The University of Danang
University of Science and Technology

CHAPTER 3
TRANSPORT LAYER

FACULTY OF INFORMATION TECHNOLOGY
PhD. LE TRAN DUC

OUTLINE

1. Transport Services & Protocols
2. Multiplexing & Demultiplexing
3. Principle of Reliable Data Transfer
4. UDP – Connectionless Transport
5. TCP – Connection-Oriented Transport
6. TCP Flow Control
7. TCP Congestion Control

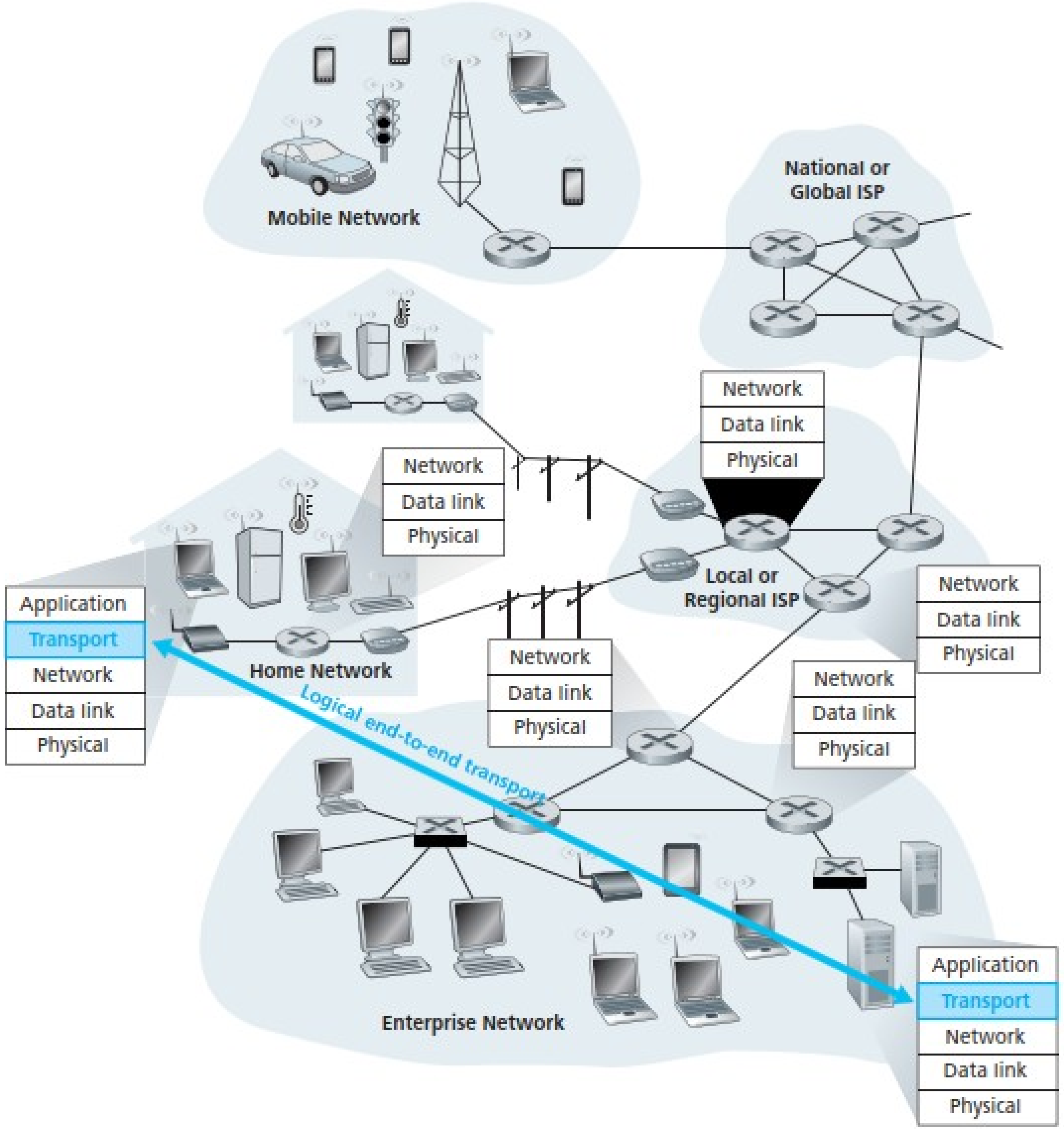
The University of Danang
University of Science and Technology

1. TRANSPORT SERVICES & PROTOCOLS

Faculty of Information Technology
PhD. Le Tran Duc

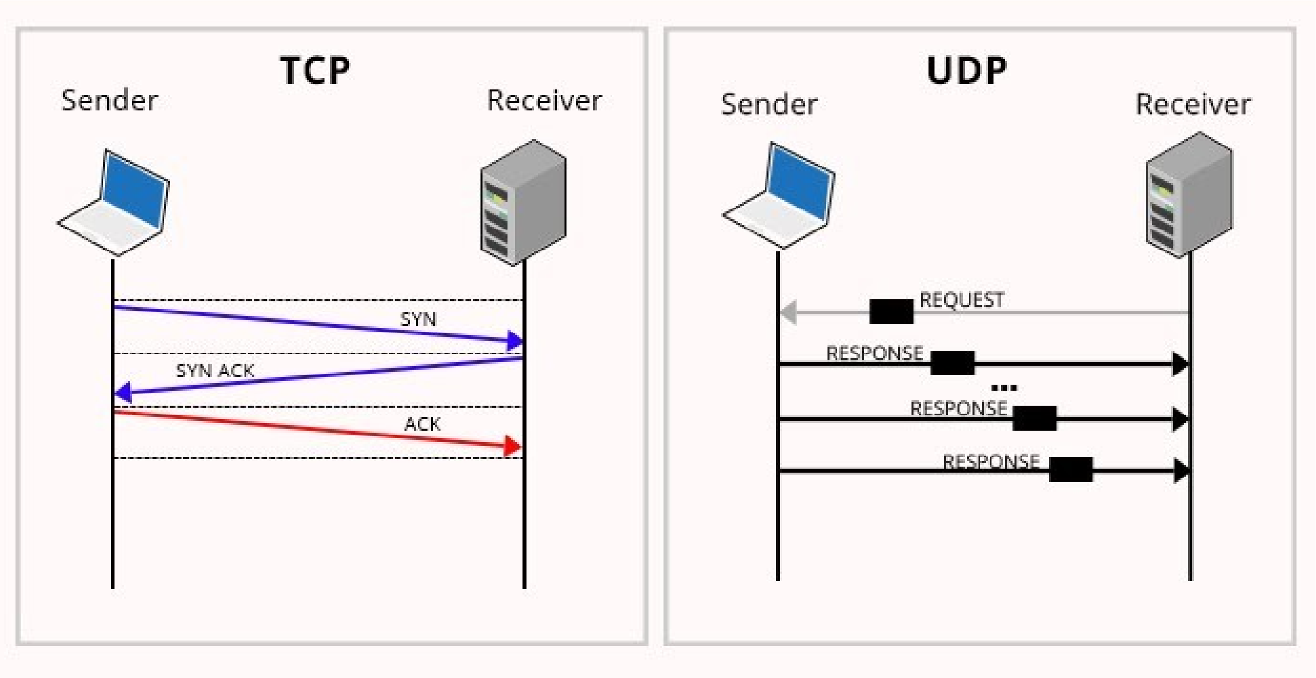
TRANSPORT SERVICES & PROTOCOLS

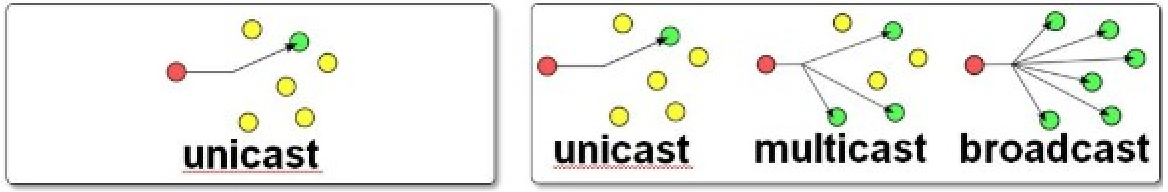
- Provide ***logical communication*** between app **processes** running on different hosts
- Transport protocols run in end systems
  - **Send side**: breaks app messages into ***segments***, passes to network layer
  - **Rcv side**: reassembles segments into messages, passes to app layer
- More than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



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
  - process-to-process data delivery and error checking—are the only two services that UDP provides
- Services not available for IP protocol:
  - delay guarantees
  - bandwidth guarantees





TCP vs UDP more details…

TCP service:

- *Reliable transport* between sending and receiving process
- *Flow control:* sender won’t overwhelm receiver
- *Congestion control:* throttle sender when network overloaded
- *Does not provide:* timing, minimum throughput guarantee, security
- *Connection-oriented:* setup required between client and server processes

UDP service:

- *Unreliable data transfer* between sending and receiving process
- *Does not provide:* reliability, flow control, congestion control, timing, throughput guarantee, security, or connection setup,

2. MULTIPLEXING & DEMULTIPLEXING

Faculty of Information Technology
PhD. Le Tran Duc

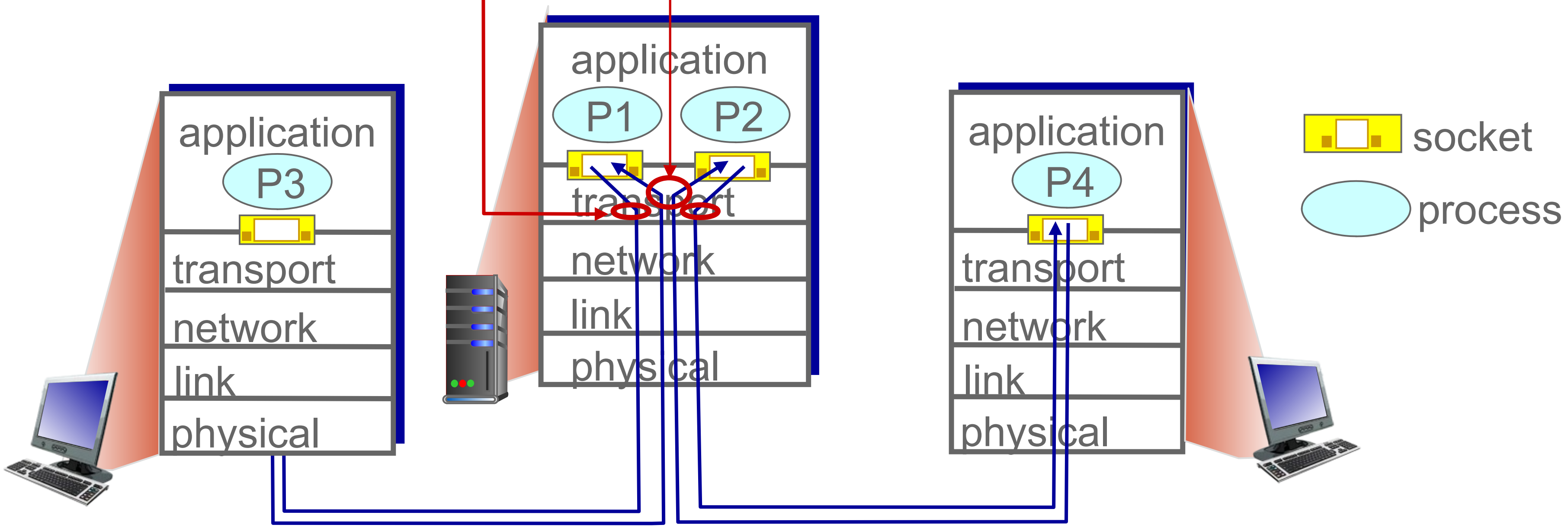
MULTIPLEXING/DEMULTIPLEXING

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

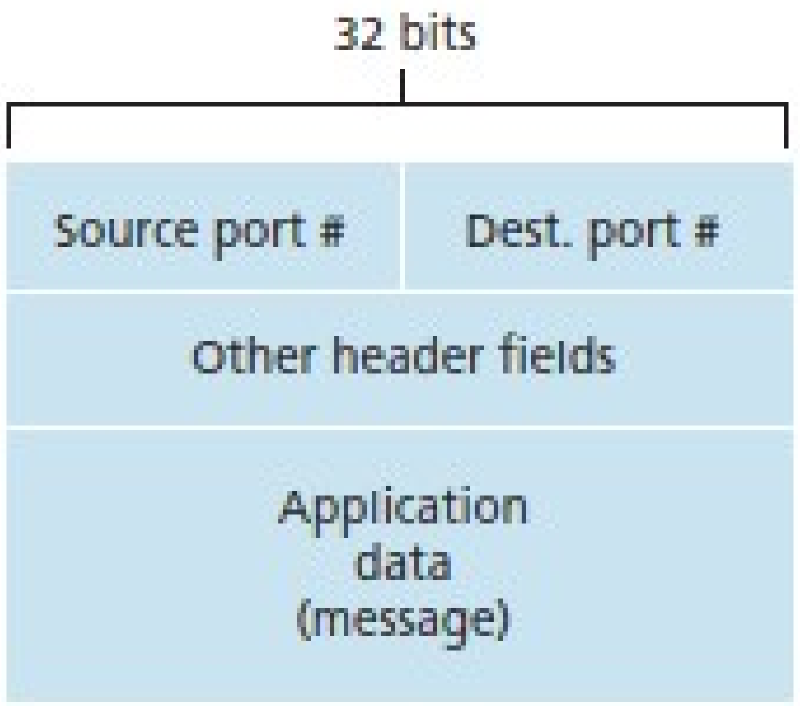


HOW DEMULTIPLEXING WORKS

Multiplexing = **gathering** data chunks at the source, **encapsulating** with header information, **passing** the segment to the network layer

Demultiplexing = **delivering** the data in a transport-layer segment **to** the **correct socket**

- **Host receives IP packets**
  - Each packet has source IP address, destination IP address
  - Each packet 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

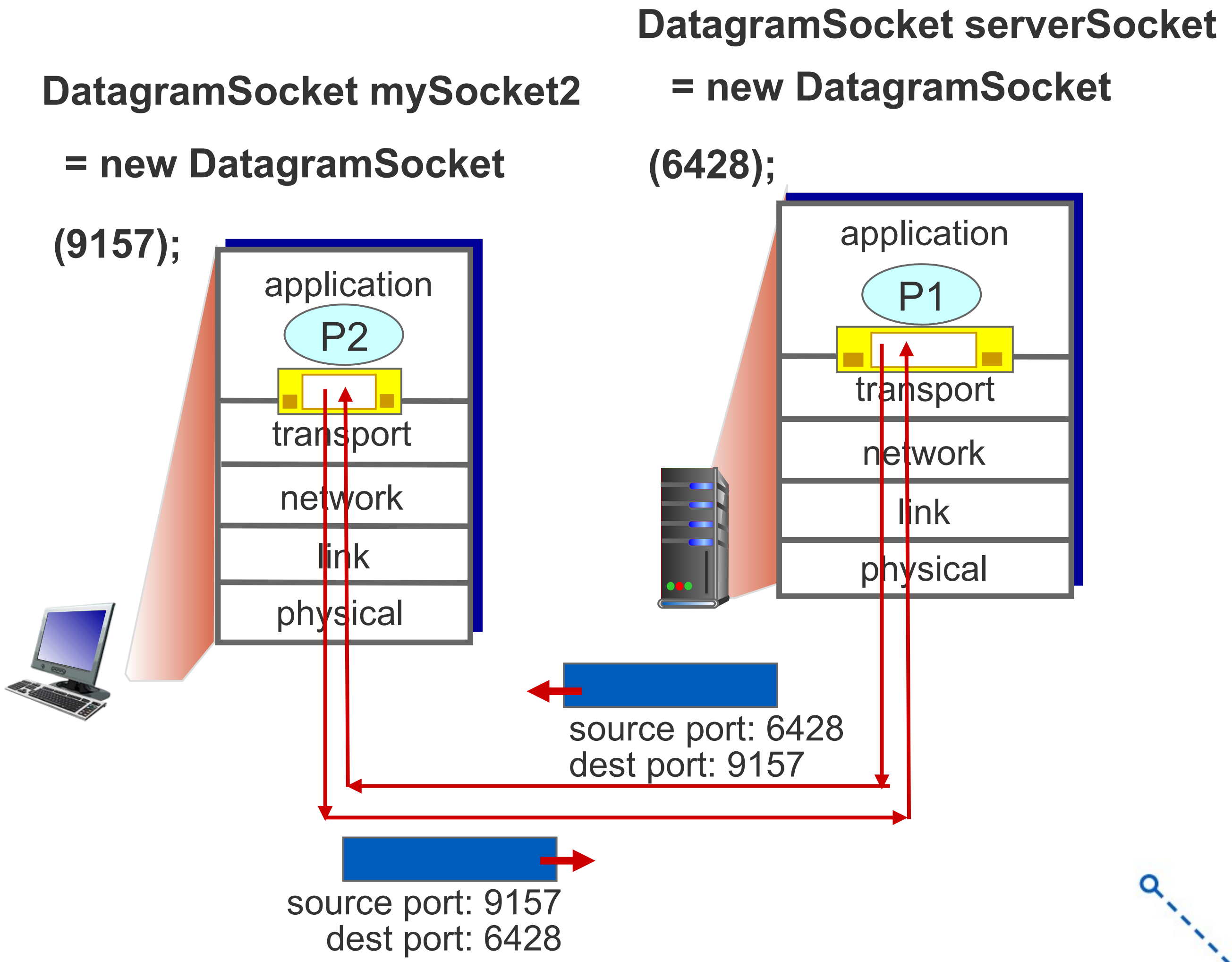
COMMON PORTS

7	Echo	554	RTSP	2745	Bagle.H	6891-6901	Windows Live
19	Chargen	546-547	DHCPv6	2967	Symantec AV	6970	Quicktime
20-21	FTP	560	rmonitor	3050	Interbase DB	7212	GhostSurf
22	SSH/SCP	563	NNTP over SSL	3074	XBOX Live	7648-7649	CU-SeeMe
23	Telnet	587	SMTP	3124	HTTP Proxy	8000	Internet Radio
25	SMTP	591	FileMaker	3127	MyDoom	8080	HTTP Proxy
42	WINS Replication	593	Microsoft DCOM	3128	HTTP Proxy	8086-8087	Kaspersky AV
43	WHOIS	631	Internet Printing	3222	GLBP	8118	Privoxy
49	TACACS	636	LDAP over SSL	3260	iSCSI Target	8200	VMware Server
53	DNS	639	MSDP (PIM)	3306	MySQL	8500	Adobe ColdFusion
67-68	DHCP/BOOTP	646	LDP (MPLS)	3389	Terminal Server	8767	TeamSpeak
69	TFTP	691	MS Exchange	3689	iTunes	8866	Bagle.B
70	Gopher	860	iSCSI	3690	Subversion	9100	HP JetDirect
79	Finger	873	rsync	3724	World of Warcraft	9101-9103	Bacula
80	HTTP	902	VMware Server	3784-3785	Ventrilo	9119	MXit
88	Kerberos	989-990	FTP over SSL	4333	mSQL	9800	WebDAV
102	MS Exchange	993	IMAP4 over SSL	4444	Blaster	9898	Dabber
110	POP3	995	POP3 over SSL	4664	Google Desktop	9988	Rbot/Spybot
113	Ident	1025	Microsoft RPC	4672	eMule	9999	Urchin
119	NNTP (Usenet)	1026-1029	Windows Messenger	4899	Radmin	10000	Webmin
123	NTP	1080	SOCKS Proxy	5000	UPnP	10000	BackupExec
135	Microsoft RPC	1080	MyDoom	5001	Slingbox	10113-10116	NetIQ
137-139	NetBIOS	1194	OpenVPN	5001	iperf	11371	OpenPGP
143	IMAP4	1214	Kazaa	5004-5005	RTP	12035-12036	Second Life
161-162	SNMP	1241	Nessus	5050	Yahoo! Messenger	12345	NetBus

CONNECTIONLESS MULTIPLEXING & DEMULTIPLEXING

- UDP socket must be specified by:
  - destination IP address
  - destination port #
- When host receives UDP segment:
  - checks destination port # in segment
  - directs UDP segment to socket with that port #

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

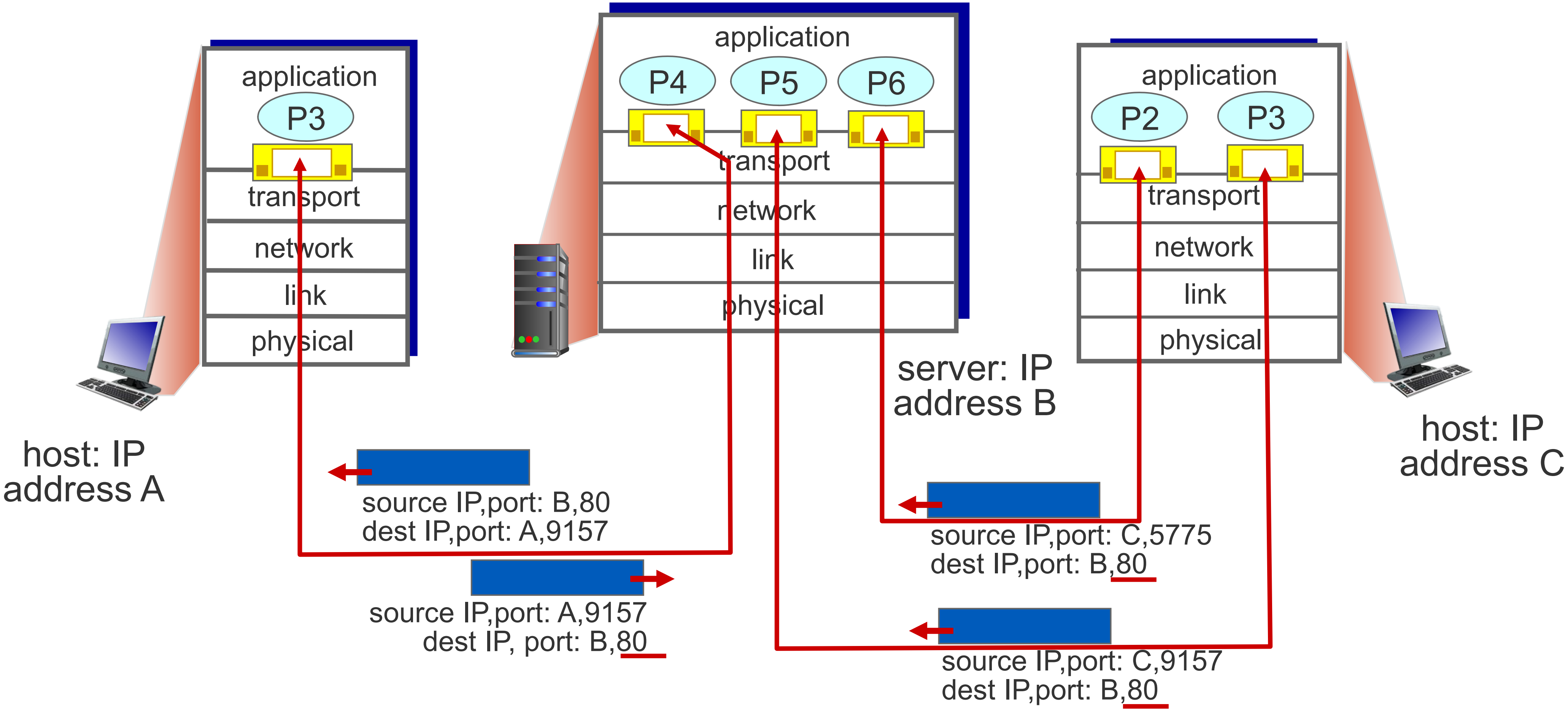


CONNECTION-ORIENTED DEMUX

- TCP socket identified by 4-tuple:
  - **source IP address**
  - **source port number**
  - **dest IP address**
  - **dest port number**
- Demux: receiver uses all four values to direct segment to appropriate socket

- Server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- Web servers have different sockets for each connecting client
  - non-persistent HTTP will have different socket for each request

CONNECTION-ORIENTED DEMUX: EXAMPLE



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

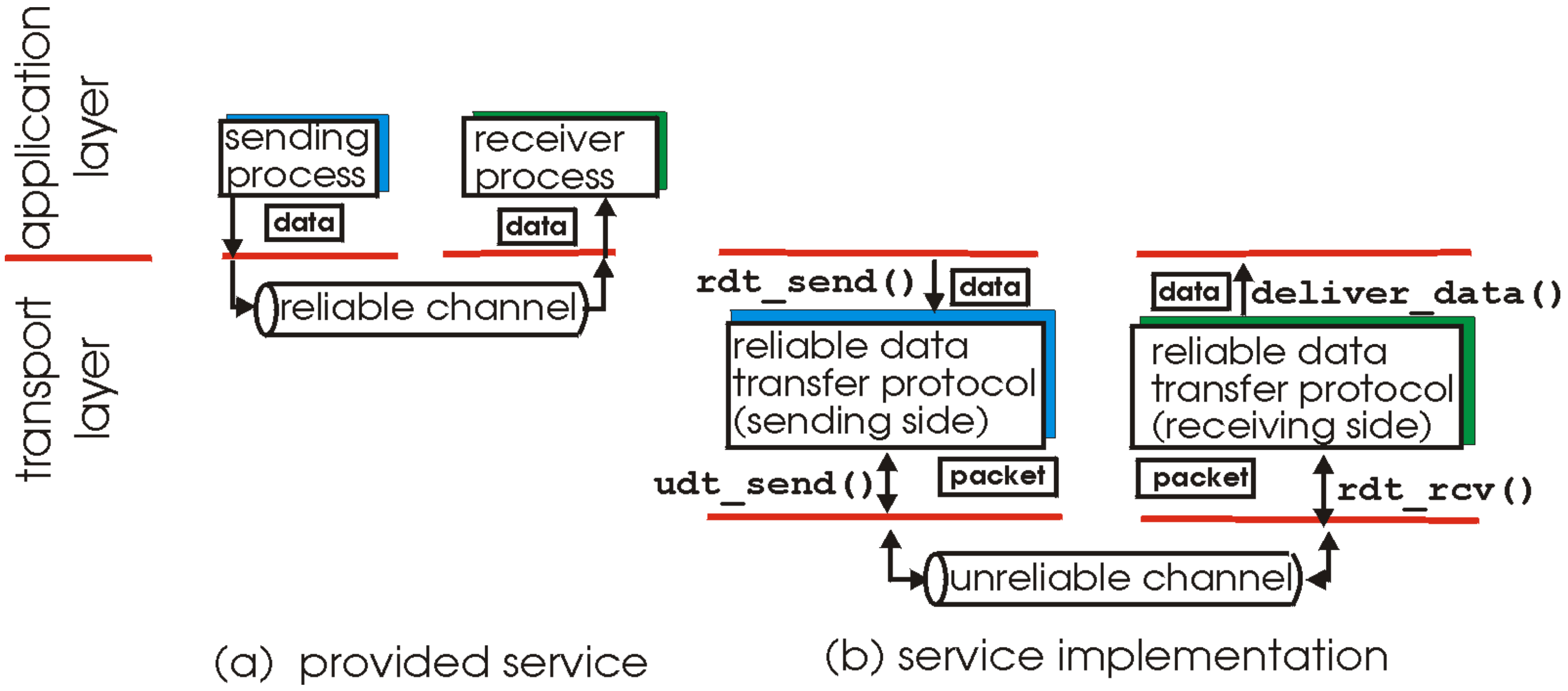
The University of Danang
University of Science and Technology

3. PRINCIPLES OF RELIABLE DATA TRANSFER

Faculty of Information Technology
PhD. Le Tran Duc

PRINCIPLES OF RELIABLE DATA TRANSFER

- Problem of implementing reliable data transfer occurs at application, transport, link layers

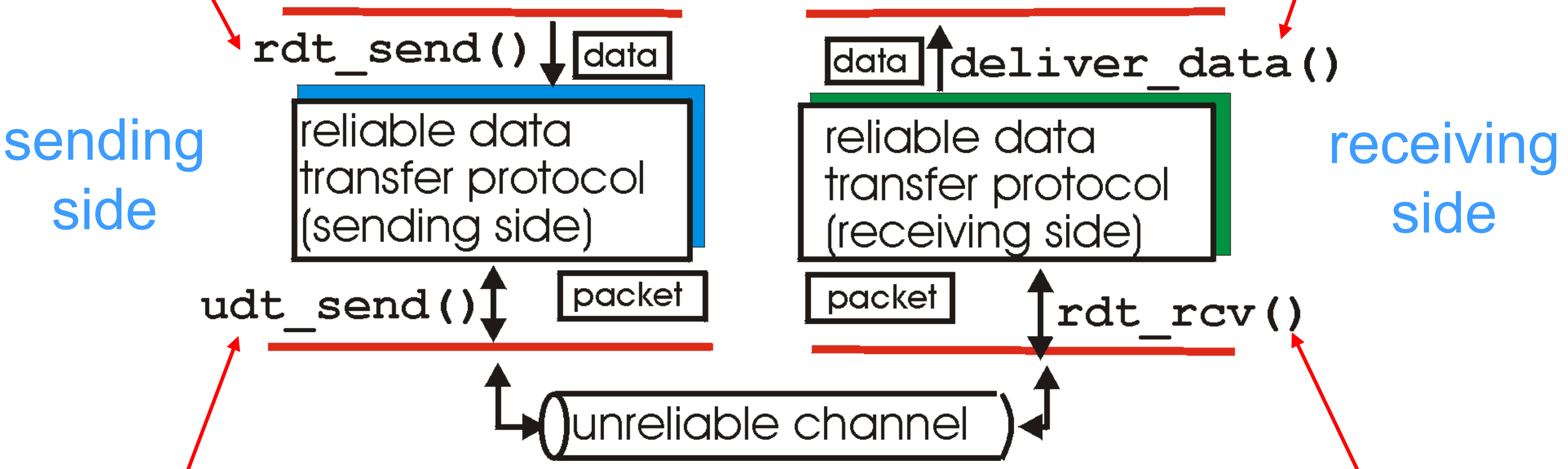


- **Difficulty**: The layer below the reliable data transfer protocol may be unreliable.
- Characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

RELIABLE DATA TRANSFER: GETTING STARTED

rdt_send(): called from above, (e.g., by app.). Pass data to be delivered to upper layer at receiver

deliver_data(): called by **rdt** to deliver data to upper



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

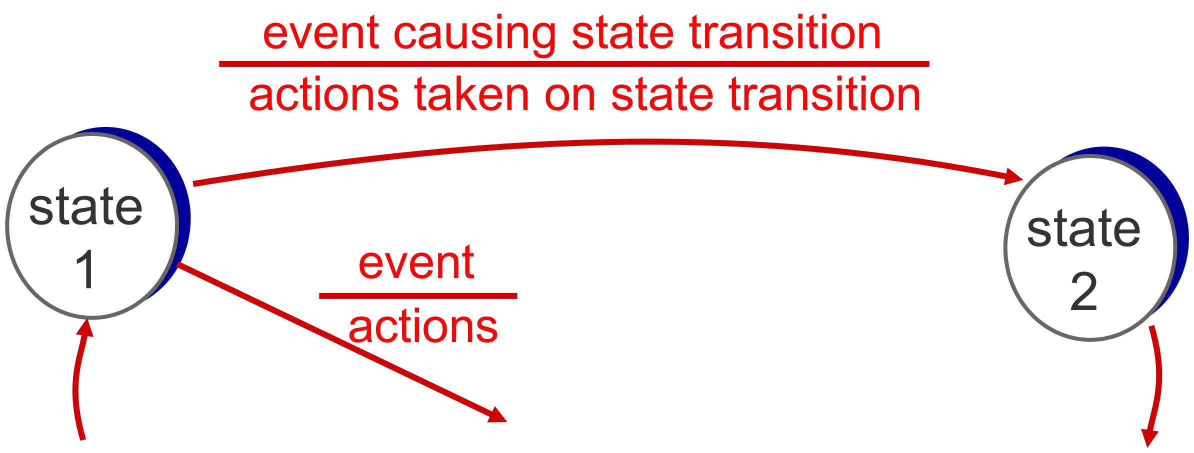
rdt_rcv(): called when packet arrives on rcv-side of channel

BUILDING A RELIABLE DATA TRANSFER PROTOCOL

We'll:

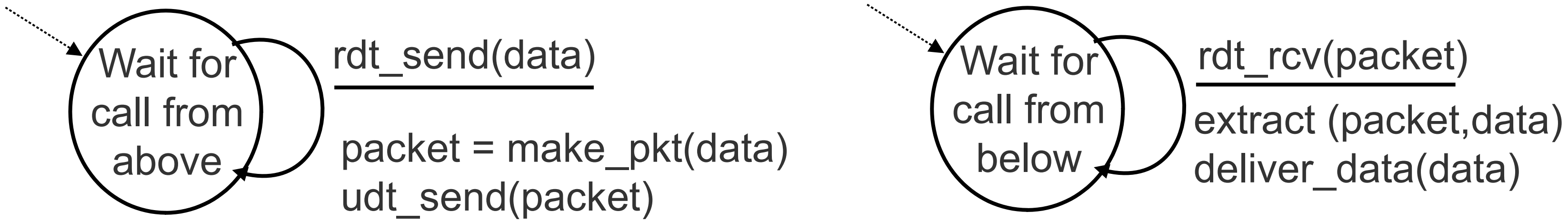
- Incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- Consider only unidirectional data transfer
    - But control info will flow on both directions!
- Use finite state machines (FSM) to specify sender, receiver

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**

- ➢ No error & loss → No need to provide any **feedback**.
- ➢ Assumption: receiving rate = sending rate → No need to ask the sender to **slow down**.

RDT2.0: CHANNEL WITH BIT ERRORS (NO LOSS)

- Underlying channel may flip bits in packet
  - checksum to detect bit errors
- *The* question: how to recover from errors?

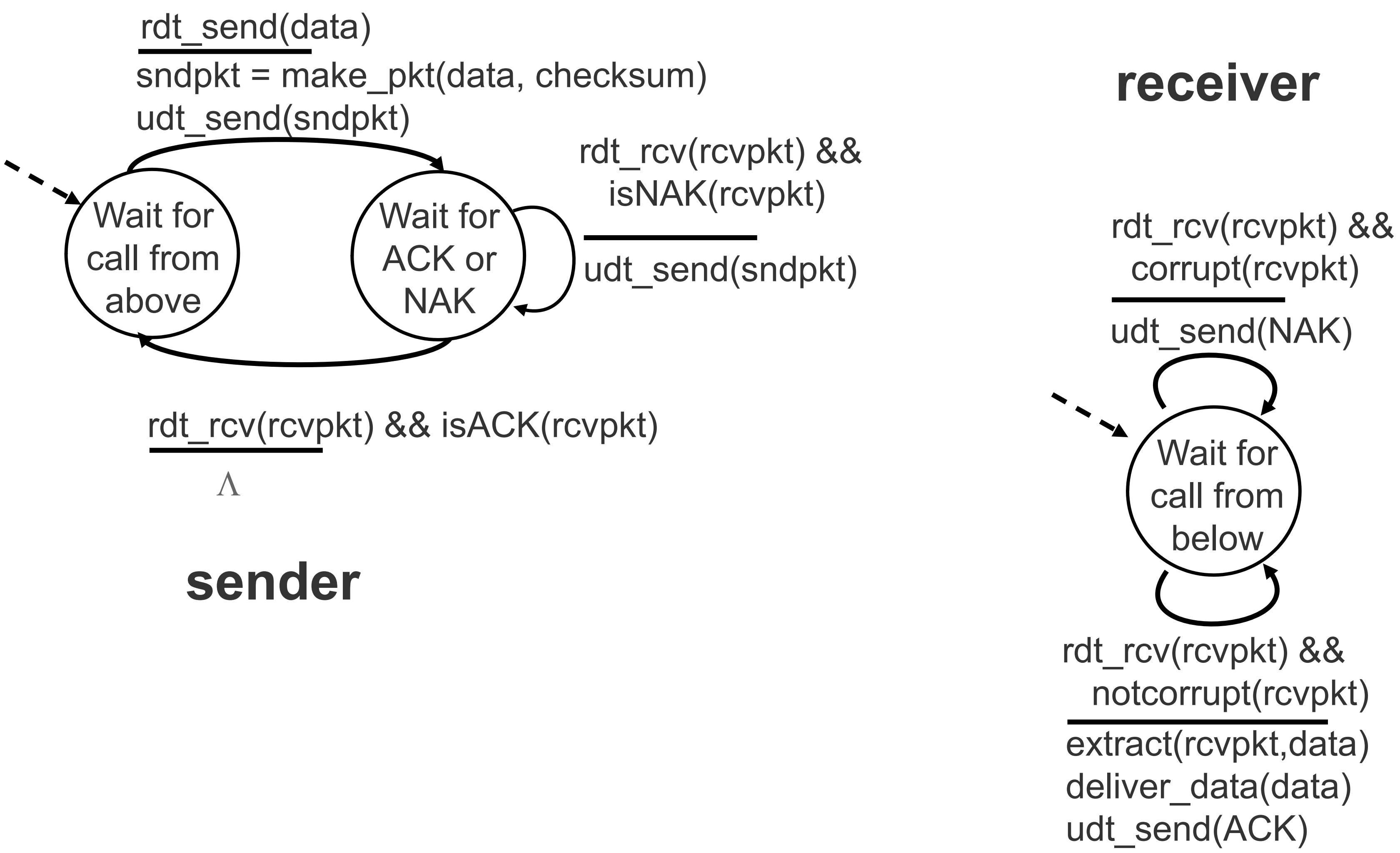
How do humans recover from “errors” during conversation?

RDT2.0: CHANNEL WITH BIT ERRORS (NO LOSS)

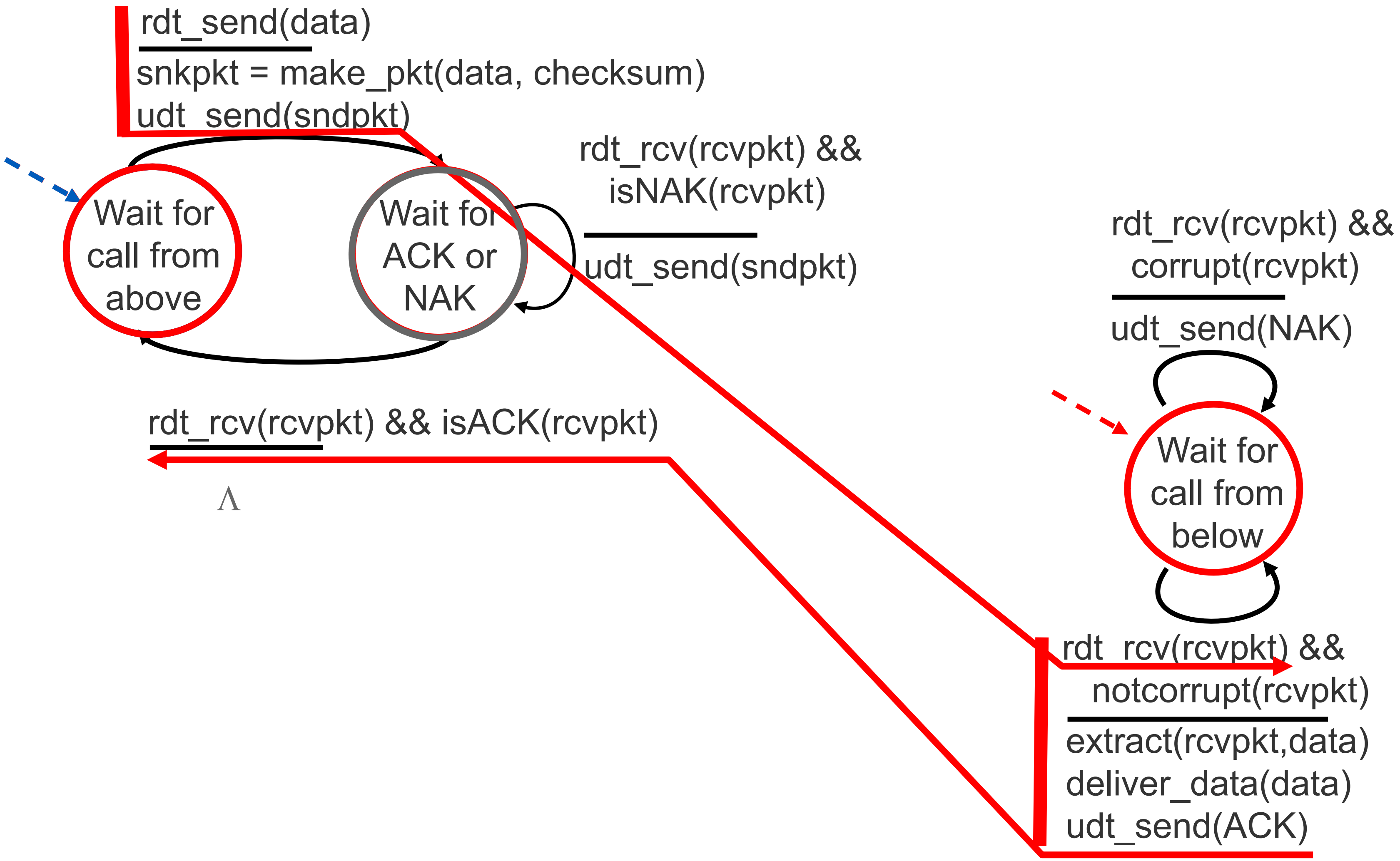
- Underlying channel may flip bits in packet
  - checksum to detect bit errors
- *The* question: how to recover from errors:
  - *(positive) acknowledgements (ACKs):* receiver explicitly tells sender that packet received OK
  - *negative acknowledgements (NAKs):* receiver explicitly tells sender that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

RDT2.0: FSM SPECIFICATION

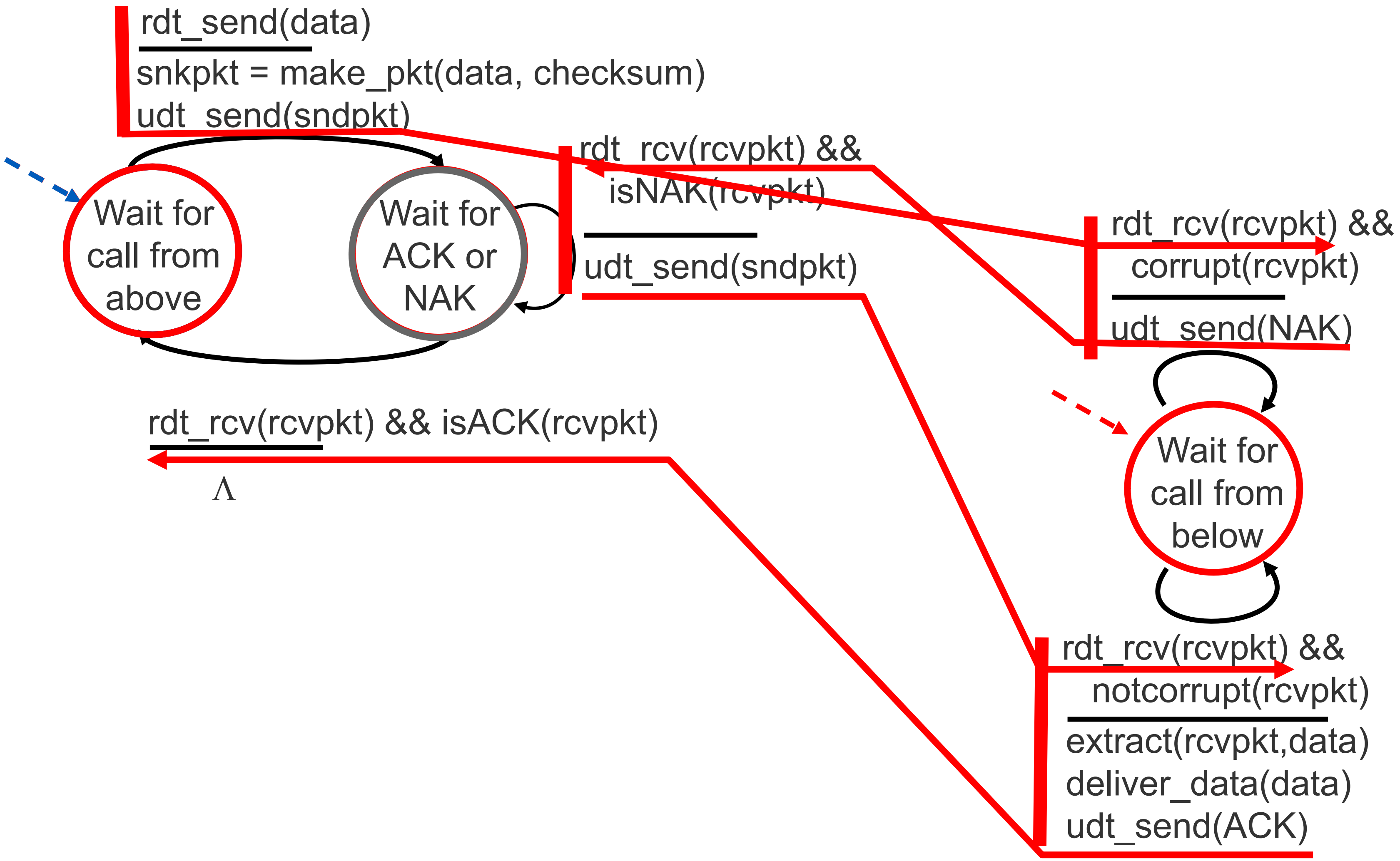
rdt2.0 protocol → employs error detection, ACK, NAK



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 → rdt2.1

- Sender adds ***sequence number*** (1 bit) to each packet; receiver adds checksum for ACK/NAK
- Sender retransmits current packet if ACK/NAK corrupted
- Receiver discards (doesn't deliver up) duplicate packet

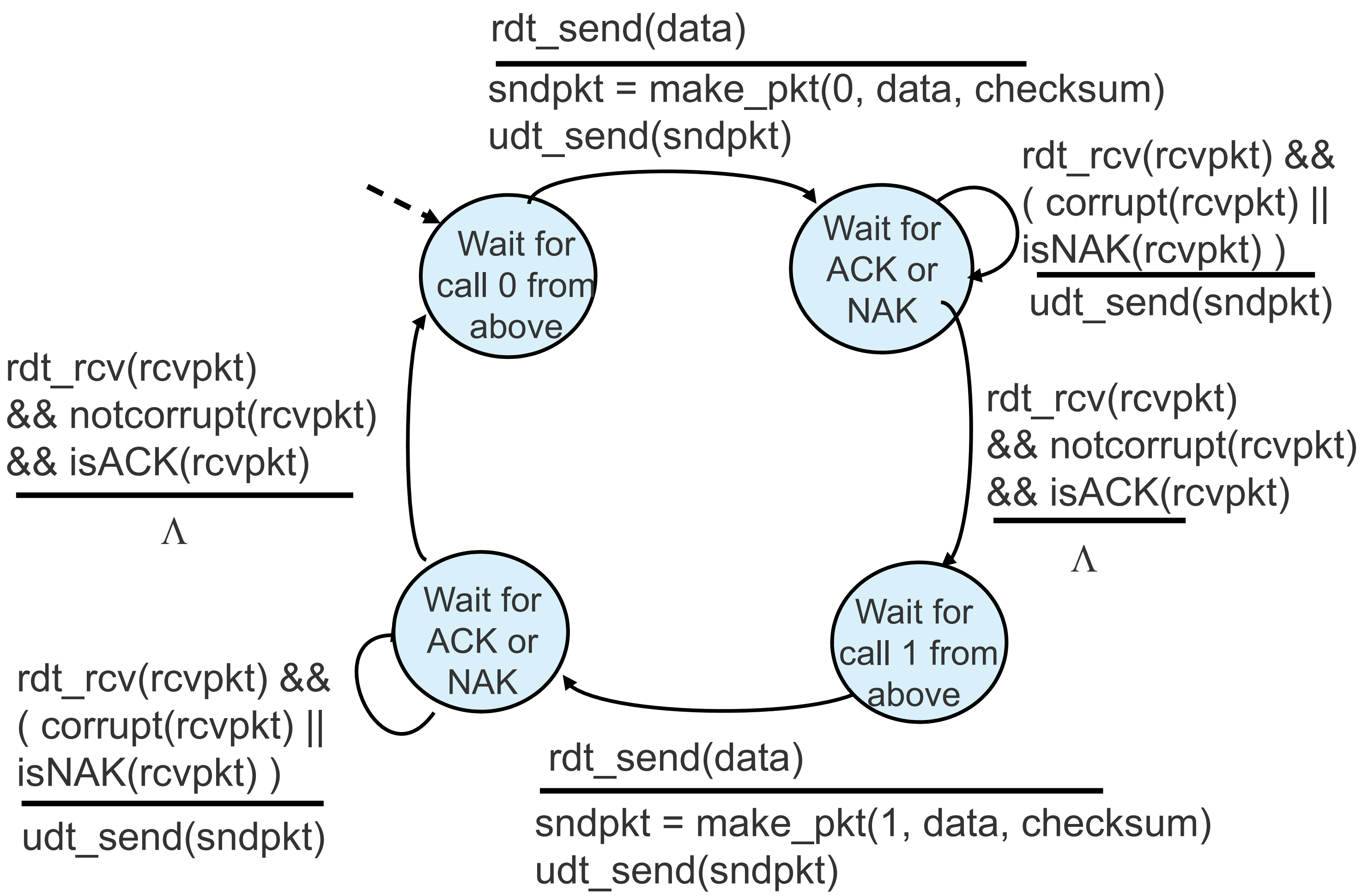
Note that: the sender cannot get more data from the upper layer in the wait-for ACK/NAK state.

→ Sender will not send new data until current packet has correctly received!!!

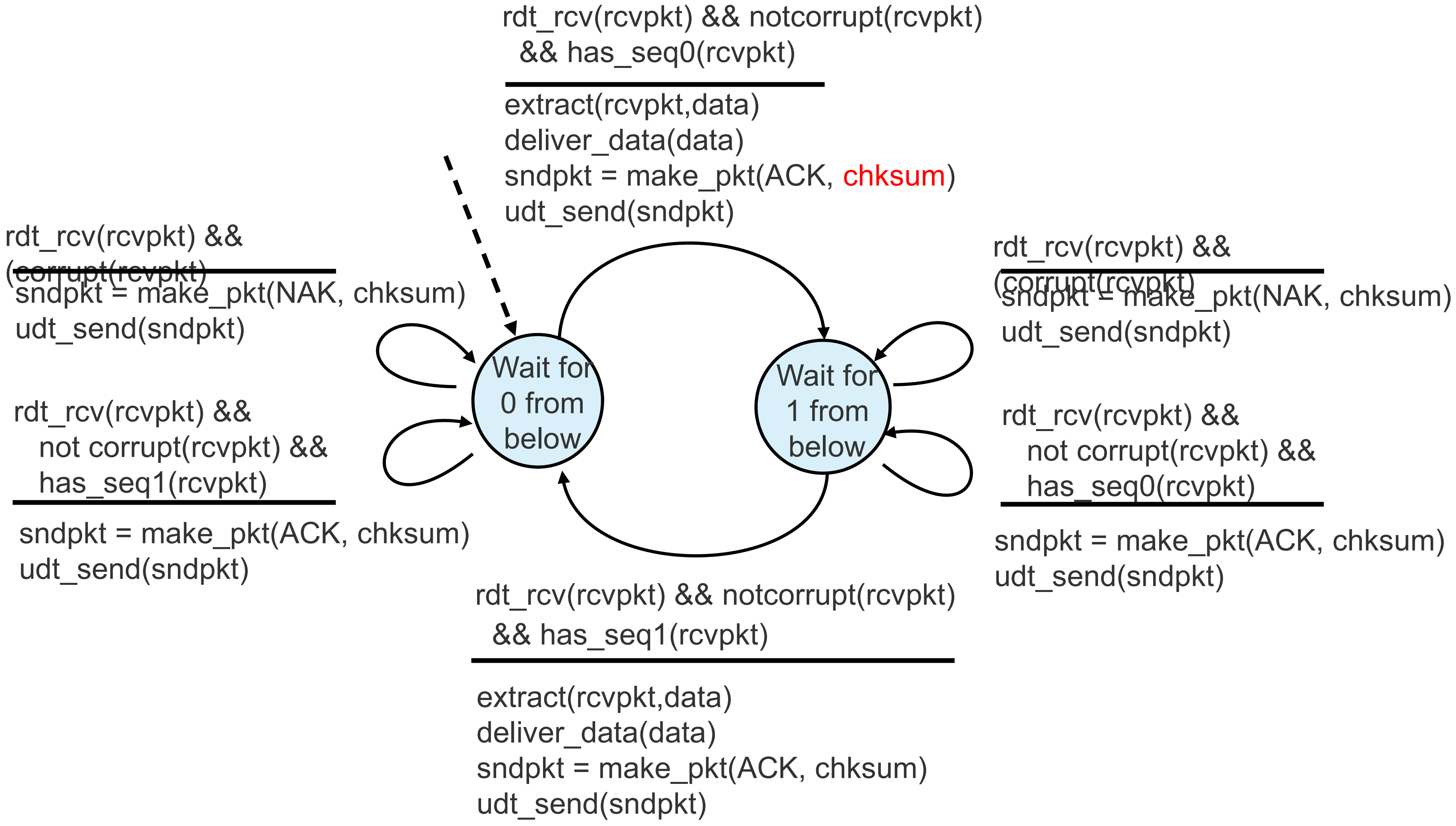
stop and wait

sender sends one packet,
then waits for receiver response

RDT2.1: SENDER HANDLES GARBLED ACK/NAKs



RDT2.1: RECEIVER HANDLES GARBLED ACK/NAKs



RDT2.1: DISCUSSION

Sender:

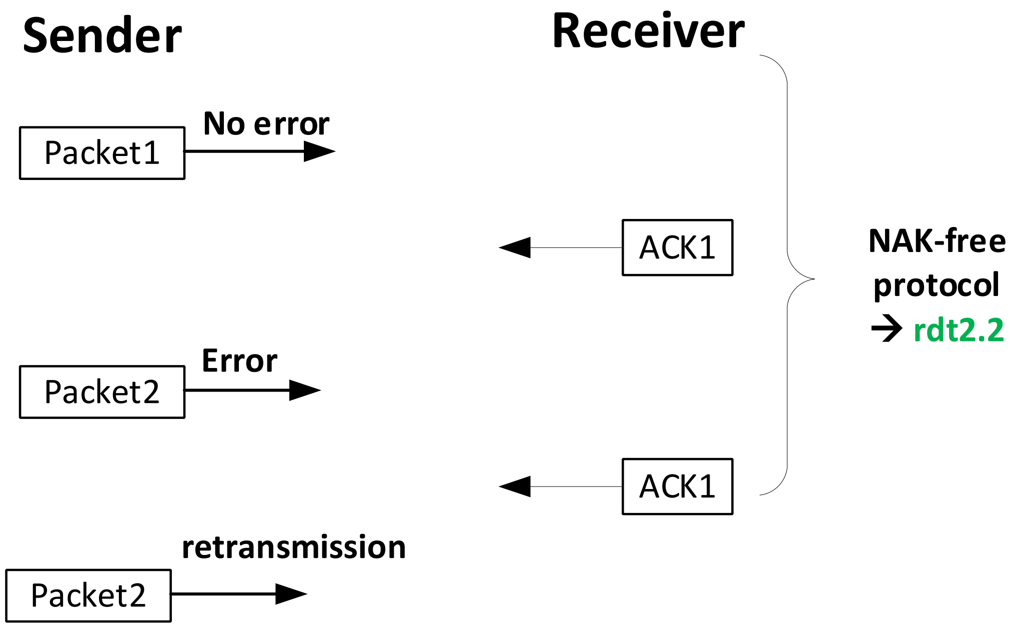
- Seq # added to pkt
- Two seq. #'s (0,1) will suffice. Why?
- Must check if received ACK/NAK corrupted
- Twice as many states
  - state must "remember" whether "expected" pkt should have seq # of 0 or 1

Receiver:

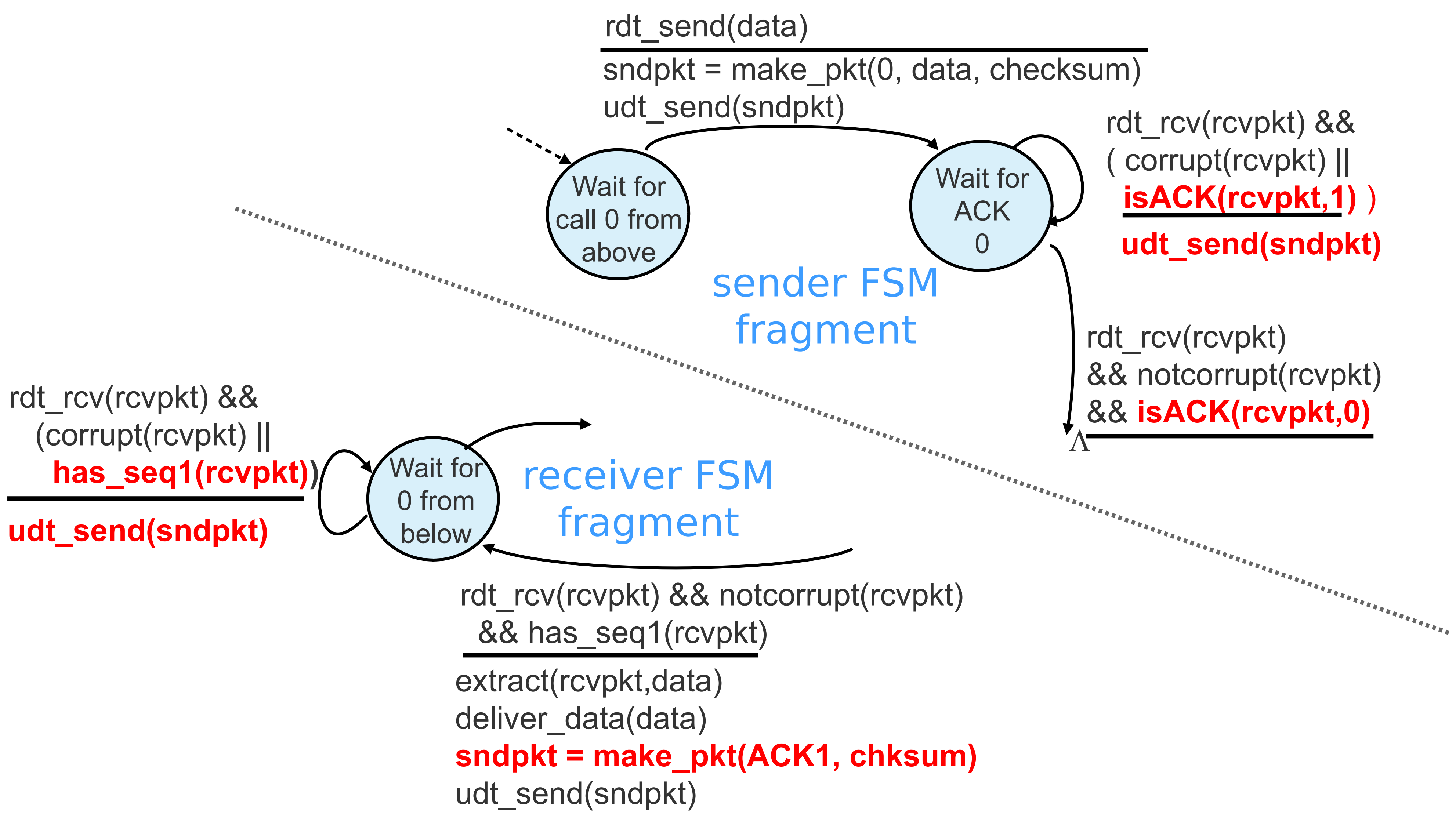
- Must check if received packet is duplicate
  - state indicates whether 0 or 1 is expected pkt seq #
- **Note**: receiver can *not* know if its last ACK/NAK received OK at sender

RDT2.2: A NAK-FREE PROTOCOL

- Same functionality as rdt2.1, using ACKs only
- Instead of NAK, receiver sends ACK for last packet 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: CHANNEL WITH ERRORS AND LOSS

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

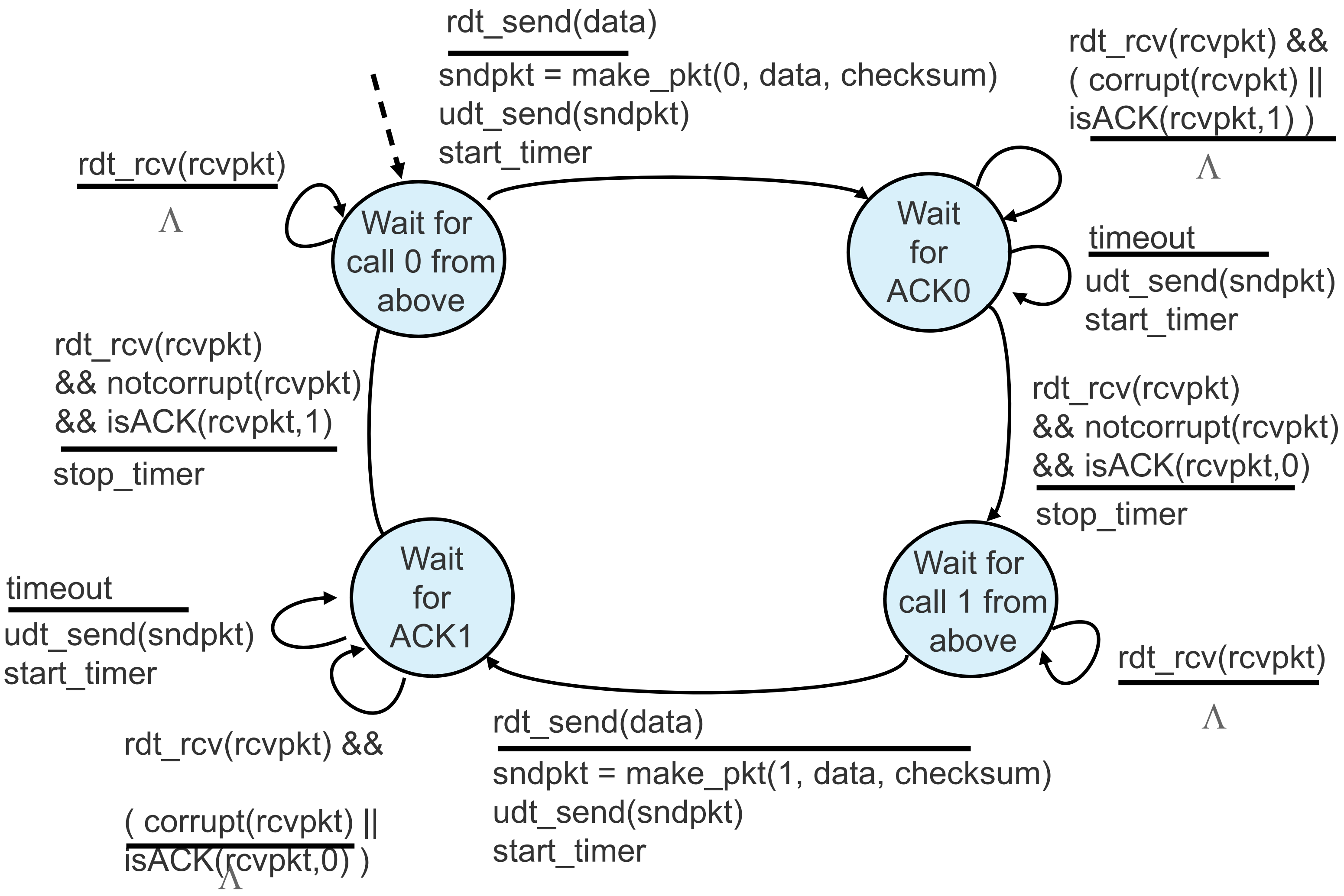
→ *How to detect packet loss & what to do when packet loss occurs?*

- checksum, seq. #, ACKs, retransmissions will answer the latter concern … but **not enough**

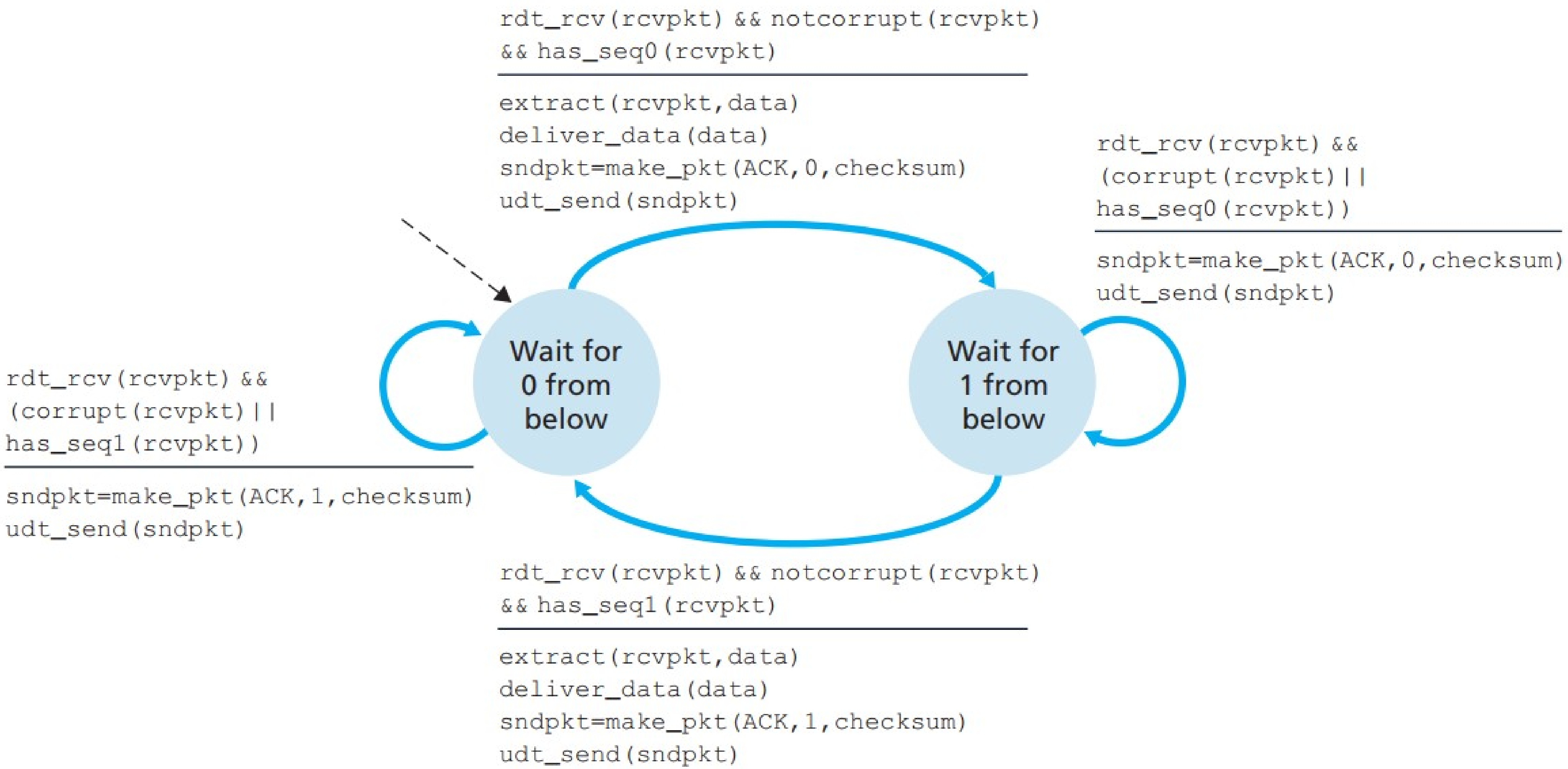
Approach: sender waits "**reasonable**" amount of time for ACK

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

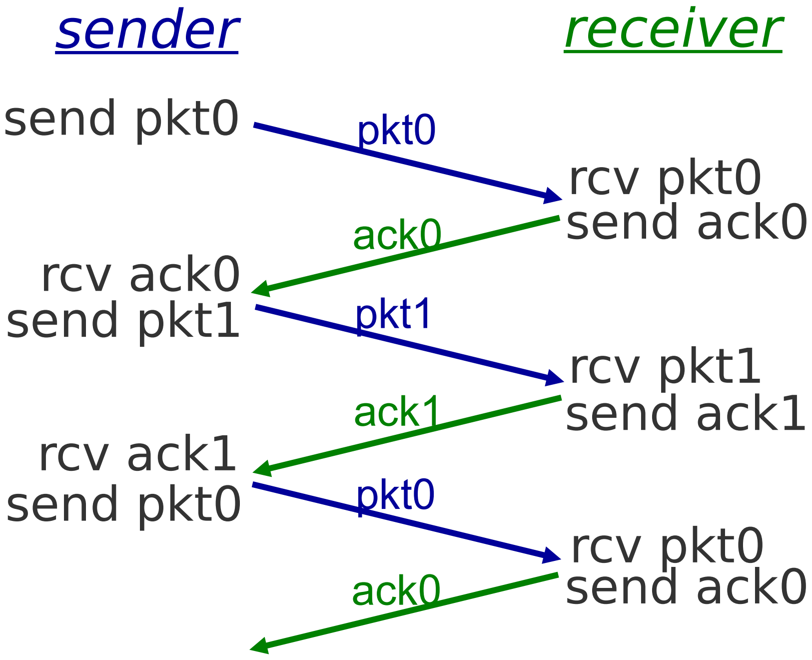
RDT3.0 SENDER



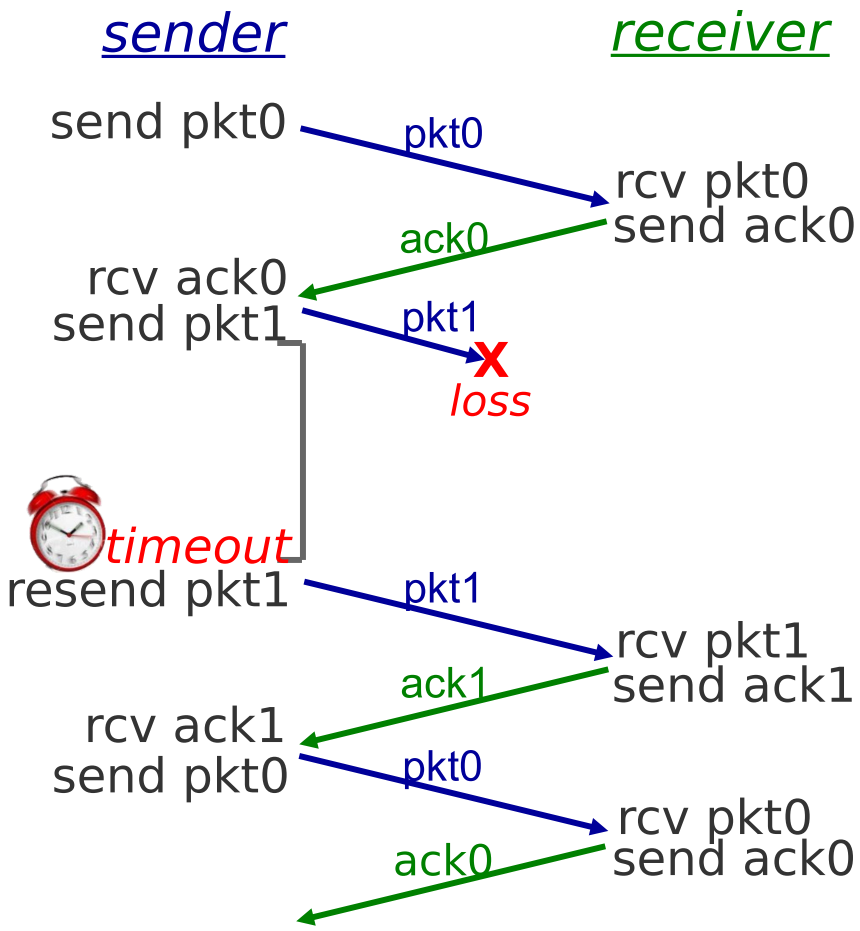
RDT3.0 RECEIVER (Similar to RDT2.2 RECEIVER)



RDT3.0 IN ACTION

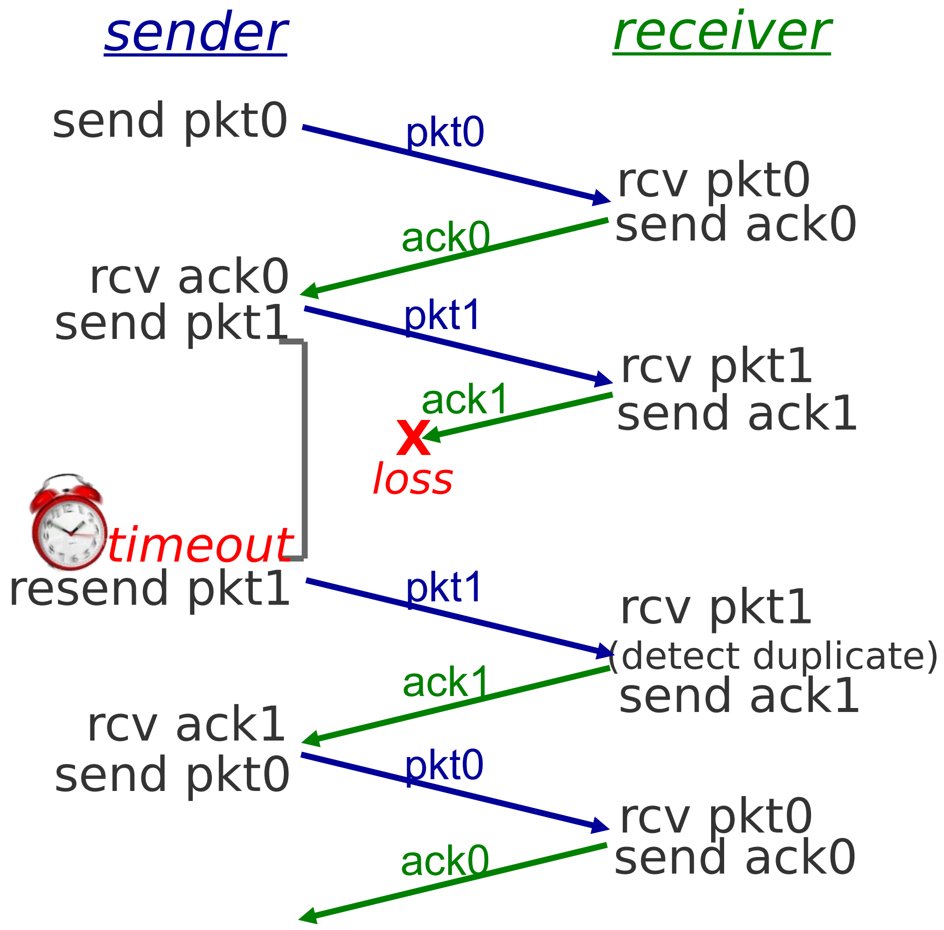


(a) no loss

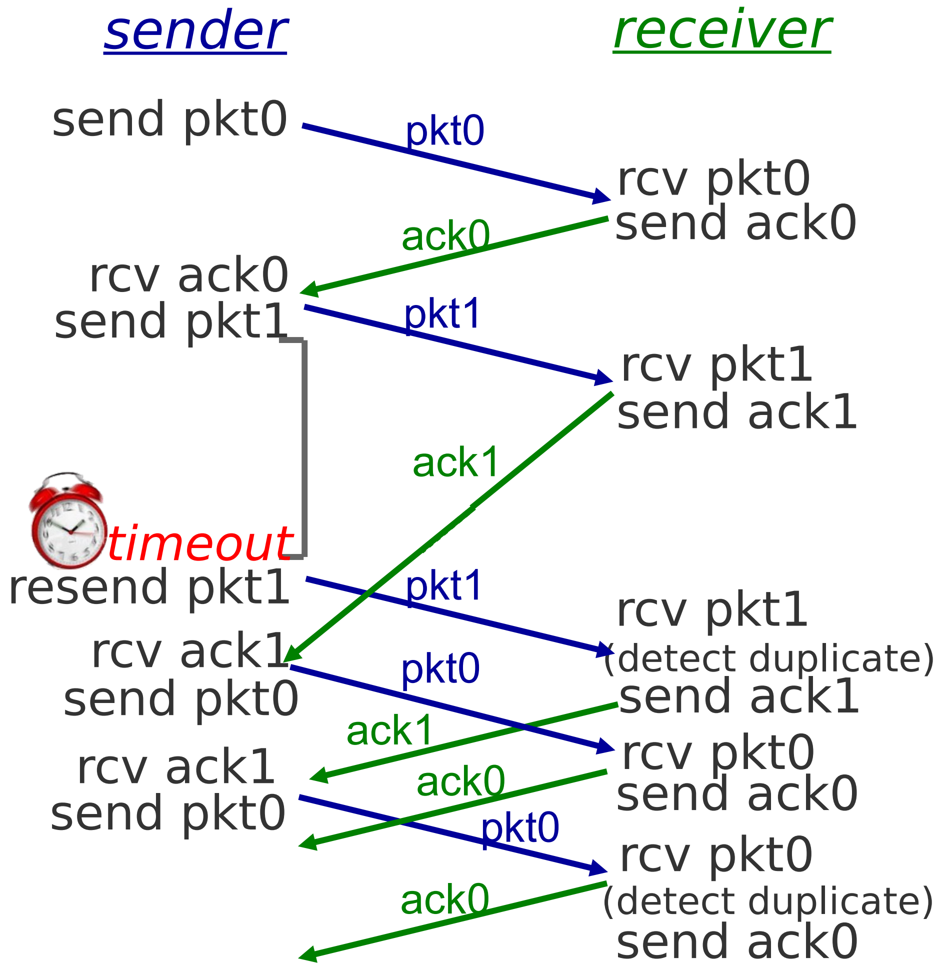


(b) packet loss

RDT3.0 IN ACTION



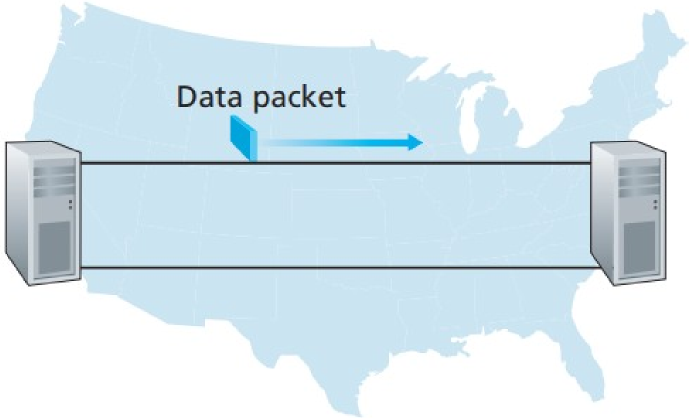
(c) ACK loss



(d) premature timeout/ delayed ACK

SUPPLEMENT 1: PERFORMANCE OF RDT3.0 (See at home)

- rdt3.0 is correct, but low performance ← because it also is stop-and-wait protocol
- e.g.: 1 Gbps link, 30 ms RTT prop. delay, packet size 8000 bits:



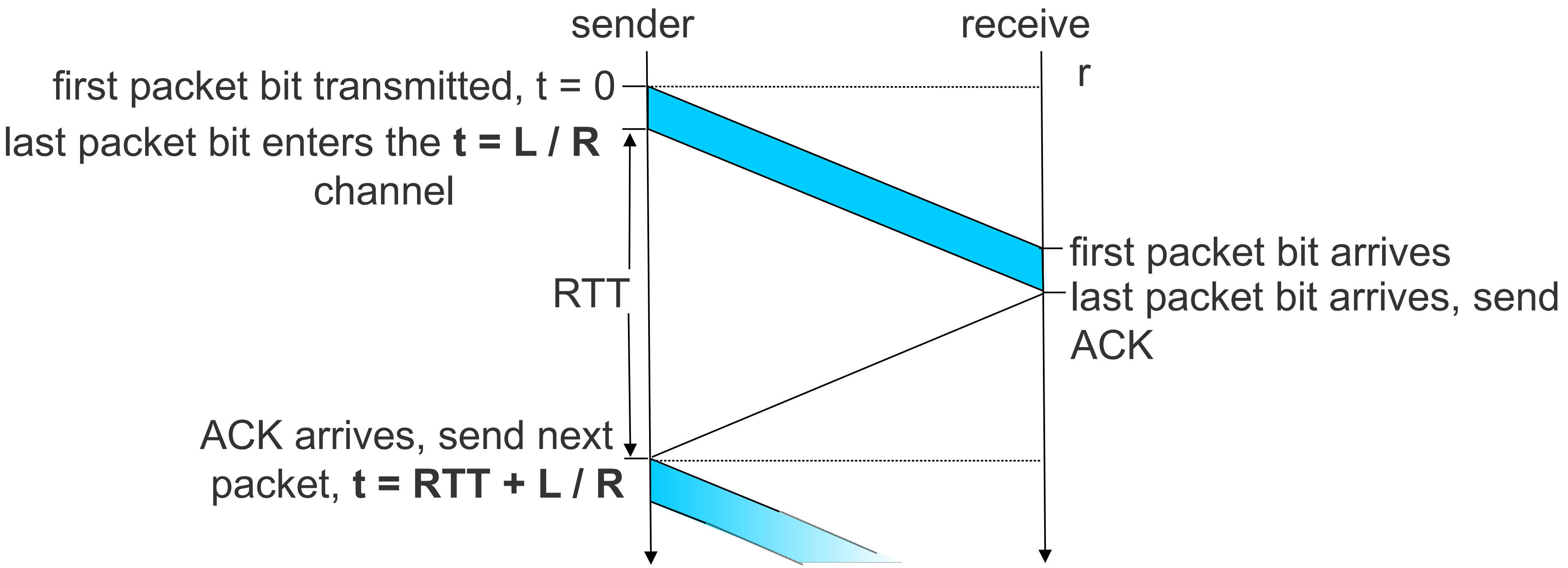
a. A stop-and-wait protocol in operation

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

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

- If RTT=30 msec, 1KB packet every 30 msec: 33kB/sec throughput over **1**Gbps link
  - → in 30.008 msec, the sender was sending for only 0.008 msec
- Network protocol limits use of physical resources!

SUPPLEMENT 2: RDT3.0 – STOP-AND-WAIT OPERATION (See at home)

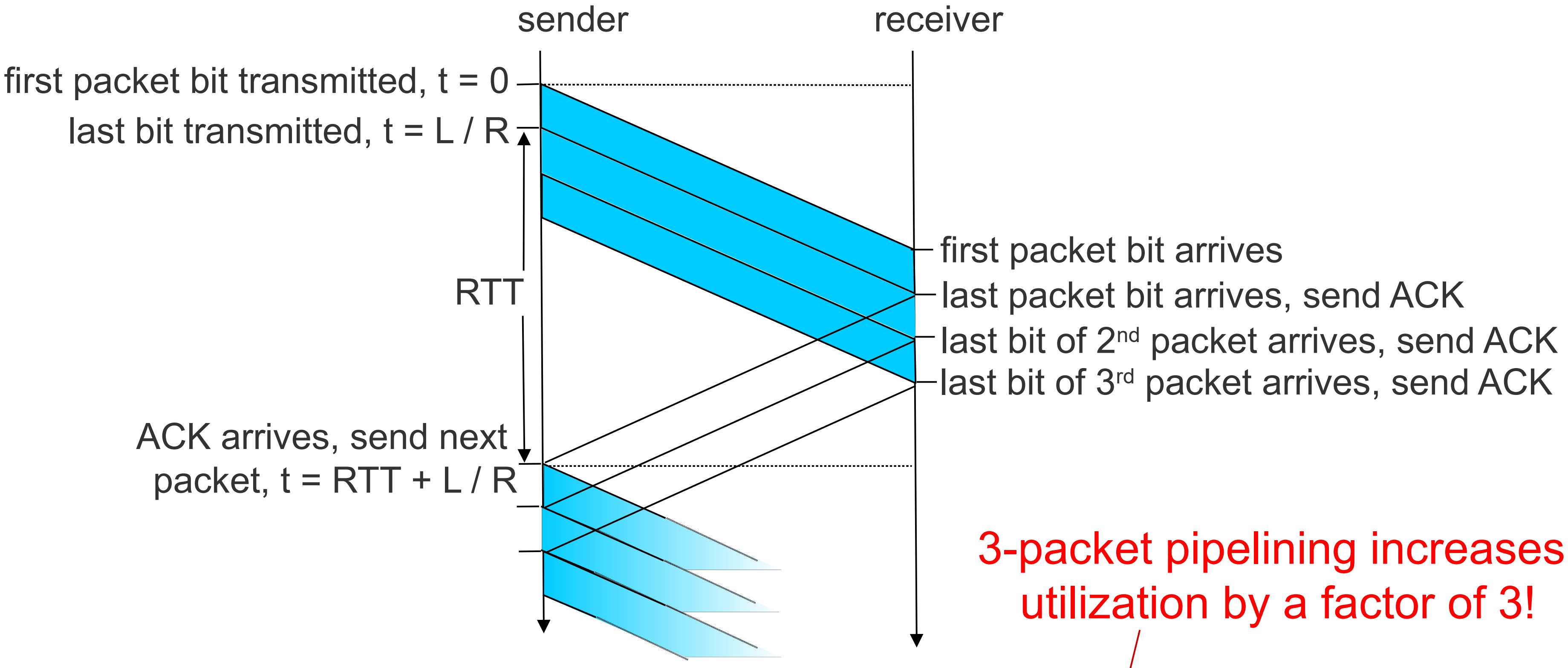


Sender utilization:

The sender was able to send only 1,000 bytes in 30.008 milliseconds, an effective throughput of only 267 kbps—even though a 1 Gbps link was available!!!

→ **SOLUTION: send multiple packets without waiting for acknowledgments**

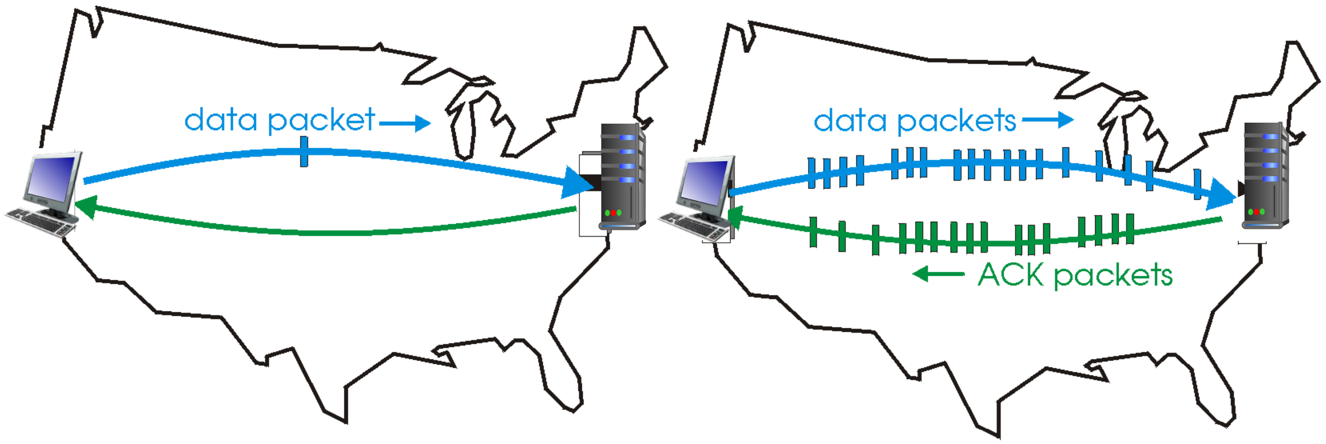
SUPPLEMENT 3: PIPELINING INCREASED UTILIZATION (See at home)



3-packet pipelining increases utilization by a factor of 3!

PIPELINED PROTOCOLS

- **Pipelining**: sender allows multiple, “in-flight”, yet-to-be-acknowledged packets
  - Range of sequence numbers must be increased
  - Buffering at sender and/or receiver



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

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

PIPELINED PROTOCOLS: OVERVIEW

Go-back-N:

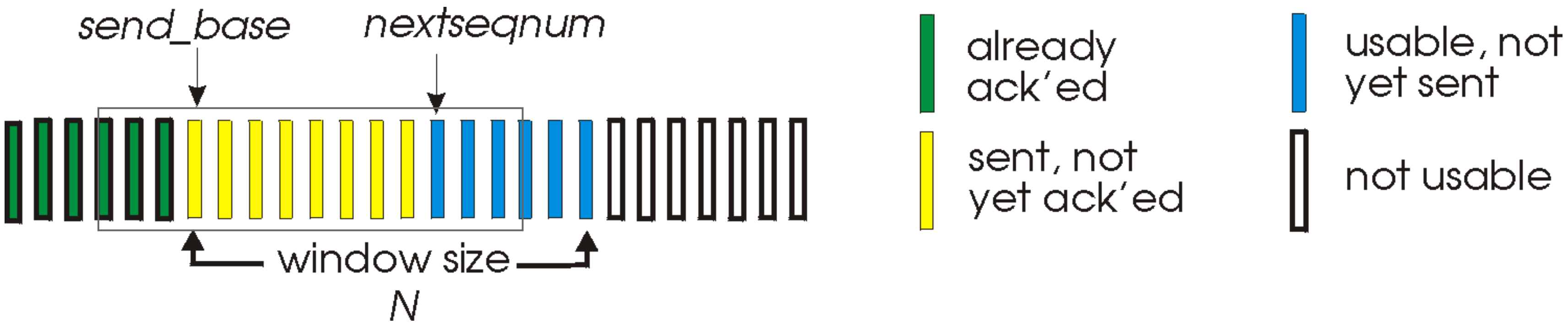
- Sender can have up to N **unACKed** packets in pipeline
- Receiver only sends ***cumulative*** *ACK*
  - Doesn't ACK packet if there's a gap
- Sender has timer for **oldest unACKed** packet
  - When timer expires, retransmit ***all*** **unACKed** packets

Selective Repeat:

- Sender can have up to N **unACKed** 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

GO-BACK-N: SENDER

- k-bit seq # in packet header
- "**window**" of up to N, consecutive unack'ed packets allowed
- 4 intervals in the range of sequence numbers can be identified:
  - **[0, *sent_base-1*], [*sent_base*, *nextseqnum-1*], [*nextseqnum*, *sent_base+N* -1], ≥ *sent_base+N***



- ACK(n): ACKs all packets up to, including seq # n - *"cumulative ACK"*
  - May receive duplicate ACKs (see receiver)
- Timer for oldest in-flight packet
- ***Timeout(n)***: retransmit packet n and all higher seq # pkts in window

SUPPLEMENT 4: GBN - SENDER EXTENDED FSM

rdt_send(data)

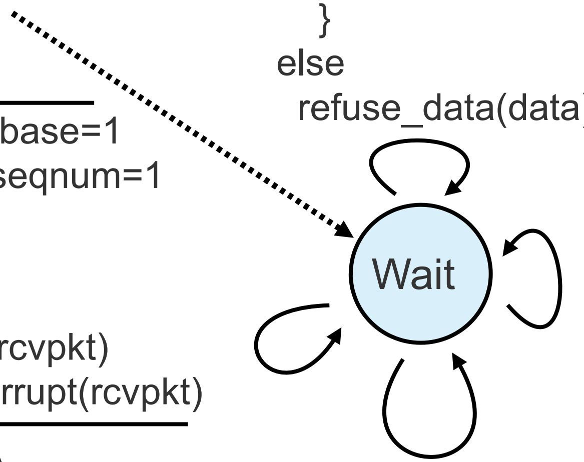
---

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

Λ

---

sendbase=1
nextseqnum=1



timeout

---

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

rdt_rcv(rcvpkt)
  && corrupt(rcvpkt)

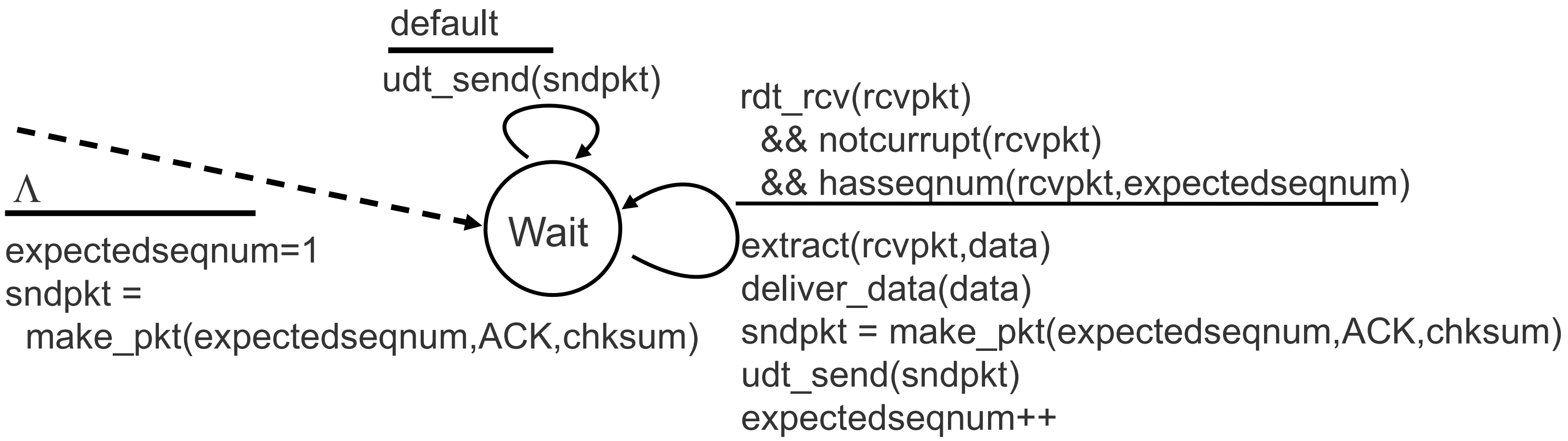
Λ

rdt_rcv(rcvpkt) &&
  notcorrupt(rcvpkt)

---

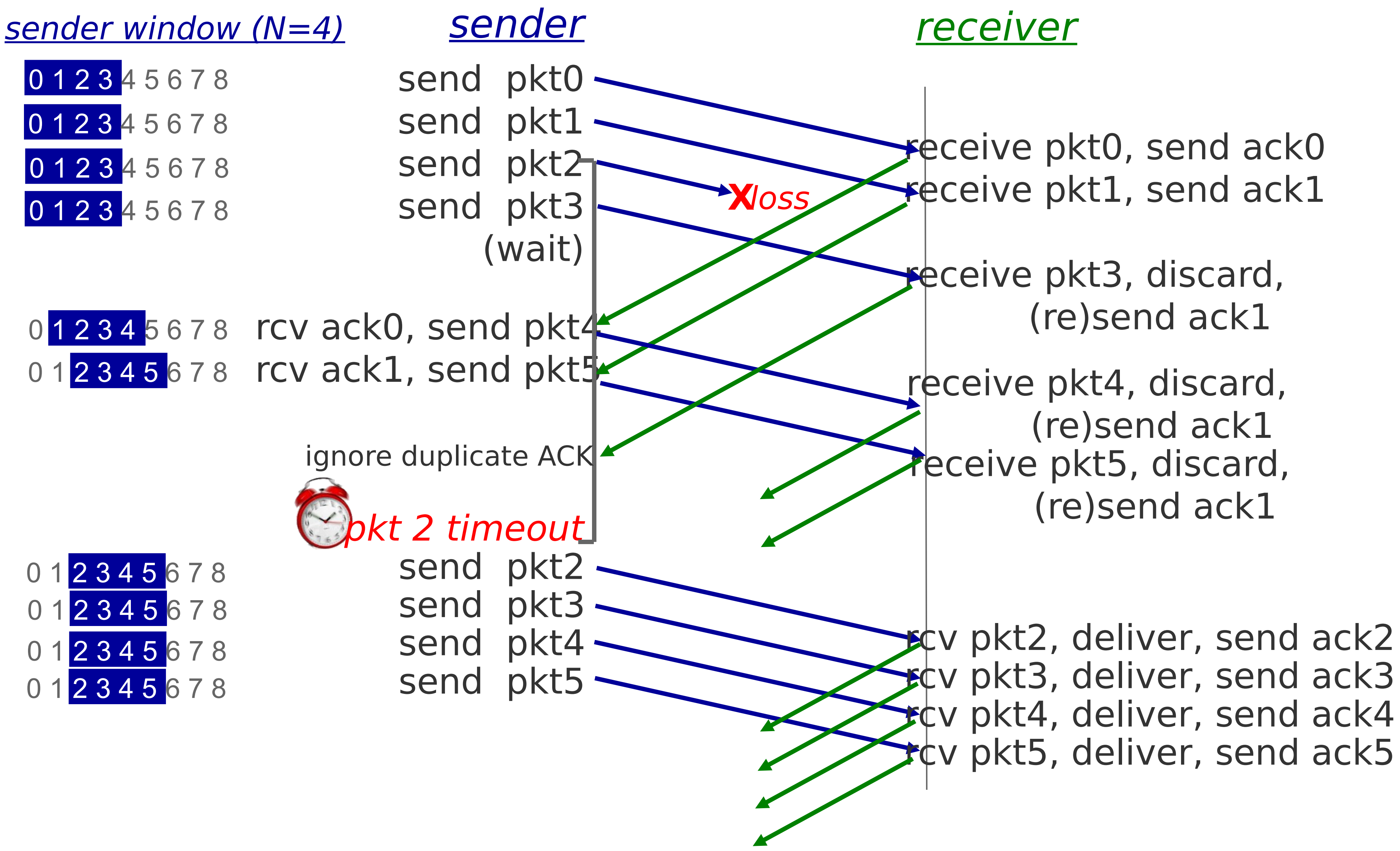
sendbase = getacknum(rcvpkt)+1
If (sendbase == nextseqnum)
    stop_timer
else
    start_timer

SUPPLEMENT 5: GBN - RECEIVER EXTENDED FSM



- ACK-only: always send ACK for correctly-received packet with **highest** *in-order* seq #
  - may generate duplicate ACKs
  - need only remember **`expectedseqnum`**
- Out-of-order packet:
  - discard (don’t buffer): ***no receiver buffering!***
  - re-ACK packet with highest in-order seq #

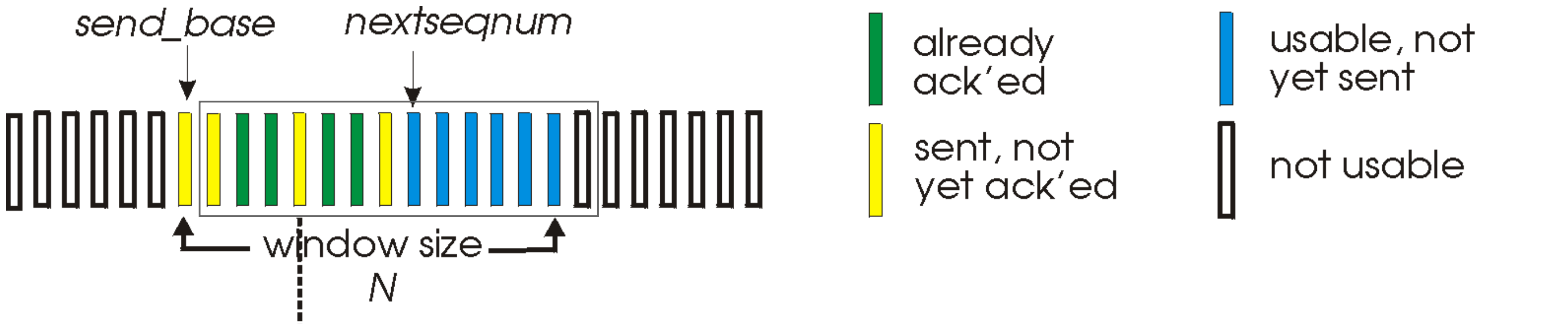
GBN IN ACTION



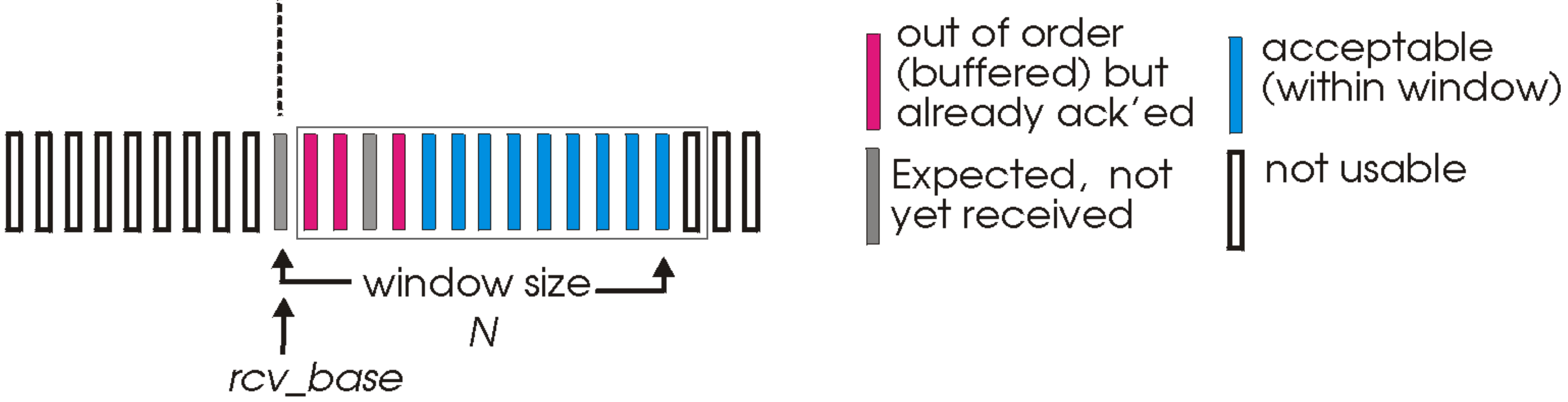
SELECTIVE REPEAT

- Receiver *individually* acknowledges all correctly received packets
  - Buffers packets, as needed, for eventual in-order delivery to upper layer
- Sender only resends packets for which ACK not received
  - Sender timer for each unACKed packet
- Sender window
  - *N* consecutive seq #'s
  - Limits seq #s of sent, unACKed packets

SELECTIVE REPEAT: SENDER, RECEIVER WINDOWS



(a) sender view of sequence numbers



(b) receiver view of sequence numbers

SELECTIVE REPEAT

Sender

When data received from above:

- If next available seq # is within the sender's window, send packet

Timeout(n):

- Each packet must now have its own logical timer
- Resend packet n, restart timer

ACK(n) received in [sendbase, sendbase+N]:

- Mark packet n as received
- If packet sequence number = send_base → the window base is moved forward to the unACKed packet with the smallest sequence number

Receiver

Packet n with seq# in [rcvbase, rcvbase+N-1]

- Out-of-order: buffer
- In-order: deliver ACK (also deliver buffered, in-order packets), advance window to next not-yet-received packet

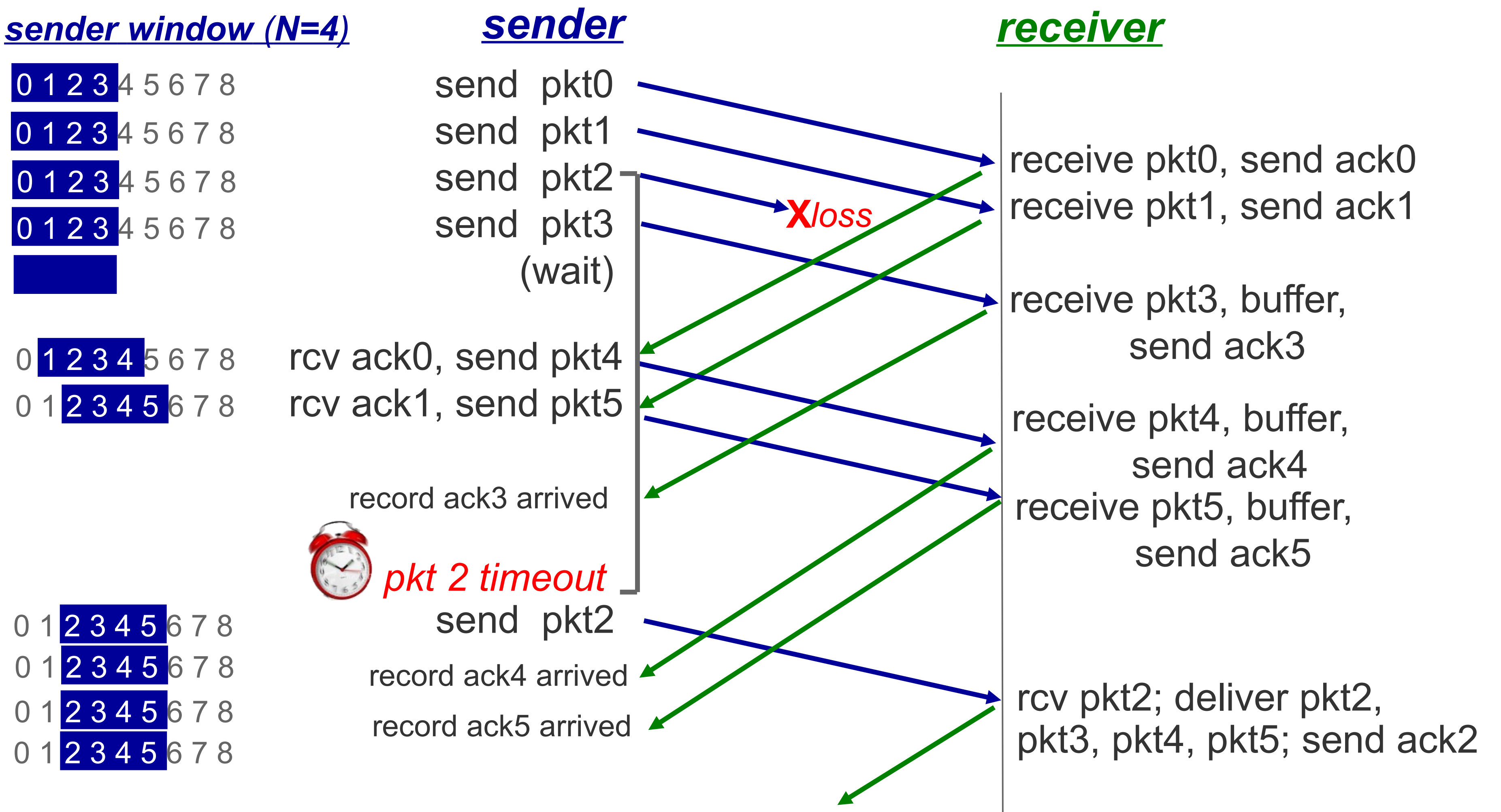
Packet n with seq# in [rcvbase-N, rcvbase-1]

- ACK(n)

Otherwise:

- Ignore

SELECTIVE REPEAT IN ACTION



Q: what happens when ack2 arrives?

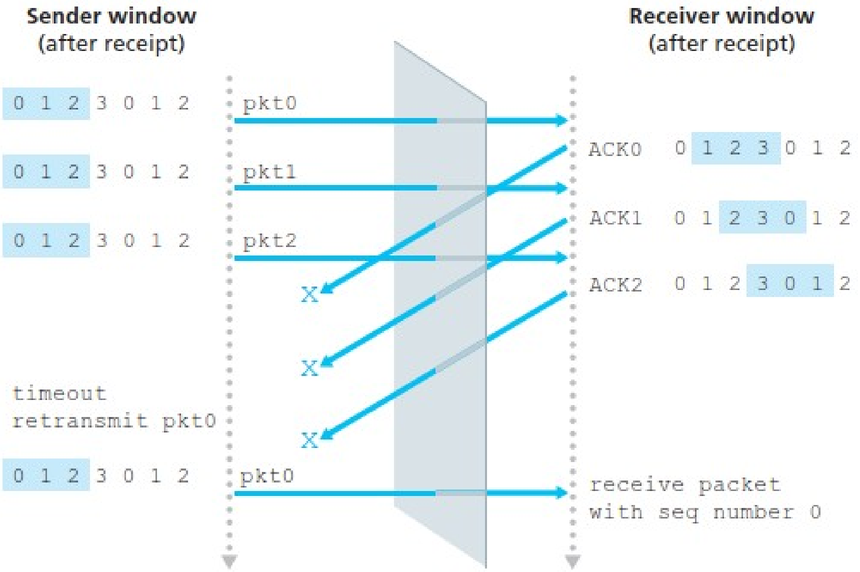
SELECTIVE REPEAT DILEMMA

Example:

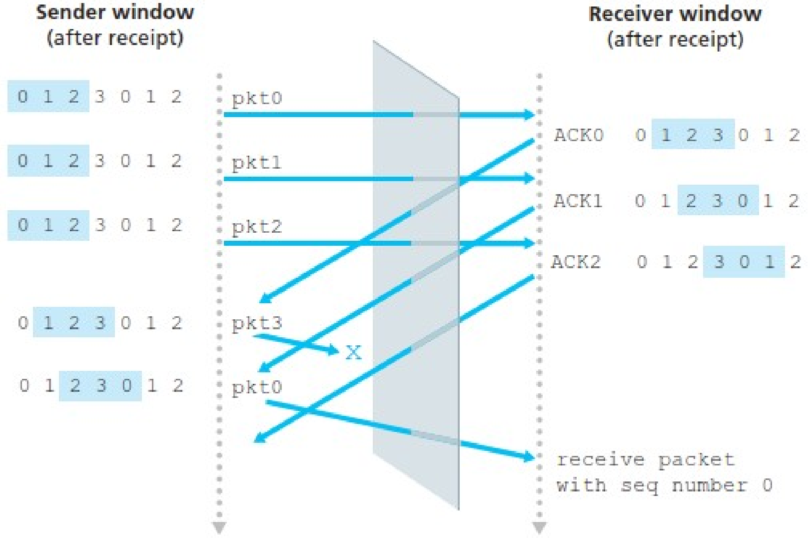
- seq #'s: 0, 1, 2, 3
- window size=3

- Receiver sees no difference in two scenarios!
- Duplicate data accepted as new in (b)

Q: what relationship between seq # size and window size to avoid problem in (b)?



a) No problem



a) oops!

4. UDP – CONNECTIONLESS TRANSPORT

IT FACULTY
Faculty of Information Technology
PhD. Le Tran Duc

UDP: USER DATAGRAM 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

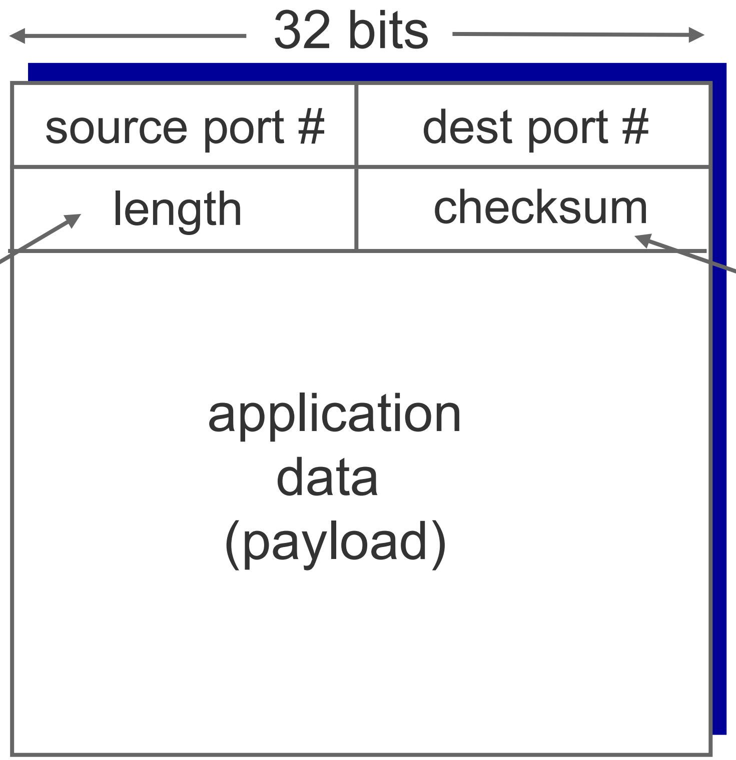
- UDP use:
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
  - SNMP
- Reliable transfer over UDP:
  - add reliability at application layer
  - application-specific error recovery!

WHY CHOSE UDP INSTEAD TCP?

- Real-time applications require minimum delay
- No connection establishment
  - No delay for connection establishment
- No connection state
  - Support more active client
- Small packet header overhead
  - TCP: 20 bytes/segment
  - UDP: 8 bytes/segment

Application	Application-Layer Protocol	Underlying Transport Protocol
Electronic mail	SMTP	TCP
Remote terminal access	Telnet	TCP
Web	HTTP	TCP
File transfer	FTP	TCP
Remote file server	NFS	Typically UDP
Streaming multimedia	typically proprietary	UDP or TCP
Internet telephony	typically proprietary	UDP or TCP
Network management	SNMP	Typically UDP
Name translation	DNS	Typically UDP

UDP SEGMENT STRUCTURE



UDP segment format

length, in bytes of UDP segment, including header

For error detection
→ Receiving host checks whether bits within the UDP segment have been altered

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 ….

UDP provides error checking → but does not do anything to recover from an error

INTERNET CHECKSUM: EXAMPLE

Example: add two 16-bit integers

	1	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
wraparound	1	0	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
sum	1	0	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum	1	1	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

The University of Danang
University of Science and Technology

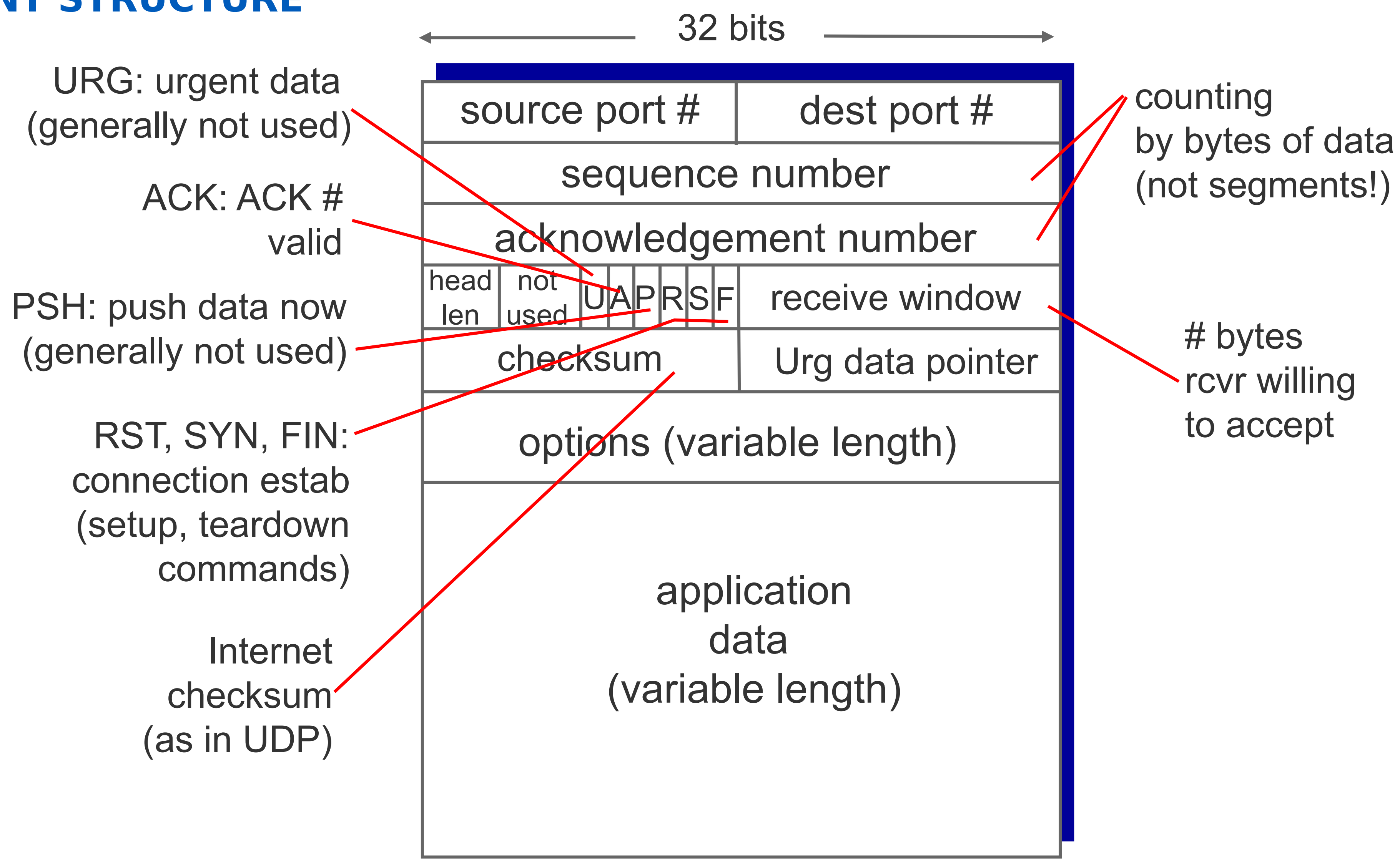
5. TCP – CONNECTION-ORIENTED TRANSPORT

Faculty of Information Technology
PhD. Le Tran Duc

TCP: TRANSMISSION CONTROL PROTOCOL

- 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

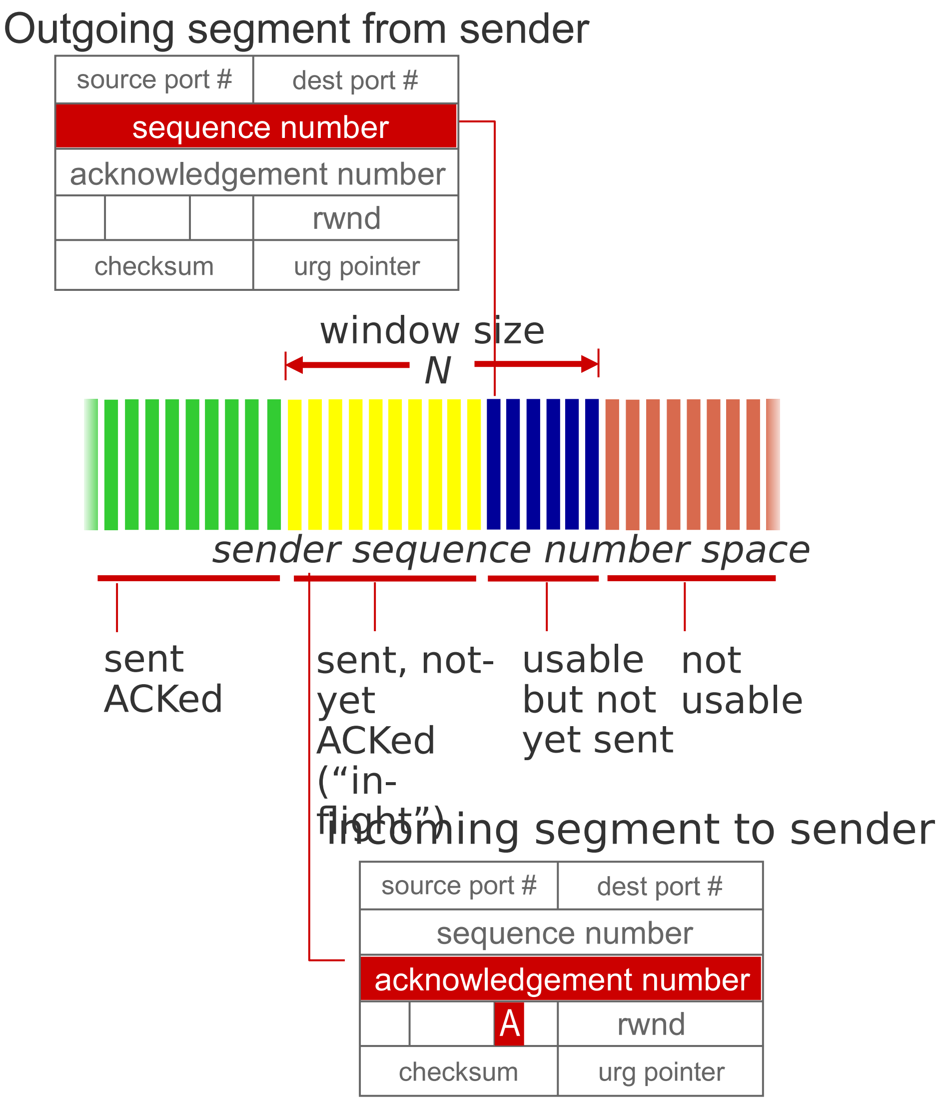


TCP SEQUENCE NUMBERS, ACKs

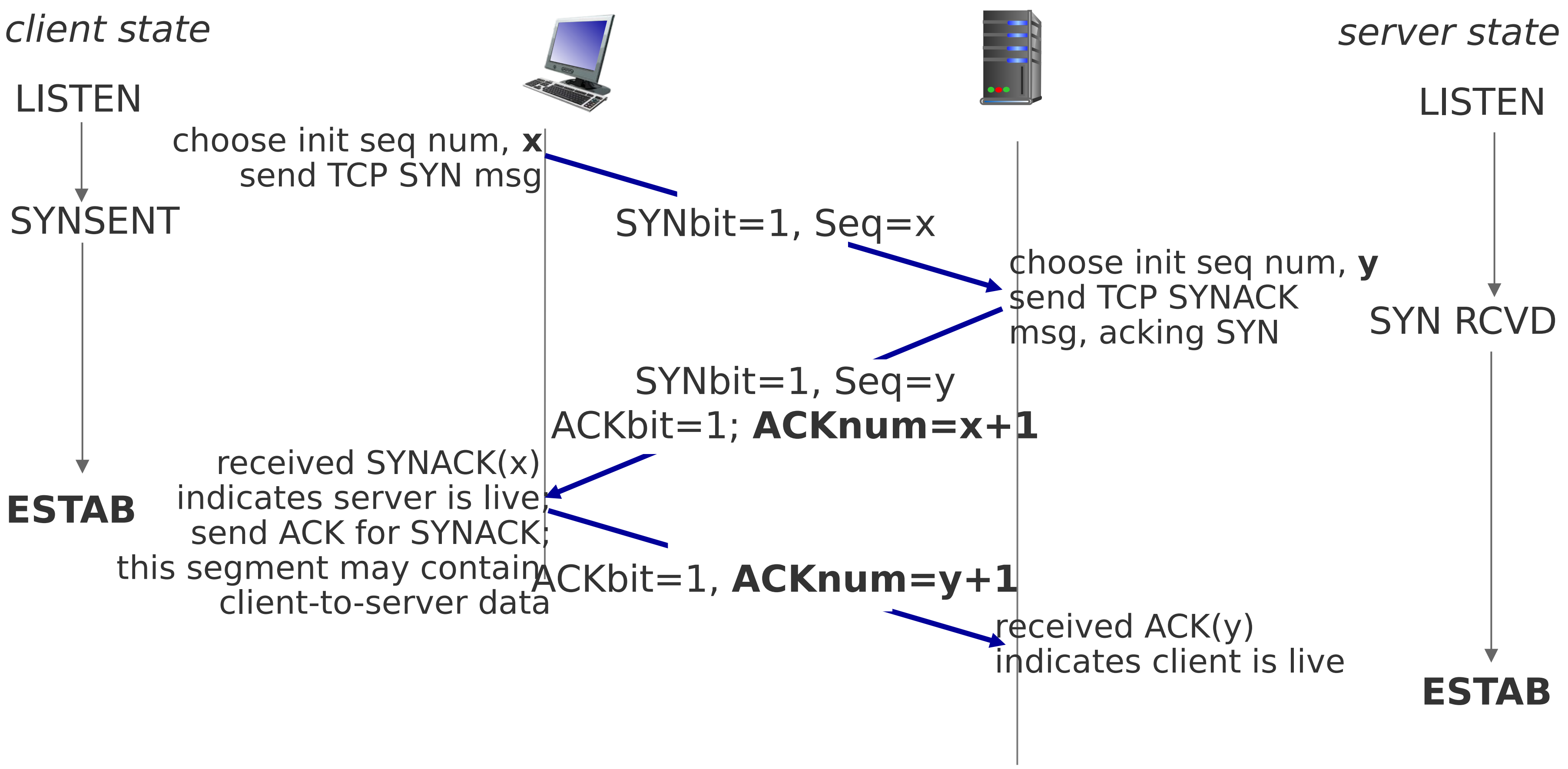
- **Sequence numbers:**
  - Byte stream "number" of first byte in segment's data
- **Acknowledgements**:
  - Seq # of next byte expected from other side
  - Cumulative ACK

Q: How receiver handles out-of-order segments

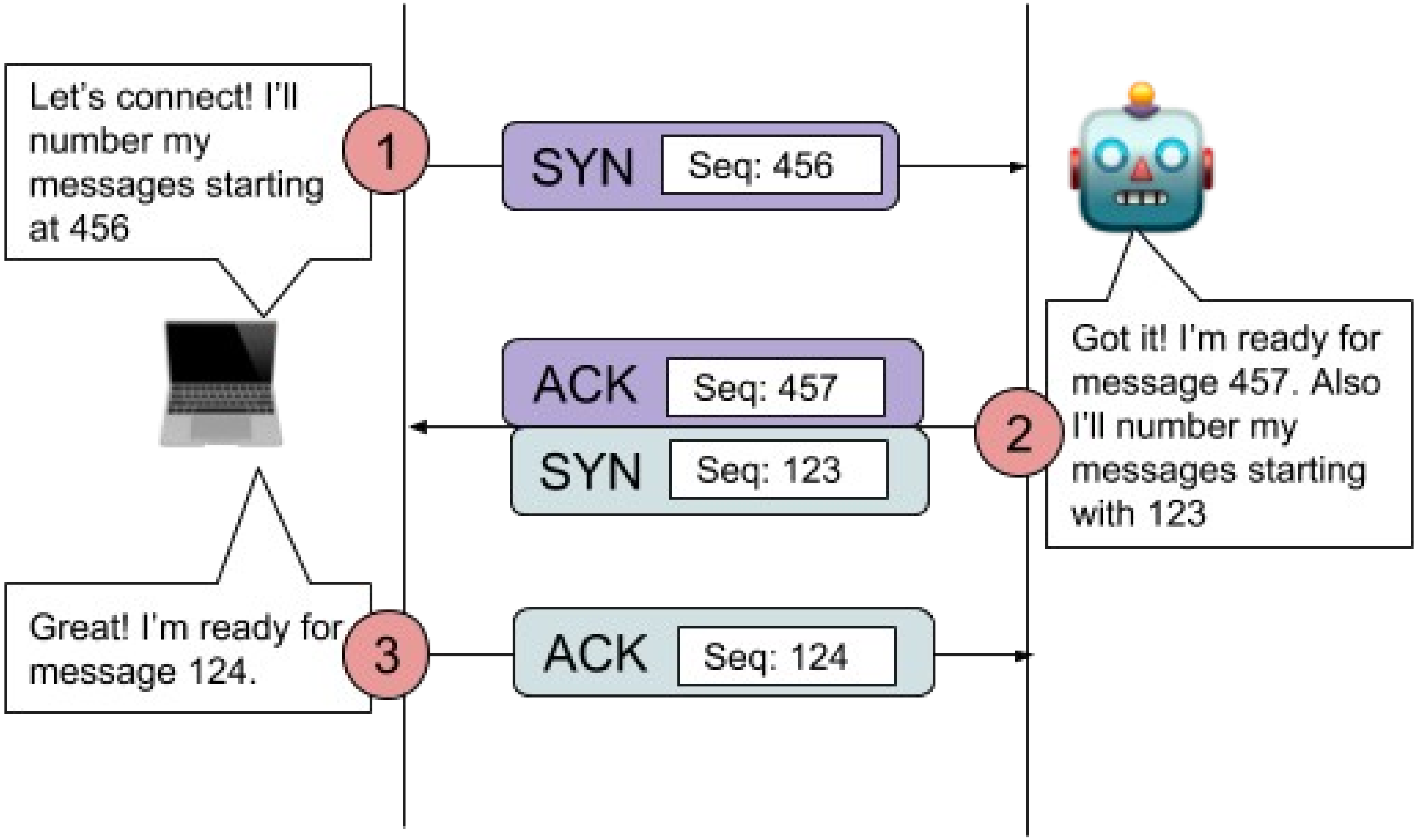
- A: TCP spec doesn't say
  - Up to implementor



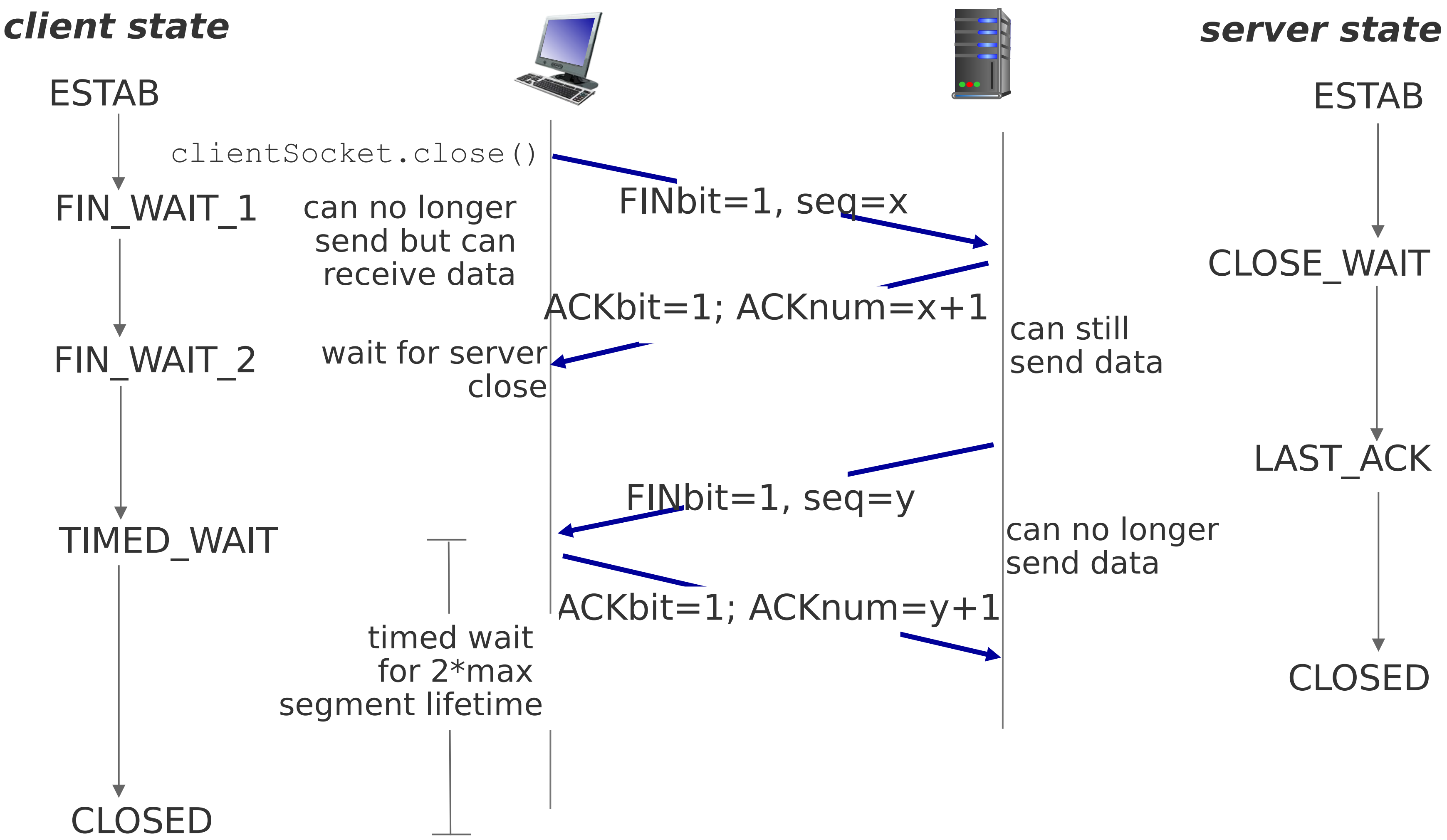
ESTABLISH TCP CONNECTION: TCP 3-WAY HANDSHAKE



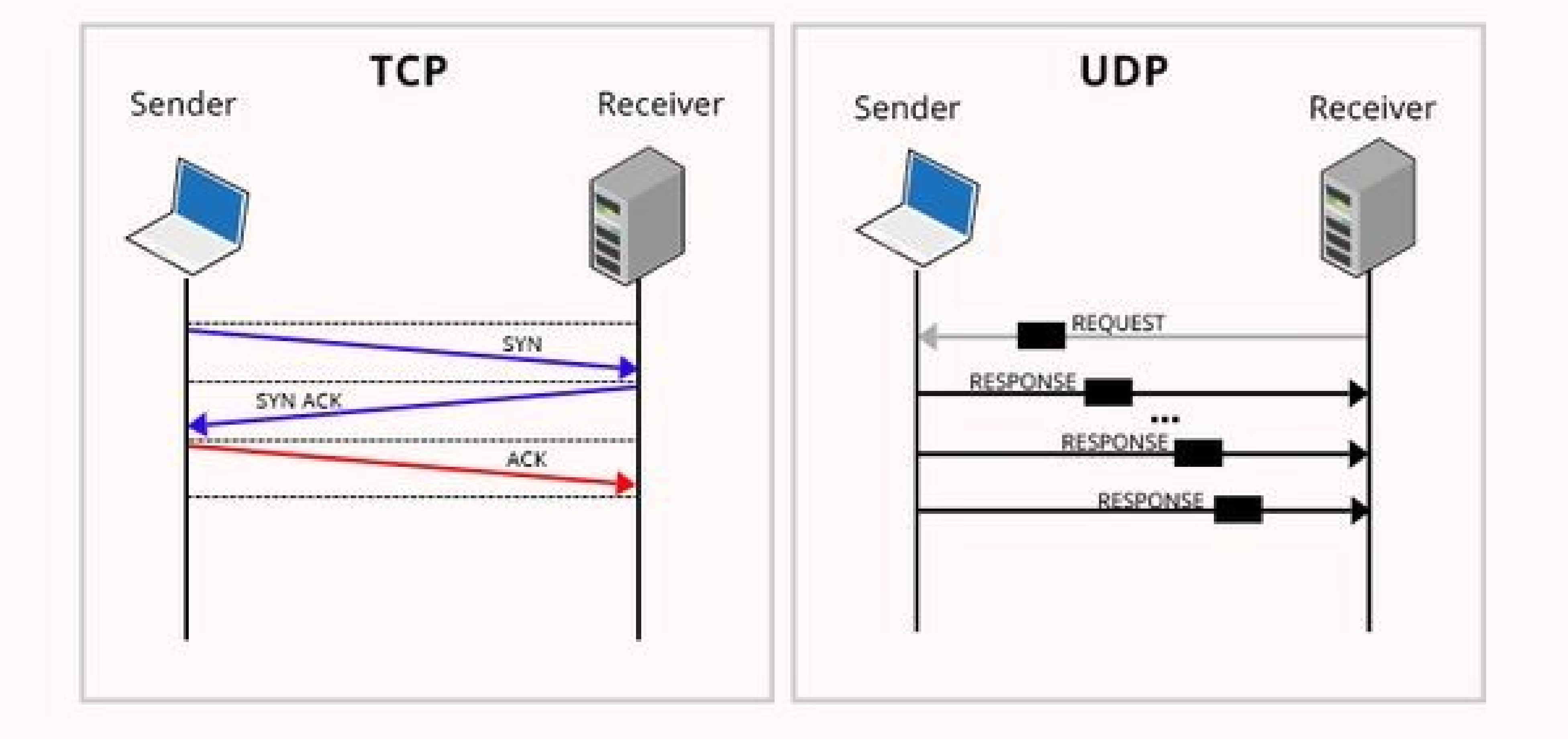
EXAMPLE


Let's connect! I'll number my messages starting at 456
1
SYN
Seq: 456
Got it! I'm ready for message 457. Also I'll number my messages starting with 123
ACK
Seq: 457
SYN
Seq: 123
2
Great! I'm ready for message 124.
3
ACK
Seq: 124

TCP: CLOSING A CONNECTION



SUPPLEMENT 6: TCP vs UDP CONNECTIONS



TCP SENDER EVENTS

Data received 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**

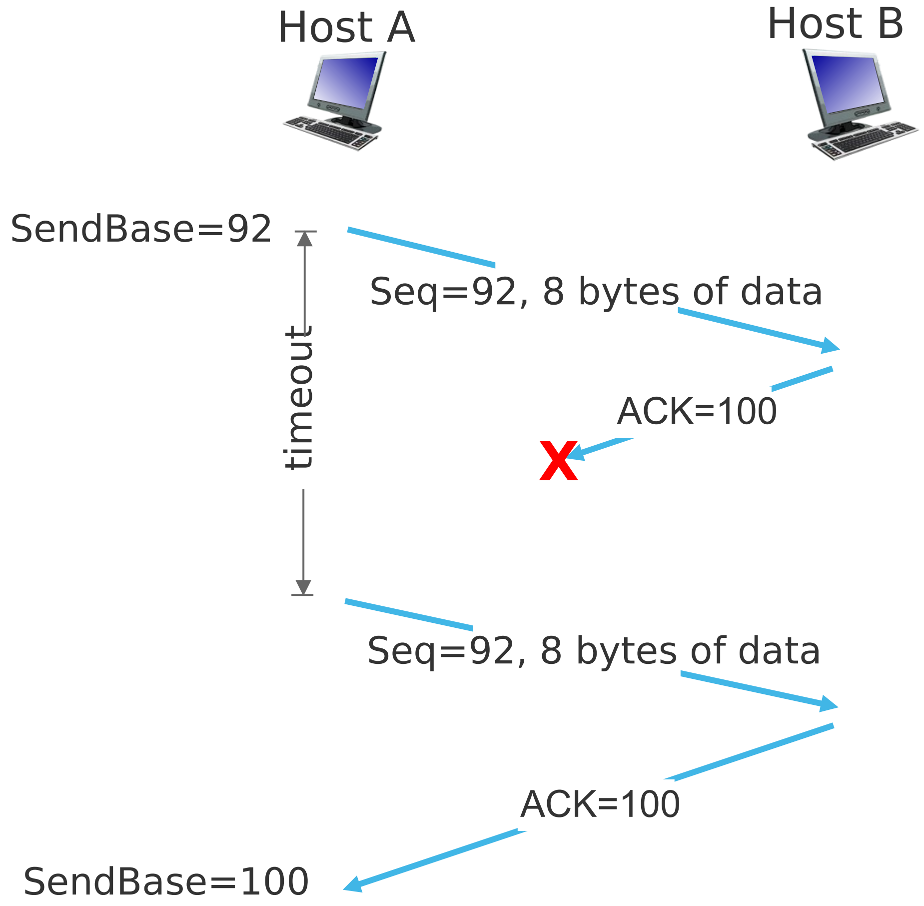
Timeout:

- Retransmit segment that caused timeout
- Restart timer

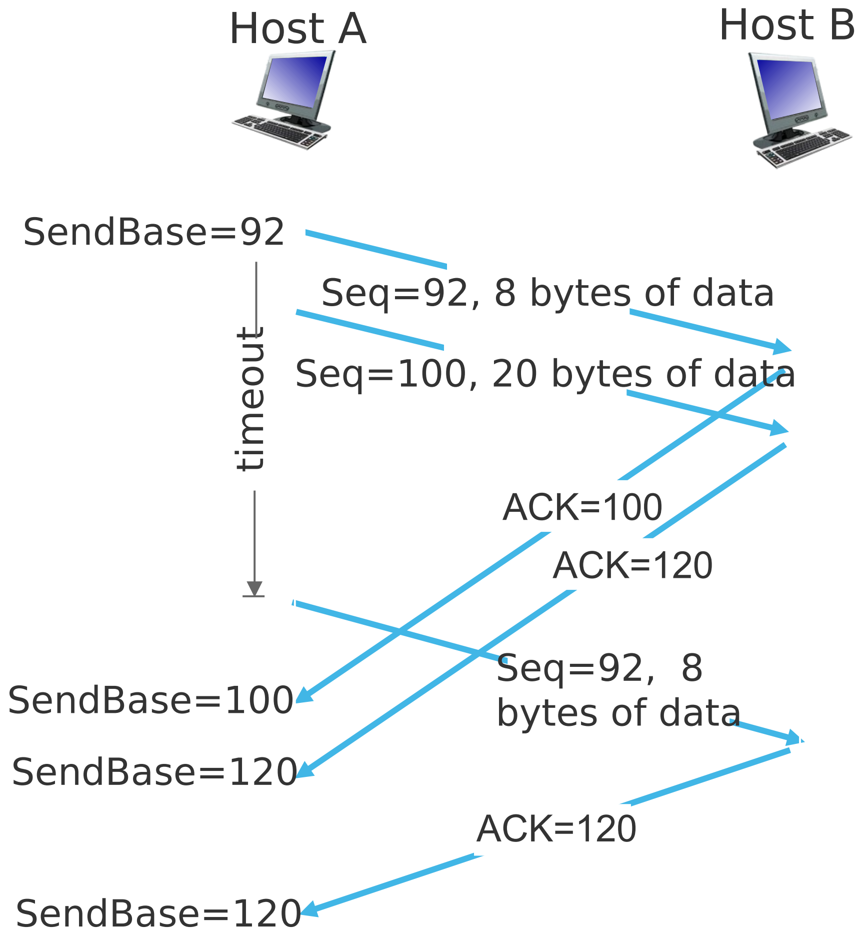
ACK received:

- If ack acknowledges previously unacked segments
  - Update what is known to be ACKed
  - Start timer if there are still unacked segments

TCP RETRANSMISSION SCENARIOS

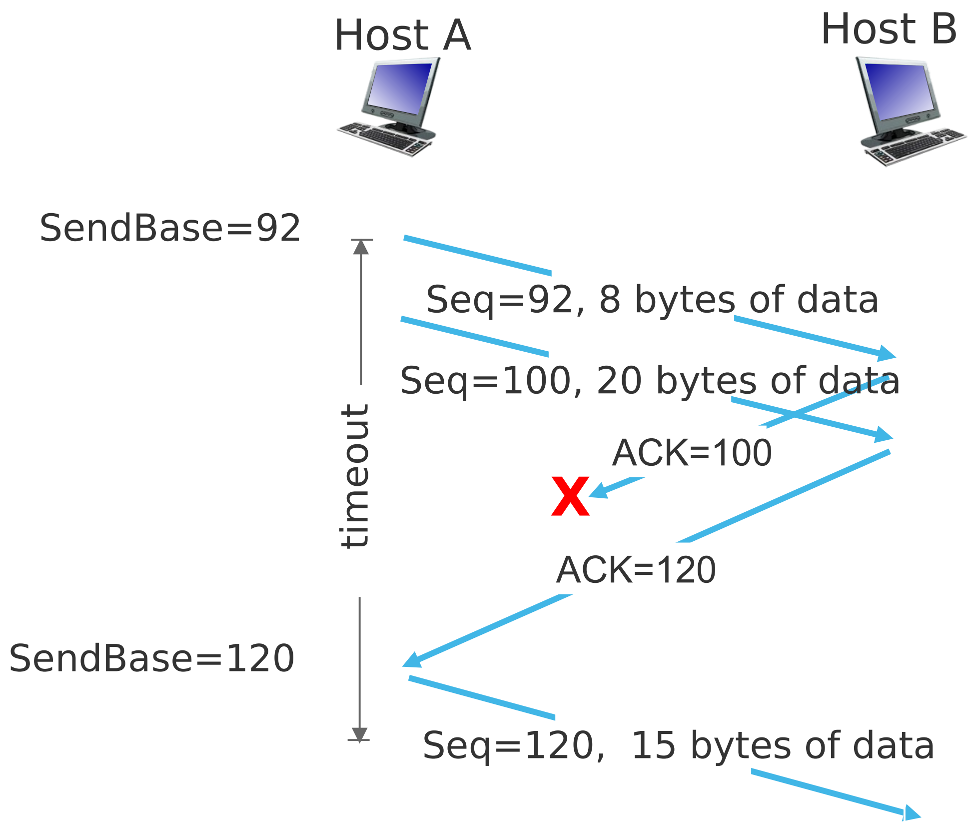


lost ACK scenario



premature timeout

TCP RETRANSMISSION SCENARIOS

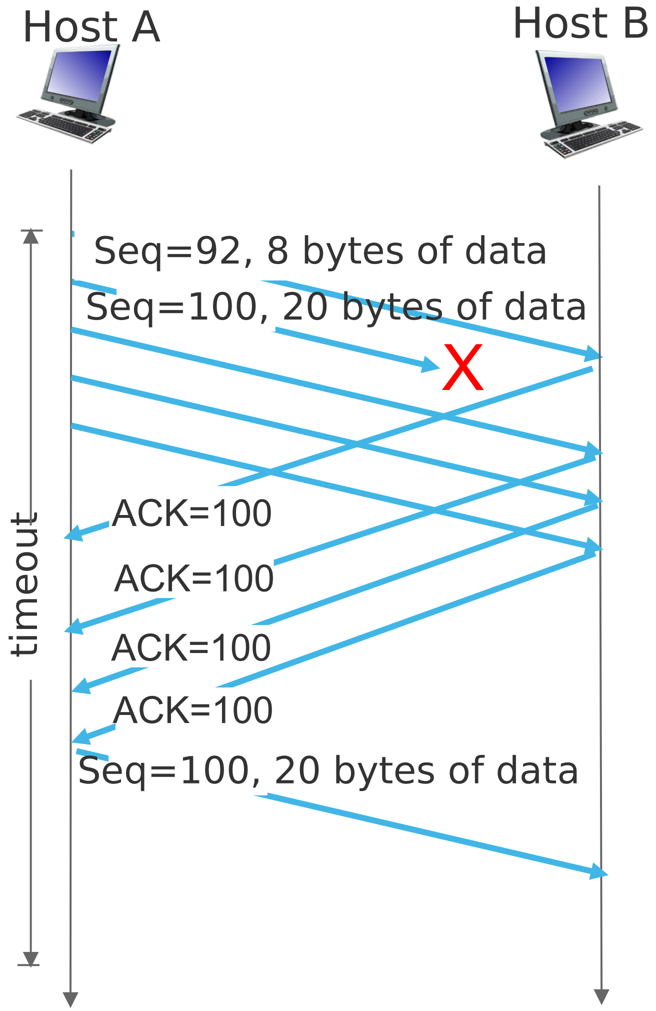


cumulative ACK

TCP RECEIVER EVENTS

Event at receiver	*TCP receiver action*
arrival of **in-order** segment with expected seq #. All data up to expected seq # **already ACKed**	delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
arrival of **in-order** segment with expected seq #. One other segment has **ACK pending**	immediately send single cumulative ACK, ACKing both in-order segments
arrival of **out-of-order** segment higher-than-expect seq. # . **Gap detected**	immediately send ***duplicate ACK***, indicating seq. # of next expected byte
arrival of segment that partially or completely **fills gap**	immediate send ACK, provided that segment starts at lower end of gap

3 DUPLICATE ACKs



fast retransmit after sender receipt of triple duplicate ACK

- When a duplicate ACK is received the sender does not know if it is because a TCP segment was lost or simply that a segment was delayed and received out of order at the receiver.
- If **more than two duplicate ACKs** are received by the sender, it is a strong indication that at least one segment has been lost.
- When three or more duplicate ACKs are received, the sender does not even wait for a re-transmission time to expire before re-transmitting the segment (the sender enters the **congestion avoidance** mode).

The University of Danang
University of Science and Technology

6. TCP FLOW CONTROL

Faculty of Information Technology
PhD. Le Tran Duc

TCP SLIDING WINDOW – SENDER SIDE

Not sent | Sent, no ACK | ACKed | Free

Sending buffer at the sender:

New data sent to transport layer by application, but not yet sent

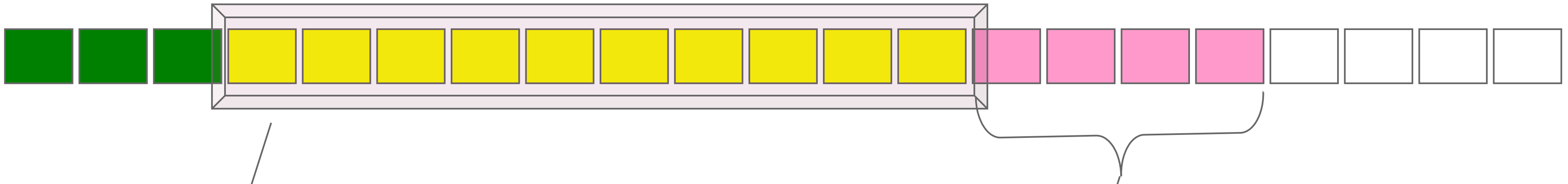
Free buffer space where application can write new data to be sent

Old data sent that has already been ACKed (Could as well be marked as free space)

TCP SLIDING WINDOW – SENDER SIDE

Not sent | Sent, no ACK | ACKed | Free

Sending buffer at the sender:



This data can not be sent yet, as the sliding window in this example has a maximum size of 10

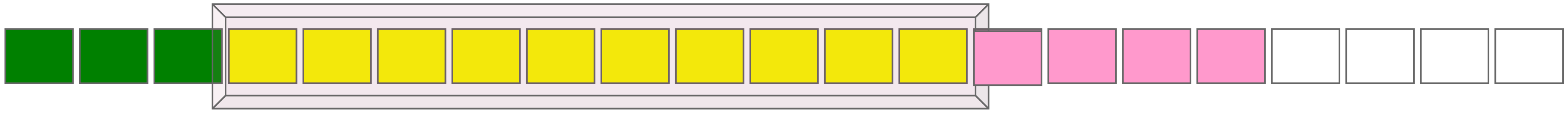
Data that has been sent, but not ACKed
Also called the *Sending window*

<u>This</u> is the *sliding window* (yes, it slides!)

TCP SLIDING WINDOW – SENDER SIDE

Not sent   Sent, no ACK   ACKed   Free

Sending buffer at the sender:



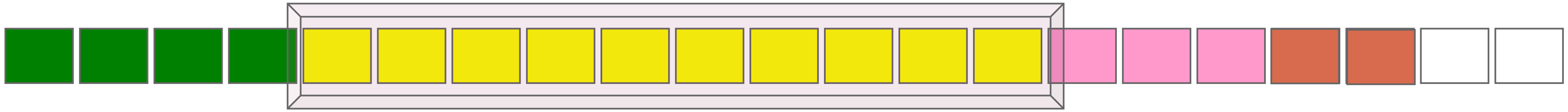
ACTION: An ACK of the oldest sent packet arrives

- The window *slides* so that the left border is in line with the oldest outstanding ACK
- The unsent segments that fit within the window are sent

TCP SLIDING WINDOW – SENDER SIDE

Not sent | Sent, no ACK | ACKed | Free

Sending buffer at the sender:



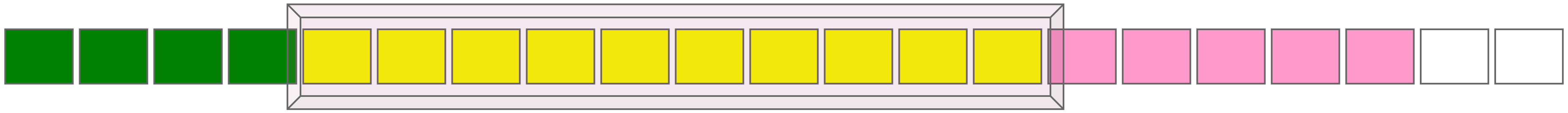
ACTION: The application has more data to send

- The data is placed in free buffer slots

TCP SLIDING WINDOW – SENDER SIDE

Not sent | Sent, no ACK | ACKed | Free

Sending buffer at the sender:



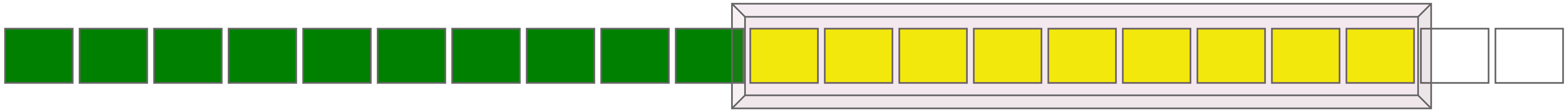
ACTION: An ACK arrives in the middle of the window

- Older sent but un-ACKed segments are now considered to be ACKed
- The window slides and unsent segments within the window are sent
- The window shrinks by one segment as there is no more than 9 segments outstanding

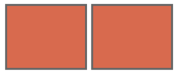
TCP SLIDING WINDOW – SENDER SIDE

Not sent   Sent, no ACK   ACKed   Free

Sending buffer at the sender:



ACTION: The application has more data to send

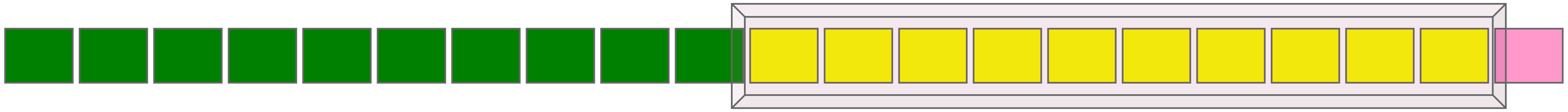


- The data is placed in free buffer slots
  - As the window is currently 9 segments wide, it can grow by one segment
  - The new data that fits within the window is sent

TCP SLIDING WINDOW – SENDER SIDE

Not sent   Sent, no ACK   ACKed   Free

Sending buffer at the sender:



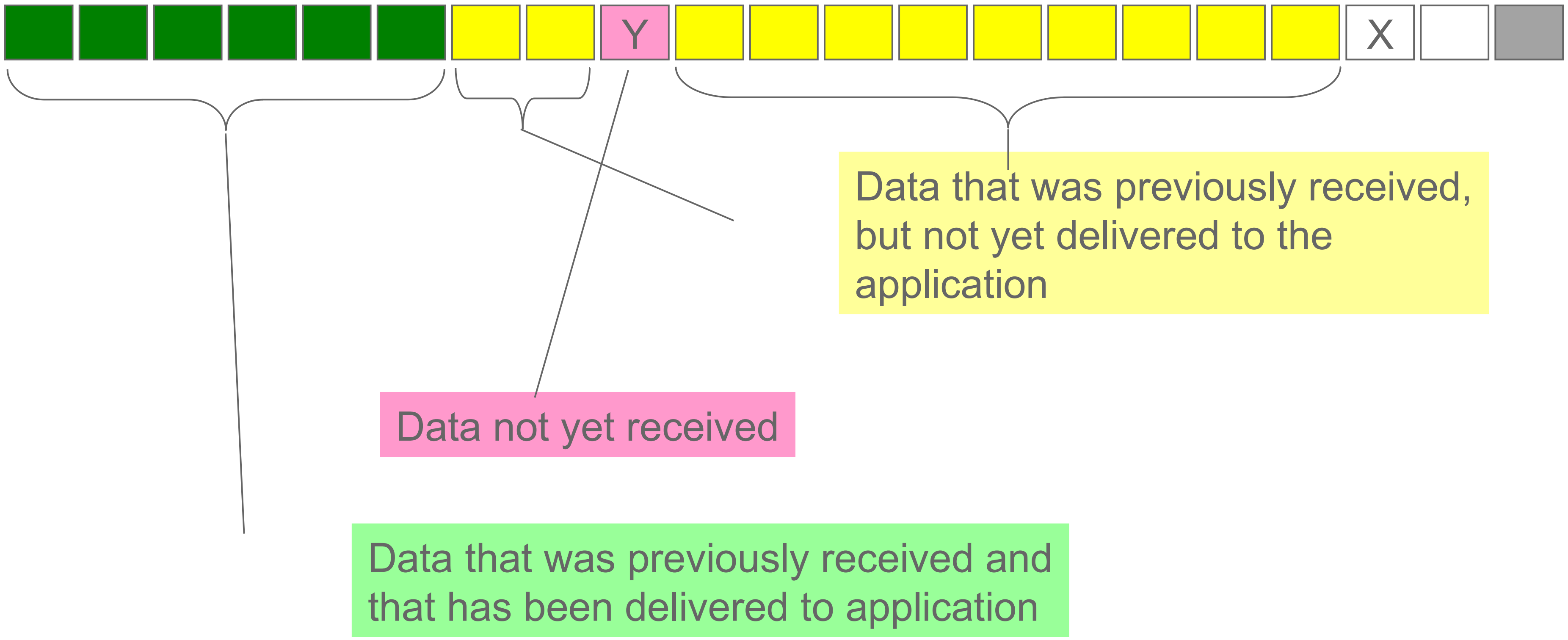
ACTION: An ACK of already ACKed segments arrives

- The ACK is silently ignored

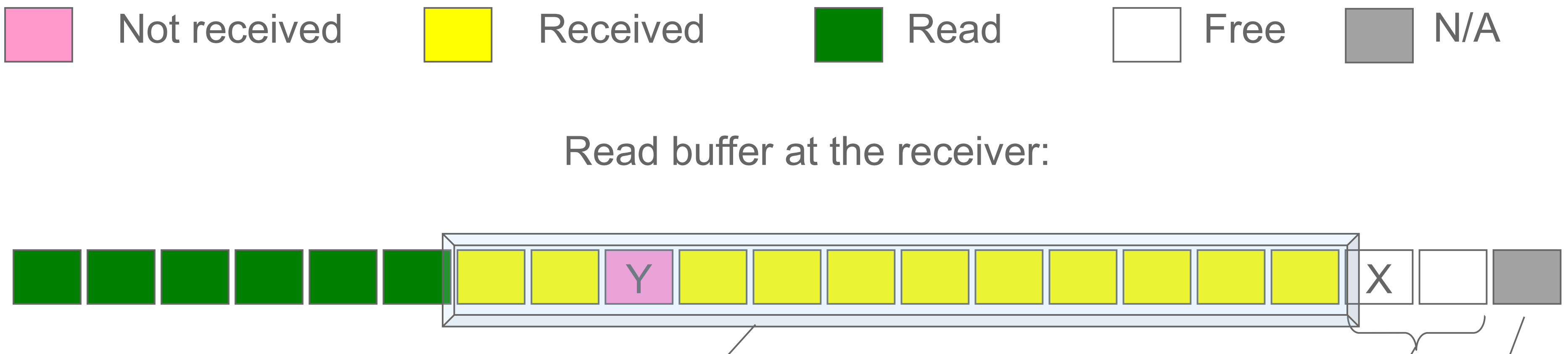
TCP SLIDING WINDOW – RECEIVER SIDE

Not received · Received · Read · Free · N/A

Read buffer at the receiver:



TCP SLIDING WINDOW – RECEIVER SIDE



The sliding window holds data received but not yet read. Must also be able to keep ”holes” like segment Y in the segments

This sliding window has size 12, max size 14

Free buffer space where new segments that are received can be stored

Space unavailable to new segments

TCP SLIDING WINDOW – RECEIVER SIDE

Not received   Received   Read   Free   N/A

Read buffer at the receiver:

ACTION: Segment X arrives

- Store in read buffer, register as received
- Send cumulative ACK Y to indicate that receiver is waiting for Y

Y

TCP SLIDING WINDOW – RECEIVER SIDE

Not received | Received | Read | Free | N/A

Read buffer at the receiver:

ACTION: Segment X+2 arrives

- Can not fit into the buffer, must be discarded
- Send cumulative ACK Y to indicate that receiver is waiting for Y

Y

TCP SLIDING WINDOW – RECEIVER SIDE

Not received   Received   Read   Free   N/A

Read buffer at the receiver:

ACTION: Applications try to read 5 segments

- Only two segments are returned, still waiting for Y
- Application is informed of how much data was read
- The unavailable segment at the end of the buffer becomes available

TCP SLIDING WINDOW – RECEIVER SIDE

Not received Received Read Free N/A

Read buffer at the receiver:

ACTION: Segment Y arrives

- Store in read buffer, register as received
  - Send cumulative ACK (X+1) to indicate that receiver is waiting for (X+1)

X+

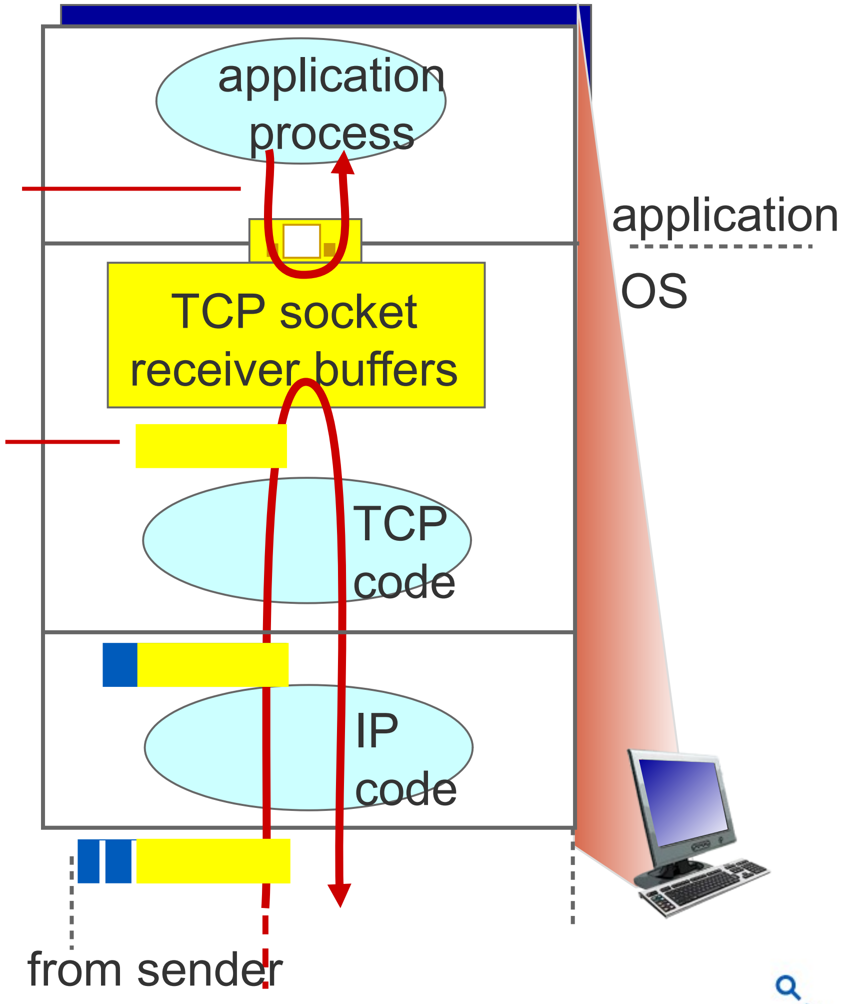
TCP FLOW CONTROL

flow control

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

application may remove data from TCP socket buffers ….

… slower than TCP receiver is delivering (sender is sending)



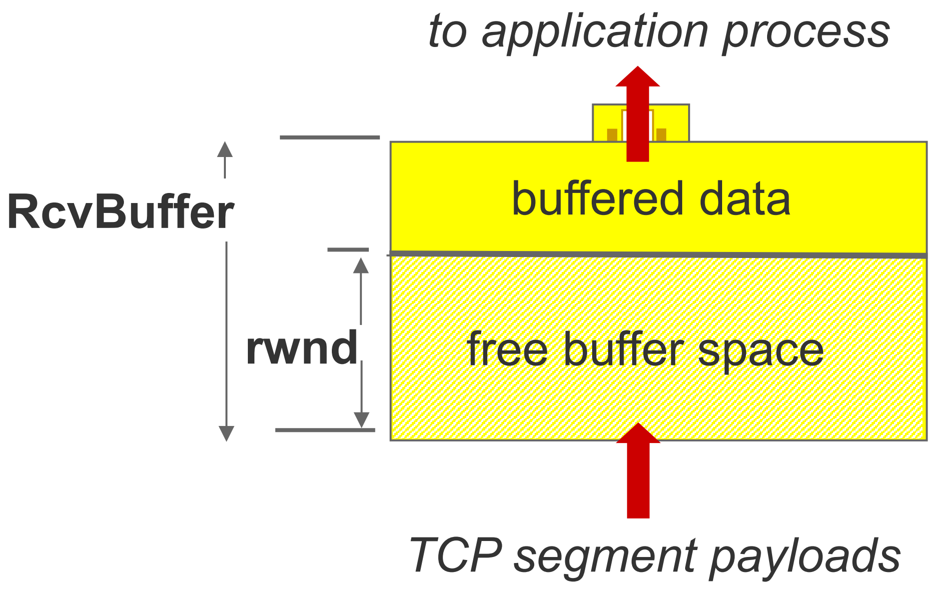
receiver protocol stack

The idea is that a node receiving data will send some kind of feedback to the node sending the data to let it know about its current condition.

TCP FLOW CONTROL

TCP provides flow control by having the *sender* maintain a variable called the **receive window (rwnd)**.

- Receiver "*advertises*" free buffer space by including **rwnd** value in TCP header of receiver-to-sender segments (***aka. in ACK***)
  - **RcvBuffer** is the receive buffer. It's size set via socket options (typical default is 4096 bytes)
  - Many operating systems auto adjust **RcvBuffer**
- Sender limits amount of unacked ("***in-flight***") data sent to receiver based on the **rwnd** value
- Guarantees receive buffer will not overflow



receiver-side buffering

TCP FLOW CONTROL

We define 2 variables:

- **LastByteRead**: the number of the last byte in the data stream read from the buffer by the app. process in host B

- **LastByteRcvd**: the number of the last byte in the data stream has arrived from the network to buffer

Because TCP is not permitted to overflow the allocated buffer, we must have:

Host B:

We need: $\boldsymbol{LastByteRcvd - LastByteRead \leq RcvBuffer}$

Receive-Window:

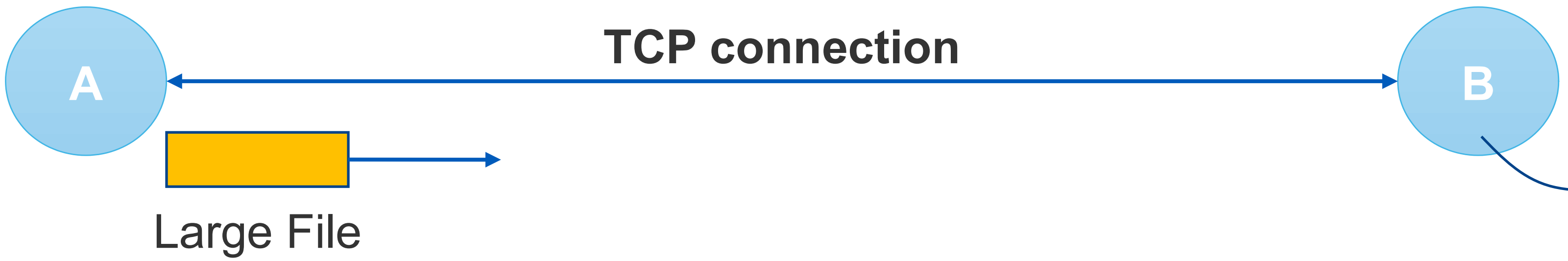
$$\boldsymbol{rwnd = RcvBuffer - [LastByteRcvd - LastByteRead]}$$

dynamic

TCP FLOW CONTROL

How to use *rwnd*?

B tells A (*Receiver "advertises" free buffer space by including **rwnd** value in TCP header of receiver-to-sender segments (ACK)*)



Buffer for this connection (**RcvBuffer**)

Host B: initially, host B sets **rwnd = RcvBuffer**

Host A: uses **sliding window** to control the number of bytes in flight it can have

- Keeps track of 2 variables: **LastByteSent** & **LastByteACKed**

$$LastByteSent - LastByteACKed = amount\ of\ unacknowledged\ data$$ (***in-flight***)

→ Need to maintain: $LastByteSent - LastByteACKed \leq rwnd$ → **No overflowing**

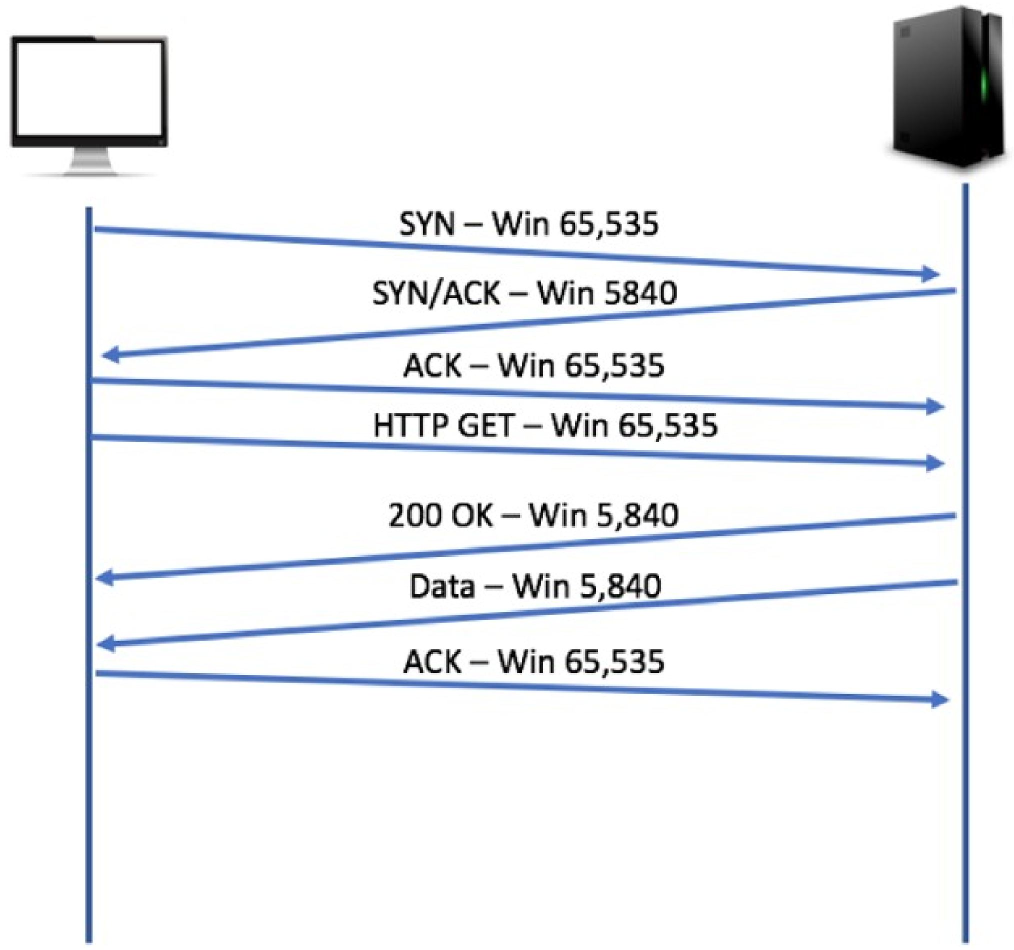
Problem: If rwnd = 0 (already informed A) and B has nothing to send to A → **A will be blocked!!!**

Solution: A still sends to B 1 data byte when receiving rwnd = 0 → B is going to acknowledge this byte

→ **If rwnd ≠ 0 then A will know it!**

TCP FLOW CONTROL

Example:



TCP Receive Window

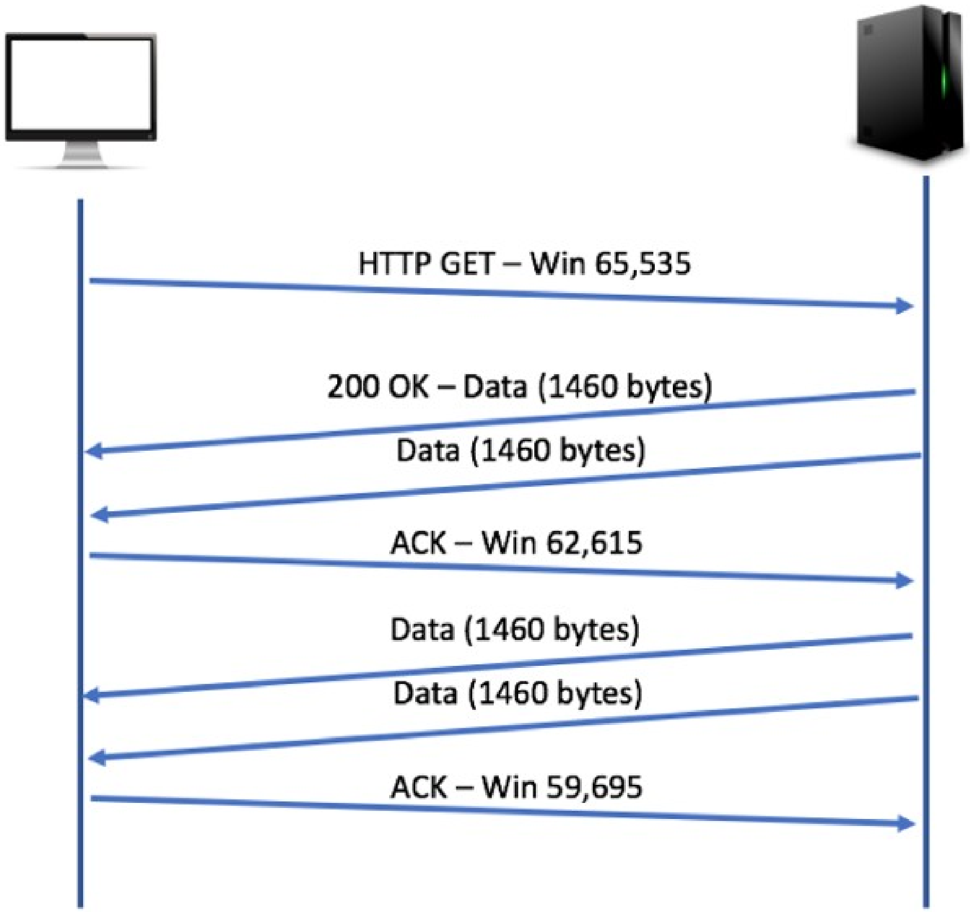
The client has a TCP receive window of 65,535 bytes, and the server has 5,840.

The client was able to process the data packets out of the TCP buffer as fast as they came in, so the window size was not reduced.

→ The client still has a full window available for receiving data – 65,535 bytes.

TCP FLOW CONTROL

Example:

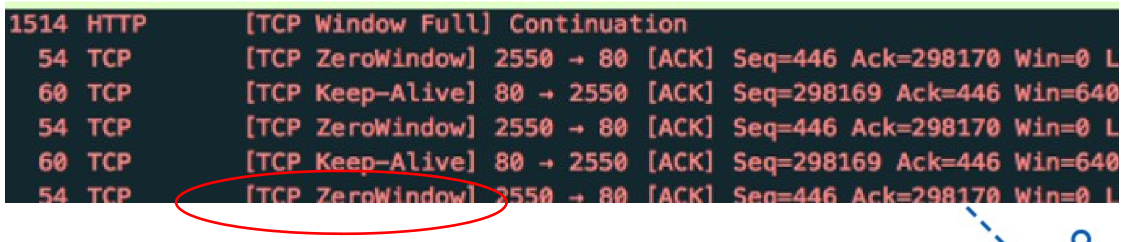


TCP Receive Window and TCP buffer

The client has a TCP receive window of 65,535 bytes, and the server has 5,840.

The client is requesting data from a server and begins to receive the data. However, in this case, the client is not able to quickly process the incoming data.

→ The acknowledgements from the client indicate that the window is shrinking.



7. TCP CONGESTION CONTROL

Faculty of Information Technology
PhD. Le Tran Duc

INTRODUCTION TO 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)

TCP Congestion Control:

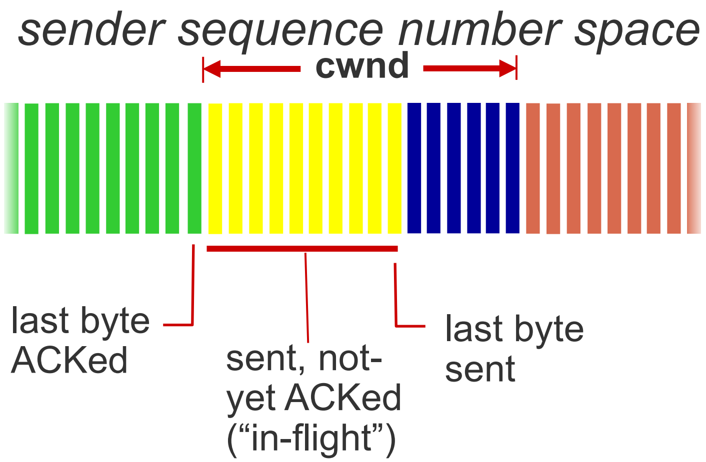
- **Basic idea:** each source determines how much capacity is available to a given flow in the network. → Each sender limits the rate!
  - Raises 3 questions:
    - How does TCP limit the rate?
    - How does TCP sender perceive that there is congestion on the path?
    - What algorithm should the sender use to change ites send rate?
- **ACKs** are used to "***pace***" the transmission of packets such that TCP is "***self-clocking***"

INTRODUCTION TO CONGESTION CONTROL

Goal:

- TCP sender should transmit as fast as possible, but **without congesting network and does not overwhelm the receiver**!
  - Call: ***swnd***, ***rwnd***, ***cwnd*** are ***Sender window***, ***Receiver window*** and ***Congestion window*** relatively.
  - We need: **swnd = min {rwnd, cwnd}** → **Answer the first question!**
  - **Effective window = swnd – (LastByteSent – LastByteAcked)**
- Each TCP sender sets its window size, based on ***implicit*** feedback: → **Answer the second & third questions!**
  - ACK segment received → network is not congested, so increase sending rate
  - Lost segment → assume loss due to congestion, so decrease sending rate

INTRODUCTION TO CONGESTION CONTROL



- sender limits transmission:

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

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

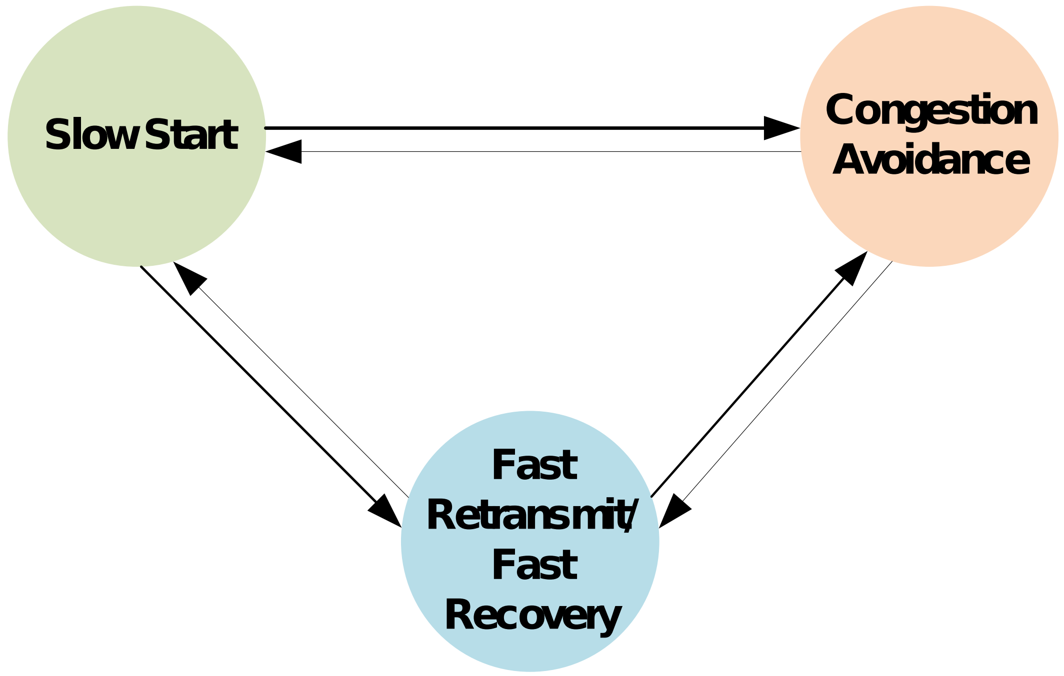
TCP sending rate:

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

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

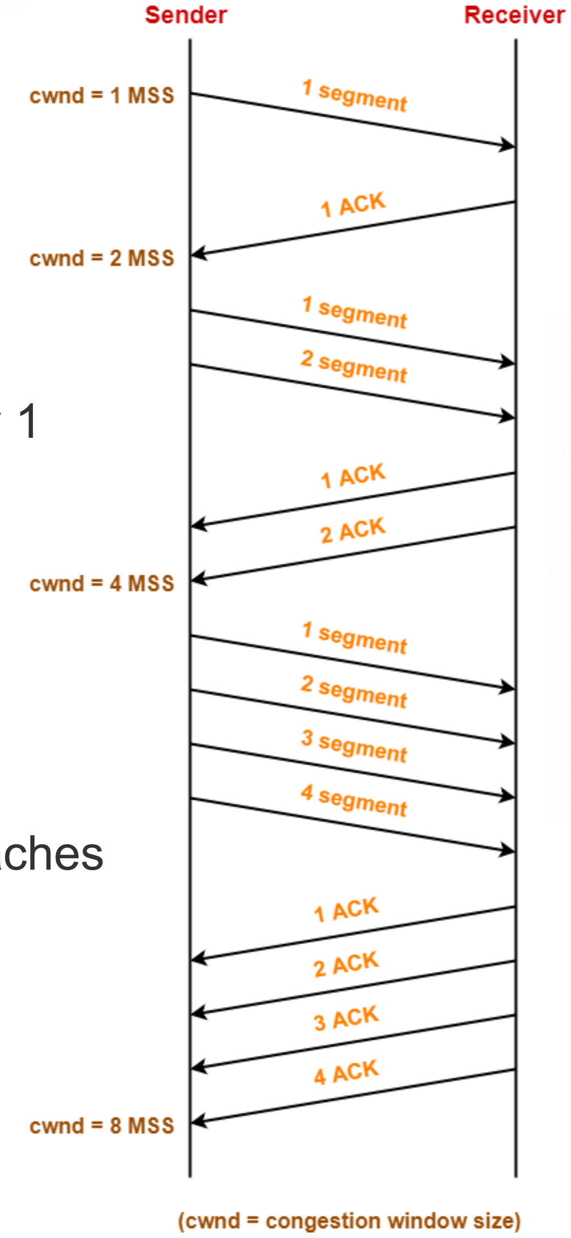
TCP CONGESTION CONTROL ALGORITHMS

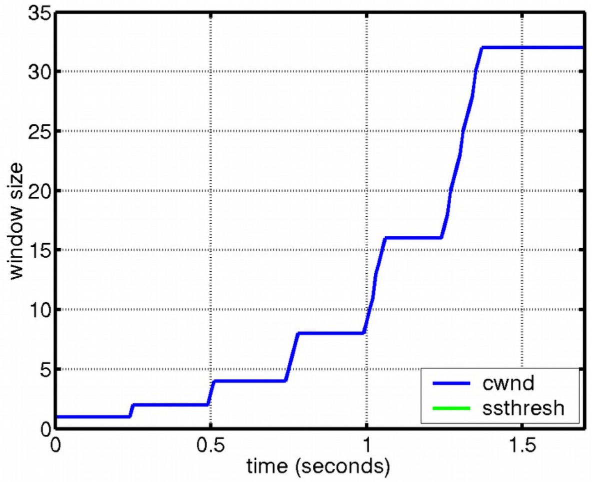
- Some TCP congestion Control algorithms:
  - **Tahoe**
    - ✓ Slow Start
    - ✓ Congestion Avoidance
    - ✓ Fast Retransmit
  - **Reno**
    - ✓ Fast Recovery
  - **Vegas**
    - ✓ New Congestion Avoidance
  - **RED**
  - **REM**
  - **SACK**
  - **FACK**…



SLOW START PHASE

- Initially, sender sets congestion window size = Maximum Segment Size (1 MSS)
  - → **cwnd = 1**
- On each successful ACK, sender increases **each** cwnd by 1 MSS
  - → **cwnd = cwnd + 1**
  - → In this phase, cwnd increases exponentially
    - **Each RTT: cwnd = 2 x cwnd**
- This phase continues until the congestion window size reaches the slow start ssthresh **(ssthresh = (rwnd/MSS) / 2)**
  - **cwnd ≥ ssthresh**

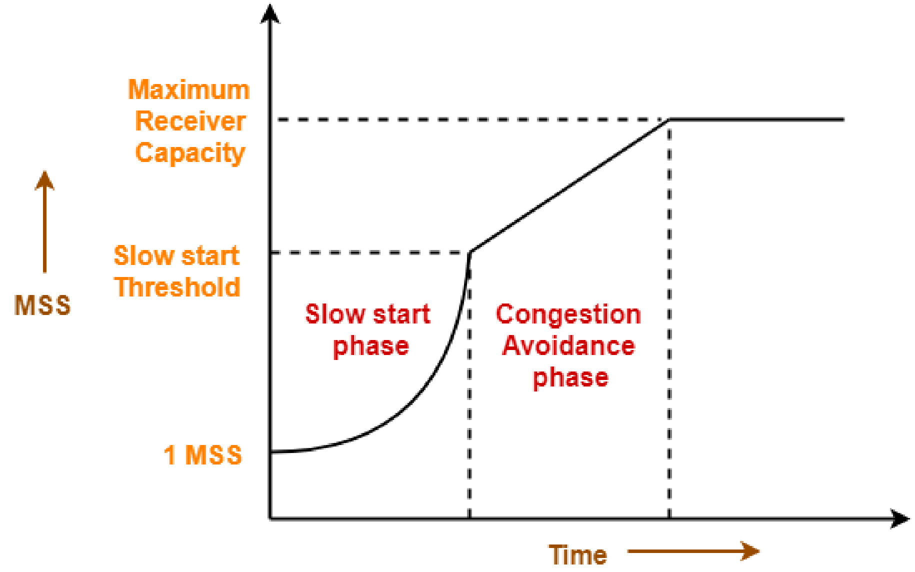


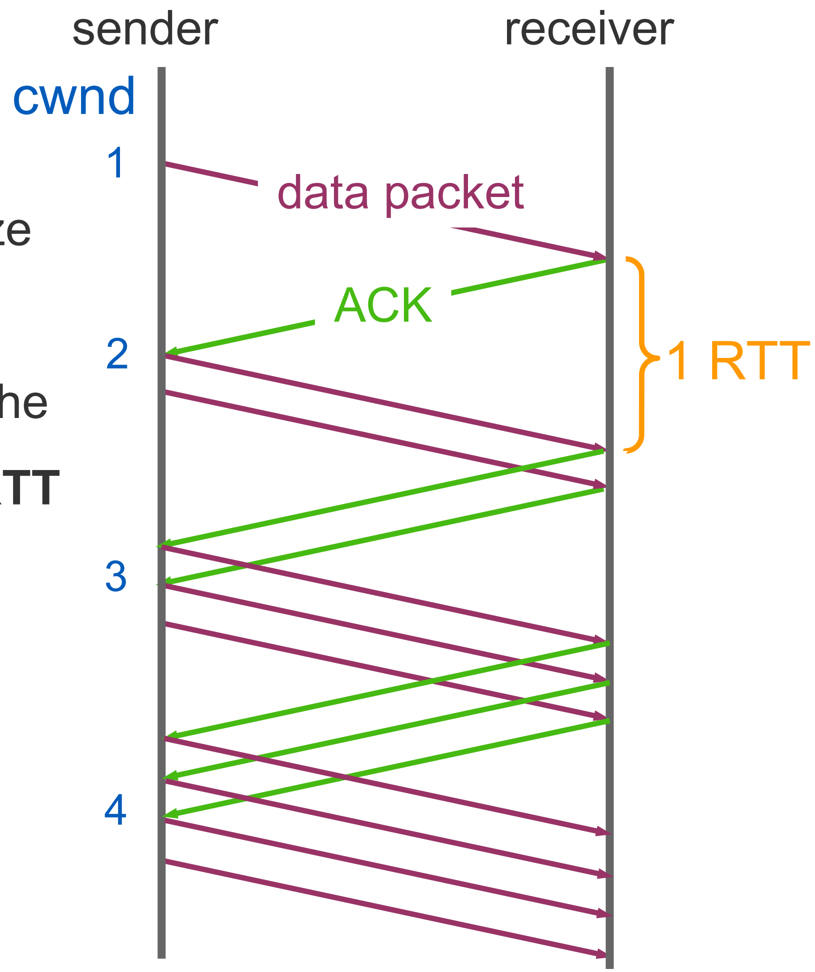


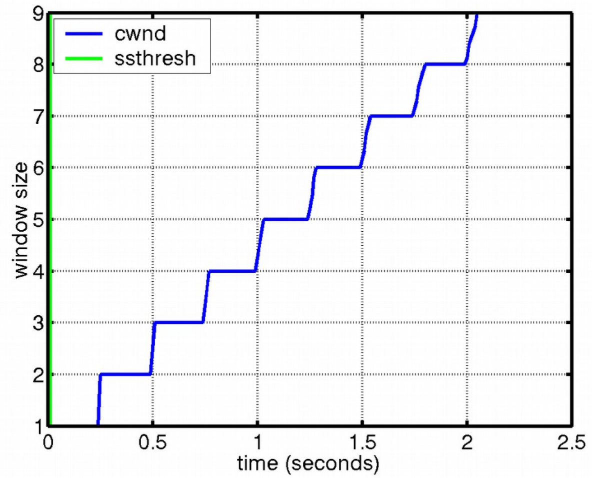
CONGESTION AVOIDANCE PHASE

- Starts when **cwnd ≥ ssthresh**
- Sender increases the congestion window size **linearly** to avoid the congestion.
- On each successful ACK, sender increases the value of cwnd by **just a single MSS every RTT**

cwnd = cwnd + MSS x (MSS/cwnd)



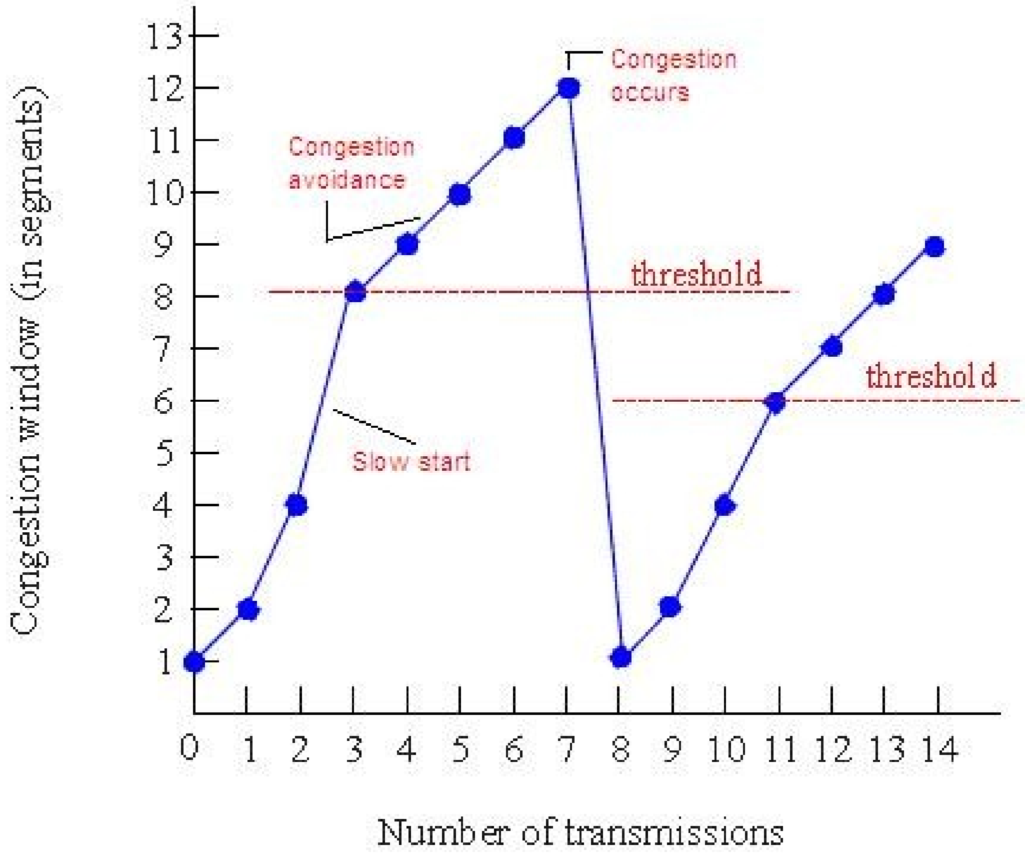




cwnd ← cwnd + 1 (for each cwnd ACKs)

TAHOE ALGORITHM

- Detecting congestion based on **timeout**.
- In congestion avoidance, if sender detects congestion (timeout), then:
  - **ssthresh = ½ $cwnd_{current}$**
  - **$cwnd_{new}$ = 1 MSS**
  - Then enter **Slow Start** again
- Disadvantages:
  - Timeout period often relatively long → Long delay before resending lost packet
  - Tahoe algorithm uses "*go-back-N" method*→ every time a packet is lost, the transmission link is empty for a period of time → Waste of resources.



RENO ALGORITHM

- Similar to TCP Tahoe at the slow start phase and also uses timeout but there are some improvements:
  - Does not accumulate ACKs as in Tahoe
  - Early detecting congestion control based on **3 duplicate ACKs**.
  - Immediately retransmits after 3 duplicate ACKs without waiting for timeout → **Fast Retransmission**

    - Adjust **$ssthresh_{new}$ = $cwnd_{current}$ /2**

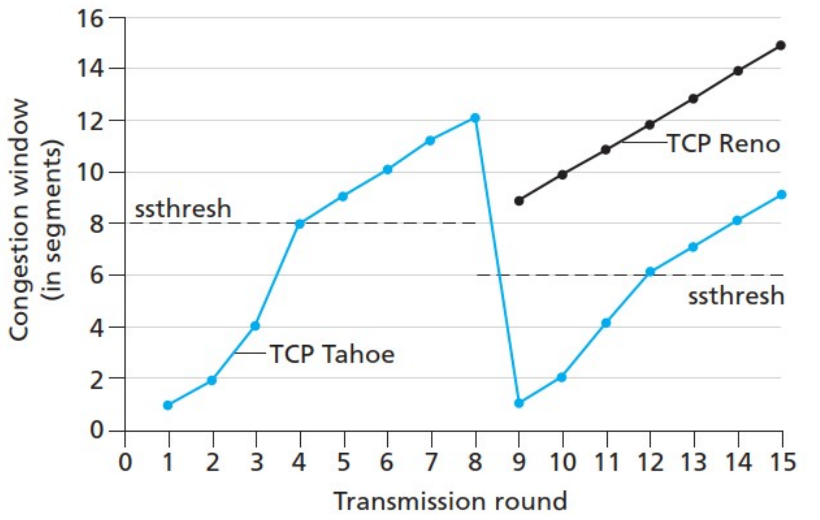
    - And set **$cwnd_{new}$ = $ssthresh_{new}$ + 3.MSS**

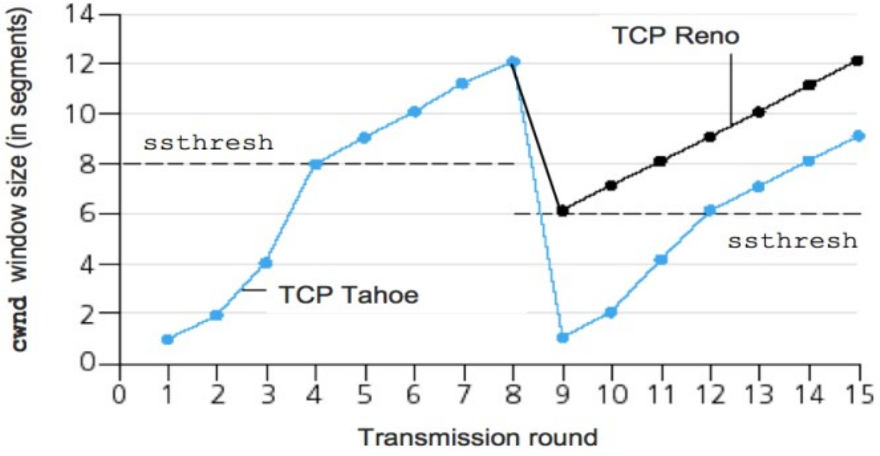
    (*some people do not add 3MSS into above formular*)

    - Resends the packet
    - Then enter the **Fast Recovery phase**

- **Advantages**:
  - Less delay than Tahoe algorithm
  - Reno works well with a small number of packet loss
- **Disadvantages**:
  - If Reno's cwnd size is too small, it may not receive enough 3 ACKs packets to run the algorithm
  - Reno can detect 1 packet loss each time & it is not possible to recognize the case of many lost packets.

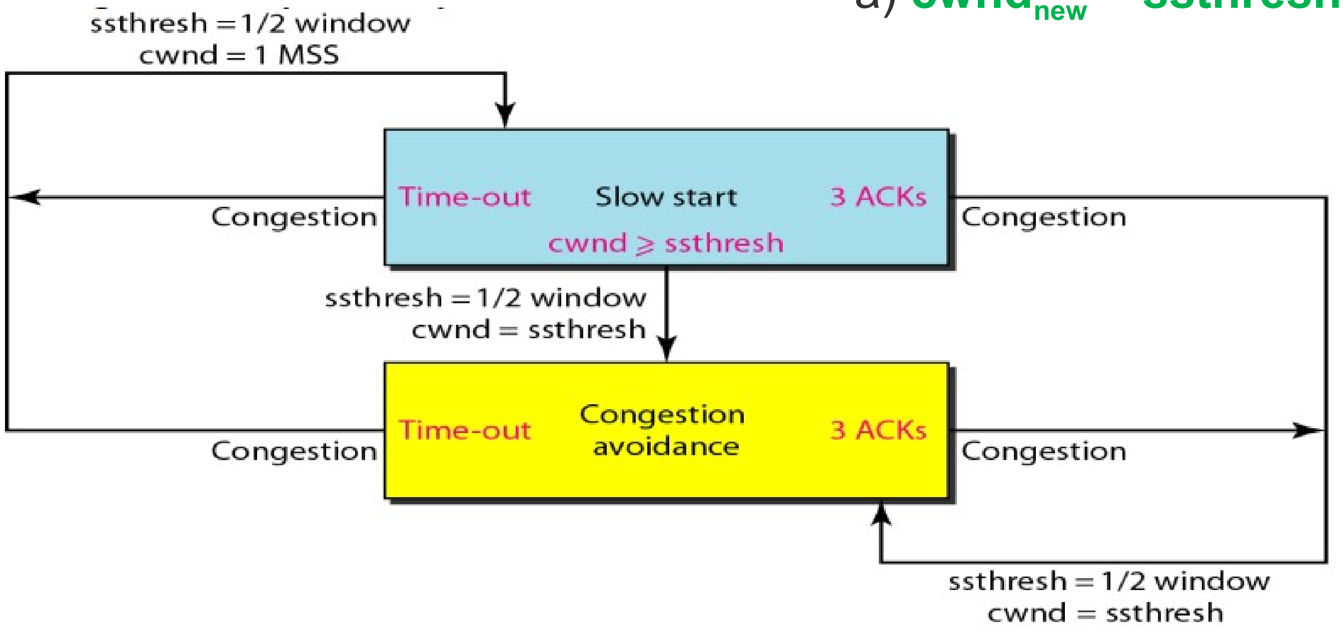
RENO ALGORITHM





a) $\text{cwnd}_{new} = \text{ssthresh}_{new} + 3.\text{MSS}$

b) $\text{cwnd}_{new} = \text{ssthresh}_{new}$



TCP FAST RECOVERY PHASE

- After Fast Retransmission (3 duplicate ACKs)
  - If all missing packets are ACKed → quit Fast Recovery → Back to Congestion Avoidance with **cwnd = ssthresh**$_{current}$
  - If only some missing packets are ACKed:
    - **cwnd = cwnd + 1 MSS** for every duplicate ACK received for the missing segment → then transmit new segment
    - If a new ACK is received, means **all missing packets are ACKed** → Back to Congestion Avoidance with **cwnd = ssthresh**$_{current}$

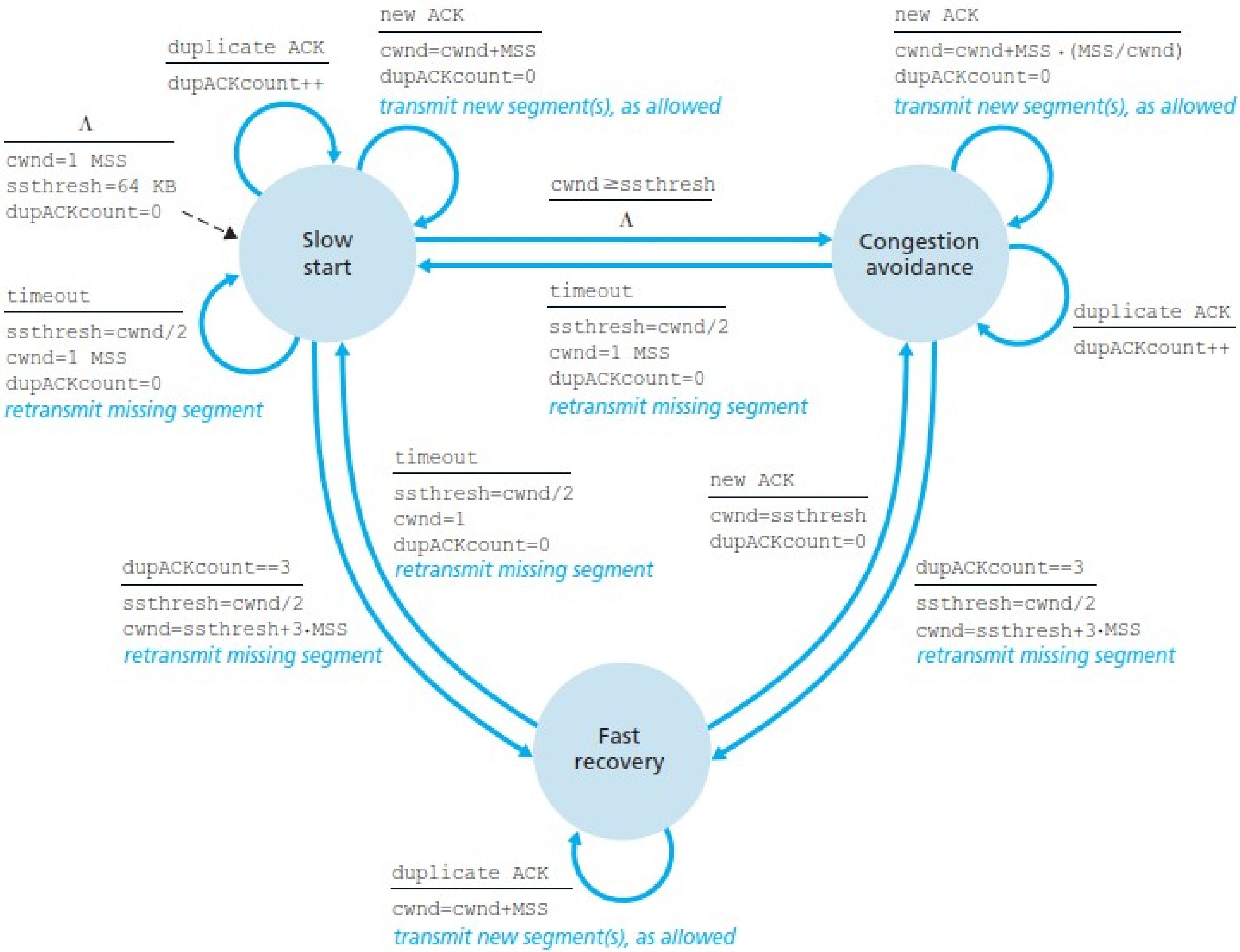
new Reno algorithm

  - If timeout occurs → Back to Slow Start
- **Advantages**:
    - Detecting of some missing packets at the same time (In Reno, quit Fast Recovery phase when **first** missing packet is ACKed) → no need to multiple times reduce the cwnd multiple times
    - Retransmit multiple missing packets

SUMMARY 1

Event	State	TCP Sender Action	Commentary
ACK receipt for previously unacked data	Slow Start (SS)	cwnd = cwnd + MSS, If (cwnd > ssthresh) set state to "Congestion Avoidance"	Resulting in a doubling of cwnd every RTT
ACK receipt for previously unacked data	Congestion Avoidance (CA)	cwnd = cwnd+MSS * (MSS/cwnd)	Additive increase, resulting in increase of cwnd by 1 MSS every RTT
Loss event detected by triple duplicate ACK	SS or CA	ssthresh = cwnd/2, cwnd = ssthresh, Set state to "Congestion Avoidance"	Fast recovery, implementing multiplicative decrease. cwnd will not drop below 1 MSS.
Timeout	SS or CA	ssthresh = cwnd/2, cwnd = 1 MSS, Set state to "Slow Start"	Enter slow start
Duplicate ACK	SS or CA	Increment duplicate ACK count for segment being acked	cwnd and ssthresh not changed

SUMMARY 2


duplicate ACK
dupACKcount++
new ACK
cwnd=cwnd+MSS
dupACKcount=0
transmit new segment(s), as allowed
new ACK
cwnd=cwnd+MSS • (MSS/cwnd)
dupACKcount=0
transmit new segment(s), as allowed
Λ
cwnd=1 MSS
ssthresh=64 KB
dupACKcount=0
Slow start
cwnd ≥ ssthresh
Λ
Congestion avoidance
timeout
ssthresh=cwnd/2
cwnd=1 MSS
dupACKcount=0
retransmit missing segment
timeout
ssthresh=cwnd/2
cwnd=1 MSS
dupACKcount=0
retransmit missing segment
duplicate ACK
dupACKcount++
timeout
ssthresh=cwnd/2
cwnd=1
dupACKcount=0
retransmit missing segment
new ACK
cwnd=ssthresh
dupACKcount=0
dupACKcount==3
ssthresh=cwnd/2
cwnd=ssthresh+3•MSS
retransmit missing segment
dupACKcount==3
ssthresh=cwnd/2
cwnd=ssthresh+3•MSS
retransmit missing segment
Fast recovery
duplicate ACK
cwnd=cwnd+MSS
transmit new segment(s), as allowed