Computer Networks

PhD. Saúl Pomares Hernández

Transport Layer

Transport Layer

Our goals:

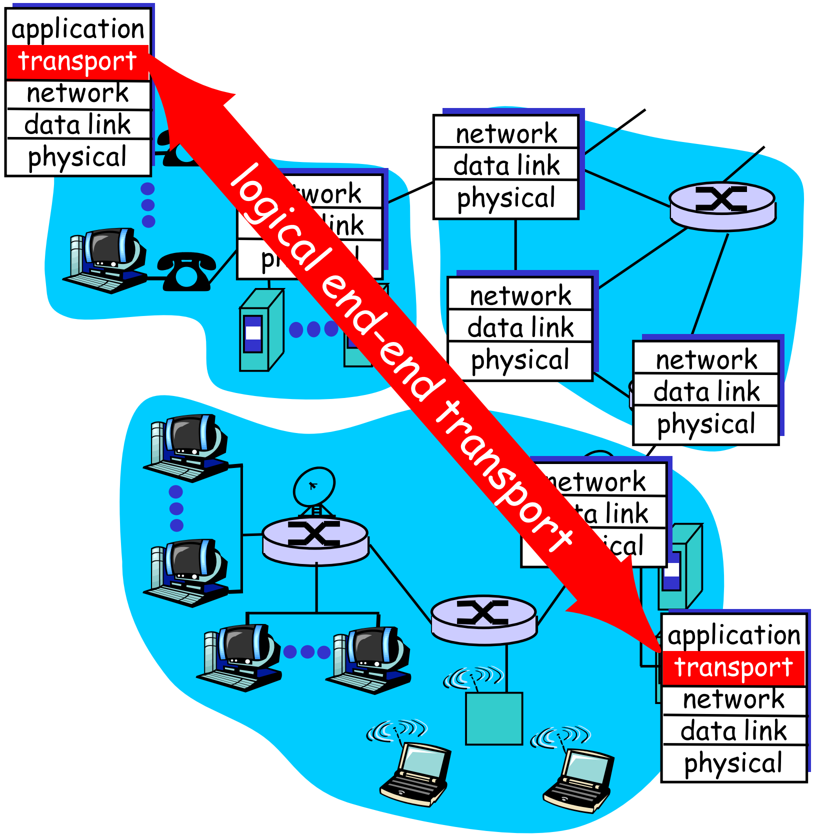
- understand principles behind transport layer services:
  - multiplexing/demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- learn about transport layer protocols in the Internet:
  - UDP: connectionless transport
  - TCP: connection-oriented transport
  - TCP congestion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

Outline

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- Principles of congestion control
- TCP congestion control

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
  - rcv 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

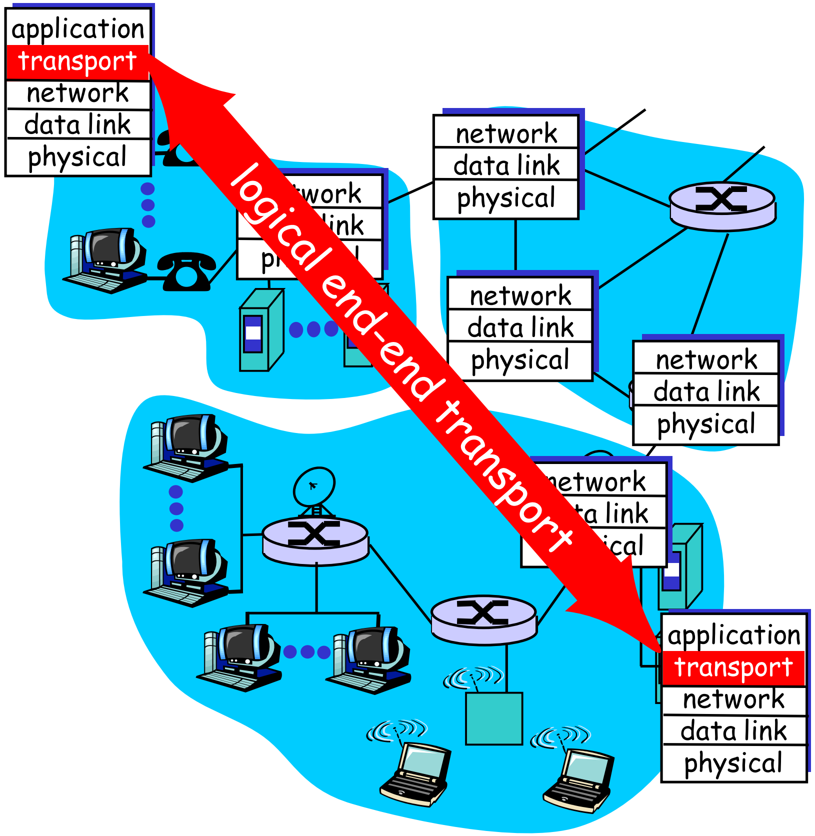
Household analogy:

12 kids sending letters to 12 kids

- processes = kids
- app messages = letters in envelopes
- hosts = houses
- transport protocol = Ann and Bill
- 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

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- Principles of congestion control
- TCP congestion control

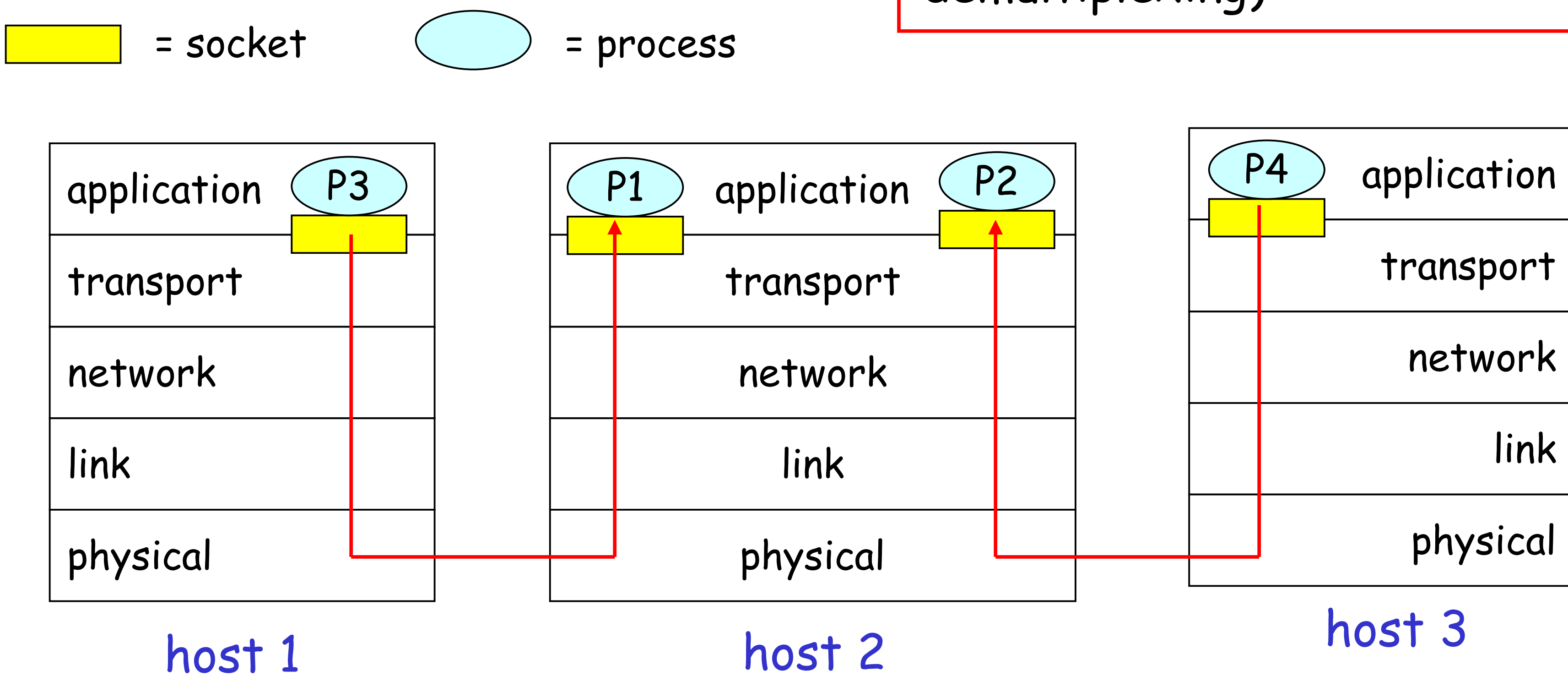
Multiplexing/demultiplexing

Demultiplexing at rcv host:

delivering received segments to correct socket

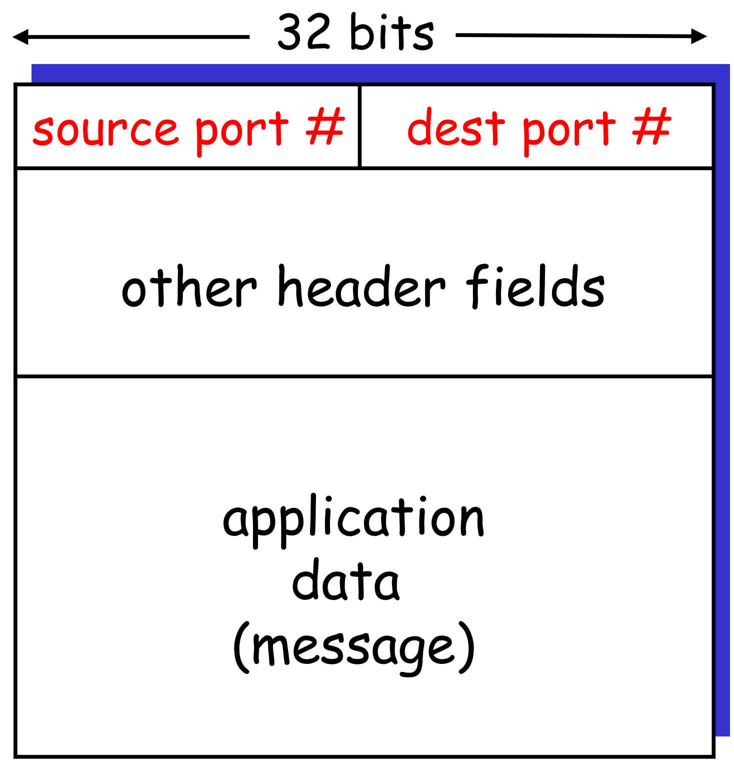
Multiplexing at send host:

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



How demultiplexing works

- host receives IP datagrams
  - each datagram has source IP address, destination IP address
  - each datagram carries 1 transport-layer segment
  - each segment has source, destination port number (recall: well-known port numbers for specific applications)
- host uses IP addresses & port numbers to direct segment to appropriate socket



TCP/UDP segment format

Connectionless demultiplexing

- Create sockets with port numbers:

```
DatagramSocket mySocket1 = new
   DatagramSocket(99111);
DatagramSocket mySocket2 = new
   DatagramSocket(99222);
```

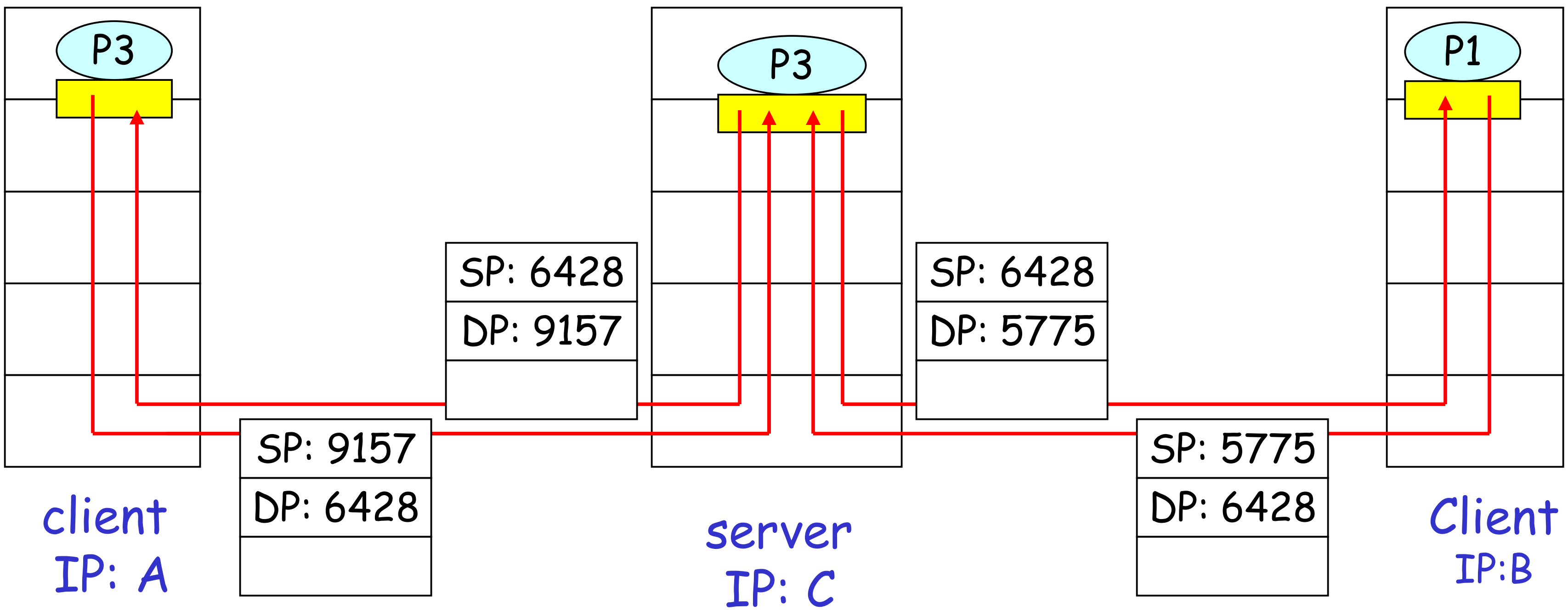
- UDP socket identified by two-tuple:

(dest IP address, dest port number)

- When host receives UDP segment:
  - checks destination port number in segment
  - directs UDP segment to socket with that port number
- IP datagrams with different source IP addresses and/or source port numbers directed to same socket

Connectionless demux (cont)

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

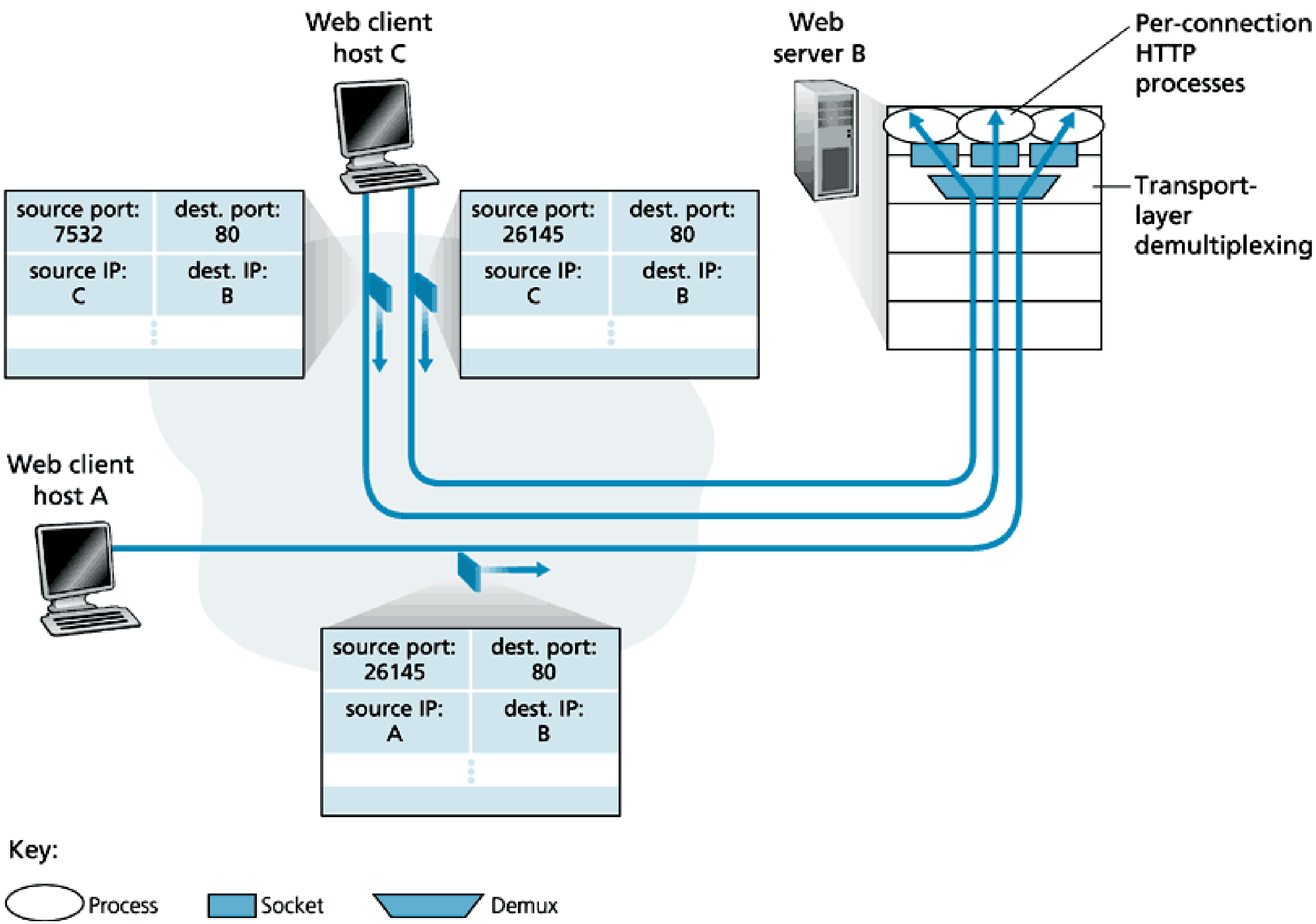


SP provides "return address"

Connection-oriented demux

- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- recv host 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 (cont)



Outline

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- Principles of congestion control
- TCP congestion control

UDP: User Datagram Protocol [RFC 768]

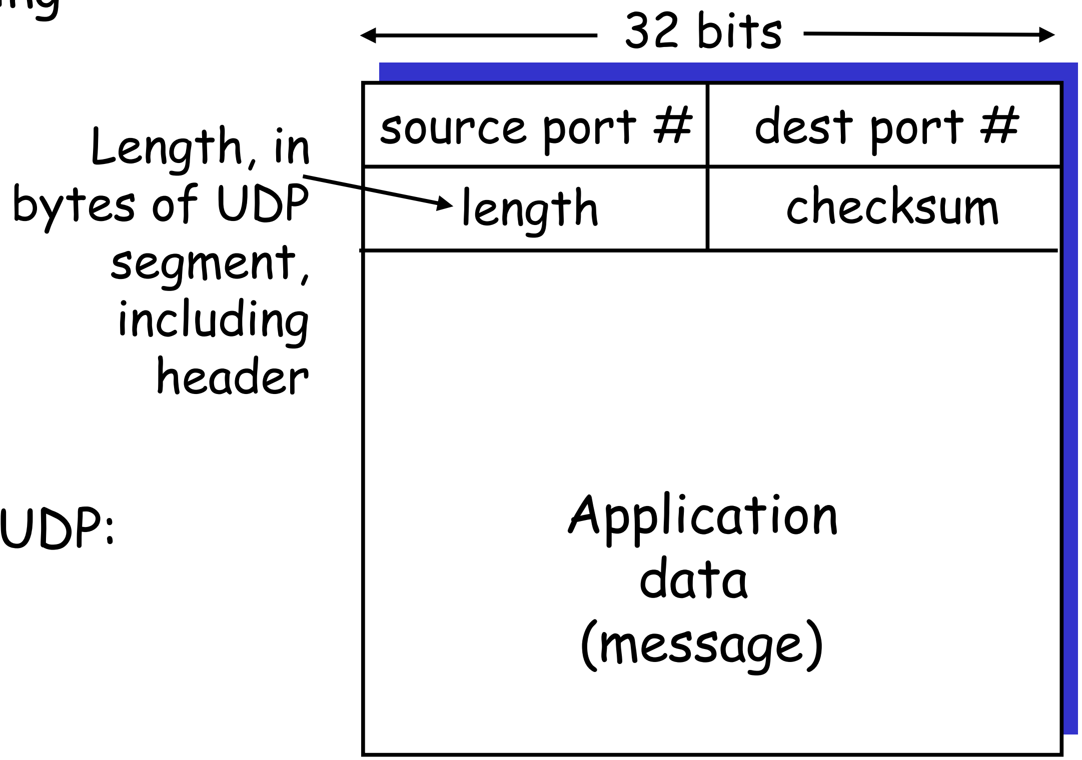
- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out of order to app
- *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

Why is there a UDP?

- no connection establishment (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

UDP: more

- 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!



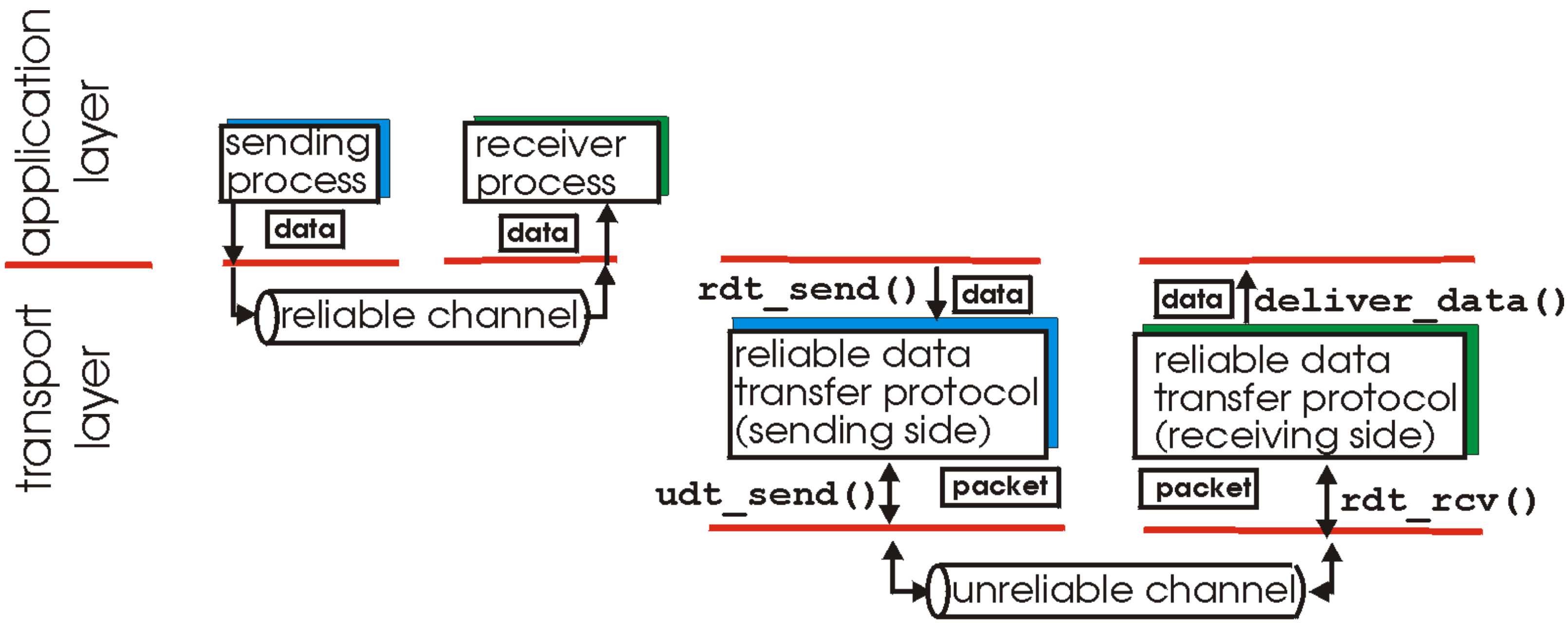
UDP segment format

Outline

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- Principles of congestion control
- TCP congestion control

Principles of Reliable data transfer

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



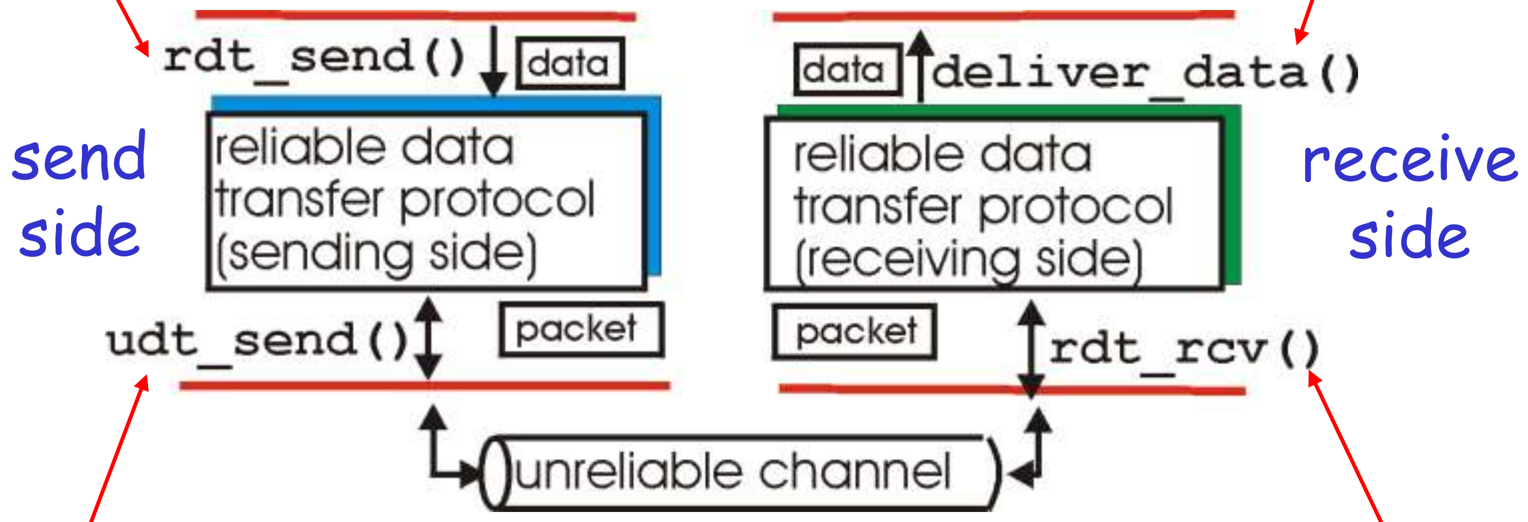
(a) provided service (b) service implementation

- 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

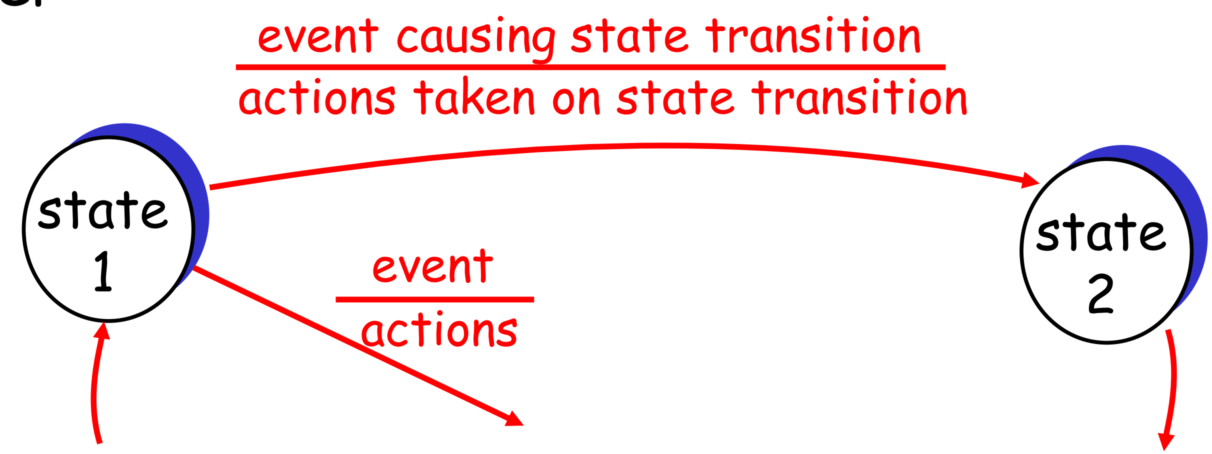
rdt_rcv(): called when packet arrives on rcv-side of channel

Reliable data transfer: getting started

We'll:

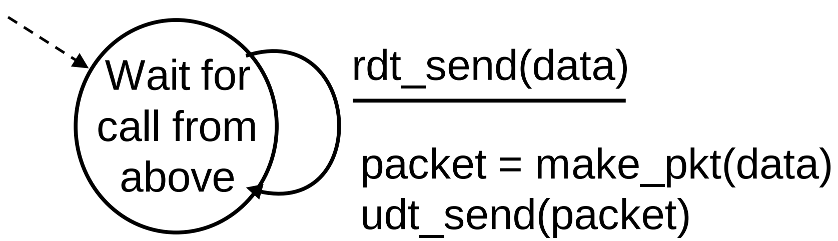
- incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- consider only unidirectional data transfer
  - but control info will flow on both directions!
- use finite state machines (FSM) to specify sender, receiver

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

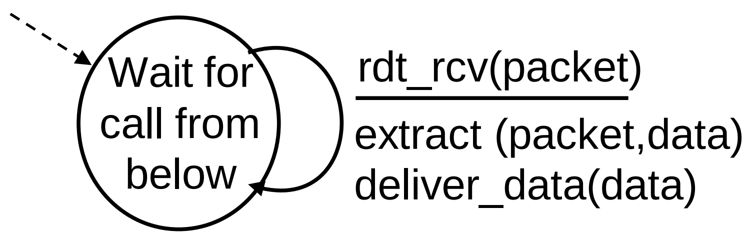


Rdt1.0: reliable transfer over a reliable channel

- underlying channel perfectly reliable
  - no bit errors
  - no loss of packets
- separate FSMs for sender, receiver:
  - sender sends data into underlying channel
  - receiver read data from underlying channel



sender

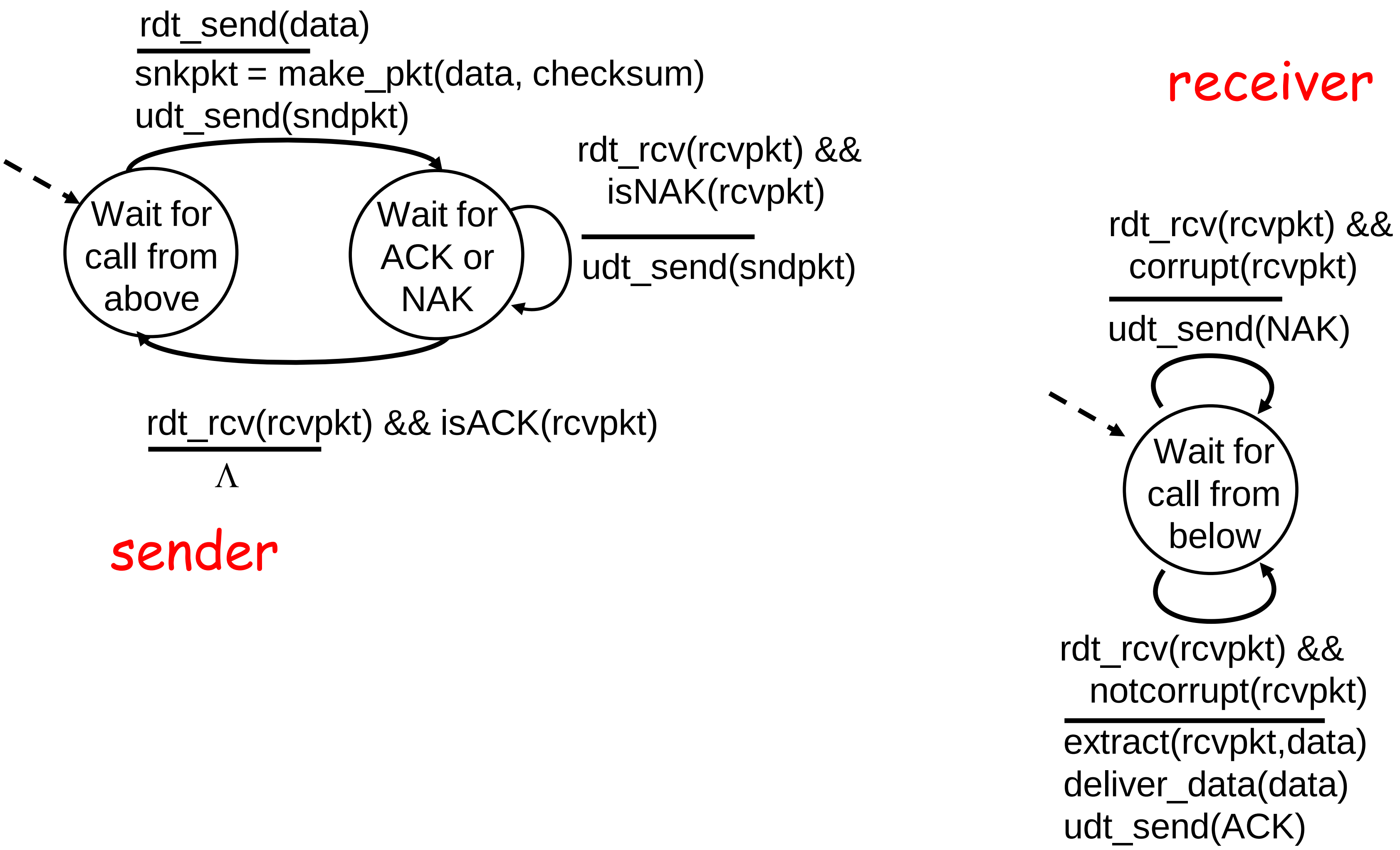


receiver

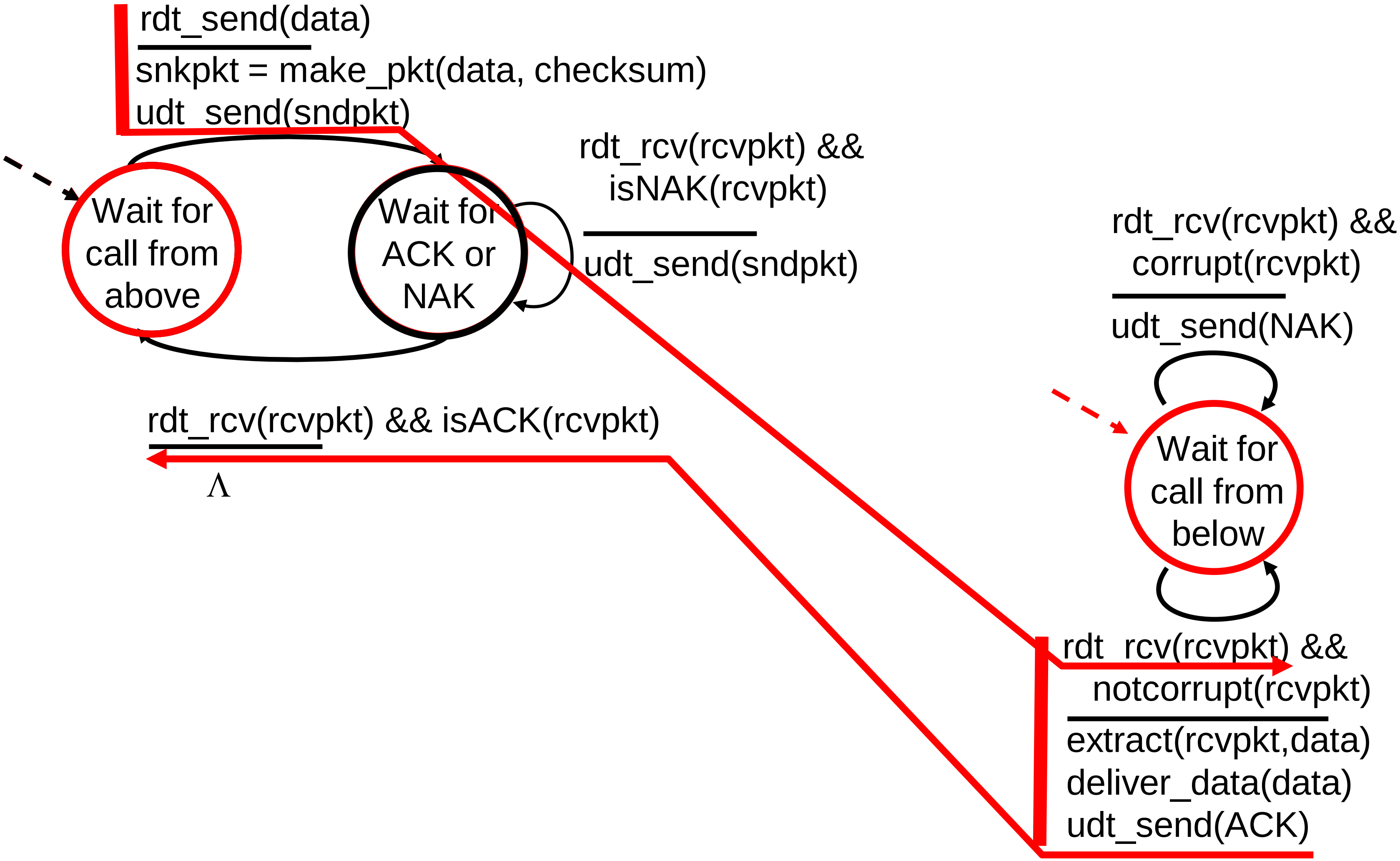
Rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
  - recall: UDP 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
  - human scenarios using ACKs, NAKs?
- new mechanisms in `rdt2.0` (beyond `rdt1.0`):
  - error detection
  - receiver feedback: control msgs (ACK,NAK) rcvr->sender

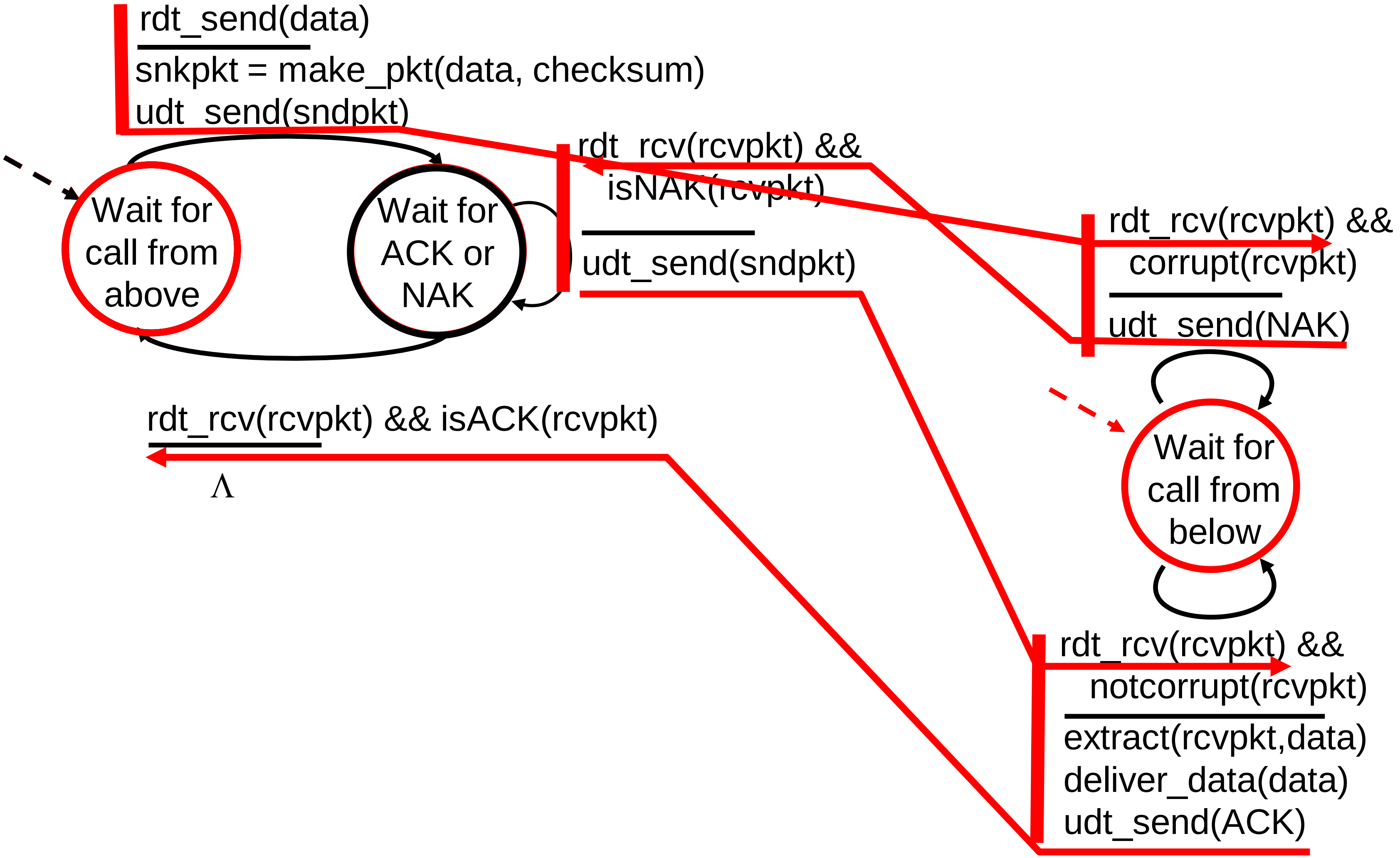
rdt2.0: FSM specification



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

What to do?

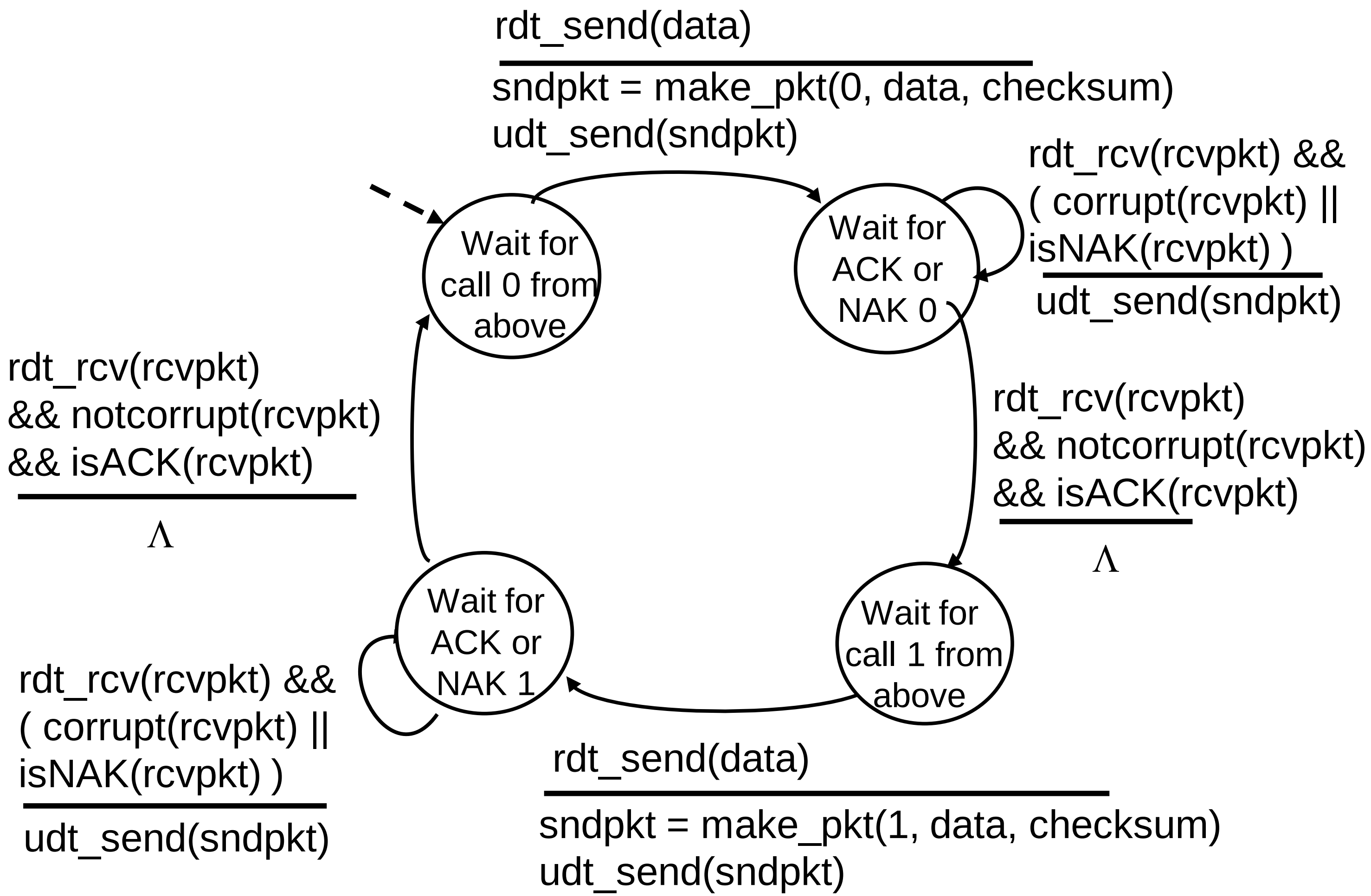
- sender ACKs/NAKs receiver's ACK/NAK? What if sender ACK/NAK lost?
- retransmit, but this might cause retransmission of correctly received pkt!

Handling duplicates:

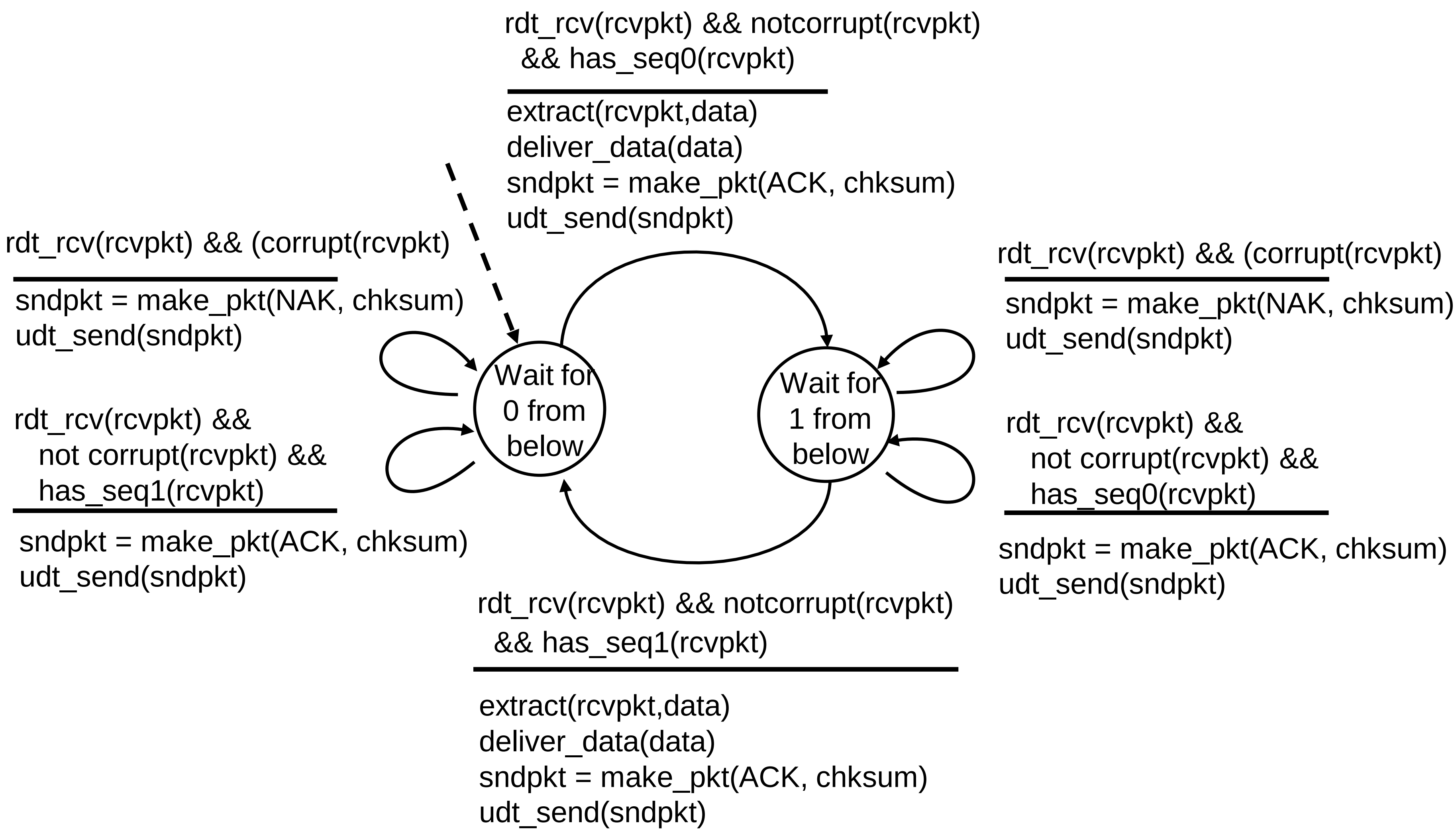
- sender adds *sequence number* to each pkt
- sender retransmits current pkt if ACK/NAK garbled
- receiver discards (doesn't deliver up) duplicate pkt

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 "current" pkt has 0 or 1 seq. #

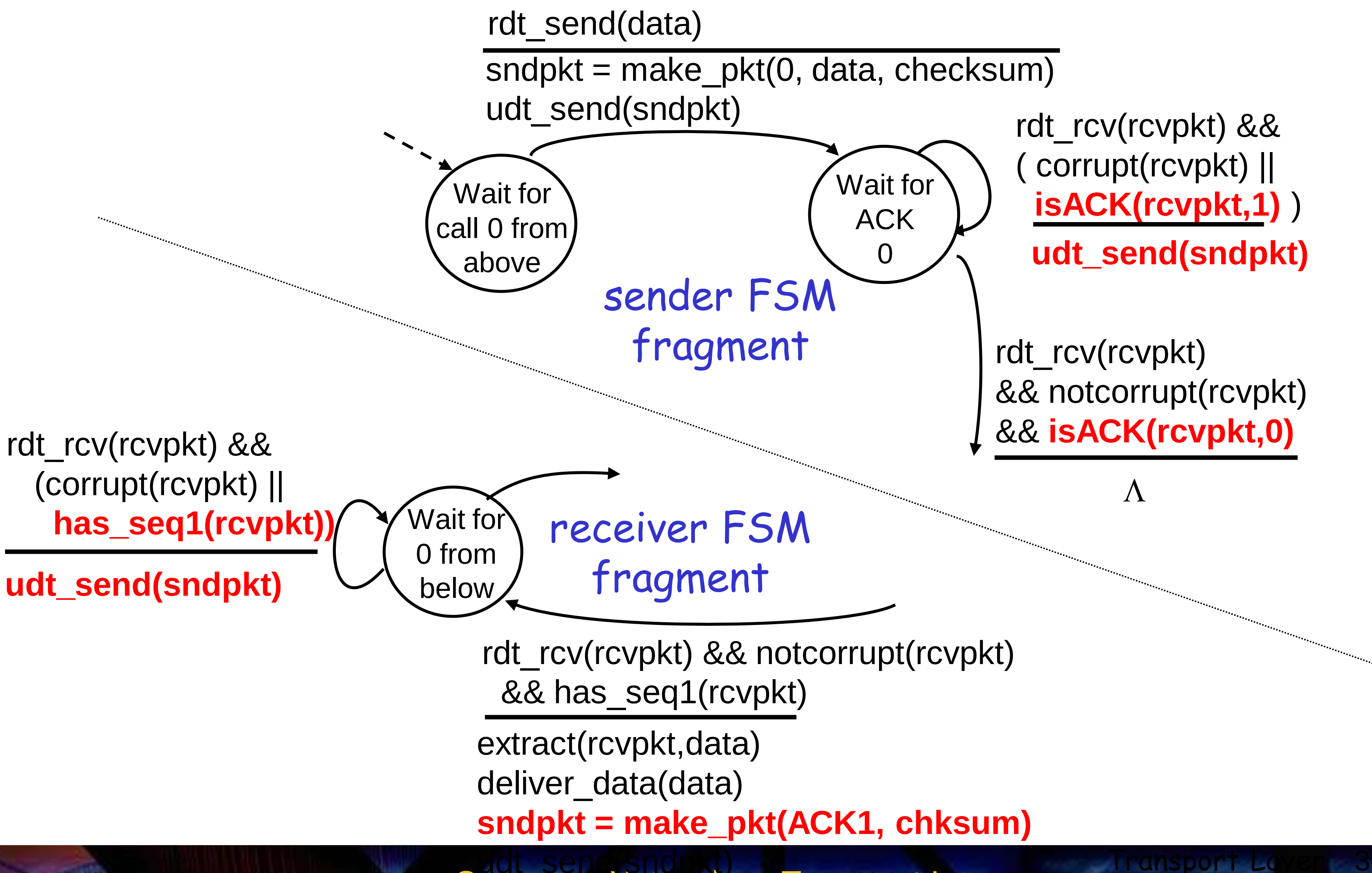
Receiver:

- 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 NAKs 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 or ACKs)

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

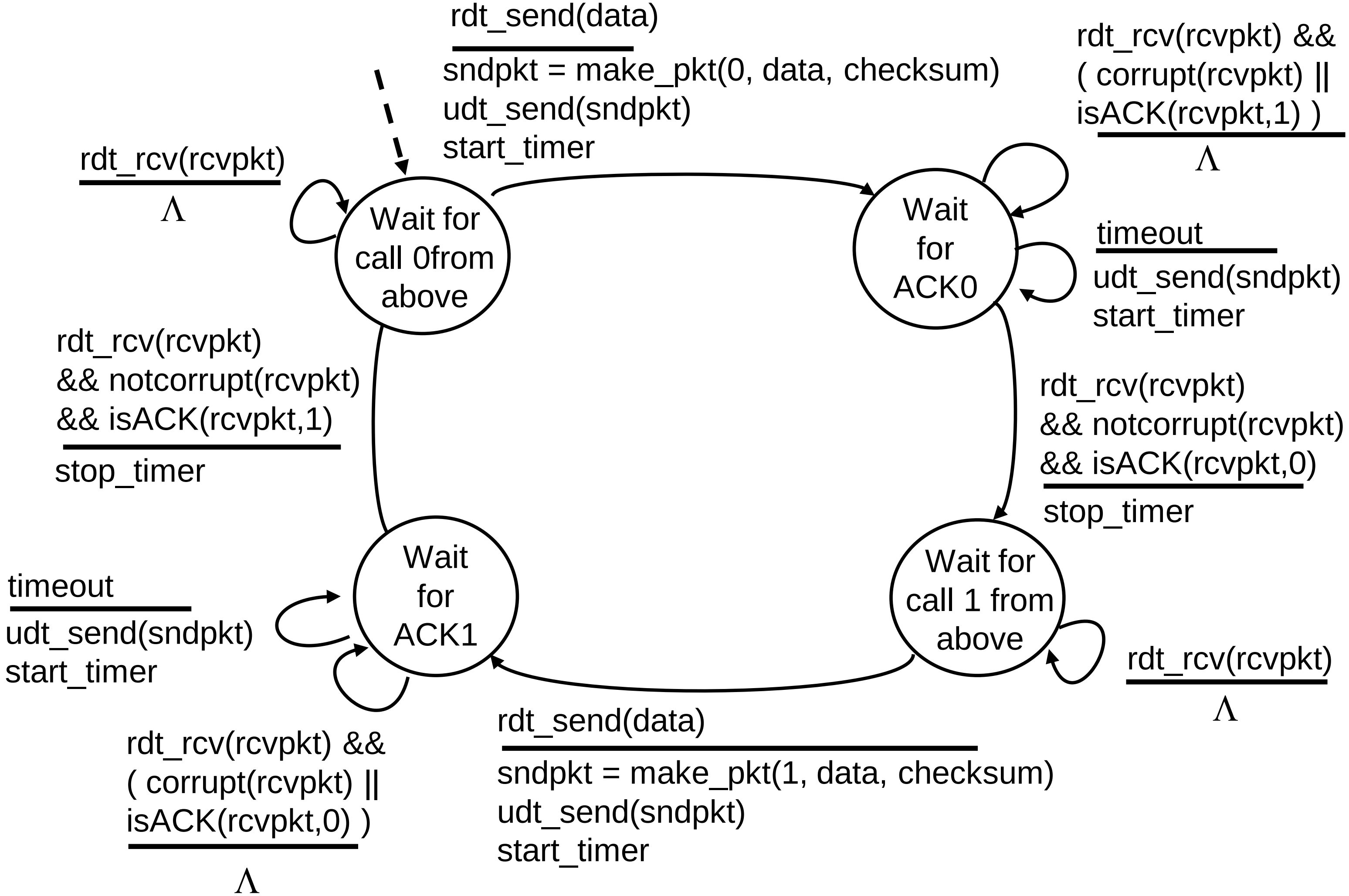
Q: how to deal with loss?

- sender waits until certain data or ACK lost, then retransmits
- yuck: drawbacks?

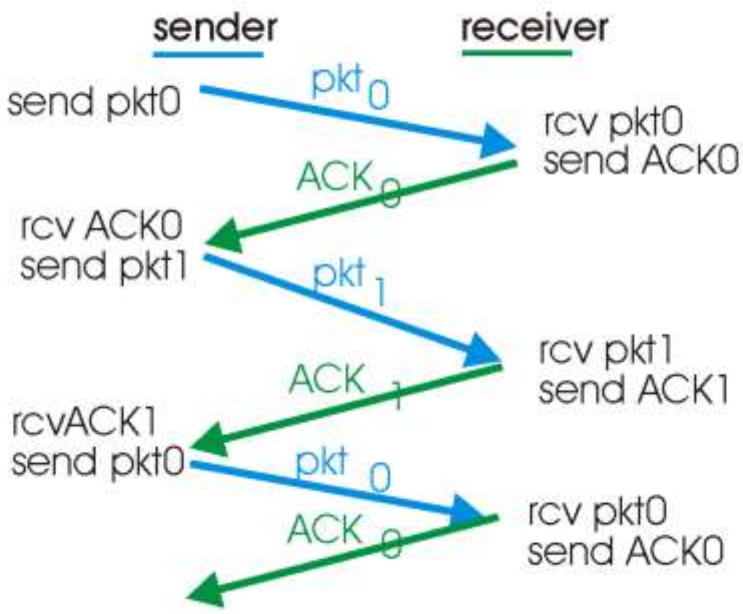
Approach: sender waits "reasonable" amount of time for ACK

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

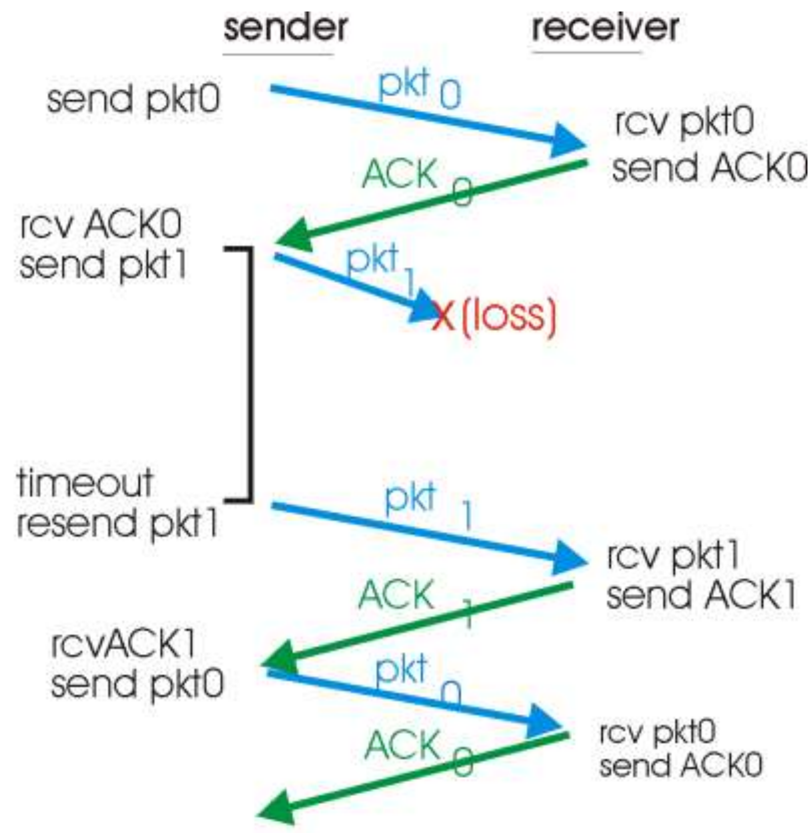
rdt3.0 sender



rdt3.0 in action

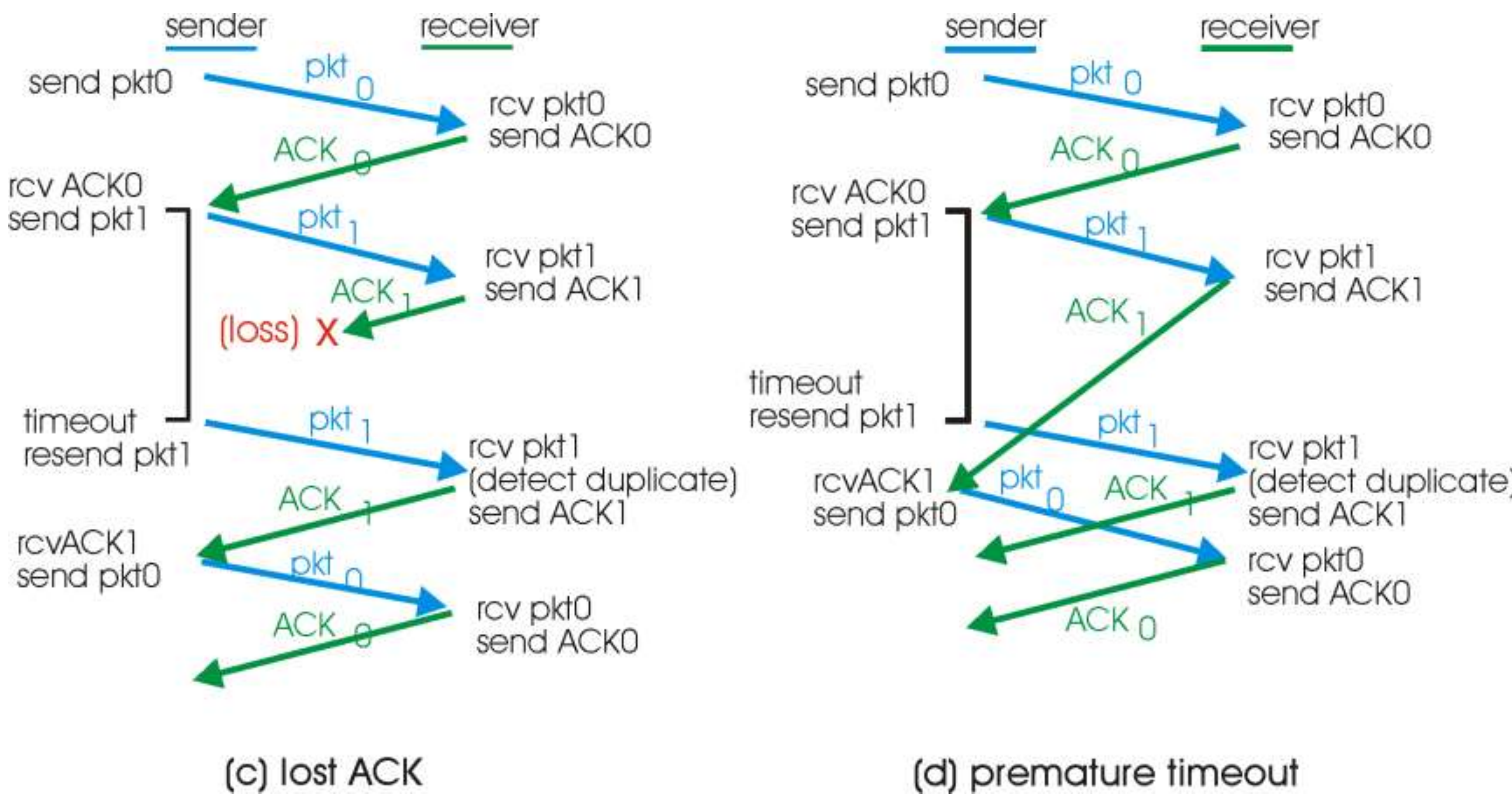


(a) operation with no loss



(b) lost packet

rdt3.0 in action



Performance of rdt3.0

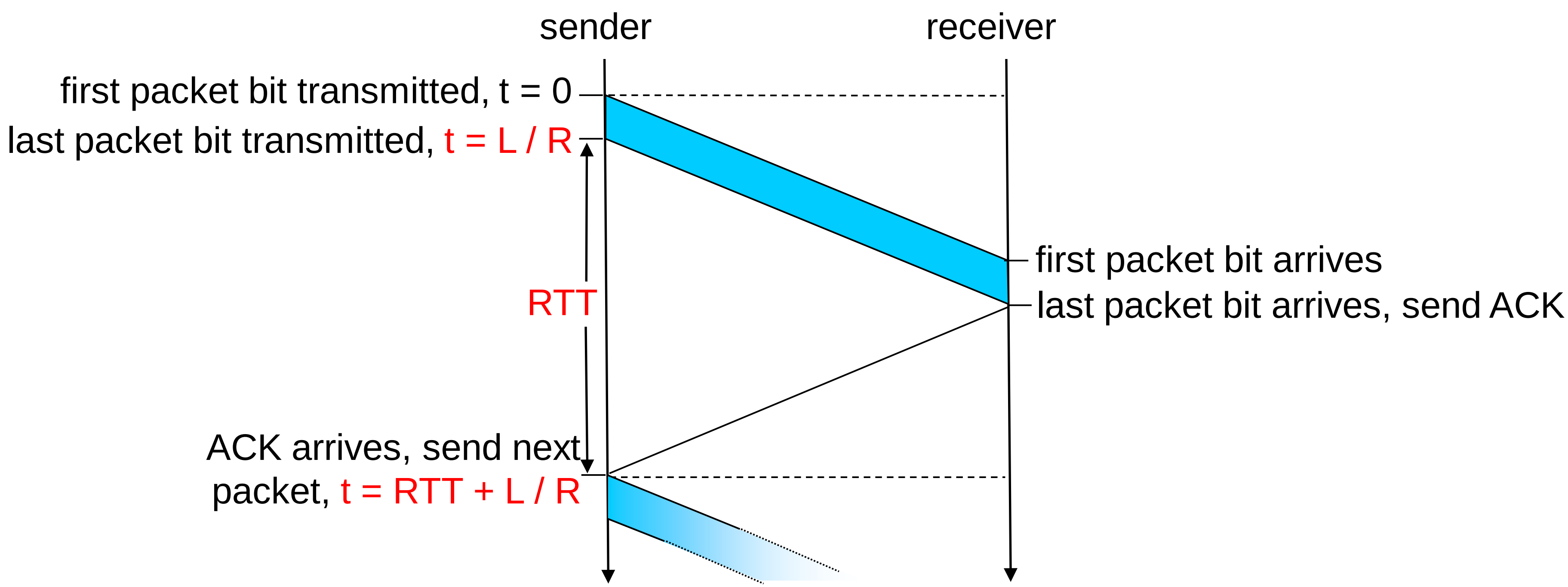
- rdt3.0 works, but performance stinks
- example: 1 Gbps link, 15 ms e-e prop. delay, 1KB packet:

$$T_{transmit} = \frac{L \text{ (packet length in bits)}}{R \text{ (transmission rate, bps)}} = \frac{8\text{kb/pkt}}{10^{**}9 \text{ b/sec}} = 8 \text{ microsec}$$

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

  - U_{sender}: Utilization = fraction of time sender busy sending
  - 1KB pkt every 30 msec -> 33kB/sec throughput over 1 Gbps link
  - network protocol limits use of physical resources!

rdt3.0: stop-and-wait operation

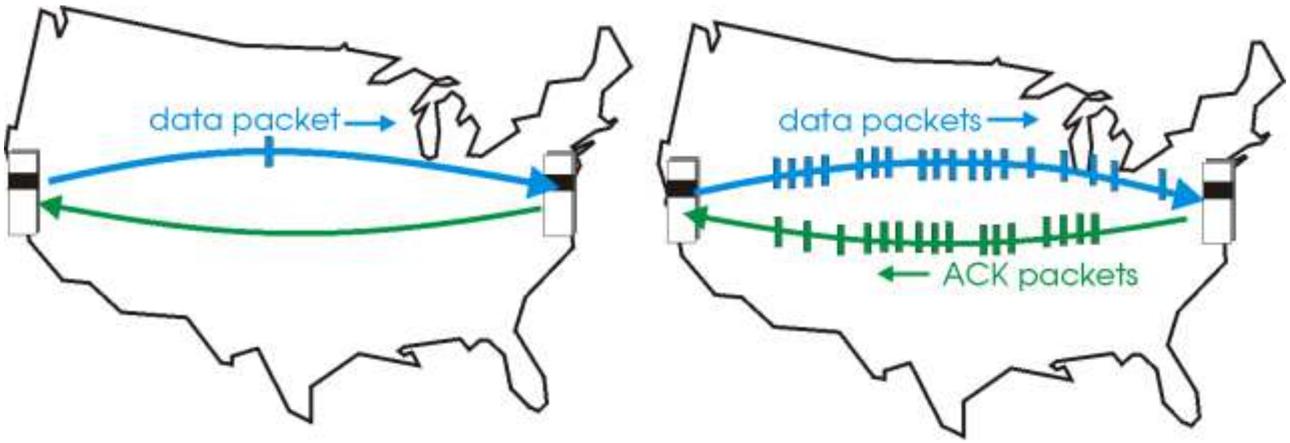


$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

Pipelined protocols

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

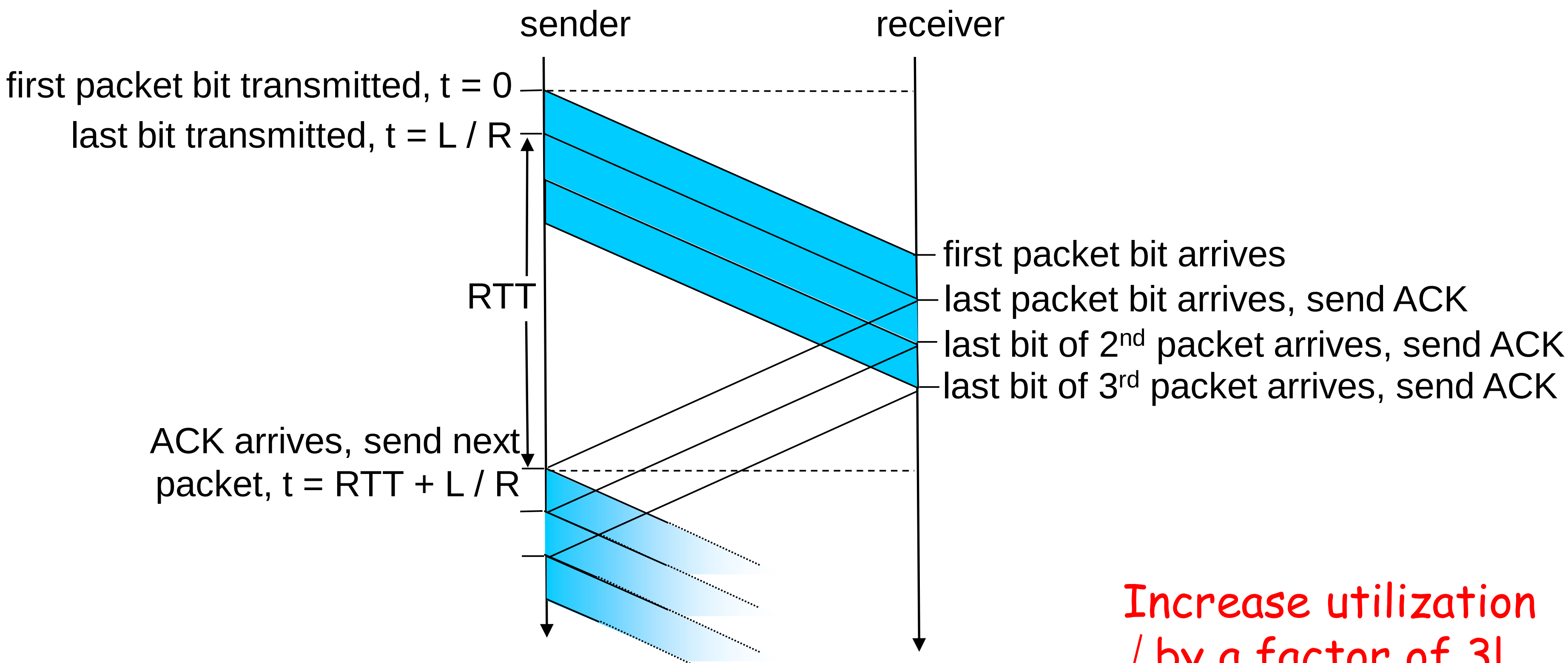
- range of sequence numbers must be increased
- buffering at sender and/or receiver



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

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

Pipelining: increased utilization



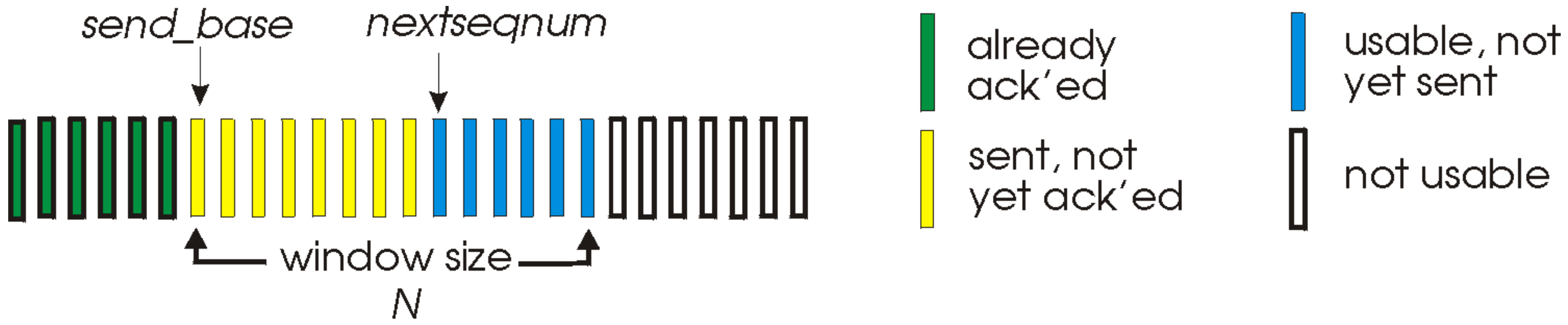
Increase utilization by a factor of 3!

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

Go-Back-N

Sender:

- k-bit seq # in pkt header
- "window" of up to N, consecutive unack'ed pkts allowed

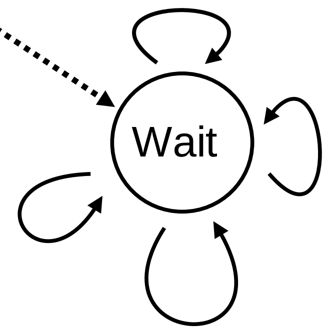


- ACK(n): ACKs all pkts up to, including seq # n - "cumulative ACK"
  - may deceive duplicate ACKs (see receiver)
- timer for each in-flight pkt
- *timeout(n):* retransmit pkt n and all higher seq # pkts in window

GBN: sender extended FSM

```
rdt_send(data)
_____________________________
if (nextseqnum < base+N) {
    sndpkt[nextseqnum] = make_pkt(nextseqnum,data,chksum)
    udt_send(sndpkt[nextseqnum])
    if (base == nextseqnum)
       start_timer
    nextseqnum++
    }
else
  refuse_data(data)
```

```
Λ
_____________
base=1
nextseqnum=1
```

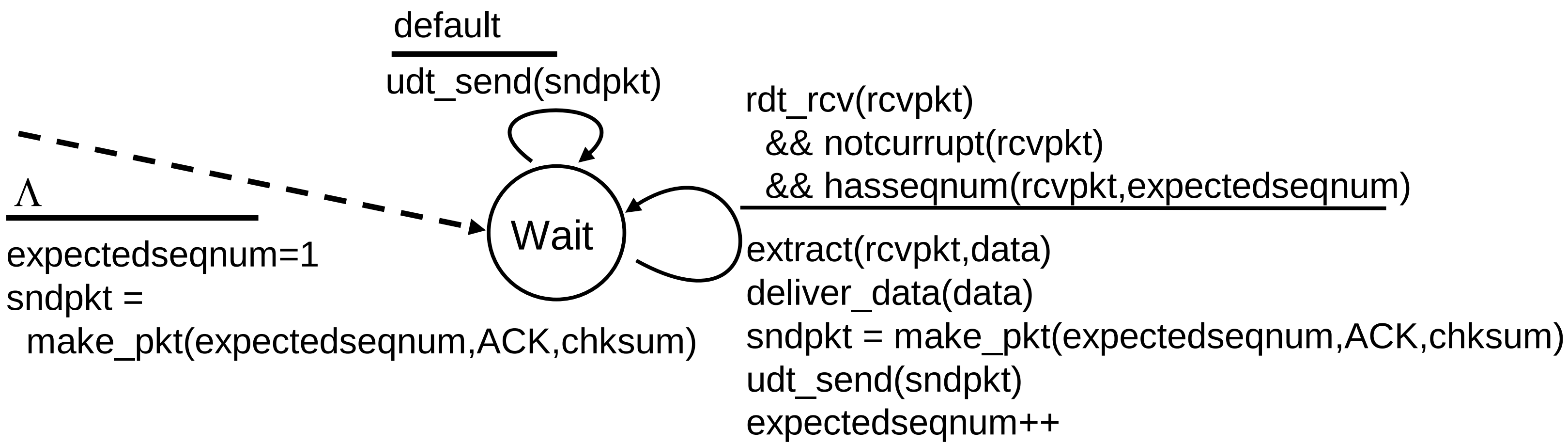



```
rdt_rcv(rcvpkt)
   && corrupt(rcvpkt)
_____________________
```

```
timeout
______________________
start_timer
udt_send(sndpkt[base])
udt_send(sndpkt[base+1])
…
udt_send(sndpkt[nextseqnum-1])
```

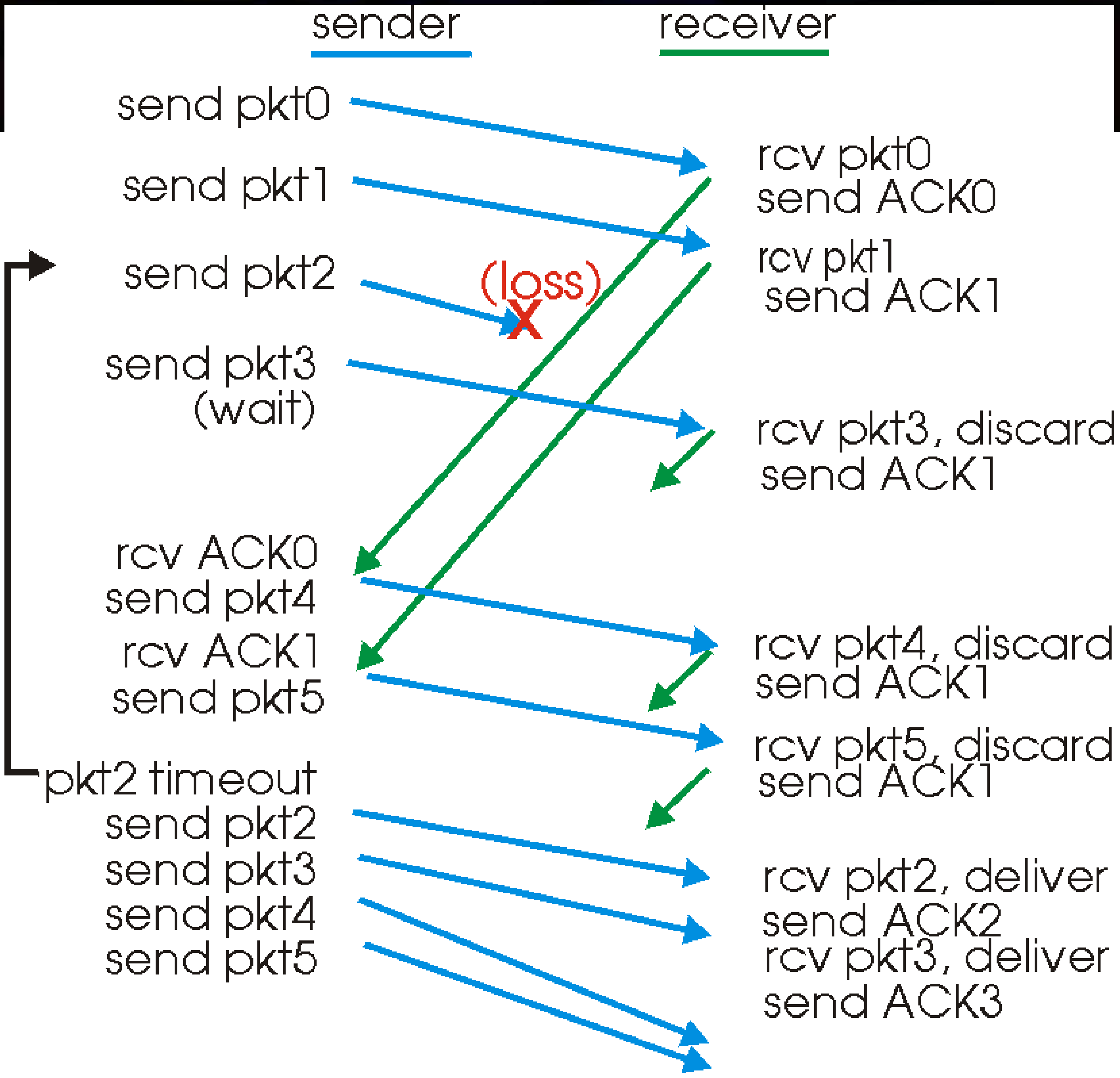
```
rdt_rcv(rcvpkt) &&
   notcorrupt(rcvpkt)
_________________________
base = getacknum(rcvpkt)+1
If (base == nextseqnum)
    stop_timer
  else
    start_timer
```

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**

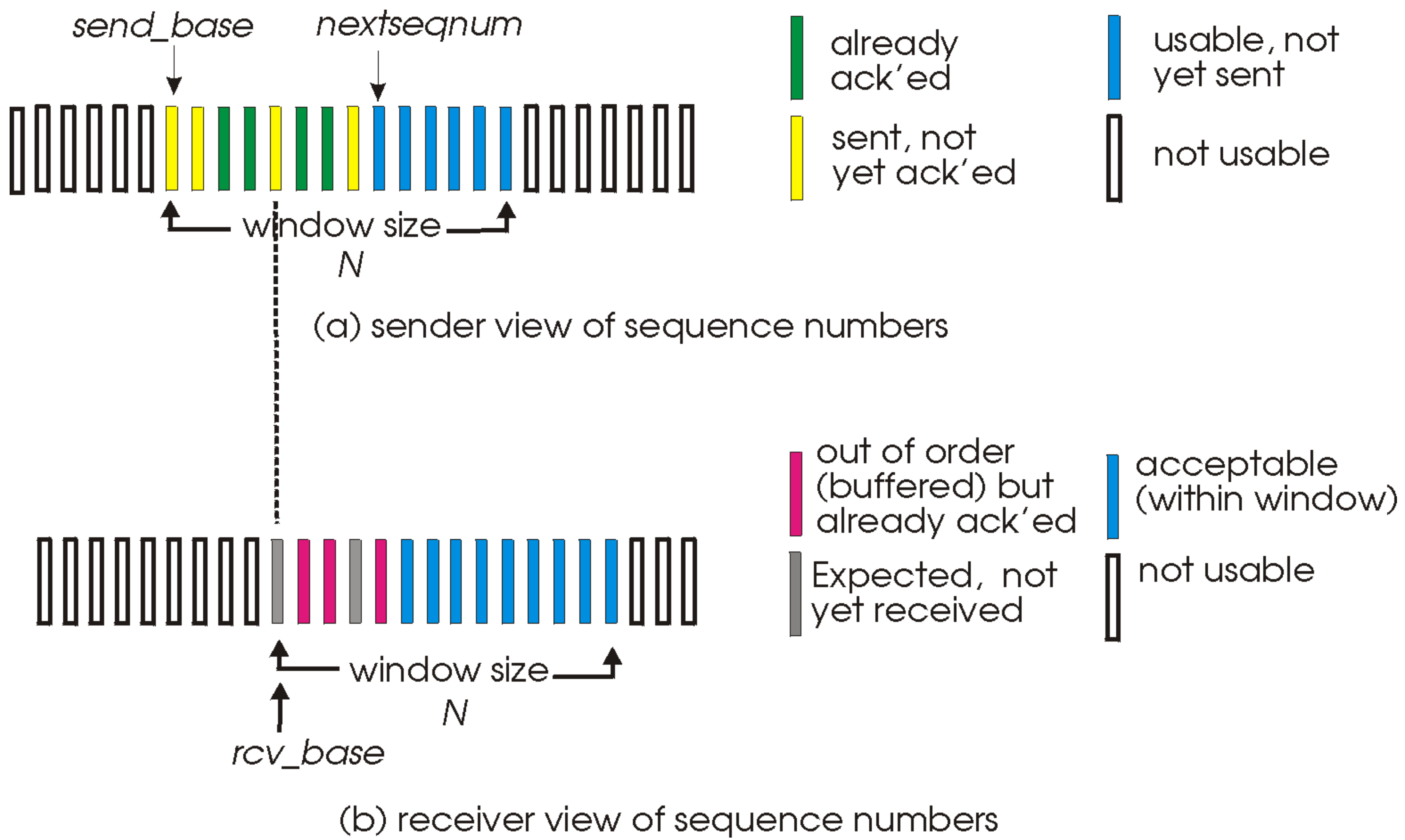
- out-of-order pkt:
  - discard (don't buffer) -> no receiver buffering!
  - Re-ACK pkt with highest in-order seq #


sender
receiver
send pkt0
send pkt1
send pkt2
(loss)
X
send pkt3
(wait)
rcv pkt0
send ACK0
rcv pkt1
send ACK1
rcv pkt3, discard
send ACK1
rcv ACK0
send pkt4
rcv ACK1
send pkt5
rcv pkt4, discard
send ACK1
rcv pkt5, discard
send ACK1
pkt2 timeout
send pkt2
send pkt3
send pkt4
send pkt5
rcv pkt2, deliver
send ACK2
rcv pkt3, deliver
send ACK3

Selective Repeat

- receiver *individually* acknowledges all correctly received pkts
  - buffers pkts, as needed, for eventual in-order delivery to upper layer
- sender only resends pkts for which ACK not received
  - sender timer for each unACKed pkt
- sender window
  - N consecutive seq #'s
  - again limits seq #s of sent, unACKed pkts

Selective repeat: sender, receiver windows



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 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

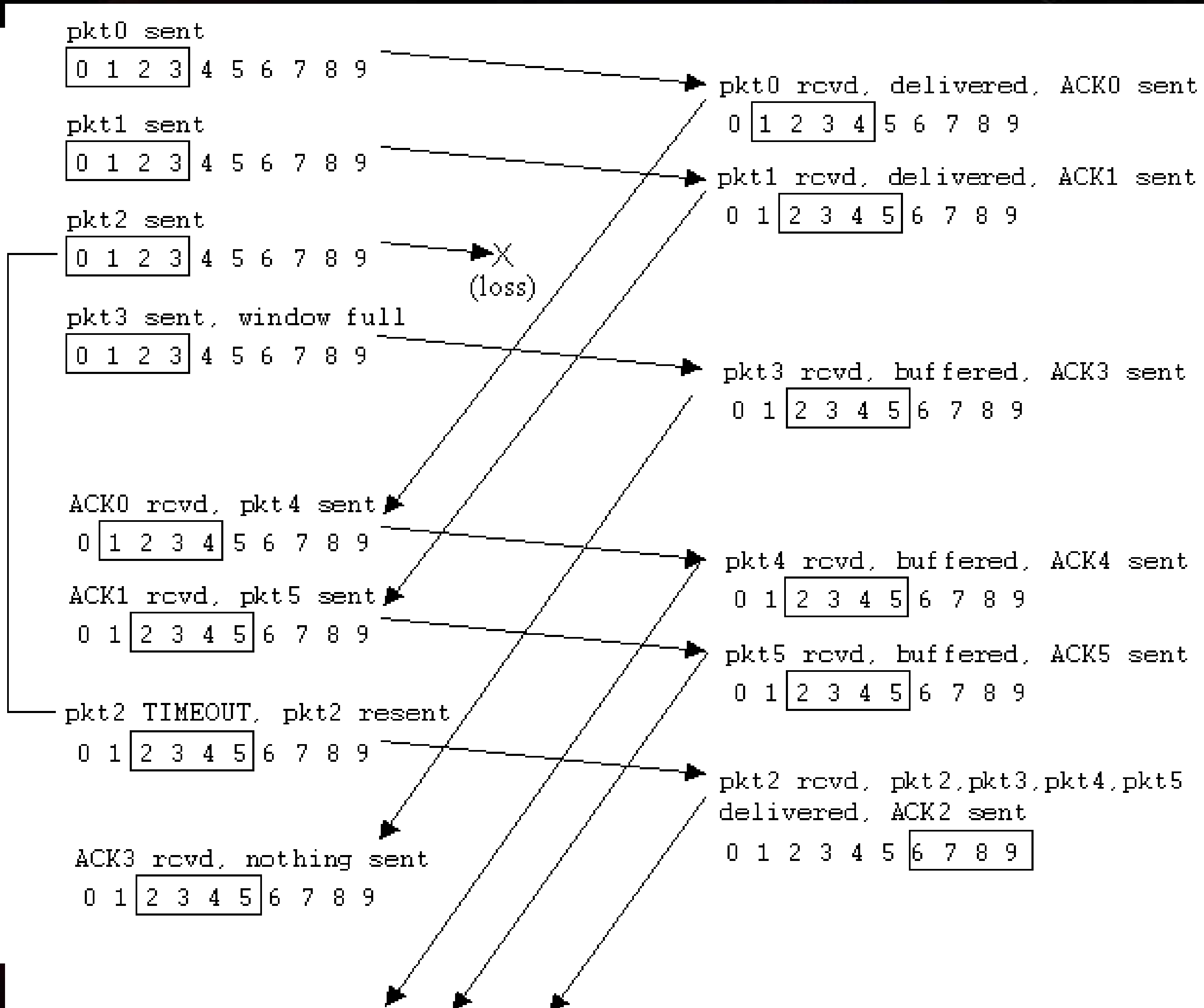
pkt n in [rcvbase-N,rcvbase-1]

- ACK(n)

otherwise:

- ignore

Selective repeat in action



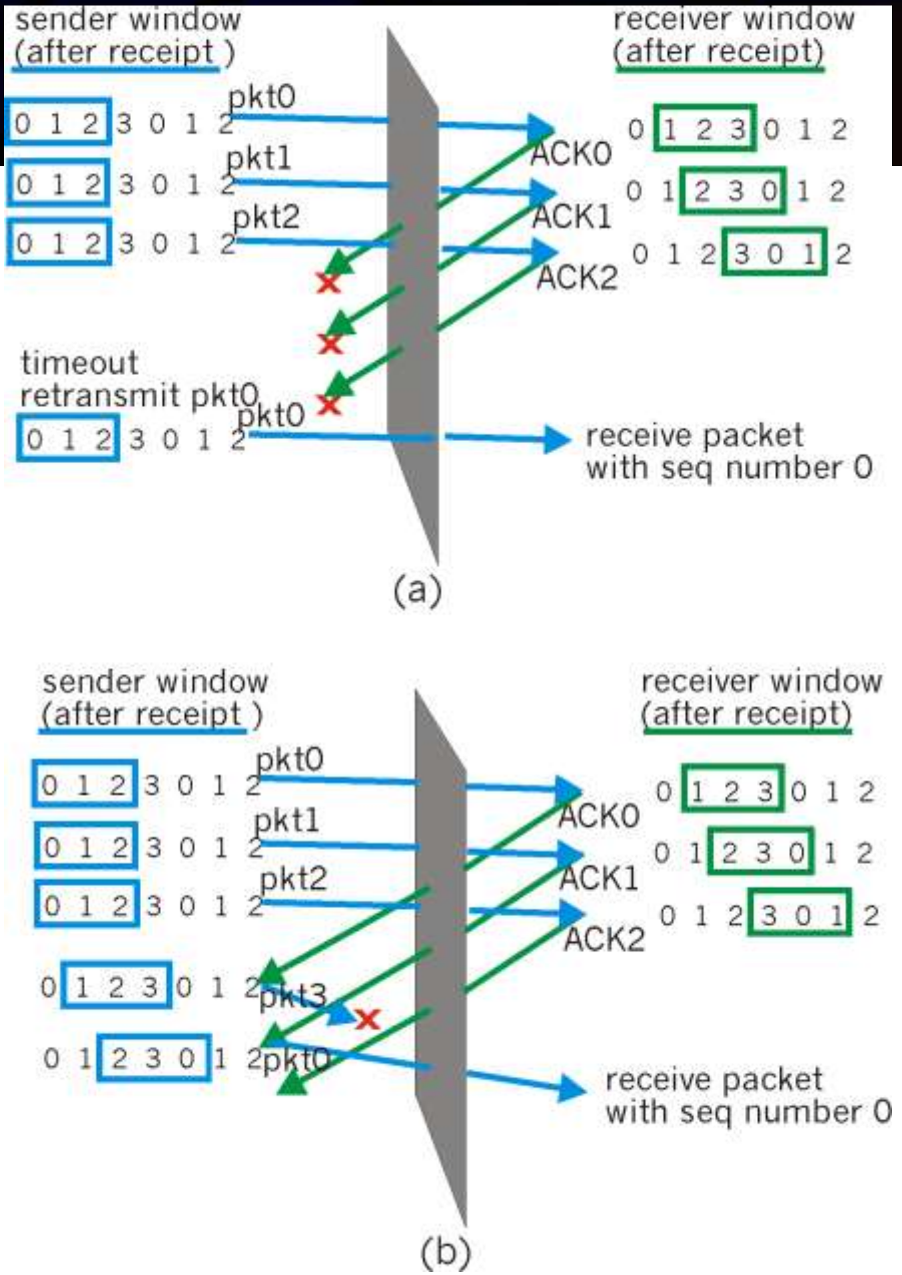
Selective repeat: dilemma

Example:

- seq #'s: 0, 1, 2, 3
- window size=3

- receiver sees no difference in two scenarios!
- incorrectly passes duplicate data as new in (a)

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

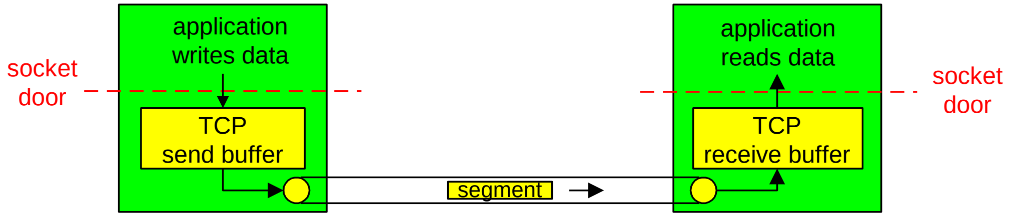


Outline

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- Principles of congestion control
- TCP congestion control

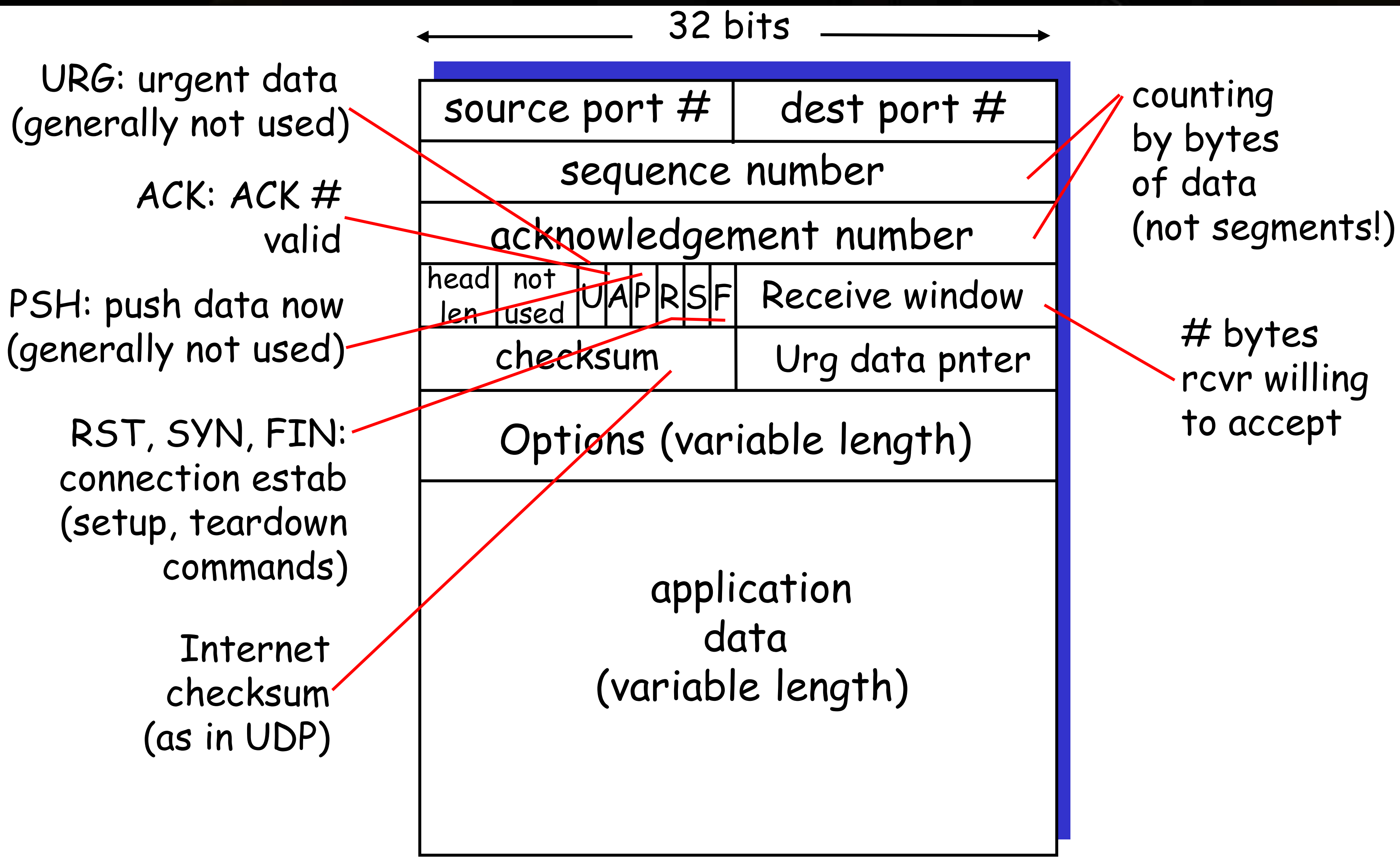
TCP: Overview RFCs: 793, 1122, 1323, 2018, 2581

- 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) init's sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

TCP segment structure



TCP seq. #'s and ACKs

Seq. #'s:

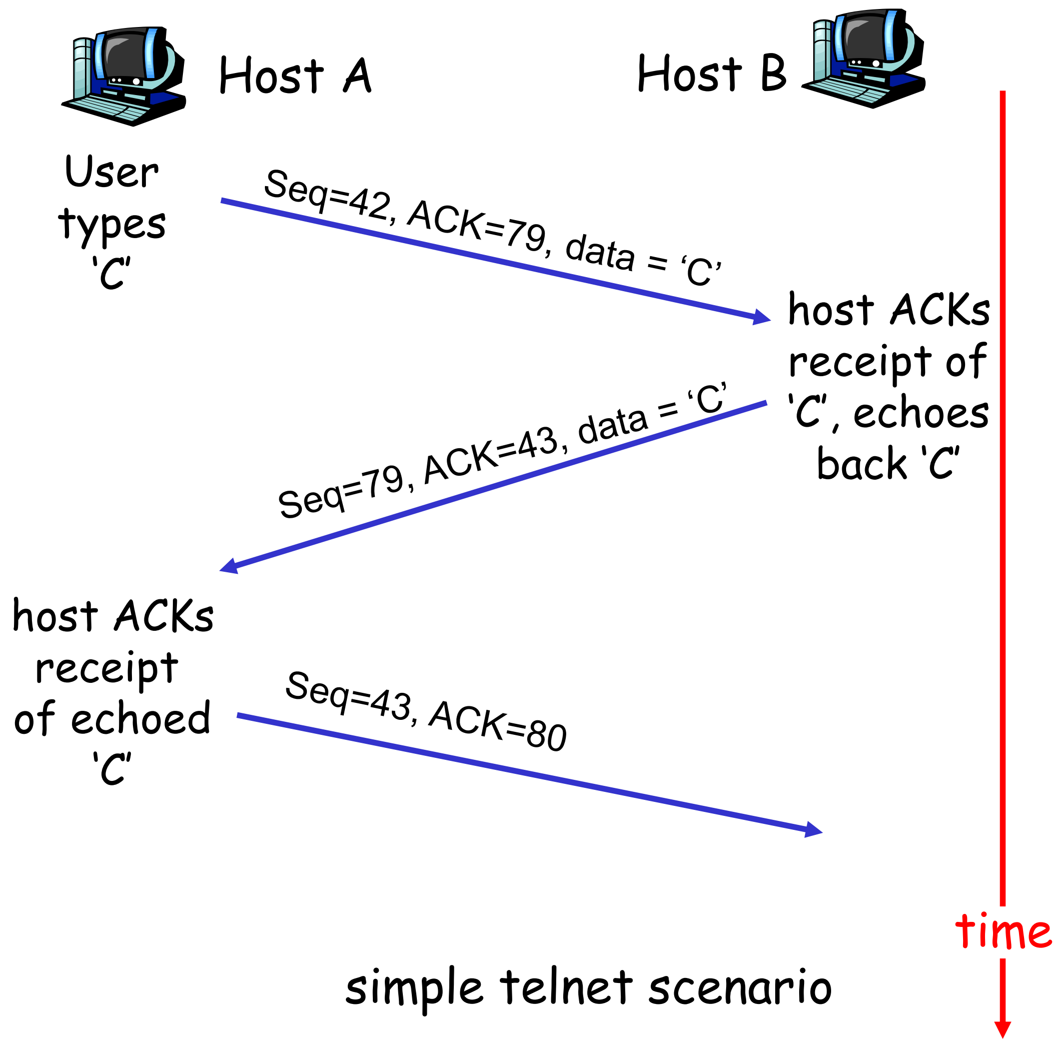
- byte stream "number" of first byte in segment's data

ACKs:

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

Q: how receiver handles out-of-order segments

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



simple telnet scenario

TCP Round Trip Time and Timeout

Q: how to set TCP timeout value?

- longer than RTT
  - but RTT varies
- too short: premature timeout
  - unnecessary retransmissions
- too long: slow reaction to segment loss

Q: how to estimate RTT?

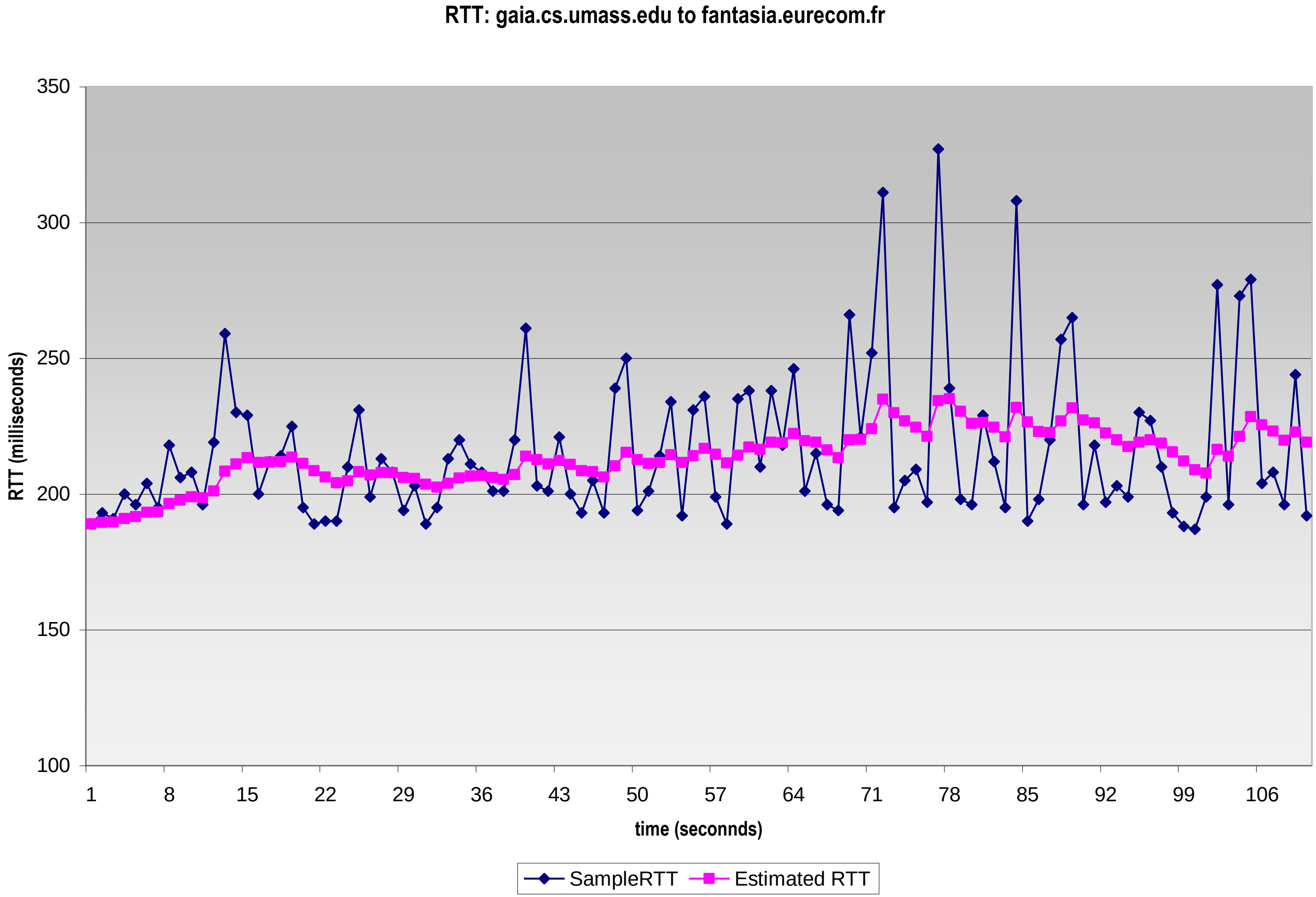
- **`SampleRTT`**: measured time from segment transmission until ACK receipt
  - ignore retransmissions
- **`SampleRTT`** will vary, want estimated RTT "smoother"
  - average several recent measurements, not just current **`SampleRTT`**

TCP Round Trip Time and Timeout

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

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

Example RTT estimation:



TCP Round Trip Time and Timeout

Setting the timeout

- `EstimtedRTT` plus "safety margin"
  - large variation in `EstimatedRTT` -> larger safety margin
- first estimate of how much SampleRTT deviates from EstimatedRTT:

```
DevRTT = (1-β)*DevRTT +
             β*|SampleRTT-EstimatedRTT|

(typically, β = 0.25)
```

Then set timeout interval:

```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```

Problems

21. Consider the TCP procedure for estimating RTT. Suppose that α=0.1. Let $SampleRTT_1$ be the most recent sample RTT, let $SampleRTT_2$ be the next most recent sample RTT, and so on.
    1. For a given TCP connection, suppose four acknowledgments have been returned with corresponding sample RTTs $SampleRTT_4$, $SampleRTT_3$, $SampleRTT_2$, $SampleRTT_1$. Express EstimatedRTT in terms of four sample RTTs.
    2. Generalize your formula for *n* sample round-trip times.
    3. For the formula in part (2) let *n* approach infinity.

22. Why do you think TCP avoids measuring the SampleRTT for retransmitted segments?

Outline

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- Principles of congestion control
- TCP congestion control

TCP reliable data transfer

- TCP creates rdt service on top of IP's unreliable service
- 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, congestion control

TCP sender events:

data rcvd 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 rcvd:

- If acknowledges previously unacked segments
  - update what is known to be acked
  - start timer if there are outstanding segments

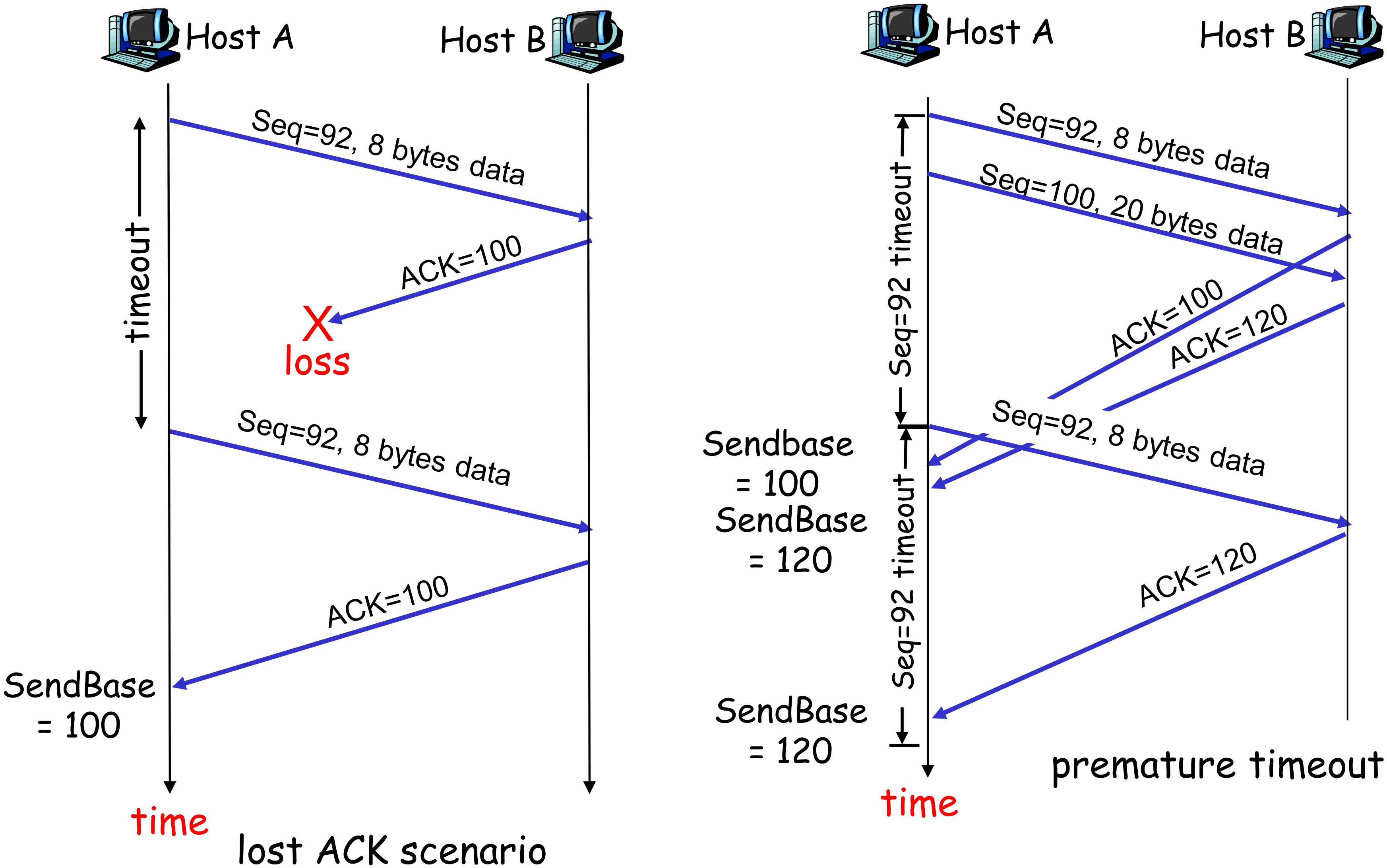
TCP sender (simplified)

```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

loop (forever) {
  switch(event)

  event: data received from application above
      create TCP segment with sequence number NextSeqNum
      if (timer currently not running)
          start timer
      pass segment to IP
      NextSeqNum = NextSeqNum + length(data)

  event: timer timeout
      retransmit not-yet-acknowledged segment with
           smallest sequence number
      start timer

  event: ACK received, with ACK field value of y
      if (y > SendBase) {
          SendBase = y
         if (there are currently not-yet-acknowledged segments)
              start timer
         }

} /* end of loop forever */
```

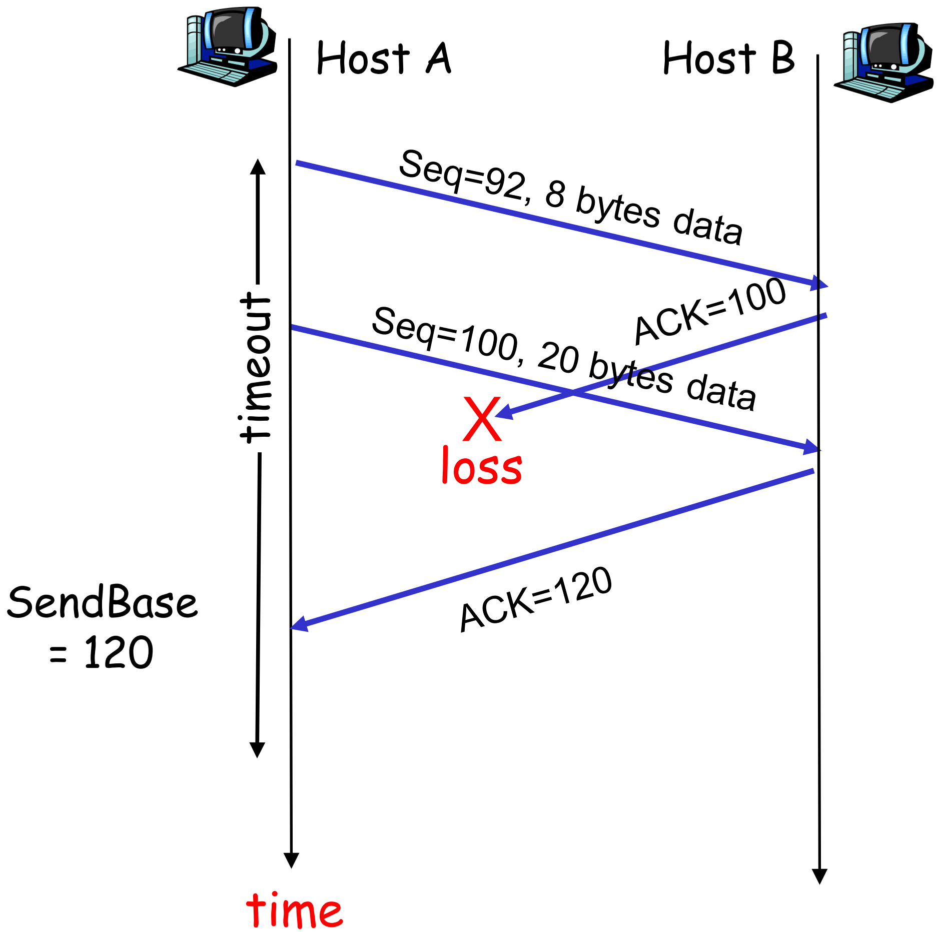
Comment:
- SendBase-1: last cumulatively ack'ed byte

Example:
- SendBase-1 = 71; y= 73, so the rcvr wants 73+ ; y > SendBase, so that new data is acked

TCP: retransmission scenarios



TCP retransmission scenarios (more)



Cumulative ACK scenario

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.

- If sender receives 3 ACKs for the same data, it supposes that segment after ACKed data was lost:
  - fast retransmit: resend segment before timer expires

Fast retransmit algorithm:

```
event: ACK received, with ACK field value of y
          if (y > SendBase) {
              SendBase = y
              if (there are currently not-yet-acknowledged segments)
                  start timer
             }
          else {
                increment count of dup ACKs received for y
                if (count of dup ACKs received for y = 3) {
                    resend segment with sequence number y
                 }
```

a duplicate ACK for already ACKed segment

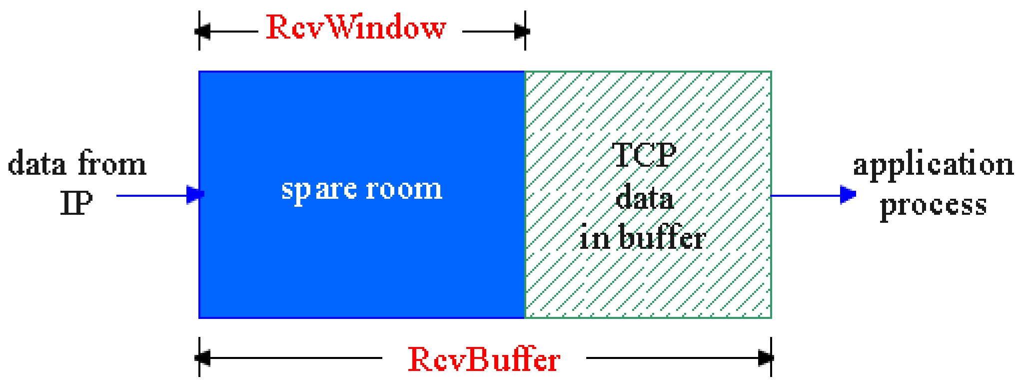
fast retransmit

Outline

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- Principles of congestion control
- TCP congestion control

TCP Flow Control

- receive side of TCP connection has a receive buffer:



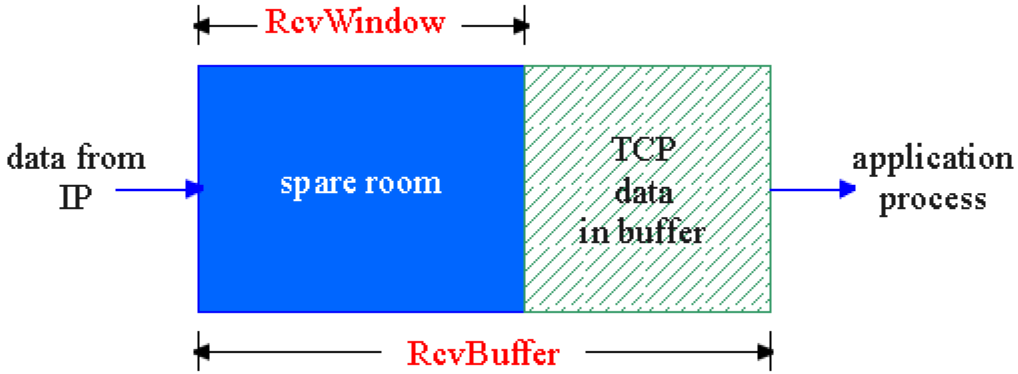
- app process may be slow at reading from buffer

flow control

sender won't overflow receiver's buffer by transmitting too much, too fast

- speed-matching service: matching the send rate to the receiving app's drain rate

TCP Flow control: how it works



(Suppose TCP receiver discards out-of-order segments)

- spare room in buffer

`= RcvWindow`

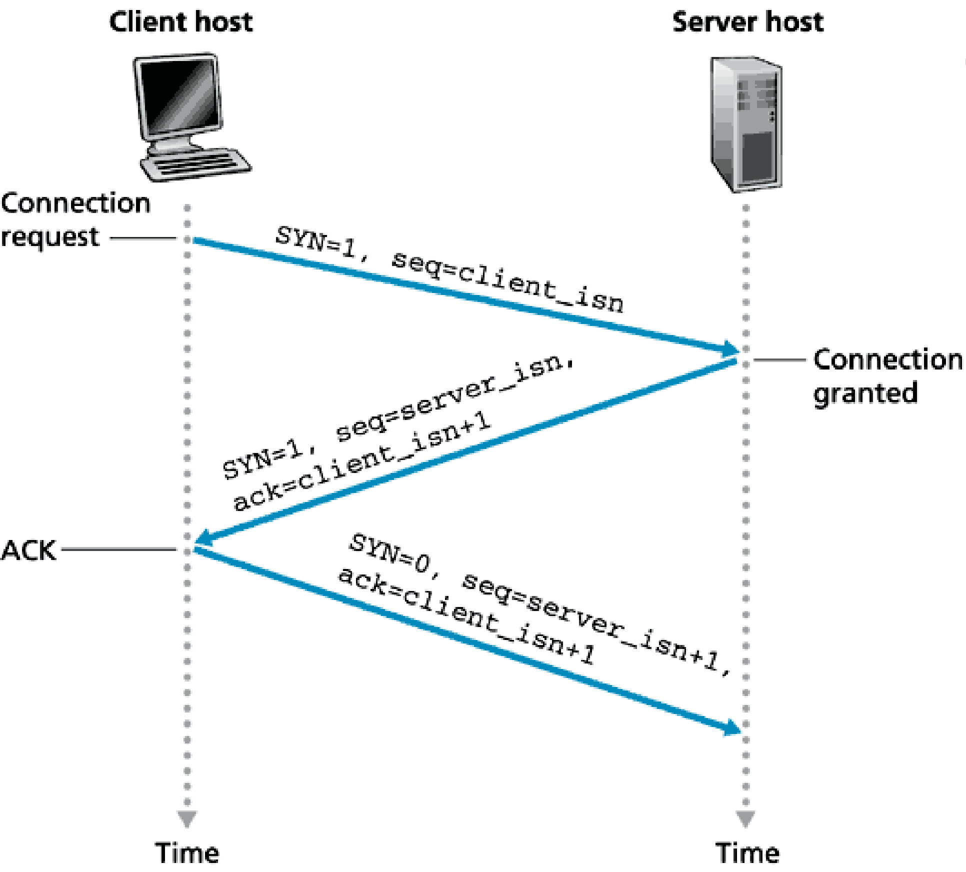
`= RcvBuffer-[LastByteRcvd - LastByteRead]`

- Rcvr advertises spare room by including value of **`RcvWindow`** in segments
- Sender limits unACKed data to **`RcvWindow`**
  - guarantees receive buffer doesn't overflow

Outline

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- Principles of congestion control
- TCP congestion control

TCP Connection Management



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

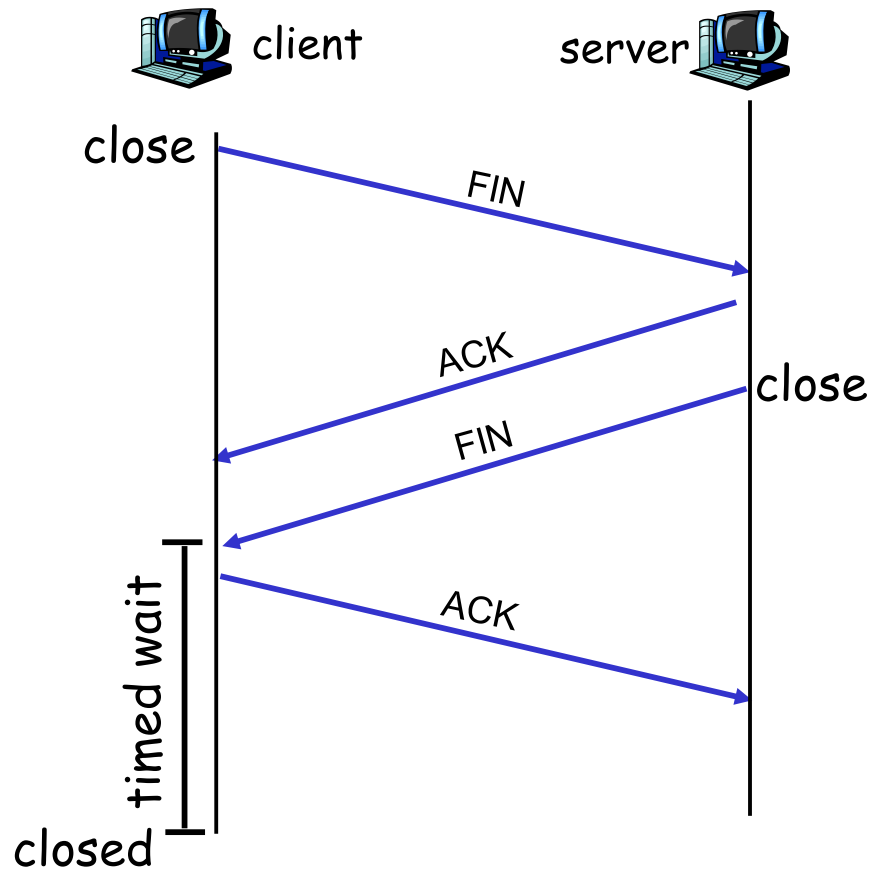
TCP Connection Management (cont.)

Closing a connection:

client closes socket:
```
clientSocket.close();
```

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

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



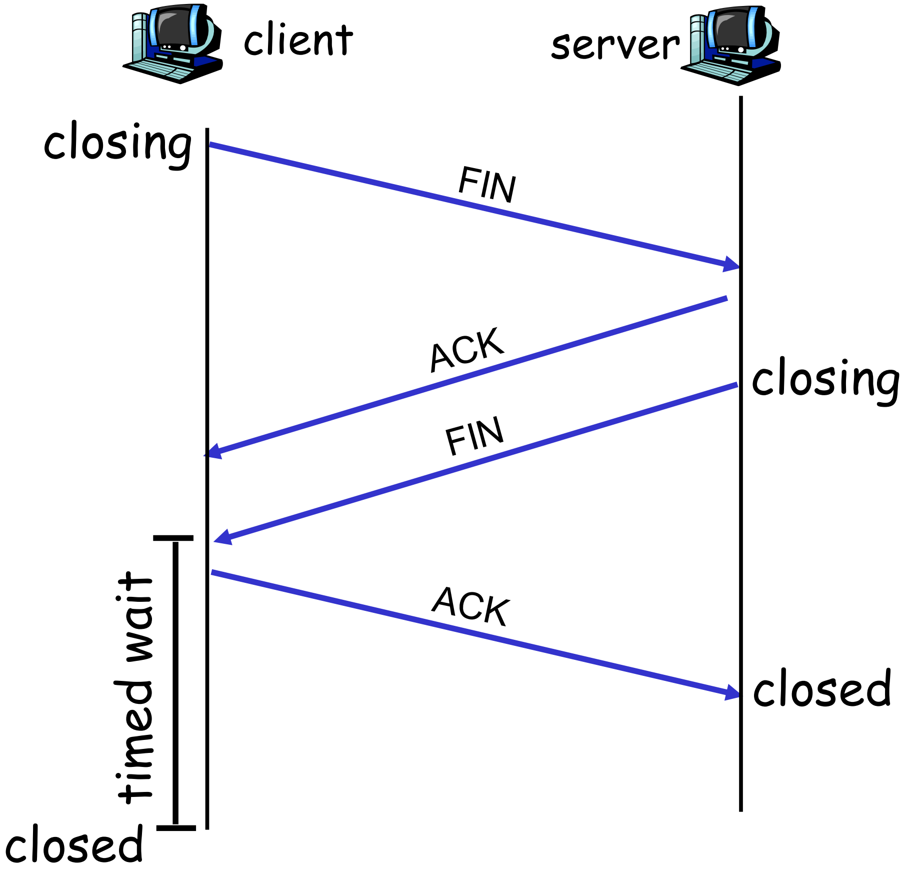
TCP Connection Management (cont.)

Step 3: client receives FIN, replies with ACK.

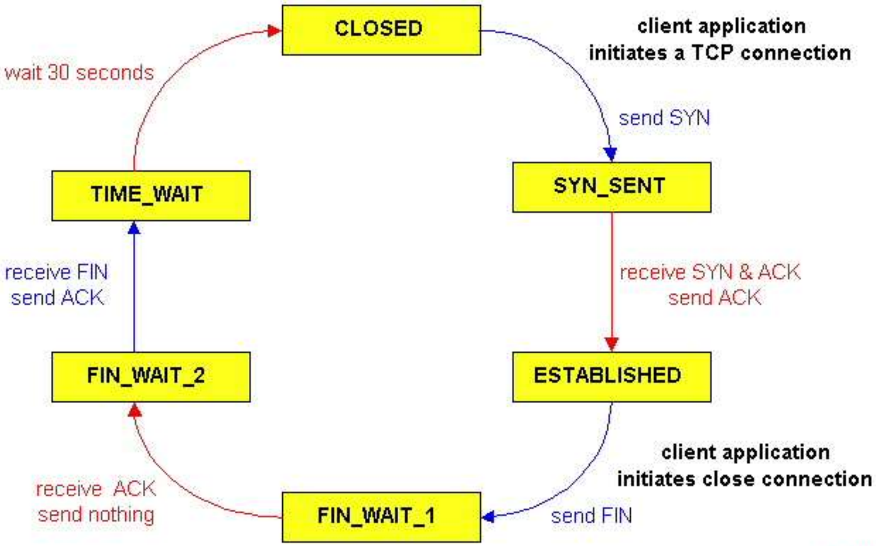
- Enters "timed wait" - will respond with ACK to received FINs

Step 4: server, receives ACK. Connection closed.

Note: with small modification, can handle simultaneous FINs.

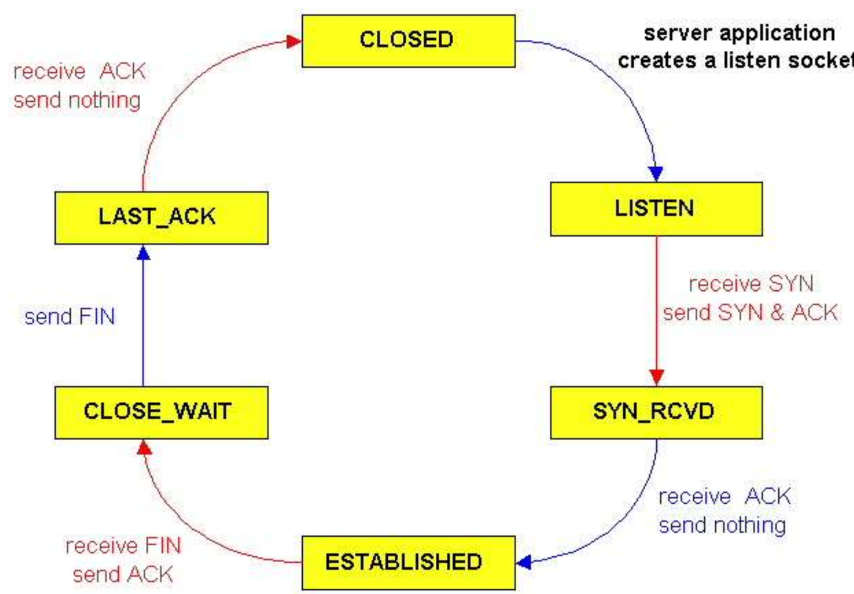


TCP Connection Management (cont)



TCP client lifecycle

TCP server lifecycle



Outline

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- Principles of congestion control
- TCP congestion control
- Delay modeling

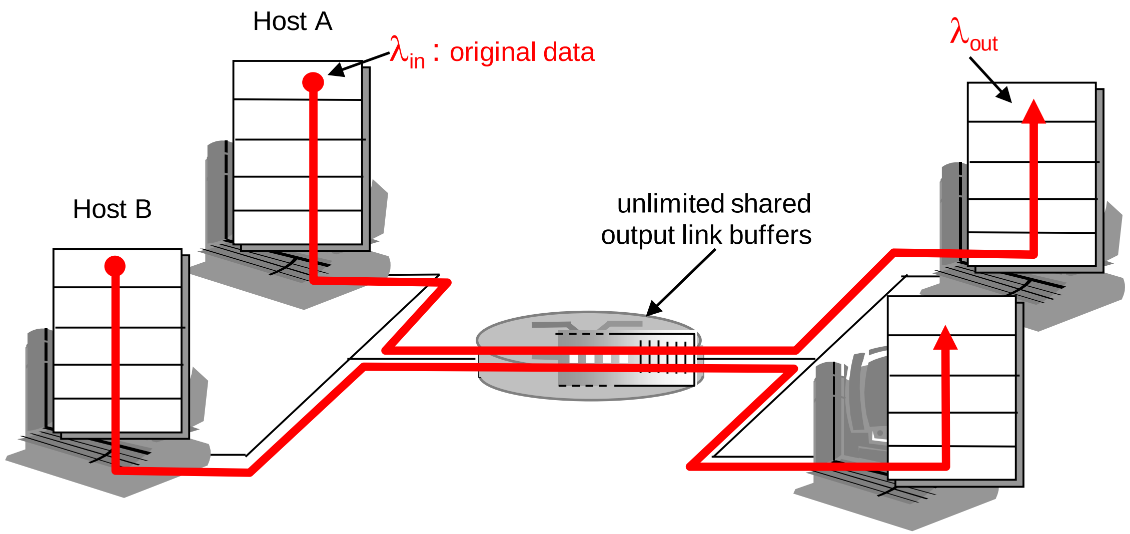
Principles of Congestion Control

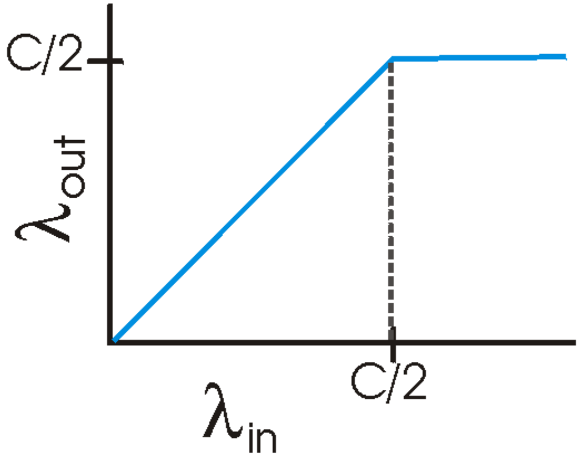
Congestion:

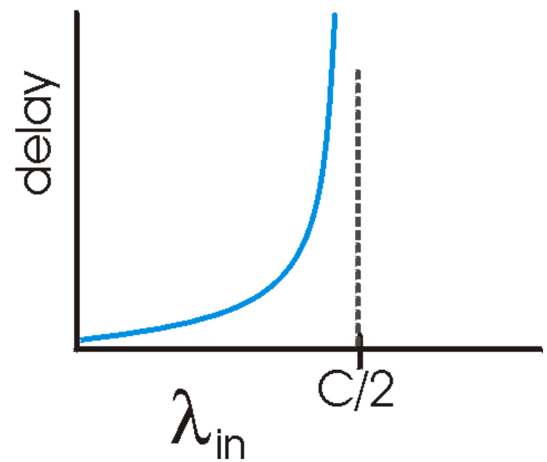
- informally: "too many sources sending too much data too fast for *network* to handle"
- different from flow control!
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)
- a top-10 problem!

Causes/costs of congestion: scenario 1

- two senders, two receivers
- one router, infinite buffers
- no retransmission



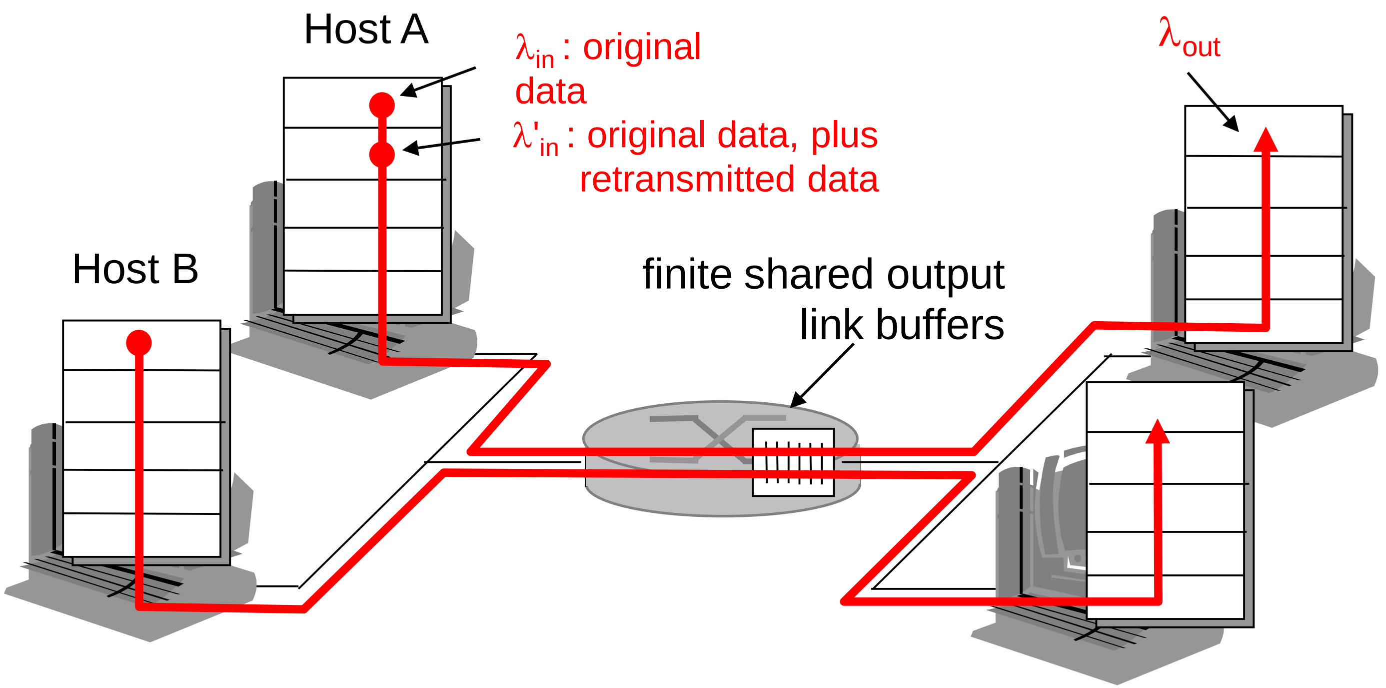




- large delays when congested
- maximum achievable throughput

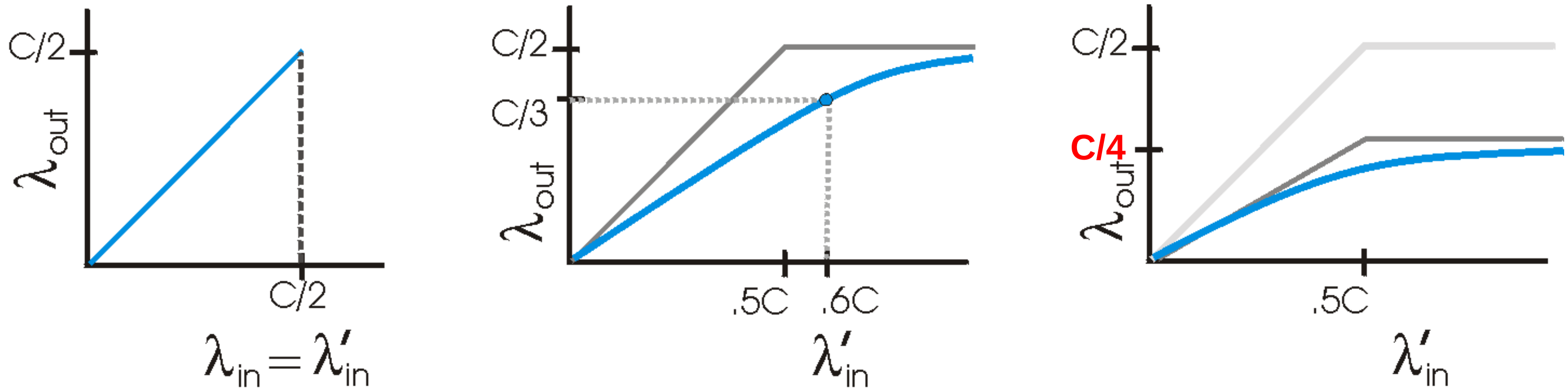
Causes/costs of congestion: scenario 2

- one router, *finite* buffers
- sender retransmission of lost packet



Causes/costs of congestion: scenario 2

- always: $\lambda_{in} = \lambda_{out}$ (goodput)
- "perfect" retransmission only when loss: $\lambda'_{in} > \lambda_{out}$
- retransmission of delayed (not lost) packet makes λ'_{in} larger (than perfect case) for same λ_{out}



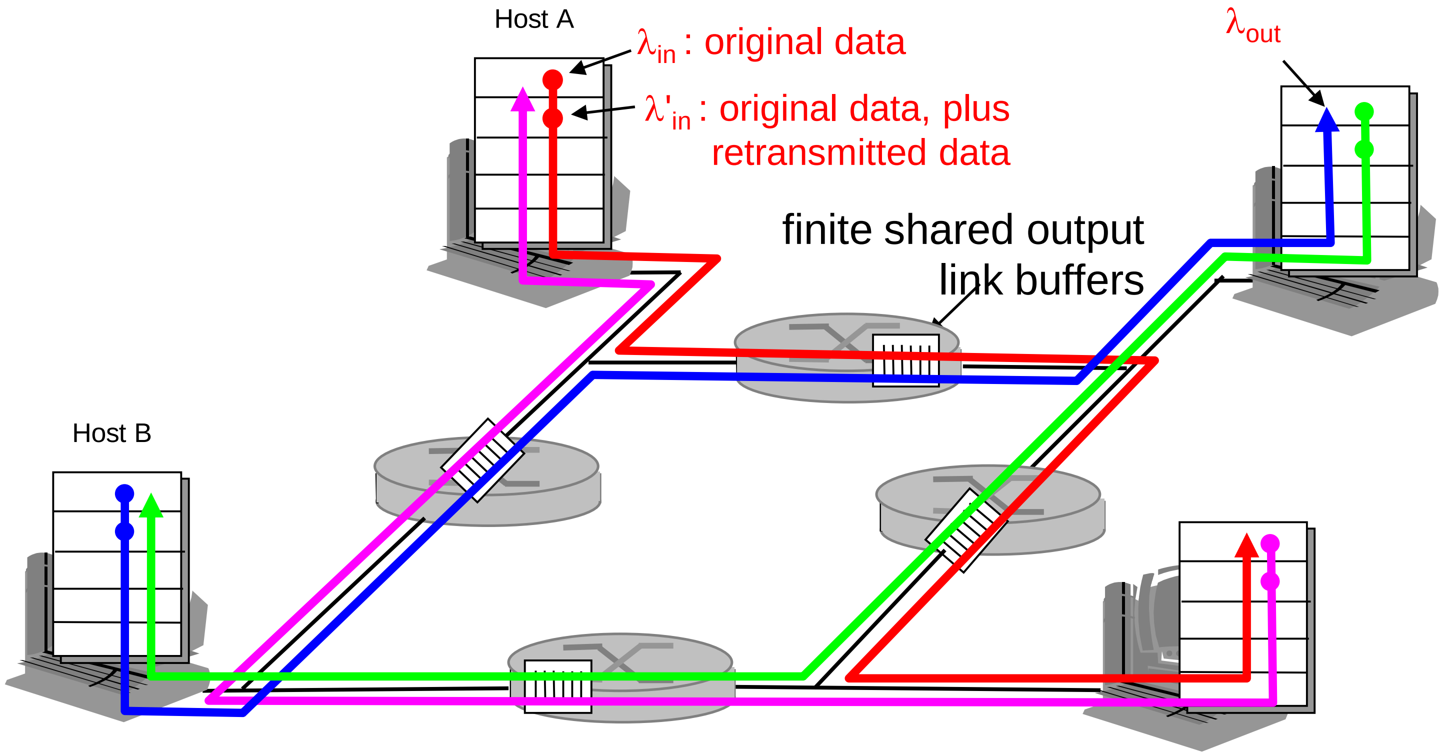
"costs" of congestion:

- more work (retrans) for given "goodput"
- unneeded retransmissions: link carries multiple copies of pkt

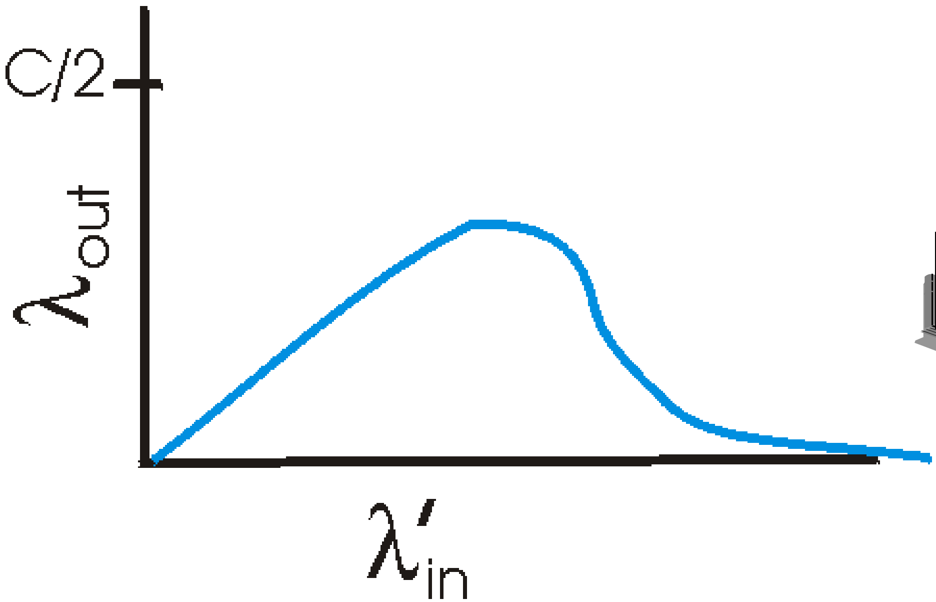
Causes/costs of congestion: scenario 3

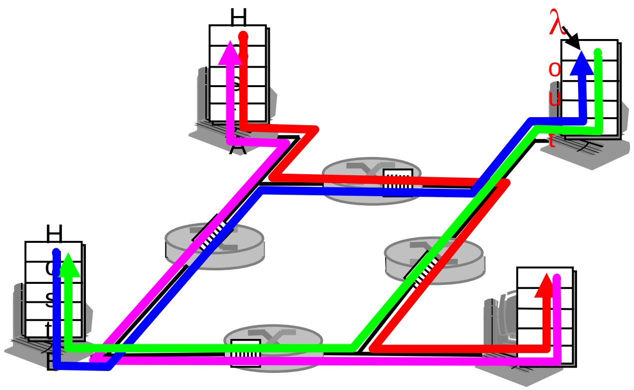
- four senders
- multihop paths
- timeout/retransmit

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



Causes/costs of congestion: scenario 3





Another "cost" of congestion:

- when packet dropped, any "upstream transmission capacity used for that packet was wasted!

Approaches towards congestion control

Two broad approaches towards 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

Outline

- Transport-layer services
- Multiplexing and demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- Principles of congestion control
- TCP congestion control

TCP Congestion Control

- end-end control (no network assistance)
- sender limits transmission:

  `LastByteSent-LastByteAcked` $\leq$ `CongWin`
- Roughly,

$$\text{rate} = \frac{\text{CongWin}}{\text{RTT}} \text{ Bytes/sec}$$

- `CongWin` is dynamic, function of perceived network congestion

How does sender perceive congestion?

- loss event = timeout *or* 3 duplicate acks
- TCP sender reduces rate (`CongWin`) after loss event

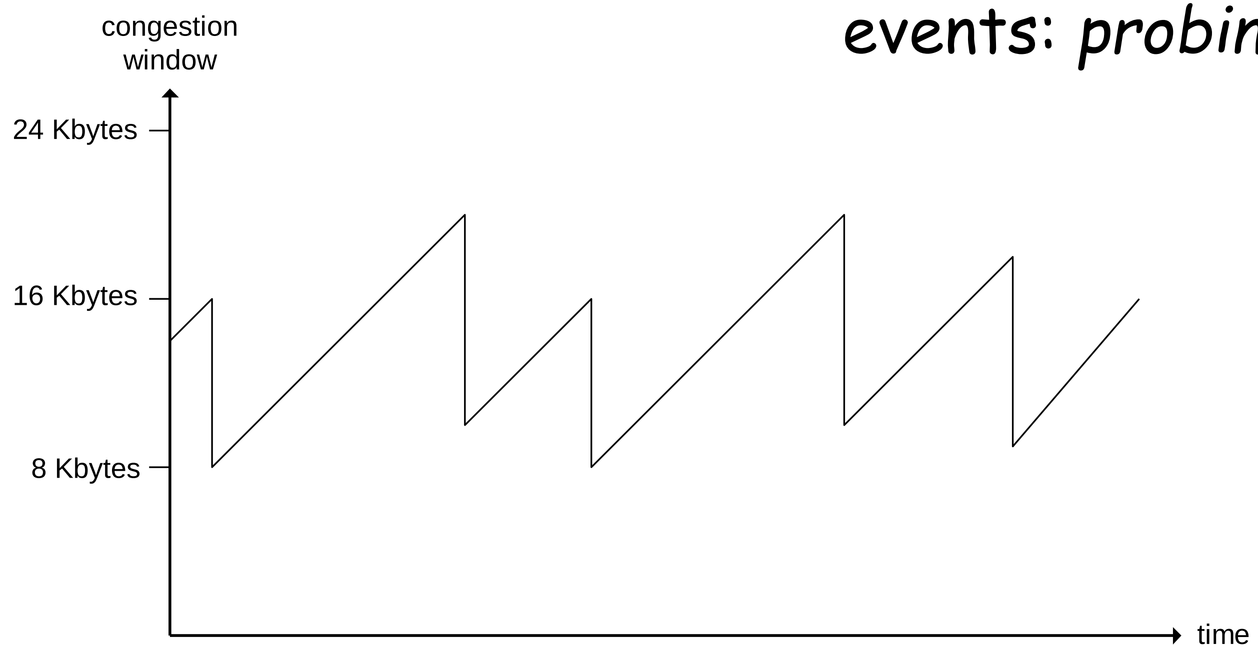
three mechanisms:

- AIMD
- slow start
- conservative after timeout events

TCP AIMD

multiplicative decrease: cut `CongWin` in half after loss event

additive increase: increase `CongWin` by 1 MSS every RTT in the absence of loss events: *probing*



Long-lived TCP connection

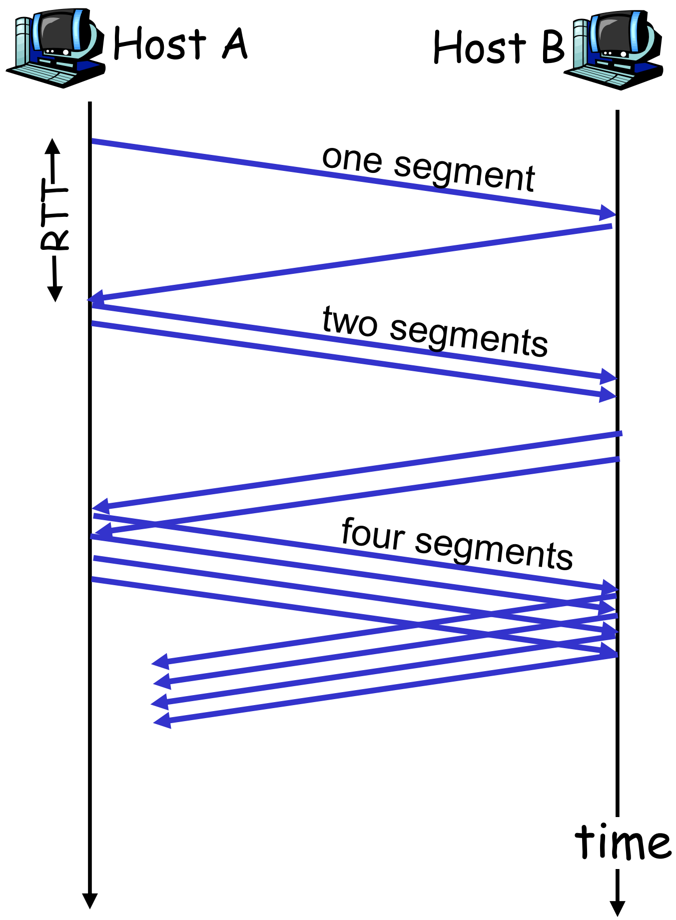
TCP Slow Start

- When connection begins, `CongWin` = 1 MSS
  - Example: MSS = 500 bytes & RTT = 200 msec
  - initial rate = 20 kbps
- available bandwidth may be >> MSS/RTT
  - desirable to quickly ramp up to respectable rate

- When connection begins, increase rate exponentially fast until first loss event

TCP Slow Start (more)

- When connection begins, increase rate exponentially until first loss event:
  - double `CongWin` every RTT
  - done by incrementing `CongWin` for every ACK received
- Summary: initial rate is slow but ramps up exponentially fast



Refinement

- After 3 dup ACKs:
  - `CongWin` is cut in half
  - window then grows linearly
- <u>But</u> after timeout event:
  - `CongWin` instead set to 1 MSS;
  - window then grows exponentially
  - to a threshold, then grows linearly

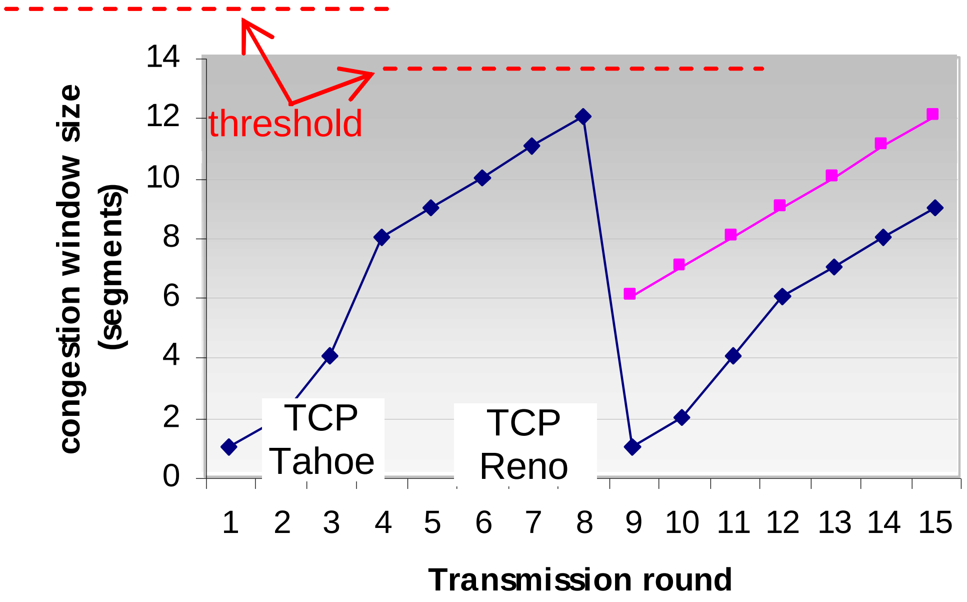
Philosophy:

- 3 dup ACKs indicates network capable of delivering some segments
- timeout before 3 dup ACKs is "more alarming"

Refinement (more)

Q: When should the exponential increase switch to linear?

A: When **CongWin** gets to 1/2 of its value before timeout.



Implementation:

- Variable Threshold
- At loss event, Threshold is set to 1/2 of CongWin just before loss event

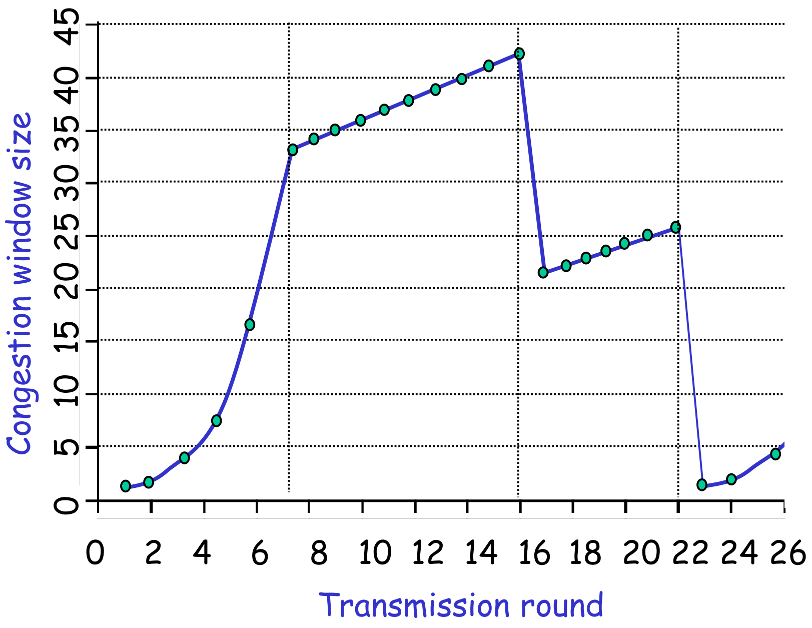
Summary: TCP Congestion Control

- When **`CongWin`** is below **`Threshold`**, sender in slow-start phase, window grows exponentially.
- When **`CongWin`** is above **`Threshold`**, sender is in congestion-avoidance phase, window grows linearly.
- When a triple duplicate ACK occurs, **`Threshold`** set to **`CongWin/2`** and **`CongWin`** set to **`Threshold`**.
- When timeout occurs, **`Threshold`** set to **`CongWin/2`** and **`CongWin`** is set to 1 MSS.

Problem

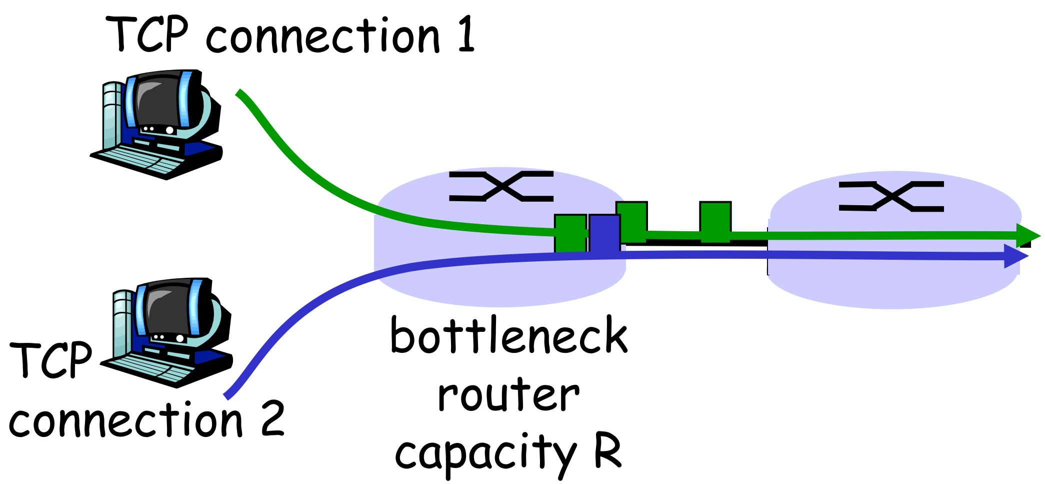
- Consider the following plot of TCP window size as a function of time. Assuming TCP Reno is the protocol experiencing the behavior shown below, answer the following questions. In all cases, you should provide a short discussion justifying your answer

1. Identify the intervals of time when TCP slow star is operating..
2. Identify the intervals of time when TCP congestion avoidance is operating.
3. After the 16th transmission round, is segment loss detected by a triple duplicate ACK or by timeout.
4. What is the initial value of Threshold at the first transmission round?
5. What is the value of Threshold at the 18th transmission round?



TCP Fairness

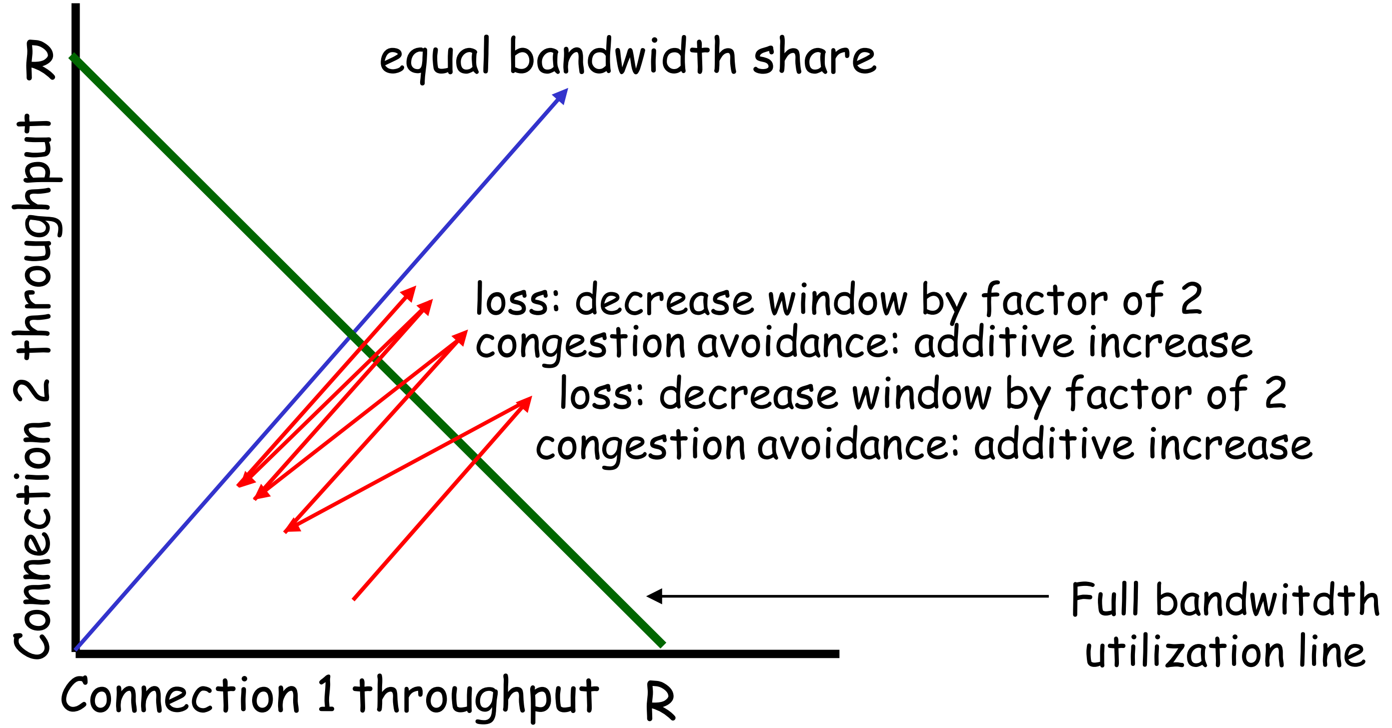
Fairness goal: if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K



Why is TCP fair?

Two competing sessions:

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



Fairness (more)

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
- Research area: TCP friendly

Fairness and parallel TCP connections

- nothing prevents app from opening parallel cnctions between 2 hosts.
- Web browsers do this
- Example: link of rate R supporting 9 cnctions;
  - new app asks for 1 TCP, gets rate R/10
  - new app asks for 11 TCPs, gets R/2 !

Delay modeling

Q: How long does it take to receive an object from a Web server after sending a request?

Ignoring congestion, delay is influenced by:

- TCP connection establishment
- data transmission delay
- slow start

Notation, assumptions:

- Assume one link between client and server of rate R
- S: MSS (bits)
- O: object size (bits)
- no retransmissions (no loss, no corruption)

Window size:

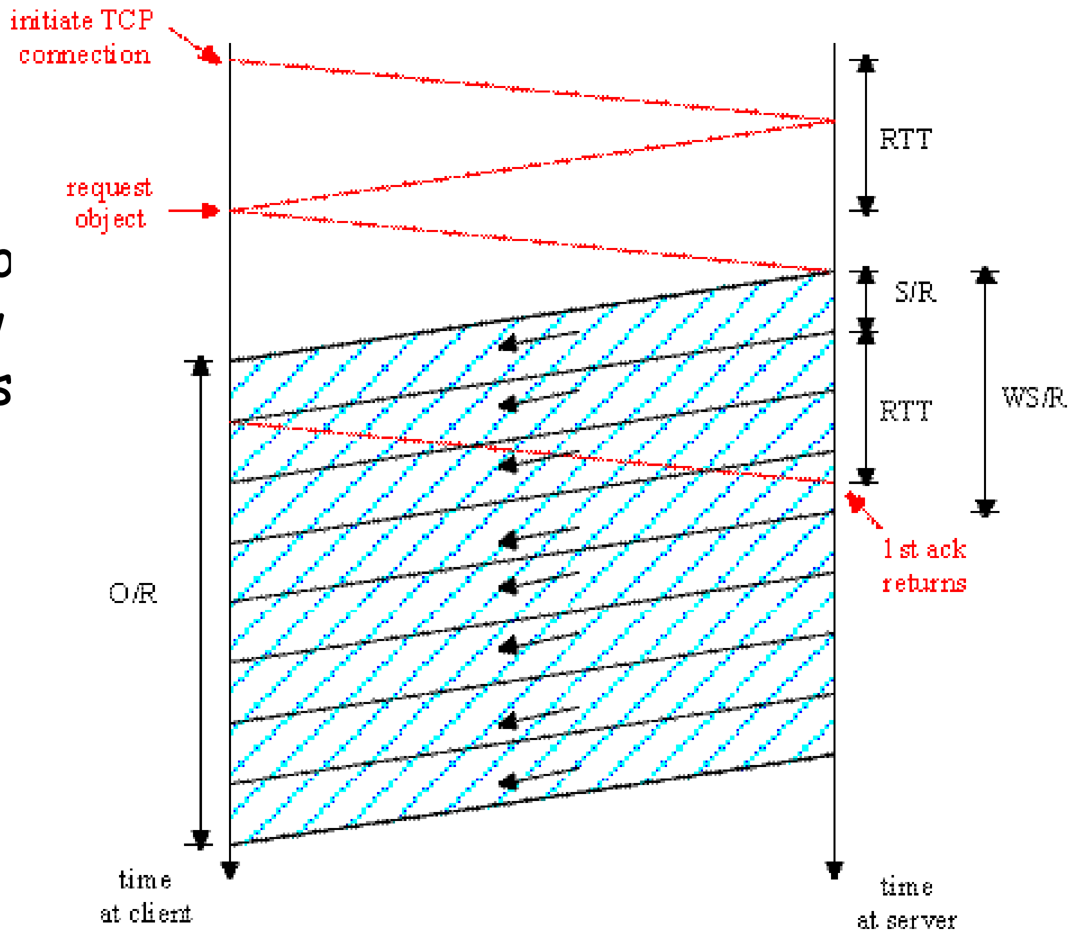
- First assume: fixed congestion window, W segments
- Then dynamic window, modeling slow start

Fixed congestion window (1)

First case:

WS/R > RTT + S/R: ACK fo
first segment in window
returns before window's
worth of data sent

latency = 2RTT + O/R



Fixed congestion window (2)

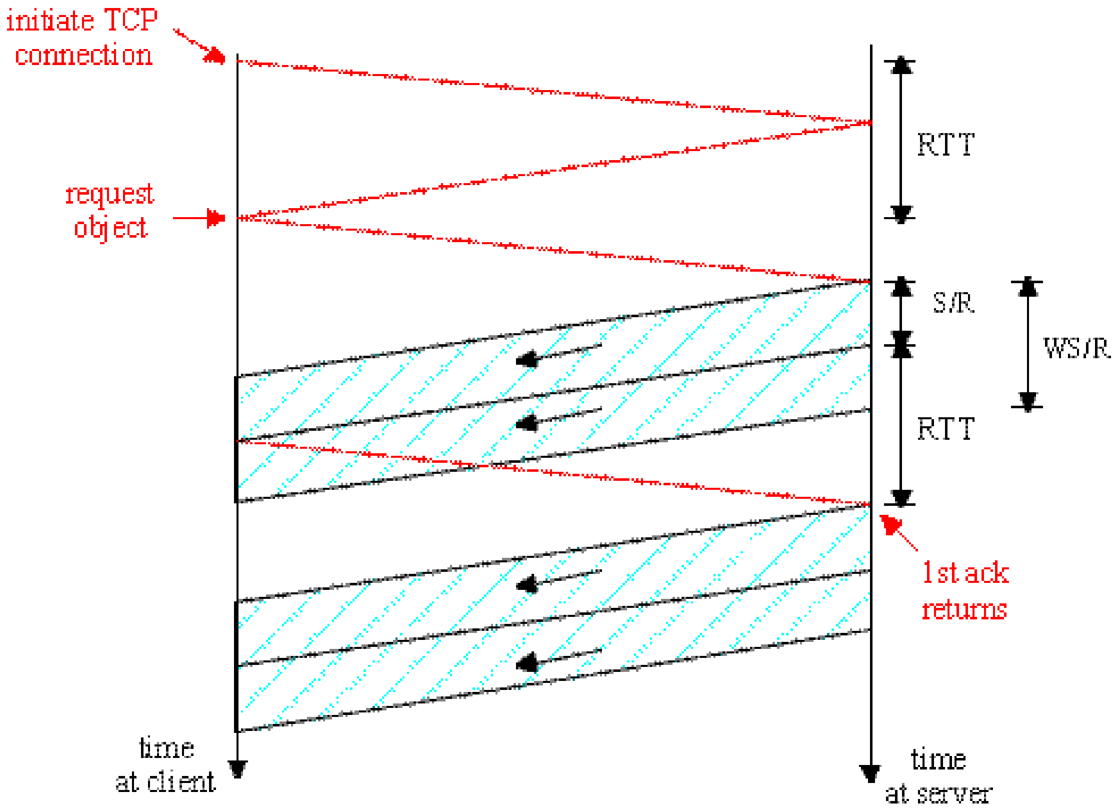
Second case:

- WS/R < RTT + S/R: wait for ACK after sending window's worth of data sent

K = O/WS

Wait time = RTT – (W-1)S/R]

latency = 2RTT + O/R + (K-1)[S/R + RTT - WS/R]



congestion window exercise

- Consider sending an object of size O = 100 kbytes from server to client. Let S = 536 bytes and RTT = 100 msec. Suppose the transport protocol uses static windows with window size W.
  1. For a transmission rate of 28 kbps, determine the minimum possible latency. Determine the minimum window size that achieves this latency.

Answer:

The minimum latency is $2RTT+O/R$. The minimum W that achieves this latency is

R	min latency	W
28 Kbps	28.77 sec	2
100 Kbps	8.2 sec	4
1 Mbps	1 sec	25
10 Mbps	0.28 sec	235

TCP Delay Modeling: Slow Start (1)

Now suppose window grows according to slow start

Will show that the delay for one object is:

$$\boxed{Latency = 2RTT + \frac{O}{R} + \sum_{k=1}^{k-1}\left[\frac{S}{R} + RTT + 2^{k-1}\frac{S}{R}\right]}$$

$$\boxed{Latency = 2RTT + \frac{O}{R} + P\left[RTT + \frac{S}{R}\right] - (2^{P} - 1)\frac{S}{R}}$$

]

where *P* is the number of times TCP idles at server:

$$P = \min\{Q, K-1\}$$

- where Q is the number of times the server idles
  if the object were of infinite size.

- and K is the number of windows that cover the object.

TCP Delay Modeling: Slow Start (2)

Delay components:

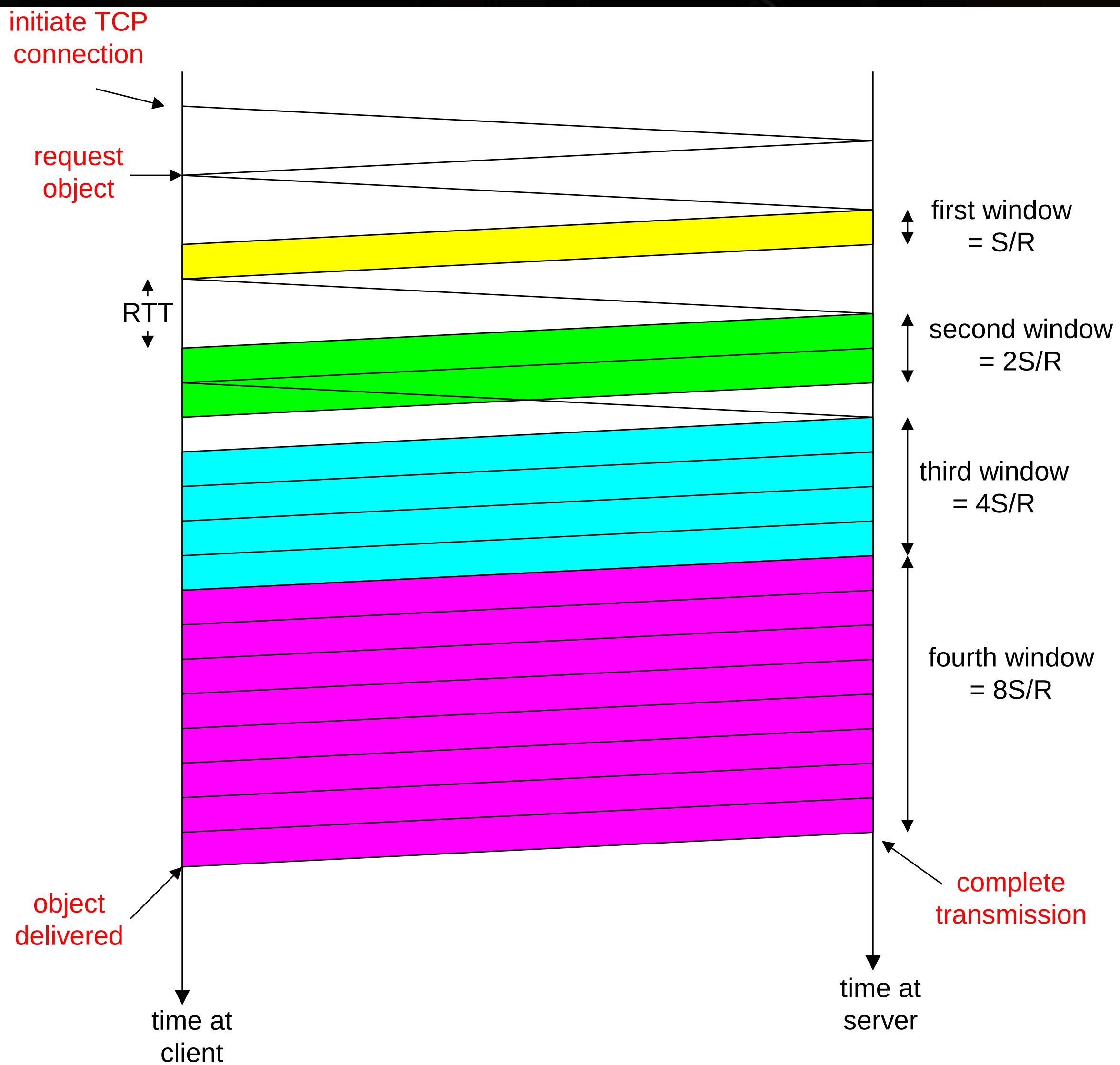
- 2 RTT for connection estab and request
- O/R to transmit object
- time server idles due to slow start

Server idles:
P = min{K-1,Q} times

Example:

- O/S = 15 segments
- K = 4 windows
- Q = 2
- P = min{K-1,Q} = 2

Server idles P=2 times



TCP Delay Modeling (3)

$$\frac{S}{R} + RTT = \text{time from when server starts to send segment until server receives acknowledgement}$$

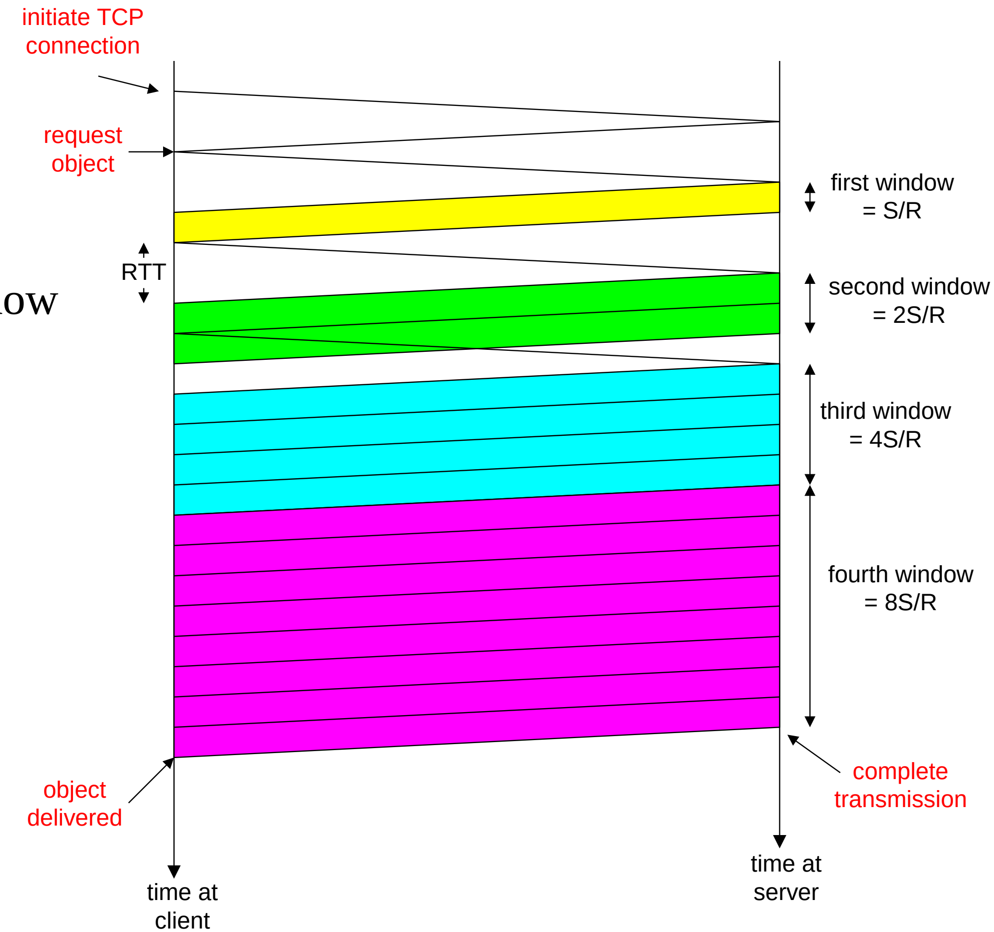
$$2^{k-1}\frac{S}{R} = \text{time to transmit the kth window}$$

$$\left[\frac{S}{R} + RTT - 2^{k-1}\frac{S}{R}\right]^{+} = \text{idle time after the } k\text{th window}$$

$$\text{delay} = \frac{O}{R} + 2RTT + \sum_{p=1}^{P} idleTime_p$$

$$= \frac{O}{R} + 2RTT + \sum_{k=1}^{P}\left[\frac{S}{R} + RTT - 2^{k-1}\frac{S}{R}\right]$$

$$= \frac{O}{R} + 2RTT + P\left[RTT + \frac{S}{R}\right] - (2^{P} - 1)\frac{S}{R}$$



TCP Delay Modeling (4)

Recall K = number of windows that cover object

How do we calculate K ?

$$
\begin{aligned}
K &= \min\{k : 2^0 S + 2^1 S + \cdots + 2^{k-1} S \geq O\} \\
&= \min\{k : 2^0 + 2^1 + \cdots + 2^{k-1} \geq O/S\} \\
&= \min\{k : 2^k - 1 \geq \frac{O}{S}\} \\
&= \min\{k : k \geq \log_2(\frac{O}{S} + 1)\} \\
&= \left\lceil \log_2(\frac{O}{S} + 1) \right\rceil
\end{aligned}
$$

TCP Delay Modeling (4)

Recall K = number of windows that cover object

How do we calculate Q ?

$$
\begin{aligned}
Q &= \max\left\{k : RTT + \frac{S}{R} - \frac{S}{R}2^{k-1} \geq 0\right\} \\
&= \max\left\{k : 2^{k-1} \leq 1 + \frac{RTT}{S/R}\right\} \\
&= \max\left\{k : k \leq \log_2\left(1 + \frac{RTT}{S/R}\right) + 1\right\} \\
&= \left\lfloor \log_2\left(1 + \frac{RTT}{S/R}\right)\right\rfloor + 1
\end{aligned}
$$

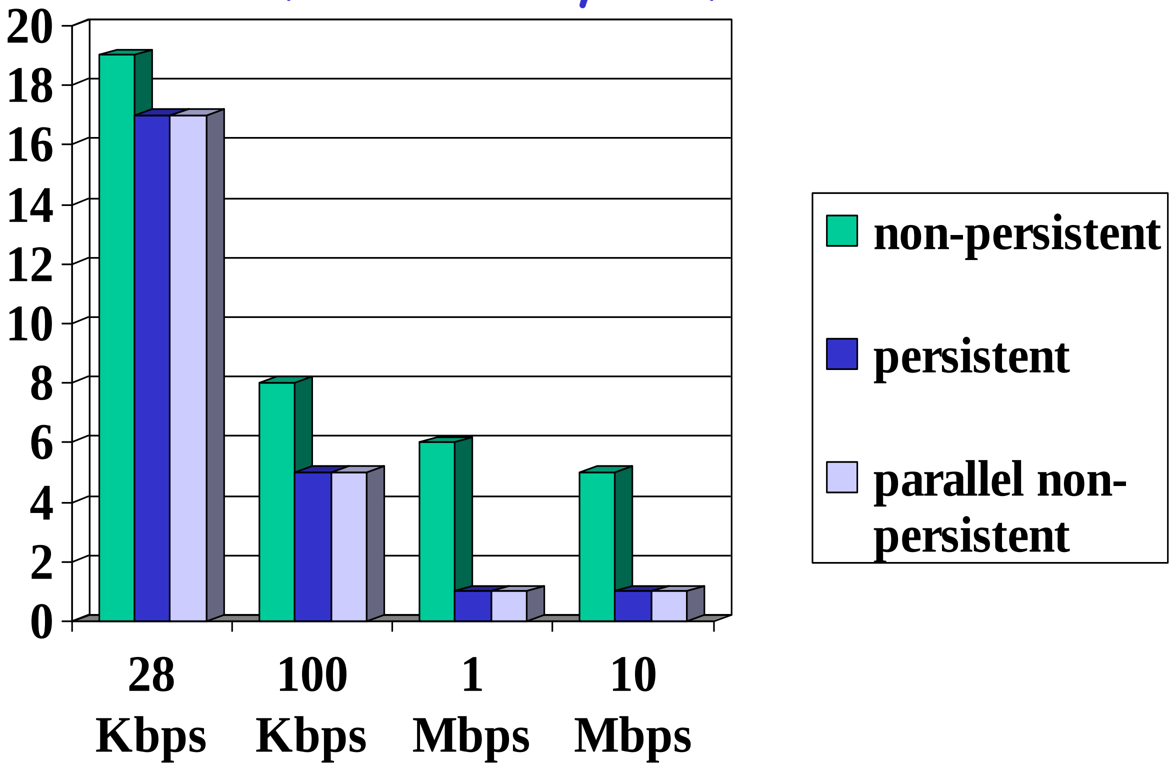
Calculation of Q, number of idles for infinite-size object, is similar to *K*

HTTP Modeling

- Assume Web page consists of:
  - *1* base HTML page (of size *O* bits)
  - *M* images (each of size *O* bits)
- Non-persistent HTTP:
  - *M+1* TCP connections in series
  - *Response time = (M+1)O/R + (M+1)2RTT + sum of idle times*
- Persistent HTTP:
  - *2 RTT* to request and receive base HTML file
  - *1 RTT* to request and receive M images
  - *Response time = (M+1)O/R + 3RTT + sum of idle times*
- Non-persistent HTTP with X parallel connections
  - Suppose M/X integer.
  - 1 TCP connection for base file
  - M/X sets of parallel connections for images.
  - *Response time = (M+1)O/R + (M/X + 1)2RTT + sum of idle times*

HTTP Response time (in seconds)

RTT = 100 msec, O = 5 Kbytes, M=10 and X=5

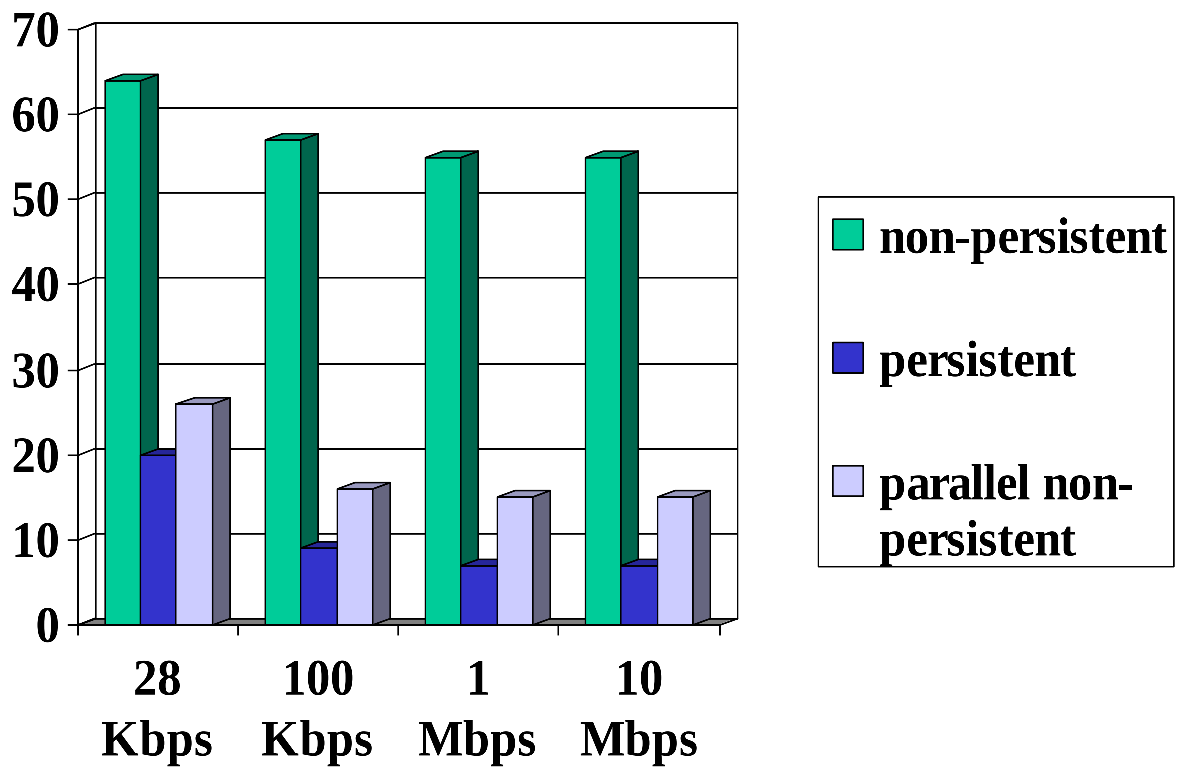


For low bandwidth, connection & response time dominated by transmission time.

Persistent connections only give minor improvement over parallel connections.

HTTP Response time (in seconds)

RTT =1 sec, O = 5 Kbytes, M=10 and X=5



For larger RTT, response time dominated by TCP establishment & slow start delays. Persistent connections now give important improvement: particularly in high delay•bandwidth networks.

Summary

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

Next:

- leaving the network "edge" (application, transport layers)
- into the network "core"

Advanced Networks

PhD. Saúl Pomares Hernández

Causal Delivery

Clocks, events and process states

- *A* distributed system is defined as a collection *P* of *N* processes p_i, $i = 1,2,\ldots N$
- Each process p_i has a state s_i consisting of its variables (which it transforms as it executes)
- Processes communicate only by messages (via a network)
- Actions of processes:
  - *Send*, *Receive*, change own state
- *Event:* the occurrence of a single action that a process carries out as it executes e.g. *Send*, *Receive*, change state
- Events at a single process p*i*, can be placed in a total ordering denoted by the relation $\rightarrow_i$ between the events. i.e.

  $e \rightarrow_i e'$ if and only if e occurs before e' at p_i
- A history of process p_i: is a series of events ordered by $\rightarrow_i$

  $history(p_i) = h_i = \langle e_i^0, e_i^1, e_i^2, \ldots \rangle$

Logical time and logical clocks Happened-before(Lamport 1978)

- Instead of synchronizing clocks, event ordering can be used

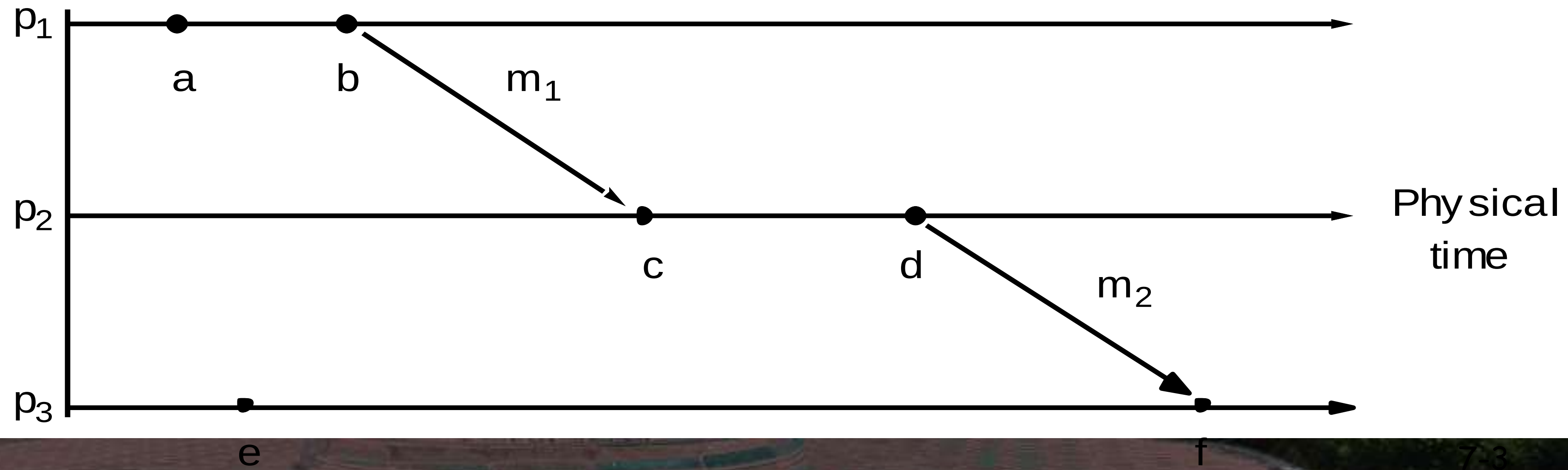
1. If two events occurred at the same process p_i (i = 1, 2, ... N) then

Not all events are related by $\rightarrow$
consider a and e (different processes and no chain of messages to relate them)
they are not related by $\rightarrow$; they are said to be concurrent; write as $a \parallel e$

$a \rightarrow b$ (at p_1) $c \rightarrow d$ (at p_2)

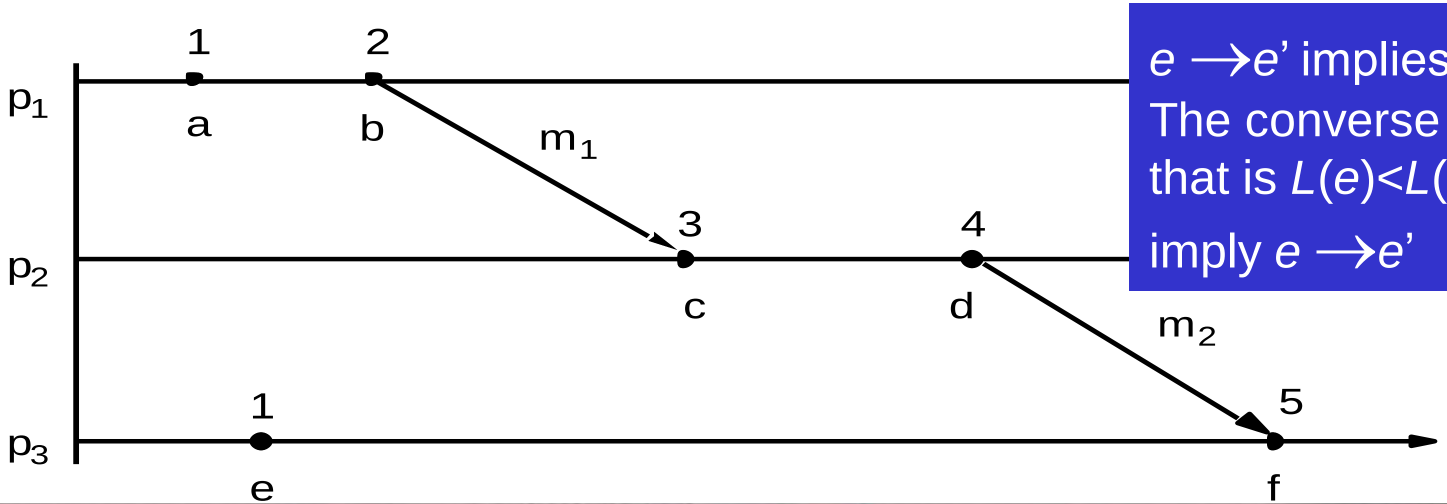
$b \rightarrow c$ because of m_1

also $d \rightarrow f$ because of m_2



Lamport's logical clocks

- A logical clock is a monotonically increasing software counter. It need not relate to a physical clock.
- Each process *pi* has a logical clock, *Li* which can be used to apply logical timestamps to events
  - LC1: *Li* is incremented by 1 before each event at process *pi*
  - LC2:
    - (a) when process *pi* sends message *m*, it piggybacks *t* = *Li*
    - (b) when *pj* receives *(m,t)* it sets Lj := *max*(Lj, *t*) and applies LC1 before timestamping the event *receive* (*m*)

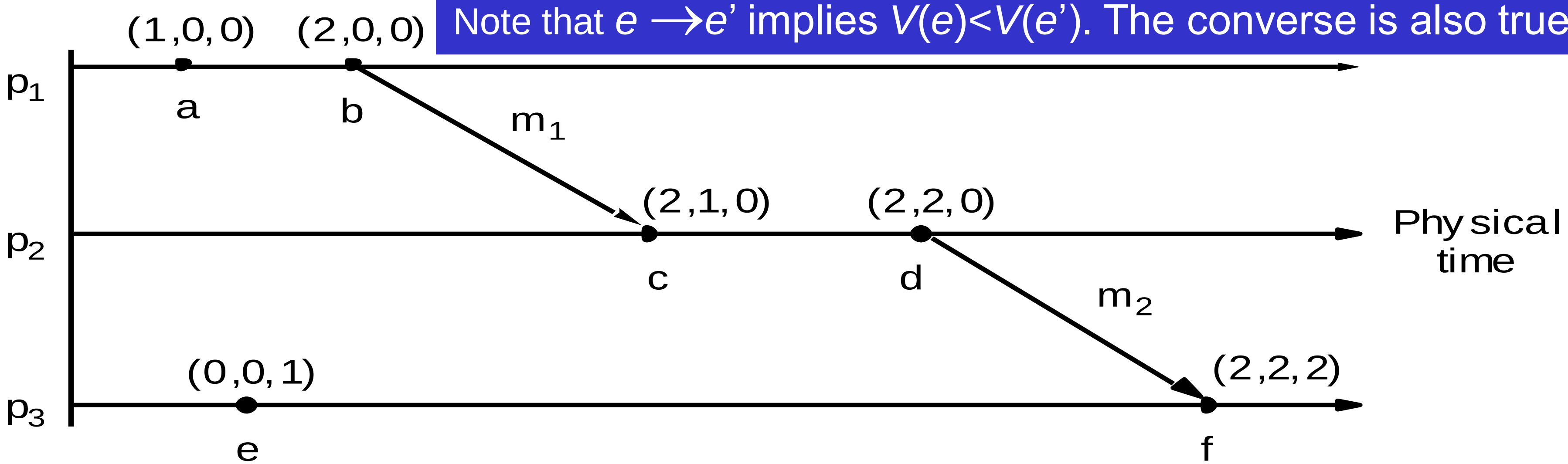


e →*e*' implies *L*(*e*)<*L*(*e*')
The converse is not true, that is *L*(*e*)<*L*(*e*') does not imply *e* →*e*'

Vector clocks

- Vector clock V_i at process p_i is an array of N integers
- VC1:initially $V_i[j] = 0$ for $i, j = 1, 2, \ldots N$
- VC2:before p_i timestamps an event it sets $V_i[i] := V_i[i] + 1$
- VC3:p_i piggybacks $t = V_i$ on every message it sends
- VC4:when p_i receives (m,t) it sets $V_i[j] := \max(V_i[j]\ , t[j])$ $j = 1, 2, \ldots N$ *(then before next event adds 1 to own element using VC2)*

Note that $e \rightarrow e'$ implies $V(e)<V(e')$. The converse is also true

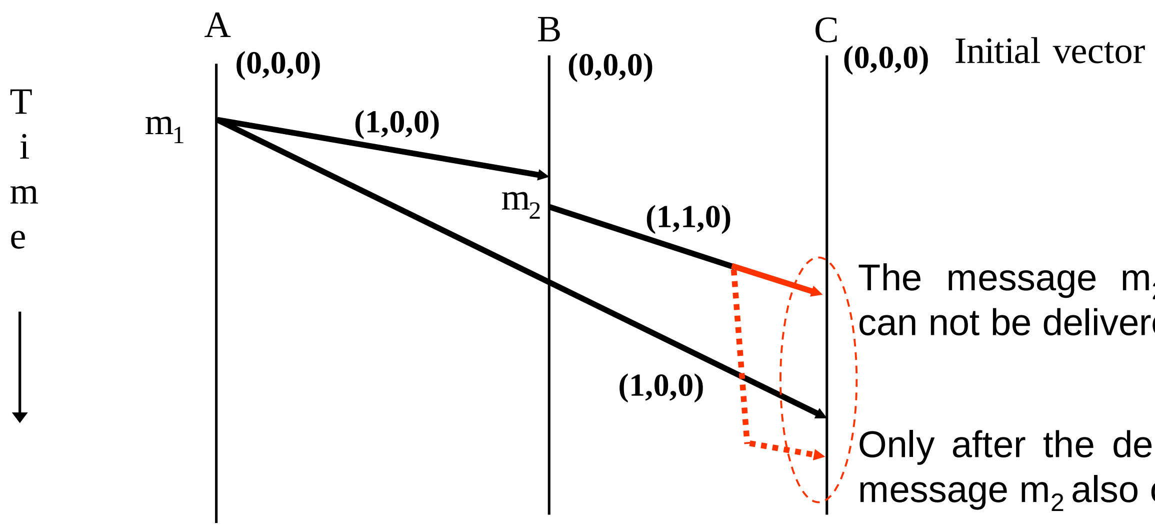


Causal Order deliver [BIR91]

Causal Ordering:

$$\textbf{If } send(m) \rightarrow send(m'),\ \textbf{then } \forall k \in g$$

$$delivery_k(m) \rightarrow delivery_k(m')$$



The message m_2 arrives but can not be delivered

Only after the delivery of m_1, The message m_2 also can be delivered.

Delivery condition

if $(VT(m')[i] = VT(p_j)[i] + 1$ **and** $VT(m')[k] \leq VT(p_j)[k] \ \ \forall\ (k \neq i,\ k=1\ldots n)$
then delivery(m)

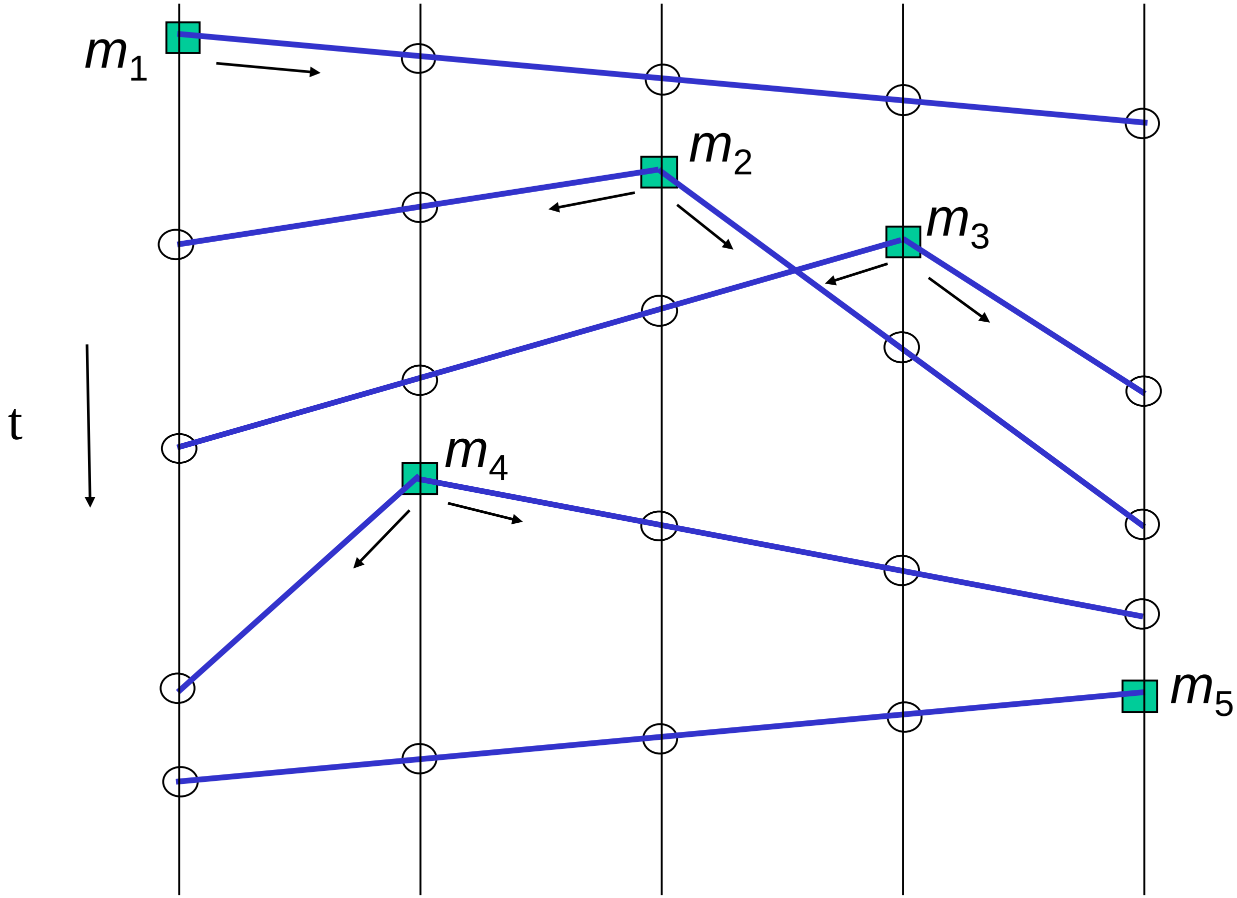
Causal Order deliver

- The general algorithm of vector time for causal delivery is as follows:
- Initially, $VT(pi)[j] = 0 \ \forall\ j=1\ldots n$.
- For each event *send*(*m*) at *pi*,
- $VT(pi)[i] = VT(pi)[i] + 1$.
- Each multicast message by process *pi* is timestamped with the updated value of *VT*(*pi*).
- For each event *deliveredj*(*m'*), *pj* modifies its vector time in the following manner:
- $VT(pj)[k]=VT(m')$

For each reception *receive(m') à pj , i≠j,   m'=(i,VT(m'),message)*

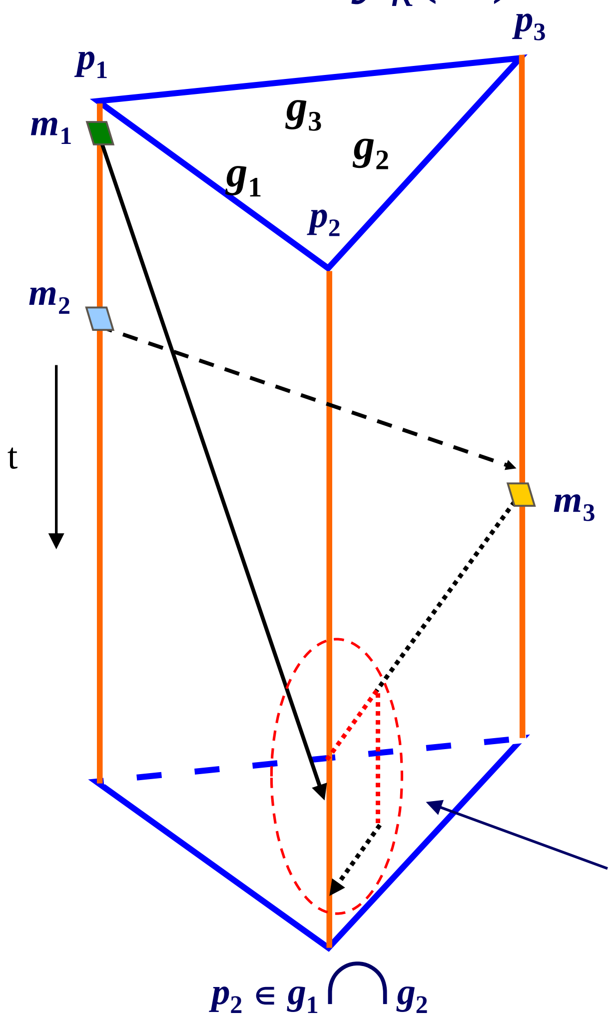
- *To enforce a causal delivery of m'*
- *i. Delivery condition*
- ***if not*** $(VT(m')[i] = VT(pj)[i] +1$ **and** $VT(m')[k] \leq VT(pj)[k] \ \forall\ (k \neq i, k=1\ldots n)$
- ***then***
- *wait*
- ***else***
- *ii*   *delivery(m)*

Problem

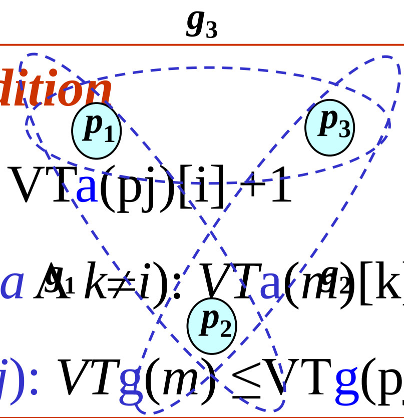


Causal Order deliver, Multigroup Case

If $send_i(m,g) \rightarrow send_j(m',g')$, ***then*** $\forall k \in g \cap g'$ $delivery_k(m) \rightarrow delivery_k(m')$



$m_1 \begin{cases} (1,0,x) \\ (x,0,0) \\ (0,x,0) \end{cases}$ $m_2 \begin{cases} (1,0,x) \\ (x,0,0) \\ (1,x,0) \end{cases}$ $m_3 \begin{cases} (1,0,x) \rightarrow g_1=\{p_1, p_2\} \\ (x,0,1) \rightarrow g_2=\{p_2, p_3\} \\ (1,x,0) \rightarrow g_3=\{p_1, p_3\} \end{cases}$

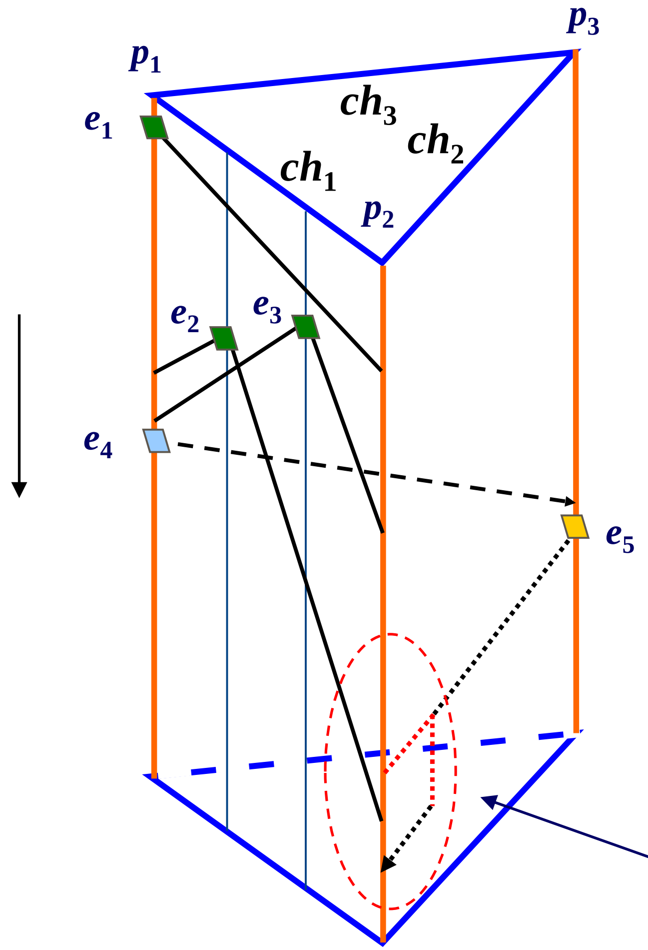


Delivery condition

- $VTa(m)[i] = VTa(pj)[i] +1$
- $\forall k : (pk \in ga \wedge k \neq i)$: $VTa(m)[k] \leq VT(pj)[k]$ and
- $\forall g : (g \in Gj)$: $VTg(m) \leq VTg(pj)$

Delivery of event m_3 must be delayed.

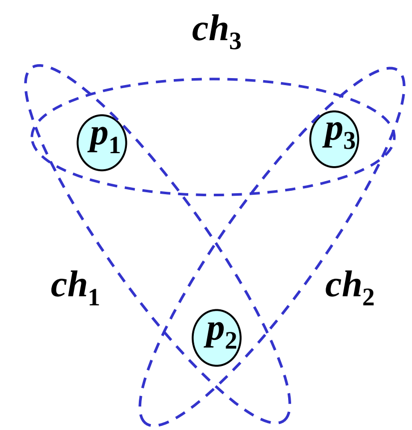
Exercise



e_1 { ?

.
.
.

e_5 { ?



$ch_1=\{?\}$
$ch_2=\{?\}$
$ch_3=\{?\}$

Delivery of event e_5 must be delayed.

$p_2 \in ch_1 \cap ch_2$

The Basic Principles (cont.)

The causal relation, denoted by $\rightarrow$:

1. $\langle x, a\rangle \rightarrow \langle y, b\rangle$ ***if*** $x{=}y \wedge a < b$
2. $\langle x, a\rangle \rightarrow \langle y, b\rangle$ **if** $\langle x, a\rangle$ *is the sending of an event and* $\langle y, b\rangle$ *is the delivery of that event* .
3. $\langle x, a\rangle \rightarrow \langle y, b\rangle$ ***if*** $\exists\langle z, c\rangle \mid (\langle x, a\rangle \rightarrow \langle z, c\rangle \wedge \langle z, c\rangle \rightarrow \langle y, b\rangle)$

Immediate Dependency Relation

The problem with causal ordering :

The amount of control information emitted for large values of $n = |G|$ is prohibitively high.

Immediate Dependency Relation $\downarrow$:

$$e \downarrow e' \Leftrightarrow [\,(e \rightarrow e') \wedge \forall\, e'' \in E, \neg (e \rightarrow e'' \rightarrow e')]$$

Immediate Dependency Relation (cont.)

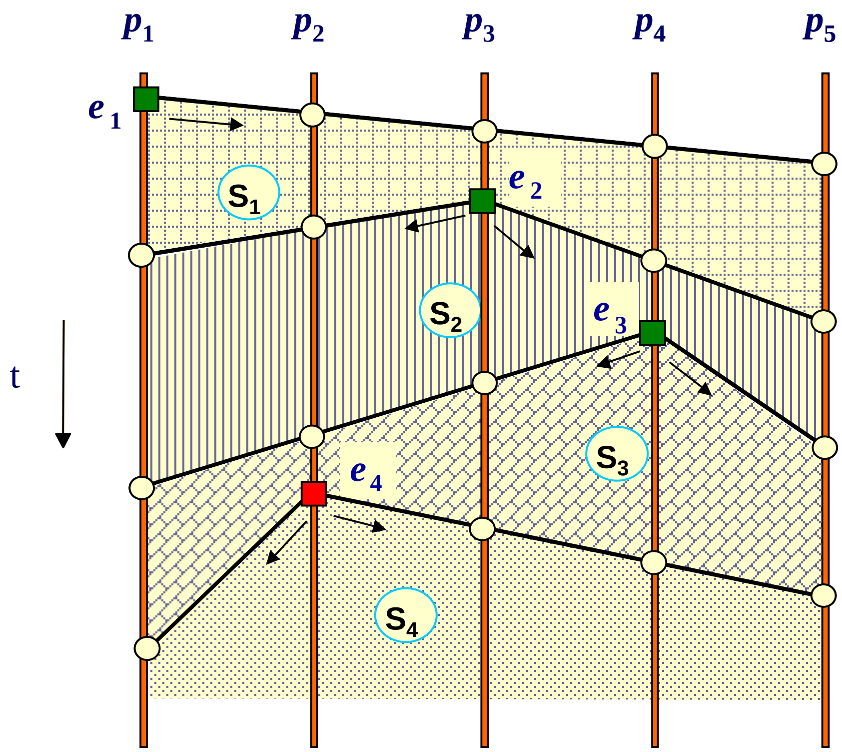
Causal Intra-Channel Ordering:

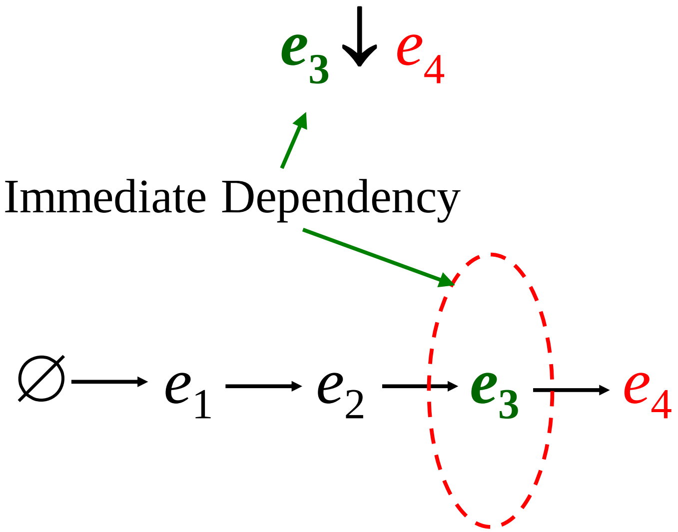
$$\textbf{\textit{If}}\ send(e) \rightarrow send(e'),\ \textbf{\textit{then}}\ \forall k \in c$$
$$delivery_k(e) \rightarrow delivery_k(e')$$

Proposition 1:

$$\textbf{\textit{If}}\ \forall\ e,e' \in E\ send(e) \downarrow send(e'),\ \textbf{\textit{then}}\ \forall k \in c$$
$$delivery_k(e) \rightarrow delivery_k(e')$$

Serial Events



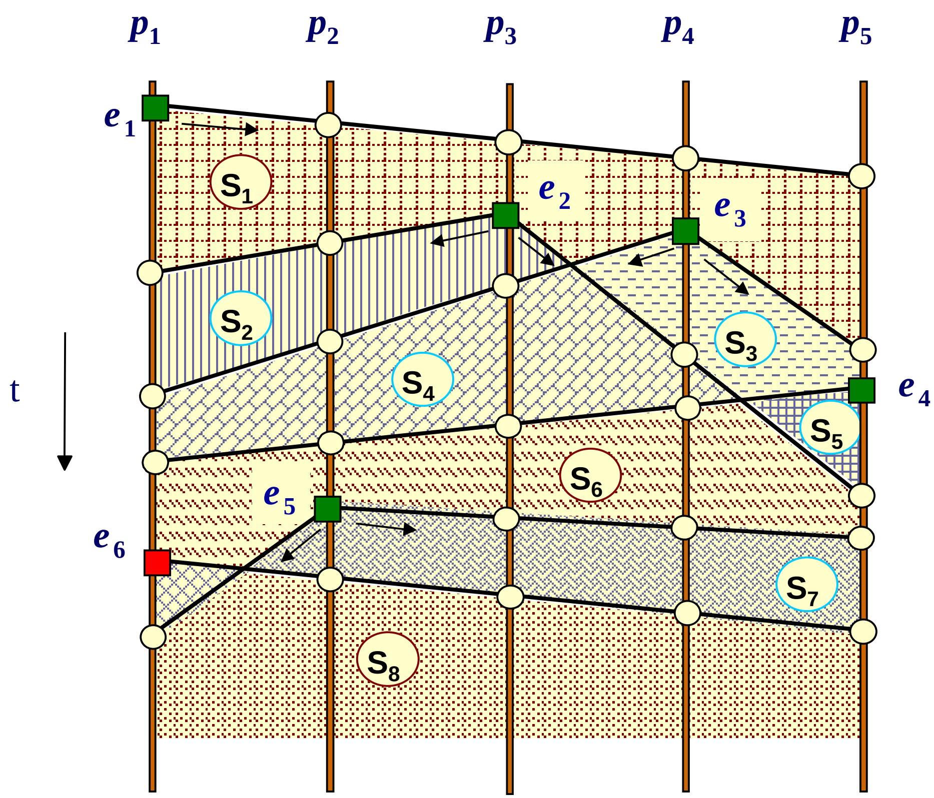


IDR Graph

Immediate Dependency Relation $\downarrow$:

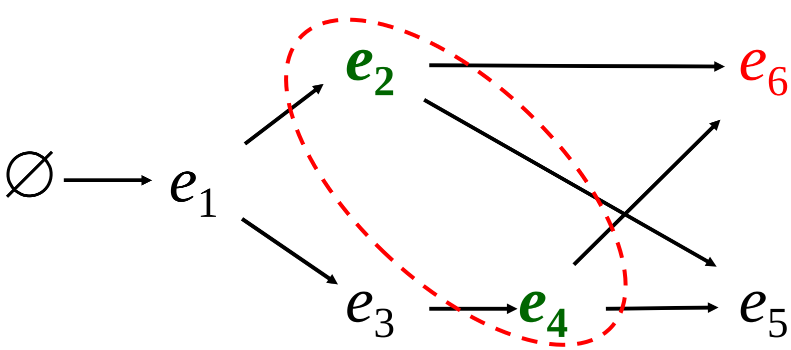
$e \downarrow e' \Leftrightarrow [\,(e \rightarrow e') \wedge \forall\, e'' \in E,\ \neg(e \rightarrow e'' \rightarrow e')]$

Concurrent Events



$$(e_2 \parallel (e_3 \downarrow e_4)) \downarrow e_6$$

Immediate Dependency



IDR Graph

Immediate Dependency Relation $\downarrow$:

$$e \downarrow e' \Leftrightarrow [\,(e \to e') \wedge \forall e'' \in E, \neg(e \to e'' \to e')]$$

Immediate Dependency Relation $\downarrow$:

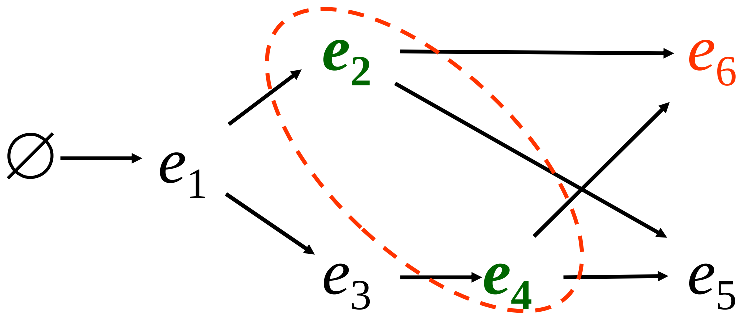
$$e \downarrow e' \Leftrightarrow [\,(e \rightarrow e') \wedge \forall\, e'' \in E, \neg(e \rightarrow e'' \rightarrow e')].$$

Concurrent Relation $\parallel$:

$$e \parallel e' \Leftrightarrow \neg\,(e \rightarrow e' \vee e' \rightarrow e)$$

Observation :

$$(e' \downarrow e \wedge e'' \downarrow e) \Rightarrow e' \parallel e''$$



For Multi-Channel Case

Immediate Inter-Channel Dependency Relation $\uparrow$:

$$(e,c)\uparrow(e',c') \Leftrightarrow [((e,c) \rightarrow (e', c'))\wedge\forall(e'', c'')\in E,$$
$$((e,c)\rightarrow (e'', c'')\rightarrow (e', c') \Rightarrow c''\neq c \wedge c''\neq c')]$$

Observation:

If only one channel exists in the system, ***then***

$$\uparrow = \downarrow$$

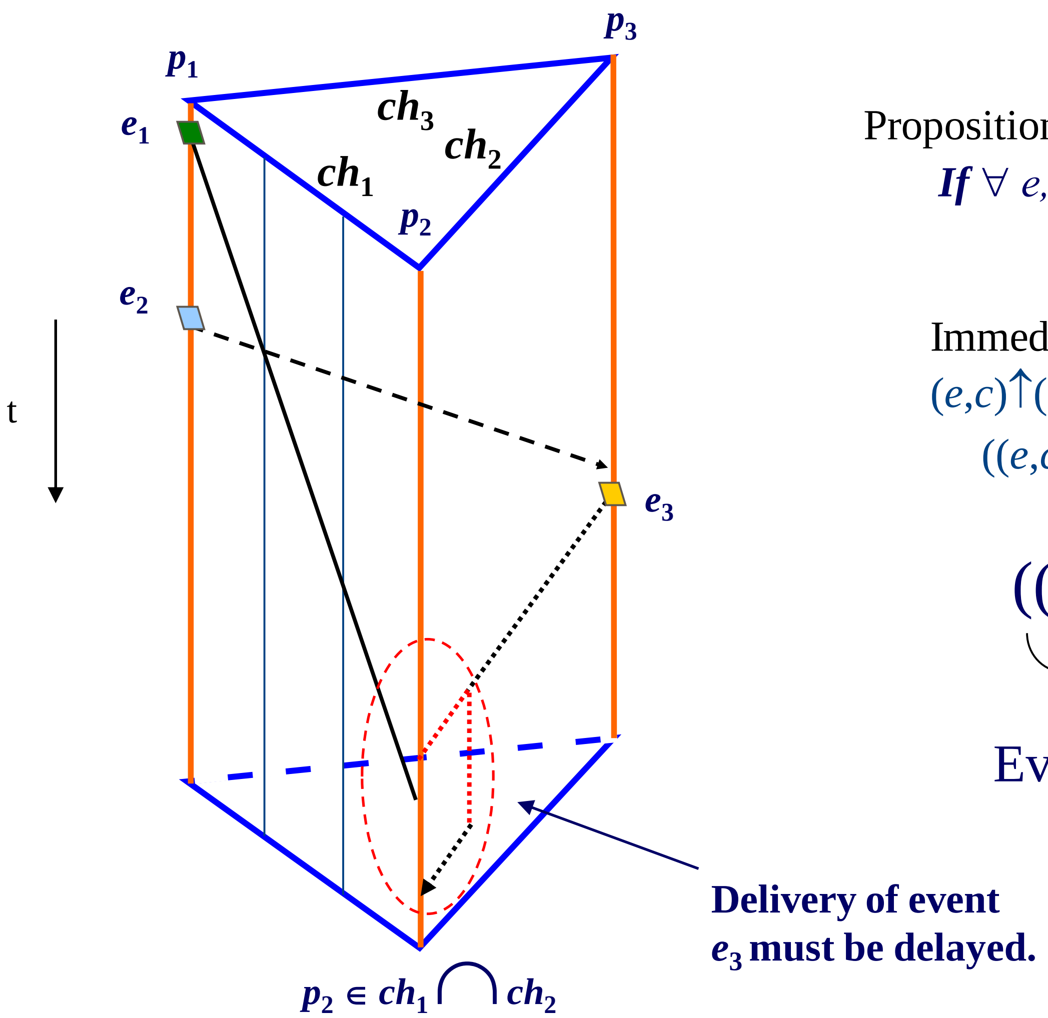
Causal Inter-Channel Ordering:

$$\textbf{\textit{If}}\ send(e,c) \rightarrow send(e',c'),\ \ \textbf{\textit{then}}\ \forall k \in c \cap c'$$
$$delivery_k(e) \rightarrow delivery_k(e')$$

Proposition 2:

$$\textbf{\textit{If}}\ \forall\ e,e' \in E\ send(e,c) \uparrow send(e',c'),\ \ \textbf{\textit{then}}\ \forall k \in c \cap c'$$
$$delivery_k(e) \rightarrow delivery_k(e')$$

Inter-channel Dependency



Proposition 2:

If $\forall\ e,e' \in E\ send(e,c) \uparrow send(e',c')$, ***then*** $\forall k \in c \cap c'$
$delivery_k(e) \rightarrow delivery_k(e')$

Immediate Inter-Channel Dependency Relation $\uparrow$:

$(e,c)\uparrow(e',c') \Leftrightarrow [((e,c) \rightarrow (e', c')) \wedge \forall(e'', c'') \in E,$
$((e,c) \rightarrow (e'', c'') \rightarrow (e', c') \Rightarrow c'' \neq c \wedge c'' \neq c')]$

$$\underbrace{((e_1,ch_1)\uparrow(e_2,ch_3))}_{\text{Events with IICDR to } e_3}\uparrow(e_3,ch_2)$$

The Multi-Channel Causal Algorithm

II. For each *message* diffused by p_i into group *d*

1. $VT(p_i)[i] = VT(p_i)[i] + 1$
2. **for all** $ci \in CI_i$ **:** $ci=(k, x, c, ch\_dests)$
3. **if** $d \in ci.ch\_dests$ **then**
4. $H_e \leftarrow H_e \cup \{(k,x,c)\}$
5. $ci.ch\_dests \leftarrow ci.ch\_dests \setminus d$
6. **endif**
7. **if** $ci.dests = \varnothing$ **then**
8. $CI_i \leftarrow CI_i \setminus ci$
9. **endif**
10. **endfor**
11. $e=(i, t=VT(p_i)[i], d, message, H_e)$
12. **send**(e) into **the group *d***
13. $CI_i \leftarrow CI_i \cup \{(i, VT(p_i)[i], d, CH_i \setminus d)\}$

The Multi-Channel Causal Algorithm

III. For each $\boldsymbol{e} = (k,t,d,event,H_e)$ **received by** p_j

To impose a causal delivery

Condition of Multi-group delivery

If not $(t = VT(p_j)[k] + 1 \wedge$

$x \leq VT(p_j)[l] \; \forall \; (l,x,c) \in H_e \big| c \in CH_j$) **then**

    wait

else

    Delivery(*message*)

    $VT(p_j)[k] = VT(p_j)[k] + 1$

    if $\left(\exists x \; (k,x,d) \in CI_j\right)$ **then**

        $CI_j \leftarrow CI_j \setminus \{ci_{k,x,d}\}$

    endif

    $CI_j \leftarrow CI_j \cup \{(k,t,d,CH_j)\}$

The Multi-Channel Causal Algorithm

for all $(l, x, c) \in H_e$

if $(c \in CH_j)$ **then**

if $\left(\exists y \; (l, y, c) \in CI_j\right)$ **then**/*$x \leq y$*/

if $x < y$ **then** /* don´t do anything */

endif

if $x = y$ **then**

if $(c \neq d)$ **then**

$MAJ(ci_{l,y,c}, d)$

else /* $c = d$ */

$CI_j \leftarrow CI_j \setminus ci_{l,y,c}$

endif

endif

endif

else /* $c \notin CH_i$ */

The Multi-Channel Causal Algorithm

else /* $c \notin CH_j$ */
  if $\left(\exists y\ (l, y, c) \in CI_j\right)$ **then**
    if $x < y$ **then /*** don´t do anything*/
    endif
    if $x = y$ **then**
      $MAJ(ci_{l,y,c}, d)$
    endif
    if $x > y$ **then**
      $VT(p_j)[l] = x$
      $CI_j \leftarrow CI_j \setminus \{ci_{l,y,c}\}$
      $CI_j \leftarrow CI_j \cup \{(l, x, c, CH_j)\}$
    endif
  else /* $\neg\exists y\ (l, y, c) \in CI_j$ ***/**
    if $(VT(p_j)[l] < x)$ **then**
      $VT(p_j)[l] = x$
      $CI_j \leftarrow CI_j \cup \{(l, x, c, CH_j)\}$
    endif

The Multi-Channel Causal Algorithm

IV. Updating

$MAJ(ci_{k,x,c}, d)$ {

    if $(c \neq d)$ **then**

        $ci_{k,x,c}.ch\_dests \leftarrow ci_{k,x,c}.ch\_dests \setminus d$

        if $\left(ci_{k,x,c}.ch\_dests = \varnothing\right)$ **then**

            $CI_j \leftarrow CI_j \setminus ci_{k,x,c}$

        endif

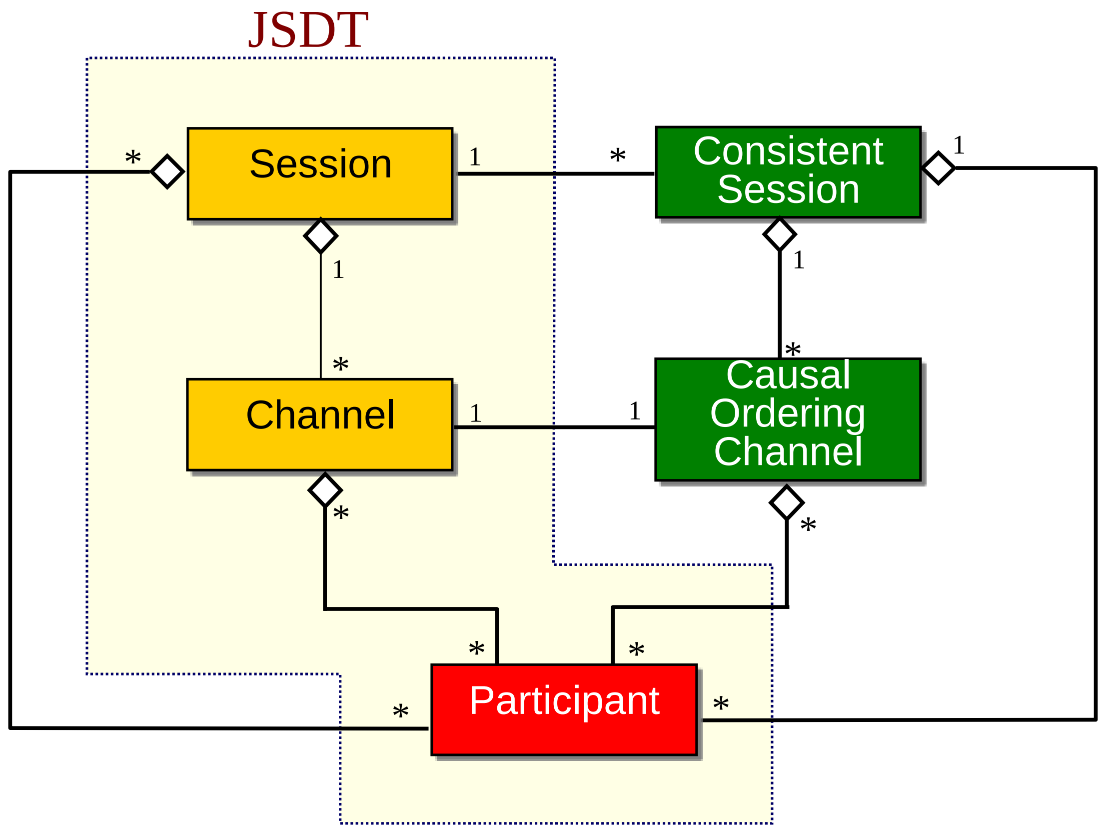
    else /* $c = d$ */

        $CI_j \leftarrow CI_j \setminus ci_{k,x,c}$

    endif

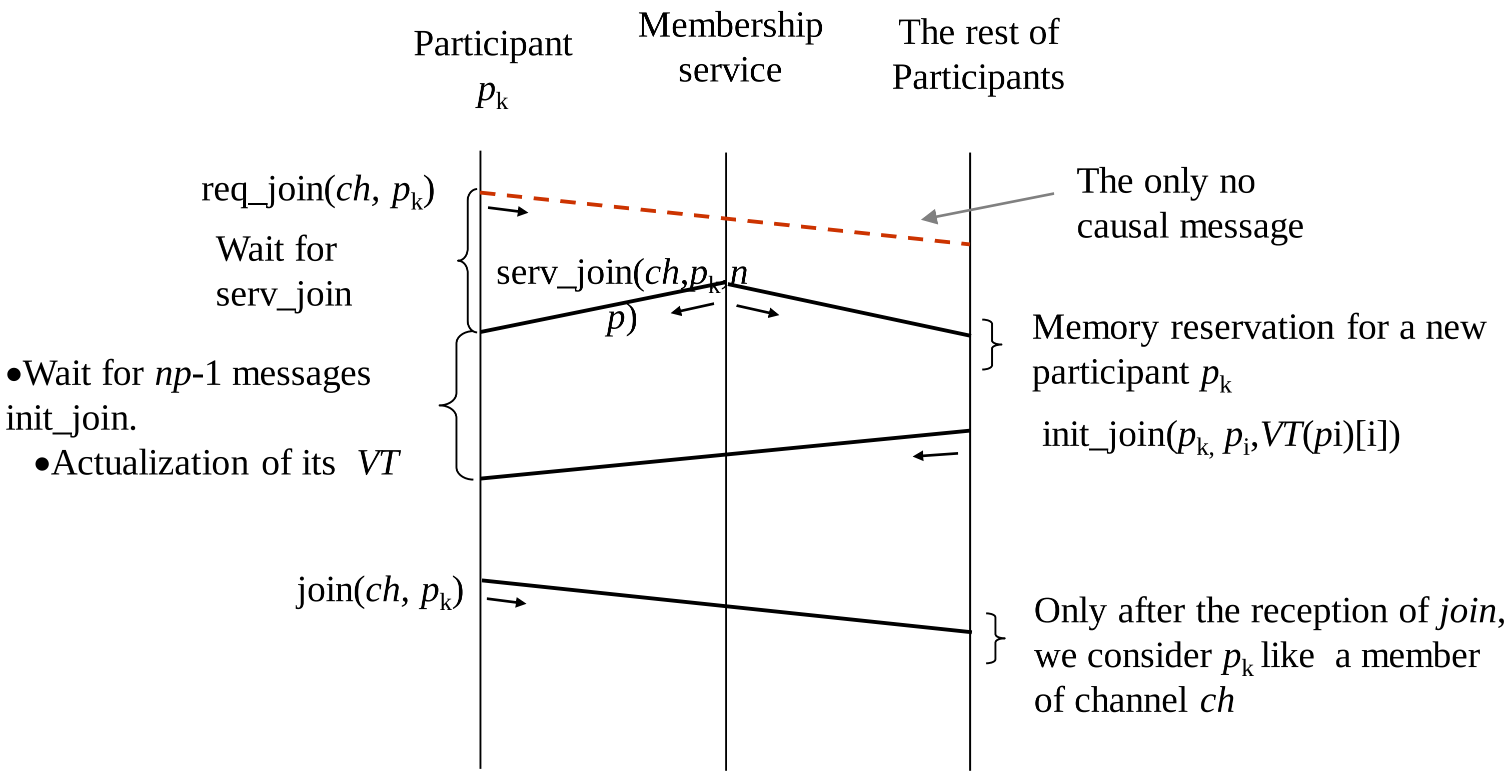
}

Implementation



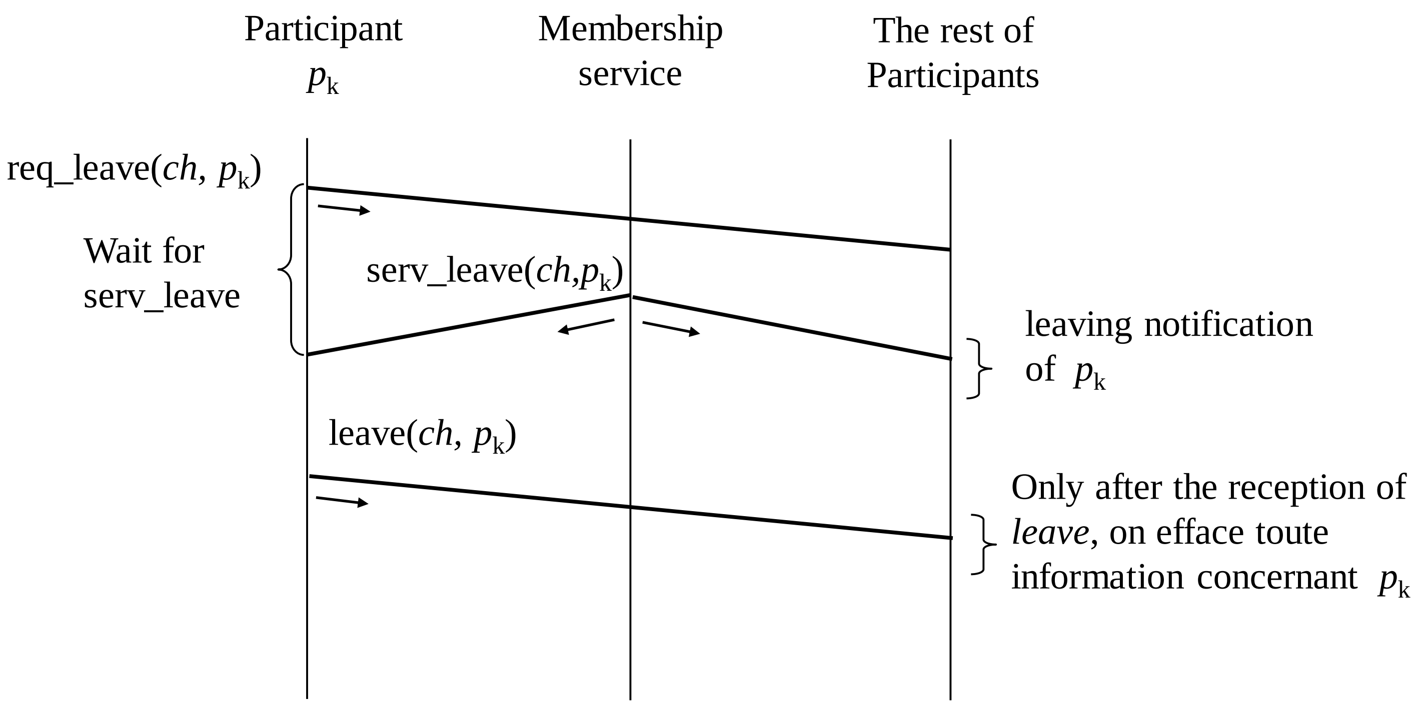
The MCP General Structure

Membership



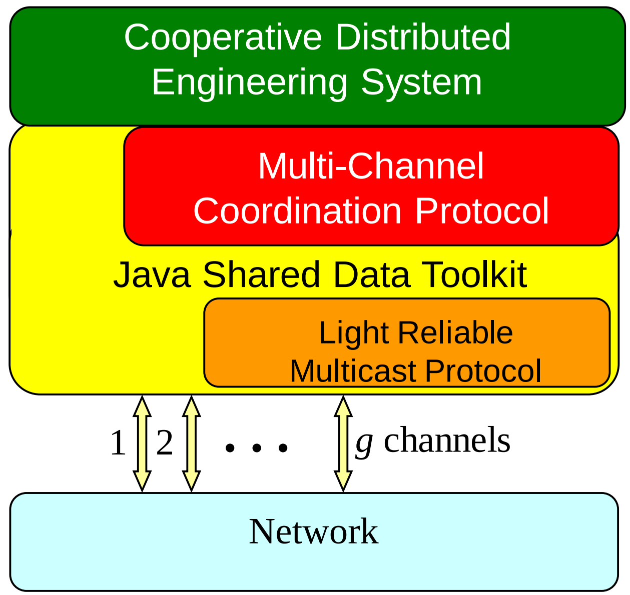
Join Procedure

Membership



Leave Procedure

Implementation (cont.)



The MCP Architecture