GITAM
(DEEMED TO BE UNIVERSITY)
(Estd. u/s 3 of the UGC Act, 1956)

Module 3

Transport Layer

Chapter 3: Transport Layer

AGENDA

- Understand principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- Learn about Internet transport layer protocols:
  - UDP: connectionless transport
  - TCP: connection-oriented reliable transport
  - TCP congestion control

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

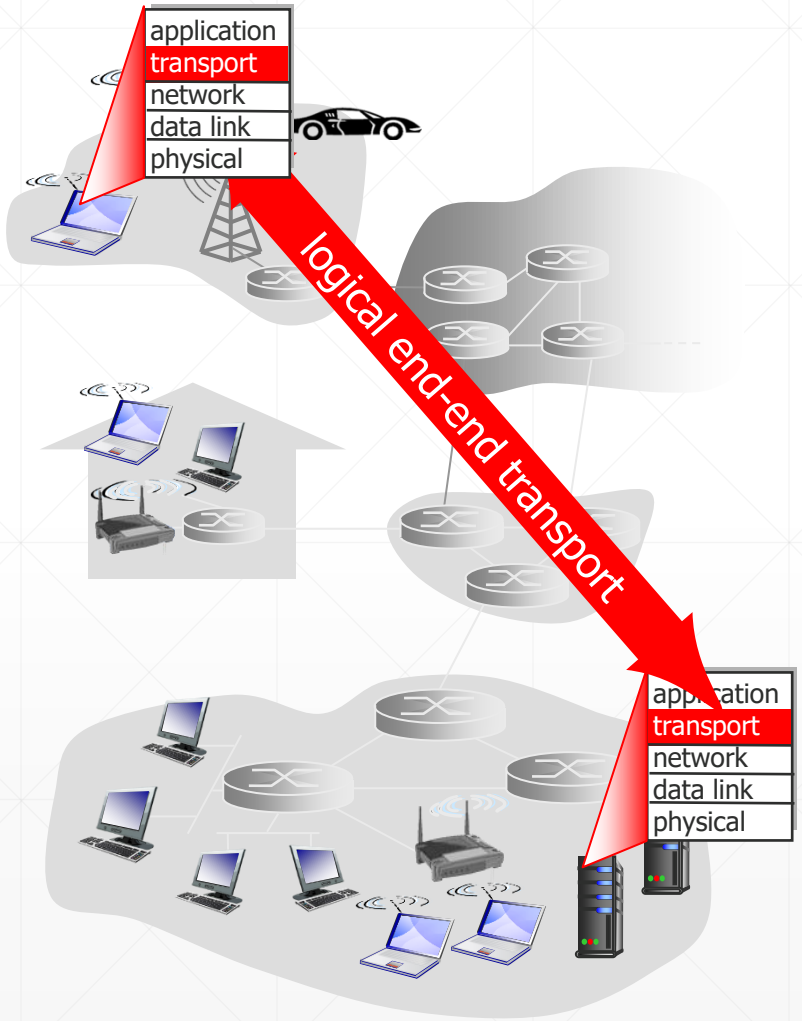
- segment structure
- reliable data transfer
- flow control
- connection management

3.6 Principles of congestion control

3.7 TCP congestion control

Transport services and protocols

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



Transport vs. network layer

- *network layer:* logical communication between hosts
- *transport layer:* logical communication between processes
  - relies on, enhances, network layer services

household analogy:

12 kids 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

Overview of Transport layer in the Internet

- Internet makes two transport layer protocols available:
  - UDP (user datagram protocol)
    - Unreliable
    - connectionless
  - TCP (transmission control protocol)
    - Reliable
    - Connection oriented
- Application developer should mention one of these protocols while designing a network application

Overview of Transport layer in the Internet

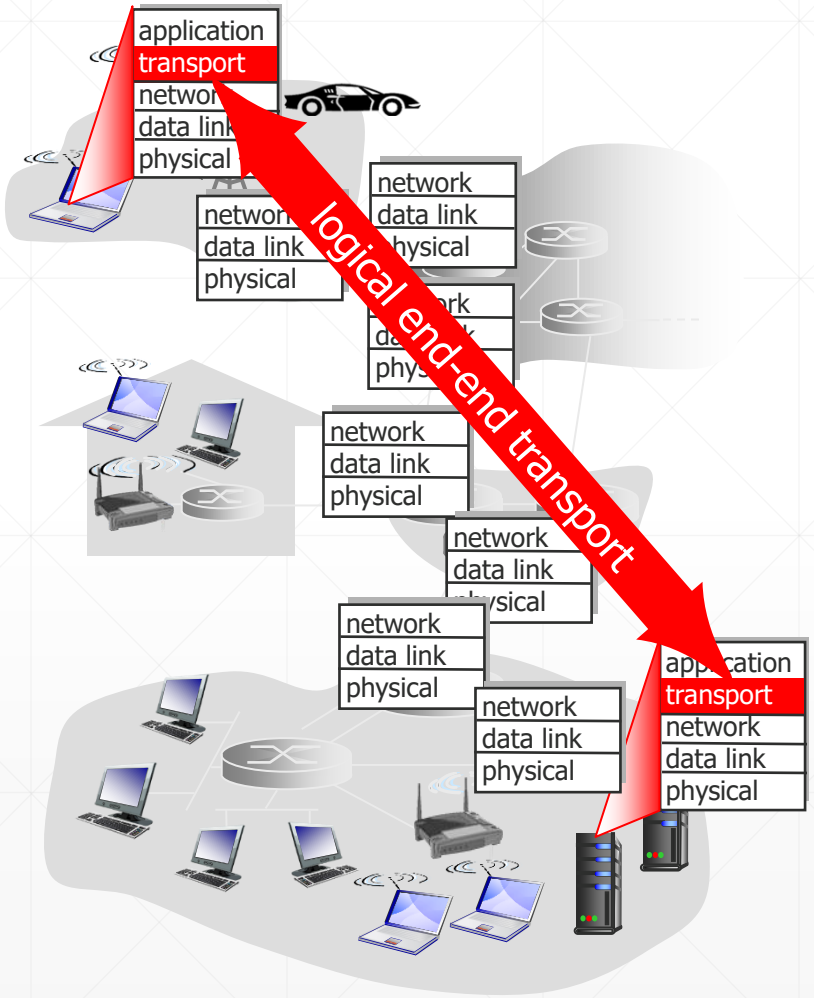
- Transport layer packet
  - Segments
- Network layer packet
  - Datagrams
- Internet's network layer protocol
  - **IP (internet protocol)**
    - IP service model is a "best-effort delivery service" (makes best effort to deliver the segments, but no guarantee of delivery, orderly delivery & integrity)
    - IP is **unreliable service**
    - Each host has an IP address

Overview of Transport layer in the Internet

- **Fundamental responsibility of UDP & TCP**
  - Extend IP's delivery system
- **Transport layer Multiplexing & Demultiplexing**
  - Extending host to host delivery to process to process delivery
- UDP & TCP provides integrity checking by including error detection fields in segment headers

Internet transport-layer protocols

- reliable, in-order delivery (TCP)
  - congestion control
    - TCP gives each connection traversing a congested link an equal share of link bandwidth
  - flow control
  - connection setup
  - Converts IP's unreliable service between end systems into reliable service between processes
- unreliable, unordered delivery: UDP



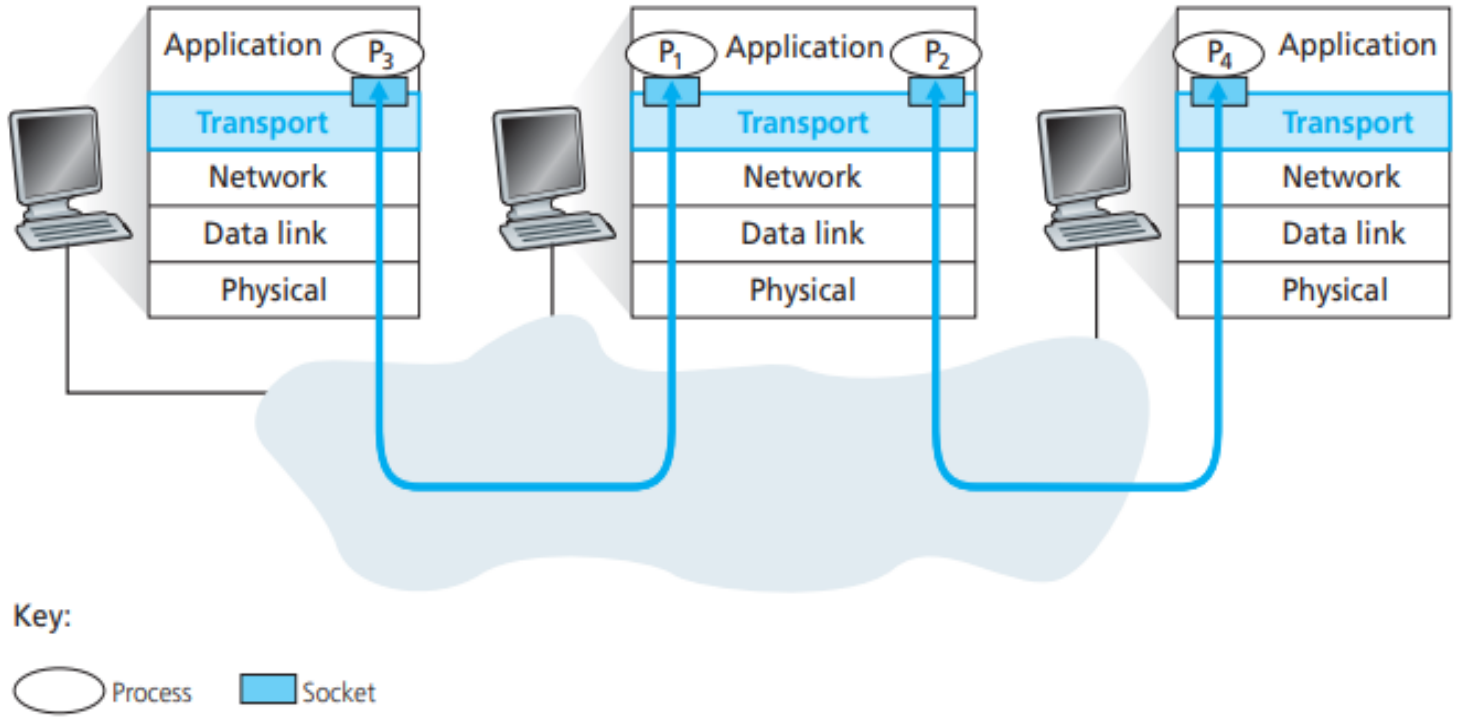
Multiplexing and Demultiplexing

- Extending the host-to-host delivery service provided by the network layer to a process-to-process delivery service for applications running on the hosts
- **Multiplexing/Demultiplexing**
- Example:
  - Someone is downloading **Web pages** while running **one FTP session** and **two Telnet sessions**.
  - Therefore, have four network application processes running—two Telnet processes, one FTP process, and one HTTP process.
  - a process can have one or more **sockets**
  - each socket has a unique **identifier**

Multiplexing and Demultiplexing

- How a receiving host directs an incoming transport-layer segment to appropriate socket?
- transport-layer segment has a set of fields in the segment
- At the receiving end, the transport layer examines these fields to deliver the data
- The job of delivering the data in a transport-layer segment to the correct socket is called **demultiplexing**
- The job of **gathering data chunks** at the source host from different sockets, **encapsulating each data chunk** with header to create segments, and **passing the segments to the network layer** is called **multiplexing**

Transport-layer multiplexing and demultiplexing



Multiplexing and Demultiplexing Example

- **Household Analogy**
  - Each of the kids is identified by his or her name
  - When Bill receives a batch of mail from the mail carrier
  - Bill performs a **demultiplexing operation** by observing to whom the letters are addressed and then hand delivering the mail to his brothers and sisters
  - Ann performs a **multiplexing operation** when she collects letters from her brothers and sisters and gives the collected mail to the mail person.

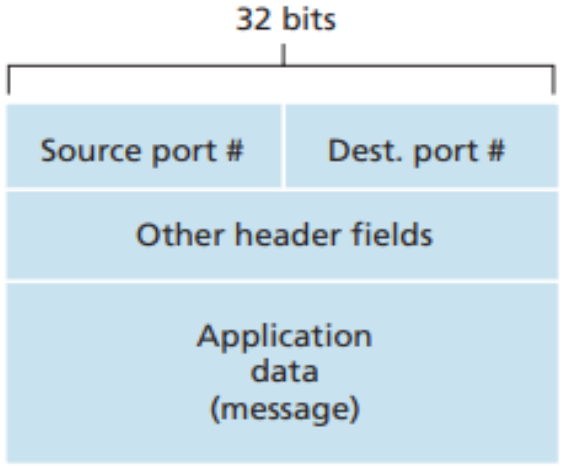
Roles of transport-layer multiplexing and demultiplexing

- **Transport-layer multiplexing requires**
  - that sockets have unique identifiers, and
  - that each segment have special fields that indicate the socket to which the segment is to be delivered
- **Special fields are**
  - source port number field
  - destination port number field
- Port number is a **16-bit number**, ranging from **0 to 65535**
- Port numbers ranging from **0 to 1023** are called **well-known port numbers** and are restricted

Source and destination port-number fields in a transport-layer segment

Demultiplexing service:

- Each socket in the host is assigned a port number
- when a segment arrives at the host, the transport layer examines the destination port number in the segment and directs the segment to the corresponding socket
- The segment's data then passes through the socket into the attached process



Connectionless Multiplexing and Demultiplexing

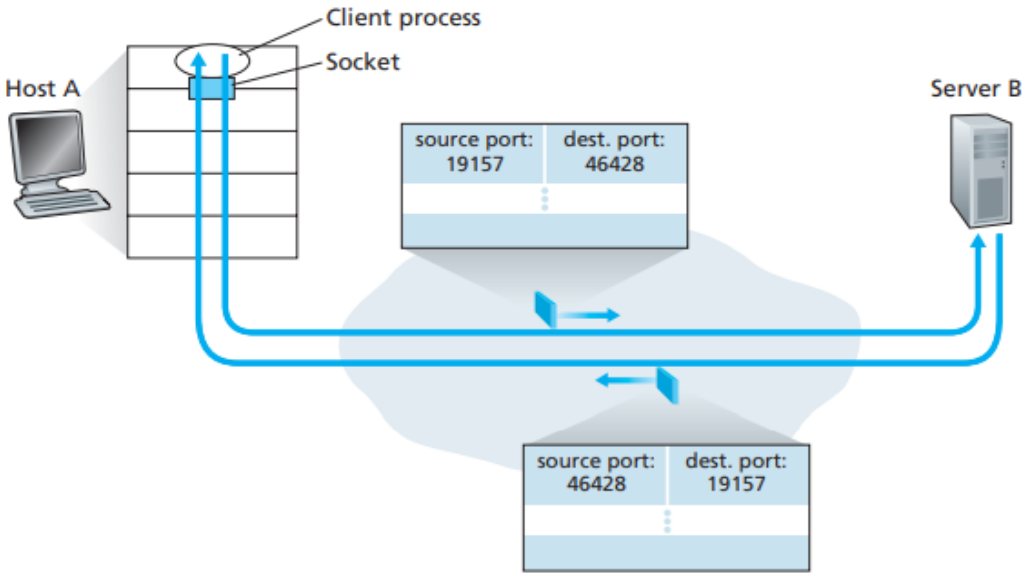
- create a UDP socket:
  - **clientSocket = socket(socket.AF_INET, socket.SOCK_DGRAM)**
- When a UDP socket is created, the transport layer automatically assigns a port number to the socket in the range 1024 to 65535
- After creating the socket to associate a specific port number use bind() method:
  - **clientSocket.bind((‘’, 19157))**
- The client side of the application lets the transport layer automatically assign the port number
- The server side of the application assigns a specific port number

UDP multiplexing/demultiplexing (Example)

- Suppose a process in **Host A**, with UDP **port 19157**, wants to send a chunk of **application data** to a process with UDP **port 46428** in **Host B**
- The transport layer in **Host A creates a transport-layer segment** that includes the application data, the source port number (19157), the destination port number (46428)
- The transport layer then **passes the resulting segment** to the **network layer**
- The network layer **encapsulates the segment in an IP datagram** and makes a **best-effort** attempt to deliver the segment to the receiving host.
- If the segment arrives at the receiving Host B, the transport layer at the receiving host examines the destination port number in the segment (46428) and **delivers the segment to its socket** identified by port 46428.
- Host B could be **running multiple processes**, each with its own UDP socket and associated port number.
- As UDP segments arrive from the network, Host B directs (demultiplexes) each segment to the appropriate socket by **examining the segment's destination port number**

UDP multiplexing/demultiplexing

- Purpose of the **source port number**
- In the A-to-B segment the source port number serves as part of a “return address”—when B wants to send a segment back to A



Connection-Oriented Multiplexing and Demultiplexing

- **Difference between a TCP socket and a UDP socket**
- TCP socket is identified by a **four-tuple:**
  - source IP address,
  - source port number,
  - destination IP address,
  - destination port number
- When a TCP segment arrives from network to a host, host uses all four values to direct the segment to appropriate socket

TCP multiplexing/demultiplexing

- **TCP client-server programming**
- The TCP server application has a “welcoming socket,” that waits for connection establishment requests from TCP clients on port
- The TCP client creates a socket and sends a connection establishment request segment with the lines:

```
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName,12000))
```

- When the host operating system of the computer running the server process receives the incoming connection-request segment with destination port 12000, it locates the server process that is waiting to accept a connection on port number 12000.
- The server process then creates a new socket:

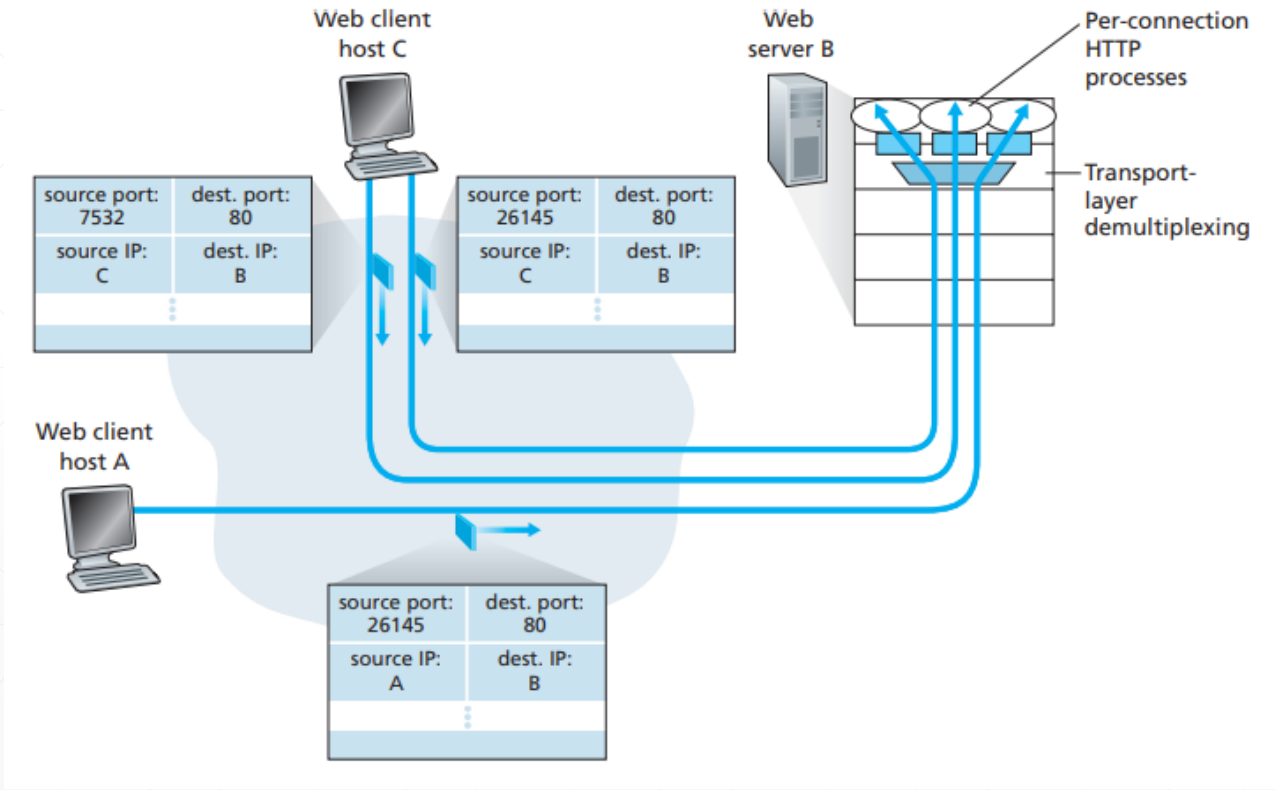
```
connectionSocket, addr = serverSocket.accept()
```

TCP multiplexing/demultiplexing

- The transport layer at the server notes the following four values in the connection-request segment:
  - the source port number in the segment,
  - the IP address of the source host,
  - the destination port number in the segment, and
  - its own IP address
- **newly created connection** socket is identified by these four values

TCP multiplexing/demultiplexing

- Host C initiates two HTTP sessions to server B
- Host A initiates one HTTP session to B
- Each have their own unique IP address—A, C, and B
- B will still be able to correctly demultiplex the two connections having the same source port number
  - since the two connections have different source IP addresses



Web Servers and TCP

- **Web servers and how they use port numbers**
- If the client and server are using **persistent HTTP**, then throughout the duration of the persistent connection the client and server exchange HTTP messages via the same server socket
- if the client and server use **non-persistent HTTP**, then a new TCP connection is created and closed for every request/response, and hence a new socket is created and later closed for every request/response
  - This can impact the performance of a busy Web server

Connectionless Transport : UDP

- UDP is defined in RFC 768
  - multiplexing/demultiplexing function
  - light error checking
- UDP **uses the destination port number** to deliver the segment's data to the correct application process
- With UDP there is **no handshaking** between sending and receiving transport-layer entities before sending a segment
- UDP is said to be **connectionless**
- **Example** : DNS (an application-layer protocol) uses UDP

Connectionless Transport : UDP

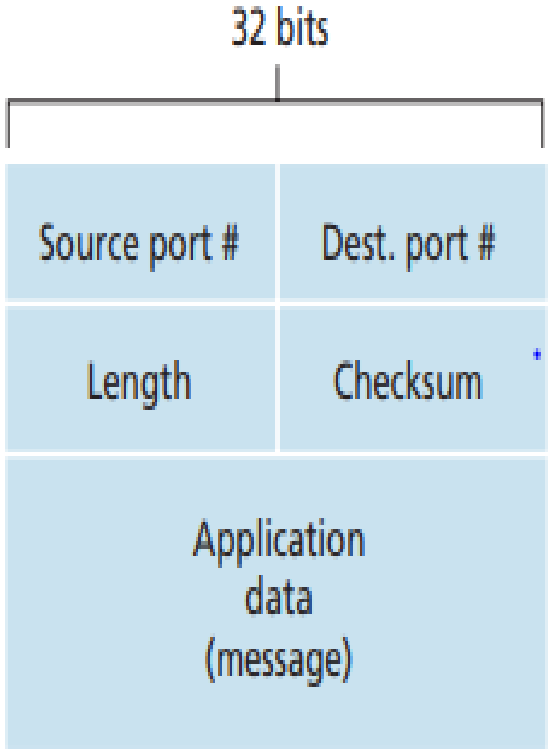
- Many applications are better suited for UDP for the following reasons:
- **Finer application-level control over what data is sent, and when**
- **No connection establishment :**
  - UDP does not introduce any delay to establish a connection
  - The reason why DNS runs over UDP rather than TCP
- **No connection state : (**receive and send buffers, congestion-control parameters, and sequence and acknowledgment number parameters)
  - a server devoted to a particular application can typically support many more active clients when the application runs over UDP rather than TCP
- **Small packet header overhead :**
  - TCP segment has 20 bytes of header overhead, whereas UDP has only 8 bytes of overhead

Popular Internet applications and their underlying transport protocols

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
Routing protocol	RIP	Typically UDP
Name translation	DNS	Typically UDP

UDP Segment Structure

- The UDP segment structure as defined in RFC 768
- application data occupies the data field
  - Example : streaming audio application, audio samples fill the data field
- UDP header has only **four fields**, each consisting of **two bytes**
- Length field specifies the **number of bytes** in the UDP segment (header plus data)
- The checksum is used by the receiving host to check whether **errors have been introduced** into the segment



UDP Checksum

- The UDP checksum provides for error detection
- Checksum is used to determine whether bits within the UDP segment have been altered
- Last addition had overflow, which was wrapped around
- 1s complement is obtained by converting all the 0s to 1s and converting all the 1s to 0s
- 1s complement of the sum 0100101011000010 is 1011010100111101
- At the receiver, all four 16-bit words are added, including the checksum
- 1111111111111111 – no error
- One of the bit is a 0 -- error

Three 16-bit words :

0110011001100000
0101010101010101
1000111100001100

The sum of first two of these 16-bit words is

0110011001100000
0101010101010101
1011101110110101

Adding the third word to the above sum gives

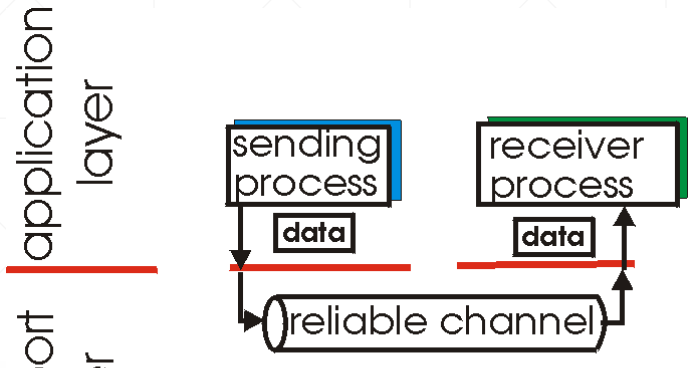
1011101110110101
1000111100001100
0100101011000010

UDP Checksum

- Why UDP provides a checksum as many link layer protocols also provide error checking?
  - there is no guarantee that all the links between source and destination provide error checking
  - possibility that bit errors could be introduced when a segment is stored in a router's memory
- neither link-by-link reliability nor in-memory error detection is **guaranteed**, UDP **must provide error detection** at the transport layer, on an end-end basis
- UDP provides error checking, it does not do anything to recover from an error
- Few implementations of UDP simply discard the damaged segment
- Few pass the damaged segment to the application with a warning

Principles of reliable data transfer

- important in application, transport, link layers

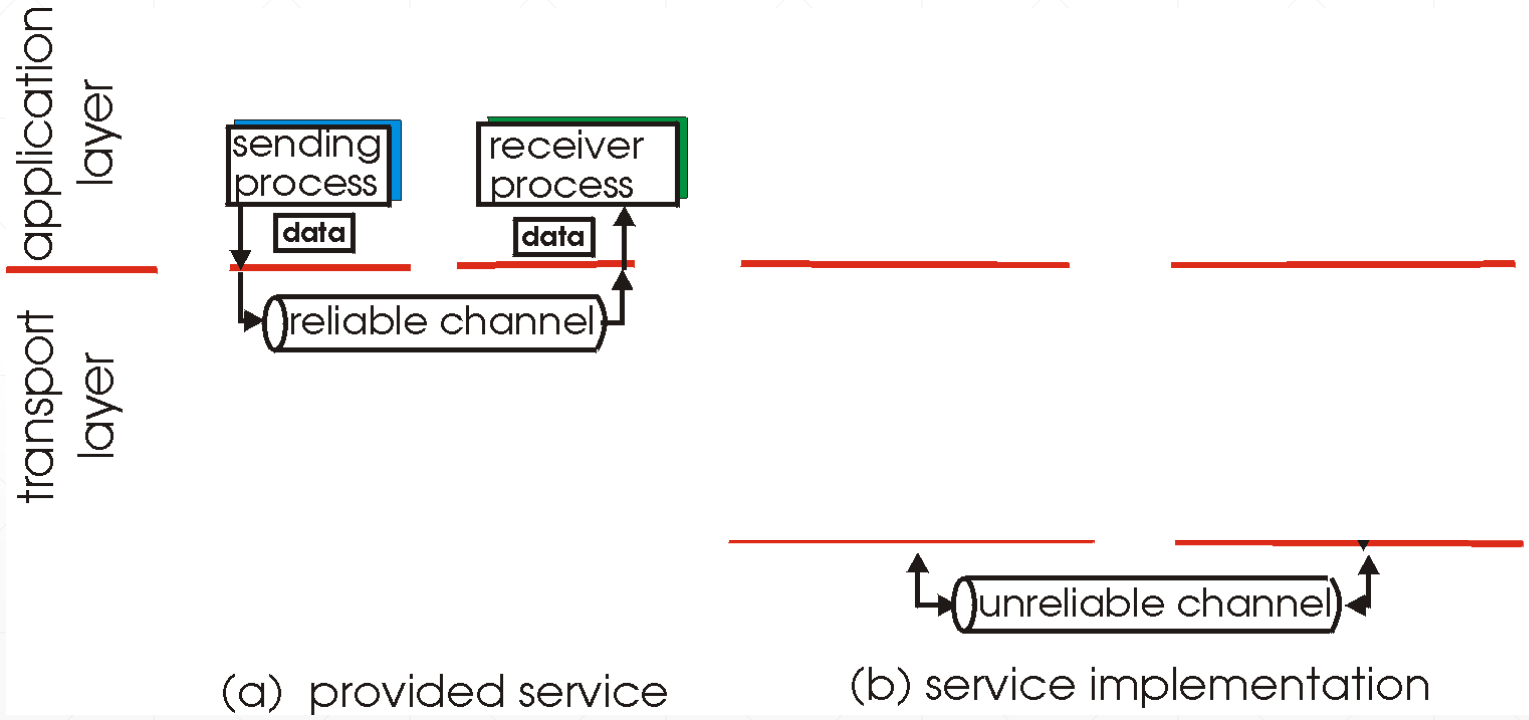


(a) provided service

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

Principles of reliable data transfer

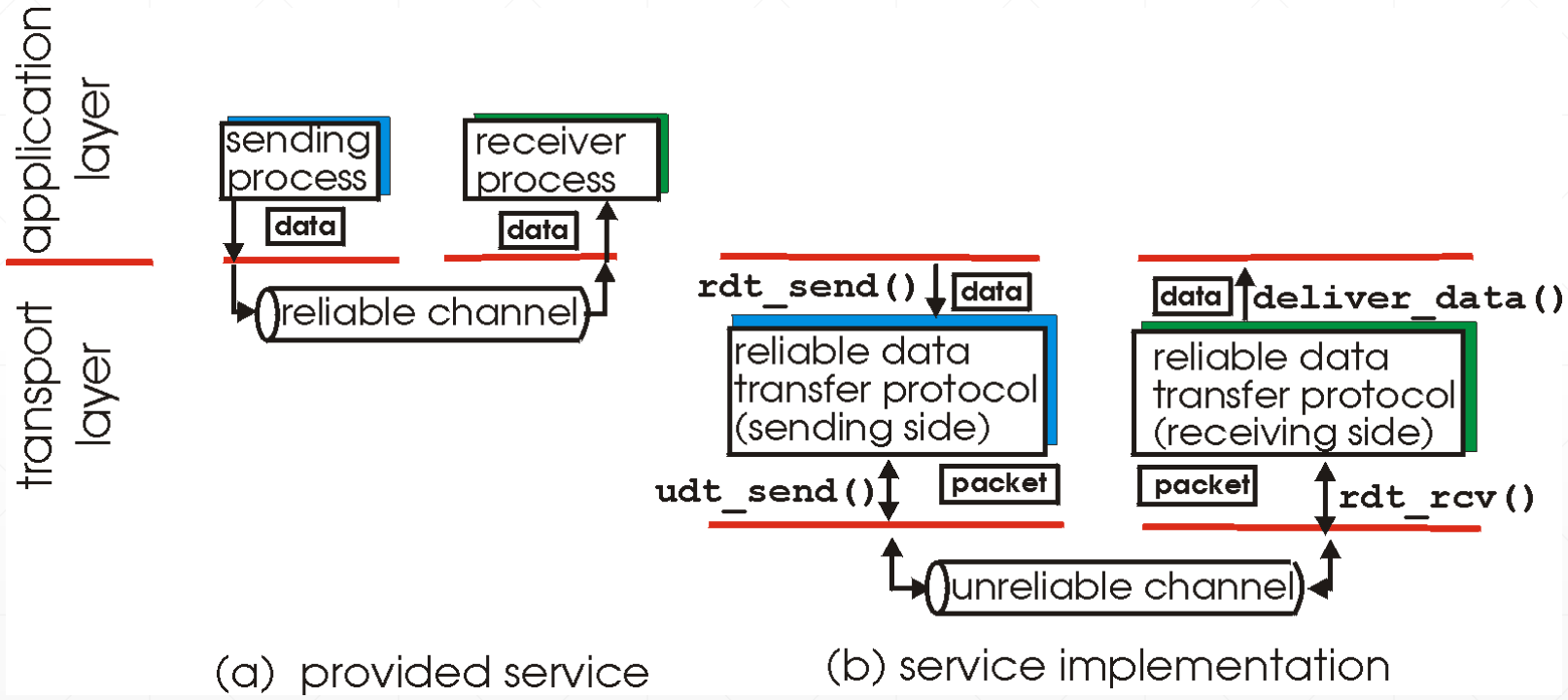
- important in application, transport, link layers



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

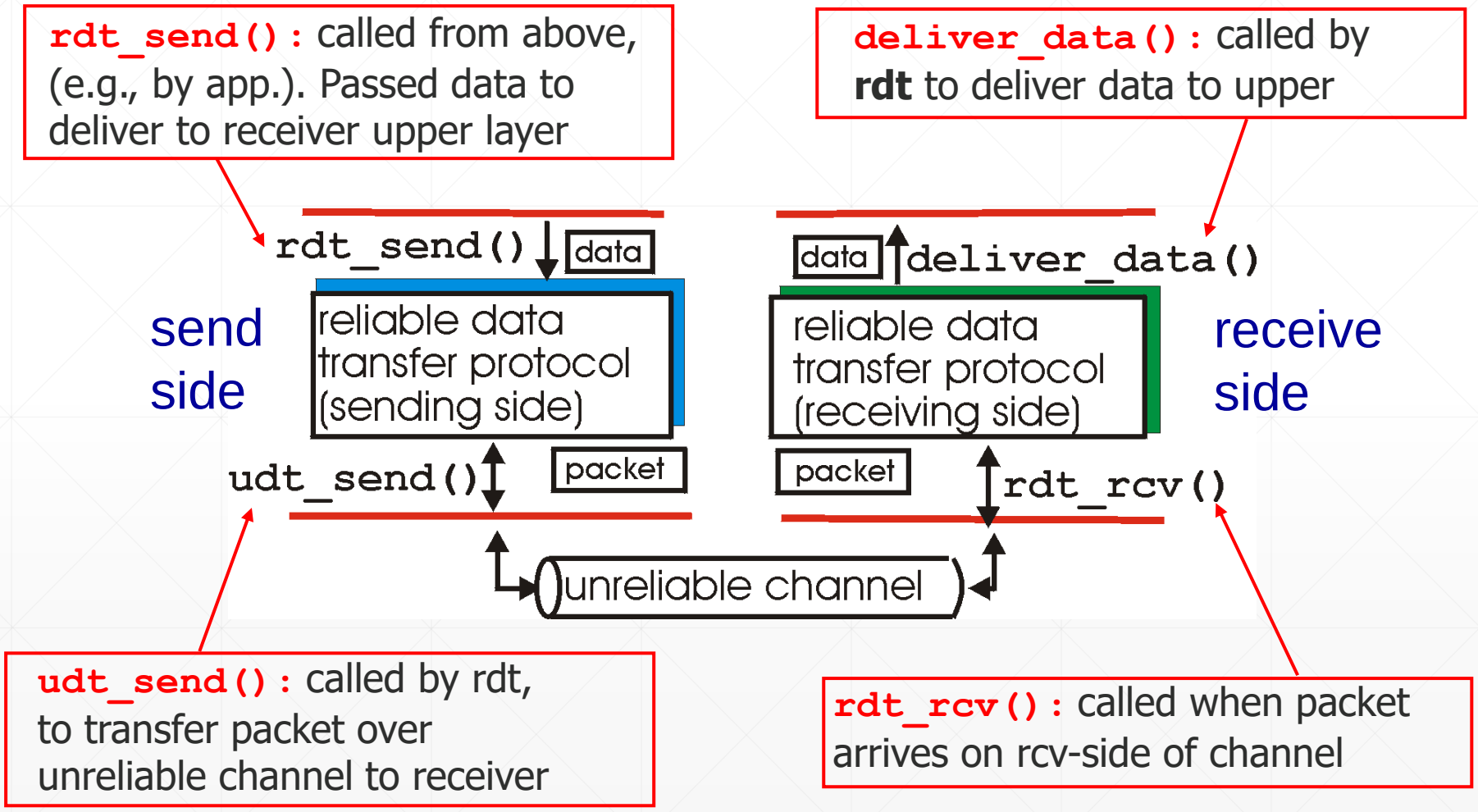
Principles of reliable data transfer

- important in application, transport, link layers
- Case of unidirectional data transfer

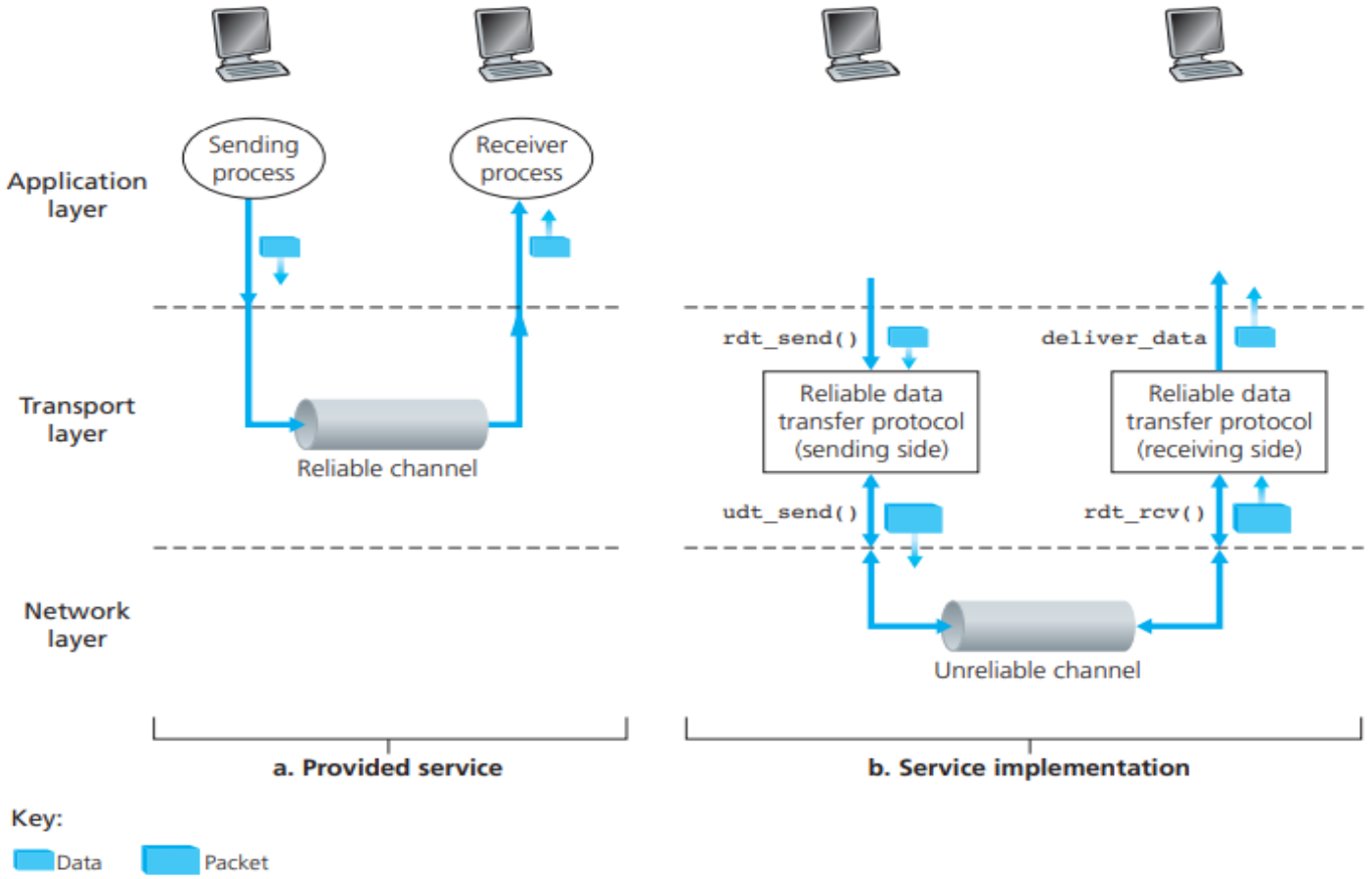


- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Reliable data transfer: getting started



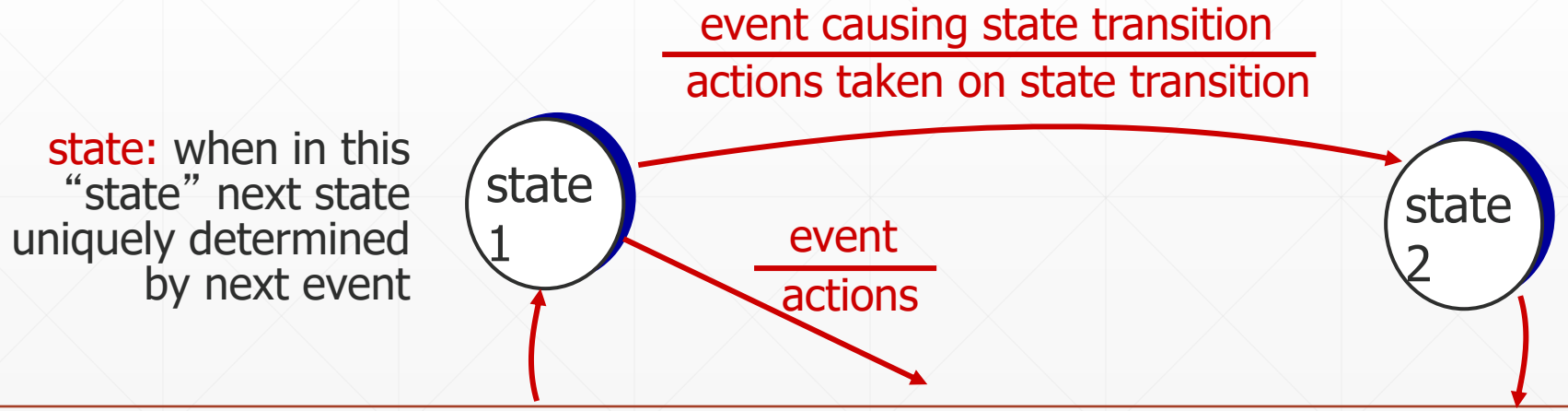
Reliable data transfer: Service model and service implementation



Reliable data transfer: getting started

We'll:

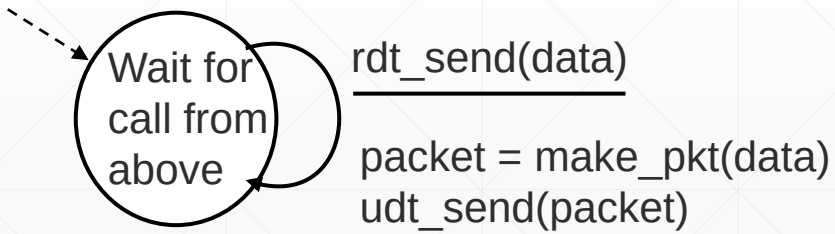
- incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- consider only unidirectional data transfer
  - but control info will flow on both directions!
- use finite state machines (FSM) to specify sender, receiver



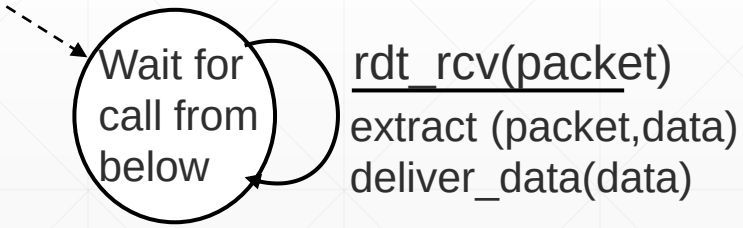
rdt1.0: reliable transfer over a reliable channel

- 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

- A **symbol Λ** below or above horizontal line denotes the lack of an action or event.
- Initial state of FSM is indicated by a **dashed arrow**



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:

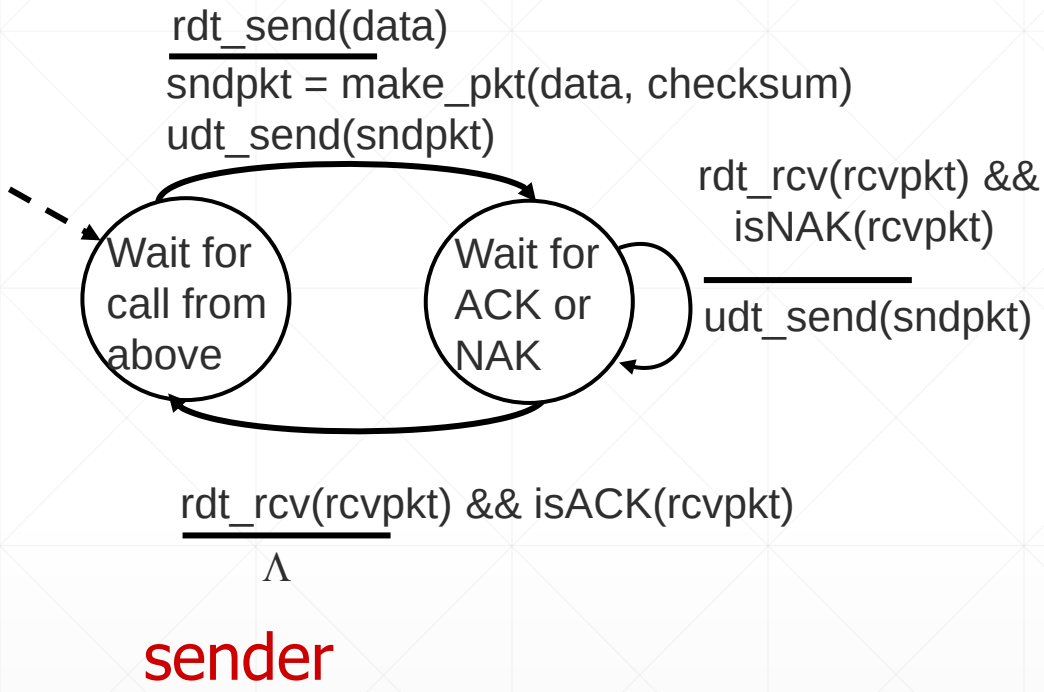
How do humans recover from "errors" during conversation?

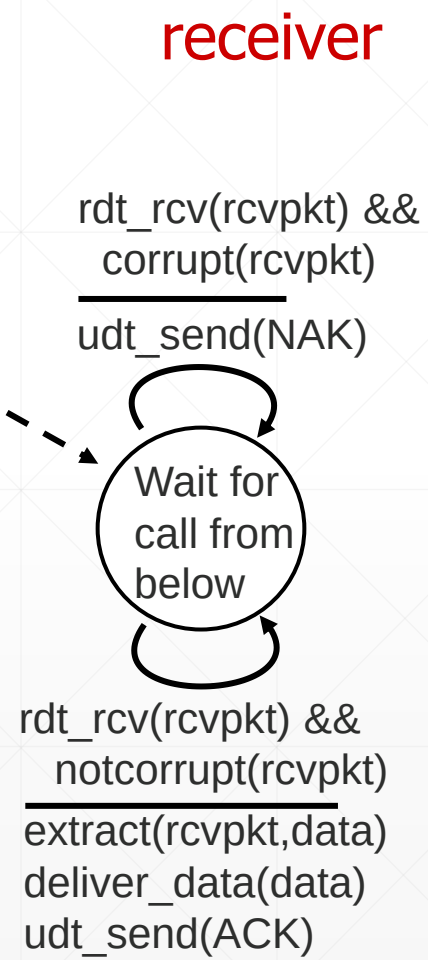
rdt2.0: channel with bit errors

- 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

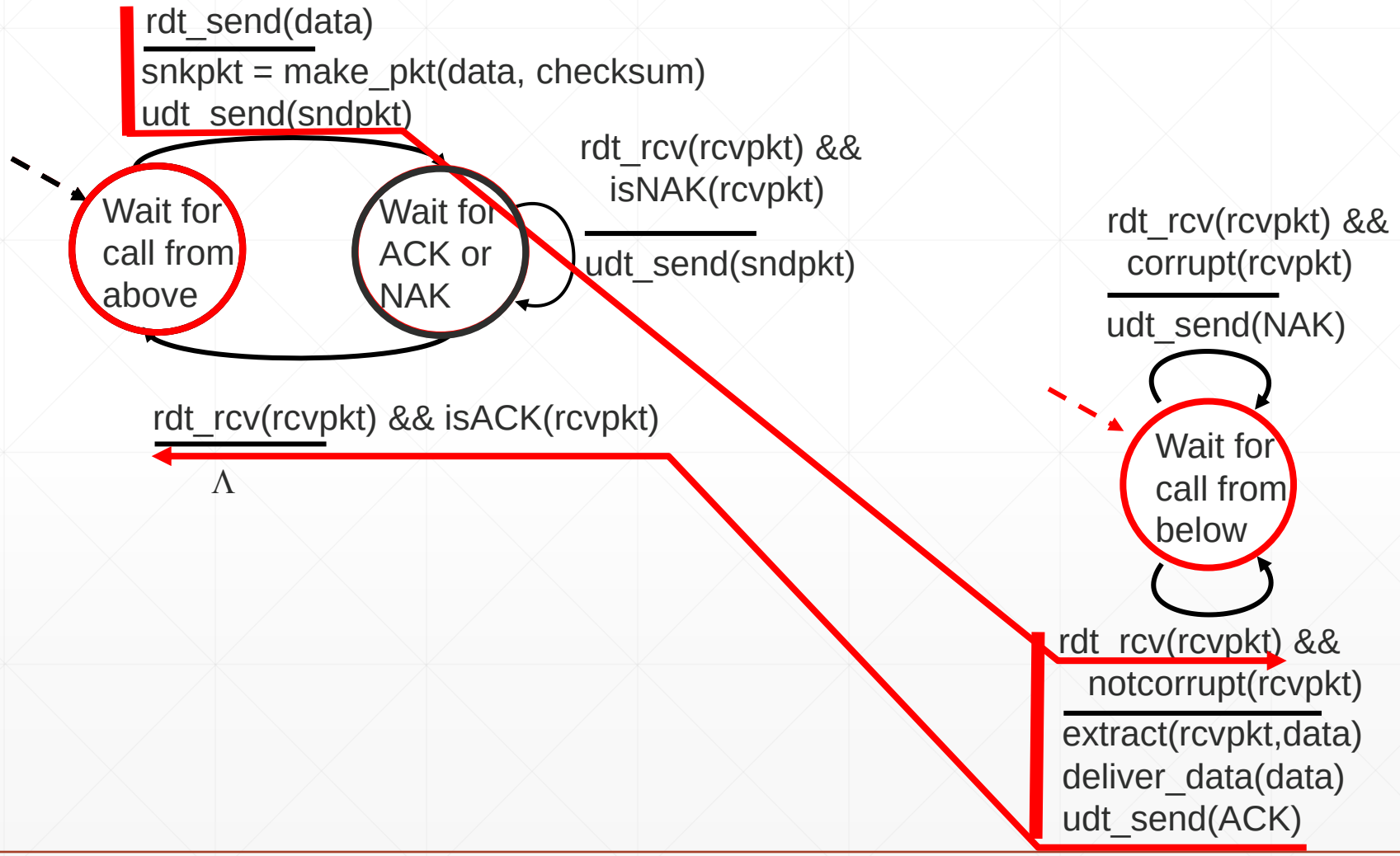
- In computer network setting, RDT protocols based on retransmissions are known as **Automatic Repeat request (ARQ) protocols**

rdt2.0: FSM specification

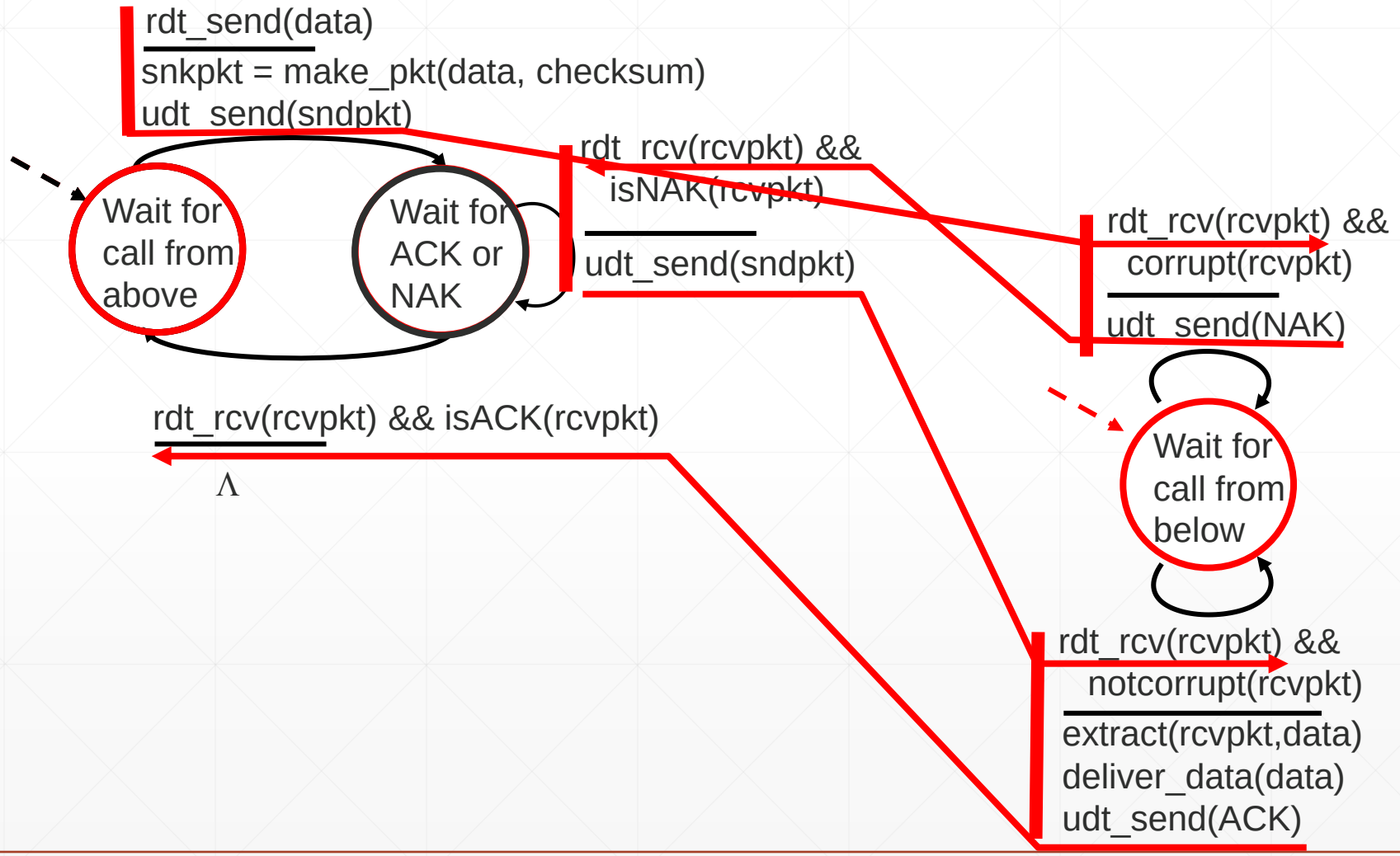




rdt2.0: operation with no errors



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

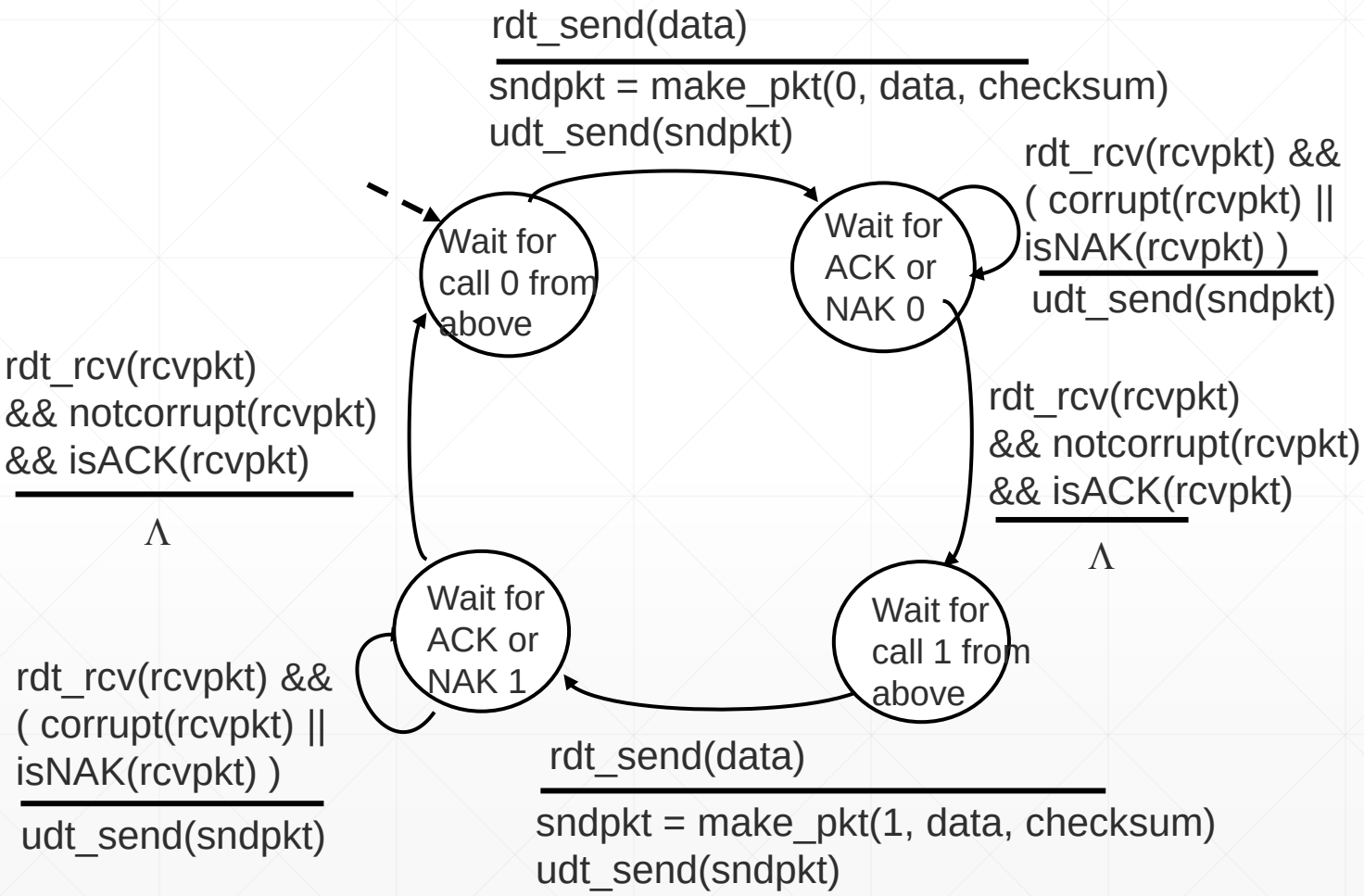
stop and wait

sender sends one packet, then waits for receiver response

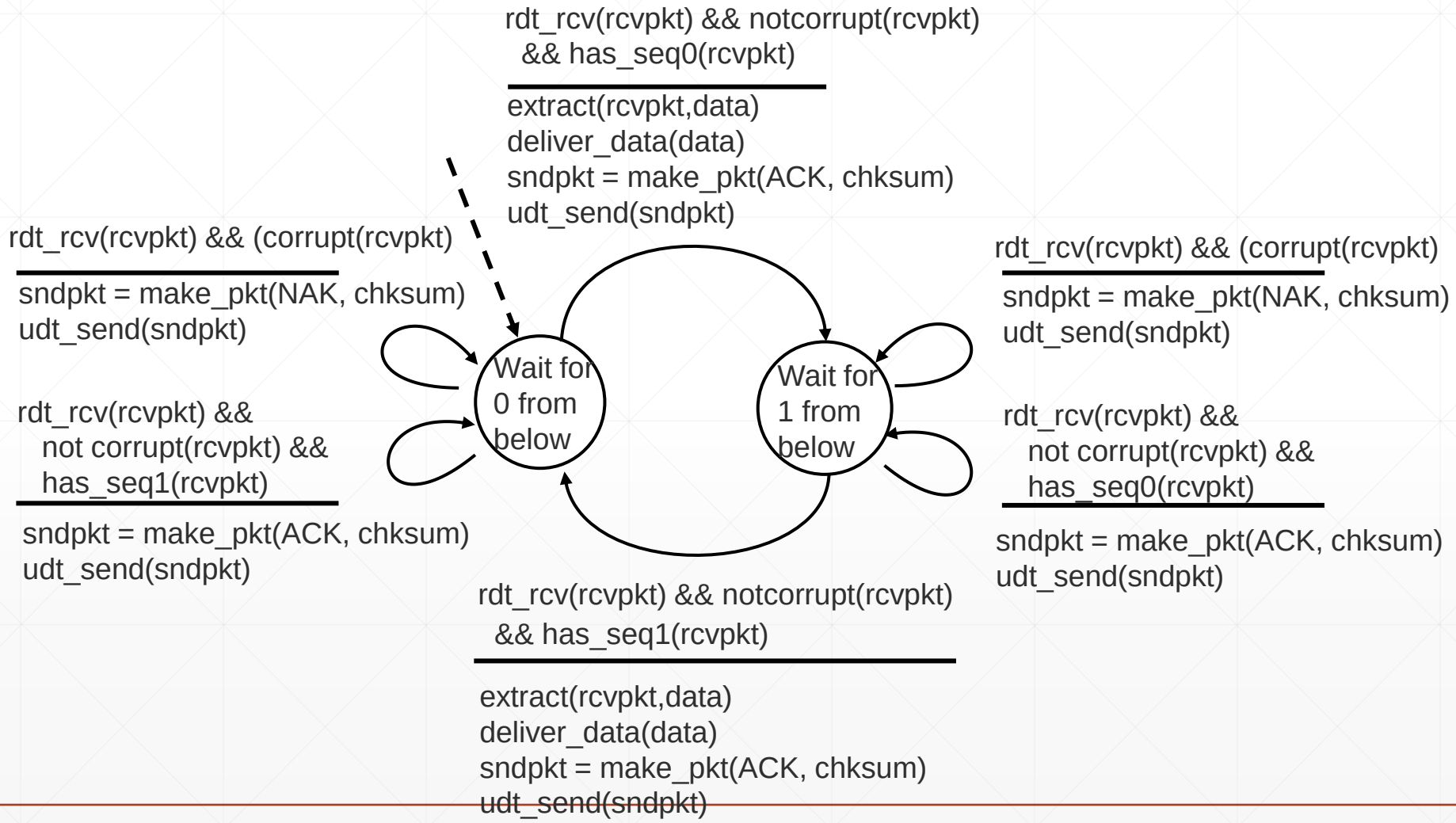
handling duplicates:

- sender retransmits current pkt if ACK/NAK corrupted
- sender adds *sequence number* to each data pkt (rdt2.1)
- receiver discards (doesn't deliver up) duplicate pkt
- When the sender is in wait for ACK or NAK state, it **cannot get more data from the upper layer**, i.e rdt_send() event cannot occur

rdt2.1: sender, handles garbled ACK/NAKs



rdt2.1: receiver, handles garbled ACK/NAKs



rdt2.1: discussion

sender:

- seq # added to pkt
- must check if received ACK/NAK corrupted
- twice as many states
  - state must "remember" whether "expected" pkt should have seq # of 0 or 1

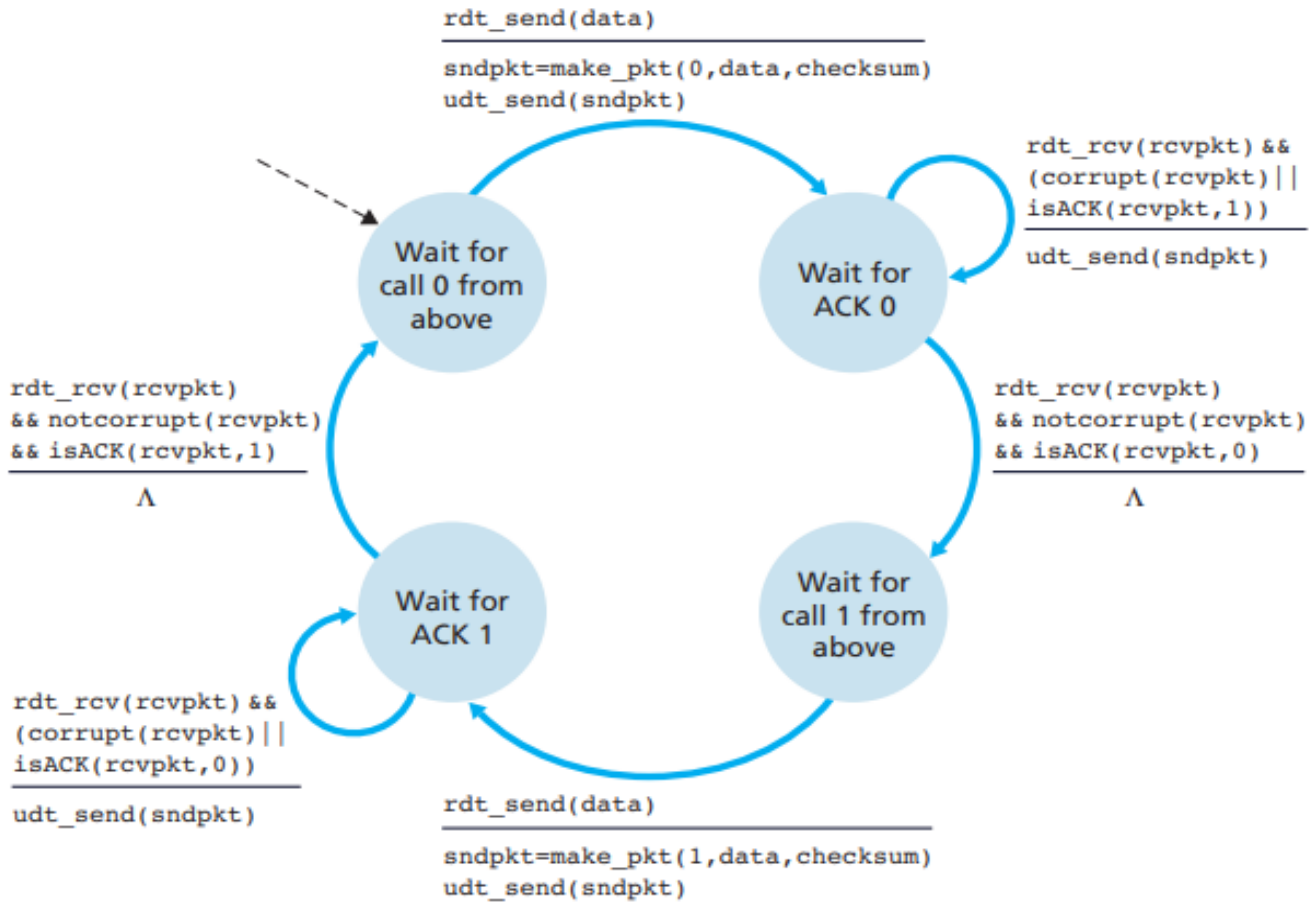
receiver:

- 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

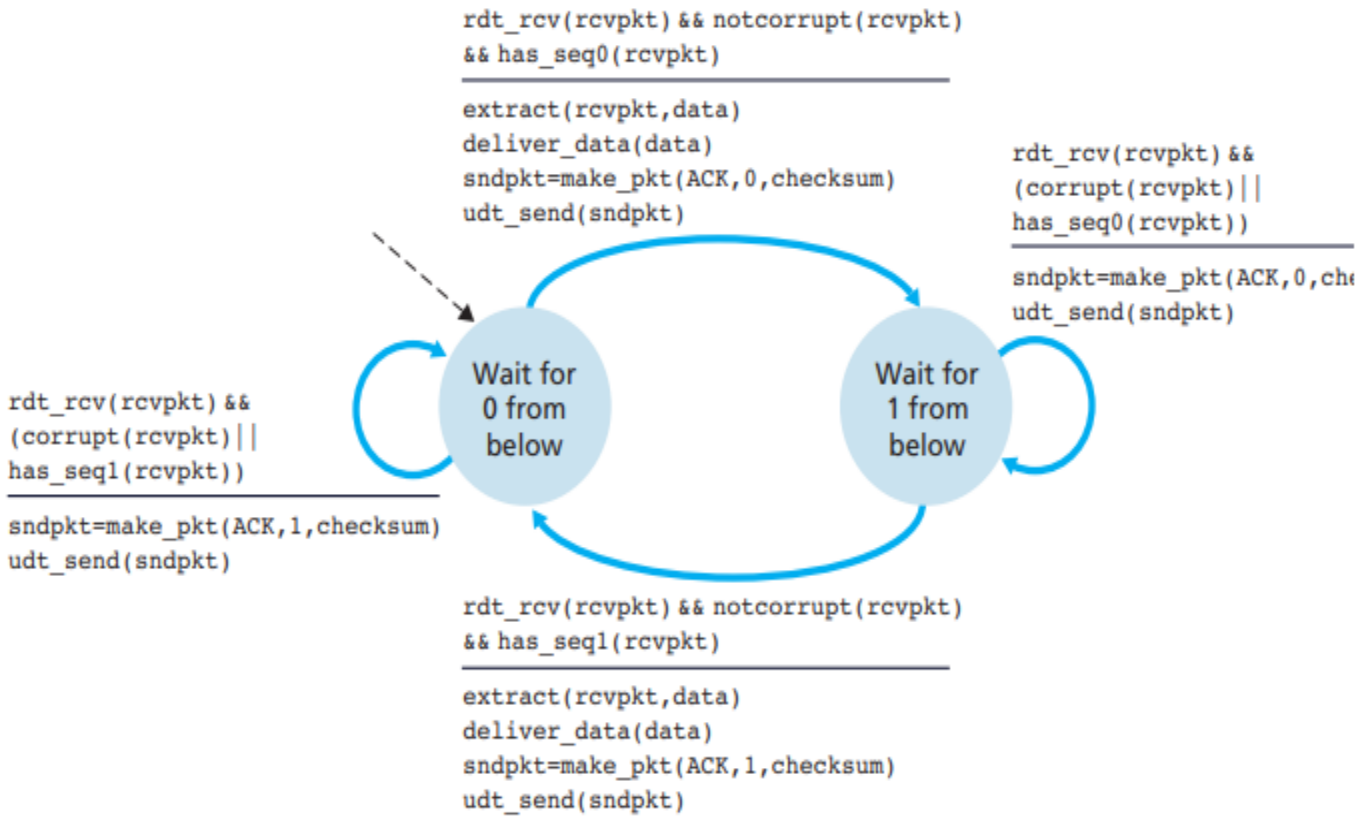
rdt2.2: a NAK-free protocol

- 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*
- Receiver must now include the **sequence number of the packet being acknowledged** and sender must check the sequence number of received acknowledge message

rdt2.2 sender



rdt2.2 receiver



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

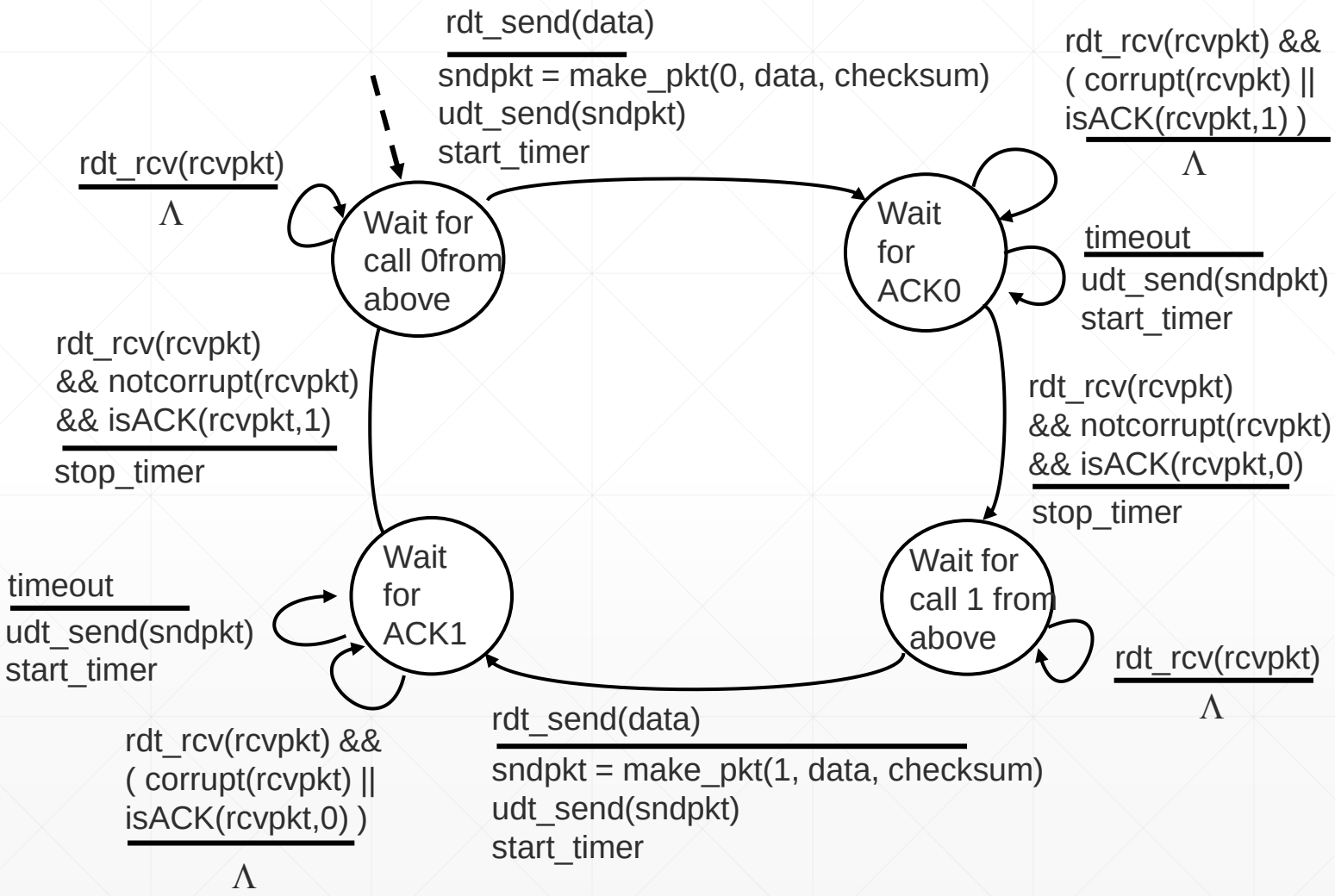
approach: sender waits "reasonable" amount of time for ACK

- 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

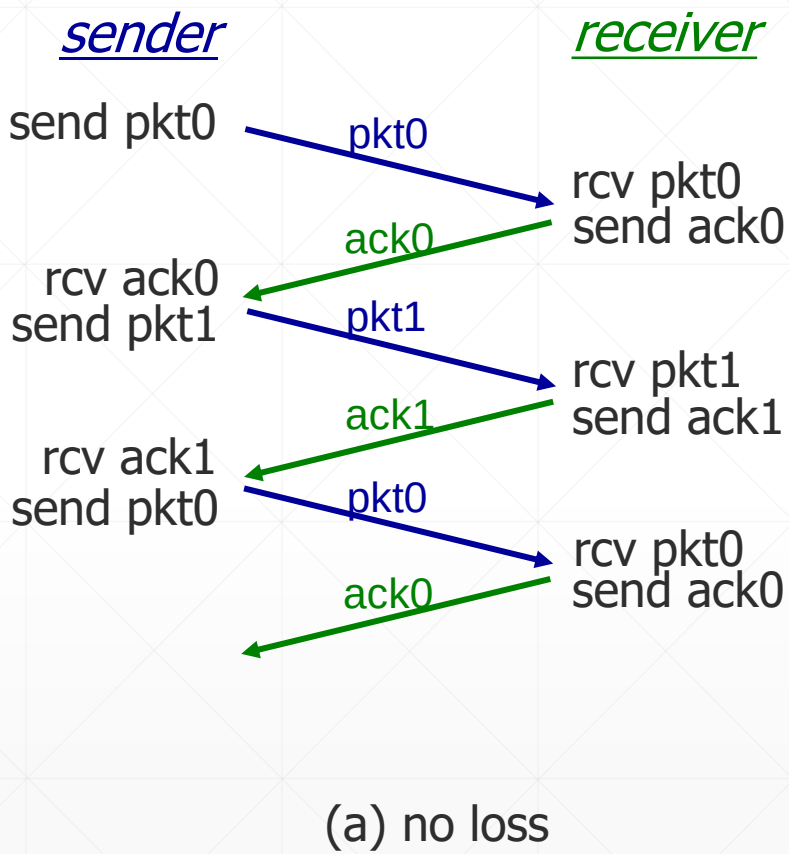
rdt3.0

- Implementing a time-based retransmission mechanism requires a **countdown timer** that can interrupt the sender after a given amount of time has expired.
- The sender will thus need to be able to
  - start the timer each time a packet is sent
  - respond to a timer interrupt (taking appropriate actions) and
  - stop the timer
- Packet sequence numbers alternate between 0 and 1, **protocol rdt3.0** is sometimes known as the **alternating-bit protocol**

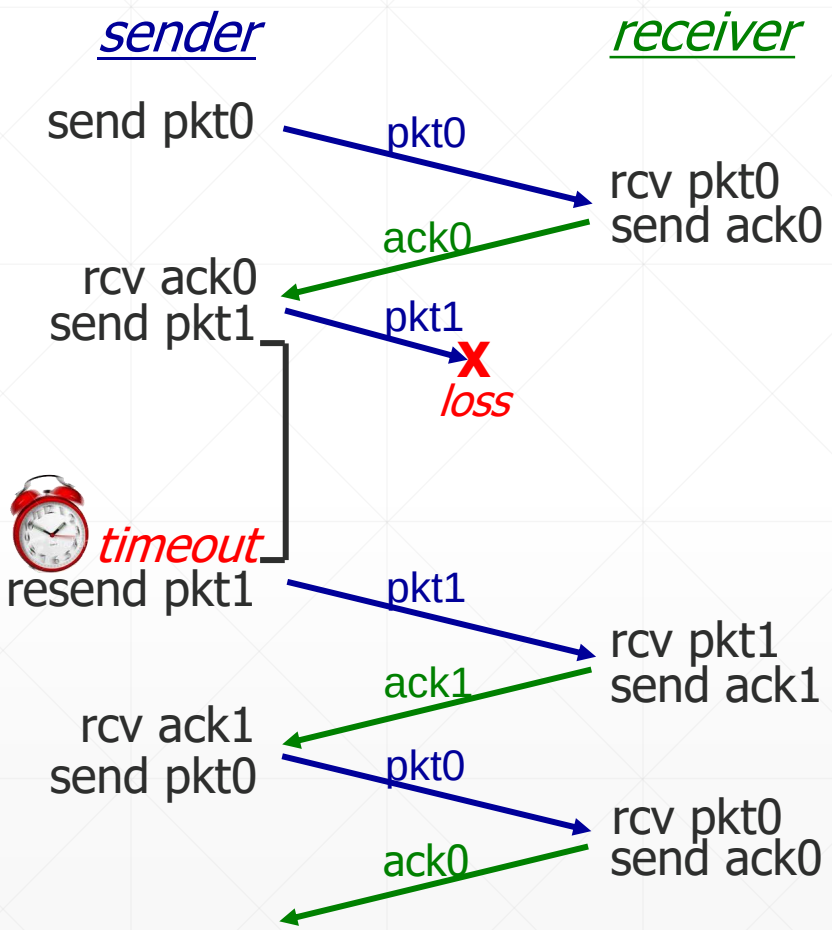
rdt3.0 sender



rdt3.0 in action

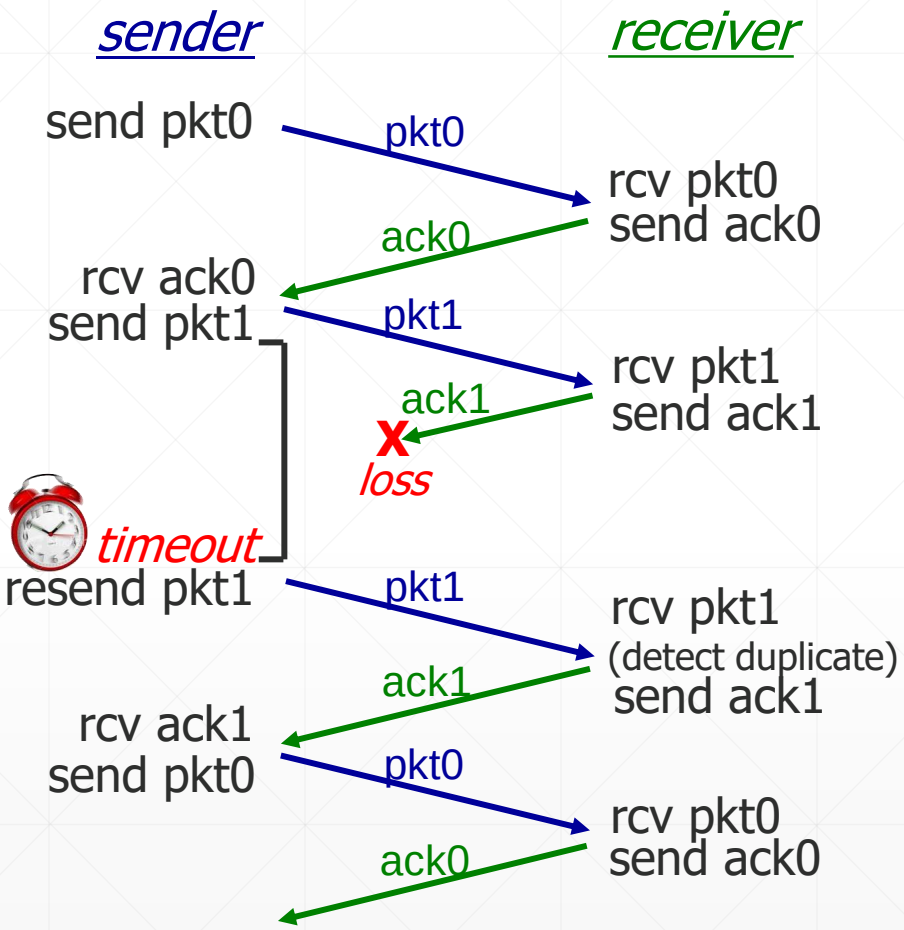


(a) no loss

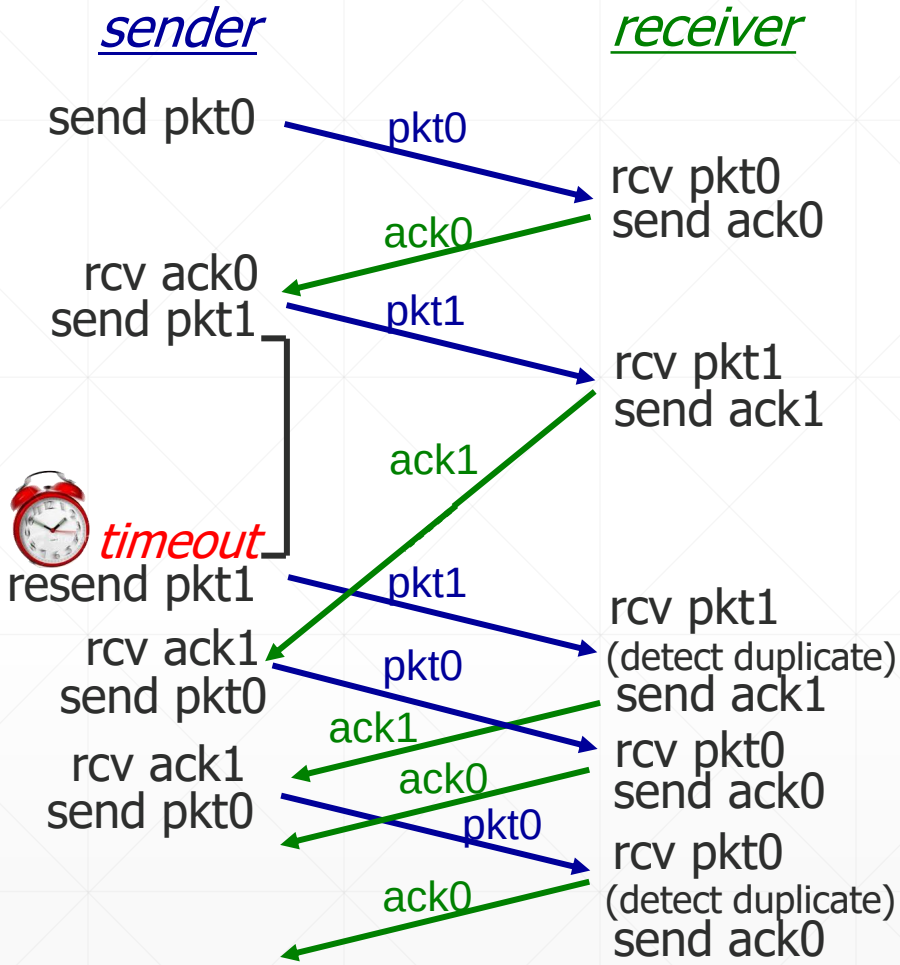


(b) packet loss

rdt3.0 in action



(c) ACK loss



(d) premature timeout/ delayed ACK

PIPELINED RELIABLE DATA TRANFER PROTOCOLS

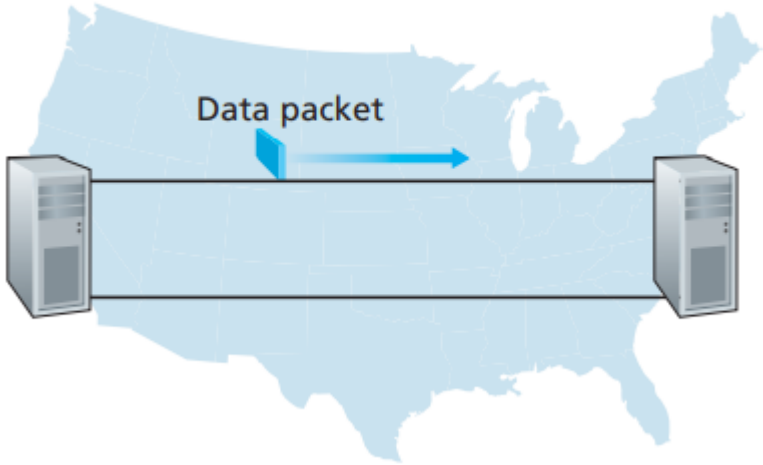
Performance of rdt3.0

- rdt3.0 is correct, but performance is bad
- e.g.:

  RTT = 30 msec

  R = 1 Gbps (10^9 bps),

  L = 1000 bytes (8000-bit packet)



Time needed to transmit the packet:

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

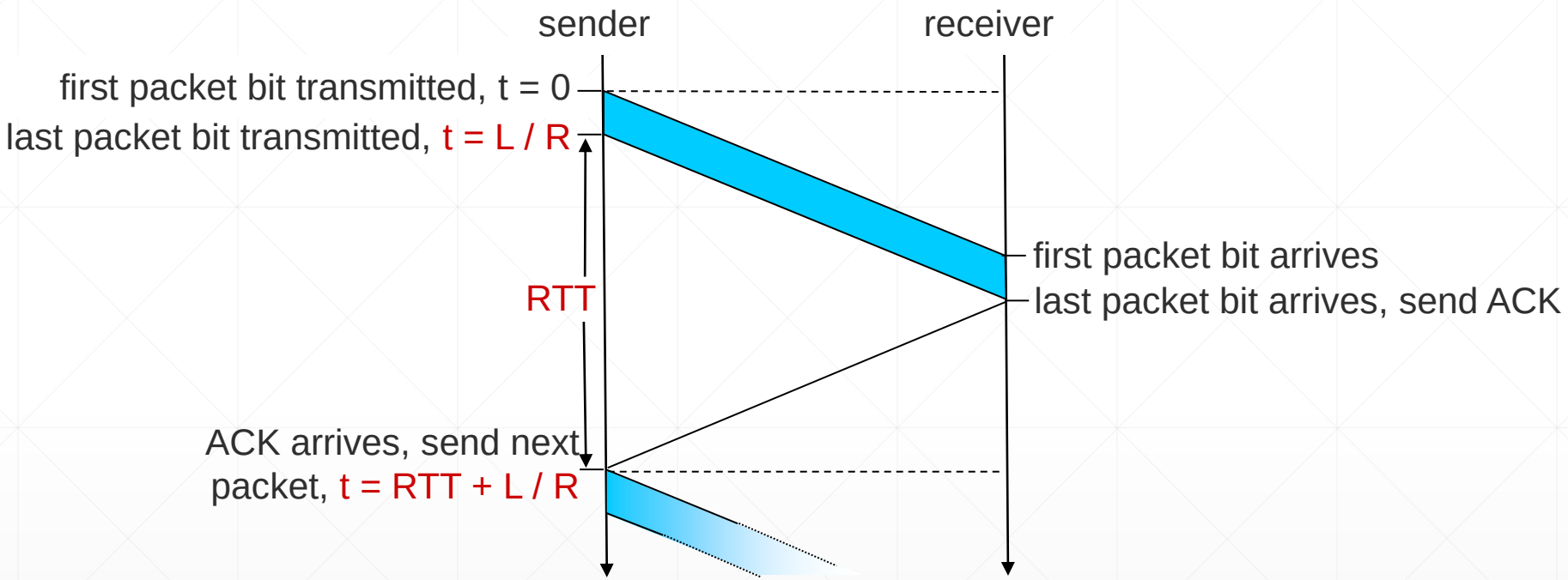
Performance of rdt3.0

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

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

- Sender sent only 1000 bytes in 30.008 msec
- Effective throughput of 267 kbps over a 1Gbps link

rdt3.0: stop-and-wait operation

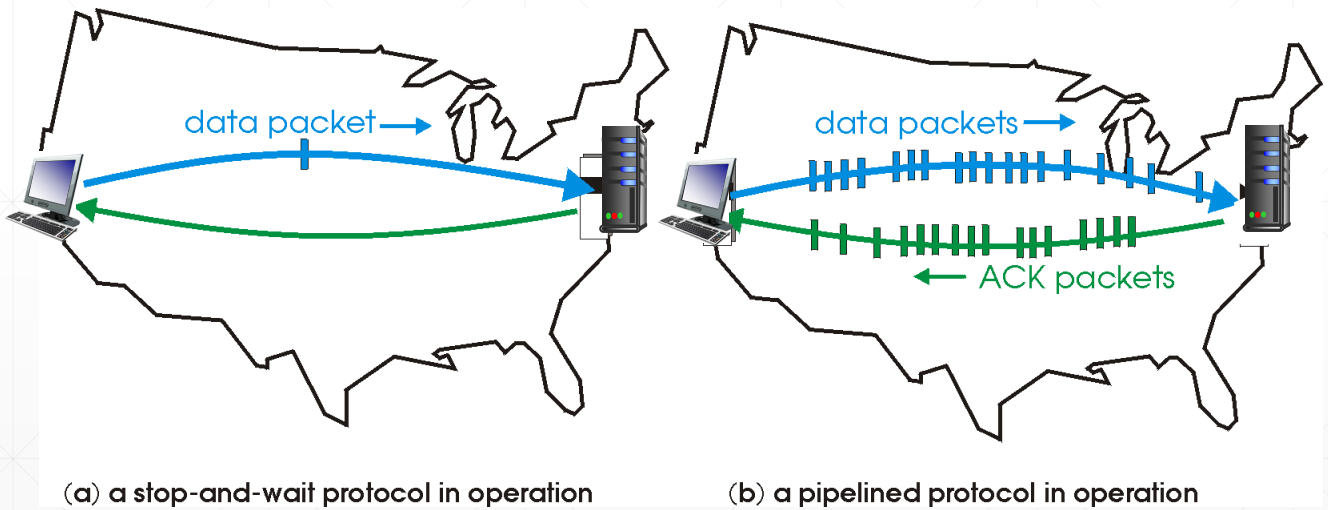


$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

Pipelined protocols

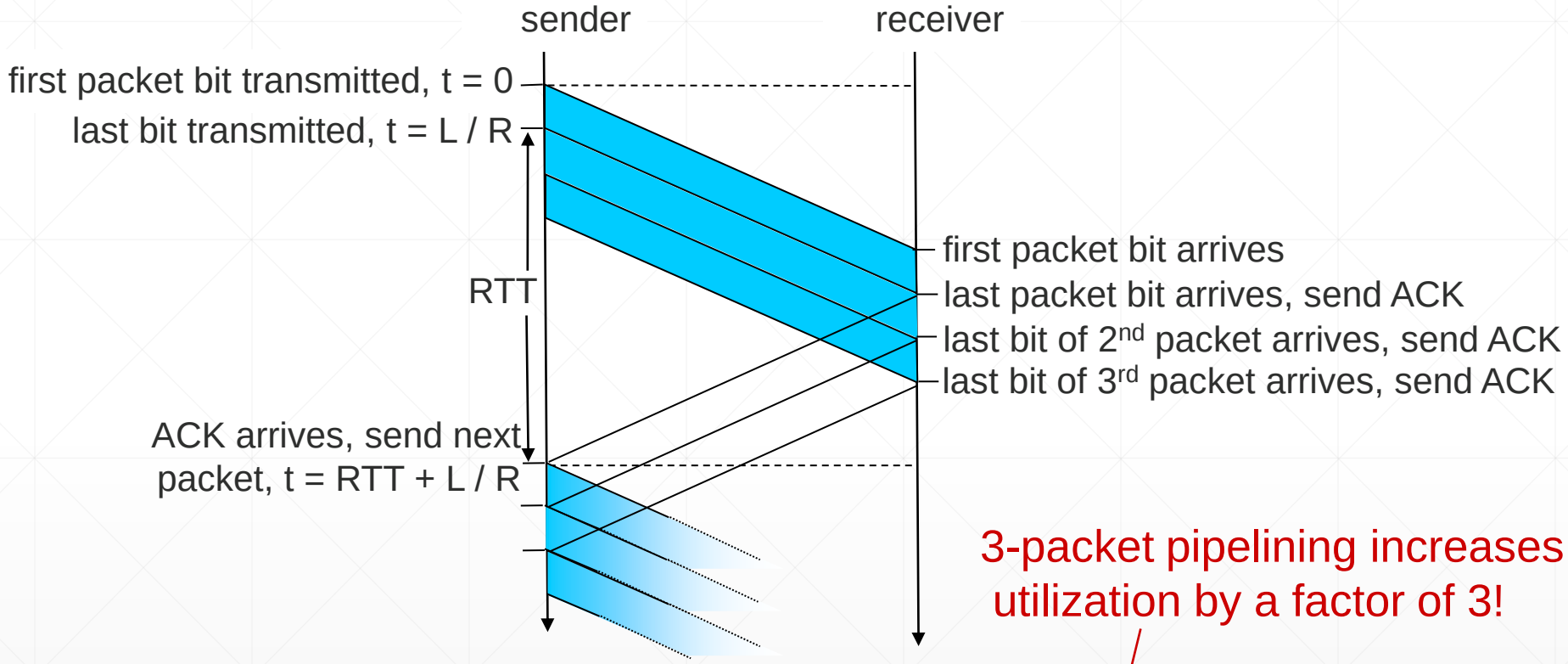
pipelining: sender allows multiple, "in-flight", yet-to-be-acknowledged pkts

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



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

Pipelining: increased utilization



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

$$U_{sender} = \frac{3L / R}{RTT + L / R} = \frac{.0024}{30.008} = 0.00081$$

Pipelined protocols: overview

Go-back-N:

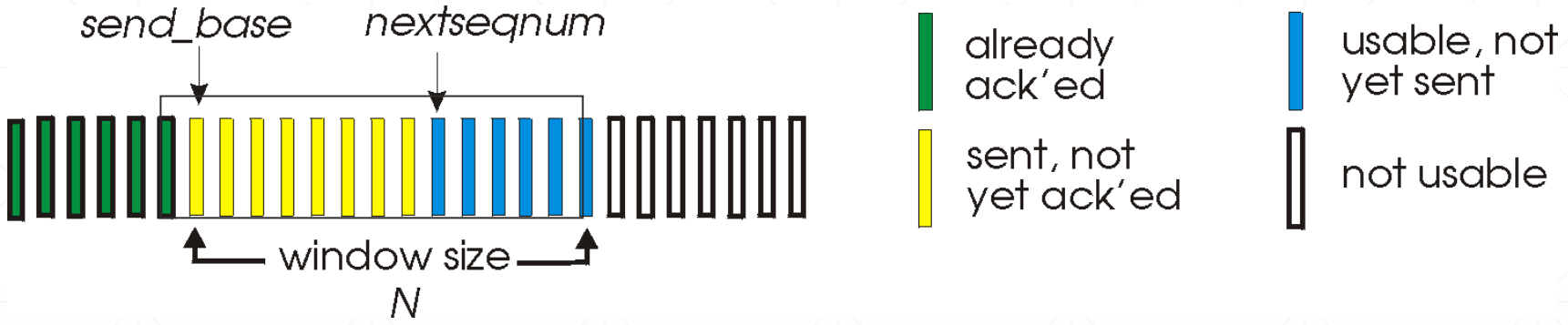
- sender can have up to N unacked packets in pipeline
- receiver only sends *cumulative ack*
  - Doesn't ack packet if there's a gap
- sender has timer for oldest unacked packet
  - when timer expires, retransmit *all* unacked packets

Selective Repeat:

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

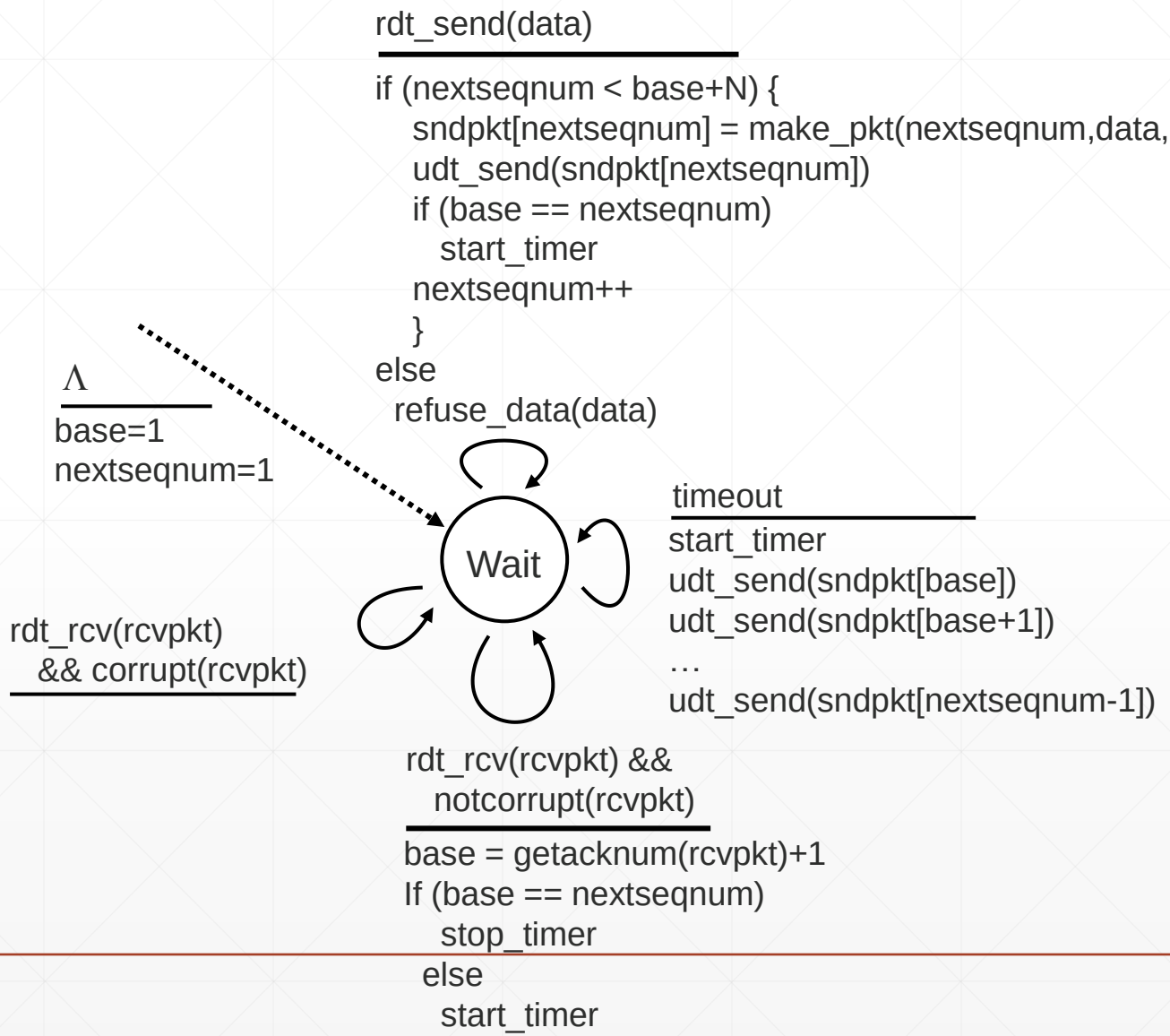
Go-Back-N: sender

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

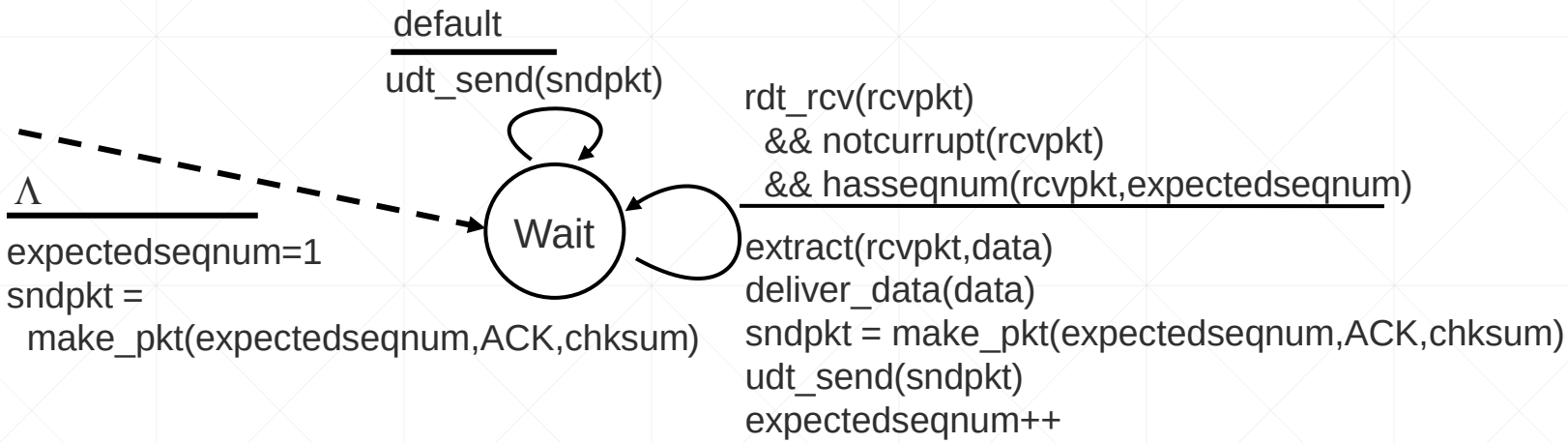


- ACK(n): ACKs all pkts up to, including seq # n - *"cumulative ACK"*
  - may receive duplicate ACKs (see receiver)
- timer for oldest in-flight pkt
- *timeout(n):* retransmit packet n and all higher seq # pkts in window

GBN: sender extended FSM



GBN: receiver extended FSM

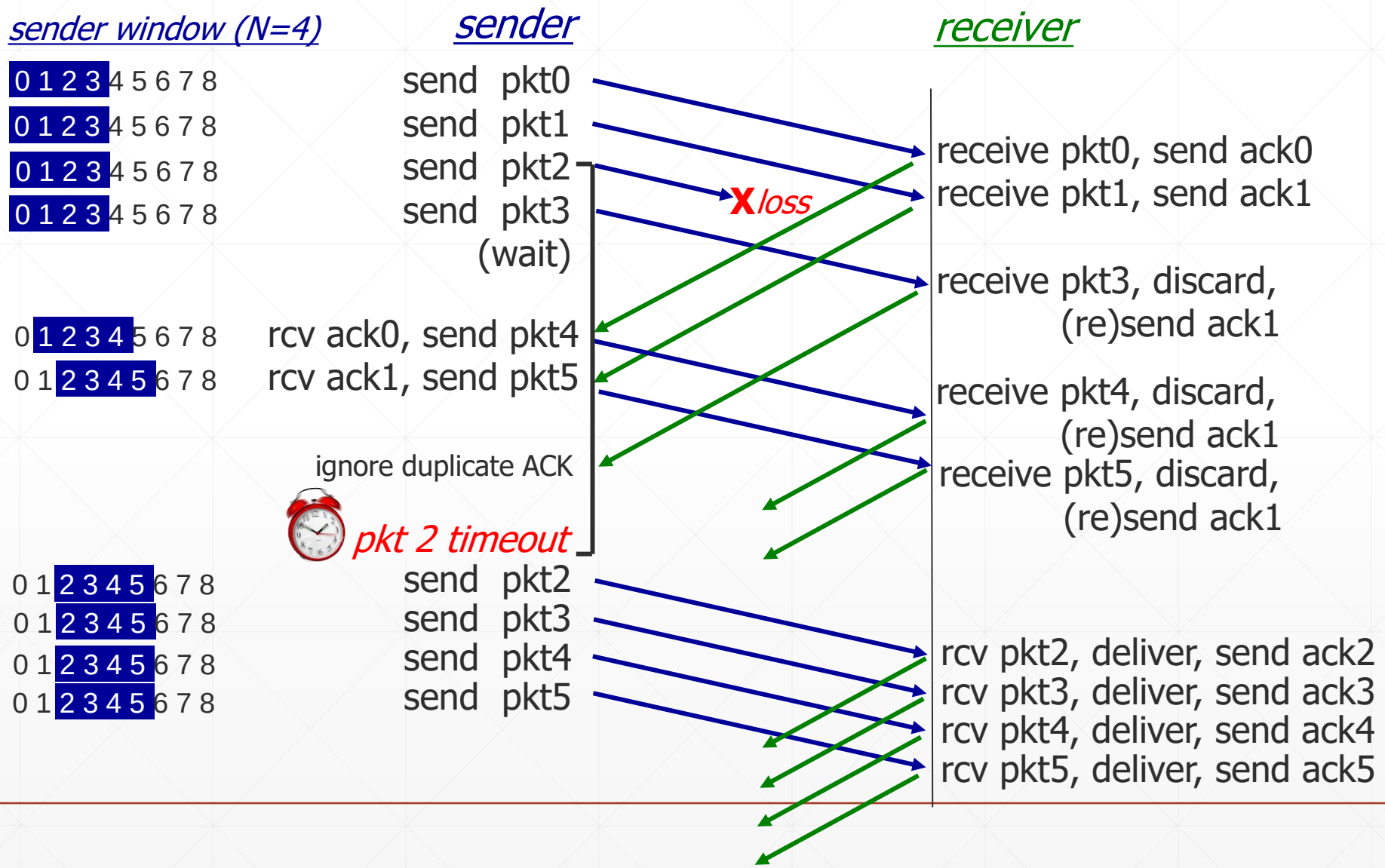


ACK-only: always send ACK for correctly-received pkt with highest *in-order* seq #

- may generate duplicate ACKs
- need only remember **`expectedseqnum`**

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

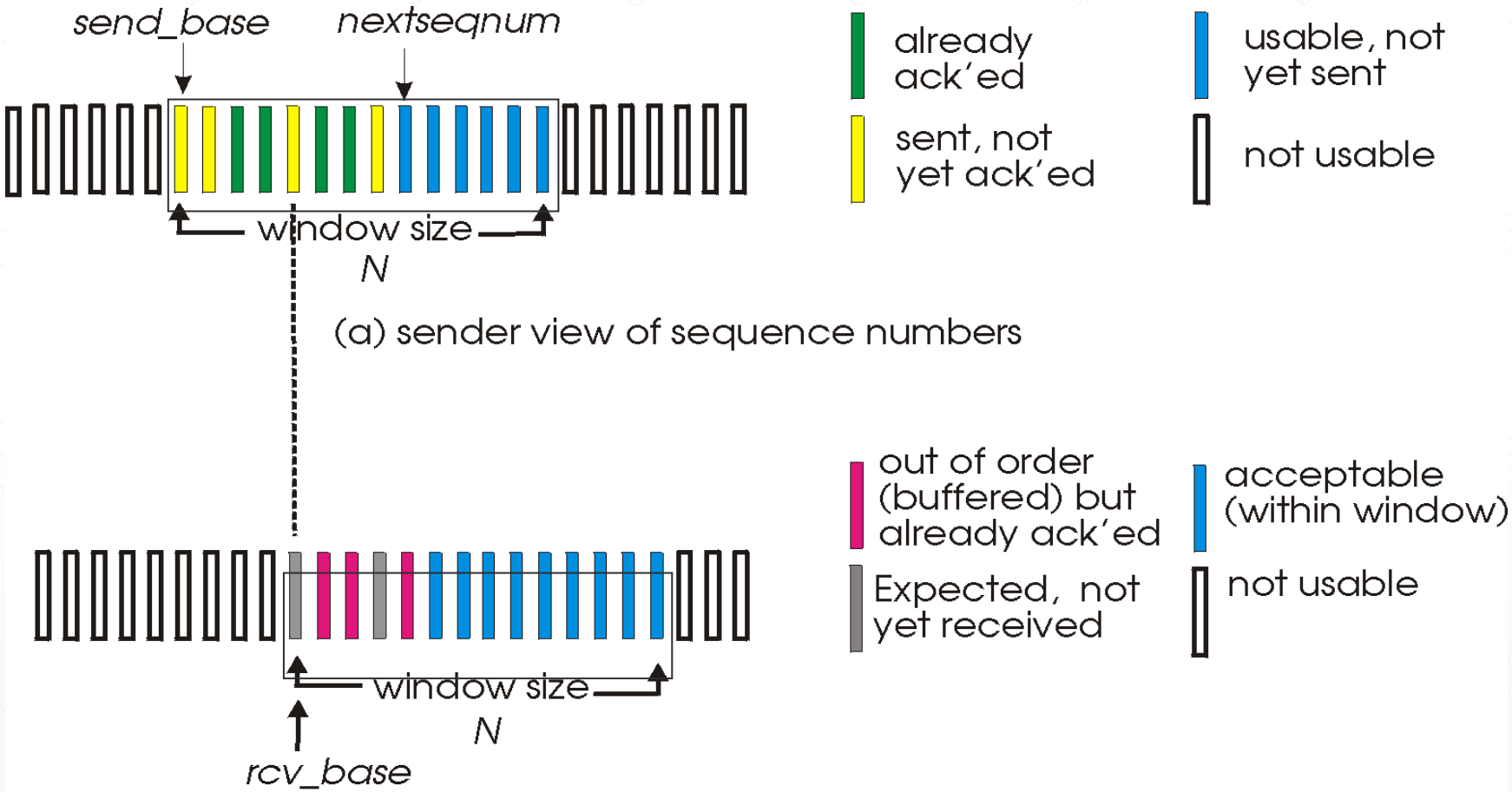
GBN in action



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
  - limits seq #s of sent, unACKed pkts

Selective repeat: sender, receiver windows



Selective repeat

sender

data from above:

- if next available seq # in window, send pkt

timeout(n):

- resend pkt n, restart timer

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

- mark pkt n as received
- if n smallest unACKed pkt, advance window base to next unACKed seq #

receiver

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

- send ACK(n)
- out-of-order: buffer
- in-order: deliver (also deliver buffered, in-order pkts), advance window to next not-yet-received pkt

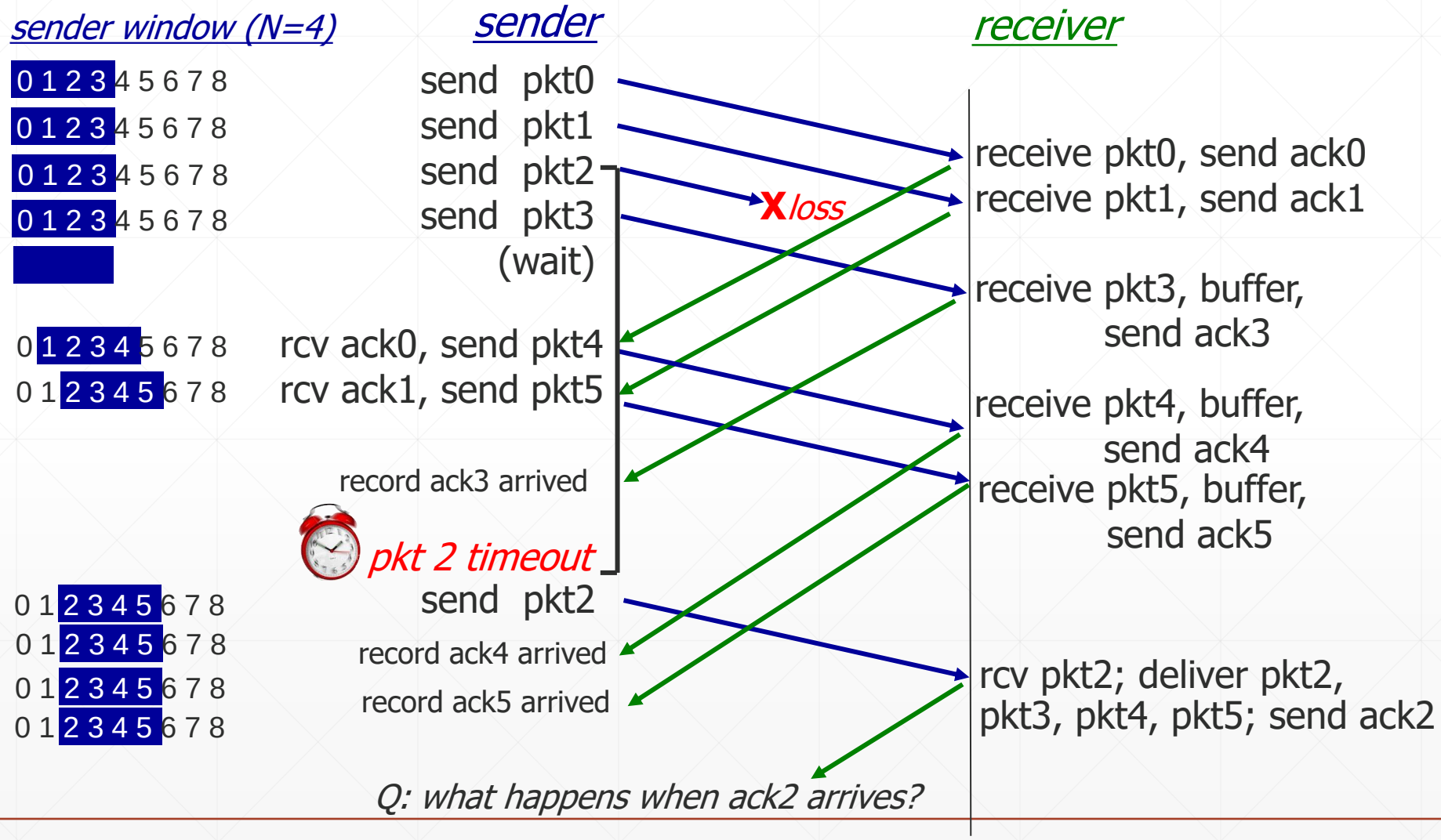
pkt n in [rcvbase-N,rcvbase-1]

- ACK(n)

otherwise:

- ignore

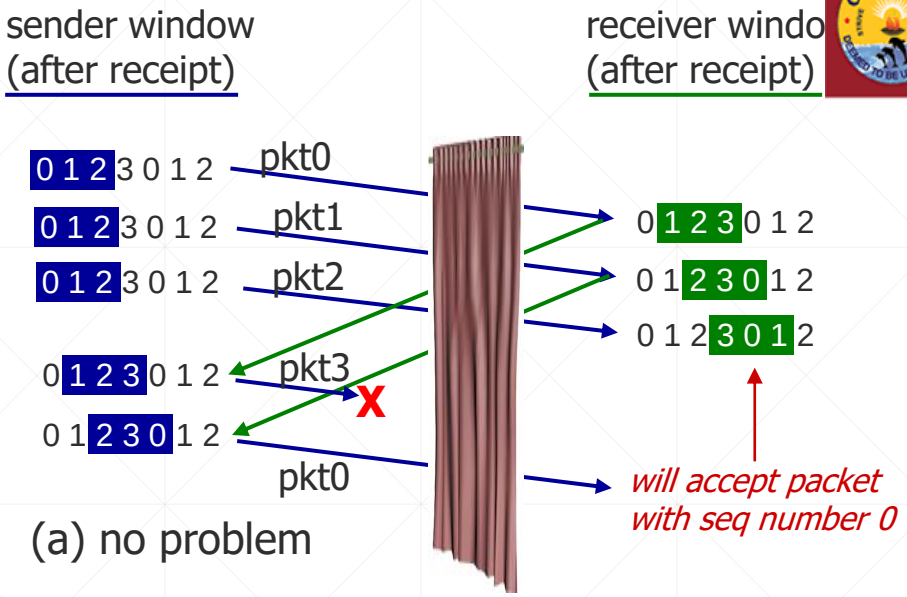
Selective repeat in action



Selective repeat: dilemma

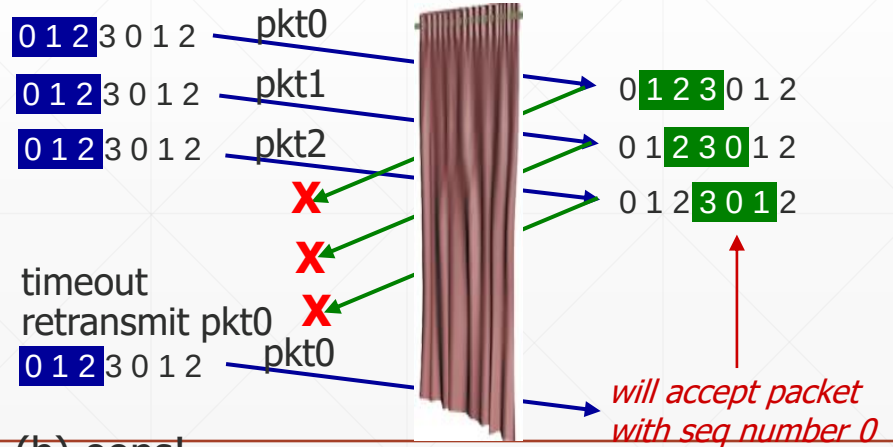
example:

- seq #'s: 0, 1, 2, 3
- window size=3
  - receiver sees no difference in two scenarios!
  - duplicate data accepted as new in (b)



(a) no problem

receiver can't see sender side.
receiver behavior identical in both cases!
something's (very) wrong!



(b) oops!

Selective repeat

- Window size should be less than or equal to half the sequence number in SR protocol.
- This is to avoid packets being recognized incorrectly.
- If the windows size is greater than half the sequence number space,
  - then if an ACK is lost, the sender may send new packets that the receiver believes are retransmissions.

Connection Oriented Protocol : TCP

- 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

❖full duplex data:

- bi-directional data flow in same connection
- MSS: maximum segment size

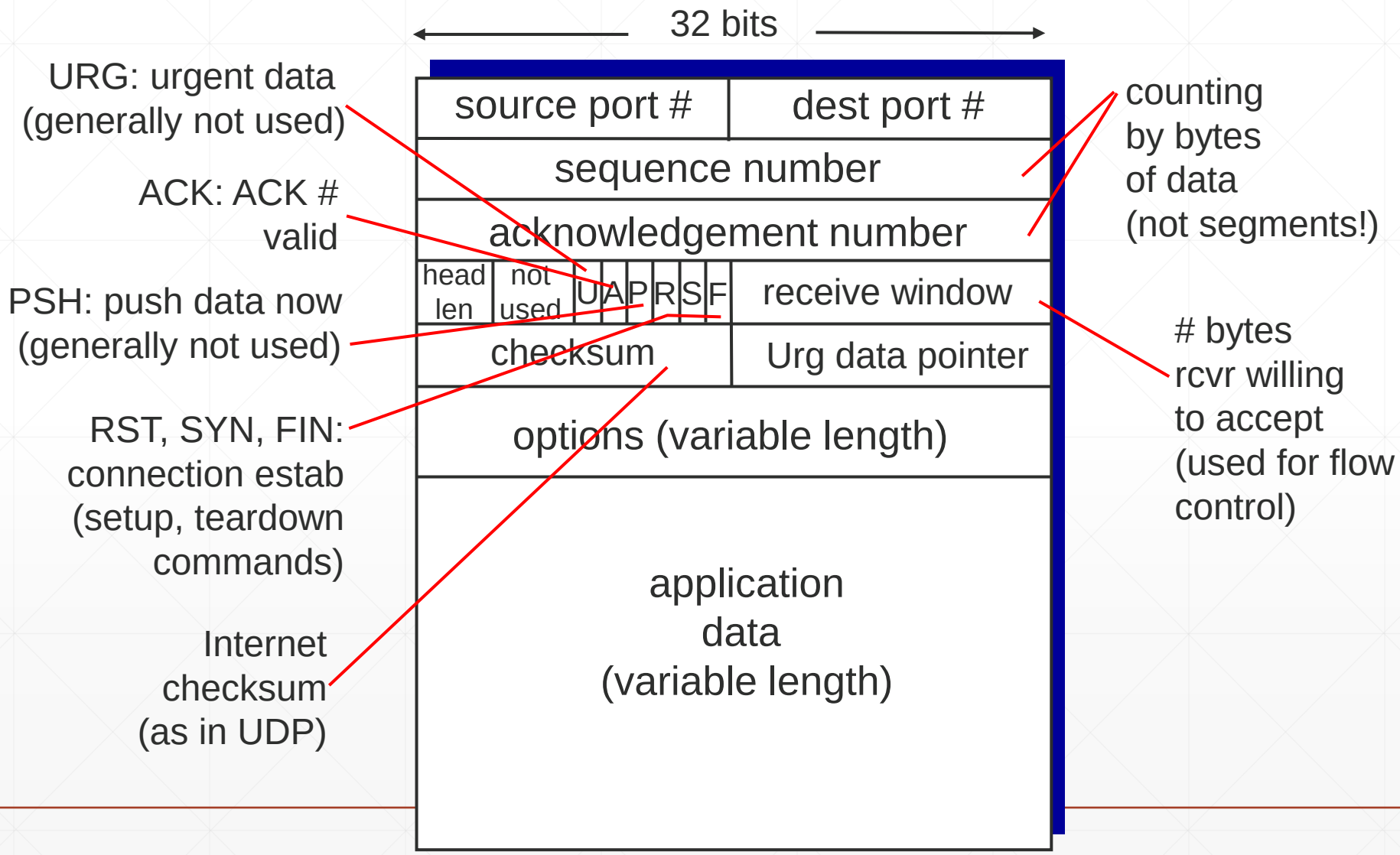
❖connection-oriented:

- handshaking (exchange of control msgs) inits sender, receiver state before data exchange

❖flow controlled:

- sender will not overwhelm receiver

TCP segment structure



TCP Seq. numbers, ACKs

sequence numbers:

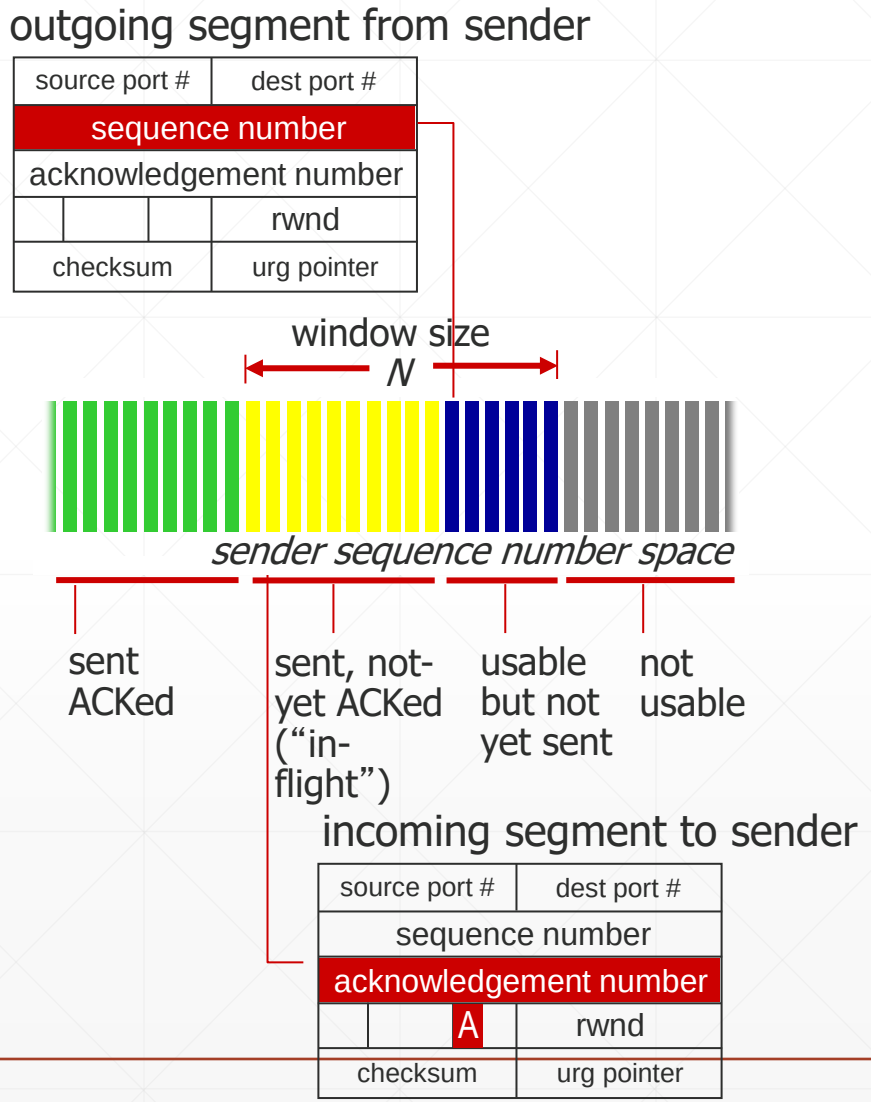
- byte stream "number" of first byte in segment's data

acknowledgements:

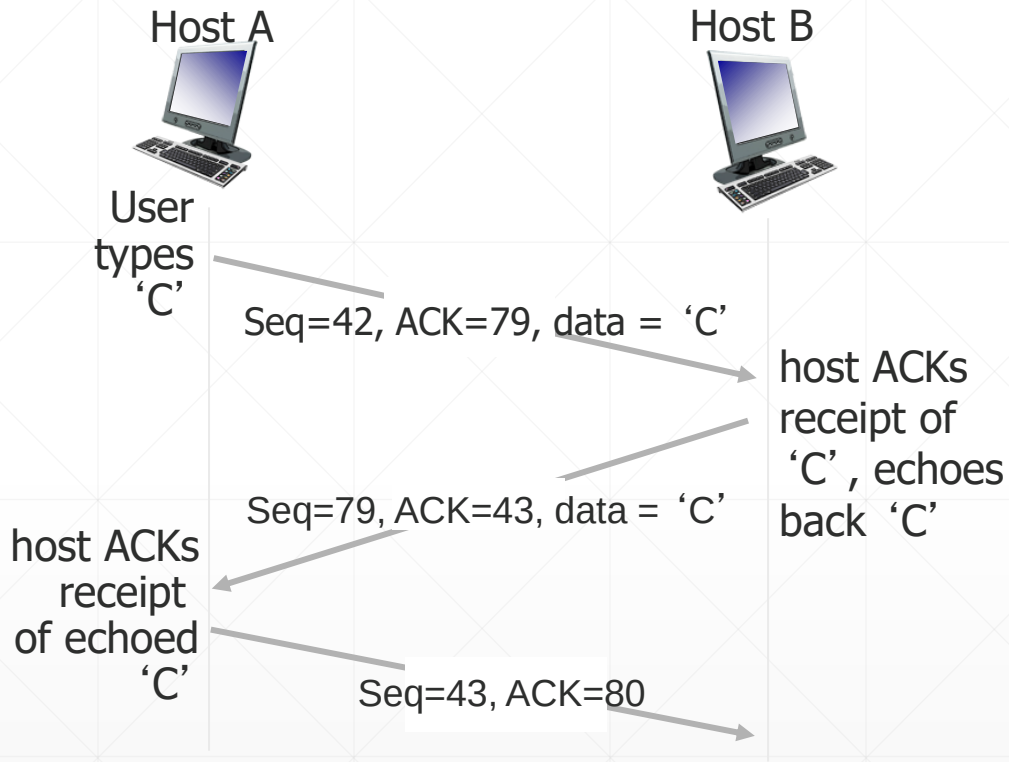
- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

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



TCP seq. numbers, ACKs



simple telnet scenario

TCP Round trip time, Timeout

Q: how to set TCP timeout value?

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

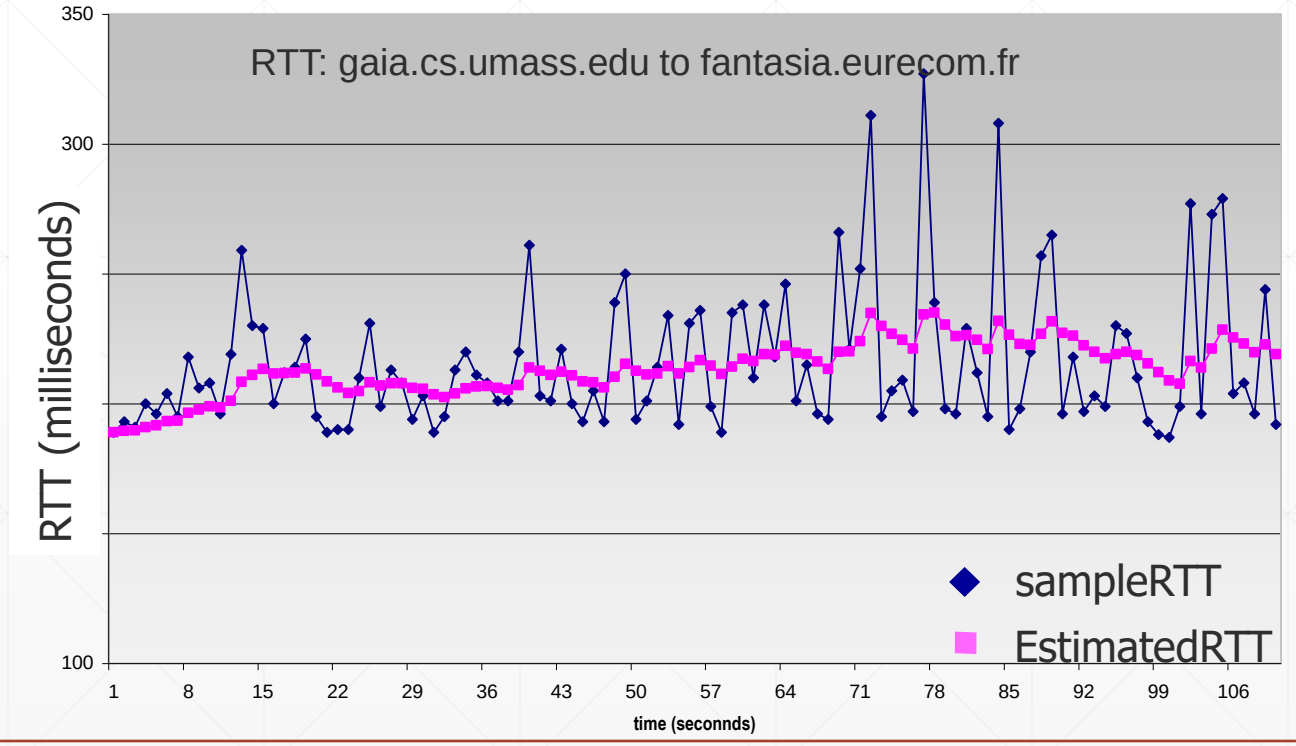
Q: how to estimate RTT?

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

TCP round trip time, timeout

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

- exponential weighted moving average
- $\alpha = 0.125$



time (seconds)

TCP round trip time, timeout

- timeout interval: **EstimatedRTT** plus "safety margin"
  - large variation in **EstimatedRTT** -> larger safety margin
- estimate SampleRTT deviation from EstimatedRTT(RTT variation):

$$DevRTT = (1-\beta)*DevRTT + \beta*|SampleRTT-EstimatedRTT|$$

(typically, $\beta = 0.25$)

$$TimeoutInterval = EstimatedRTT + 4*DevRTT$$

estimated RTT

"safety margin"

TCP reliable data transfer

- TCP creates rdt service on top of IP's unreliable service
  - pipelined segments
  - cumulative acks
  - single retransmission timer
- retransmissions triggered by:
  - timeout events
  - duplicate acks

Let's initially consider simplified TCP sender:

- ignore duplicate acks
- ignore flow control, congestion control

TCP sender events:

data rcvd from app:

- create segment with seq #
- seq # is byte-stream number of first data byte in segment
- start timer if not already running
  - think of timer as for oldest unacked segment
  - expiration interval: **`TimeOutInterval`**

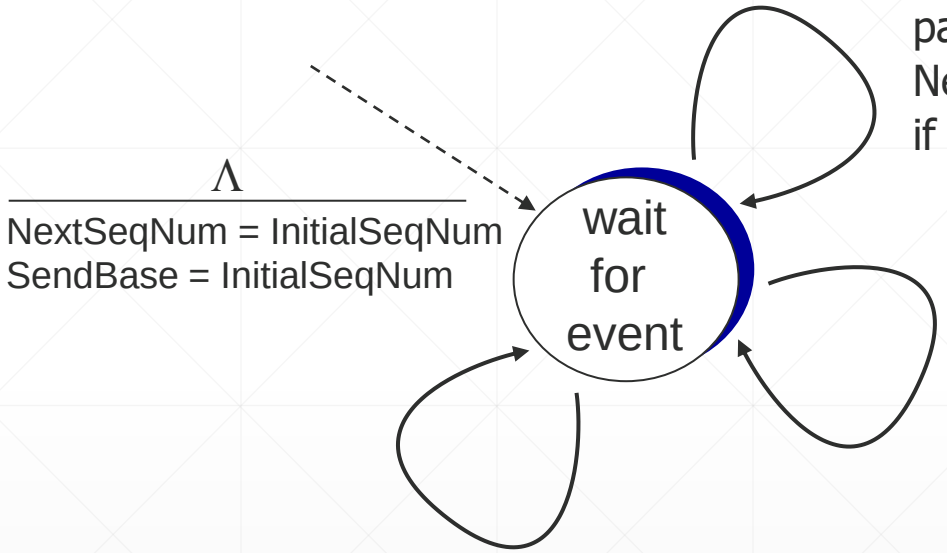
timeout:

- retransmit segment that caused timeout
- restart timer

ack rcvd:

- if ack acknowledges previously unacked segments
  - update what is known to be ACKed
  - start timer if there are still unacked segments

TCP sender (simplified)



data received from application above

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

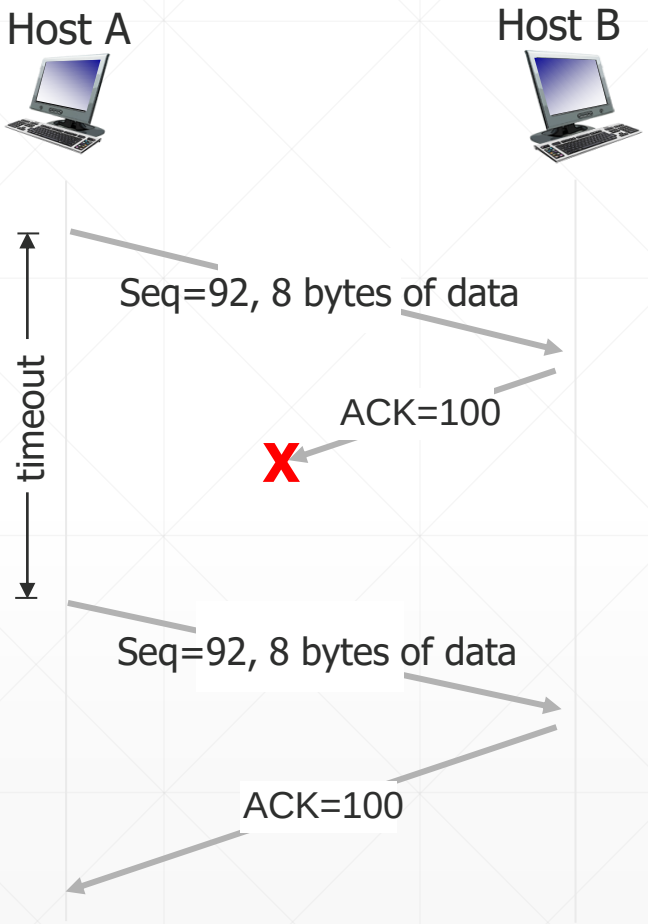
timeout

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

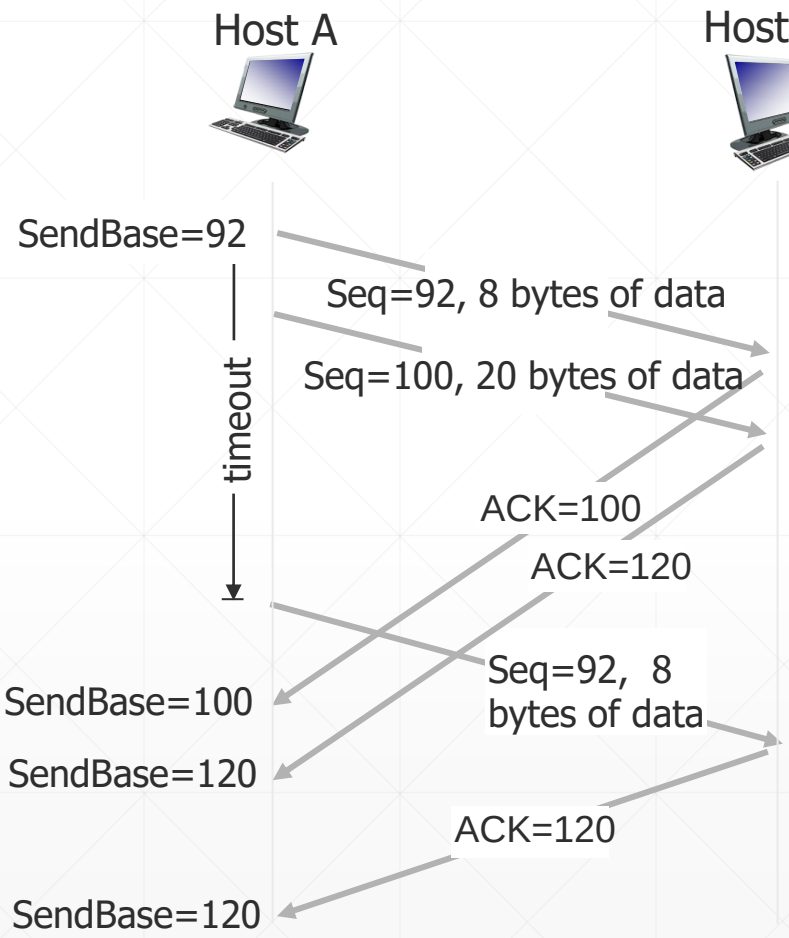
ACK received, with ACK field value y

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

TCP: retransmission scenarios

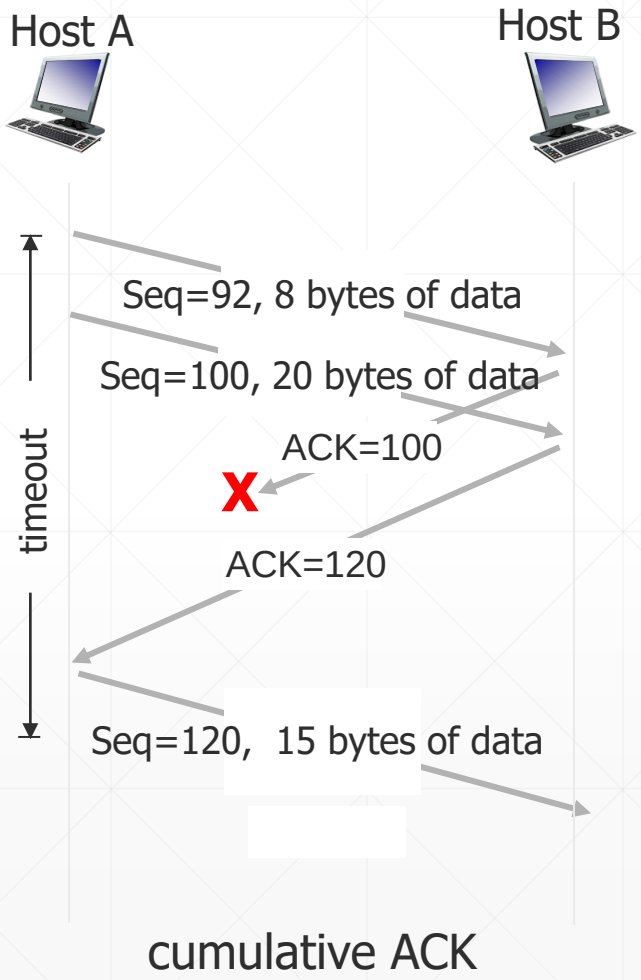


lost ACK scenario



premature timeout

TCP: retransmission scenarios



cumulative ACK

TCP ACK generation [RFC 1122, RFC 2581]

Event	TCP Receiver Action
Arrival of in-order segment with expected sequence number. All data up to expected sequence number already acknowledged.	Delayed ACK. Wait up to 500 msec for arrival of another in-order segment. If next in-order segment does not arrive in this interval, send an ACK.
Arrival of in-order segment with expected sequence number. One other in-order segment waiting for ACK transmission.	Immediately send single cumulative ACK, ACKing both in-order segments.
Arrival of out-of-order segment with higher-than-expected sequence number. Gap detected.	Immediately send duplicate ACK, indicating sequence number of next expected byte (which is the lower end of the gap).
Arrival of segment that partially or completely fills in gap in received data.	Immediately send ACK, provided that segment starts at the lower end of gap.

TCP fast retransmit

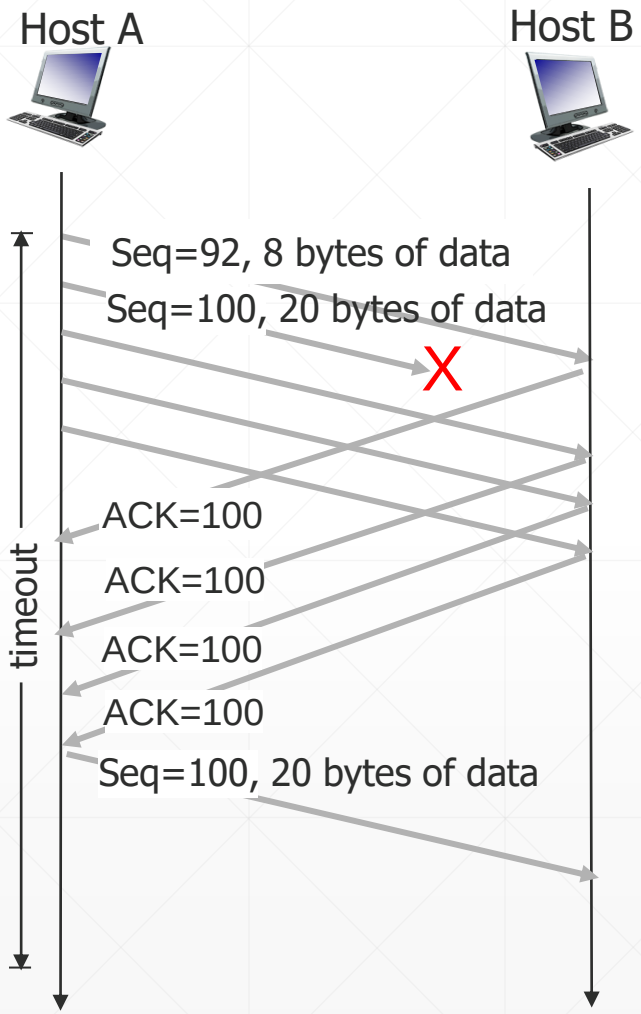
- time-out period often relatively long:
  - long delay before resending lost packet
- detect lost segments via duplicate ACKs.
  - sender often sends many segments back-to-back
  - if segment is lost, there will likely be many duplicate ACKs.

TCP fast retransmit

if sender receives 3 ACKs for same data ("triple duplicate ACKs"), resend unacked segment with smallest seq #

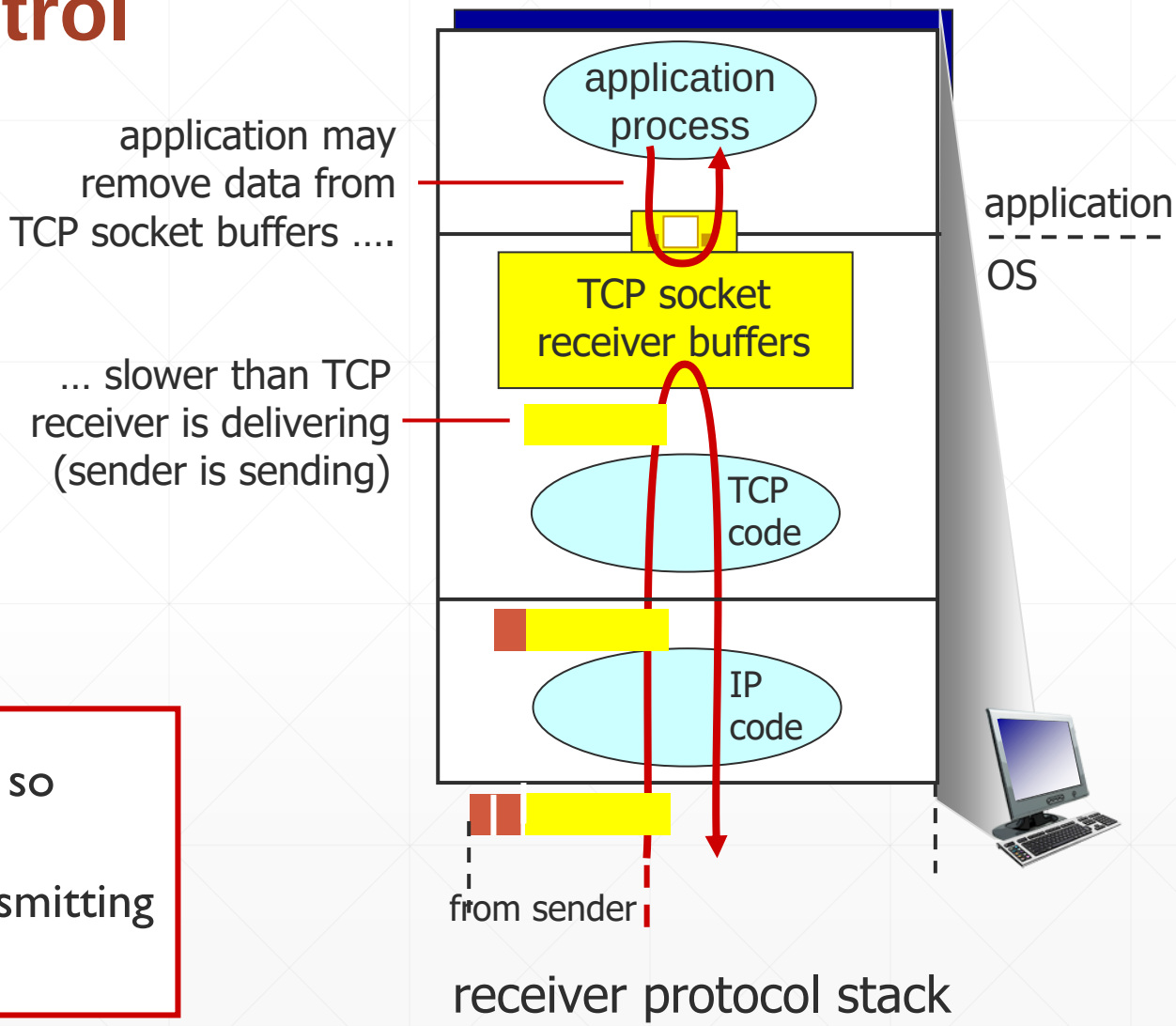
- likely that unacked segment lost, so don't wait for timeout

TCP Fast retransmit



fast retransmit after sender receipt of triple duplicate ACK

TCP Flow control

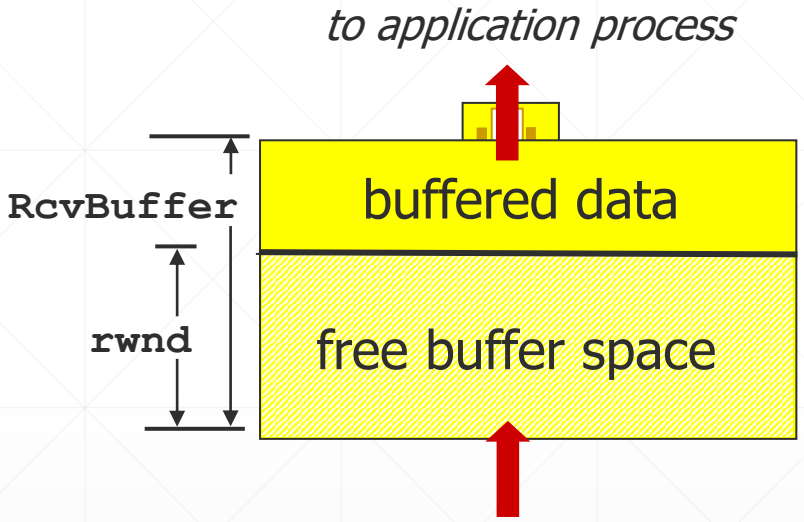


flow control

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

TCP Flow control

- receiver "advertises" free buffer space by including **`rwnd`** value in TCP header of receiver-to-sender segments
  - **`RcvBuffer`** size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust **`RcvBuffer`**
- sender limits amount of unacked ("in-flight") data to receiver's **`rwnd`** value
- guarantees receive buffer will not overflow

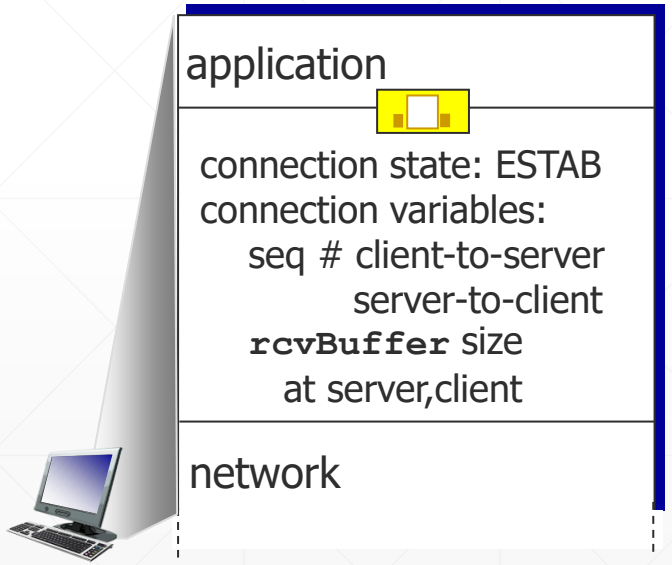


receiver-side buffering

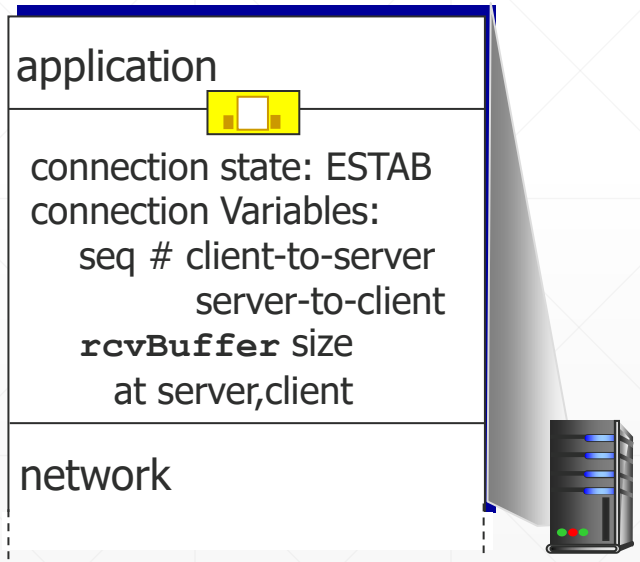
Connection Management

before exchanging data, sender/receiver ”handshake” :

- agree to establish connection (each knowing the other willing to establish connection)
- agree on connection parameters

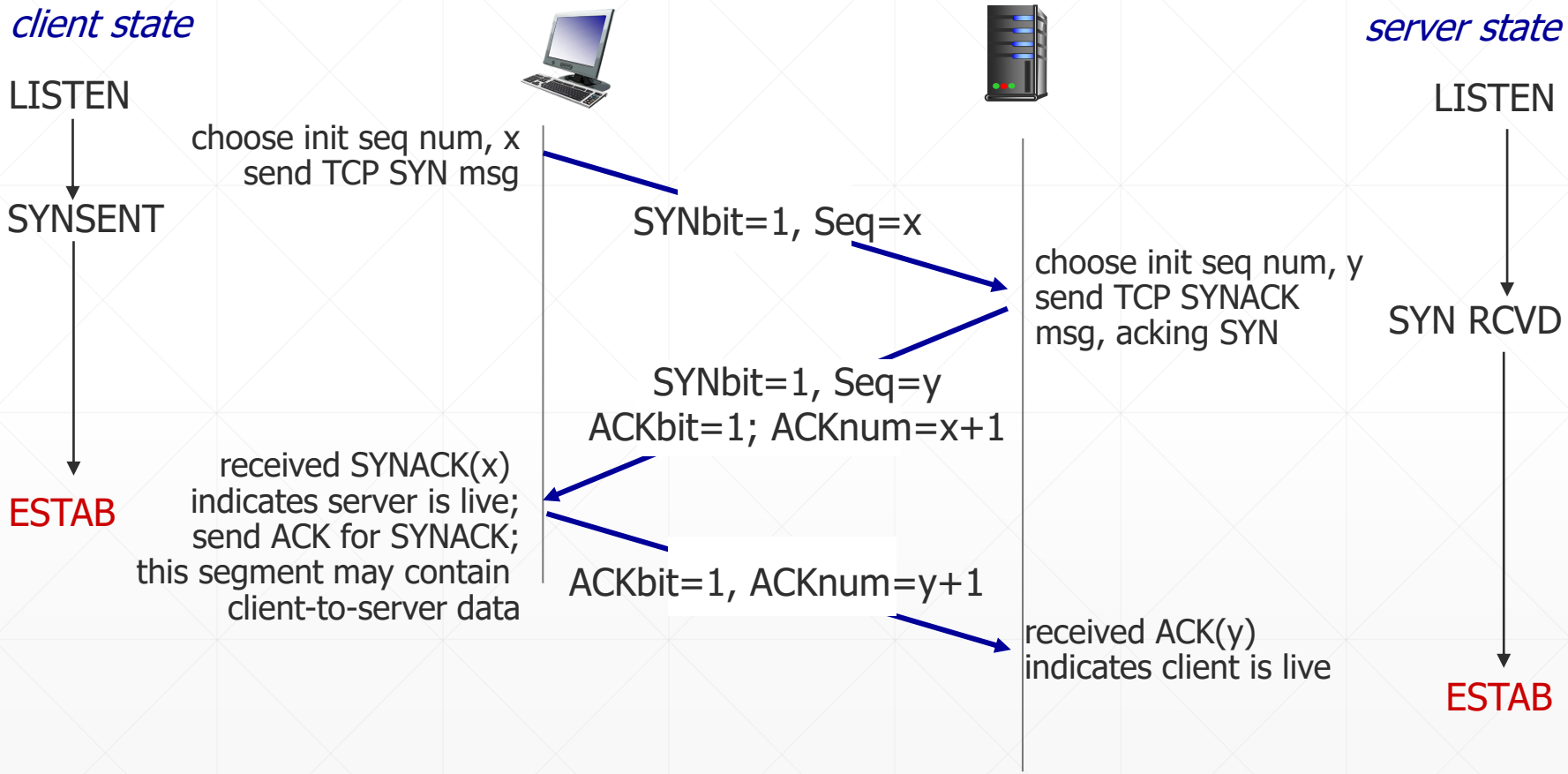



```
Socket clientSocket =
  newSocket("hostname","port
   number");
```




```
Socket connectionSocket =
  welcomeSocket.accept();
```

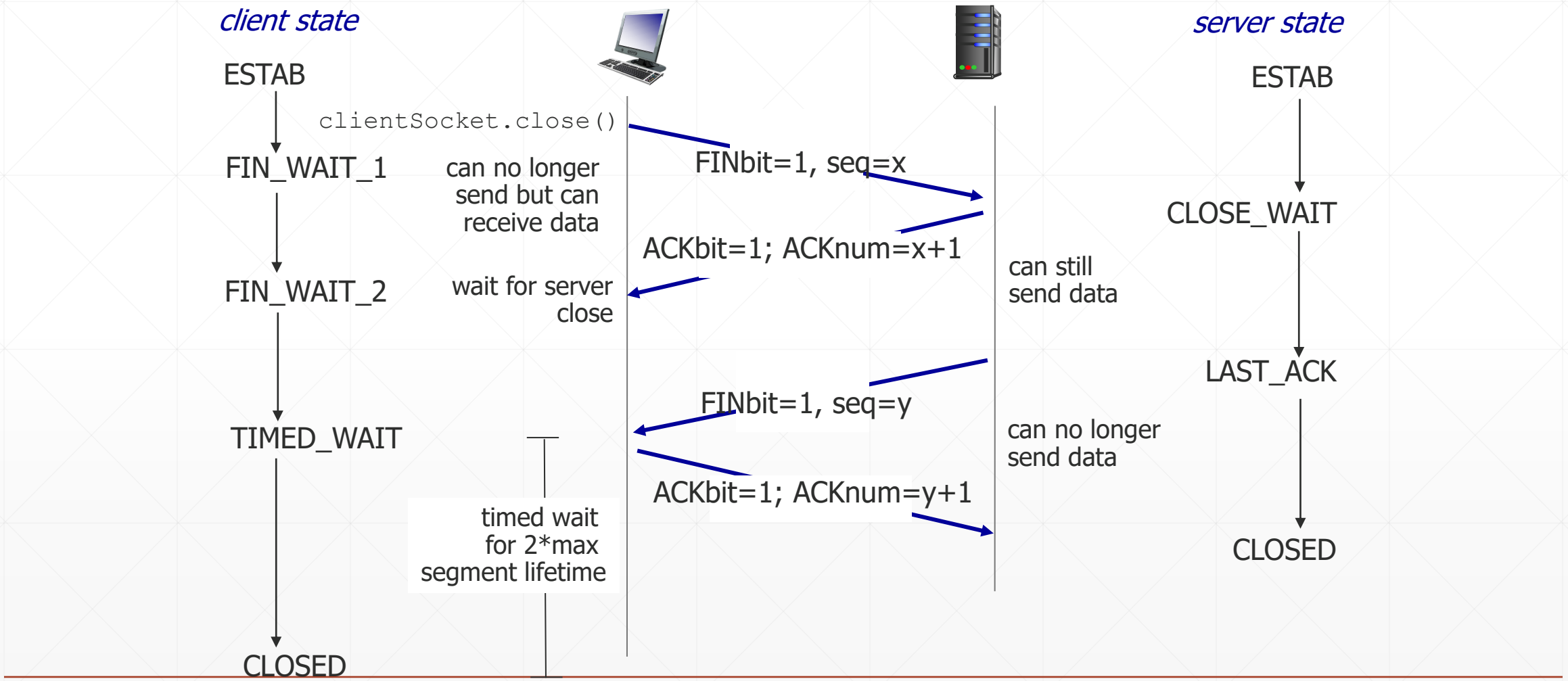
TCP 3-way handshake



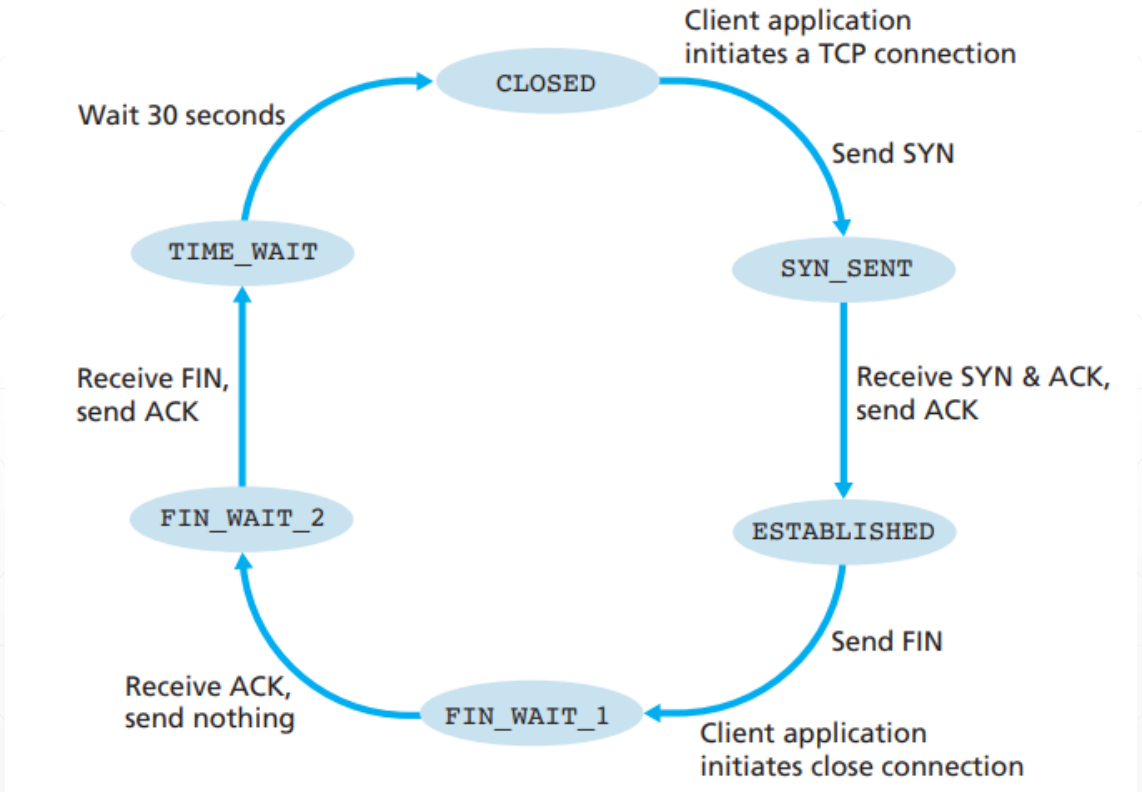
TCP: closing a connection

- client, server each close their side of connection
  - send TCP segment with FIN bit = 1
- respond to received FIN with ACK
  - on receiving FIN, ACK can be combined with own FIN
- simultaneous FIN exchanges can be handled

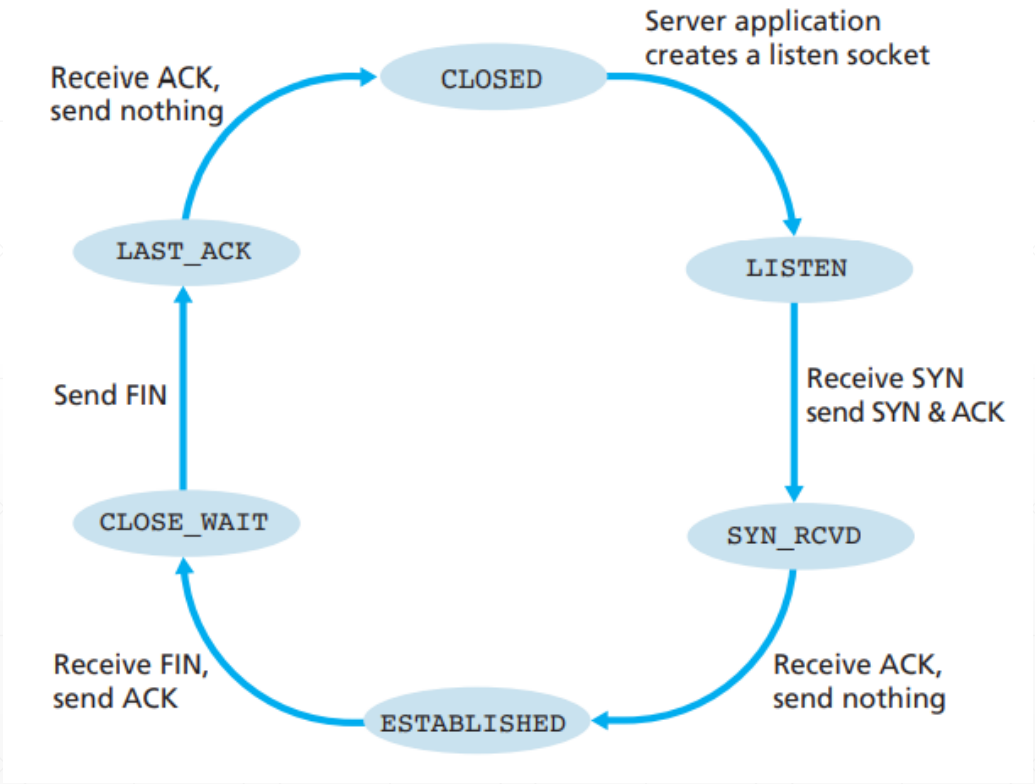
TCP: closing a connection



A typical sequence of TCP states visited by a client TCP



A typical sequence of TCP states visited by a server-side TCP



THANK YOU