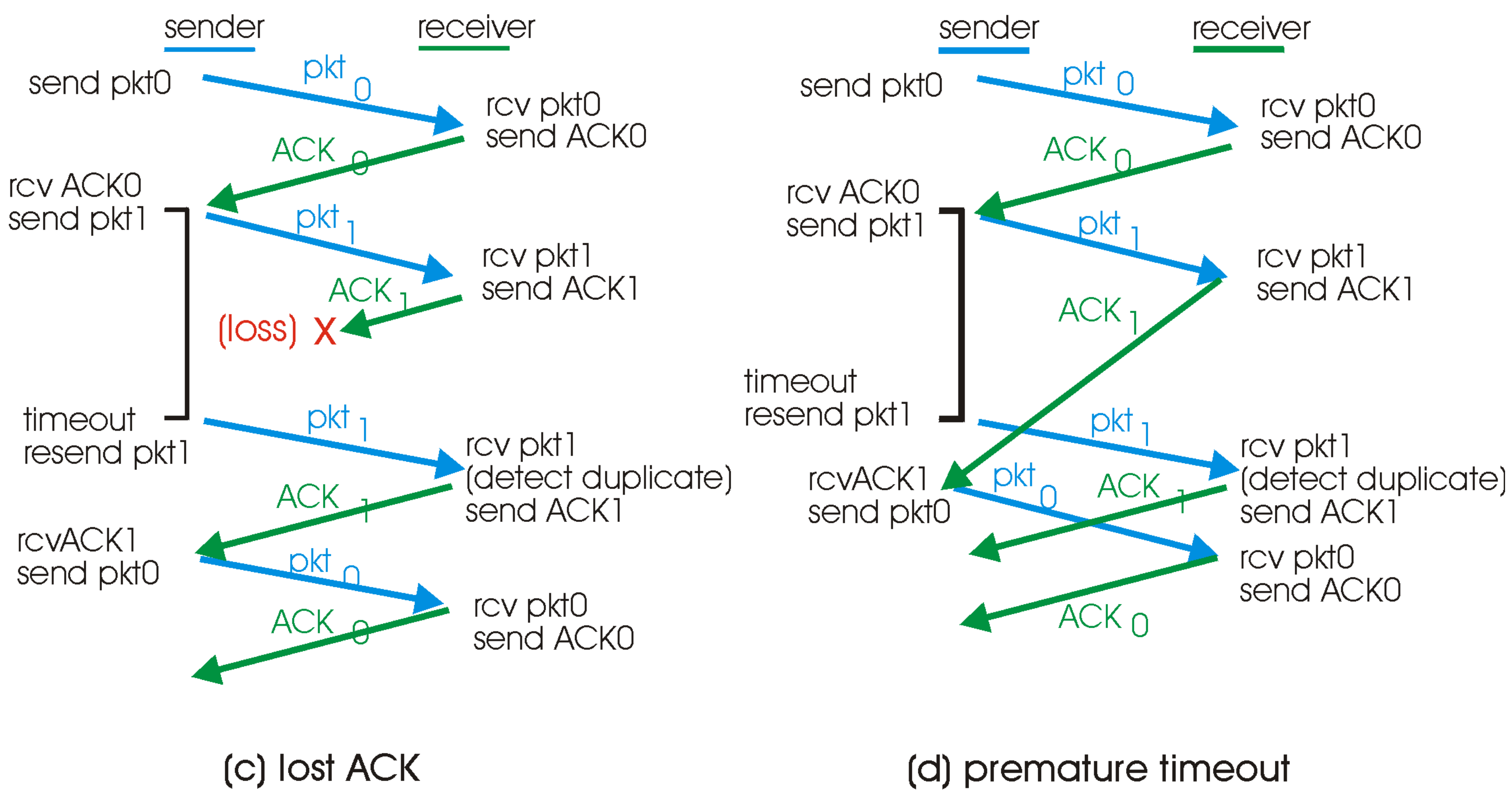
rdt3.0 Sender



rdt3.0 in Action



rdt3.0 in Action



Performance of rdt3.0

- rdt3.0 works, but **performance stinks**
- For 1 Gbps link, 15 ms prop. delay, 8000 bit packet:

$$d_{trans} = \frac{L}{R} = \frac{8000\text{bits}}{10^9\text{bps}} = 8\,\text{microseconds}$$

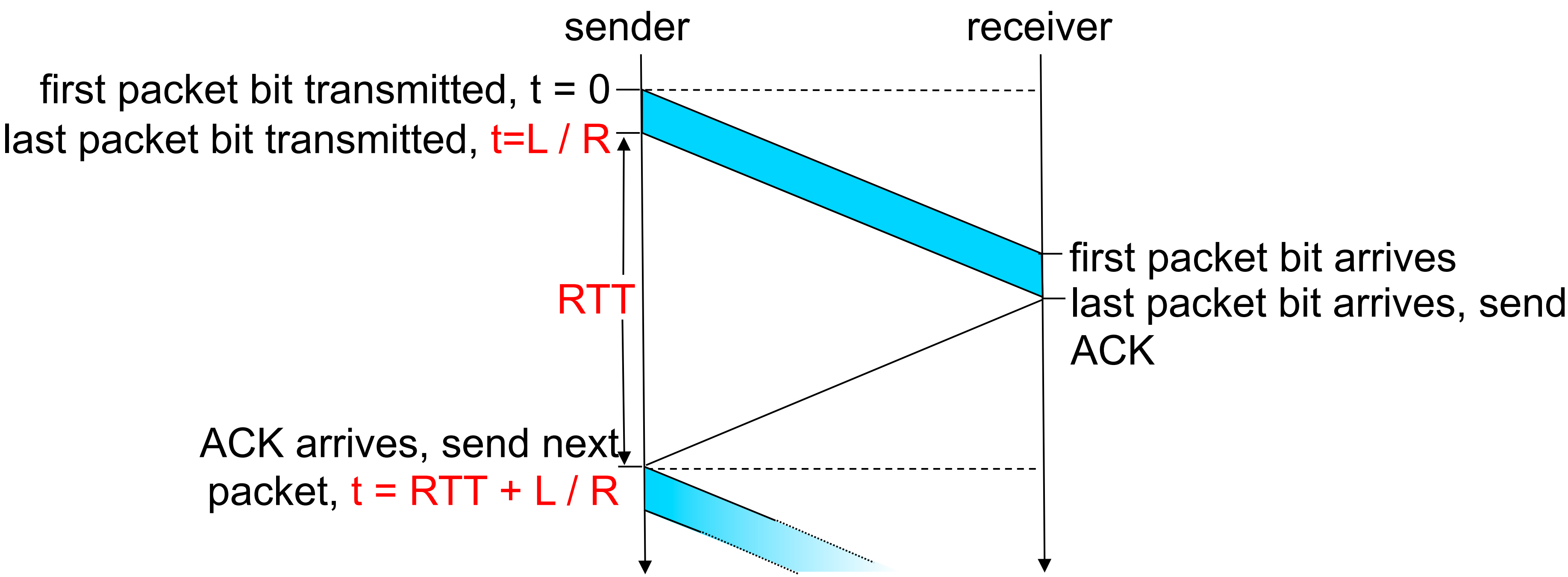
How long it takes to push packet out onto wire

- U_{sender}: utilization : fraction of time sender busy sending

$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

- 1KB packet every 30 msec
  - **33kB/sec throughput over 1 Gbps link**
  - **Network protocol limits use of physical resources!**

rdt3.0: Stop-and-Wait Operation

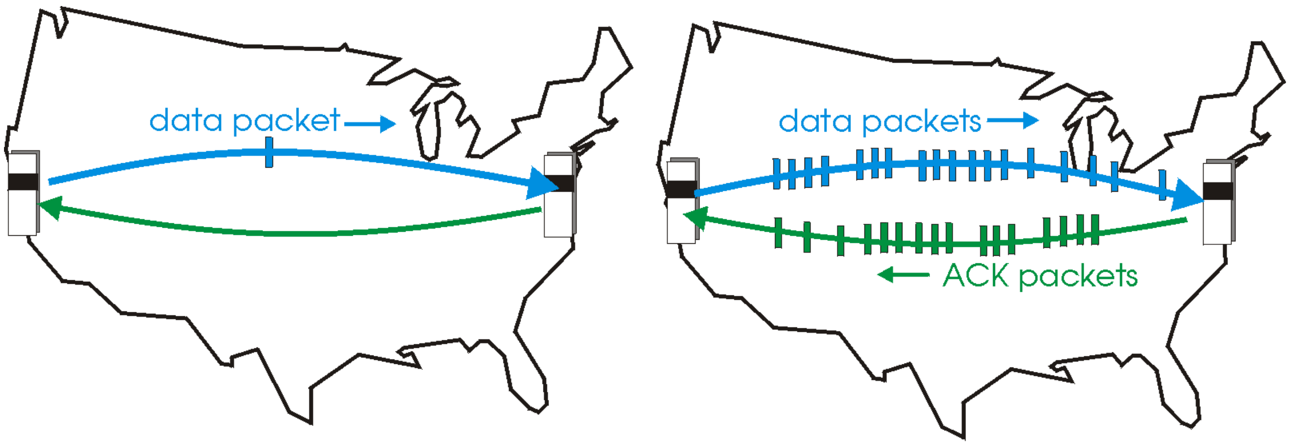


$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

Pipelined protocols

Pipelining: sender allows multiple, "in-flight", yet-to-be-acknowledged packets

- Range of sequence numbers must be increased
- Buffering at sender and/or receiver

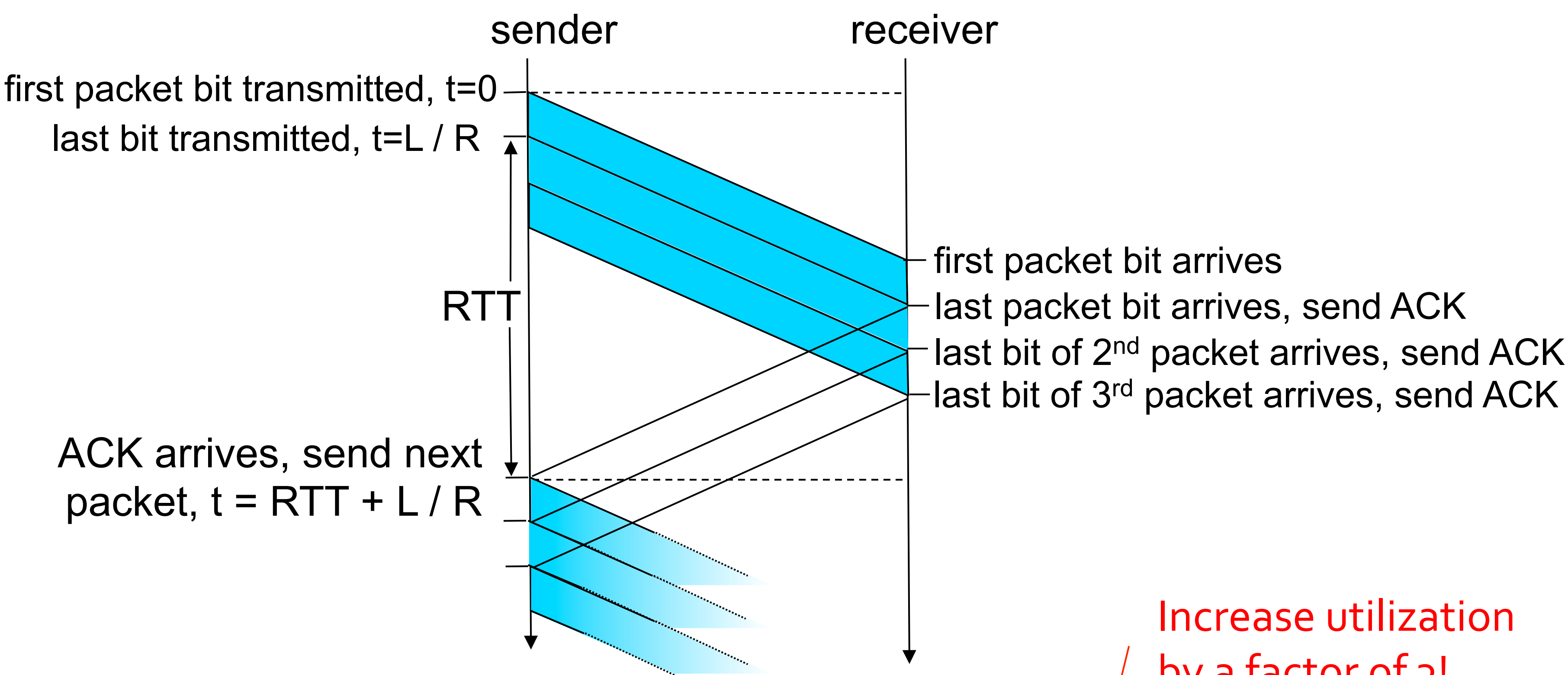


(a) a stop-and-wait protocol in operation

(b) a pipelined protocol in operation

- Two generic forms of pipelined protocols: *go-Back-N and selective repeat*

Pipelining: Increased Utilization



Increase utilization by a factor of 3!

$$U_{sender} = \frac{3 * L / R}{RTT + L / R} = \frac{.024}{30.008} = 0.0008$$

Pipelining Protocols

GO-BACK-N

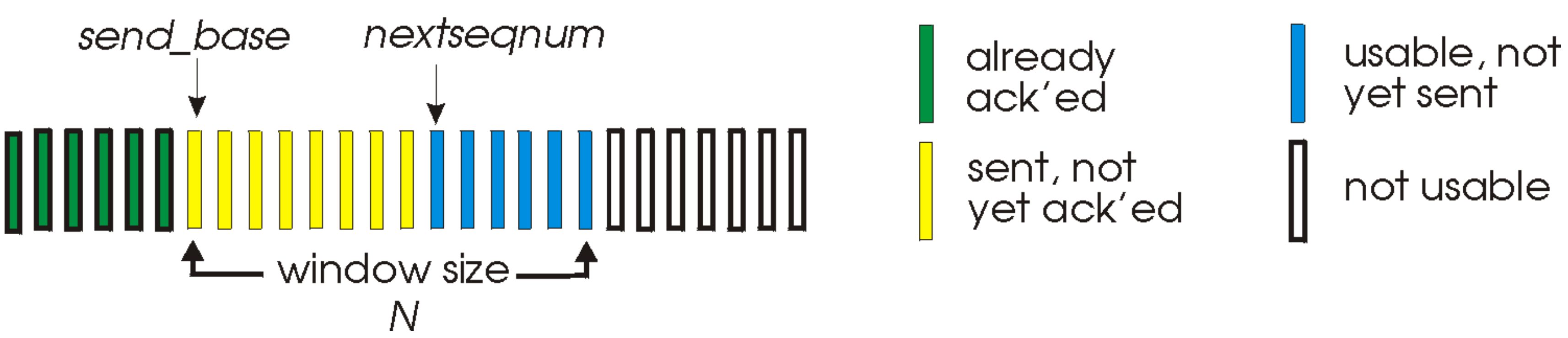
- Sender
  - Up to N unACKed pkts in pipeline
- Receiver
  - Only sends cumulative ACKs
  - Doesn't ACK packet if there's a gap
- Sender
  - Has timer for oldest unACKed packet
  - If timer expires: retransmit all unACKed packets

SELECTIVE REPEAT

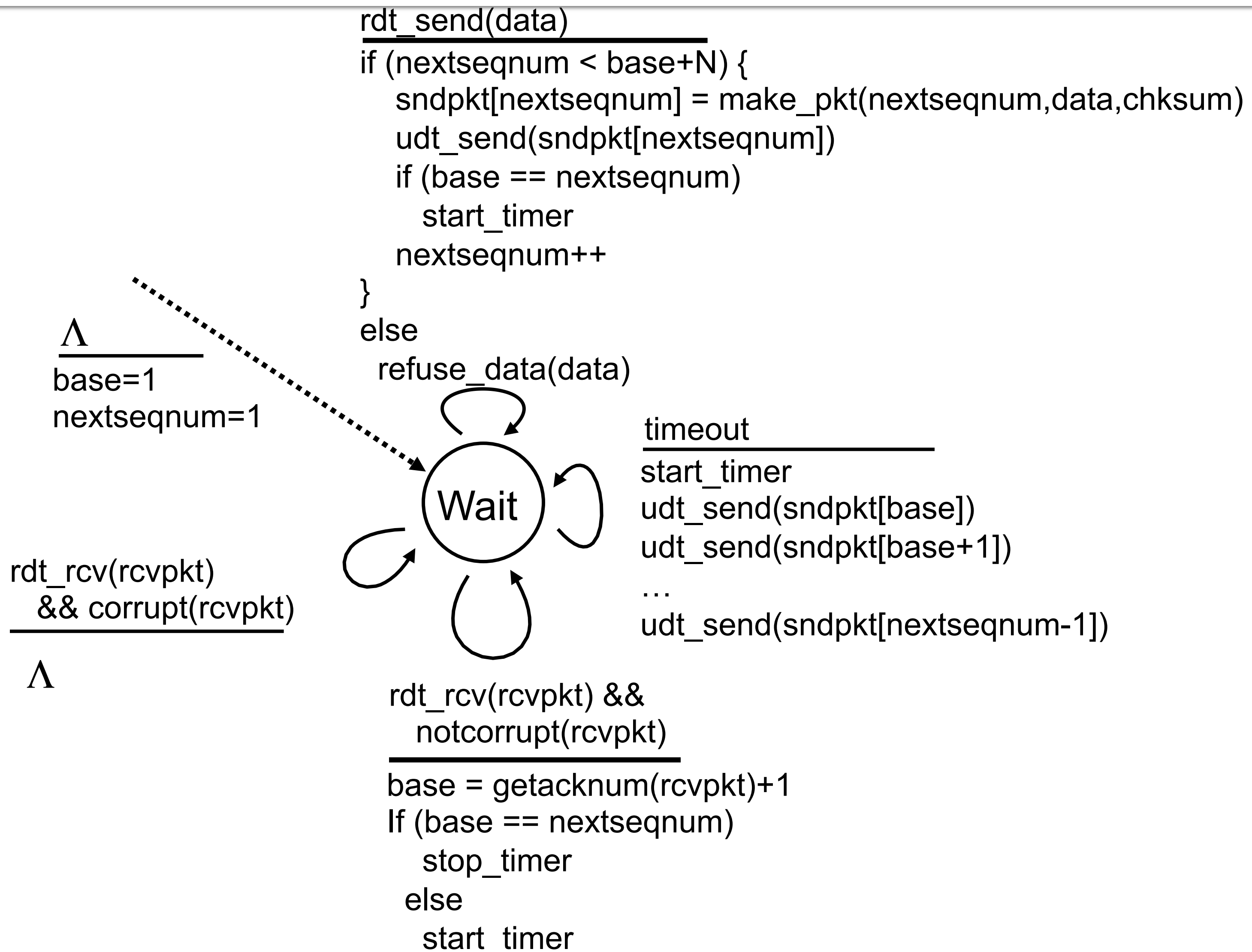
- Sender
  - Up to N unACKed packets in pipeline
- Receiver
  - ACKs individual pkts
- Sender
  - Maintains timer for each unACKed pkt
  - If timer expires: retransmit only unACKed packet

Go-Back-N : Sender

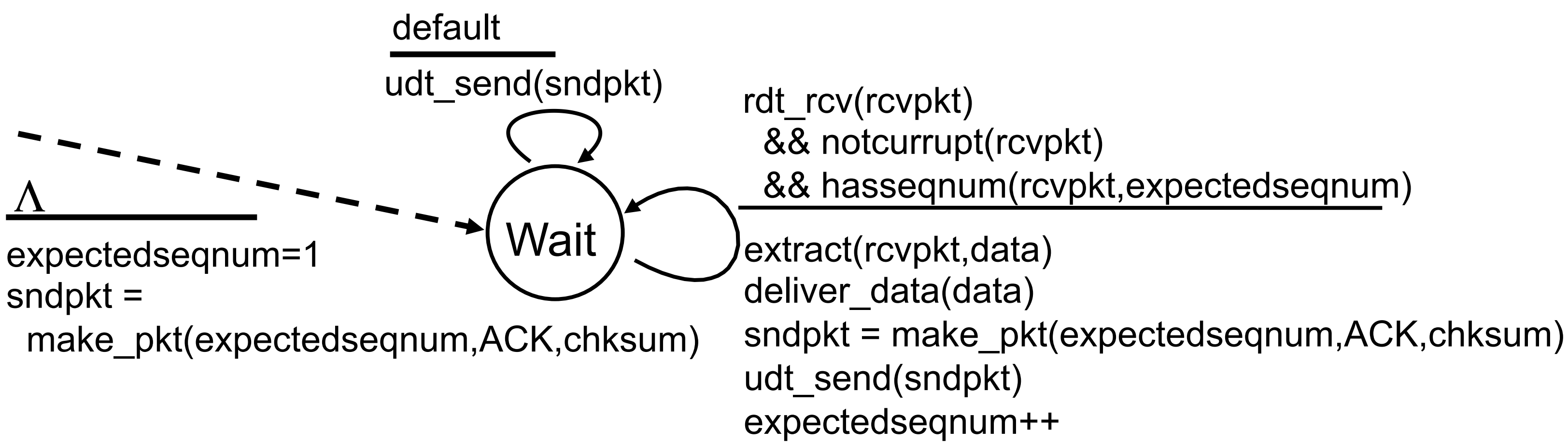
- k-bit sequence # in packet header
- "Window" of up to N, consecutive unACKed pkts allowed
- ACK(n): ACKs all pkts up to, including seq # n
  - Referred to as a "cumulative ACK"
  - May receive duplicate ACKs *(see receiver)*
- Timer for oldest in-flight packet
- timeout(n): retransmit packet n **and all higher** seq # packets in window



GBN: Sender Extended FSM

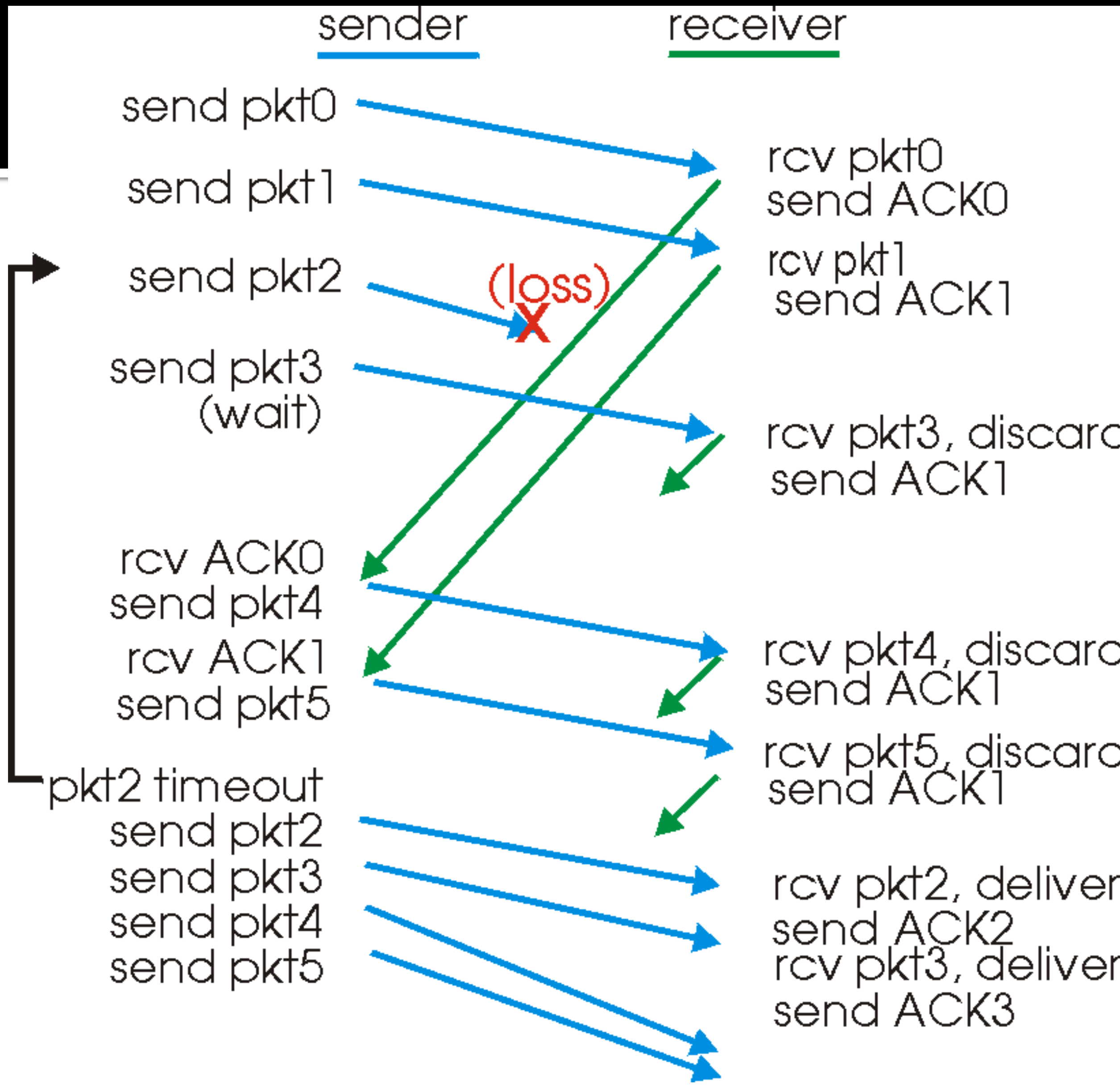


GBN: Receiver Extended FSM



- ACK-only: always send ACK for correctly-received pkt with highest ***in-order*** seq #
  - May generate duplicate ACKs
  - Need only remember **`expectedseqnum`**
- Out-of-order pkt:
  - Discard (don't buffer) -> no receiver buffering! (reduces complexity)
  - Re-ACK pkt with highest in-order seq #

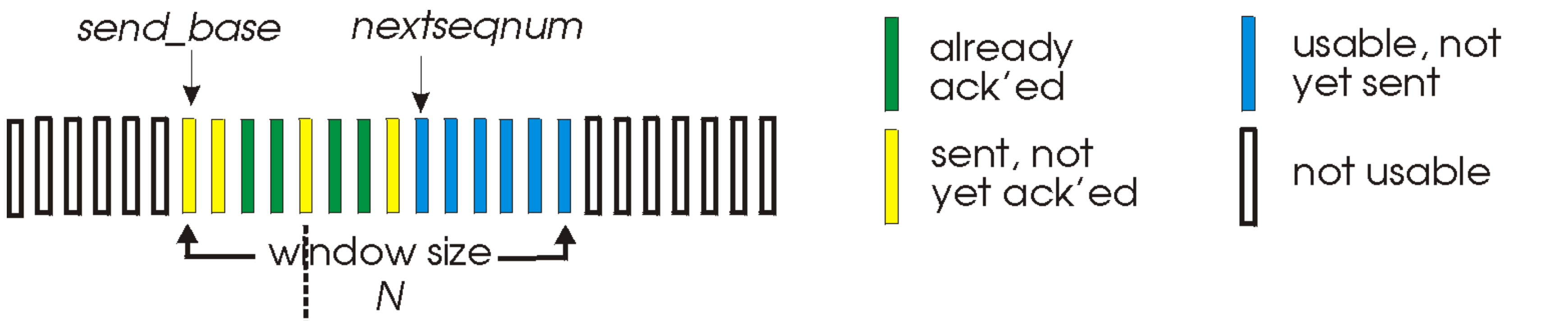
GBN in action



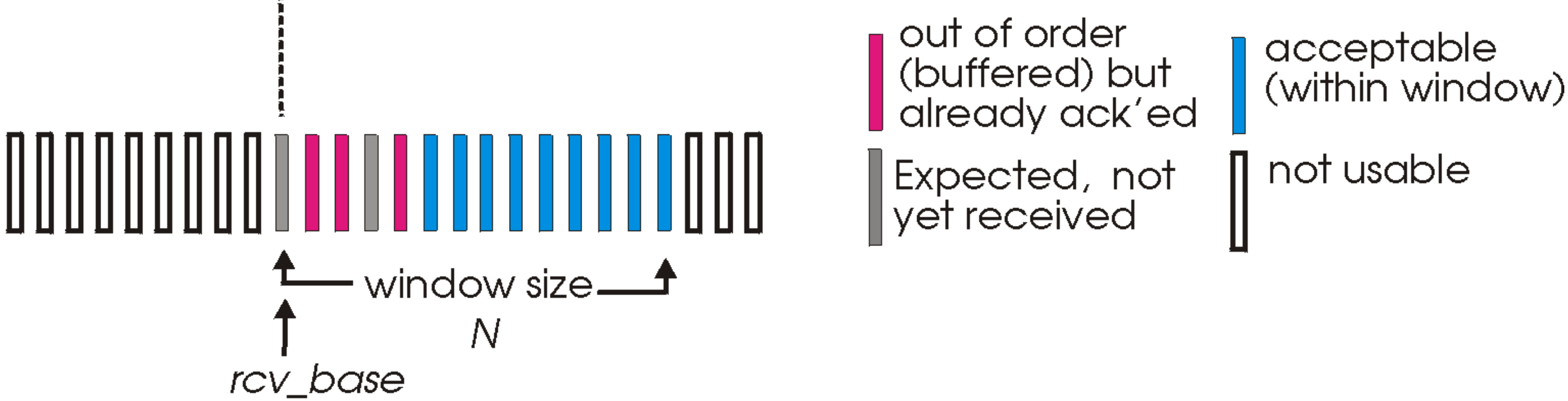
Selective Repeat

- Receiver *individually* acknowledges all correctly received packets
  - Buffers packets, as needed, for eventual in-order delivery to upper layer
- Sender only resends packets for which ACK was not received
  - Sender timer for each unACKed packets
- Sender window
  - N consecutive seq #'s
  - Again limits seq #s of sent, unACKed packets

Selective Repeat: Sender and Receiver windows



(a) sender view of sequence numbers



(b) receiver view of sequence numbers

Selective Repeat

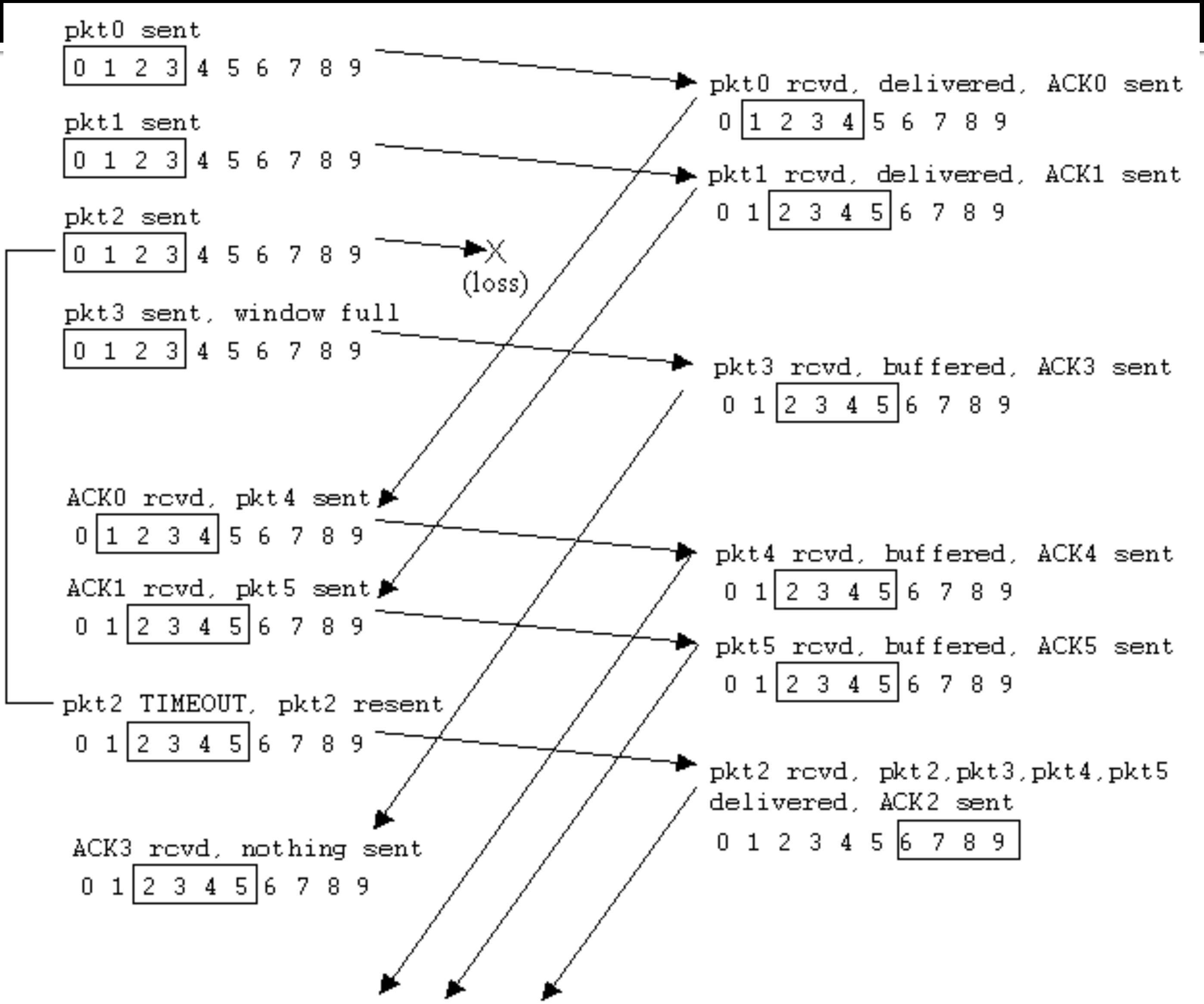
SENDER

- Data from above :
  - If next available seq # in window, send pkt
- timeout(n):
  - Resend pkt n, restart timer
- ACK(n) in [sendbase,sendbase+N]:
  - Mark pkt n as received
  - If n is smallest unACKed pkt, advance window base to next unACKed seq #

RECEIVER

- pkt n in [rcvbase, rcvbase+N-1]
  - Send ACK(n)
  - Out-of-order: buffer
  - In-order: deliver (also deliver buffered, in-order pkts), advance window to next not-yet-received pkt
- pkt n in [rcvbase-N,rcvbase-1]
  - ACK(n) *(this is a packet that was previously received…)*
- Otherwise:
  - Ignore

Selective Repeat in Action

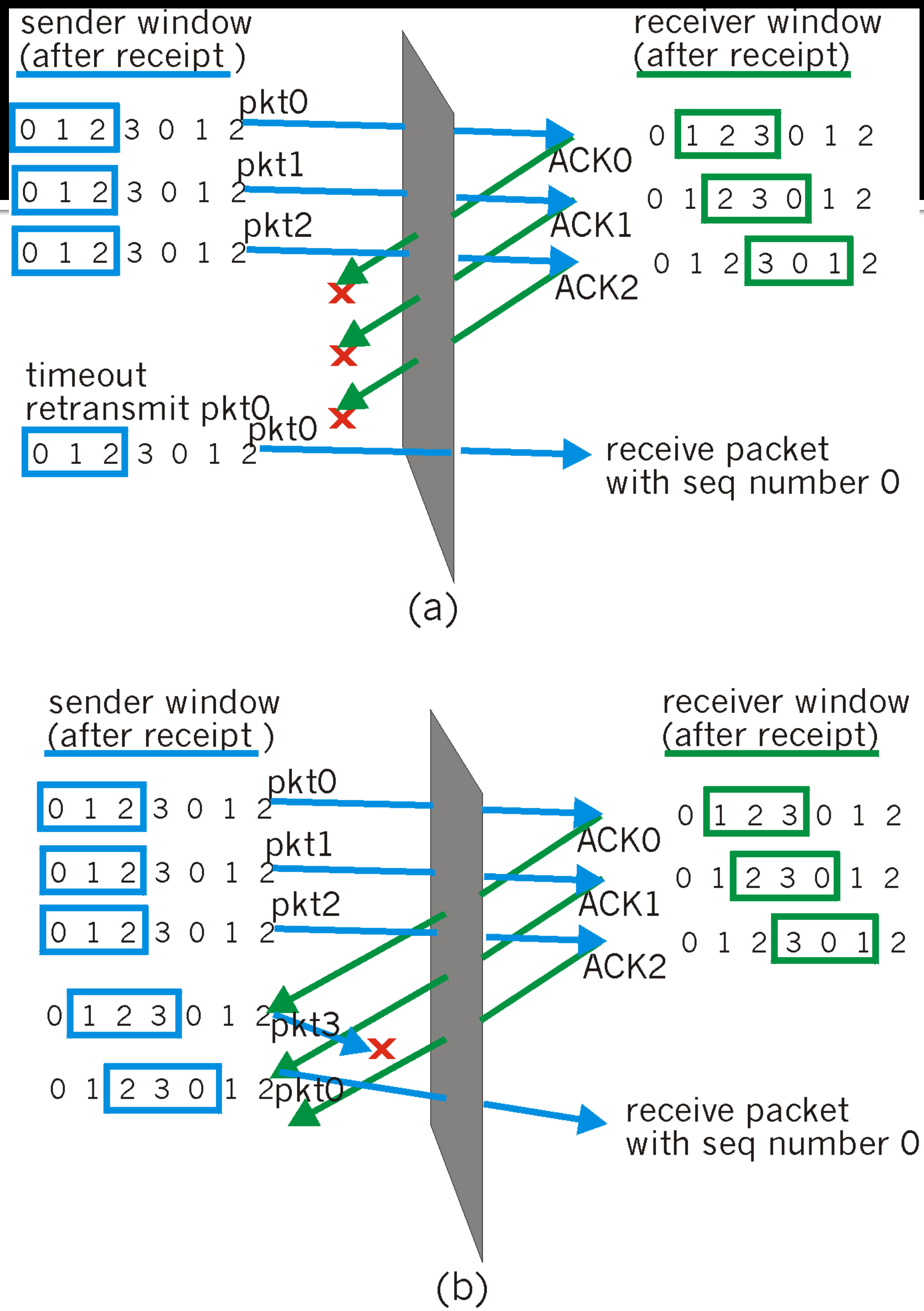


Selective Repeat: Dilemma

- Example
  - seq #'s: 0, 1, 2, 3
  - window size=3
- Receiver sees no difference in two scenarios!
  - Is it a retransmitted packet, or a new packet?
  - Incorrectly passes duplicate data as new in (a)

Q: What relationship is needed between seq # size and window size?

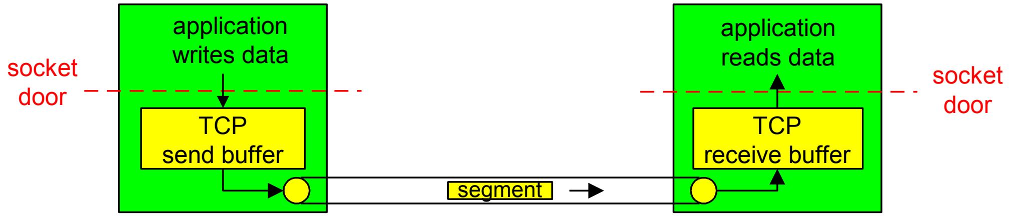
A: Twice as many seq #'s



TCP – Transmission Control Protocol

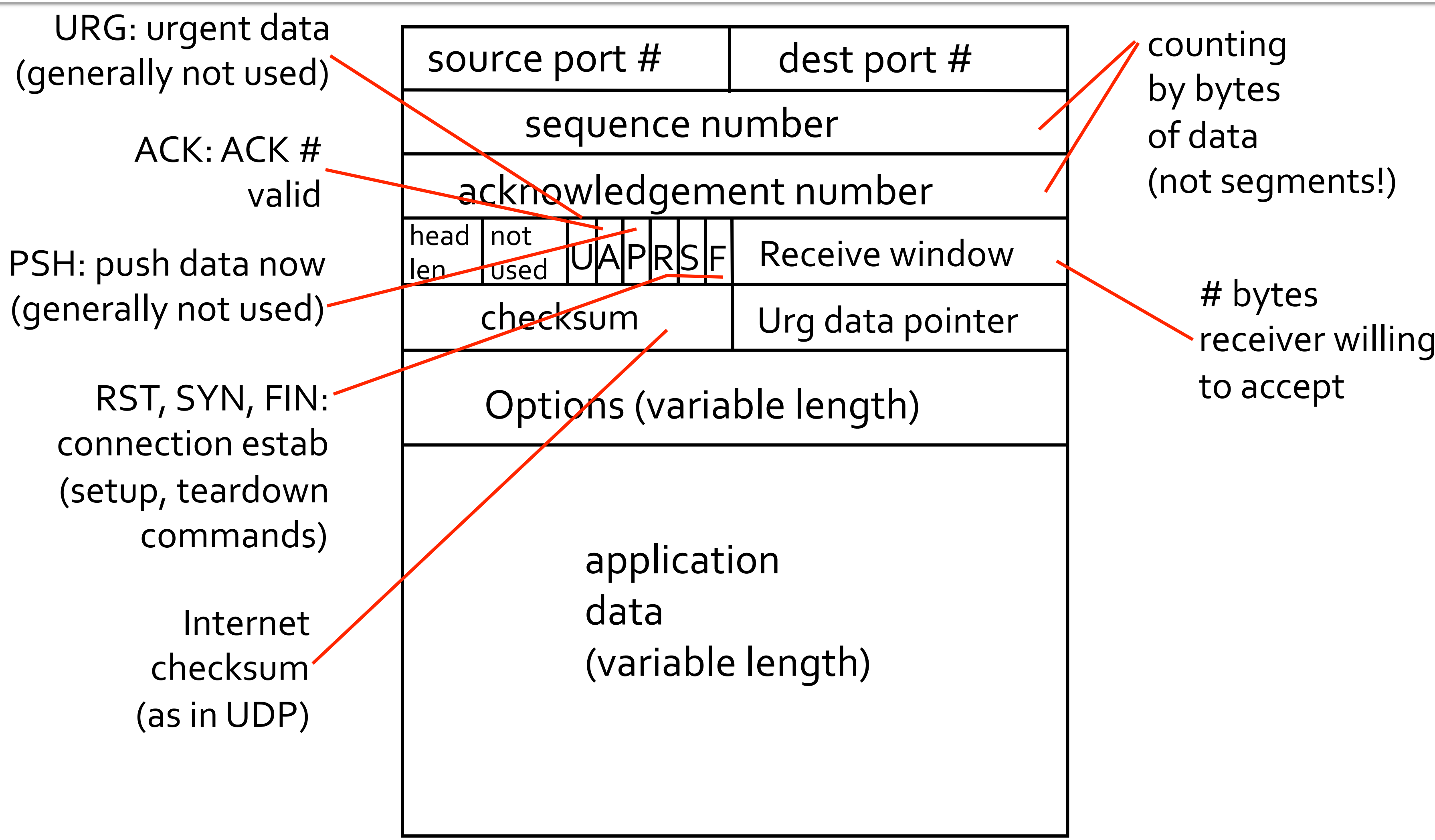
TCP: Overview RFCs: 793, 1122, 1323, 2018, 2581

- Point-to-point:
  - One sender, one receiver
- Reliable, in-order *byte steam:*
  - No "message boundaries"
- Pipelined:
  - TCP congestion and flow control set window size
- *Send & receive buffers*



- Full duplex data:
  - Bi-directional data flow in same connection
  - MSS: maximum segment size
- Connection-oriented:
  - Handshaking (exchange of control msgs) initializes sender, receiver state before data exchange
- Flow controlled:
  - Sender will not overwhelm receiver

TCP segment structure



TCP seq. #'s and ACKs

Seq. #'s:

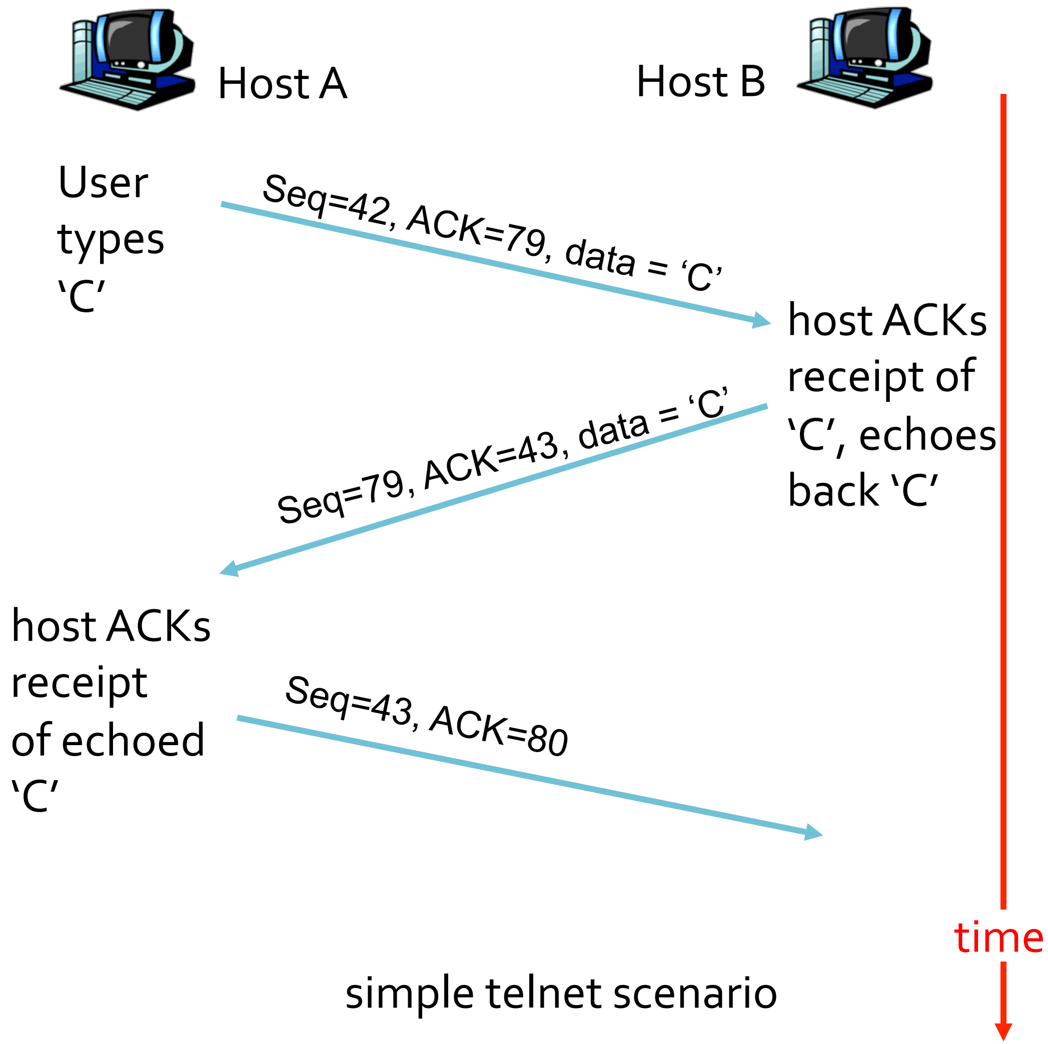
- Byte stream "number" of first byte in segment's data

ACKs:

- Seq # of next byte expected from other side
- Cumulative ACK

How does receiver handle out-of-order segments?

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



simple telnet scenario

TCP Round Trip Time and Timeout

- **How to set TCP timeout value?**
- Should be longer than RTT (round-trip-time)
  - But RTT varies…
- If it is too short
  - Premature timeout
  - Unnecessary retransmissions…
- If it is too long
  - Slow reaction to segment loss

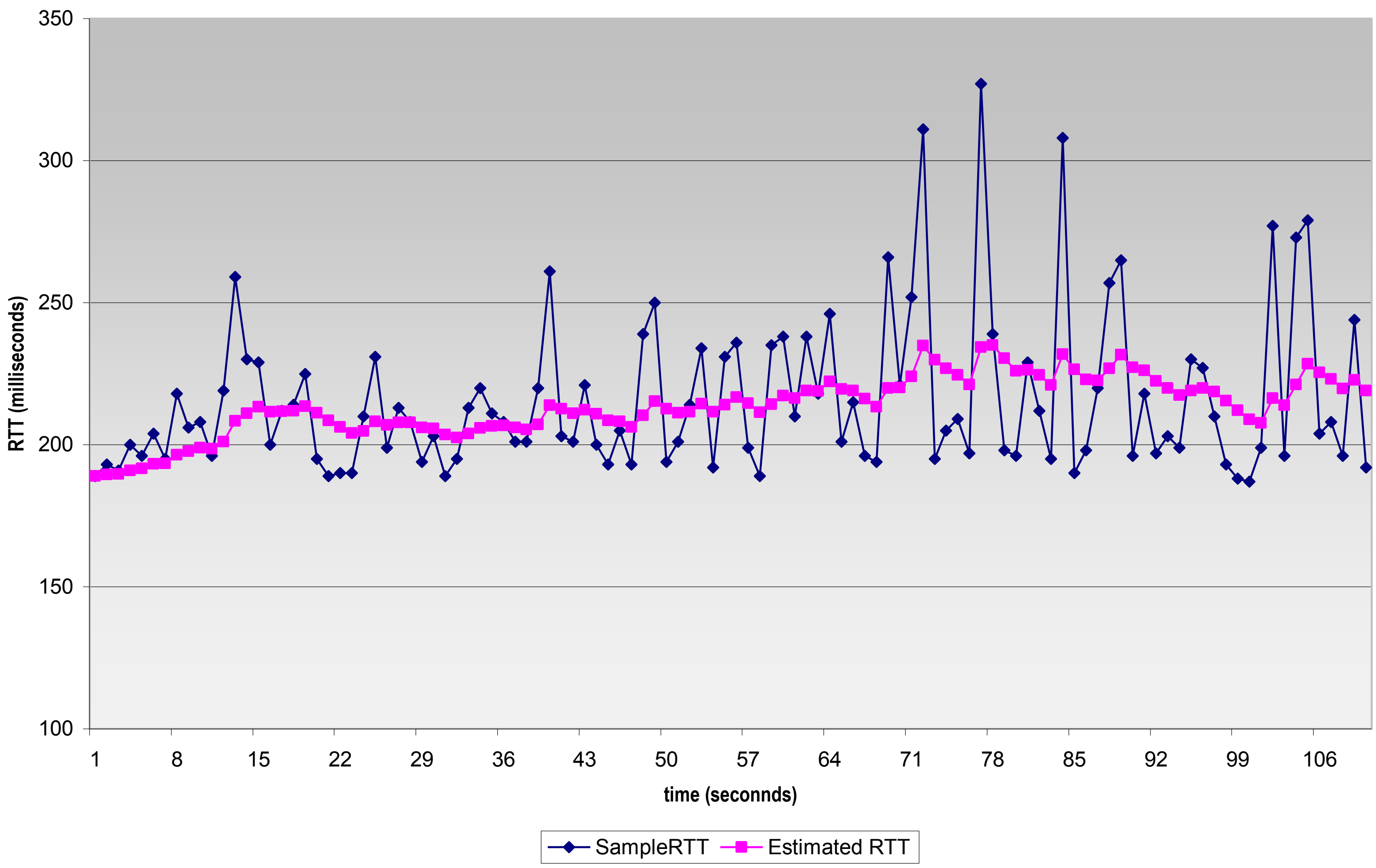
- **How can we estimate RTT?**
- Measure time from segment transmission until ACK receipt
  - Ignore retransmissions
  - Call this "SampleRTT"
- SampleRTT will vary
  - We want a "smoother" estimated RTT
  - Average several recent measurements, not just current SampleRTT

TCP Round Trip Time and Timeout

- Exponential weighted moving average
- Influence of past sample decreases exponentially fast
- Typical value: $\alpha = 0.125$

```
EstimatedRTT = (1- α)*EstimatedRTT + α*SampleRTT
```

Example RTT Estimation



TCP Round Trip Time and Timeout

- Setting the TCP timeout
  - EstimtedRTT plus “safety margin”
  - The larger the variation in EstimatedRTT, the larger the safety margin

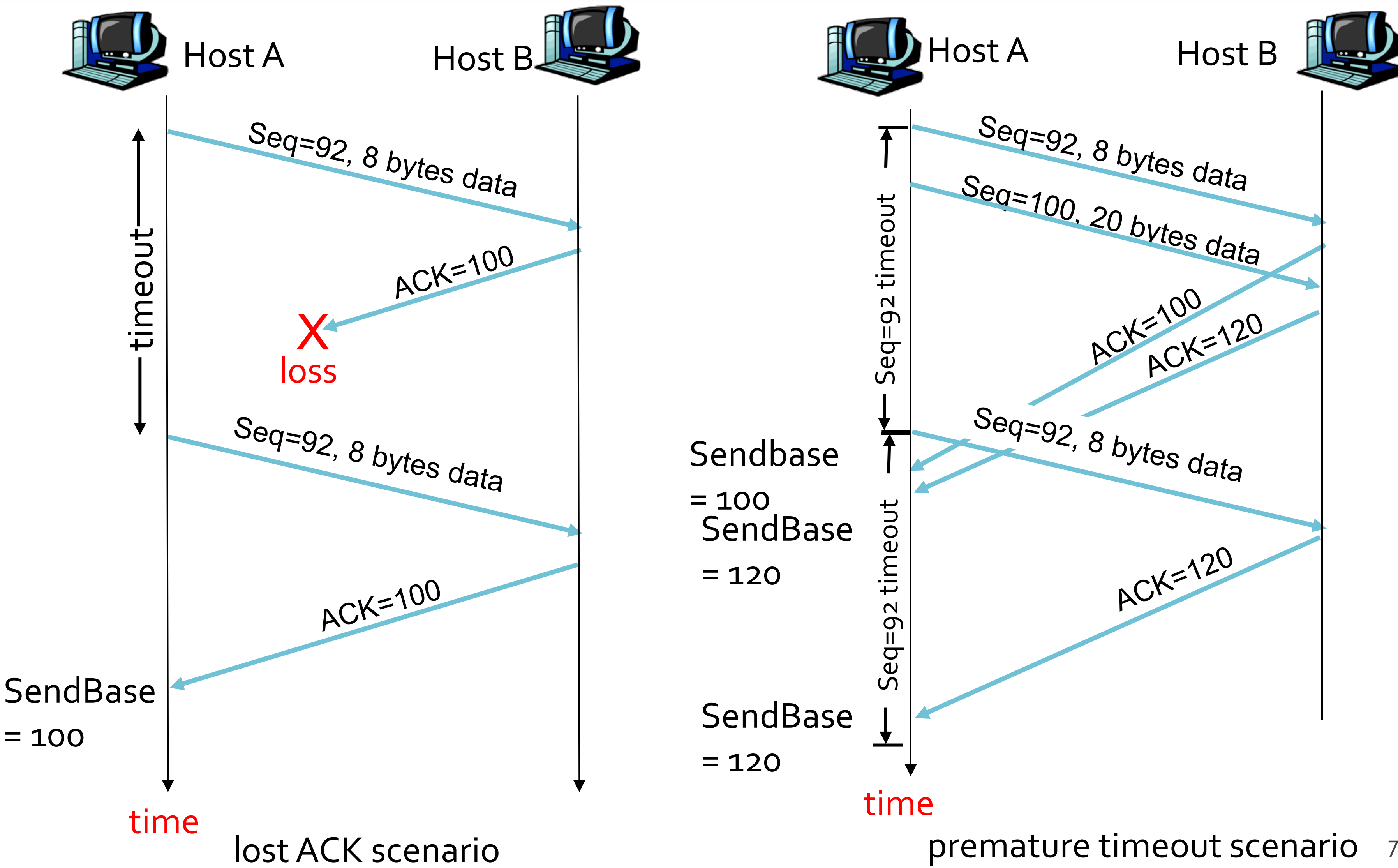
TCP Reliable Data Transfer

- TCP creates rdt service on top of unreliable IP
- Features of TCP
  - Pipelined segments
  - Cumulative ACKs
- TCP uses single retransmission timer
- Retransmissions are triggered by
  - timeout events
  - duplicate ACKs
- Initially consider simplified TCP sender:
  - Ignore duplicate ACKs
  - Ignore flow control
  - Ignore congestion control

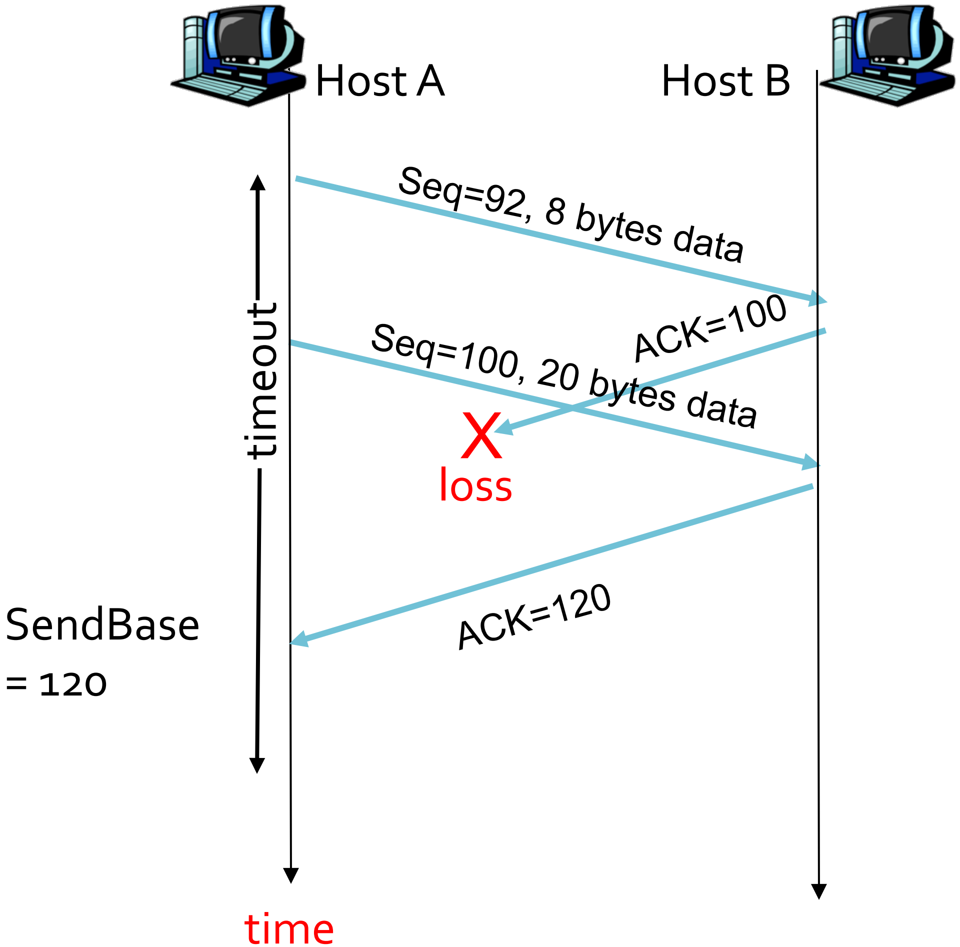
TCP Sender Events

- Data received from app:
  - Create segment with seq #
  - seq # is byte-stream number of first data byte in segment
  - Start timer if not already running (think of timer as for oldest unACKed segment)
  - Expiration interval: TimeOutInterval
- Timeout:
  - Retransmit segment that caused timeout
  - Restart timer
- ACK received:
  - If acknowledges previously unACKed segments
    - Update what is known to be ACKed
    - Start timer if there are outstanding segments

TCP Retransmission Scenarios



TCP Retransmission Scenarios



Cumulative ACK scenario

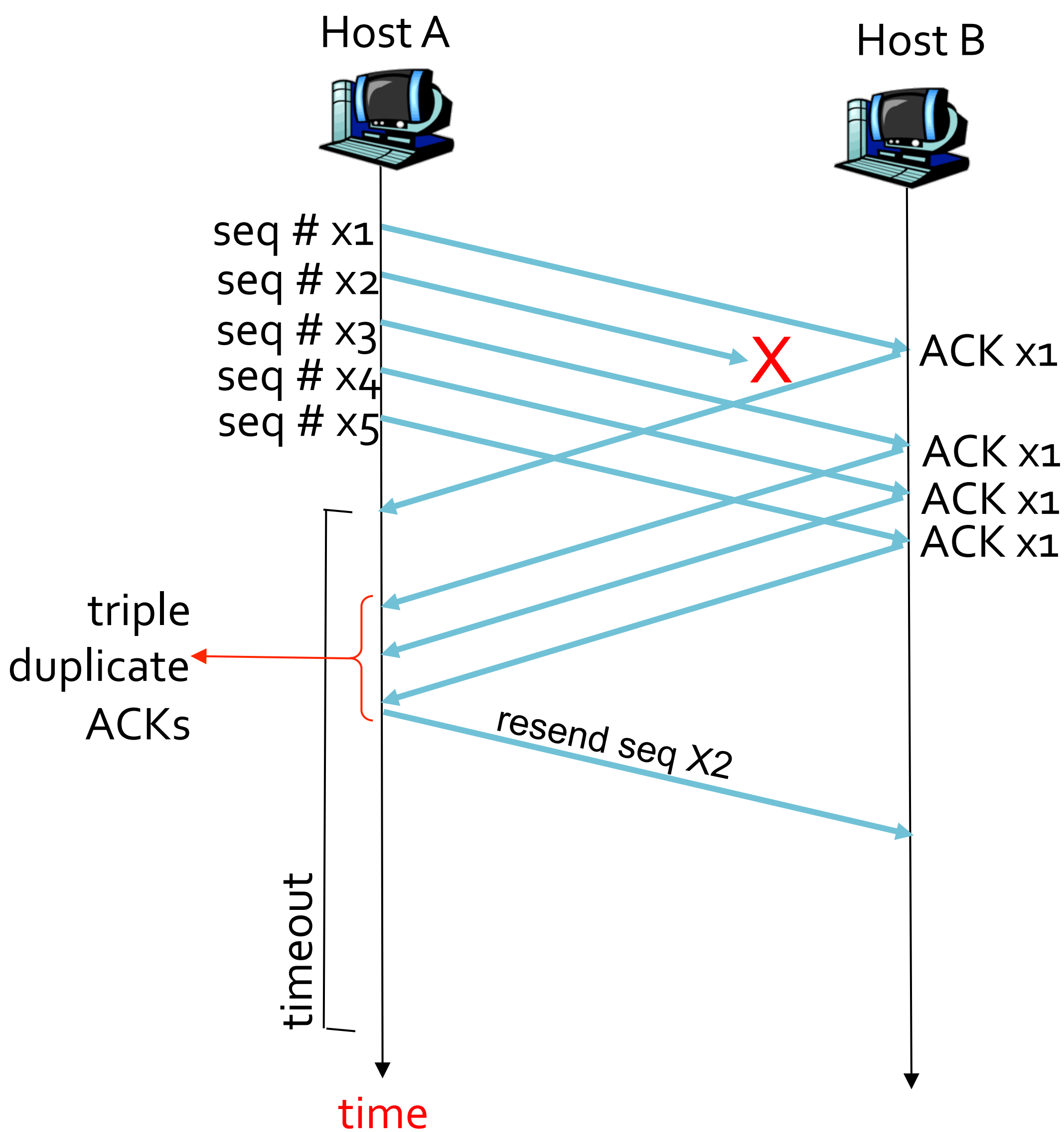
TCP ACK generation [RFC 1122, RFC 2581]

Event at Receiver	TCP Receiver action
Arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
Arrival of in-order segment with expected seq #. One other segment has ACK pending	Immediately send single cumulative ACK, ACKing both in-order segments
Arrival of out-of-order segment higher-than-expect seq. # . Gap detected	Immediately send *duplicate ACK*, indicating seq. # of next expected byte
Arrival of segment that partially or completely fills gap	Immediate send ACK, provided that segment starts at lower end of gap

Fast Retransmit

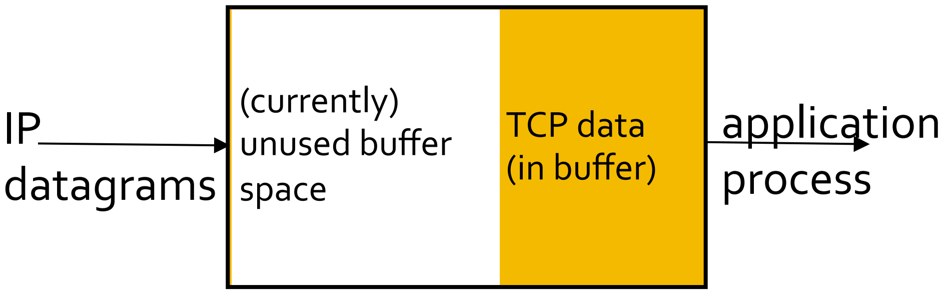
- Time-out period often relatively long
  - Long delay before resending lost packet
- Detect lost segments via duplicate ACKs.
  - Sender often sends many segments back-to-back
  - If segment is lost, there will likely be many duplicate ACKs for that segment

- If sender receives 3 ACKs for same data, it assumes that segment after ACKed data was lost:
  - **Fast retransmit**: resend segment before timer expires


Host A
Host B
seq # x1
seq # x2
seq # x3
seq # x4
seq # x5
X
ACK x1
ACK x1
ACK x1
ACK x1
triple
duplicate
ACKs
resend seq X2
timeout
time

TCP Flow Control

- Receive side of TCP connection has a receive buffer:



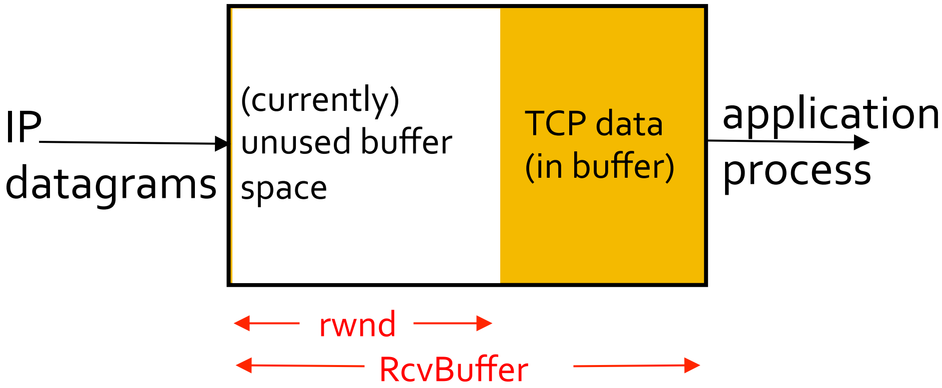
- Application process may be slow at reading from buffer
  - What if buffer fills up?

Flow Control:

Prevents **sender** from **overflowing receiver's buffer** by transmitting too much, too fast

Speed matching service: matching send rate to receiving application's drain rate

TCP Flow Control: How it Works



- Suppose TCP receiver discards out-of-order segments…
- Unused buffer space = rwnd

  = RcvBuffer-[LastByteRcvd - LastByteRead]

- Receiver notifies sender of unused buffer space
  - Segment header includes the `rwnd` value
- Sender limits # of unACKed bytes to `rwnd`
  - Guarantees receiver's buffer doesn't overflow

TCP Connection Management

- TCP sender and receiver establish "connection" before exchanging data segments
  - Client initiates connection
    - Calls connect() to an IP/port
  - Server is contacted by client
    - Calls accept()
- TCP variables initialized while establishing connection
  - Sequence #s
  - Buffers and flow control info (e.g. RcvWindow)

- Three way handshake:
- **Step 1:** client host sends TCP SYN segment to server
  - Specifies initial seq #
  - No data
- **Step 2:** server host receives SYN, replies with SYNACK segment
  - Server allocates buffers
  - Specifies server initial seq. #
- **Step 3:** client receives SYNACK, replies with ACK segment, which may contain data

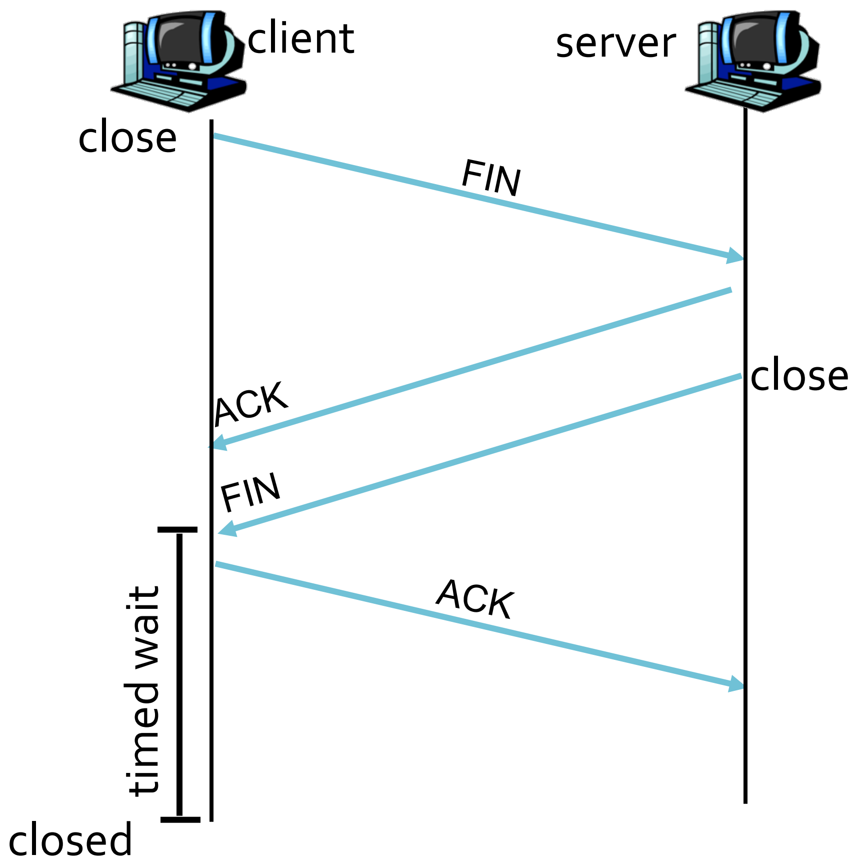
TCP Connection Management

Closing a connection:

client closes socket via close():

Step 1: client system sends TCP FIN control segment to server

Step 2: server receives FIN, replies with ACK. Closes connection, sends FIN.

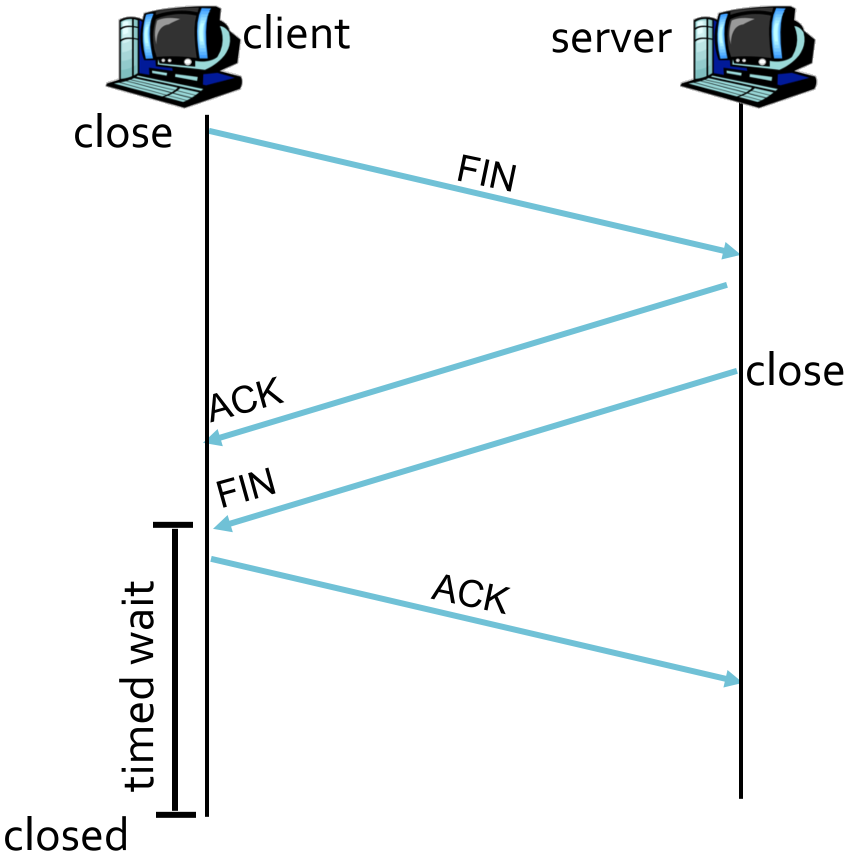


TCP Connection Management

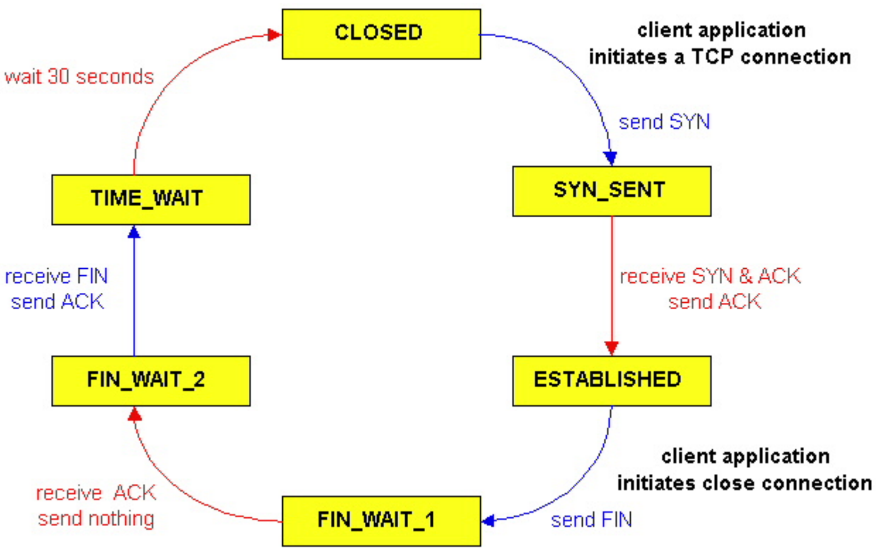
Step 3: client receives FIN, replies with ACK.

- Enters "timed wait" - will respond with ACK to received FINs
- Why not close immediately? Avoids potential problems if a *new* socket (with same ports and IPs) is created, and then old delayed data is delivered…

Step 4: server, receives ACK. Connection closed

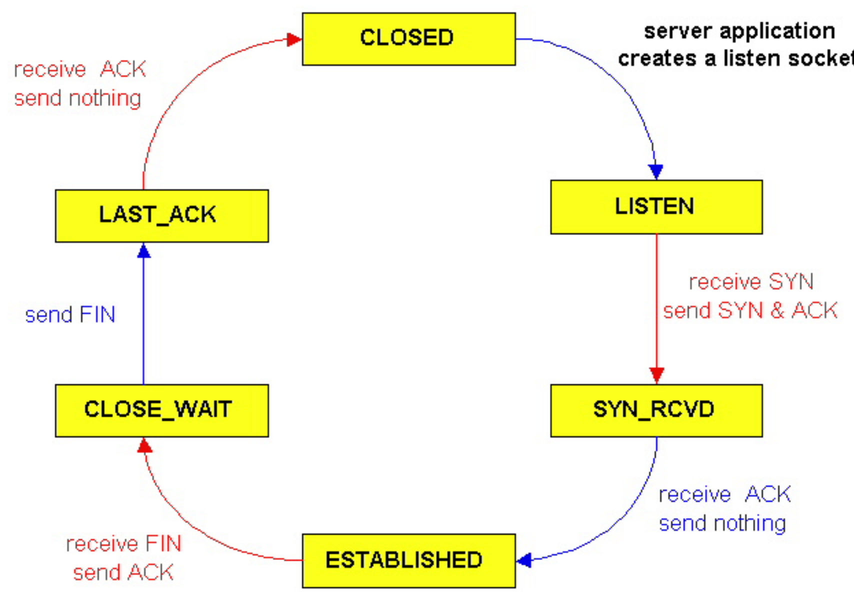


TCP Connection Management



TCP client lifecycle

TCP server lifecycle



Congestion Control

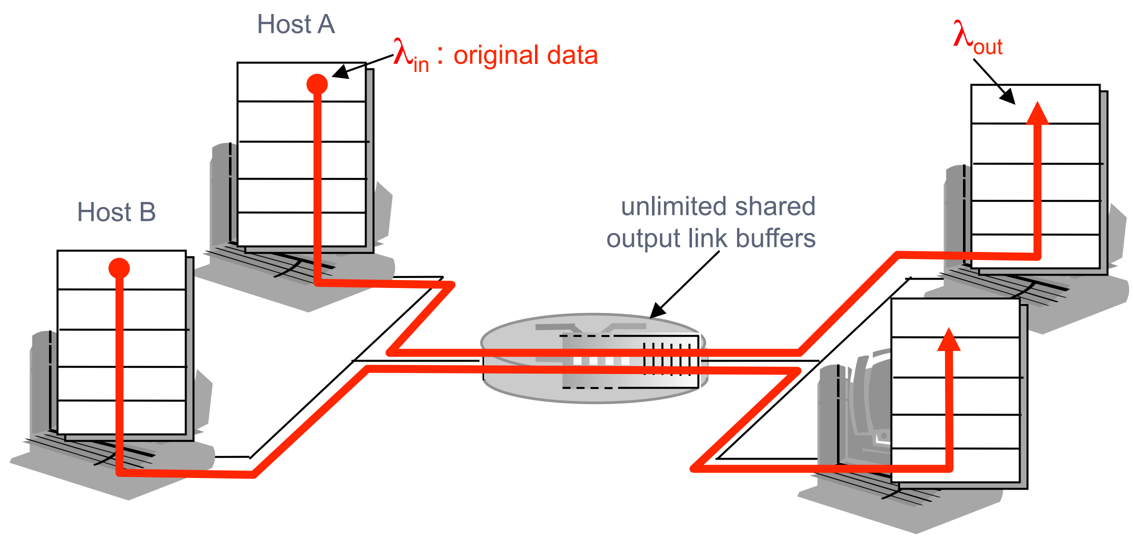
Principles and TCP Specifics

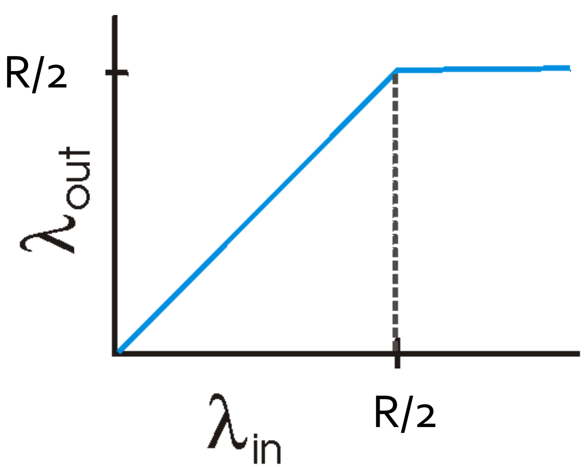
Principles of Congestion Control

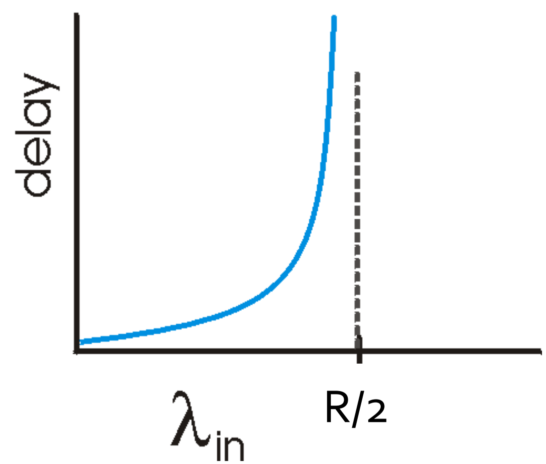
- What is congestion?
  - Informally: "too many sources sending too much data too fast for **network** to handle"
- Different from flow control!
- Manifestations
  - Lost packets (buffer overflow at routers)
  - Long delays (queueing in router buffers)

Causes/costs of Congestion: Scenario 1

- Two senders, two receivers
- One router, **infinite** buffers
- No retransmission
- Link BW of R



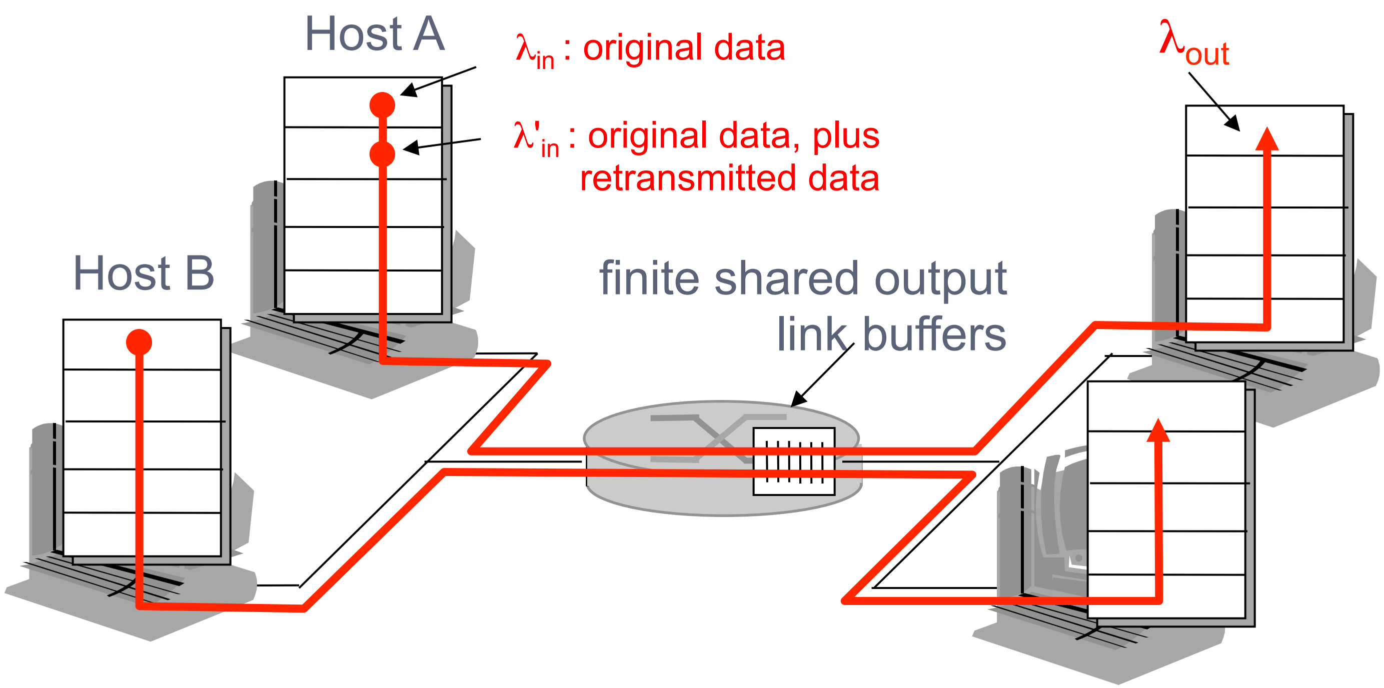




- **Large delays** when congested
- Maximum achievable throughput

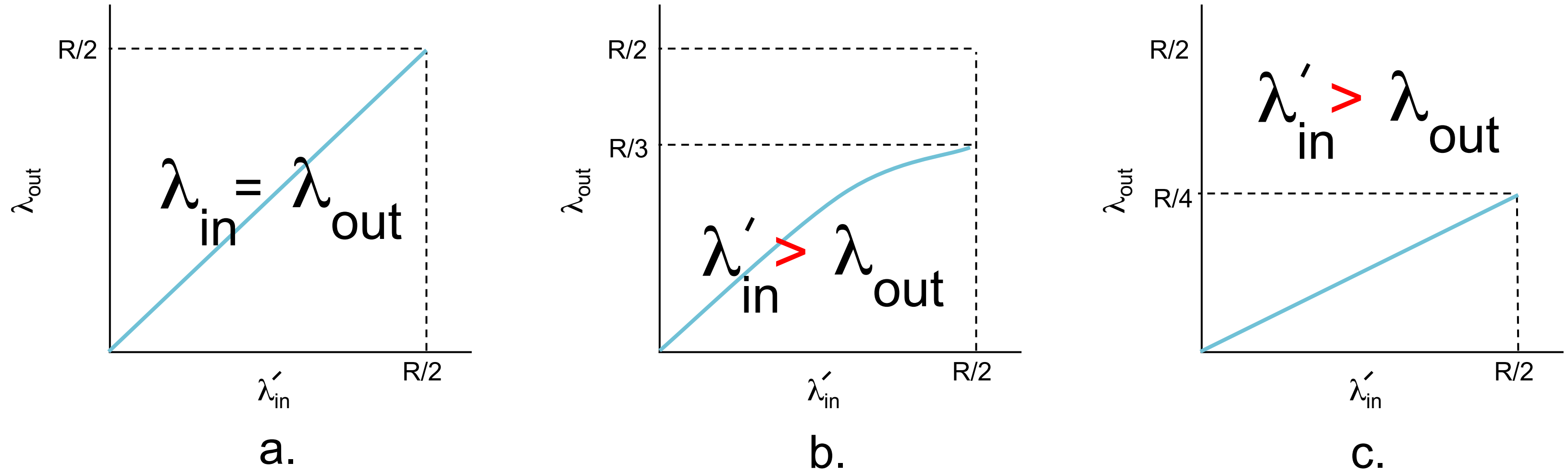
Causes/costs of Congestion: Scenario 2

- One router, **finite** buffers
- Sender retransmission of lost packet



Causes/costs of Congestion: Scenario 2

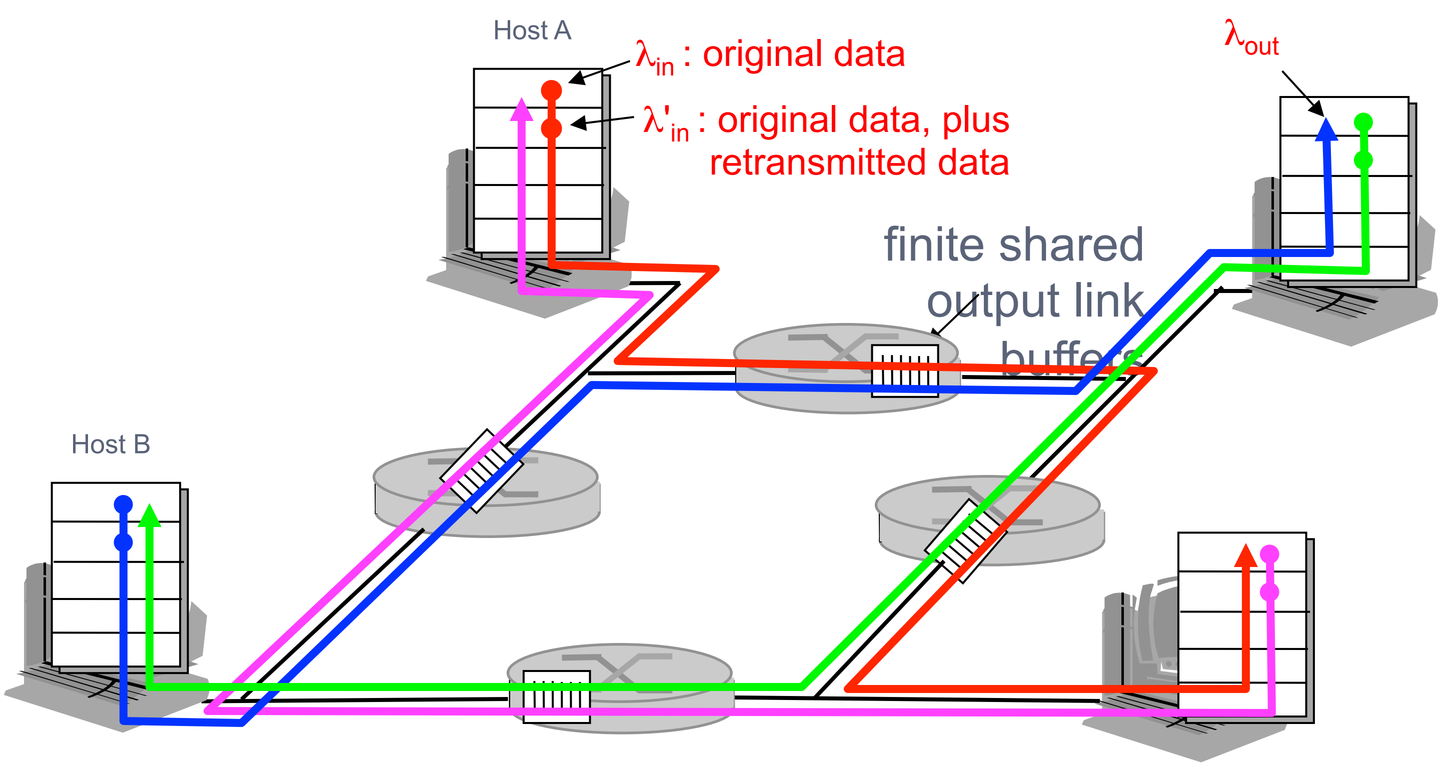
- Case a: Sender only transmits when it knows buffer space is available in router (unrealistic)
- Case b: Sender retransmits only when packet is known to be lost
  - New cost of congestion: More sender work (retrans) for given "goodput"
- Case c: Assume sender also retransmits when a packet is delayed (not lost), i.e. a premature timeout (bigger λ_{in}')
  - New cost of congestion: router output link carries multiple copies of packet



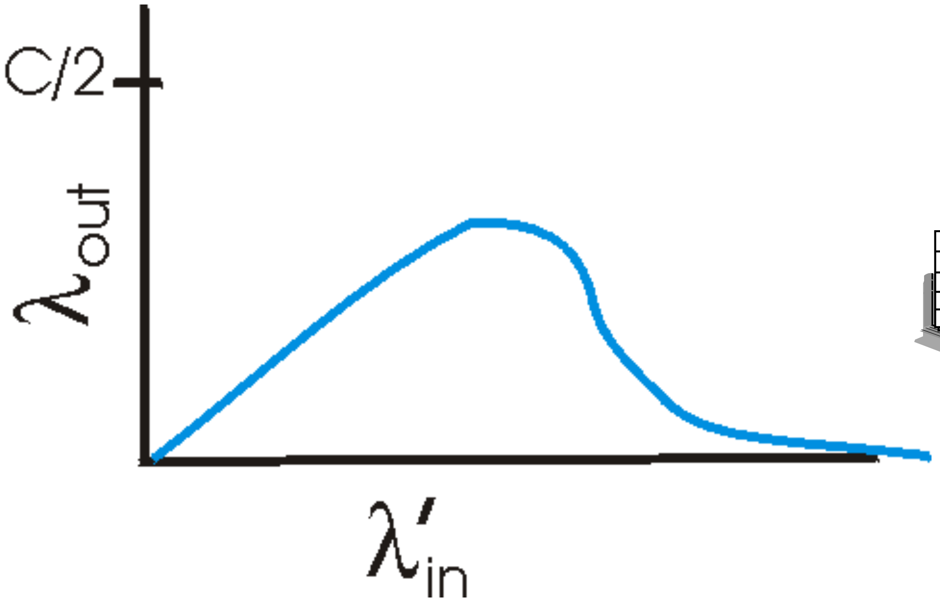
Causes/costs of Congestion: Scenario 3

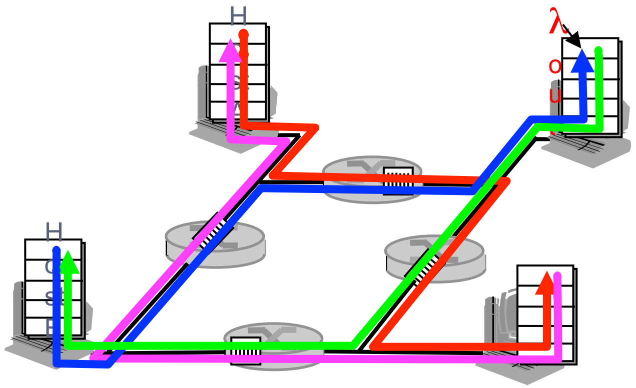
- Four senders
- Multihop paths
- Timeout/retransmit

Q: what happens as λ_{in} and λ'_{in} increase?



Causes/costs of Congestion: Scenario 3





- A new cost of congestion
  - When packet dropped, any upstream transmission capacity used for that packet was wasted!

Congestion Control Approaches

Two broad approaches to congestion control:

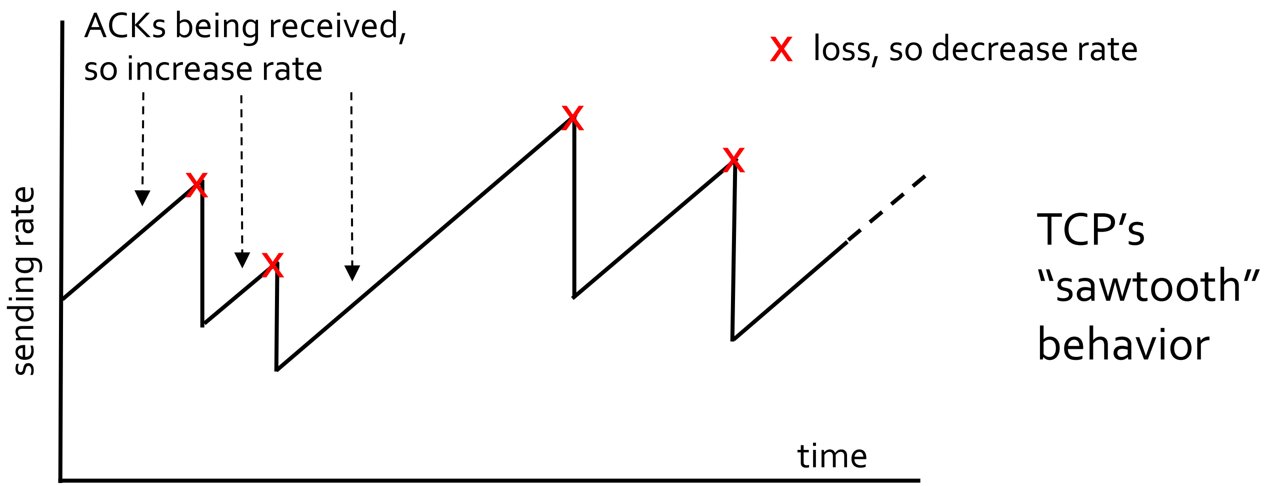
- **End-end congestion control:**
  - No explicit feedback from network
  - Congestion inferred from end-system observed loss, delay
  - **Approach taken by TCP**
- **Network-assisted congestion control:**
  - Routers provide feedback to end systems
  - Single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
  - Explicit rate sender should send at

TCP Congestion Control

- Goal: TCP sender should transmit **as fast as possible**, but without congesting network
- How do we find the rate just below congestion level?
  - Decentralized approach – each TCP sender sets its own rate, based on *implicit* feedback:
  - ACK indicates segment received (a good thing!)
    - Network not congested, so increase sending rate
  - Lost segment – assume loss is due to congested network, so decrease sending rate

TCP Congestion Control: Bandwidth Probing

- Probing for bandwidth
  - Increase transmission rate on receipt of ACK, until eventually loss occurs, then decrease transmission rate



- How fast to increase or decrease?

TCP Congestion Control: Details

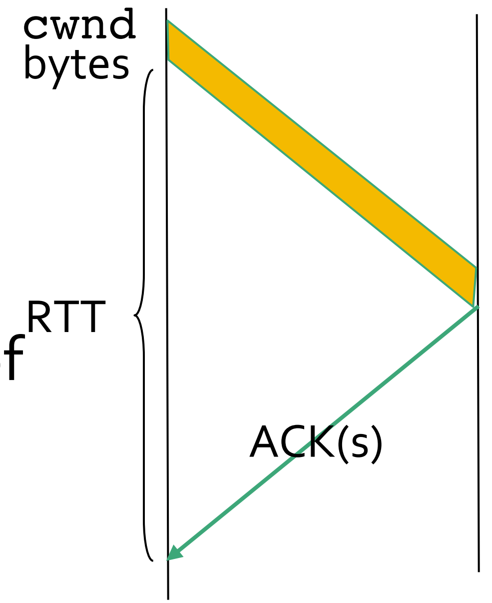
- Sender limits rate by limiting number of unACKed bytes "in pipeline":

  LastByteSent-LastByteAcked ≤ cwnd

  - **cwnd:** differs from **rwnd** (how, why?)
  - sender limited by **min(cwnd,rwnd)**
- Roughly,

$$\text{rate} = \frac{\text{cwnd}}{\text{RTT}} \text{ bytes/sec}$$

- **cwnd** (congestion window) is a function of perceived network congestion
- **rwnd** (receiver window) is explicitly changed by receiver



TCP Congestion Control Details

BAD: DATA LOSS

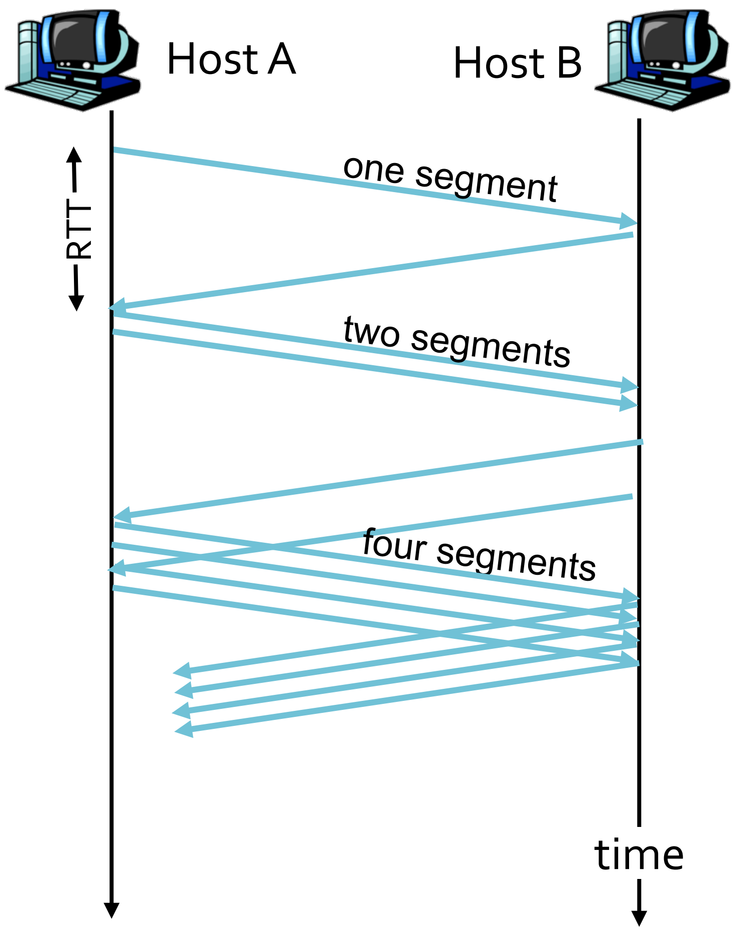
- **3 duplicate ACKs?**
  - At least some segments getting through (*recall fast retransmit*)
  - Cut `cwnd` in half
- **Timeout?**
  - No response from receiver
  - Cut `cwnd` to 1
    - Very aggressive slowdown!

GOOD: ACK RECEIVED

- **Slowstart phase** (connection start or after timeout):
  - Increase `cwnd` exponentially fast
  - Think name is backwards? Think "avoiding a slow start"
- **Congestion avoidance**
  - Increase `cwnd` linearly

TCP Slow Start Phase

- When connection begins, **cwnd** = 1 MSS
  - Example: MSS = 500 bytes & RTT = 200 msec
  - Initial rate = 20 kbps
- Available bandwidth may be >> MSS/RTT
  - Goal: quickly ramp up to respectable rate
- **Increase rate exponentially** until first loss event or when threshold reached
  - Double **cwnd** every RTT
  - Done by incrementing **cwnd** by 1 for every ACK received



Leaving Slow Start via ssthresh

- Slow start phase must end
  - If we keep exponentially increasing our bandwidth usage, we're sure to cause congestion!
- When do we stop increasing rapidly?
  - Based on a new value: **`ssthresh`**
    - Slow start threshold maintained by TCP
    - When **`cwnd`** >= **`ssthresh`**: transition from slowstart to congestion avoidance phase
- Starts off high at first
- Set to **`ssthresh=cwnd/2`** when data loss occurs
    - Remember (half of) TCP rate when congestion last occurred

TCP Congestion Avoidance Phase

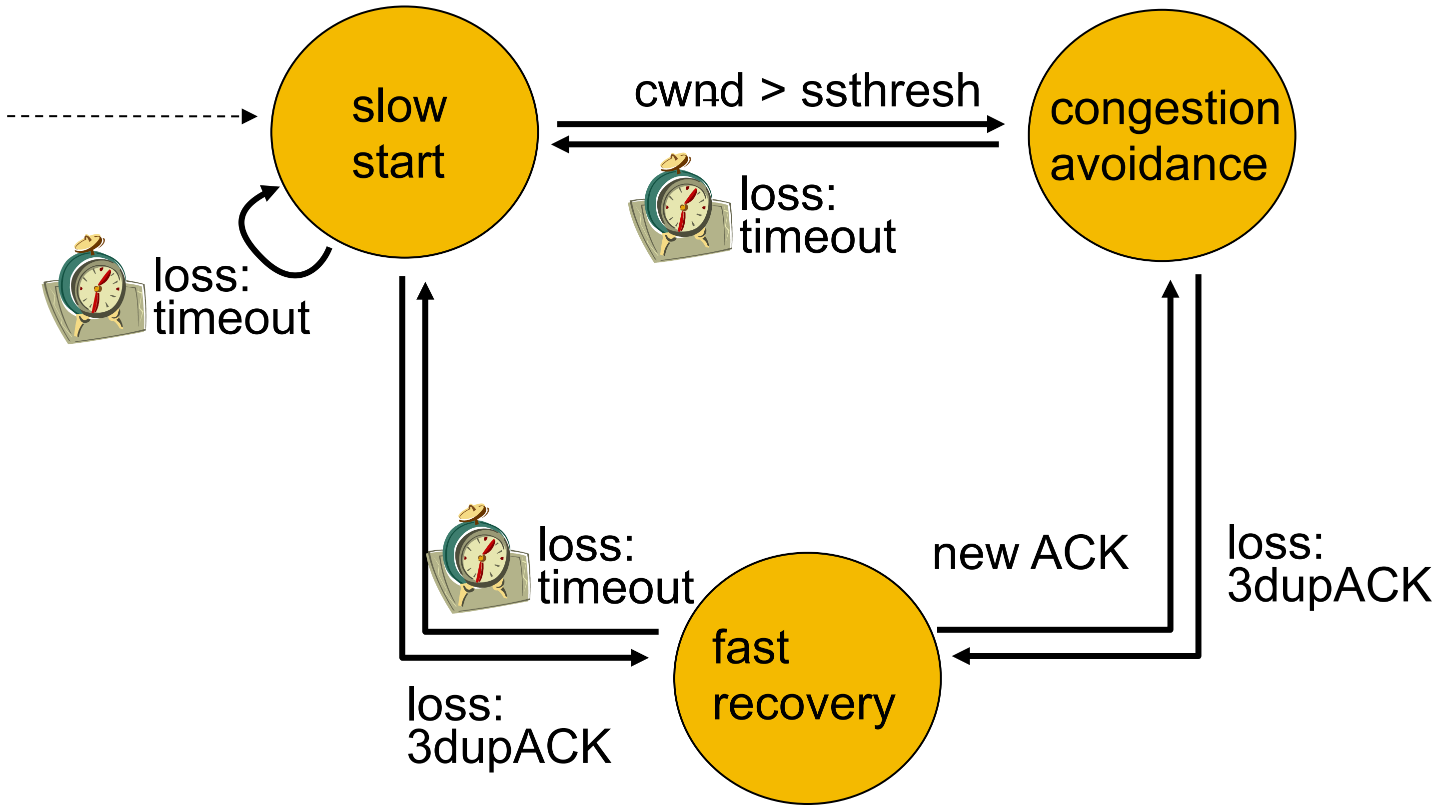
- New phase: when `cwnd > ssthresh`, grow `cwnd` linearly
  - Increase `cwnd` by 1 MSS per RTT
  - Approach possible congestion slower than in slowstart

AIMD

- ACKs: increase `cwnd` by 1 MSS per RTT: additive increase
- loss: cut `cwnd` in half (non-timeout-detected loss): multiplicative decrease

AIMD: Additive Increase Multiplicative Decrease

TCP Congestion Control FSM: Overview

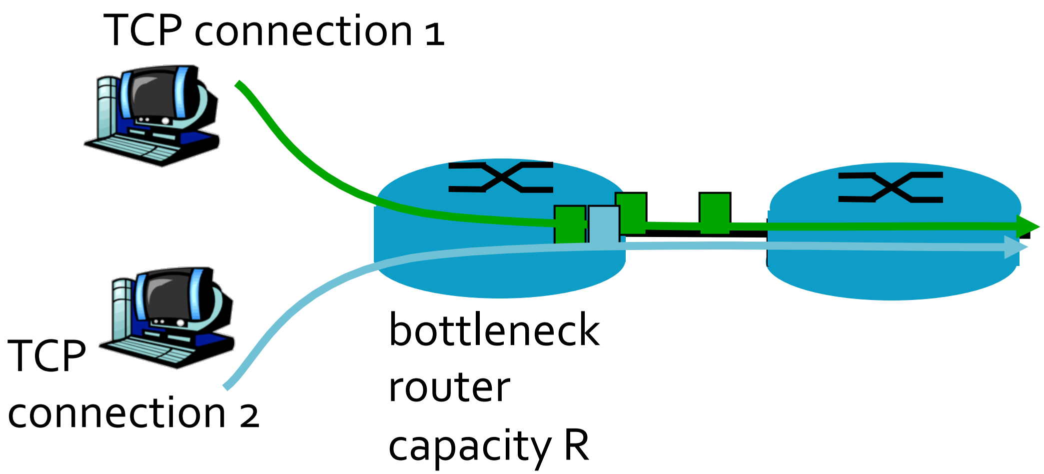


Summary: TCP Congestion Control

- When `cwnd` < `ssthresh`, sender in **slow-start phase**
  - Window grows exponentially.
- When `cwnd` >= `ssthresh`, sender is in **congestion-avoidance phase**
  - Window grows linearly.
- When 3 duplicate ACKs received
  - `ssthresh` set to `cwnd/2`
  - `cwnd` set to ~ `ssthresh`
- When timeout occurs
  - `ssthresh` set to `cwnd/2`
  - `cwnd` set to 1 MSS.

TCP Fairness

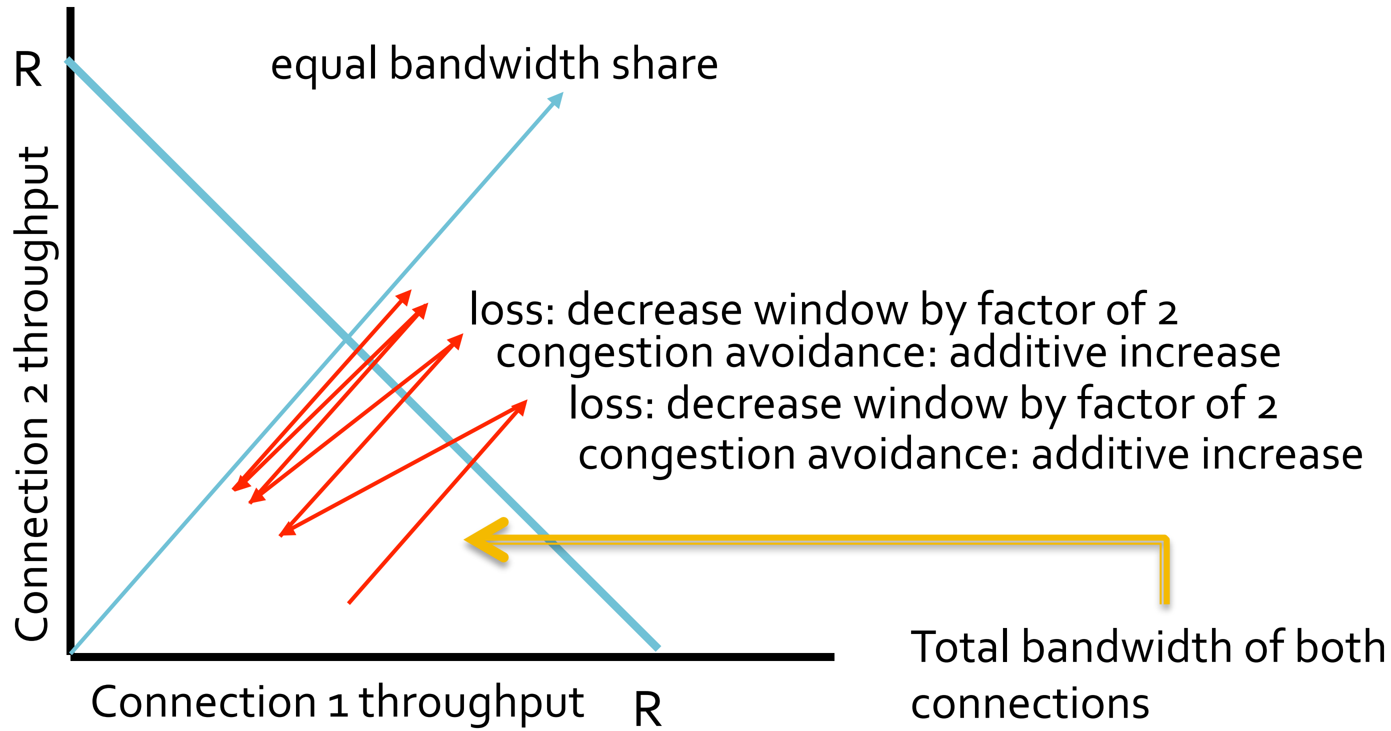
fairness goal: if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K



Why is TCP fair?

Two competing sessions:

- Additive increase gives slope of 1, as throughout increases
- Multiplicative decrease decreases throughput proportionally



Fairness

- **Fairness and UDP**
  - Multimedia apps often do not use TCP
    - Do not want rate throttled by congestion control
  - Instead use UDP
    - Pump audio/video at constant rate
    - Tolerate packet loss
  - UDP can "crowd out" TCP
- **Fairness and parallel TCP Connections**
  - Nothing prevents app from opening parallel connections between 2 hosts.
  - Web browsers do this
  - Example: link of rate R supporting 9 connections;
    - New app asks for 1 TCP, gets rate R/10
    - New app asks for 11 TCPs, gets > R/2 !

Summary

User Datagram Protocol (UDP) Characteristics

- UDP is a connectionless datagram service.
  - There is no connection establishment: packets may show up at any time.
- UDP packets are self-contained.
- UDP is unreliable:
  - No acknowledgements to indicate delivery of data.
  - Checksums cover the header, and only optionally cover the data.
  - Contains no mechanism to detect missing or mis-sequenced packets.
  - No mechanism for automatic retransmission.
  - No mechanism for flow control or congestion control (sender can overrun receiver or network)

TCP Characteristics

- TCP is connection-oriented.
  - 3-way handshake used for connection setup
- TCP provides a stream-of-bytes service
- TCP is reliable:
  - Acknowledgements indicate delivery of data
  - Checksums are used to detect corrupted data
  - Sequence numbers detect missing, or mis-sequenced data
  - Corrupted data is retransmitted after a timeout
  - Mis-sequenced data is re-sequenced
  - (Window-based) Flow control prevents over-run of receiver
- TCP uses congestion control to share network capacity among users