

# CS 3205 COMPUTER NETWORKS

**JAN-MAY 2020**

**LECTURE 4,5,6: 23<sup>RD</sup> 27<sup>TH</sup> 28<sup>TH</sup> JAN 2020**

Text book and section(s) covered in this lecture:

Book Kurose and Ross – Sections 5.1, 5.2, 3.4

Book Patterson and Davie – Sections 2.1, 2.2, 2.3, 2.4, 2.5,

# Link Layer

- ❖ Second from bottom of the stack
- ❖ Nodes connected via Links
- ❖ Links which enable connection need a physical media
  - Wired, Wireless
  - LAN, Enterprise, MAN, WAN.
- ❖ Five Additional Problems
  - Encoding
  - Delineating, Framing
  - Error Detection, Error Correction
  - Reliable Delivery – making link as reliable one
  - Medium Access Control
- ❖ Half Duplex, Full Duplex

# Link Capacity and Shannon-Hartley Theorem (Upper / Theoretical bound)

- ❖  $C = B * \log_2 ( 1 + S / N )$
- ❖ C is the capacity of the link
- ❖ B is the bandwidth (in frequency)
- ❖ S is the signal power
- ❖ N is the noise power
  
- ❖ Signal to Noise Ratio (SNR) is usually denoted in decibals (dB).
- ❖  $SNR = 10 * \log_{10}(S/N)$ . If signal power is 1000 times the Noise power, then  $SNR = 30dB$

# Link Capacity and Shannon-Hartley Theorem (Contd)

- ❖ Capacity of voice phone line:
- ❖  $B = (3300 \text{ Hz} - 300 \text{ Hz}) = 3000 \text{ Hz}$
- ❖ Assume 30 dB SNR,
- ❖ Then  $C = 3000 * \log_2 (1 + 1000)$
- ❖ Which approximates to 30 kbps
  
- ❖ Shannon-Hartley theorem is applicable to all media.
- ❖ High capacity achieved: high bandwidth or high SNR, or both.
- ❖ Even if both are high, it still depends on the channel encoding schemes to achieve theoretical limits.

# Data in Links

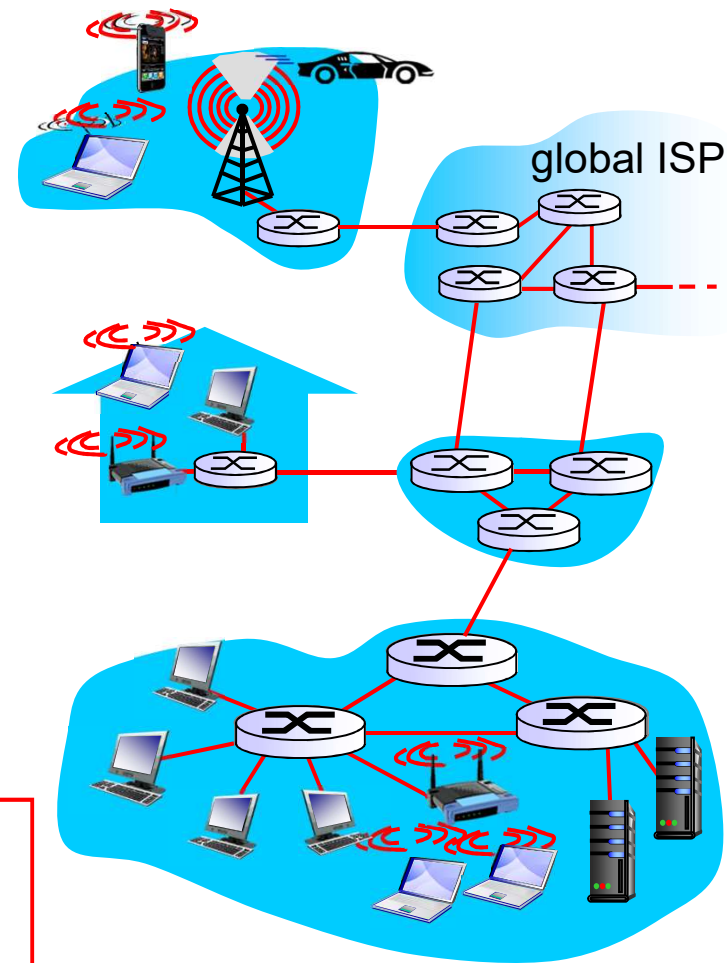
- ❖ Physical medium carry signals
- ❖ Binary data is encoded onto the signals.
- ❖ Encoding binary data onto electro magnetic signals is very challenging.
- ❖ Two parts:
- ❖ (Lower layer) How signal is represented, modulated. Binary coding, Amplitude modulation, Frequency modulation, other types..
- ❖ (Upper layer) How it is encoded, i.e., value is represented.
- ❖ Lower layer signal representation, optimization done by Electrical Communication Specialists.

# Link layer: introduction

## *terminology:*

- ❖ hosts and routers: **nodes**
- ❖ communication channels that connect adjacent nodes along communication path: **links**
  - wired links
  - wireless links
  - LANs
- ❖ layer-2 packet: **frame**, encapsulates datagram

*data-link layer* has responsibility of transferring datagram from one node to *physically adjacent* node over a link



# Link layer: context

- ❖ datagram transferred by different link protocols over different links:
  - e.g., Ethernet on first link, frame relay on intermediate links, 802.11 on last link
- ❖ each link protocol provides different services
  - e.g., may or may not provide rdt over link

## *transportation analogy:*

- ❖ trip from IIT-M to UC-Berkley
  - Uber: Campus to Airport
  - plane: Chennai to SFO – 1 or 2 hops (Emirates - 1 hop - Dubai, Singapore airlines - 2 hops - Singapore, Narita (Tokyo))
  - BART train: SFO to UC-Berkley
- ❖ Student = **datagram**
- ❖ transport segment = **communication link**
- ❖ transportation mode = **link layer protocol**
- ❖ travel agent = **routing algorithm**

# Link layer services

## ❖ *framing, link access:*

- encapsulate datagram into frame, adding header, trailer
- channel access if shared medium
- “MAC” addresses used in frame headers to identify source, dest
  - different from IP address!

## ❖ *reliable delivery between adjacent nodes*

- seldom used on low bit-error link (fiber, some twisted pair)
- wireless links: high error rates
  - *Q*: why both link-level and end-end reliability?



# Link layer services (more)

## ❖ *flow control:*

- pacing between adjacent sending and receiving nodes

## ❖ *error detection:*

- errors caused by signal attenuation, noise.
- receiver detects presence of errors:
  - signals sender for retransmission or drops frame

## ❖ *error correction:*

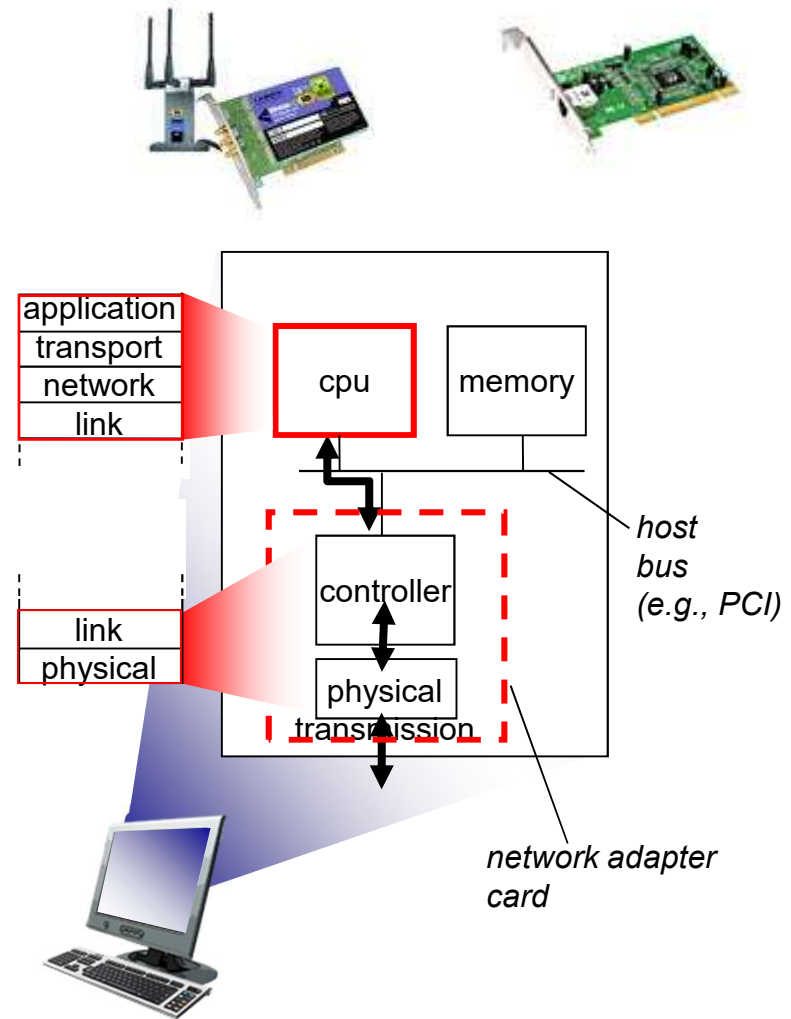
- receiver identifies *and corrects* bit error(s) without resorting to retransmission

## ❖ *half-duplex and full-duplex*

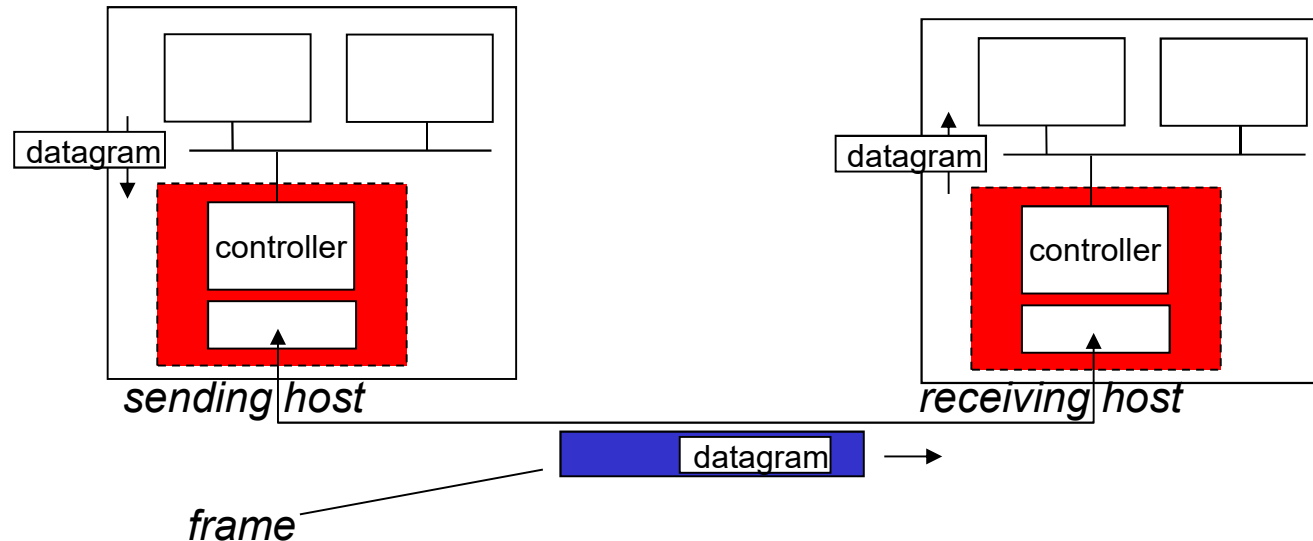
- with half duplex, nodes at both ends of link can transmit, but not at same time

# Where is the link layer implemented?

- ❖ in each and every host
- ❖ link layer implemented in “adaptor” (aka *network interface card* NIC) or on a chip
  - Ethernet card, 802.11 card; Ethernet chipset
  - implements link, physical layer
- ❖ attaches into host's system buses
- ❖ combination of hardware, software, firmware



# Adaptors communicating



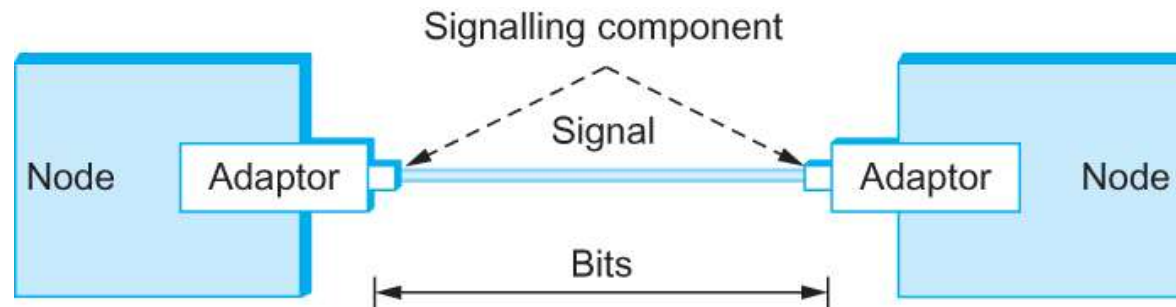
## ❖ sending side:

- encapsulates datagram in frame
- adds error checking bits, reliable data transfer, flow control, etc.

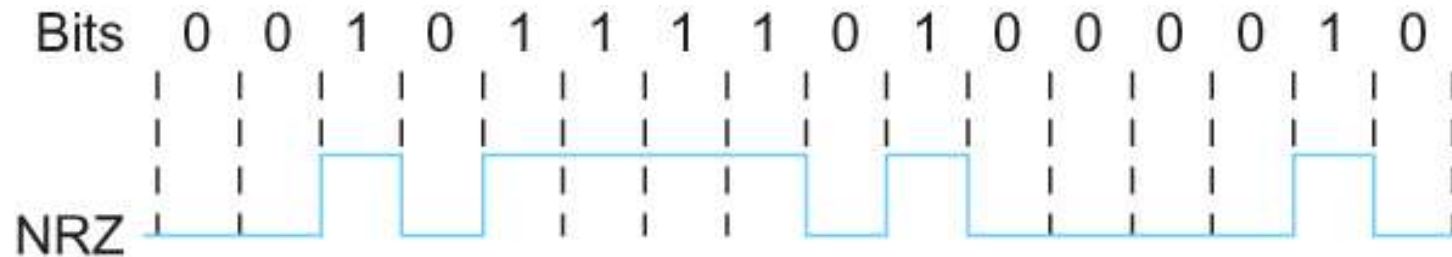
## ❖ receiving side

- looks for errors, rdt, flow control, etc
- extracts datagram, passes to upper layer at receiving side

# Encoding



Signals travel between signaling components; bits flow between adaptors



NRZ encoding of a bit stream

NRZ – Non Return to Zero

# Encoding

## ❖ Problem with NRZ

### ■ Baseline wander

- The receiver keeps an average of the signals it has seen so far
- Uses the average to distinguish between low and high signal
- When a signal is significantly low than the average, it is 0, else it is 1
- Too many consecutive 0's and 1's cause this average to change, making it difficult to detect

# Encoding

## ❖ Problem with NRZ

### ■ Clock recovery

- Frequent transition from high to low or vice versa are necessary to enable clock recovery
- Both the sending and decoding process is driven by a clock
- Every clock cycle, the sender transmits a bit and the receiver recovers a bit
- The sender and receiver have to be precisely synchronized

# Encoding

## ❖ NRZI

- Non Return to Zero Inverted
- Sender makes a transition from the current signal to encode 1 and stay at the current signal to encode 0
- Solves for consecutive 1's

# Encoding

## ❖ Manchester encoding

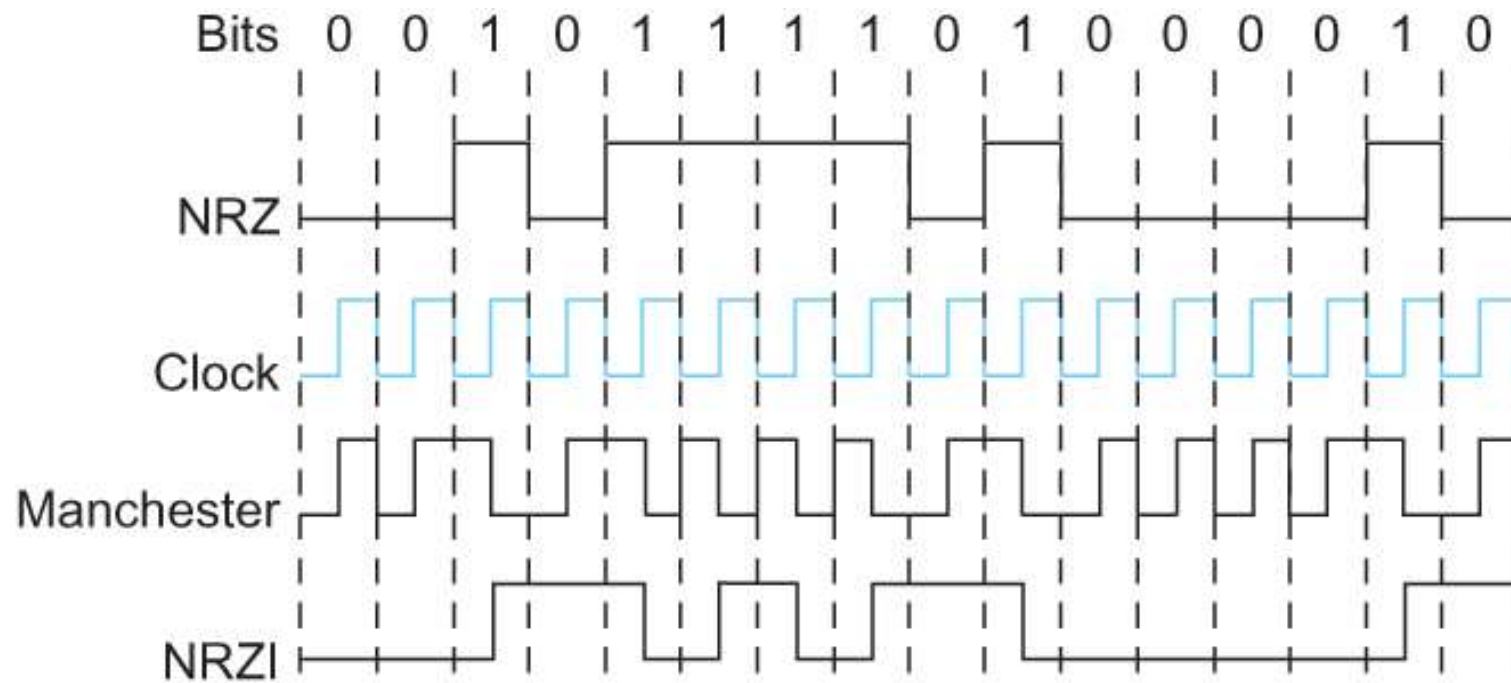
- Merging the clock with signal by transmitting Ex-OR of the NRZ encoded data and the clock
- Clock is an internal signal that alternates from low to high, a low/high pair is considered as one clock cycle
- In Manchester encoding
  - 0: low  $\rightarrow$  high transition
  - 1: high  $\rightarrow$  low transition



# Encoding

- ❖ Problem with Manchester encoding
  - Doubles the rate at which the signal transitions are made on the link
    - Which means the receiver has half of the time to detect each pulse of the signal
  - The rate at which the signal changes is called the link's baud rate
  - In Manchester the bit rate is half the baud rate

# Encoding



Different encoding strategies

# Encoding

## ❖ 4B/5B encoding

- Insert extra bits into bit stream so as to break up the long sequence of 0's and 1's
- Every 4-bits of actual data are encoded in a 5-bit code that is transmitted to the receiver
- 5-bit codes are selected in such a way that each one has no more than one leading 0(zero) and no more than two trailing 0's.
- No pair of 5-bit codes results in more than three consecutive 0's

# Encoding

## ❖ 4B/5B encoding

0000 → 11110

0001 → 01001

0010 → 10100

..

..

1111 → 11101

16 left

11111 – when the line is idle

00000 – when the line is dead

00100 – to mean halt

13 left : 7 invalid, 6 for various  
control signals

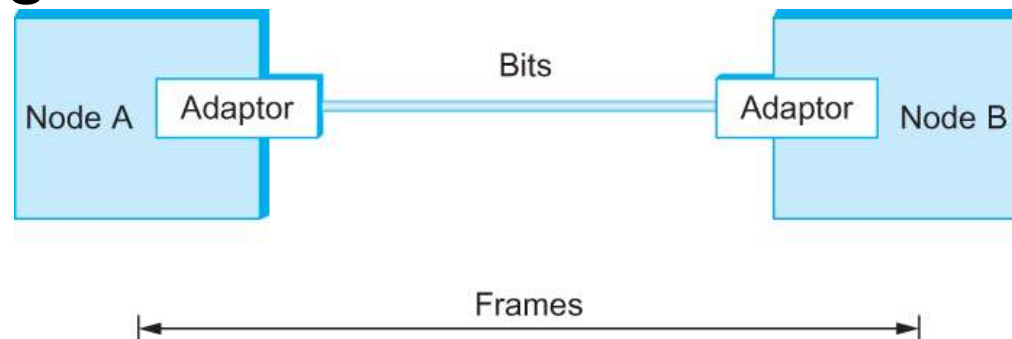
# Framing

Section 2.3

Computer Networks 5<sup>th</sup> Edn: Patterson and Davie

# Framing

- ❖ We are focusing on packet-switched networks, which means that blocks of data (called *frames* at this level), not bit streams, are exchanged between nodes.
- ❖ It is the network adaptor that enables the nodes to exchange frames.



Bits flow between adaptors, frames between hosts

# Framing

- ❖ When node A wishes to transmit a frame to node B, it tells its adaptor to transmit a frame from the node's memory. This results in a sequence of bits being sent over the link.
- ❖ The adaptor on node B then collects together the sequence of bits arriving on the link and deposits the corresponding frame in B's memory.
- ❖ Recognizing exactly what set of bits constitute a frame—that is, determining where the frame begins and ends—is the central challenge faced by the adaptor

# Framing

## ❖ Byte-oriented Protocols

- To view each frame as a collection of bytes (characters) rather than bits
- BISYNC (Binary Synchronous Communication) Protocol
  - Developed by IBM (late 1960)
- DDCMP (Digital Data Communication Protocol)
  - Used in DECNet
- Sentinel Approach (body guard – convoys)

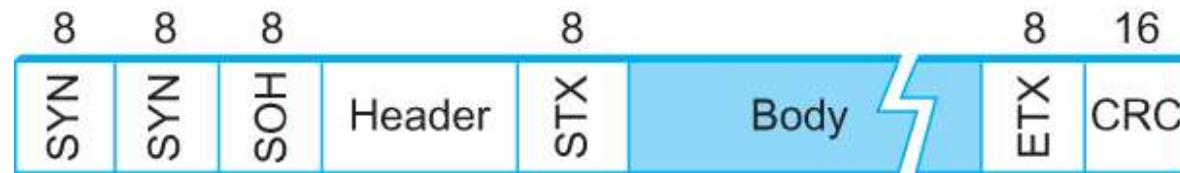


# Framing

## ❖ BISYNC – sentinel approach

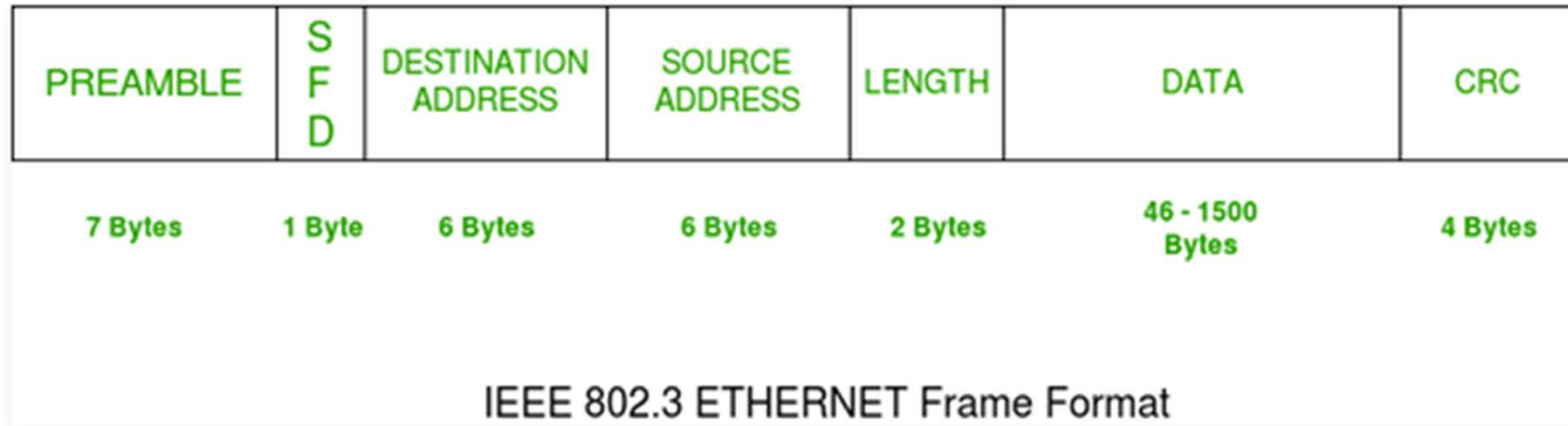
- Frames transmitted beginning with leftmost field
- Beginning of a frame is denoted by sending a special SYN (synchronize) character
- Data portion of the frame is contained between special sentinel character STX (start of text) and ETX (end of text)
- SOH : Start of Header
- DLE : Data Link Escape
- Character Stuffing
- CRC: Cyclic Redundancy Check

# Framing



BISYNC Frame Format

# Ethernet Frame



Preamble – Alternative 0s and 1s

SFD – Start of the frame delimiter - 10101011

CRC – Cyclic Redundancy Check

<https://www.geeksforgeeks.org/computer-network-ethernet-frame-format/>

<https://www.electronics-notes.com/articles/connectivity/ethernet-ieee-802-3/basics-tutorial.php>

[http://media.klinkmann.lv/pdf/lv/exfo/Exfo\\_Ethernet\\_Reference\\_Guide\\_en.pdf](http://media.klinkmann.lv/pdf/lv/exfo/Exfo_Ethernet_Reference_Guide_en.pdf)

# Inter Frame Gap

## Ethernet [\[ edit \]](#)

[Ethernet](#) devices must allow a minimum idle period between transmission of [Ethernet packets](#) known as the **interpacket gap (IPG)**, **interframe spacing**, or **interframe gap (IFG)**.<sup>[1]</sup> A brief recovery time between packets allows devices to prepare for reception of the next packet. While some physical layer variants literally transmit nothing during the idle period, most modern ones transmit a constant signal and send an idle pattern. The standard minimum interpacket gap for transmission is 96 bit times (the time it takes to transmit 96 bits of data on the medium), which is

- 9.6  $\mu$ s for 10 Mbit/s Ethernet,
- 0.96  $\mu$ s for 100 Mbit/s ([Fast](#)) Ethernet,
- 96 ns for [Gigabit Ethernet](#),
- 38.4 ns for [2.5 Gigabit Ethernet](#),
- 19.2 ns for [5 Gigabit Ethernet](#),
- 9.6 ns for [10 Gigabit Ethernet](#),
- 2.4 ns for [40 Gigabit Ethernet](#), and
- 0.96 ns for [100 Gigabit Ethernet](#).<sup>[1]</sup>

<https://www.oreilly.com/library/view/ethernet-the-definitive/9781449362980/ch04.html>  
[https://en.wikipedia.org/wiki/Interpacket\\_gap](https://en.wikipedia.org/wiki/Interpacket_gap)

## Error Detection and Correction

### **Section 5.2**

Computer Networking – A top-down approach,  
Kurose and Ross, 6<sup>th</sup> Edition.

Section 2.4

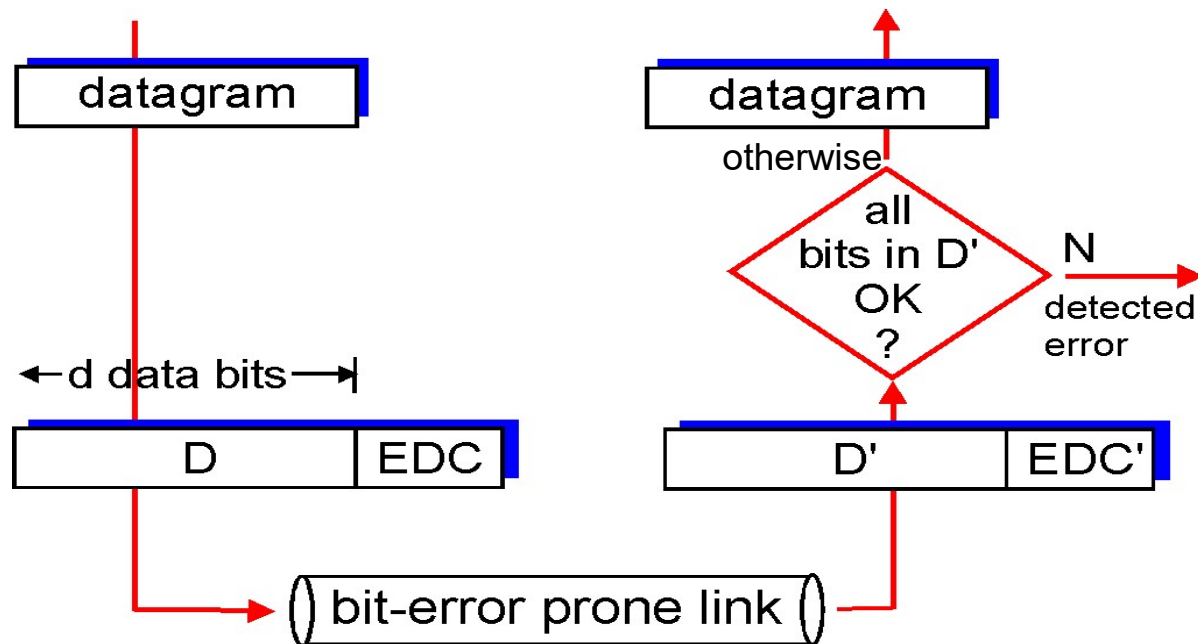
Patterson and Davie

# Error detection

EDC= Error Detection and Correction bits (redundancy)

D = Data protected by error checking, may include header fields

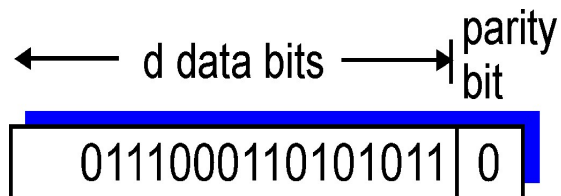
- Error detection not 100% reliable!
  - protocol may miss some errors, but rarely
  - larger EDC field yields better detection and correction



# Parity checking

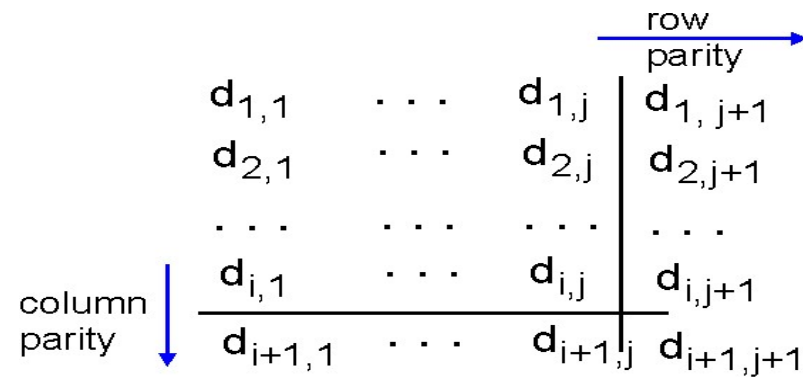
## *single bit parity:*

- ❖ detect single bit errors



## *two-dimensional bit parity:*

- ❖ detect and correct single bit errors



1	0	1	0	1	1
1	1	1	1	0	0
0	1	1	1	0	1
0	0	1	0	1	0

*no errors*

1	0	1	0	1	1
1	0	1	1	0	0
0	1	1	1	0	1
0	0	1	0	1	0

parity error

*correctable  
single bit error*

# Internet checksum (review)

*goal:* detect “errors” (e.g., flipped bits) in transmitted packet  
(note: used at transport layer *only*)

## *sender:*

- ❖ treat segment contents as sequence of 16-bit integers
- ❖ **checksum: addition (1's complement sum) of segment contents**
- ❖ sender puts checksum value into the checksum field (ex. UDP, IP)

## *receiver:*

- ❖ compute checksum of received segment
- ❖ check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected.  
*But maybe errors nonetheless?*



# Internet checksum: example

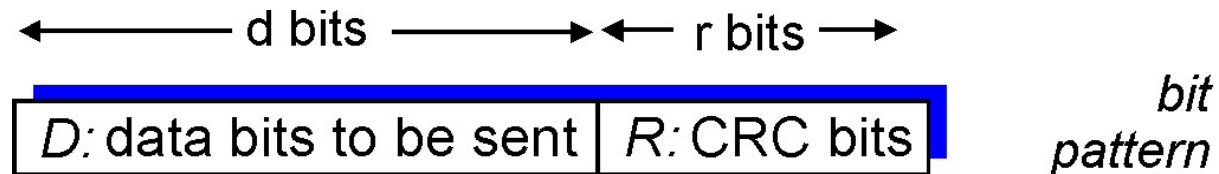
example: add two 16-bit integers

	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
<hr/>																
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1
<hr/>																
sum	1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum	0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

*Note:* when adding numbers, a carryout from the most significant bit needs to be added to the result

# Cyclic redundancy check

- ❖ more powerful error-detection coding
- ❖ view data bits, **D**, as a binary number
- ❖ choose  $r+1$  bit pattern (generator), **G**
- ❖ goal: choose  $r$  CRC bits, **R**, such that
  - $\langle D, R \rangle$  exactly divisible by  $G$  (modulo 2)
  - receiver knows  $G$ , divides  $\langle D, R \rangle$  by  $G$ . If non-zero remainder: error detected!
  - can detect all burst errors less than  $r+1$  bits
- ❖ widely used in practice (Ethernet, 802.11 WiFi, ATM)



$$D * 2^r \text{ XOR } R$$

*mathematical formula*

# Cyclic redundancy check

- ❖ All CRC calculations are done in modulo-2 arithmetic
  - Without carries in addition
  - Without borrows in subtraction
- ❖ This implies Addition and Subtraction are identical and both are equivalent to bitwise exclusive-or (XOR) of the operands
- ❖  $1101 \text{ XOR } 0101 = 1110$
- ❖  $1001 \text{ XOR } 1101 = 0110$
- ❖  $1101 - 0101 = 1110$
- ❖  $1001 - 1101 = 0110$

# CRC example

want:

## D.2<sup>r</sup> XOR R = nG

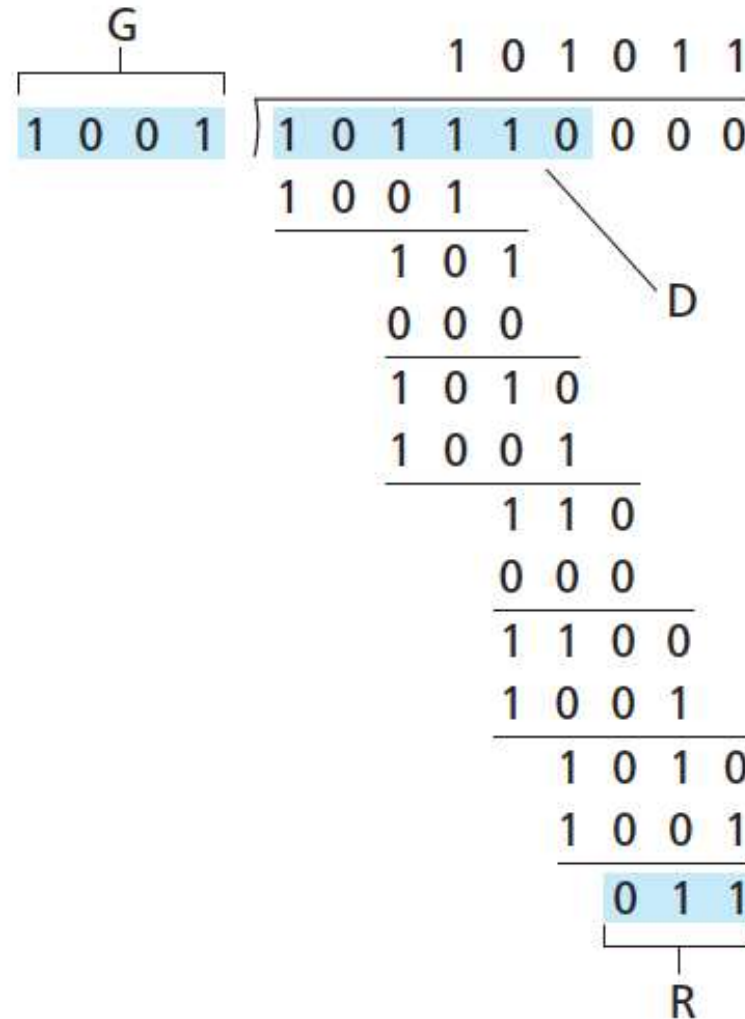
*equivalently:*

$$D \cdot 2^r = nG \text{ XOR } R$$

*equivalently:*

if we divide  $D \cdot 2^r$  by  $G$ , want remainder  $R$  to satisfy:

$$R = \text{remainder}\left[\frac{D \cdot 2^r}{G}\right]$$



original message  
1 0 1 0 0 0 0

@ means X-OR

Generator polynomial  
 $x^3+1$   
 $1.x^3+0.x^2+0.x^1+1.x^0$   
CRC generator  
1 0 0 1 4-bit

If CRC generator is of  $n$  bit then append  $(n-1)$  zeros in the end of original message

Sender

```

1001 | 10100000000
@1001
-----
00110000000
@1001
-----
01010000
@1001
-----
0011000
@1001
-----
01010
@1001
-----
0011
  
```

Message to be transmitted

```

10100000000
+ 011
-----
1010000011
  
```

```

1001 | 1010000011
@1001
-----
0011000011
@1001
-----
01010011
@1001
-----
0011011
@1001
-----
01001
@1001
-----
0000
  
```

Receiver

Zero means data is accepted

# Reliable Transmission

Section 2.5

Computer Networks 5<sup>th</sup> Edn: Patterson and Davie

# Reliable Transmission

- ❖ CRC is used to detect errors.
- ❖ Some error codes are strong enough to correct errors.
- ❖ The overhead is typically too high.
- ❖ Corrupt frames must be discarded.
- ❖ A link-level protocol that wants to deliver frames reliably must recover from these discarded frames.
- ❖ This is accomplished using a combination of two fundamental mechanisms
  - Acknowledgements and Timeouts

# Reliable Transmission

- ❖ An *acknowledgement* (ACK for short) is a small control frame that a protocol sends back to its peer saying that it has received the earlier frame.
  - A control frame is a frame with header only (no data).
- ❖ The receipt of an *acknowledgement* indicates to the sender of the original frame that its frame was successfully delivered.



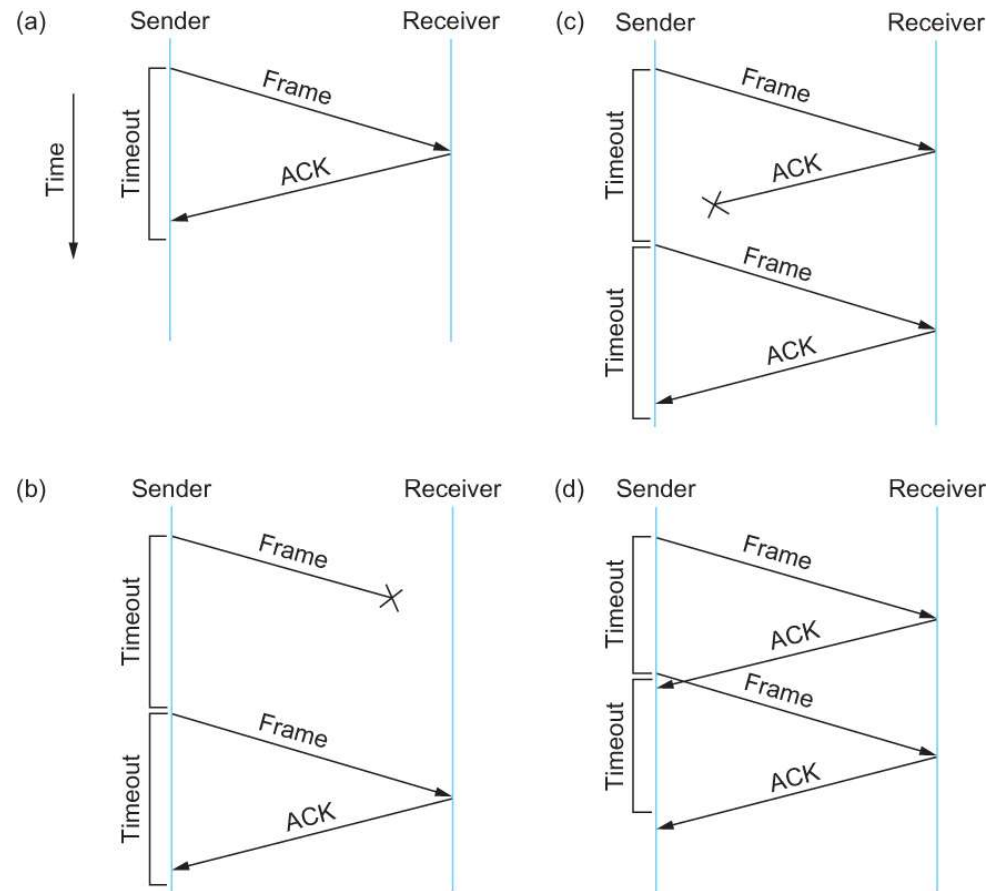
# Reliable Transmission

- ❖ If the sender does not receive an *acknowledgment* after a reasonable amount of time, then it retransmits the original frame.
- ❖ The action of waiting a reasonable amount of time is called a *timeout*.
- ❖ The general strategy of using *acknowledgements* and *timeouts* to implement reliable delivery is sometimes called **Automatic Repeat reQuest (ARQ)**.

# Stop and Wait Protocol

- ❖ Idea of stop-and-wait protocol is straightforward
  - After transmitting one frame, the sender waits for an acknowledgement before transmitting the next frame.
  - If the acknowledgement does not arrive after a certain period of time, the sender times out and retransmits the original frame

# Stop and Wait Protocol



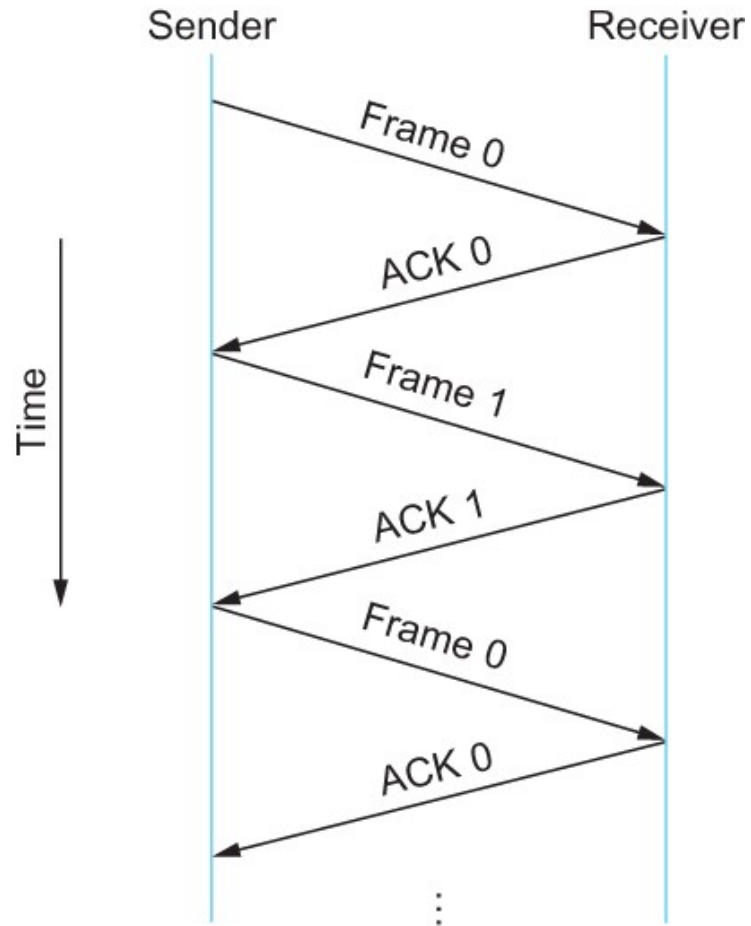
Timeline showing four different scenarios for the stop-and-wait algorithm.

(a) The ACK is received before the timer expires; (b) the original frame is lost; (c) the ACK is lost; (d) the timeout fires too soon

# Stop and Wait Protocol

- ❖ If the acknowledgment is lost or delayed in arriving
  - The sender times out and retransmits the original frame, but the receiver will think that it is the next frame since it has correctly received and acknowledged the first frame
  - As a result, duplicate copies of frames will be delivered
- ❖ How to solve this
  - Use 1 bit sequence number (0 or 1)
  - When the sender retransmits frame 0, the receiver can determine that it is seeing a second copy of frame 0 rather than the first copy of frame 1 and therefore can ignore it (the receiver still acknowledges it, in case the first acknowledgement was lost)

# Stop and Wait Protocol

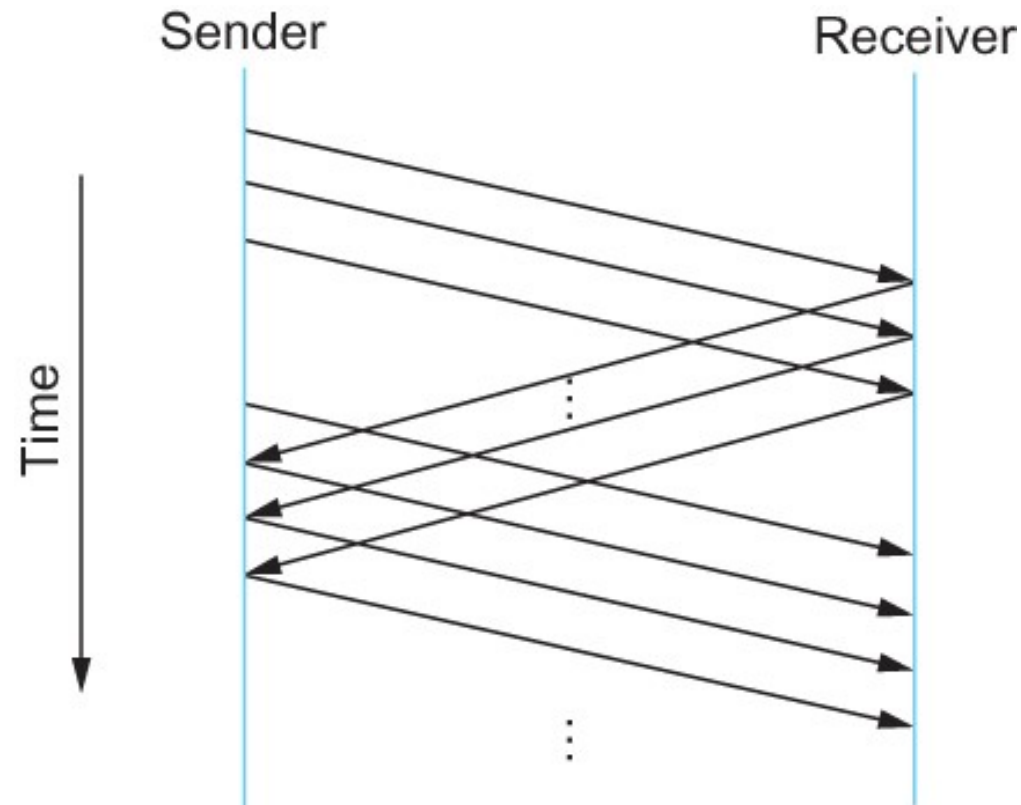


Timeline for stop-and-wait with 1-bit sequence number

# Stop and Wait Protocol

- ❖ The sender has only one outstanding frame on the link at a time
  - This may be far below the link's capacity
- ❖ Consider a 1.5 Mbps link with a 45 ms RTT
  - The link has a delay  $\times$  bandwidth product of 67.5 Kb or approximately 8 KB
  - Since the sender can send only one frame per RTT and assuming a frame size of 1 KB
  - Maximum Sending rate
    - $\text{Bits per frame} \div \text{Time per frame} = 1024 \times 8 \div 0.045 = 182 \text{ Kbps}$   
Or about one-eighth of the link's capacity
  - To use the link fully, then sender should transmit up to eight frames before having to wait for an acknowledgement

# Sliding Window Protocol



Timeline for Sliding Window Protocol

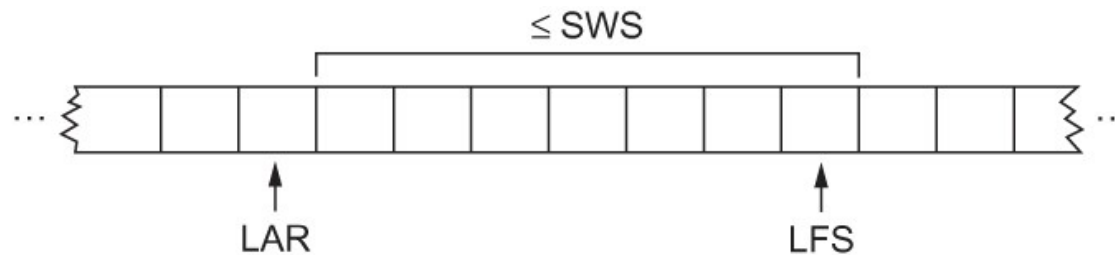
# Sliding Window Protocol

- ❖ Sender assigns a sequence number denoted as SeqNum to each frame.
  - Assume it can grow infinitely large
- ❖ Sender maintains three variables
  - Sending Window Size (SWS)
    - Upper bound on the number of outstanding (unacknowledged) frames that the sender can transmit
  - Last Acknowledgement Received (LAR)
    - Sequence number of the last acknowledgement received
  - Last Frame Sent (LFS)
    - Sequence number of the last frame sent



# Sliding Window Protocol

- ❖ Sender also maintains the following invariant  
$$\text{LFS} - \text{LAR} \leq \text{SWS}$$



Sliding Window on Sender

# Sliding Window Protocol

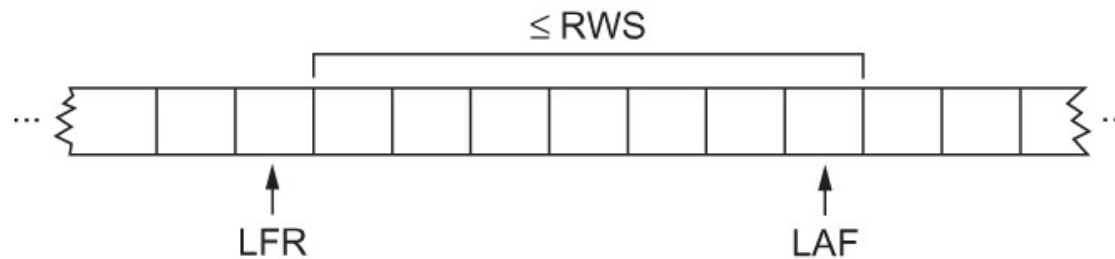
- ❖ When an acknowledgement arrives
  - the sender moves LAR to right, thereby allowing the sender to transmit another frame
- ❖ Also the sender associates a timer with each frame it transmits
  - It retransmits the frame if the timer expires before the ACK is received
- ❖ Note that the sender has to be willing to buffer up to SWS frames
  - WHY?

# Sliding Window Protocol

- ❖ Receiver maintains three variables
  - Receiving Window Size (RWS)
    - Upper bound on the number of out-of-order frames that the receiver is willing to accept
  - Largest Acceptable Frame (LAF)
    - Sequence number of the largest acceptable frame
  - Last Frame Received (LFR)
    - Sequence number of the last frame received

# Sliding Window Protocol

- ❖ Receiver also maintains the following invariant  
$$\text{LAF} - \text{LFR} \leq \text{RWS}$$



Sliding Window on Receiver

# Sliding Window Protocol

- ❖ When a frame with sequence number  $\text{SeqNum}$  arrives, what does the receiver do?
  - If  $\text{SeqNum} \leq \text{LFR}$  or  $\text{SeqNum} > \text{LAF}$ 
    - Discard it (the frame is outside the receiver window)
  - If  $\text{LFR} < \text{SeqNum} \leq \text{LAF}$ 
    - Accept it
    - Now the receiver needs to decide whether or not to send an ACK

# Sliding Window Protocol

- Let SeqNumToAck
  - Denote the largest sequence number not yet acknowledged, such that all frames with sequence number less than or equal to SeqNumToAck have been received
- The receiver acknowledges the receipt of SeqNumToAck even if high-numbered packets have been received
  - This acknowledgement is said to be cumulative.
- The receiver then sets
  - $LFR = SeqNumToAck$  and adjusts
  - $LAF = LFR + RWS$

# Sliding Window Protocol

For example, suppose  $LFR = 5$  and  $RWS = 4$

(i.e. the last ACK that the receiver sent was for seq. no. 5)

⇒  $LAF = 9$

If frames 7 and 8 arrive, they will be buffered because they are within the receiver window

But no ACK will be sent since frame 6 is yet to arrive

Frames 7 and 8 are out of order

Frame 6 arrives (it is late because it was lost first time and had to be retransmitted)

Now Receiver Acknowledges Frame 8

and bumps  $LFR$  to 8

and  $LAF$  to 12

# Issues with Sliding Window Protocol

- ❖ When timeout occurs, the amount of data in transit decreases
  - Since the sender is unable to advance its window
- ❖ When the packet loss occurs, this scheme is no longer keeping the pipe full
  - The longer it takes to notice that a packet loss has occurred, the more severe the problem becomes
- ❖ How to improve this
  - Negative Acknowledgement (NAK)
  - Additional Acknowledgement
  - Selective Acknowledgement



# Issues with Sliding Window Protocol

- ❖ Negative Acknowledgement (NAK)
  - Receiver sends NAK for frame 6 when frame 7 arrive (in the previous example)
    - However this is unnecessary since sender's timeout mechanism will be sufficient to catch the situation
- ❖ Additional Acknowledgement
  - Receiver sends additional ACK for frame 5 when frame 7 arrives
    - Sender uses duplicate ACK as a clue for frame loss
- ❖ Selective Acknowledgement
  - Receiver will acknowledge exactly those frames it has received, rather than the highest number frames
    - Receiver will acknowledge frames 7 and 8
    - Sender knows frame 6 is lost
    - Sender can keep the pipe full (additional complexity)

# Issues with Sliding Window Protocol

## How to select the window size

- SWS is easy to compute
  - $\text{Delay} \times \text{Bandwidth}$
- RWS can be anything
  - Two common setting

- »  $\text{RWS} = 1$

No buffer at the receiver for frames that arrive out of order

- »  $\text{RWS} = \text{SWS}$

The receiver can buffer frames that the sender transmits

It does not make any sense to keep  $\text{RWS} > \text{SWS}$

WHY?

# Issues with Sliding Window Protocol

## ❖ Finite Sequence Number

- Frame sequence number is specified in the header field
  - Finite size
    - » 3 bit: eight possible sequence number: 0, 1, 2, 3, 4, 5, 6, 7
  - It is necessary to wrap around

# Issues with Sliding Window Protocol

- ❖ How to distinguish between different incarnations of the same sequence number?
  - Number of possible sequence number must be larger than the number of outstanding frames allowed
    - Stop and Wait: One outstanding frame
      - » 2 distinct sequence number (0 and 1)
    - Let *MaxSeqNum* be the number of available sequence numbers
    - $SWS + 1 \leq \text{MaxSeqNum}$ 
      - Is this sufficient?

# Issues with Sliding Window Protocol

$$SWS + 1 \leq \text{MaxSeqNum}$$

- Is this sufficient?

- Depends on RWS

- If  $RWS = 1$ , then sufficient

- If  $RWS = SWS$ , then not good enough

- ❖ For example, we have eight sequence numbers

0, 1, 2, 3, 4, 5, 6, 7

$RWS = SWS = 7$

Sender sends 0, 1, ..., 6

Receiver receives 0, 1, ..., 6

Receiver acknowledges 0, 1, ..., 6

ACK (0, 1, ..., 6) are lost

Sender retransmits 0, 1, ..., 6

Receiver is expecting 7, 0, ..., 5

# Issues with Sliding Window Protocol

To avoid this,

If  $RWS = SWS$

$SWS < (MaxSeqNum + 1)/2$

# Issues with Sliding Window Protocol

## ❖ Serves three different roles

- Reliable
- Preserve the order
  - Each frame has a sequence number
  - The receiver makes sure that it does not pass a frame up to the next higher-level protocol until it has already passed up all frames with a smaller sequence number
- Frame control
  - Receiver is able to throttle the sender
    - Keeps the sender from overrunning the receiver
      - » From transmitting more data than the receiver is able to process

## Principles of Reliable Data transfer

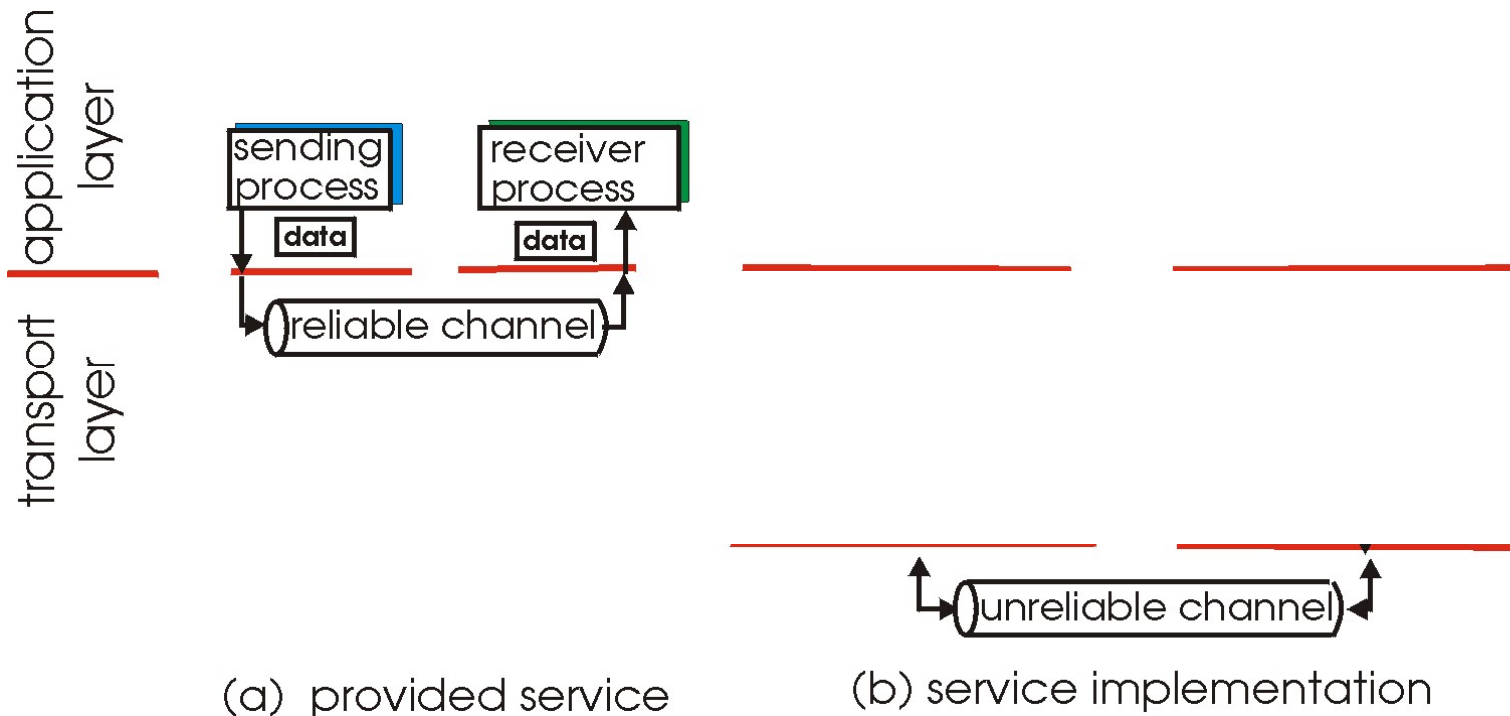
### **Section 3.4**

Computer Networking – A top-down approach,  
Kurose and Ross, 6<sup>th</sup> Edition.



# Principles of reliable data transfer

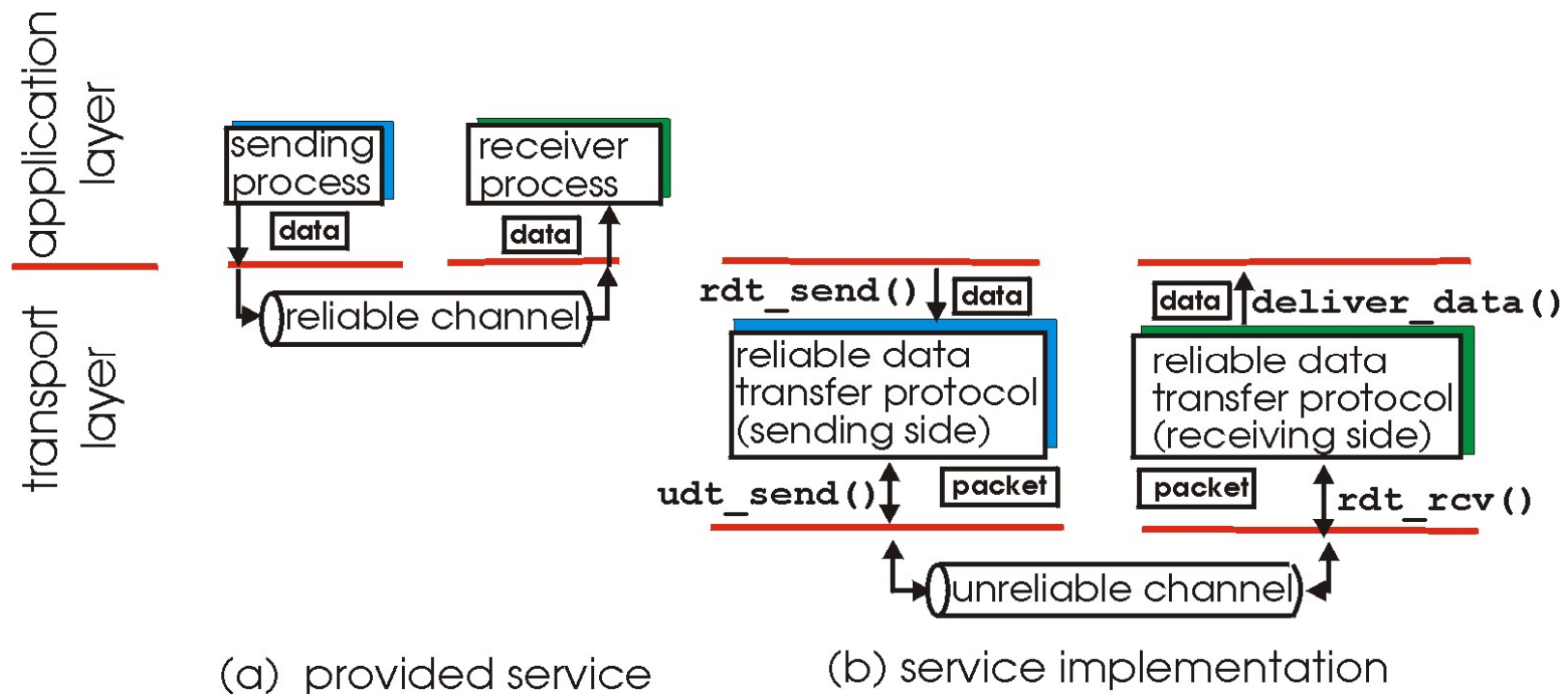
- ❖ important in application, transport, link layers
  - top-10 list of important networking topics!



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

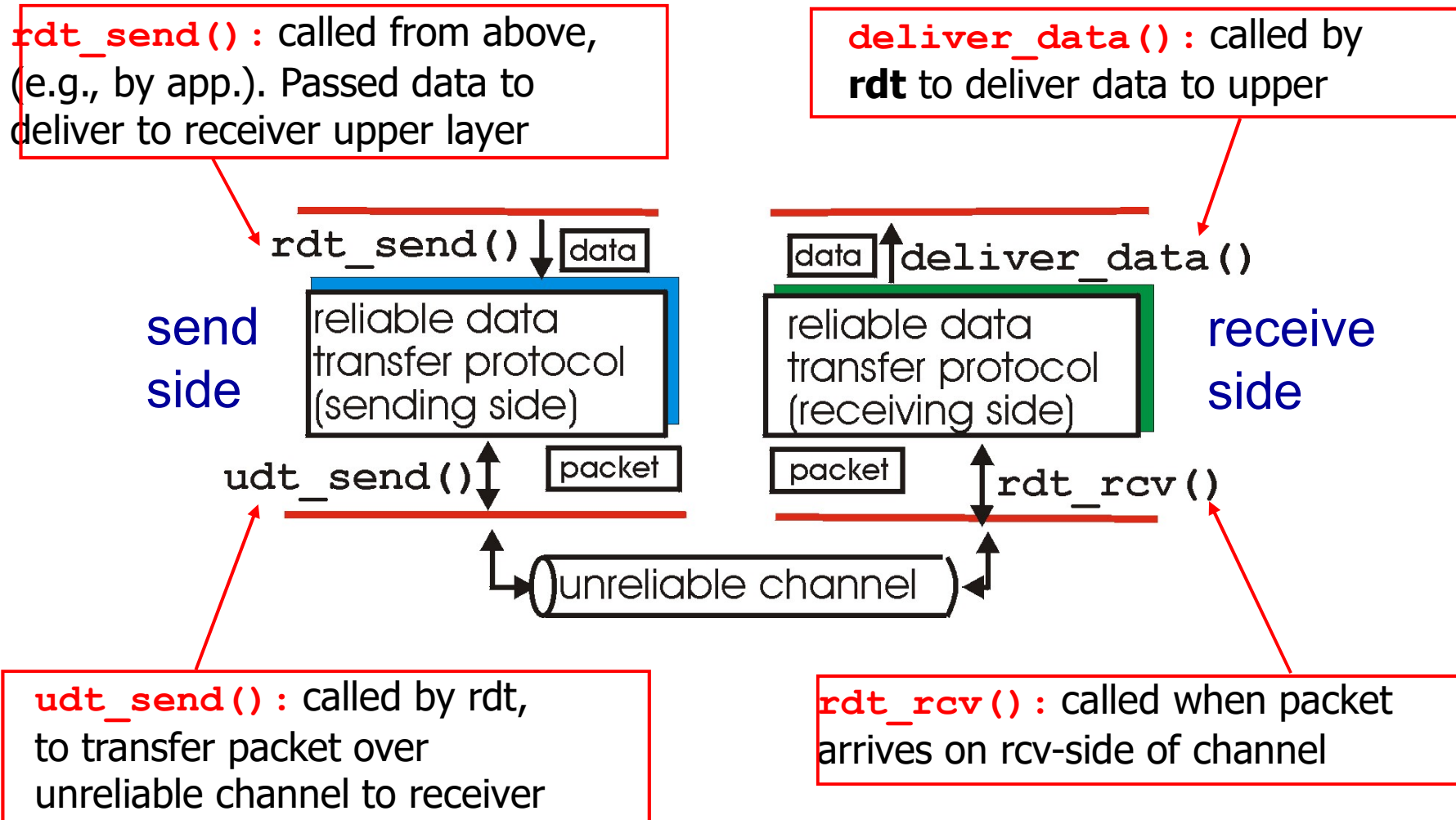
# Principles of reliable data transfer

- ❖ important in application, 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

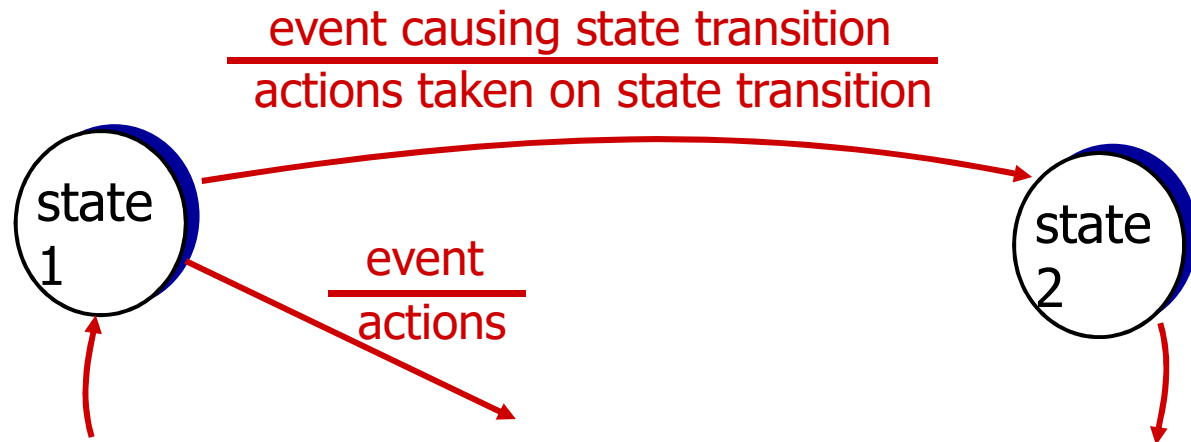


# Reliable data transfer: getting started

we'll:

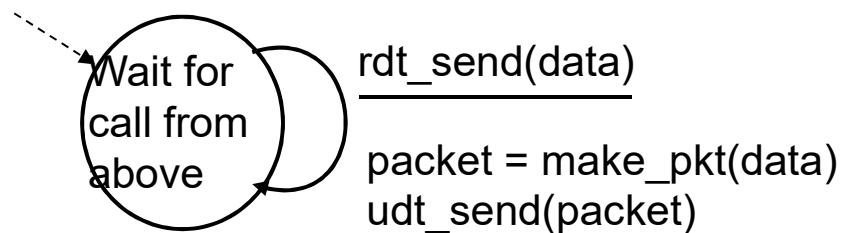
- ❖ incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- ❖ consider only unidirectional data transfer
  - but control info will flow on both directions!
- ❖ use finite state machines (FSM) to specify sender, receiver

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

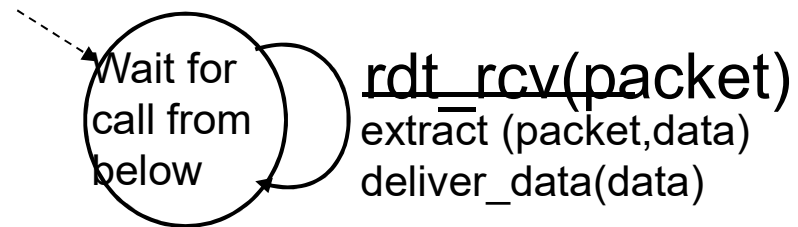


# rdt1.0: reliable transfer over a reliable channel

- ❖ 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

# rdt2.0: channel with bit errors

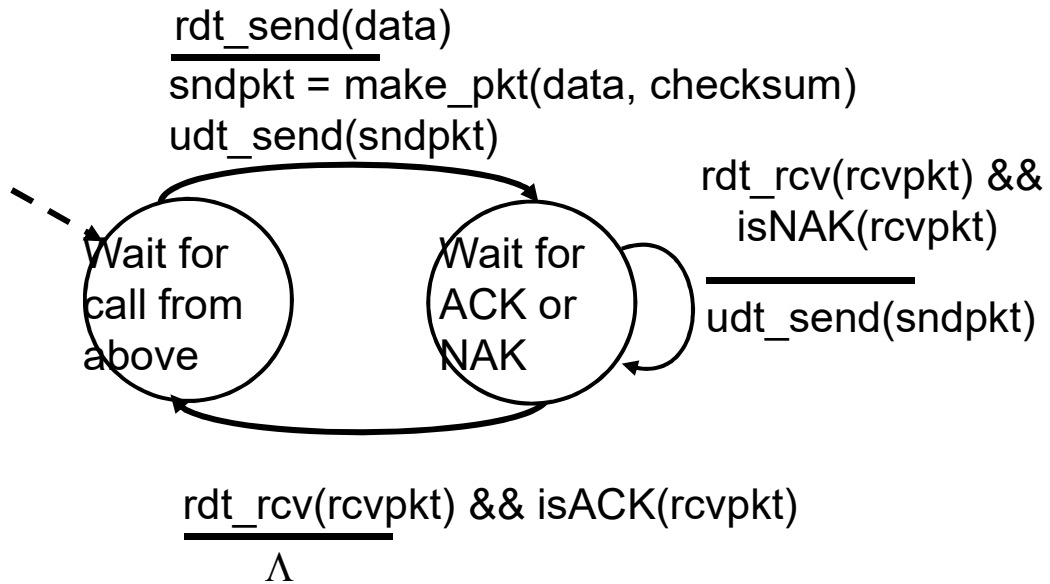
- ❖ underlying channel may flip bits in packet
  - checksum to detect bit errors
- ❖ *the question: how to recover from errors:*

*How do humans recover from “errors” during conversation?*

# rdt2.0: channel with bit errors

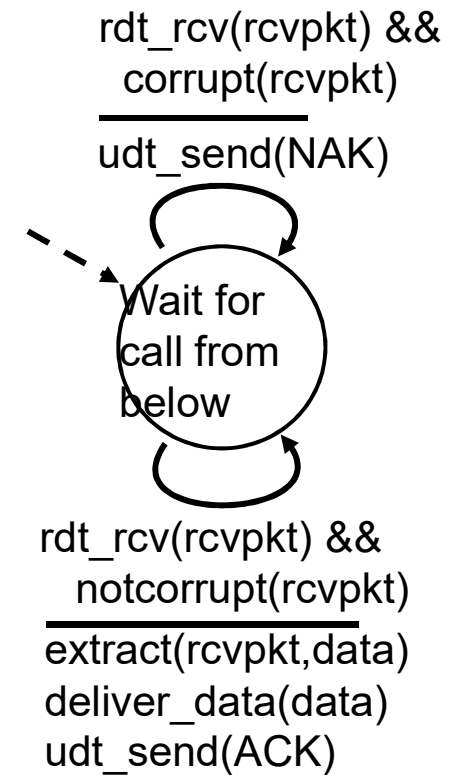
- ❖ underlying channel may flip bits in packet
  - checksum to detect bit errors
- ❖ *the question: how to recover from errors:*
  - *acknowledgements (ACKs)*: receiver explicitly tells sender that pkt received OK
  - *negative acknowledgements (NAKs)*: receiver explicitly tells sender that pkt had errors
  - 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: FSM specification



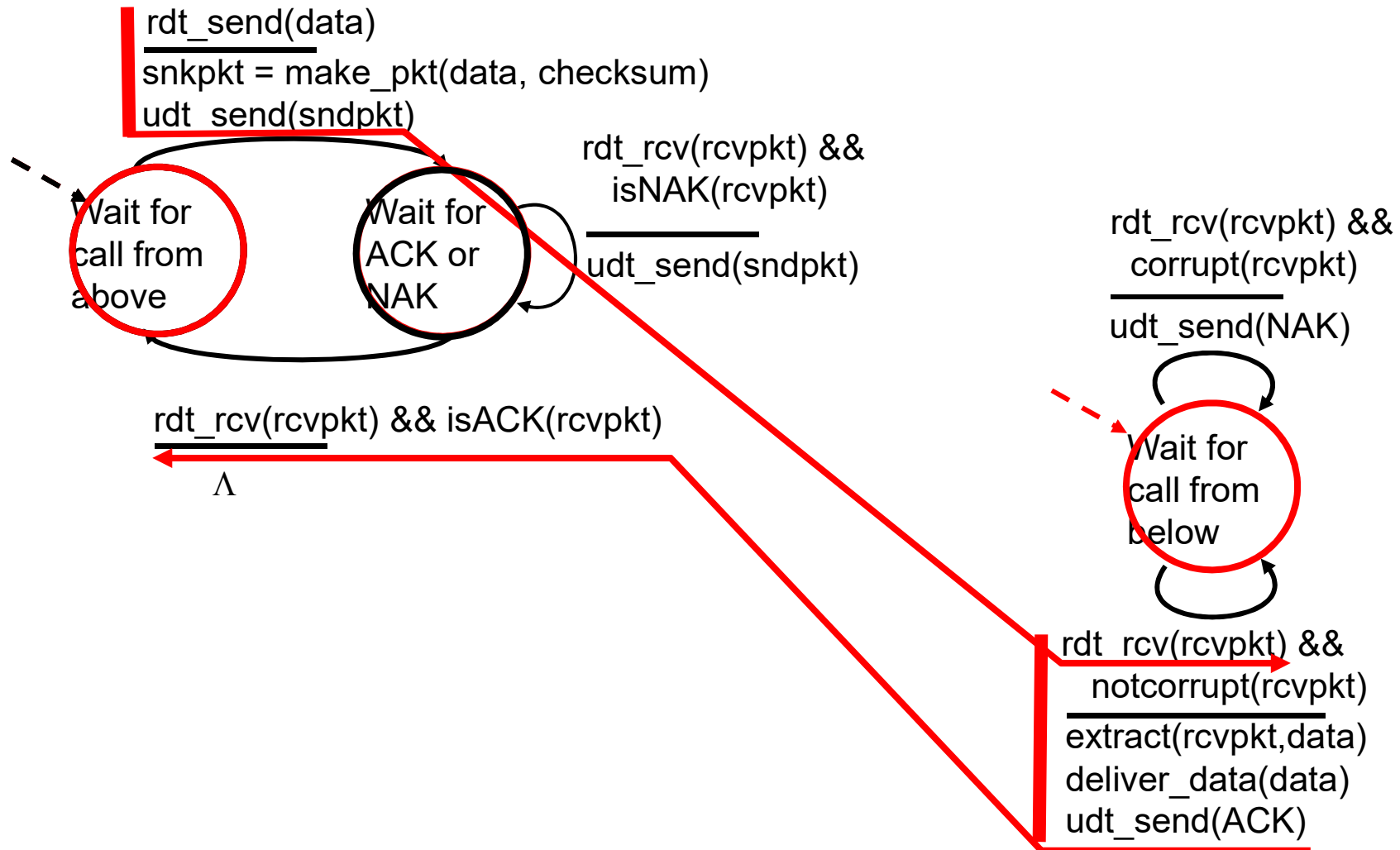
sender

receiver

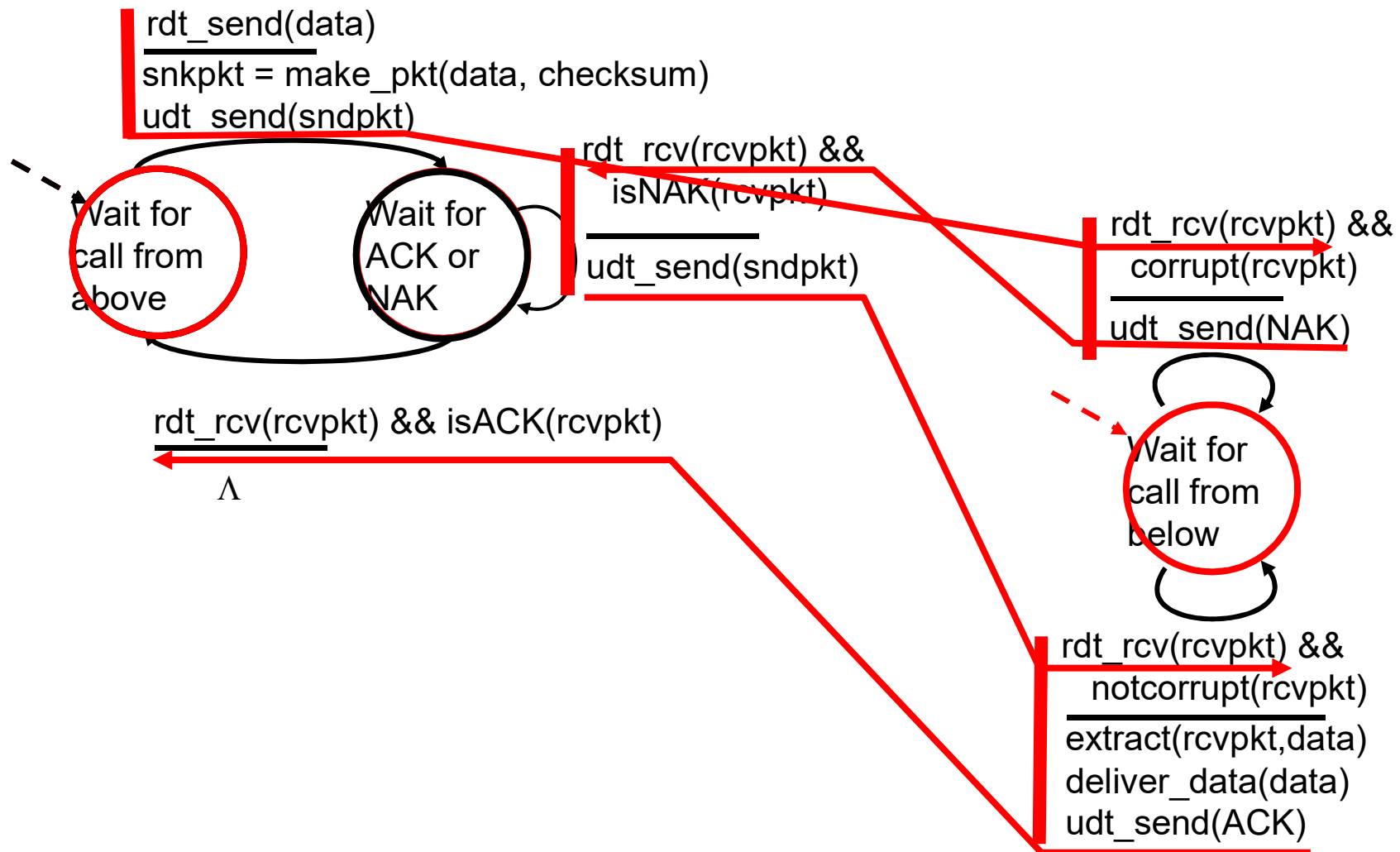




# rdt2.0: operation with no errors



# 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

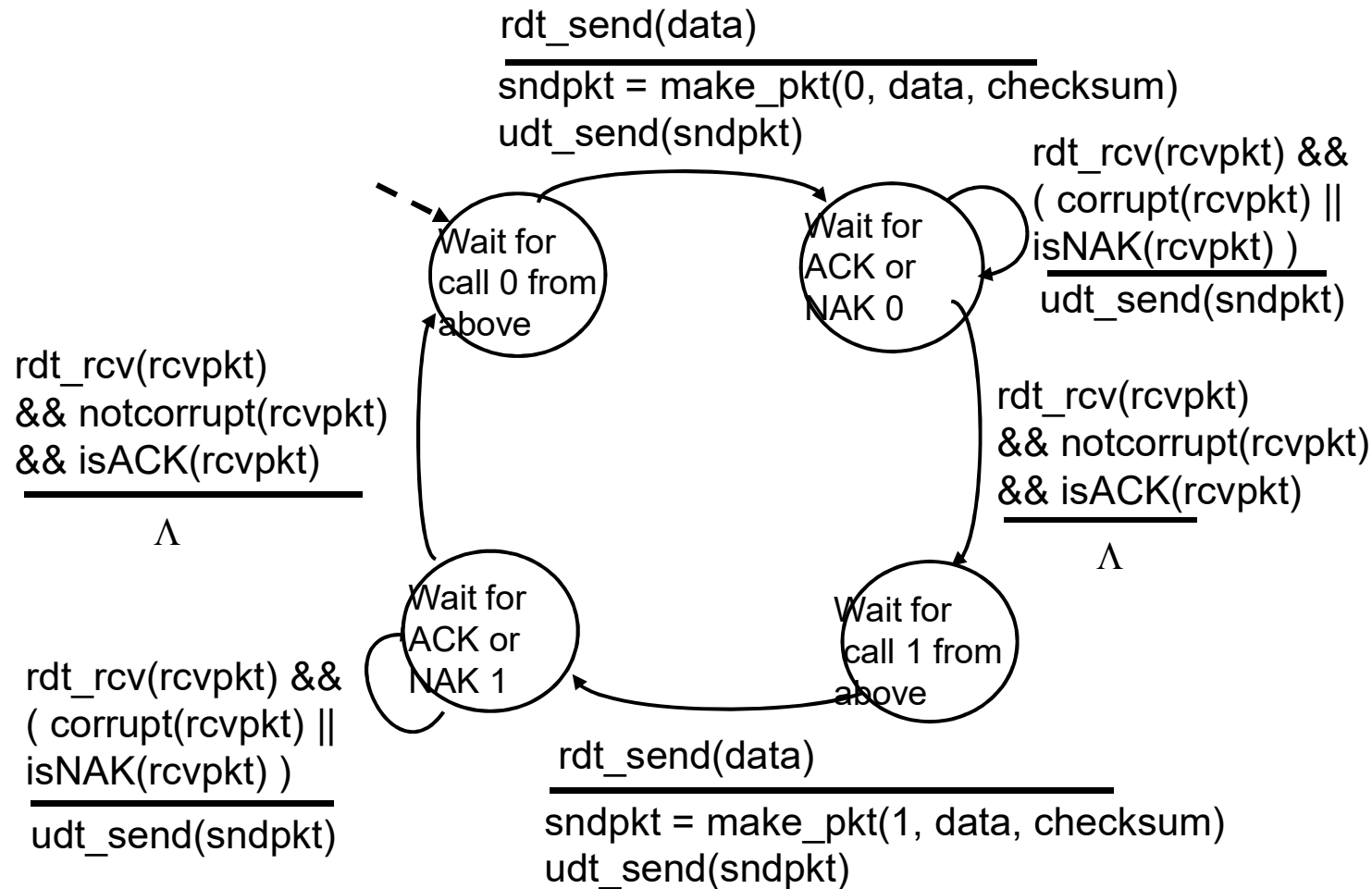
## handling duplicates:

- ❖ sender retransmits current pkt if ACK/NAK corrupted
- ❖ sender adds *sequence number* to each pkt
- ❖ receiver discards (doesn't deliver up) duplicate pkt

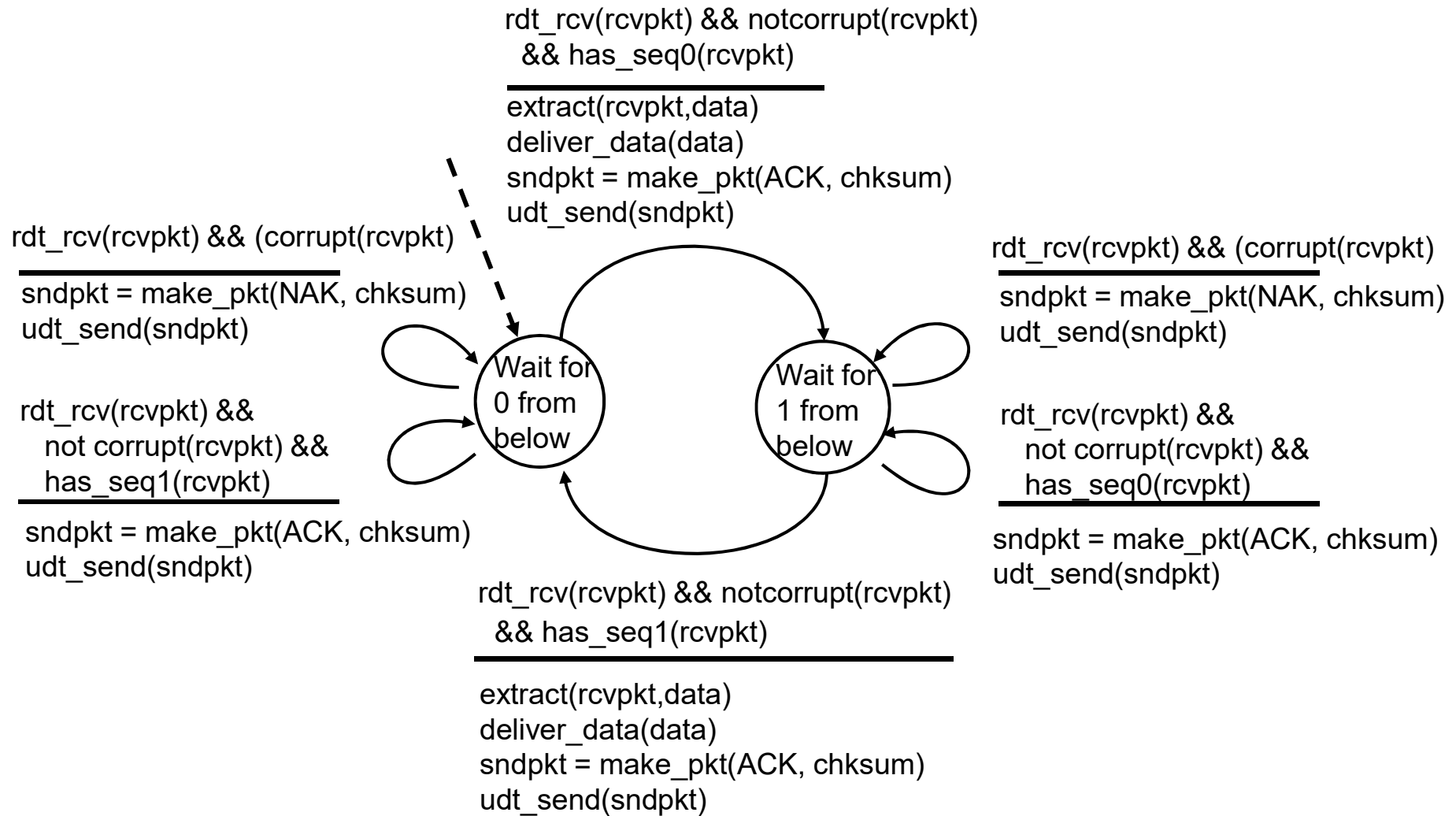
## stop and wait

sender sends one packet,  
then waits for receiver  
response

## rdt2.1: sender, handles garbled ACK/NAKs



# rdt2.1: receiver, handles garbled ACK/NAKs



# rdt2.1: discussion

## sender:

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

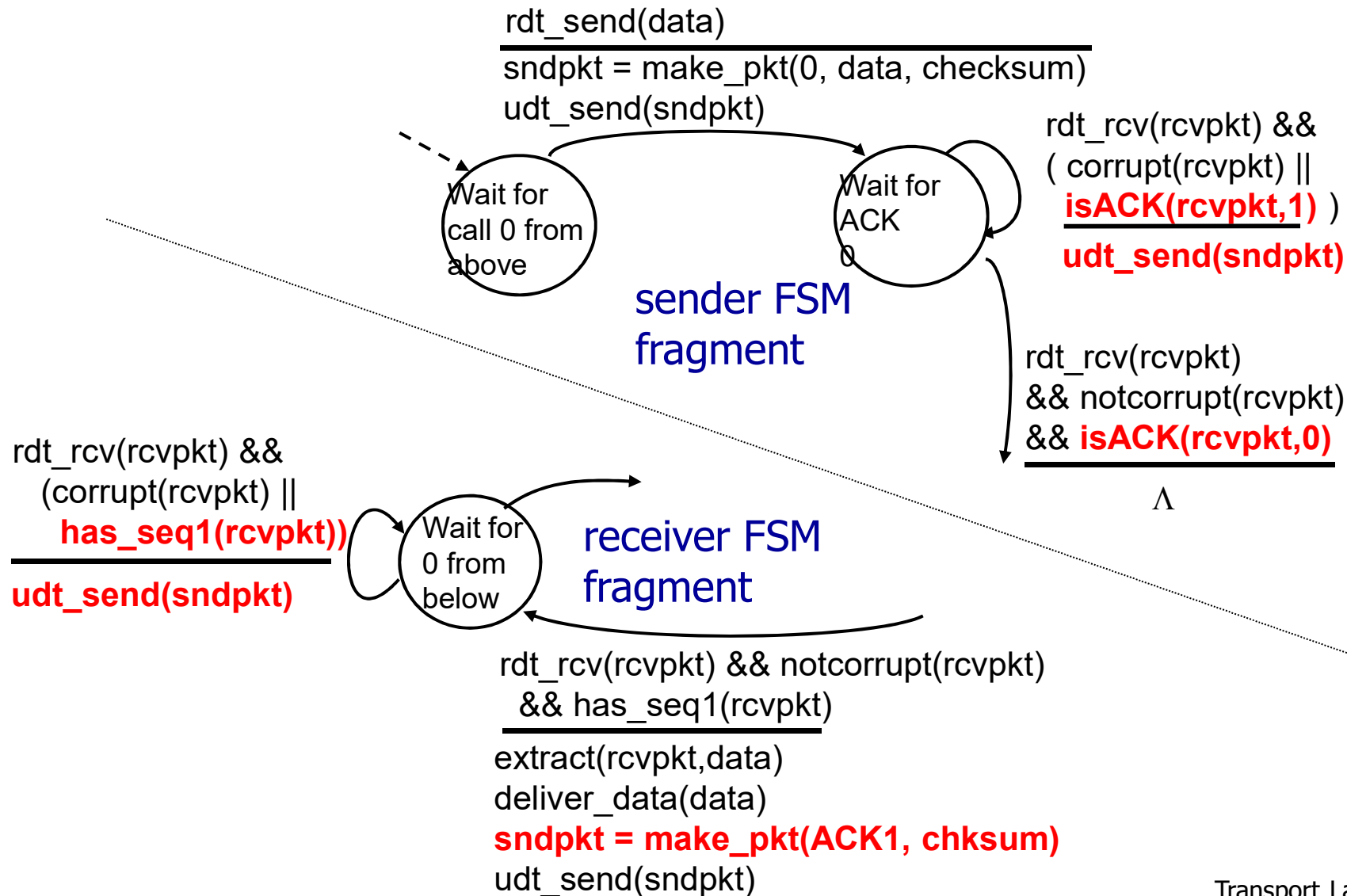
## receiver:

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

## rdt2.2: a NAK-free protocol

- ❖ same functionality as rdt2.1, using ACKs only
- ❖ instead of NAK, receiver sends ACK for last 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:

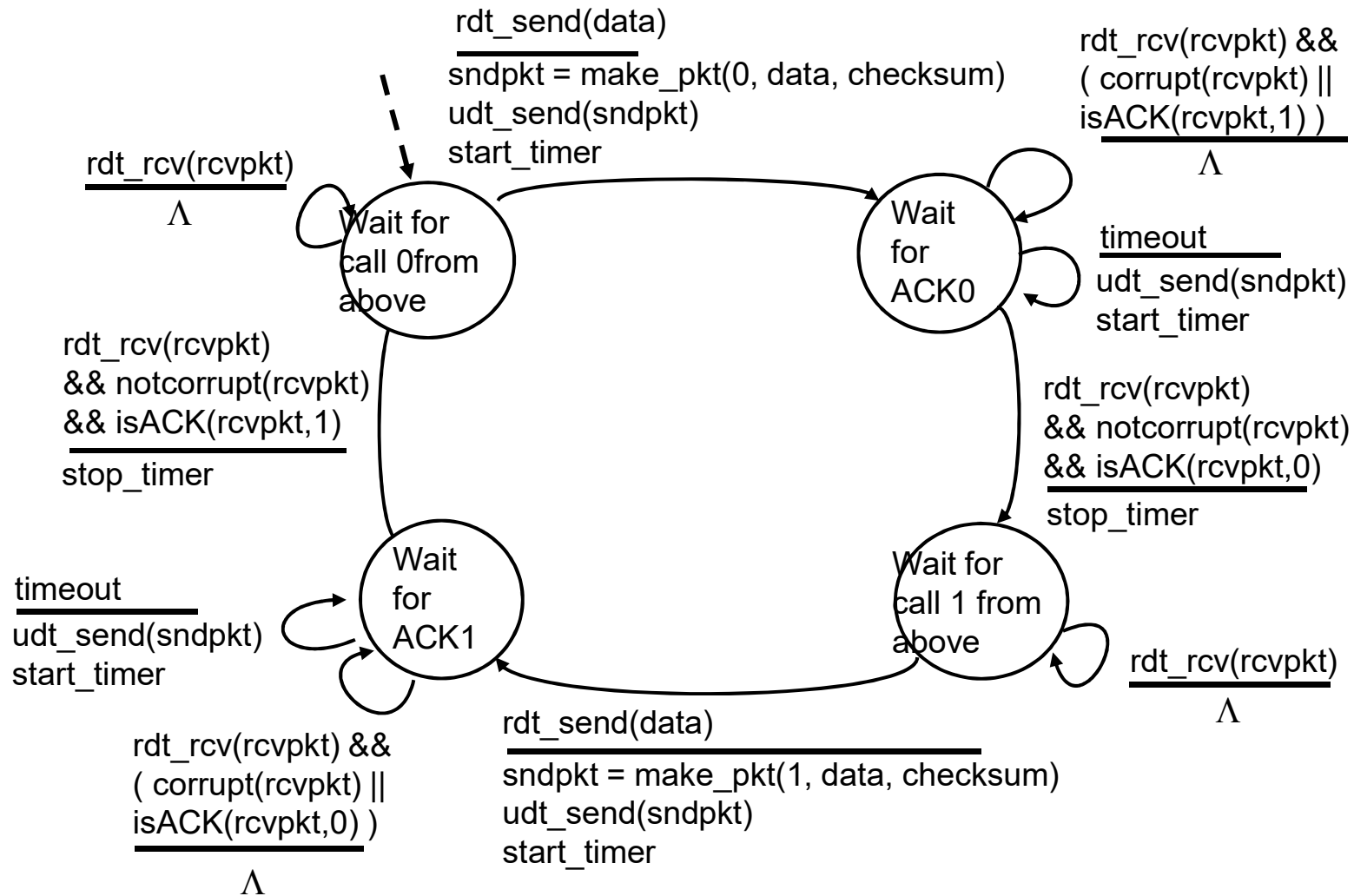
underlying channel can also lose packets (data, ACKs)

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

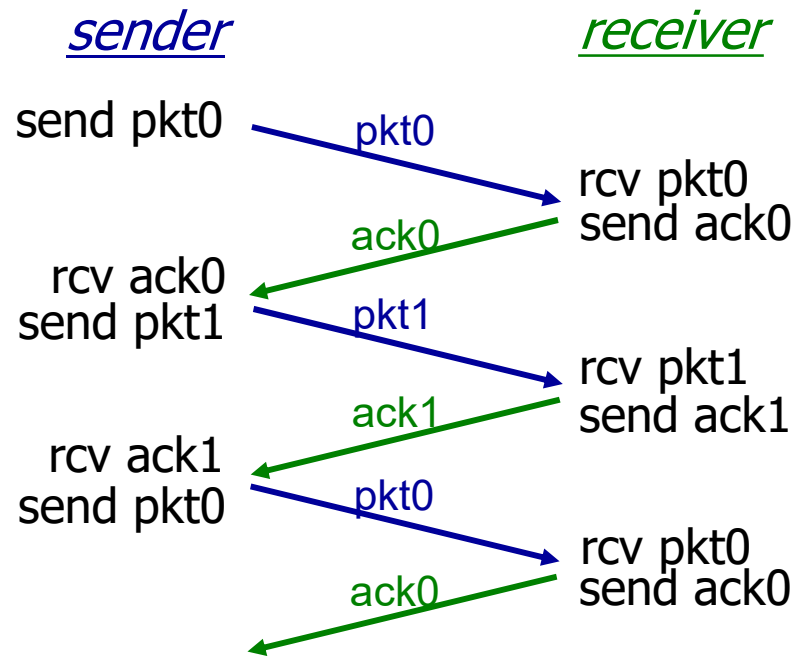
approach: sender waits “reasonable” amount of time for ACK

- ❖ 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

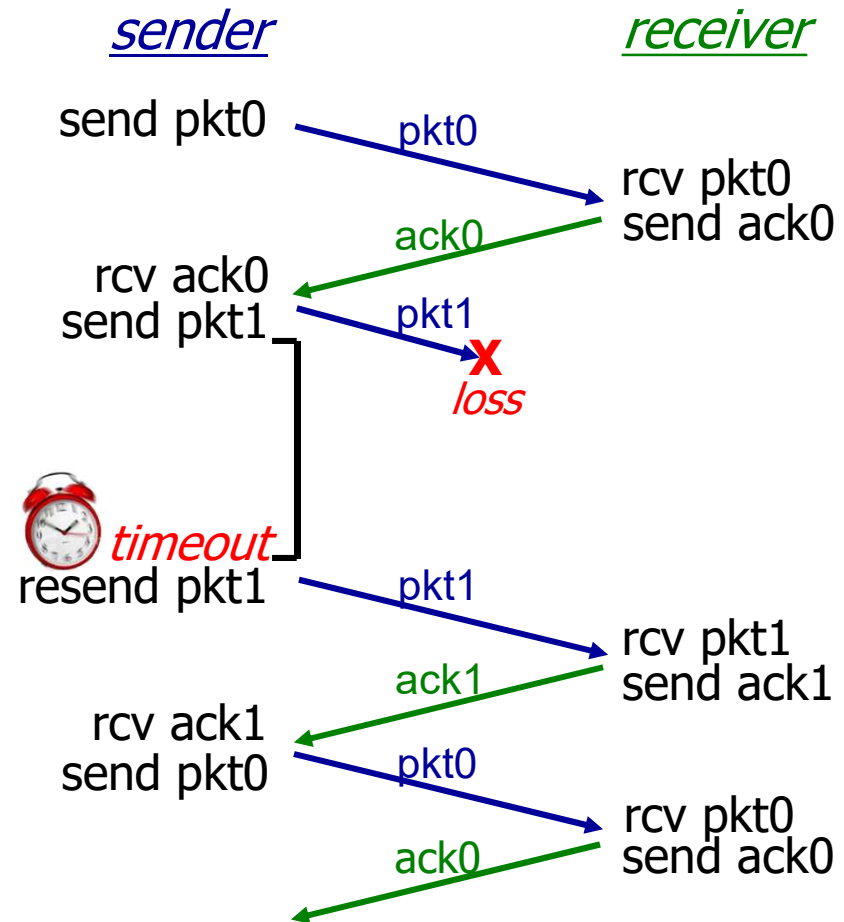
# rdt3.0 sender



# rdt3.0 in action

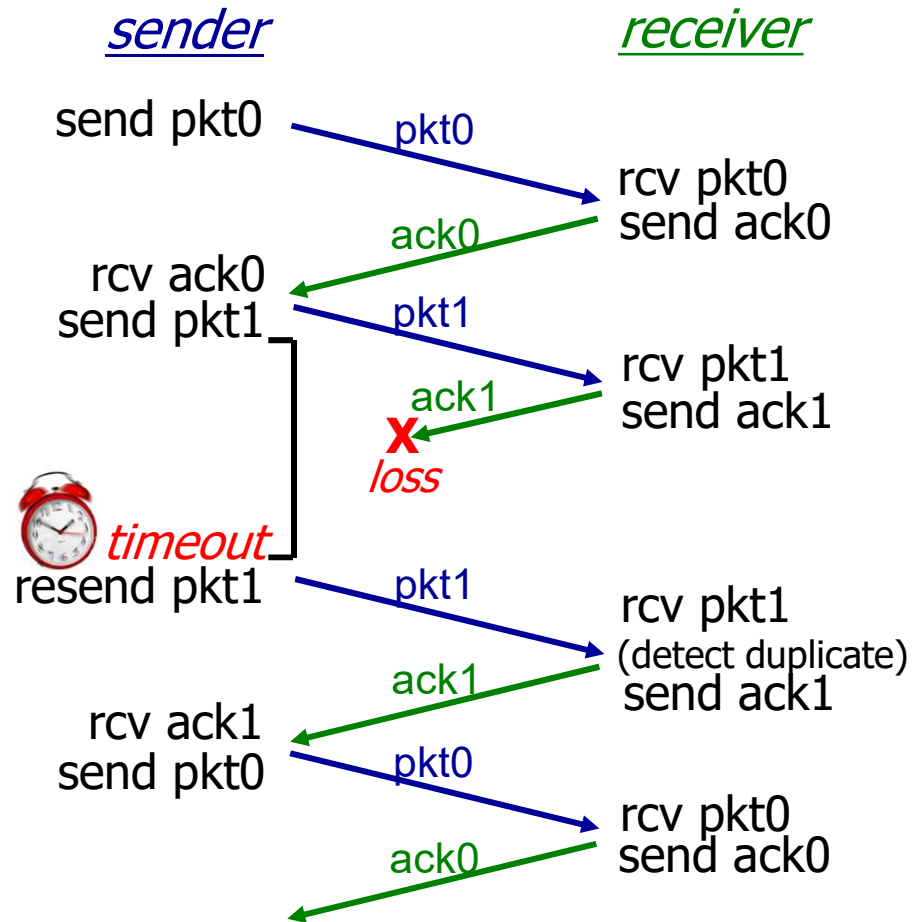


(a) no loss

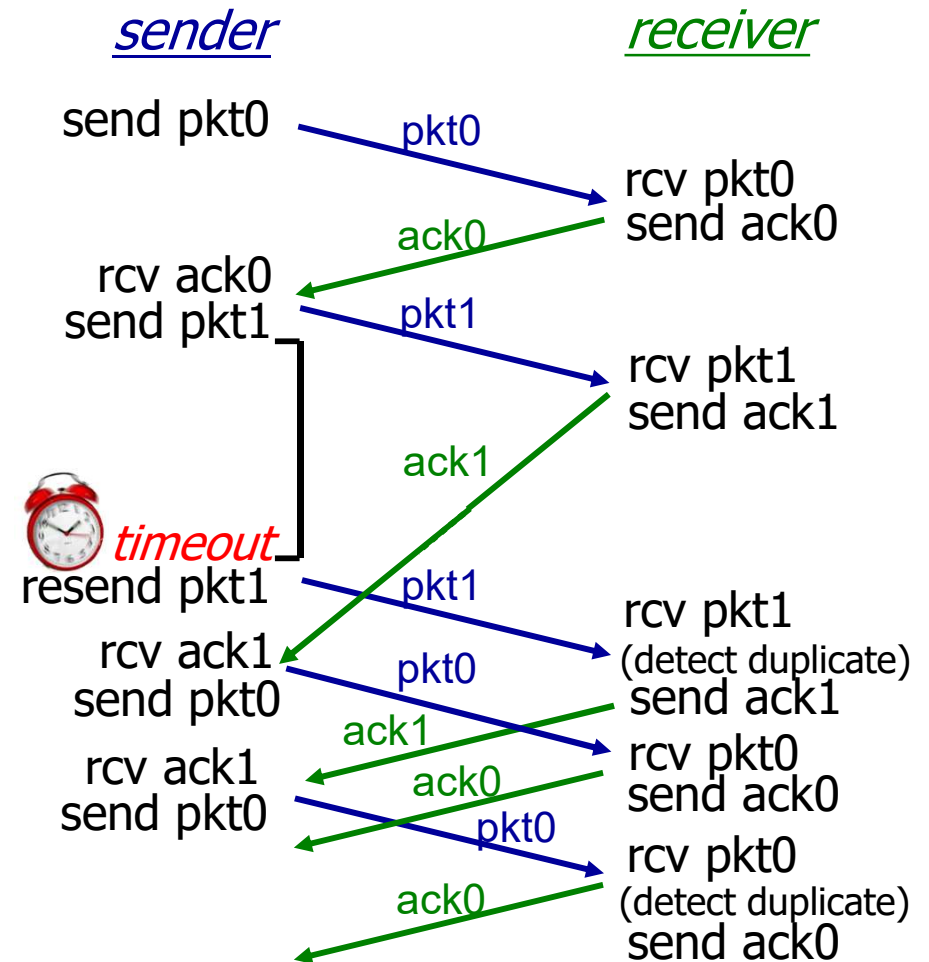


(b) packet loss

# rdt3.0 in action



(c) ACK loss



(d) premature timeout/ delayed ACK

# Performance of rdt3.0

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

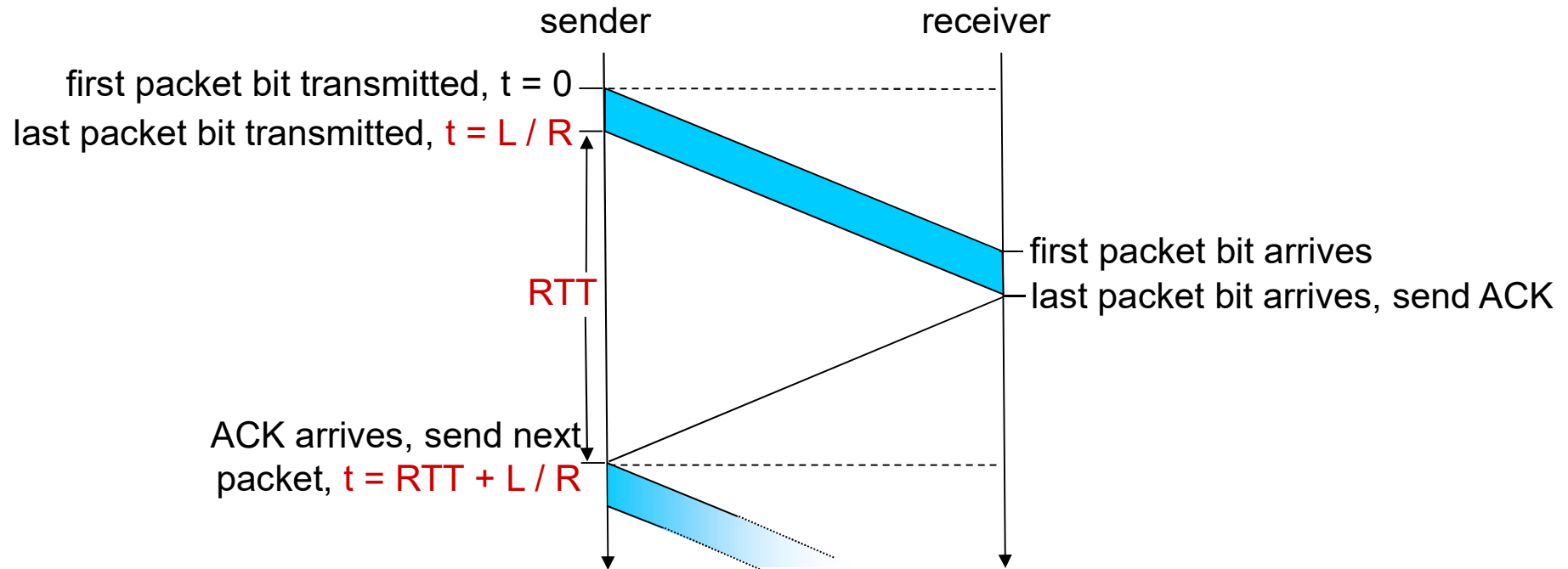
$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$

- $U_{sender}$ : *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

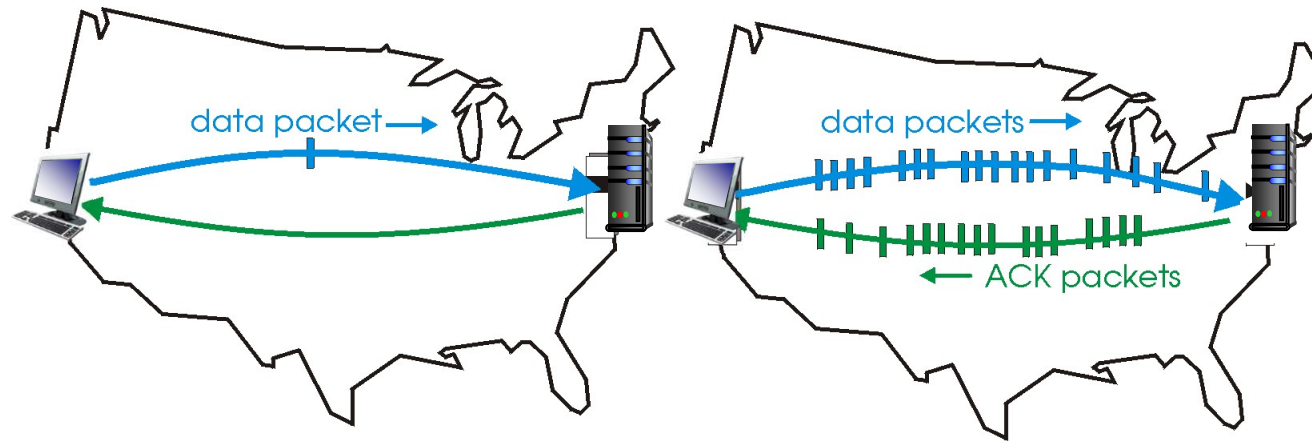


$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

# 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

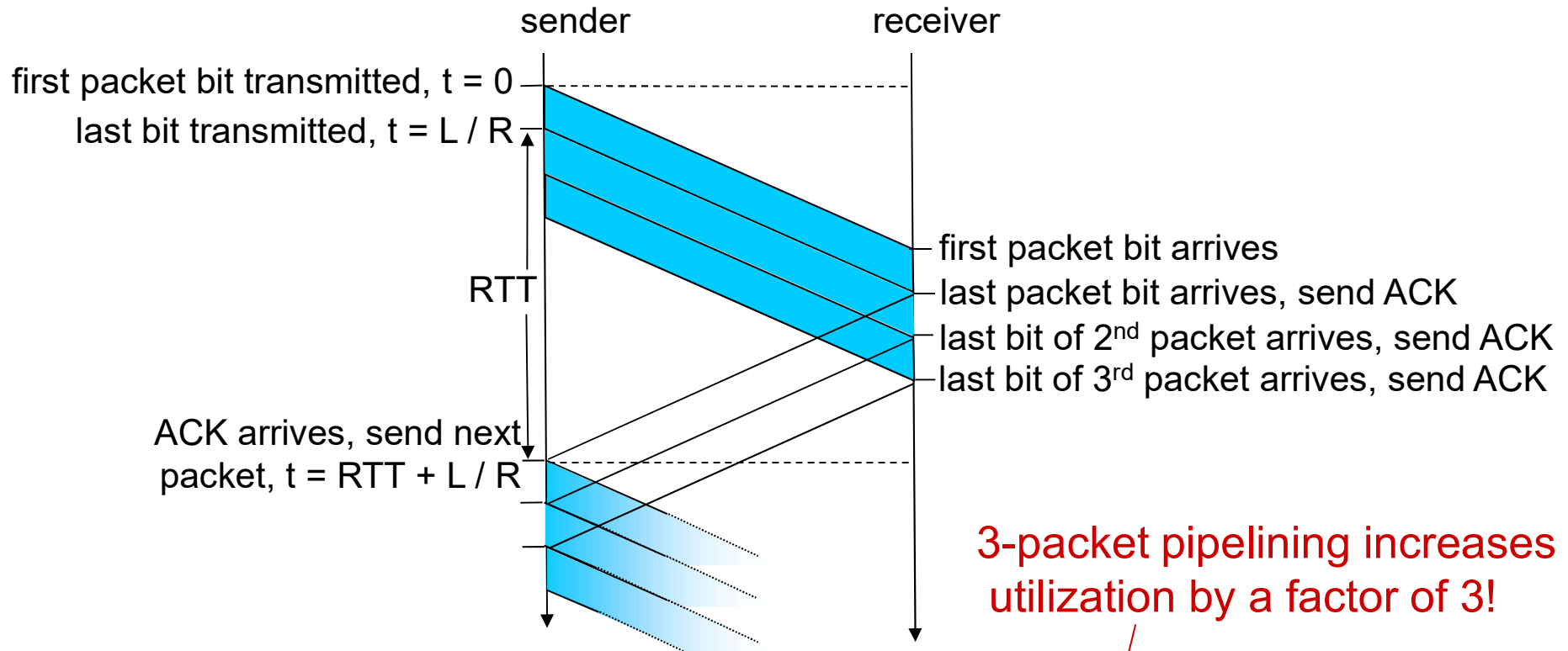


(a) a stop-and-wait protocol in operation

(b) a pipelined protocol in operation

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

# Pipelining: increased utilization



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

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



# Pipelined protocols: overview

## Go-back-N:

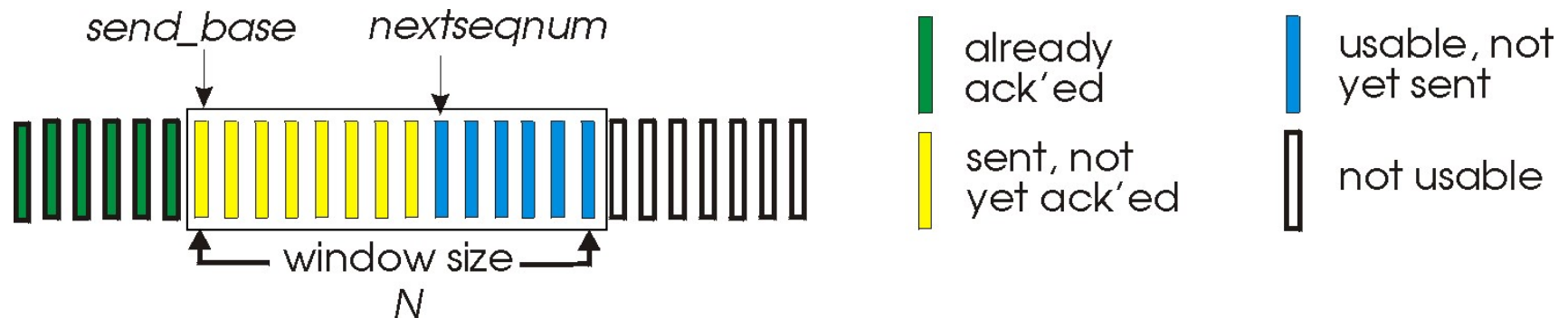
- ❖ sender can have up to N unacked packets in pipeline
- ❖ receiver only sends *cumulative ack*
  - doesn't ack packet if there's a gap
- ❖ sender has timer for oldest unacked packet
  - when timer expires, retransmit *all* unacked packets

## Selective Repeat:

- ❖ sender can have up to N unack'ed 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

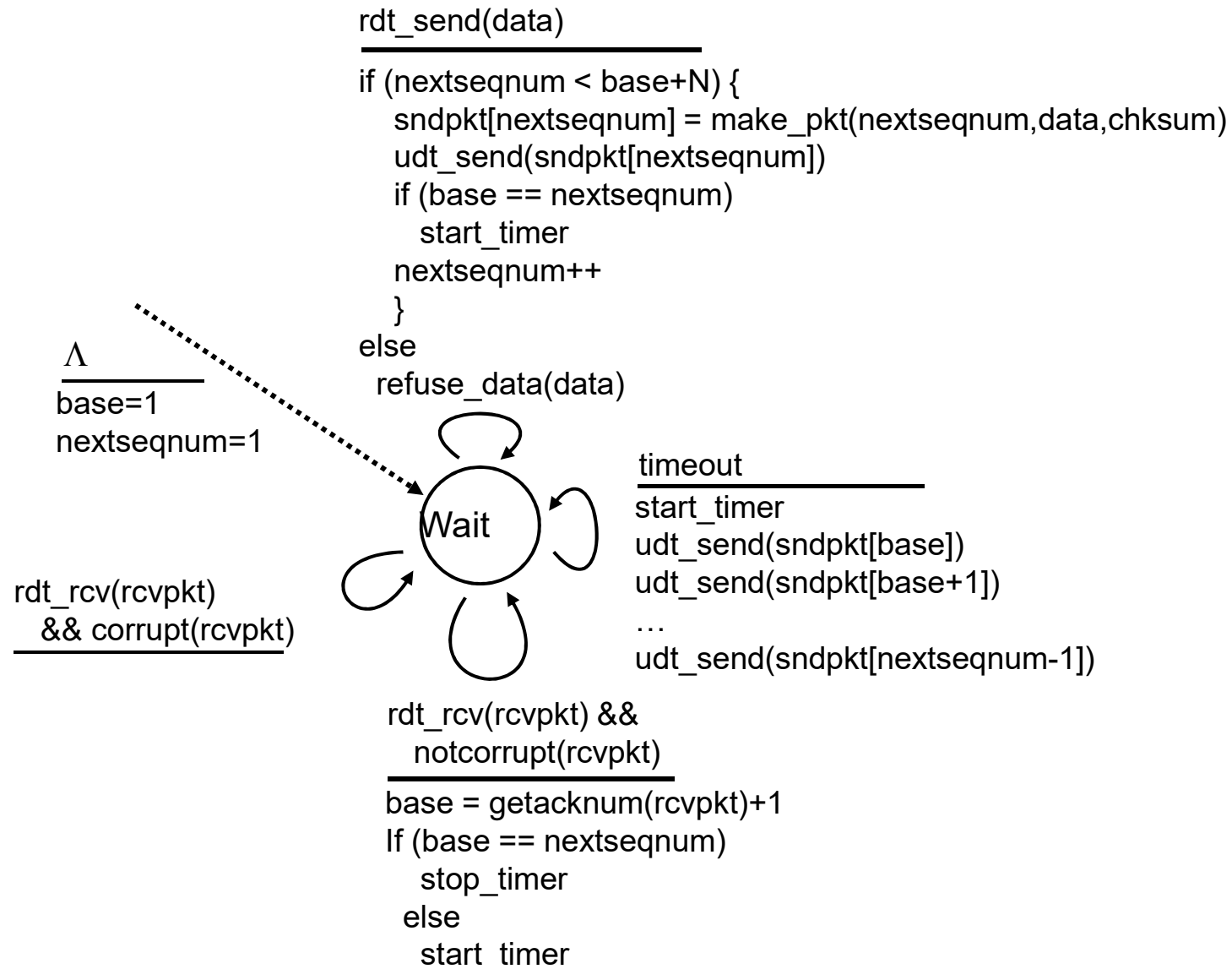
# Go-Back-N: sender

- ❖ k-bit seq # in pkt header
- ❖ “window” of up to N, consecutive unack’ed pkts allowed

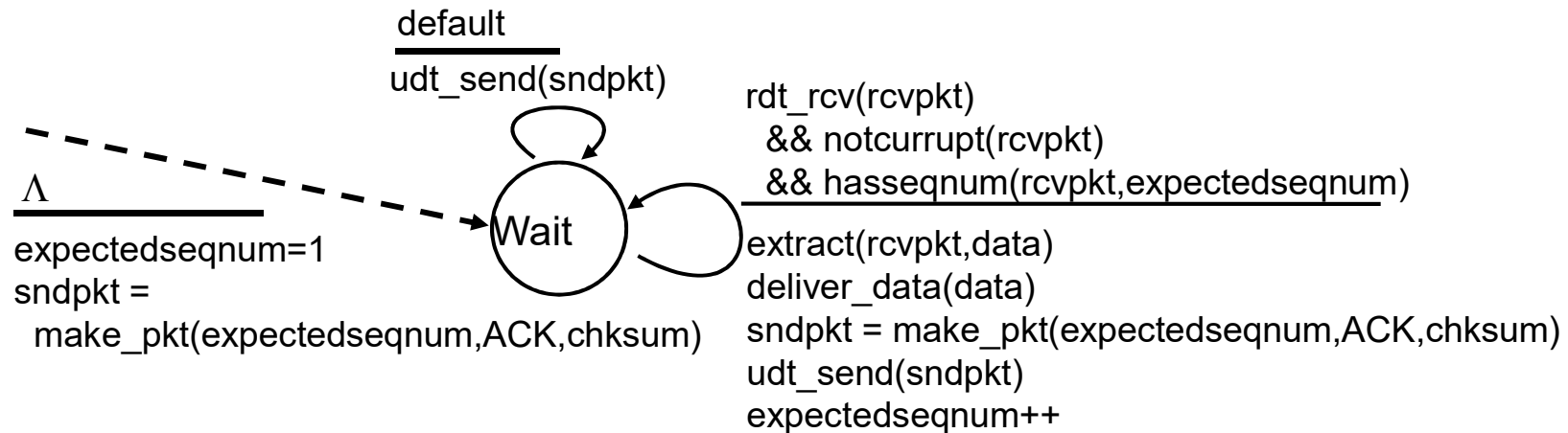


- ❖ ACK(n): ACKs all pkts up to, including seq # n - “cumulative ACK”
  - may receive duplicate ACKs (see receiver)
- ❖ timer for oldest in-flight pkt
- ❖ timeout(n): retransmit packet n and all higher seq # pkts in window

# 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

sender window (N=4)

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

sender

send pkt0

send pkt1

send pkt2

send pkt3

(wait)

rcv ack0, send pkt4

rcv ack1, send pkt5

ignore duplicate ACK



*pkt 2 timeout*

send pkt2

send pkt3

send pkt4

send pkt5

receiver

receive pkt0, send ack0

receive pkt1, send ack1

receive pkt3, discard,  
(re)send ack1

receive pkt4, discard,  
(re)send ack1

receive pkt5, discard,  
(re)send ack1

rcv pkt2, deliver, send ack2

rcv pkt3, deliver, send ack3

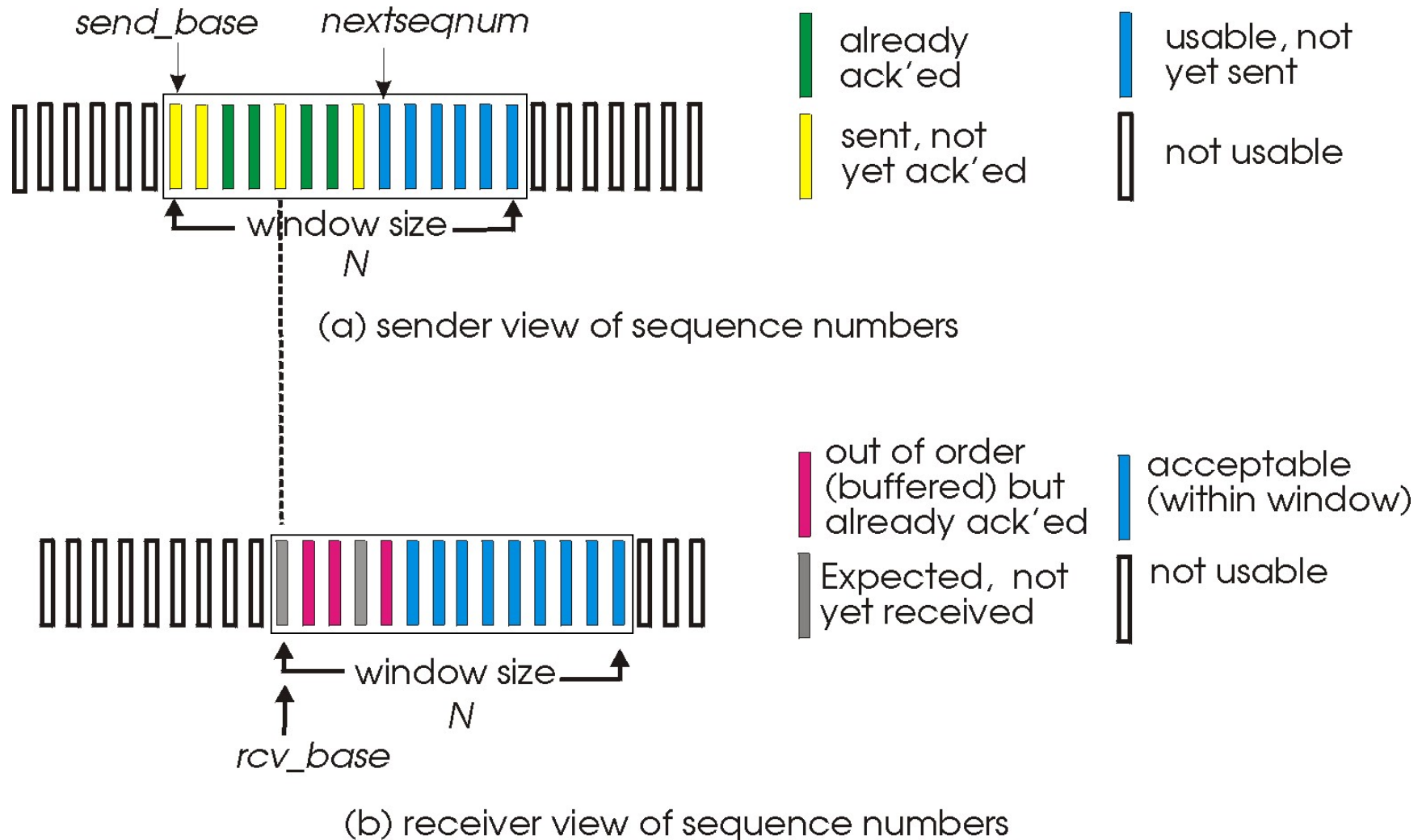
rcv pkt4, deliver, send ack4

rcv pkt5, deliver, send ack5

# 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]

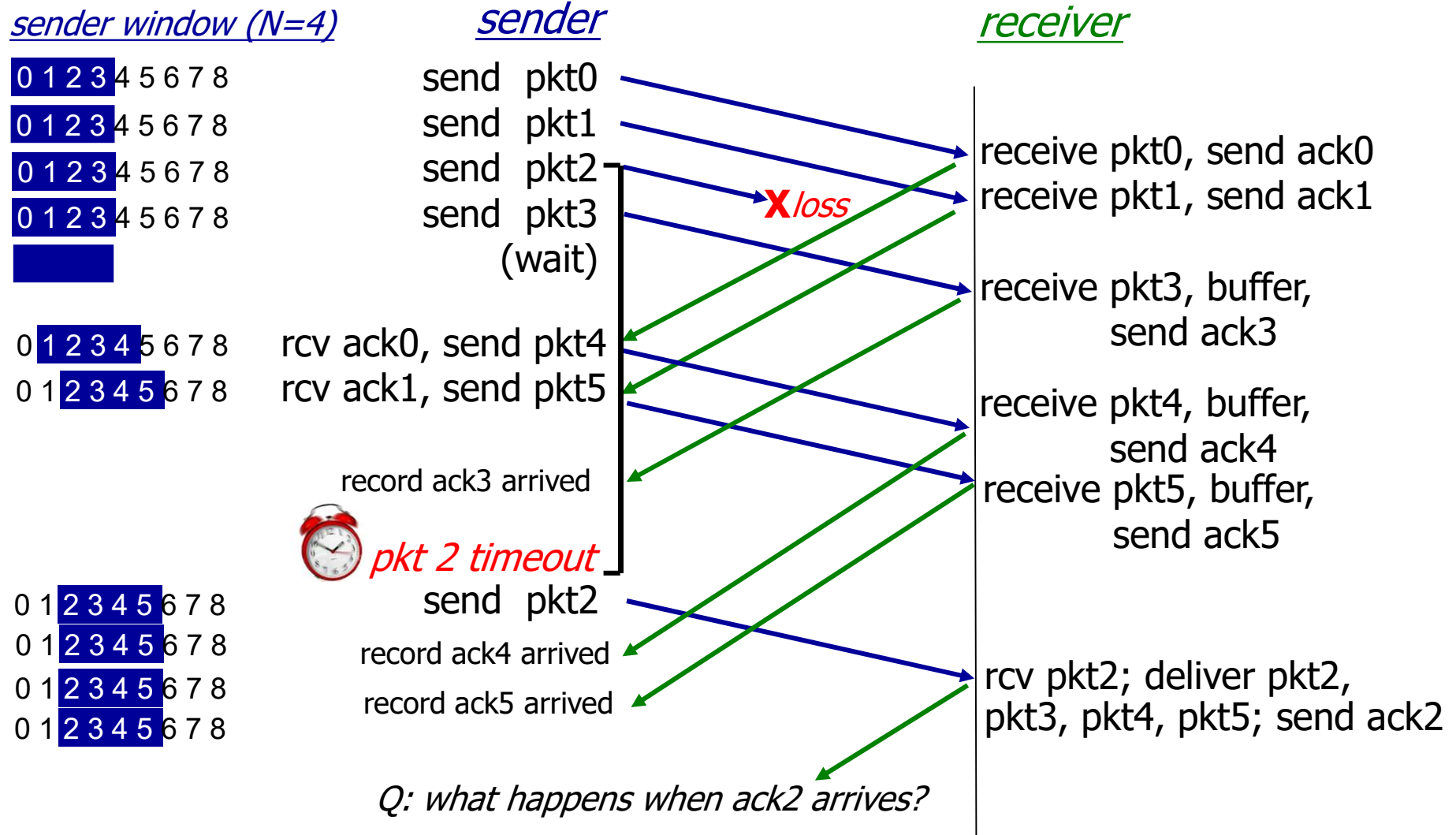
- ❖ ACK(n)

### otherwise:

- ❖ ignore



# Selective repeat in action

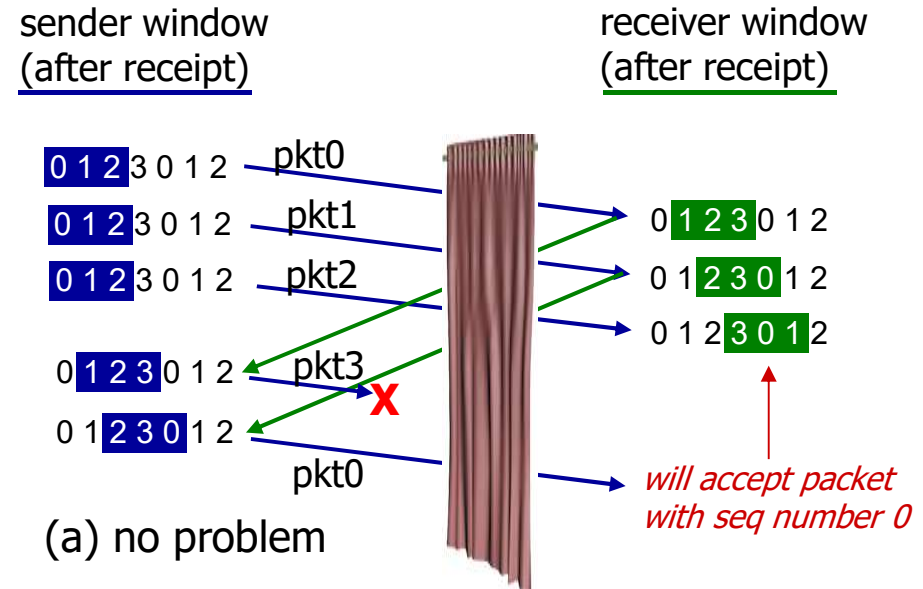


# Selective repeat: dilemma

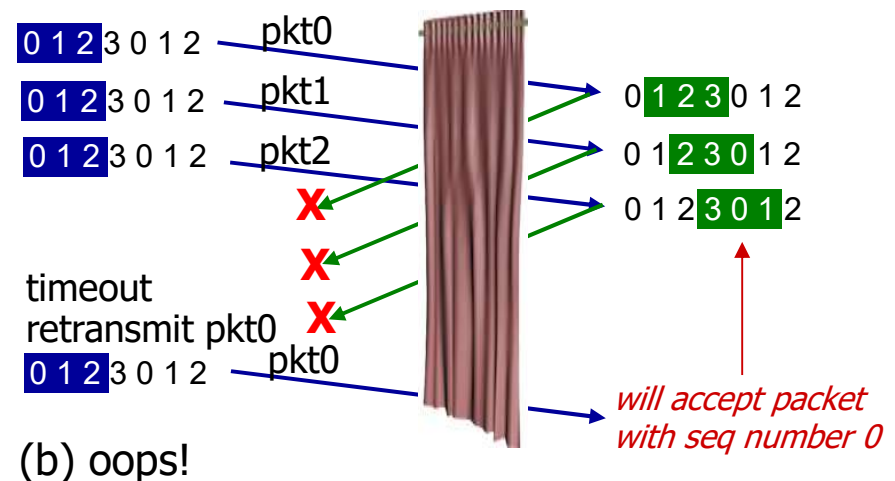
example:

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

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



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



# Sequence number

- ❖ Related to Sender window size - SWS
- ❖ Related to Receiver window size - RWS
- ❖ Minimum sequence number =  $(SWS + RWS)$
- ❖ Mostly SWS is equal to RWS.  $SWS > RWS$  does not help.
- ❖ Number of bits for sequence number
  - $\text{Ceil}(\log_2(\text{minimum sequence number}))$

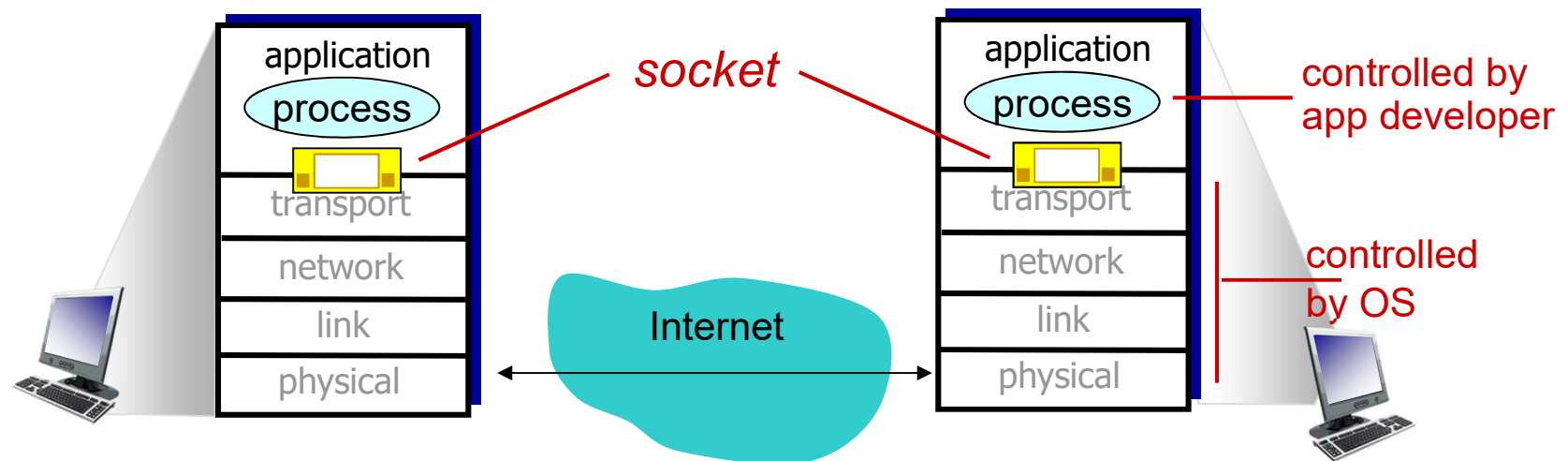
## Network Application Programming

### **Sections 2.1.2, 2.7**

Computer Networking – A top-down approach,  
Kurose and Ross, 6<sup>th</sup> Edition.

# Sockets

- ❖ process sends/receives messages to/from its **socket**
- ❖ socket analogous to door
  - sending process shoves message out door
  - sending process relies on transport infrastructure on other side of door to deliver message to socket at receiving process



# Addressing processes

- ❖ to receive messages, process must have *identifier*
- ❖ host device has unique 32-bit IP address
- ❖ Q: does IP address of host on which process runs suffice for identifying the process?
  - A: no, *many* processes can be running on same host
- ❖ *identifier* includes both **IP address** and **port numbers** associated with process on host.
- ❖ example port numbers:
  - HTTP server: 80
  - mail server: 25
- ❖ to send HTTP message to gaia.cs.umass.edu web server:
  - **IP address**: 128.119.245.12
  - **port number**: 80
- ❖ more shortly...

# Socket programming *with TCP*

## *client must contact server*

- ❖ server process must first be running
- ❖ server must have created socket (door) that welcomes client's contact

## *client contacts server by:*

- ❖ Creating TCP socket, specifying IP address, port number of server process
- ❖ *when client creates socket:* client TCP establishes connection to server TCP

- ❖ when contacted by client, *server TCP creates new socket* for server process to communicate with that particular client
  - allows server to talk with multiple clients
  - source port numbers used to distinguish clients (more in Chap 3)

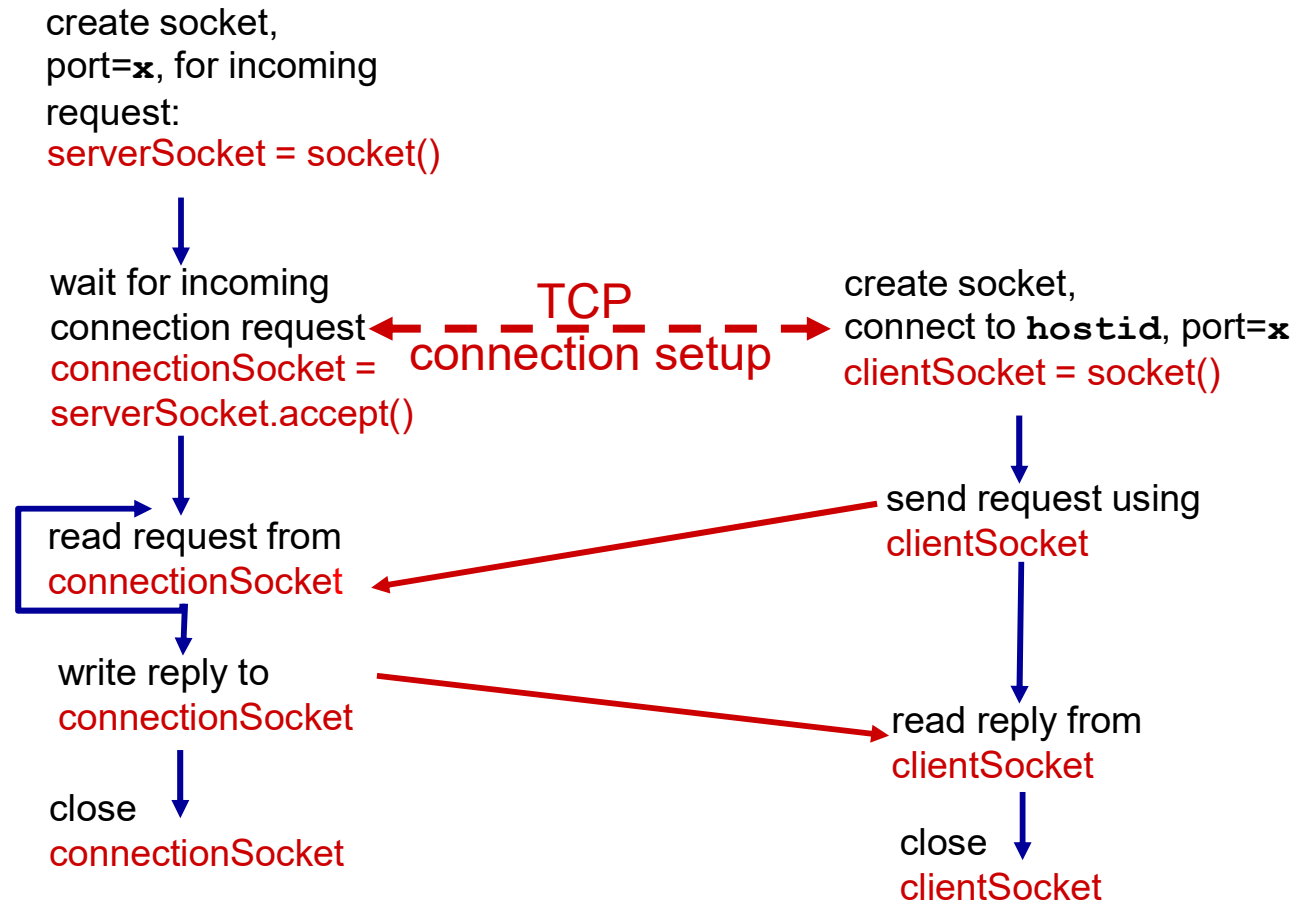
## *application viewpoint:*

TCP provides reliable, in-order byte-stream transfer (“pipe”) between client and server

# Client/server socket interaction: TCP

server (running on `hostid`)

client





# Example app:TCP client

## *Python TCPClient*

create TCP socket for  
server, remote port 12000

```
from socket import *
serverName = 'servername'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName,serverPort))
sentence = raw_input('Input lowercase sentence:')
clientSocket.send(sentence)
modifiedSentence = clientSocket.recv(1024)
print 'From Server:', modifiedSentence
clientSocket.close()
```

No need to attach server  
name, port

# Example app: TCP server

## *Python TCPServer*

create TCP welcoming socket	→	from socket import *
		serverPort = 12000
server begins listening for incoming TCP requests	→	serverSocket = socket(AF_INET, SOCK_STREAM)
		serverSocket.bind(('', serverPort))
		serverSocket.listen(1)
		print 'The server is ready to receive'
loop forever	→	while 1:
server waits on accept() for incoming requests, new socket created on return	→	connectionSocket, addr = serverSocket.accept()
read bytes from socket (but not address as in UDP)	→	sentence = connectionSocket.recv(1024)
		capitalizedSentence = sentence.upper()
		connectionSocket.send(capitalizedSentence)
close connection to this client (but <i>not</i> welcoming socket)	→	connectionSocket.close()