

ELEC / COMP 177 – Fall 2013

Computer Networking

→ Transport Layer (TCP & UDP)

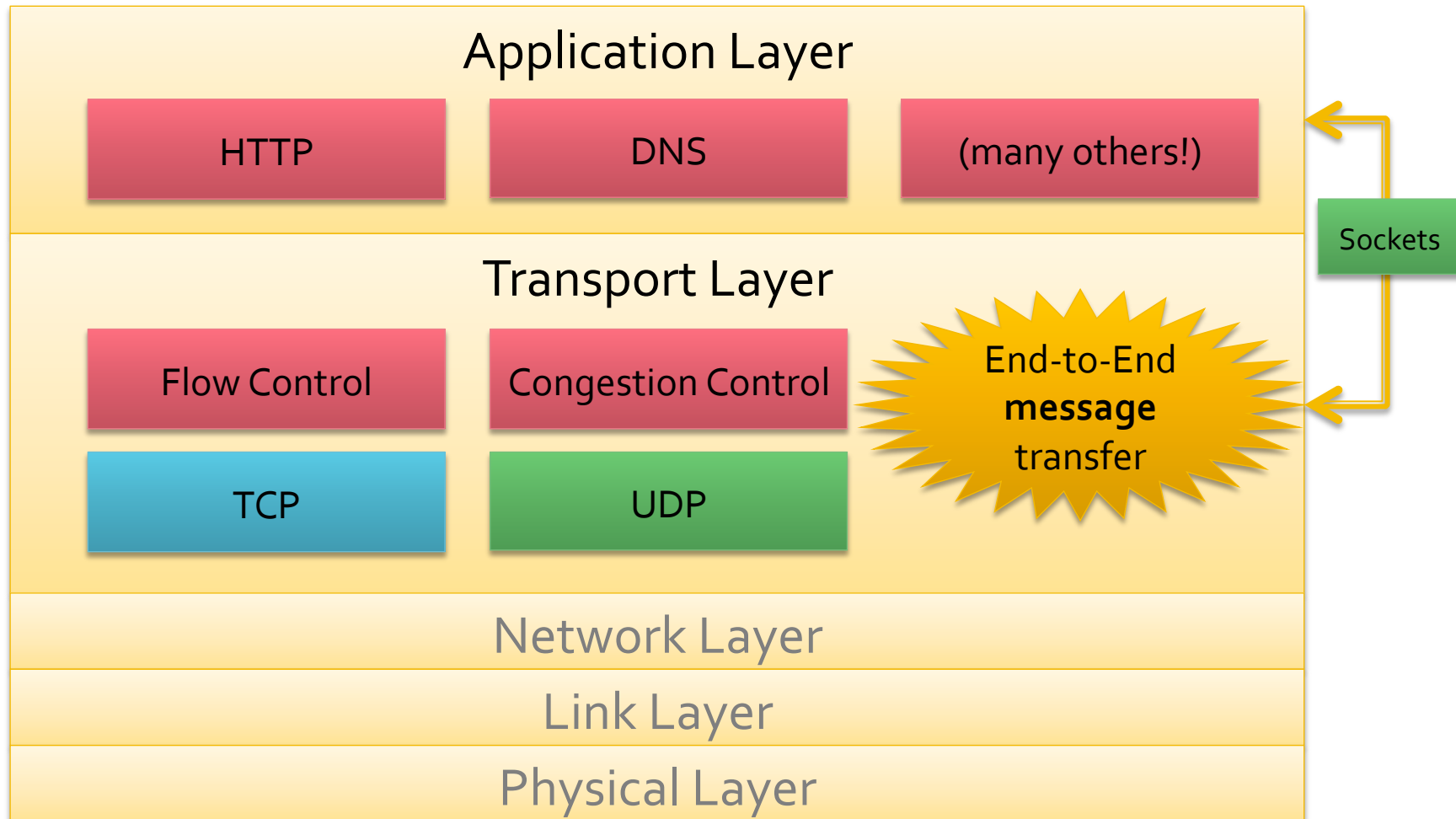
Some slides from Kurose and Ross, *Computer Networking*, 5th Edition

Upcoming Schedule

- Midterm Exam
 - Tuesday, October 15th
 - Bring laptop / USB key
- Open notes, open computer, open internet
- 1 programming problem using Python
 - Something to do with HTTP...
 - If you have your web server from projects 1/2 handy, you can repurpose it quickly...

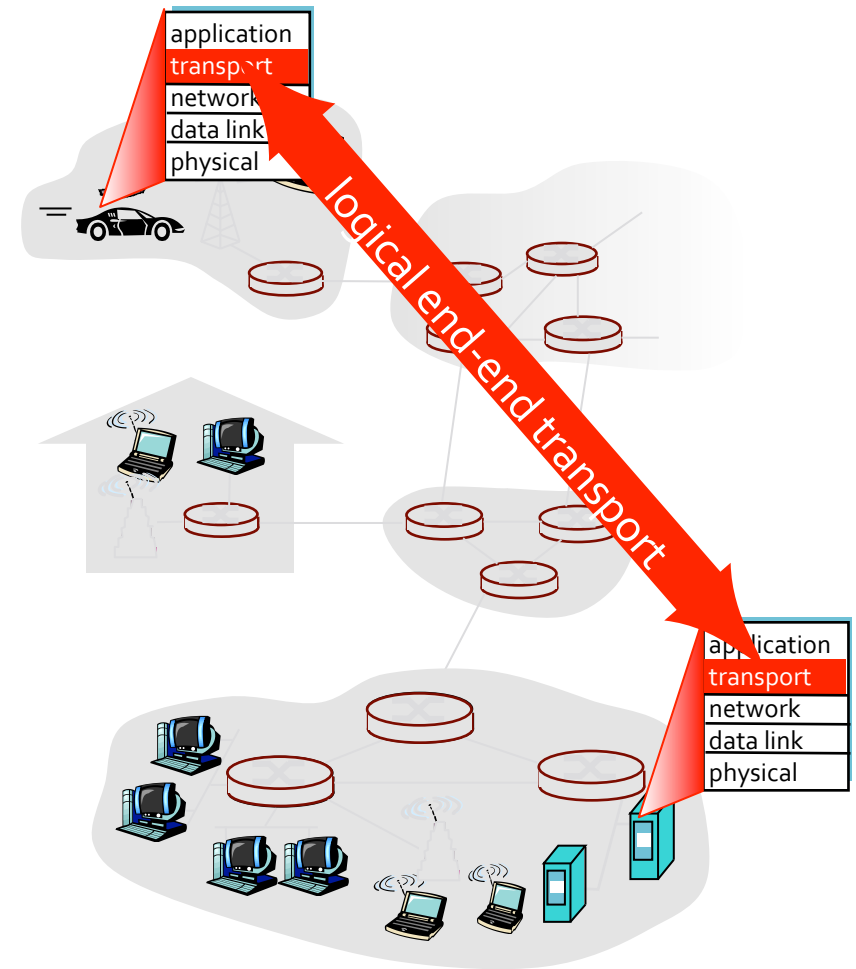
Transport Layer

Introducing the Transport Layer



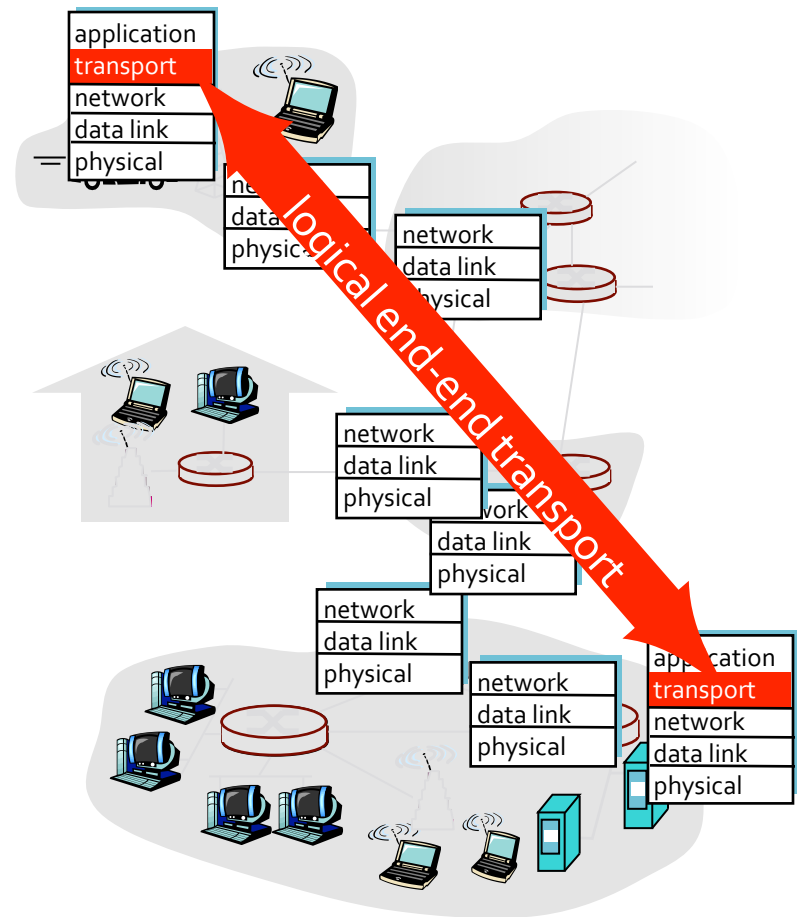
Goal of Transport Layer

- 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



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



UDP – User Datagram Protocol

Connectionless Transport

UDP: User Datagram Protocol [RFC 768]

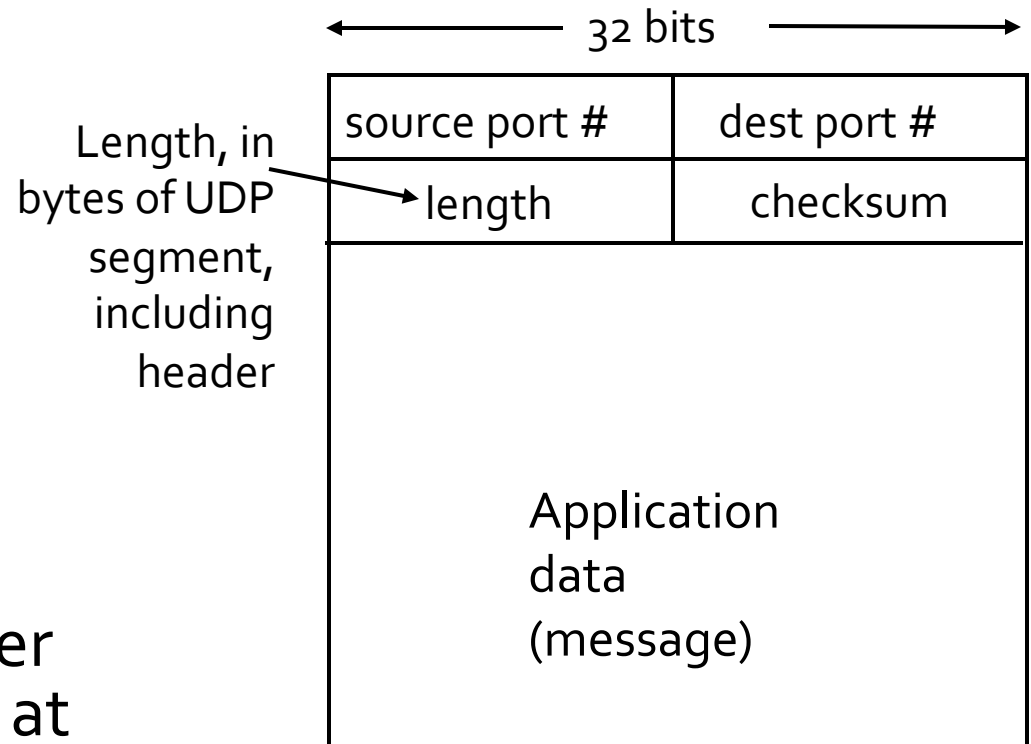
- “No frills, bare bones” Internet transport protocol
- “Best effort” service
- UDP segments may be:
 - Lost
 - Delivered out of order to app
- Connectionless
 - No handshaking between UDP sender, receiver
 - Each UDP segment handled independently of others

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

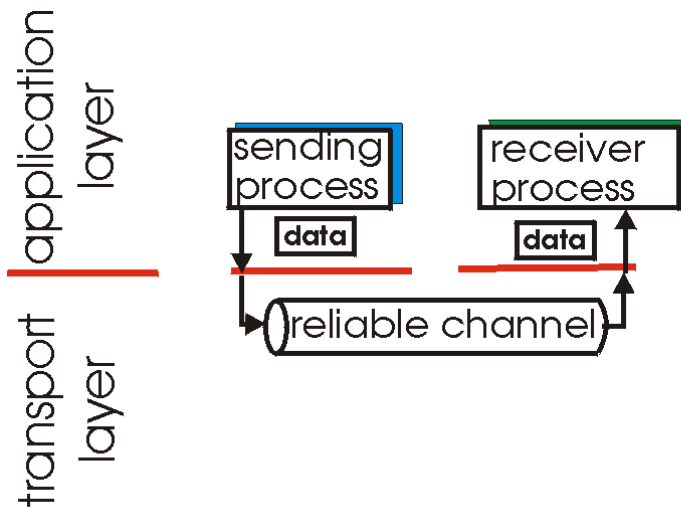
- Compute checksum of received segment
- Check if computed checksum equals checksum field value:
 - NO - error detected
 - YES - no error detected. But maybe errors nonetheless?

Reliable Data Transfer

Stepping through the design of TCP

Principles of Reliable data transfer

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

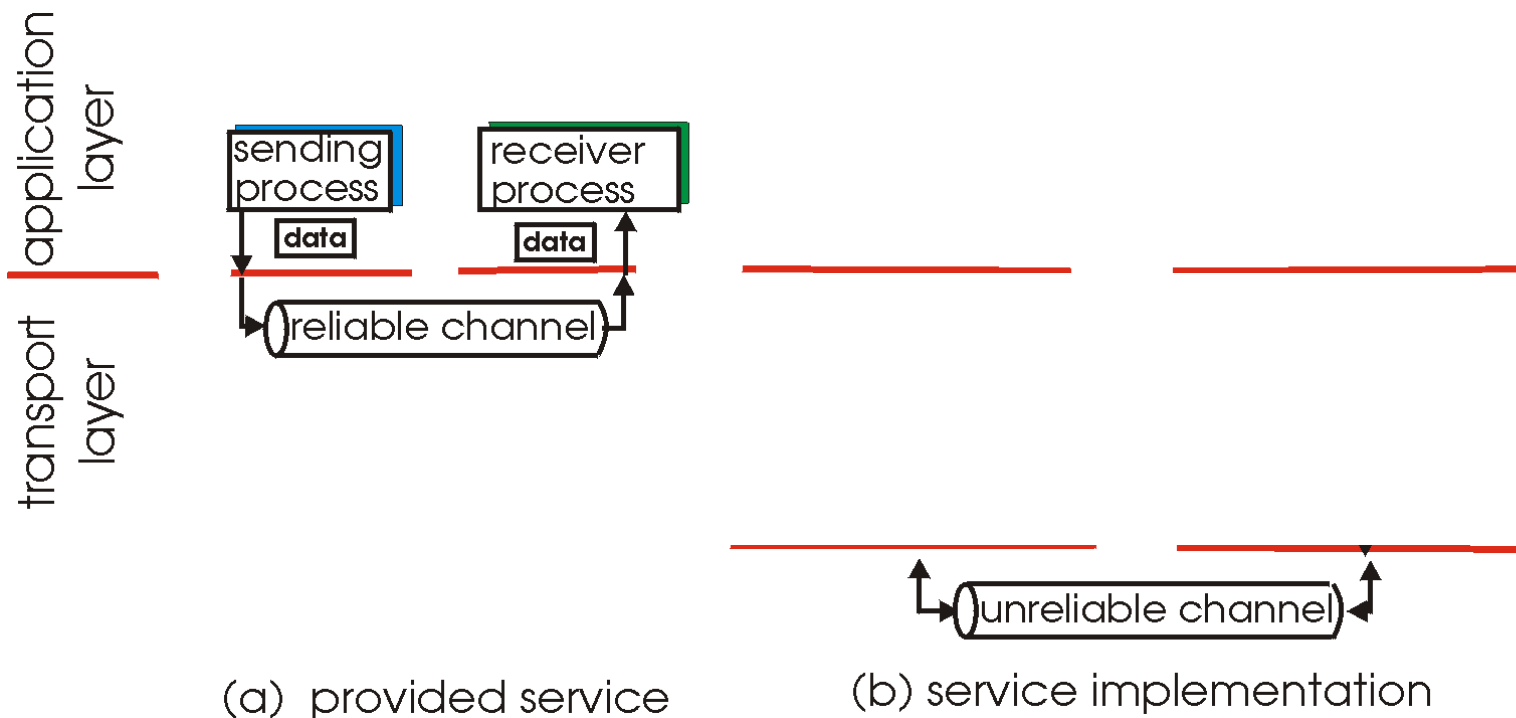


(a) provided service

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

Principles of Reliable data transfer

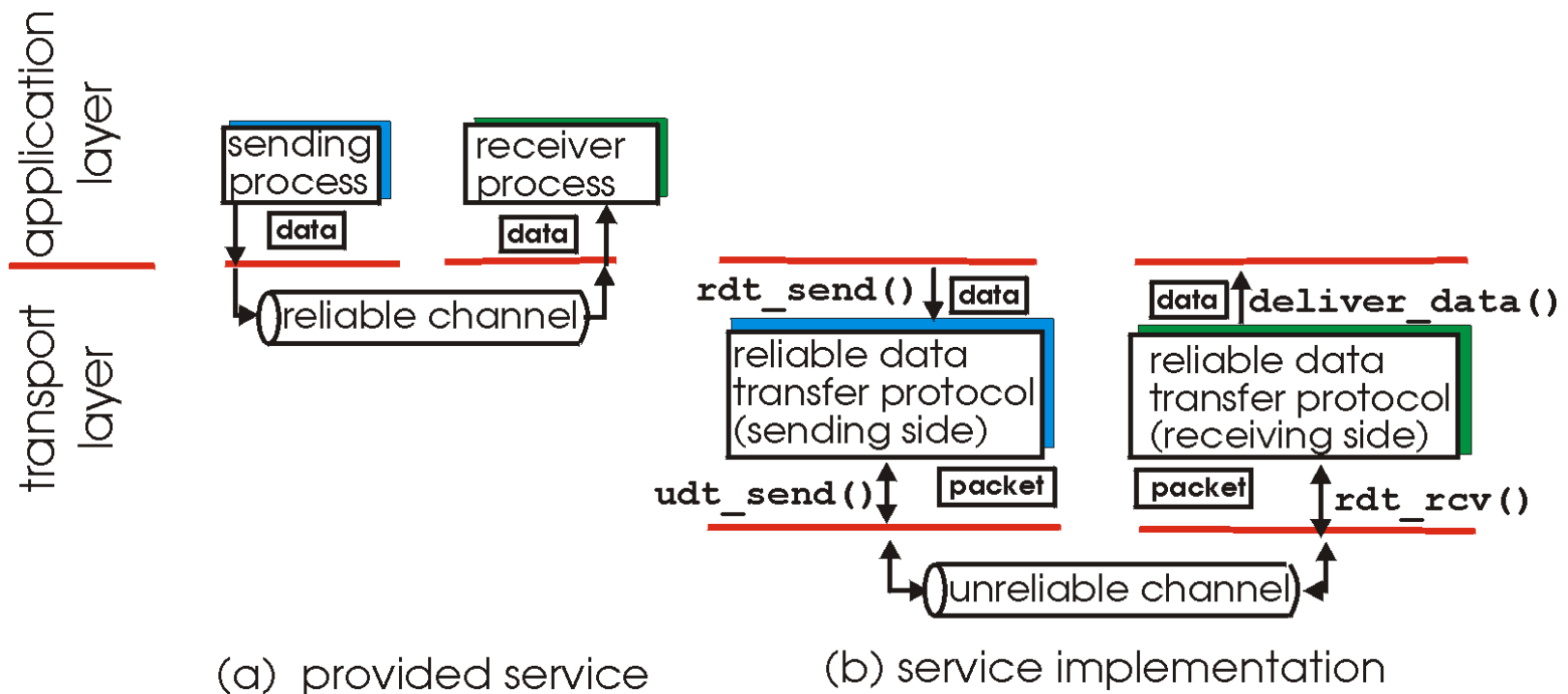
- Reliability is important in application, transport, and link layers



- 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

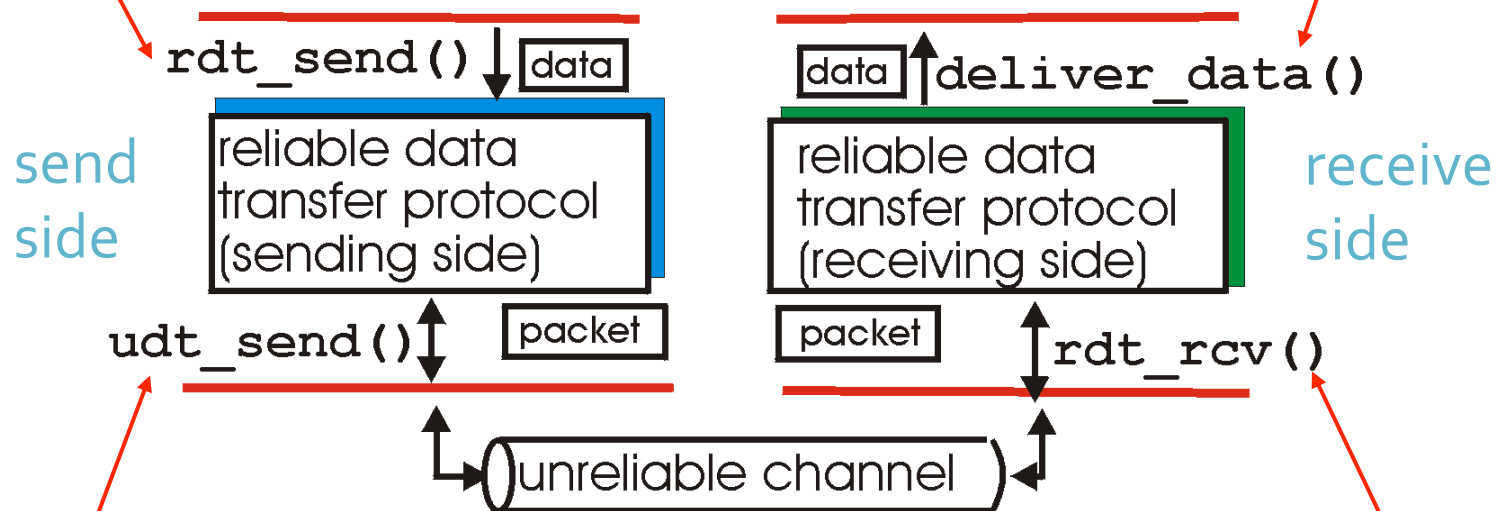


- Characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

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.

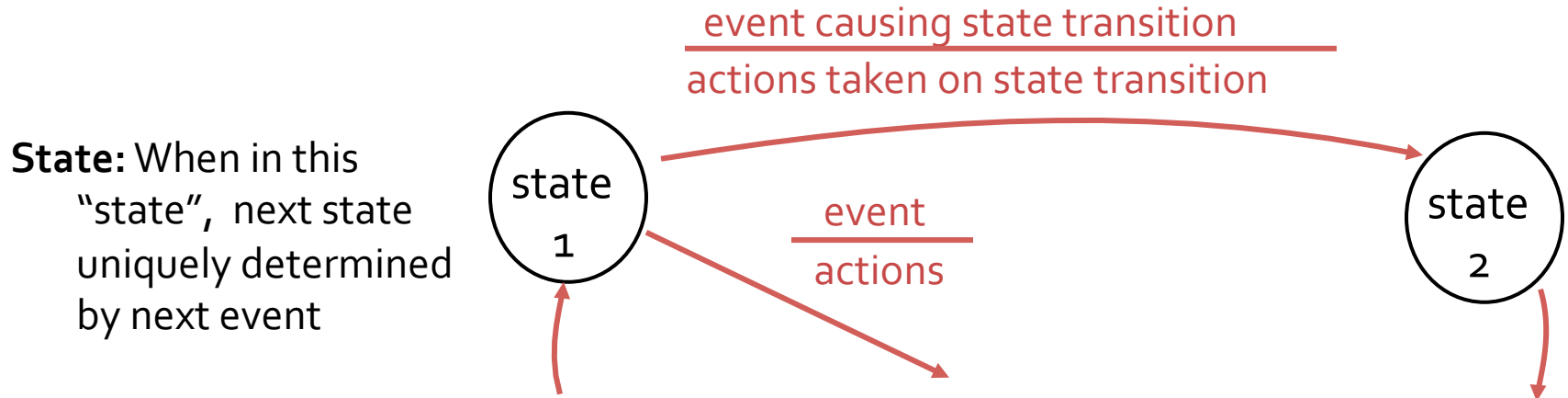


udt_send() : called by rdt, to transfer packet over unreliable channel to receiver

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

Intro to Reliable Data Transfer

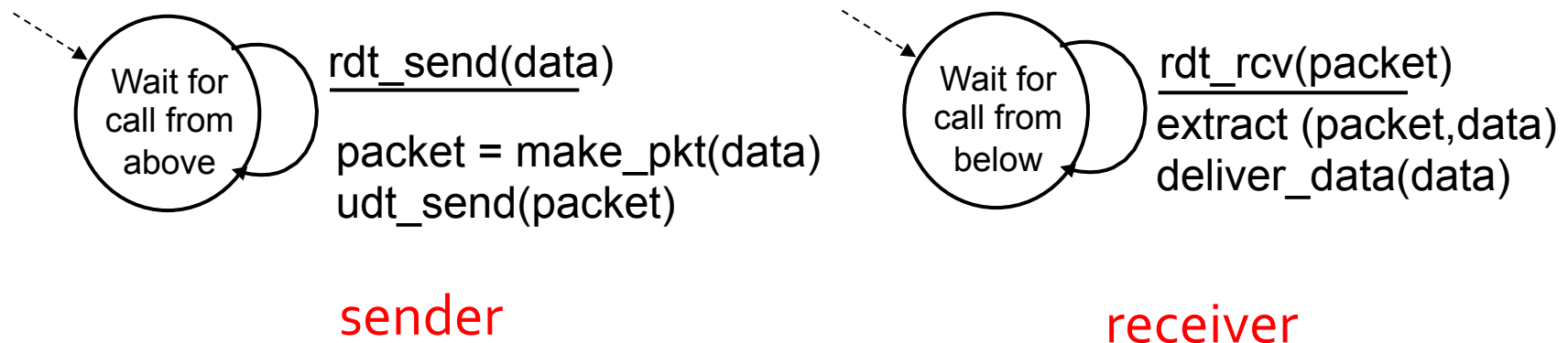
- The plan: **Incrementally** develop sender / receiver sides of **reliable data transfer protocol (rdt)**, a **fictional** protocol
 - TCP is similar to RDT but too complex to describe all at once
- Consider only unidirectional data transfer
 - but control info will flow on both directions!
- Use **finite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



Λ (uppercase Lambda = empty set)

rdt1.0: Reliable Transfer Over a Reliable Channel

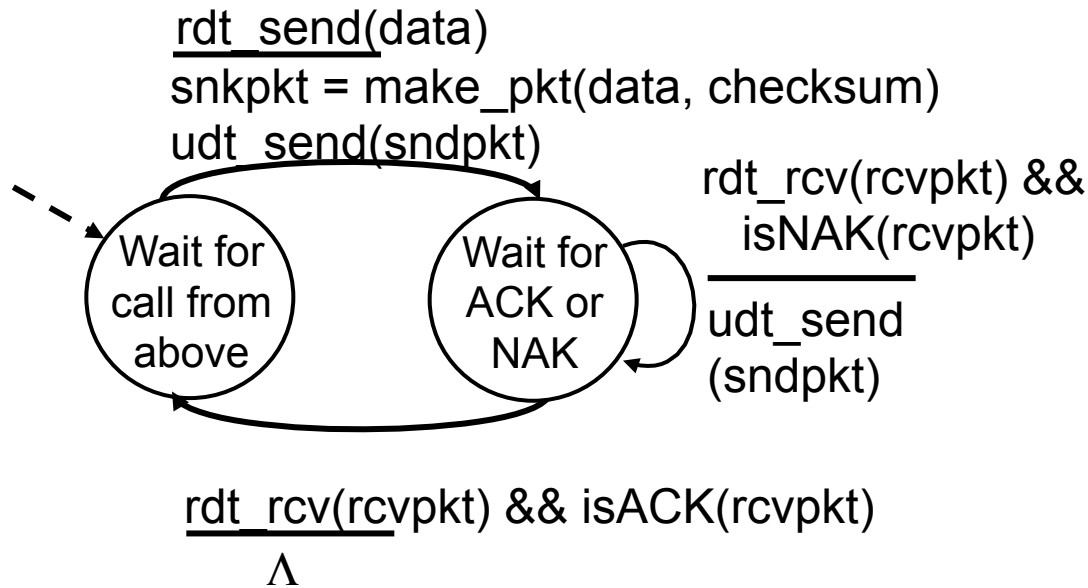
- Underlying channel perfectly reliable
 - No bit errors
 - No loss of packets
- Separate FSMs for sender, receiver:
 - Sender sends data into underlying channel
 - Receiver reads data from underlying channel



rdt2.0: Channel with Bit Errors

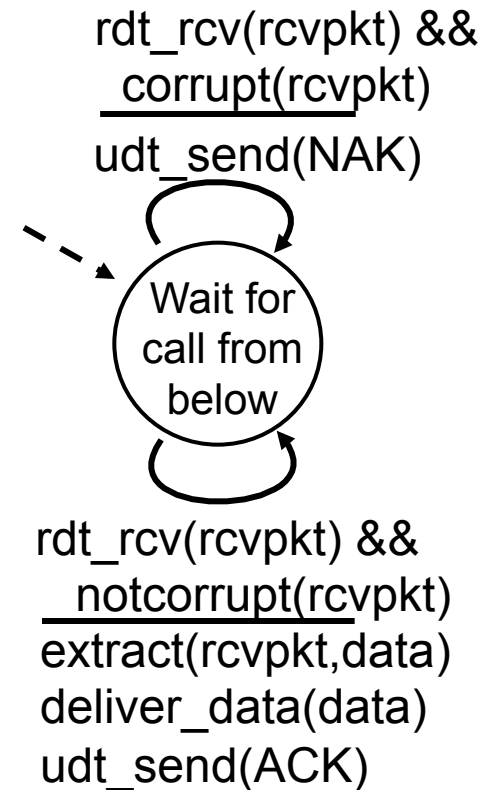
- Underlying channel may **flip bits** in packet
 - Checksum to detect bit errors
- But, how do we recover from errors?
 - **Acknowledgements (ACKs)**: receiver explicitly tells sender that packet received OK
 - **Negative acknowledgements (NAKs)**: receiver explicitly tells sender that packet had errors
 - 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

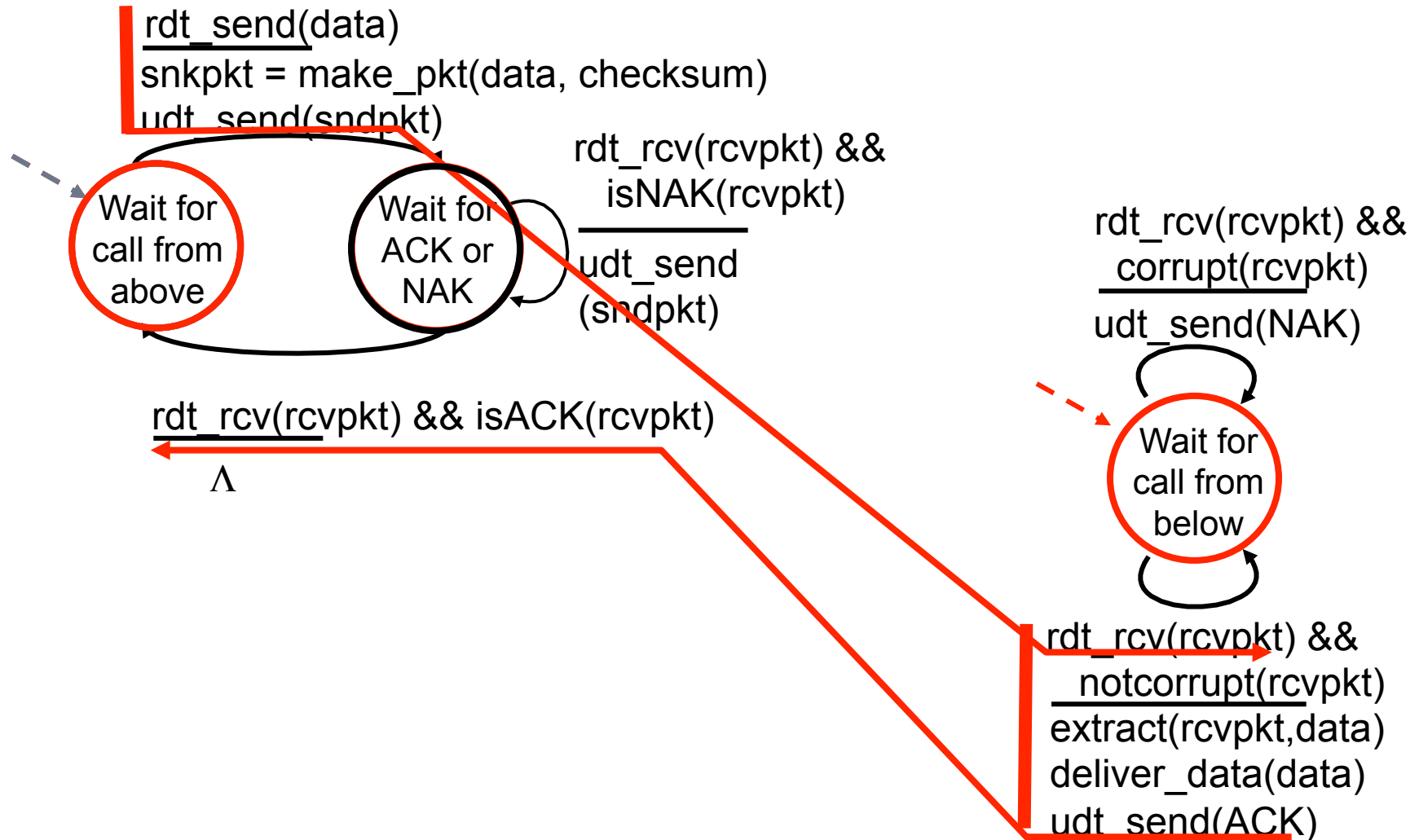


sender

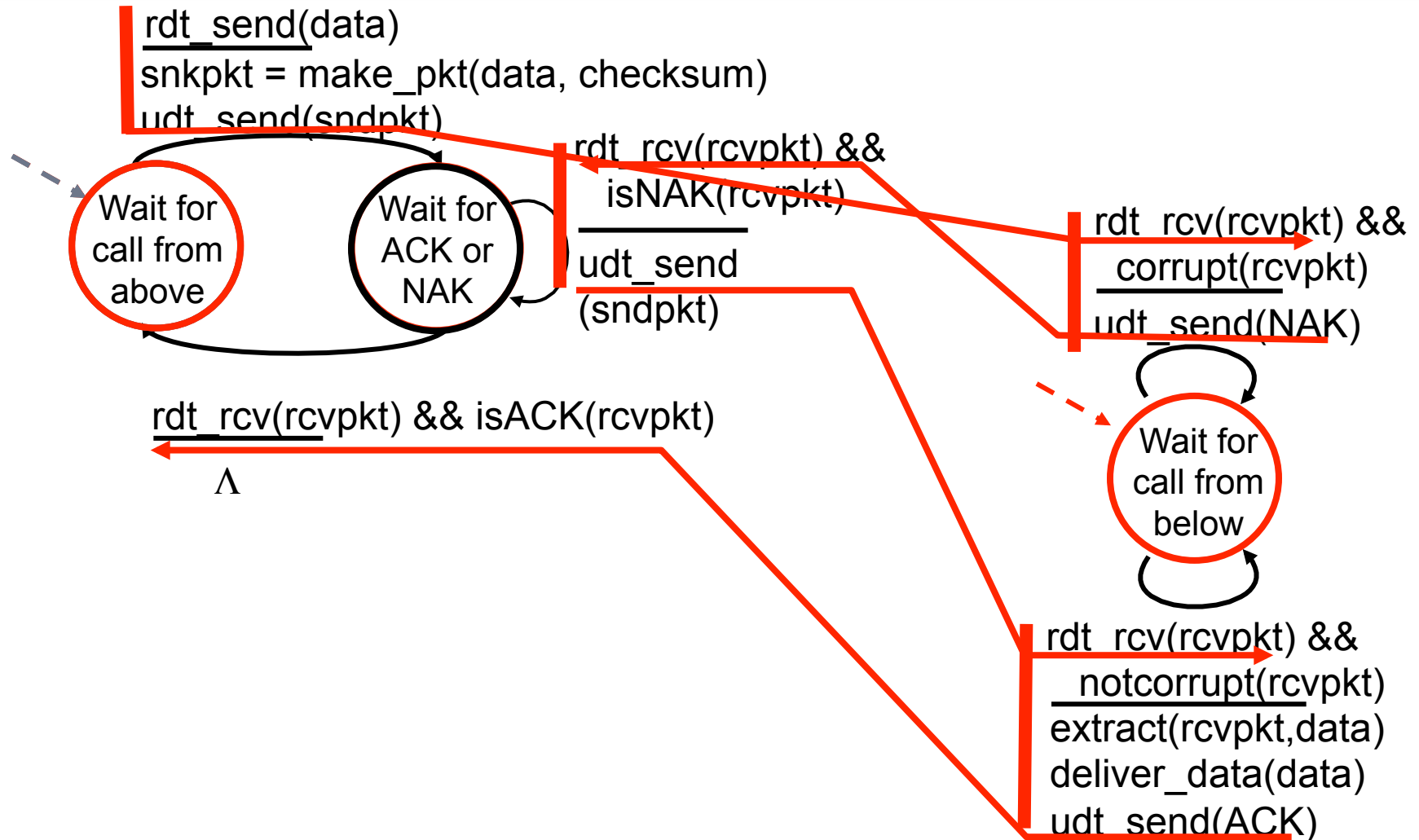
receiver



rdt2.0: Operation with No Errors



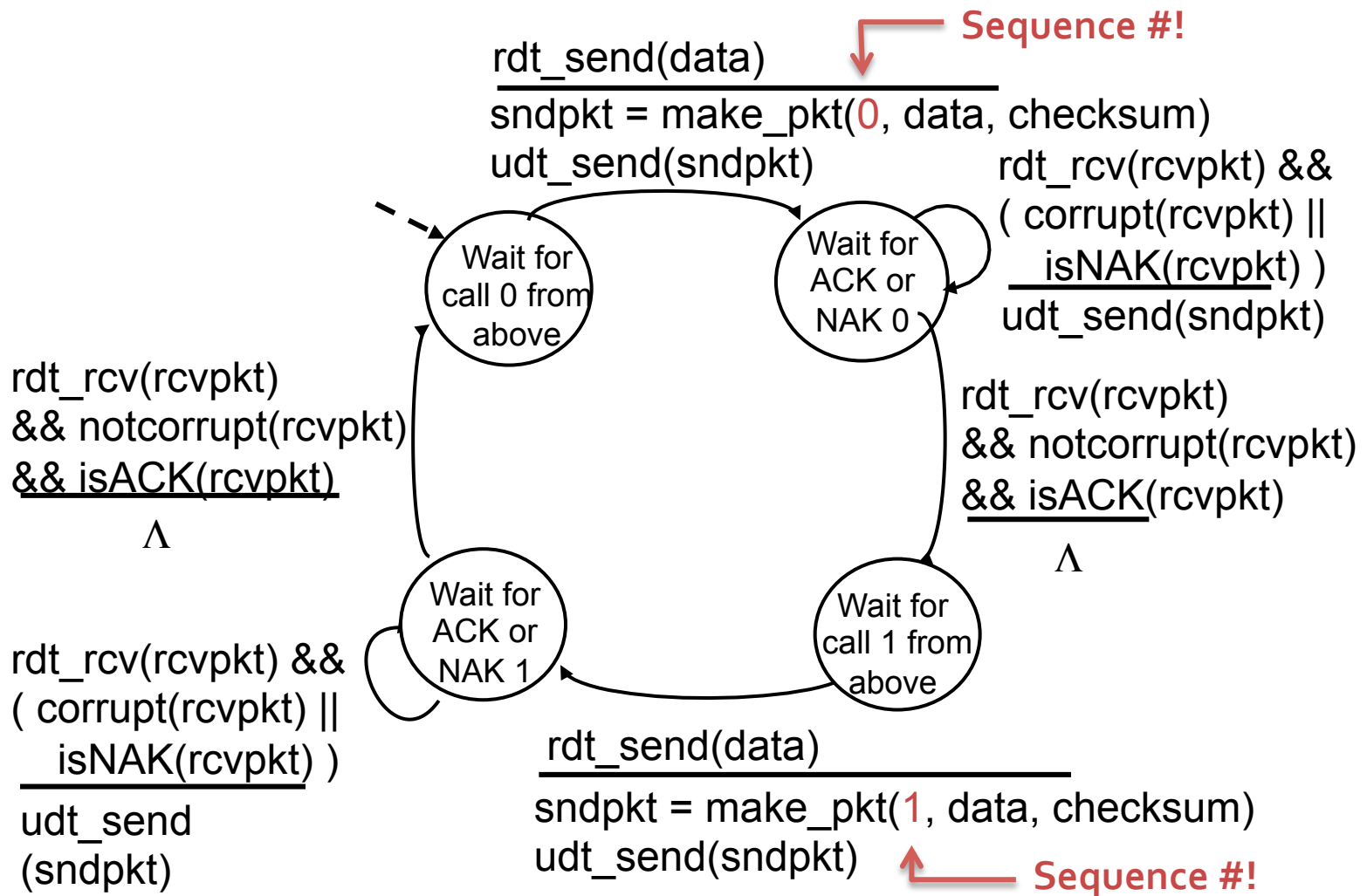
rdt2.0: Error Scenario



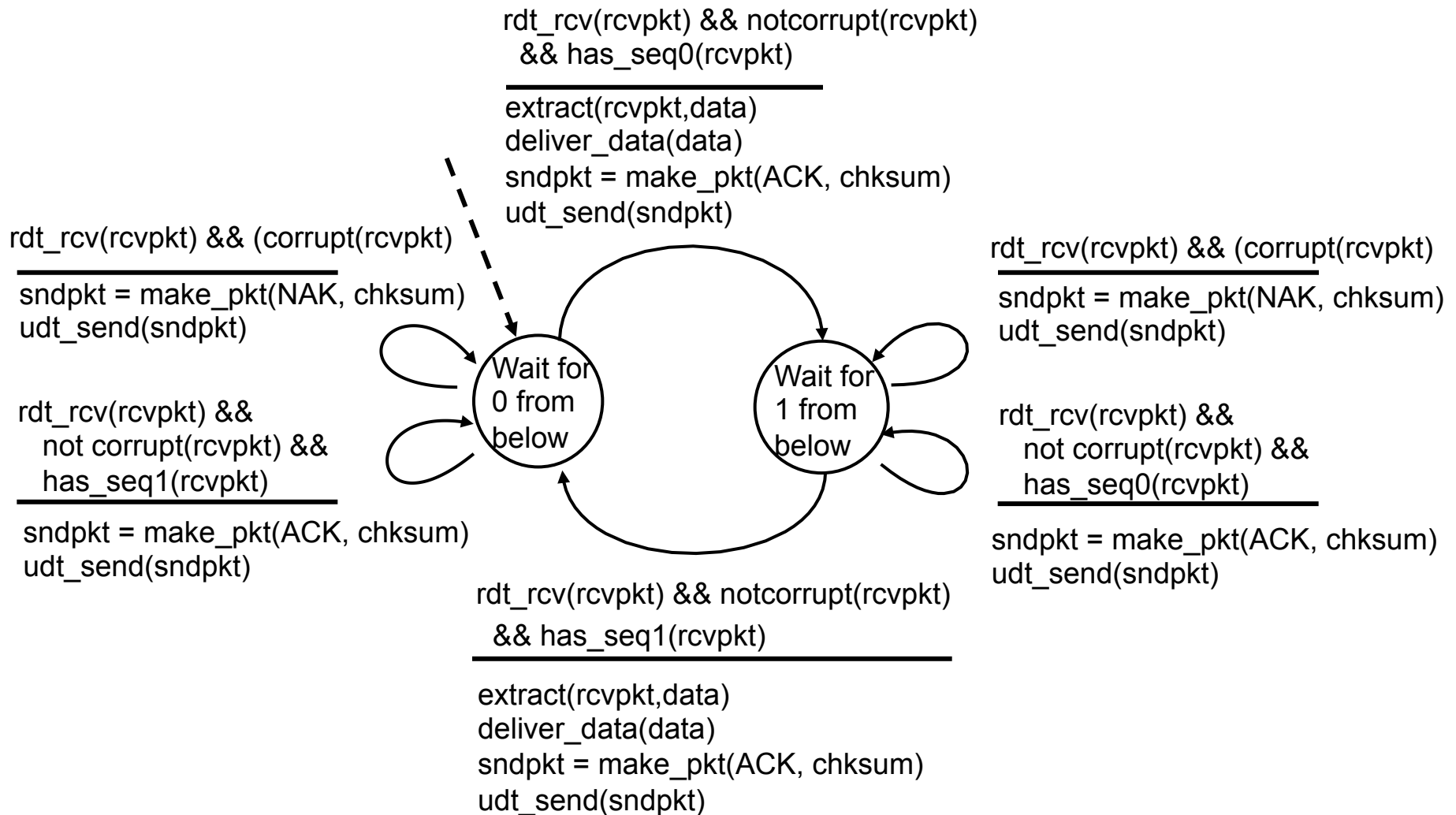
rdt2.0 has a Fatal Flaw!

- 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

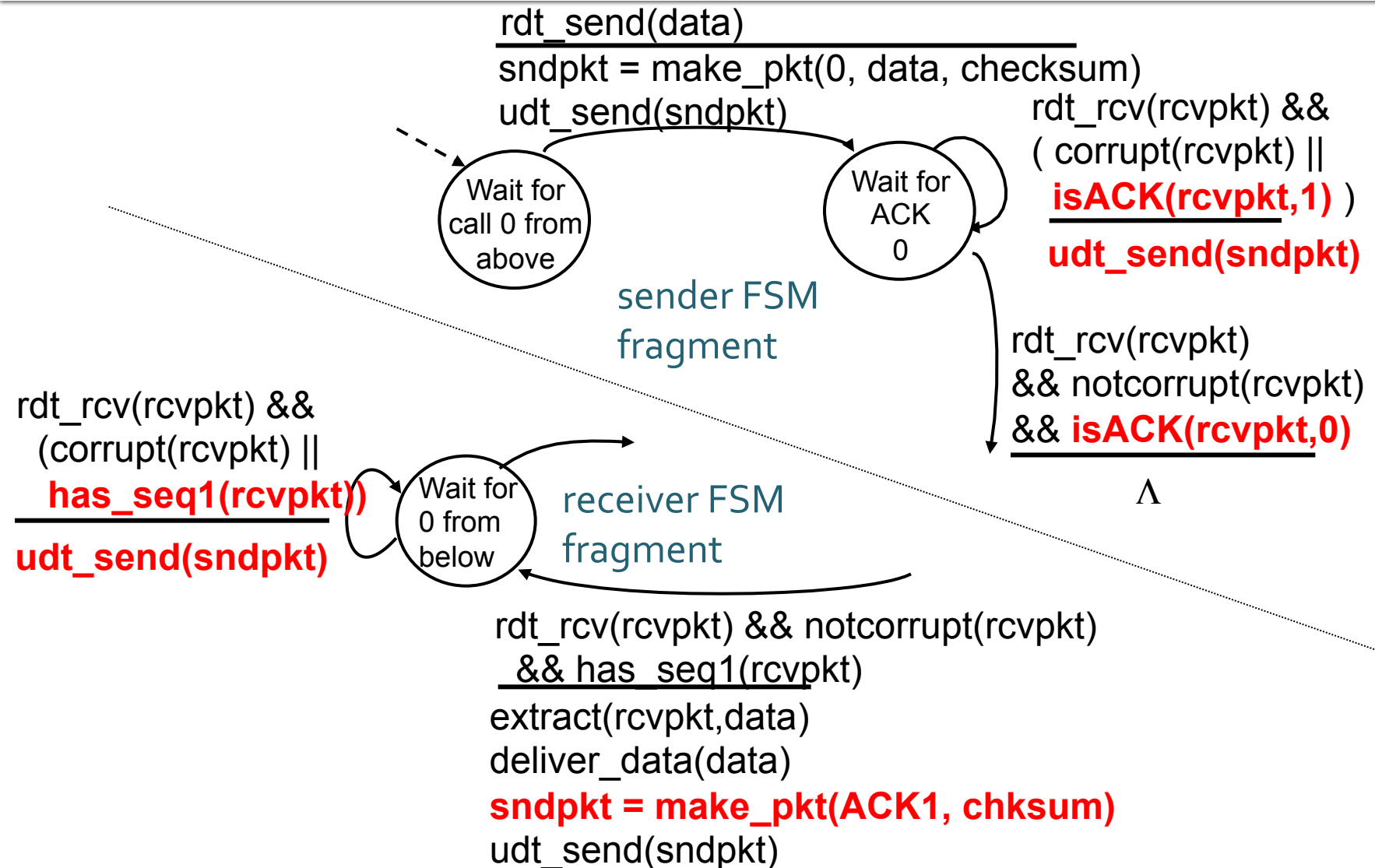
Receiver:

- 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

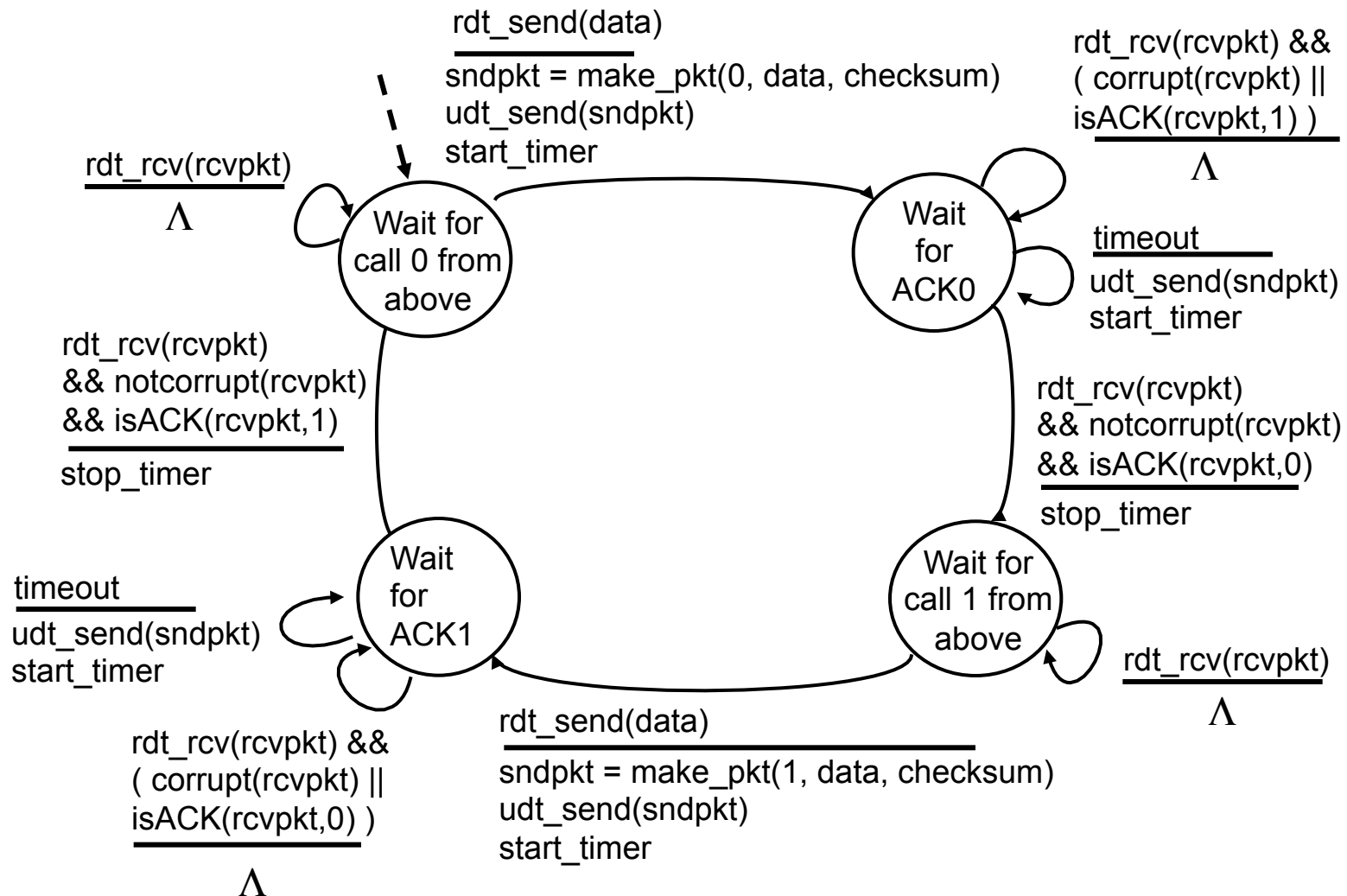
rdt2.2: Partial Sender and Receiver



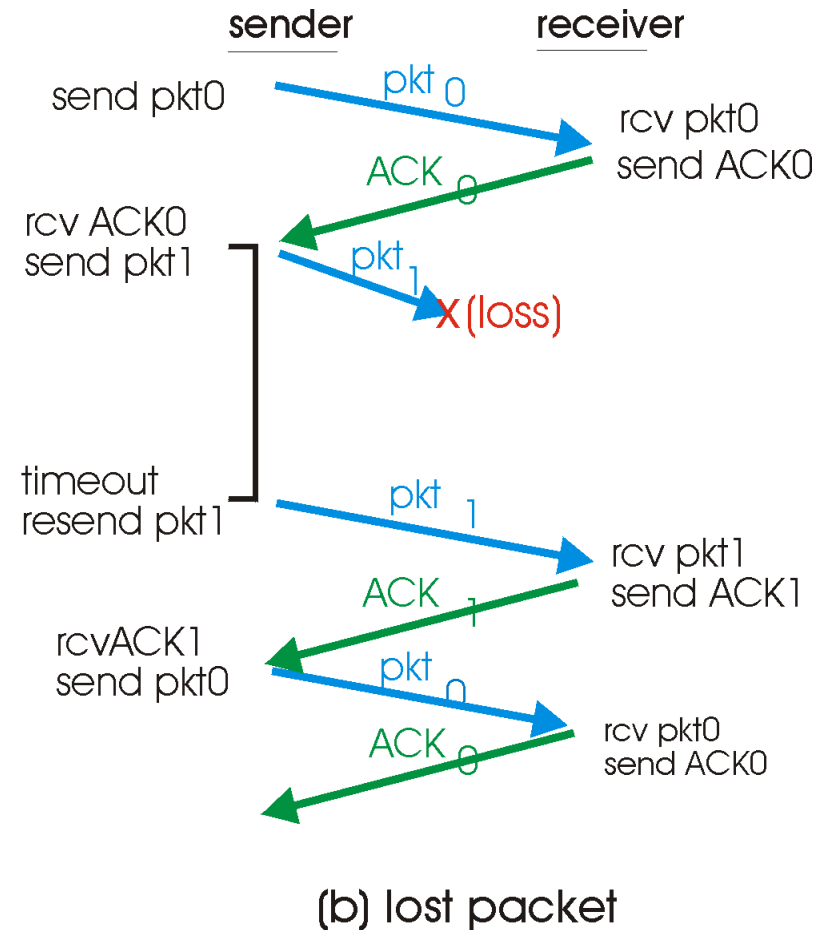
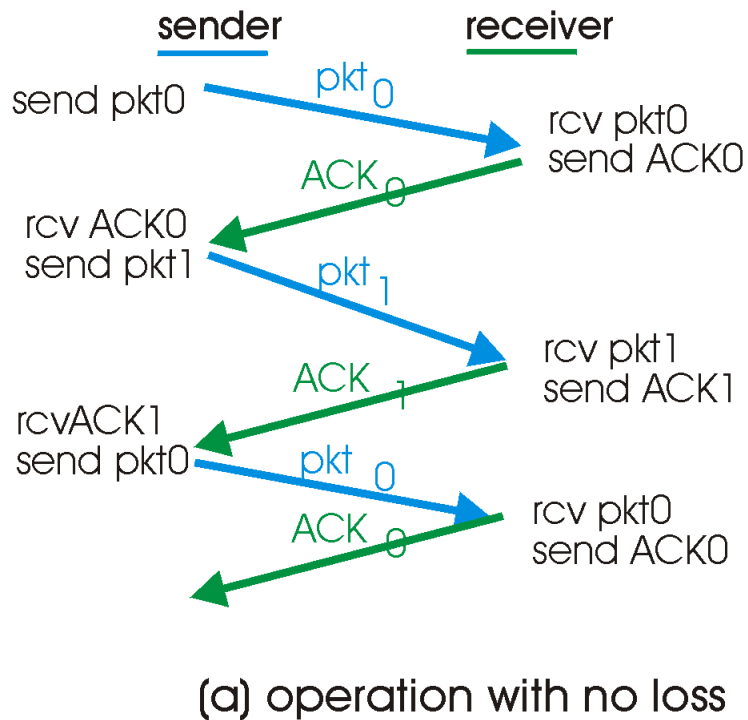
rdt3.0: Channels with Errors and Loss

- 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

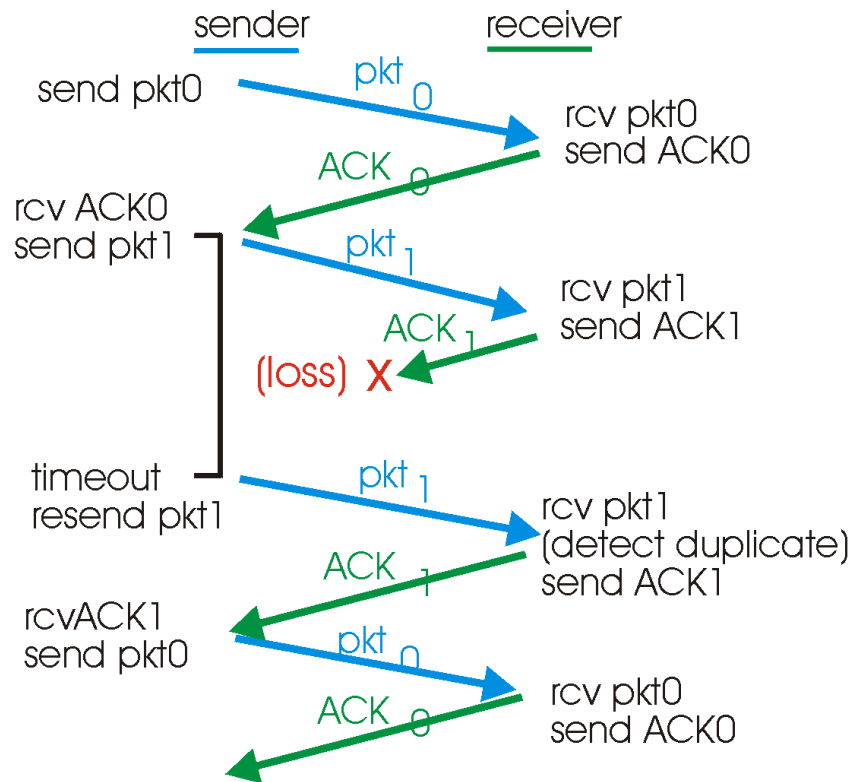
rdt3.0 Sender



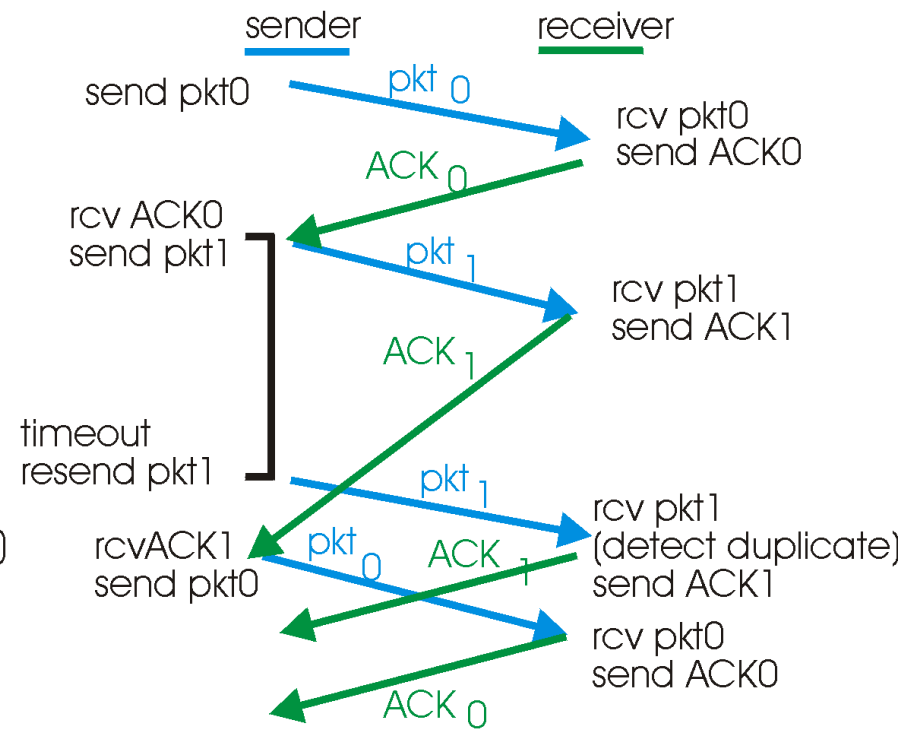
rdt3.0 in Action



rdt3.0 in Action



(c) lost ACK



(d) premature timeout

Performance of rdt3.0

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

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

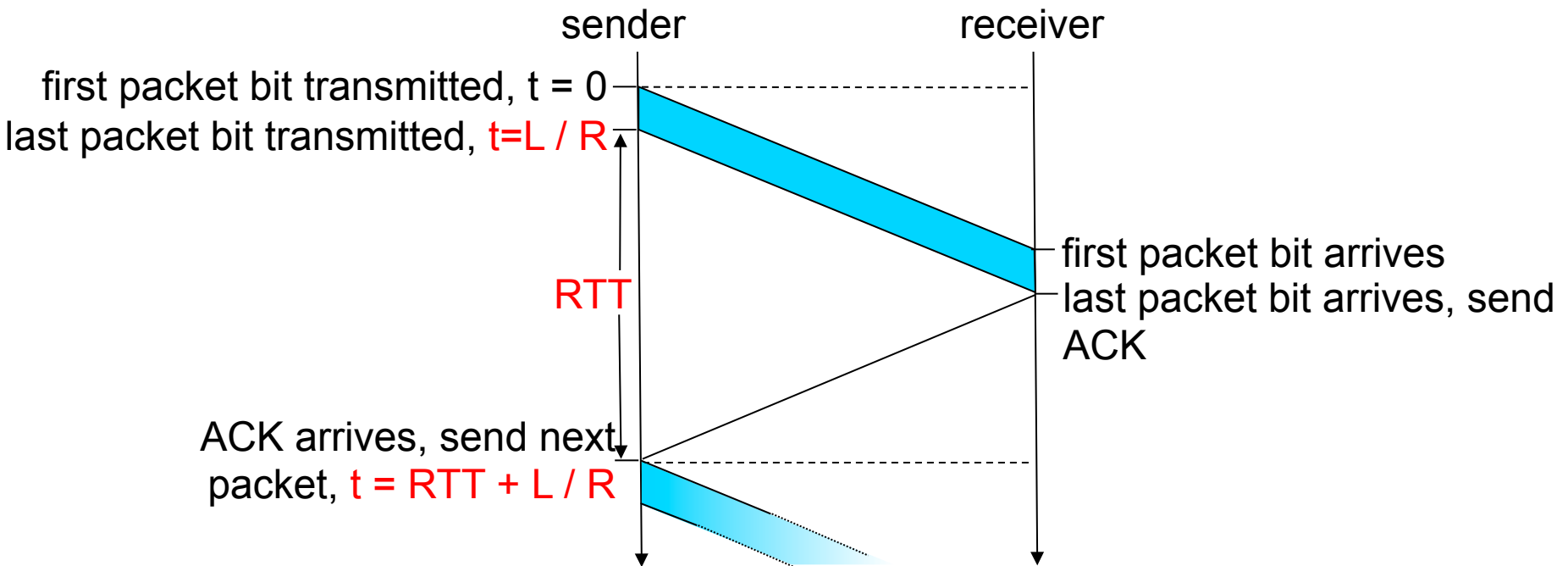
How long it takes to push packet out onto wire

- 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

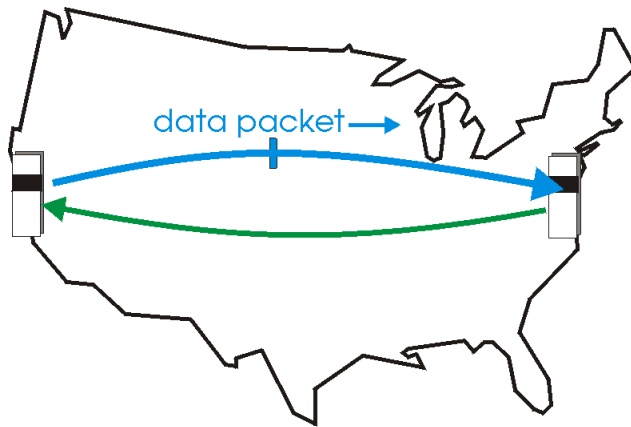


$$U_{\text{sender}} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

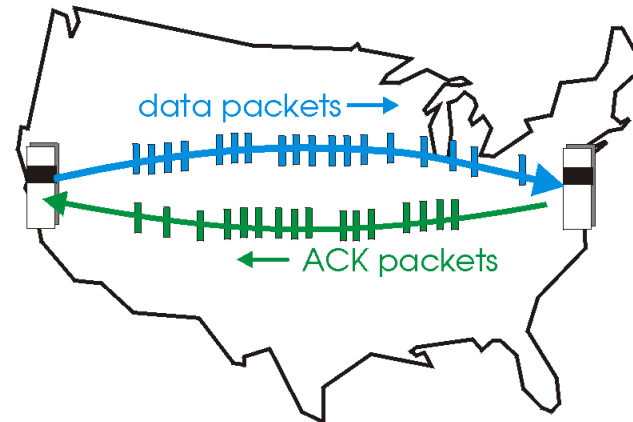
Pipelined protocols

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

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

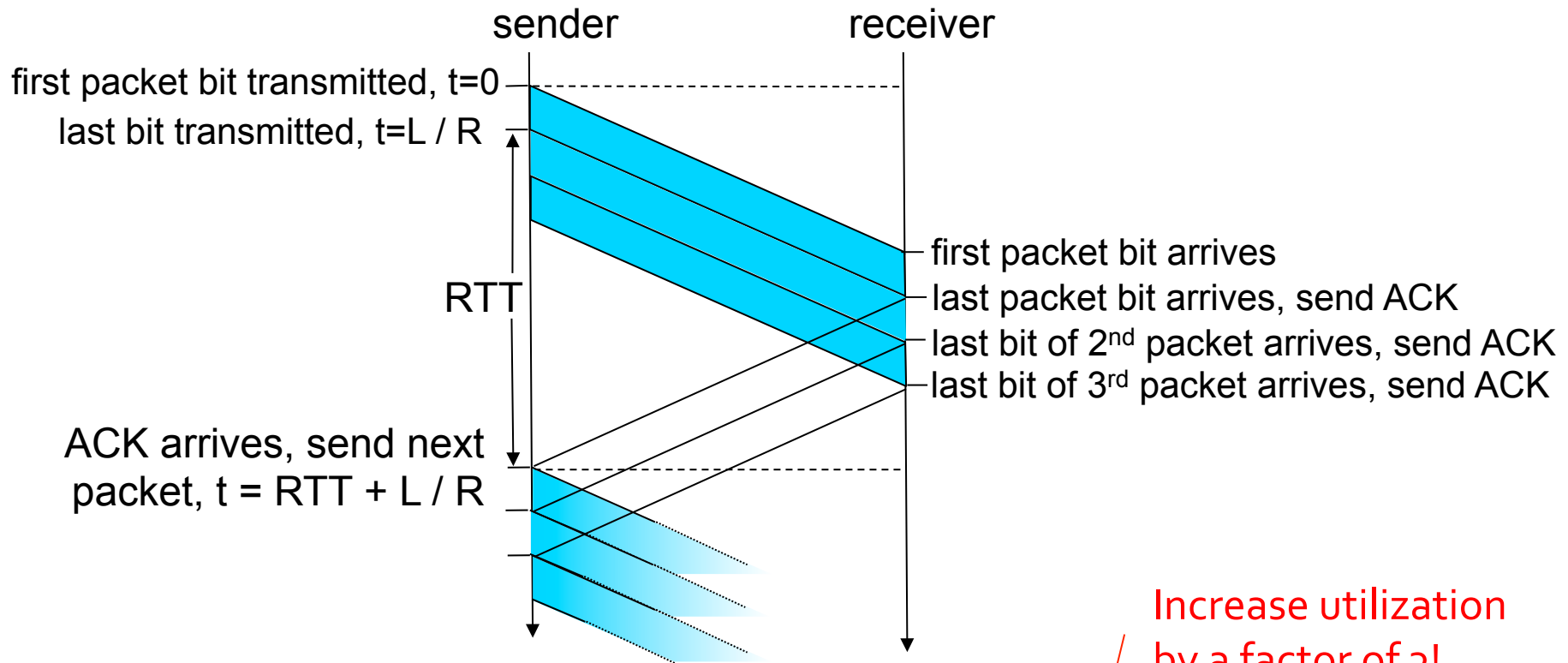


(a) a stop-and-wait protocol in operation



(b) a pipelined protocol in operation

Pipelining: Increased Utilization



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

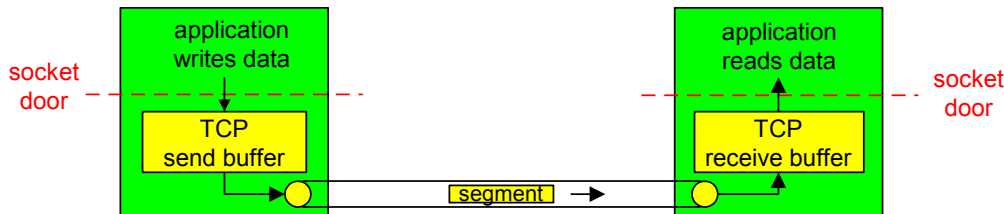
Increase utilization
by a factor of 3!

TCP – Transmission Control Protocol

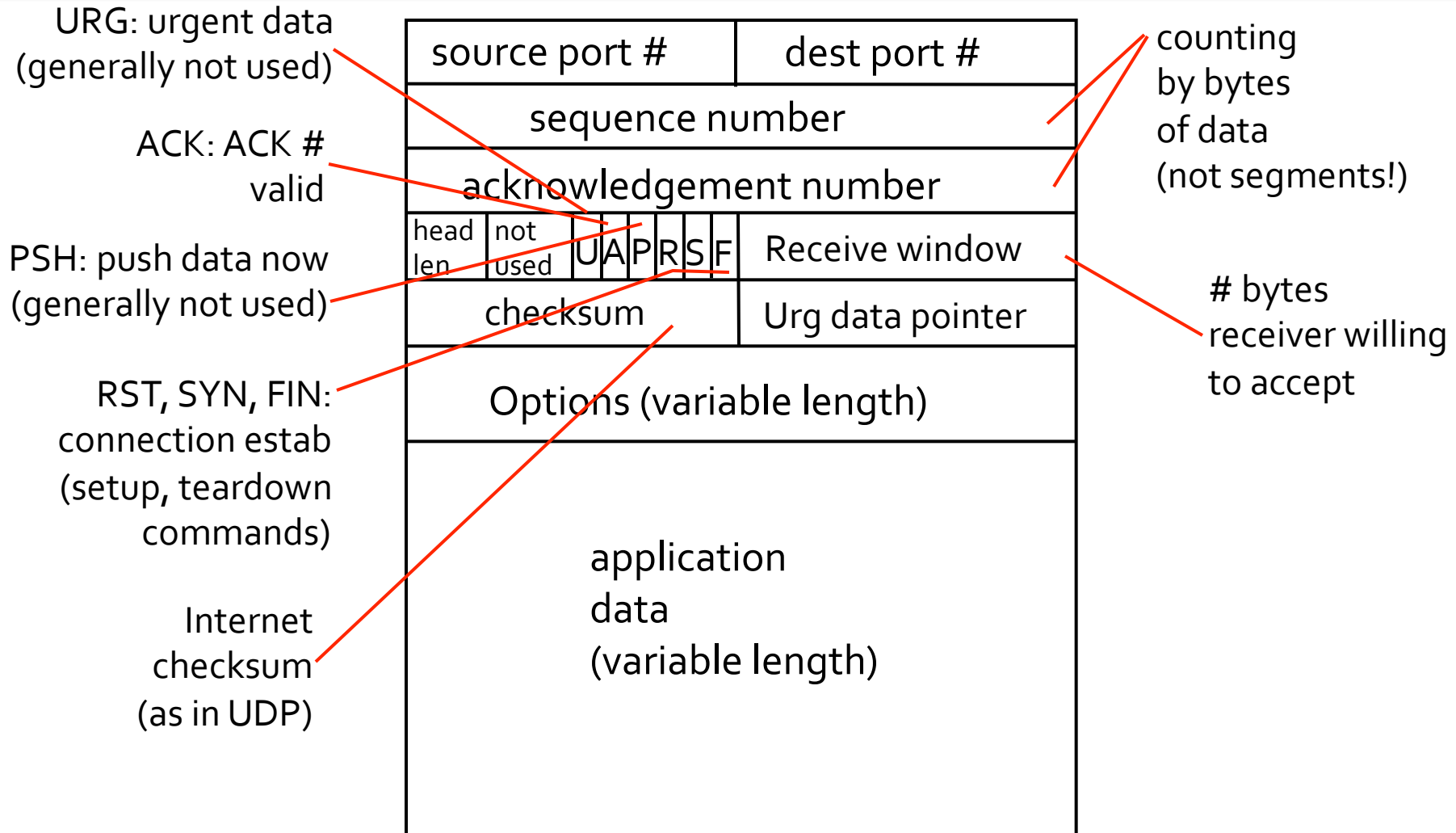
TCP: Overview

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
- ***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:

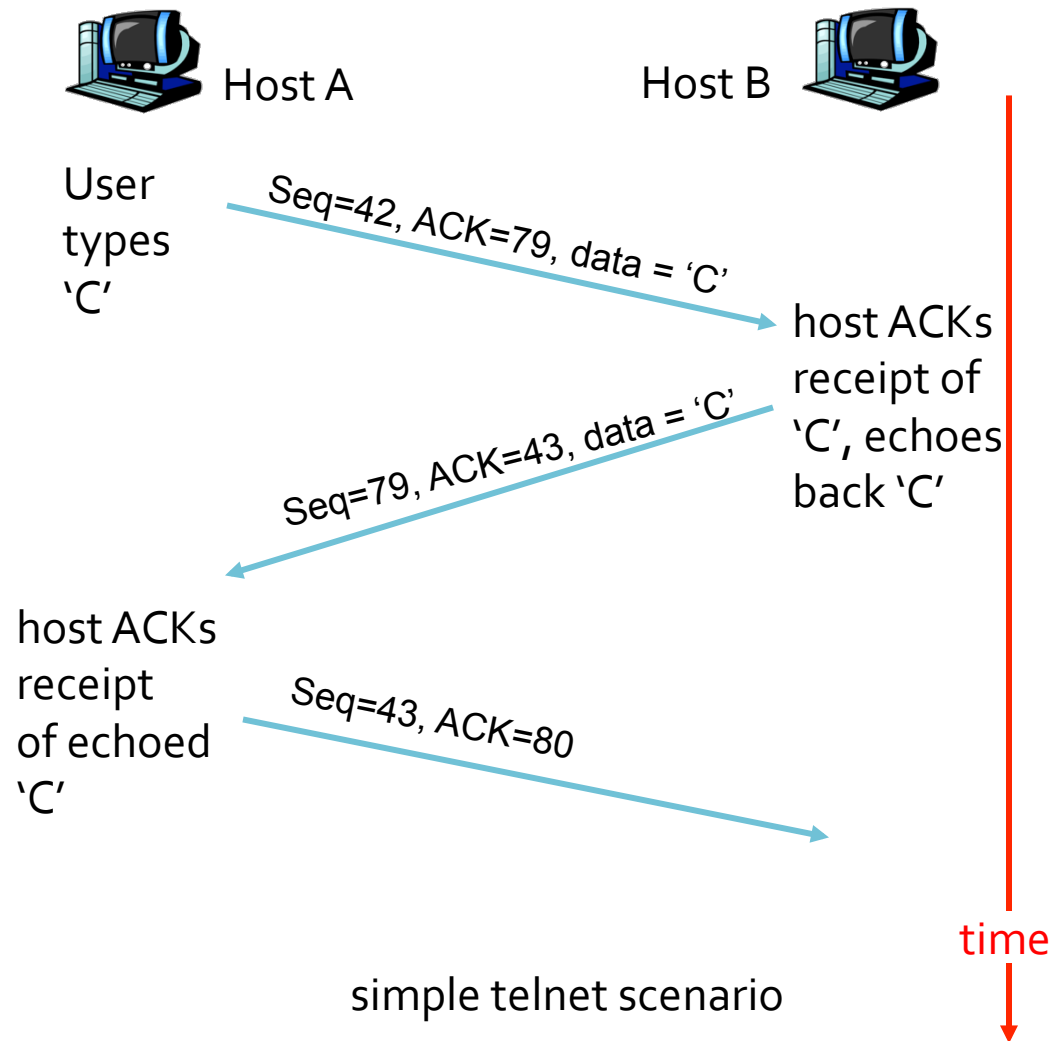
- Byte stream “number” of first byte in segment’s data

ACKs:

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

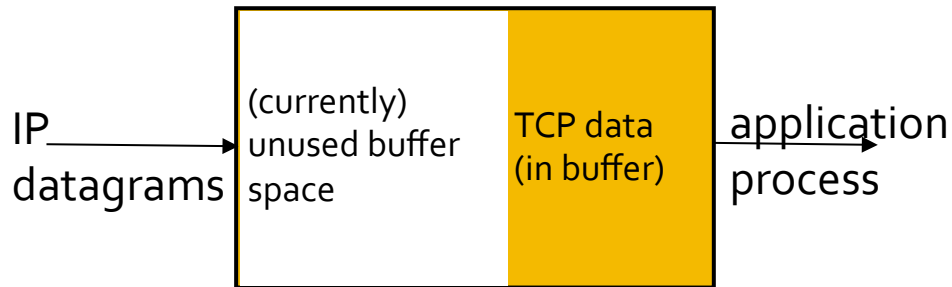
How does receiver handle out-of-order segments?

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



TCP Flow Control

- Receive side of TCP connection has a receive buffer:



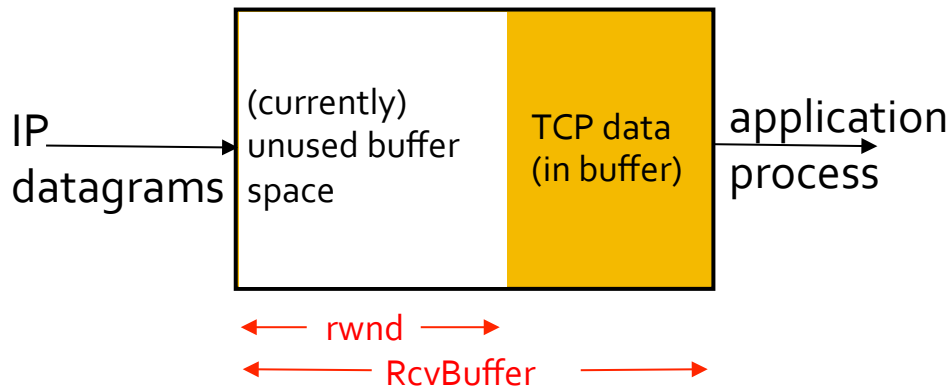
- Application process may be slow at reading from buffer
 - What if buffer fills up?

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 = $rwnd$
= $RcvBuffer - [LastByteRcvd - LastByteRead]$

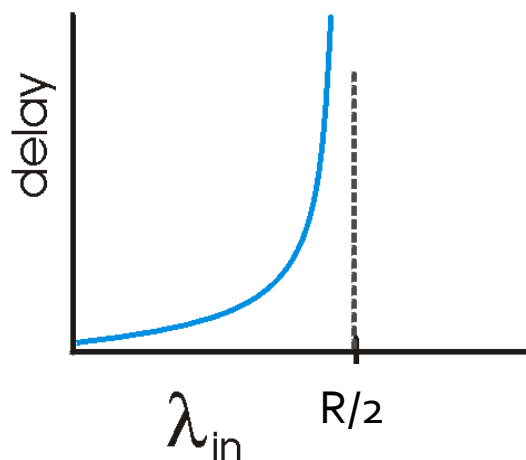
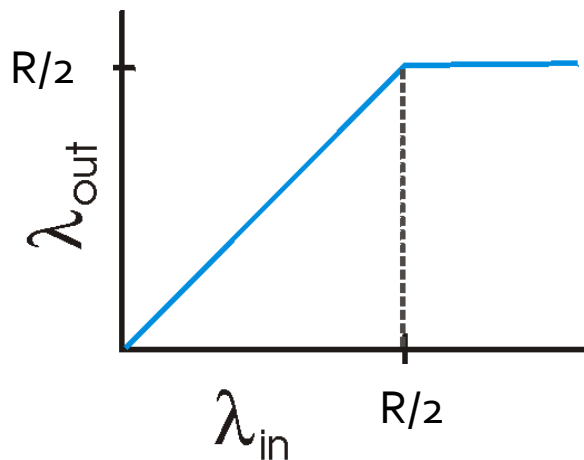
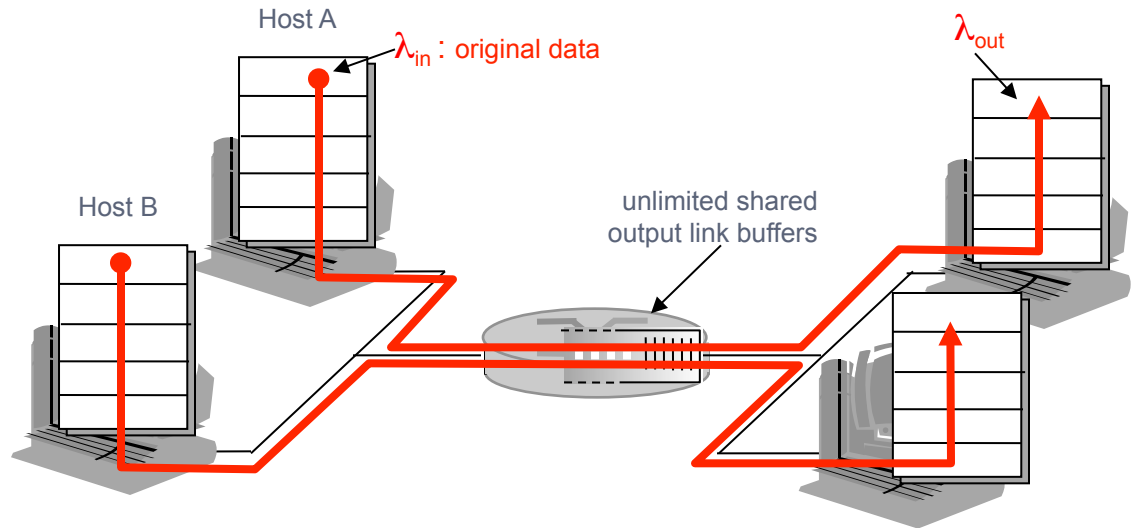
- Receiver notifies sender of unused buffer space
 - Segment header includes the $rwnd$ value
- Sender limits # of unACKed bytes to $rwnd$
 - Guarantees receiver's buffer doesn't overflow

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

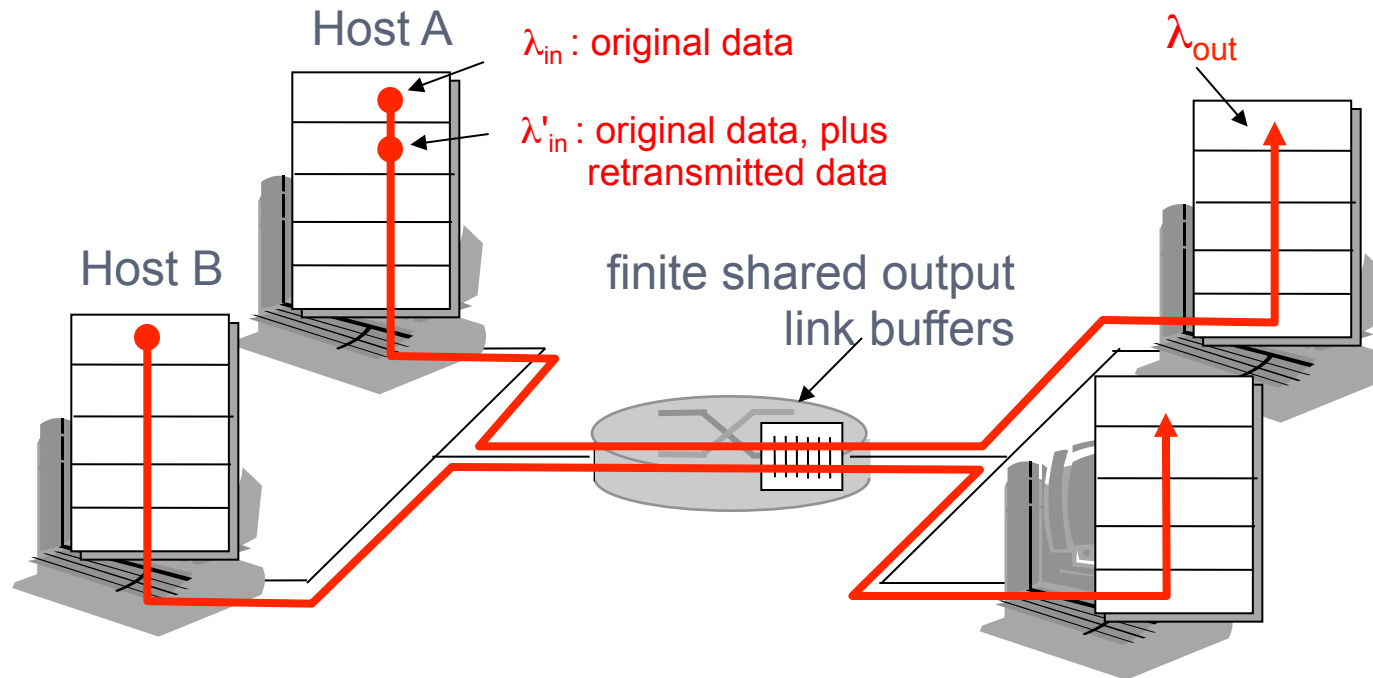
- Two senders, two receivers
- One router, **infinite** buffers
- No retransmission
- Link BW of R



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

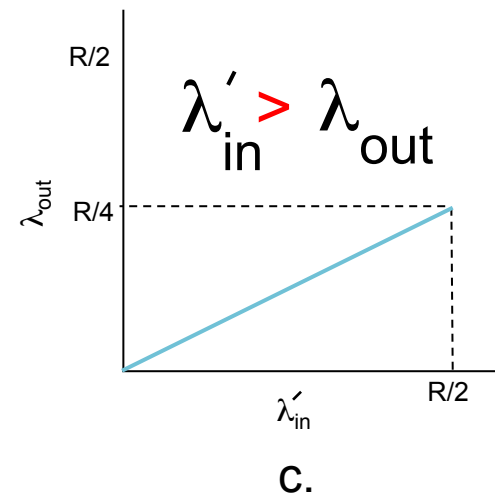
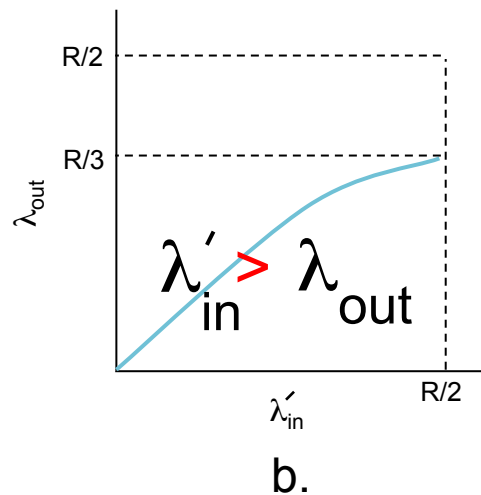
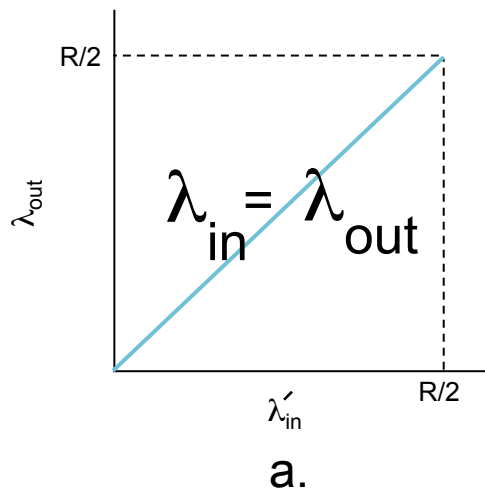
Causes/costs of Congestion: Scenario 2

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



Causes/costs of Congestion: Scenario 2

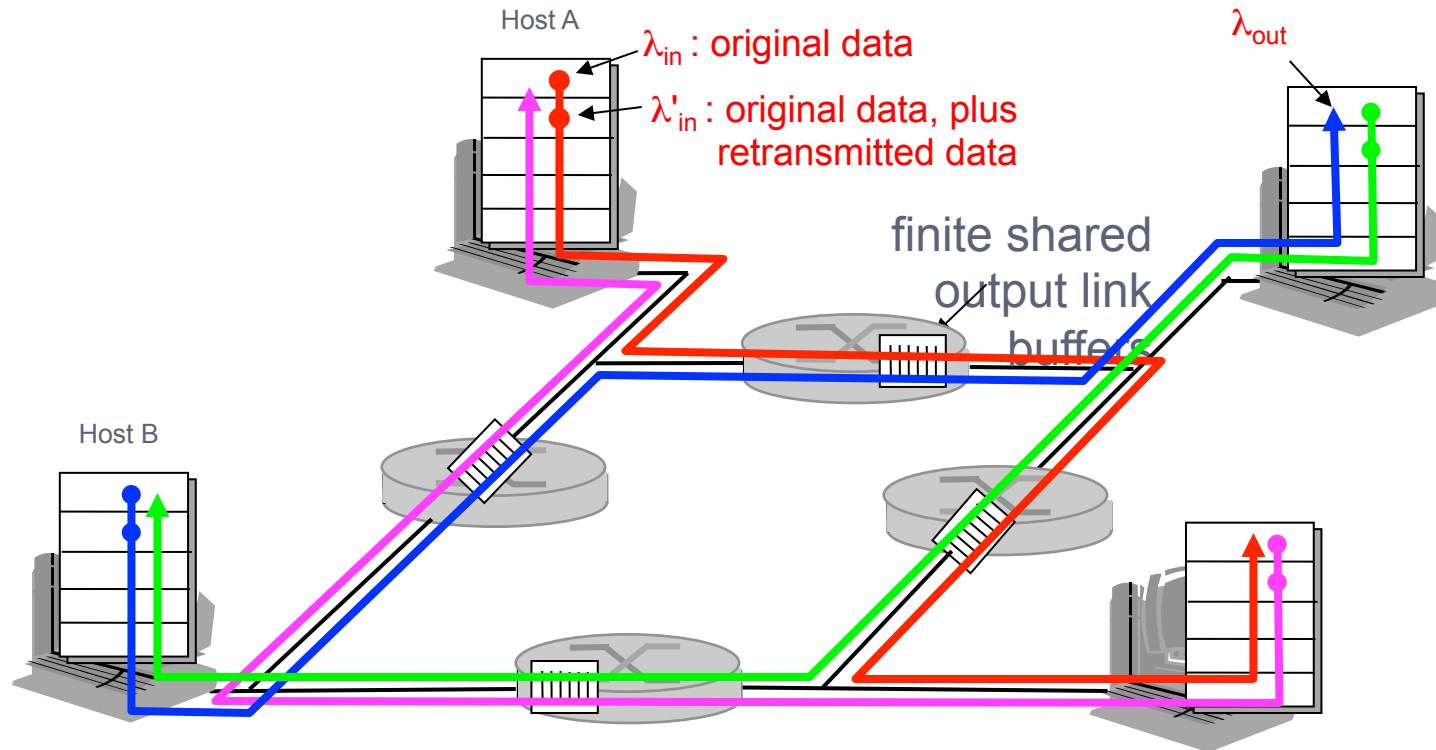
- Case a: Sender only transmits when it knows buffer space is available in router (unrealistic)
- Case b: Sender retransmits only when packet is known to be lost
 - New cost of congestion: More sender work (retrans) for given "goodput"
- Case c: Assume sender also retransmits when a packet is delayed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



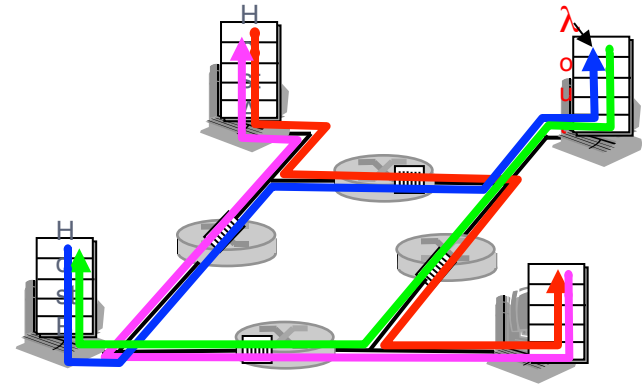
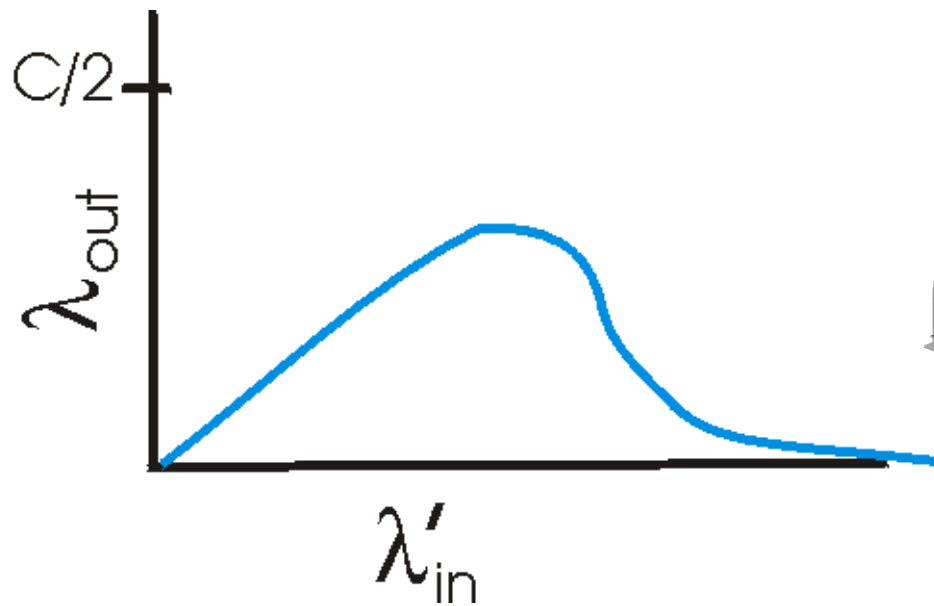
Causes/costs of Congestion: Scenario 3

- 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:

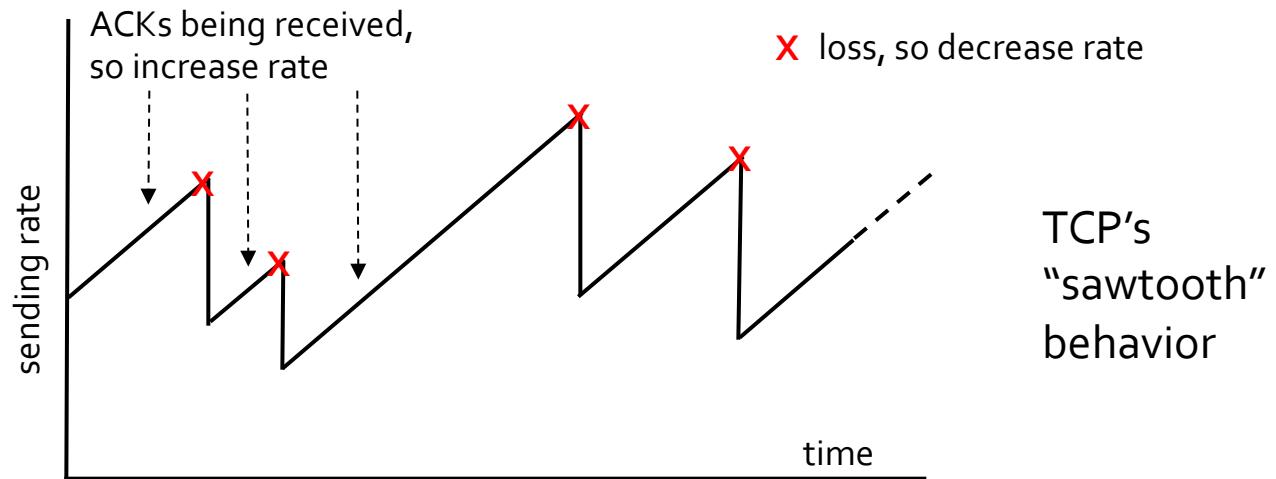
- **End-end congestion control:**
 - No explicit feedback from network
 - Congestion inferred from end-system observed loss, delay
 - **Approach taken by TCP**
- **Network-assisted congestion control:**
 - Routers provide feedback to end systems
 - Single bit indicating congestion (SNA, DECbit, TCP/IP 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?

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