CS2105

An Awesome Introduction to Computer Networks

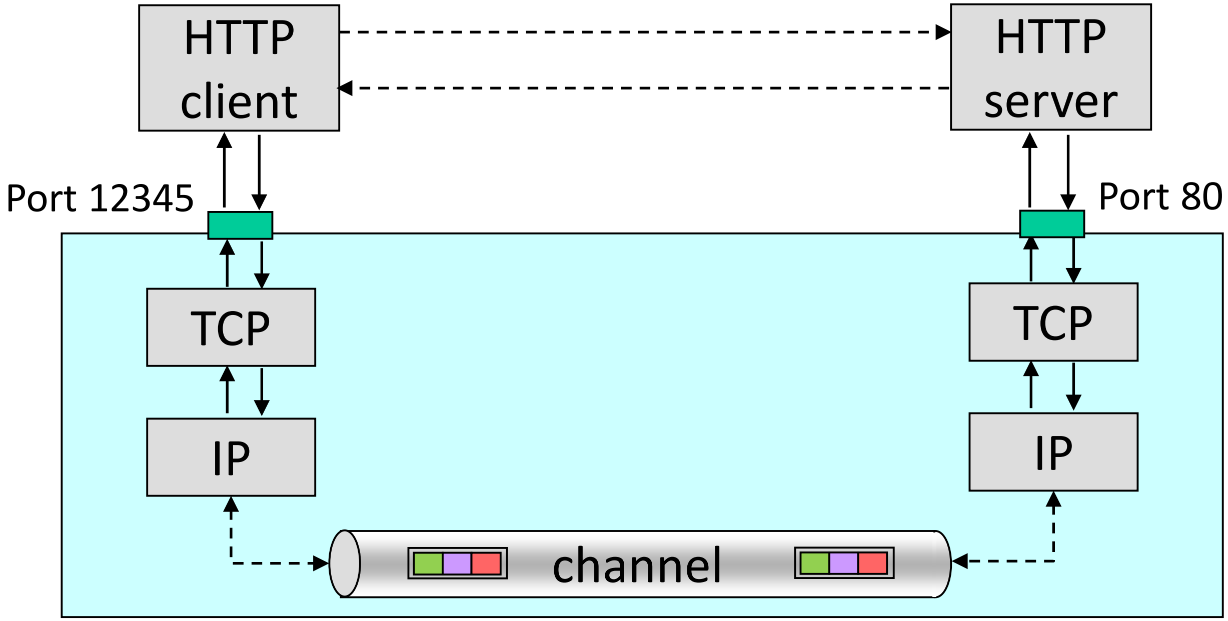
Lectures 4&5: The Transport Layer

NUS National University of Singapore | Department of Computer Science
School of Computing

PREVIOUS LECTURE

Web and HTTP

- A Web page consists of a *base HTML file* and *some other objects* referenced by the HTML file.
- HTTP uses TCP as transport service.
  - TCP, in turn, uses service provided by IP!



Socket

PREVIOUS LECTURE

- Applications (processes) send messages over the network through sockets.
  - Conceptually, socket = IP address + port number
  - Programming wise, socket = a set of APIs

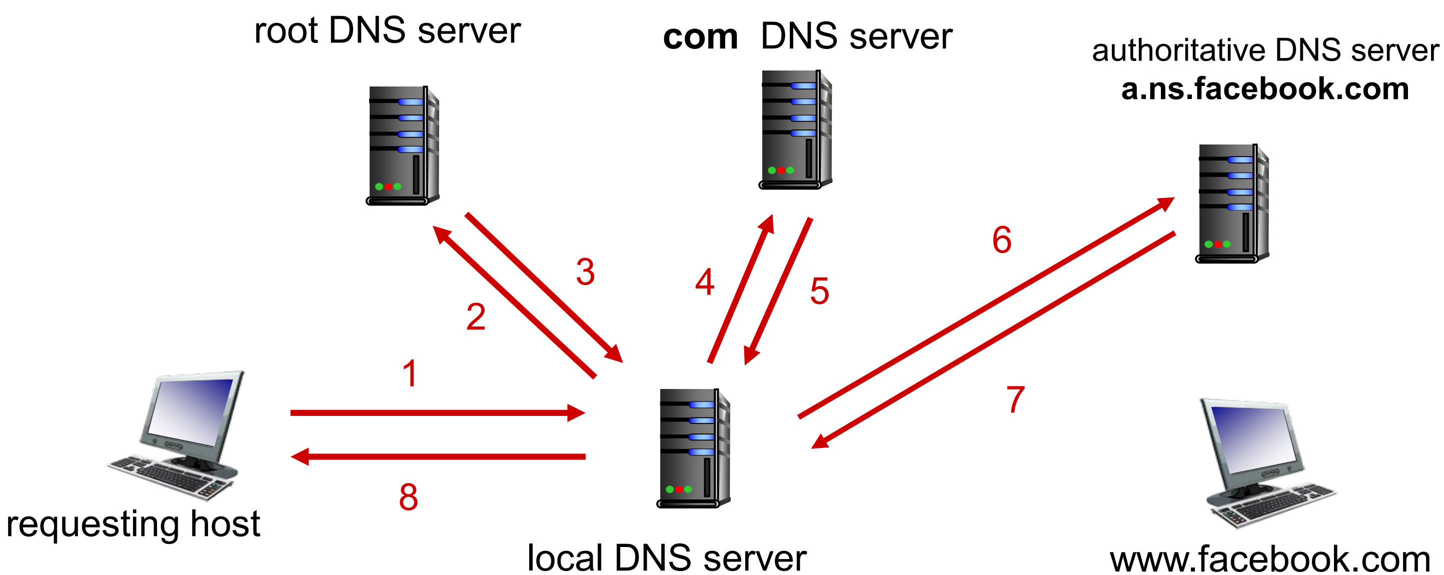
- UDP socket
  - Server uses one socket to serve all clients.
  - No connection is established before sending data.
  - Sender explicitly attaches destination IP address + port #.

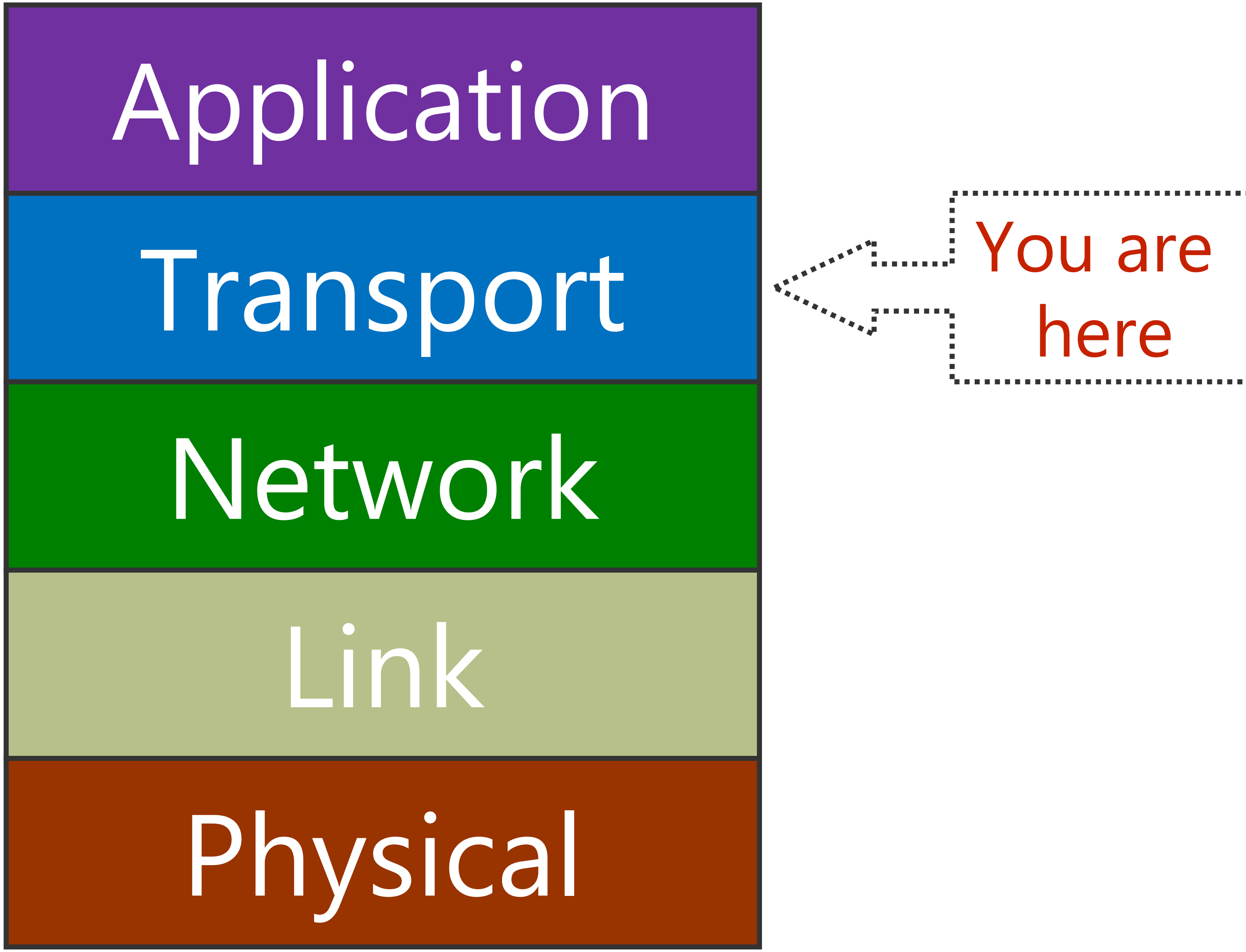
- TCP socket
  - Server creates a new socket for each client.
  - Client establishes connection to server.
  - Server uses connection to identify client.

Domain Name System

PREVIOUS LECTURE

- DNS is the Internet's primary directory service.
  - It translates host names, which can be easily memorized by humans, to numerical IP addresses used by hosts for the purpose of communication.




Application
Transport
Network
Link
Physical
You are here

Lectures 4&5: The Transport Layer

After this class, you are expected to:

- appreciate the simplicity of UDP and the service it provides.
- know how to calculate the checksum of a packet.
- be able to design your own reliable protocols with *ACK*, *NAK*, *sequence number*, *timeout* and *retransmission*.
- understand the working of *Go-Back-N* and *Selective Repeat* protocols.
- understand the operations of TCP.

Lectures 4&5: Roadmap

3.1 Transport-layer Services

3.3 Connectionless Transport: UDP

3.4 Principles of Reliable Data Transfer

3.5 Connection-oriented Transport: TCP

Kurose Textbook, Chapter 3
(Some slides are taken from the book)

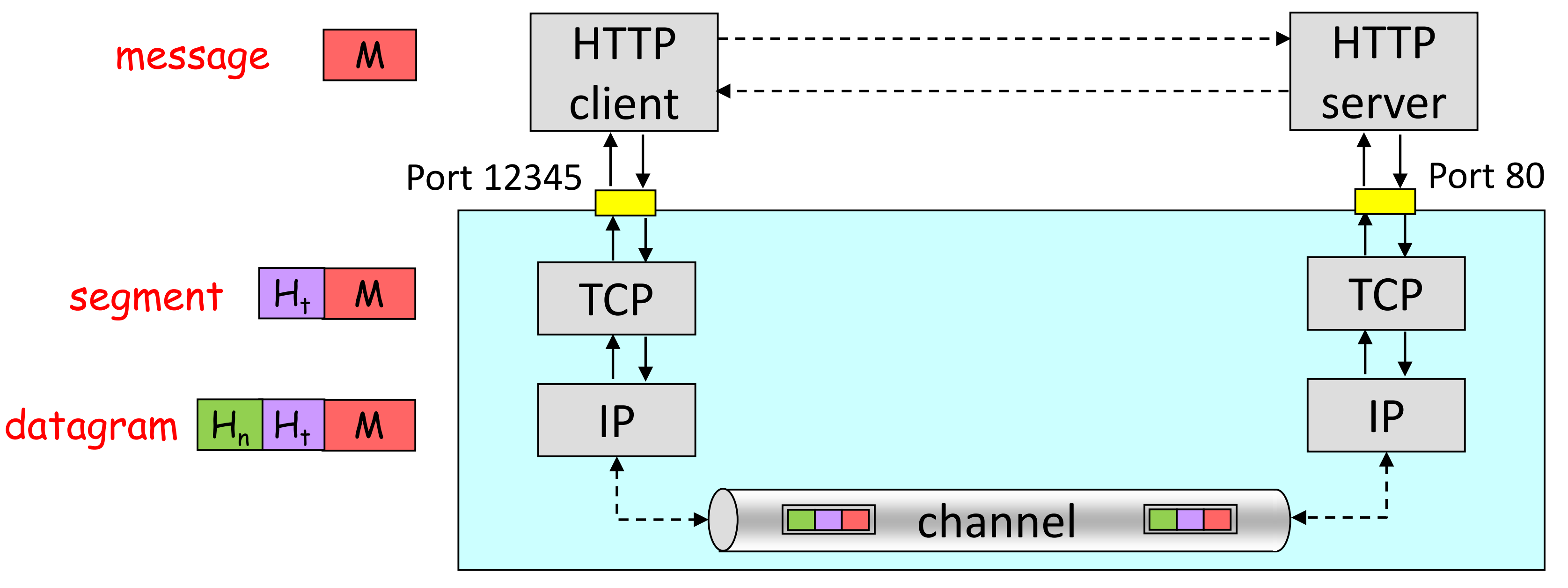
Transport Layer Services

- Deliver messages between application processes running on different hosts
  - Two popular protocols: TCP and UDP

- Transport layer protocols run in hosts.
  - Sender side: breaks app message into *segments* (as needed), passes them to network layer (aka IP layer).
  - Receiver side: reassembles segments into message, passes it to app layer.
  - Packet switches (routers) in between: only check destination IP address to decide routing.

Transport / Network Layers

- Each IP datagram contains source and dest IP addresses.
  - Receiving host is identified by dest IP address.
  - Each IP datagram carries one transport-layer segment.
  - Each segment contains source and dest port numbers.



Lectures 4&5: Roadmap

3.1 Transport-layer Services

3.3 Connectionless Transport: UDP

3.4 Principles of Reliable Data Transfer

3.5 Connection-oriented Transport: TCP

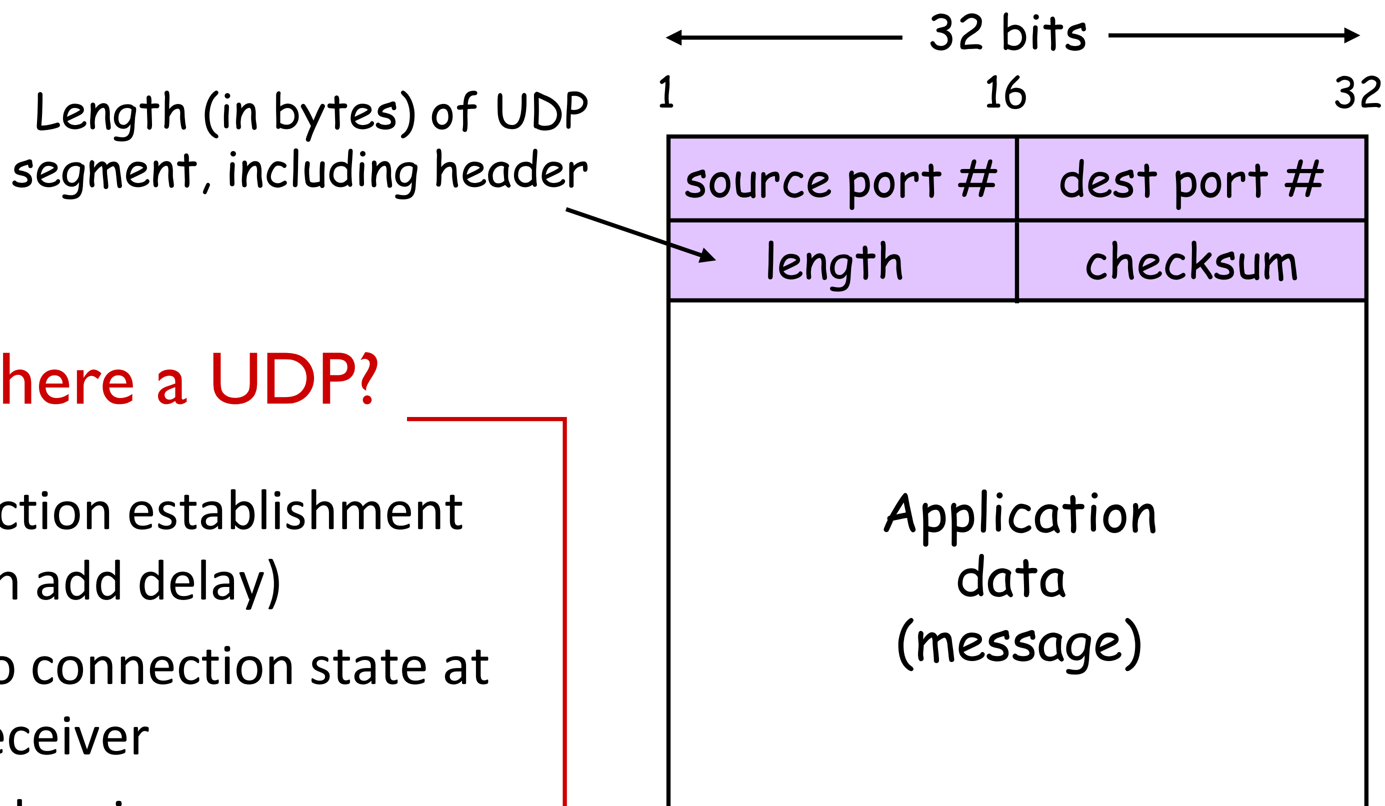
UDP: User Datagram Protocol [RFC 768]

- UDP adds very little service on top of IP:
  - Multiplexing at sender: UDP gathers data from processes, forms packets and passes them to IP
  - De-multiplexing at receiver: UDP receives packets from lower layer and dispatches them to the right processes.
  - Checksum
- UDP transmission is unreliable
  - Often used by streaming multimedia apps (loss tolerant & rate sensitive)

Connectionless De-multiplexing

- When UDP receiver receives a UDP segment:
  - Checks destination port # in segment.
  - Directs UDP segment to the socket with that port #.
  - IP datagrams (from different sources) with the same destination port # will be directed to the same UDP socket at destination.

UDP 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

UDP Checksum

Goal: to detect "errors" (i.e., flipped bits) in transmitted segment.

Sender:

- compute checksum value (next page)
- put 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 really no error?)

Checksum Computation

- How is UDP checksum computed?
  1. Treat UDP segment as a sequence of <u>16-bit</u> integers.
  2. Apply binary addition on every 16-bit integer (checksum field is currently 0).
  3. Carry (if any) from the most significant bit will be added to the result.
  4. Compute 1's complement to get UDP checksum.

x	y	x $\oplus$ y	carry
0	0	0	–
0	1	1	–
1	0	1	–
1	1	0	1

Checksum Example

- Example: add two 16-bit integers

carry

		1 1 1 0	0 1 1 0	0 1 1 0	0 1 1 0
wraparound carry		1 1 0 1	0 1 0 1	0 1 0 1	0 1 0 1
	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 (1's complement)		0 1 0 0	0 1 0 0	0 1 0 0	0 0 1 1

Lectures 4&5: Roadmap

3.1 Transport-layer Services

3.3 Connectionless Transport: UDP

3.4 Principles of Reliable Data Transfer

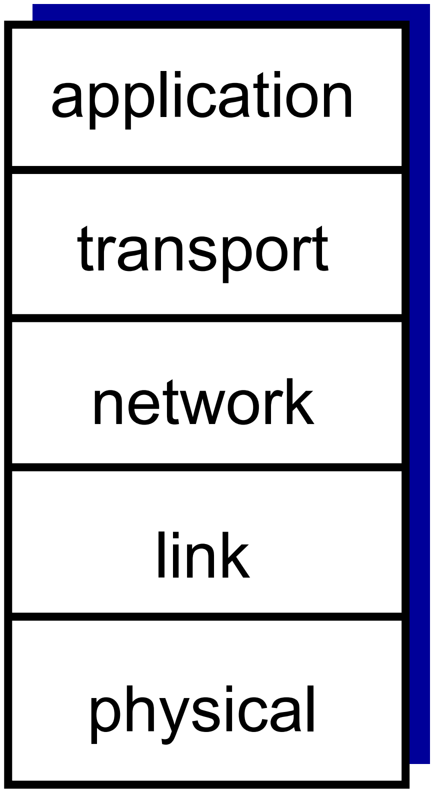
3.5 Connection-oriented transport: TCP

"Sending Data Reliably Over the Internet is Much Harder Than You Think. The Intricacy Involved in Ensuring Reliability Will Make Your Head Explode."

Transport vs. Network Layer

- **Transport layer** resides on end hosts and provides process-to-process communication.
- **Network layer** provides host-to-host, best-effort and unreliable communication.

Question: How to build a **reliable transport layer protocol** on top of **unreliable communication**?

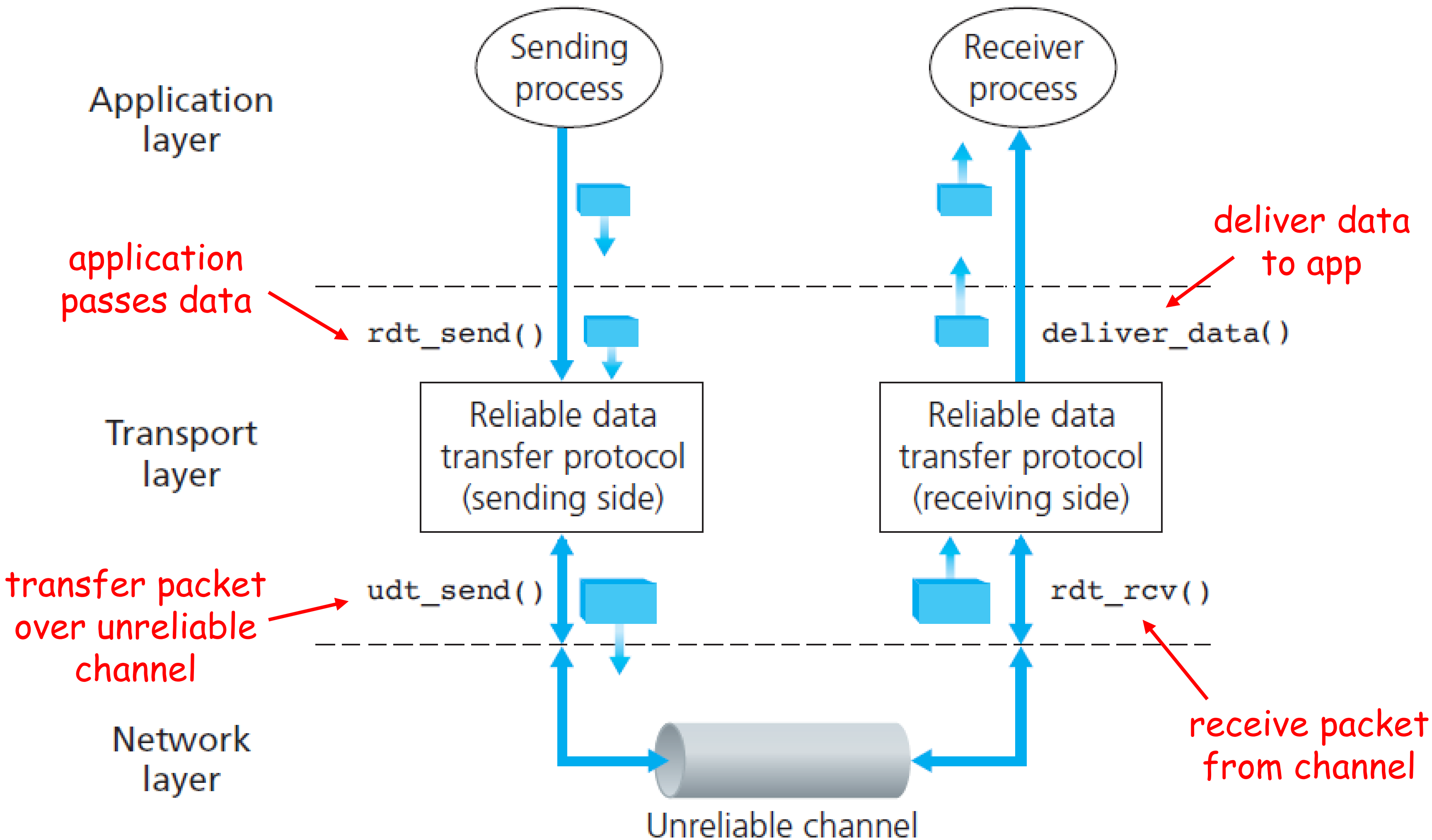


Reliable Transfer over Unreliable Channel

- Underlying network may
  - corrupt packets
  - drop packets
  - re-order packets (not considered in this lecture)
  - deliver packets after an arbitrarily long delay

- End-to-end reliable transport service should
  - guarantee packets delivery and correctness
  - deliver packets (to receiver application) in the same order they are sent

Reliable Data Transfer: Service Model

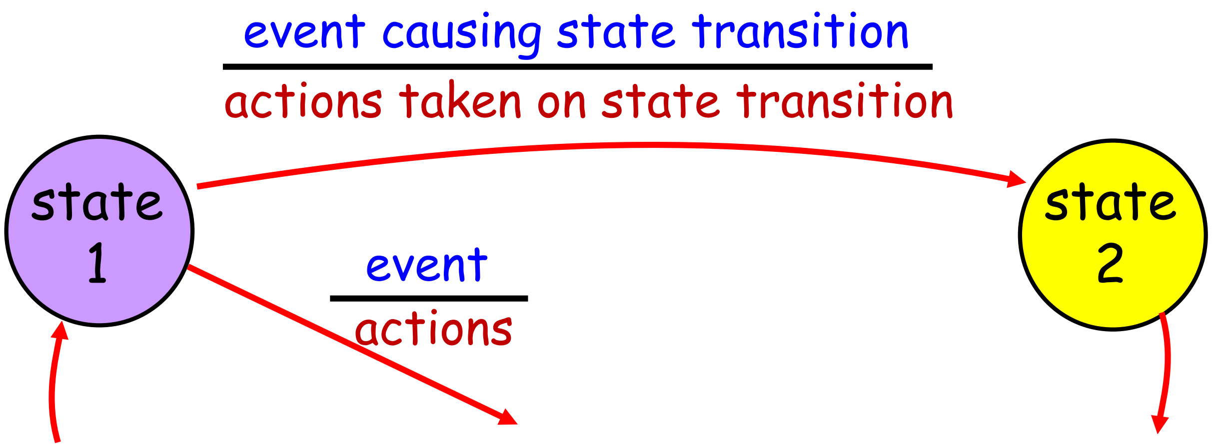


Reliable Data Transfer Protocols

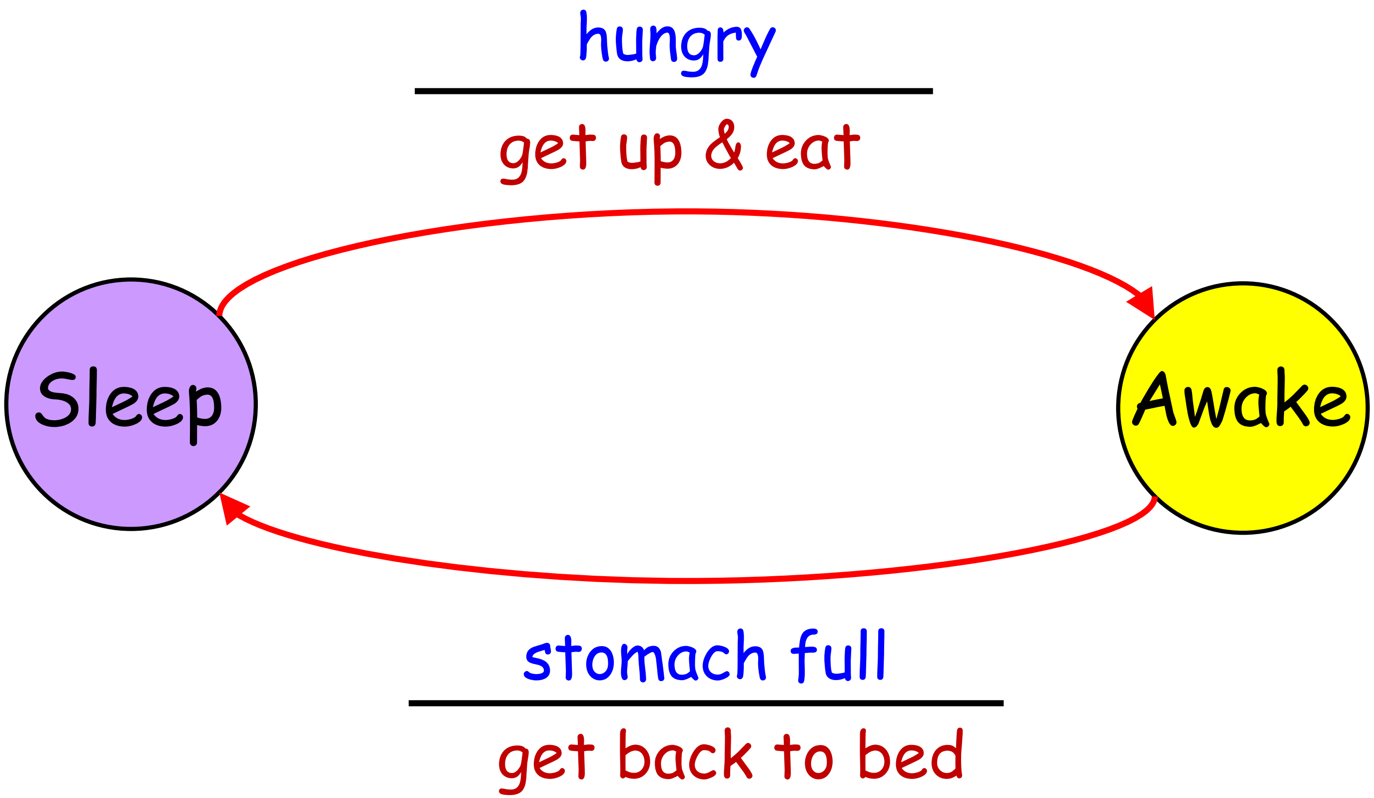
- Characteristics of unreliable channel will determine the complexity of reliable data transfer protocols (rdt).
- We will incrementally develop sender & receiver sides of rdt protocols, considering increasingly complex models of unreliable channel.
- We consider only unidirectional data transfer
  - but control info may flow in reverse direction!

Finite State Machine (FSM)

- We will use finite state machines (FSM) to describe sender and receiver of a protocol.
  - We will learn a protocol by examples, but FSM provides you the complete picture to refer to as necessary.

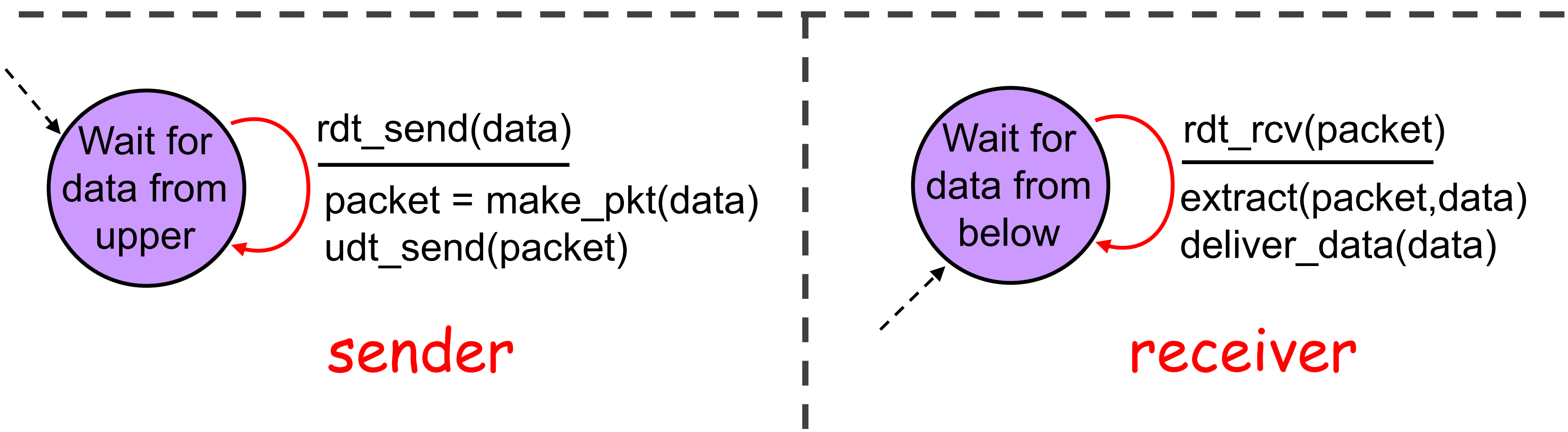


Example FSM



rdt 1.0: Perfectly Reliable Channel

- Assume underlying channel is perfectly reliable.
- Separate FSMs for sender, receiver:
  - Sender sends data into underlying (perfect) channel
  - Receiver reads data from underlying (perfect) channel

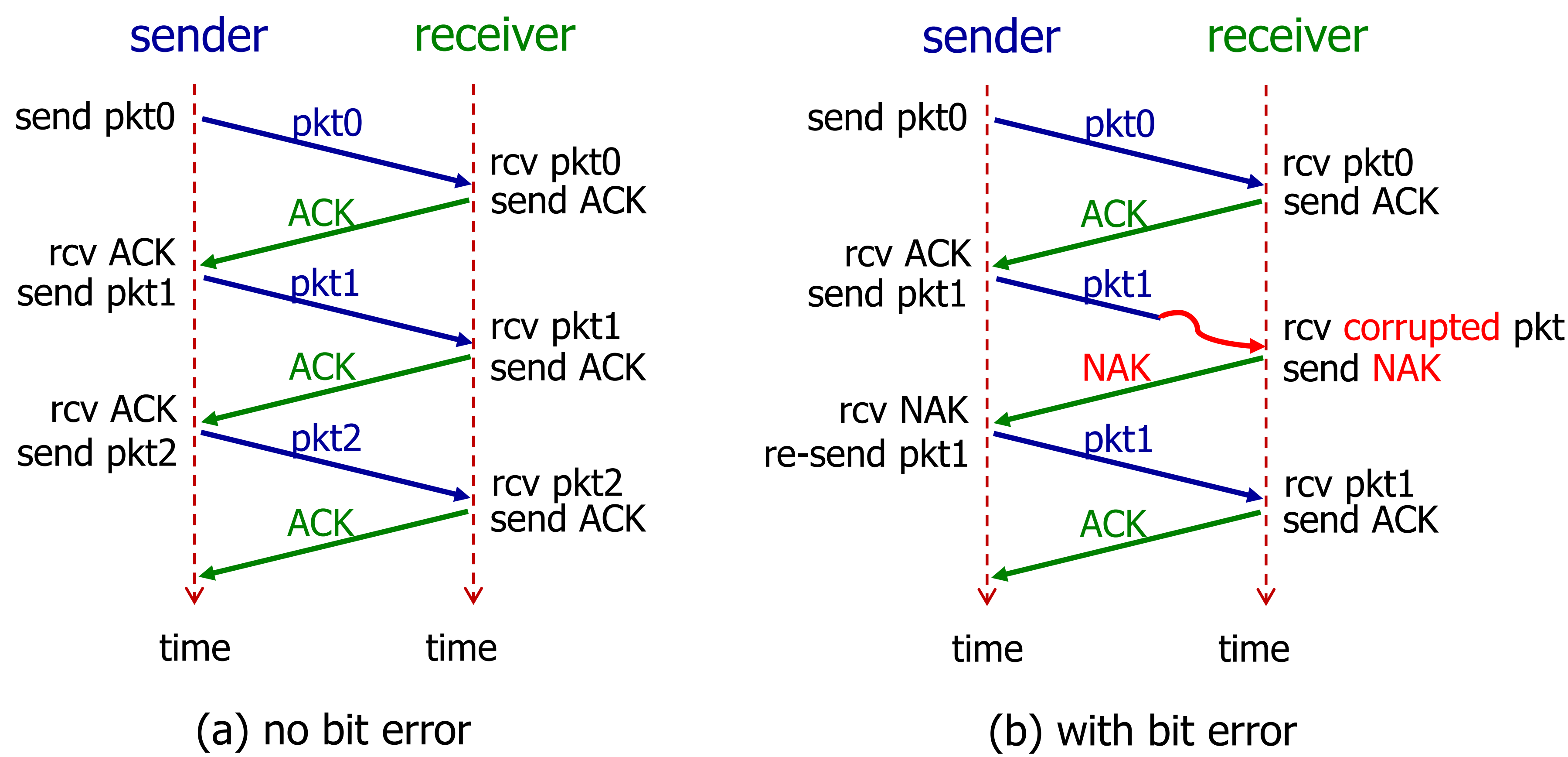


rdt 2.0: Channel with *Bit Errors*

- Assumption:
  - underlying channel may flip bits in packets
  - other than that, the channel is perfect

- Q1: how to <u>detect</u> bit errors?
  - Receiver may use ***checksum*** to <u>detect</u> bit errors.
- Q2: how to <u>recover</u> from bit errors?
  - *Acknowledgements (ACKs):* receiver explicitly tells sender that packet received is OK.
  - *Negative acknowledgements (NAKs):* receiver explicitly tells sender that packet has errors.
    - Sender retransmits packet on receipt of NAK.

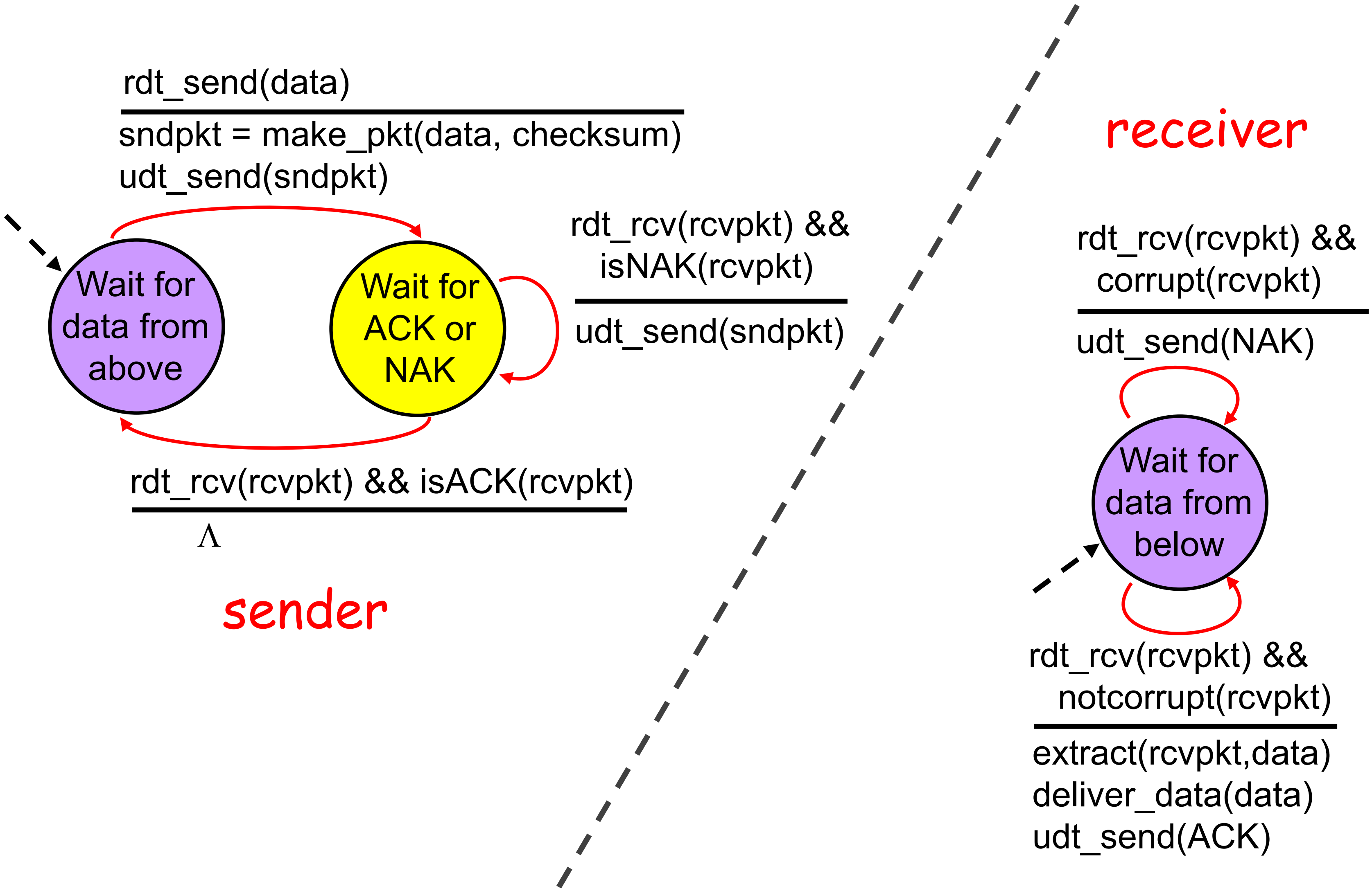
rdt 2.0 In Action



stop and wait protocol

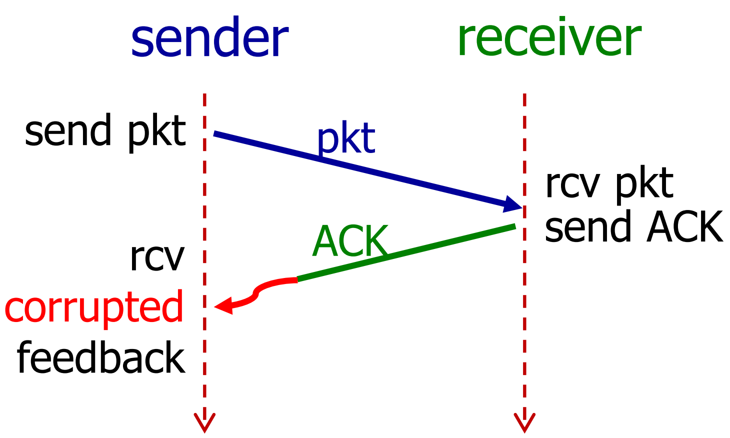
Sender sends one packet at a time, then waits for receiver response

rdt 2.0: FSM

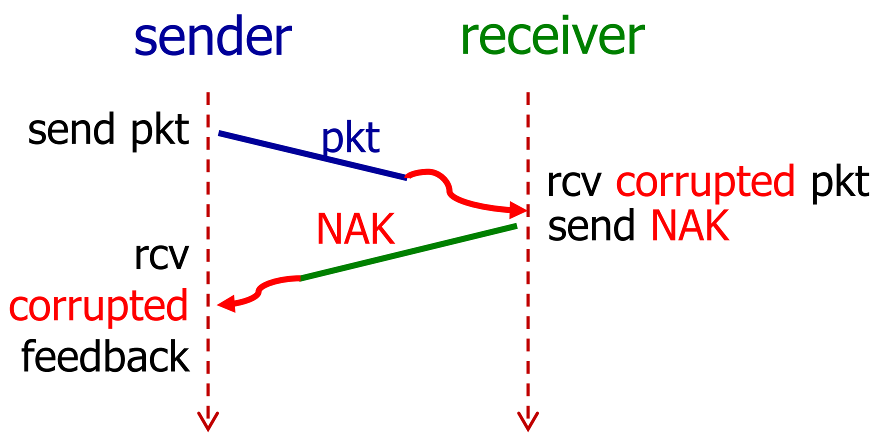

rdt_send(data)
sndpkt = make_pkt(data, checksum)
udt_send(sndpkt)
Wait for data from above
Wait for ACK or NAK
rdt_rcv(rcvpkt) && isNAK(rcvpkt)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && isACK(rcvpkt)
Λ
sender
receiver
rdt_rcv(rcvpkt) && corrupt(rcvpkt)
udt_send(NAK)
Wait for data from below
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
udt_send(ACK)

rdt 2.0 has a Fatal Flaw!

- What happens if ACK/NAK is corrupted?
  - Sender doesn't know what happened at receiver!
- So what should the sender do?
  - Sender just retransmits when receives garbled ACK or NAK.
  - Questions: does this work?



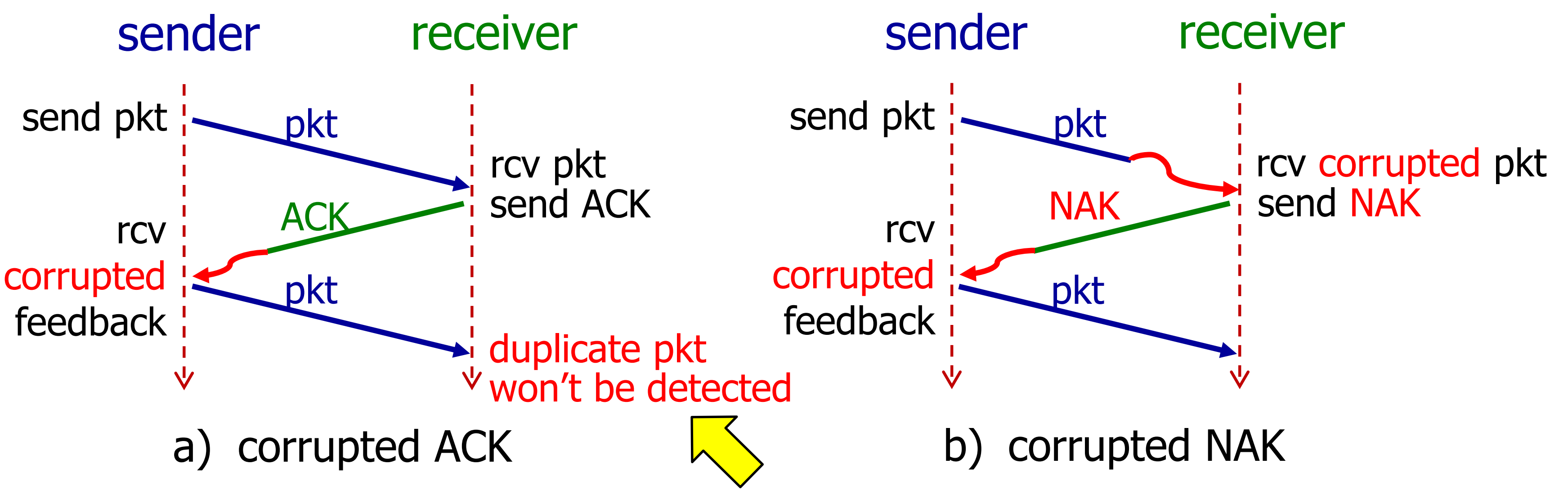
a) corrupted ACK



b) corrupted NAK

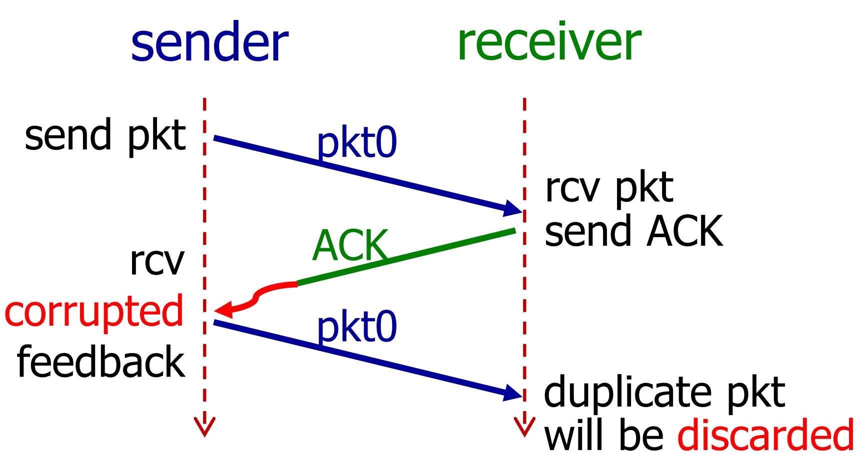
rdt 2.0 has a Fatal Flaw!

- Sender just retransmits when it receives garbled feedback.
  - This may cause retransmission of correctly received packet!
  - Question: how can receiver identify duplicate packet?

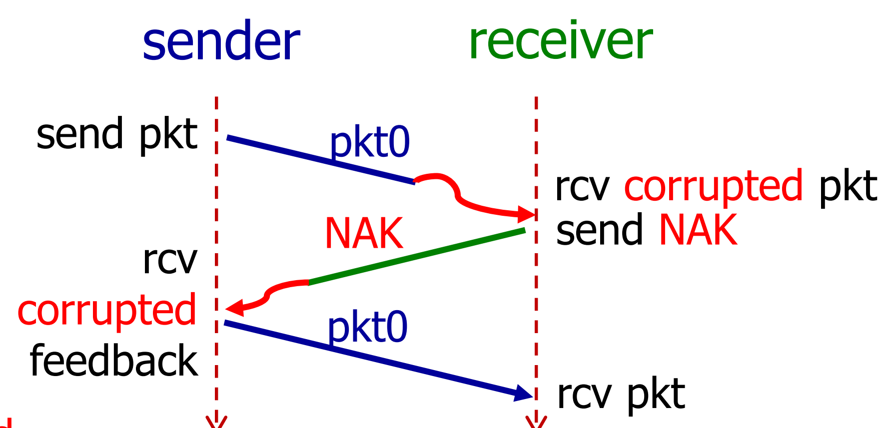


rdt 2.1: rdt 2.0 + Packet Seq.

- To handle duplicates:
  - Sender retransmits current packet if ACK/NAK is garbled.
  - Sender adds *sequence number* to each packet.
  - Receiver discards (doesn't deliver up) duplicate packet.
- This gives rise to protocol rdt 2.1.

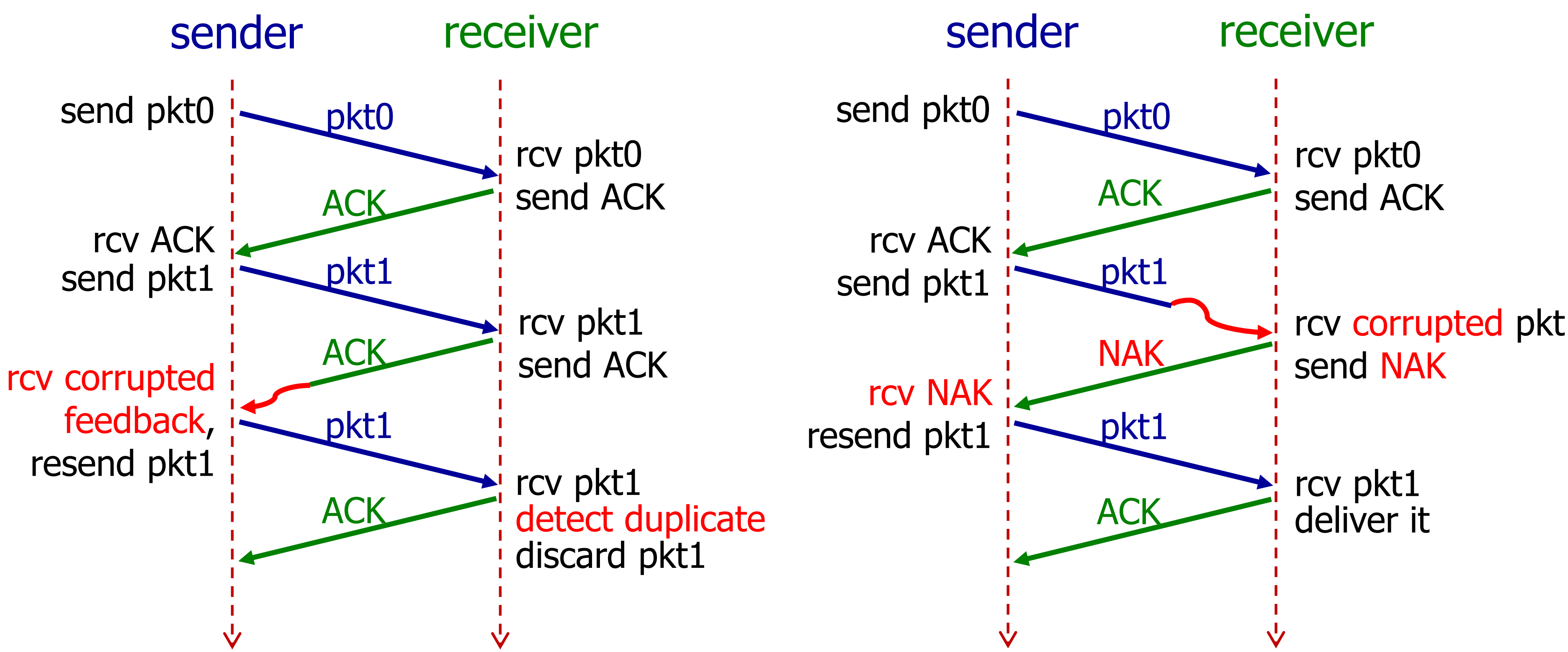


a) corrupted ACK



b) corrupted NAK

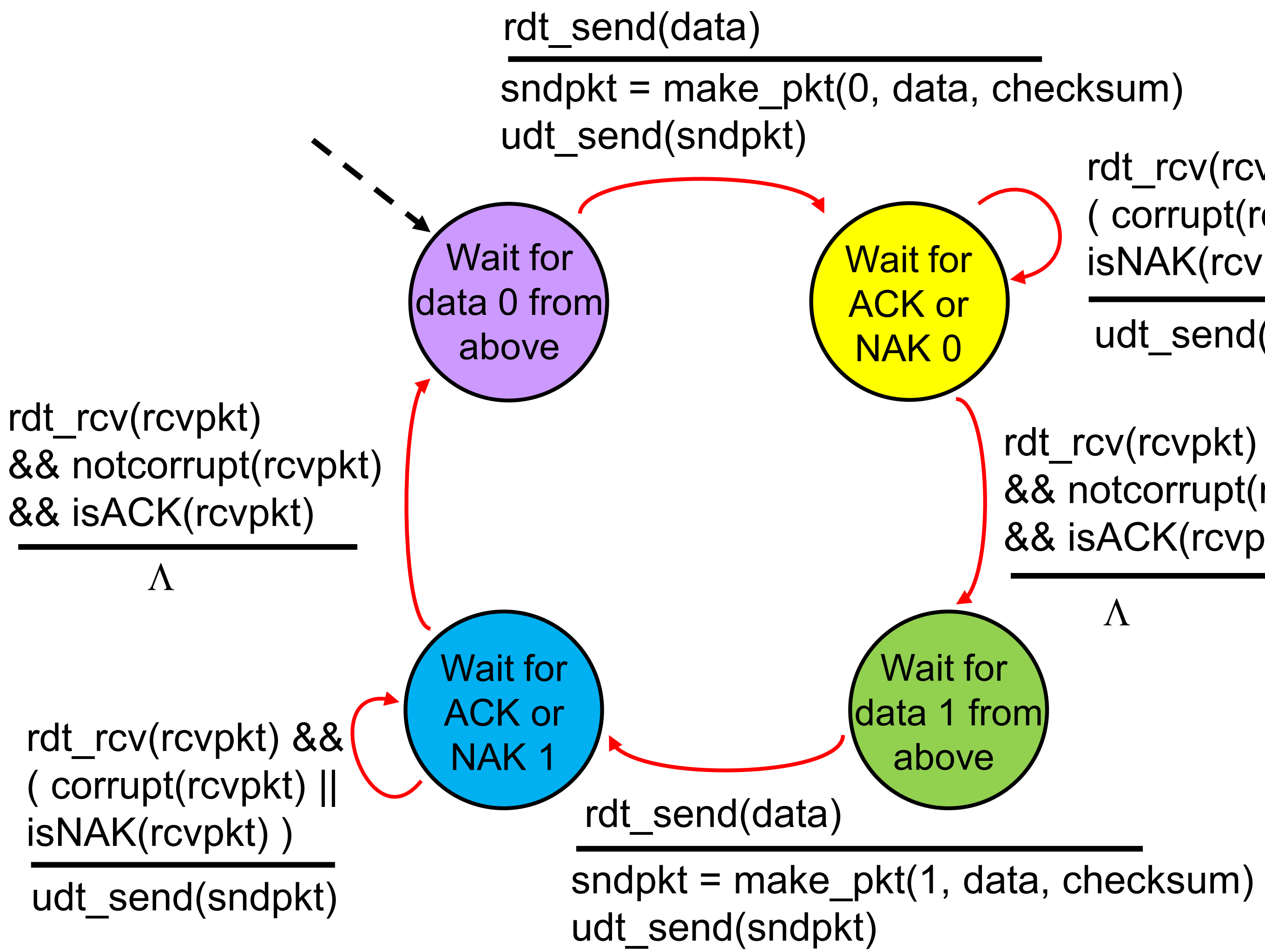
rdt 2.1 In Action



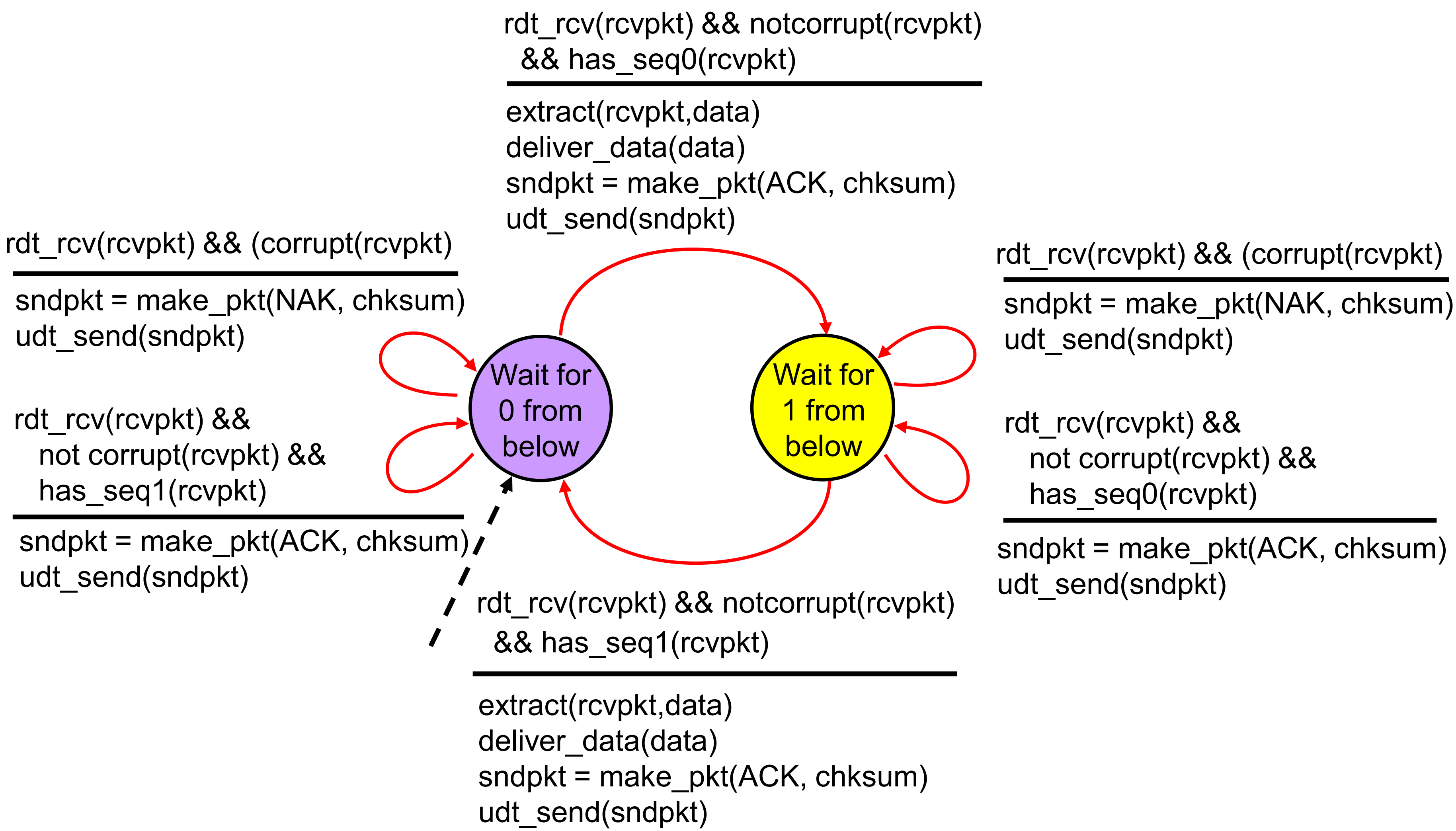
a) resend due to corrupted ACK

b) resend due to corrupted packet

rdt 2.1 Sender FSM



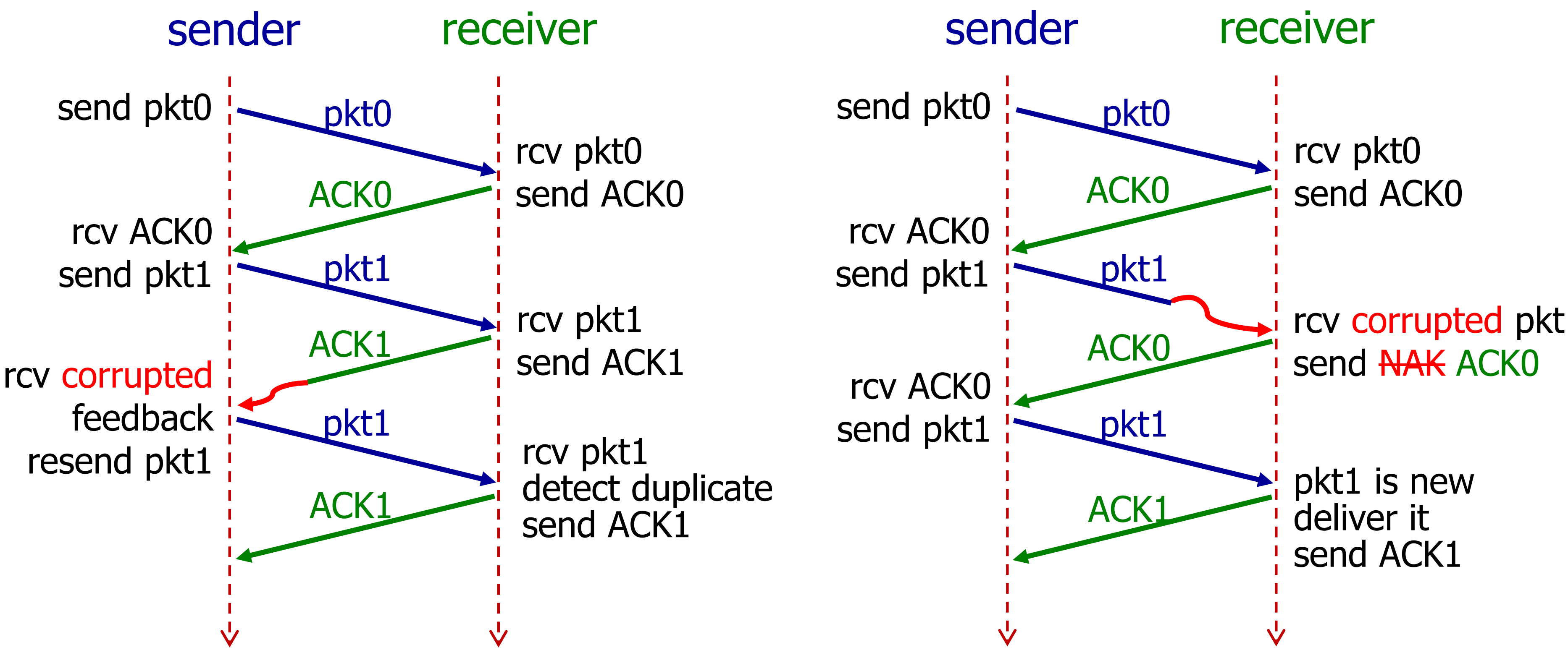
rdt 2.1 Receiver FSM



rdt 2.2: a NAK-free Protocol

- Same assumption and functionality as rdt 2.1, but use ACKs only.
- Instead of sending NAK, receiver sends ACK for the last packet received OK.
  - Now receiver must *explicitly* include seq. # of the packet being ACKed.
- Duplicate ACKs at sender results in same action as NAK: *retransmit current pkt*.

rdt 2.2 In Action



a) resend due to corrupted ACK

b) resend due to duplicate ACK

rdt 3.0: Channel with *Errors* and *Loss*

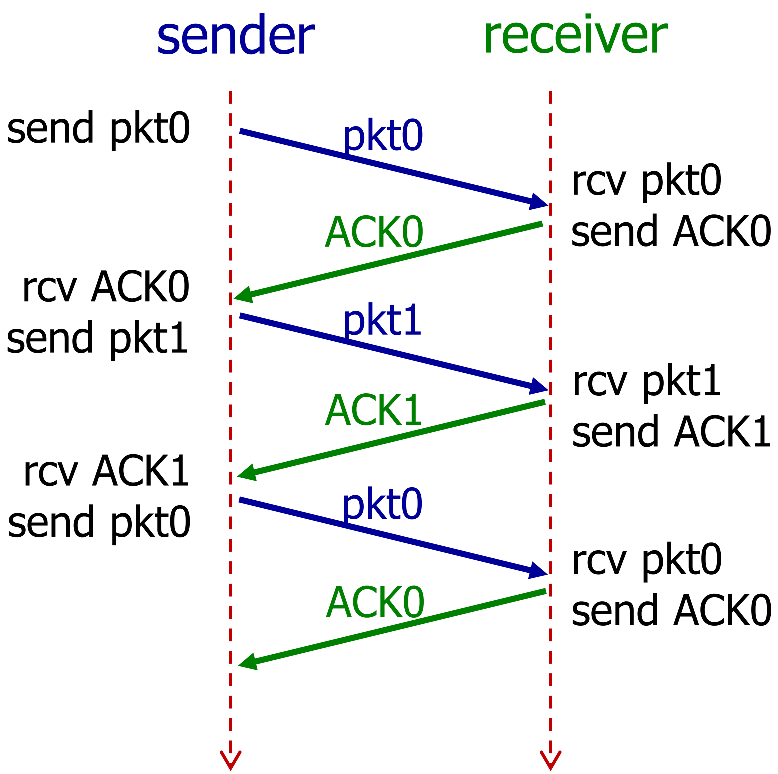
- Assumption: underlying channel
  - may flip bits in packets
  - may lose packets
  - may incur arbitrarily long packet delay
  - but won't re-order packets

- Question: how to detect packet loss?
  - checksum, ACKs, seq. #, retransmissions will be of help… but not enough

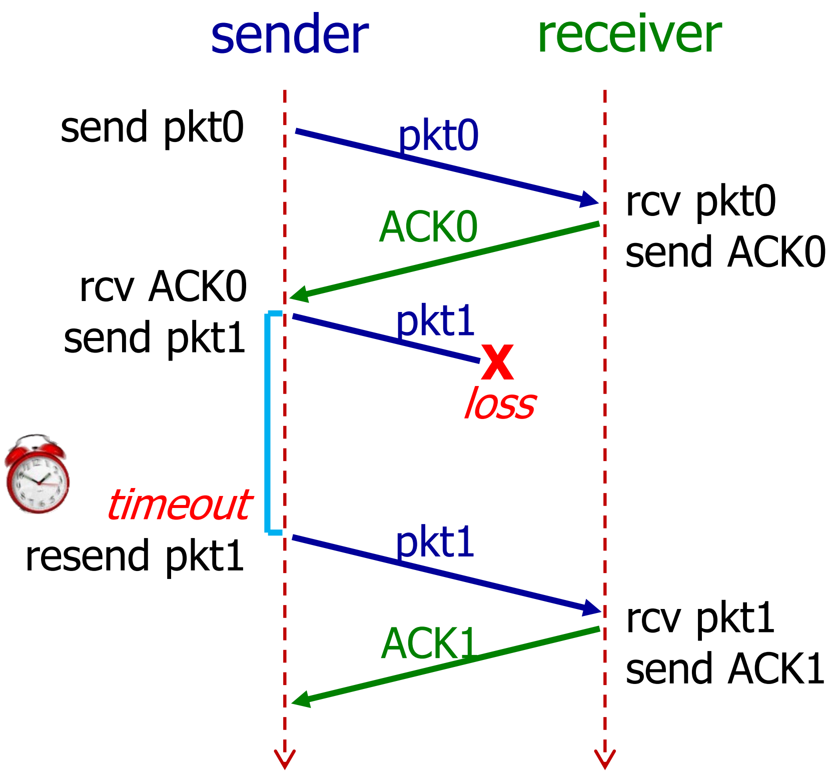
rdt 3.0: Channel with *Errors* and *Loss*

- To handle packet loss:
  - Sender waits "reasonable" amount of time for ACK.
  - Sender retransmits if no ACK is received till *timeout*.
- Question: what if packet (or ACK) is just delayed, but not lost?
  - Timeout will trigger retransmission.
  - Retransmission will generate duplicates in this case, but receiver may use seq. # to detect it.
  - Receiver must specify seq. # of the packet being ACKed (check scenario (d) two pages later).

rdt 3.0 In Action

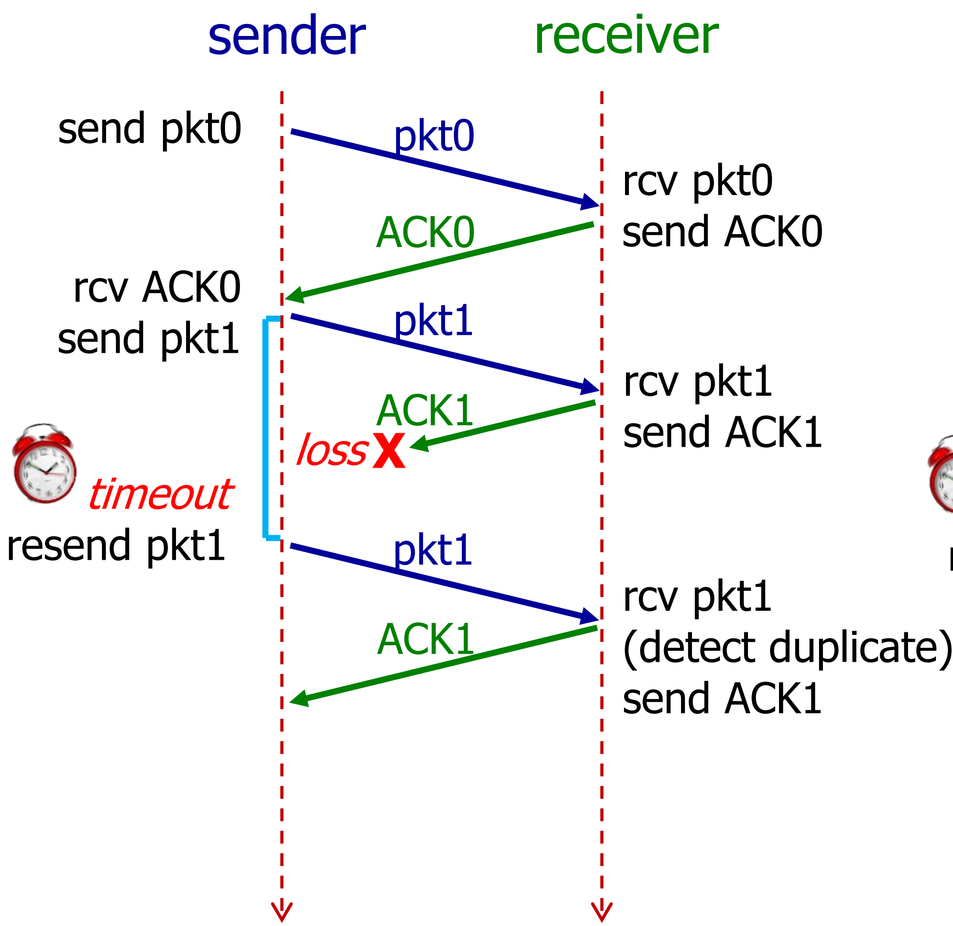


a) no packet loss

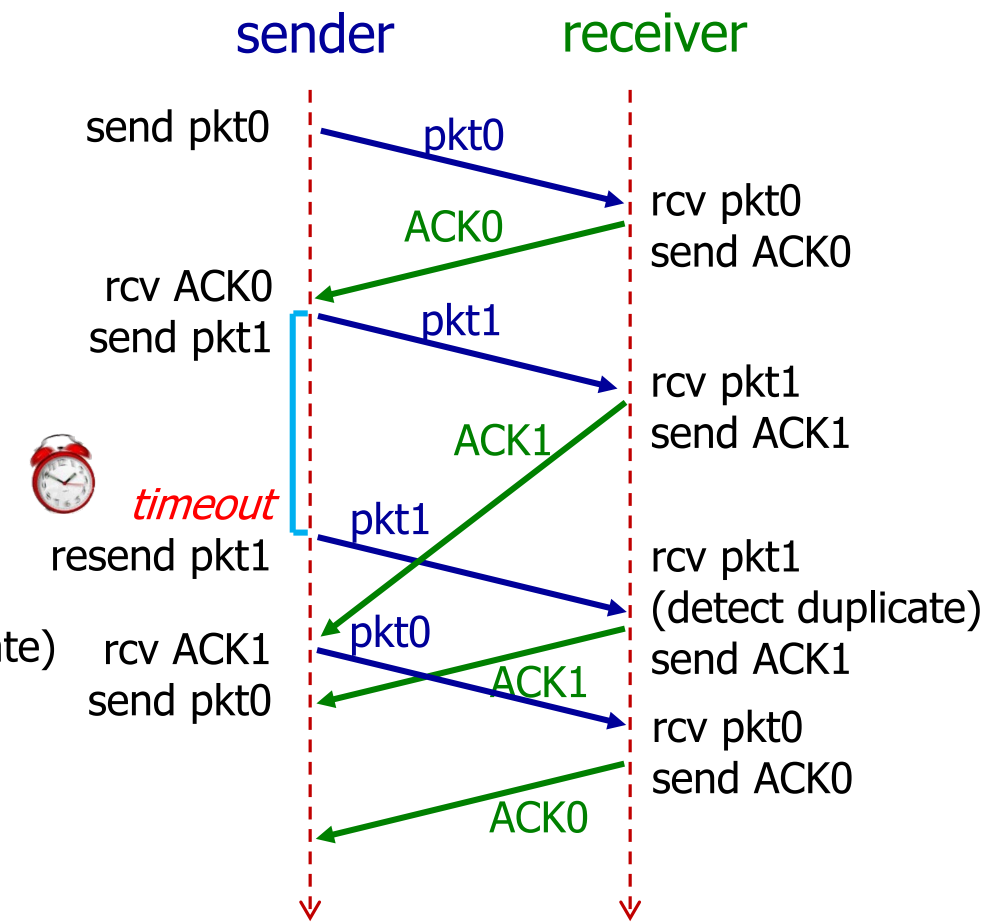


b) packet loss

rdt 3.0 In Action

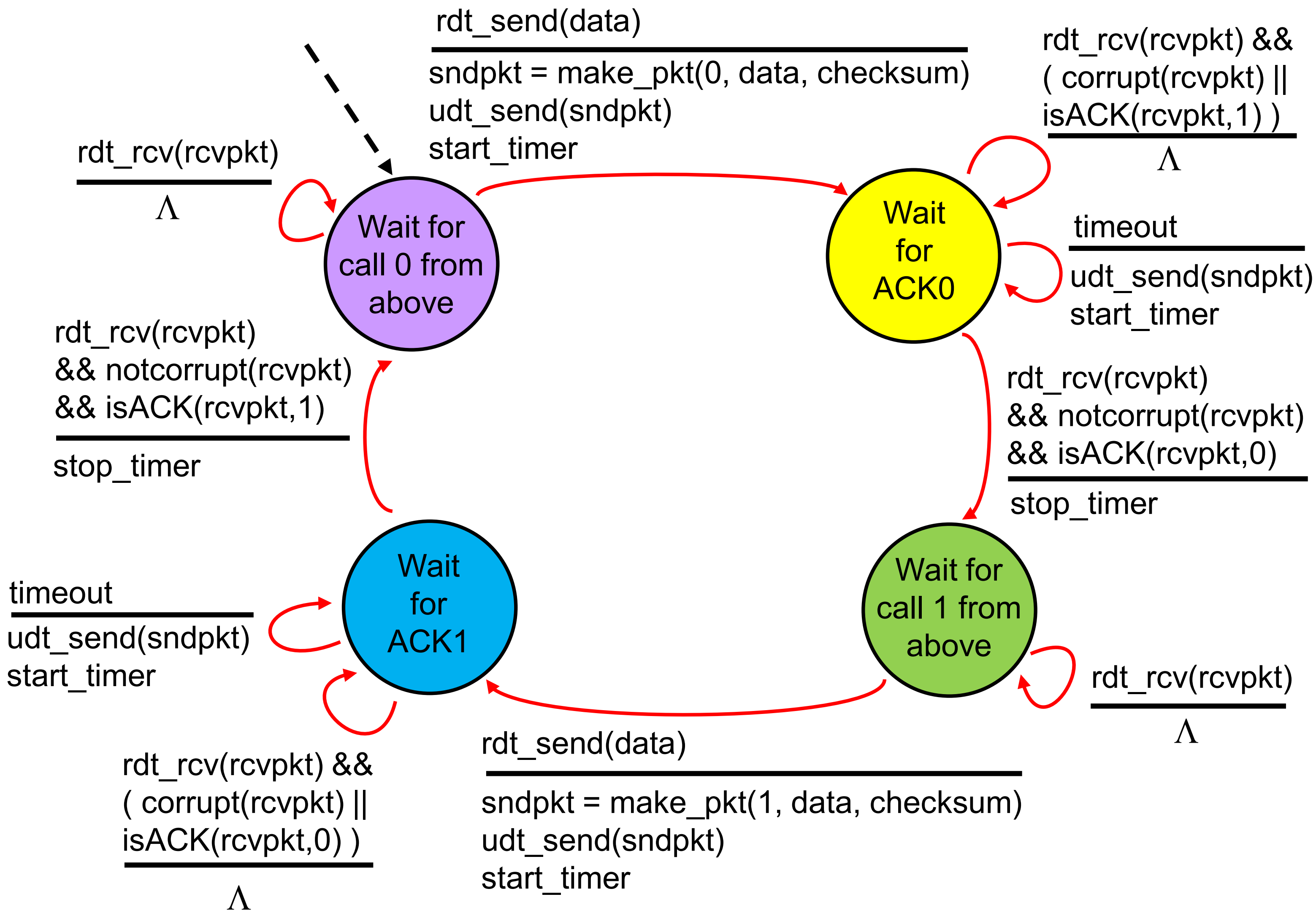


c) lost ACK



d) premature timeout / delayed ACK

rdt 3.0 Sender FSM



RDT Summary

rdt Version	Scenario	Features Used
1.0	no error	nothing
2.0	data Bit Error	checksum, ACK/NAK
2.1	data Bit Error ACK/NAK Bit Error	checksum, ACK/NAK, sequence Number
2.2	Same as 2.1	NAK free
3.0	data Bit Error ACK Bit Error packet Loss	checksum, ACK, sequence Number, timeout/re-transmission

Performance of rdt 3.0

- rdt 3.0 works, but performance stinks.
- Example: packet size = 8000 bits, link rate = 1 Gbps:

$$d_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 0.008 \text{ msec}$$

  - If RTT = 30 msec, sender sends 8000 bits every 30.008 msec.

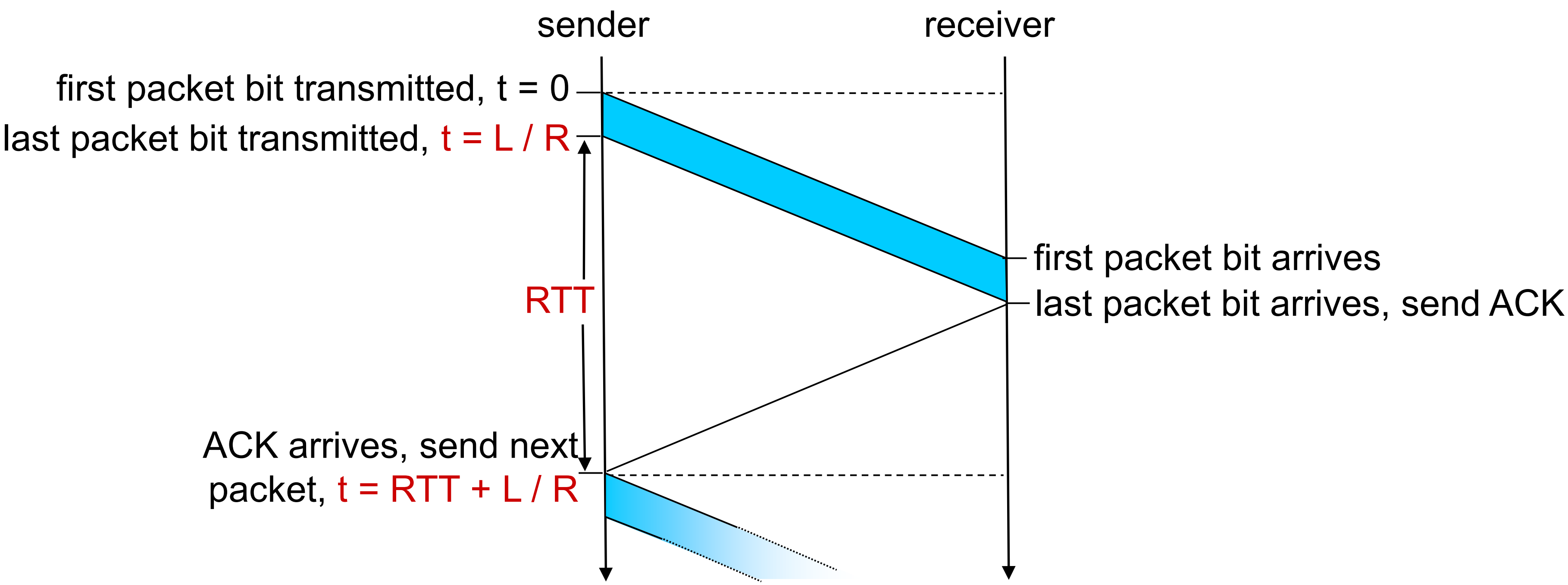
$$\text{throughput} = \frac{L}{RTT + d_{trans}} = \frac{8000}{30.008} = 267 \text{ kbps}$$

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

$$U_{sender} = \frac{d_{trans}}{RTT + d_{trans}} = \frac{0.008}{30 + 0.008} = 0.00027$$

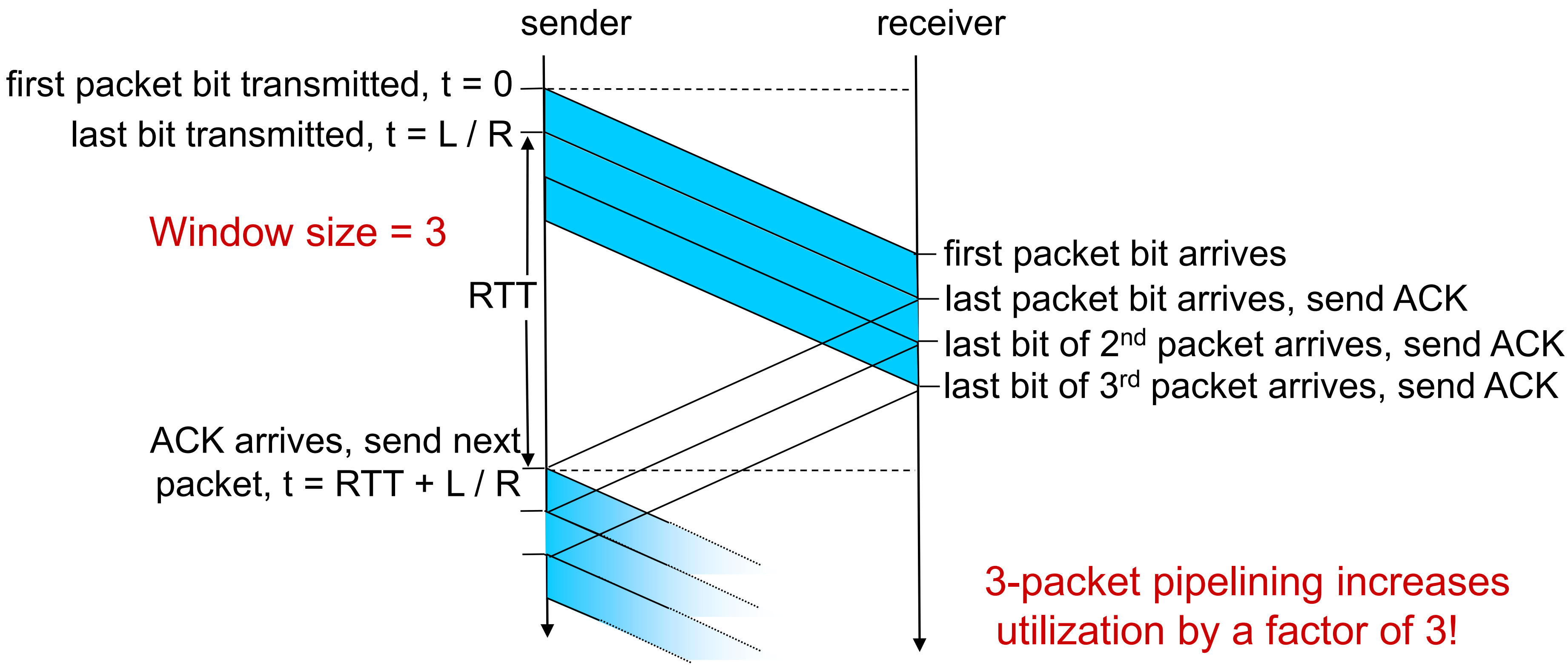
rdt 3.0: Stop-and-wait Operation

- Network protocol limits use of physical resources!



$$U_{sender} = \frac{L\ /\ R}{RTT + L/R} = \frac{0.008}{30 + 0.008} = 0.00027$$

Pipelining: Increased Utilization



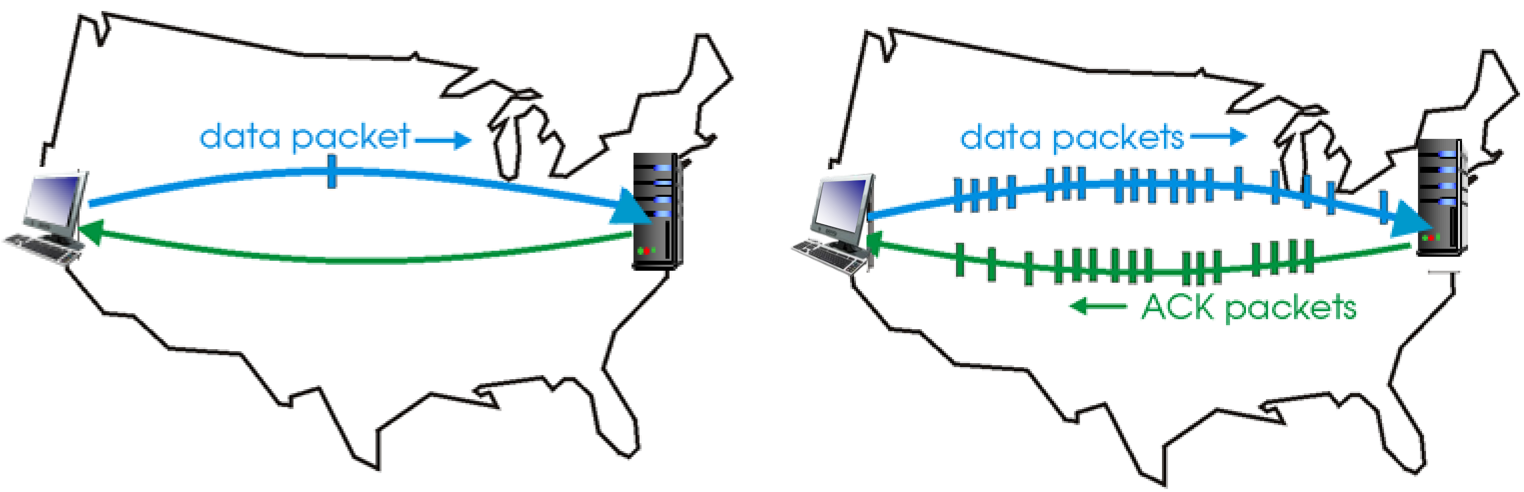
3-packet pipelining increases utilization by a factor of 3!

$$U_{sender} = \frac{3 * L / R}{RTT + L/R} = \frac{0.024}{30 + 0.008} = 0.00081$$

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



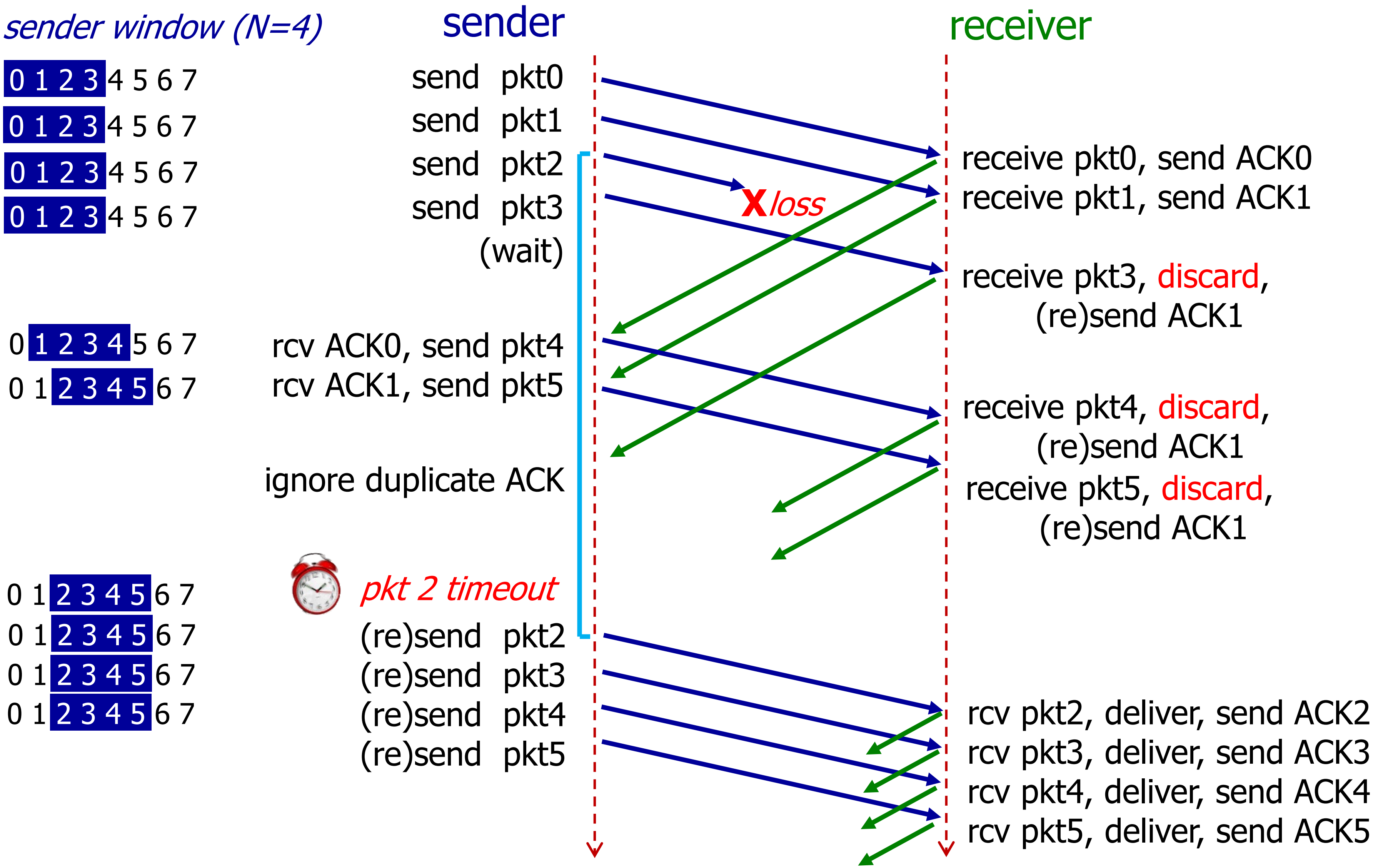
(a) a stop-and-wait protocol in operation (b) a pipelined protocol in operation

Benchmark Pipelined Protocols

- Two generic forms of pipelined protocols:
  - *Go-Back-N (GBN)*
  - *Selective repeat (SR)*

- Assumption (same as rdt 3.0): underlying channel
  - may flip bits in packets
  - may lose packets
  - may incur arbitrarily long packet delay
  - but won't re-order packets

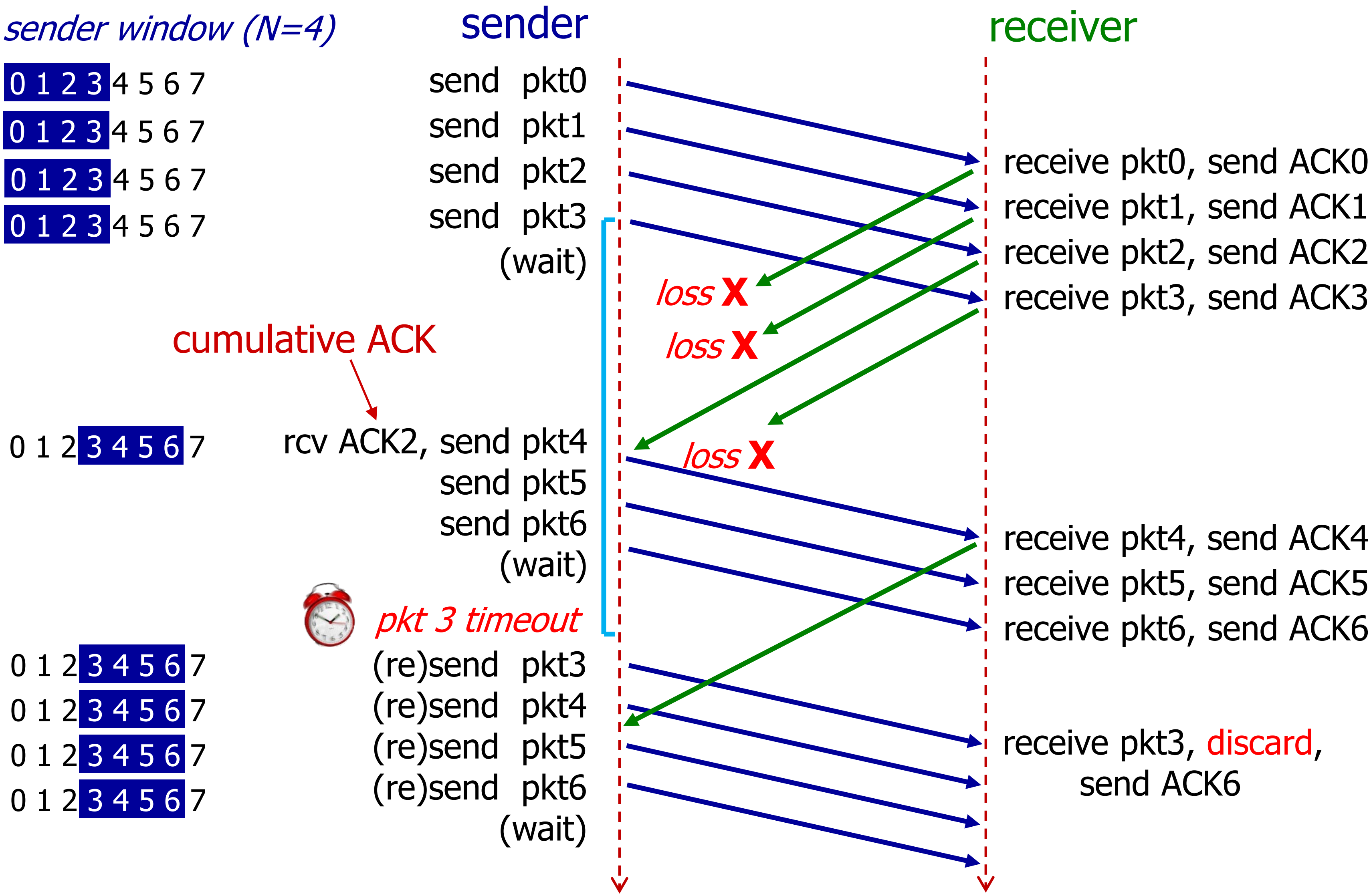
Go-back-N In Action



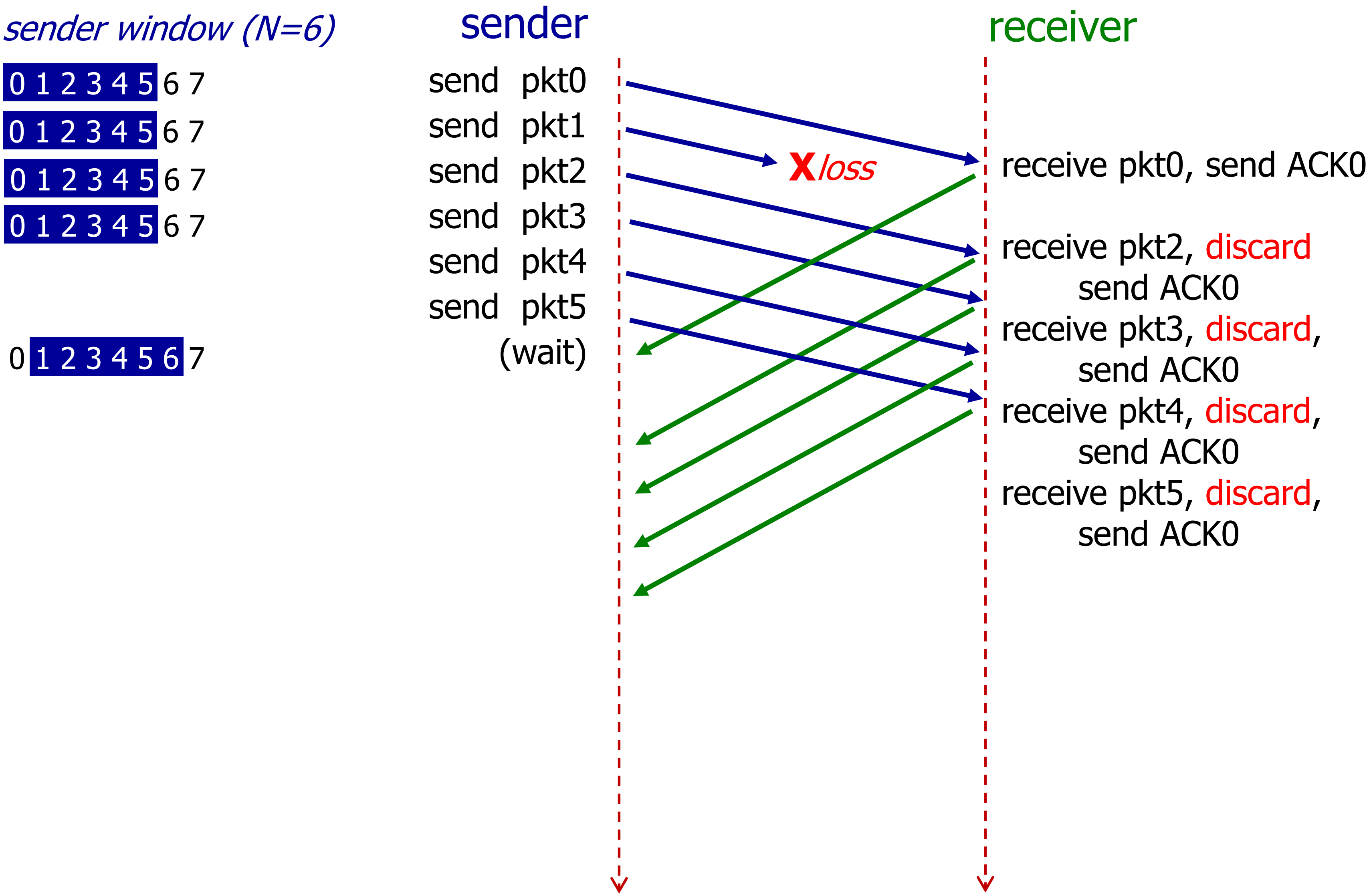
Go-back-N: Key Features

- GBN Sender
  - can have up to *N* unACKed packets in pipeline.
  - insert k-bits sequence number in packet header.
  - use a "sliding window" to keep track of unACKed packets.
  - keep a timer for the oldest unACKed packet.
  - *timeout(n):* retransmit packet *n* and all subsequent packets in the window.
- GBN Receiver
  - only ACK packets that arrive in order.
    - simple receiver: need only remember **`expectedSeqNum`**
  - discard out-of-order packets and ACK the last in-order seq. #.
    - *Cumulative ACK*: "*ACK m*" means all packets up to *m* are received.

Go-back-N In Action



Go-back-N In Action

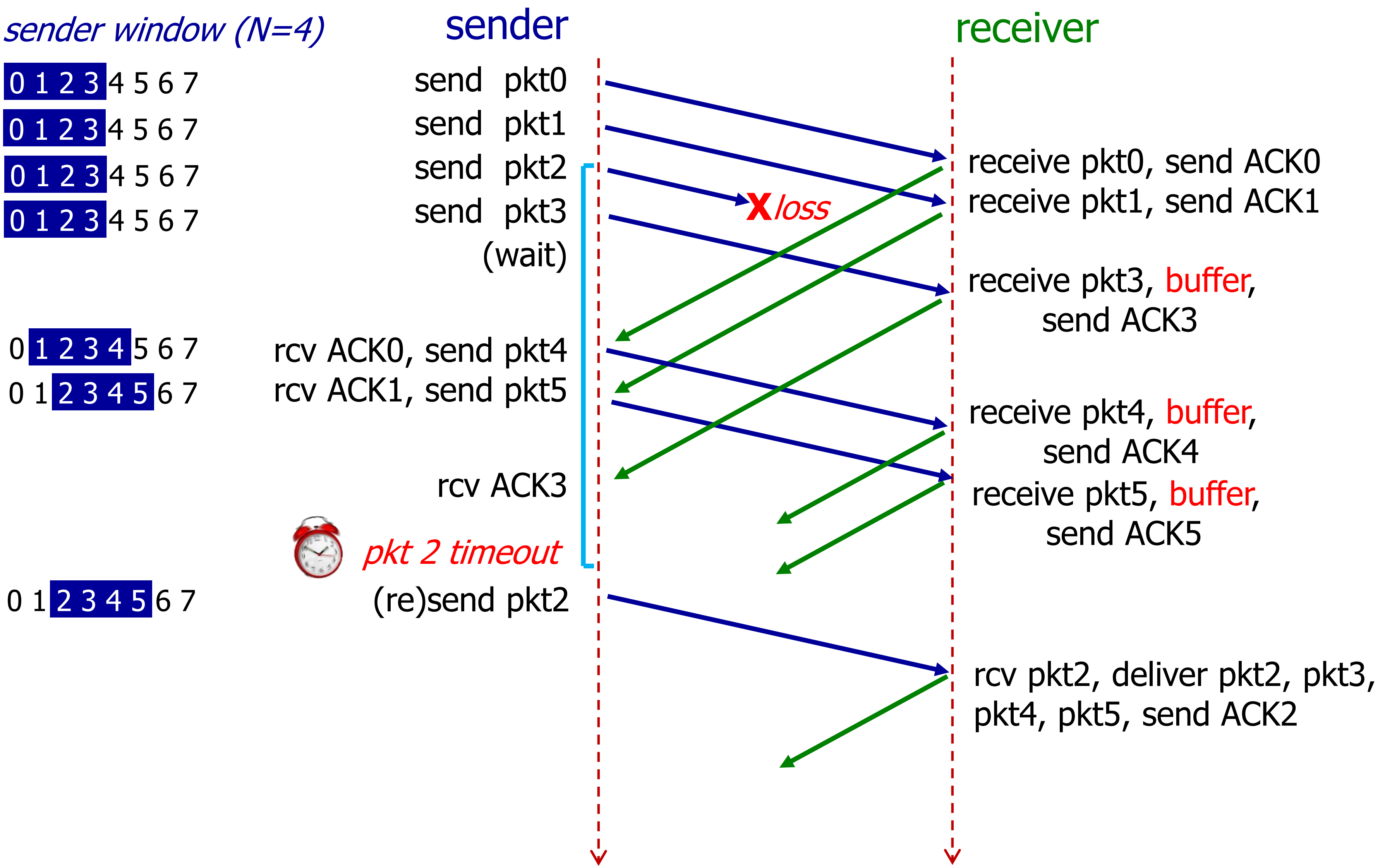


Selective Repeat: Key Features

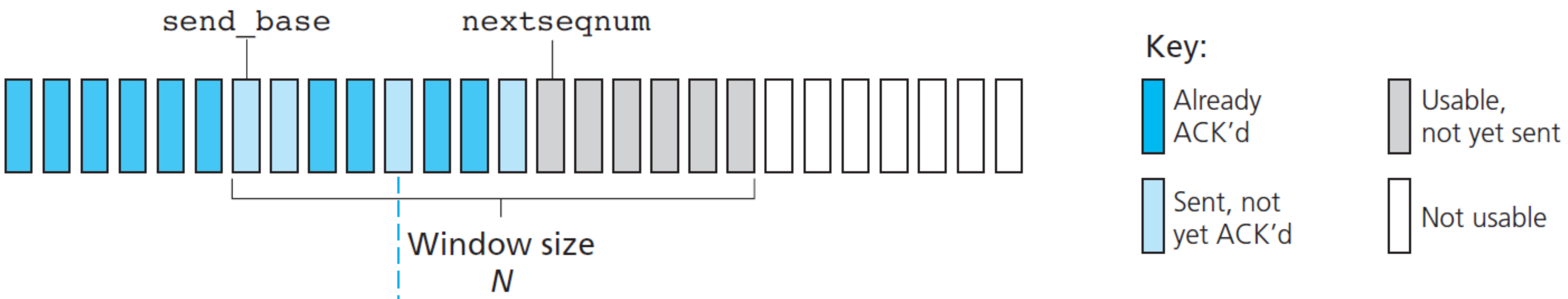
- Receiver *individually* acknowledges all correctly received packets.
  - Buffers out-of-order packets, as needed, for eventual in-order delivery to upper layer.

- Sender maintains timer for each unACKed packet.
  - When timer expires, retransmit only that unACKed packet.

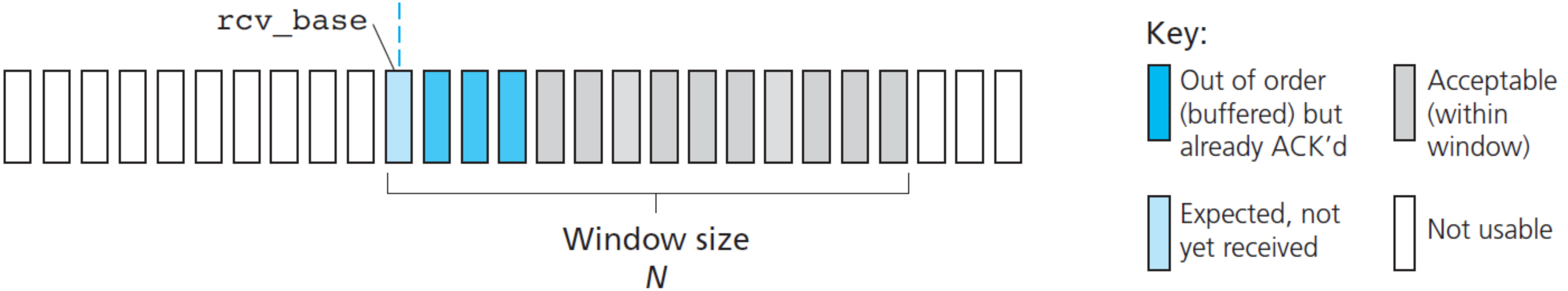
Selective Repeat In Action



SR Sender and Receiver Windows



a. Sender view of sequence numbers



b. Receiver view of sequence numbers

Selective Repeat: Behaviors

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 is 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

Lectures 4&5: Roadmap

TCP: Transport Control Protocol

- In contrast to UDP, TCP is complex and is described in tens of RFCs, with new mechanisms or tweaks introduced throughout the years, resulting in many variants of TCP.

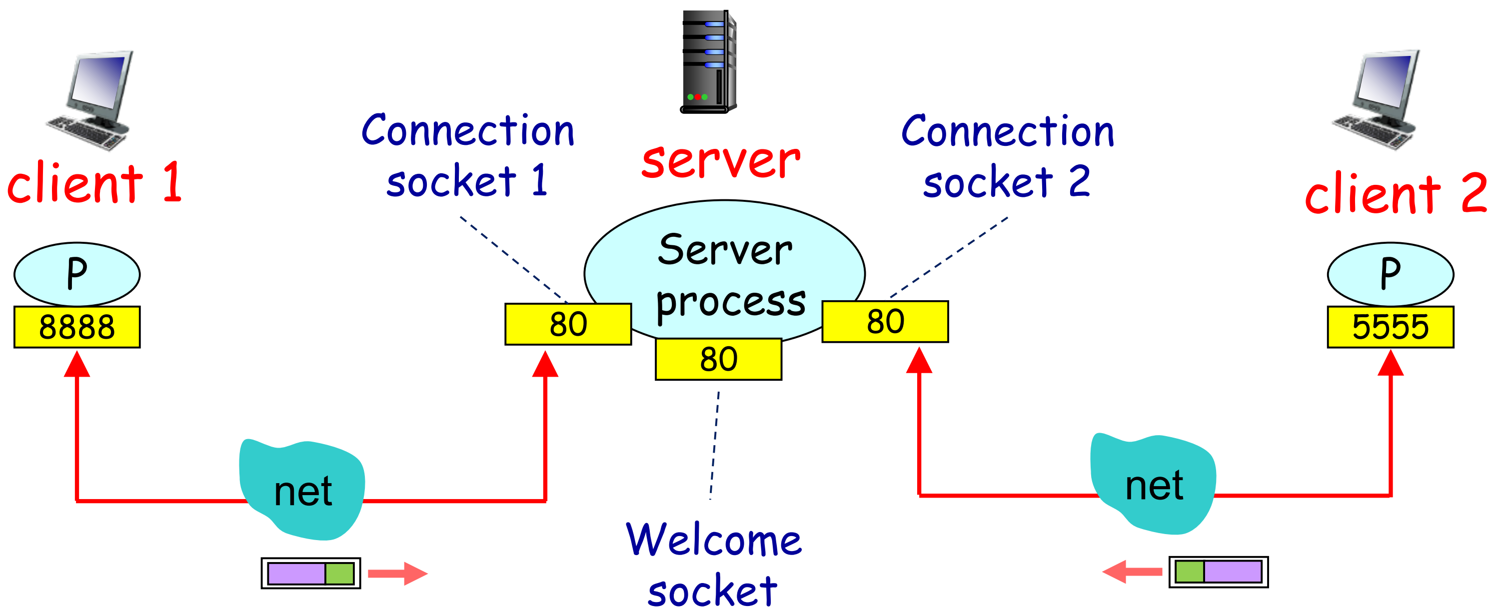
- We will only scratch the surface of TCP in CS2105.
  - More will be covered in CS3103.

TCP Overview [RFC 793, 1122, … 2581 …]

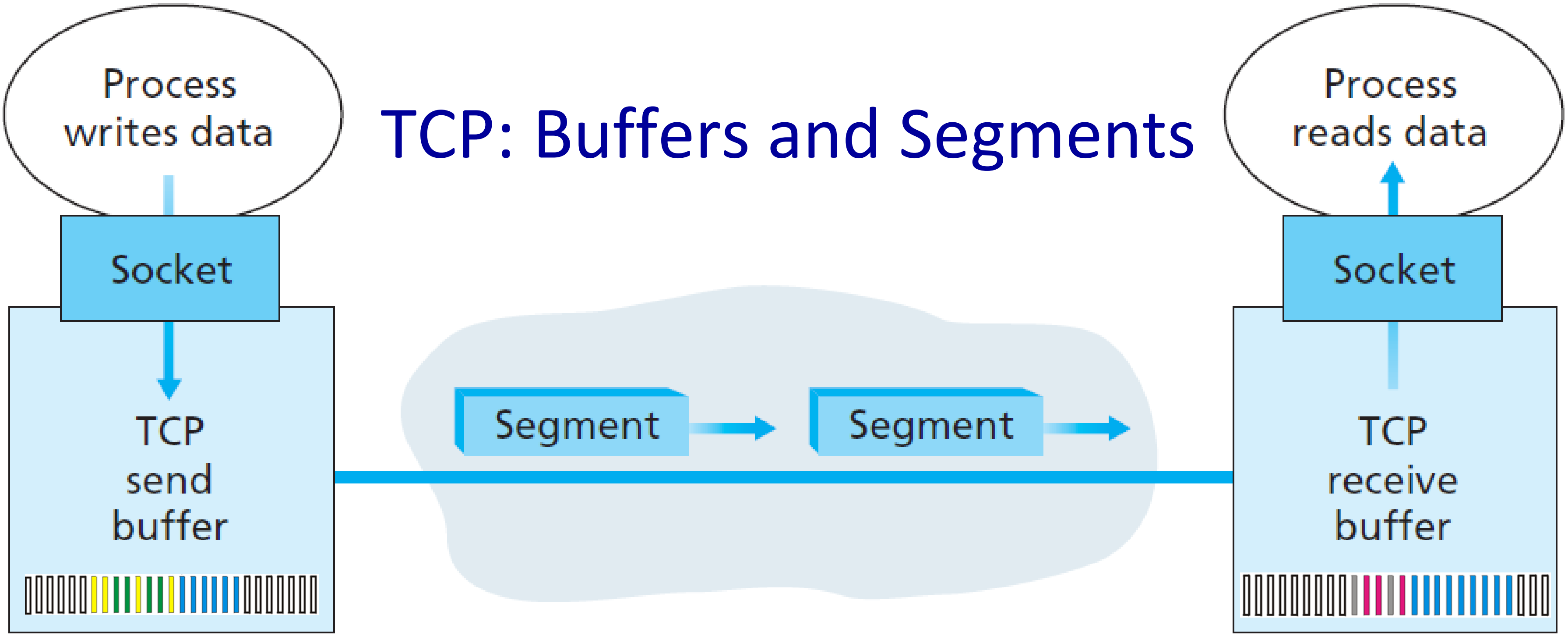
- Point-to-point:
  - One sender, one receiver.
- Connection-oriented:
  - handshaking (exchange of control messages) before sending app data.
- Full duplex service:
  - bi-directional data flow in the same connection
- Reliable, in-order *byte steam:*
  - use sequence numbers to label bytes

Connection-oriented De-mux

- A TCP connection (socket) is identified by 4-tuple:
  - (srcIPAddr, srcPort, destIPAddr, destPort)
  - Receiver uses all four values to direct a segment to the appropriate socket.

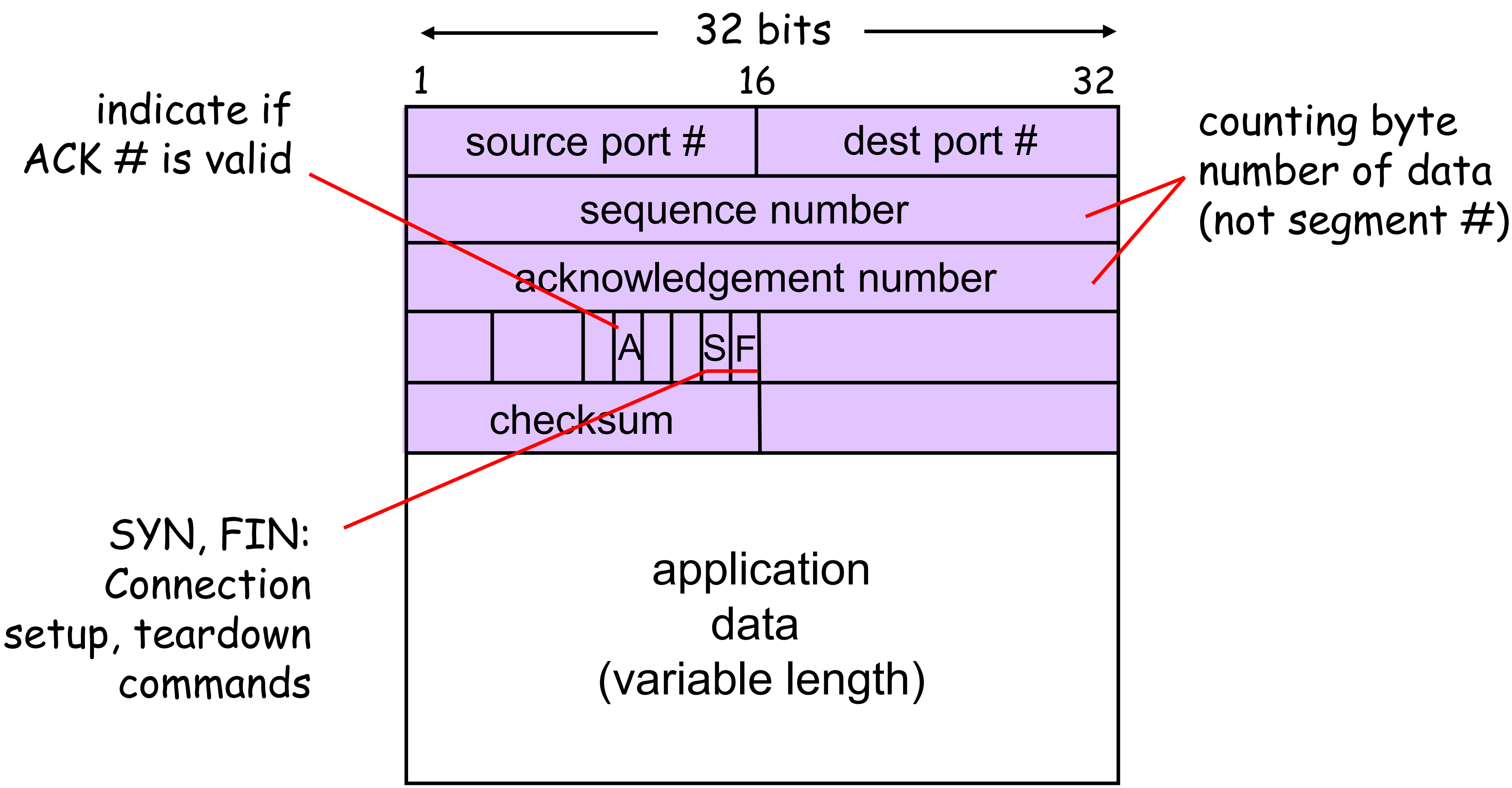


TCP: Buffers and Segments



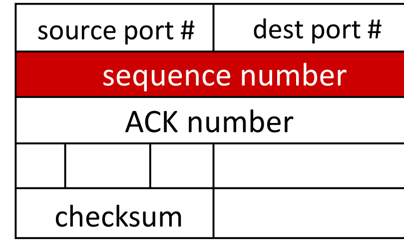
- TCP send and receive buffers
  - two buffers created after handshaking at any side.
- How much app-layer data a TCP segment can carry?
  - maximum segment size (MSS), typically 1,460 bytes
  - app passes data to TCP and TCP forms packets in view of MSS.

TCP Header

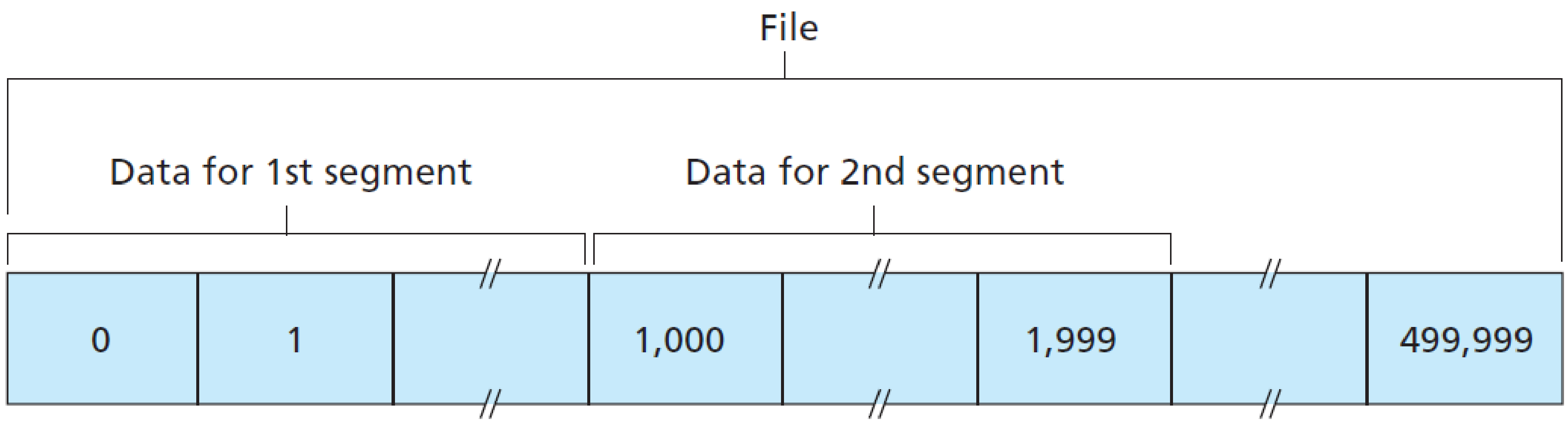


(some fields are not shown)

TCP Sequence Number



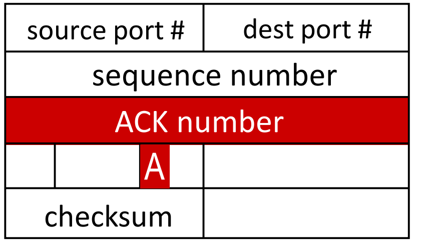
- "Byte number" of the <u>first</u> byte of data in a segment.
- Example: send a file of 500,000 bytes; MSS is 1,000 bytes.



Dividing file data into TCP segments

- Seq. # of 1st TCP segment: 0, 2nd TCP segment: 1,000, 3rd TCP segment: 2,000, 4th TCP segment: 3,000, etc.

TCP ACK Number



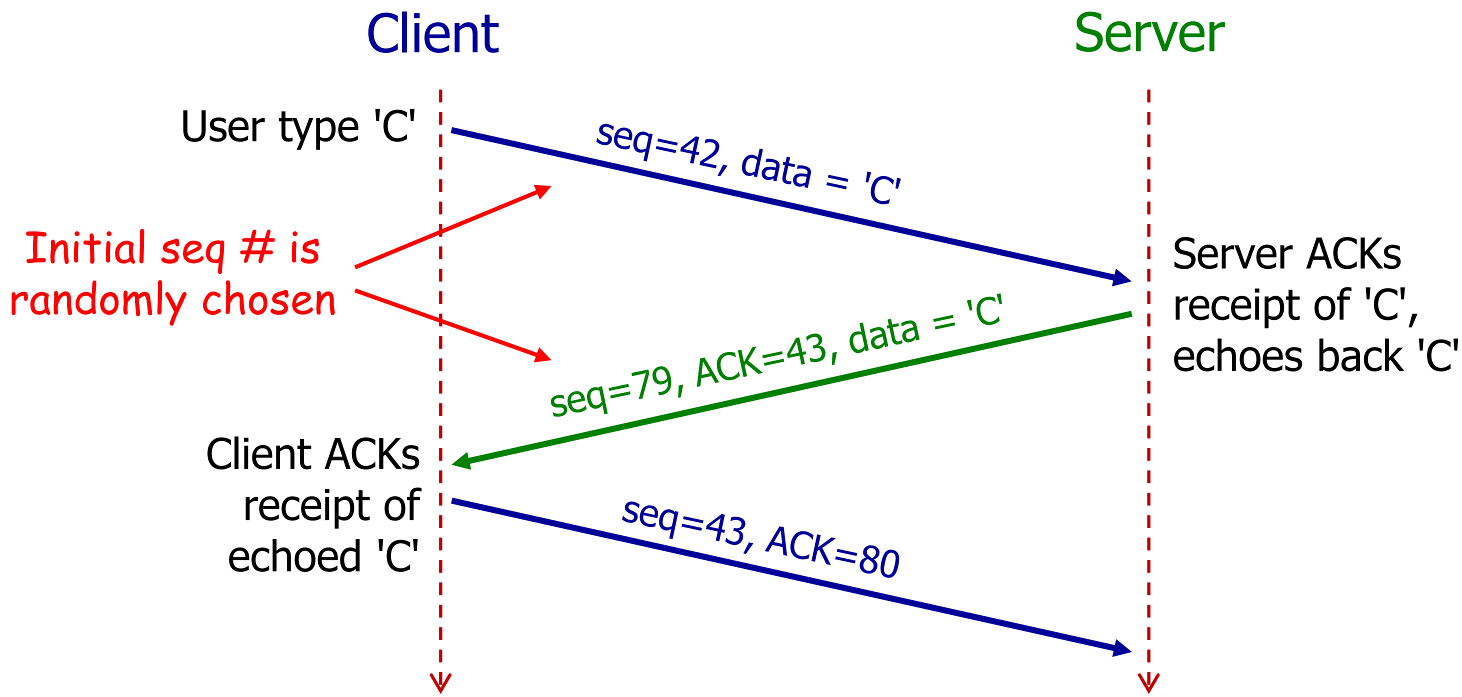
- Seq # of the next byte of data expected by receiver.

Sequence number of a segment	Amount of data carried	Corresponding ACK number
0	1,000	1,000
1,000	1,000	2,000
2,000	1,000	3,000
3,000	1,000	4,000
…	…	…

- TCP ACKs up to the first missing byte in the stream (cumulative ACK).
  - Note: TCP spec doesn't say how receiver should handle out-of-order segments - it's up to implementer.

Example: TCP Echo Server

- TCP (and also UDP) is a full duplex protocol
  - bi-directional data flow in the same TCP connection.
- Example:



TCP Echo Server

TCP Sender Events (simplified)

```
NextSeqNum=InitialSeqNumber
SendBase=InitialSeqNumber

loop (forever) {
    switch(event)

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

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

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

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

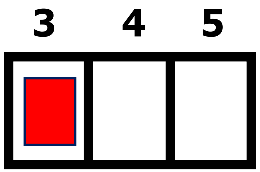
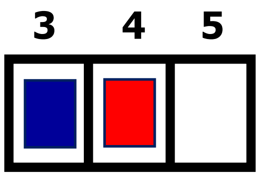
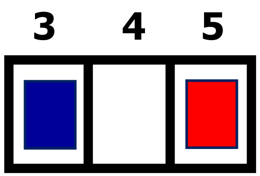
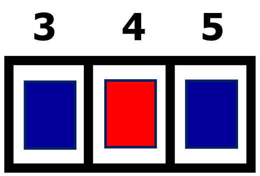
Sender keeps one timer only

Retransmit only oldest unACKed packet

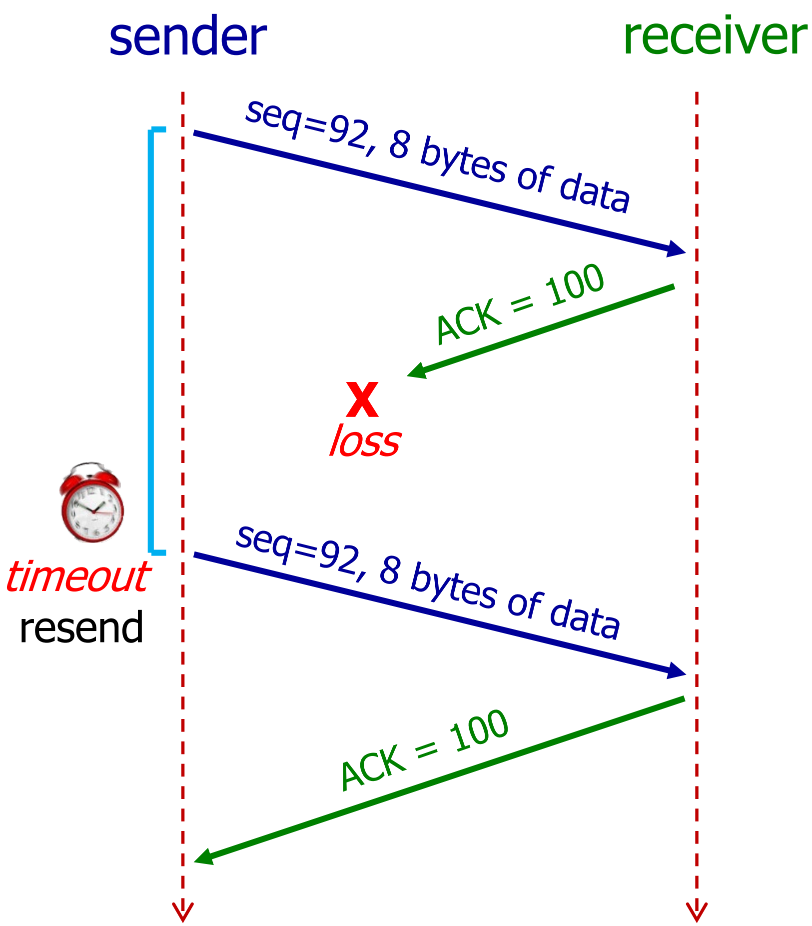
first byte of data to be ACKed.

Cumulative ACK

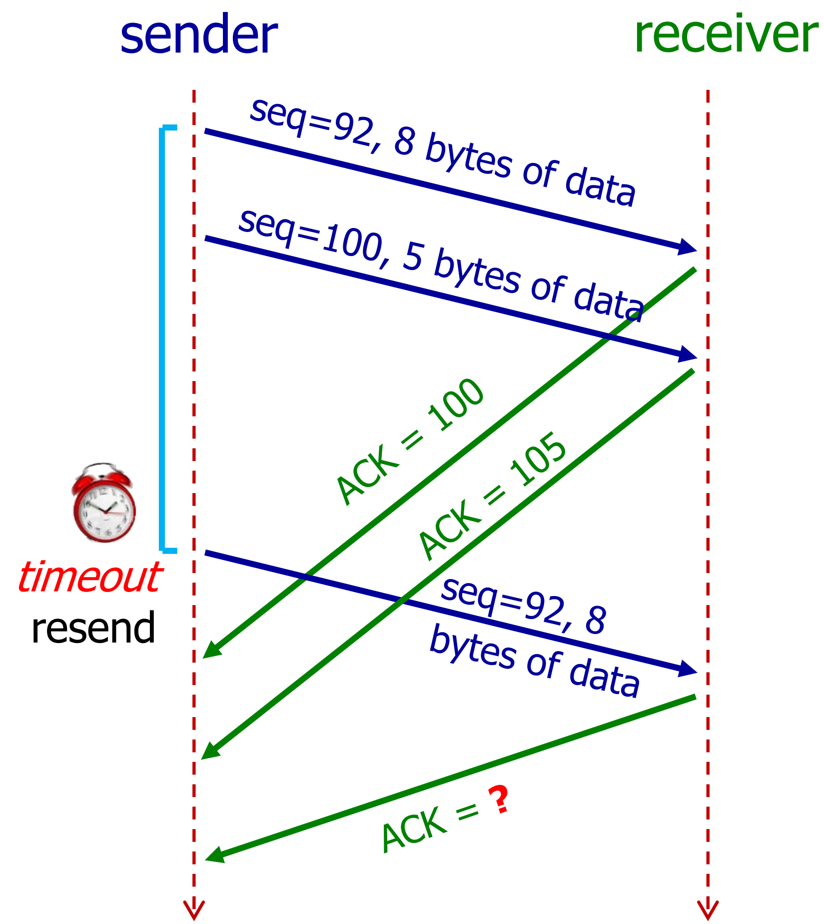
TCP ACK Generation [RFC 2581]

Event at TCP 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	Immediately send ACK, provided that segment starts at lower end of gap	

TCP Timeout / Retransmission



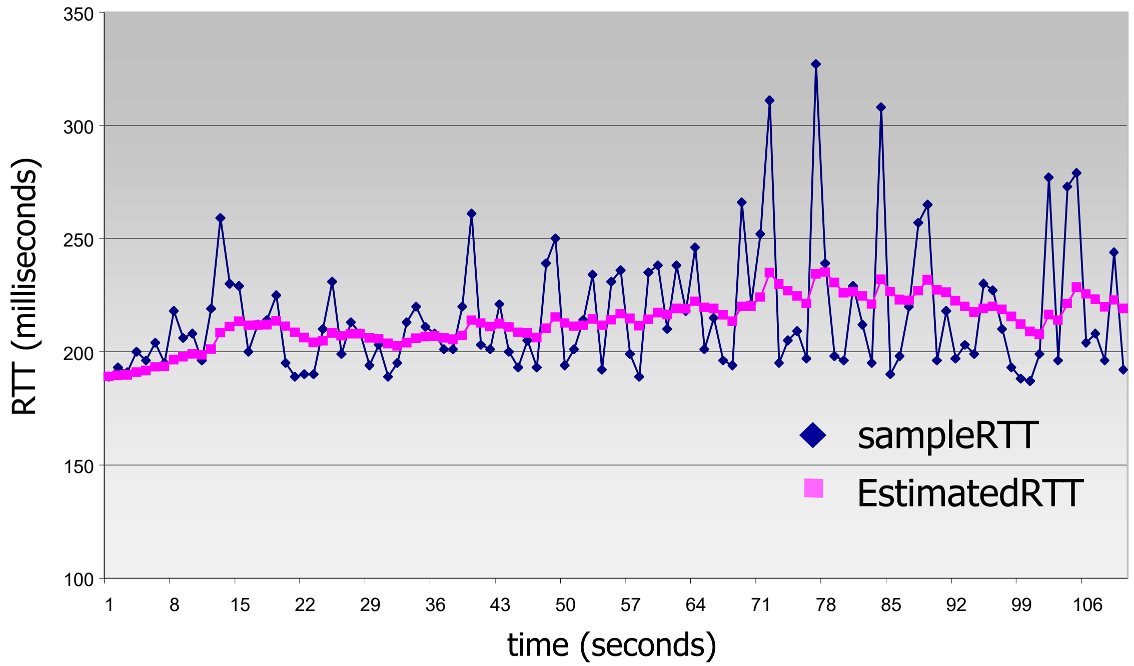
a) Lost ACK



b) premature timeout

TCP Timeout Value

- How does TCP set appropriate timeout value?
  - too short timeout: premature timeout and unnecessary retransmissions.
  - too long timeout: slow reaction to segment loss.
  - Timeout interval must be longer than RTT – but RTT varies!



TCP Timeout Value

- TCP computes (and keeps updating) timeout interval based on estimated RTT.

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

(typical value of α : 0.125)

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

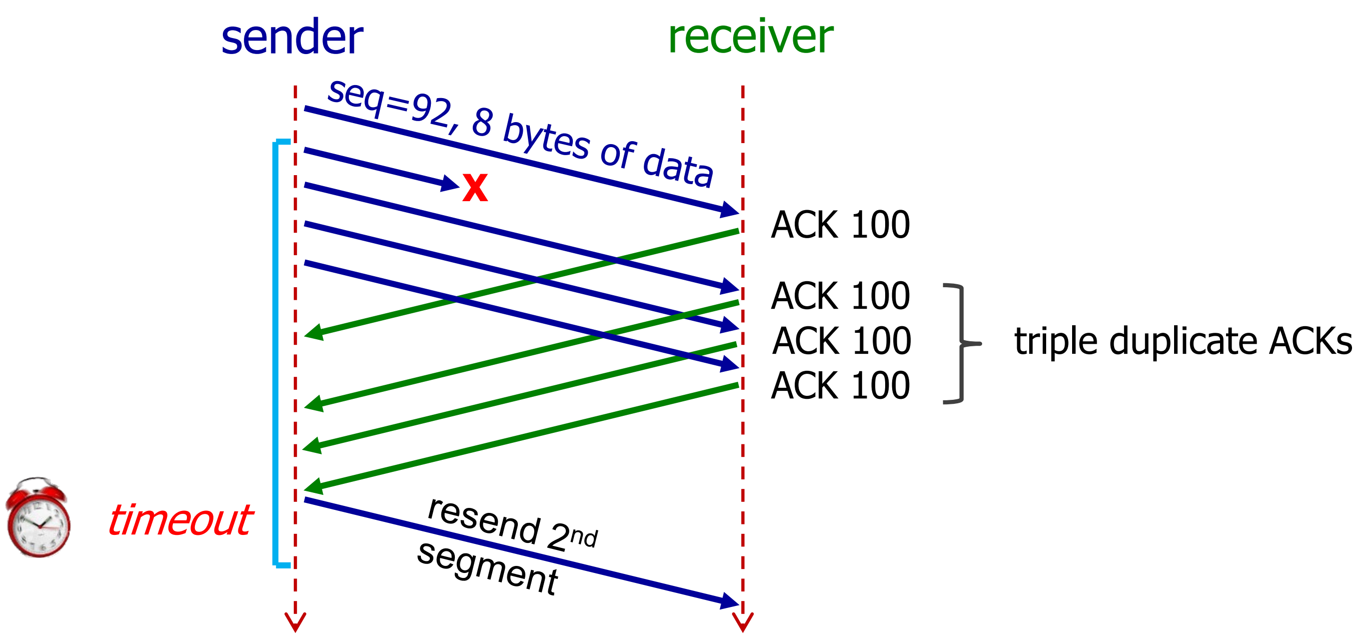
(typical value of β : 0.25)

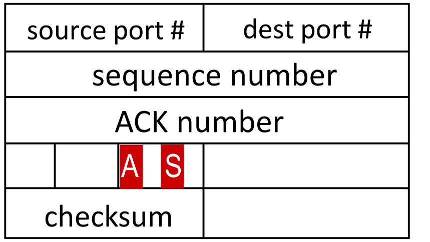
TimeoutInterval = EstimatedRTT + 4*DevRTT

"safety margin"

TCP Fast Retransmission [RFC 2001]

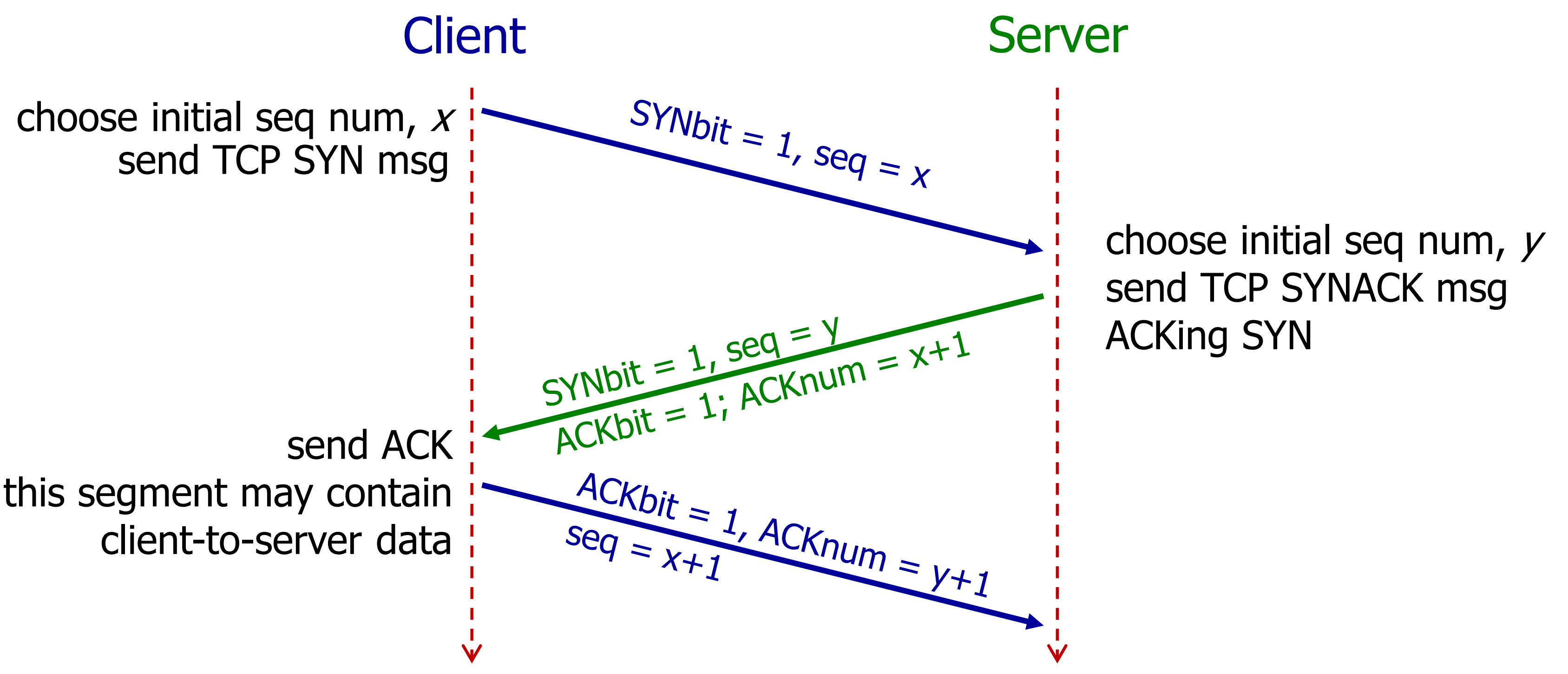
- Timeout period is often relatively long.
  - long delay before resending lost packet
- Fast retransmission:
  - **Event:** If sender receives 4 ACKs for the same segment, it supposes that segment is lost.
  - **Action:** resend segment (even before timer expires).





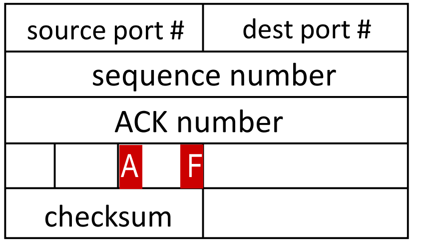
Establishing Connection

- Before exchanging app data, TCP sender and receiver "shake hands".
  - Agree on connection and exchange connection parameters.

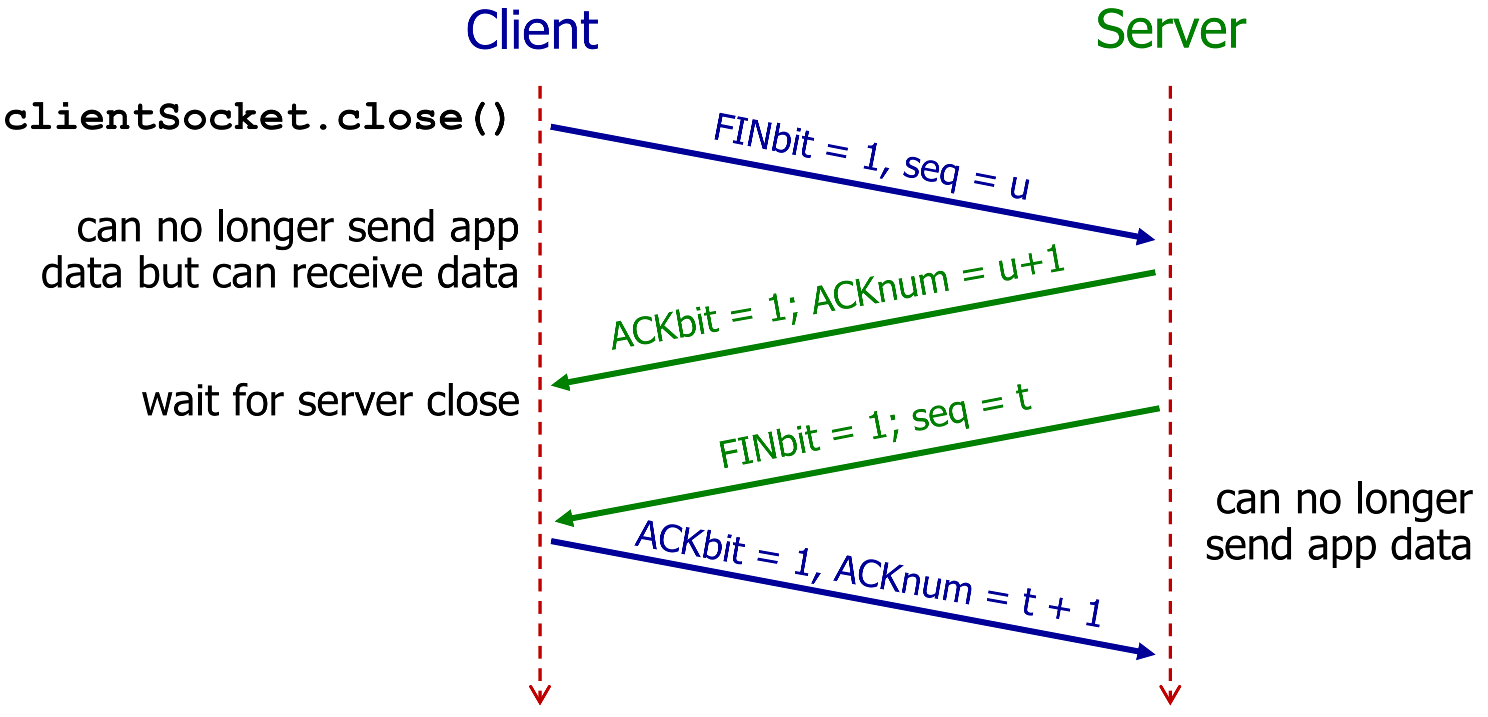


3-way handshake

Closing Connection



- Client, server each close their side of connection.
  - send TCP segment with FIN bit = 1



What we did not cover….

- TCP flow control (Chapter 3.5.5)
  - Sender won’t overflow receiver’s buffer by sending too much or too fast.
  - Receiver feeds back to sender how many more bytes it is willing to accept.
- TCP congestion control (Chapter 3.6 & 3.7)
  - Be polite and send less if network is congested.
- They will be covered in the next course (CS3103)

Lectures 4&5: Summary

Go-back-N

- Sender can have up to *N* unACKed packets in pipeline
- Receiver only sends *cumulative ACKs*
  - Out-of-order packets discarded
- Sender sets timer for the 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
  - Out-of-order packets buffered
- Sender maintains timer for ***each*** unACKed packet
  - when timer expires, retransmit only that unACKed packet

Lectures 4&5: Summary

- Connection-oriented transport: TCP
  - Segment structure
  - Reliable data transfer
  - Sequence number
  - Acknowledgement number
  - Cumulative ACK
  - Setting and updating retransmission time interval
  - Fast retransmission
  - 3-way handshake