Chapter 5
Transport Layer

Chapter 5 outline

.1 transport-layer services

2 multiplexing and demultiplexing

3 connectionless transport: UDP

4 principles of reliable data transfer

5 connection-oriented transport: TCP

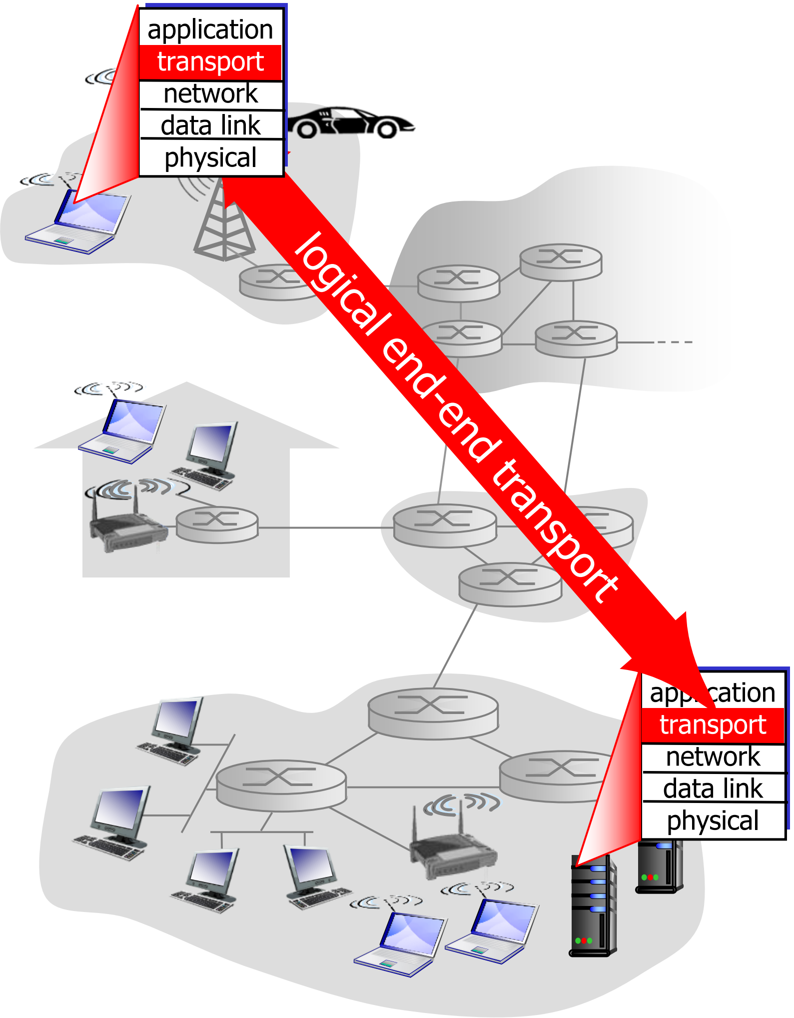
- segment structure
- reliable data transfer
- flow control
- connection management

6 principles of congestion control

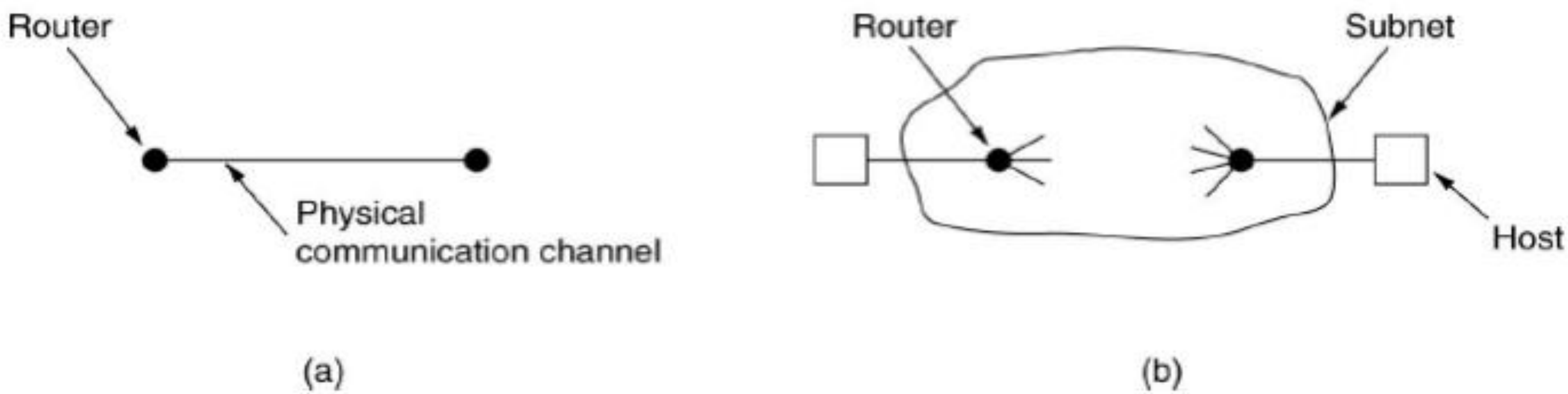
7 TCP congestion control

Transport services and protocols

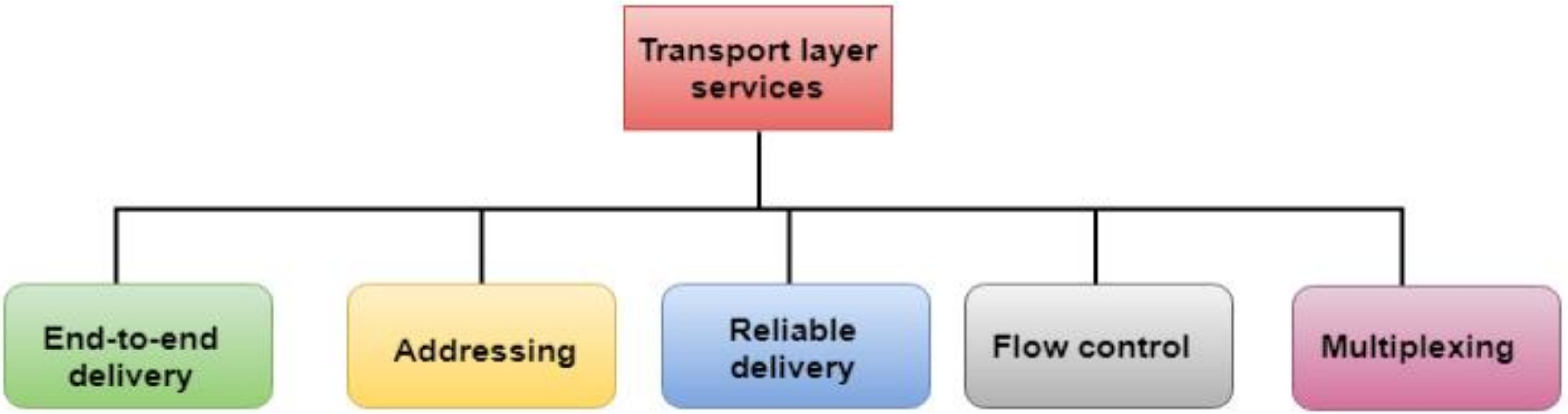
- provide *logical communication* between app processes running on different hosts
- transport protocols run in end systems
  - send side: breaks app messages into *segments*, passes to network layer
  - rcv side: reassembles segments into messages, passes to app layer
- more than one transport protocol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



Transport Protocol



(a)Environment of the data link layer.
(b)Environment of the transport layer.



End-to-end delivery:

The transport layer transmits the entire message to the destination. Therefore, it ensures the end-to-end delivery of an entire message from a source to the destination.

Reliable delivery:

The transport layer provides reliability services by retransmitting the lost and damaged packets.

Flow Control

Flow control is used to prevent the sender from overwhelming the receiver. If the receiver is **overloaded** with too much data, then the receiver discards the packets and asking for the retransmission of packets. This increases network congestion and thus, reducing the system performance. The transport layer is responsible for flow control. It uses the sliding window protocol that makes the data transmission more efficient as well as it controls the flow of data so that the receiver does not become overwhelmed.

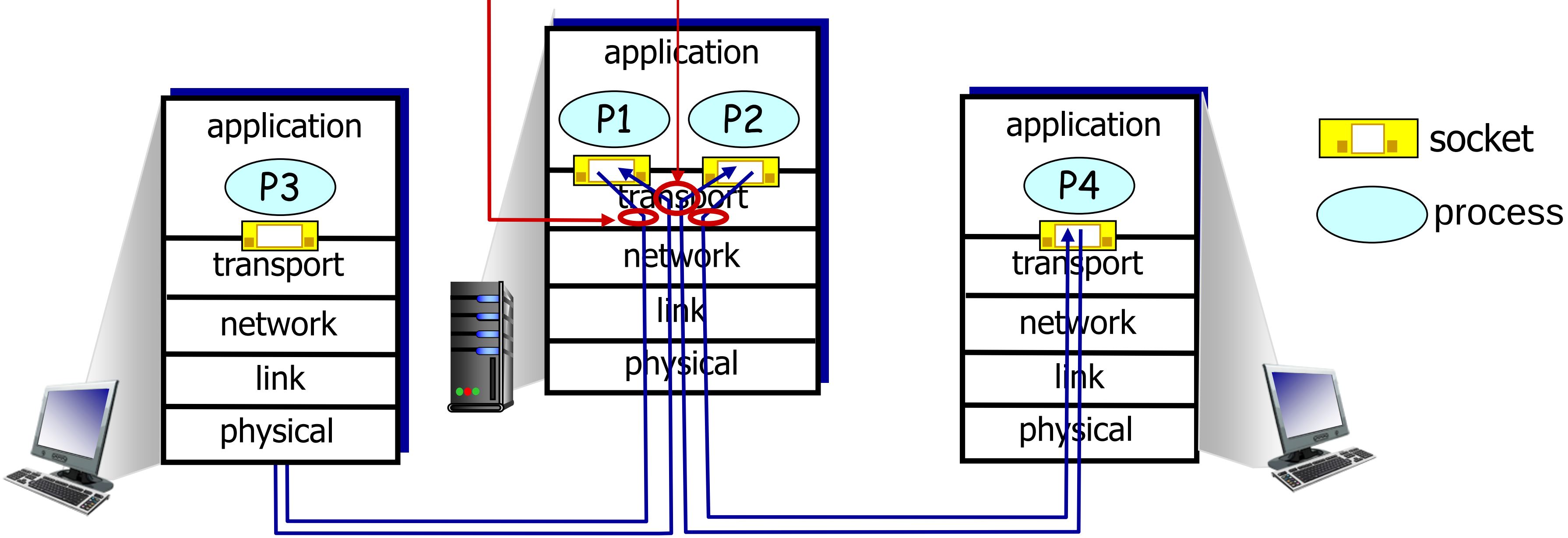
Addressing

The transport layer interacts with the functions of the session layer. Many protocols combine session, presentation, and application layer protocols into a single layer known as the application layer. In these cases, delivery to the session layer means the delivery to the application layer. Data generated by an application on one machine must be transmitted to the correct application on another machine. In this case, addressing is provided by the transport layer.

Multiplexing/demultiplexing

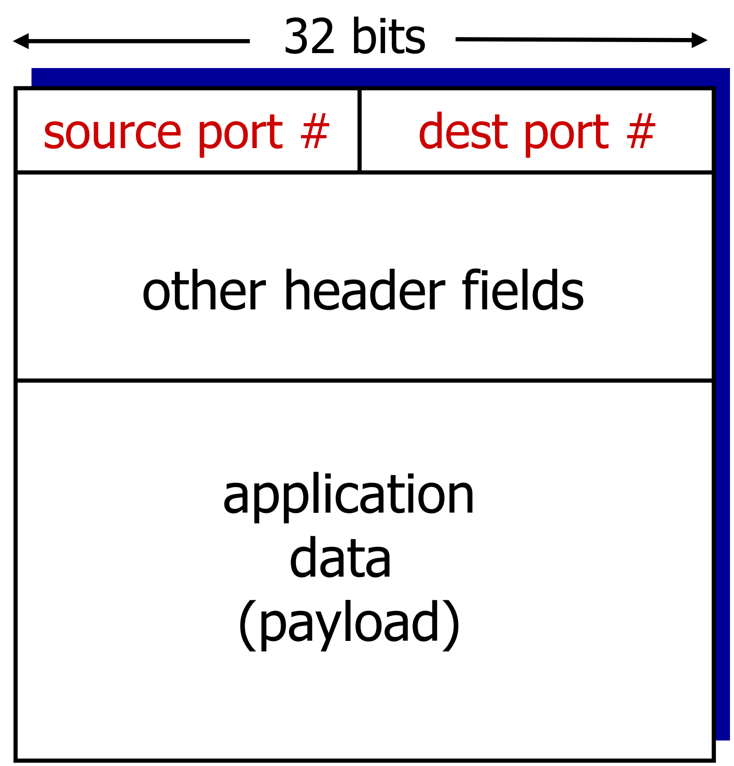
multiplexing at sender:
handle data from multiple sockets, add transport header (later used for demultiplexing)

demultiplexing at receiver:
use header info to deliver received segments to correct socket



How demultiplexing works

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



TCP/UDP segment format

Crash Recovery

Network and Transport layers automatically handle network and router crashes.
Issue here is how the transport layer recovers from host (end system) crashes.
Want clients continue to be able to work if the server quickly reboots.

Rule of Thumb: Recovery from a layer N crash can only be done by layer N + 1 because only the higher level retains enough status info to reconstruct where it was before the problem occurred.

Application needs to help recovering from a crash

- Transport can fail since A(ck) / W(rite) not atomic / C(rash)

	Strategy used by receiving host					
	First ACK, then write			First write, then ACK		
Strategy used by sending host	AC(W)	AWC	C(AW)	C(WA)	W AC	WC(A)
Always retransmit	OK	DUP	OK	OK	DUP	DUP
Never retransmit	LOST	OK	LOST	LOST	OK	OK
Retransmit in S0	OK	DUP	LOST	LOST	DUP	OK
Retransmit in S1	LOST	OK	OK	OK	OK	DUP

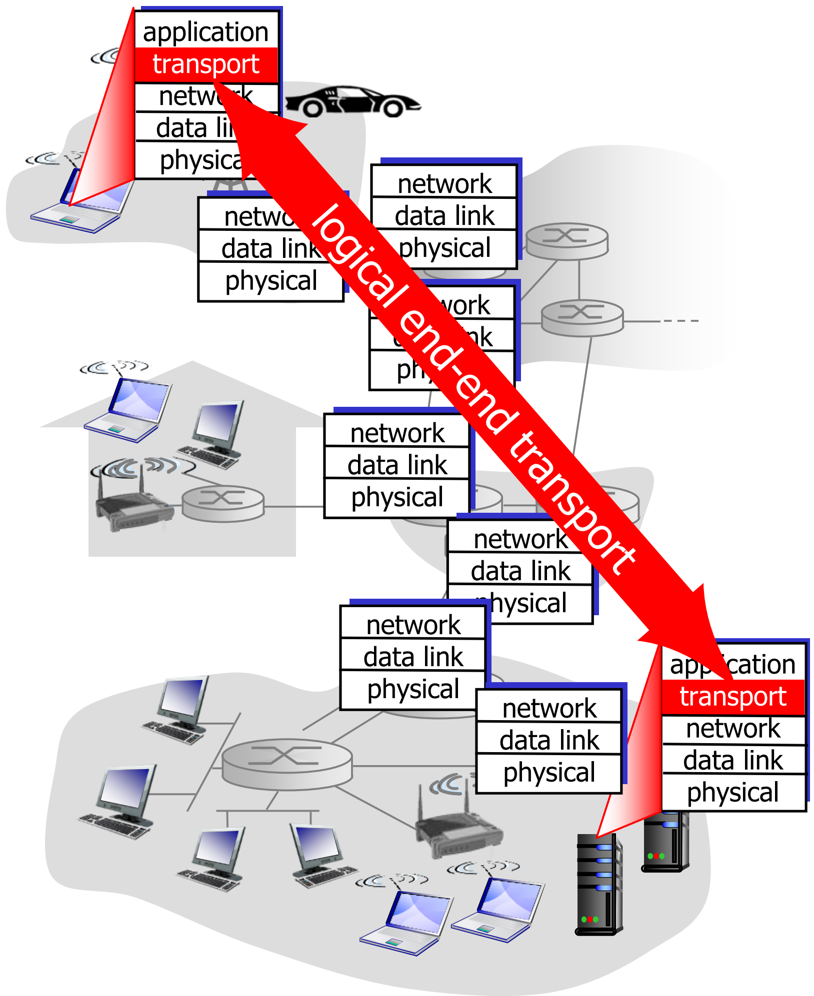
All 8 possible combos of client and server Strategies shown here.

OK = Protocol functions correctly
DUP = Protocol generates a duplicate message
LOST = Protocol loses a message

For each strategy there is some seq of events causing protocol to fail.

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



UDP

•UDP stands for **User Datagram Protocol**.

•UDP is a simple protocol and it provides nonsequenced transport functionality.

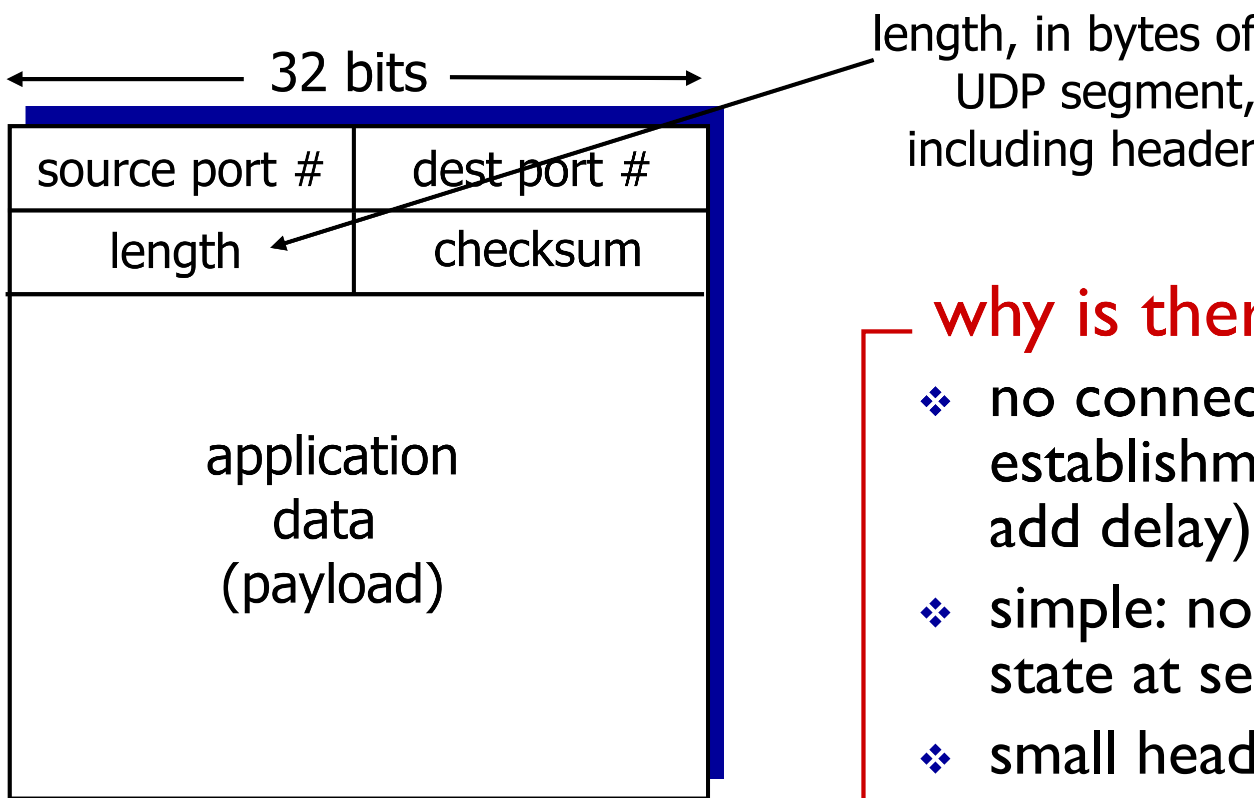
•UDP is a connectionless protocol.

•This type of protocol is used when reliability and security are less important than speed and size.

•UDP is an end-to-end transport level protocol that adds transport-level addresses, checksum error control, and length information to the data from the upper layer.

•The packet produced by the UDP protocol is known as a user datagram.

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

•**Source port address:** It defines the address of the application process that has delivered a message. The source port address is of 16 bits address.

•**Destination port address:** It defines the address of the application process that will receive the message. The destination port address is of a 16-bit address.

•**Total length:** It defines the total length of the user datagram in bytes. It is a 16-bit field.

•**Checksum:** The checksum is a 16-bit field which is used in error detection.

Disadvantages of UDP protocol

•UDP provides basic functions needed for the end-to-end delivery of a transmission.

•It does not provide any sequencing or reordering functions and does not specify the damaged packet when reporting an error.

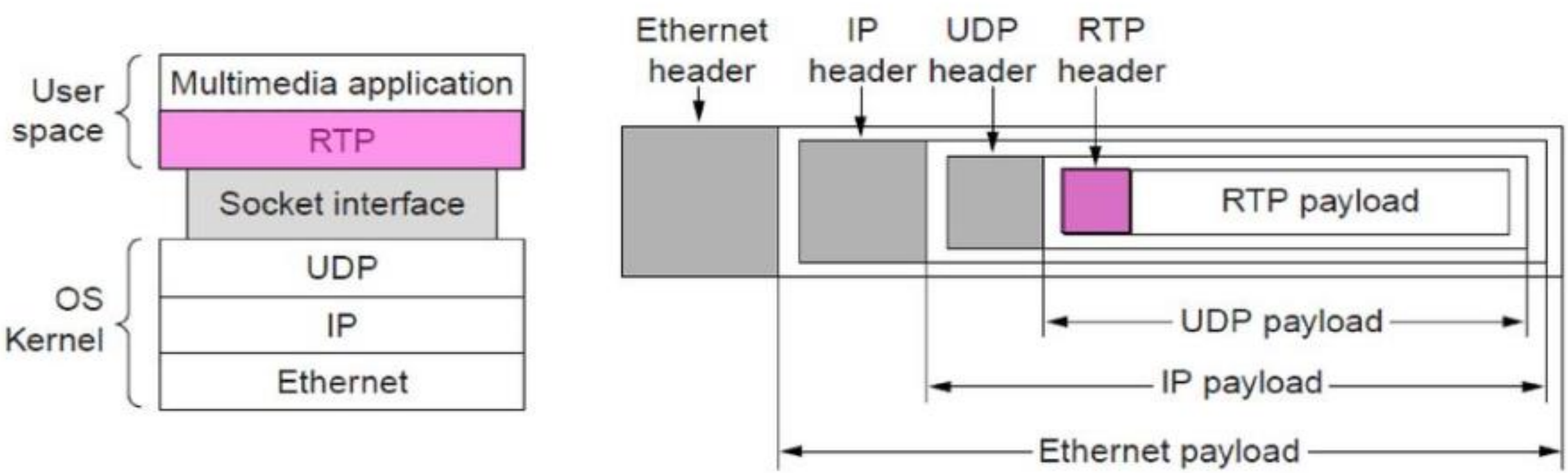
•UDP can discover that an error has occurred, but it does not specify which packet has been lost as it does not contain an ID or sequencing number of a particular data segment.

Remote Procedure Call is **a technique for building distributed systems**. Basically, it allows a program on one machine to call a subroutine on another machine without knowing that it is remote. **RPC is not a transport protocol:** rather, it is a method of using existing communications features in a transparent way.

Real-Time Protocol (1)

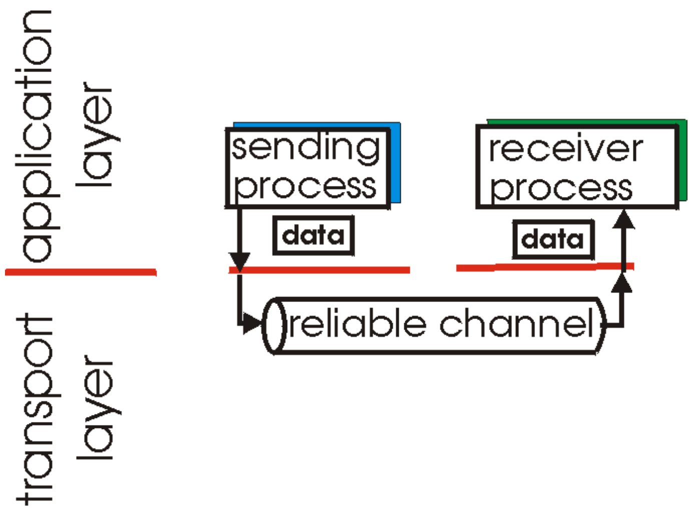
RTP (Real-time Transport Protocol) provides support for sending real-time multimedia over UDP (e.g., voice, video, …)

- Often implemented as part of the application



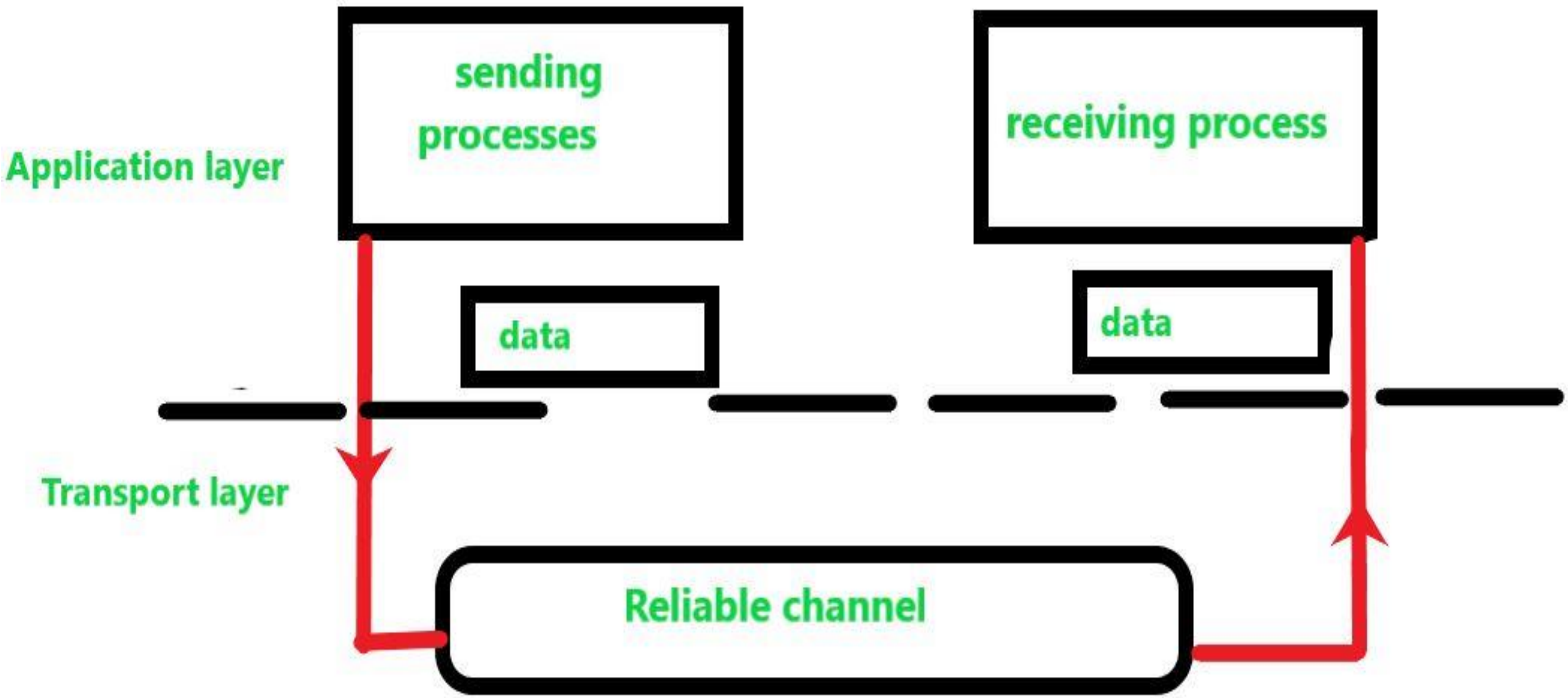
Principles of reliable data transfer

- important in application, transport, link layers



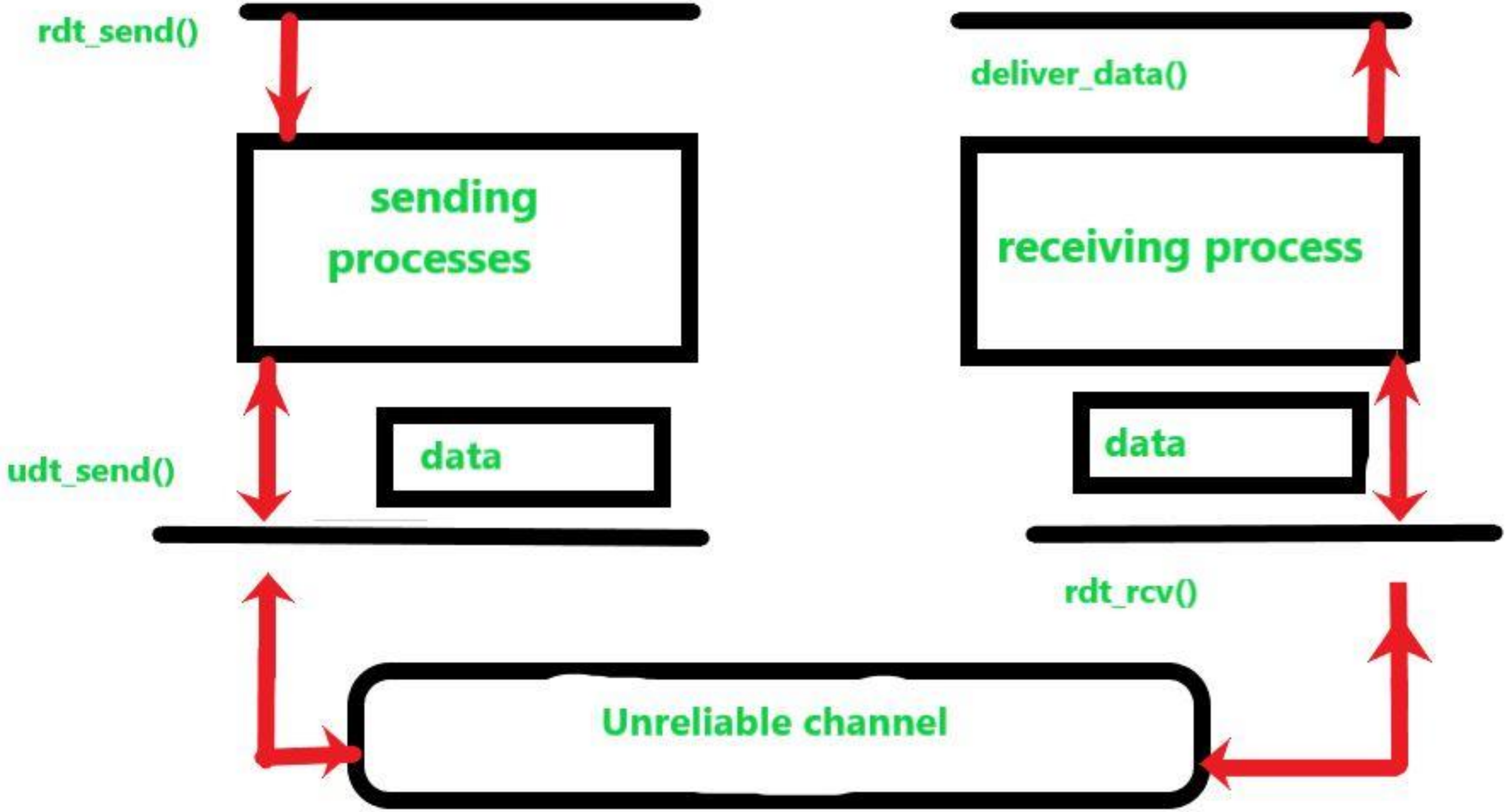
(a) provided service

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


sending processes
receiving process
Application layer
data
data
Transport layer
Reliable channel

These processes uses the logical communication to transfer data from transport layer to network layer and this transfer of data should be reliable and secure. The data is transferred in the form of packets but the problem occurs in reliable transfer of data.

In this model, we have design the sender and receiver sides of a protocol over a reliable channel. In the reliable transfer of data the layer receives the data from the above layer breaks the message in the form of segment and put the header on each segment and transfer. Below layer receives the segments and remove the header from each segment and make it a packet by adding to header.

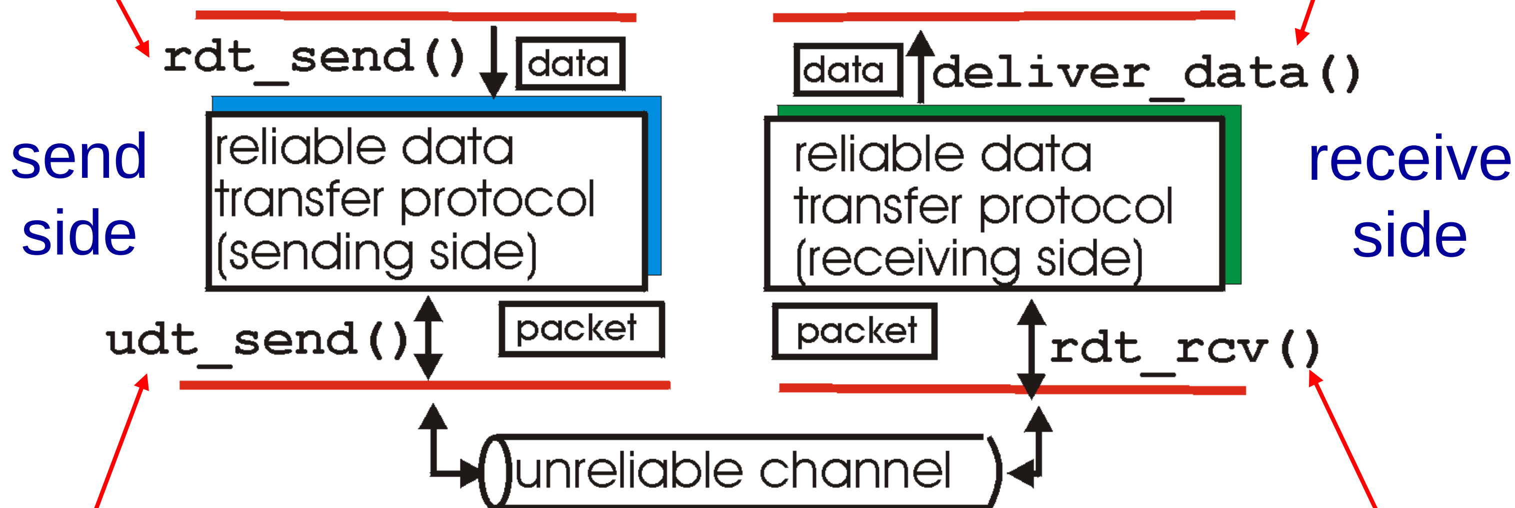

rdt_send()
deliver_data()
sending processes
receiving process
udt_send()
data
data
rdt_rcv()
Unreliable channel

The data which is transferred from the above has no transferred data bits corrupted or lost, and all are delivered in the same sequence in which they were sent to the below layer this is reliable data transfer protocol. This service model is offered by TCP to the Internet applications that invoke this transfer of data.

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

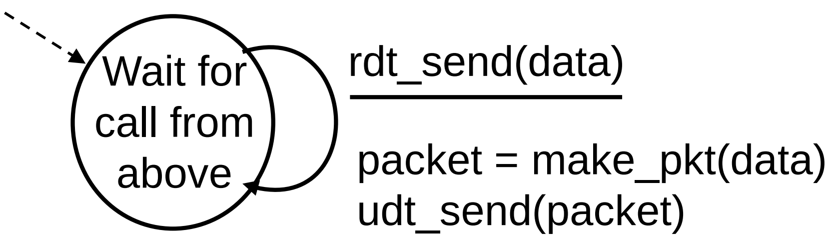
Finite State Machines

- A **finite state machine** is a model of behavior composed of states, transitions and actions.
  - A **state** stores information about the past, i.e. it reflects the input changes from the system start to the present moment.
  - A **transition** indicates a **state change** and is described by a **condition/event** that would need to be fulfilled to enable the transition.
  - An **action** is a description of an **activity** that is to be performed at a given moment.

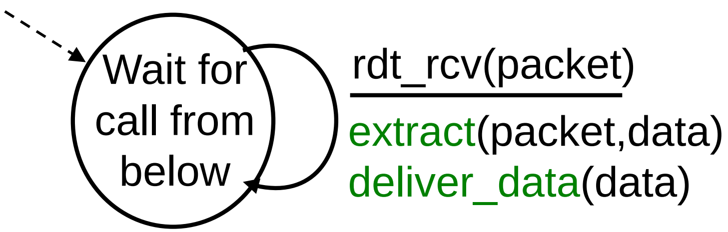
rdt1.0: reliable transfer over a reliable channel

- underlying channel perfectly reliable
  - no bit errors
  - no loss of packets

  - sender sends data into underlying channel
  - receiver reads data from underlying channel



sender

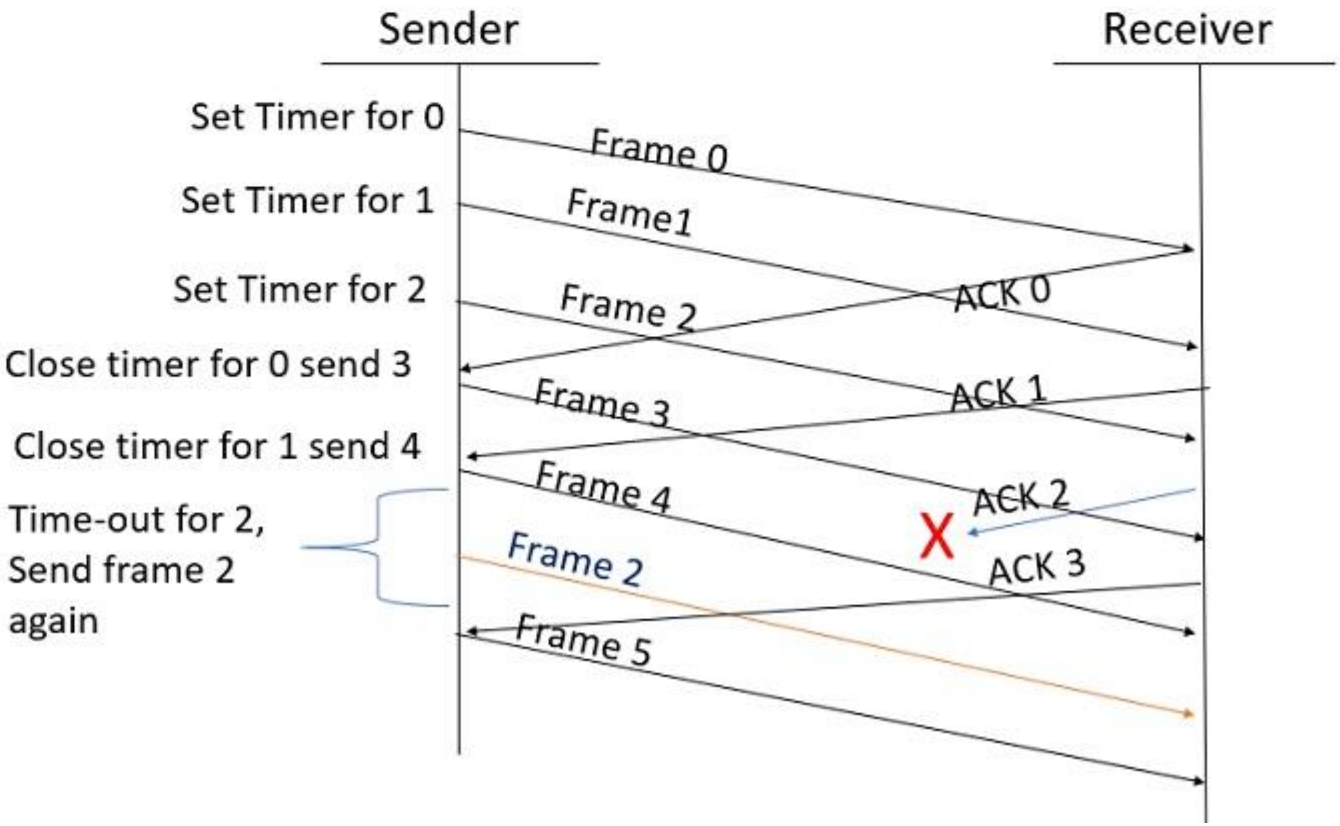


receiver

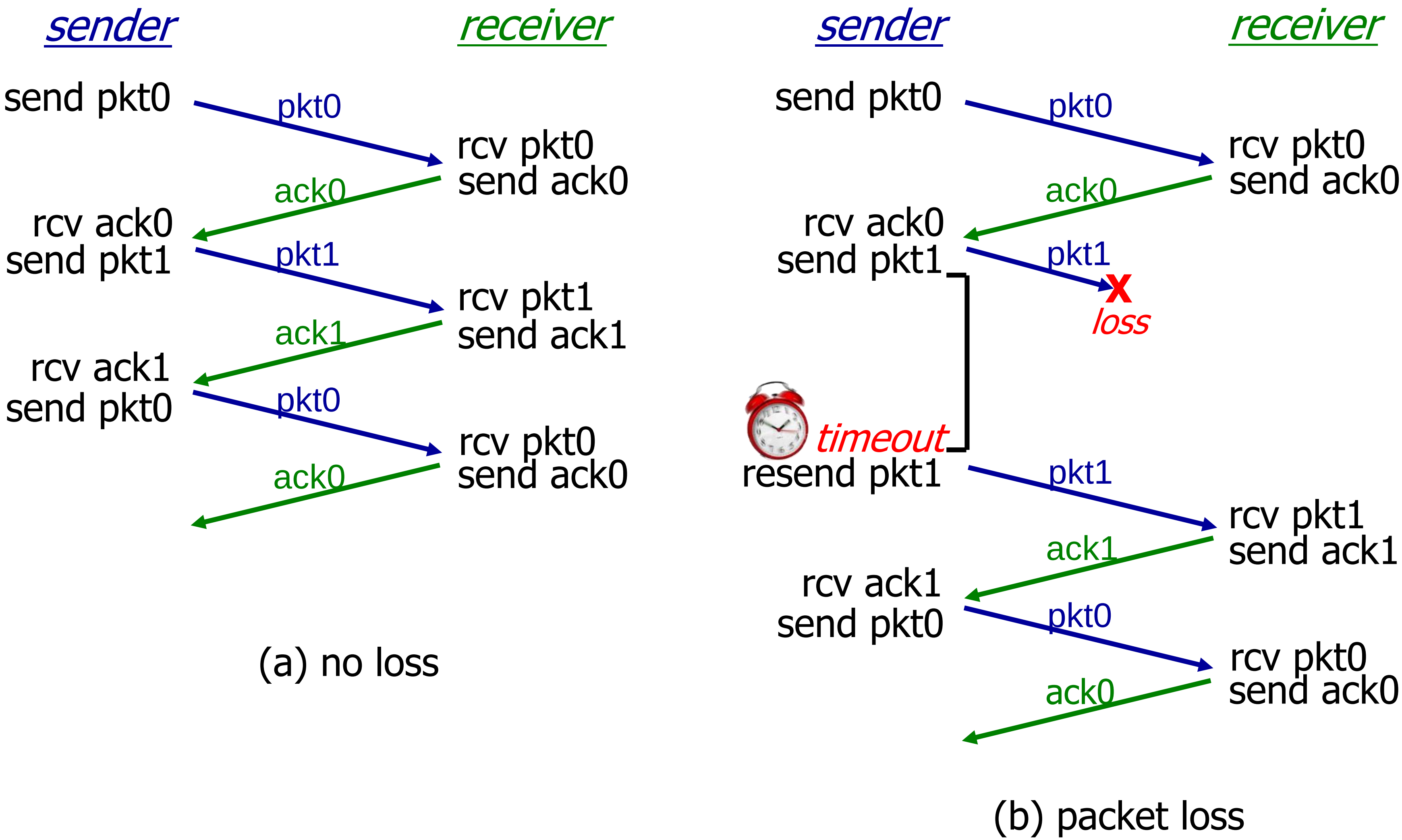
Selective Repeat ARQ

In the selective repeat, the sender sends several frames specified by a window size even without the need to wait for individual acknowledgement from the receiver as in Go-Back-N ARQ. In selective repeat protocol, the retransmitted frame is received out of sequence.

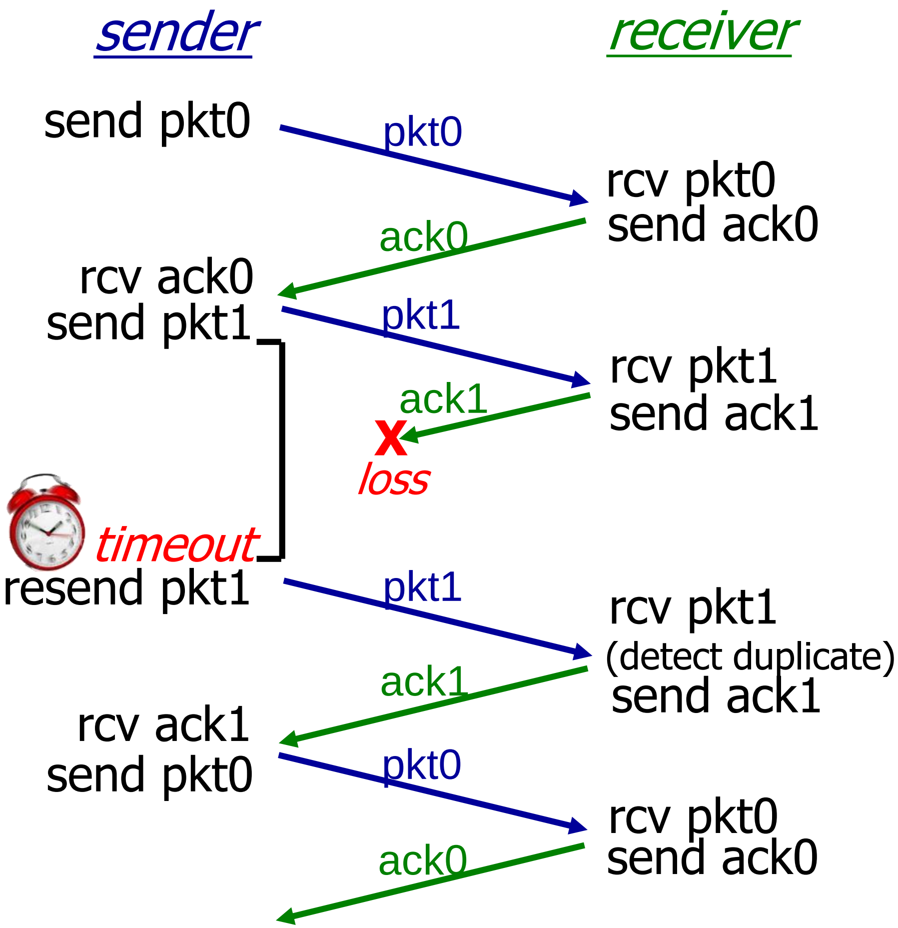
In Selective Repeat ARQ only the lost or error frames are retransmitted, whereas correct frames are received and buffered.
The receiver while keeping track of sequence numbers buffers the frames in memory and sends NACK for only frames which are missing or damaged. The sender will send/retransmit a packet for which NACK is received.


Sender
Receiver
Set Timer for 0
Frame 0
Set Timer for 1
Frame1
Set Timer for 2
ACK 0
Frame 2
Close timer for 0 send 3
ACK 1
Frame 3
Close timer for 1 send 4
Frame 4
ACK 2
Time-out for 2,
Send frame 2
again
Frame 2
ACK 3
Frame 5

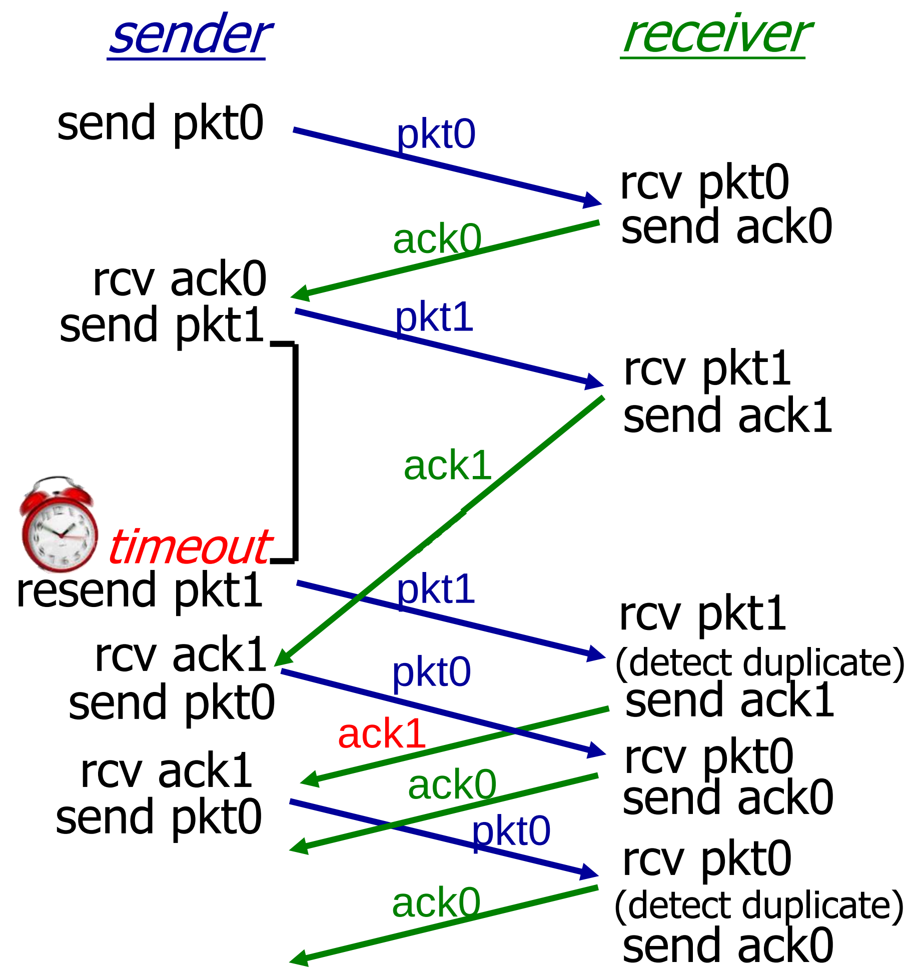
rdt3.0 in action



rdt3.0 in action



(c) ACK loss



(d) premature timeout/ delayed ACK

Performance of rdt3.0

- rdt3.0 is correct, but performance stinks
- e.g.: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:

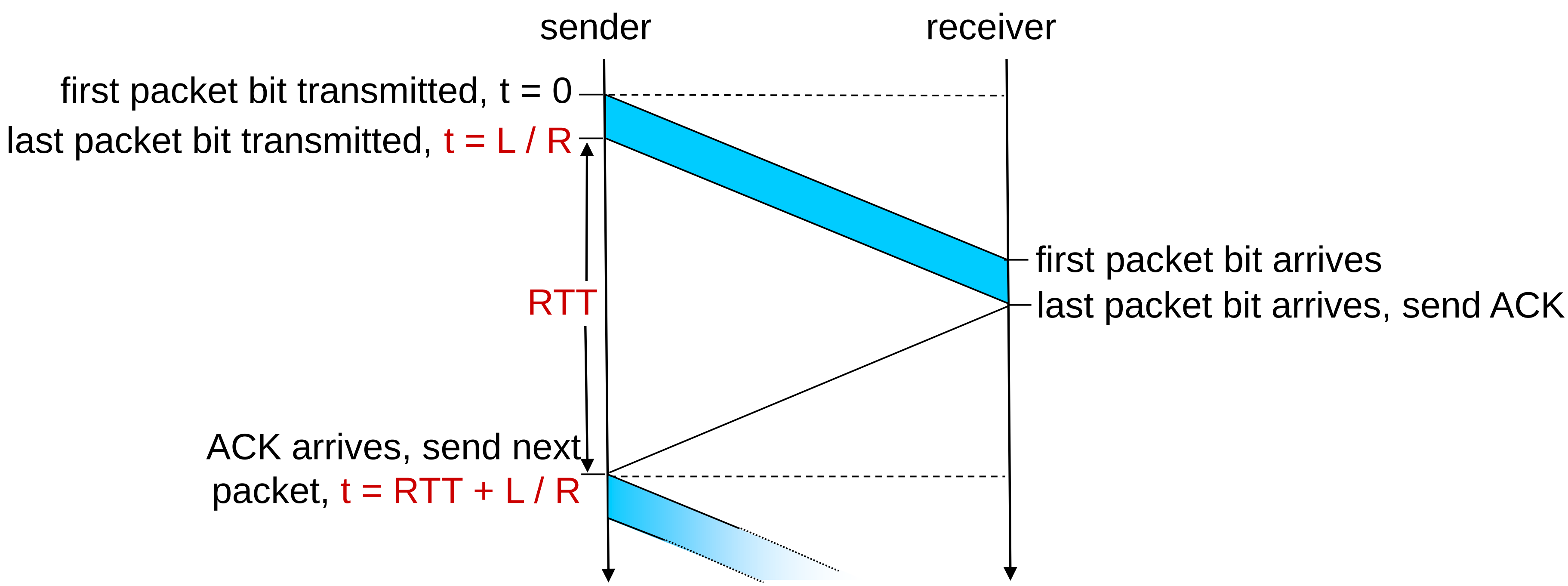
$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$

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

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

  - if RTT=30 msec, 1KB pkt every 30 msec: 33kB/sec thruput over 1 Gbps link
- network protocol limits use of physical resources!

rdt3.0: stop-and-wait operation

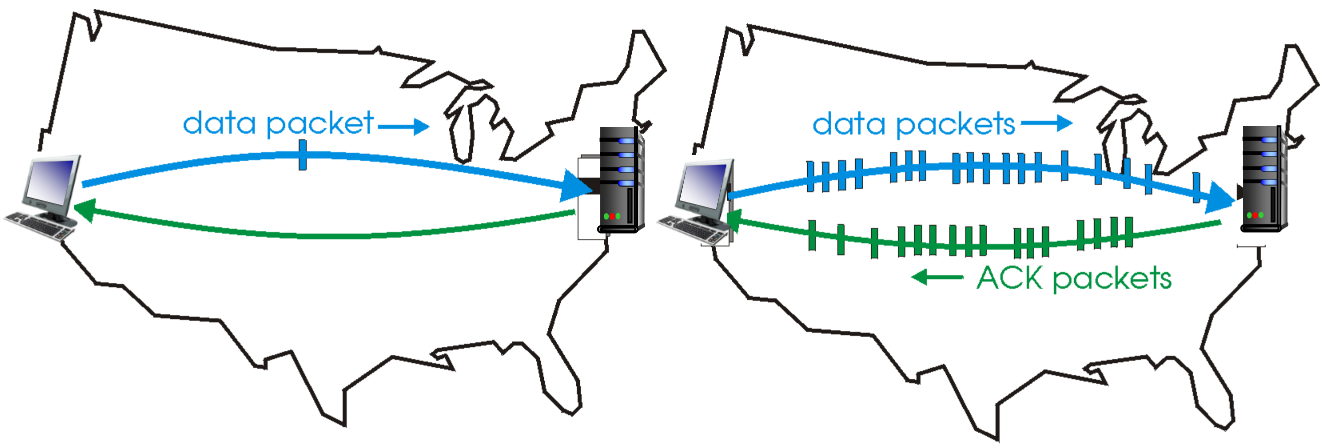


$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

Pipelined protocols

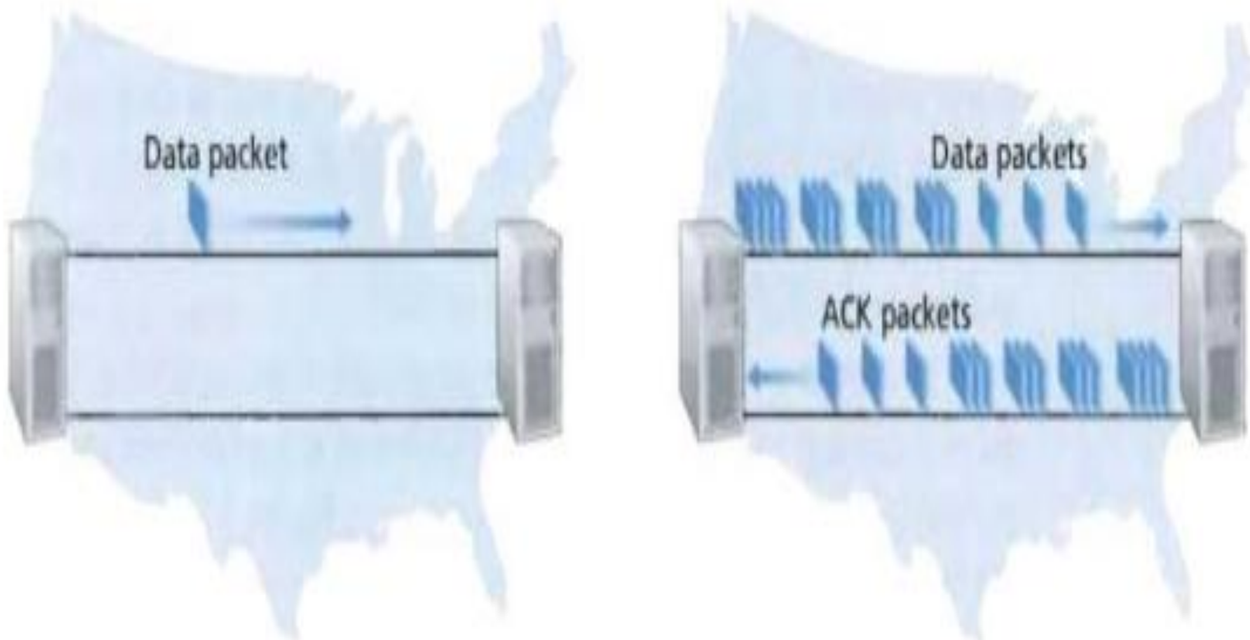
pipelining: sender allows multiple, "in-flight", yet-to-be-acknowledged pkts

- 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*



a. A stop-and-wait protocol in operation

b. A pipelined in operation

Pipelined Reliable Data Transfer Protocols

- Two basic approaches
  - Go-Back-N
  - Selective repeat

Pipelined protocols: overview

Go-back-N:

- sender can have up to N unacked packets in pipeline
- receiver only sends *cumulative ack*
  - doesn't ack packet if there's a gap
- sender has timer for oldest unacked packet
  - when timer expires, retransmit *all* unacked packets

Selective Repeat:

- sender can have up to N unack'ed packets in pipeline
- receiver sends *individual ack* for each packet
- sender maintains timer for each unacked packet
  - when timer expires, retransmit only that unacked packet

GBN: sender extended FSM

rdt_send(data)

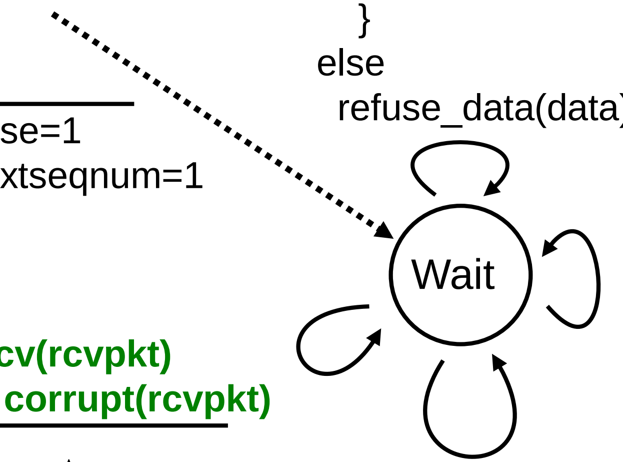
---

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



timeout

---

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

rdt_rcv(rcvpkt) && corrupt(rcvpkt)

Λ

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)

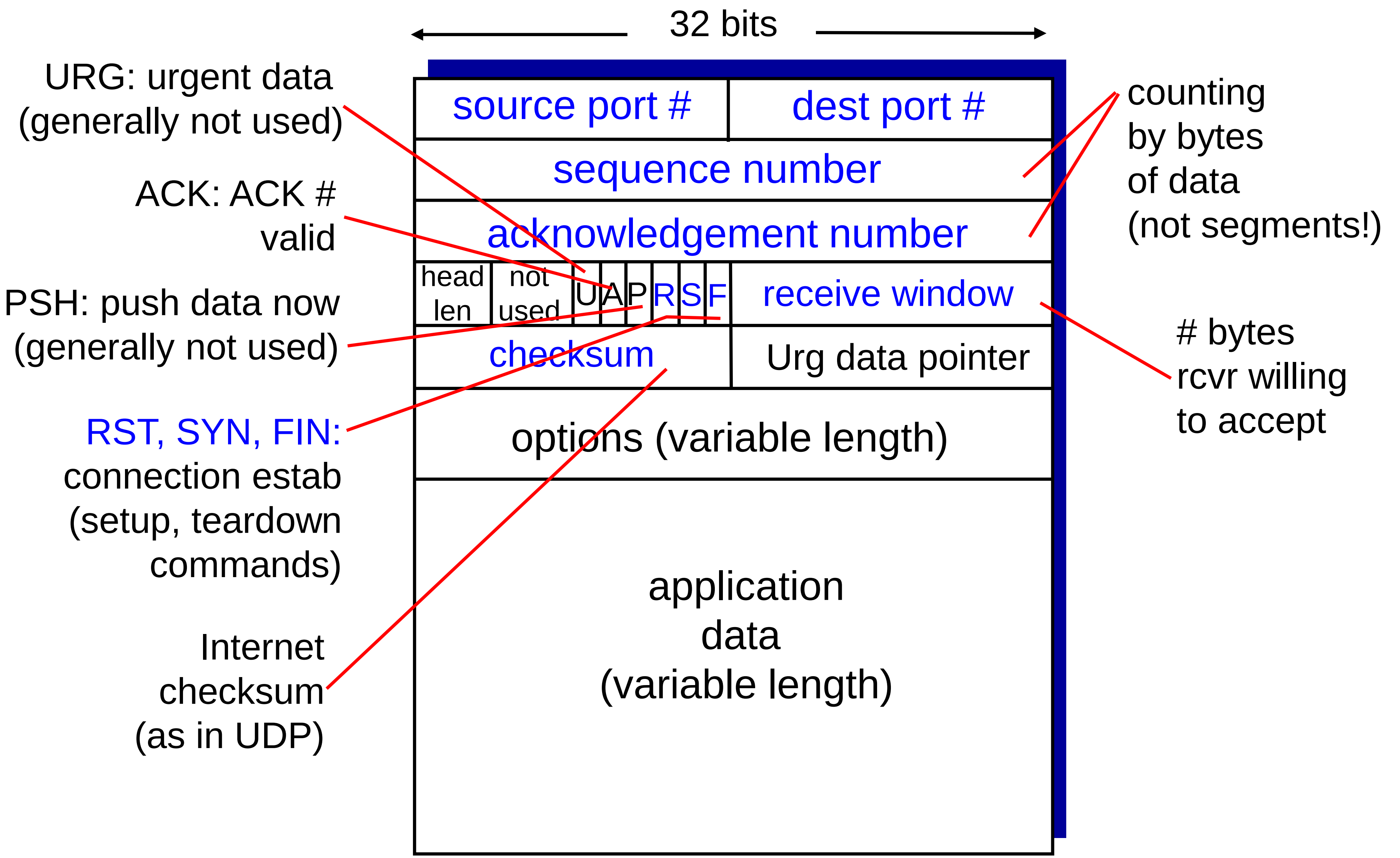
---

base = getacknum(rcvpkt)+1
If (base == nextseqnum)
  stop_timer
 else
  start_timer

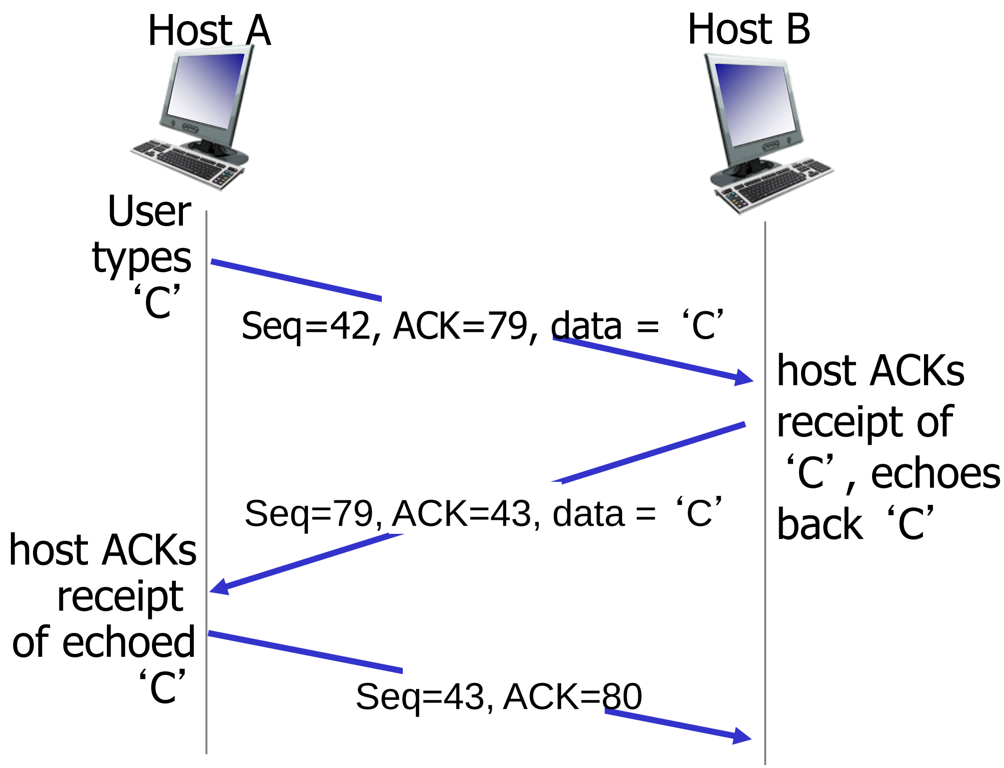
TCP: Overview

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

TCP segment structure

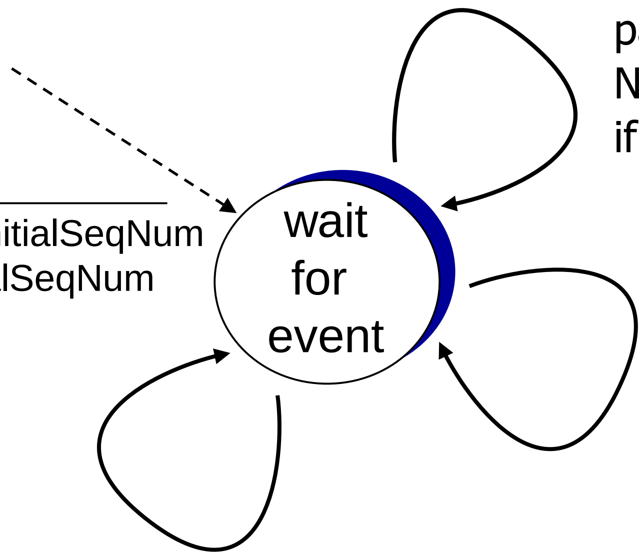


TCP seq. numbers, ACKs



simple telnet scenario

TCP sender (simplified)



Λ

NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

data received from application above

---

create segment, seq. #: NextSeqNum
pass segment to IP (i.e., “send”)
NextSeqNum = NextSeqNum + length(data)
if (timer currently not running)
    start timer

timeout

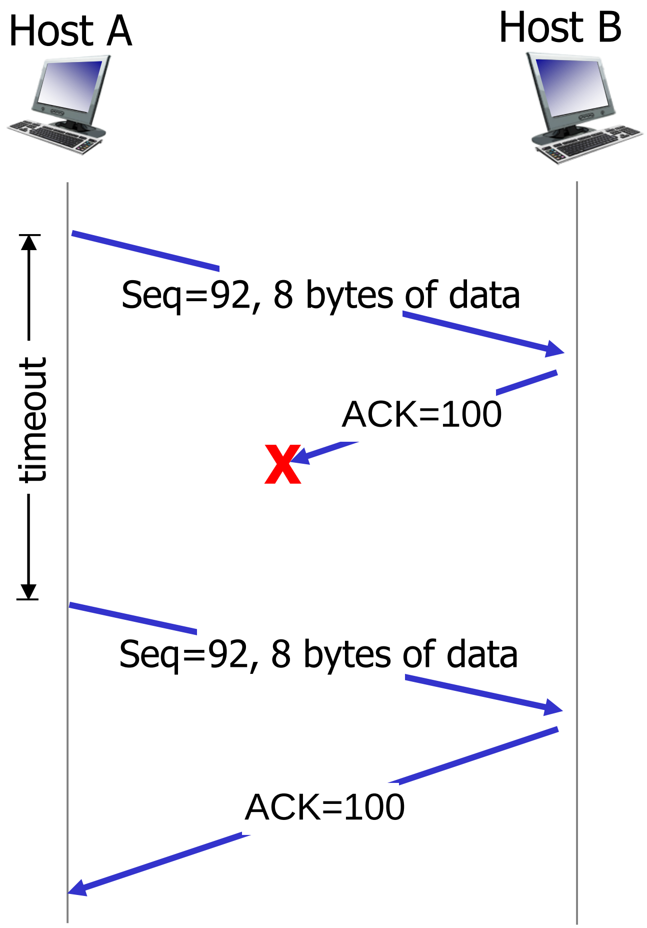
---

retransmit not-yet-acked segment
    with smallest seq. #
start timer

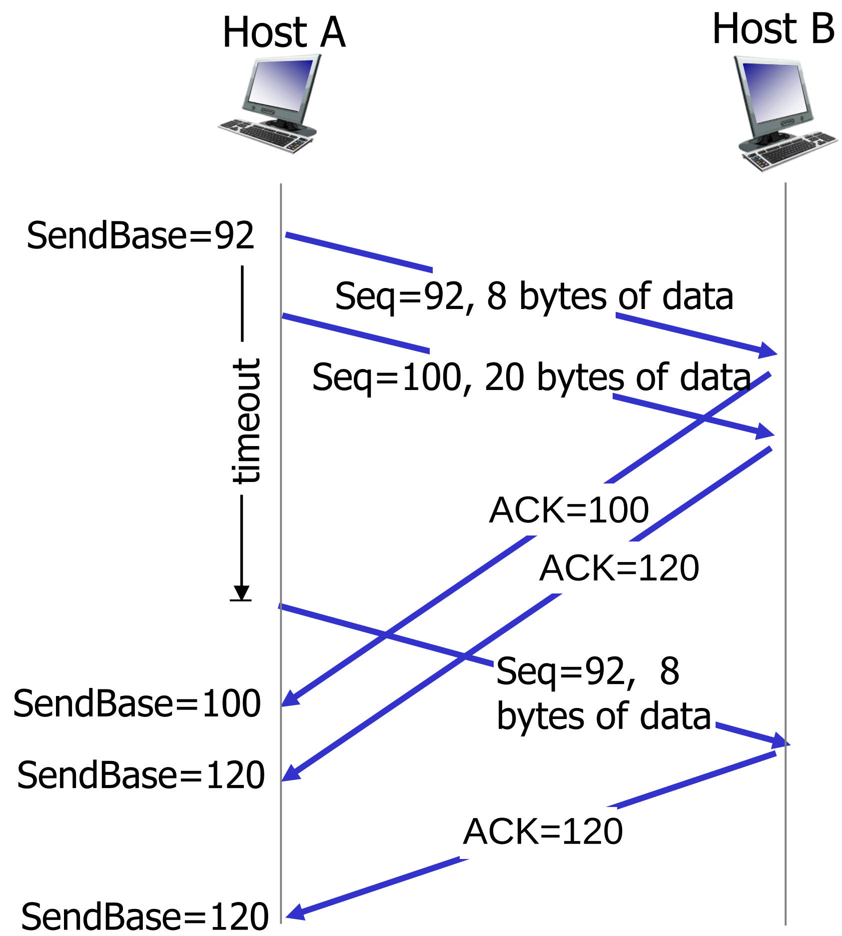
ACK received, with ACK field value y

```
if (y > SendBase) {
   SendBase = y
   /* SendBase–1: last cumulatively ACKed byte */
   if (there are currently not-yet-acked segments)
       start timer
     else stop timer
  }
```

TCP: retransmission scenarios

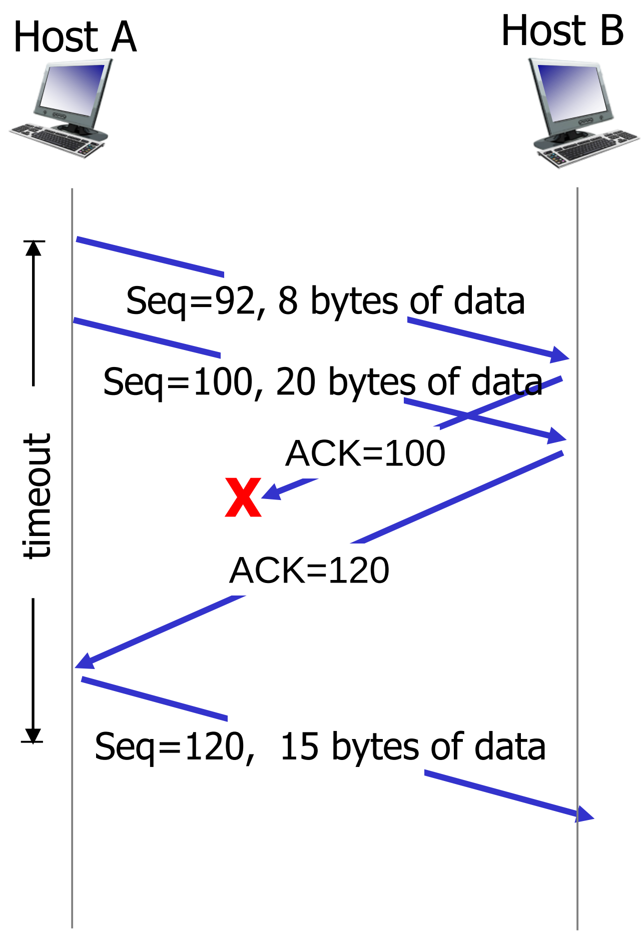


lost ACK scenario



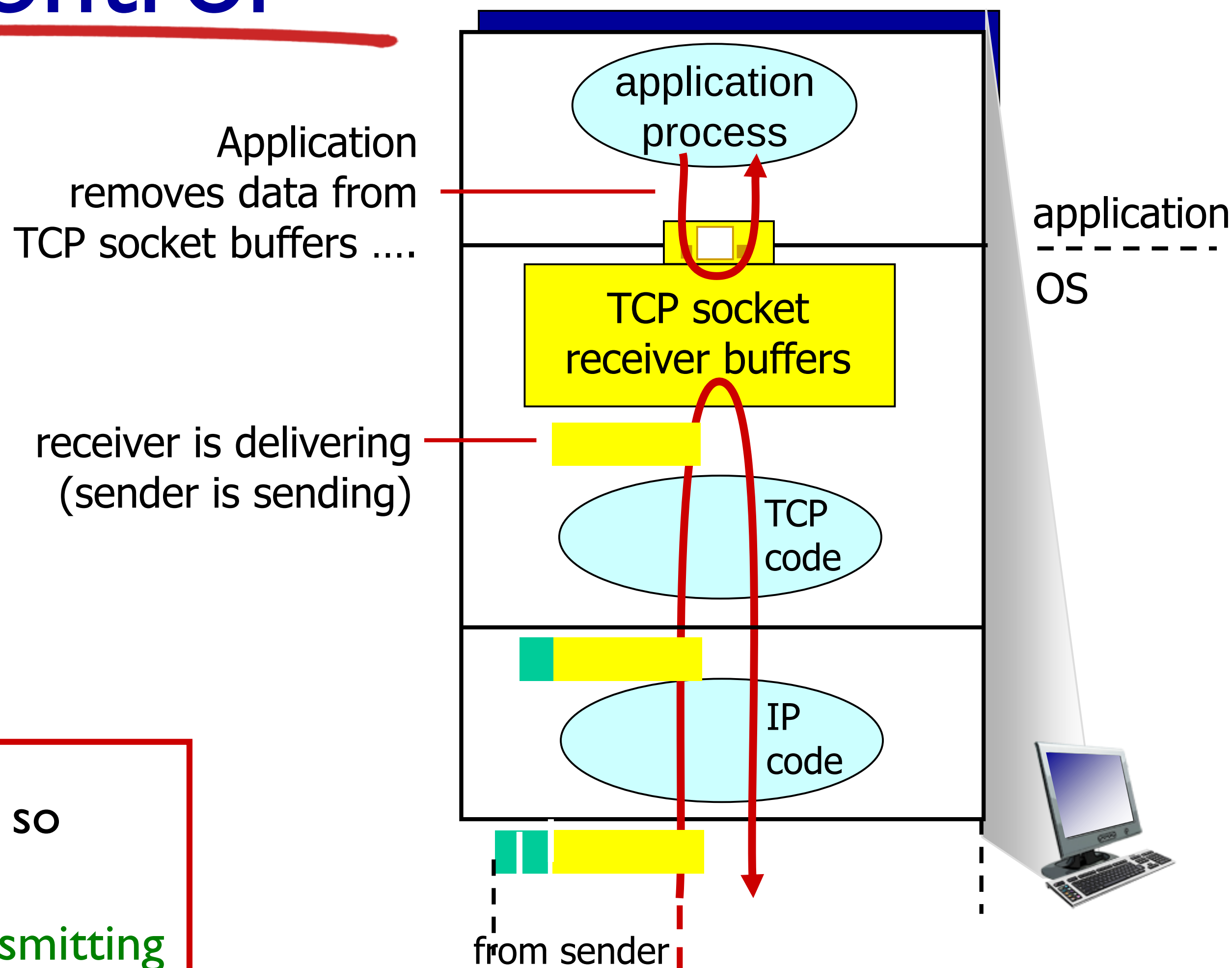
premature timeout

TCP: retransmission scenarios



cumulative ACK

TCP flow control



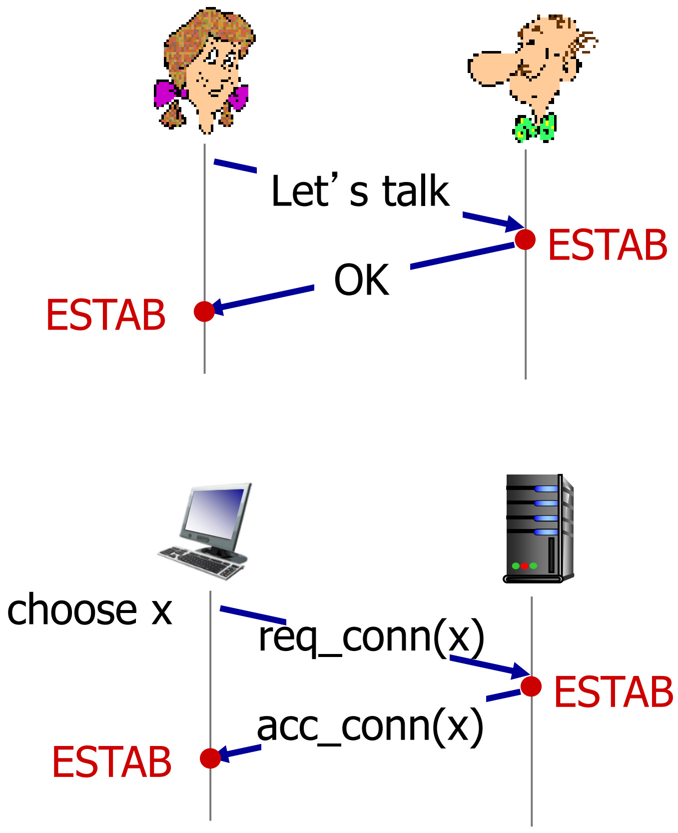
receiver protocol stack

flow control

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

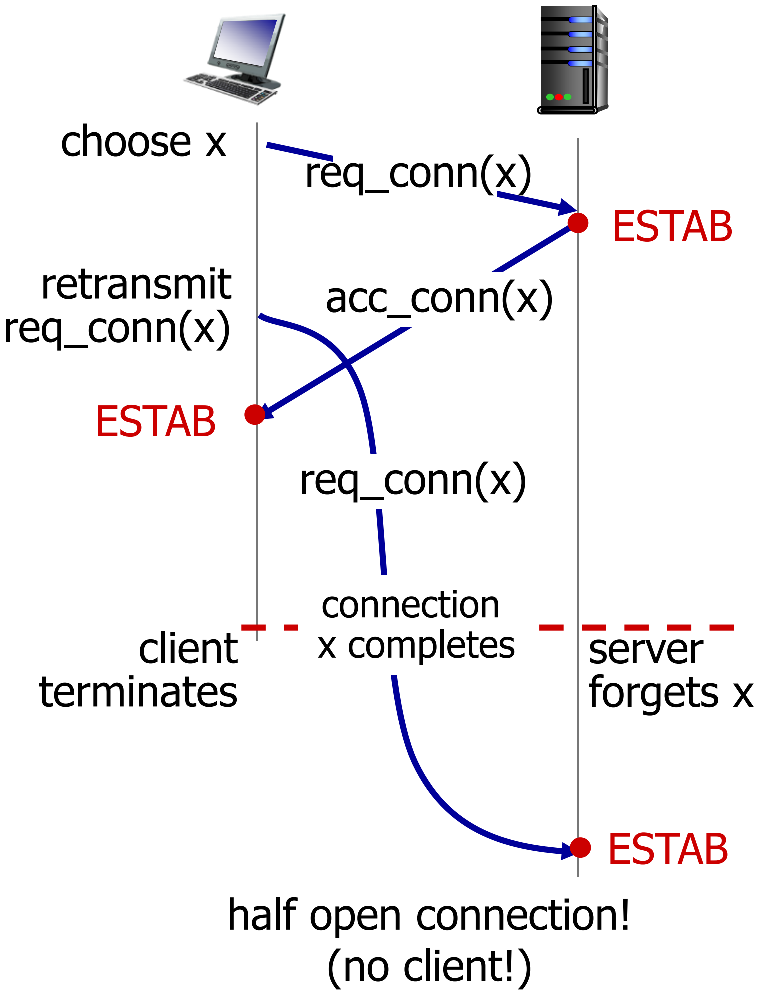
Agreeing to establish a connection

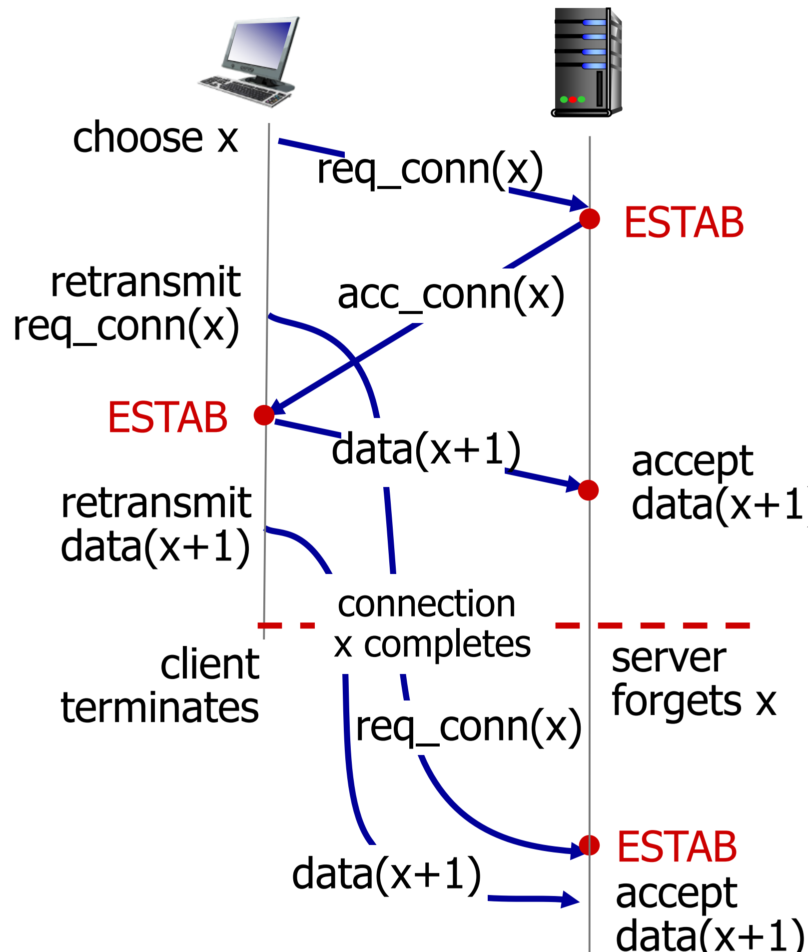
2-way handshake:



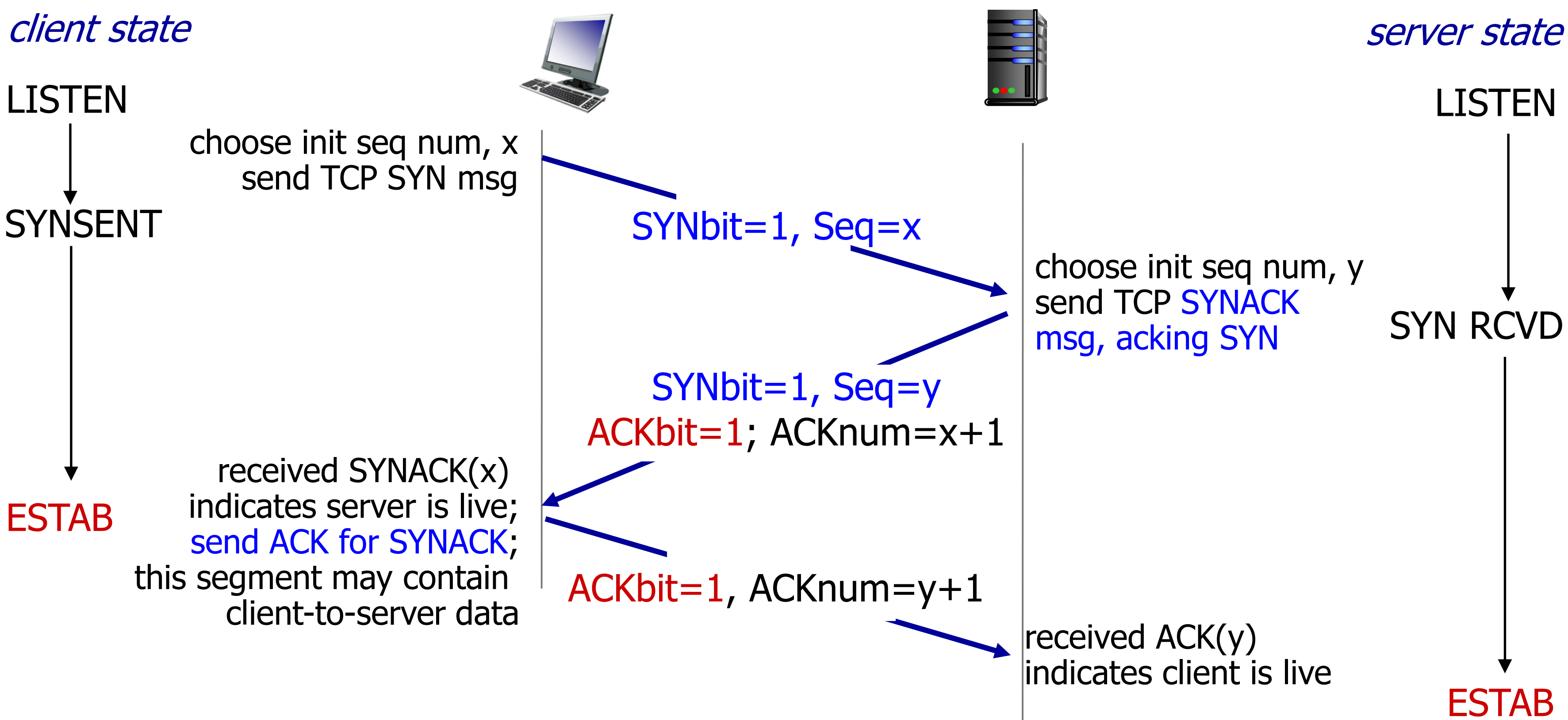
Agreeing to establish a connection

2-way handshake failure scenarios:

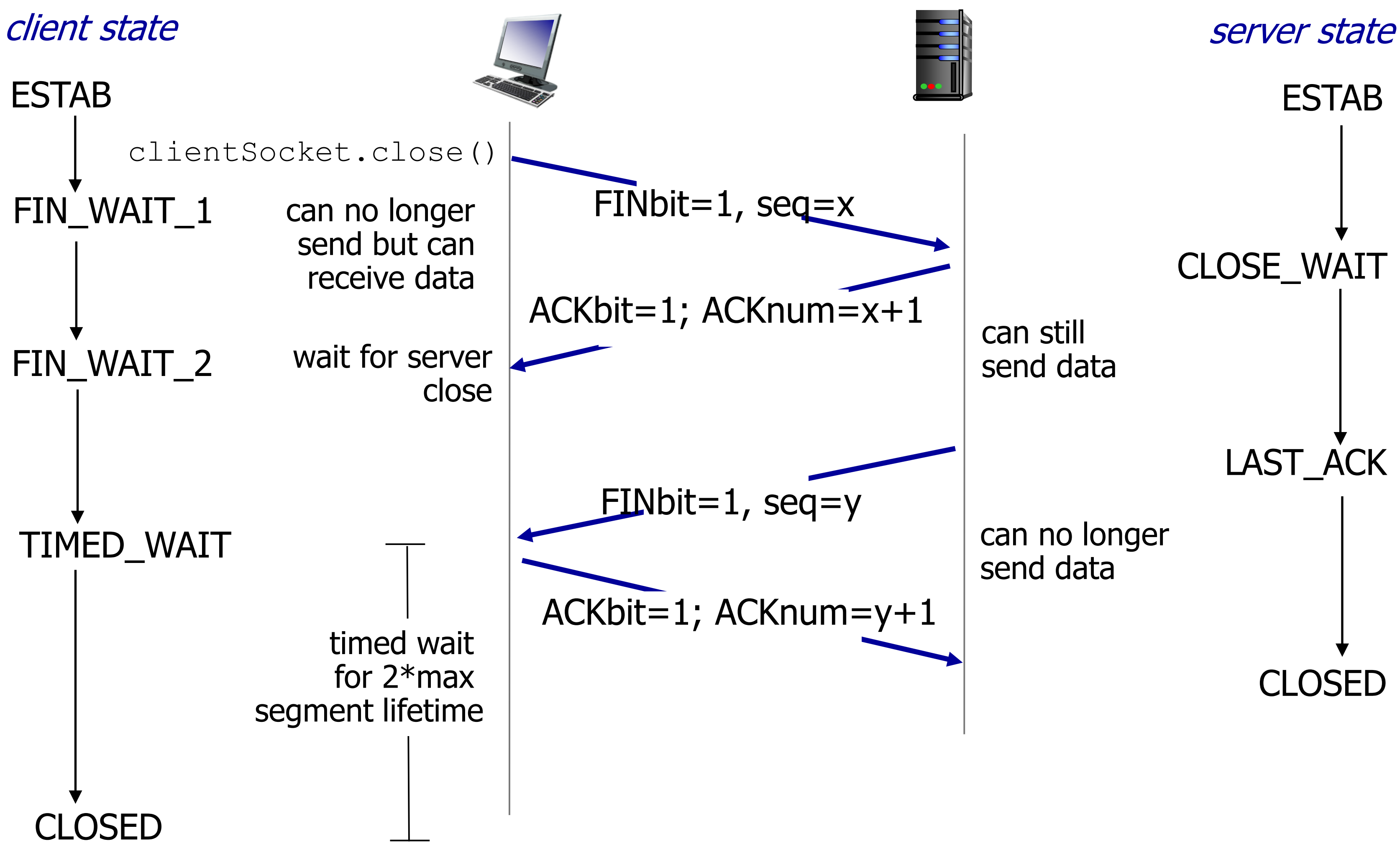




TCP 3-way handshake



TCP: closing a connection



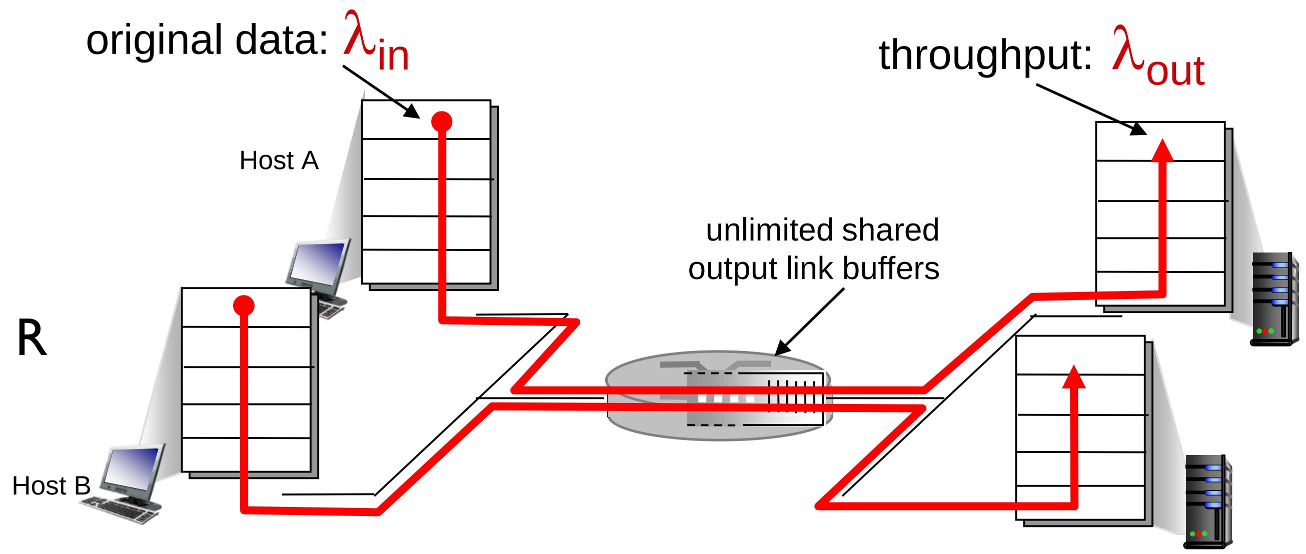
Principles of congestion control

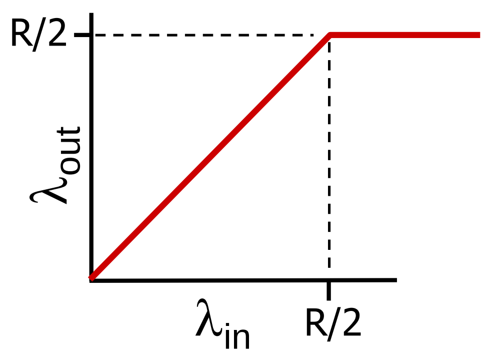
congestion:

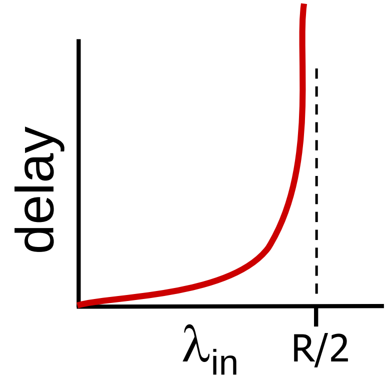
- informally: “too many sources sending too much data too fast for *network* to handle”
- different from flow control!
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)
- a top-10 problem!

Causes/costs of congestion: scenario 1

- two senders, two receivers
- one router, infinite buffers
- output link capacity: R
- no retransmission



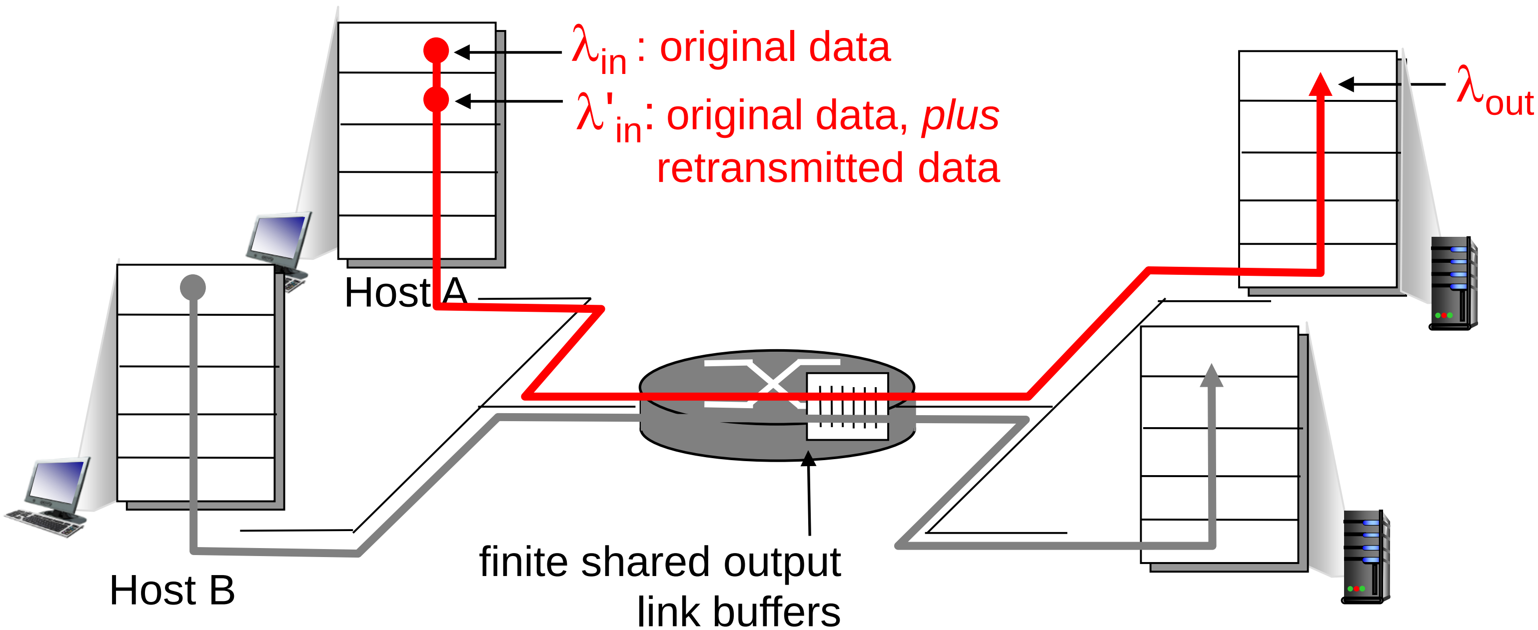




- maximum per-connection throughput: R/2
- large delays as arrival rate, λ_{in}, approaches capacity

Causes/costs of congestion: scenario 2

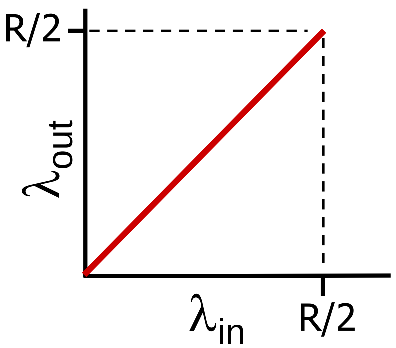
- one router, *finite* buffers
- sender retransmission of timed-out packet
  - application-layer input = application-layer output: λ_{in} = λ_{out}
  - transport-layer input includes *retransmissions* : $\lambda'_{in} \geq \lambda_{in}$

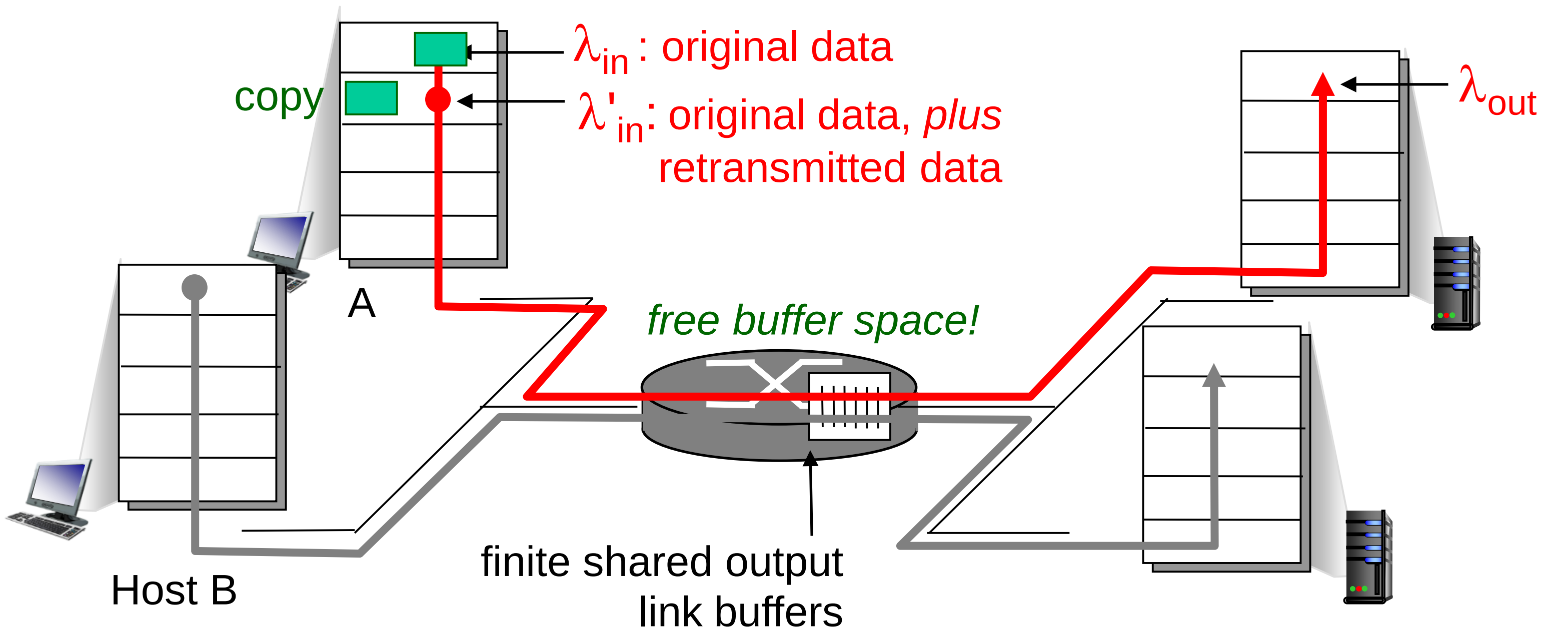


Causes/costs of congestion: scenario 2

idealization: perfect knowledge

- sender sends only when router buffers available



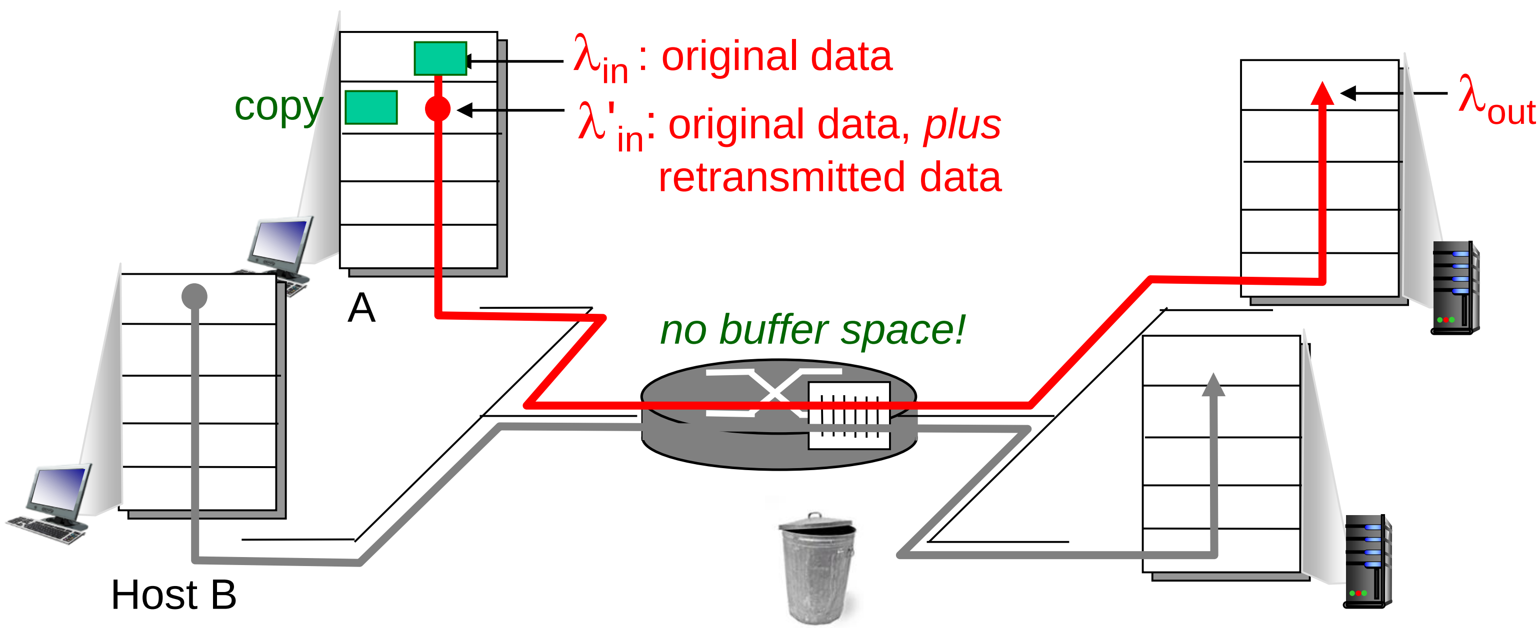


Causes/costs of congestion: scenario 2

Idealization: known loss

packets can be lost, dropped at router due to full buffers

- sender only resends if packet *known* to be lost

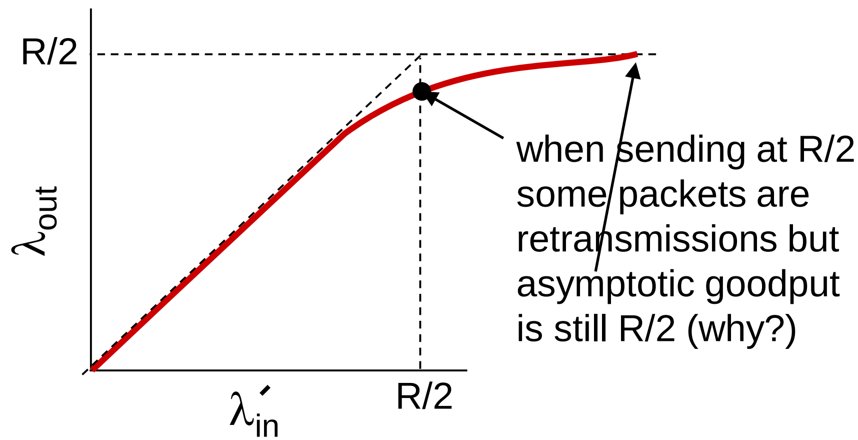


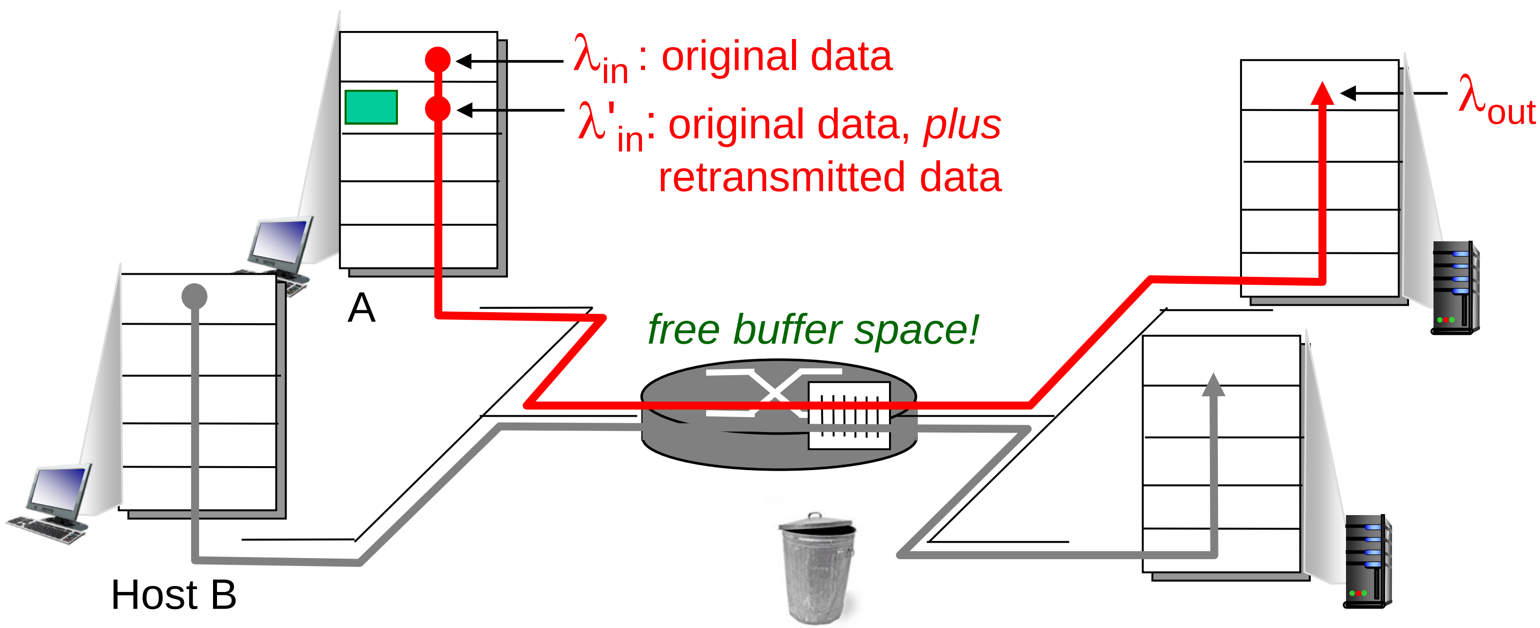
Causes/costs of congestion: scenario 2

Idealization: known loss

packets can be lost, dropped at router due to full buffers

- sender only resends if packet *known* to be lost

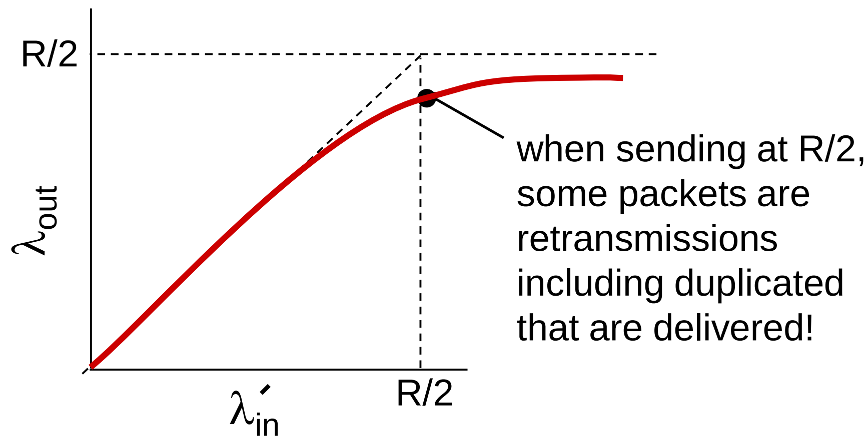


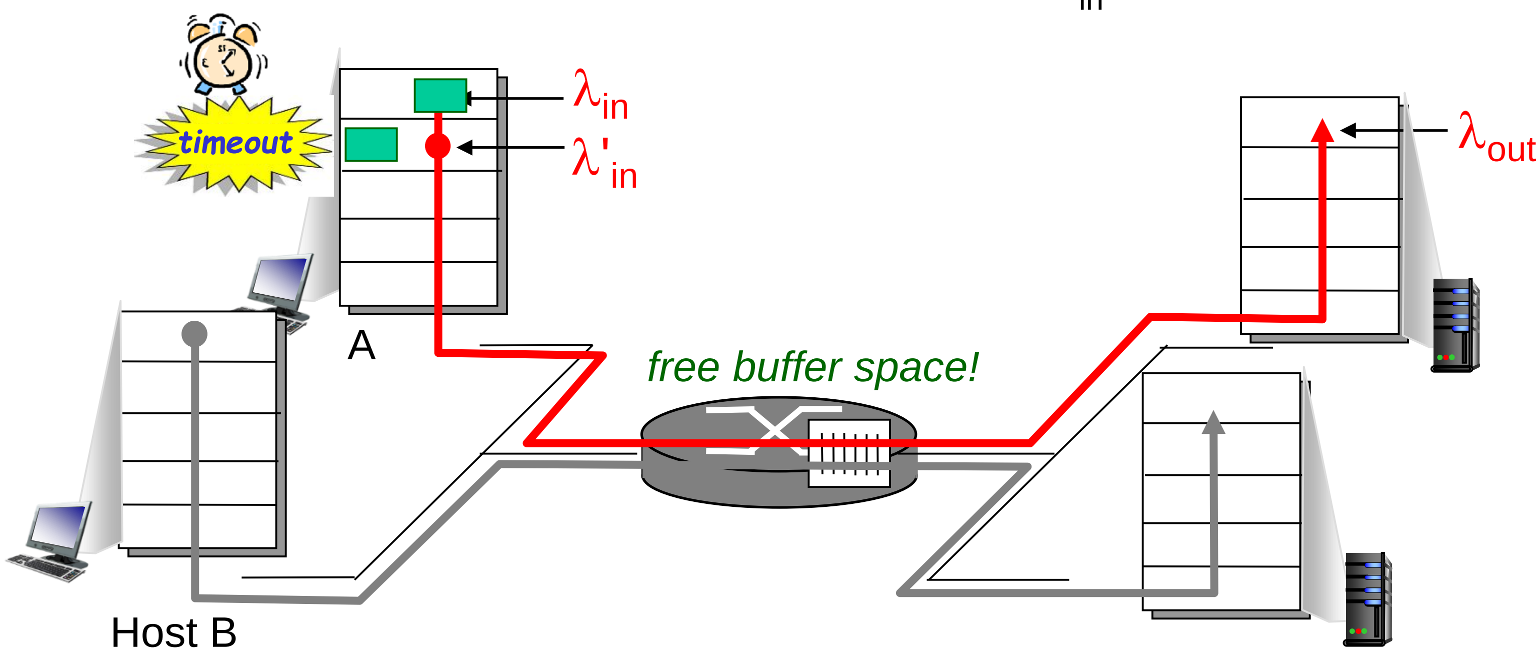


Causes/costs of congestion: scenario 2

Realistic: duplicates

- packets can be lost, dropped at router due to full buffers
- sender times out prematurely, sending *two* copies, both of which are delivered

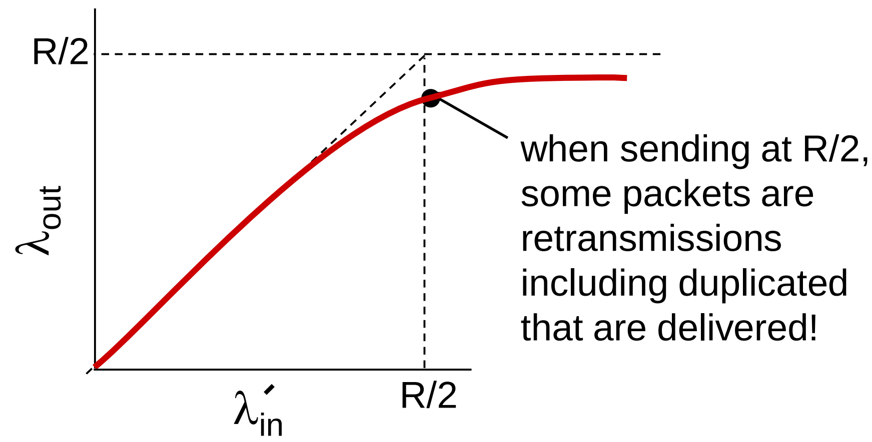




Causes/costs of congestion: scenario 2

Realistic: duplicates

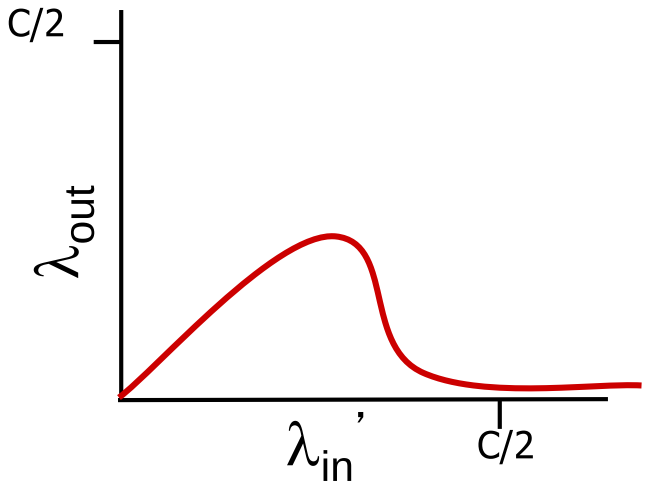
- packets can be lost, dropped at router due to full buffers
- sender times out prematurely, sending *two* copies, both of which are delivered

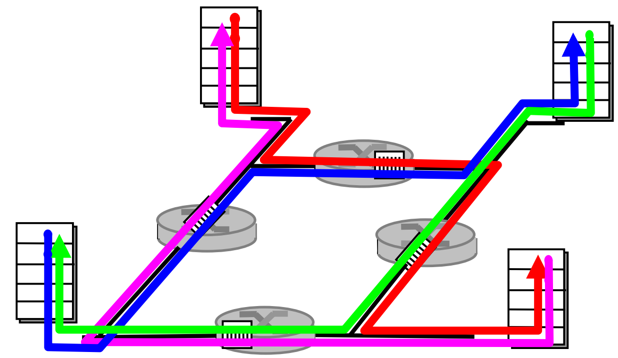


"costs" of congestion:

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

Causes/costs of congestion: scenario 3





another "cost" of congestion:

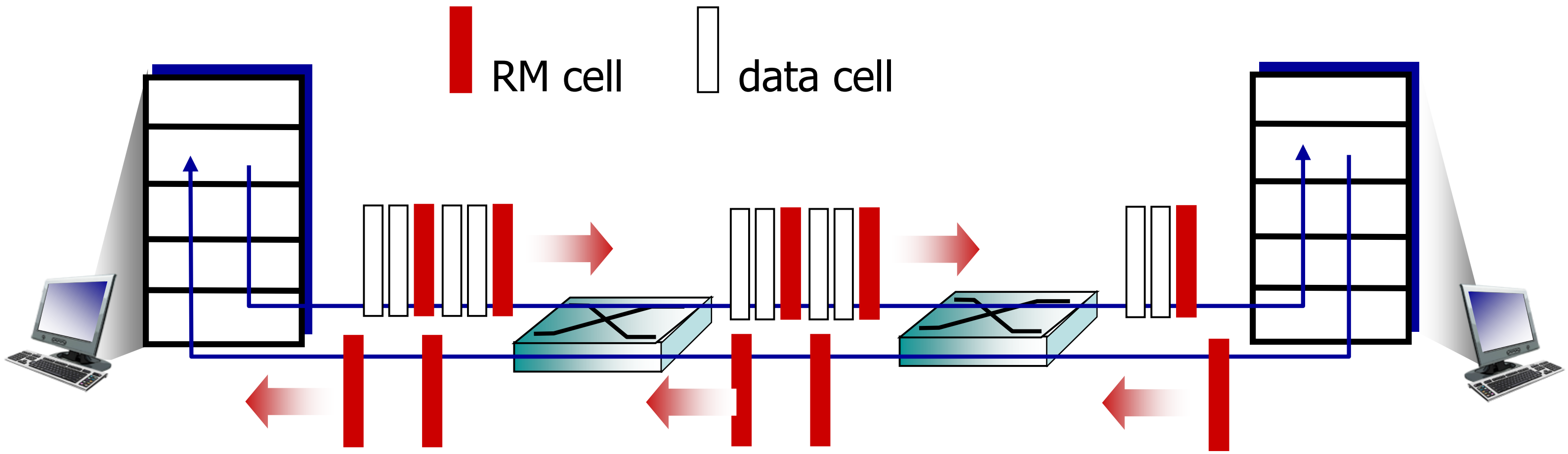
- when packet dropped, any "upstream transmission capacity used for that packet was wasted!

Case study: ATM ABR congestion control

ABR: available bit rate:
- "elastic service"
- if sender's path "underloaded":
  - sender should use available bandwidth
- if sender's path congested:
  - sender throttled to minimum guaranteed rate

RM (resource management) cells:
- sent by sender, interspersed with data cells (1 RM-32 data)
- bits in RM cell set by switches ("*network-assisted*")
  - *NI bit:* no increase in rate (mild congestion)
  - *CI bit:* congestion indication
- RM cells returned to sender by receiver, with bits intact

Case study: ATM ABR congestion control

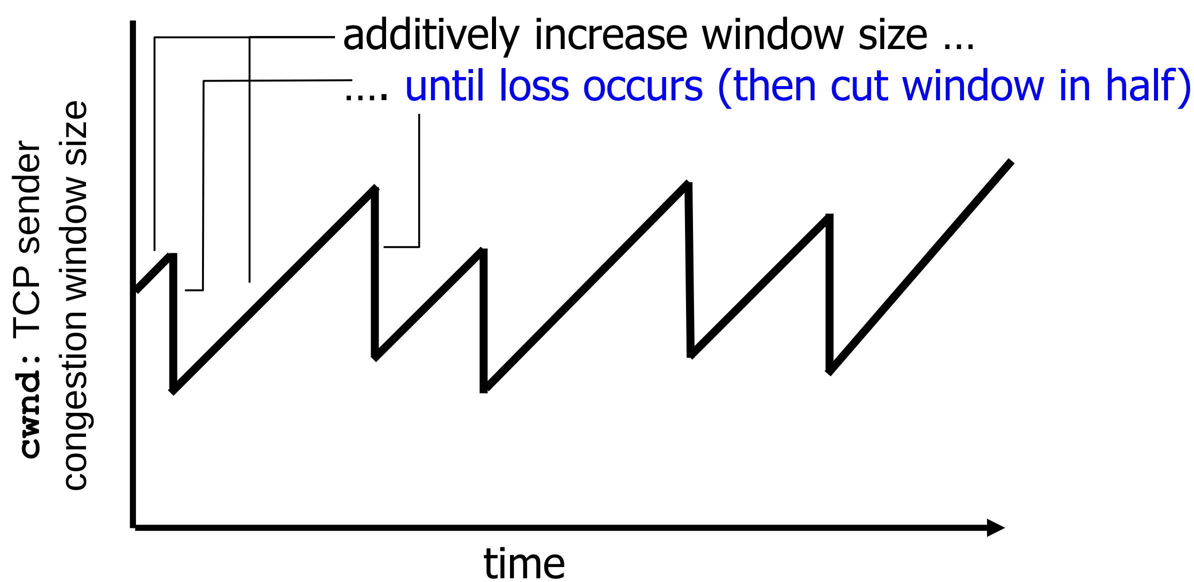


- two-byte ER (explicit rate) field in RM cell
  - congested switch may lower ER value in cell
  - senders' send rate thus max supportable rate on path
- EFCI (Explicit Forward CI) bit in data cells: set to 1 in congested switch
  - if data cell preceding RM cell has EFCI set, receiver sets CI bit in returned RM cell

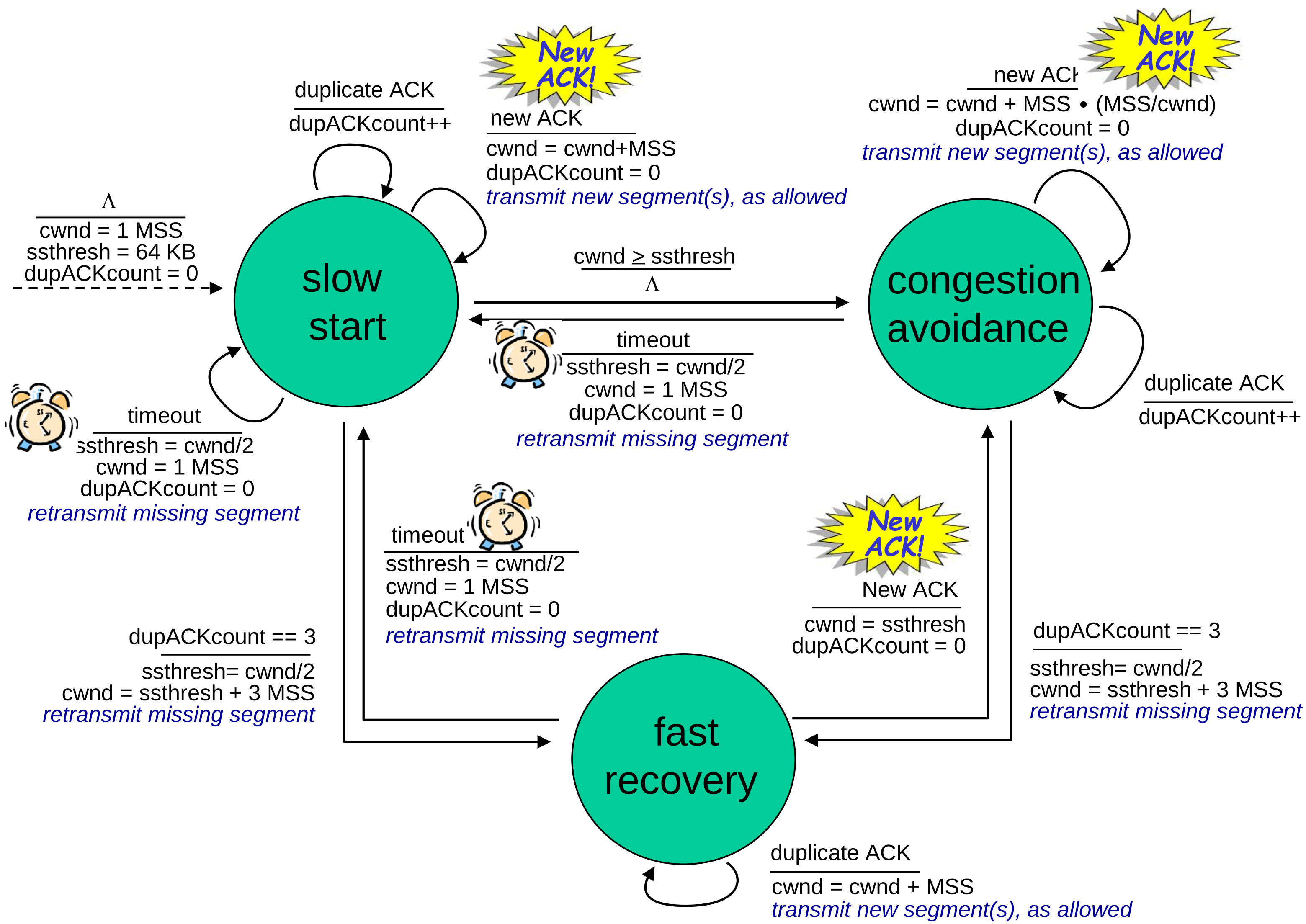
TCP congestion control: additive increase multiplicative decrease

- *approach:* sender increases transmission rate (window size), probing for usable bandwidth, until loss occurs
  - *additive increase:* increase **`cwnd`** by 1 MSS every RTT until loss detected
  - *multiplicative decrease*: cut **`cwnd`** in half after loss

AIMD saw tooth behavior: probing for bandwidth



Summary: TCP Congestion Control



TCP Fairness

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