UMBC

AN HONORS UNIVERSITY IN MARYLAND

# IS 450/IS 650–
# Data Communications and Networks

# Transport Layer

Nirmalya Roy
Department of Information Systems
University of Maryland Baltimore County

# Chapter 3 Outline

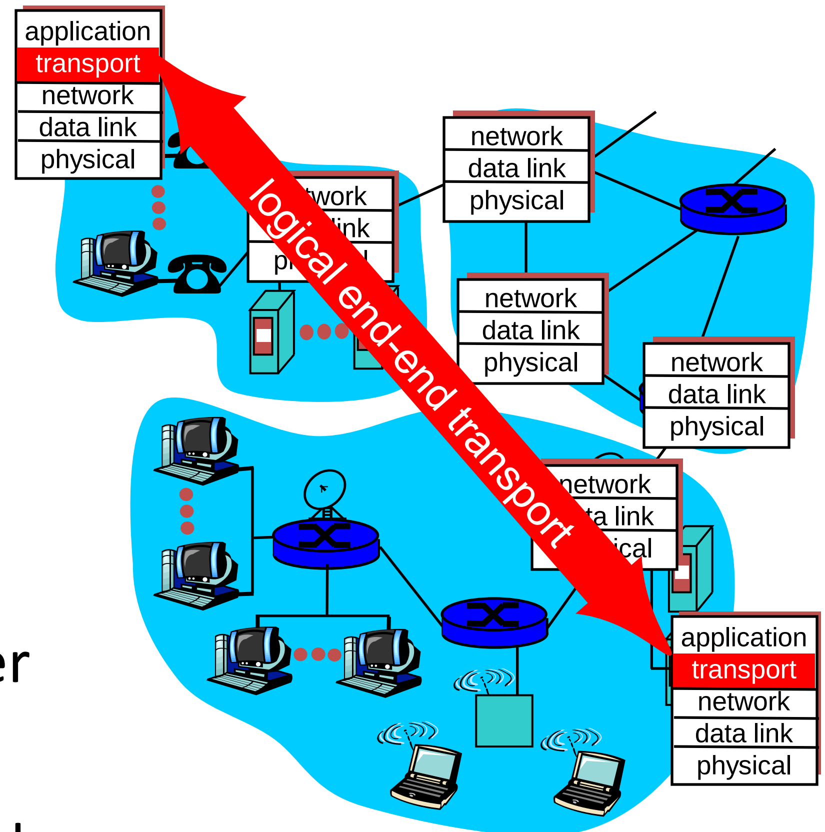
- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

# Need for Transport Layer

- Network Layer offers connections
  - IP (Internet Protocol) service model
    - Best-effort delivery service, unreliable service
  - Connections not reliable
    - Losses, delays due to out-of-order, queue overflow, …
- Transport Layer Goals
  - End to end reliability
  - In Order delivery
  - Performance
    - Congestion control
    - Flow control

# Transport services and protocols

- *logical communication* between processes
- transport protocols run in end systems
  - breaks app messages into segments
  - 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

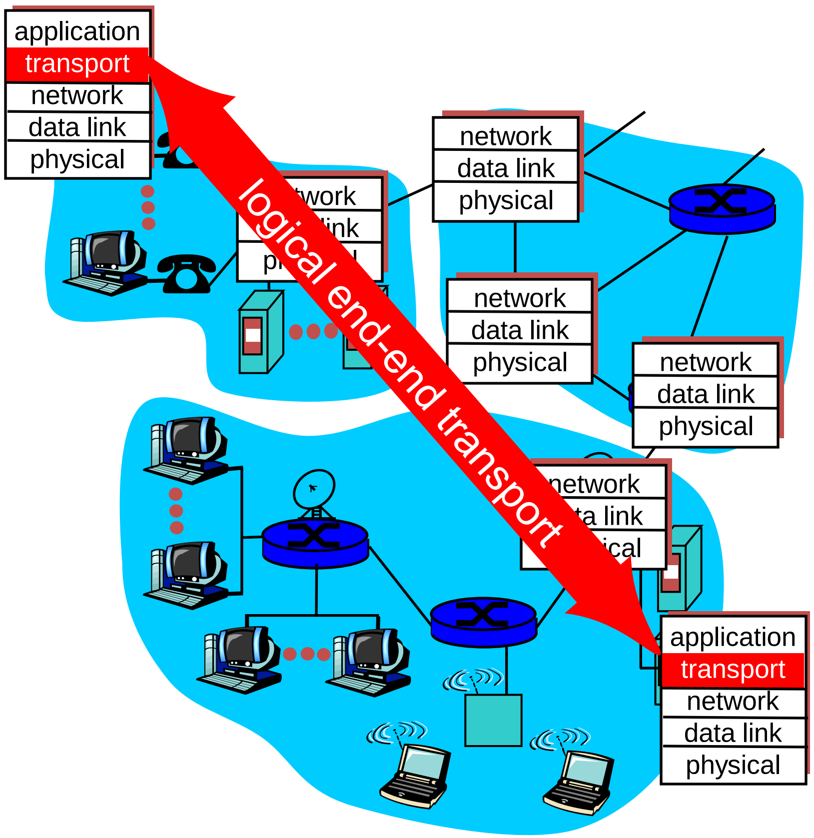
***household analogy:***

12 kids in Ann's house sending letters to 12 kids in Bill's house:

- hosts = houses
- processes = kids
- app messages = letters in envelopes
- transport protocol = Ann and Bill who demux to in-house siblings
- network-layer protocol = postal service

# Transport-layer protocols (TCP, UDP):

- 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
- Host-to-host delivery to process-to-process delivery
  - transport-layer multiplexing demultiplexing



# Agenda

- Transport Layer
- Multiplexing / Demultiplexing

- Reliable Transport
  - Stop-and-wait
  - Pipelined
    - Go back N
    - Selective Request

- TCP
  - Congestion Control
  - Flow Control

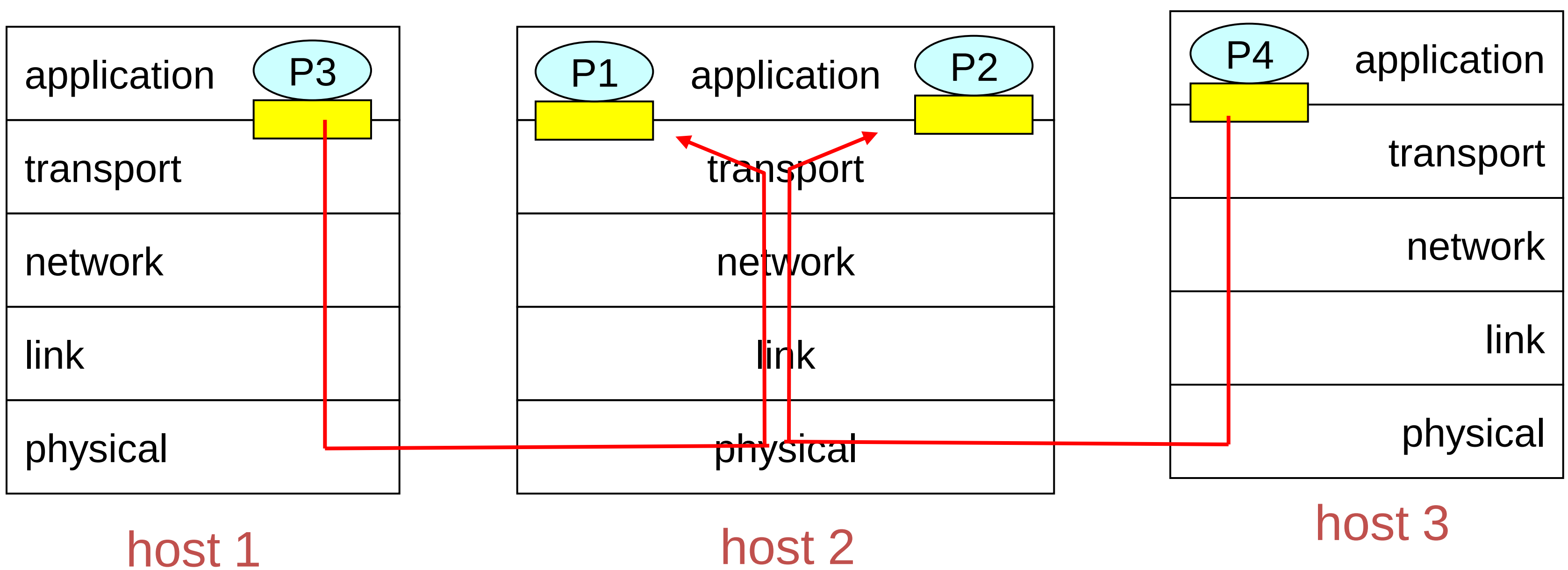
# Multiplexing/demultiplexing

**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)

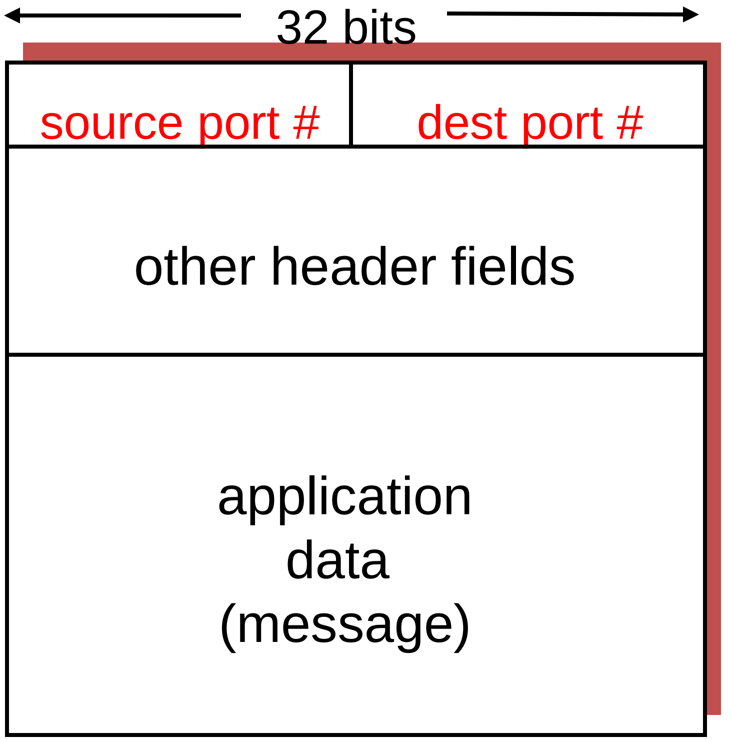

One HTTP process, one FTP process, one Telnet process
More than one socket, each socket has unique identifier

# 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
- host uses IP addresses & port numbers to direct segment to appropriate socket

Analogous to car rentals at airports

Shuttles MUX passengers and take them
To rental office -- DeMUX to diff cars




TCP/UDP segment format

# Connectionless demultiplexing

- Create sockets with port numbers:

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

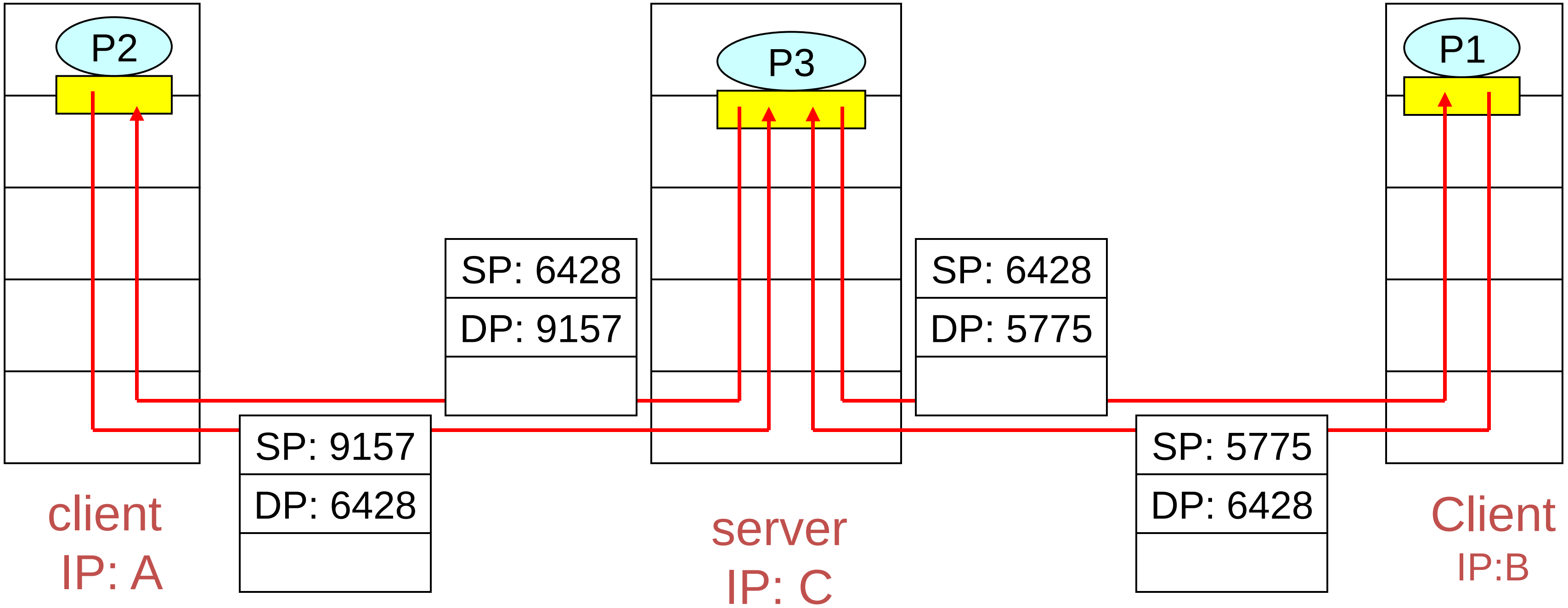
- UDP socket fully identified by two-tuple:

  (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);
```

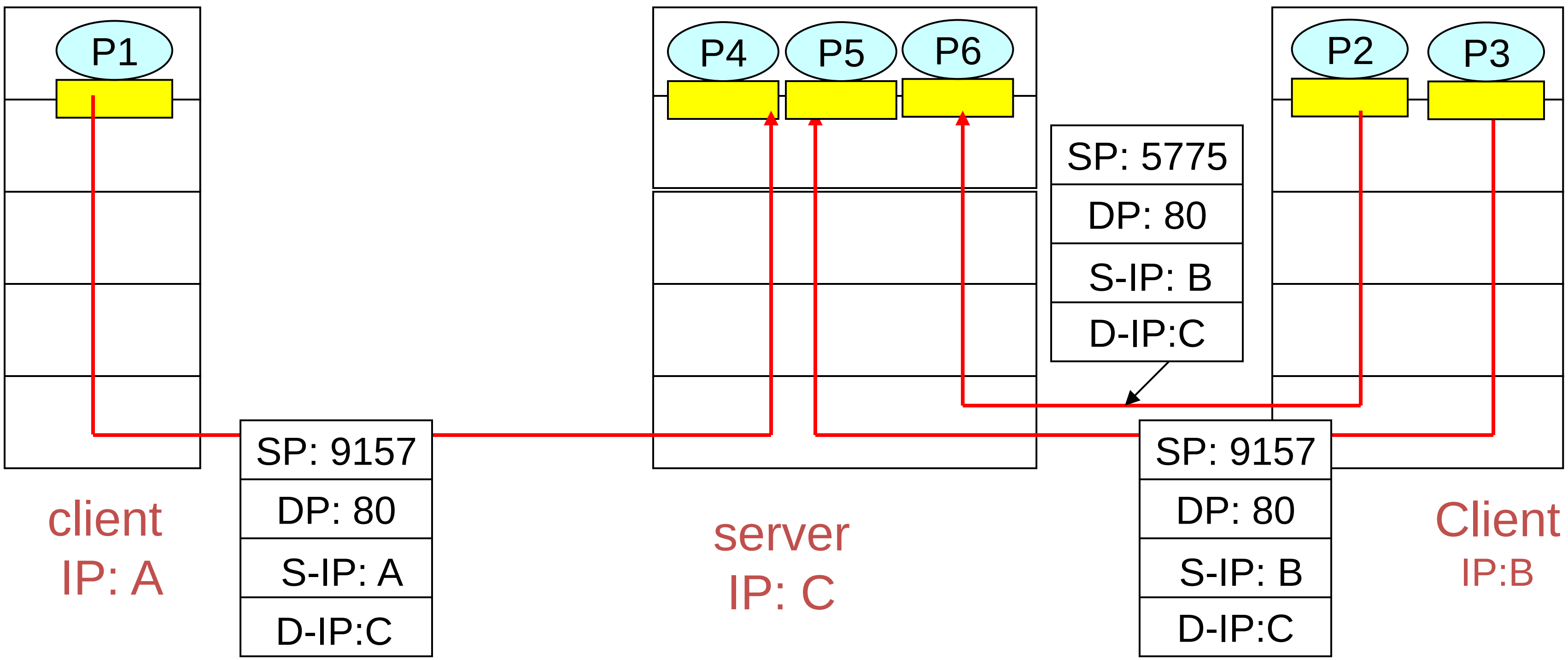



SP provides “return address”

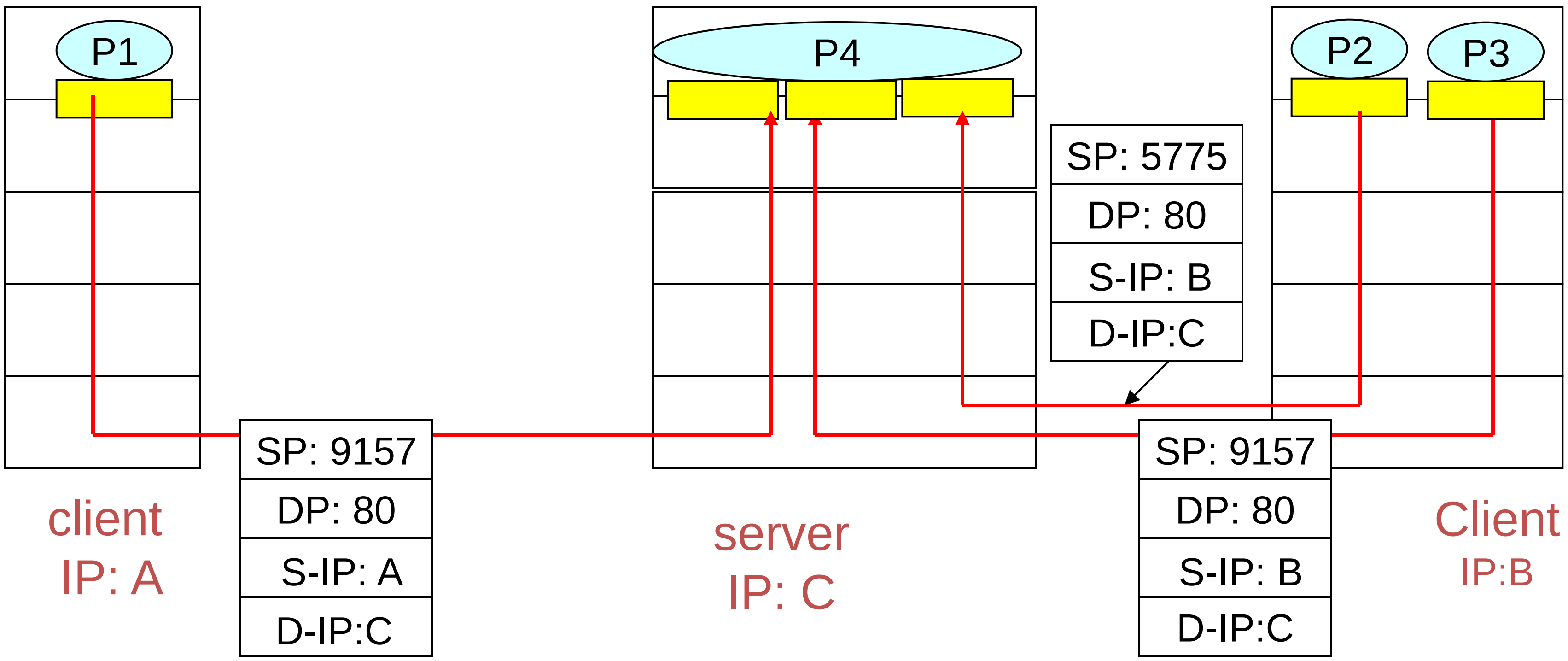
# Connection-oriented demux

- 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)



# Connection-oriented demux: Threaded Web Server



# Chapter 3 Outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

# UDP: User Datagram Protocol [RFC 768]

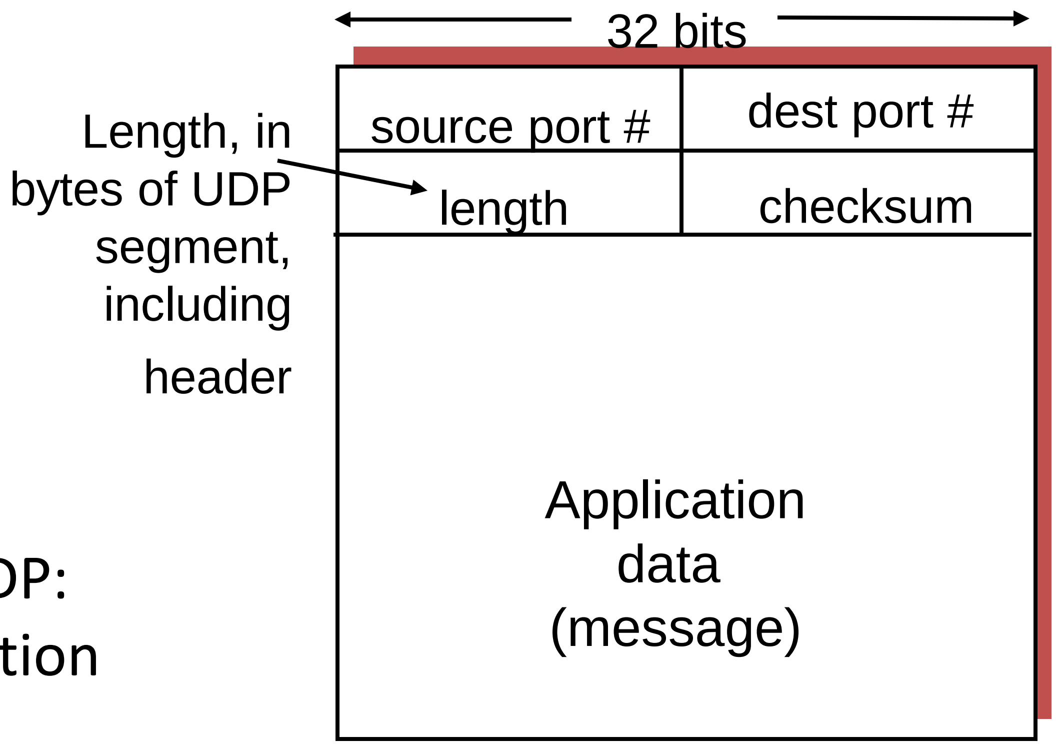
- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out of order to app
- *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

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: segment header

- often used for streaming multimedia apps
  - loss tolerant
  - rate sensitive
- other UDP uses
  - DNS
  - SNMP
- reliable transfer over UDP: add reliability at application layer
  - application-specific error recovery!




UDP segment format

# UDP checksum

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

**sender:**

- treat segment contents, including header fields, as sequence of 16-bit integers
- checksum: addition (one's complement sum) of segment contents
- sender puts checksum value into UDP checksum field

**receiver:**

- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected. *But maybe errors nonetheless?* More later ….

# Internet checksum: example

example: add two 16-bit integers

| | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| | | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| wraparound | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 |
| sum | | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 |
| checksum | | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 |

*Note:* when adding numbers, a carryout from the most significant bit needs to be added to the result

# Chapter 3 outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP

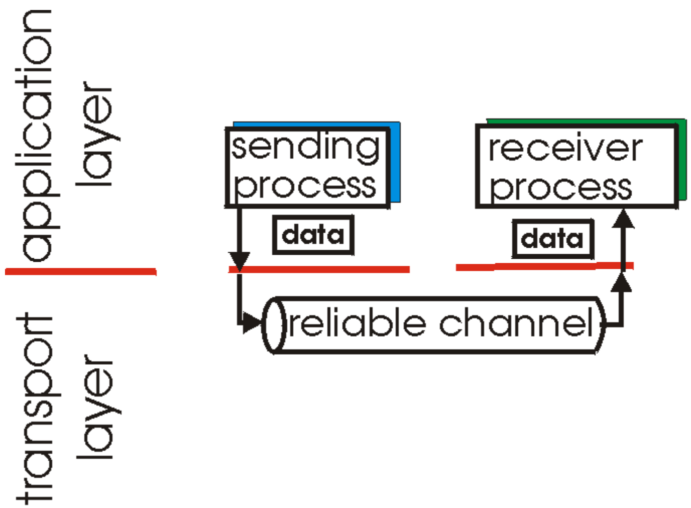
- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control

3.7 TCP congestion control

# Principles of reliable data transfer

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

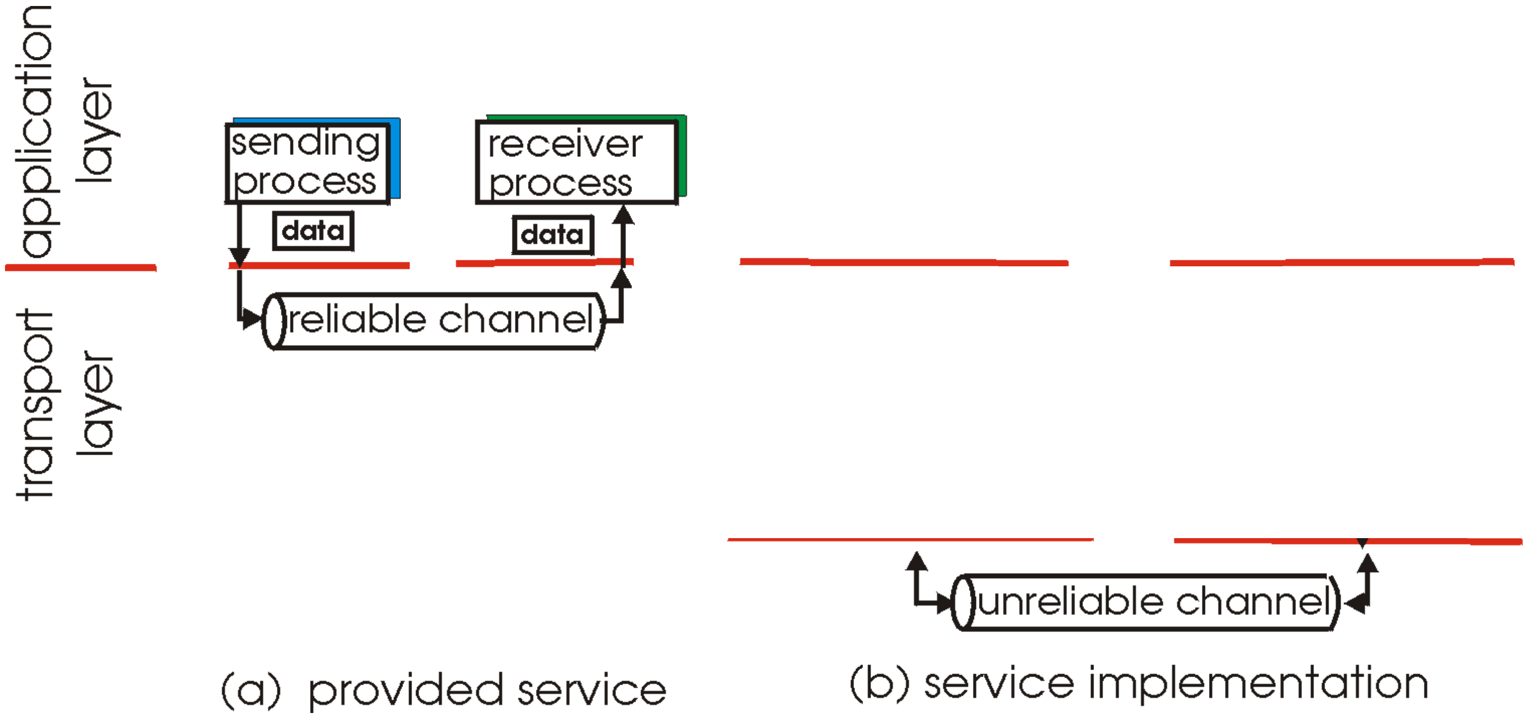



(a) provided service

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

# Principles of reliable data transfer

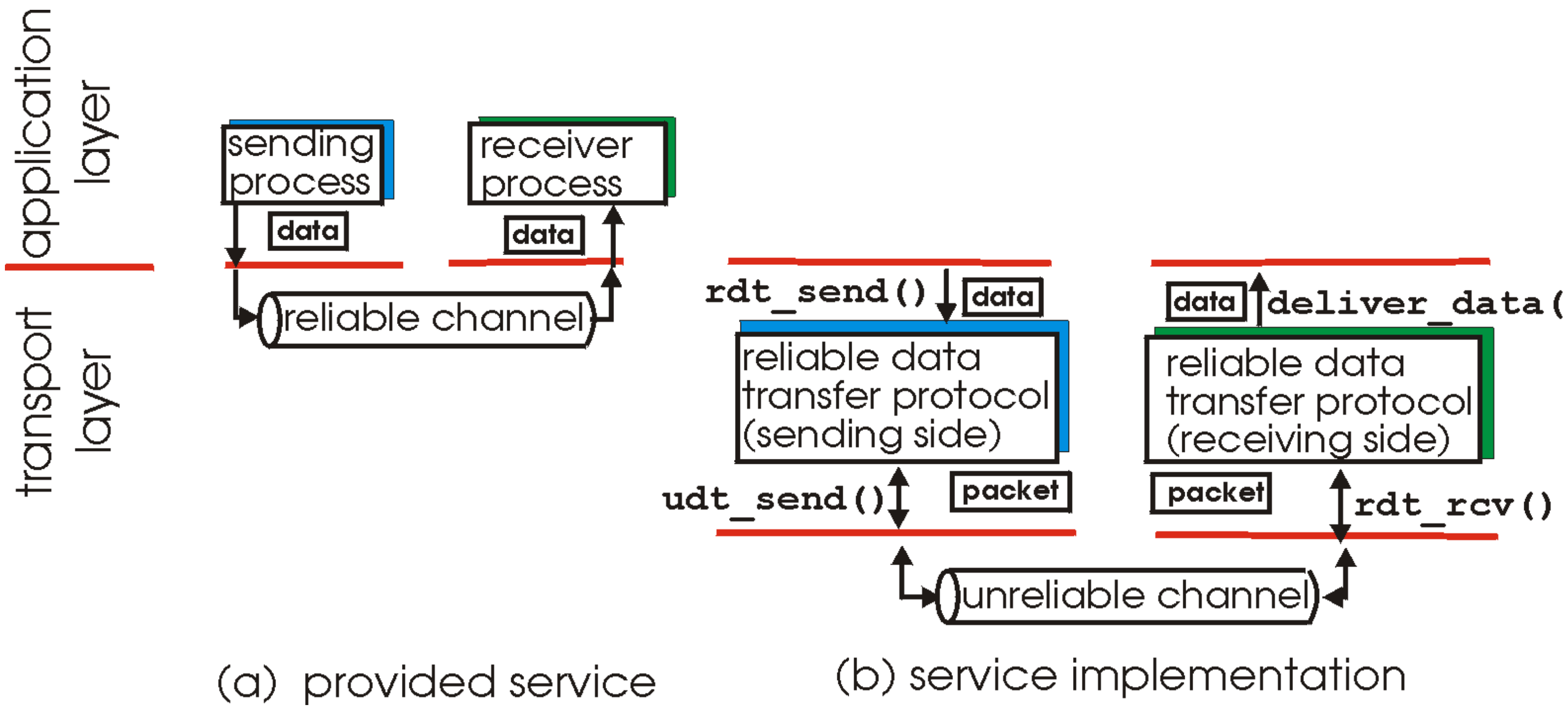
- important in application, transport, link layers
  - top-10 list of important networking topics!




- 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!

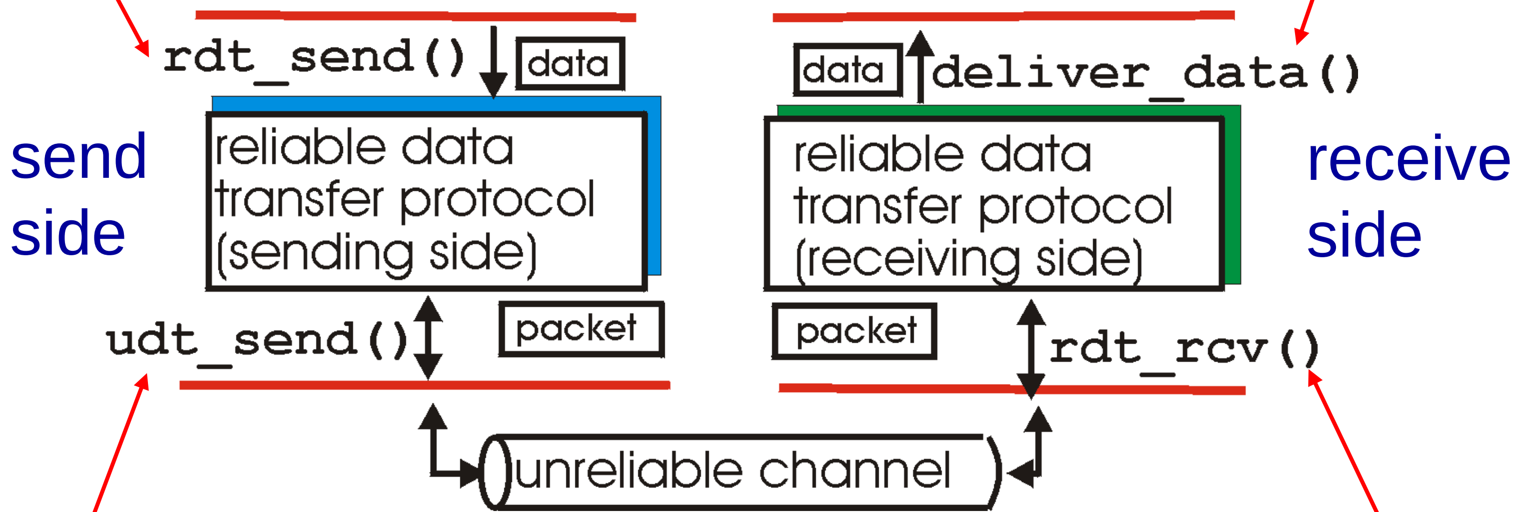



- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

# 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

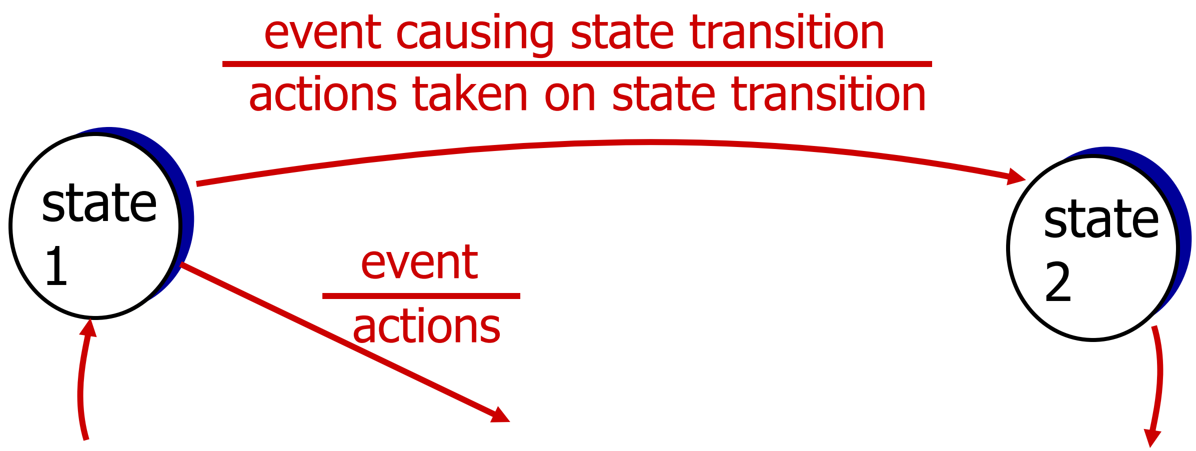
**rdt_rcv():** called when packet arrives on rcv-side of channel

# Reliable data transfer: getting started

We'll:

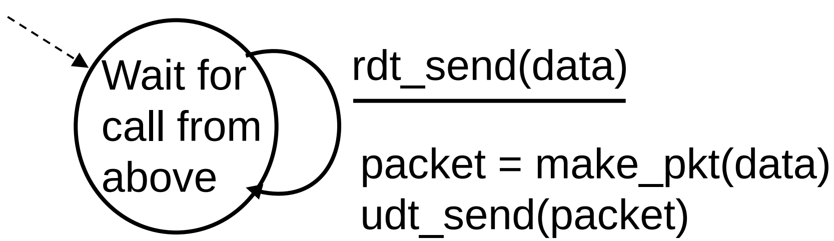
- incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- consider only unidirectional data transfer
  - but control info will flow on both directions!
- use finite state machines (FSM) to specify sender, receiver

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

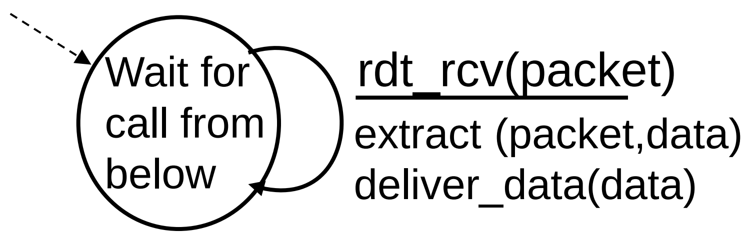


# rdt1.0: reliable transfer over a reliable channel

- 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

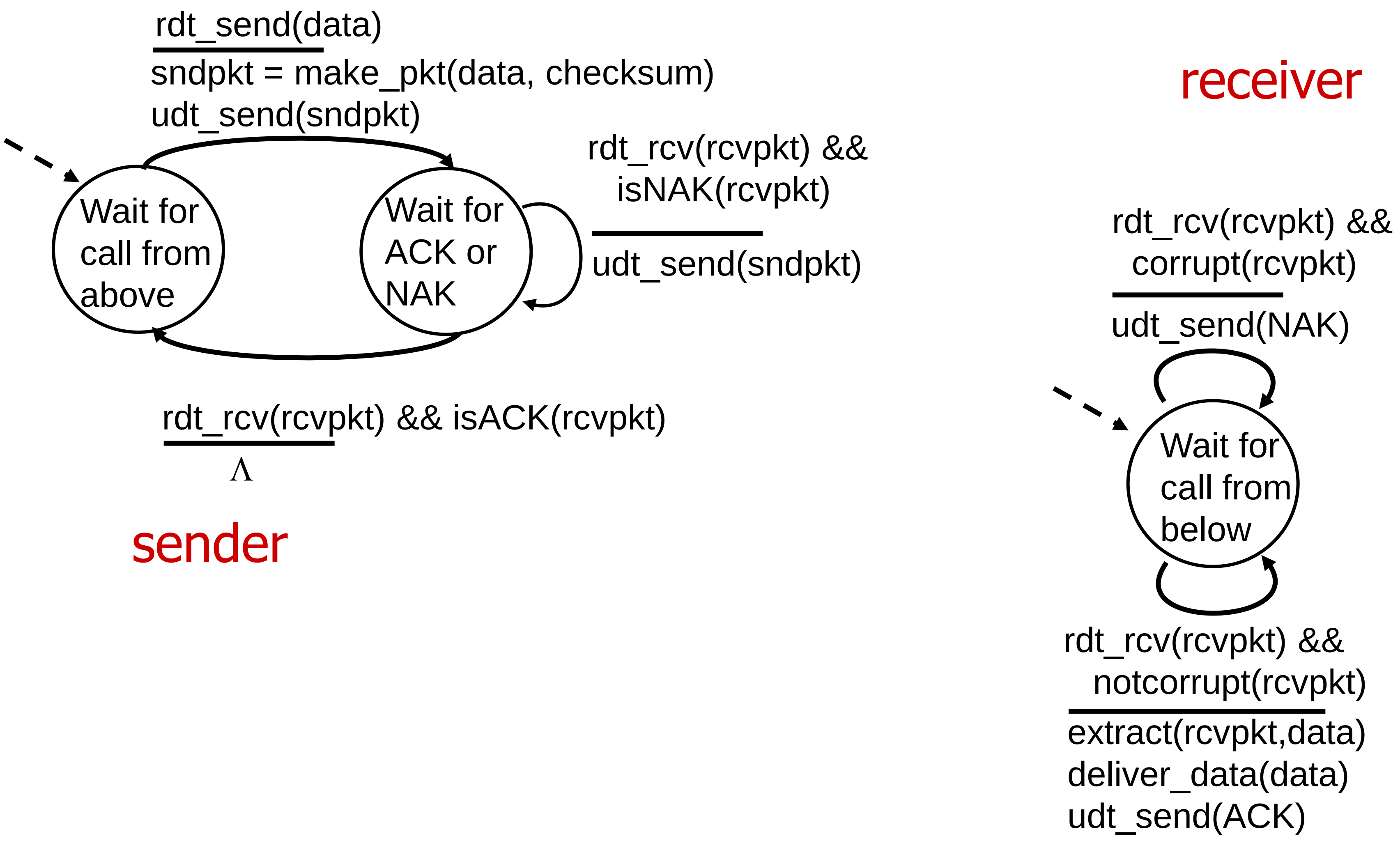
- underlying channel may flip bits in packet
  - checksum to detect bit errors
- *the* question: how to recover from errors:

*How do humans recover from "errors" during conversation?*

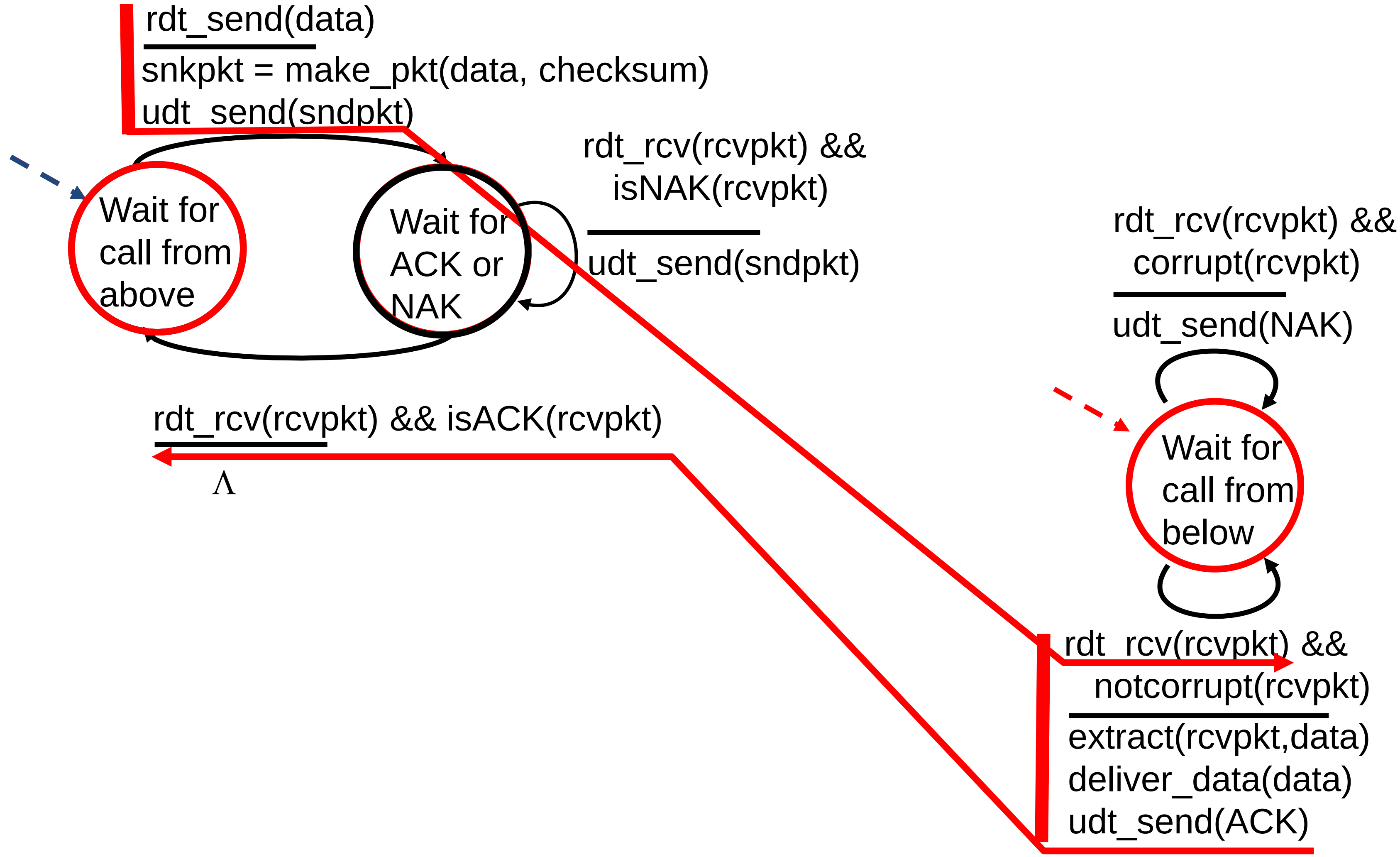
# rdt2.0: channel with bit errors

- 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

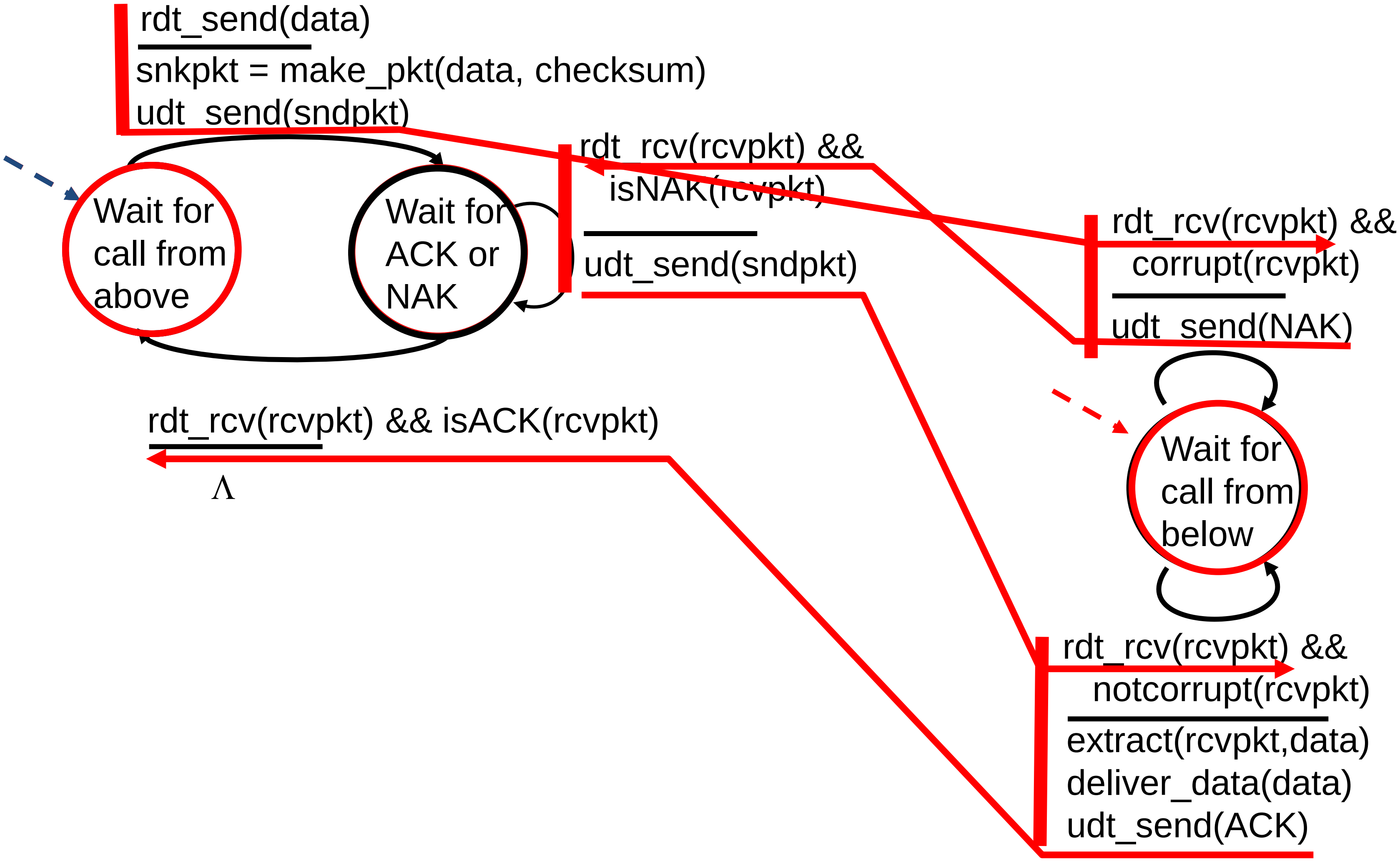
# rdt2.0: FSM specification



# 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

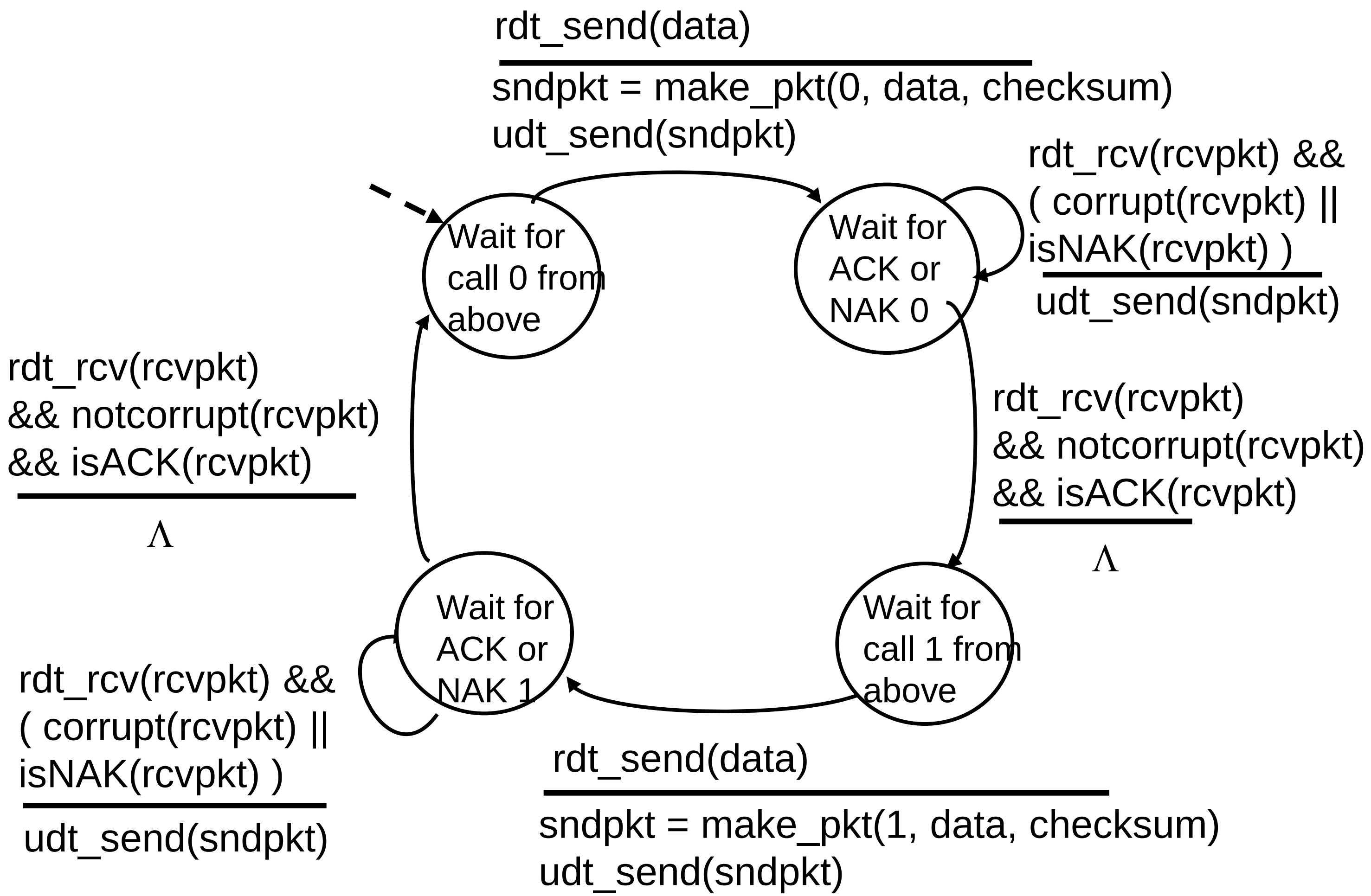
handling duplicates:

- sender retransmits current pkt if ACK/NAK corrupted
- sender adds *sequence number* to each pkt
- receiver discards (doesnt 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

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq0(rcvpkt)

---

extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && (corrupt(rcvpkt)

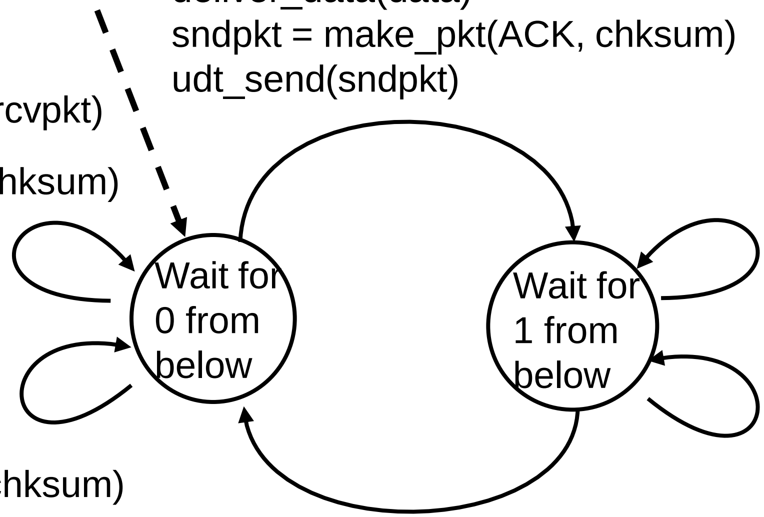
---

sndpkt = make_pkt(NAK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
not corrupt(rcvpkt) &&
has_seq1(rcvpkt)

---

sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)




rdt_rcv(rcvpkt) && (corrupt(rcvpkt)

---

sndpkt = make_pkt(NAK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
not corrupt(rcvpkt) &&
has_seq0(rcvpkt)

---

sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq1(rcvpkt)

---

extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

# 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

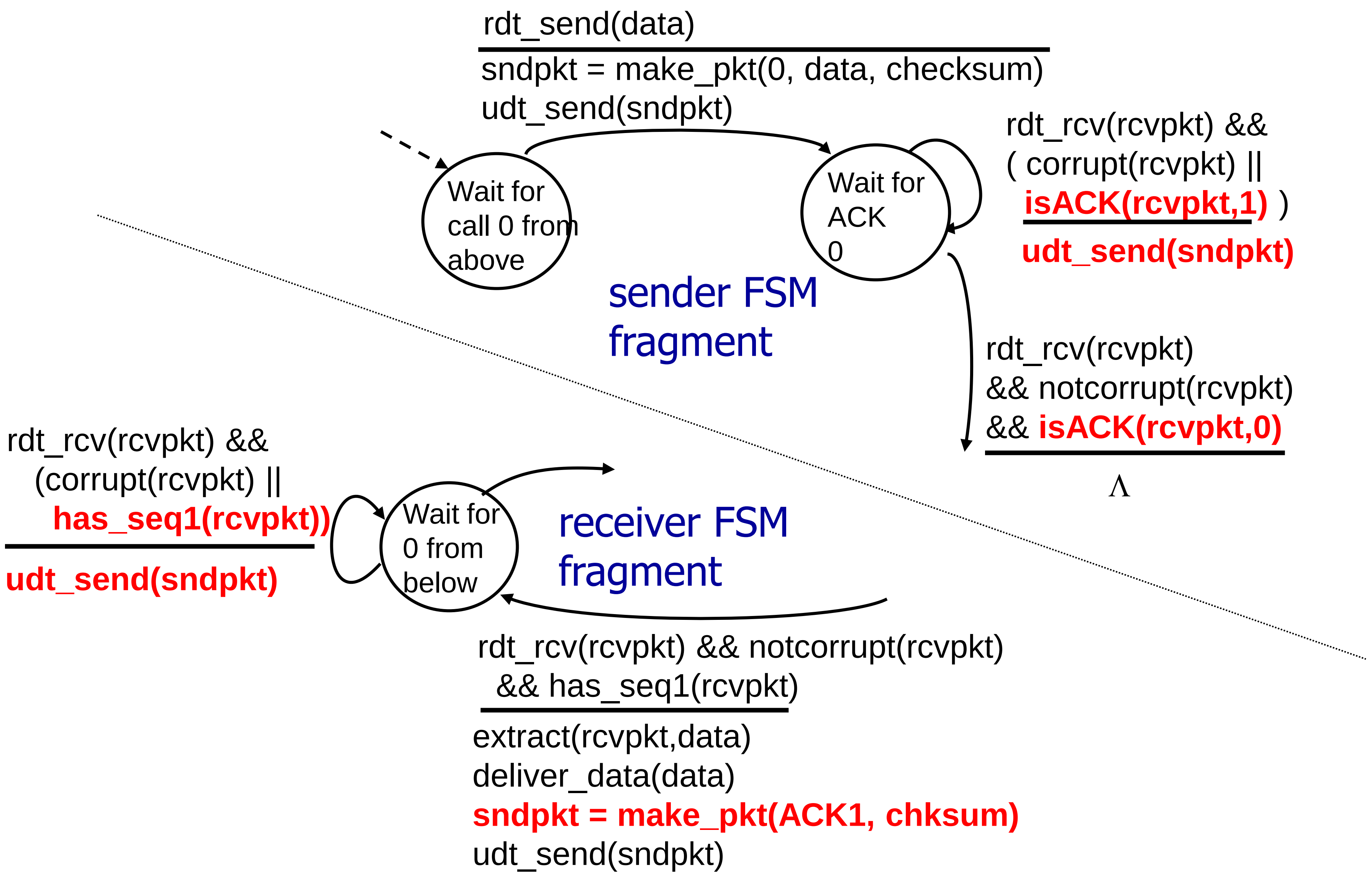
receiver:

- 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

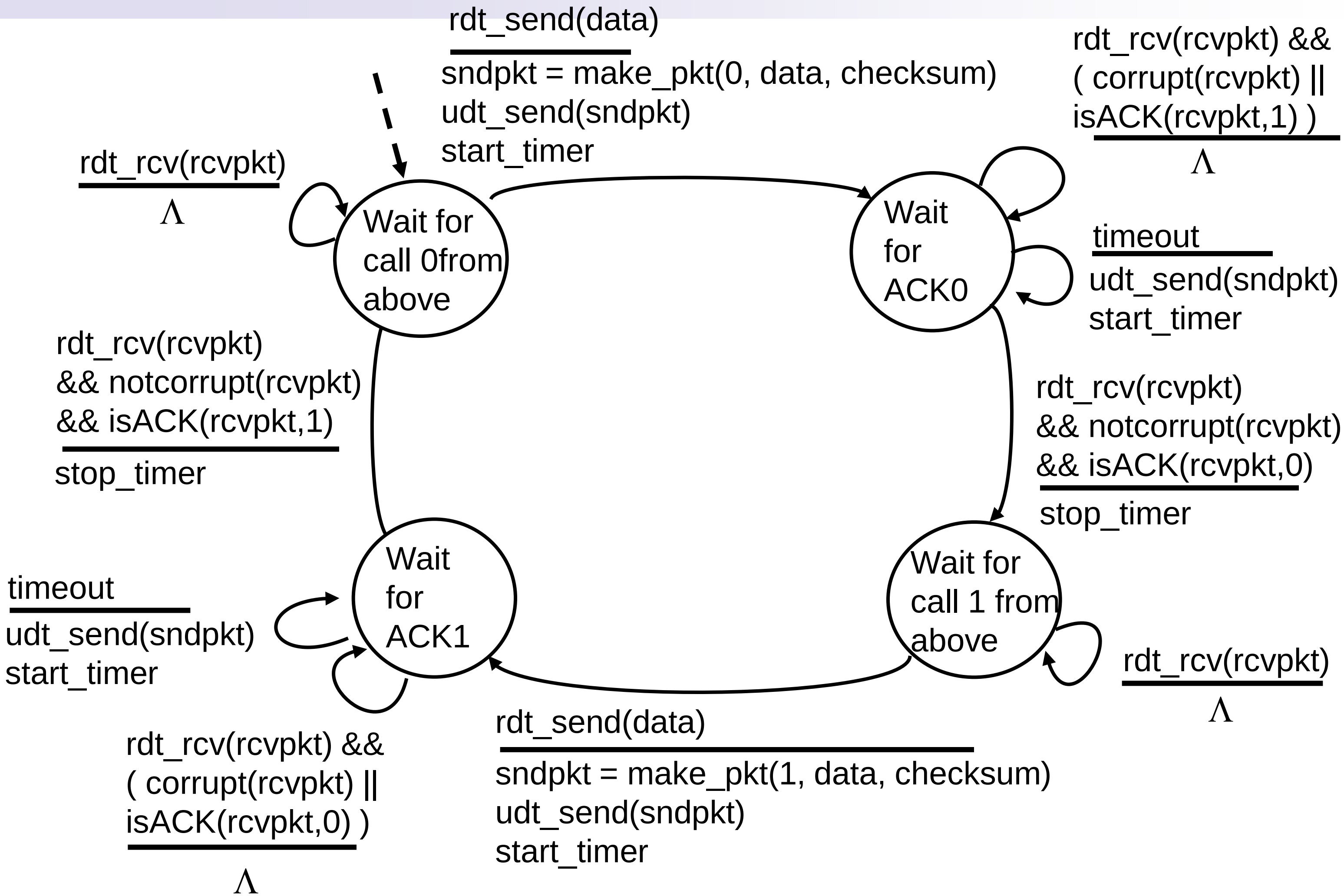
new assumption: underlying channel can also lose packets (data, ACKs)

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

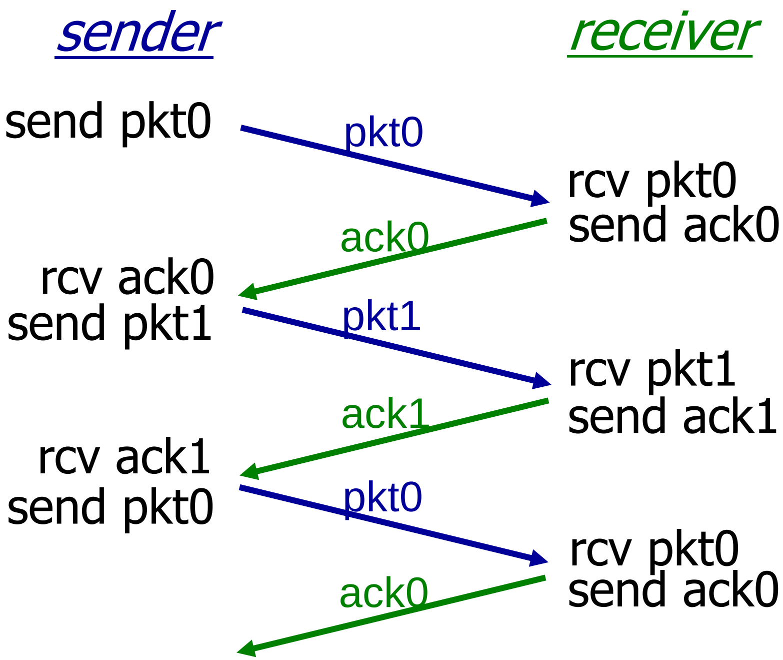
approach: sender waits "reasonable" amount of time for ACK

- 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

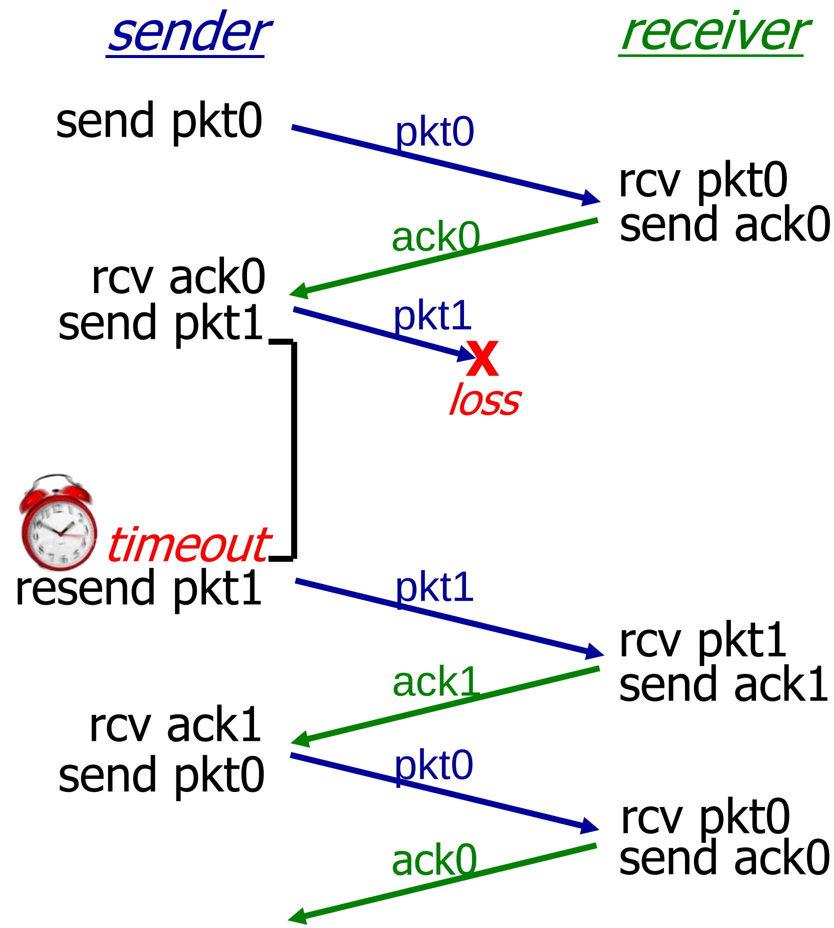
# rdt3.0 sender



# rdt3.0 in action

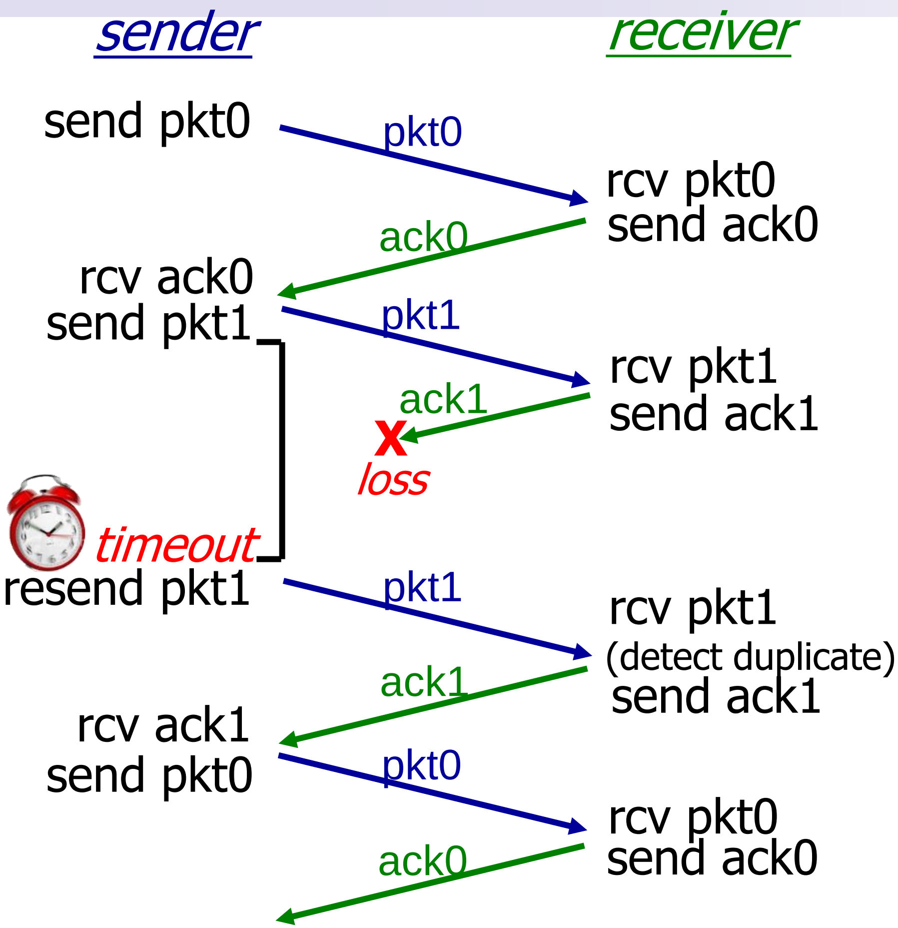



(a) no loss

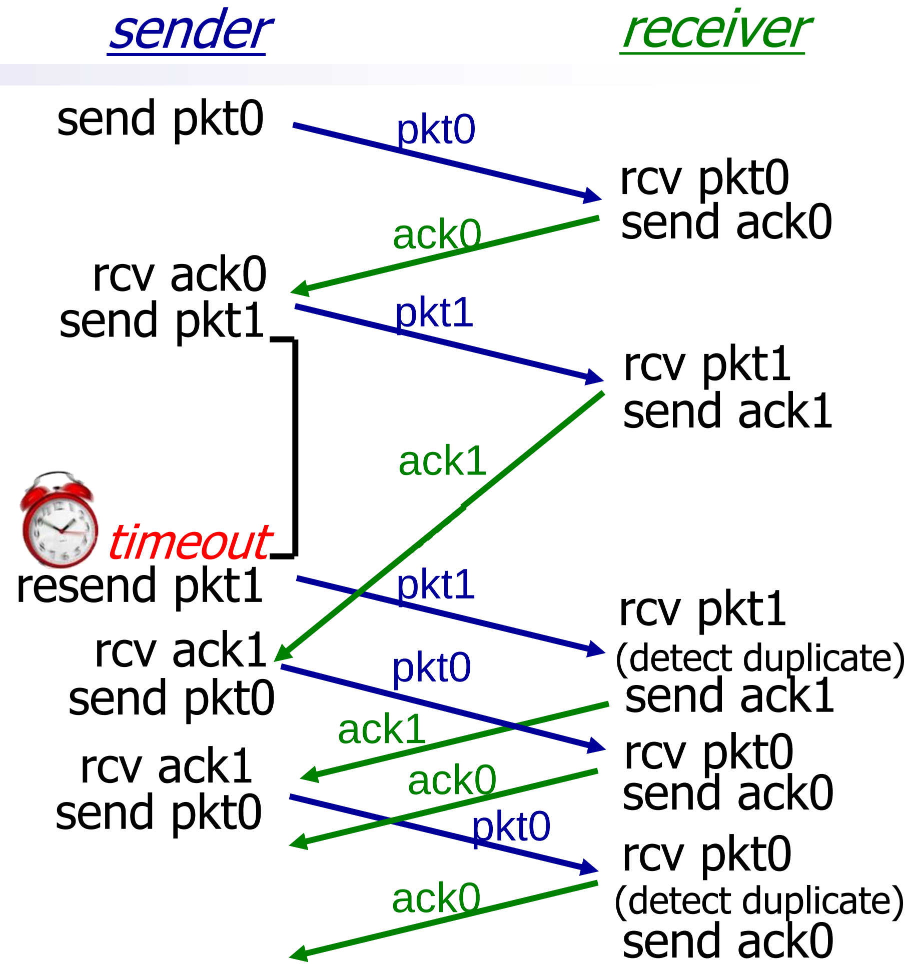



(b) packet loss

# rdt3.0 in action




(c) ACK loss




(d) premature timeout/ delayed ACK

# Summary of transmission methods

- Reliable data transfer over a channel with Bit Errors
  - Positive acknowledgement (ACK)
  - Negative acknowledgement (NAK)
  - ARQ (Automatic Repeat reQuest) protocols
    - Error detection
    - Receiver feedback
    - Retransmission

- Stop & Wait
- Pipelined
  - Go Back N
  - Selective Repeat

# Problem: Performance of rdt3.0

- 1 Gbps link, 15 ms prop. delay, 8000 bit packet:

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

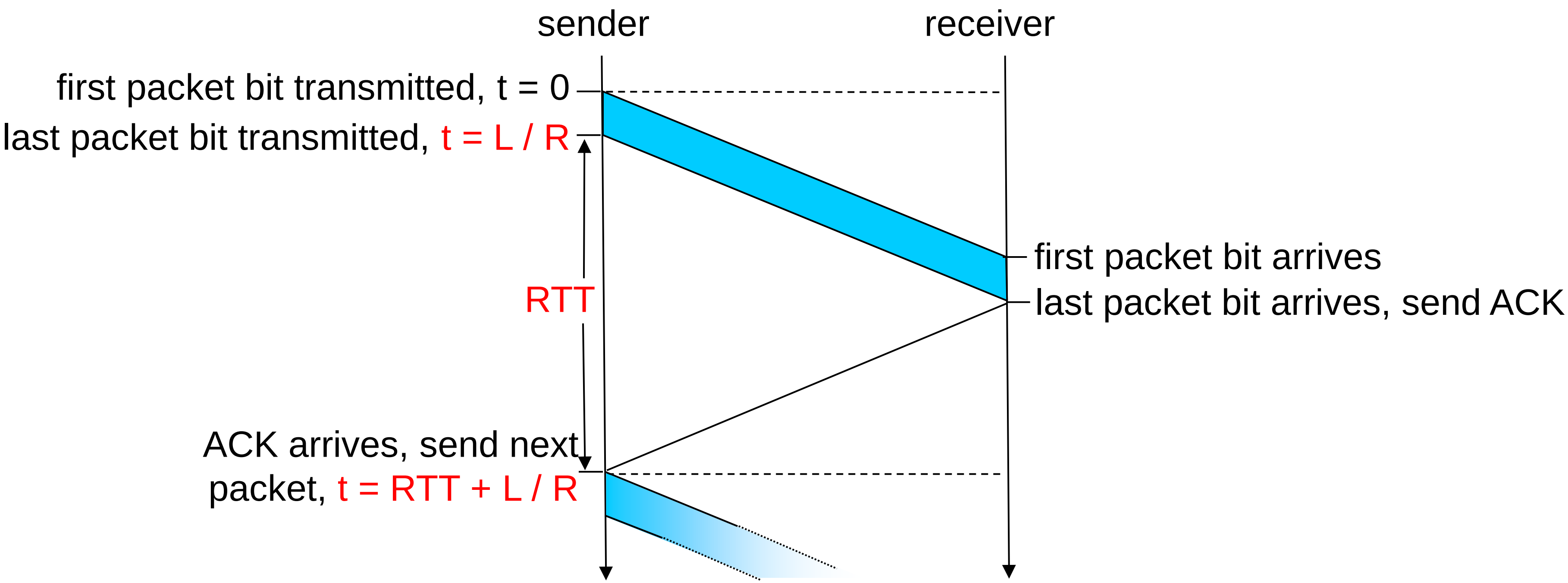
  - $U_{sender}$: *utilization* – fraction of time sender busy sending

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

    - if RTT=30 msec, 1KB pkt every 30 msec: 33KB/sec thruput over 1 Gbps link
    - 1KB = 1000 bytes = 8000 bits

- network protocol limits use of physical resources!

# Stop-and-wait operation

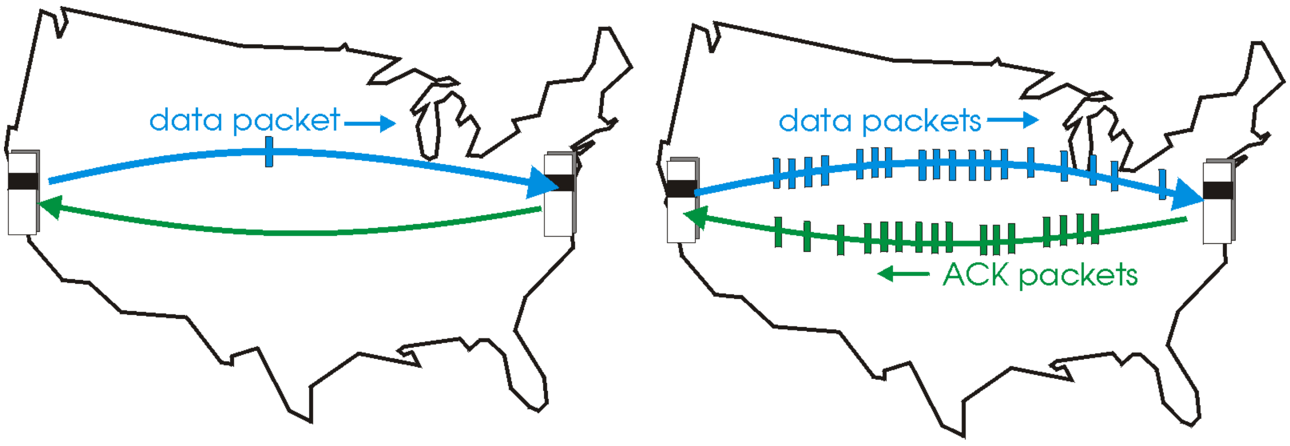



$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

# 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

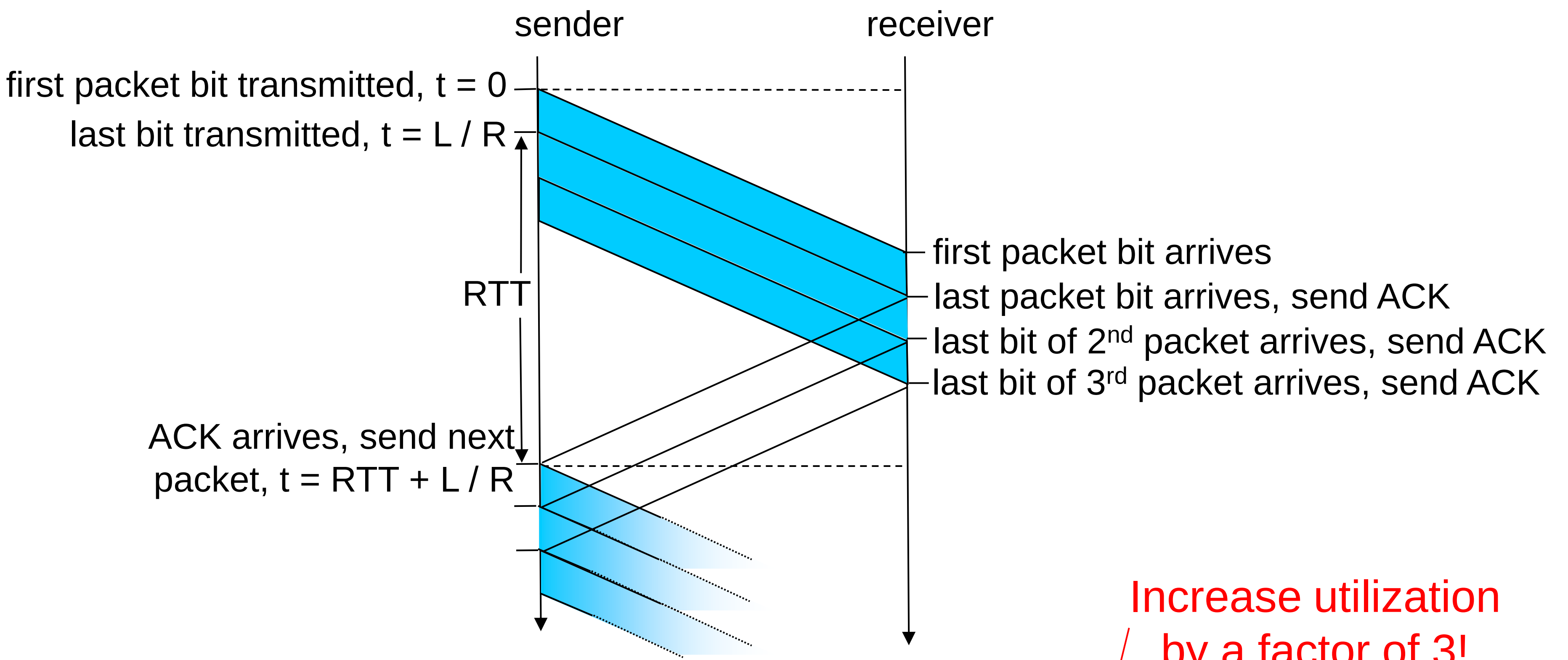



(a) a stop-and-wait protocol in operation

(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!

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

# Agenda

- Transport Layer
- Multiplexing / Demultiplexing

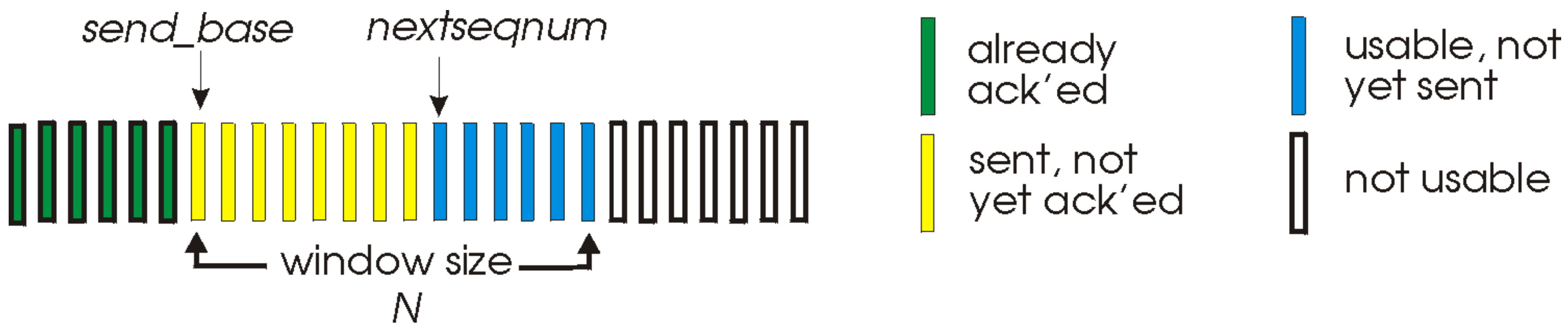
- Reliable Transport
  - Stop-and-wait
  - Pipelined
    - Go back N
    - Selective Request

- TCP
  - Congestion Control
  - Flow Control

# Go-Back-N

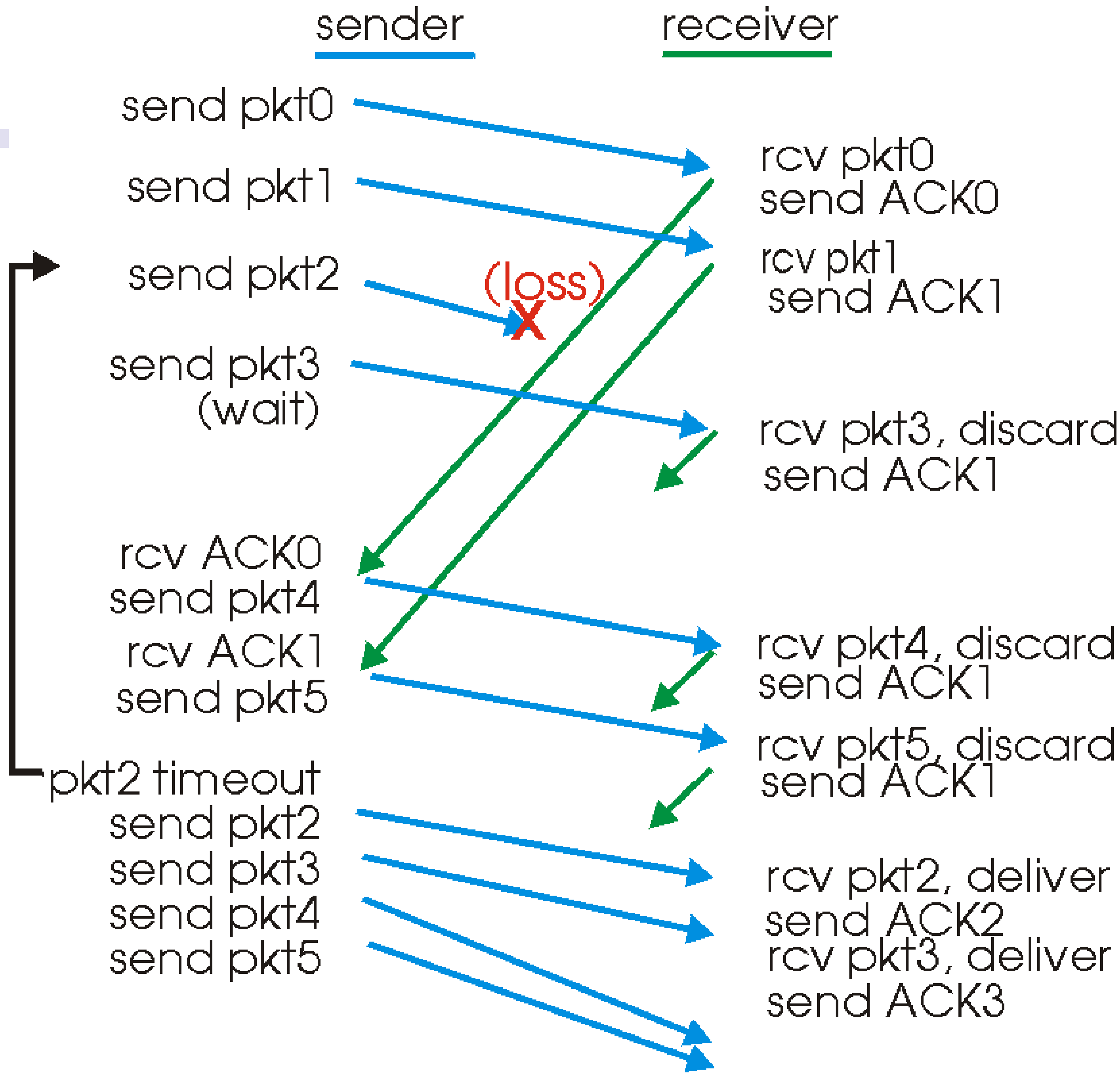
Sender:

- 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 receive duplicate ACKs
- timer for each in-flight pkt
- *timeout(n):* retransmit pkt n and all higher seq # pkts in window
- N referred to as the window size and GBN as a sliding-window protocol
- Why we limit N? (flow control, TCP congestion control)

# GBN in action



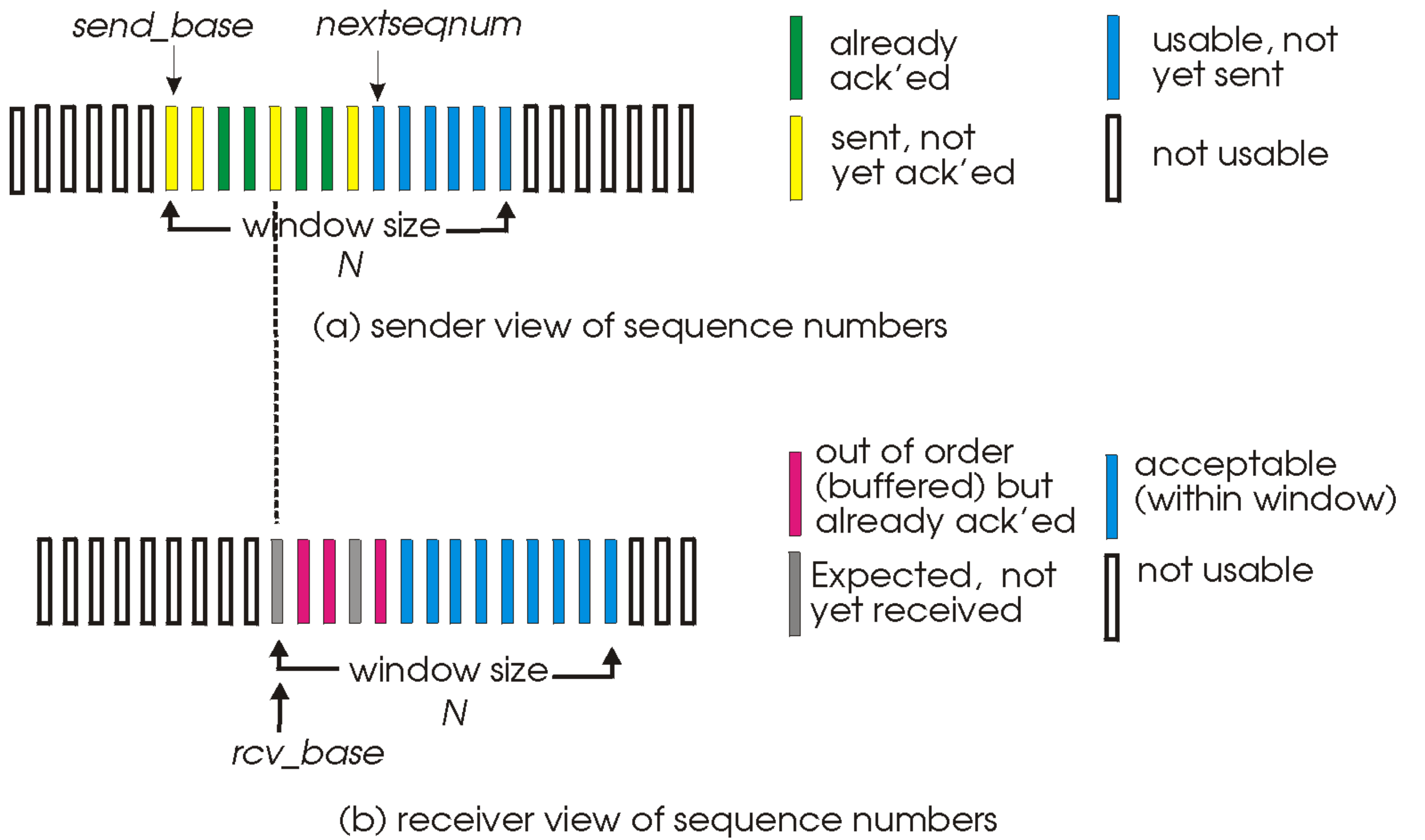
# Selective Repeat

- 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 Request

- Makes sense to transmit only the lost packets
  - But this is true under what assumption ?
    - GBN suffers from performance  problems
      - Window size and bandwidth-delay product are both large, many pkts in pipeline
      - Single pkt error cause GBN to retransmit a large # of pkts, many unnecessarily
      - Probability of channel error increases, the pipeline can become filled with these unnecessary transmissions
  - Can you say a case in which Go-BACK-N might be better
    - GBN protocol allows the sender to potentially "fill the pipeline" with packets
      - Increase the channel utilization than stop-and-wait protocols
  - SR protocols avoid unnecessary retransmissions
    - Sender only retransmits pkts that are received in error at receiver (lost/corrupted)

# 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

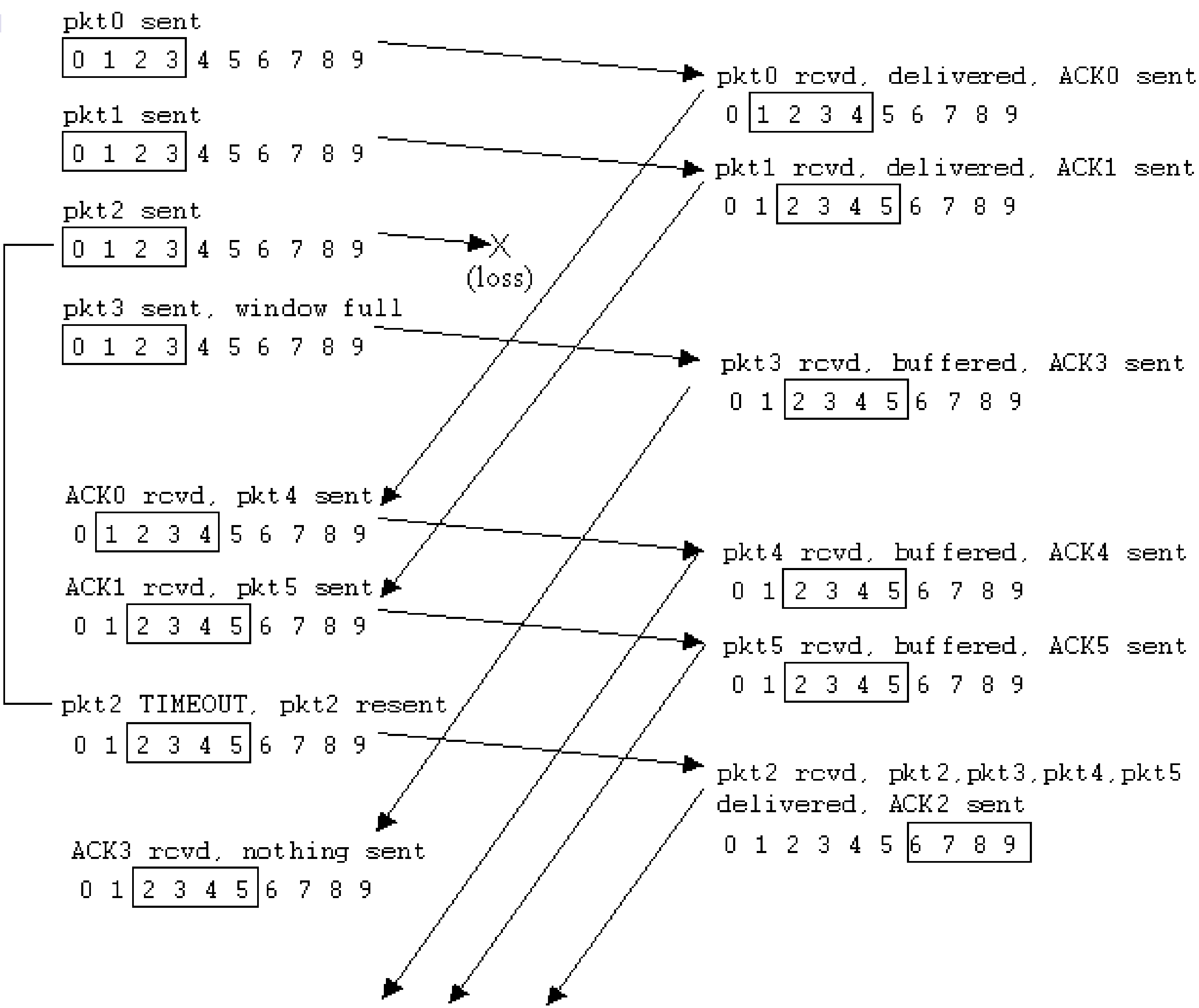
pkt n in [rcvbase-N,rcvbase-1]

- ACK(n)

otherwise:

- ignore

# Selective repeat in action



# Selective Repeat (SR)

- SR receiver acknowledge a correctly received packet whether or not it is in order

- Out-of-order pkts are buffered
  - If any missing pkts (with lower seq #) are received, a batch of pkts can be delivered in order to the upper layer.

# Selective repeat: dilemma

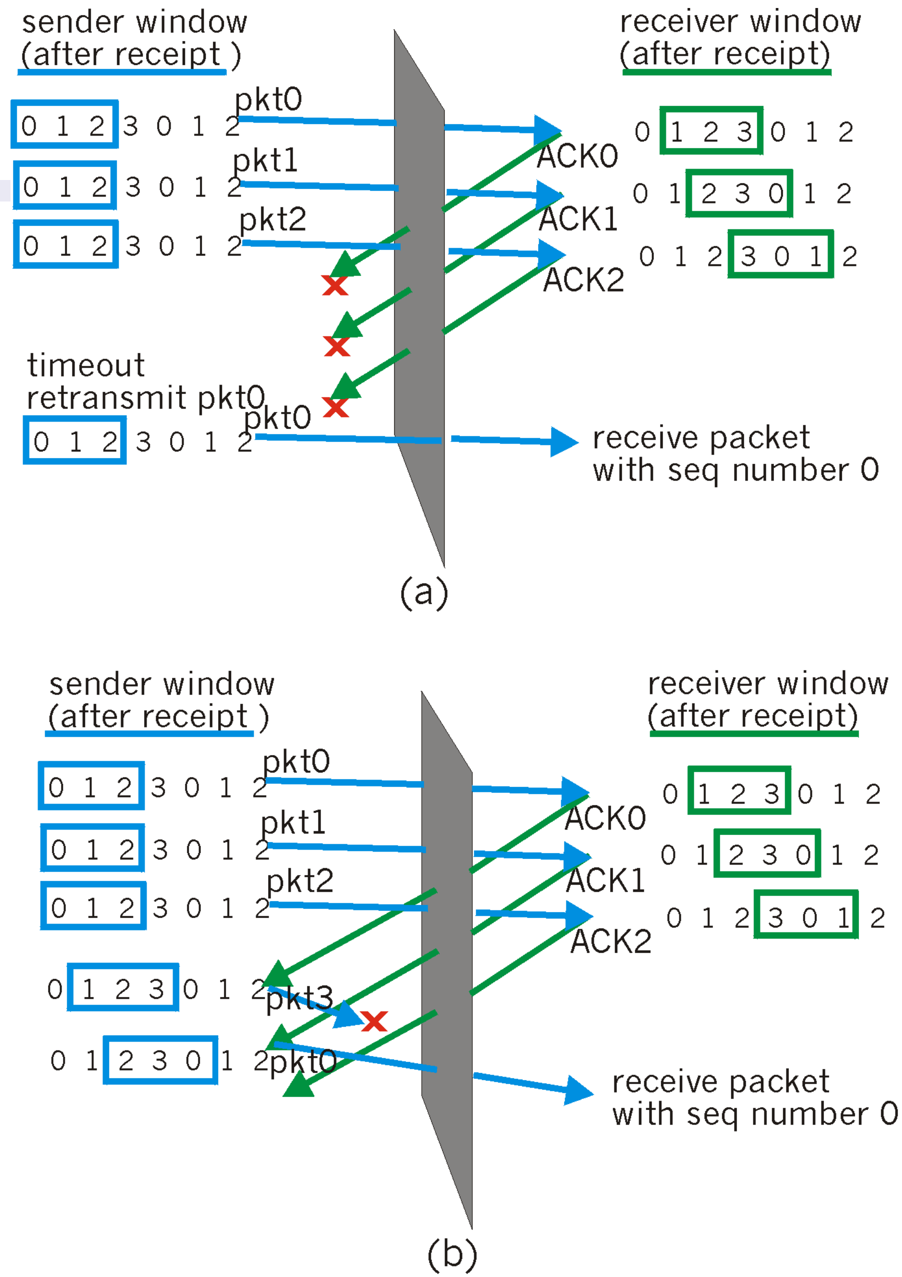
Example:

- seq #’s: 0, 1, 2, 3
- window size=3

- receiver sees no difference in two scenarios!
- incorrectly passes duplicate data as new in (a)
- No way of distinguishing the retransmission of the 1st pkt from an original transmission of the 5th pkt

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

Ans: window size must be ≤ half the size of the seq # space for SR protocols.




What if pkts go out of order

# Agenda

- Transport Layer
- Multiplexing / Demultiplexing

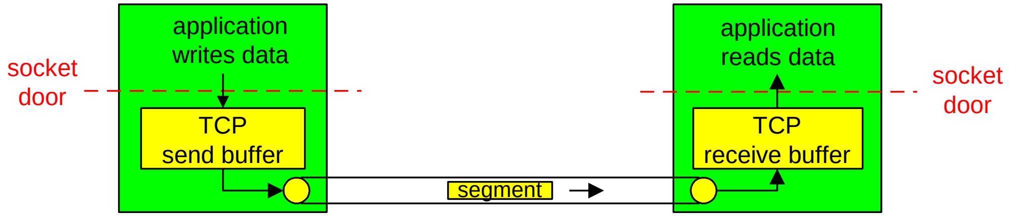
- Reliable Transport
  - Stop-and-wait
  - Pipelined
    - Go back N
    - Selective Request

- TCP
  - Congestion Control
  - Flow Control

# TCP: Overview

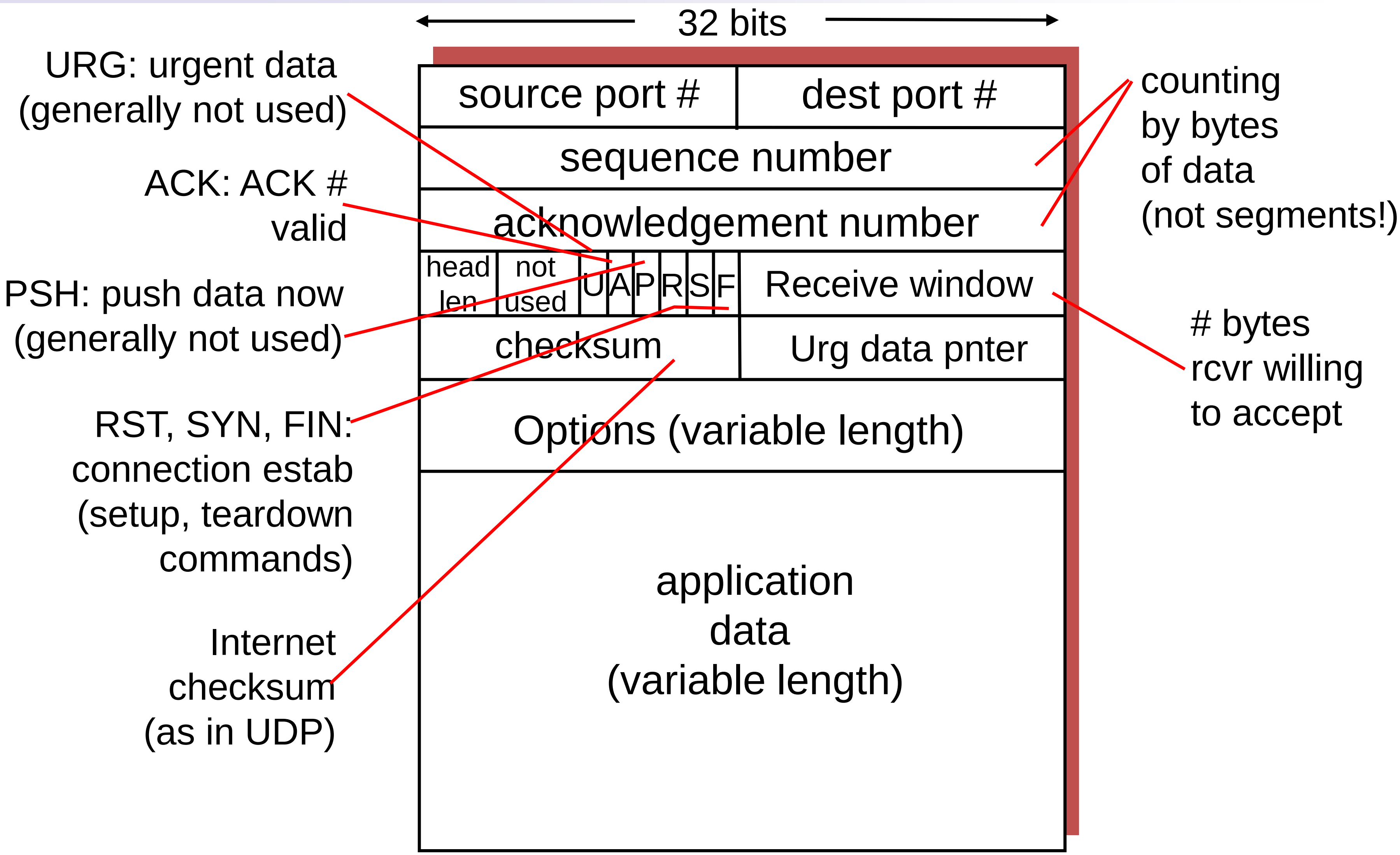
RFCs: 793, 1122, 1323, 2018, 2581

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte steam:*
  - no "message boundaries"
- pipelined:
  - TCP congestion and flow control set window size
- *send & receive buffers*




- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- connection-oriented:
  - handshaking (exchange of control msgs) init's sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

# TCP segment structure



# TCP: Connection-Oriented Transport

- TCP has 3 main components
  - Reliable transmission
  - Congestion Control
  - Flow Control

# Reliable Transmission

- TCP is connection-oriented
  - Sender sends control packets (SYN) and receiver replies (ACK)
  - Receiver also opens a similar connection
  - Full-duplex service; point-to-point connection

- Sender sends a small burst of packets
  - Receiver ACKs: ACK contains the next expected packet (actually byte)
  - Sender receives ACK, and sends a bigger burst
  - Called "Self-clocking" behavior

# Reliable Transmission

- If train of packets lost
  - Sender will not get any ACKs
  - Will timeout (gets alarmed)
  - Retransmit from first un-ACK-ed packet,
  - Drastically reduces window size
- If packet n lost, but (n+1) successful
  - Receiver will send Duplicate ACK
  - Three DupACKs, Resends (n)
  - Fast Retransmit
    - Retransmitting the missing segments before that segment's timer expires
  - Cuts window size by half

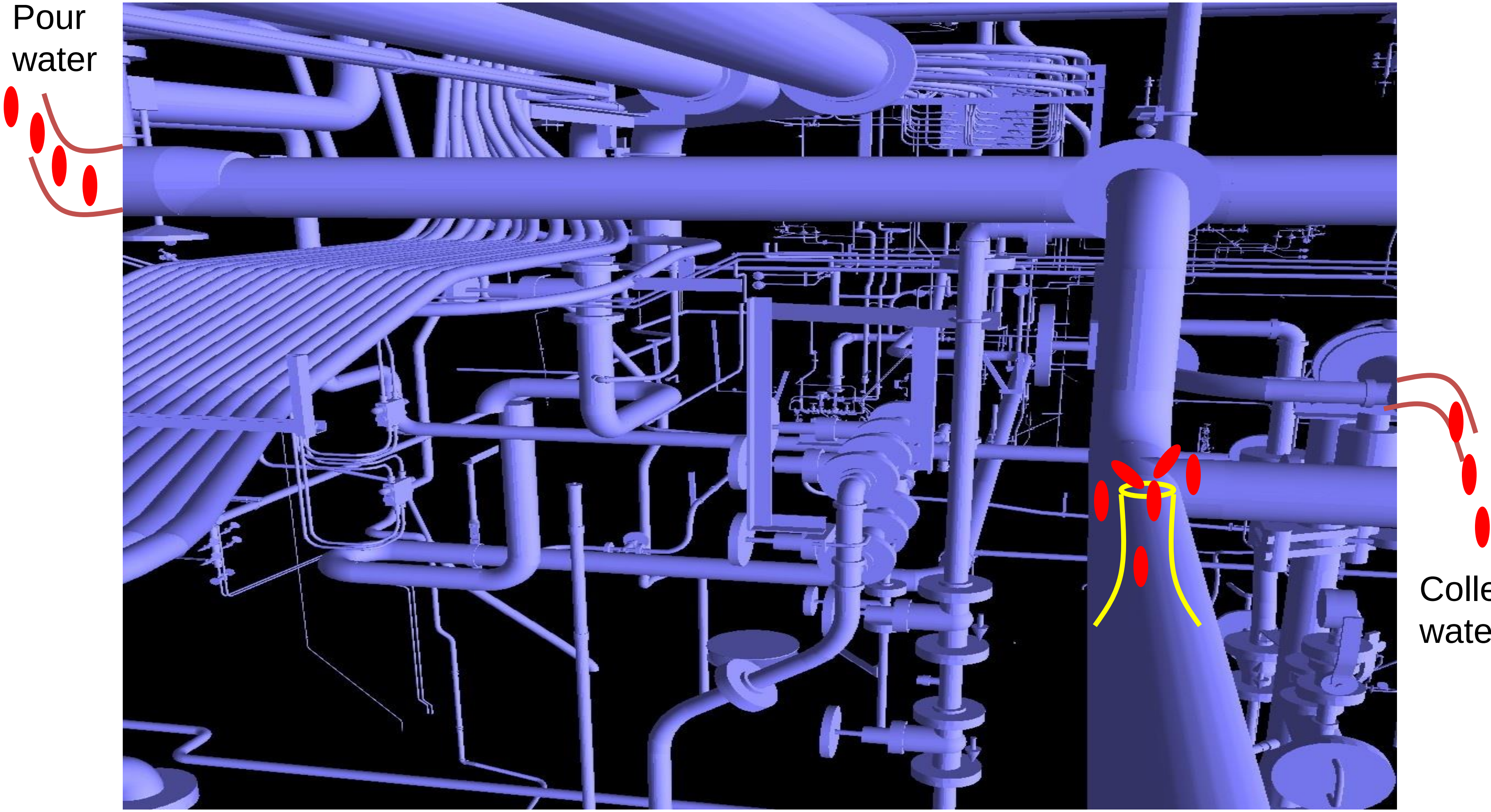
# Congstion Control

# TCP Congestion Control

- Problem Definition
  - How much data should I pump into the network to ensure
    - Intermediate router queues not filling up
    - Fairness achieved among multiple TCP flows
- Why is this problem difficult?
  - TCP cannot have information about the network
  - Only TCP receiver can give some feedbacks
- Approach: sender limit the rate of sending traffic as a function of perceived network congestion
  - How does a TCP sender limit the rate?
  - How does TCP sender perceive that there is congestion?
  - What algorithm should the sender use?

# The TCP Intuition

Pour water

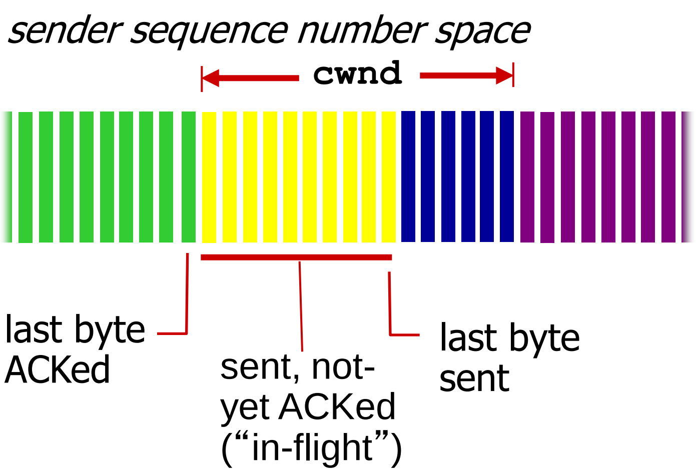


Collect water

# The TCP Protocol (in a nutshell)

- T (sender) transmits few packets, waits for ACK
  - Called slow start
- R (receiver) acknowledges all packet till seq #i by ACK i (optimizations possible)
  - ACK sent out only on receiving a packet
  - Can be Duplicate ACK if expected packet not received
- ACK reaches T → indicator of more capacity
  - T transmits larger burst of packets (self clocking) … so on
  - Burst size increased until packet drops (i.e., DupACK or timeout)
- When T gets DupACK or waits for longer than RTO (Retransmission TimeOut)
  - Assumes congestion → reduces burst size (congestion window)

# TCP Congestion Control: details




- sender limits transmission:

$$\texttt{LastByteSent-LastByteAcked} \leq \texttt{cwnd}$$

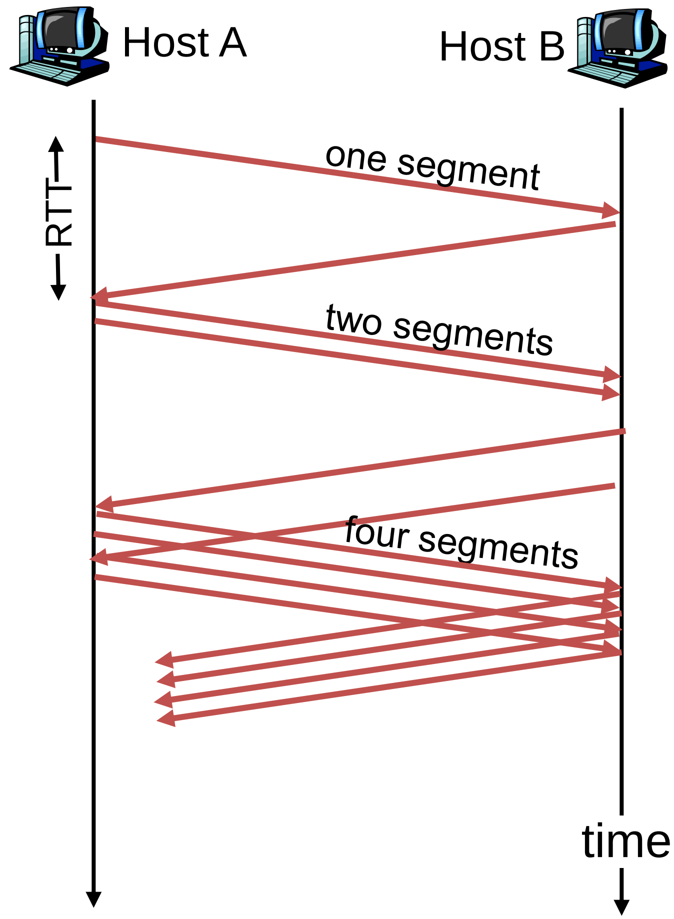
- **cwnd** is dynamic, function of perceived network congestion

*TCP sending rate:*

- *roughly:* send cwnd bytes, wait RTT for ACKS, then send more bytes

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

# TCP Timeline




Think of a blind person trying to stand up in a low ceiling room

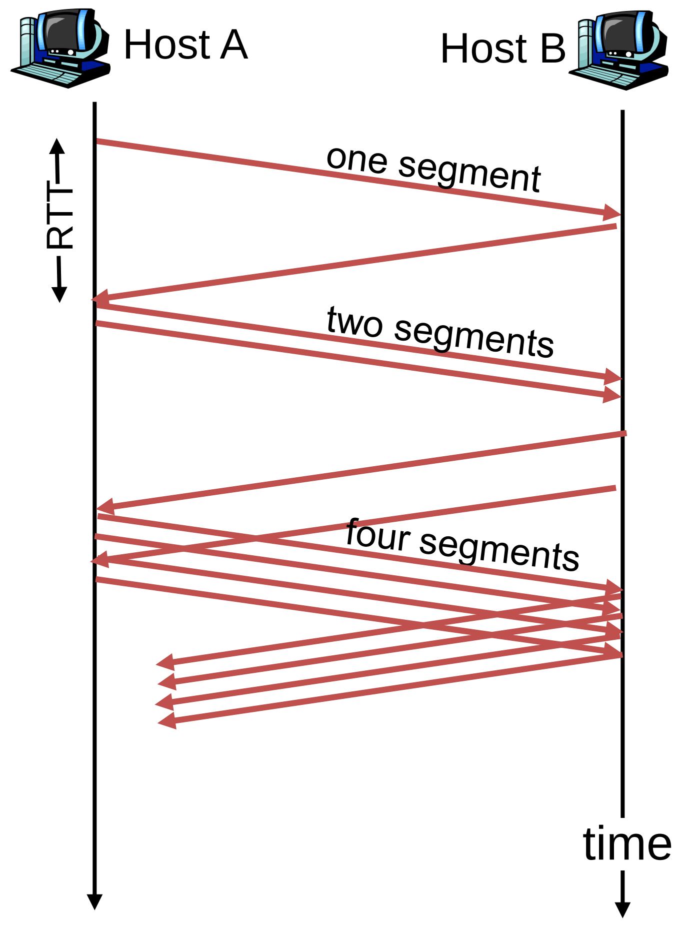
Objective:
Don't bang your head, but stand up quickly

# TCP Congestion Control Algorithm

- A loss segment implies congestion
  - TCP sender's rate should be decreased
- An ACK indicates that network is delivering the sender's segment to the receiver
  - TCP sender's rate can be increased
- Bandwidth probing
  - TCP sender increases transmission rate to probe when congestion onset begins
  - backs off from that rate and then begins probing to see if congestion rate has changed
- Jacobson 1988
  - Slow start; congestion avoidance; fast recovery

# TCP Slow Start

- 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
- When this exponential growth rate should end?



# TCP Slow Start (more)

- If there is a loss event (i.e., congestion) indicated by a timeout
  - TCP sender sets the value of `cwnd` to 1
  - Begin the slow start process anew
  - Sets the value of 2nd state variable `ssthresh` (slow start threshold) to `cwnd/2`
  - Slow start ends when `cwnd = ssthresh`
- TCP transitions into congestion avoidance (CA) mode
  - TCP increases `cwnd` more cautiously when in CA mode
  - Final end of slow start happens if 3 duplicate ACKs are detected
  - TCP performs a fast retransmit
  - Enters a fast recovery state

# TCP Slow Start (more)

- Congestion avoidance state
  - value of `cwnd` is approx. half its value when congestion was last detected
  - Rather than doubling the value of `cwnd` every RTT TCP adopts a more conservative approach
  - Increase the value of `cwnd` by just a single MSS

- TCP performs a fast retransmit
  - Enters a fast recovery state

# TCP: detecting, reacting to loss

- loss indicated by timeout:
  - **cwnd** set to 1 MSS
  - window then grows exponentially (as in slow start) to threshold, then grows linearly

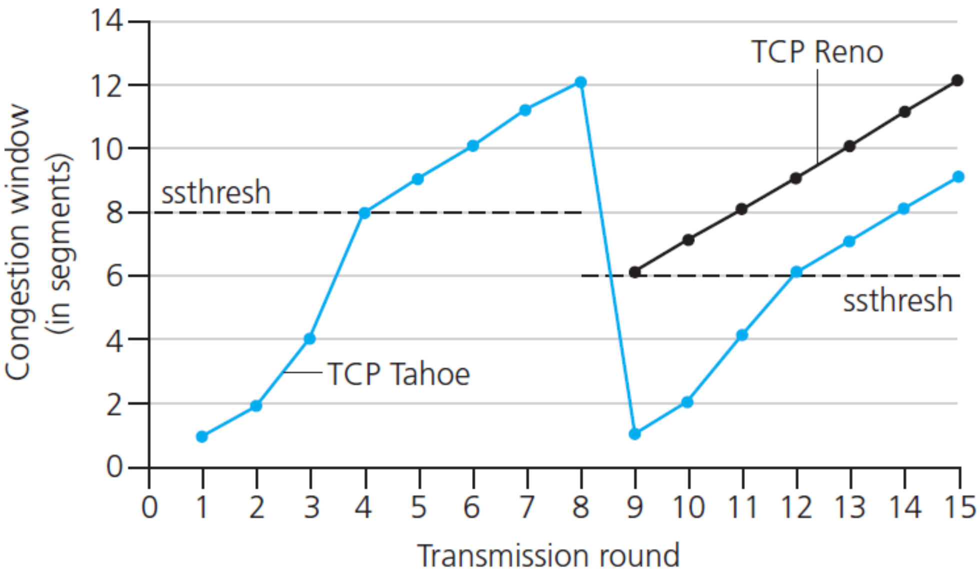
- loss indicated by 3 duplicate ACKs: TCP RENO (newer version)
  - dup ACKs indicate network capable of delivering some segments
  - **cwnd** is cut in half window then grows linearly

- TCP Tahoe (earlier version) always sets **cwnd** to 1 (timeout or 3 duplicate acks)

# TCP: switching from slow start to CA

Q: When should the exponential increase switch to linear?

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




## Implementation:

- variable **ssthresh**
- on loss event, **ssthresh** is set to 1/2 of **cwnd** just before loss event

# TCP 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.

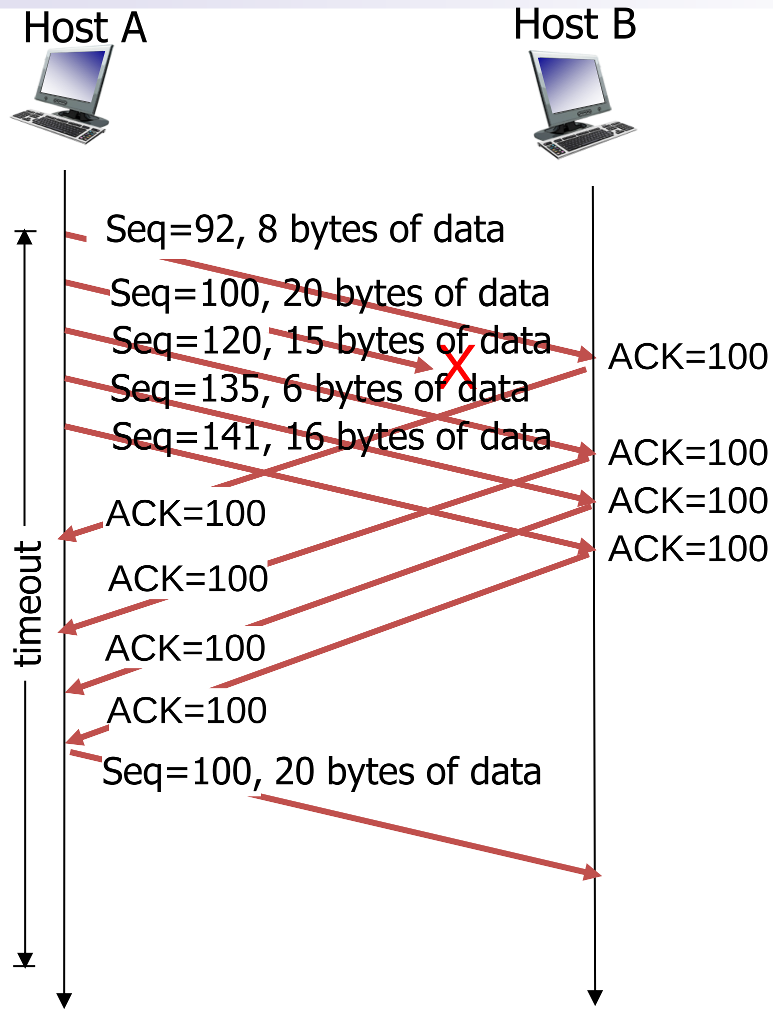
*TCP fast retransmit*

if sender receives 3 ACKs for same data

("triple duplicate ACKs"), resend unacked segment with smallest seq #

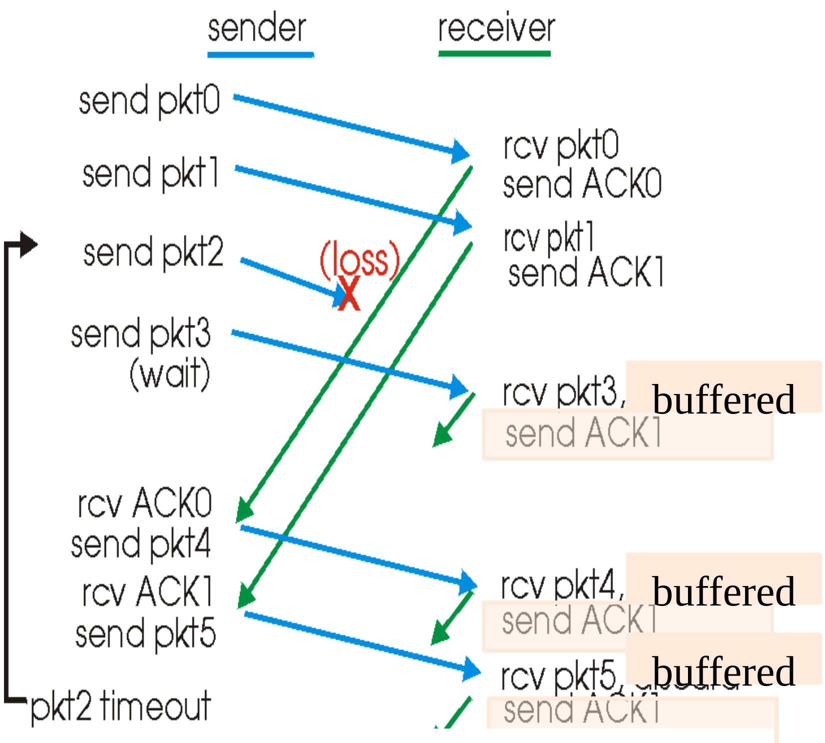
- likely that unacked segment lost, so don't wait for timeout

# TCP fast retransmit




fast retransmit after sender receipt of triple duplicate ACK

# Understanding 3 Duplicate ACKs from GBN




Q: Is TCP Go-Back-N or Selective Repeat?

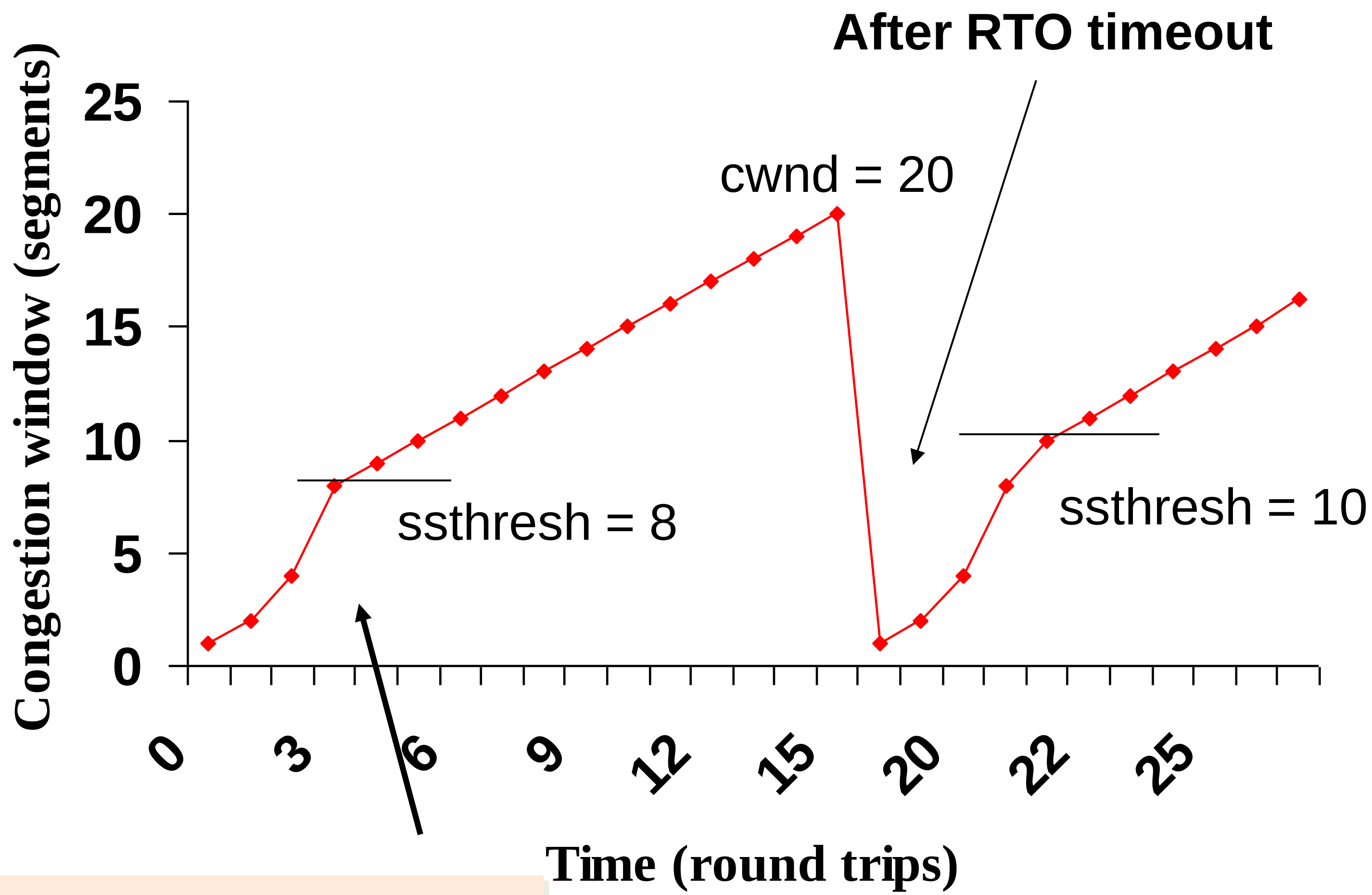
A: a) TCP implementation buffers correctly received but out-of-order segments.

b) Selective acknowledgement allows a TCP receiver to acknowledge out-of order segments selectively rather than just cumulatively acknowledging the last correctly received, in-order segment.

c) Selective retransmission: skipping the retransmission of segments that have already been selectively acknowledged by the receiver

d) TCP is a hybrid GBN and SR protocol

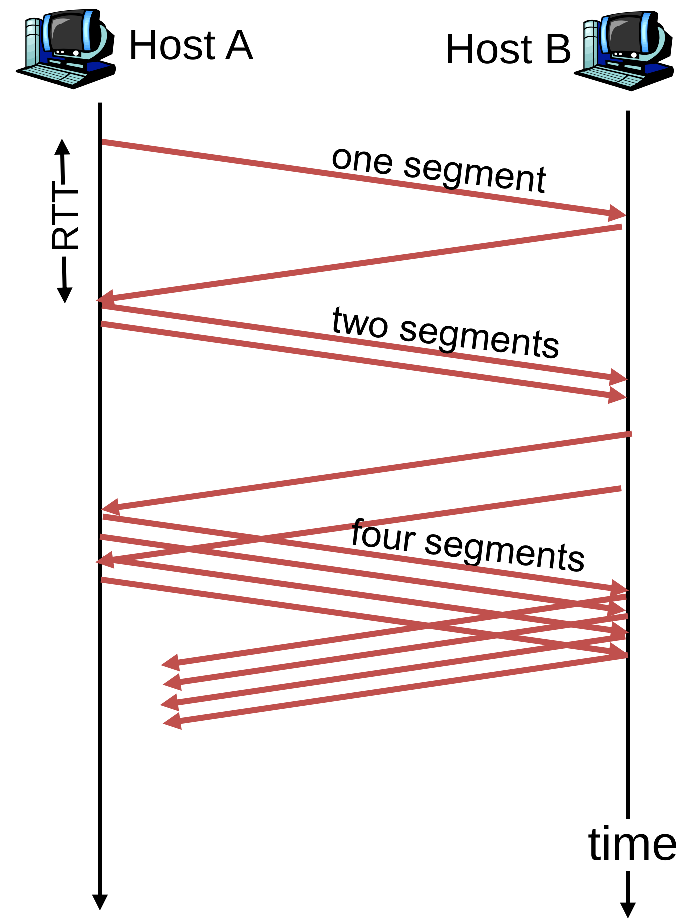
# More Example: When waited for > RTO




double **CongWin** every RTT
done by incrementing **CongWin**
for every ACK received

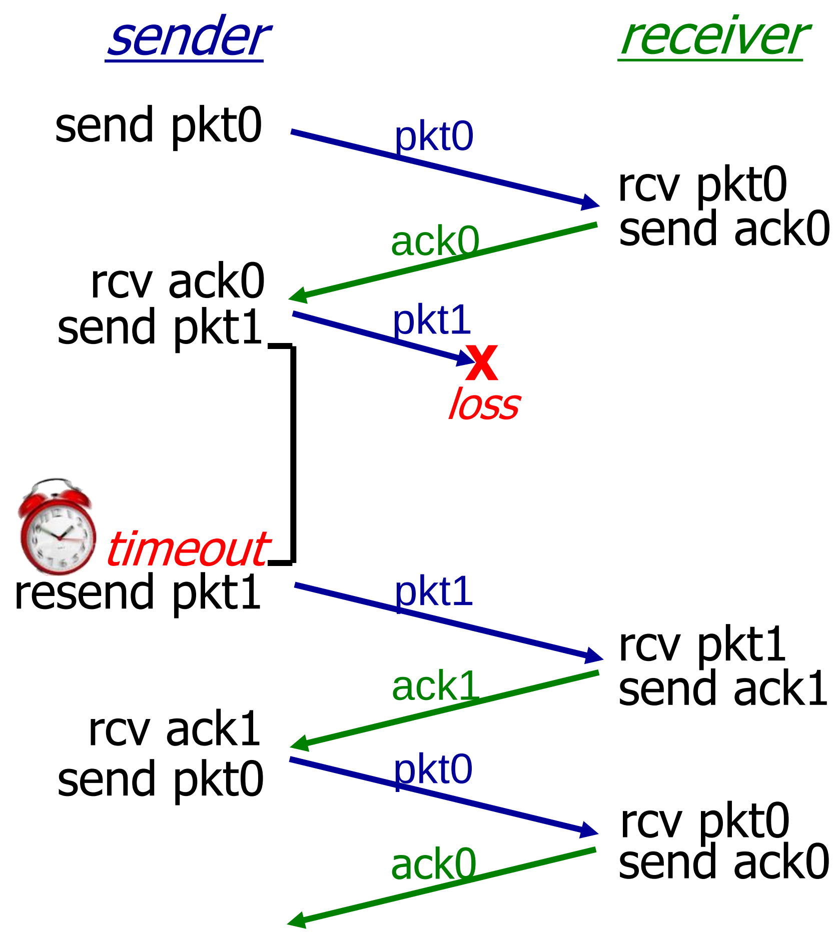
# Understanding RTT (X axis) & cwnd/segments (Y axis) relationship

- Relation between RTT (Transmission Round) and Packets Sequence Number/ MSS (Maximum Segment Size)
  - 1st RTT: pkt 1
  - 2nd RTT: pkt 2 & 3
  - 3rd RTT: pkt 4, 5, 6 & 7
  - 4th RTT: pkt 8, 9, 10, 11, 12, 13, 14, & 15
  - 5th RTT: pkt 16 to 31
  - 6th RTT: pkt 32 to 63



# Next Step

- We talked about the congestion window
  - Setting up the congestion window size
- What about RTT and Retransmission Timeout?
  - How to determine the value of RTT/RTO?




(b) packet loss

# Timeout -- function of RTT

Q: how to set TCP timeout value?

- longer than RTT
  - but RTT varies
  - How much larger?
- too short: premature timeout
  - unnecessary retransmissions
- too long: slow reaction to segment loss
- How should the RTT be estimated in first place?
- Should a timer be associated with each and every unacknowledged segment? [TCP work by Jacobson 1988]

Q: how to estimate RTT?

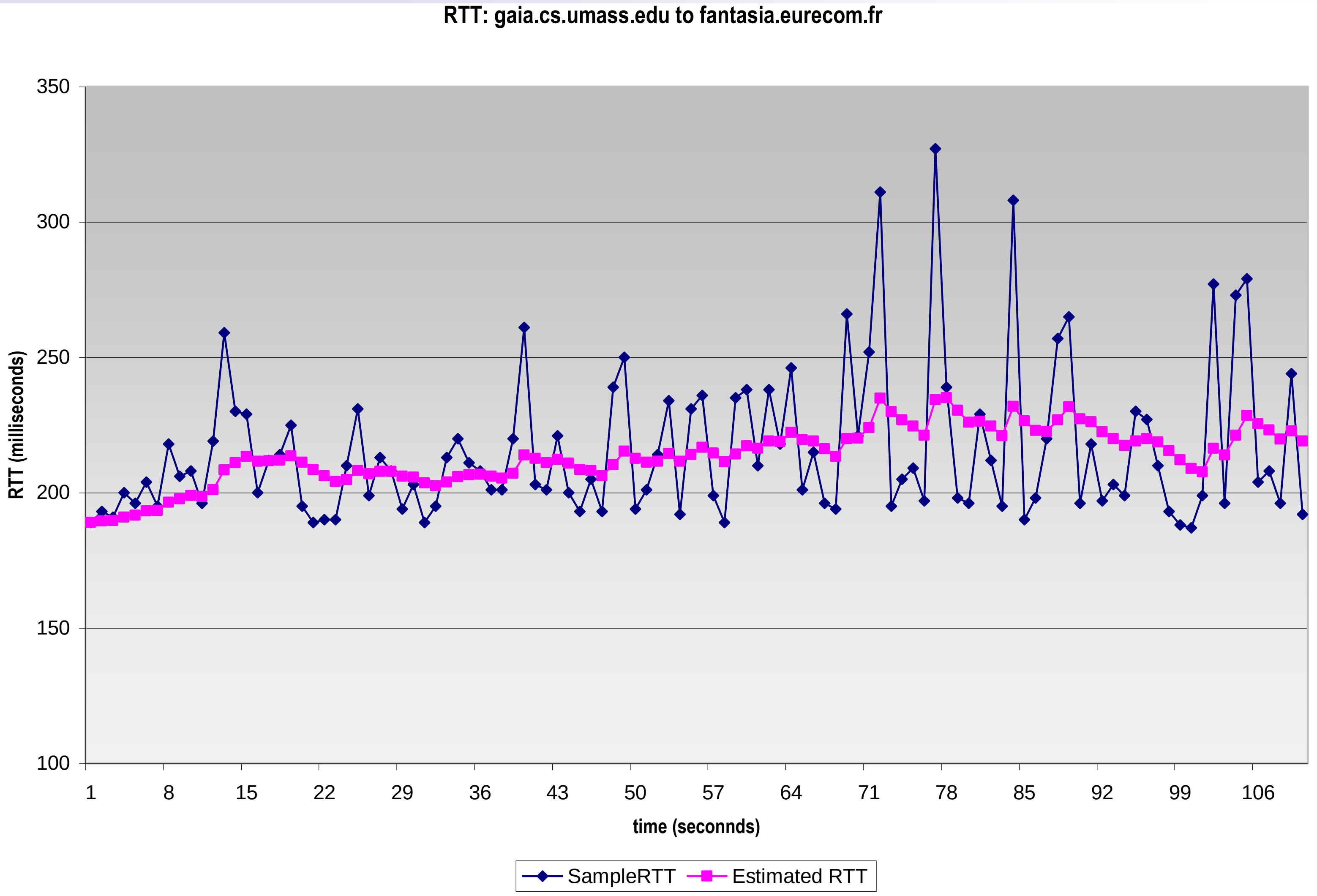
- `SampleRTT`: measured time from segment transmission until ACK receipt
- One of the transmitted but currently unacknowledged segment
- Vary due to congestion in the routers and varying load on the end systems
- **`SampleRTT`** will vary, want estimated RTT "smoother"
  - average several recent measurements, not just current **`SampleRTT`**

# TCP Round Trip Time

**EstimatedRTT = (1- $\alpha$)*EstimatedRTT + $\alpha$*SampleRTT**

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

# Example RTT estimation:



# Timeout

Setting the timeout

- **`EstimatedRTT`** 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:

- Interval should be greater than or equal to **`EstimatedRTT`**, shouldn't be too large
  - Unnecessary retransmissions would be sent or TCP would not quickly retransmit
  - **`EstimatedRTT`** + Margin

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

# TCP: Connection-Oriented Transport

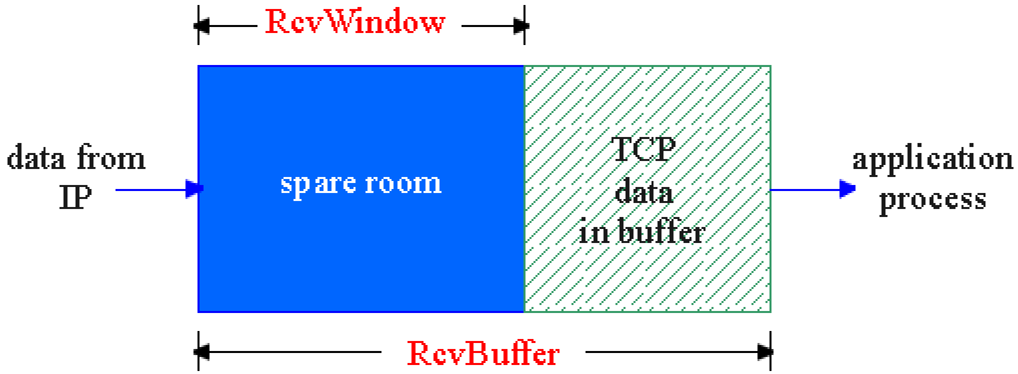
- TCP has 3 main components
  - Reliable transmission
  - Congestion Control
  - Flow Control

# TCP Flow Control

- Problem Definition
  - The receiver has limits on buffer
  - If many nodes transmitting to same receiver
    - Losses may happen at receiver
  - Need to avoid such losses

- Solution
  - Receiver tells transmitter how much space left
  - Transmitter chooses its congestion window accordingly

# 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

# Chapter 3: Transport Layer Summary

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

next:

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

Questions?

...