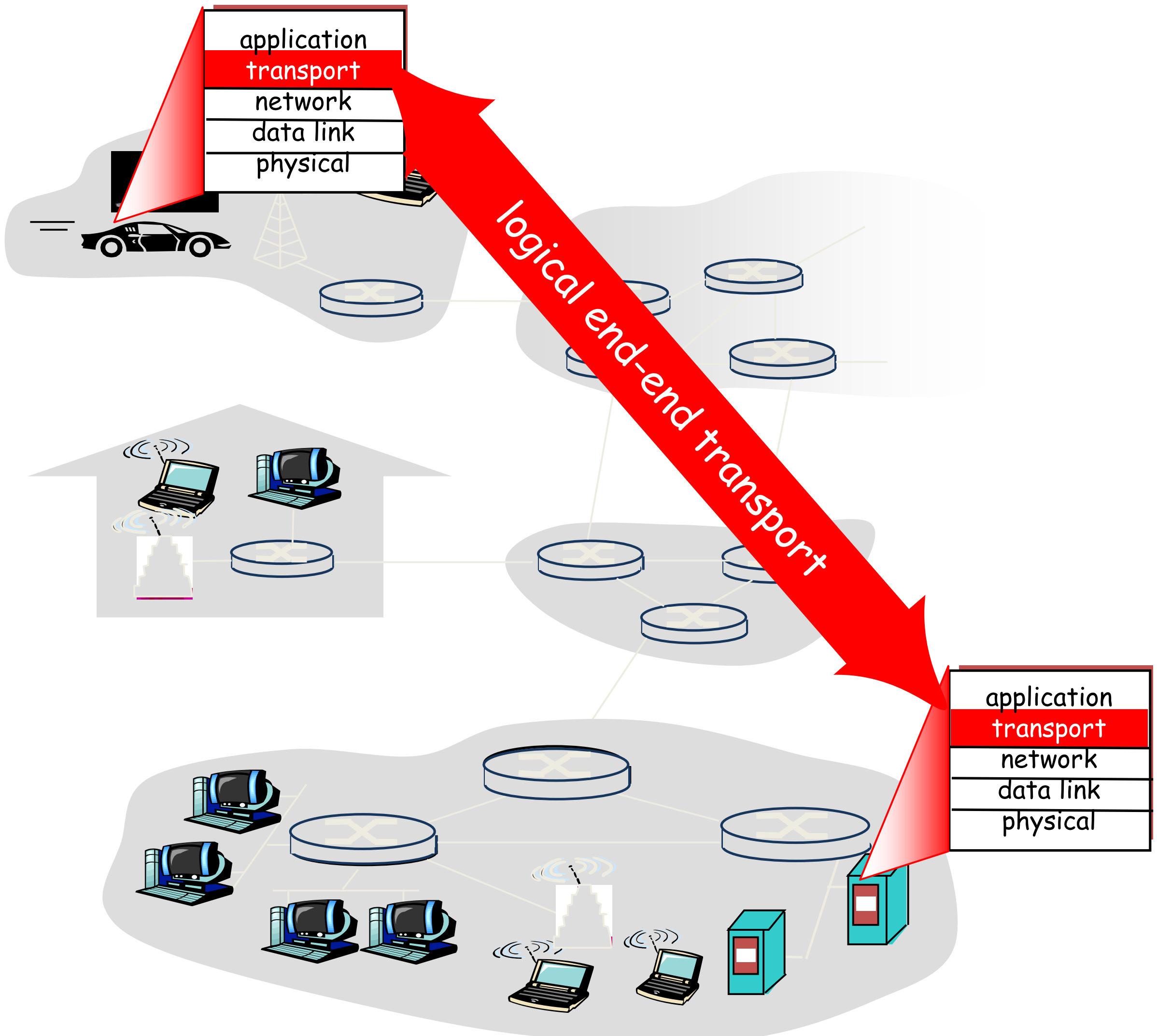
Transport Layer

- The Transport Layer is responsible for providing logical communication between processes. Uses the services of the Network Layer to (try) to transfer data between processes.
- The TL relies on the services of the **Network layer** protocol, so it is limited in the services it can provide.



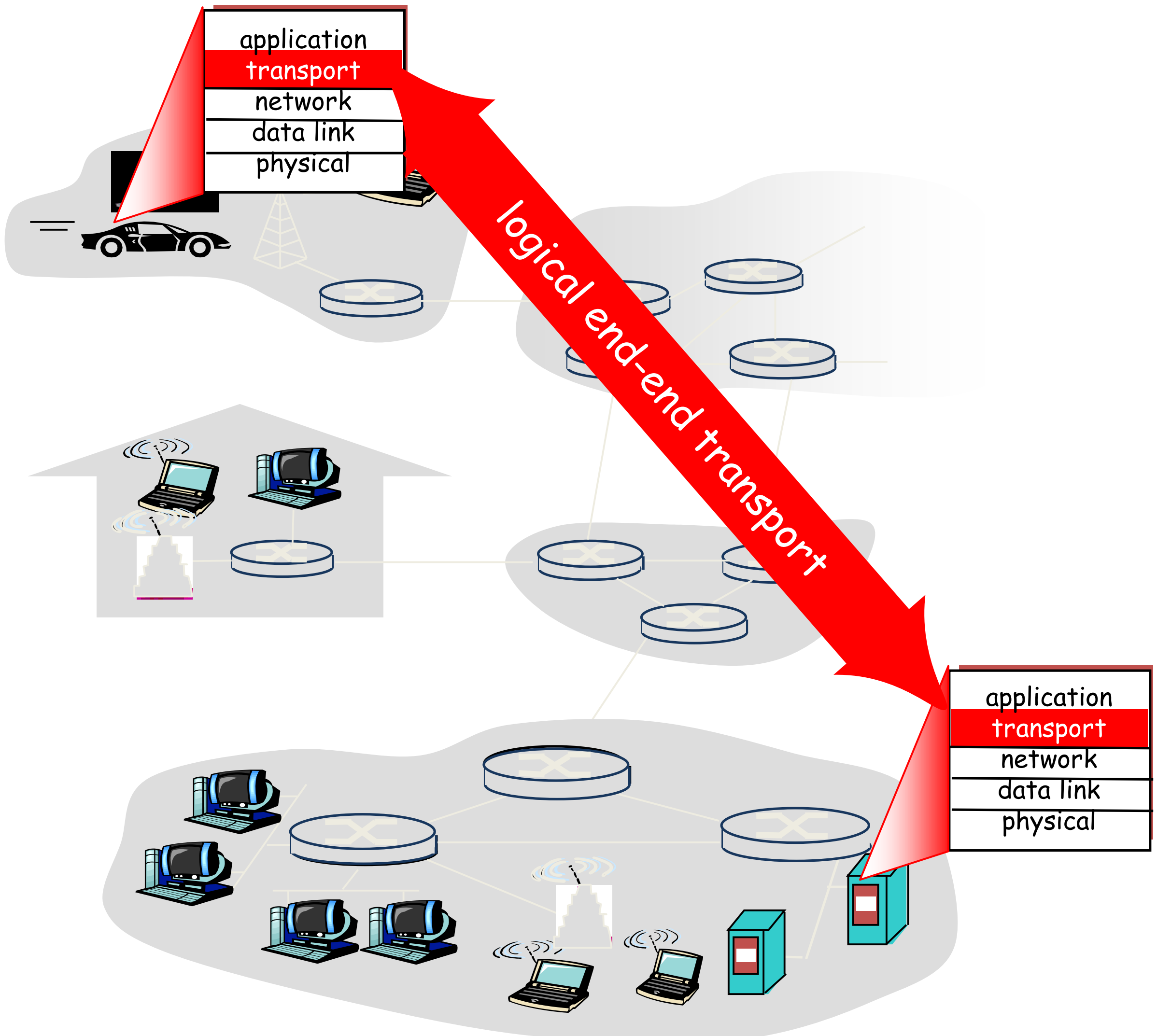
Transport Layer

The Internet Transport Layer offers two services

TCP

UDP

They are different and we will look at them both.



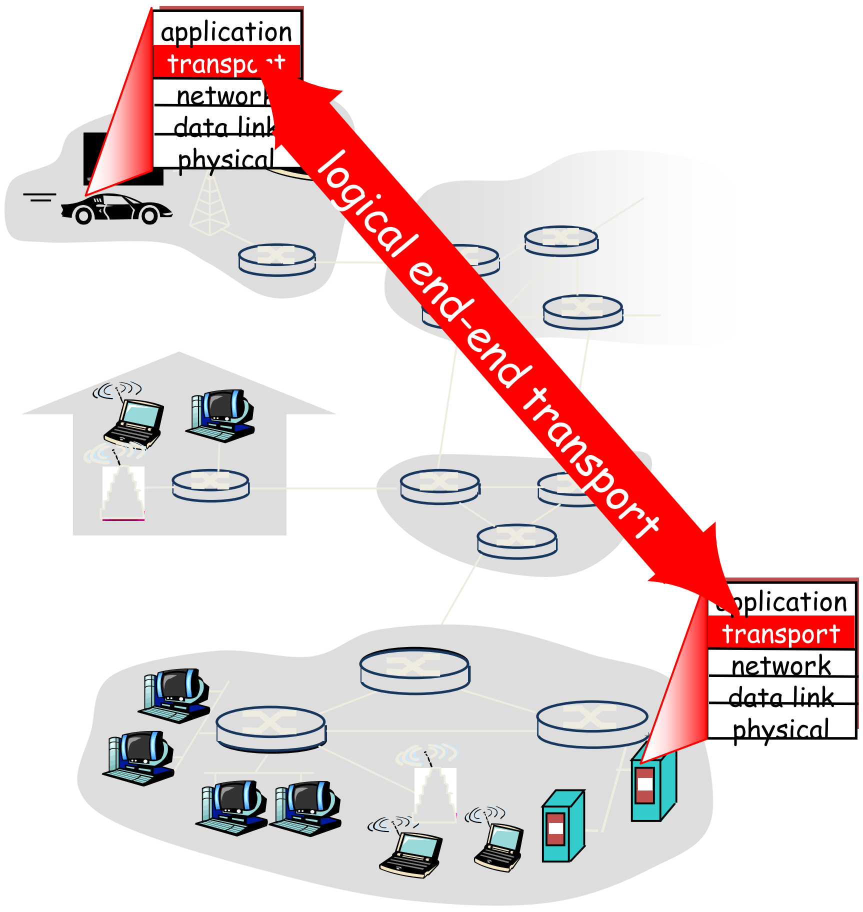
Transport Layer Services

- The ***fundamental service*** of TCP and UDP is to extend the Network Layer packet delivery service provided by IP between ***hosts*** to a delivery service between ***processes***

- ***How?***

TCP and UDP both provide

multiplexing and de-multiplexing of data from several processes



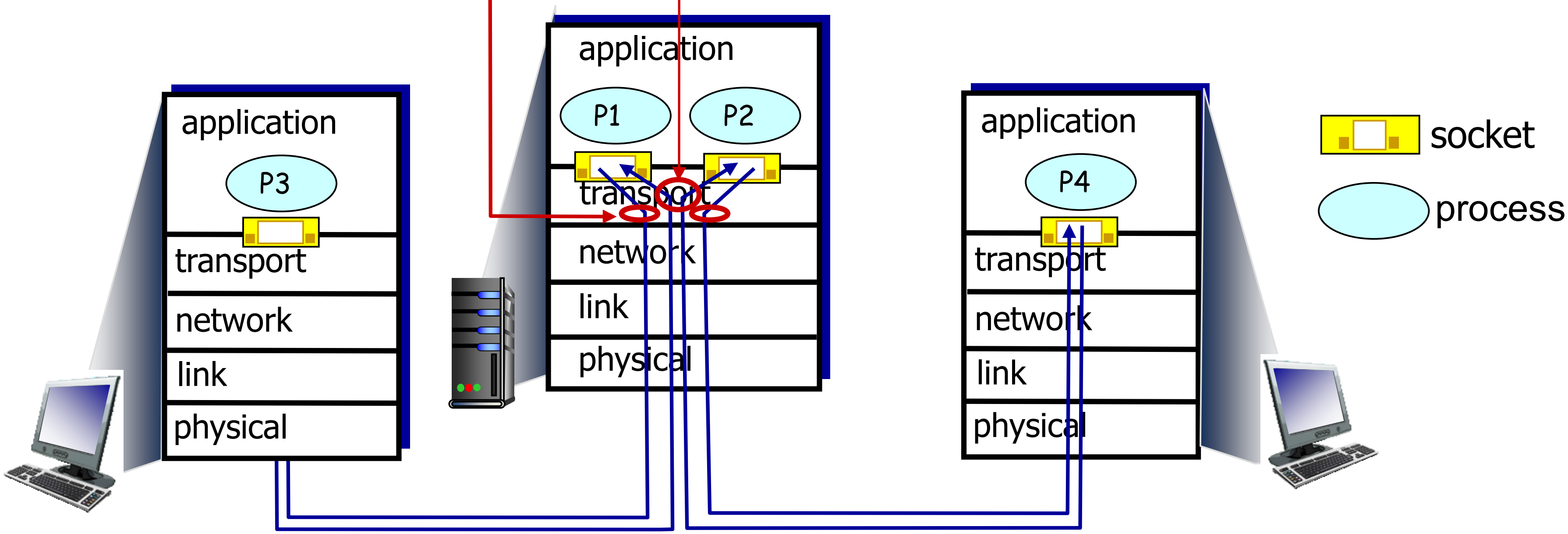
Multiplexing/demultiplexing

multiplexing at sender:

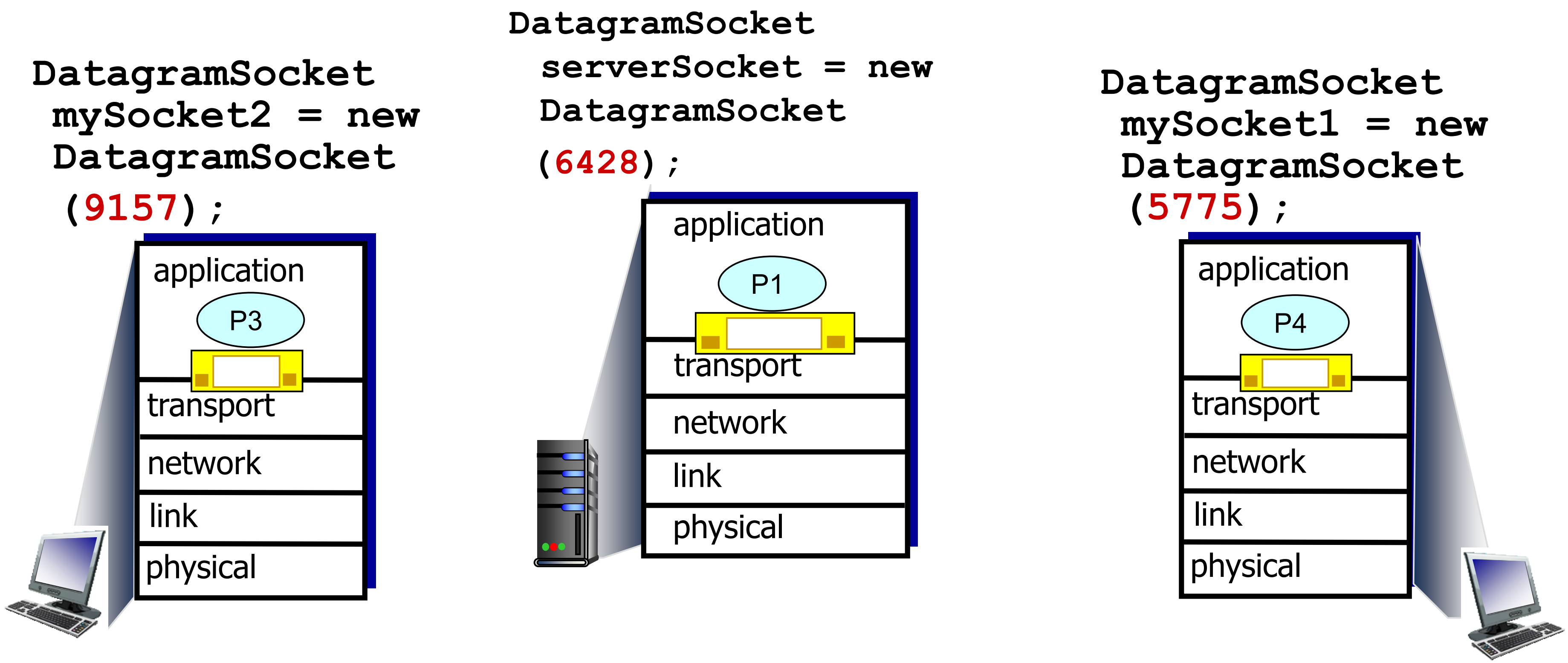
handle data from multiple sockets, add transport header (later used for demultiplexing)

demultiplexing at receiver:

use header info to deliver received segments to correct socket



Connectionless demux: example

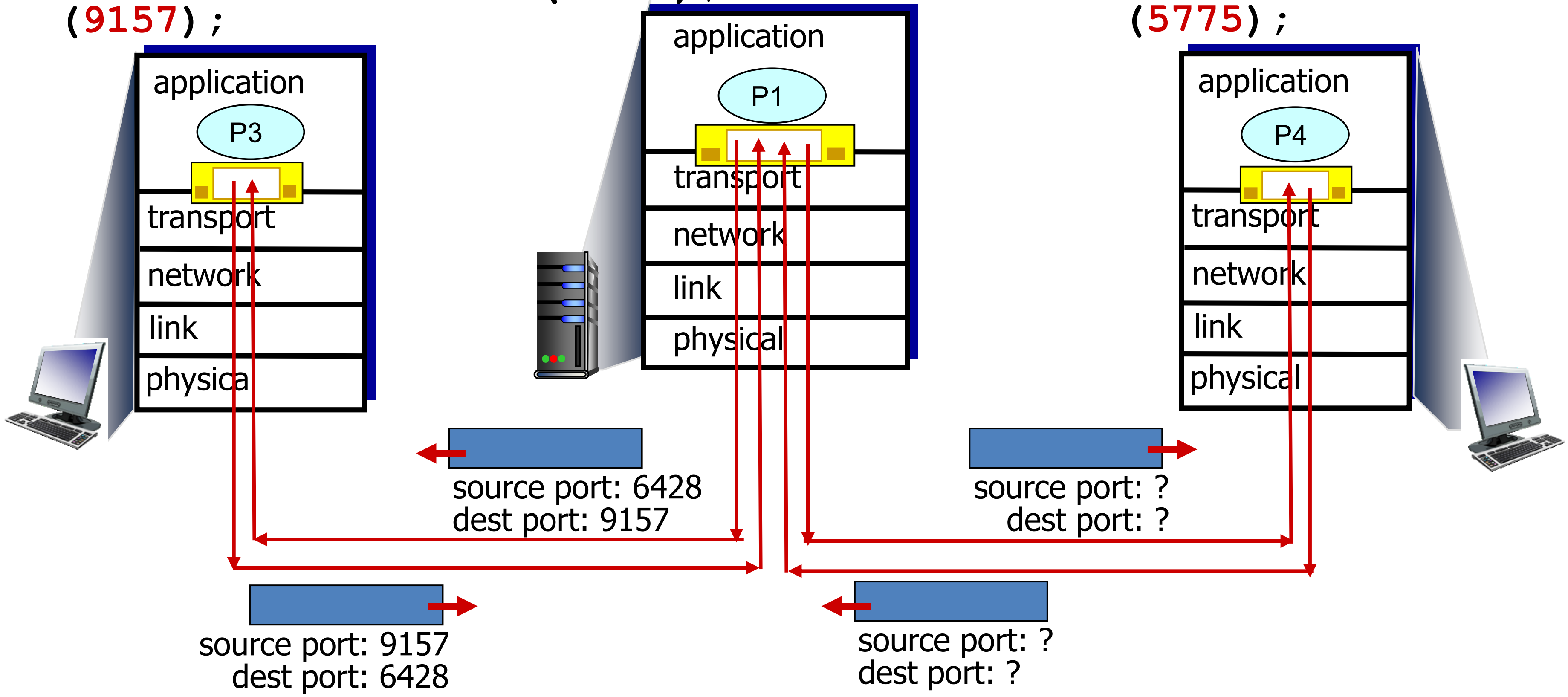


Connectionless demux: example

```
DatagramSocket
 serverSocket = new
 DatagramSocket
 (6428);
```

```
DatagramSocket
 mySocket2 = new
 DatagramSocket
 (9157);
```

```
DatagramSocket
 mySocket1 = new
 DatagramSocket
 (5775);
```



Connectionless demultiplexing

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

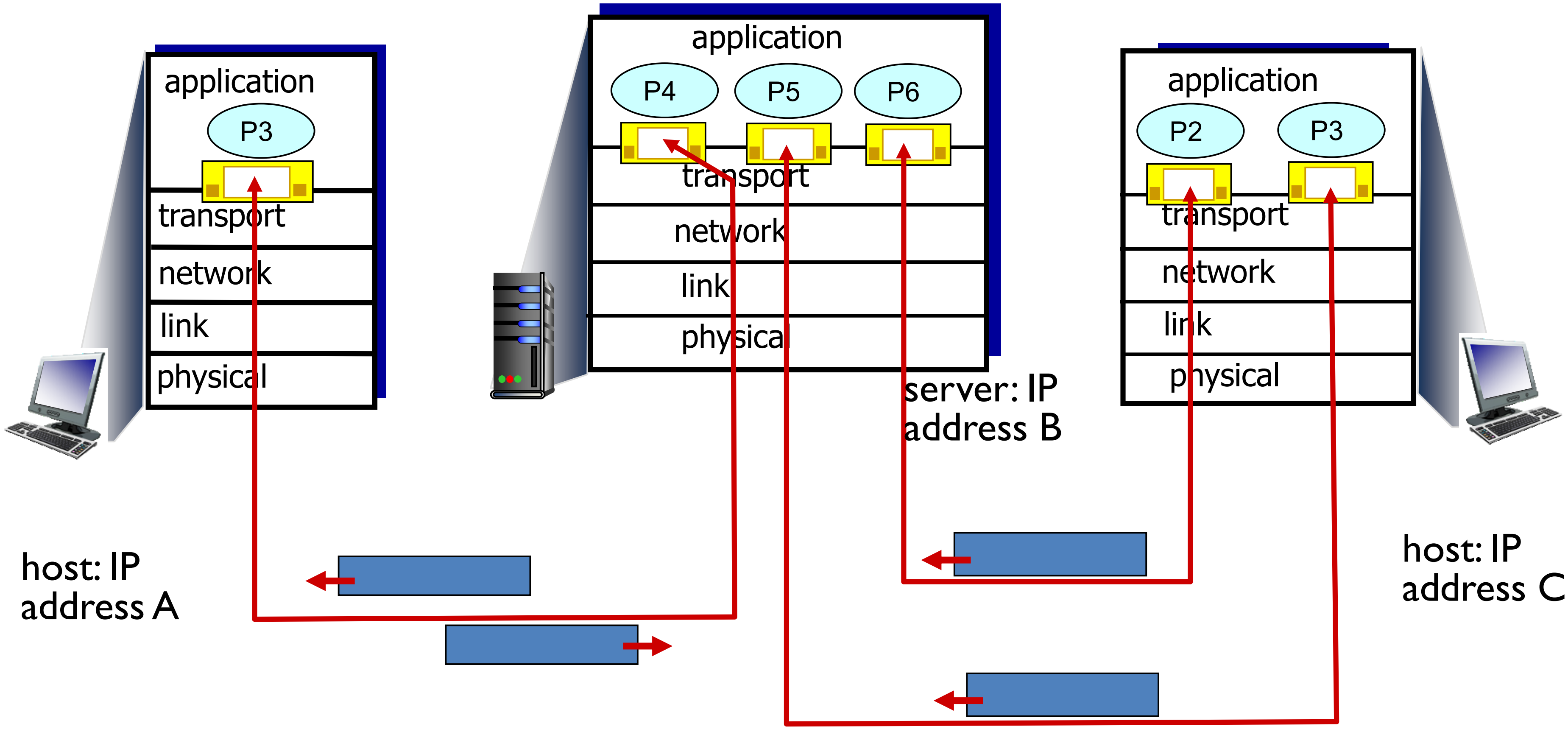
```
DatagramSocket mySocket1
= new DatagramSocket(12534);
```

- *recall:* when creating datagram to send into UDP socket, must specify
  - destination IP address
  - destination port #

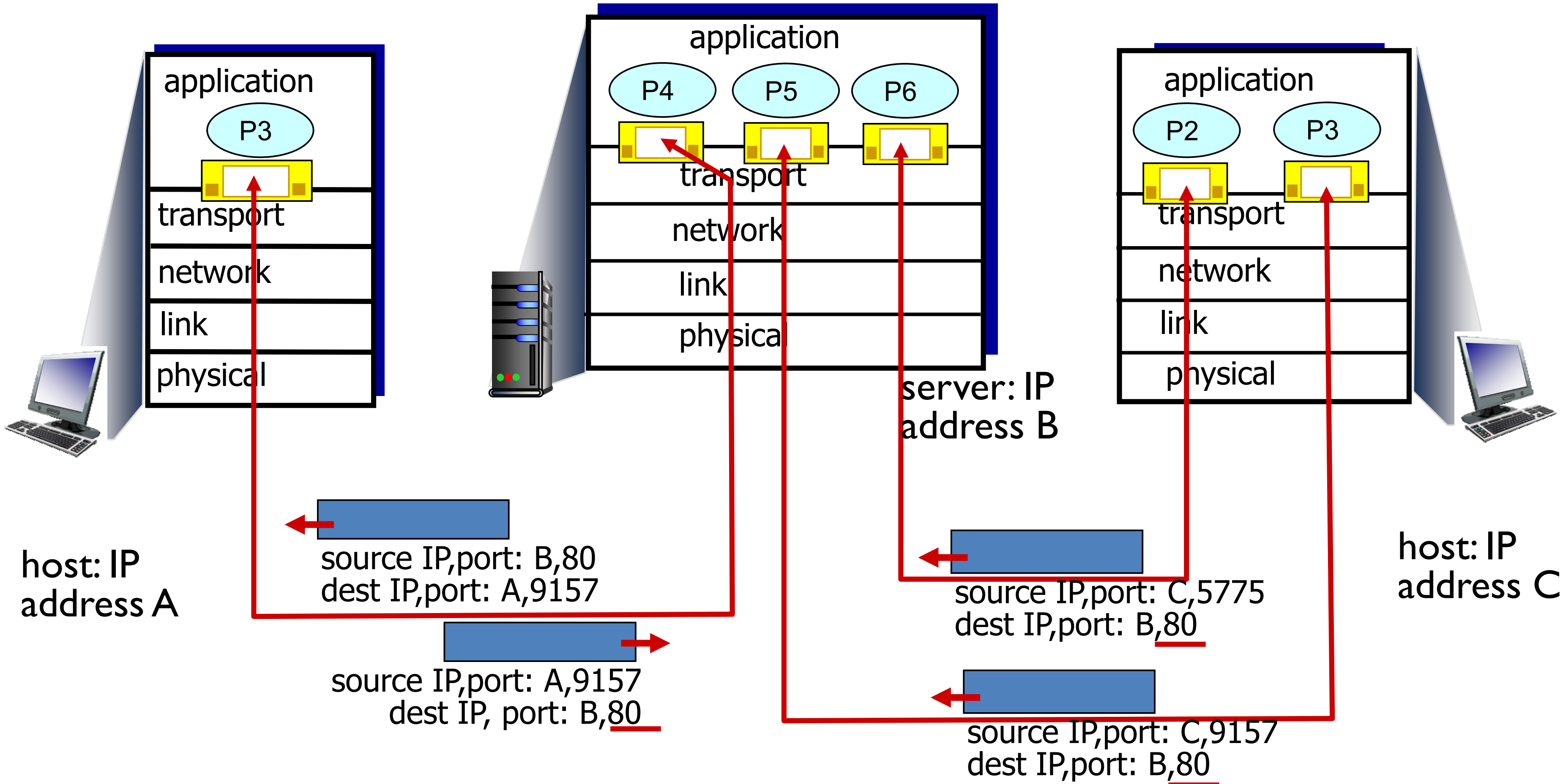
- when host receives UDP segment:
  - checks destination port # in segment
  - directs UDP segment to socket with that port #

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

Connection-oriented demux: example

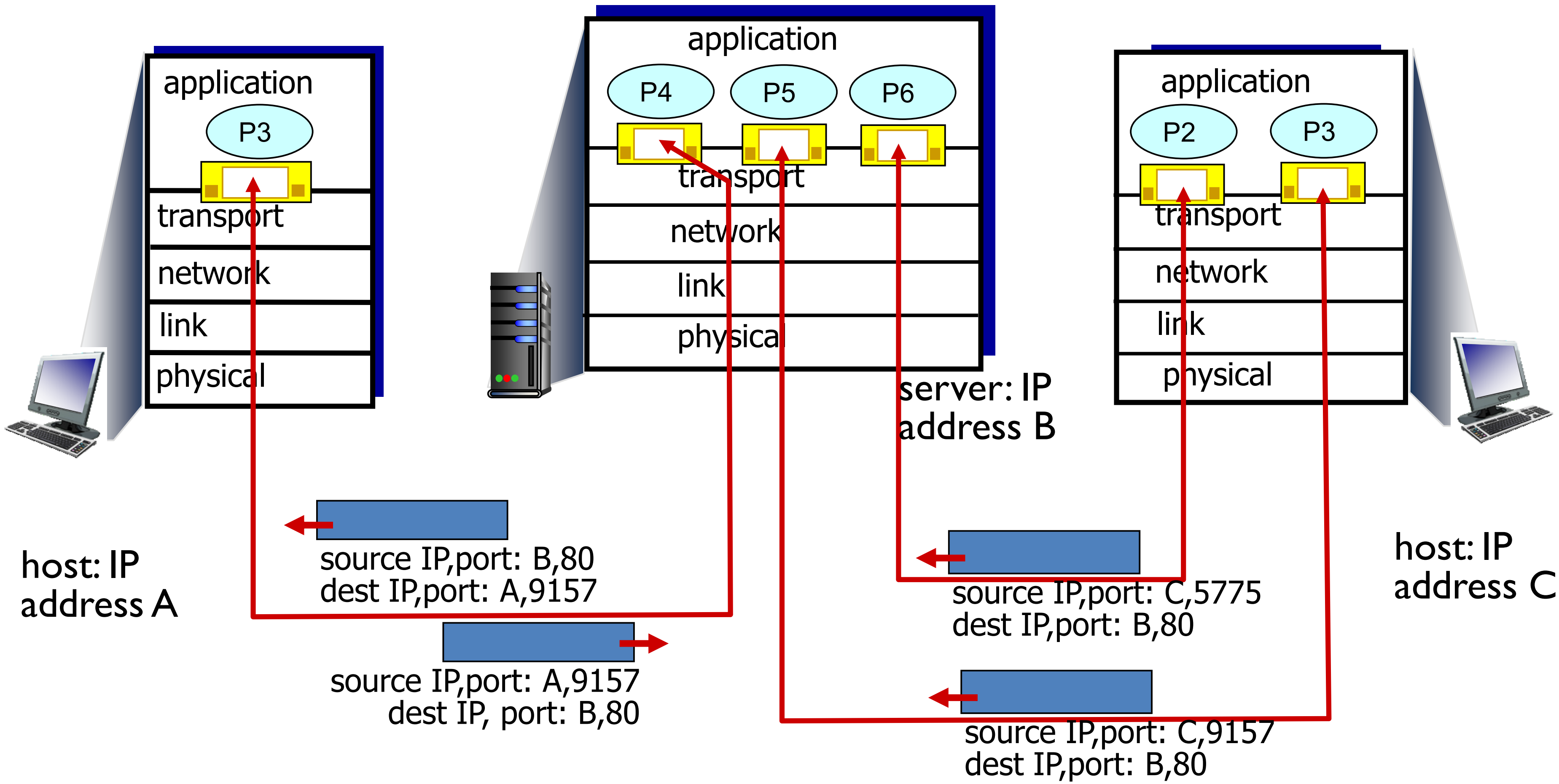


Connection-oriented demux: example



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

Connection-oriented 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

Transport Layer Services

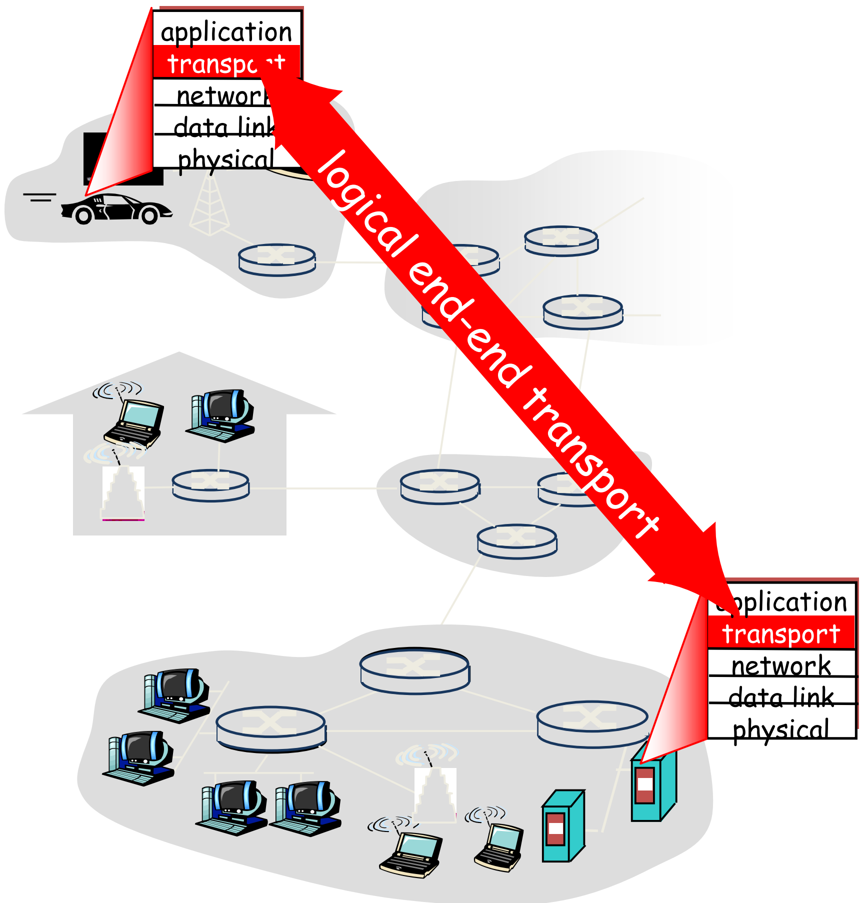
- The ***fundamental service*** of TCP and UDP is to extend the Network Layer packet delivery service provided by IP between ***hosts*** to a delivery service between ***processes***

- *How?*

  TCP and UDP both provide

  multiplexing and de-multiplexing of data from several processes

- **UDP provides**
  - best effort delivery



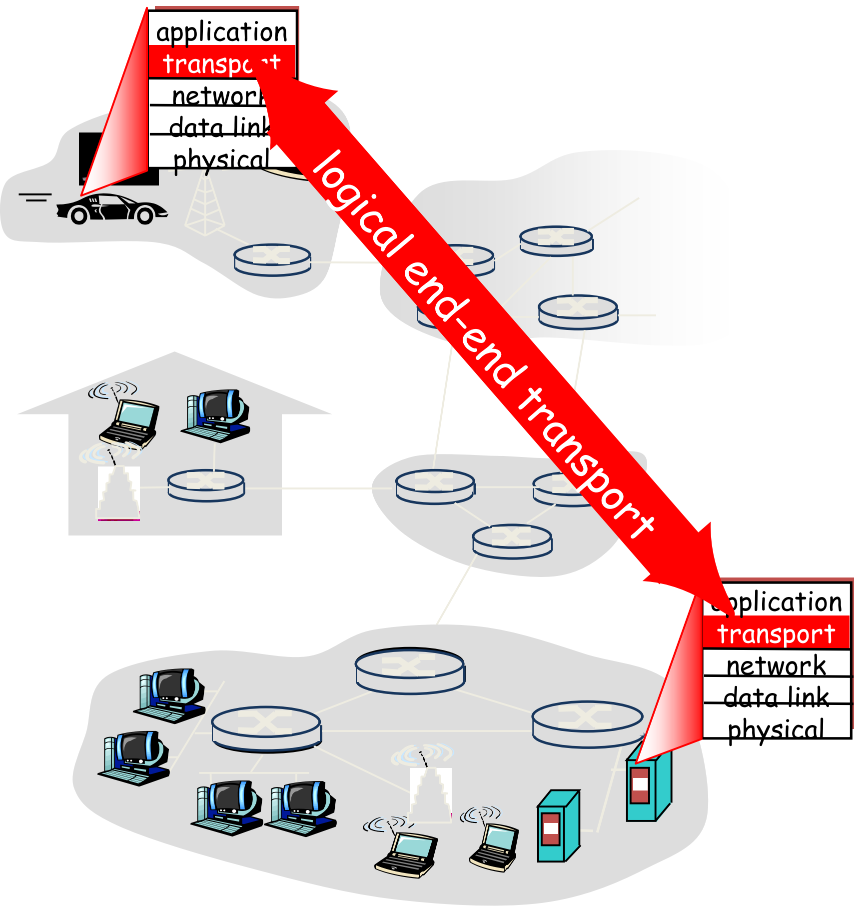
Transport Layer Services

- The ***fundamental service*** of TCP and UDP is to extend the Network Layer packet delivery service provided by IP between ***hosts*** to a delivery service between ***processes***
- *How?*

  TCP and UDP both provide

  multiplexing and de-multiplexing of data from several processes

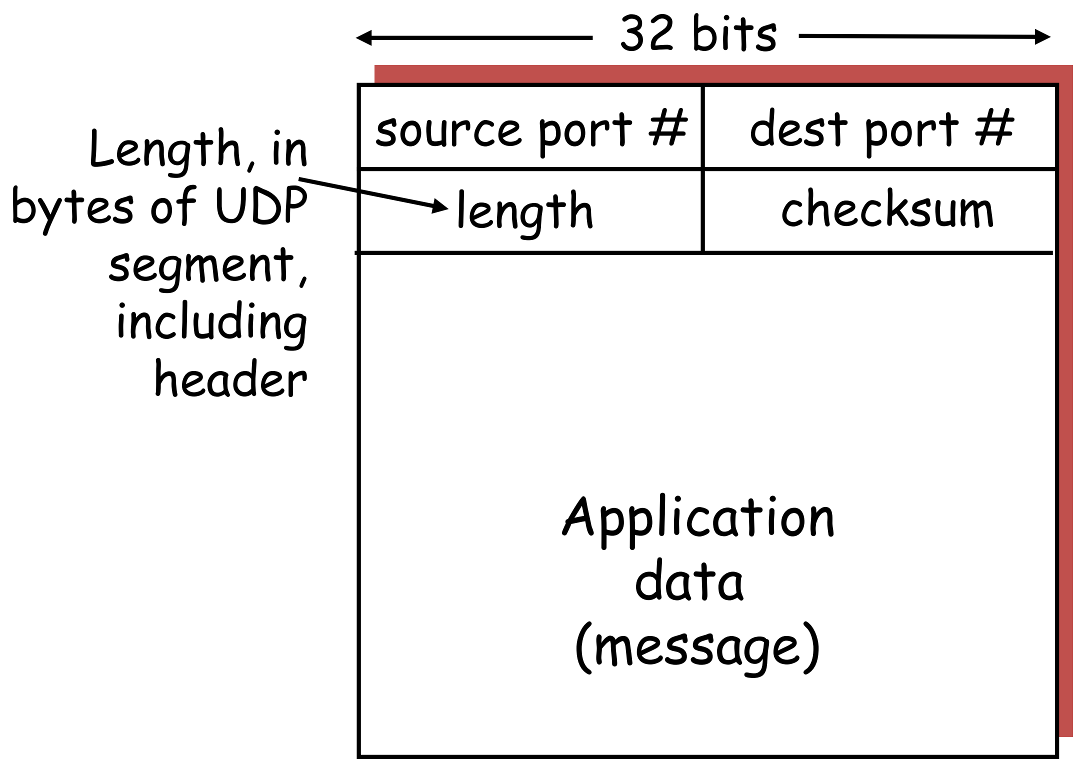
- UDP provides
  - best effort delivery
- **TCP provides**
  1. congestion management
  2. flow control
  3. connection setup
  4. reliable, in-order delivery of data



RFC 768

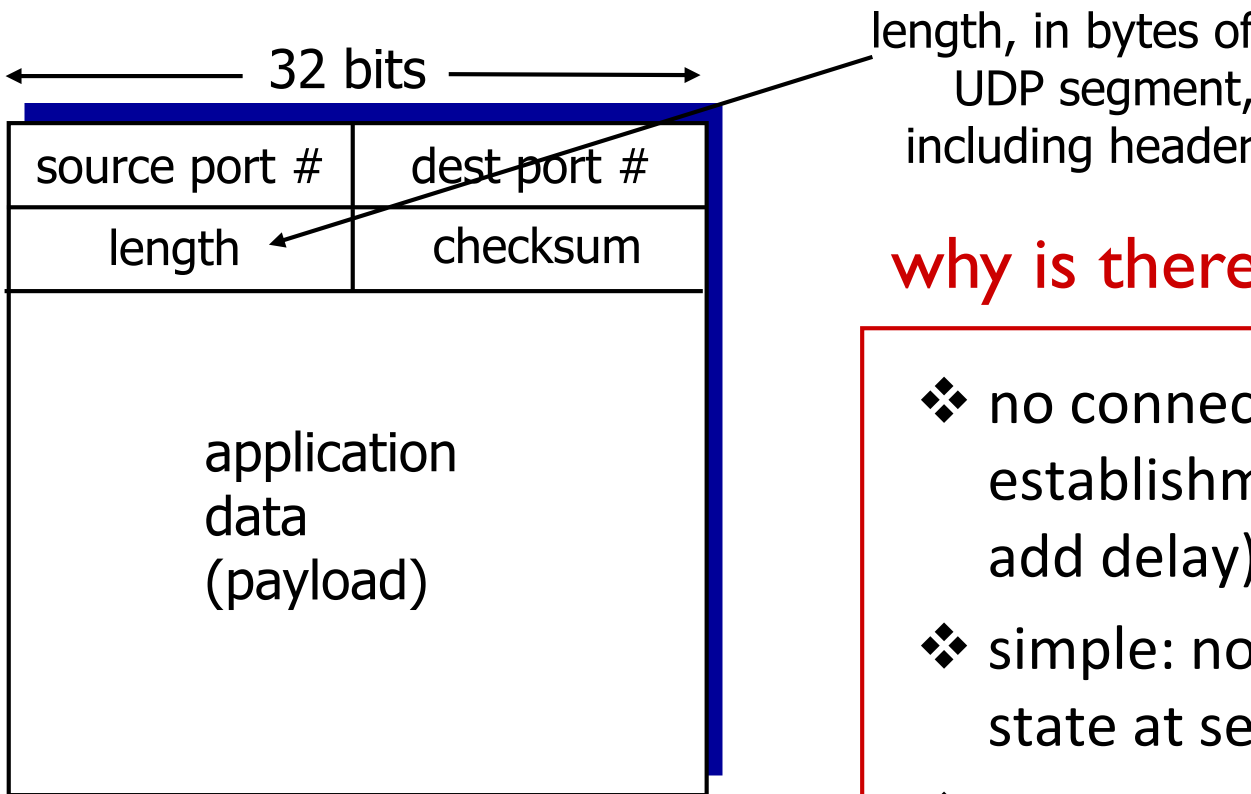
User Datagram Protocol (UDP)

- no connection establishment (which can add delay)
- simple: no connection state at sender, receiver
- small segment header
- no congestion control: UDP can blast away as fast as desired
- can add reliability at application layer
- 1’s complement checksum can be used to detect (but not correct) errors. (example)
- segments can be lost or delivered to application out of order.
- each segment is independent of others.



UDP segment format

UDP: segment header



UDP segment format

why is there a UDP?

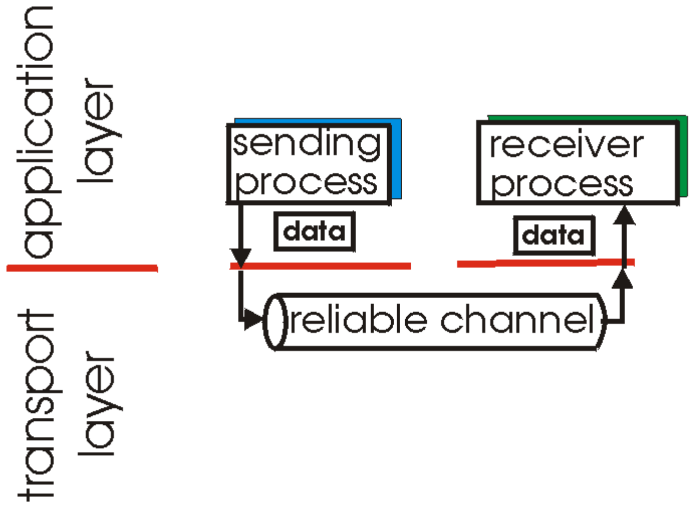
- no connection establishment (which can add delay)
- simple: no connection state at sender, receiver
- small header size
- no congestion control: UDP can blast away as fast as desired

Reliable Data Transfer

- Many applications want reliable data transfer, so many transport layer protocols provide this.
- The service level of the underlying network may vary. Assume the TL needs to deal with errors and loss of data packets.
- Start with the assumption of a reliable network and progressively add in mechanisms for dealing with errors… (on blackboard)

Principles of reliable data transfer

- important in application, transport, link layers
  - top-10 list of important networking topics!

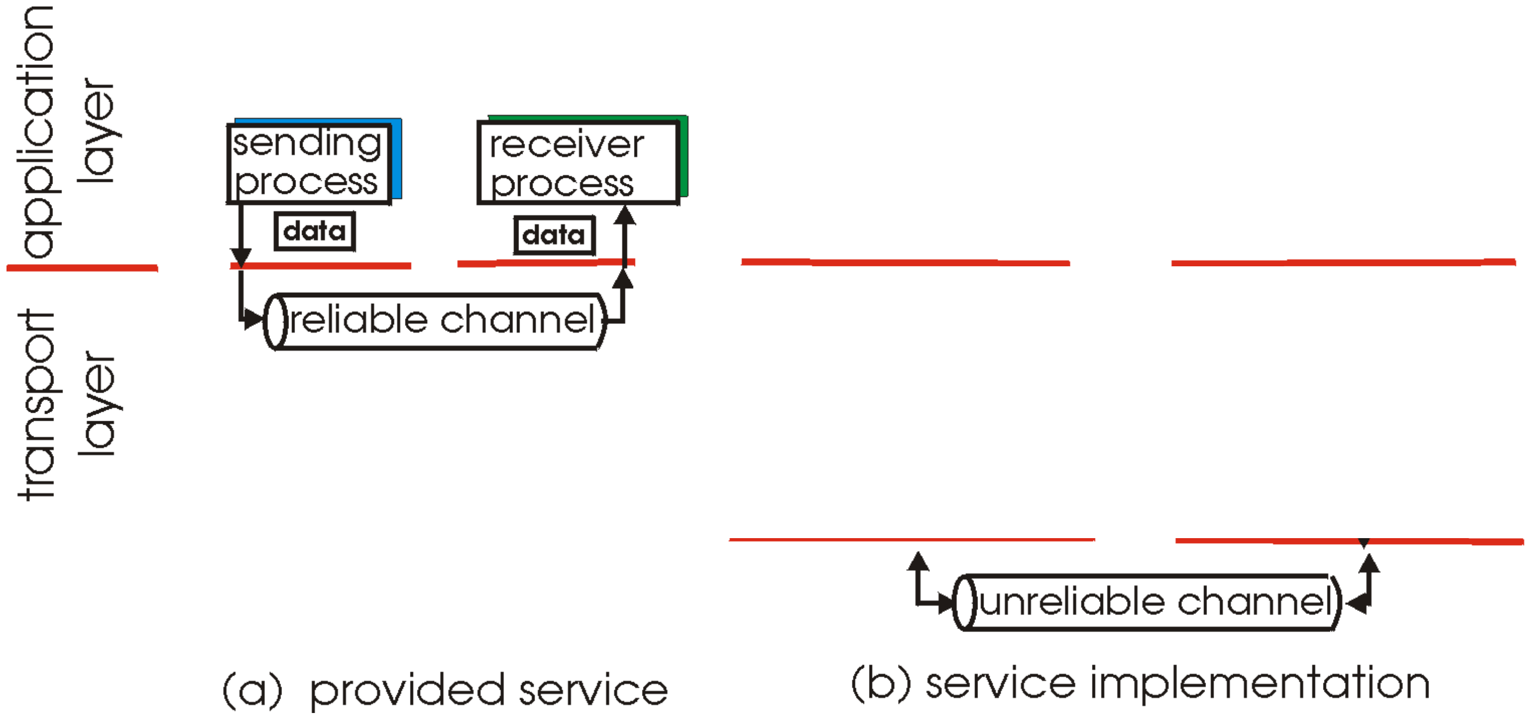


(a) provided service

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

Principles of reliable data transfer

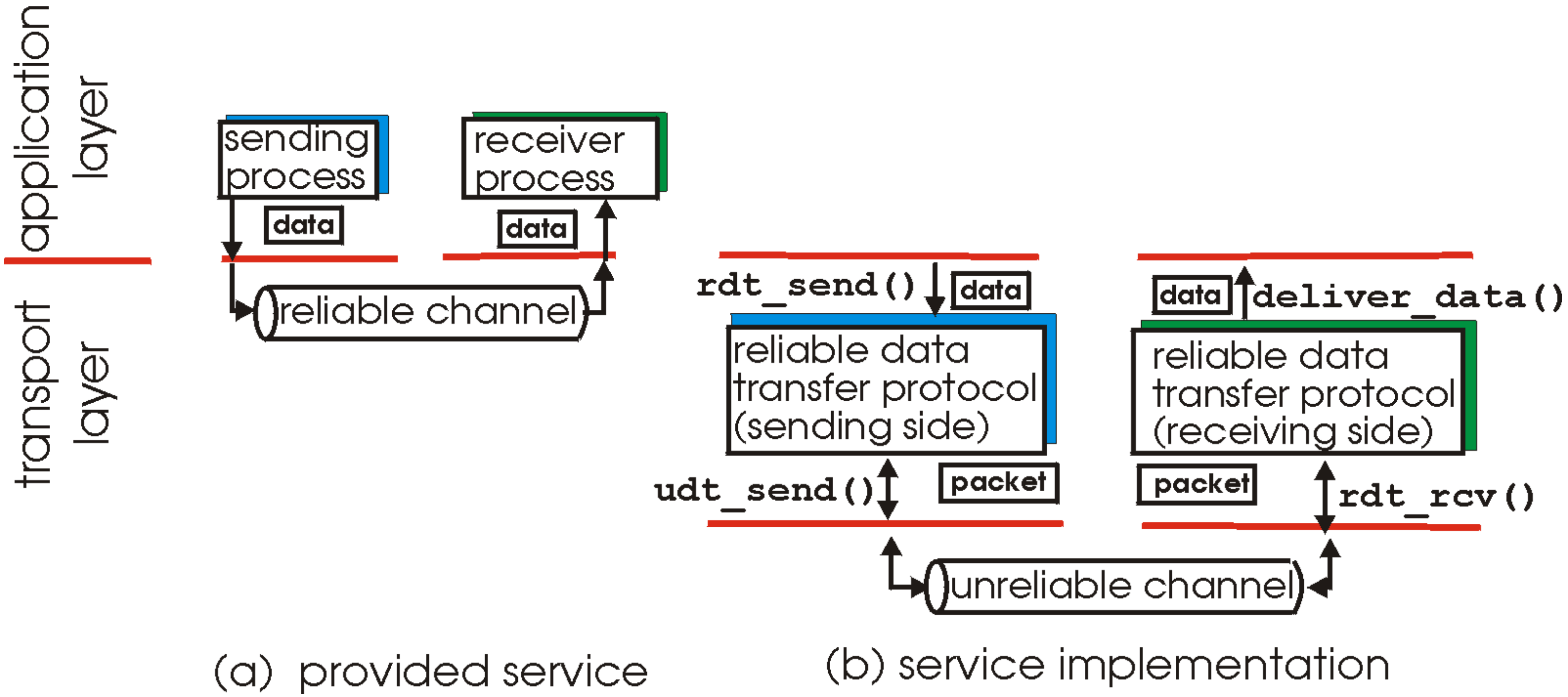
- important in application, transport, link layers
  - top-10 list of important networking topics!



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

Principles of reliable data transfer

- important in application, transport, link layers
  - top-10 list of important networking topics!

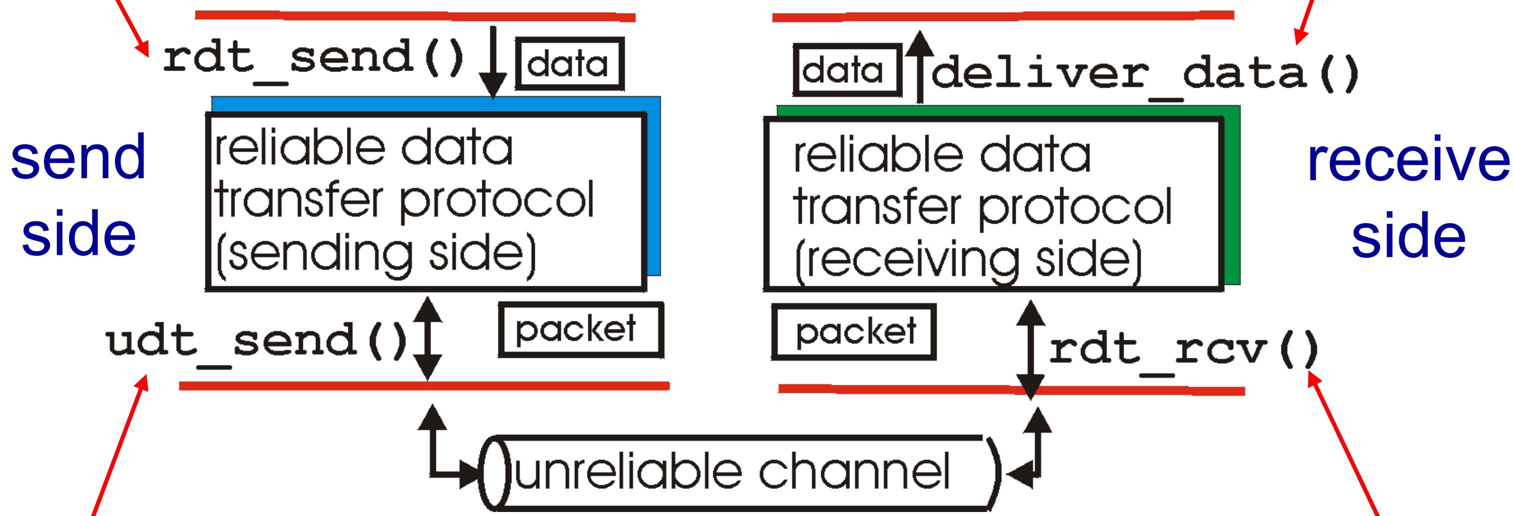


- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

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 upper



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

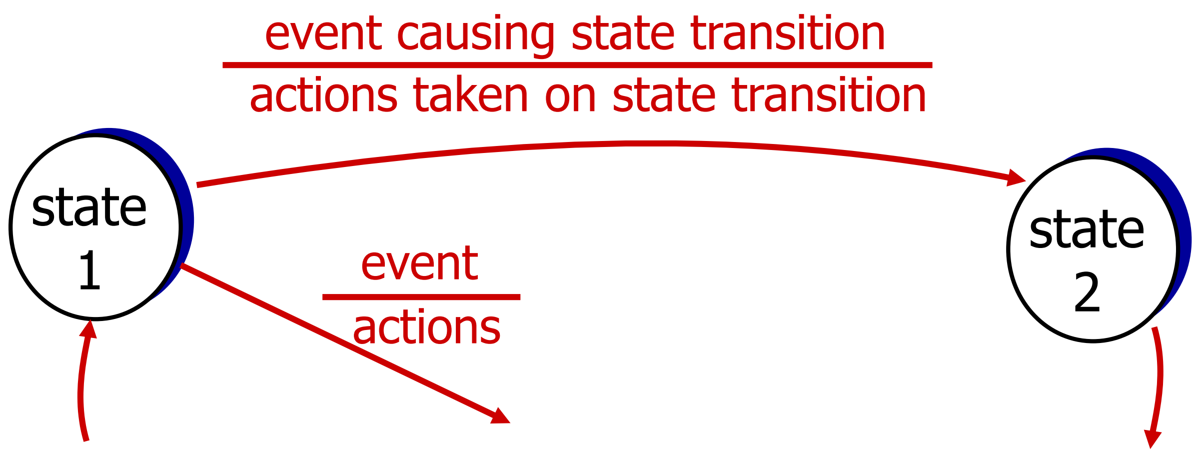
rdt_rcv(): called when packet arrives on rcv-side of channel

Reliable data transfer: getting started

we' ll:

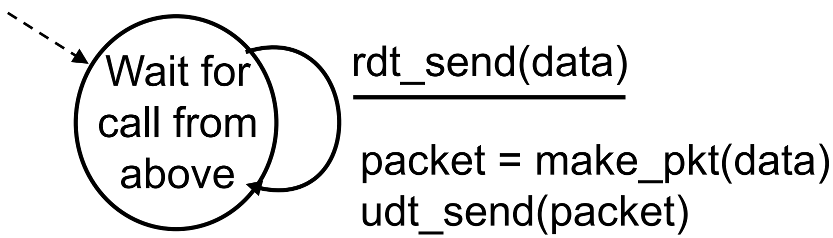
- incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- consider only unidirectional data transfer
  - but control info will flow on both directions!
- use finite state machines (FSM) to specify sender, receiver

state: when in this "state" next state uniquely determined by next event

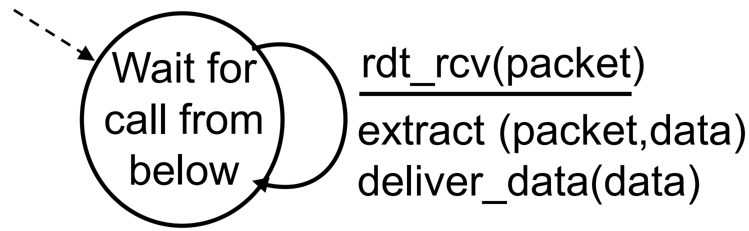


rdt1.0: reliable transfer over a reliable channel

- 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

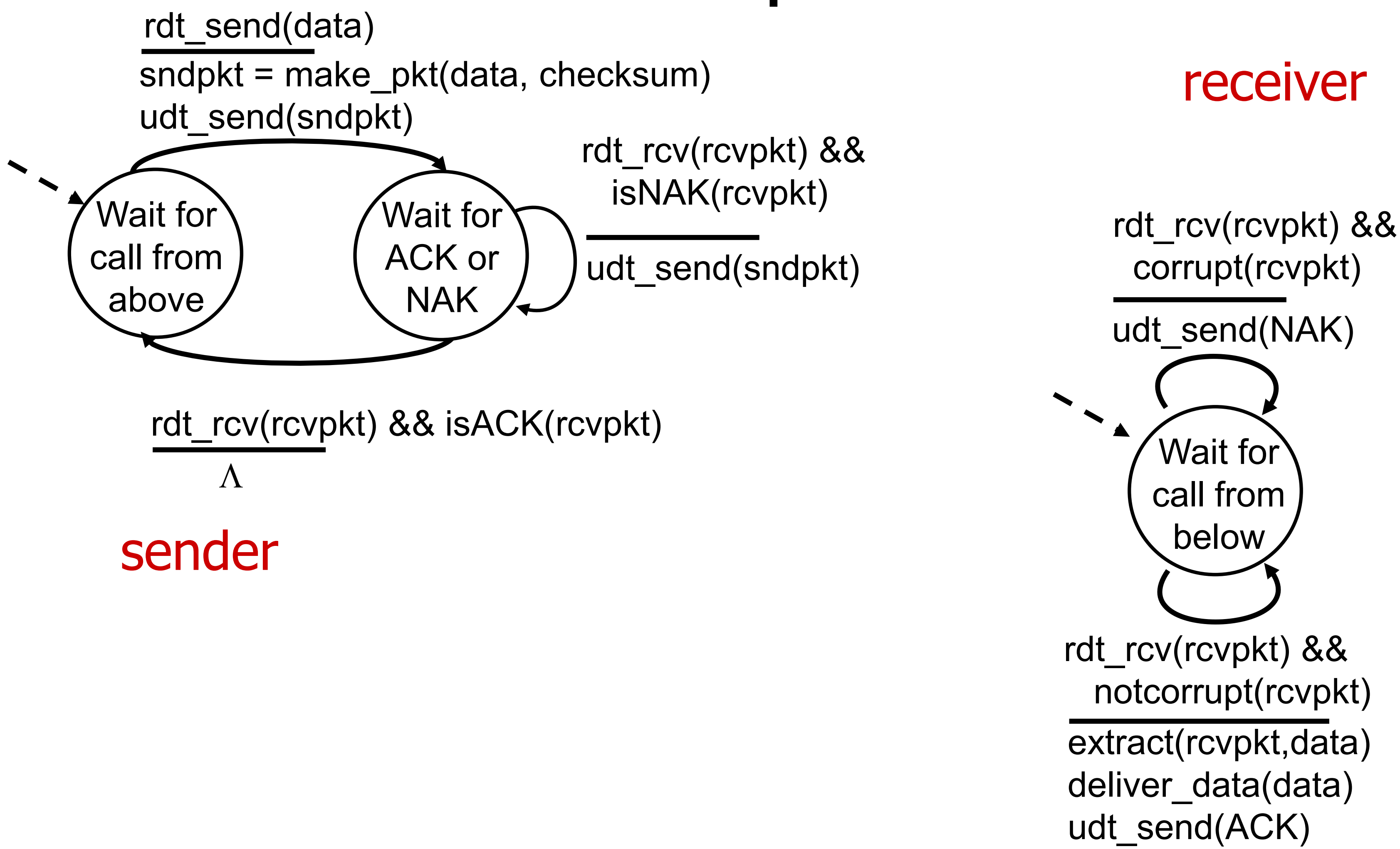
- underlying channel may flip bits in packet
  - checksum to detect bit errors
- *the* question: how to recover from errors:

How do humans recover from "errors" during conversation?

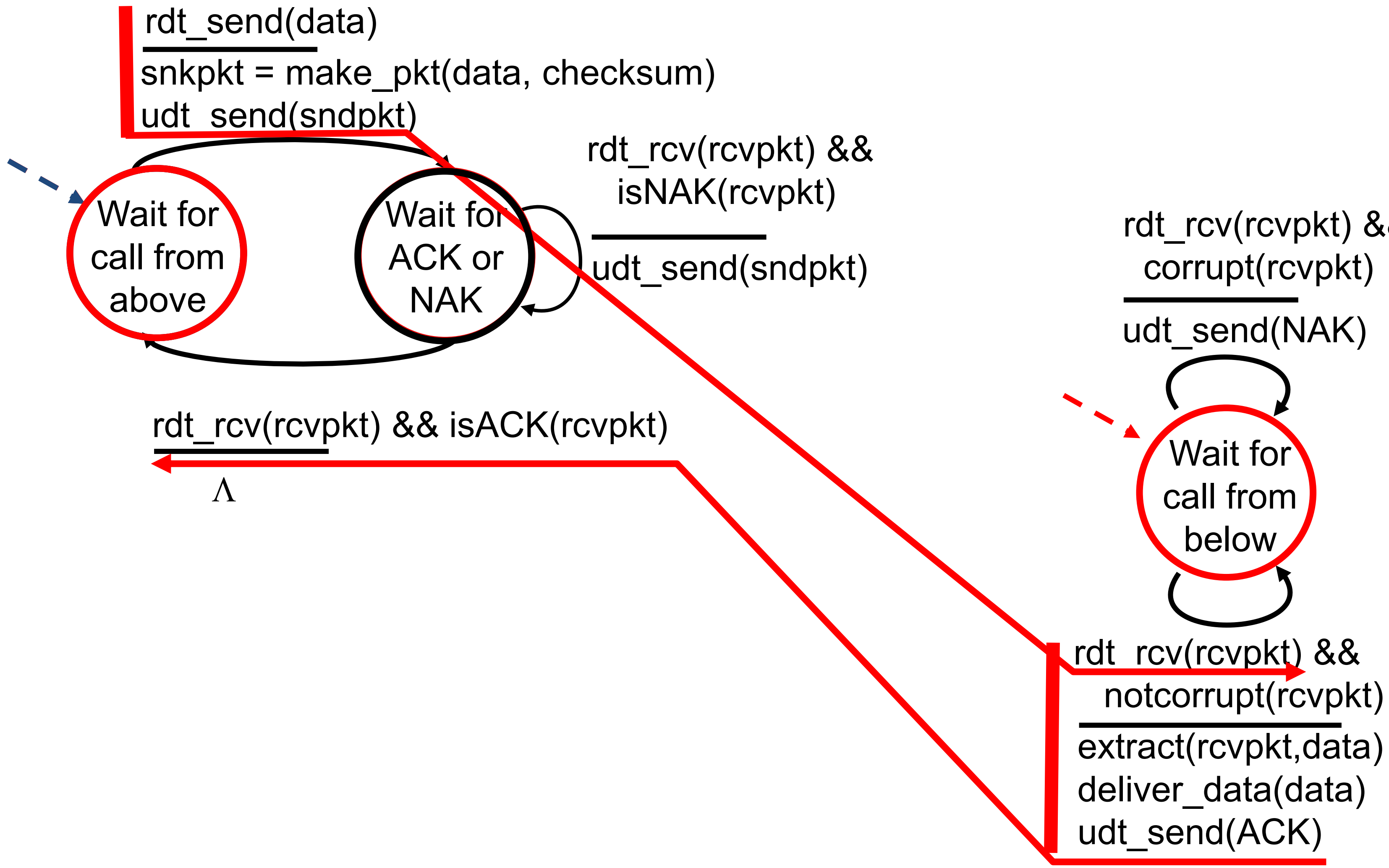
rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
  - checksum to detect bit errors
- *the* question: how to recover from errors:
  - *acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK
  - *negative acknowledgements (NAKs):* receiver explicitly tells sender that pkt had errors
  - sender retransmits pkt on receipt of NAK
- new mechanisms in **`rdt2.0`** (beyond **`rdt1.0`**):
  - error detection
  - feedback: control msgs (ACK,NAK) from receiver to sender

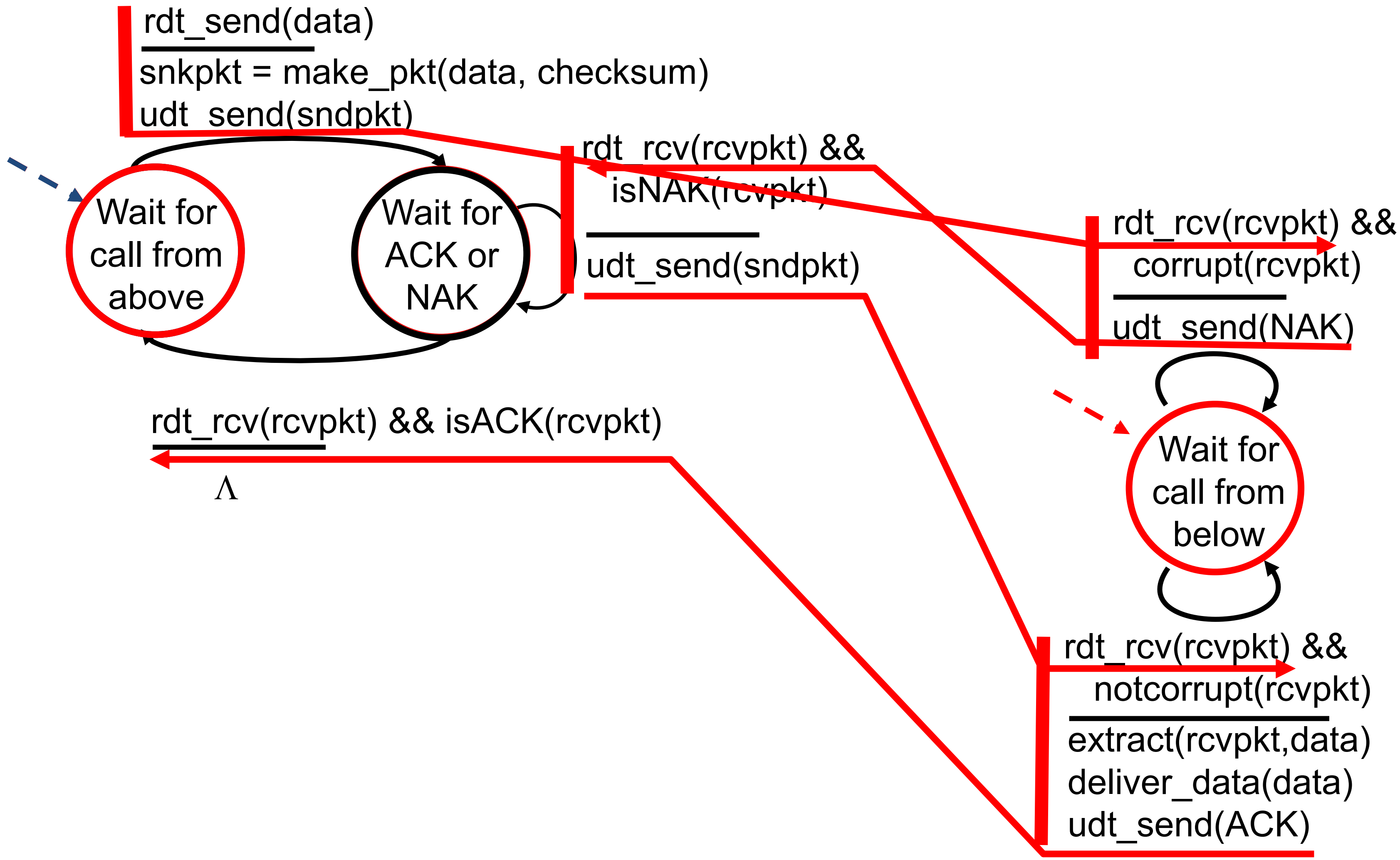
rdt2.0: FSM specification



rdt2.0: operation with no errors



rdt2.0: error scenario



rdt2.0 has a fatal flaw!

what happens if ACK/NAK corrupted?

- sender doesn't know what happened at receiver!
- can't just retransmit: possible duplicate

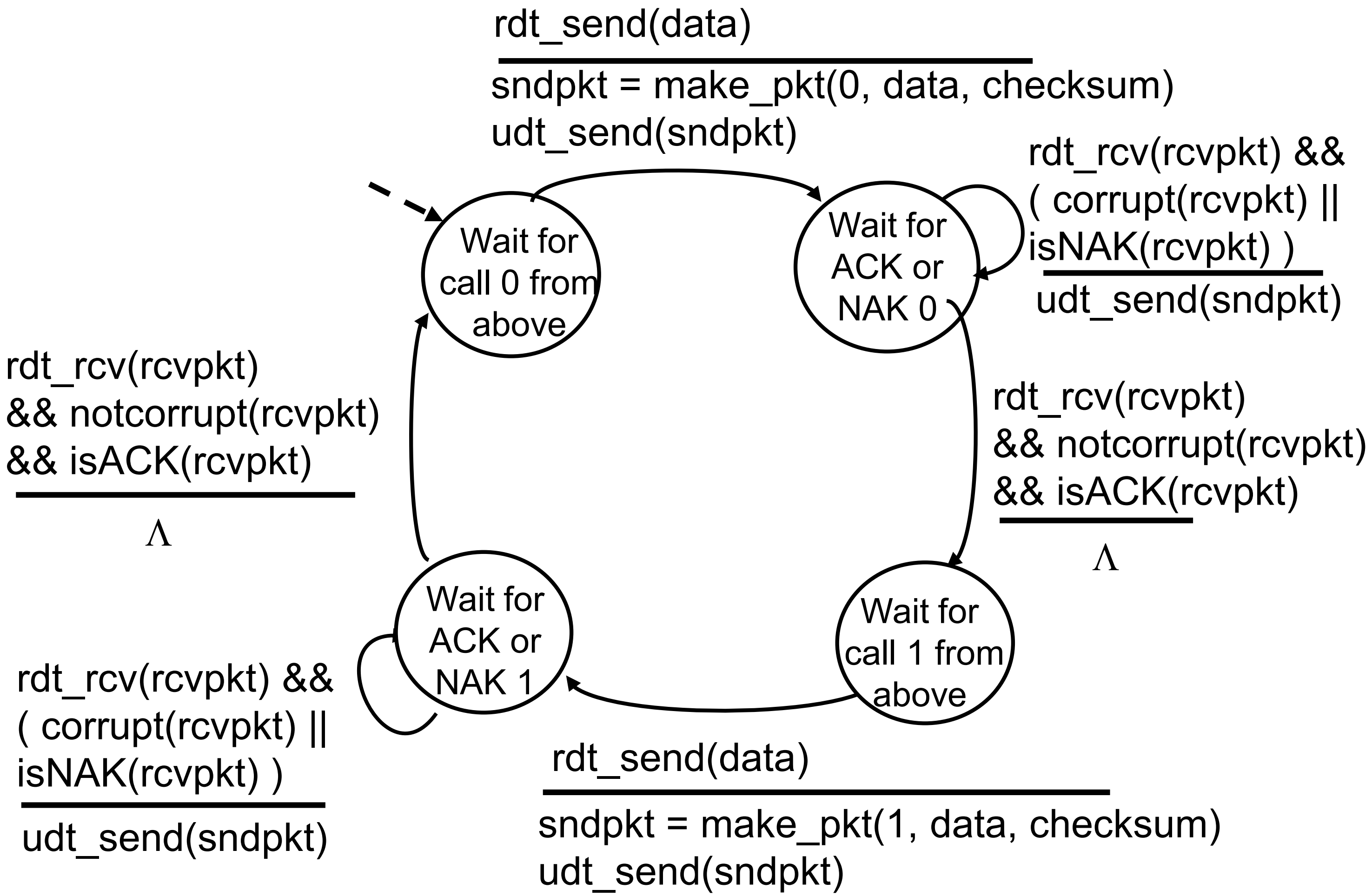
handling duplicates:

- sender retransmits current pkt if ACK/NAK corrupted
- sender adds *sequence number* to each pkt
- receiver discards (doesn't deliver up) duplicate pkt

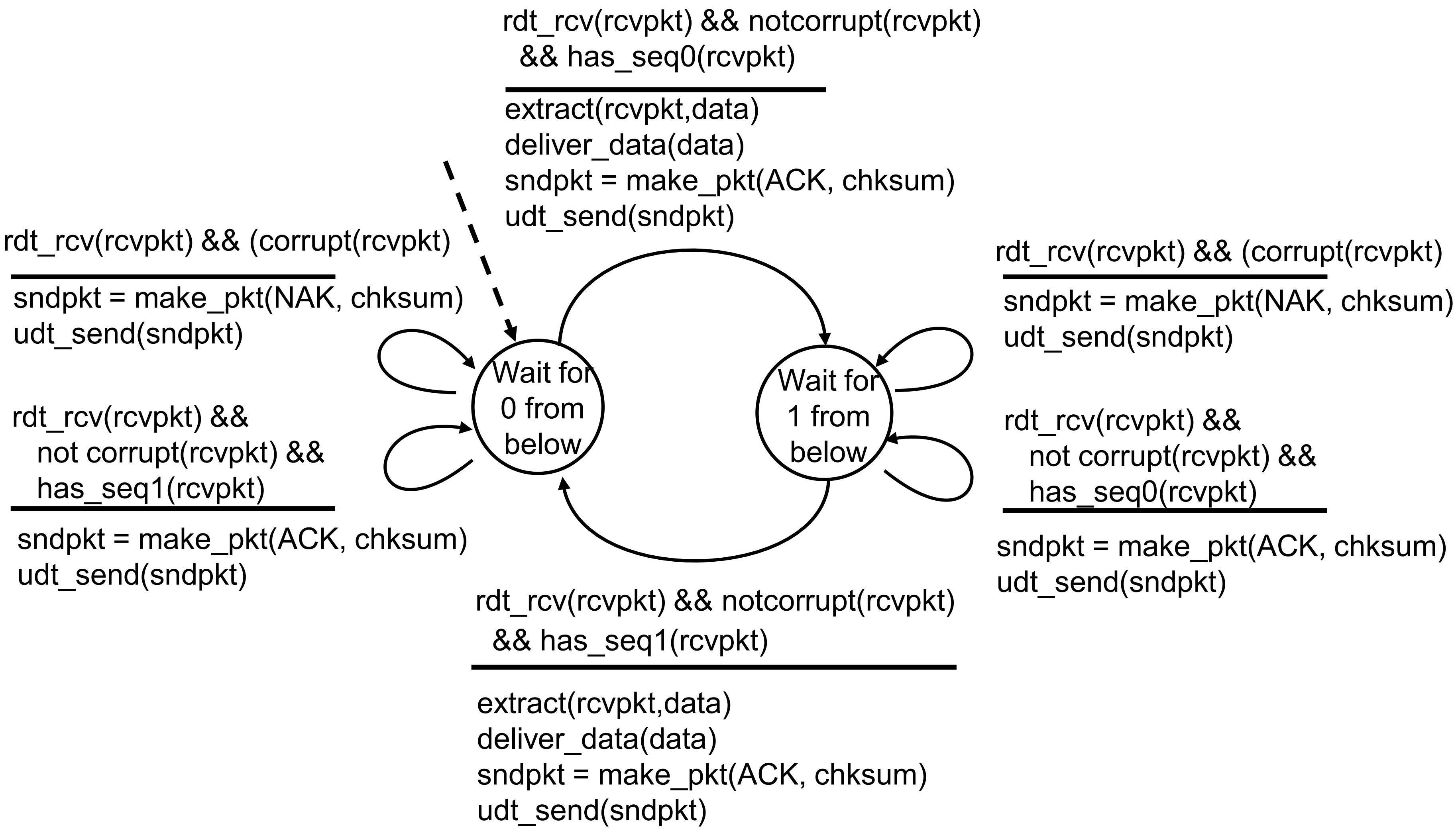
stop and wait

sender sends one 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 "expected" pkt should have seq # of 0 or 1

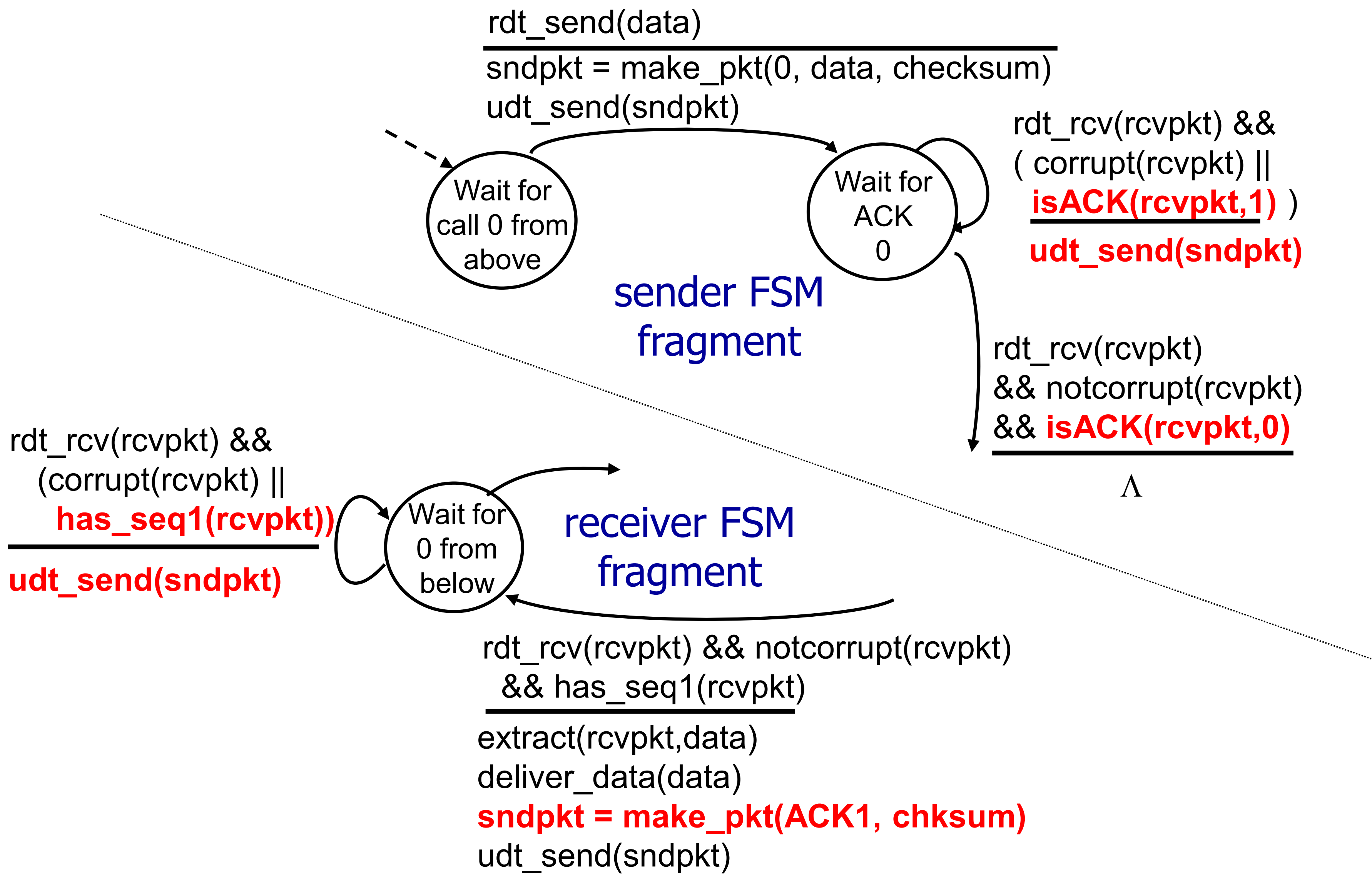
receiver:

- must check if received packet is duplicate
  - state indicates whether 0 or 1 is expected pkt seq #
- note: receiver can *not* know if its last ACK/NAK received OK at sender

rdt2.2: a NAK-free protocol

- same functionality as rdt2.1, using ACKs only
- instead of NAK, receiver sends ACK for last pkt received OK
  - receiver must *explicitly* include seq # of pkt being ACKed
- duplicate ACK at sender results in same action as NAK: *retransmit current pkt*

rdt2.2: sender, receiver fragments



rdt3.0: channels with errors *and* loss

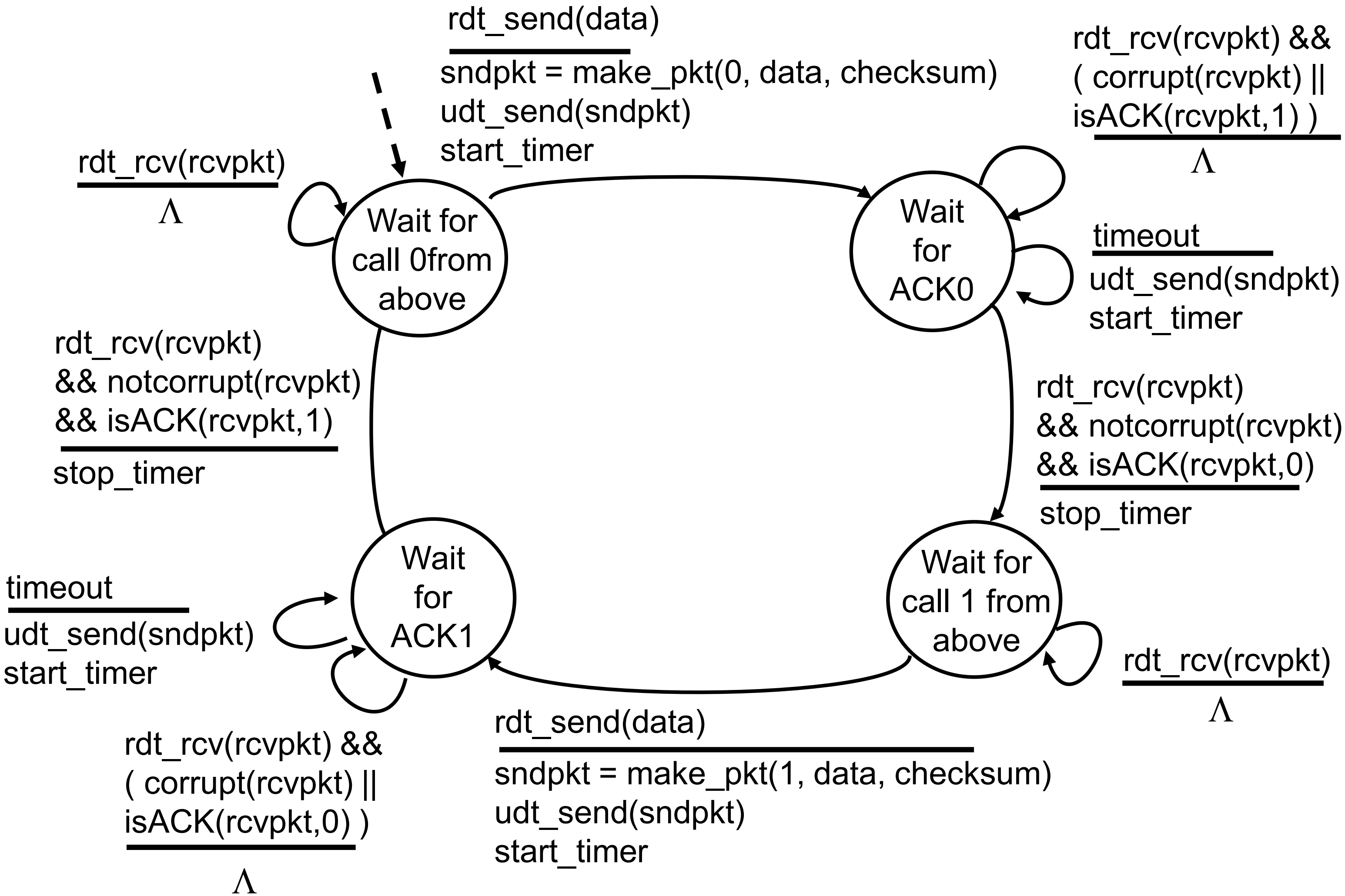
new assumption: underlying channel can also lose packets (data, ACKs)

- checksum, seq. #, ACKs, retransmissions will be of help … but not enough

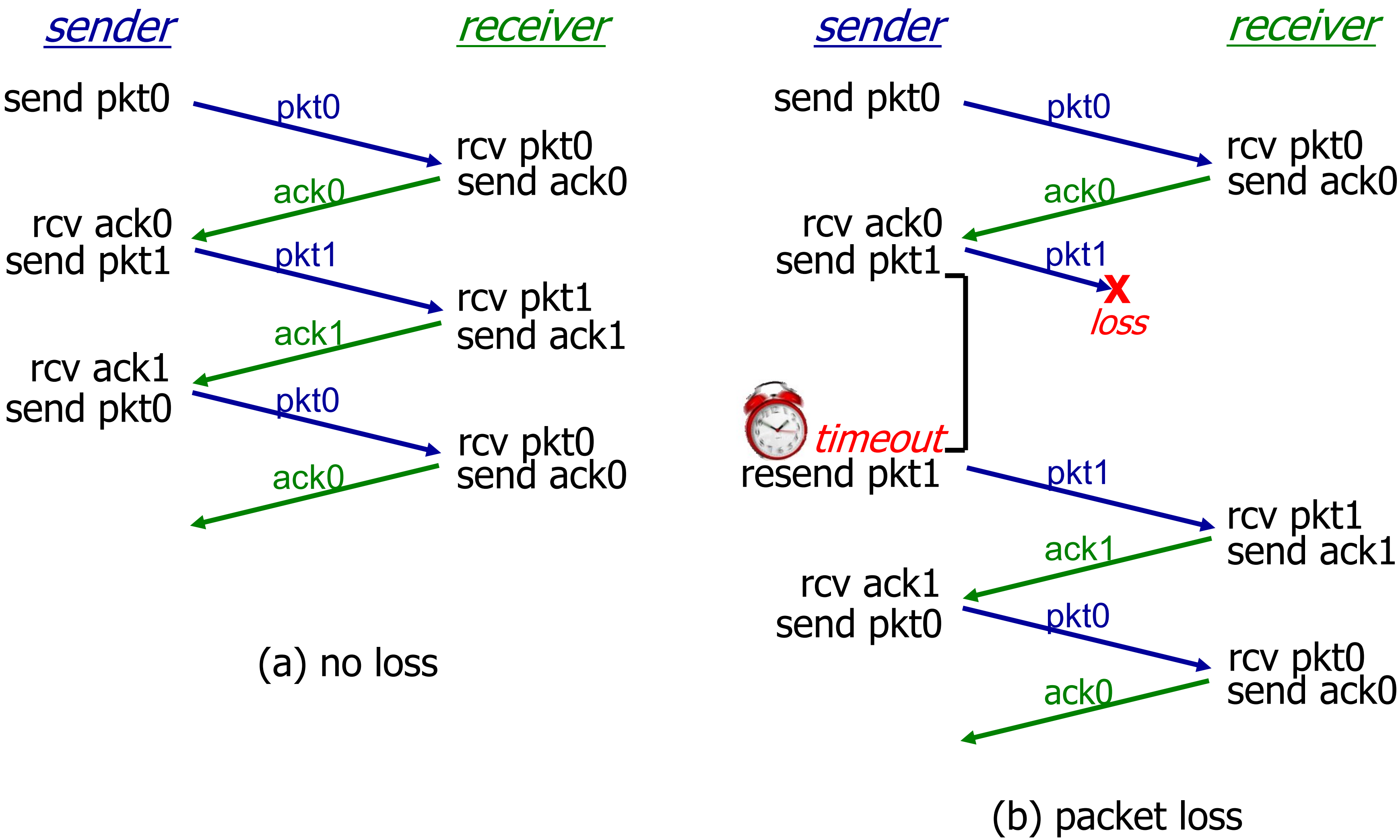
approach: sender waits "reasonable" amount of time for ACK

- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but seq. #'s already handles this
  - receiver must specify seq # of pkt being ACKed
- requires countdown timer

rdt3.0 sender

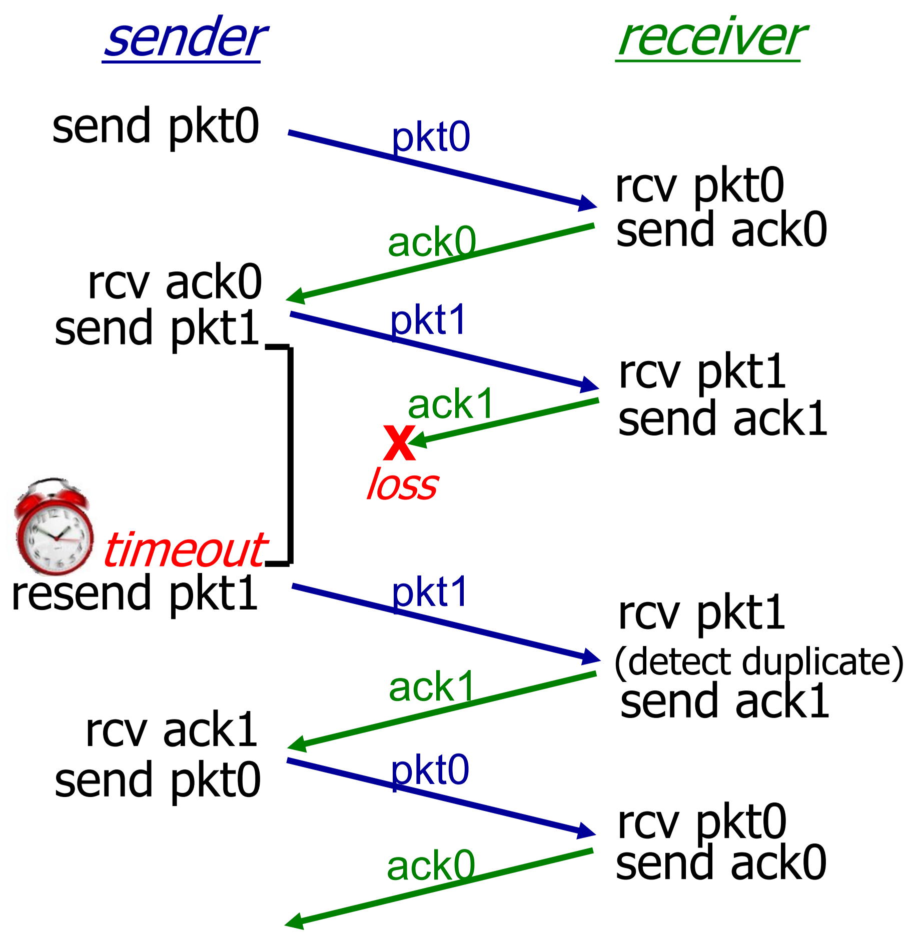


rdt3.0 in action

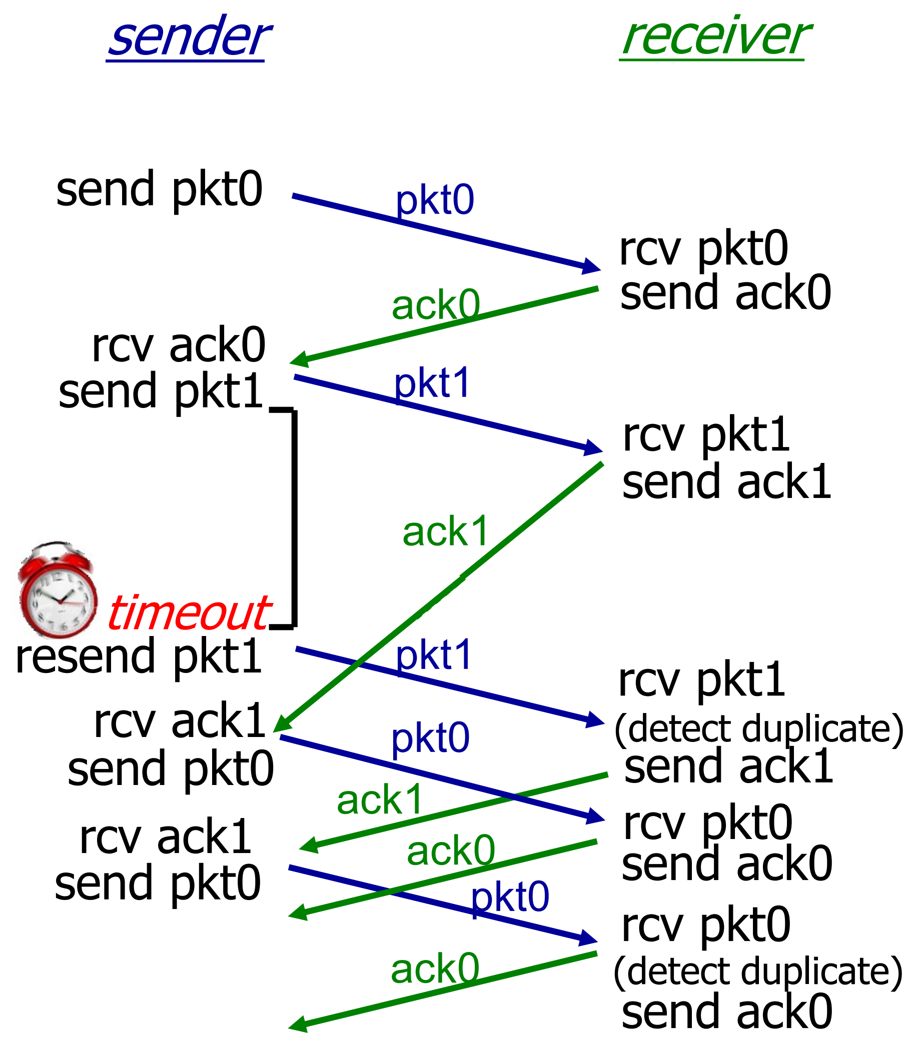


Computer Networking and Applications

rdt3.0 in action



(c) ACK loss



(d) premature timeout/ delayed ACK

Need for Pipelining

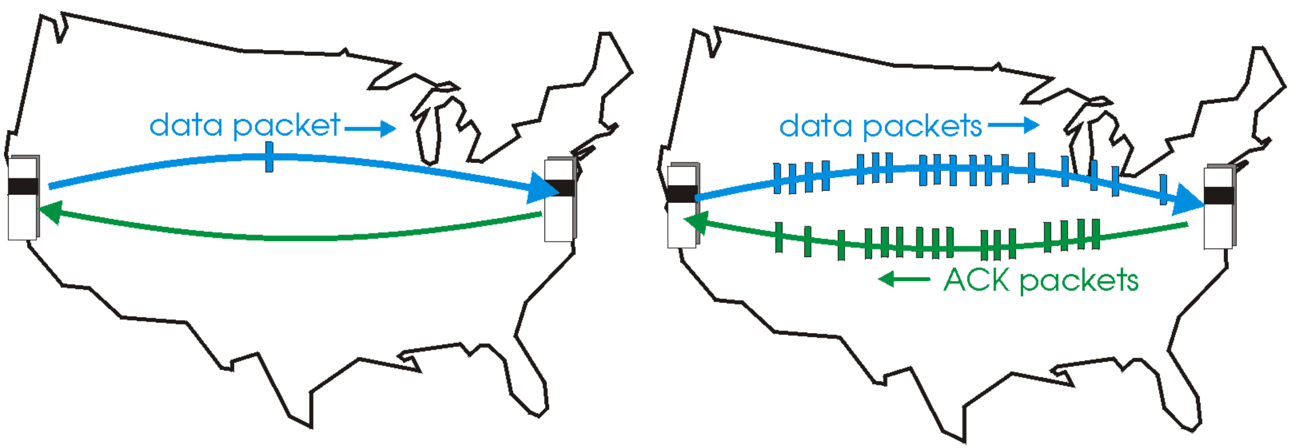
- 1 Gbps link, 15 ms e-e prop. delay, 1KB packet:

$$T_{transmit} = \frac{8\text{Kb/pkt}}{10^{**}9\ \text{b/sec}} = 8\ \text{microsec/packet}$$

$$\text{Utilization} = U = \frac{\text{fraction of time}}{\text{sender busy sending}} = \frac{8\ \text{microsec}}{30.008\ \text{msec}} = 0.00027$$

1KB pkt every 30 msec -> 33KB/sec thruput over 1 Gbps link

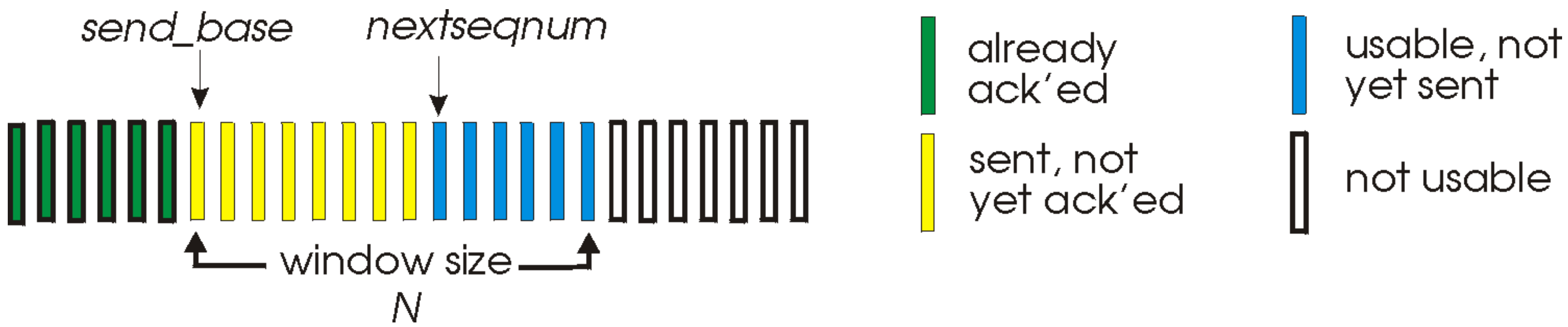
network protocol limits use of physical resources!



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

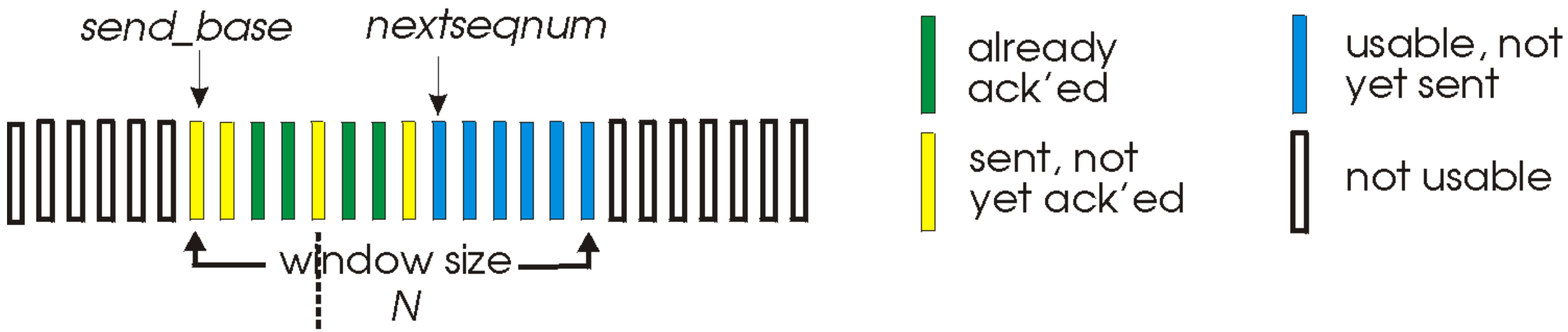
go-Back-N

- Sender

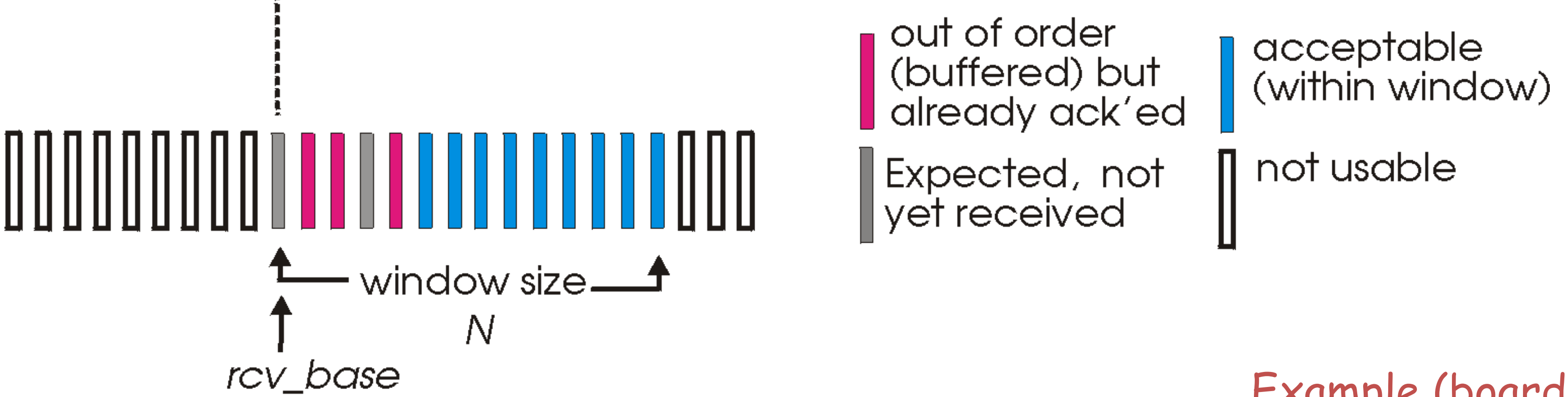


  - one timer
  - on timeout, retransmit all packets in window from last ACK + 1
- Receiver
  - cumulative ACK
  - discard out of order packets
- Example
- Advantages & disadvantages

Selective Repeat



(a) sender view of sequence numbers



(b) receiver view of sequence numbers

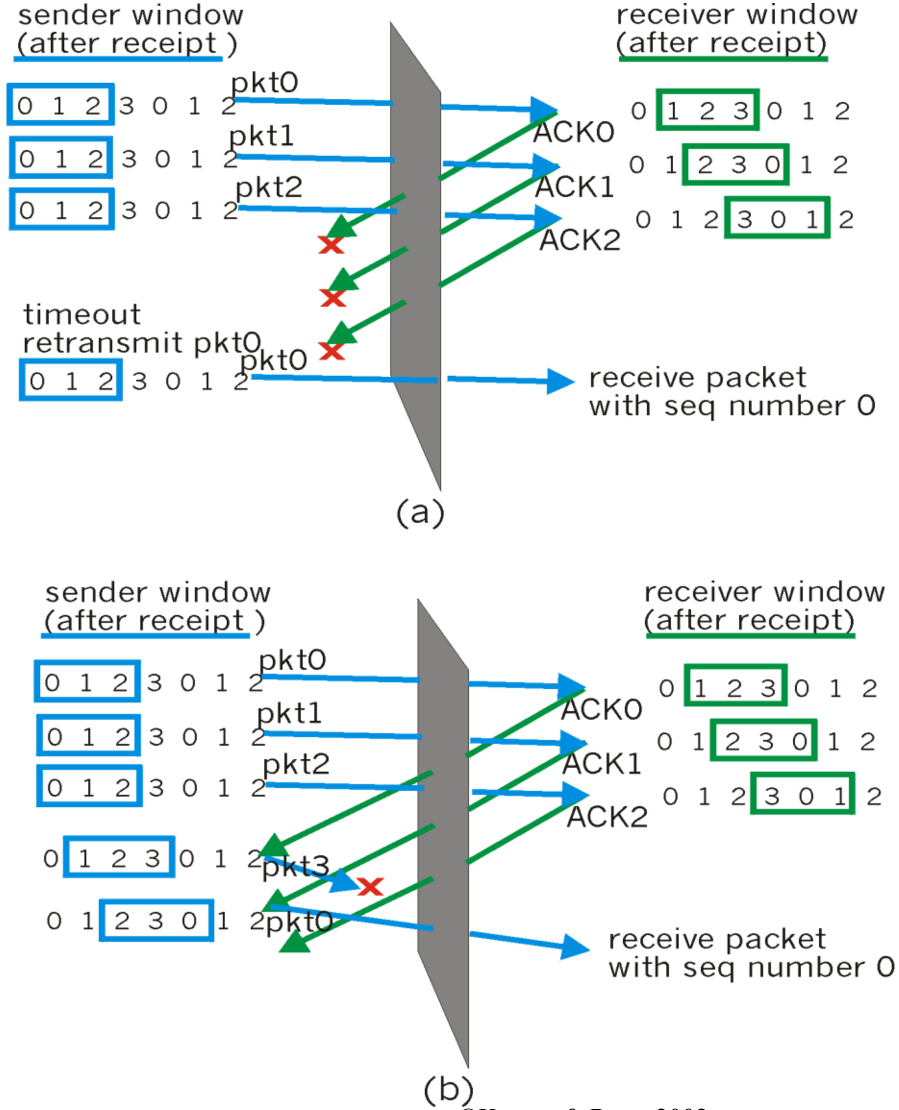
Example (board)

Selective Repeat - window size

Example:

- seq #' s: 0, 1, 2, 3
- window size=3
- receiver sees no difference in two scenarios
- incorrectly passes duplicate data as new in (a)

Q: what is the relationship between seq# size and window size?



TCP

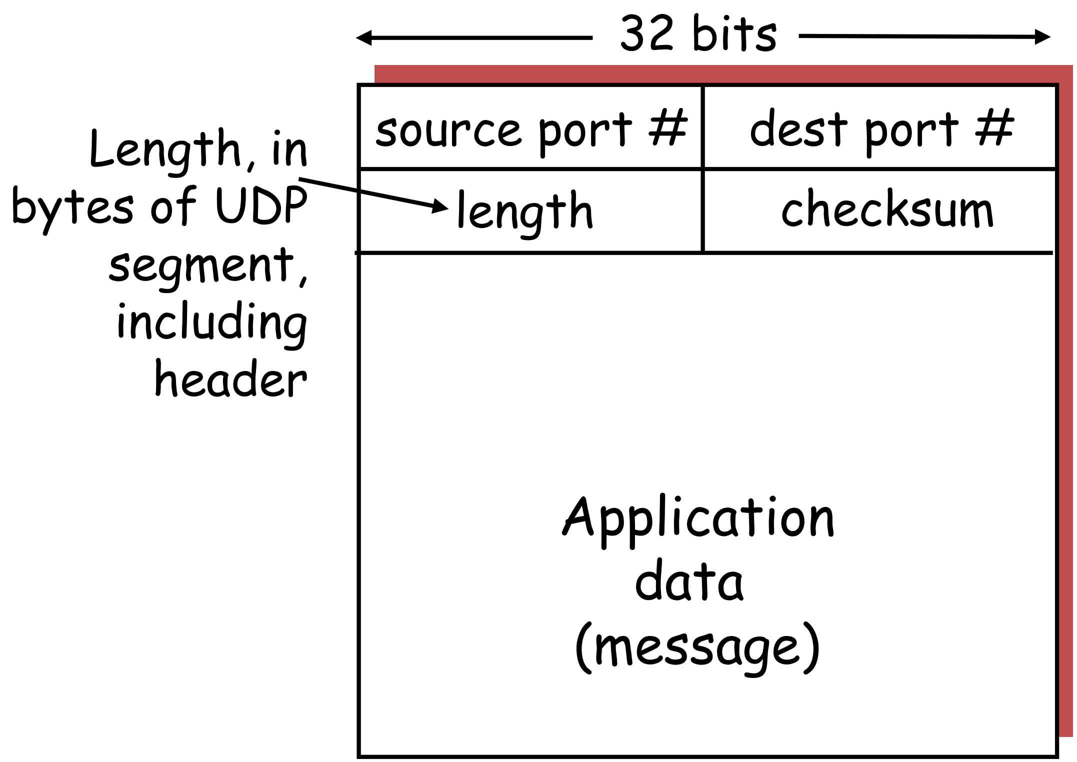
RFCs 793, 1122, 1323, 2018, 2581

- TCP is designed to provide the appearance of a reliable channel over the unreliable network layer (IP).
- In addition to the checksum and multiplexing as provided by UDP, TCP also provides **flow control** and **congestion control** and a **reliable connection**.
- The channel, or *connection*, is **not a virtual circuit**. All state information resides in the sending and receiving hosts, not in the routers.
- TCP deals with
  - Lost packets
  - Re-ordered packets
  - Delayed packets
- TCP is a modified hybrid of go-back-N and selective repeat. Cumulative ACKs. Only ACK up to correctly received segments.

RFC 768

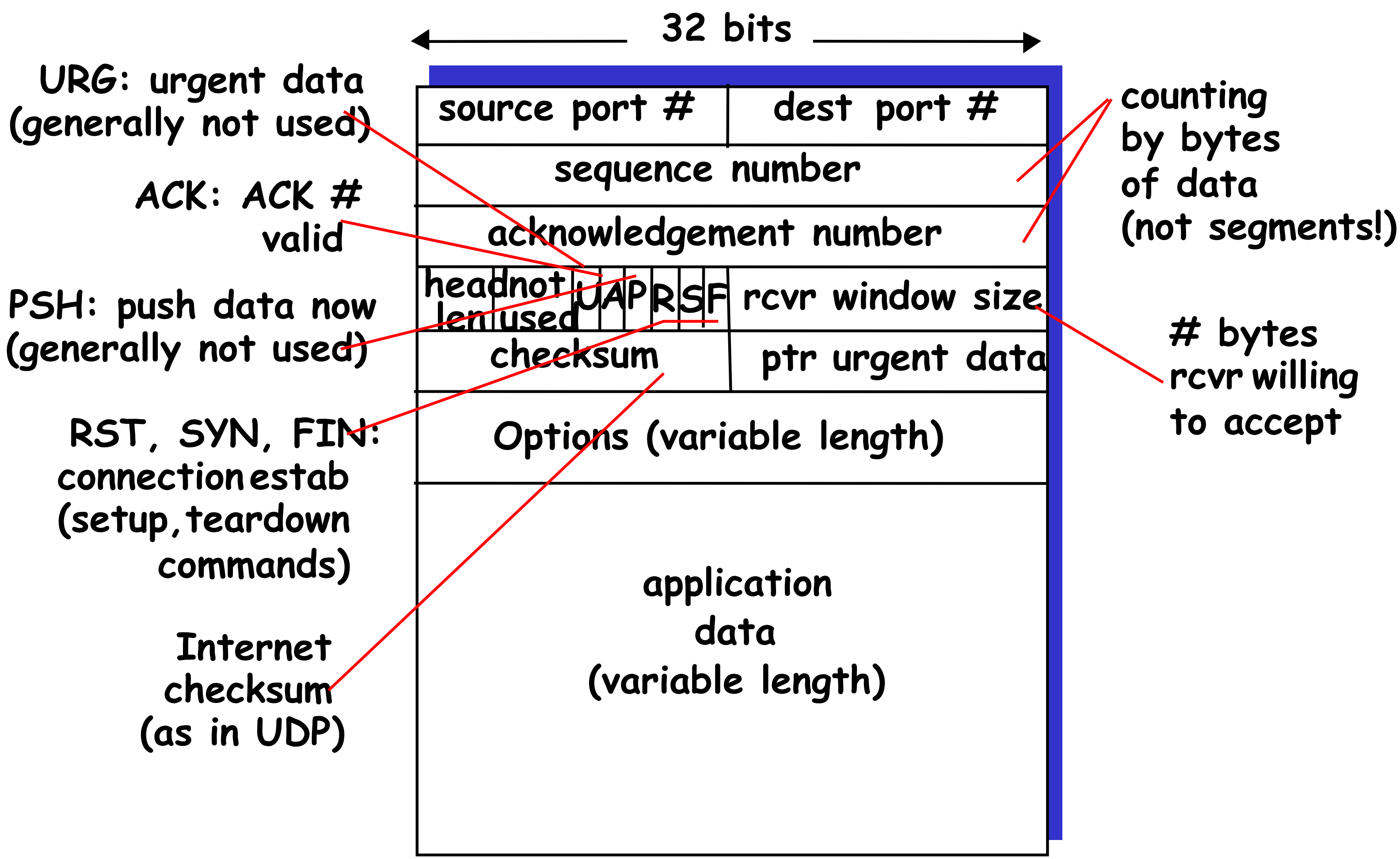
User Datagram Protocol (UDP)

- no connection establishment (which can add delay)
- simple: no connection state at sender, receiver
- small segment header
- no congestion control: UDP can blast away as fast as desired
- can add reliability at application layer
- 1’s complement checksum can be used to detect (but not correct) errors. (example)
- segments can be lost or delivered to application out of order.
- each segment is independent of others.



UDP segment format

TCP segment structure



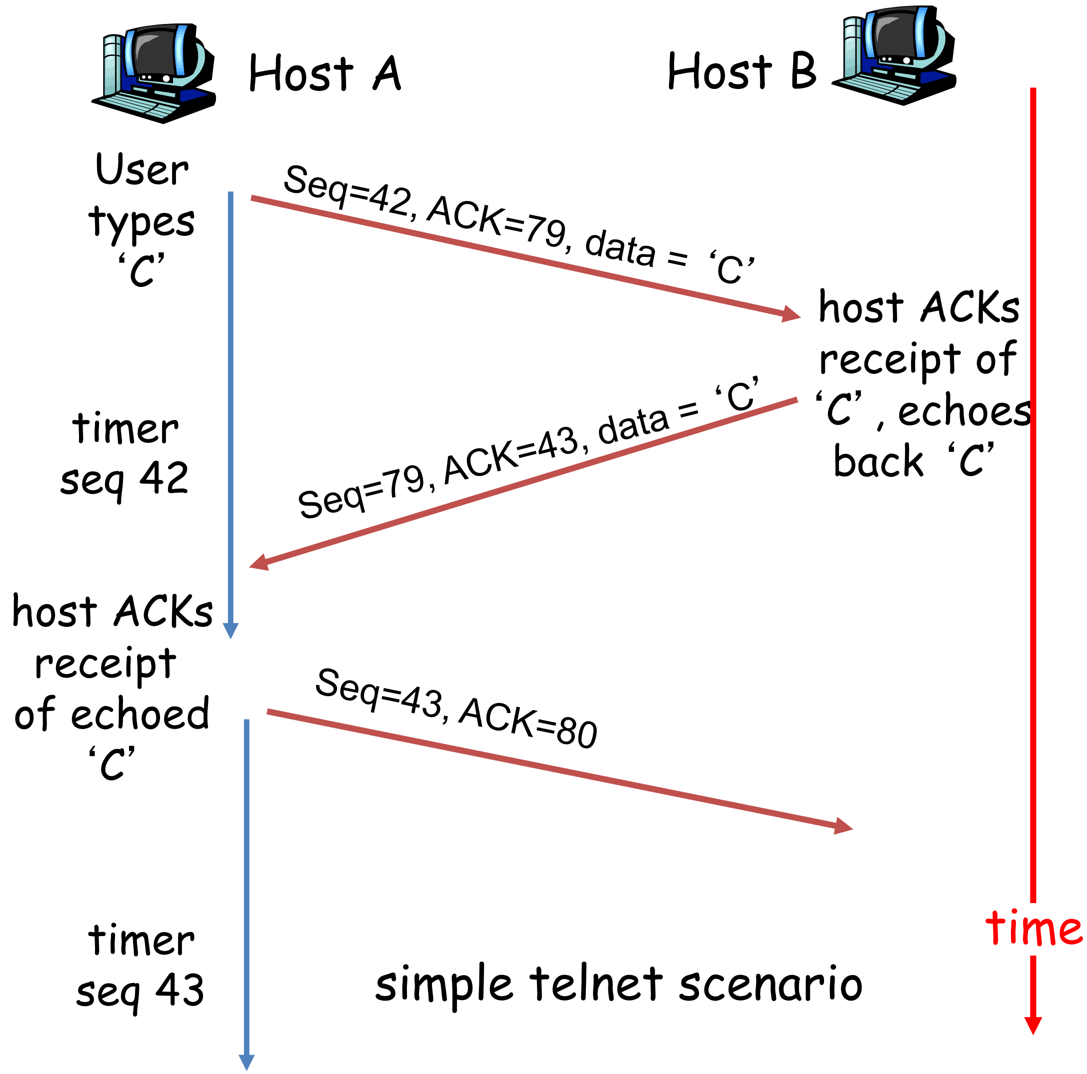
TCP

Seq #s and ACKs in TCP

Seq. #'s:
- byte stream "number" of **first byte** in segment's data

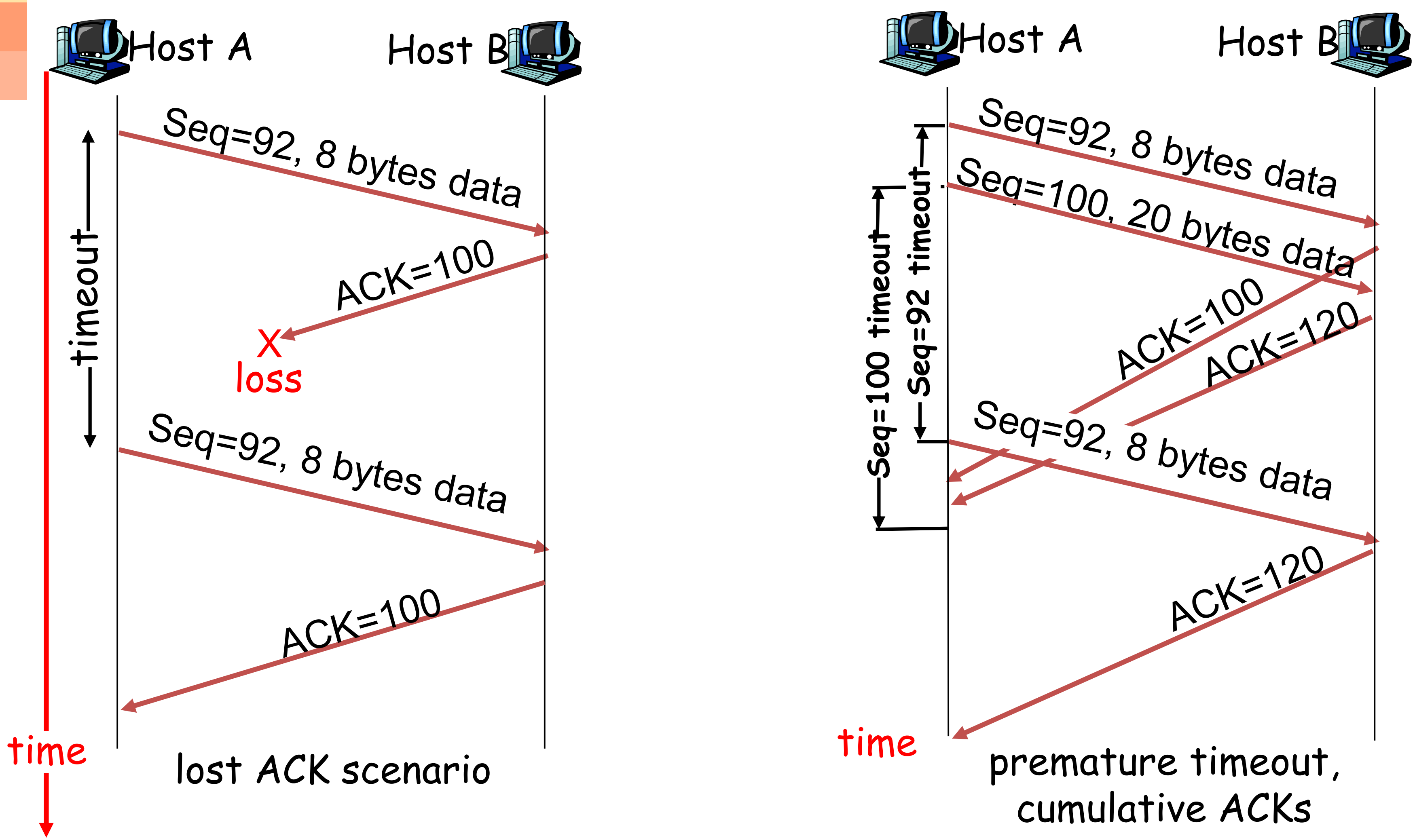
ACKs:
- seq # of next byte expected from other side
- cumulative ACK
- Piggybacked in data packets

Q: how receiver handles out-of-order segments
- A: TCP spec doesn't say, - up to implementor



simple telnet scenario

Retransmission Scenarios



How long should the timeout be?

- Timer should time out only rarely.
  - Ideally only when a segment is lost
  - Practically usually only when a segment is lost, may timeout on some long delays; but not average delays.
- The delay is expressed as a round trip time (RTT). Delay varies due to congestion and path. We need an estimate. Exponential Weighted Moving Average is used

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

Typically, existing estimate is weighted more (x = 0.125)

- To prevent delays from causing timeouts too frequently, the timer takes into consideration the deviation from the average.

```
Timeout = EstimatedRTT + 4 * Deviation
```

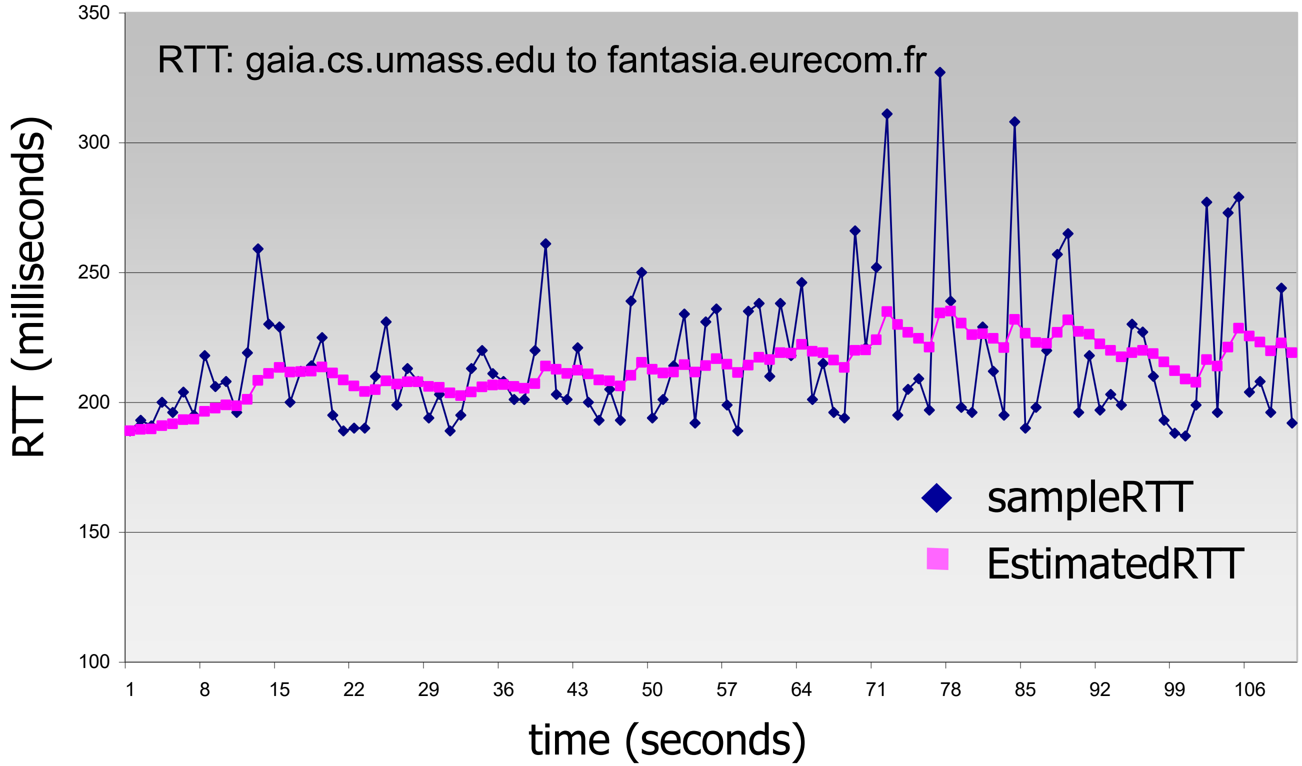
```
Deviation=(1-x)*Deviation+x*|SampleRTT-EstimatedRTT|
```

- Note that one ACK may acknowledge several segments, so calculating the timeout is non-trivial.

TCP round trip time, timeout

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

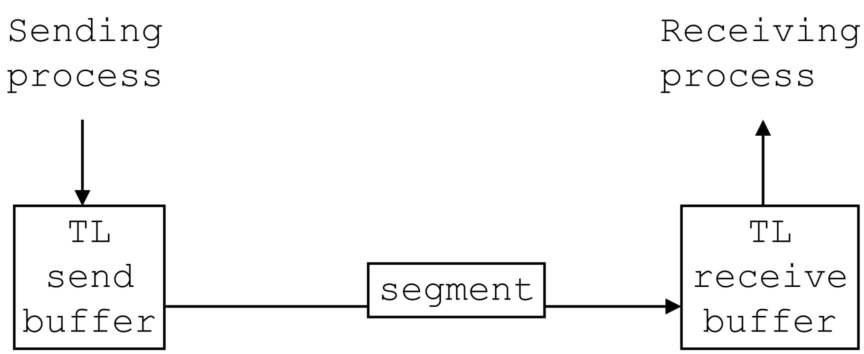
- exponential weighted moving average
- influence of past sample decreases exponentially fast
- typical value: $\alpha = 0.125$

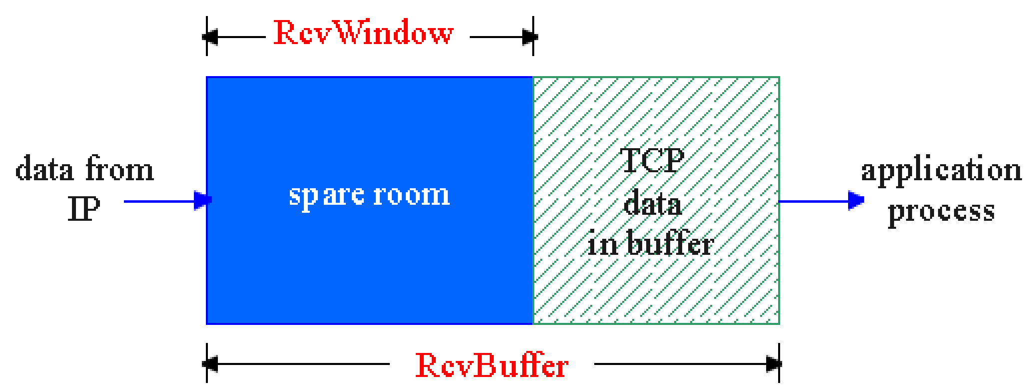


TCP - flow control

- **Flow control exists to prevent the sender from overwhelming the receiver.**
- Receiving host informs sender of the receive window size in the header of TCP segments (initially = receive buffer size)
- At sending host

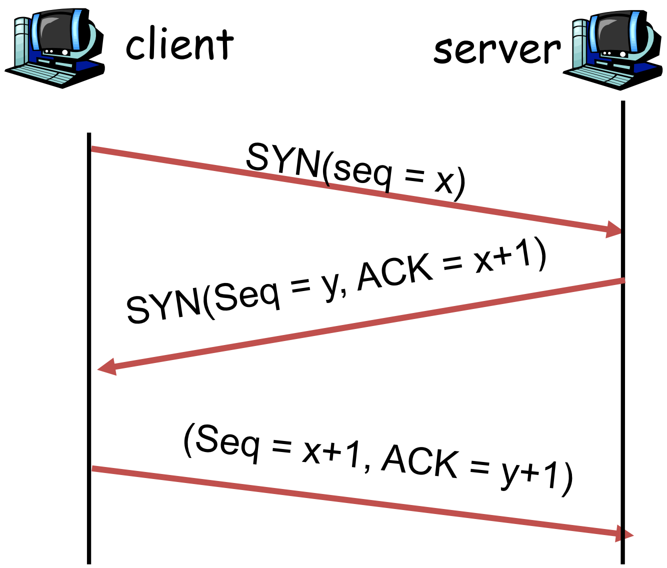
  LastByteSent - LastByteAcked <= Receive Window
- What if the receive buffer is full? Receive window = 0. How will the sender know when more space is available?





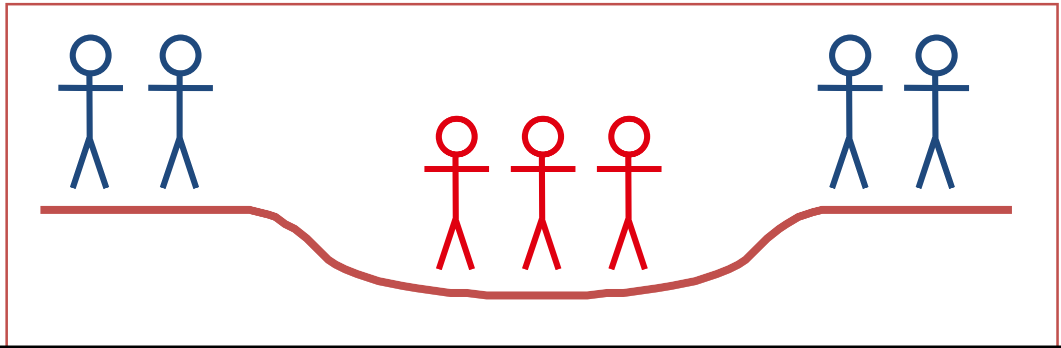
Establishing a Connection

- TCP connections are established by a 3-way handshake.
  - Request connection
  - Grant connection
  - Acknowledge
- Initial sequence numbers of the sender and receiver are exchanged as part of the connection establishment.
- A SYN bit in the header is set to 1 for the first two segments to indicate the set up.
- The third handshake is required to deal with duplicate segments/ACKs



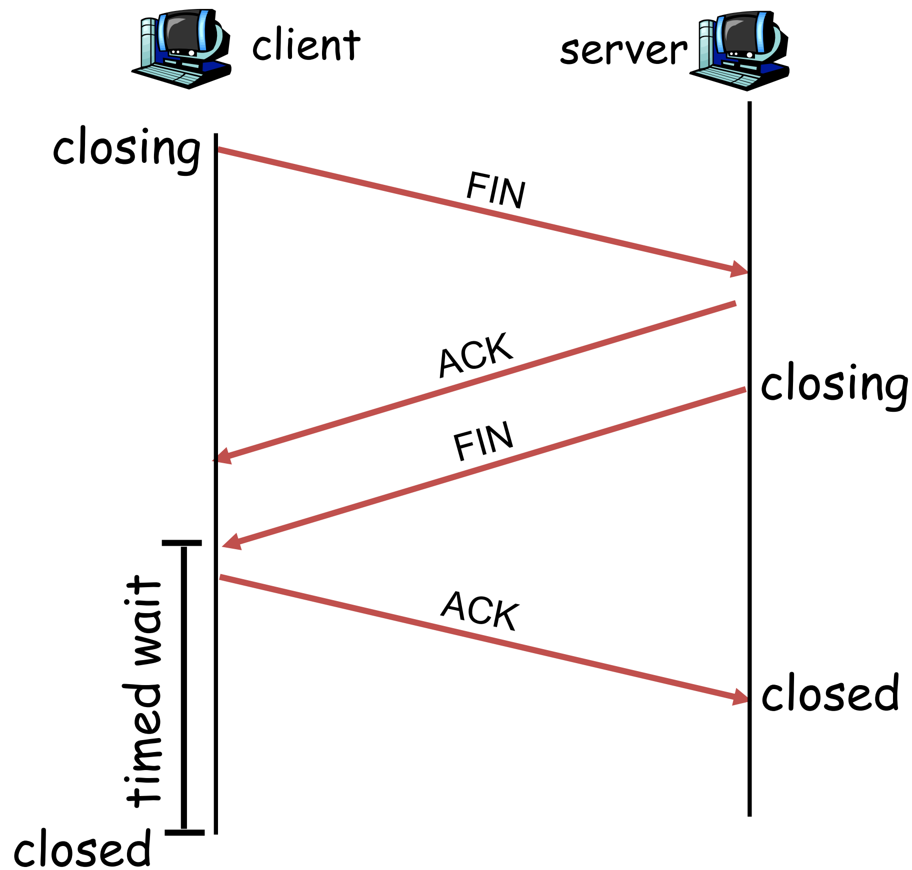
Terminating a Connection

- It is equally important to have an agreed termination of a connection
  - need to guarantee that all data has been delivered in both directions
- Such a guarantee is not possible
  (but assurance can be given with a high probability)
  - consider the problem in the diagram below
  - the blue armies (on the hills) are to attack the red army (in the valley)
  - the blue armies need to guarantee a coordinated attack



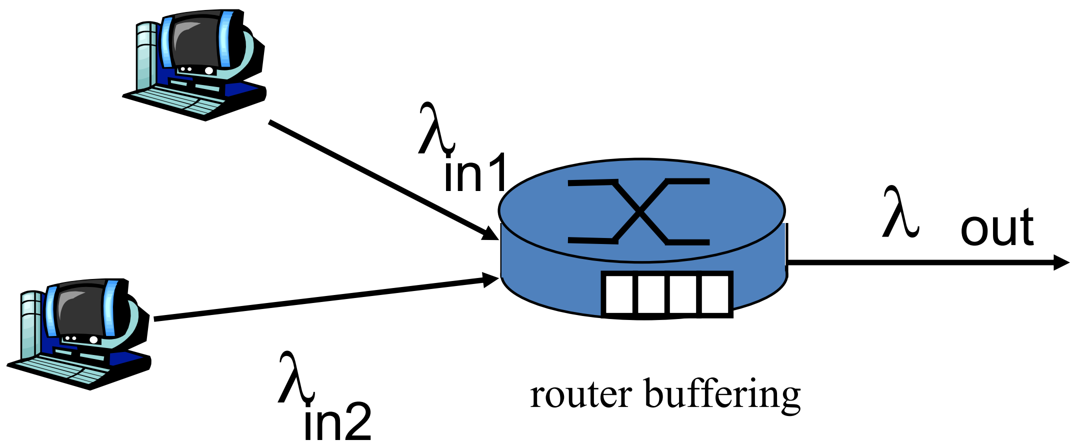
Terminating a connection in TCP

- timeouts at client and server guarantee that the connection will eventually be closed on both sides.
- timed wait increases the chance of server receiving a final ACK



Congestion

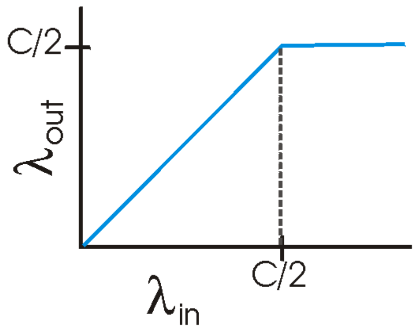
- Congestion can occur within the network. Too many sources and/or too much data being sent for the routers.

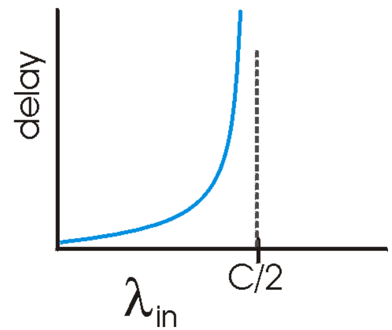


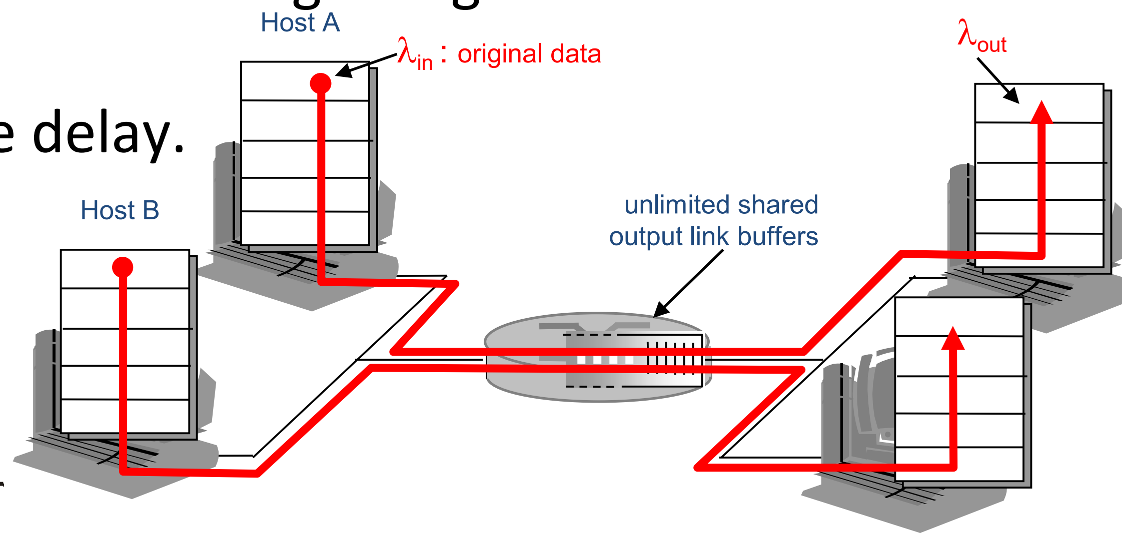
- Can result in
  - packet loss (buffer overflows)
  - high delays (queuing in buffer)

Reasons to avoid Congestion

- Throughput is capped at routers. Sending at higher rates has no benefit.
- Congestion increases average delay.







- Retransmission of packets, dropped or delayed, adds overhead to network. For delayed packets, this is wasted overhead.
- When a packet is dropped, **the effort of earlier routers**, is wasted.
- Performance degrades exponentially as congestion level approaches capacity. We want to avoid reaching this state!

Approaches to avoiding congestion

- Network layer assisted
  - Routers provide explicit feedback of congestion and/or available rate.
  - ATM-ABR, Explicit Congestion Notification (TCP/IP - proposed)
- End-to-end
  - End systems attempt to infer congestion based on packet acknowledgment times, etc.
  - Current TCP/IP implementations
- Network layer assisted increases complexity of routers
- End-to-end may make an incorrect inference of congestion (e.g. in mobile networks)

TCP congestion control - Tahoe 1988

- In addition to the Receive Window (for flow control) the senders rate is also controlled by a Congestion Window

  `LastByteSent - LastByteAcked <= min (ReceiveWin, CongWin)`

- Determining the congestion window
  - Initially CongWin = 1 Maximum Segment Size (MSS)
  - Set a threshold, ssthresh = 65535 bytes
  - While CongWin <= ssthresh
    - Send a segment
    - If ACK is received before timeout,
      CongWin = CongWin + 1 MSS
    - If timeout occurs,
      ssthresh = 1/2 min(ReceiveWin, CongWin), CongWin = 1 MSS
  - While CongWin > Thresh
    - Send a segment
    - If ACK is received before timeout,
      CongWin = CongWin + MSS * MSS/CongWin
    - If timeout occurs
      ssthresh = 1/2 min(ReceiveWin, CongWin), CongWin = 1 MSS

Slow Start

Congestion Avoidance

TCP Congestion Control & extensions

- Extensions
- Reno - 1990
  - Fast retransmit
    - 3 duplicate ACKs - retransmit immediately
  - Fast recovery
    - 3 duplicate ACKs - stay in congestion avoidance, don't re-enter slow start
- Vegas - 1995
  - detect congestion before packet loss.

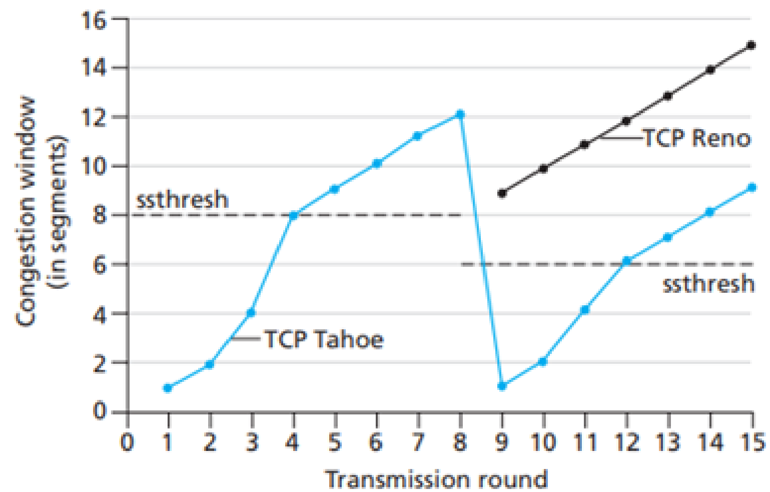


Figure 3.53 ♦ Evolution of TCP's congestion window (Tahoe and Reno)

TCP Congestion Control - open issues

- Under congestion conditions, TCP provides fair sharing of available throughput
  - When a segment is not ACK'ed, the Threshold of senders will converge
    Send1 = 20/2 = 10   Send2 = 10/2 = 5
  - Only fair if each application has same number of TCP streams open!
- TCP doesn't spend much time in slow start (due to exponential increase) unless
  - Transmission rates are high relative to latency (not getting enough ACKs back to grow window size)
  - Object being sent is small (not enough time to escape slow start)
- Somewhat problematic for the web.
- UDP is often used to *avoid* TCP congestion control. This will be a problem if UDP traffic becomes more prevalent.
- TCP congestion control is still an active area of research with many variations on the basic Reno protocol.