

Computer Networking

Lent Term M/W/F 11-midday
LT1 in Gates Building

Slide Set 4

Andrew W. Moore

andrew.moore@cl.cam.ac.uk

February 2014

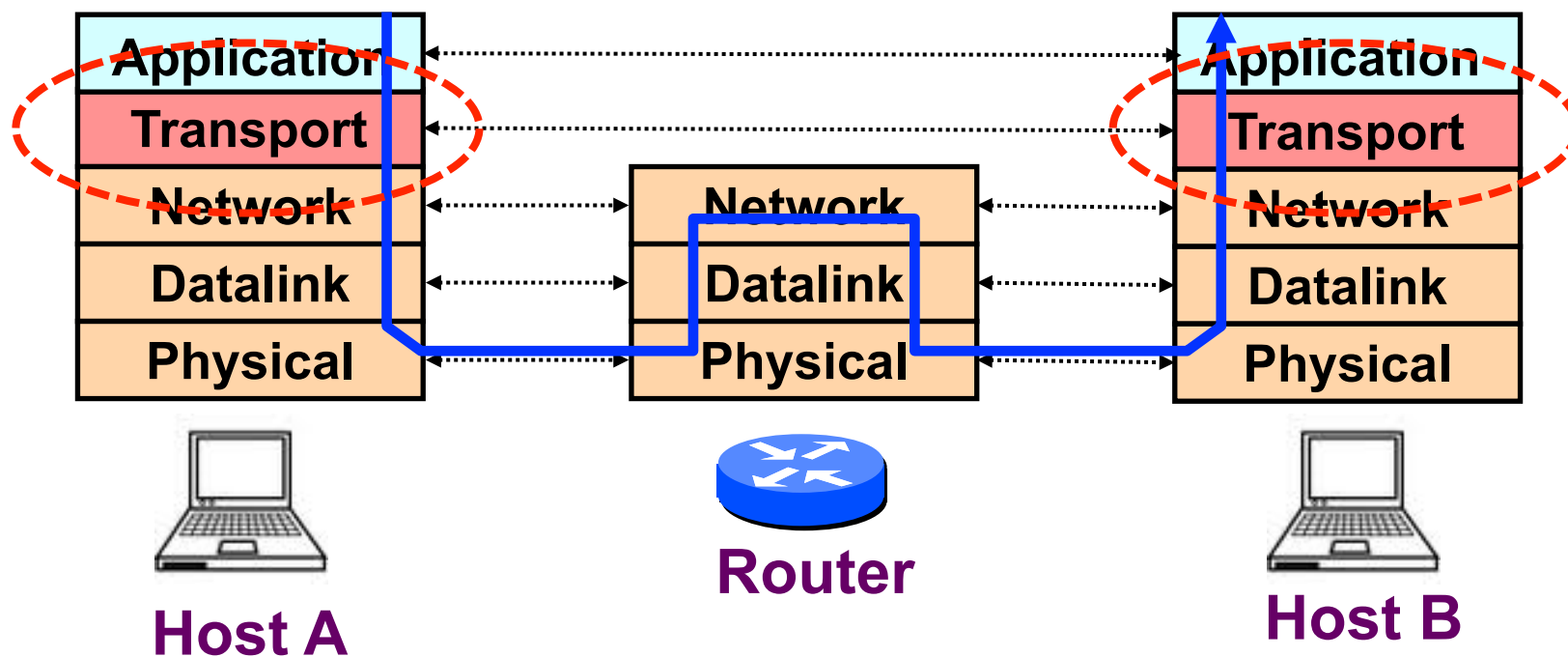
Topic 5a – Transport

Our goals:

- understand principles behind transport layer services:
 - multiplexing/
demultiplexing
 - reliable data transfer
 - flow control
 - congestion control
- learn about transport layer protocols in the Internet:
 - UDP: connectionless transport
 - TCP: connection-oriented transport
 - TCP congestion control

Transport Layer

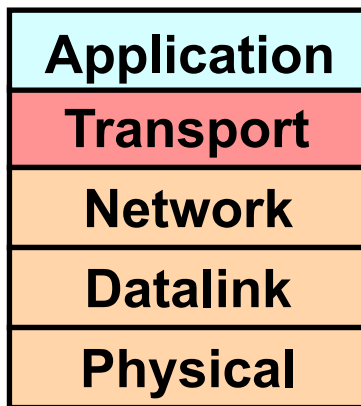
- Commonly a layer **at end-hosts**, between the application and network layer



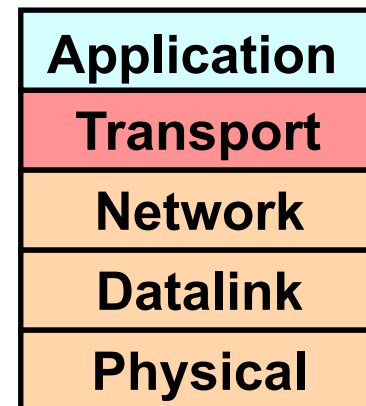
Why a transport layer?

- IP packets are addressed to a host but end-to-end communication is between application processes at hosts
 - Need a way to decide which packets go to which applications (*more multiplexing*)

Why a transport layer?

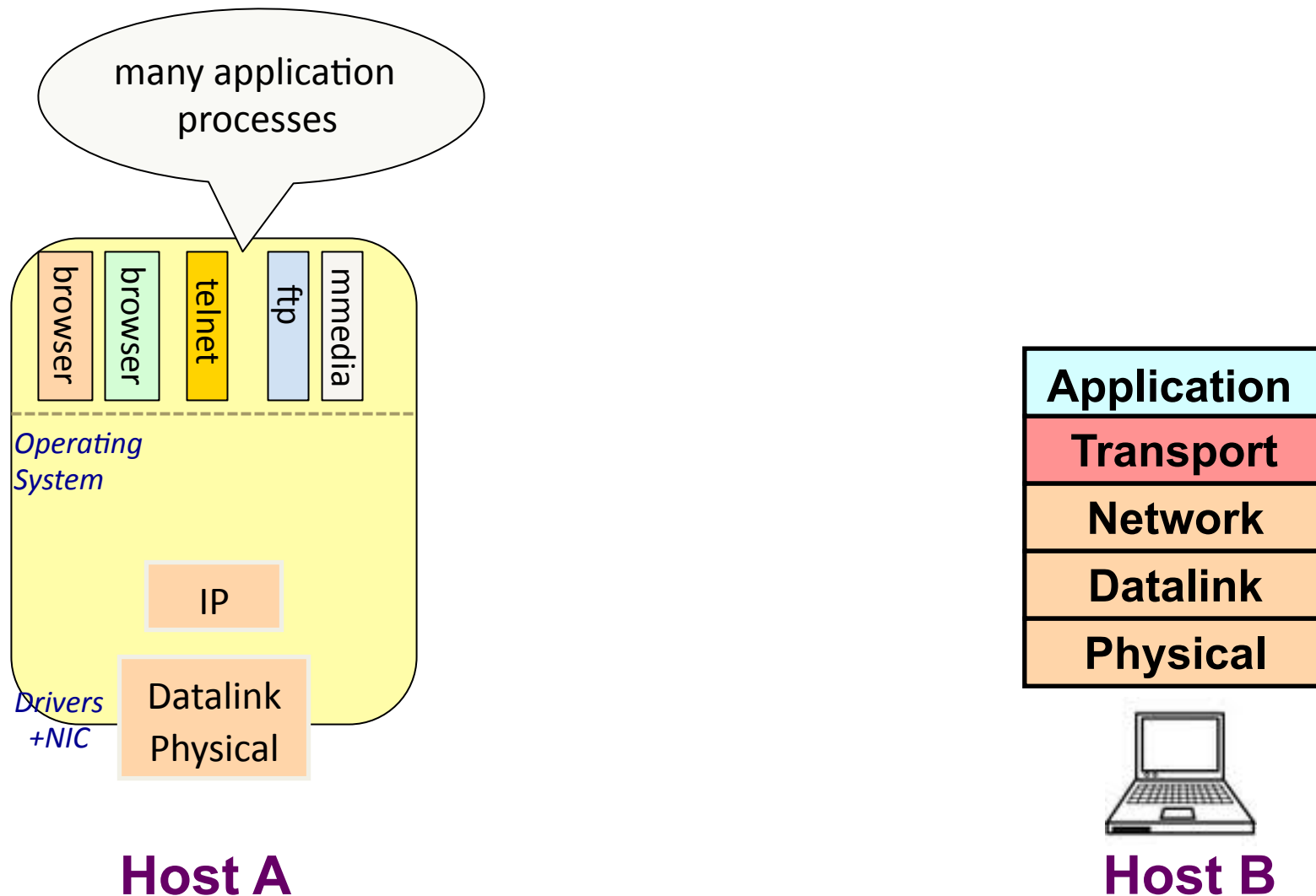


Host A

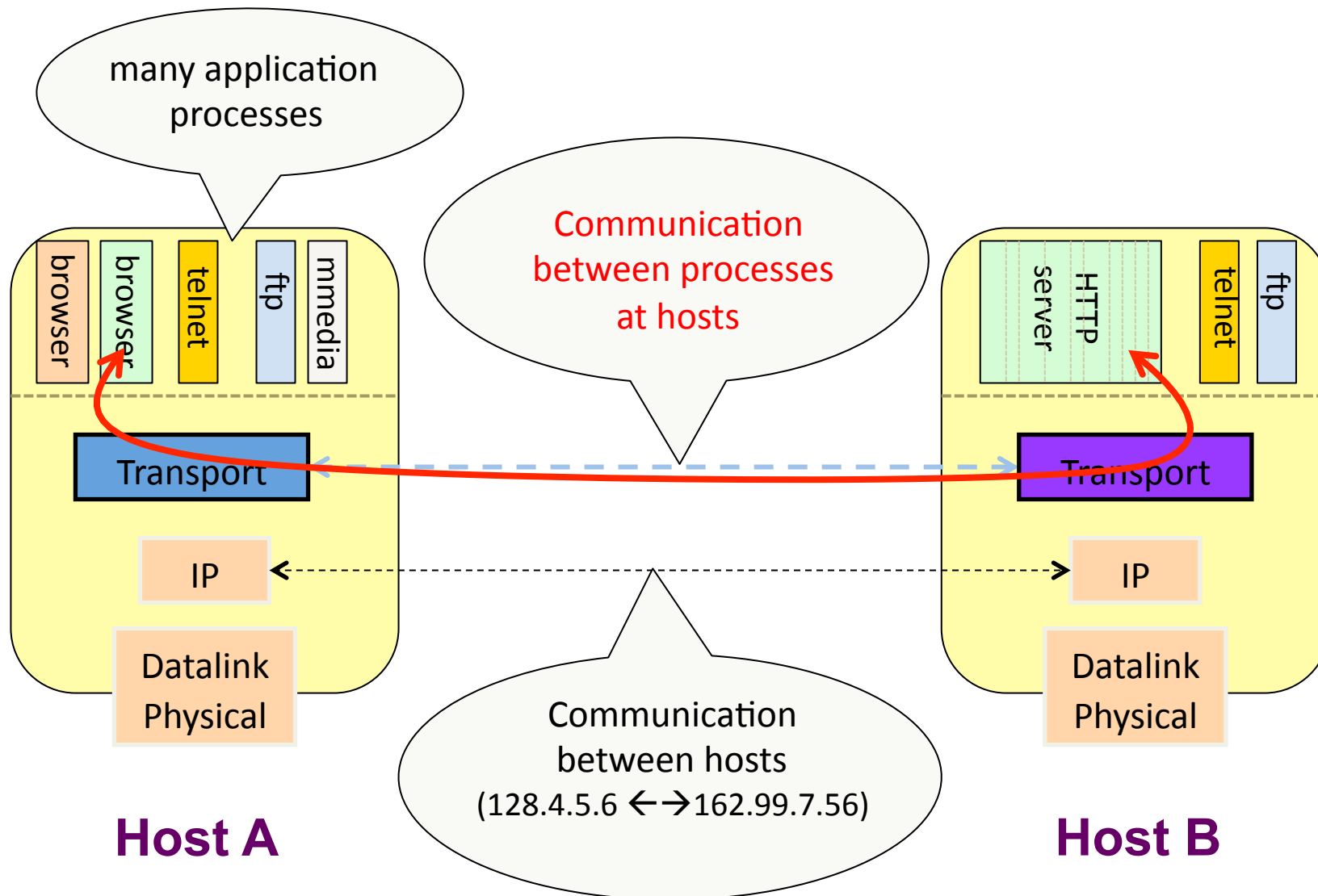


Host B

Why a transport layer?



Why a transport layer?



Why a transport layer?

- IP packets are addressed to a host but end-to-end communication is between application processes at hosts
 - Need a way to decide which packets go to which applications (mux/demux)
- IP provides a weak service model (*best-effort*)
 - Packets can be corrupted, delayed, dropped, reordered, duplicated
 - No guidance on how much traffic to send and when
 - Dealing with this is tedious for application developers

Role of the Transport Layer

- Communication between application processes
 - Multiplexing between application processes
 - Implemented using *ports*

Role of the Transport Layer

- Communication between application processes
- Provide common end-to-end services for app layer [optional]
 - Reliable, in-order data delivery
 - Paced data delivery: flow and congestion-control
 - too fast may overwhelm the network
 - too slow is not efficient

Role of the Transport Layer

- Communication between processes
- Provide common end-to-end services for app layer [optional]
- TCP and UDP are the common transport protocols
 - also SCTP, MTCP, SST, RDP, DCCP, ...

Role of the Transport Layer

- Communication between processes
- Provide common end-to-end services for app layer [optional]
- TCP and UDP are the common transport protocols
- **UDP is a minimalist, no-frills transport protocol**
 - only provides mux/demux capabilities

Role of the Transport Layer

- Communication between processes
- Provide common end-to-end services for app layer [optional]
- TCP and UDP are the common transport protocols
- UDP is a minimalist, no-frills transport protocol
- TCP is the *totus porcus* protocol
 - offers apps a reliable, in-order, byte-stream abstraction
 - with congestion control
 - but **no** performance (delay, bandwidth, ...) guarantees

Role of the Transport Layer

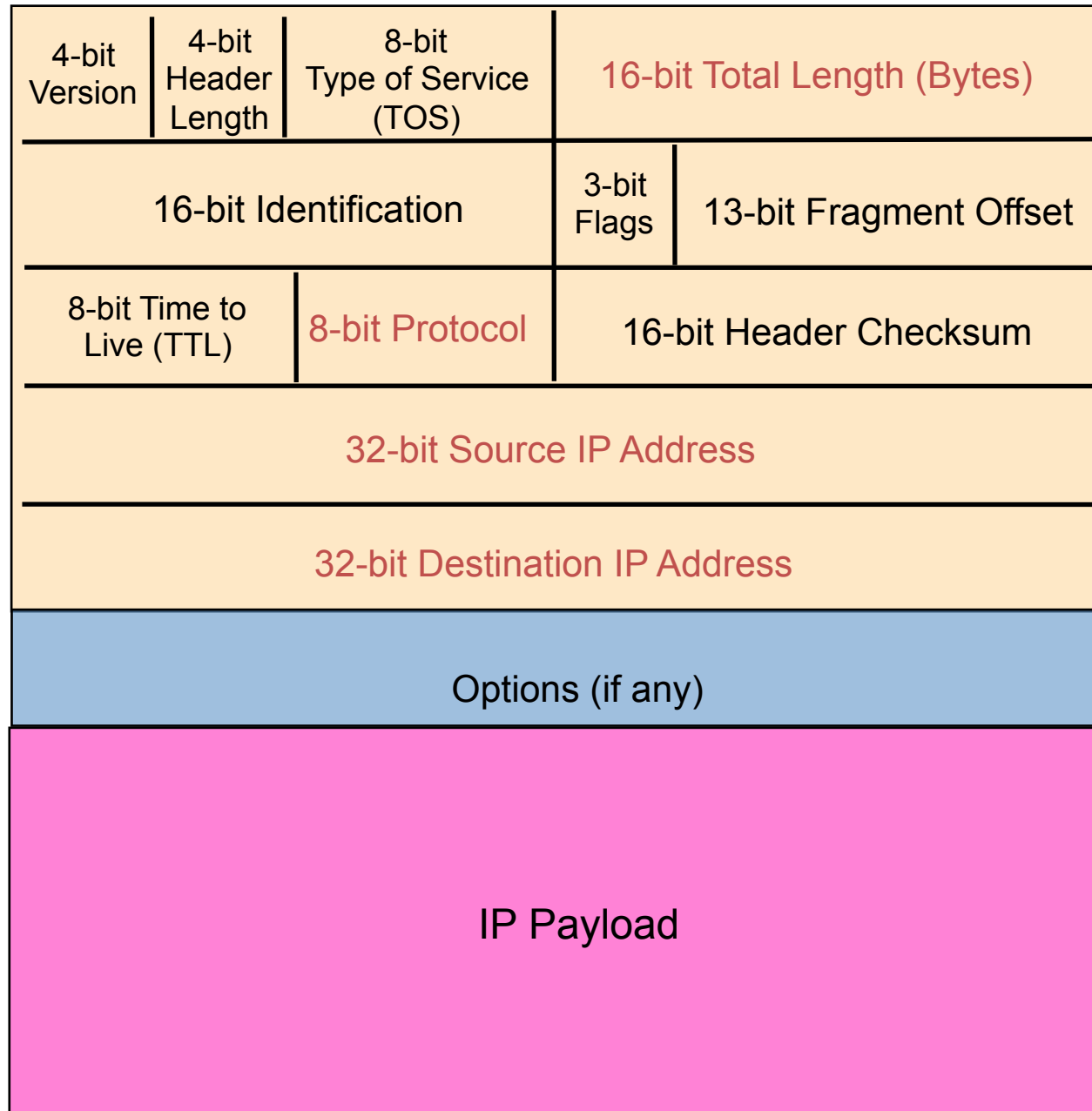
- Communication between processes
 - mux/demux from and to application processes
 - implemented using ports

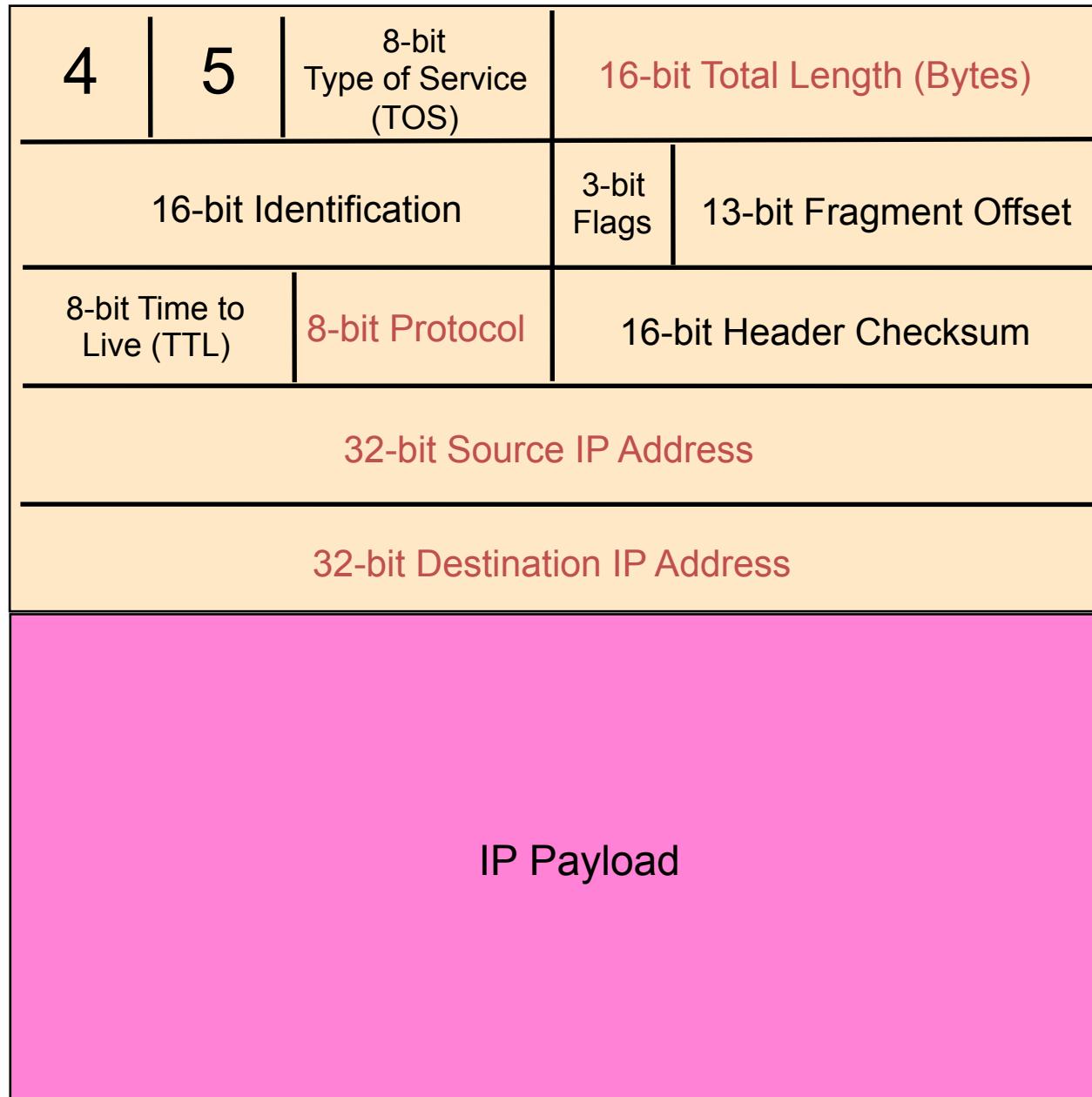
Context: Applications and Sockets

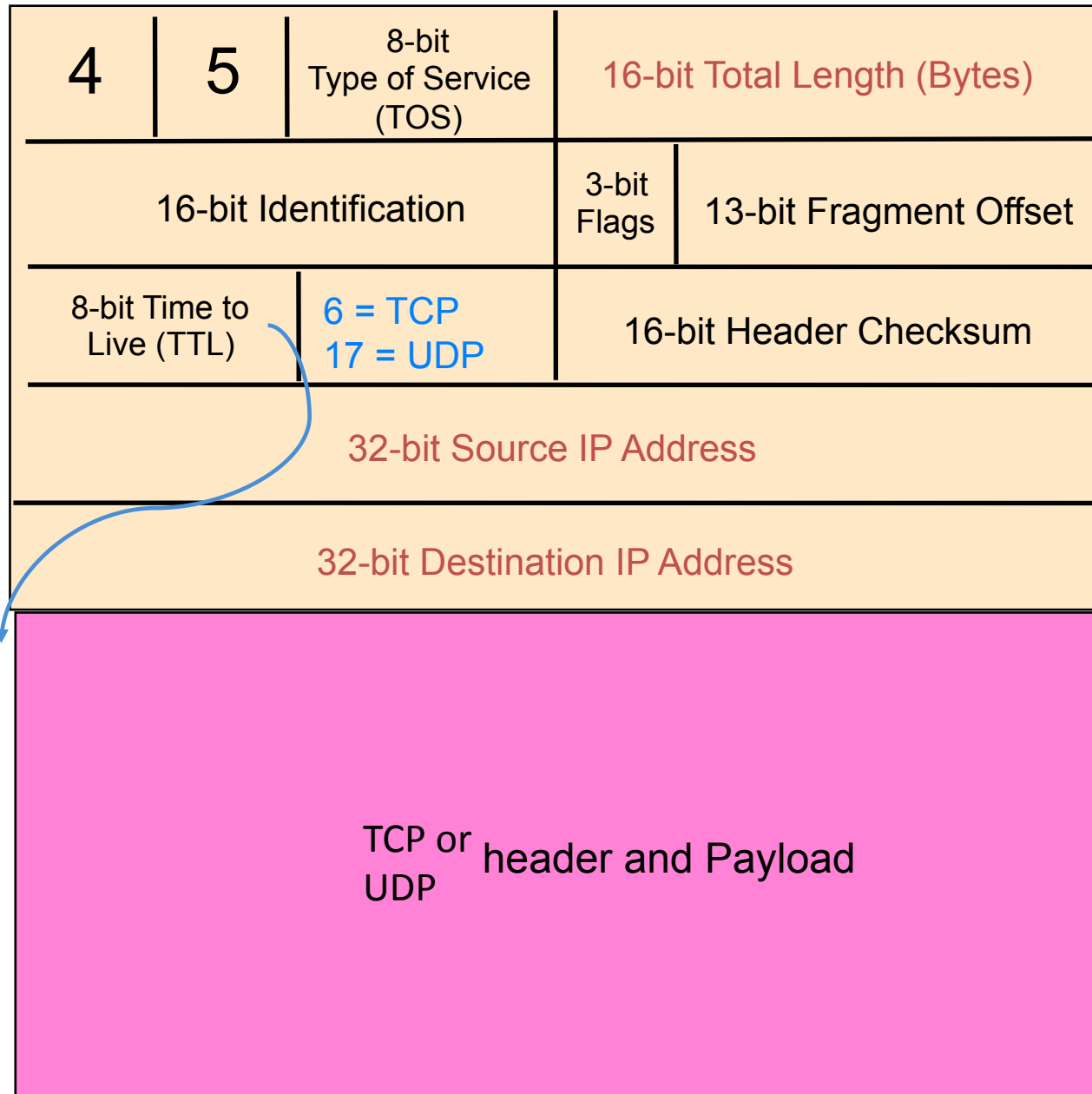
- Socket: software abstraction by which an application process exchanges network messages with the (transport layer in the) operating system
 - `socketID = socket(..., socket.TYPE)`
 - `socketID.sendto(message, ...)`
 - `socketID.recvfrom(...)`
- Two important types of sockets
 - UDP socket: TYPE is `SOCK_DGRAM`
 - TCP socket: TYPE is `SOCK_STREAM`

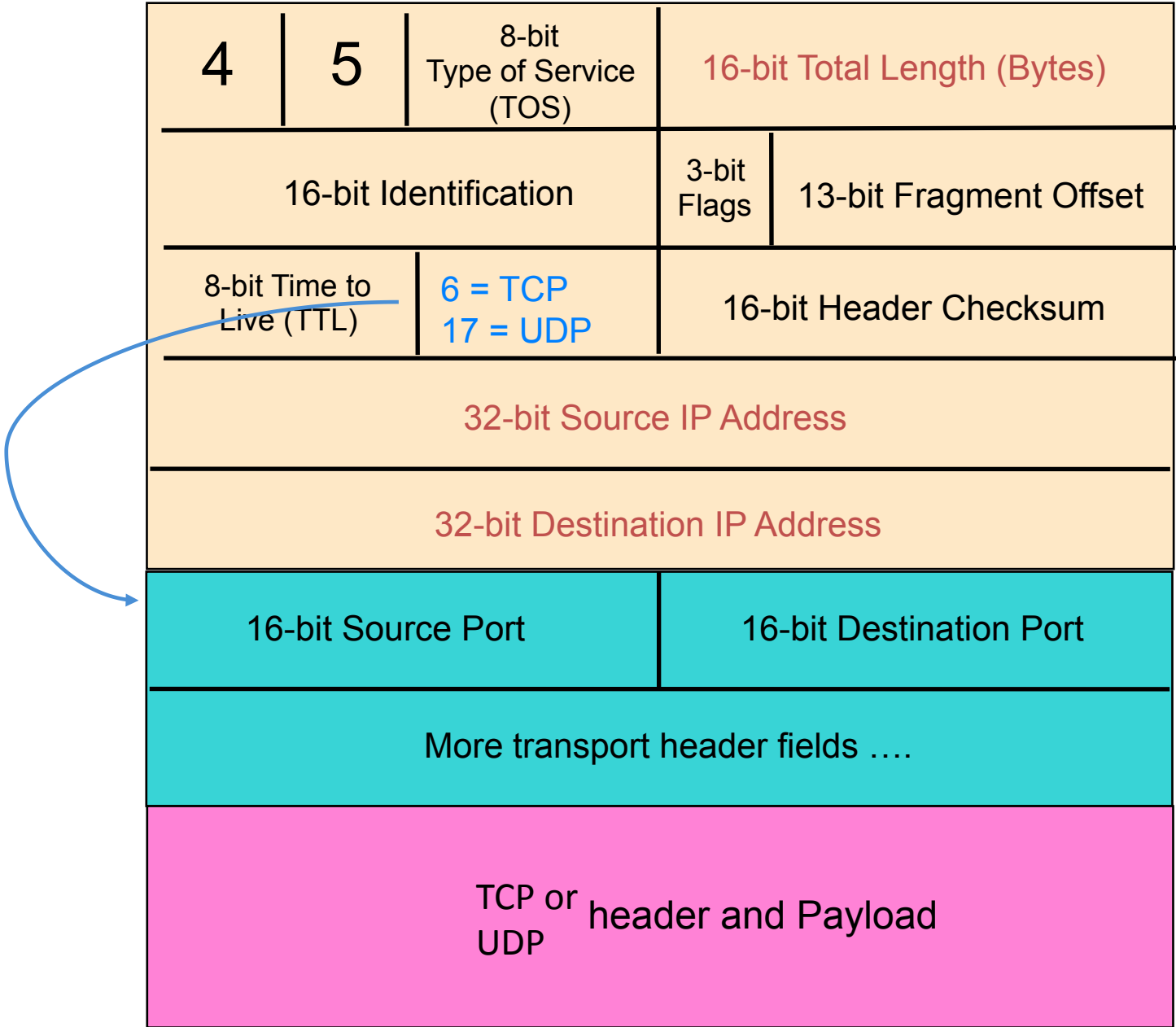
Ports

- Problem: deciding which app (socket) gets which packets
- Solution: *port* as a transport layer identifier
 - 16 bit identifier
 - OS stores mapping between sockets and *ports*
 - a packet carries a source and destination port number in its transport layer header
- For UDP ports (SOCK_DGRAM)
 - OS stores (local port, local IP address) \leftrightarrow socket
- For TCP ports (SOCK_STREAM)
 - OS stores (local port, local IP, remote port, remote IP) \leftrightarrow socket









Recap: Multiplexing and Demultiplexing

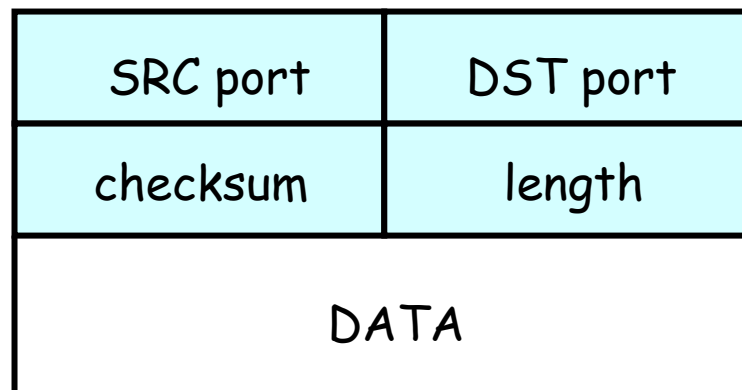
- Host receives IP packets
 - Each IP header has source and destination **IP address**
 - Each Transport Layer header has source and destination **port** number
- Host uses IP addresses and port numbers to direct the message to appropriate **socket**

More on Ports

- Separate 16-bit port address space for UDP and TCP
- “Well known” ports (0-1023): everyone agrees which services run on these ports
 - e.g., ssh:22, http:80
 - helps client know server’s port
- Ephemeral ports (most 1024-65535): dynamically selected: as the source port for a client process

UDP: User Datagram Protocol

- Lightweight communication between processes
 - Avoid overhead and delays of ordered, reliable delivery
- UDP described in RFC 768 – (1980!)
 - Destination IP address and port to support demultiplexing
 - Optional error checking on the packet contents
 - (checksum field of 0 means “don’t verify checksum”)

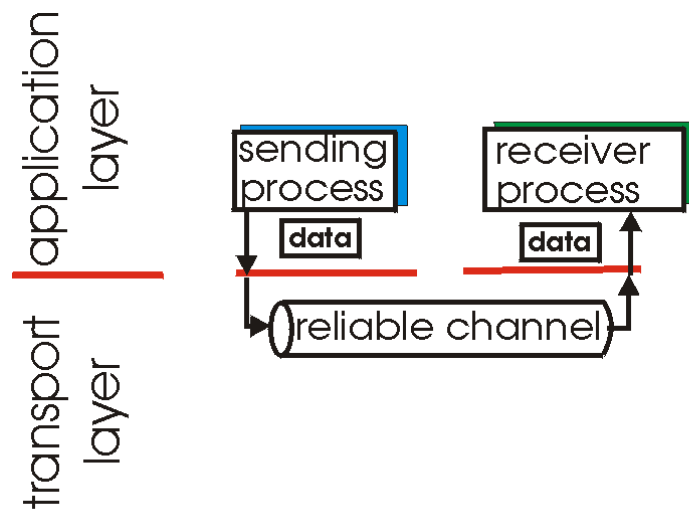


Why a transport layer?

- IP packets are addressed to a host but end-to-end communication is between application processes at hosts
 - Need a way to decide which packets go to which applications (mux/demux)
- IP provides a weak service model (*best-effort*)
 - Packets can be corrupted, delayed, dropped, reordered, duplicated

Principles of Reliable data transfer

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



(a) provided service

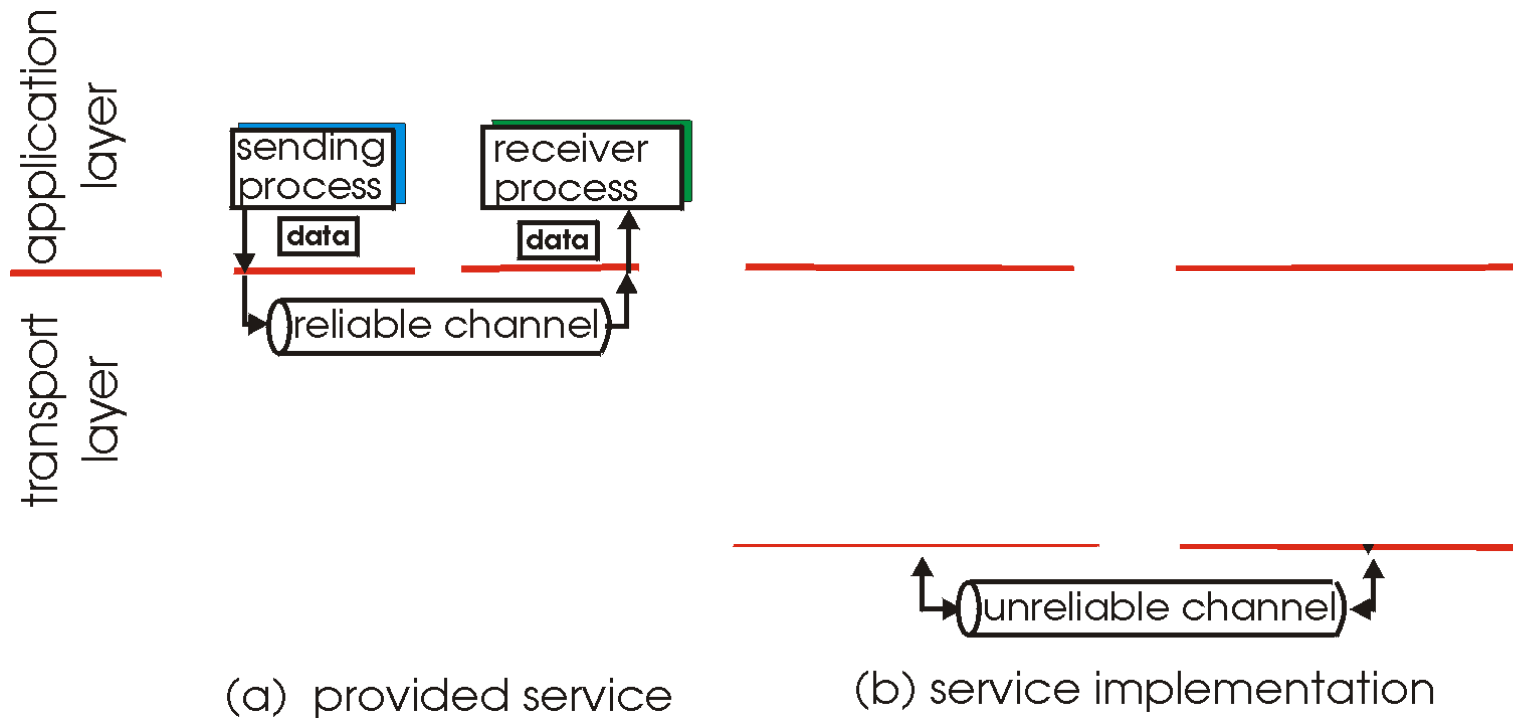
- In a perfect world, reliable transport is easy

But the Internet default is *best-effort*

- All the bad things best-effort can do
 - a packet is corrupted (bit errors)
 - a packet is lost
 - a packet is delayed (*why?*)
 - packets are reordered (*why?*)
 - a packet is duplicated (*why?*)

Principles of Reliable data transfer

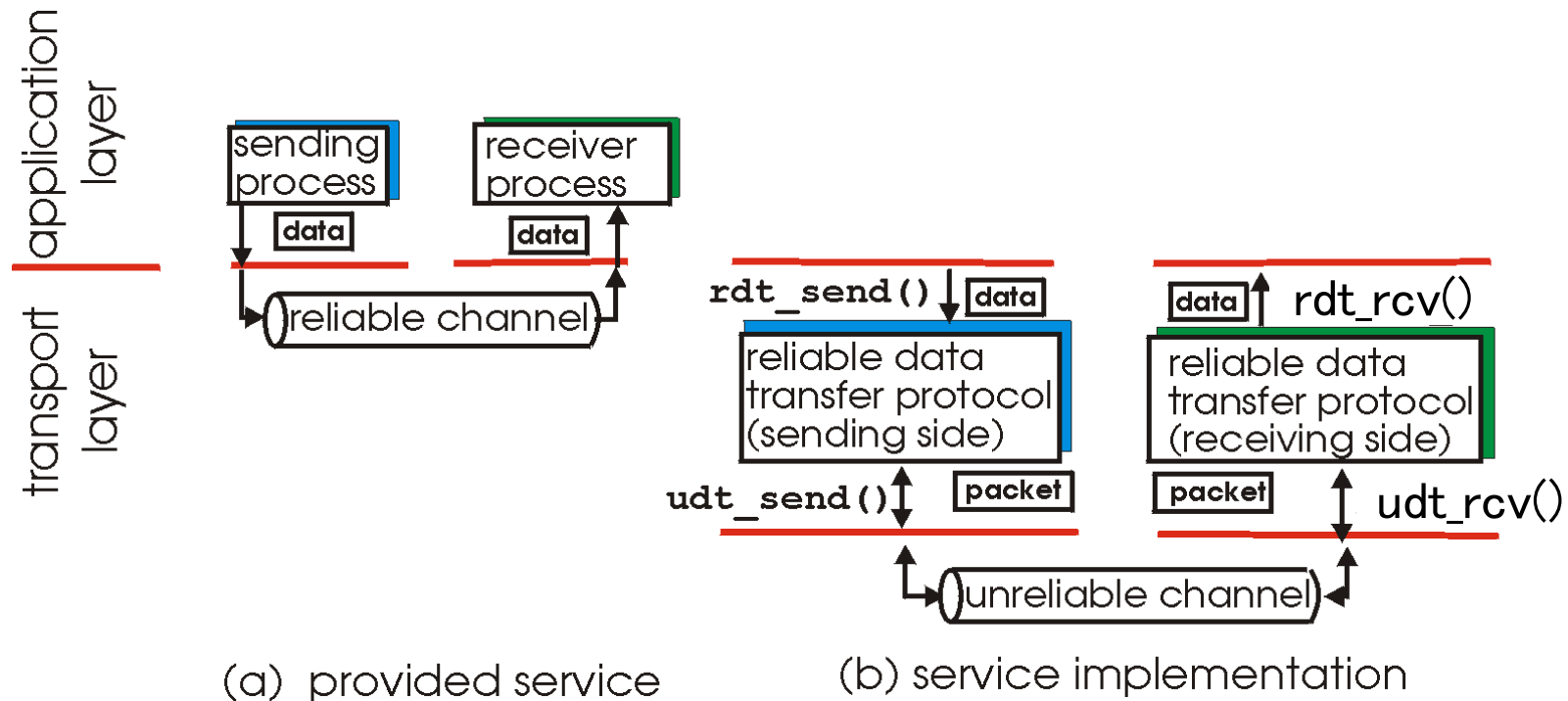
- important in app., transport, link layers
- top-10 list of important networking topics!



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

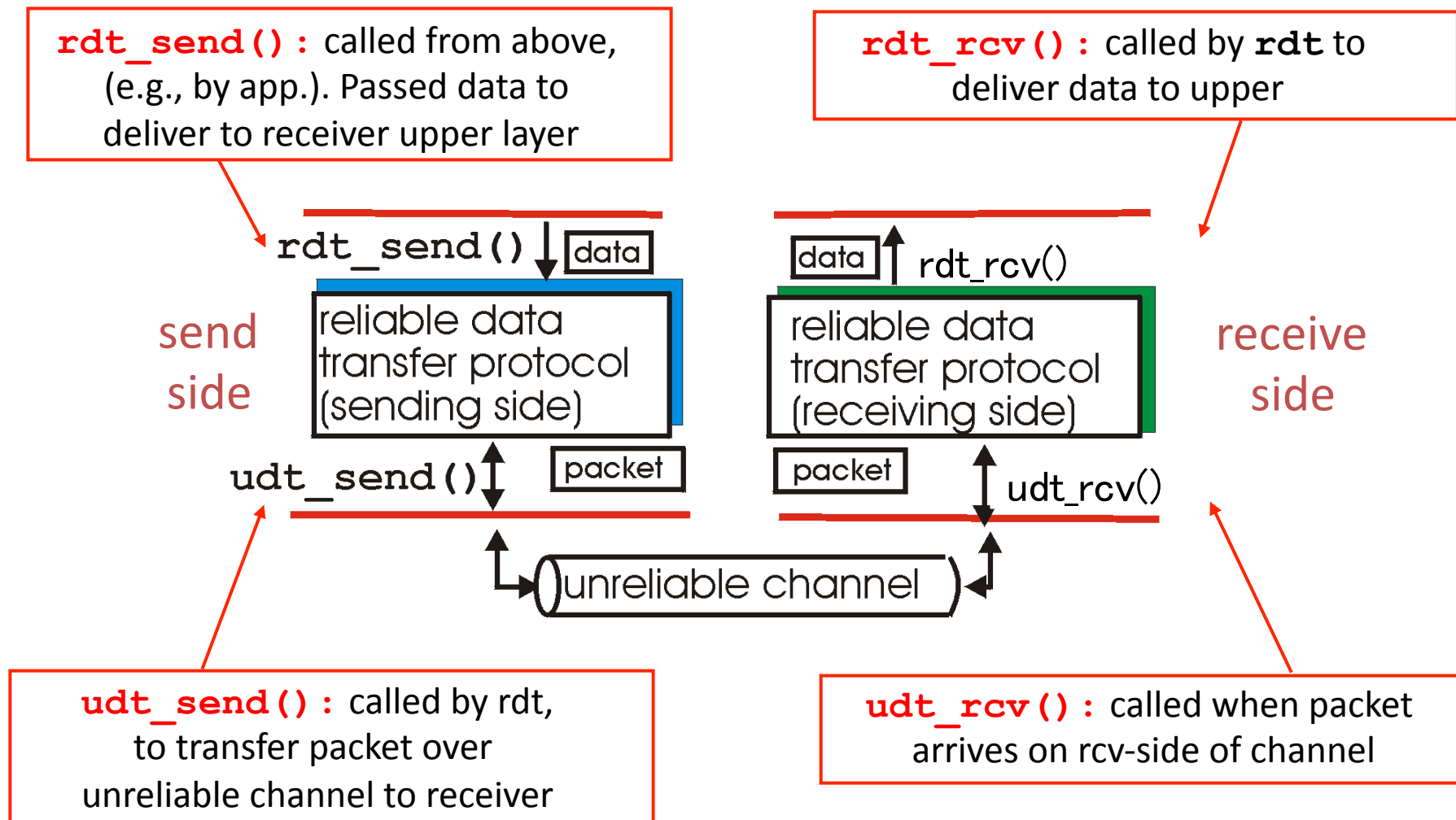
Principles of Reliable data transfer

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



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

Reliable data transfer: getting started

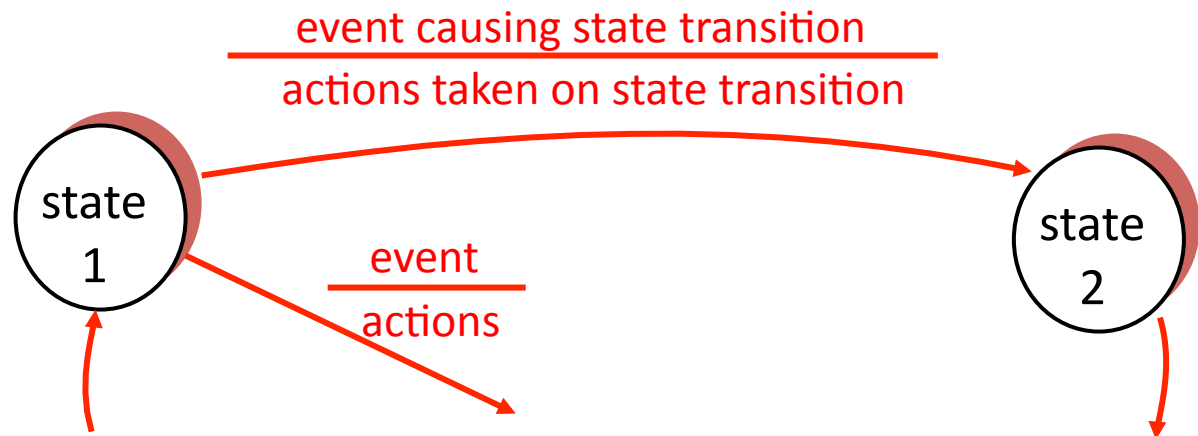


Reliable data transfer: getting started

We' ll:

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

state: when in this “state” next state uniquely determined by next event



KR state machines – a note.

Beware

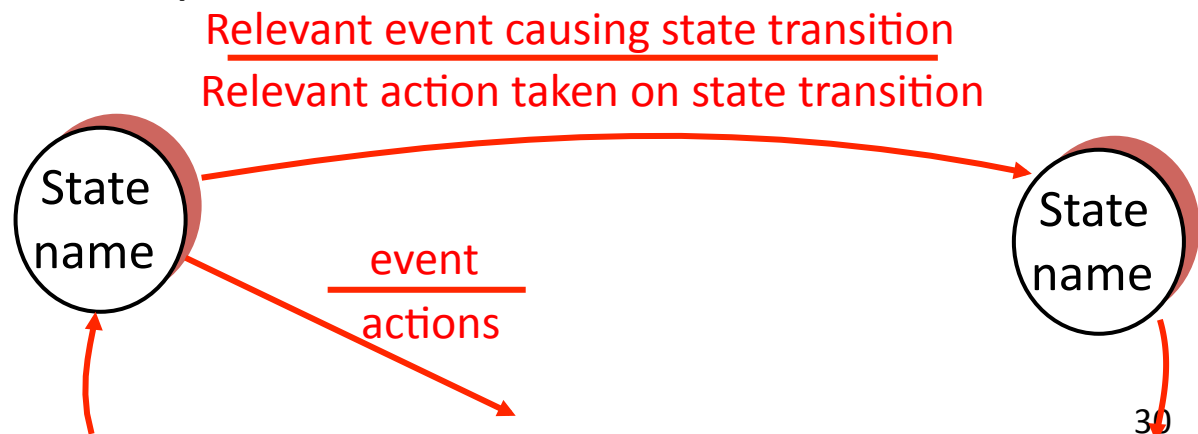
Kurose and Ross has a confusing/confused attitude to state-machines.

I've attempted to normalise the representation.

UPSHOT: these slides have differing information to the KR book (from which the RDT example is taken.)

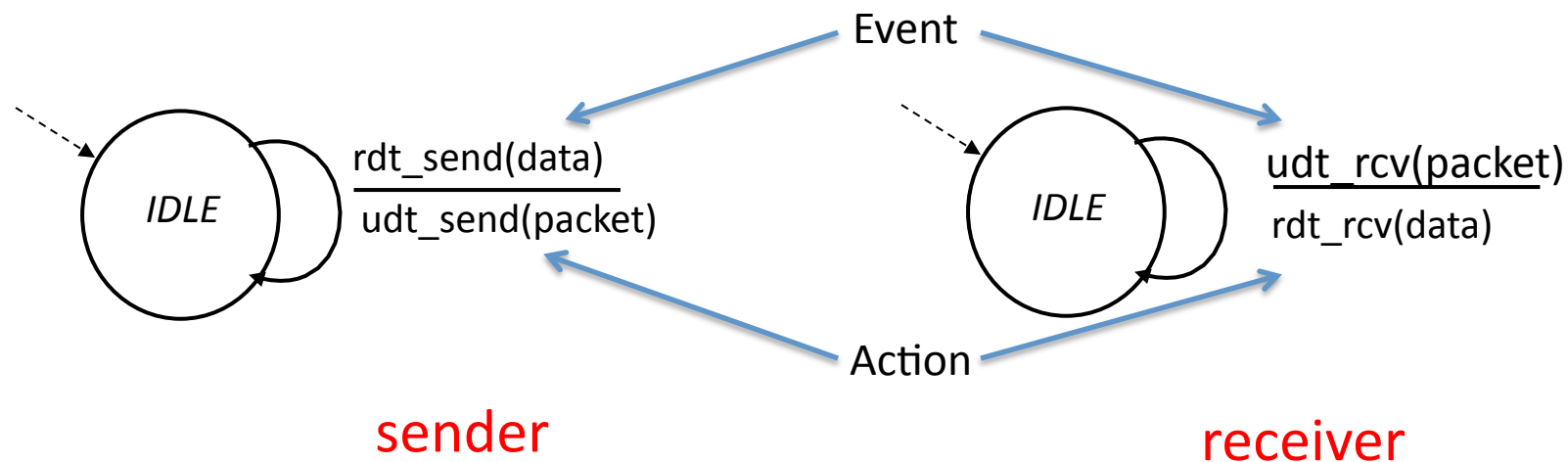
in KR “actions taken” appear wide-ranging, my interpretation is more specific/relevant.

state: when in this “state”
next state uniquely
determined by next
event



Rdt1.0: reliable transfer over a reliable channel

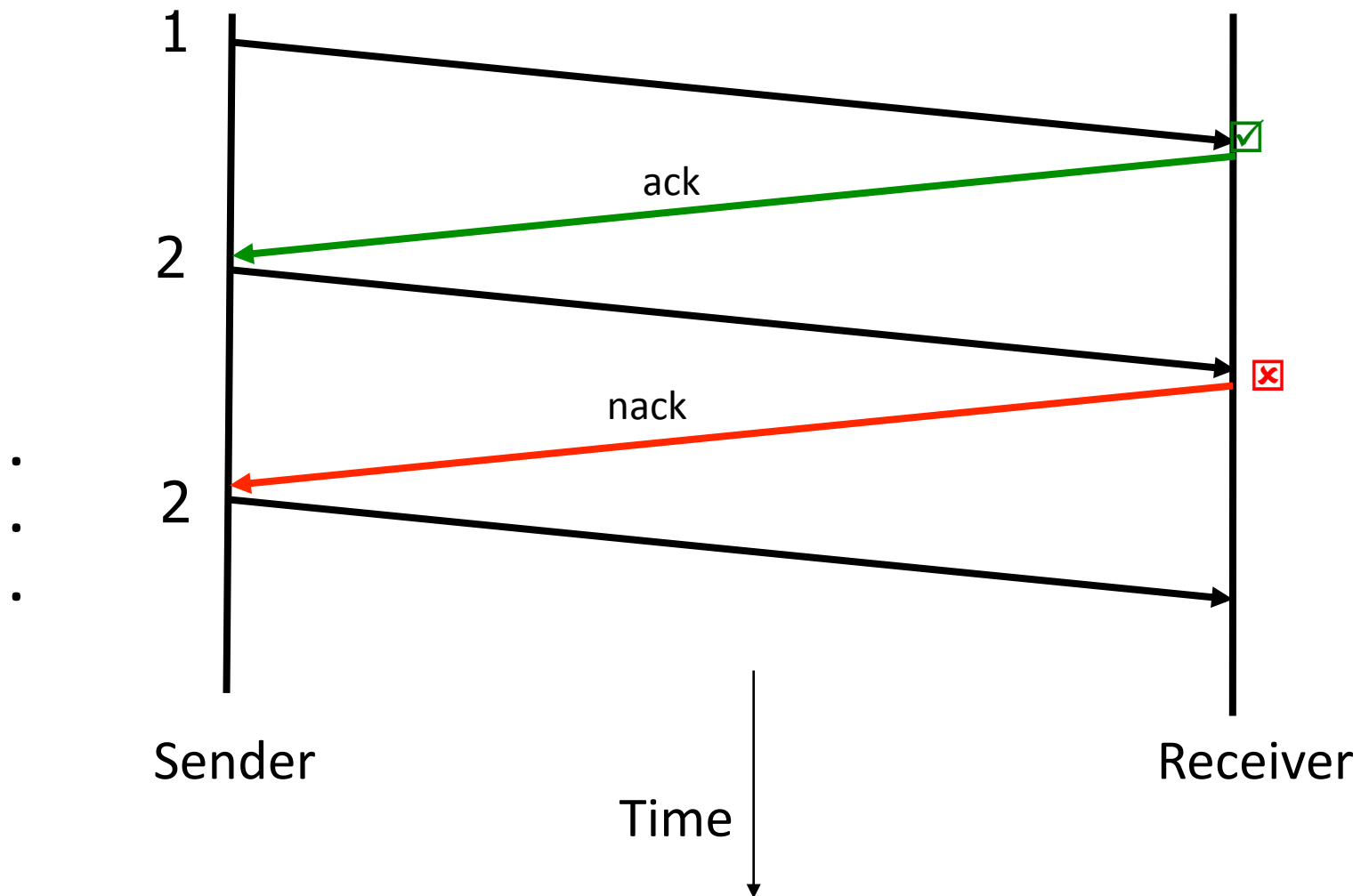
- underlying channel perfectly reliable
 - no bit errors
 - no loss of packets
- separate FSMs for sender, receiver:
 - sender sends data into underlying channel
 - receiver read data from underlying channel



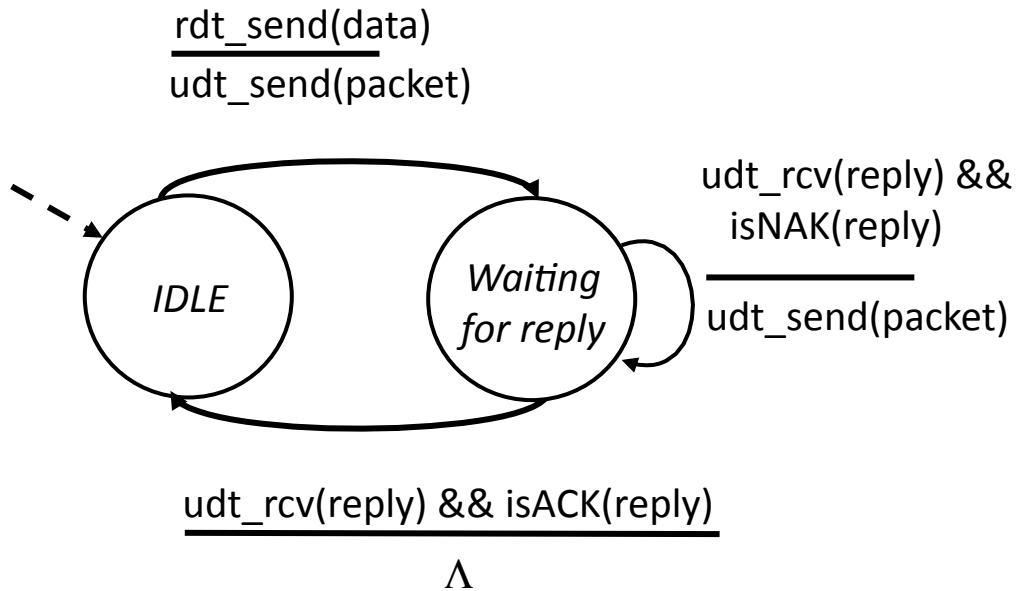
Rdt2.0: channel with bit errors

- 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 packet received is OK
 - *negative acknowledgements (NAKs)*: receiver explicitly tells sender that packet had errors
 - sender retransmits packet on receipt of NAK
- new mechanisms in **rdt2.0** (beyond **rdt1.0**):
 - error detection
 - receiver feedback: control msgs (ACK,NAK) receiver->sender

Dealing with Packet Corruption



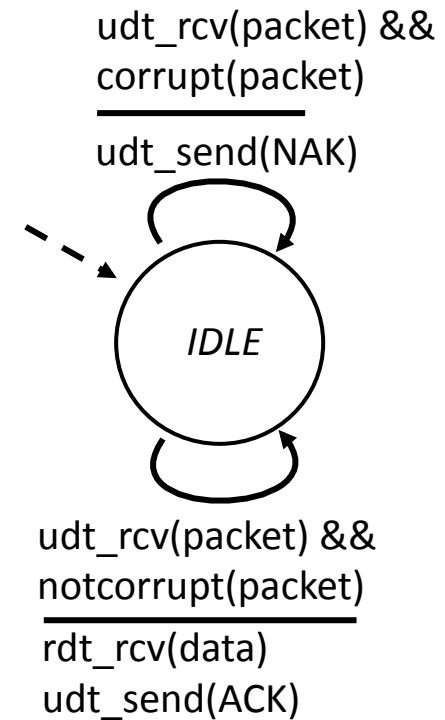
rdt2.0: FSM specification



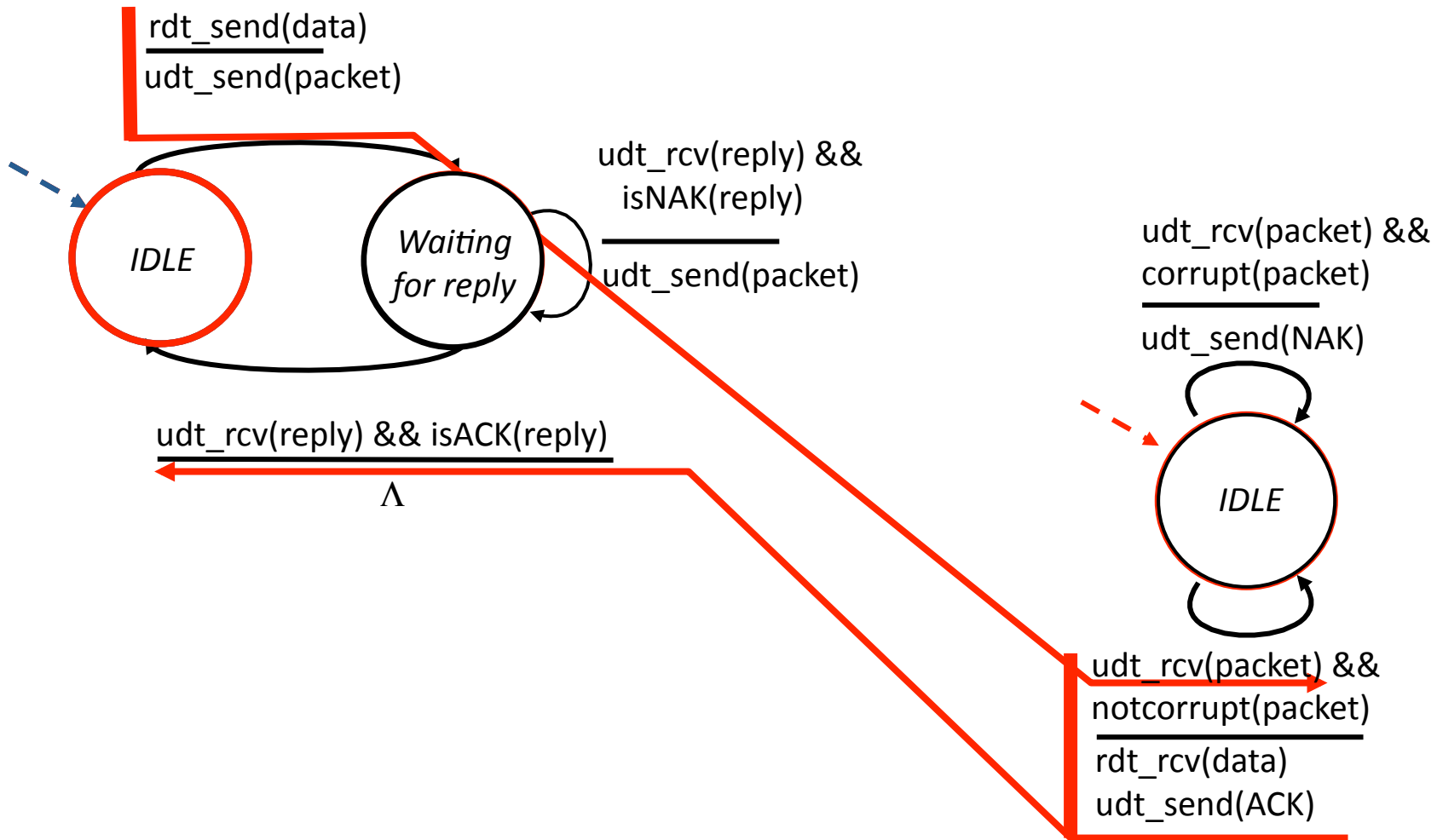
sender

Note: the sender holds a copy of the packet being sent until the delivery is acknowledged.

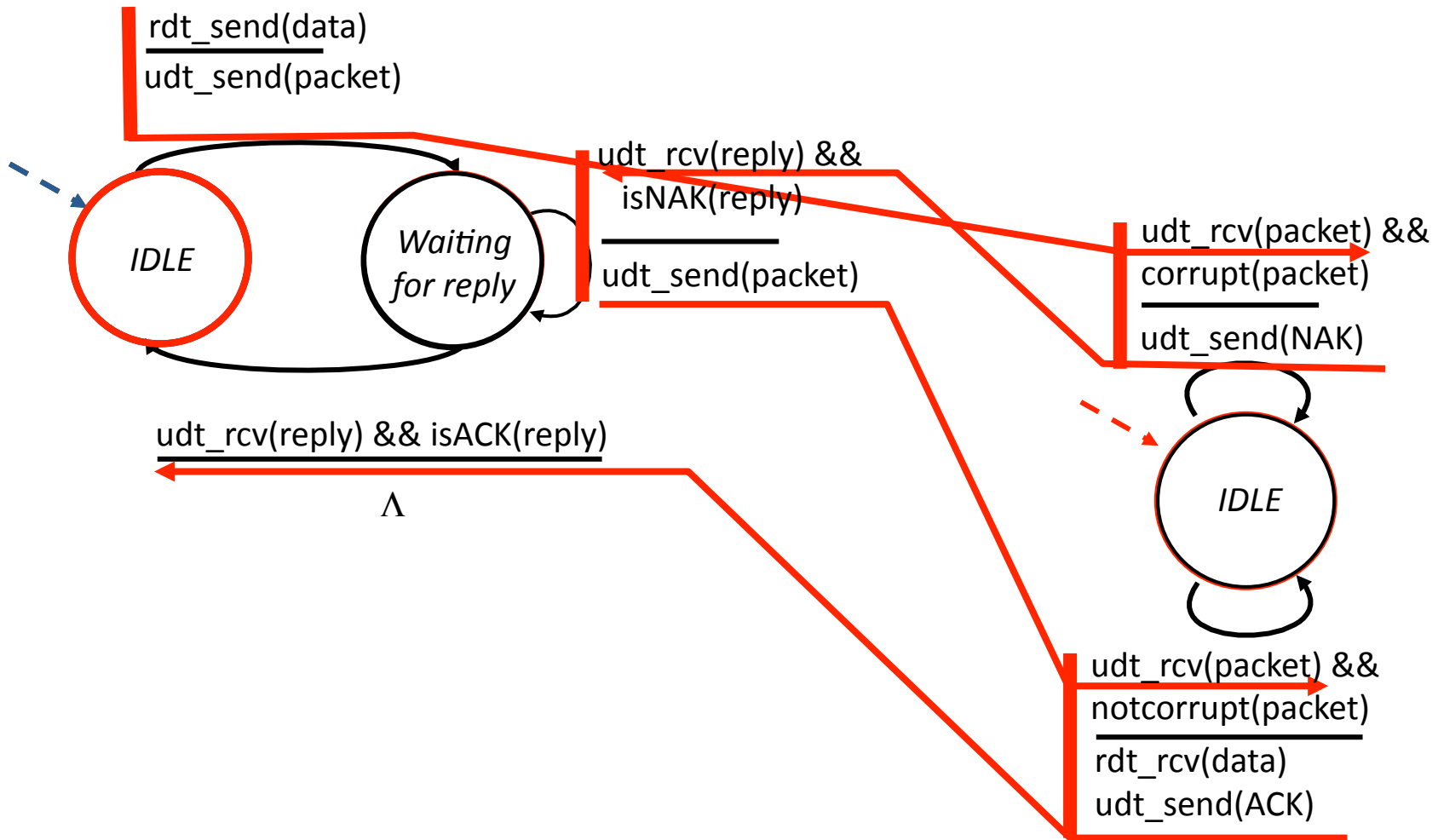
receiver



rdt2.0: operation with no errors



rdt2.0: error scenario



rdt2.0 has a fatal flaw!

What happens if ACK/NAK corrupted?

- sender doesn't know what happened at receiver!
- can't just retransmit: possible duplicate

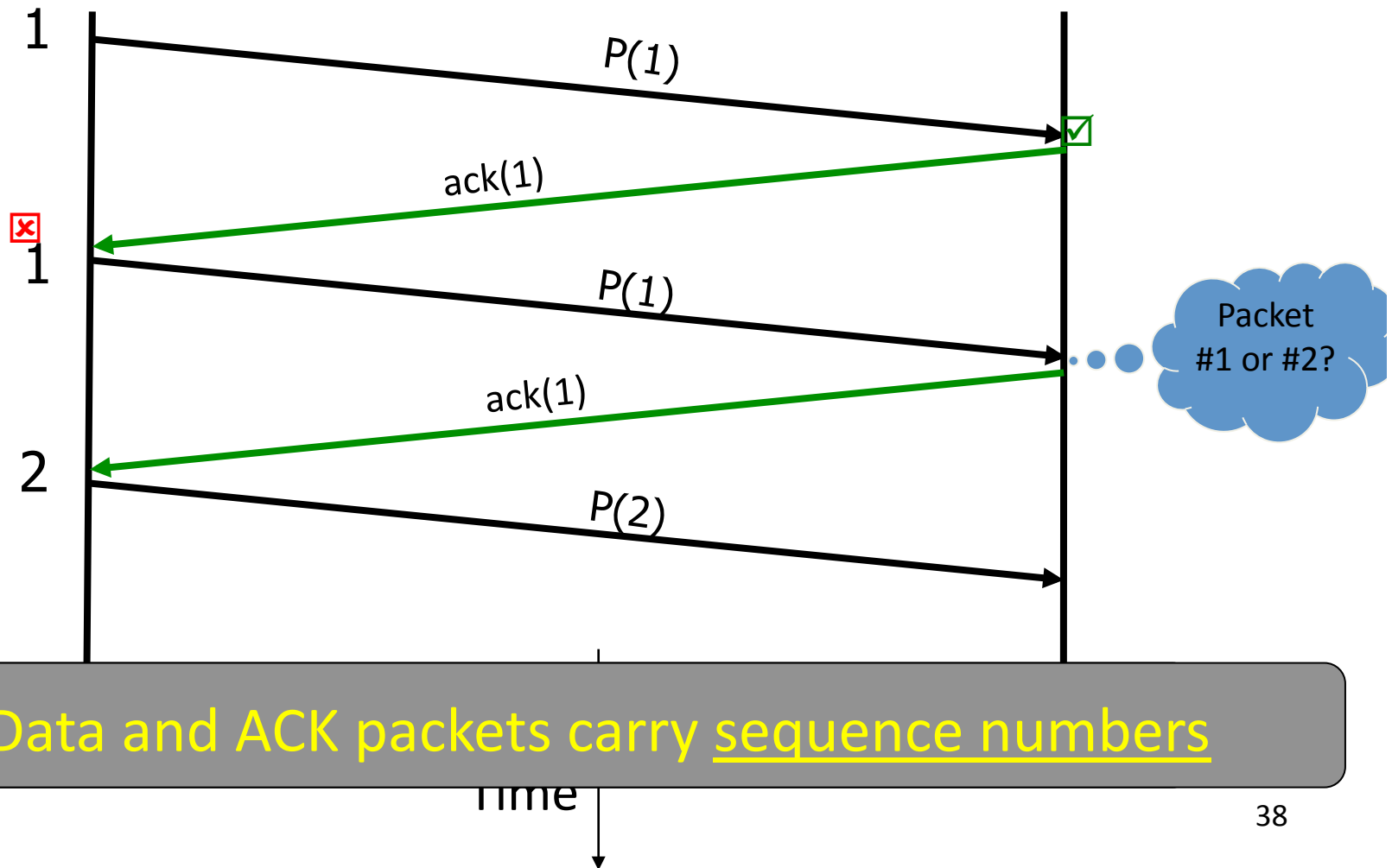
Handling duplicates:

- sender retransmits current packet if ACK/NAK garbled
- sender adds *sequence number* to each packet
- receiver discards (doesn't deliver) duplicate packet

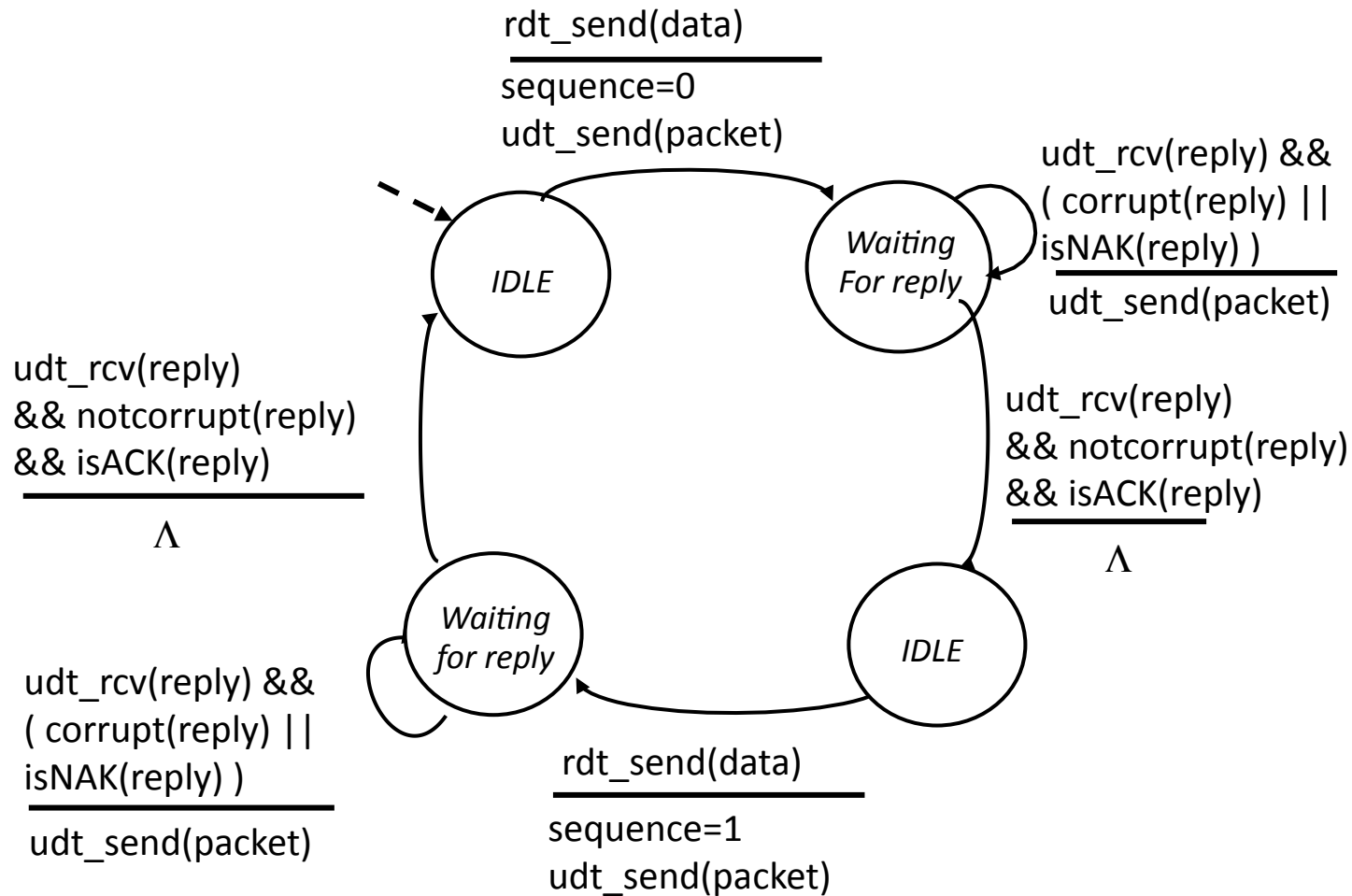
stop and wait

Sender sends one packet, then waits for receiver response

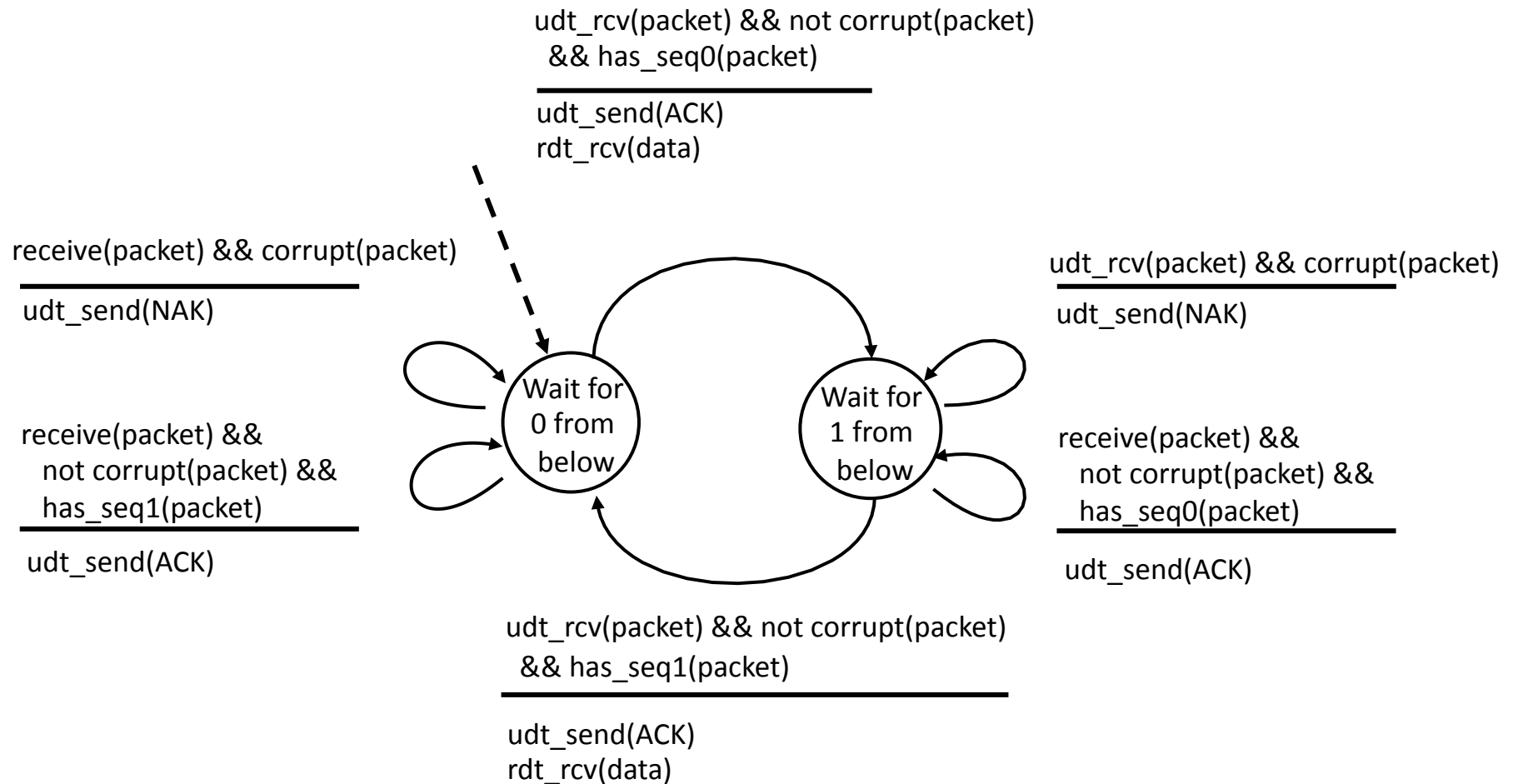
Dealing with Packet Corruption



rdt2.1: sender, handles garbled ACK/NAKs



rdt2.1: receiver, handles garbled ACK/NAKs



rdt2.1: discussion

Sender:

- seq # added to pkt
- two seq. #'s (0,1) will suffice. Why?
- must check if received ACK/NAK corrupted
- twice as many states
 - state must “remember” whether “current” pkt has a 0 or 1 sequence number

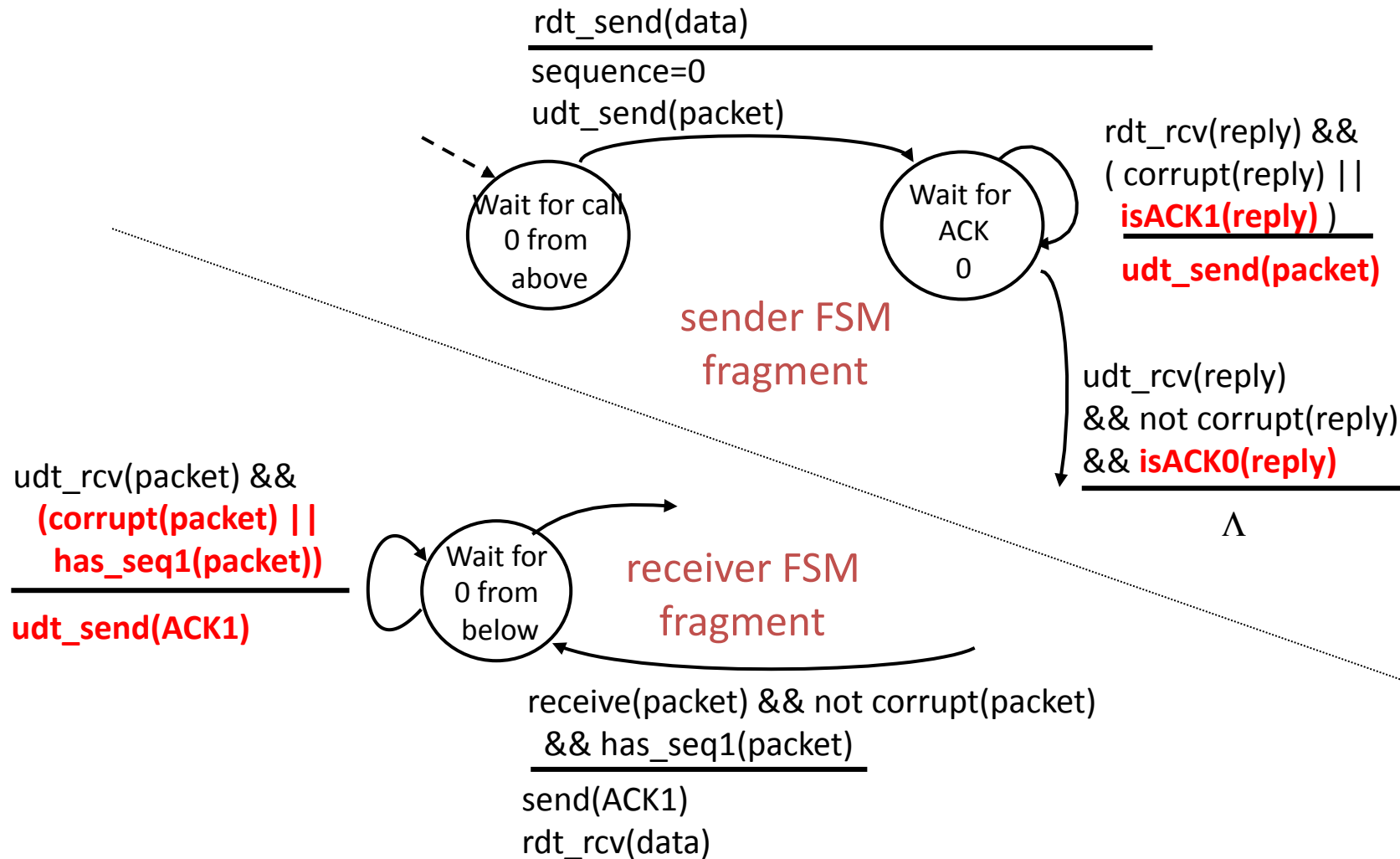
Receiver:

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

rdt2.2: a NAK-free protocol

- same functionality as rdt2.1, using ACKs only
- instead of NAK, receiver sends ACK for last pkt received OK
 - receiver must *explicitly* include seq # of pkt being ACKed
- duplicate ACK at sender results in same action as NAK:
retransmit current pkt

rdt2.2: sender, receiver fragments



rdt3.0: channels with errors *and* loss

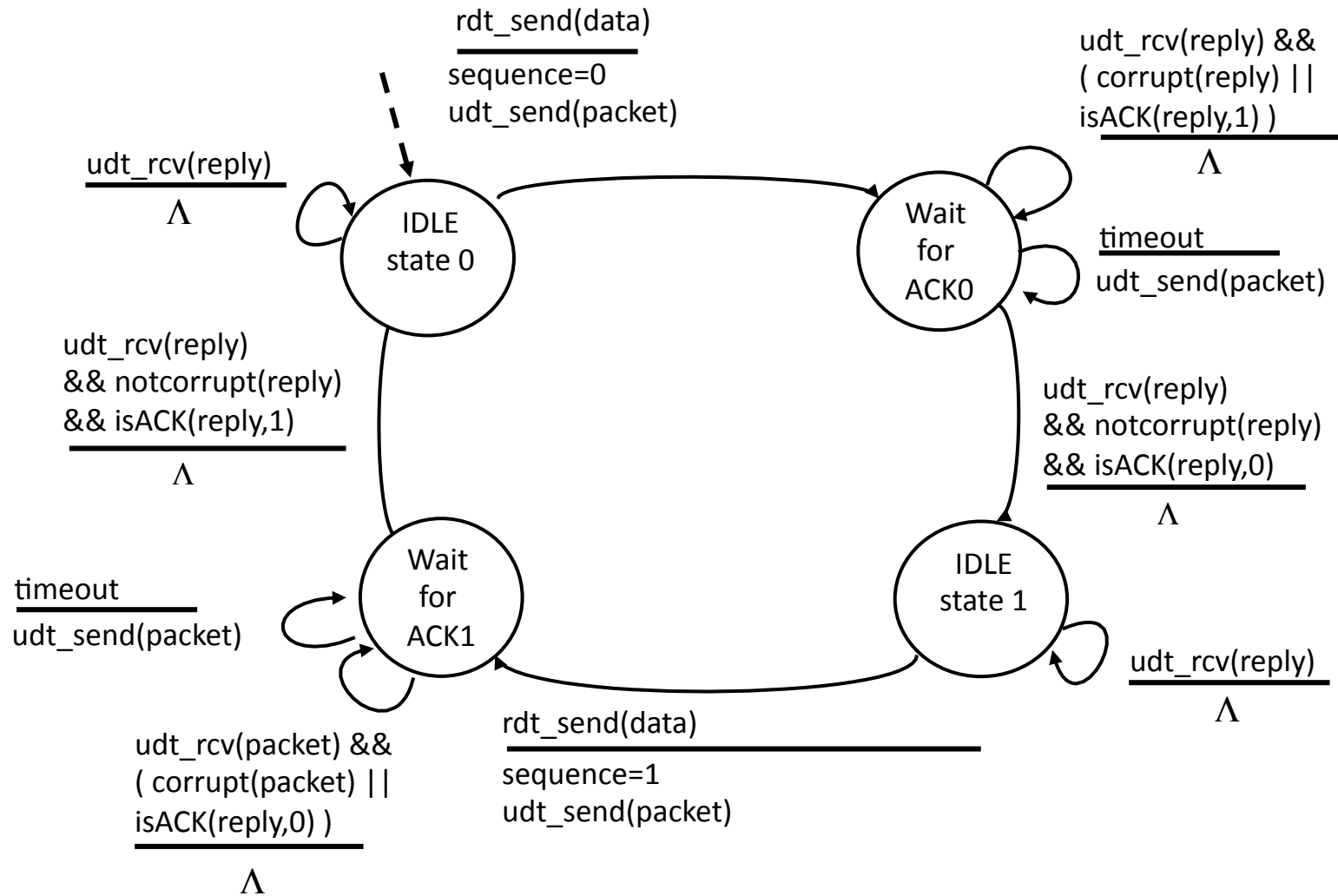
New assumption: underlying channel can also lose packets (data or ACKs)

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

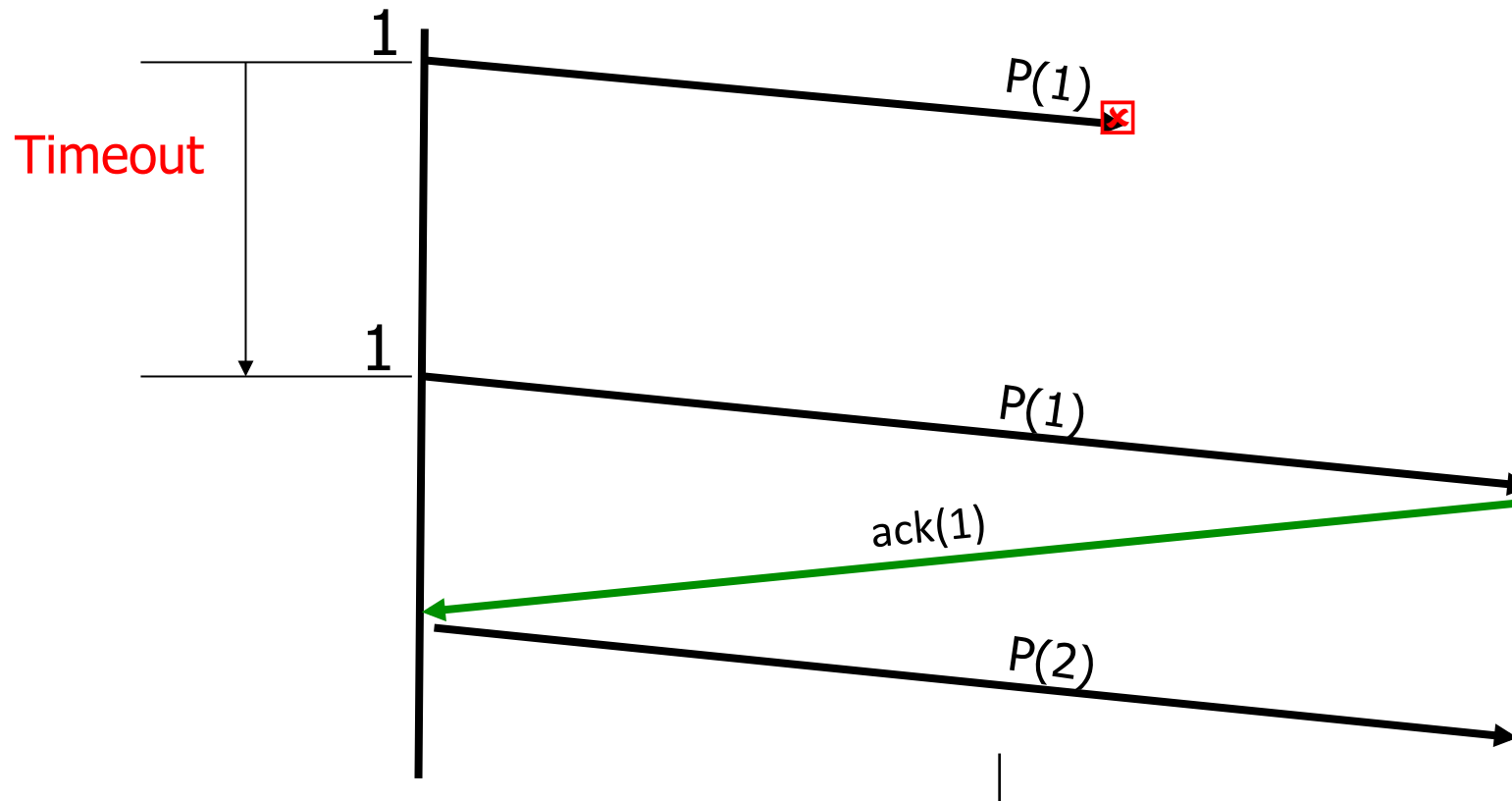
Approach: sender waits “reasonable” amount of time for ACK

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

rdt3.0 sender

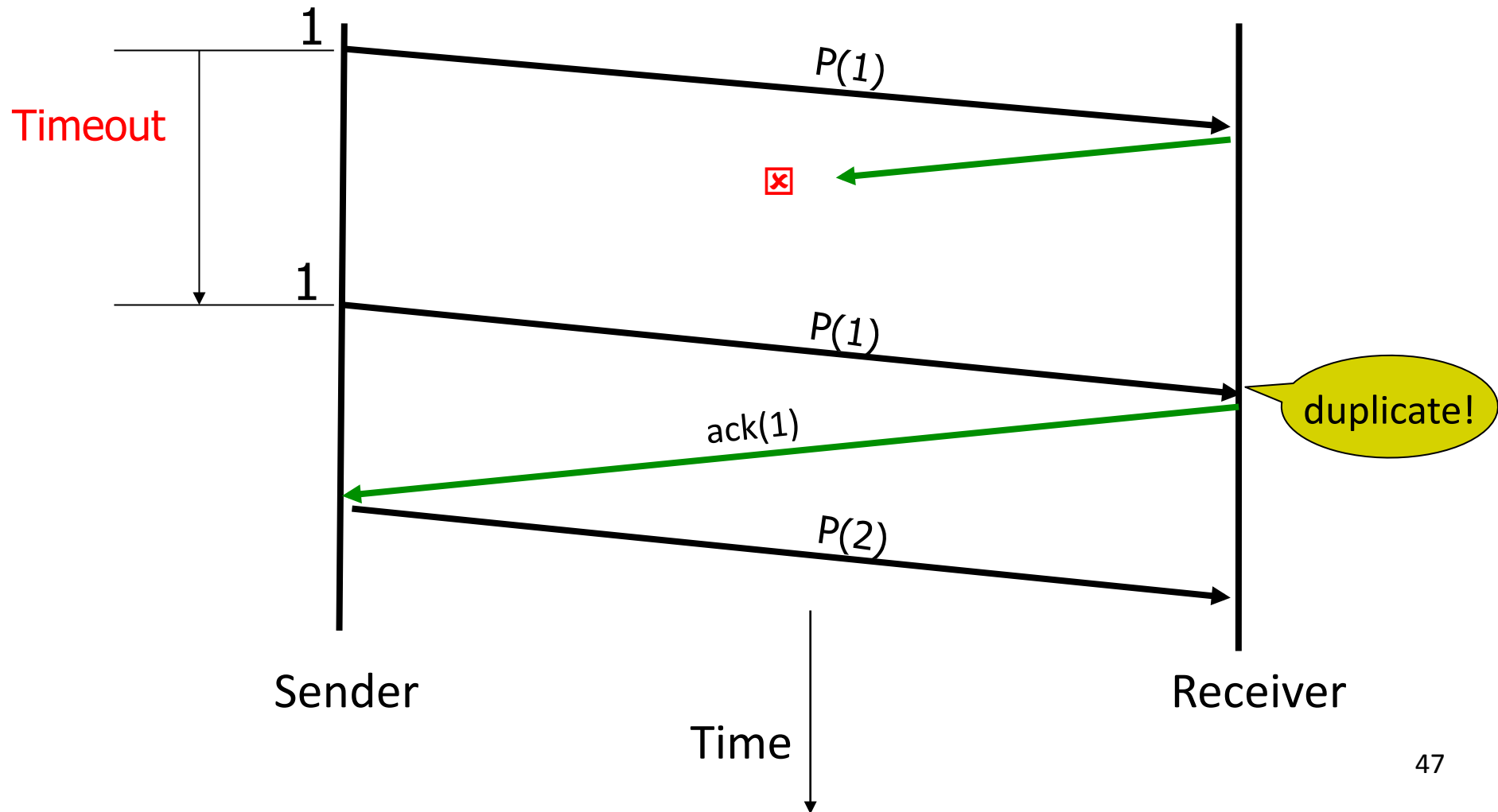


Dealing with Packet Loss

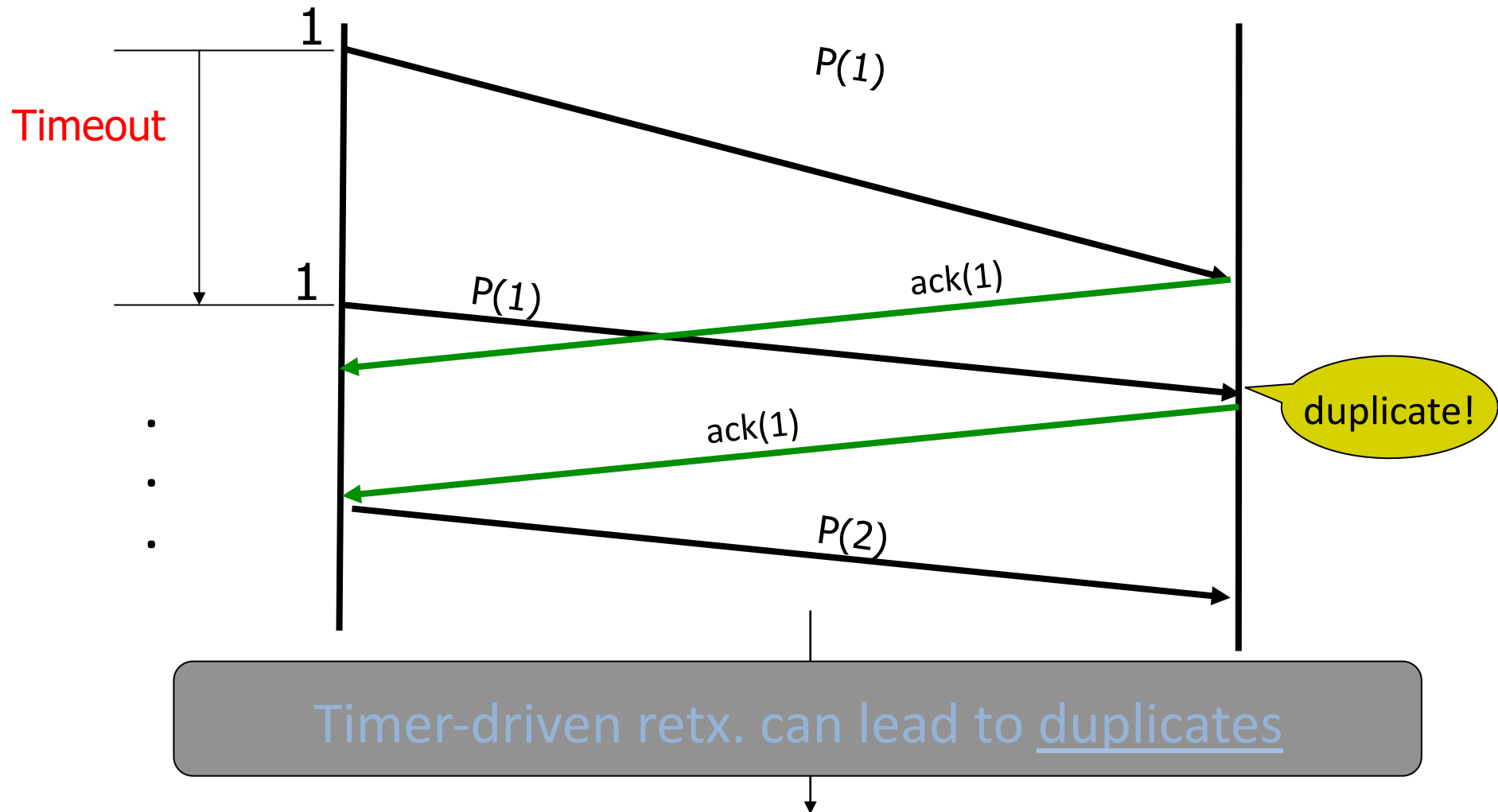


Timer-driven loss detection
Set timer when packet is sent; retransmit on timeout

Dealing with Packet Loss



Dealing with Packet Loss



Performance of rdt3.0

- rdt3.0 works, but performance stinks
- ex: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:

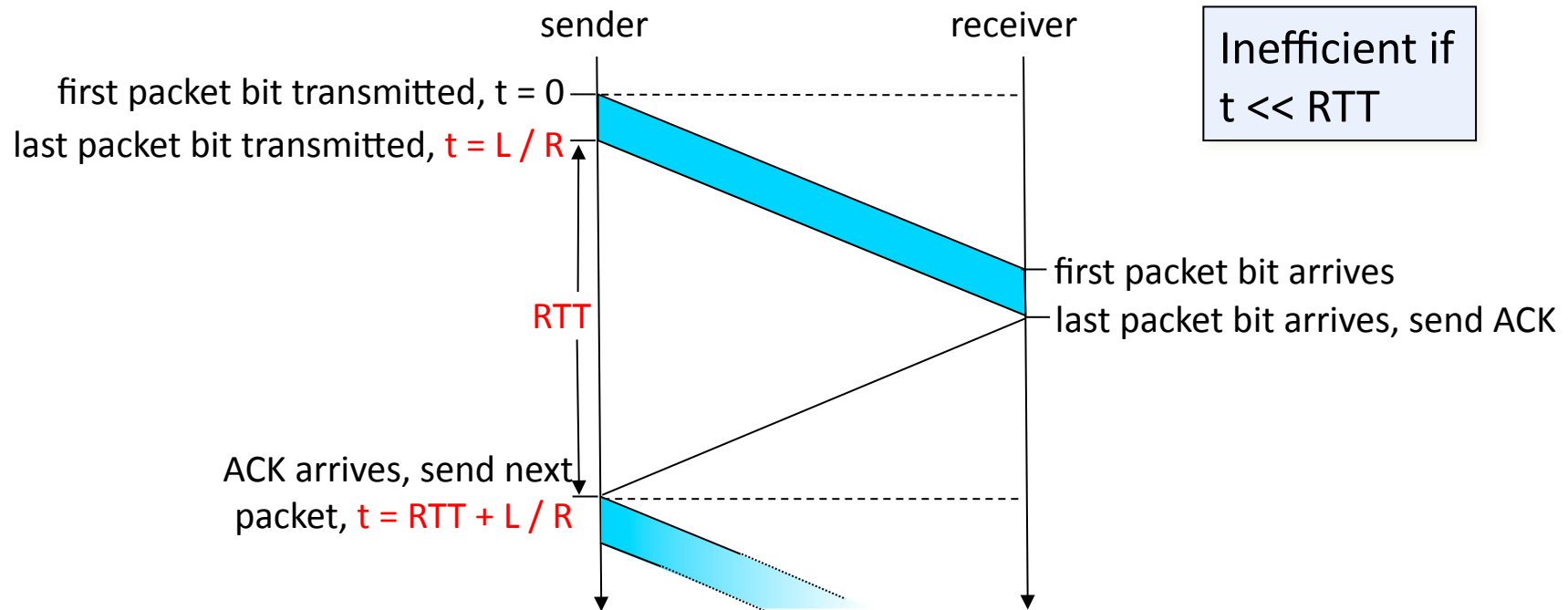
$$d_{trans} = \frac{L}{R} = \frac{8000\text{bits}}{10^9\text{bps}} = 8\text{microseconds}$$

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

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

- 1KB pkt every 30 msec -> 33kB/sec throughput over 1 Gbps link
- network protocol limits use of physical resources!

rdt3.0: stop-and-wait operation

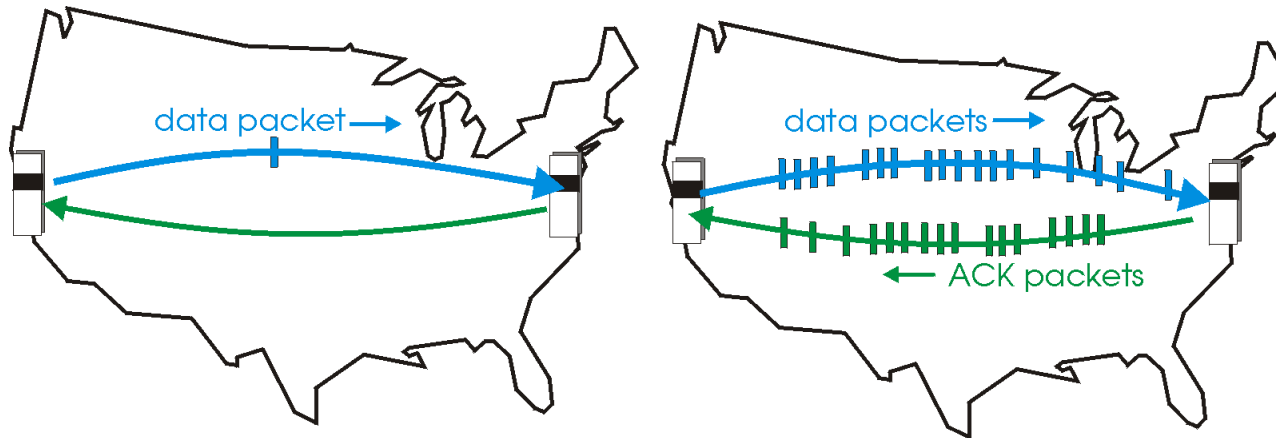


$$U_{\text{sender}} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.00027$$

Pipelined (Packet-Window) 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

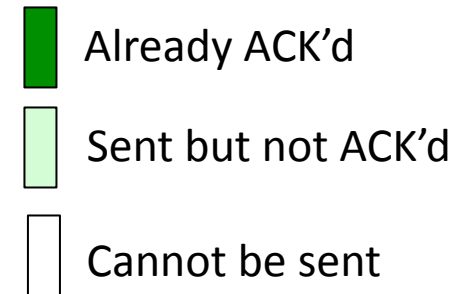
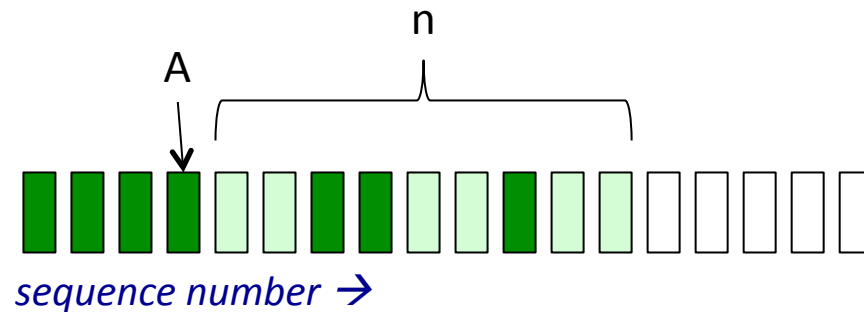
(b) a pipelined protocol in operation

A Sliding Packet Window

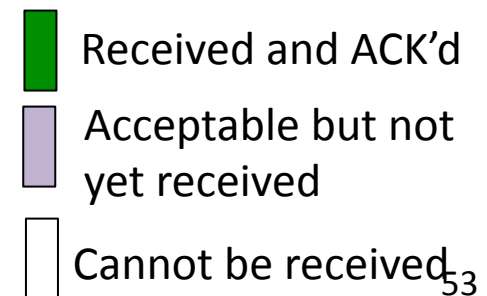
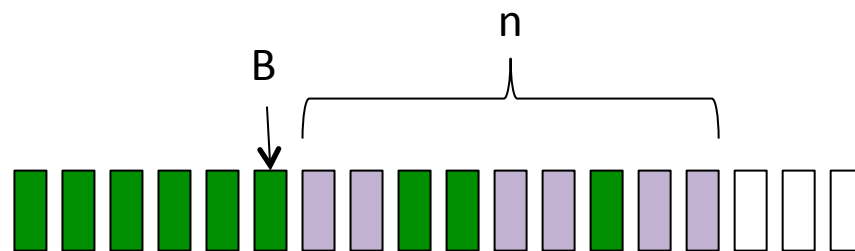
- **window** = set of adjacent sequence numbers
 - The size of the set is the **window size**; assume window size is n
- General idea: send up to n packets at a time
 - Sender can send packets in its window
 - Receiver can accept packets in its window
 - Window of acceptable packets “slides” on successful reception/acknowledgement

A Sliding Packet Window

- Let A be the **last ack'd packet of sender without gap**;
then window of sender = {A+1, A+2, ..., A+n}



- Let B be the **last received packet without gap** by receiver,
then window of receiver = {B+1, ..., B+n}

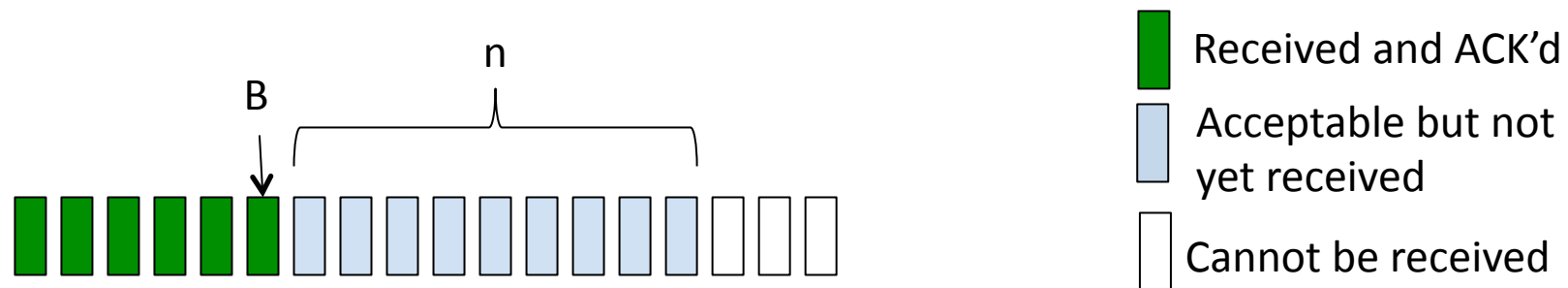


Acknowledgements w/ Sliding Window

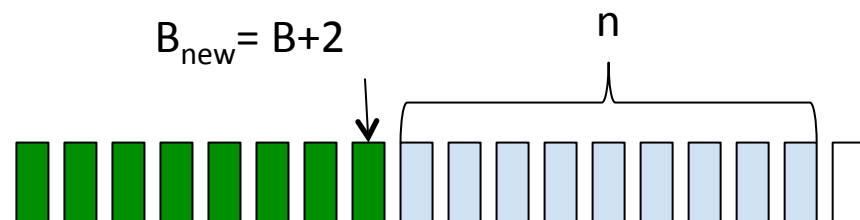
- Two common options
 - cumulative ACKs: ACK carries next in-order sequence number that the receiver expects

Cumulative Acknowledgements (1)

- At receiver



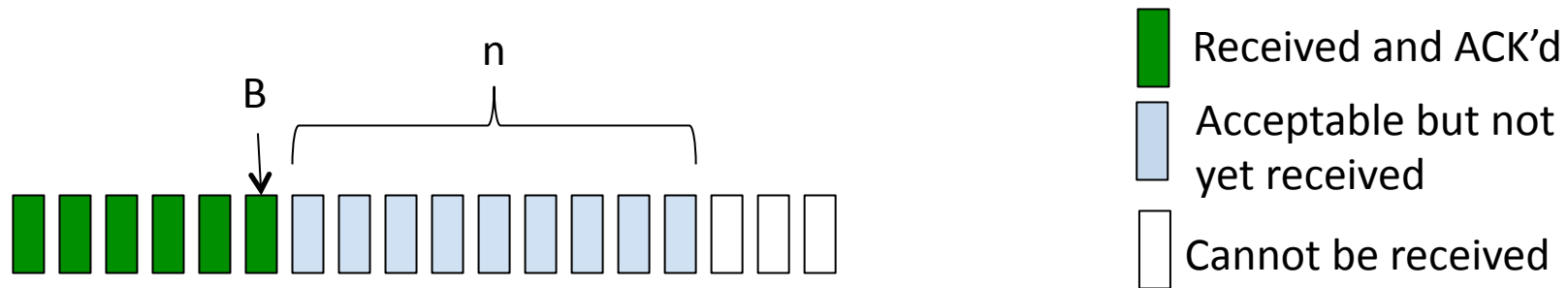
- After receiving $B+1$, $B+2$



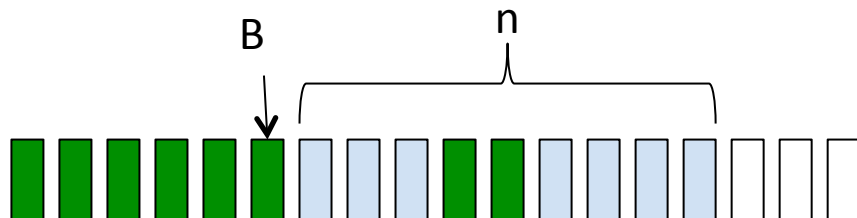
- Receiver sends $\text{ACK}(B_{\text{new}}+1)$

Cumulative Acknowledgements (2)

- At receiver



- After receiving B+4, B+5



- Receiver sends **ACK(B+1)**

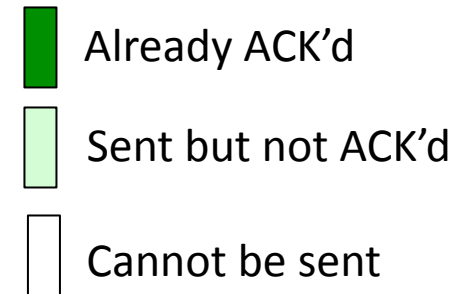
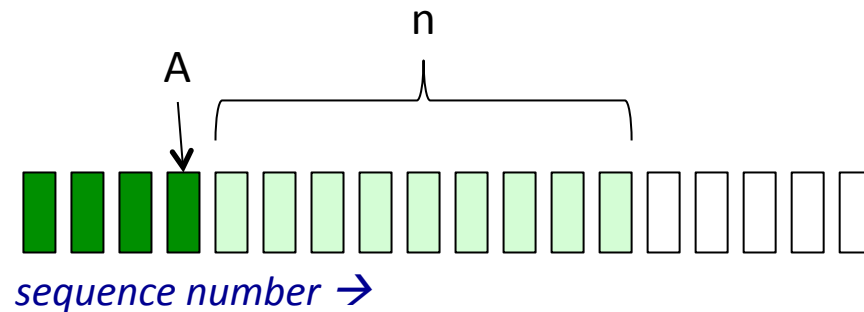
How do we recover?

Go-Back-N (GBN)

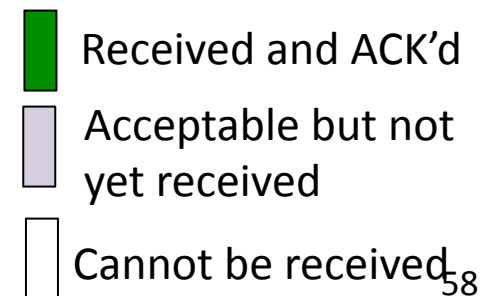
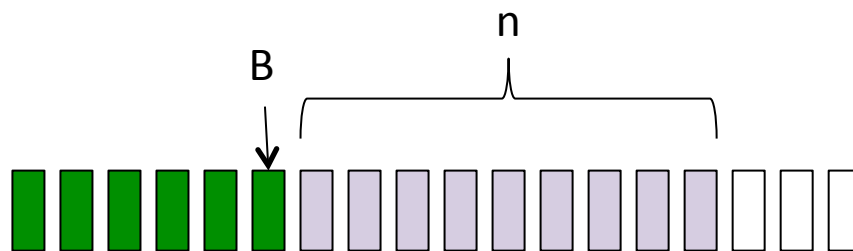
- Sender transmits up to n unacknowledged packets
- Receiver only accepts packets in order
 - discards out-of-order packets (i.e., packets other than $B+1$)
- Receiver uses **cumulative acknowledgements**
 - i.e., sequence# in ACK = next expected in-order sequence#
- Sender sets timer for 1st outstanding ack ($A+1$)
- If timeout, retransmit $A+1, \dots, A+n$

Sliding Window with GBN

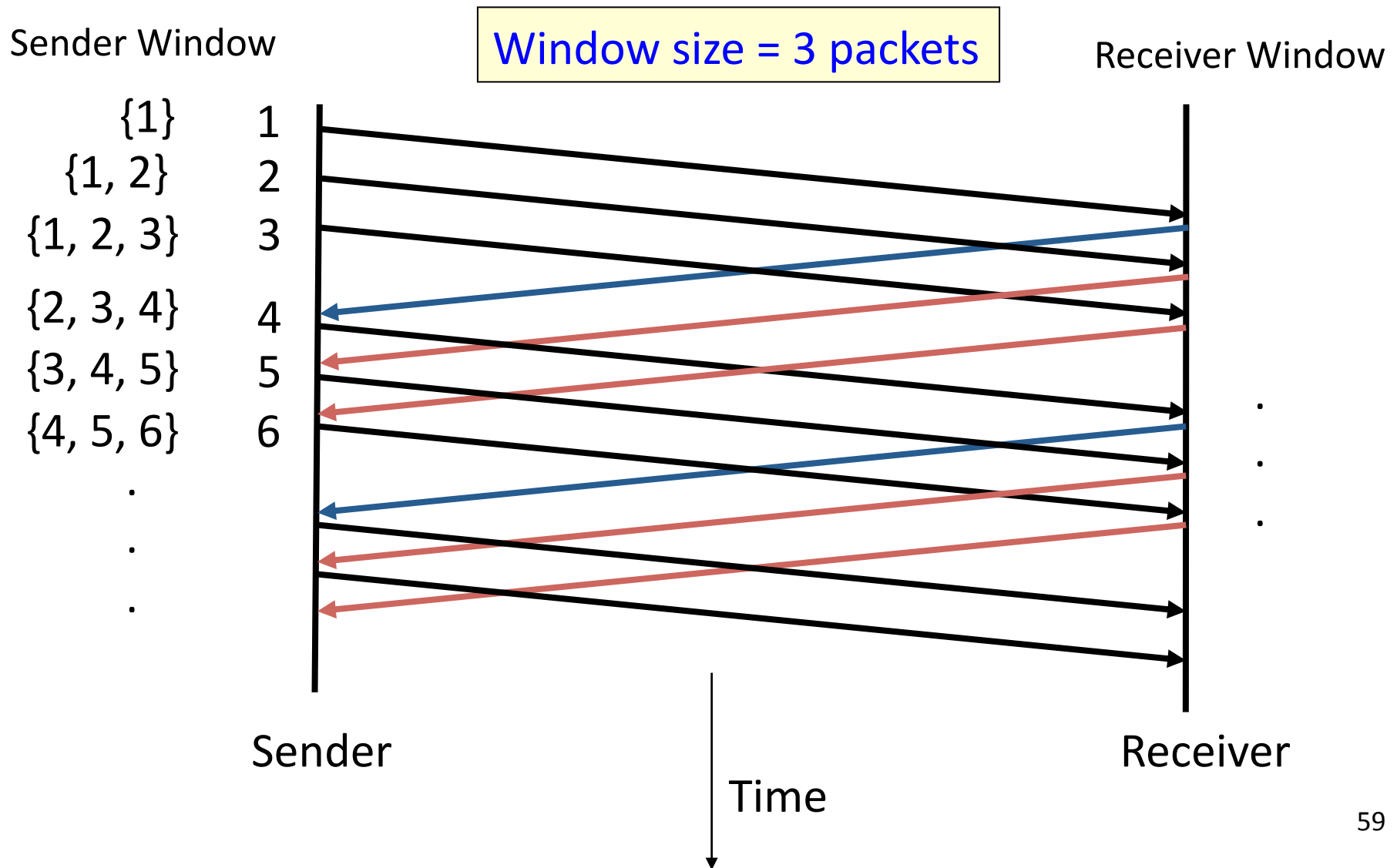
- Let A be the **last ack'd packet of sender without gap**;
then window of sender = {A+1, A+2, ..., A+n}



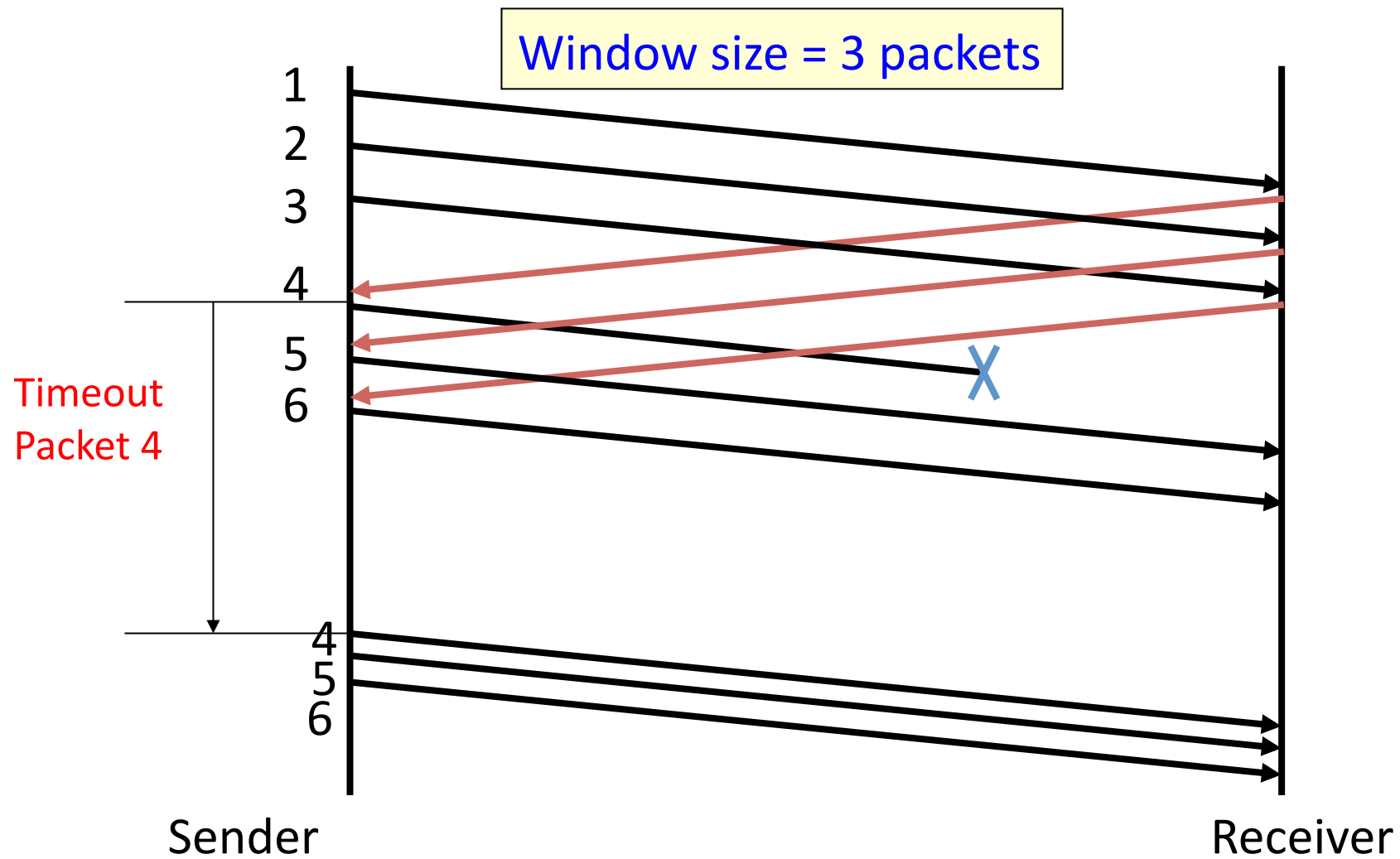
- Let B be the **last received packet without gap** by receiver,
then window of receiver = {B+1, ..., B+n}



GBN Example w/o Errors



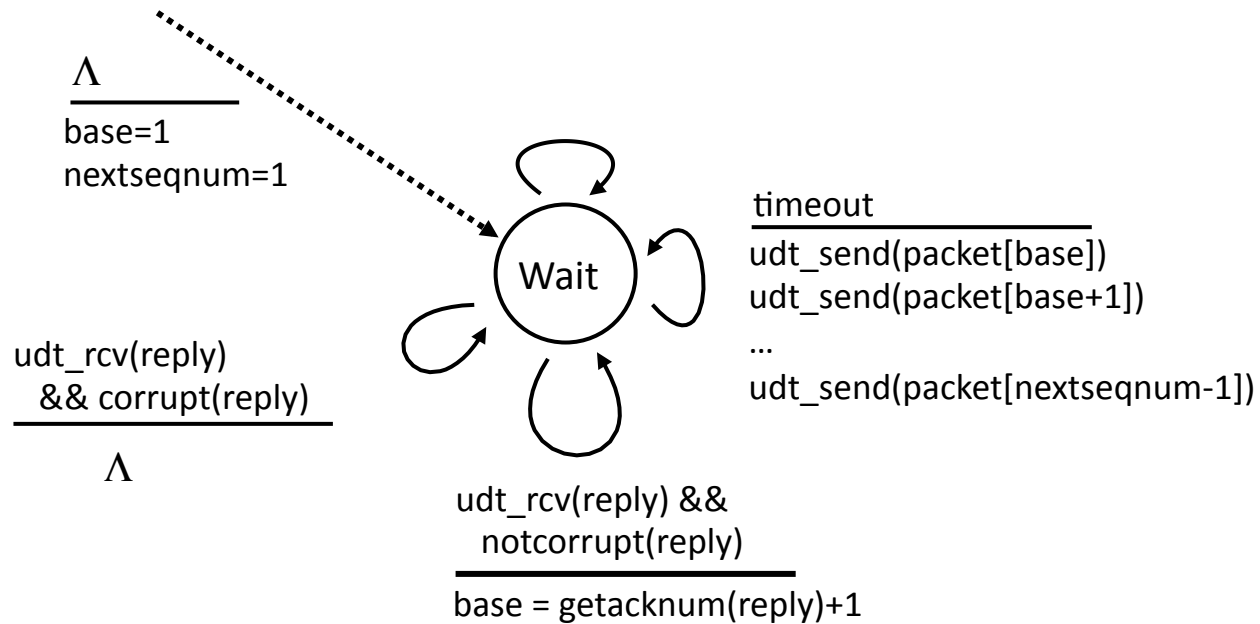
GBN Example with Errors



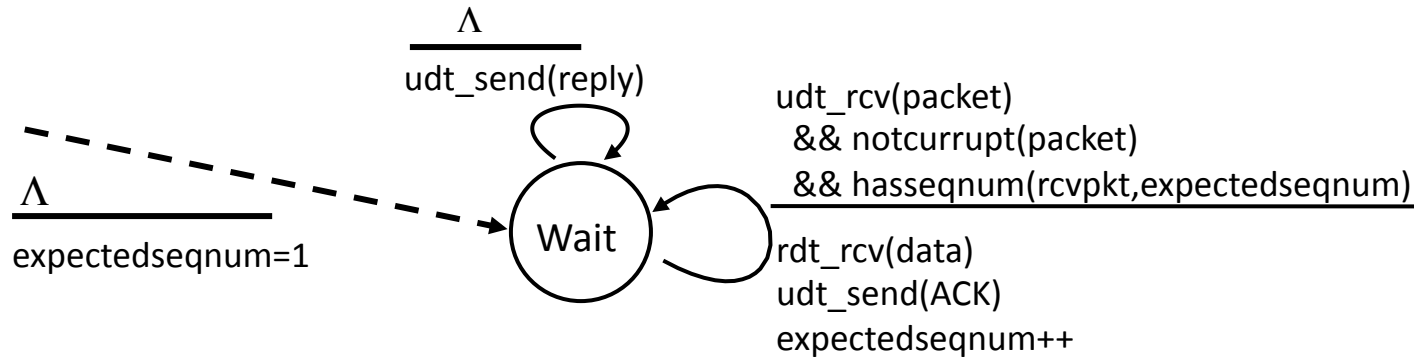
GBN: sender extended FSM

```

rdt_send(data)
if (nextseqnum < base+N) {
  udt_send(packet[nextseqnum])
  nextseqnum++
}
else
  refuse_data(data) Block?
  
```



GBN: receiver extended FSM



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

- may generate duplicate ACKs
- need only remember **expectedseqnum**
- out-of-order packet:
 - discard (don't buffer) -> **no receiver buffering!**
 - Re-ACK packet with highest in-order seq #

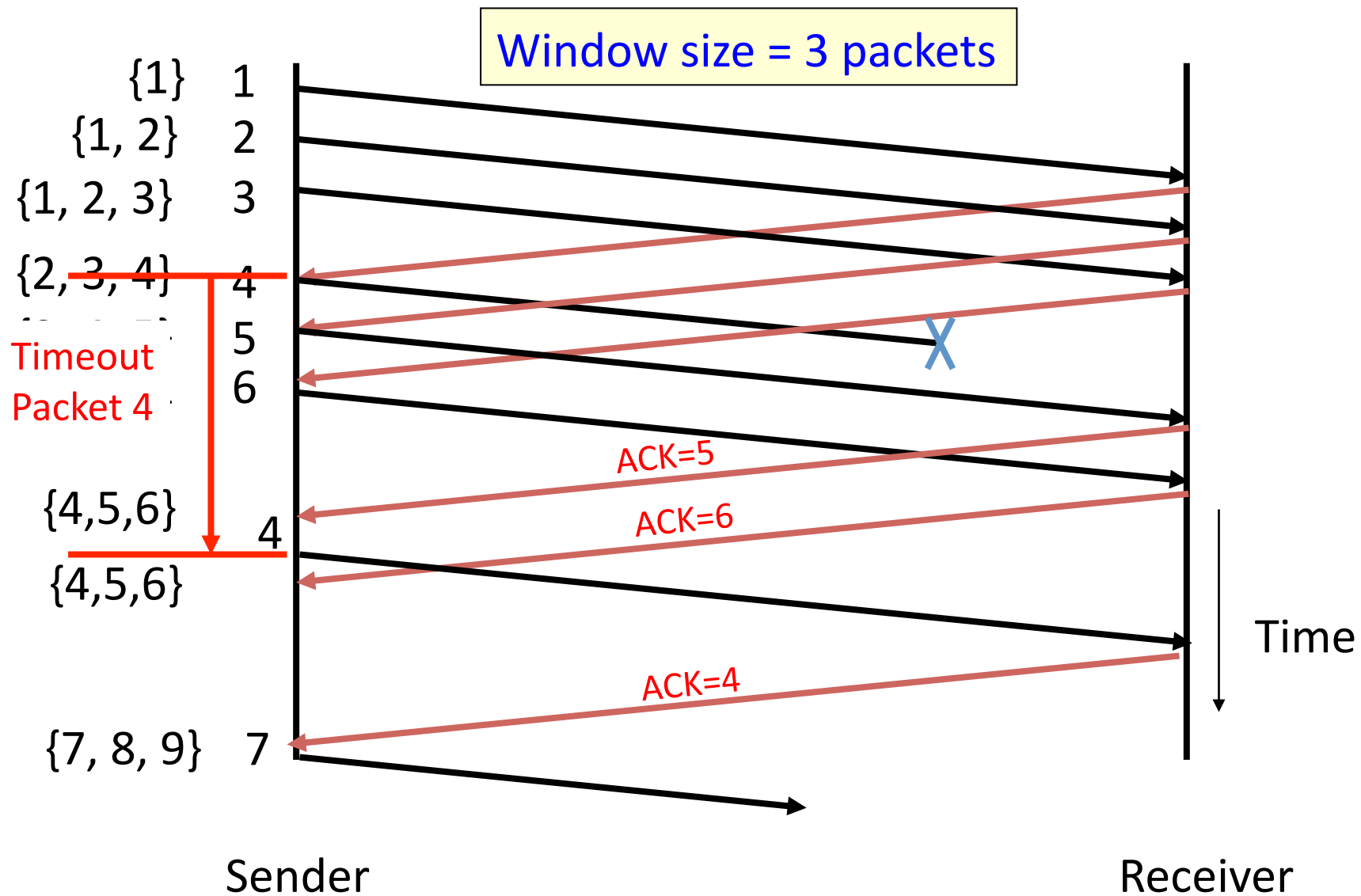
Acknowledgements w/ Sliding Window

- Two common options
 - cumulative ACKs: ACK carries next in-order sequence number the receiver expects
 - selective ACKs: ACK individually acknowledges correctly received packets
- Selective ACKs offer more precise information but require more complicated book-keeping
- Many variants that differ in implementation details

Selective Repeat (SR)

- Sender: transmit up to n unacknowledged packets
- Assume packet k is lost, $k+1$ is not
- Receiver: indicates packet $k+1$ correctly received
- Sender: retransmit only packet k on timeout
- Efficient in retransmissions but complex book-keeping
 - need a timer per packet

SR Example with Errors



Observations

- With sliding windows, it is possible to fully utilize a link, provided the window size is large enough.
Throughput is $\sim (n/RTT)$
 - Stop & Wait is like $n = 1$.
- Sender has to buffer all unacknowledged packets, because they may require retransmission
- Receiver may be able to accept out-of-order packets, but only up to its buffer limits
- Implementation complexity depends on protocol details (GBN vs. SR)

Recap: components of a solution

- Checksums (for error detection)
- Timers (for loss detection)
- Acknowledgments
 - cumulative
 - selective
- Sequence numbers (duplicates, windows)
- Sliding Windows (for efficiency)

- Reliability protocols use the above to decide when and what to retransmit or acknowledge

What does TCP do?

Most of our previous tricks + a few differences

- Sequence numbers are byte offsets
- Sender and receiver maintain a sliding window
- Receiver sends cumulative acknowledgements (like GBN)
- Sender maintains a single retx. timer
- Receivers do not drop out-of-sequence packets (like SR)
- Introduces **fast retransmit** : optimization that uses duplicate ACKs to trigger early retx (next time)
- Introduces timeout estimation algorithms (next time)