CSE 321
Computer Networks
July 2018 Term

Class 17-19: Reliable Data Transfer

Khaled Mahmud Shahriar
Assistant Professor, Dept. of CSE, BUET
khaledshahriar@cse.buet.ac.bd

Chapter 3
Transport Layer

A note on the use of these ppt slides:

We're making these slides freely available to all (faculty, students, readers). They're in PowerPoint form so you see the animations; and can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a *lot* of work on our part. In return for use, we only ask the following:

- ❖ If you use these slides (e.g., in a class) that you mention their source (after all, we'd like people to use our book!)
- ❖ If you post any slides on a www site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

Thanks and enjoy! JFK/KWR
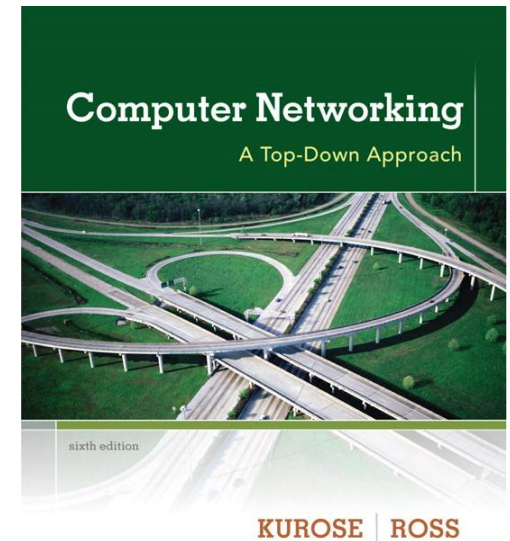
Computer Networking: A Top Down Approach
6th edition
Jim Kurose, Keith Ross
Addison-Wesley
March 2012

Chapter 3 outline

Principles of reliable data transfer

❖ important in application, transport, link layers
  ▪ top-10 list of important networking topics!
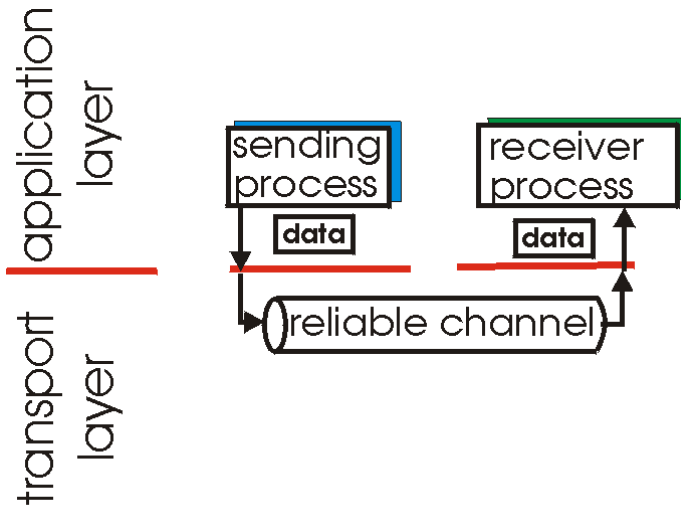
(a) provided service

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

Principles of reliable data transfer

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

(a) provided service

(b) service implementation

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

Principles of reliable data transfer

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

(a) provided service   (b) service implementation
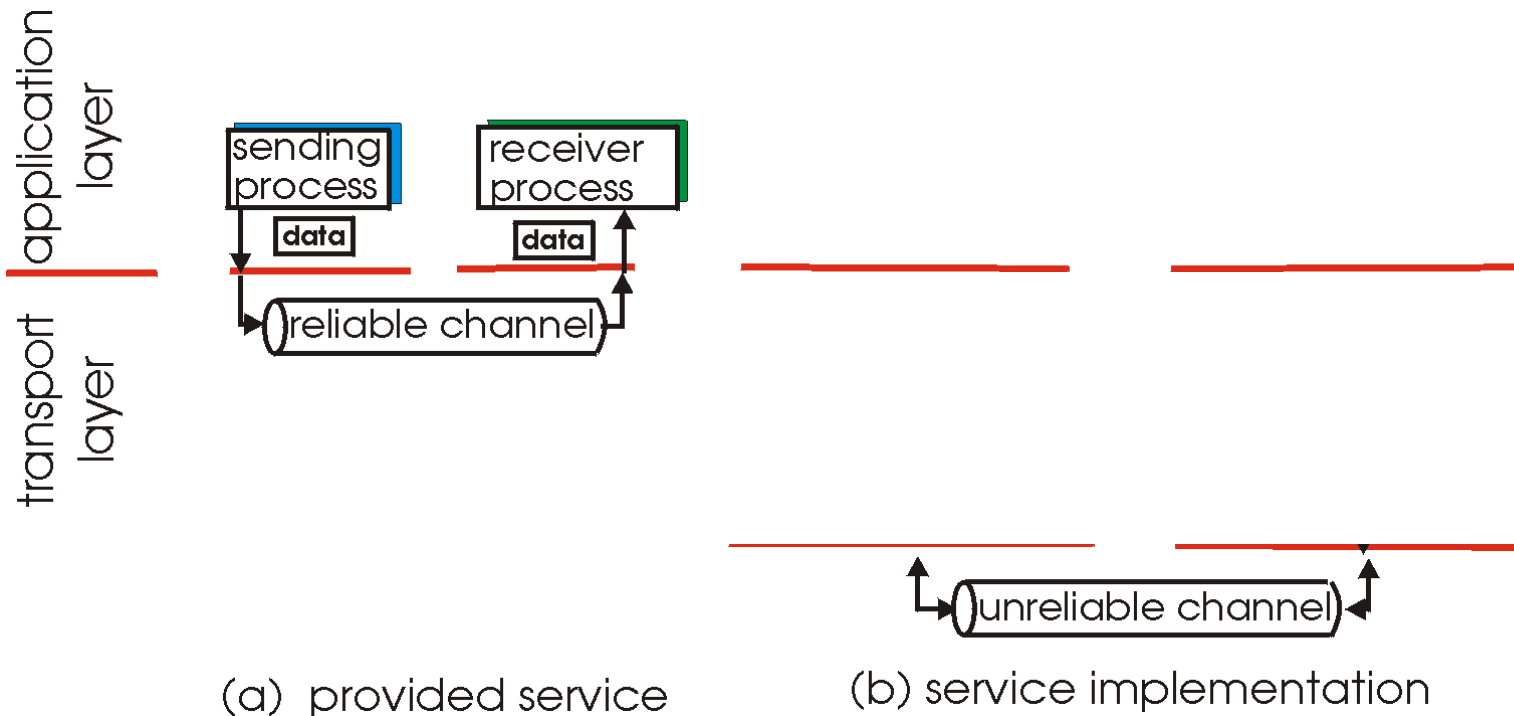
❖ **characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)**

Reliable data transfer: getting started

rdt_send(): called from above, (e.g., by app.). Passed data to deliver to receiver upper layer
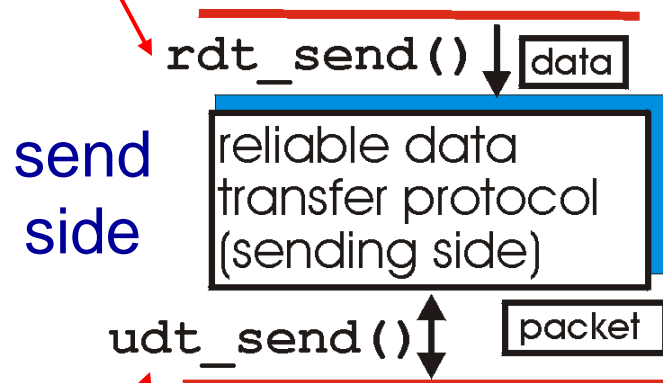
deliver_data(): called by **rdt** to deliver data to upper

rdt_send() ↓ data

data ↑ deliver_data()

send side

reliable data transfer protocol (sending side)

reliable data transfer protocol (receiving side)

receive side

udt_send() ↕ packet

packet ↕ rdt_rcv()

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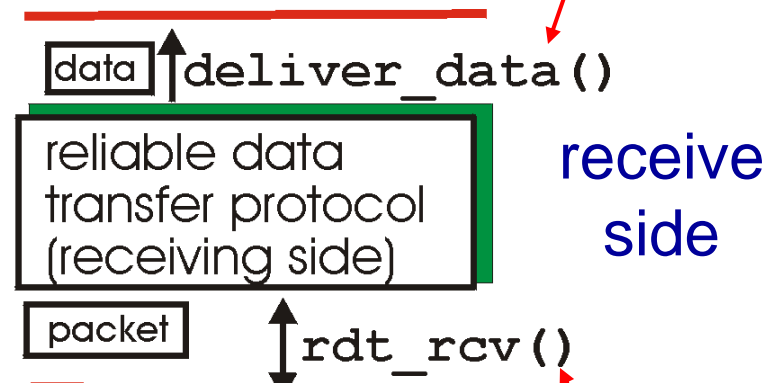
udt_send(): called by rdt, to transfer packet over unreliable channel to receiver

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

Characteristics of Channel

- ❖ Packet may get corrupted due to noise, device error etc. but always reaches the receiver
- ❖ Packets may get lost in transit
- ❖ Packets may be reordered
  - ▪ problem of delayed duplicate

Reliable data transfer: getting started

we'll:

❖ incrementally develop sender, receiver sides of <u>r</u>eliable <u>d</u>ata <u>t</u>ransfer 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

event causing state transition
actions taken on state transition

state 1

event
actions

state 2
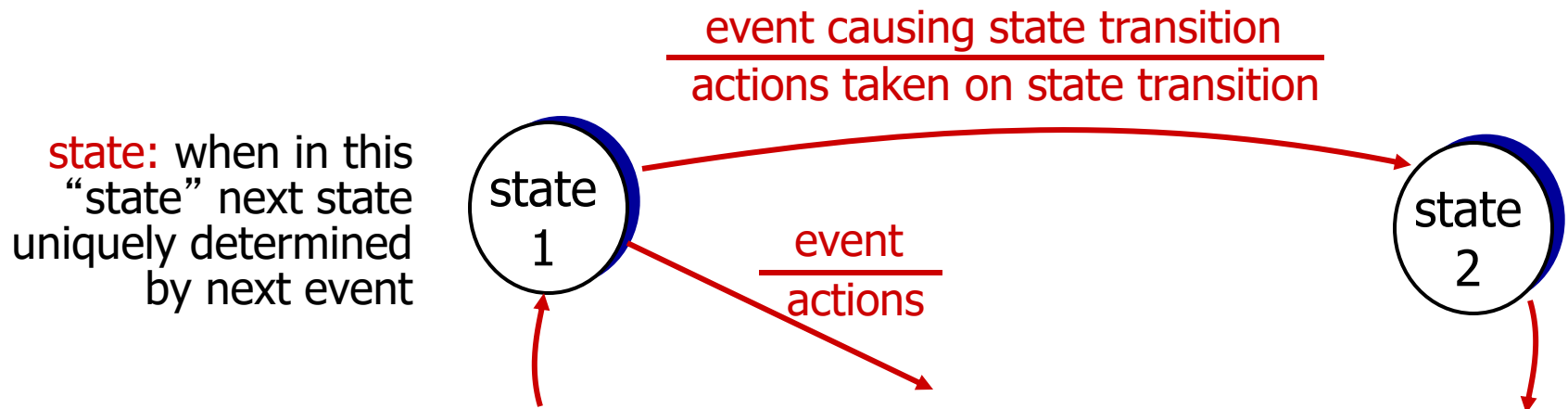
rdt1.0: reliable transfer over a reliable channel

- ❖ **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*

send pkt ——— pkt ——→
send pkt ——— pkt ——→ rcv pkt
send pkt ——— pkt ——→ rcv pkt
send pkt ——— pkt ——→ rcv pkt
send pkt ——— pkt ——→ rcv pkt
send pkt ——— pkt ——→ rcv pkt
send pkt ——— pkt ——→ rcv pkt
         ——— pkt ——→ rcv pkt

Wait for
call from
above

rdt_send(data)
_____

packet = make_pkt(data)
udt_send(packet)

sender

Wait for
call from
below

rdt_rcv(packet)
_____

extract (packet,data)
deliver_data(data)
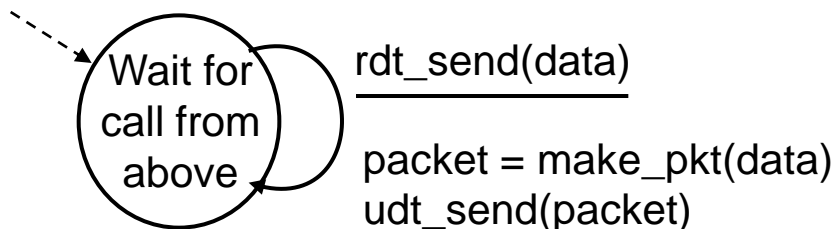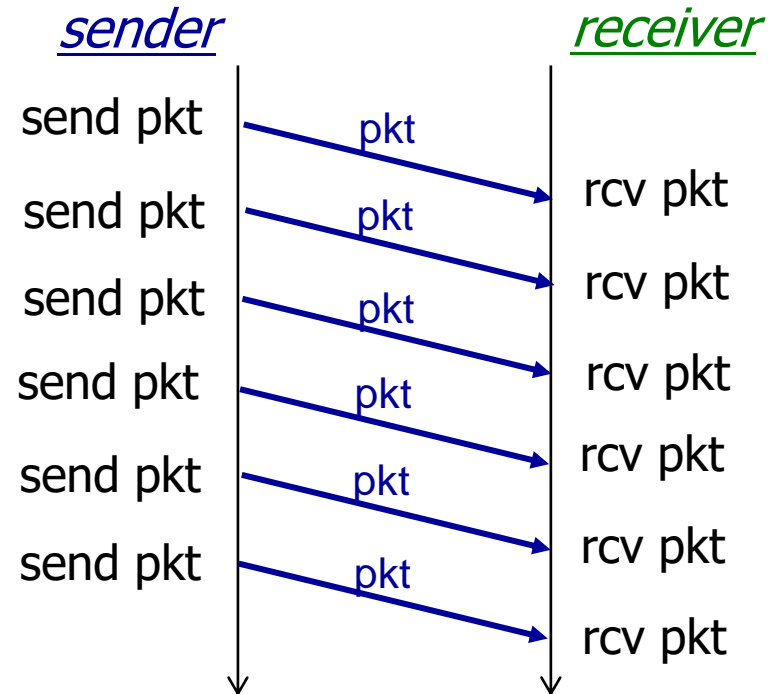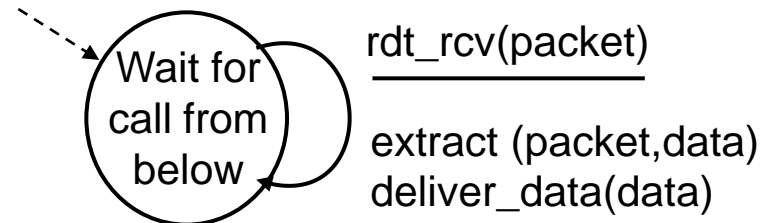
receiver

rdt2.0: channel with bit errors

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

How do humans recover from "errors"
during conversation?

rdt2.0: channel with bit errors

❖ underlying channel may flip bits in packet
  ▪ checksum to detect bit errors

❖ *the* question: how to recover from errors:

  ▪ *acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK

  ▪ *negative acknowledgements (NAKs):* receiver explicitly tells sender that pkt had errors

  ▪ sender retransmits pkt on receipt of NAK

❖ new mechanisms in `rdt2.0` (beyond `rdt1.0`):
  ▪ error detection
  ▪ feedback: control msgs (ACK,NAK) from receiver to sender

rdt2.0 in action space-time diagram

sender   receiver

send pkt
→ pkt →
     rcv pkt
     send ack
← ack ←

rcv ack
send pkt
→ pkt →   **error**
     rcv corrupted packet
     send NAK
← NAK ←

rcv NAK
resend pkt
→ pkt →
     rcv pkt
     send ack
← ack ←

rcv ack
send pkt
→ pkt →
     rcv pkt
     send ack
← ack ←

rcv ack
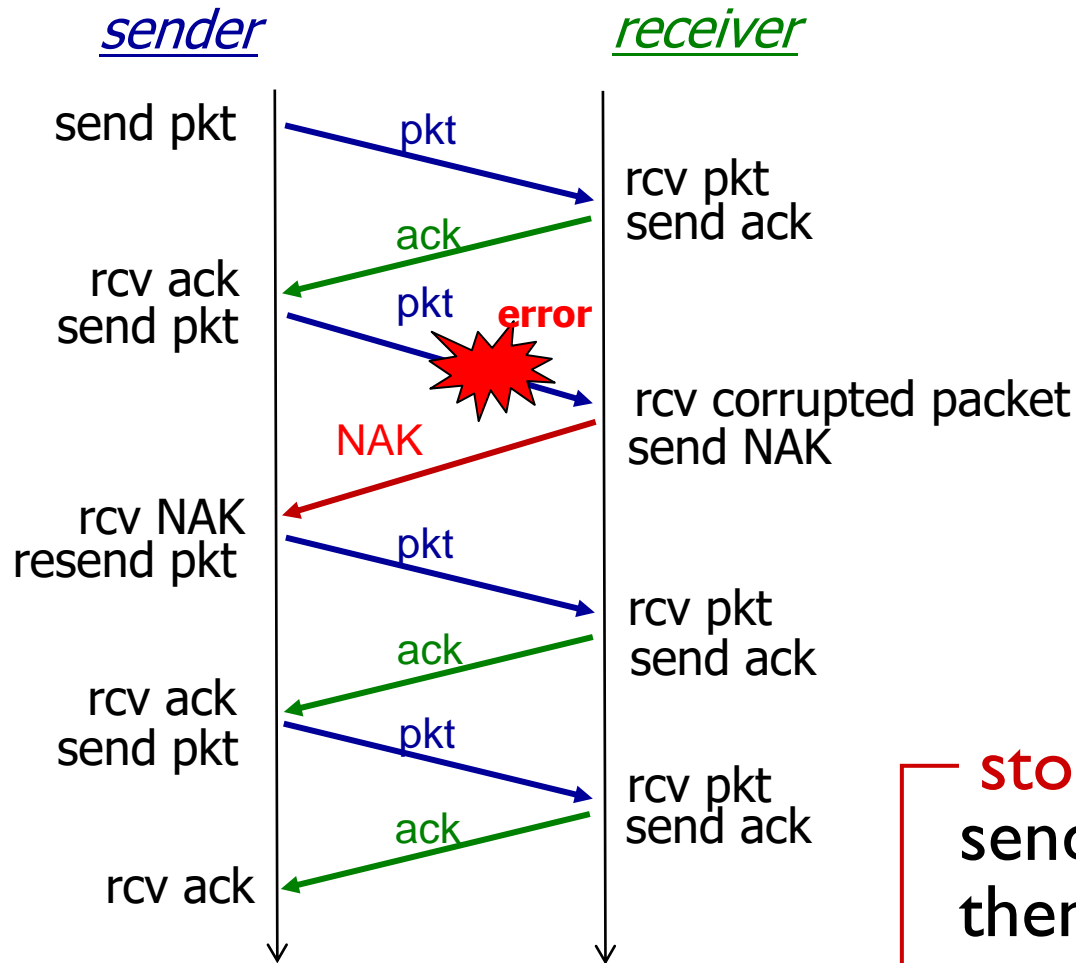
stop and wait
sender sends one packet, then waits for receiver response

rdt2.0: FSM specification

rdt_send(data)

sndpkt = make_pkt(data, checksum)
udt_send(sndpkt)

receiver

rdt_rcv(rcvpkt) &&
isNAK(rcvpkt)

udt_send(sndpkt)

Wait for call from above

Wait for ACK or NAK

rdt_rcv(rcvpkt) && isACK(rcvpkt)

Λ

sender

rdt_rcv(rcvpkt) &&
corrupt(rcvpkt)

udt_send(NAK)

Wait for call from below

rdt_rcv(rcvpkt) &&
notcorrupt(rcvpkt)

extract(rcvpkt,data)
deliver_data(data)
udt_send(ACK)
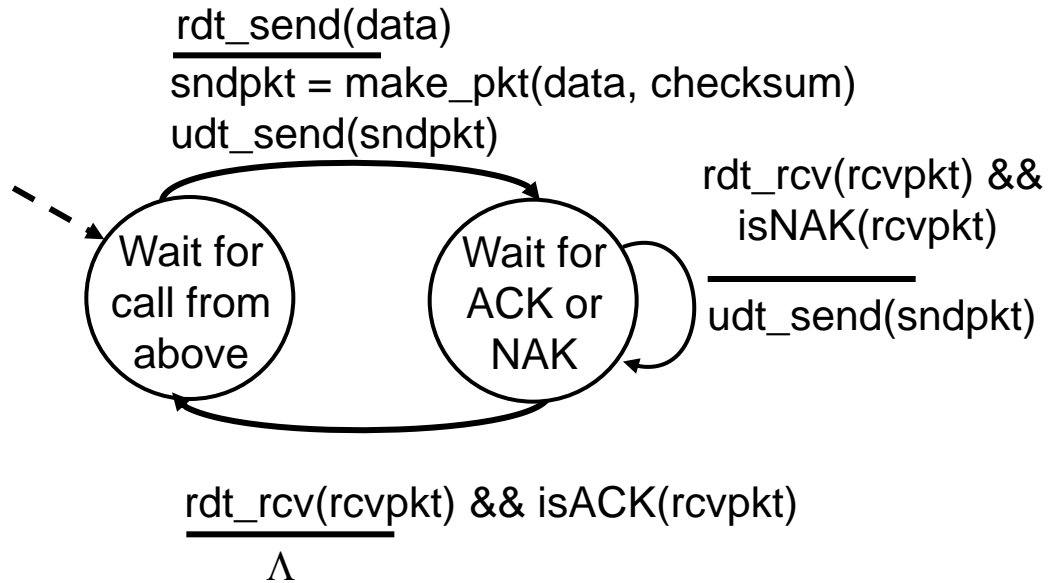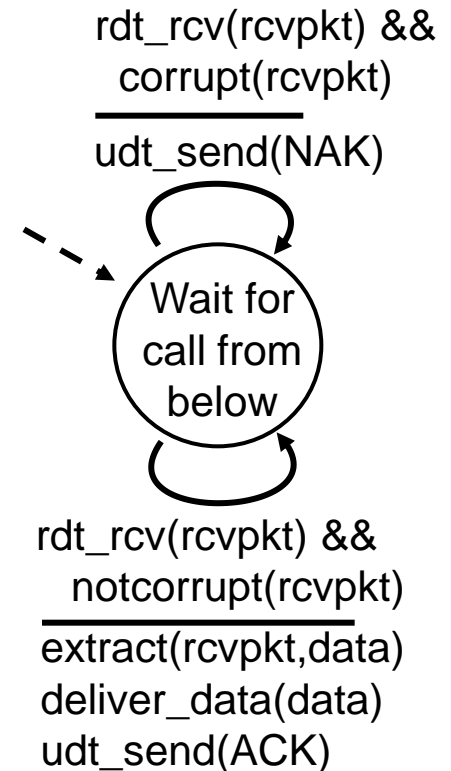
rdt2.0: operation with no errors

rdt_send(data)
―――――――
snkpkt = make_pkt(data, checksum)
udt_send(sndpkt)

Wait for call from above

Wait for ACK or NAK

rdt_rcv(rcvpkt) &&
isNAK(rcvpkt)
――――――――
udt_send(sndpkt)

rdt_rcv(rcvpkt) && isACK(rcvpkt)
――――――――――――――――
Λ

rdt_rcv(rcvpkt) &&
corrupt(rcvpkt)
――――――――
udt_send(NAK)

Wait for call from below

rdt_rcv(rcvpkt) &&
notcorrupt(rcvpkt)
――――――――――
extract(rcvpkt,data)
deliver_data(data)
udt_send(ACK)
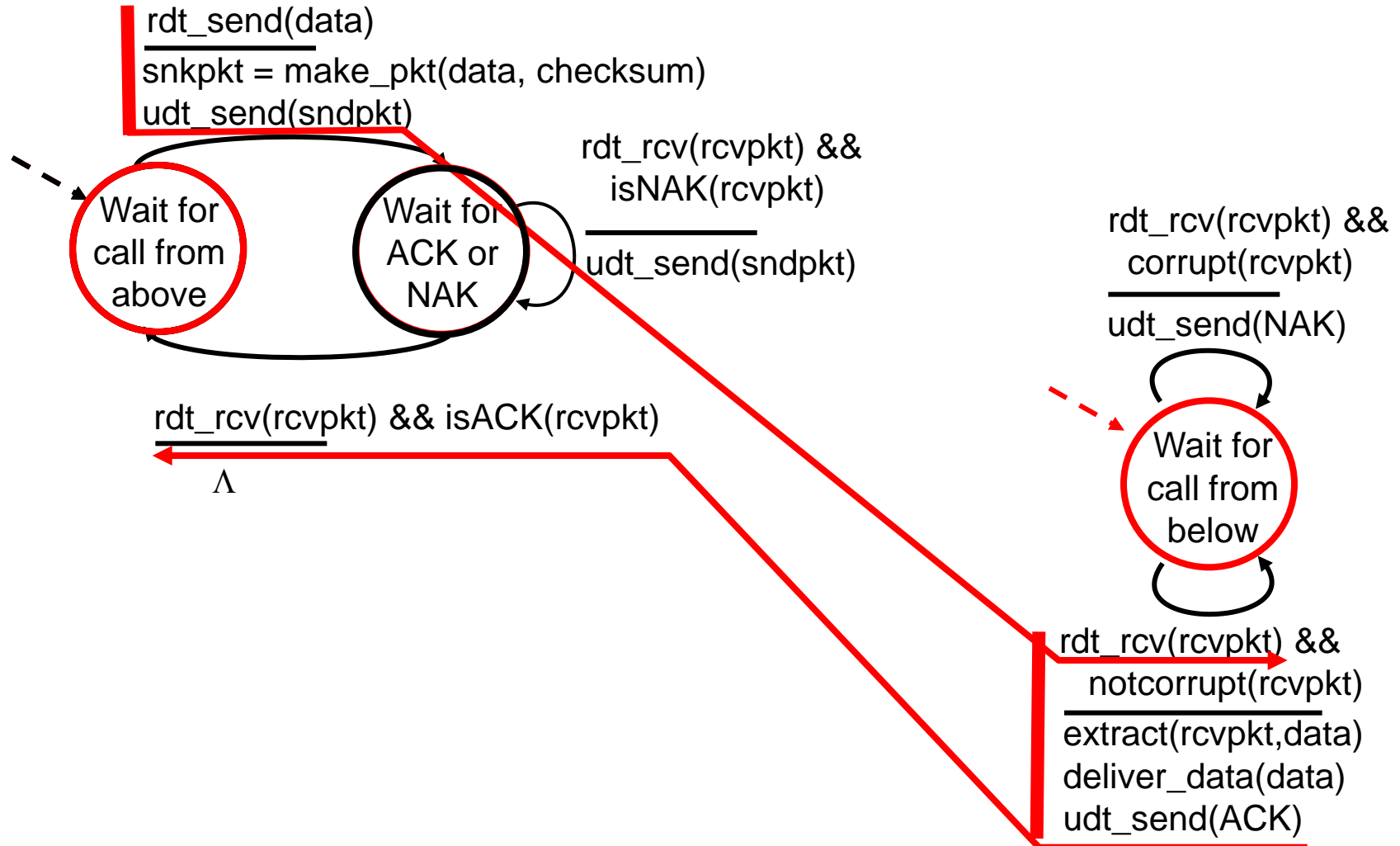
rdt2.0: error scenario

rdt_send(data)
_____
snkpkt = make_pkt(data, checksum)
udt_send(sndpkt)

Wait for
call from
above

Wait for
ACK or
NAK

rdt_rcv(rcvpkt) &&
isNAK(rcvpkt)
_____
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
corrupt(rcvpkt)
_____
udt_send(NAK)

rdt_rcv(rcvpkt) && isACK(rcvpkt)
_____
Λ

Wait for
call from
below

rdt_rcv(rcvpkt) &&
notcorrupt(rcvpkt)
_____
extract(rcvpkt,data)
deliver_data(data)
udt_send(ACK)

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

sender   receiver

send pkt → pkt → rcv pkt / send ack

rcv ack / send pkt ← ack

pkt **error** → rcv corrupted packet / send NAK

rcv corrupted pkt what was it ? ACK or NAK!! ← NAK
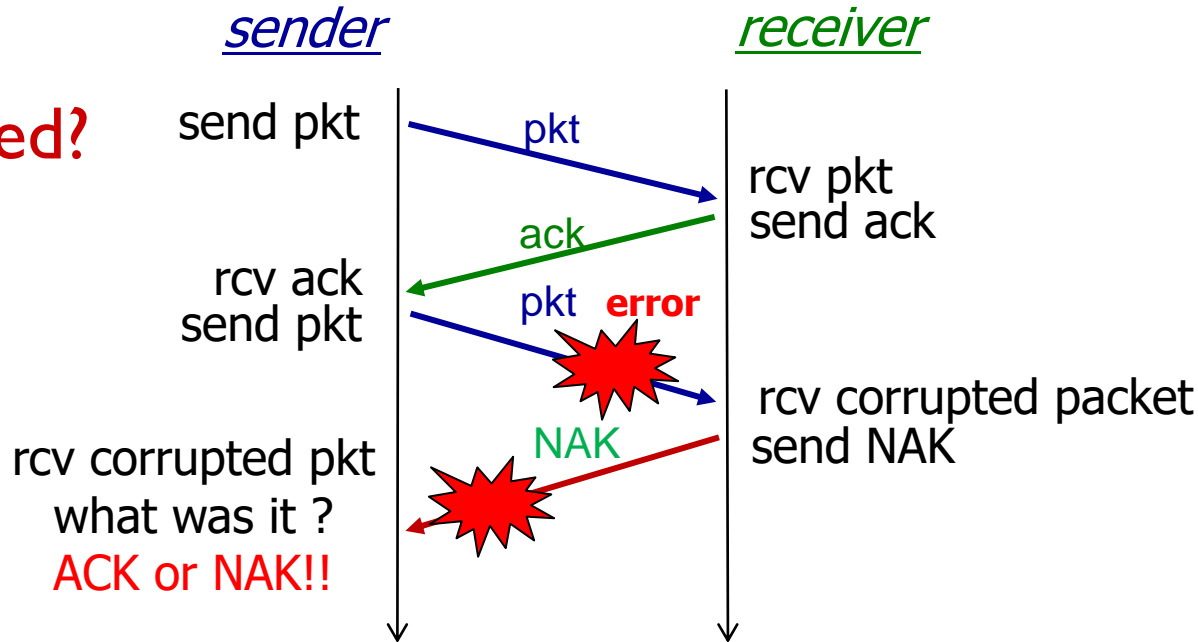
handling duplicates:

❖ sender retransmits current pkt if ACK/NAK corrupted

❖ sender adds *sequence number* to each pkt

❖ receiver discards (doesn't deliver up) duplicate pkt

rdt2.1: Frames with Seq No.

sender   receiver   *seq expected*

send pkt — pkt0 →

rcv pkt
send ack | 0 |

← ack —

rcv ack
send pkt — pkt1 →

rcv packet
send ack | 1 |

rcv corrupted pkt
resend packet   ← ack — **error**

— pkt1 →

packet already rcvd;
discard
send ack | 0 |

rcv nak
send pkt   ← ack —

— pkt0 →

rcv packet
send ack | 0 |

rcv ack   ← ack0 —
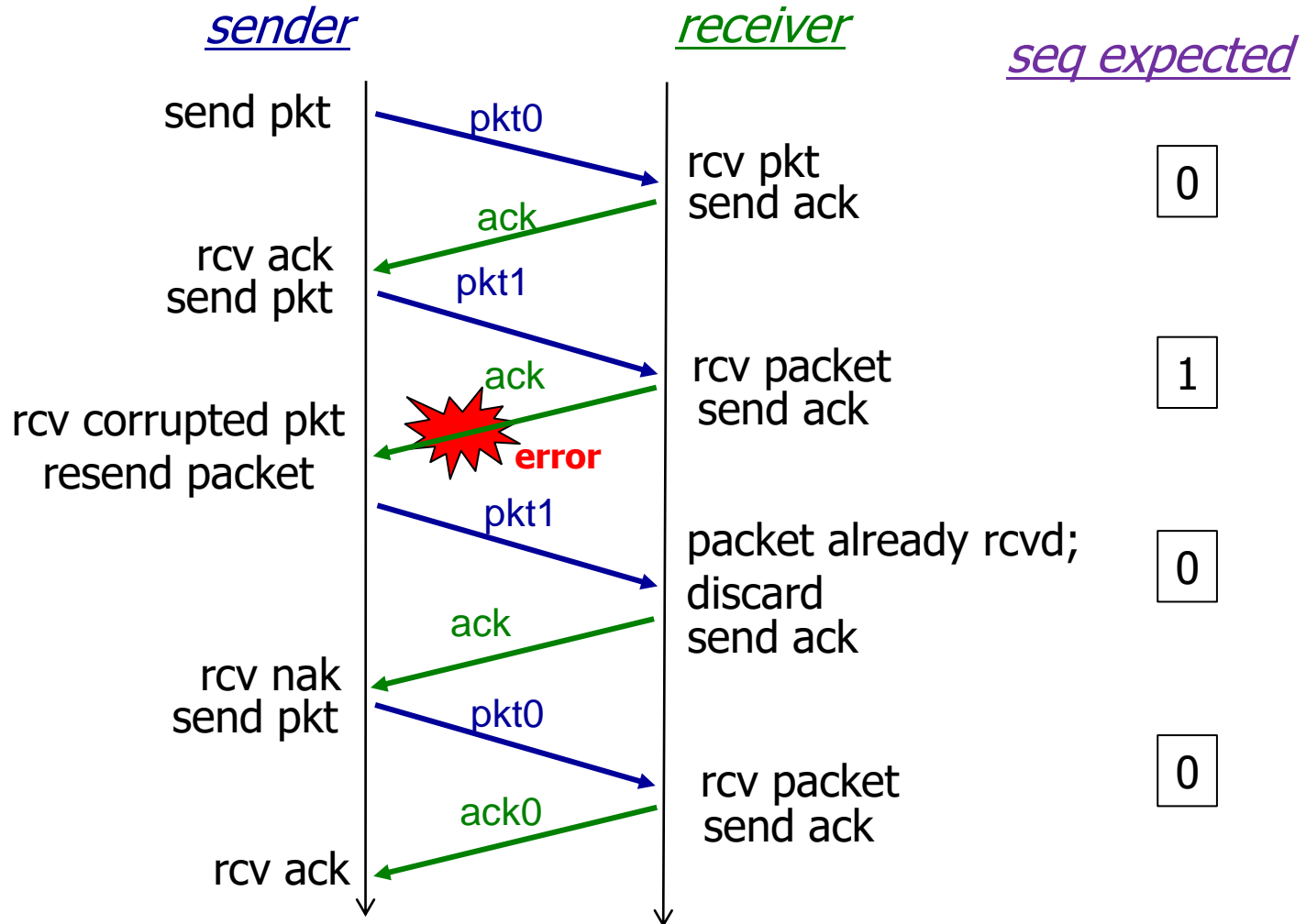
rdt2.1: sender, handles garbled ACK/NAKs

rdt_send(data)
_____

sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isNAK(rcvpkt))
_____
udt_send(sndpkt)

Wait for call 0 from above

Wait for ACK or NAK 0

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt)
_____
Λ

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt)
_____
Λ

Wait for ACK or NAK 1

Wait for call 1 from above

rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isNAK(rcvpkt))
_____
udt_send(sndpkt)

rdt_send(data)
_____

sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)

rdt2.1: receiver, handles garbled ACK/NAKs

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
 && has_seq0(rcvpkt)
──────────────────────────────
extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && (corrupt(rcvpkt)
──────────────────────────────
sndpkt = make_pkt(NAK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
 not corrupt(rcvpkt) &&
 has_seq1(rcvpkt)
──────────────────────────────
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && (corrupt(rcvpkt)
──────────────────────────────
sndpkt = make_pkt(NAK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
 not corrupt(rcvpkt) &&
 has_seq0(rcvpkt)
──────────────────────────────
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

Wait for 0 from below

Wait for 1 from below

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
 && has_seq1(rcvpkt)
──────────────────────────────
extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

rdt2.1: discussion

sender:

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

receiver:

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

rdt2.2: a NAK-free protocol

❖ same functionality as rdt2.1, using ACKs only

❖ instead of NAK, receiver sends ACK for last pkt received OK

  ▪ receiver must *explicitly* include seq # of pkt being ACKed

❖ duplicate ACK at sender results in same action as NAK: *retransmit current pkt*

rdt2.1: No NAK

ack expected

seq expected

send pkt
pkt0
rcv pkt
send ack
0

ack0

0   rcv ack
send pkt
pkt1   **error**

rcv corrupted packet
send ack (no nak)
1

ack0

1   rcv ack; not expected
resend packet

pkt1

~~packet already rcvd;~~
~~discard~~
send ack
0

ack1

1   rcv nak
send pkt
pkt0

rcv packet
send ack
0

ack0

0   rcv ack

Transport Layer 3-23
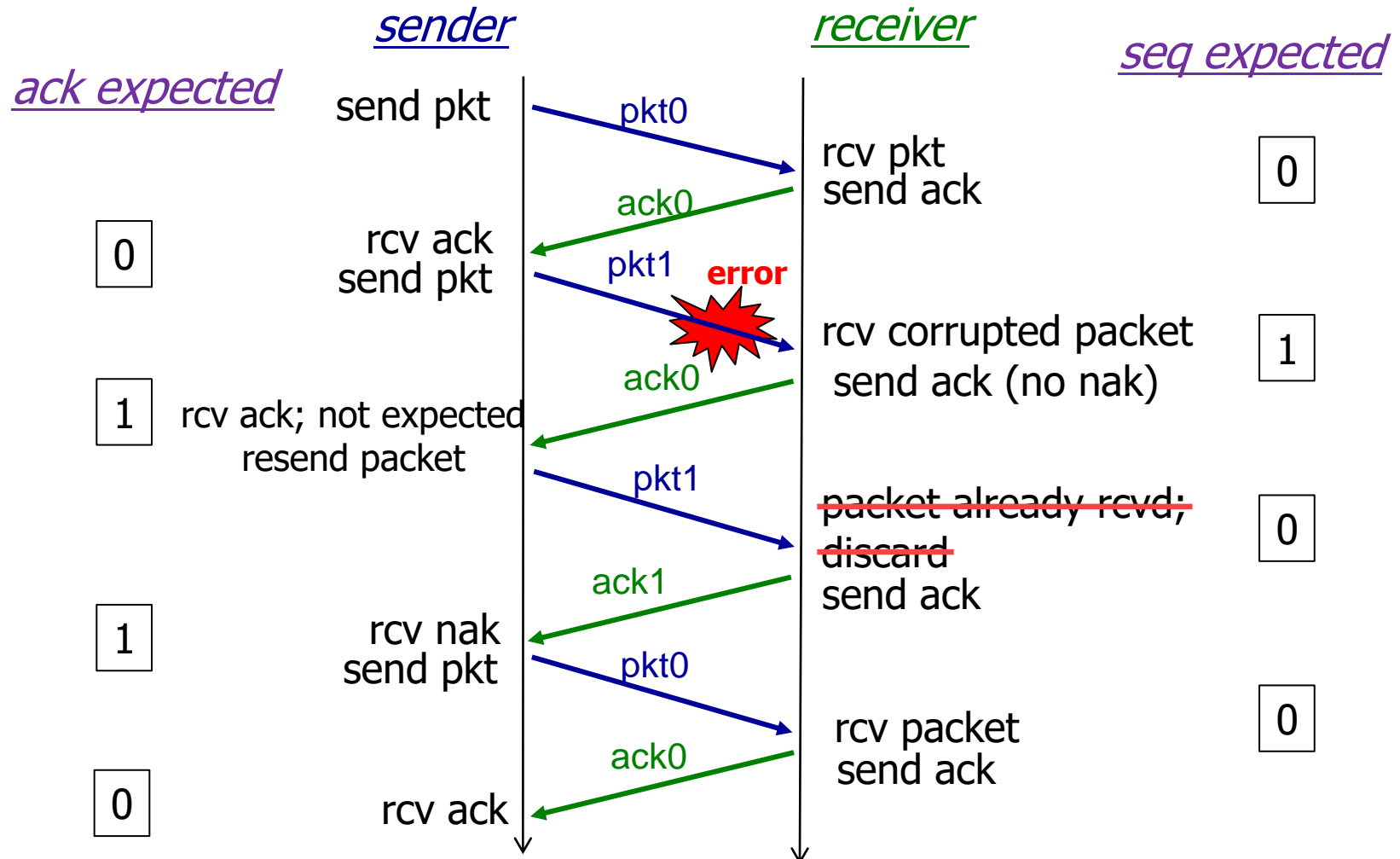
rdt2.2: sender, receiver fragments

rdt_send(data)
—————————————————
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)

Wait for call 0 from above

Wait for ACK 0

rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isACK(rcvpkt,1))
udt_send(sndpkt)

sender FSM
fragment

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& **isACK(rcvpkt,0)**
—————————————————
Λ

rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
has_seq1(rcvpkt))
—————————————————
udt_send(sndpkt)

Wait for 0 from below

receiver FSM
fragment

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq1(rcvpkt)
—————————————————
extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK1, chksum)
udt_send(sndpkt)

rdt3.0: channels with errors *and* loss

new assumption: underlying channel can also lose packets (data, ACKs)
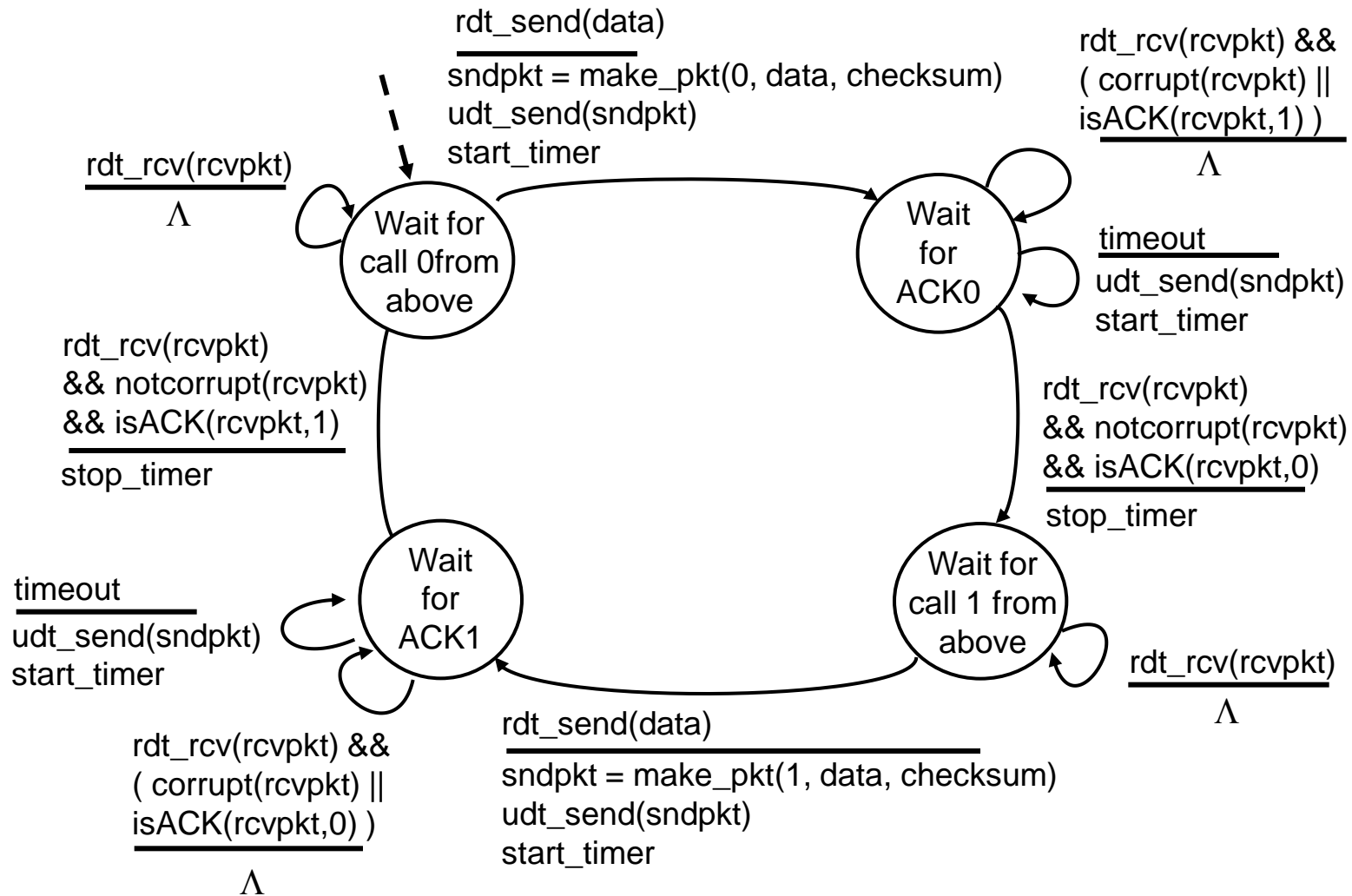
- checksum, seq. #, ACKs, retransmissions will be of help … but not enough

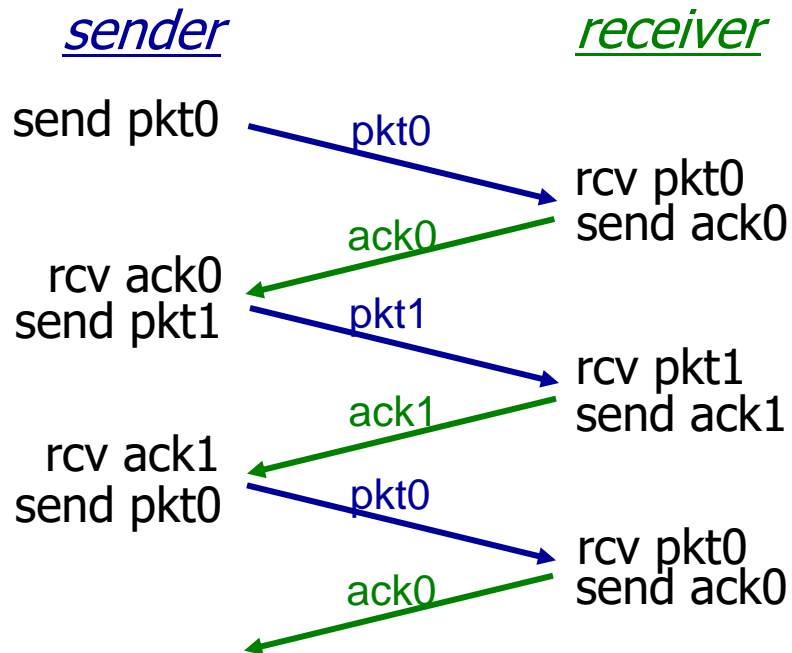**approach:** sender waits "reasonable" amount of time for ACK

- ❖ 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

Also known as alternating-bit protocol

rdt3.0 sender

rdt_send(data)
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
start_timer

rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isACK(rcvpkt,1))
Λ

rdt_rcv(rcvpkt)
Λ

Wait for call 0from above

Wait for ACK0

timeout
udt_send(sndpkt)
start_timer

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt,1)
stop_timer

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt,0)
stop_timer

timeout
udt_send(sndpkt)
start_timer

Wait for ACK1

Wait for call 1 from above

rdt_rcv(rcvpkt)
Λ

rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isACK(rcvpkt,0))
Λ

rdt_send(data)
sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)
start_timer

Transport Layer 3-26

rdt3.0 in action

sender   **receiver**

send pkt0 → pkt0
      rcv pkt0
      send ack0
rcv ack0 ← ack0
send pkt1 → pkt1
      rcv pkt1
      send ack1
rcv ack1 ← ack1
send pkt0 → pkt0
      rcv pkt0
      send ack0
← ack0

(a) no loss

sender   **receiver**

send pkt0 → pkt0
      rcv pkt0
      send ack0
rcv ack0 ← ack0
send pkt1 → pkt1
      X
      loss

timeout
resend pkt1 → pkt1
      rcv pkt1
      send ack1
rcv ack1 ← ack1
send pkt0 → pkt0
      rcv pkt0
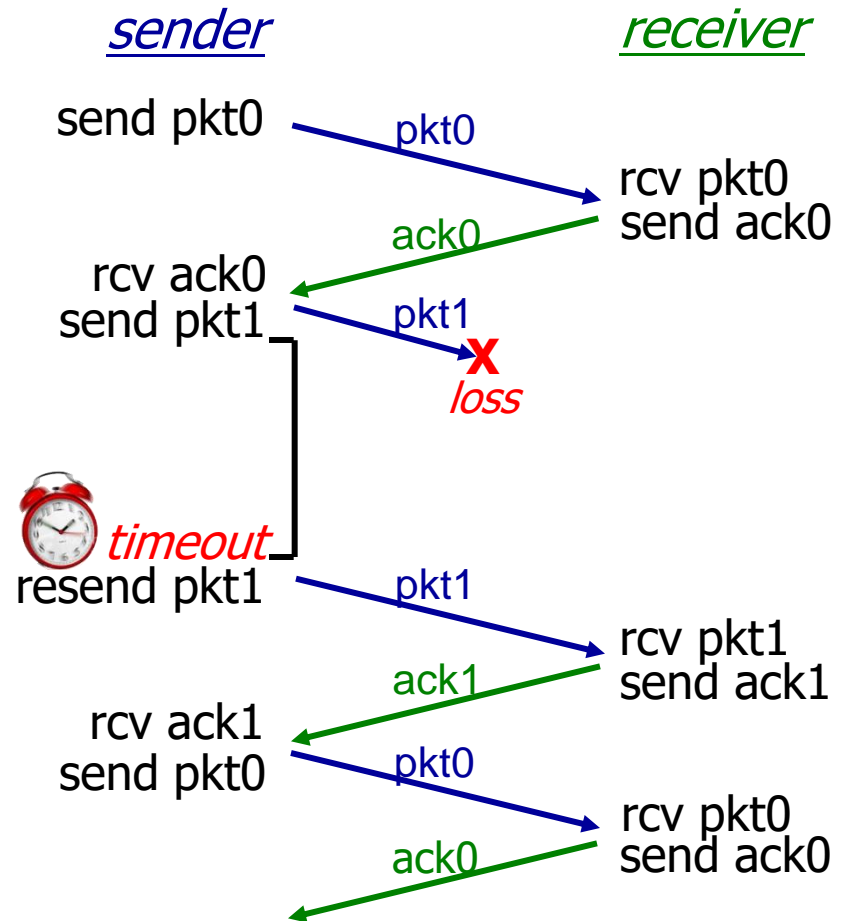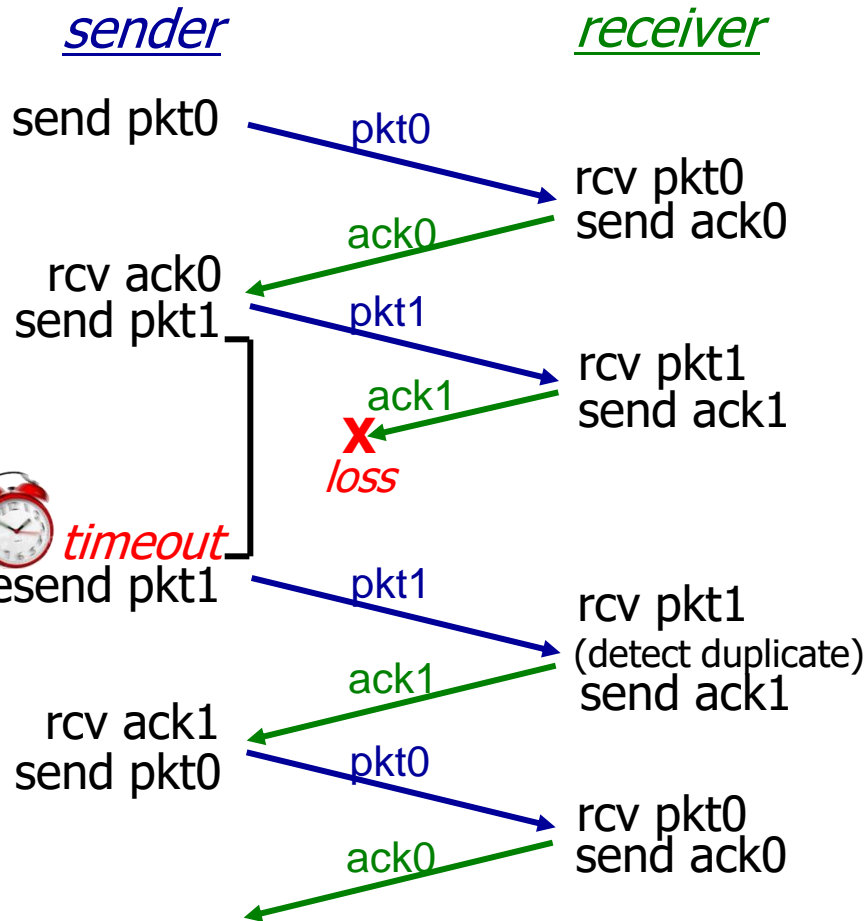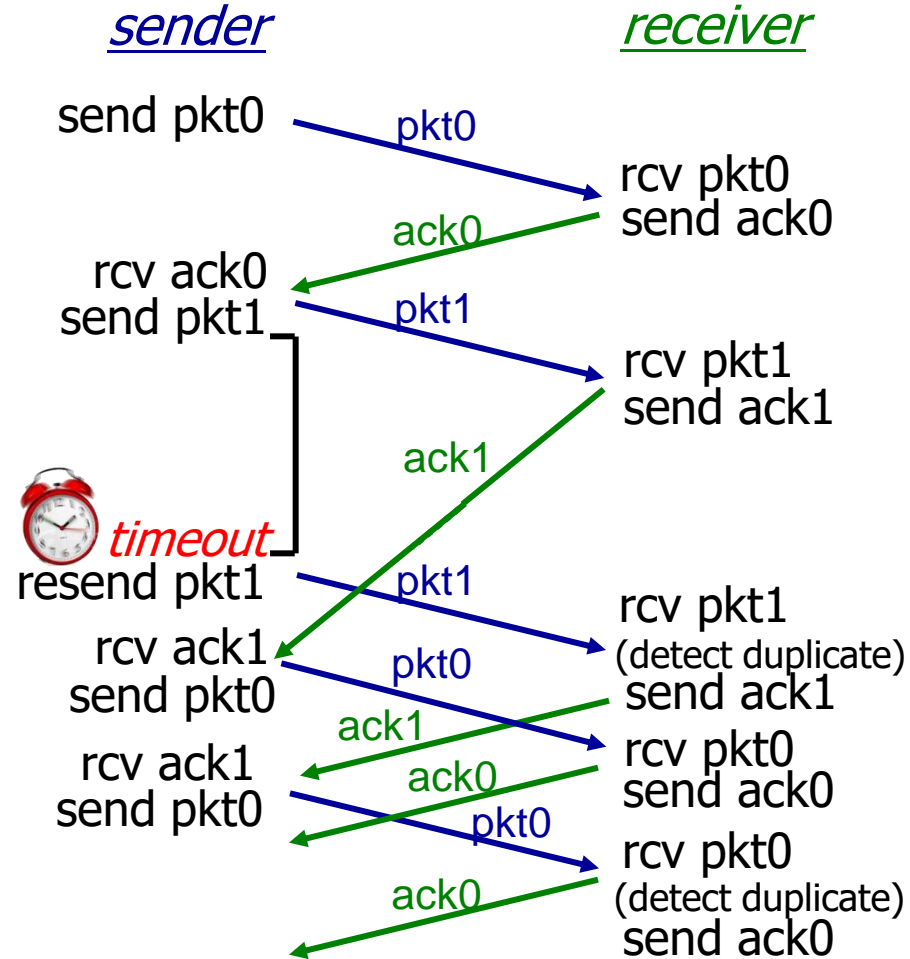   send ack0
← ack0

(b) packet loss

rdt3.0 in action

(c) ACK loss

(d) premature timeout/ delayed ACK

Determining the Value of Timeout

❖ Must be as long as the round-trip time (RTT) between the sender and receiver
❖ Easy for point to point link
❖ Difficult to compute end to end; need to consider
  ▪ buffering at intermediate routers
  ▪ time needed to process packet at each hop
❖ Waiting for worst case maximum delay would degrade performance
  ▪ Determined dynamically as in TCP

Performance of rdt3.0

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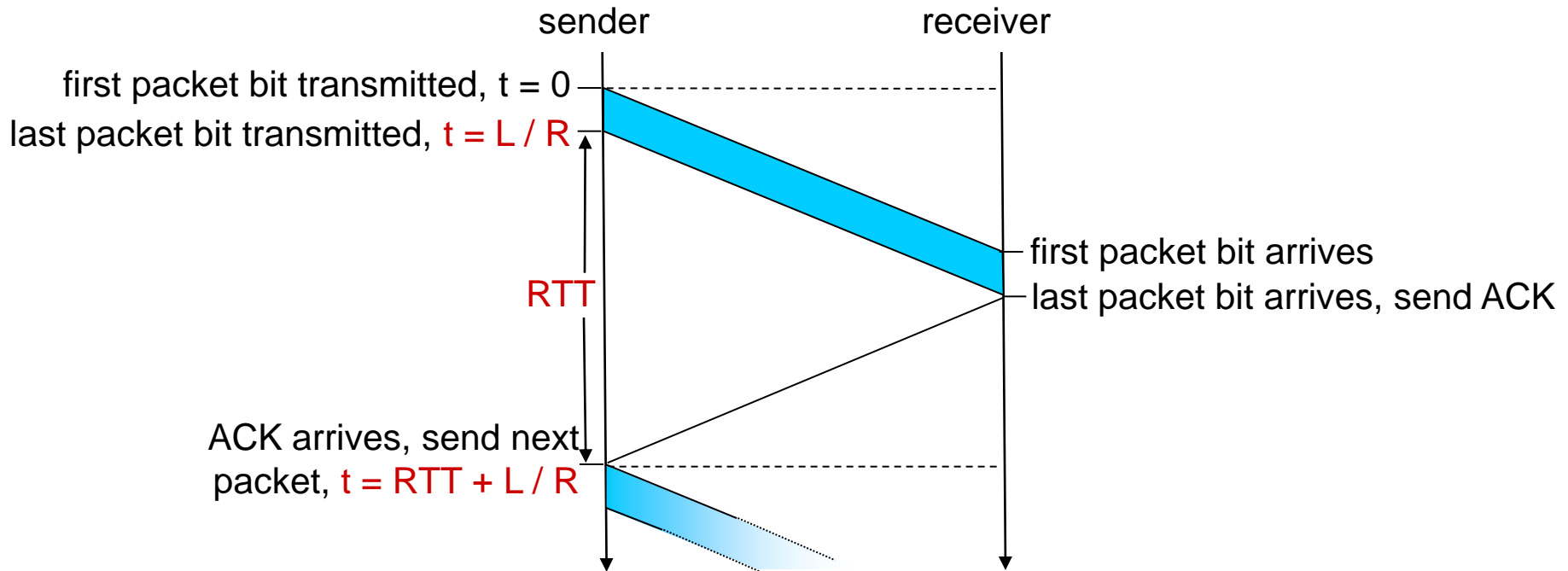
❖ e.g.: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:

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

▪ U $_{sender}$: *utilization* – fraction of time sender busy sending

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

▪ if RTT=30 msec, 1KB pkt every 30 msec: 33kB/sec thruput over 1 Gbps link

❖ network protocol limits use of physical resources!

rdt3.0: stop-and-wait operation

sender   receiver

first packet bit transmitted, t = 0

last packet bit transmitted, t = L / R

first packet bit arrives

RTT

last packet bit arrives, send ACK

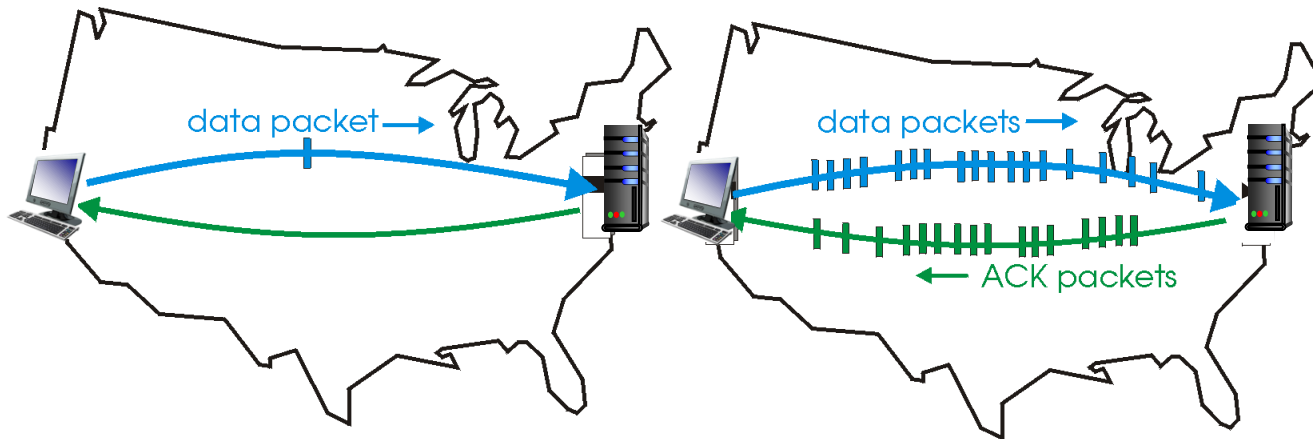ACK arrives, send next
packet, t = RTT + L / R

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

Pipelined protocols

pipelining: sender allows multiple, "in-flight", yet-to-be-acknowledged pkts
- range of sequence numbers must be increased
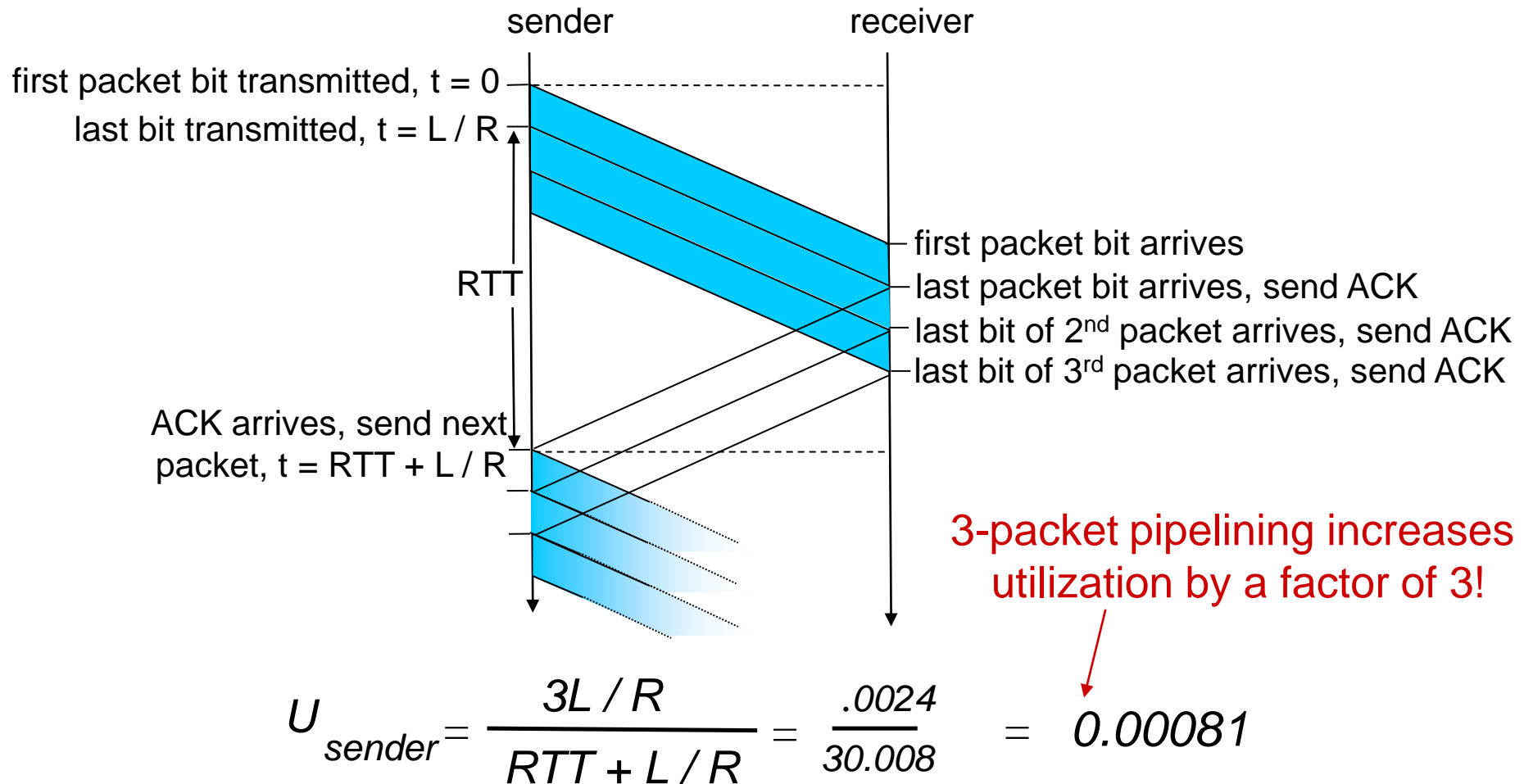- buffering at sender and/or receiver

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

Pipelining: increased utilization

sender   receiver

first packet bit transmitted, t = 0

last bit transmitted, t = L / R

RTT

first packet bit arrives

last packet bit arrives, send ACK

last bit of 2nd packet arrives, send ACK

last bit of 3rd packet arrives, send ACK

ACK arrives, send next
packet, t = RTT + L / R

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

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

❖ two generic forms of pipelined protocols: *go-Back-N, selective repeat*

Pipelined protocols: 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
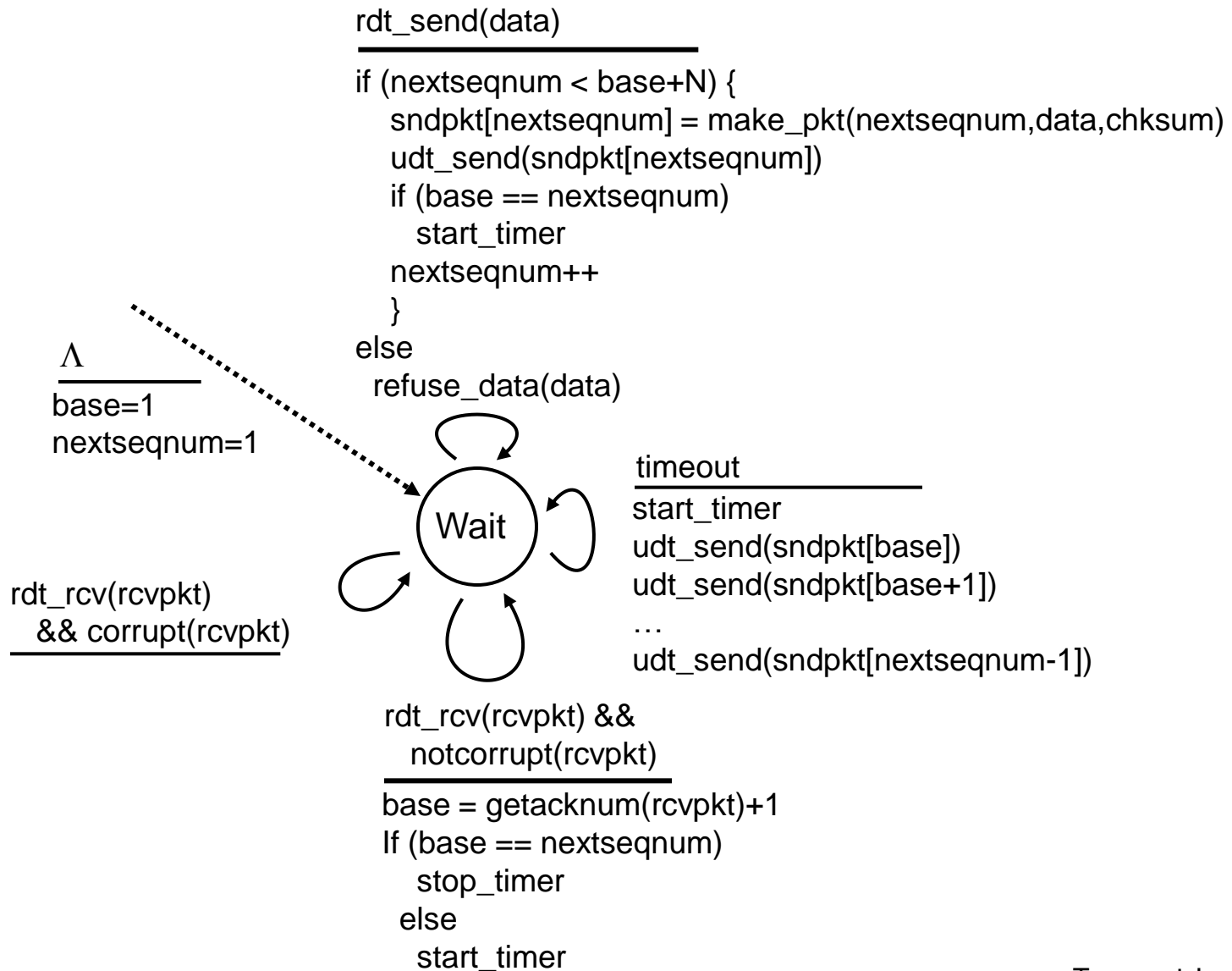 - when timer expires, retransmit *all* unacked packets

Go-Back-N: sender

❖ k-bit seq # in pkt header
❖ "window" of up to N, consecutive unack'ed pkts allowed
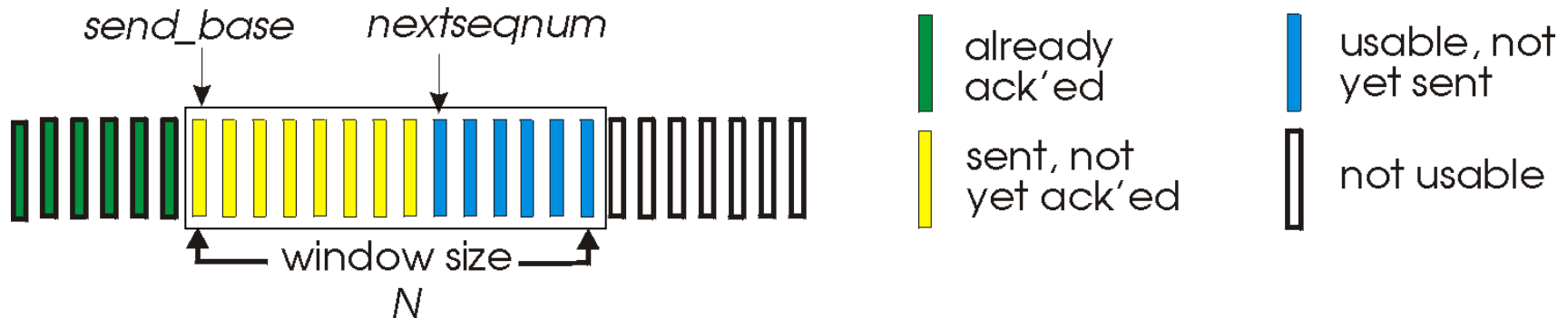
❖ ACK(n): ACKs all pkts up to, including seq # n - *"cumulative ACK"*
  ▪ may receive duplicate ACKs (see receiver)
❖ timer for oldest in-flight pkt
❖ *timeout(n):* retransmit packet n and all higher seq # pkts in window

GBN: sender extended FSM

rdt_send(data)
————————————

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

Λ
————
base=1
nextseqnum=1

rdt_rcv(rcvpkt)
  && corrupt(rcvpkt)
————————————

Wait

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

rdt_rcv(rcvpkt) &&
  notcorrupt(rcvpkt)
————————————
base = getacknum(rcvpkt)+1
If (base == nextseqnum)
    stop_timer
  else
    start_timer

GBN: receiver extended FSM

default
_____
udt_send(sndpkt)

rdt_rcv(rcvpkt)
&& notcurrupt(rcvpkt)
&& hasseqnum(rcvpkt,expectedseqnum)
_____

Λ
_____
expectedseqnum=1
sndpkt =
make_pkt(expectedseqnum,ACK,chksum)

Wait

extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(expectedseqnum,ACK,chksum)
udt_send(sndpkt)
expectedseqnum++
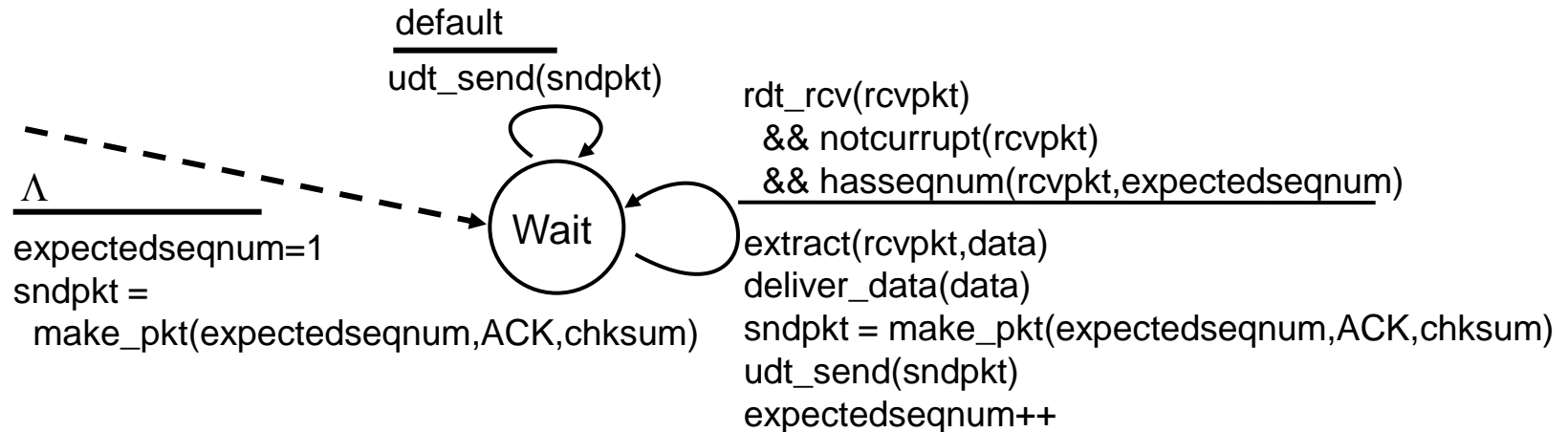
ACK-only: always send ACK for correctly-received pkt with highest *in-order* seq #

  ▪ 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

sender window (N=4)   sender   receiver

`0 1 2 3` 4 5 6 7 8   send pkt0
`0 1 2 3` 4 5 6 7 8   send pkt1
`0 1 2 3` 4 5 6 7 8   send pkt2 ┐   receive pkt0, send ack0
`0 1 2 3` 4 5 6 7 8   send pkt3 │   **X** *loss*   receive pkt1, send ack1
                                    (wait)   │
                                               │   receive pkt3, discard,
0 `1 2 3 4` 5 6 7 8   rcv ack0, send pkt4   │   (re)send ack1
0 1 `2 3 4 5` 6 7 8   rcv ack1, send pkt5   │
                                               │   receive pkt4, discard,
                                               │   (re)send ack1
            ignore duplicate ACK   │   receive pkt5, discard,
                                               │   (re)send ack1
                        pkt 2 timeout ┘

0 1 `2 3 4 5` 6 7 8   send pkt2
0 1 `2 3 4 5` 6 7 8   send pkt3
0 1 `2 3 4 5` 6 7 8   send pkt4   rcv pkt2, deliver, send ack2
0 1 `2 3 4 5` 6 7 8   send pkt5   rcv pkt3, deliver, send ack3
                                                               rcv pkt4, deliver, send ack4
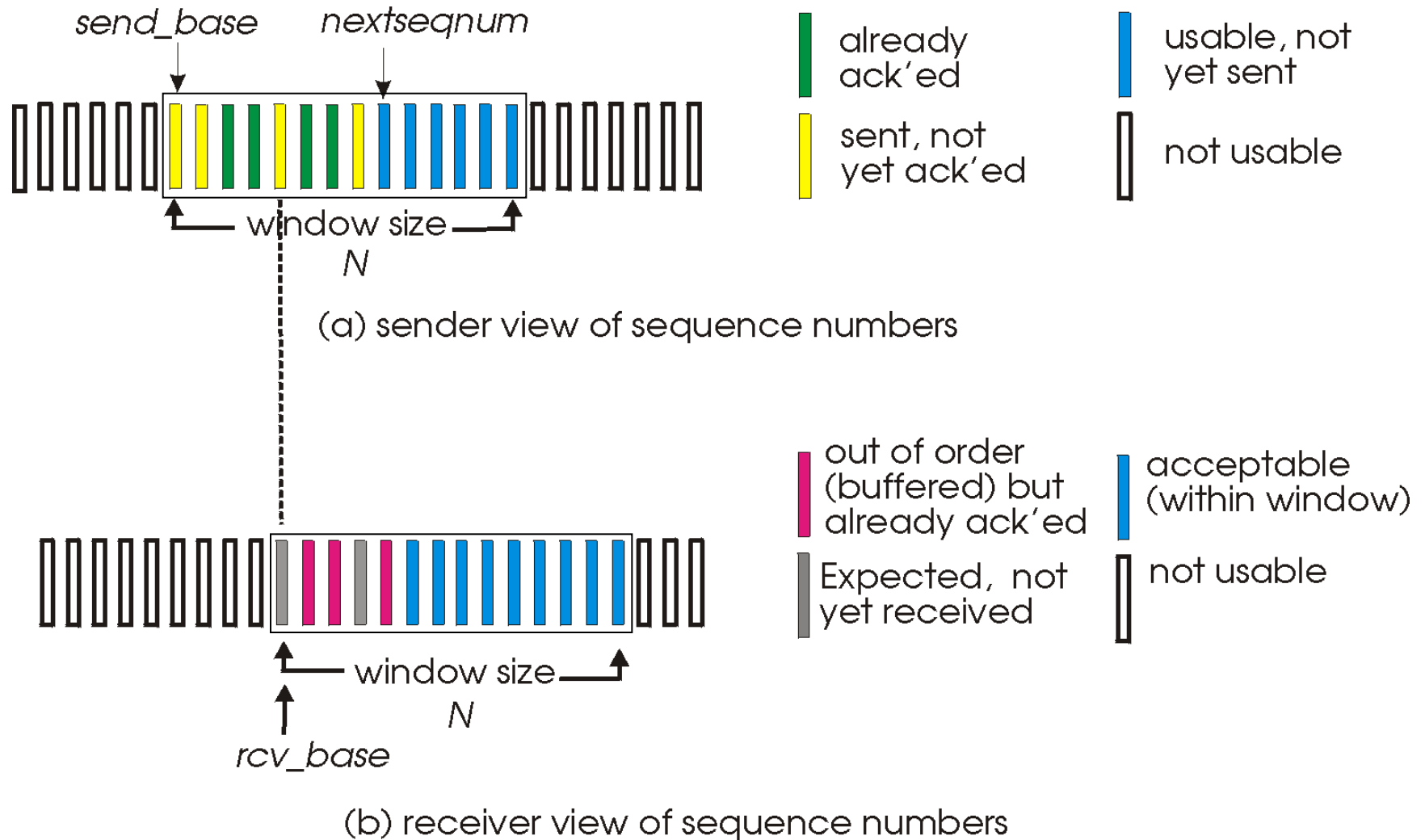   rcv pkt5, deliver, send ack5

Problems of GBN

❖ For a large window size, a single packet error causes retransmission of a large number of packets

❖ The pipeline becomes filled with unnecessary retransmissions

❖ Can be solved by
  ▪ accepting out of order packets by the receiver
  ▪ selective retransmission
    • Selective Repeat (SR) protocol

Pipelined protocols: Selective repeat

❖ sender can have up to N unack'ed packets in pipeline

❖ 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

(a) sender view of sequence numbers

send_base   nextseqnum

	already ack'ed
	usable, not yet sent
	sent, not yet ack'ed
	not usable

window size N

(b) receiver view of sequence numbers

rcv_base

window size N

	out of order (buffered) but already ack'ed
	acceptable (within window)
	Expected, not yet received
	not usable

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

sender window (N=4)　　　sender　　　　　　　receiver

0 1 2 3 4 5 6 7 8　　　send pkt0
0 1 2 3 4 5 6 7 8　　　send pkt1
0 1 2 3 4 5 6 7 8　　　send pkt2　　　　　　receive pkt0, send ack0
0 1 2 3 4 5 6 7 8　　　send pkt3　　**X** *loss*　receive pkt1, send ack1

　　　　　　　　　　　(wait)
　　　　　　　　　　　　　　　　　　　receive pkt3, buffer,
　　　　　　　　　　　　　　　　　　　　　　send ack3
0 1 2 3 4 5 6 7 8　rcv ack0, send pkt4
0 1 2 3 4 5 6 7 8　rcv ack1, send pkt5
　　　　　　　　　　　　　　　　　　　receive pkt4, buffer,
　　　　　　　　　　　　　　　　　　　　　　send ack4
　　　　　　　record ack3 arrived　　　receive pkt5, buffer,
　　　　　　　　　　　　　　　　　　　　　　send ack5

　　　　　　　pkt 2 timeout
0 1 2 3 4 5 6 7 8　　　send pkt2
0 1 2 3 4 5 6 7 8　record ack4 arrived
0 1 2 3 4 5 6 7 8　record ack4 arrived　rcv pkt2; deliver pkt2,
0 1 2 3 4 5 6 7 8　　　　　　　　　　　pkt3, pkt4, pkt5; send ack2
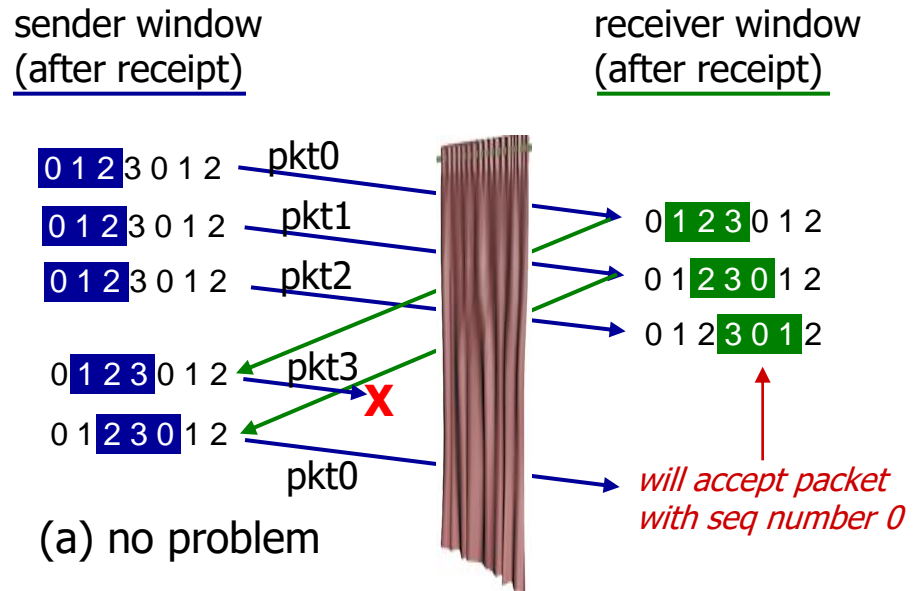
　　　　Q: what happens when ack2 arrives?

Problems with Finite Sequence No.

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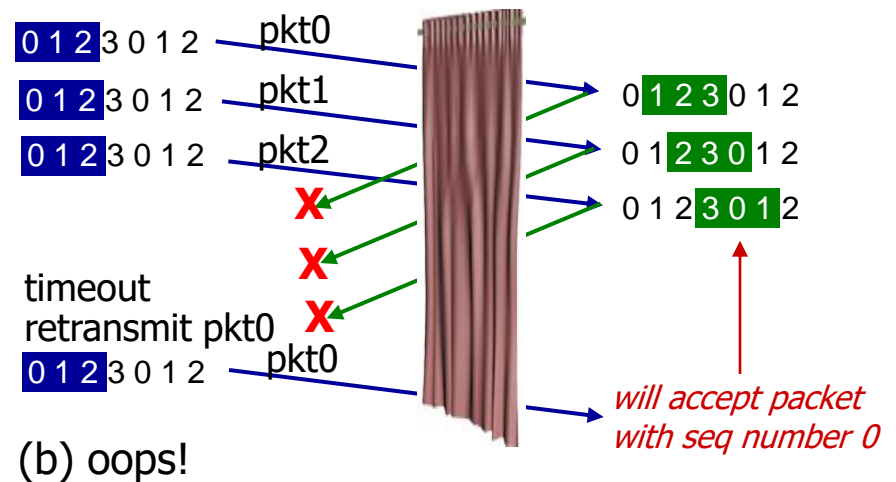- seq #'s: 0, 1, 2, 3
- window size=3

- 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)?

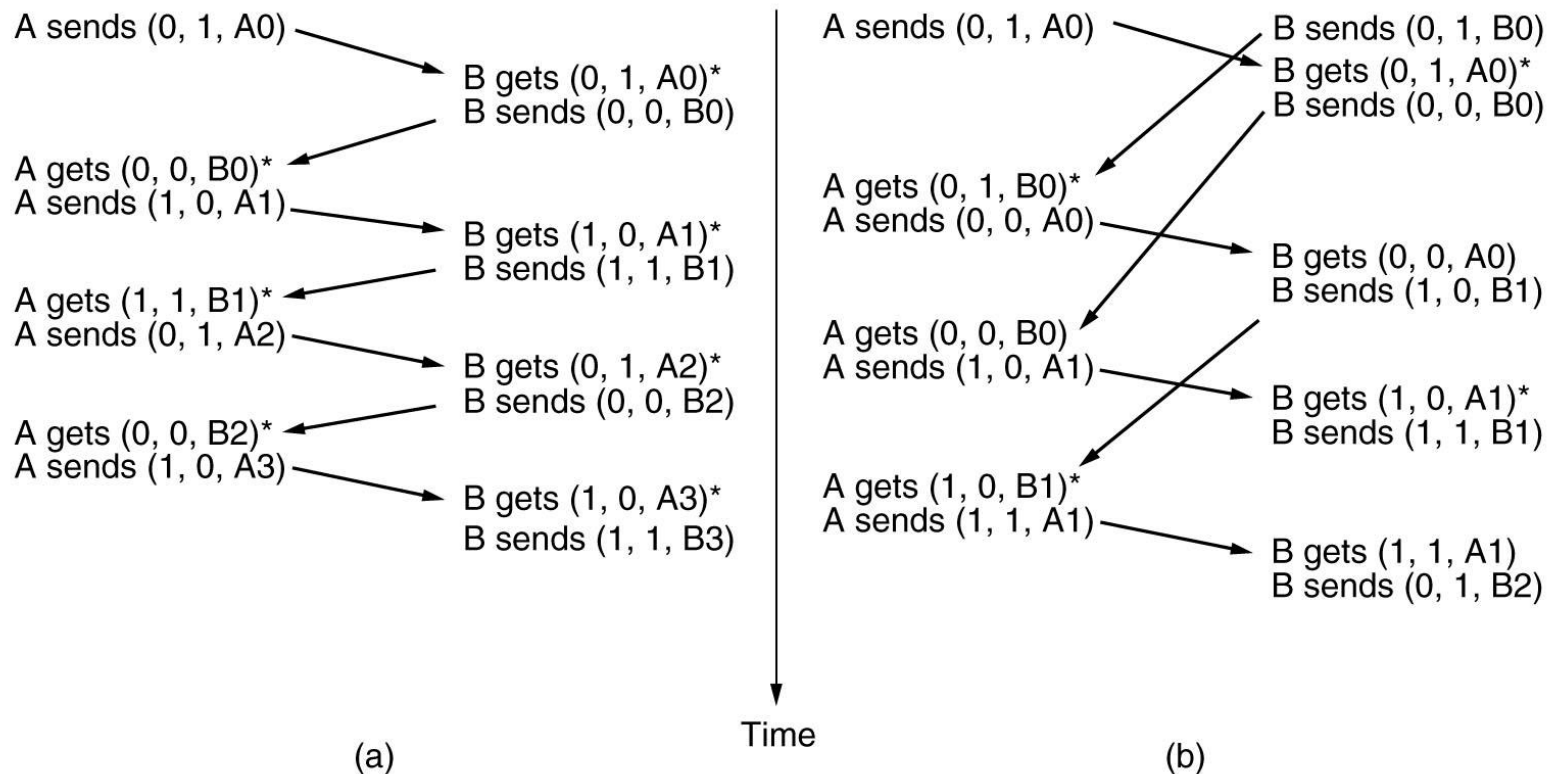Q: What should the relation in GBN?

pkt0
pkt1
pkt2

0 1 2 3 0 1 2
0 1 2 3 0 1 2
0 1 2 3 0 1 2

0 1 2 3 0 1 2
0 1 2 3 0 1 2

pkt3
X
pkt0

0 1 2 3 0 1 2
0 1 2 3 0 1 2
0 1 2 3 0 1 2
0 1 2 3 0 1 2

will accept packet with seq number 0

(a) no problem

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

pkt0
pkt1
pkt2

0 1 2 3 0 1 2
0 1 2 3 0 1 2
0 1 2 3 0 1 2

0 1 2 3 0 1 2
0 1 2 3 0 1 2
0 1 2 3 0 1 2

X
X

timeout
retransmit pkt0
X
pkt0

0 1 2 3 0 1 2

will accept packet with seq number 0

(b) oops!

Bi-directional Transfer

❖ Send ack in the same frame for data
❖ Known as Piggybacking

A sends (0, 1, A0)
    B gets (0, 1, A0)*
    B sends (0, 0, B0)
A gets (0, 0, B0)*
A sends (1, 0, A1)
    B gets (1, 0, A1)*
    B sends (1, 1, B1)
A gets (1, 1, B1)*
A sends (0, 1, A2)
    B gets (0, 1, A2)*
    B sends (0, 0, B2)
A gets (0, 0, B2)*
A sends (1, 0, A3)
    B gets (1, 0, A3)*
    B sends (1, 1, B3)

(a)

A sends (0, 1, A0)
    B sends (0, 1, B0)
    B gets (0, 1, A0)*
    B sends (0, 0, B0)
A gets (0, 1, B0)*
A sends (0, 0, A0)
    B gets (0, 0, A0)
    B sends (1, 0, B1)
A gets (0, 0, B0)
A sends (1, 0, A1)
    B gets (1, 0, A1)*
    B sends (1, 1, B1)
A gets (1, 0, B1)*
A sends (1, 1, A1)
    B gets (1, 1, A1)
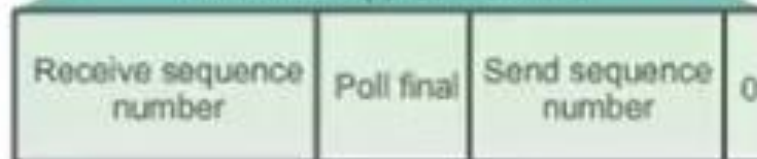   B sends (0, 1, B2)

(b)

Time

Delayed Duplicate Problem

❖ Old copies of a packet may appear at the sender or receiver

❖ Since sequence numbers wrap around, care must be taken to guard against such duplicate packet

- Do not use a sequence number, until all the earlier packets with the same sequence number have died out.
- TCP extension for high speed network assumes maximum packet lifetime of approx. three minutes
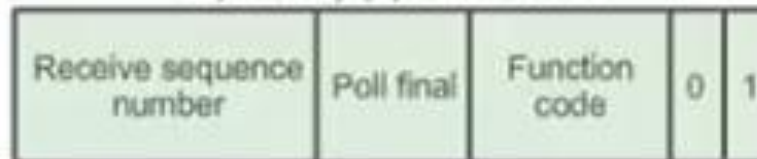
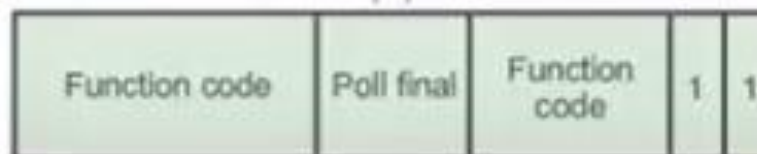# Sliding Window at Link Layer: HDLC
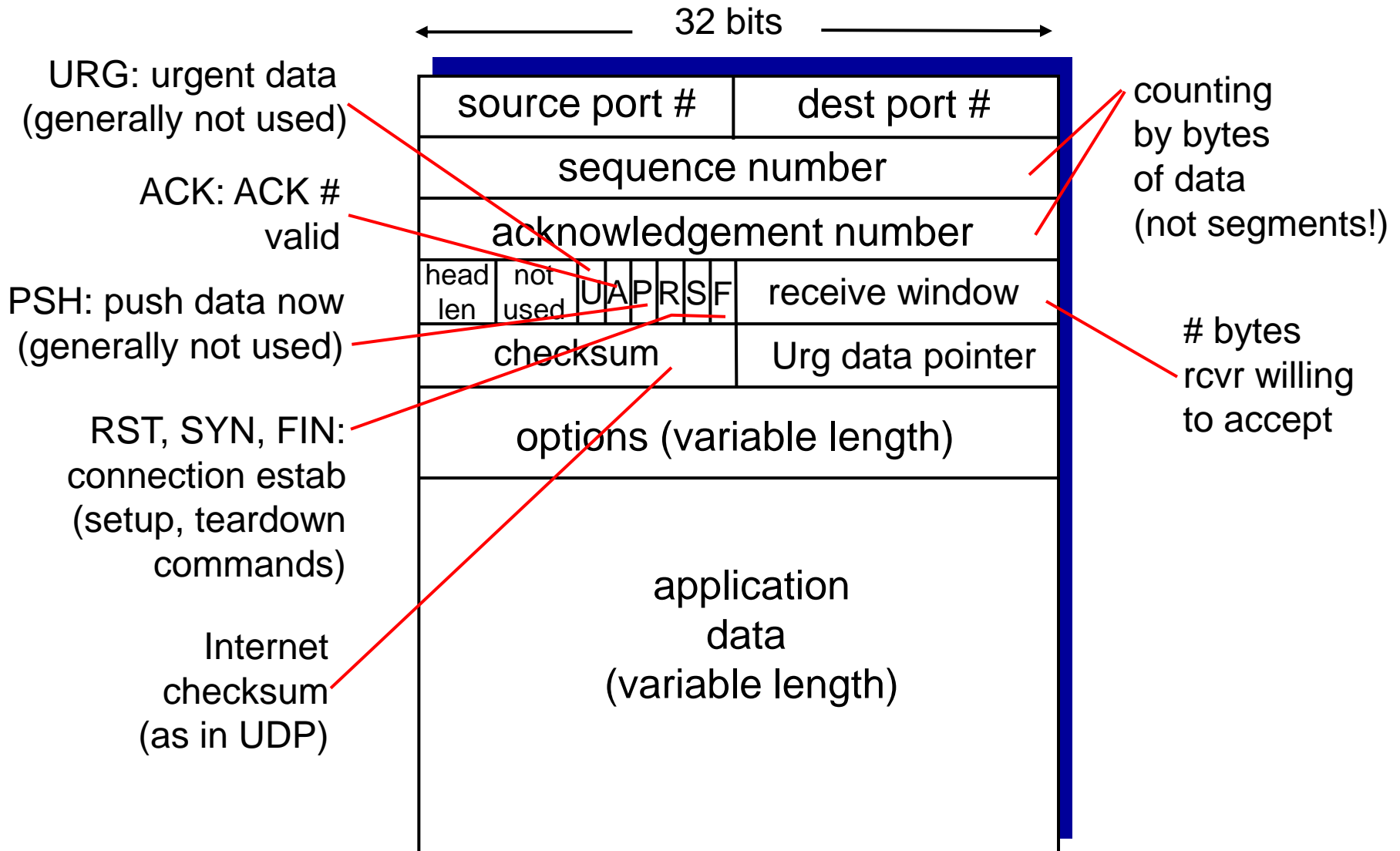

HDLC Frame Structure

Sliding Window at Transport Layer: TCP

32 bits

URG: urgent data
(generally not used)

ACK: ACK #
valid

PSH: push data now
(generally not used)

RST, SYN, FIN:
connection estab
(setup, teardown
commands)

Internet
checksum
(as in UDP)

source port #	dest port #
sequence number	
acknowledgement number	
head len	not used	U A P R S F	receive window
checksum	Urg data pointer		
options (variable length)			
application data (variable length)			

counting
by bytes
of data
(not segments!)

bytes
rcvr willing
to accept

TCP Header

Sliding Window in TCP (2)

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

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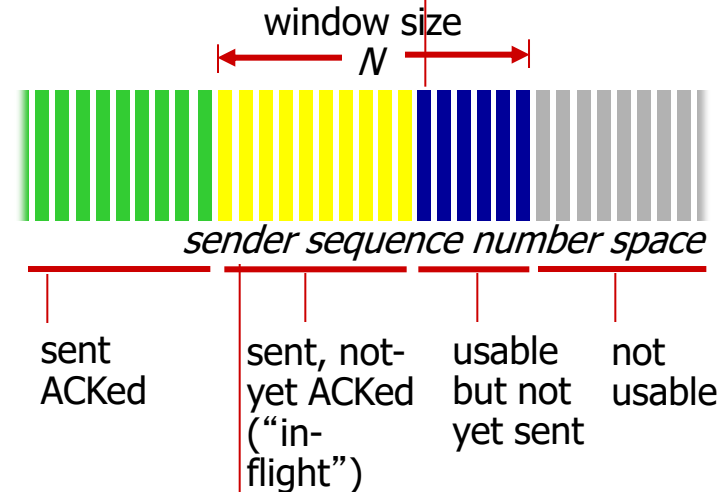- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments
- A: TCP spec doesn't say, - up to implementor

outgoing segment from sender

source port #	dest port #
sequence number	
acknowledgement number	
	rwnd
checksum	urg pointer

window size
N

sender sequence number space

sent ACKed | sent, not-yet ACKed ("in-flight") | usable but not yet sent | not usable

incoming segment to sender

source port #	dest port #
sequence number	
acknowledgement number	
	A	rwnd
checksum	urg pointer

TCP round trip time, timeout

❖ **timeout interval: `EstimatedRTT` plus "safety margin"**
  - large variation in `EstimatedRTT` -> larger safety margin

❖ estimate SampleRTT deviation from EstimatedRTT:

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

         (typically, β = 0.25)
```

`TimeoutInterval = EstimatedRTT + 4*DevRTT`

estimated RTT   "safety margin"

TCP ACK generation [RFC 1122, RFC 2581]

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

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

Host A   Host B

Seq=92, 8 bytes of data
Seq=100, 20 bytes of data

X

timeout

ACK=100
ACK=100
ACK=100
ACK=100

Seq=100, 20 bytes of data

fast retransmit after sender
receipt of triple duplicate ACK