# Transport Layer

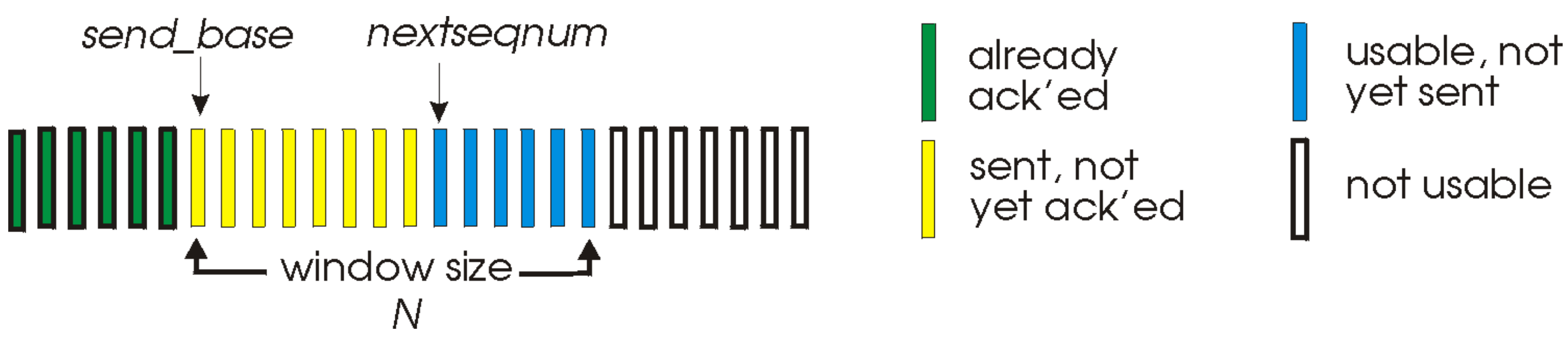
Instructor: C. Pu (Ph.D., Assistant Professor)

Lecture 10

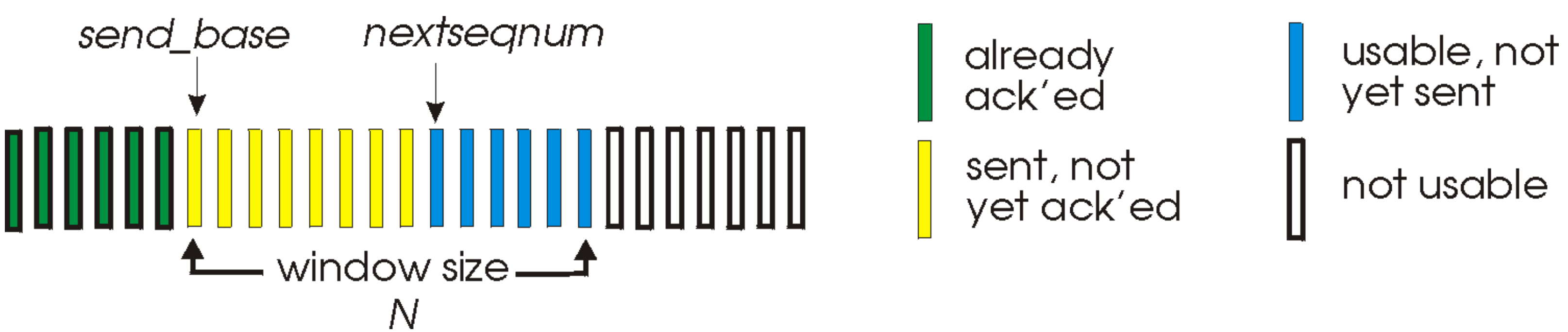
*puc@marshall.edu*

# Go-Back-N

- Sender:
  - allow to transmit *multiple packets without waiting for ACK*
    - constrained to ***N*** unack'ed packets in the pipeline
- Sender's view of the sequence numbers in GBN



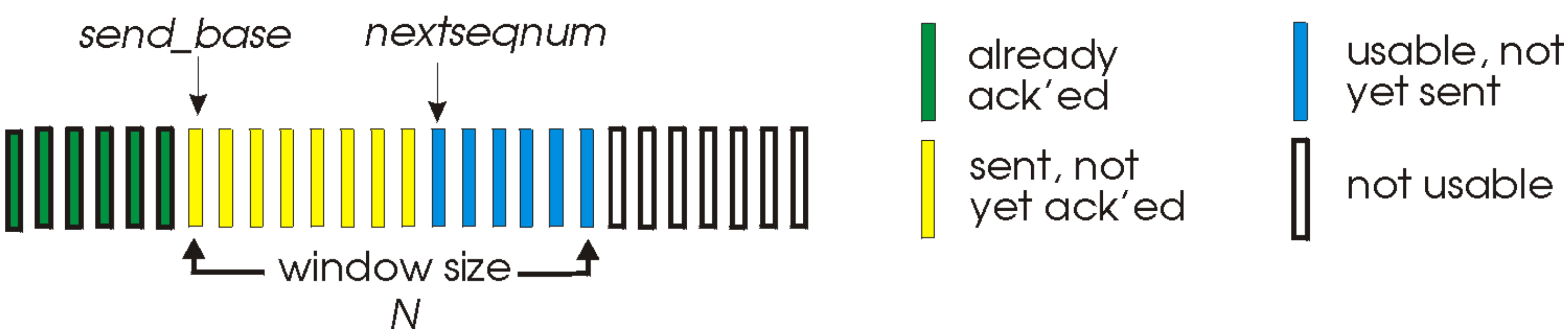
# Go-Back-N



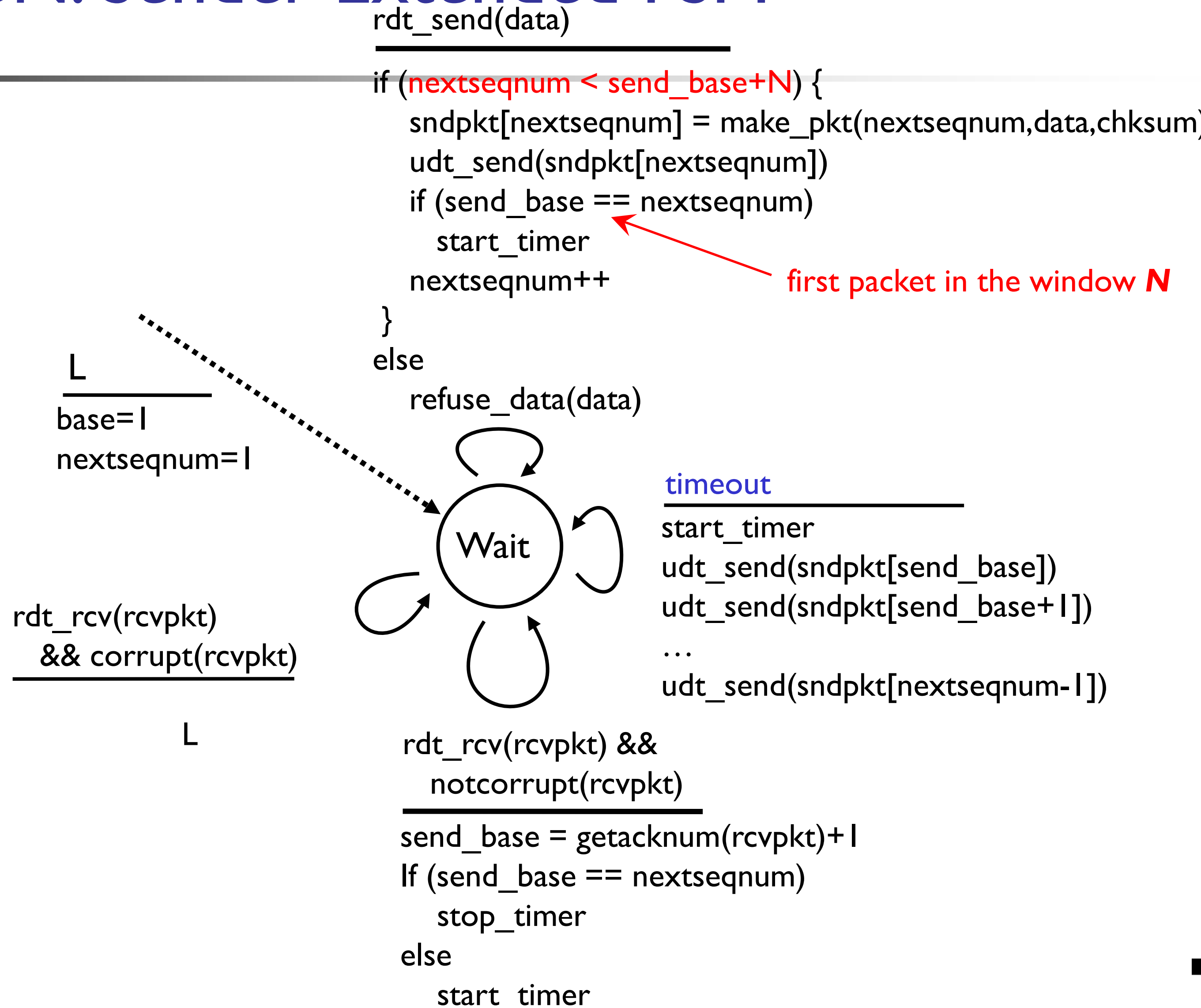

- *send_base*: the seq. number of the ***oldest unack'ed packet***
- *nextseqnum:* the ***smallest unused seq. number***
- [0, send_base - 1]: packets that have already been ***transmitted*** and ***acked***
- [send_base, nextseqnum - 1]: packets that have been ***sent but not yet acked***
- [nextseqnum, send_base + N -1]: packets can be sent immediately

# Go-Back-N

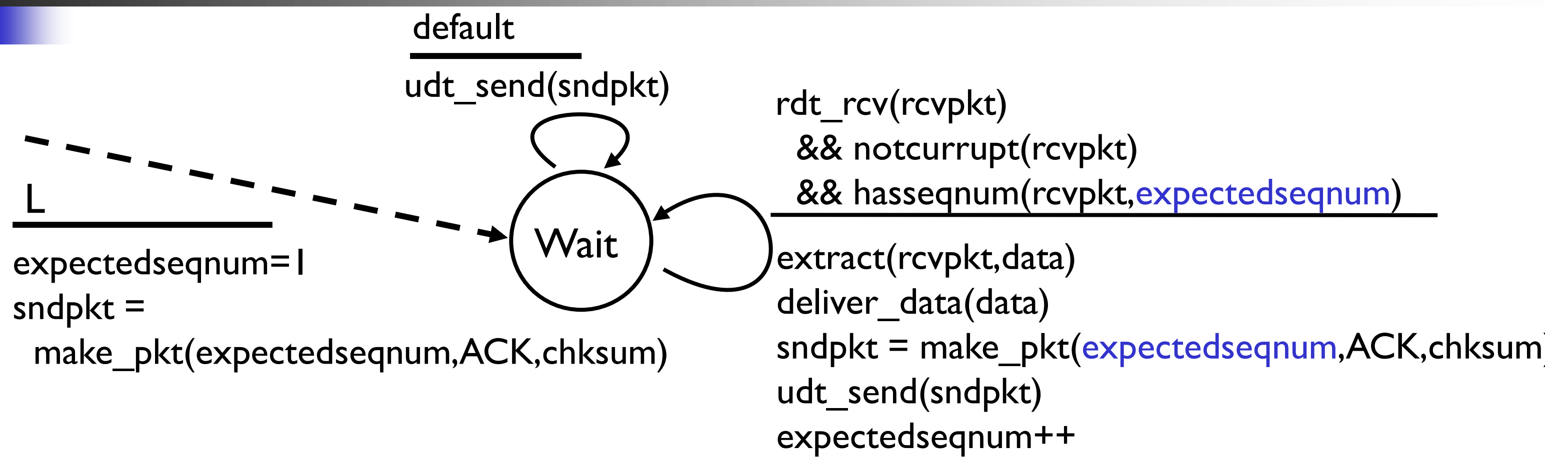
- Sender:
  - allow to transmit *multiple packets without waiting for ACK*
    - constrained to ***N*** unack'ed packets in the pipeline
  - "window" of up to ***N***, consecutive unack'ed pkts allowed
  - seq. number is carried in a fixed-length field in the packet header
  - k-bit seq # in pkt header
    - $[0, 2^k - 1]$
    - e.g., TCP: 32-bits seq #



# GBN: Sender Extended FSM



# GBN: Receiver Extended FSM

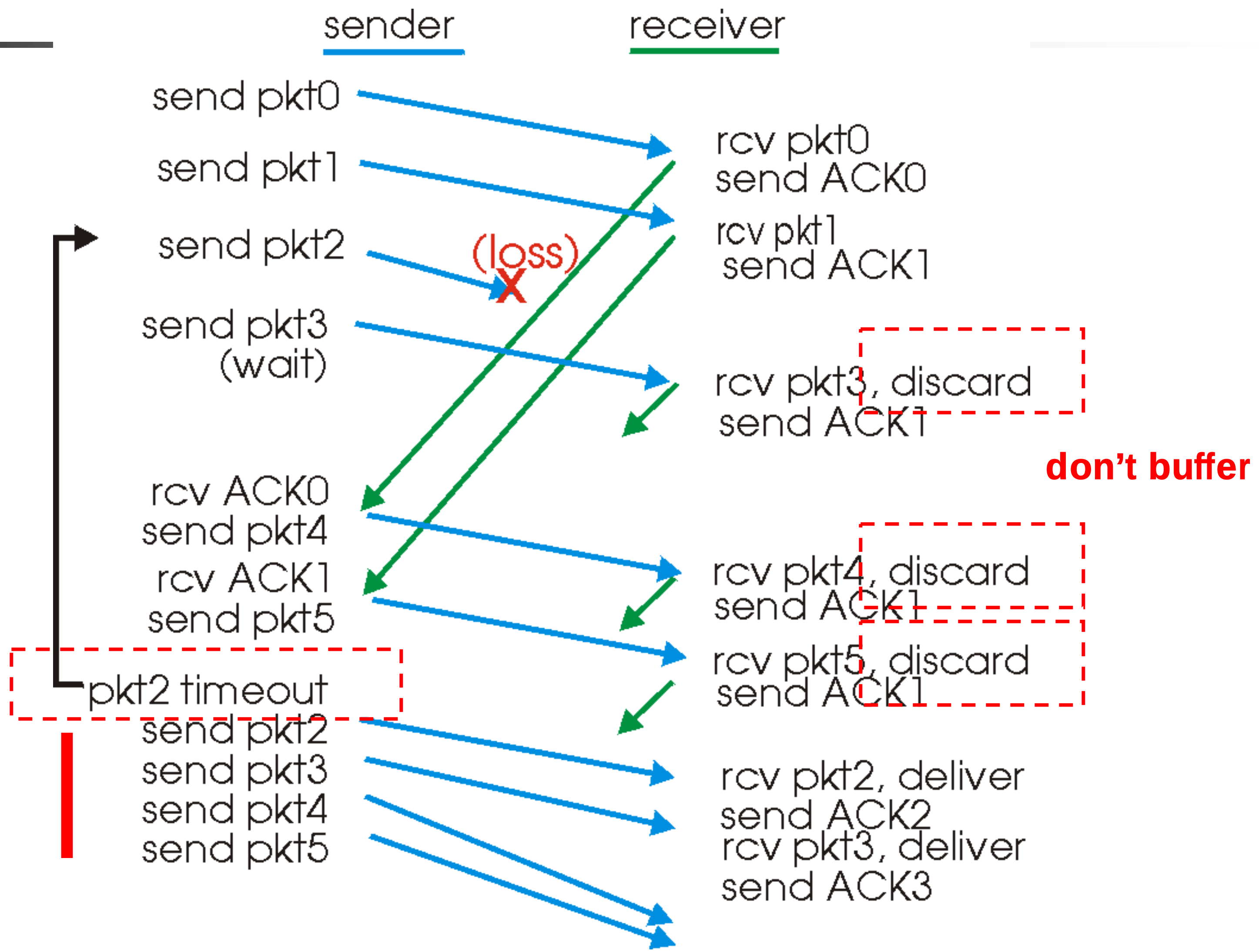



- ACK(***n***): ***ACKs all pkts up to, including seq # n*** - "***cumulative ACK***"
  - may receive duplicate ACKs (see receiver)
- timer for each in-flight pkt
- *timeout(**n**):* retransmit pkt ***n*** and all higher seq # pkts in window
- ACK-only: always send ACK for correctly-received pkt with highest *in-order* seq #
  - may generate duplicate ACKs
  - need only remember **expectedseqnum**

# GBN: Receiver Extended FSM (cont.)

- out-of-order pkt:
  - packet *n* is expected, but packet *n+1* arrives
  - discard (don't buffer) -> no receiver buffering! why?
    - based on GBN, the sender transmit the packet
    - adv.?
      - simplicity of receiver buffering
    - disadv.?
      - throwing away a correctly received packet
      - subsequent retransmission of that packet might be lost or garbled
      - even more retransmission would be required

# GBN in Action



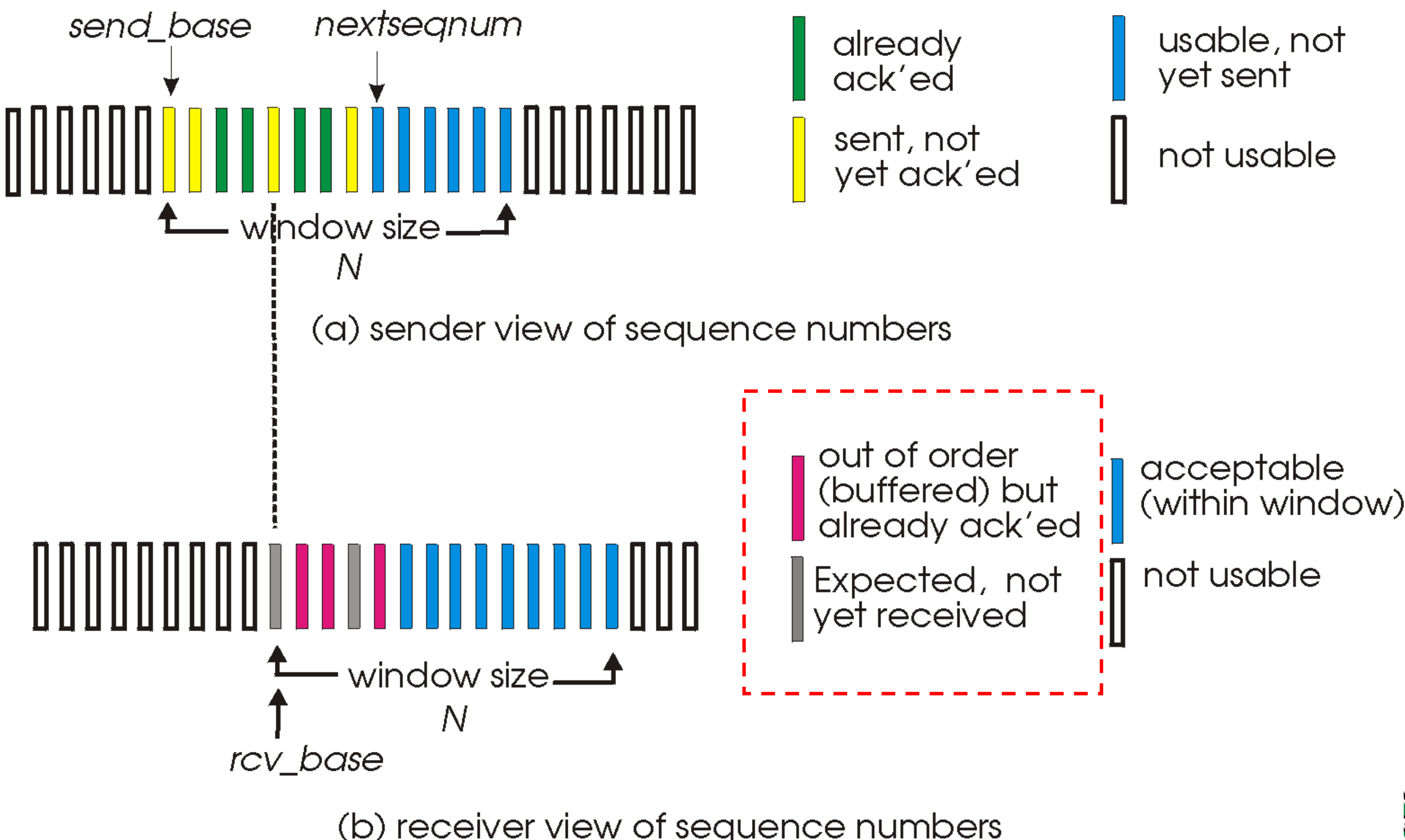
# Selective Repeat

- In GBN protocol,
  - allow the sender to potentially "fill the pipeline"
  - avoiding the channel utilization problem
  - → suffer from performance problem, e.g. many packets in the pipeline, but a single packet error?
    - retransmit a large number of packets (many unnecessarily)

# Selective Repeat (cont.)

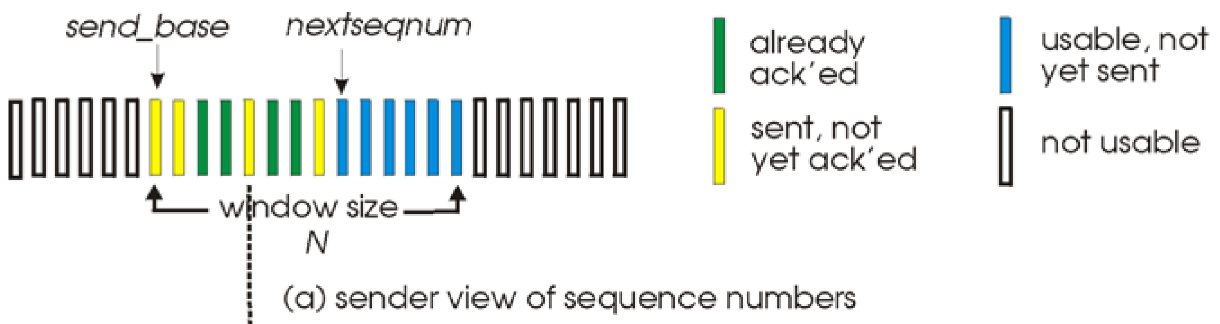
- selective-repeat protocol *avoids unnecessary retransmissions* by having the sender retransmit only those packets that it suspects were received in error at the receiver
- receiver *individually acknowledges all correctly received pkts*, whether or not they are in order
  - buffers pkts, as needed, 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
  - again limits seq #s of sent, unACKed pkts

# Selective Repeat: Sender, Receiver Windows



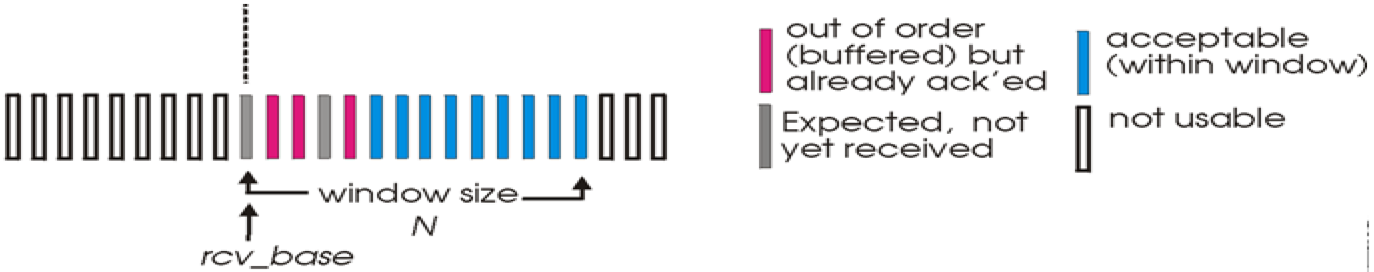
# Selective Repeat (cont.)

- Sender:
  - data received from above :
    - if next available seq # in window, send pkt
  - timeout(n):
    - resend pkt n, restart timer
  - ACK(n) in [send_base, send_base+N-1]:
    - mark pkt n as received
    - If the packet's sequence# == send_base
      - advance the window base to the unacked packet with the smallest sequence #



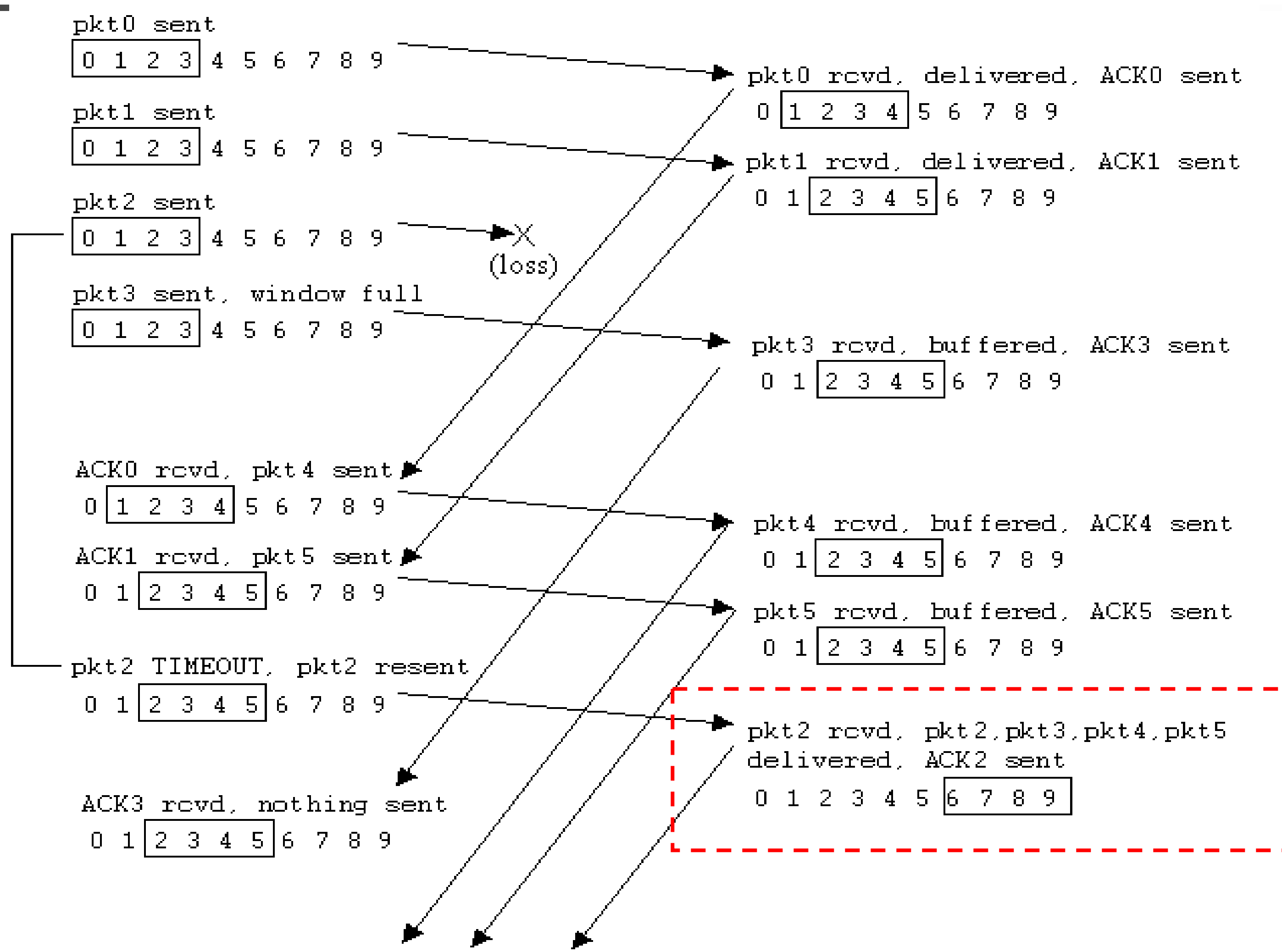
# Selective Repeat (cont.)

- Receiver:
  - pkt n in [rcv_base, rcv_base+N-1]
    - send ACK(n)
    - out-of-order: buffering
    - in-order: deliver (also deliver buffered, in-order pkts), advance window to next not-yet-received pkt
  - pkt n in [rcv_base-N, rcv_base-1]
    - ACK(n)
    - although, the receiver has previously acknowledged
  - otherwise: ignore the packet




(b) receiver view of sequence numbers

# Selective repeat in Action



# Pipelined Protocols: Summary

**Go-back-N:**

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

**Selective Repeat:**

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

# TCP: Overview
# RFCs: 793, 1122, 1323, 2018, 2581

- TCP is said to be connection-oriented
  - before one application process can begin to send data to another
  - the two processes must first "*handshake*" with each other
    - send some preliminary segments to each other to establish the parameters of the ensuing data transfer
  - both sides of the connection will initialize many TCP state variables associated with the TCP connection

# TCP: Overview
# RFCs: 793, 1122, 1323, 2018, 2581

- The TCP "connection" is not an end-to-end TDM or FDM circuit as in a circuits switched network
  - the connection state resides entirely in the two end systems
    - because the TCP protocol runs only in the end systems and not in the intermediate network elements (routers and link-layer switches)
    - the intermediate network elements do not maintain TCP connection state

# TCP: Overview
# RFCs: 793, 1122, 1323, 2018, 2581

- A TCP connection provides a *full-duplex service*
  - if there is a TCP connection between Process A on one host and Process B on another host
  - then application layer data can flow from Process A to Process B at the same time as application layer data flows from Process B to Process A
- A TCP connection is always *point-to-point*:
  - one sender and one receiver
- *multicasting*
  - the transfer of data from one sender to many receivers in a single send operation
  - not possible with TCP

# TCP: How a TCP Connection is Established

- suppose a process running in one host wants to initiate a connection with another process in another host
  - *client process*
    - *the process that is initiating the connection*
  - *server process*
    - *the other process*
  - ***three-way handshake***

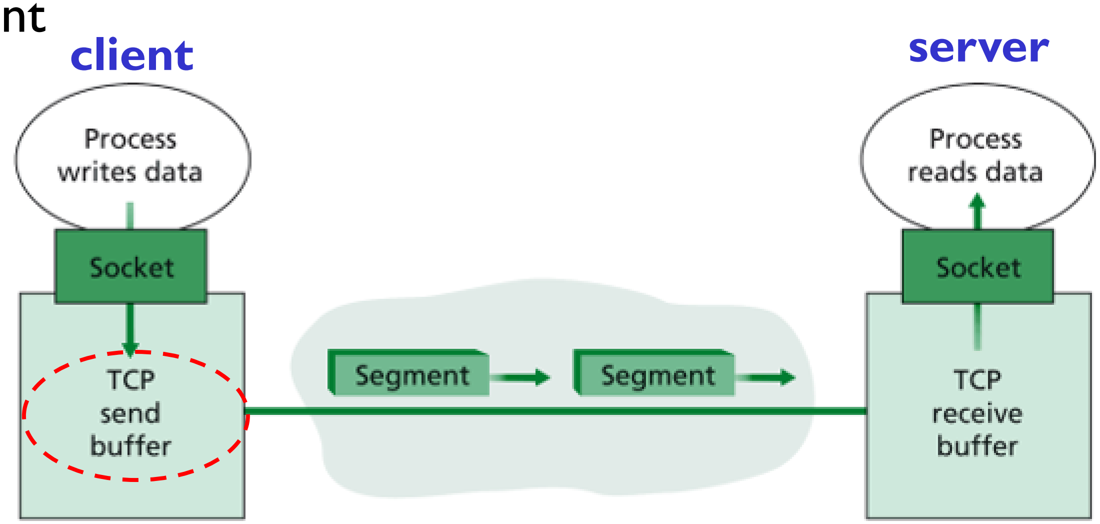
1. *the client first sends a special TCP segment*
2. *the server responds with a second special TCP segment*
3. *the client responds again with a third special segment*
    - the first two segments carry no payload
    - the third of these segments may carry a payload

# TCP: How a TCP Connection is Established

- Once a TCP connection is established, the two application processes can send data to each other
  - the client process passes a stream of data through the socket
  - once the data passes through the door, the data is in the hands of TCP running in the client

send buffer:

- set aside during initial 3-way handshake
- grab chunks of data from send buffer and pass the data to the network layer
- maximum segment size (MSS): the maximum amount of data that can be grabbed and placed in a segment
  - MSS is typically set by first determining the length of the largest link-layer frame that can be sent by the sending host

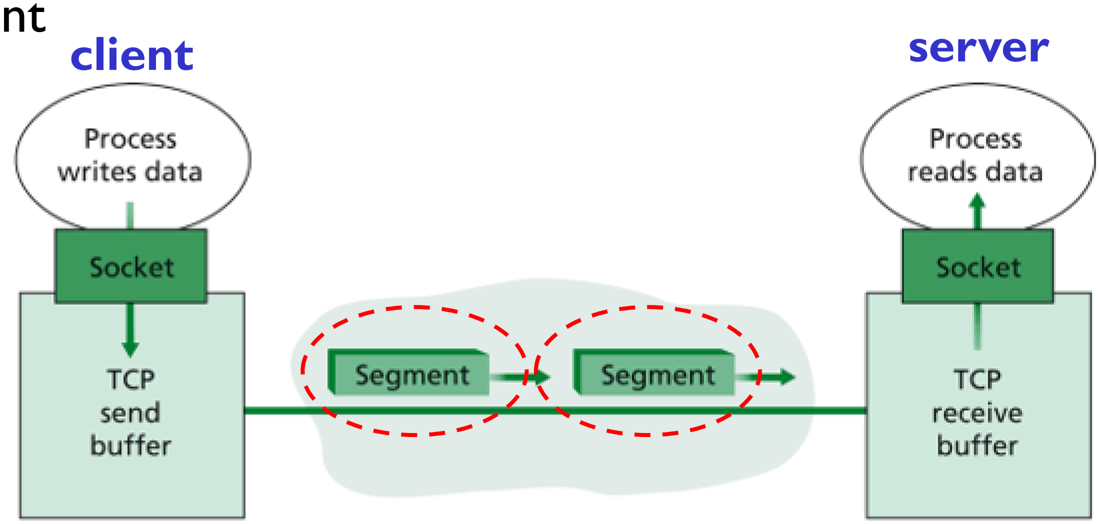



TCP send and receive buffers

# TCP: How a TCP Connection is Established

- Once a TCP connection is established, the two application processes can send data to each other
  - the client process passes a stream of data through the socket
  - once the data passes through the door, the data is in the hands of TCP running in the client

segment:
- TCP pairs each chunk of client data with a TCP header, thereby forming TCP segments



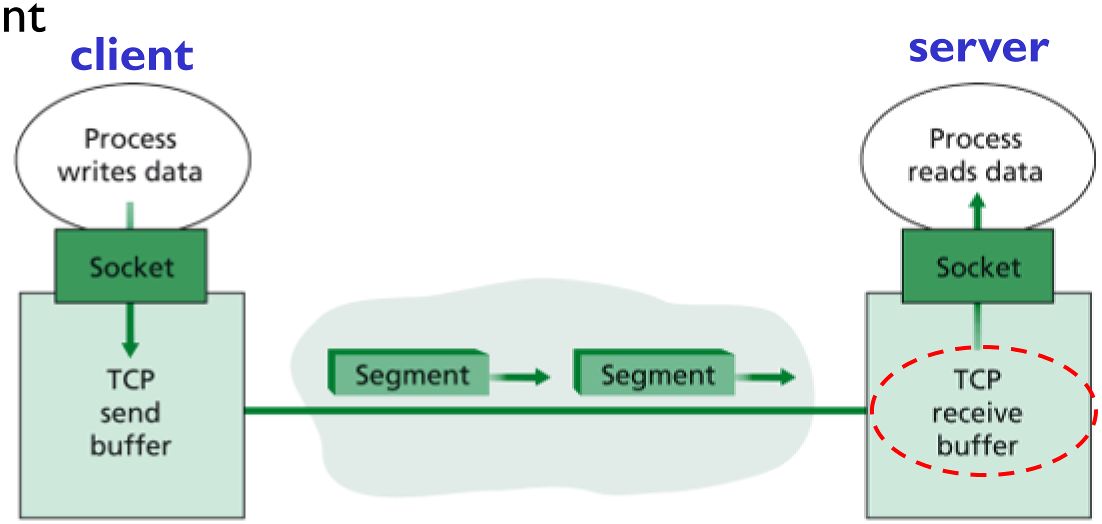

TCP send and receive buffers

# TCP: How a TCP Connection is Established

- Once a TCP connection is established, the two application processes can send data to each other
  - the client process passes a stream of data through the socket
  - once the data passes through the door, the data is in the hands of TCP running in the client

receiver buffer:

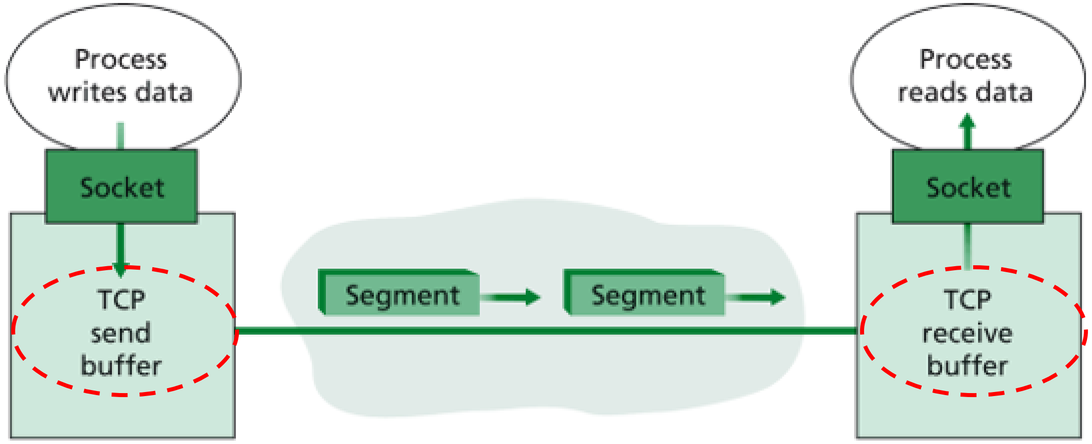
- when TCP receives a segment at the other end, the segment's data is placed in the TCP connection's receive buffer
- app. reads the stream of data from buffer




TCP send and receive buffers

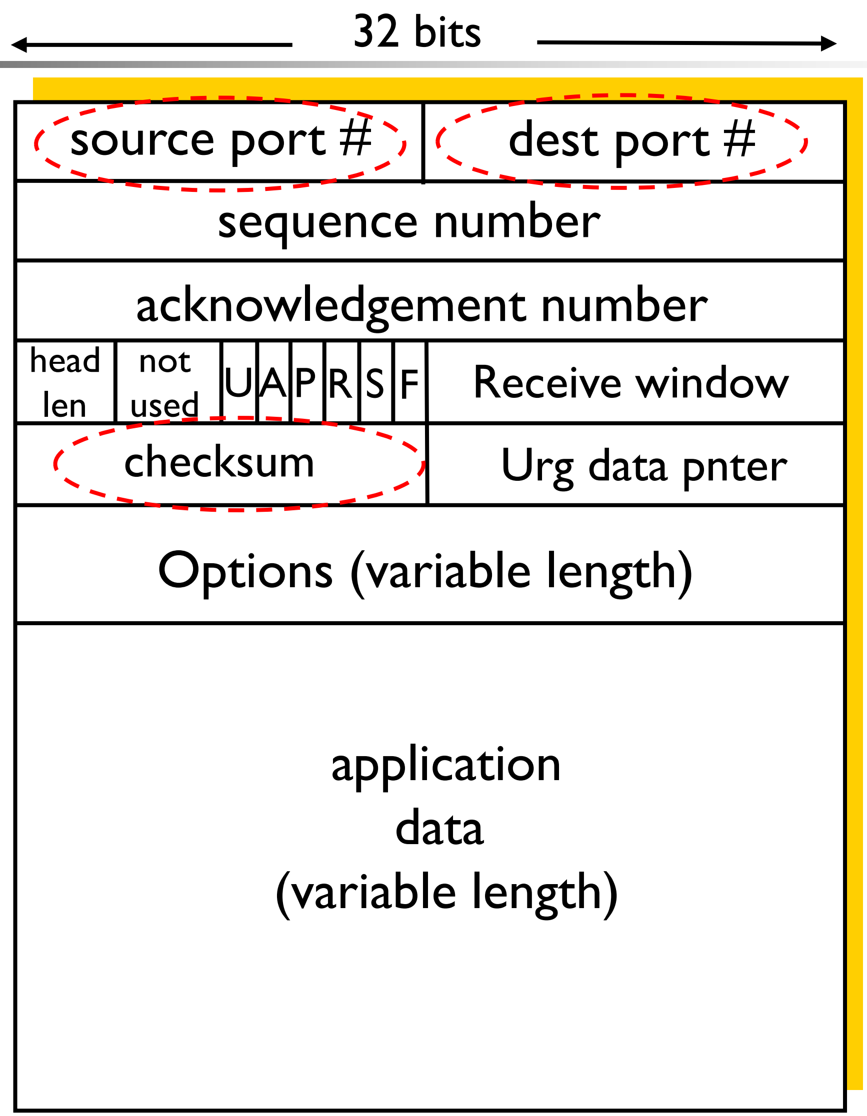
# TCP: How a TCP Connection is Established

- Once a TCP connection is established, the two application processes can send data to each other
  - the client process passes a stream of data through the socket
  - once the data passes through the door, the data is in the hands of TCP running in the client
- each side of the connection has its own send buffer and its own receive buffer



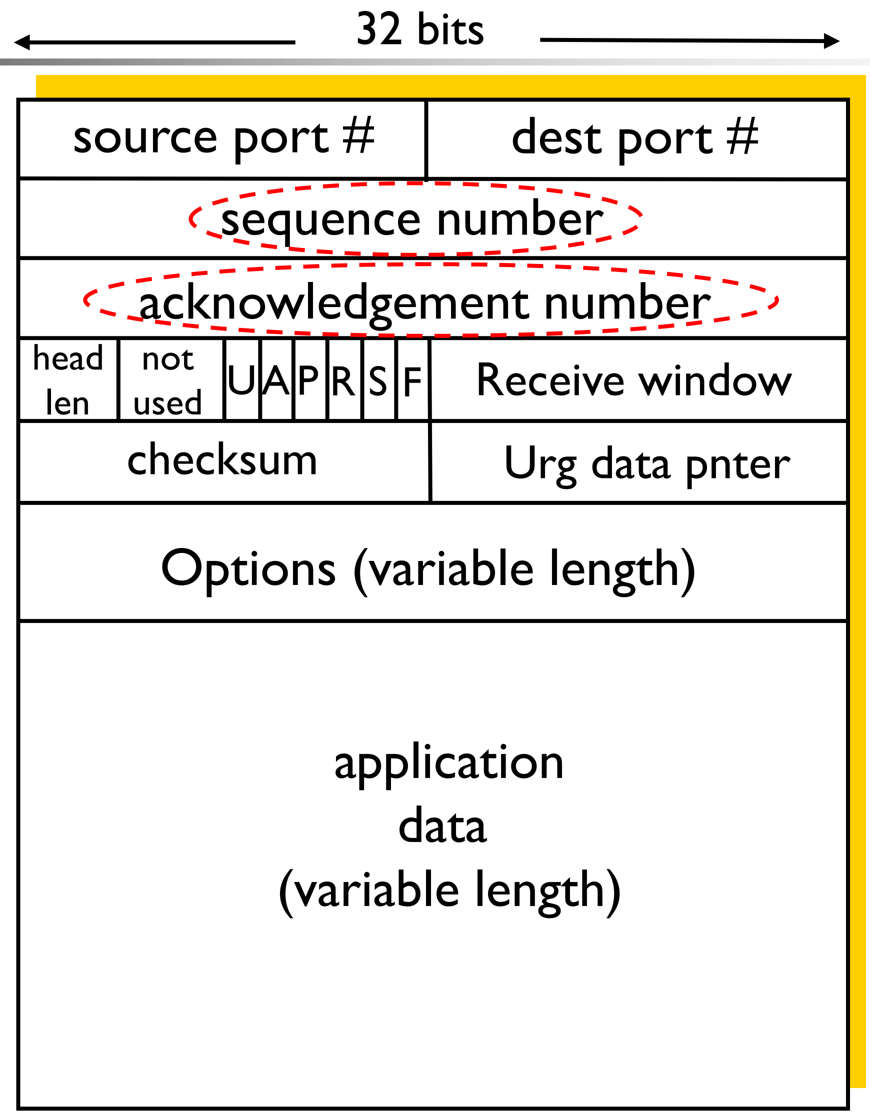

TCP send and receive buffers

# TCP Segment Structure

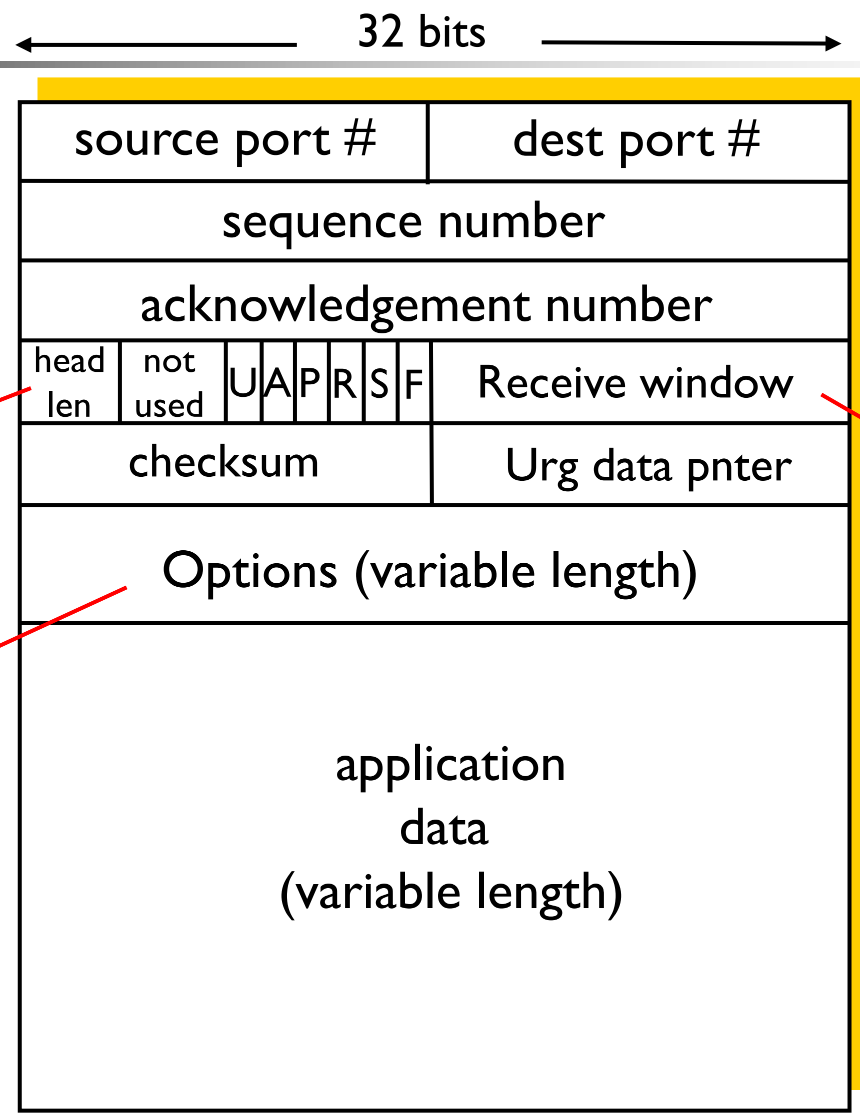


# TCP Segment Structure

- 32-bit sequence number field
- 32-bit acknowledgment number field



# TCP Segment Structure

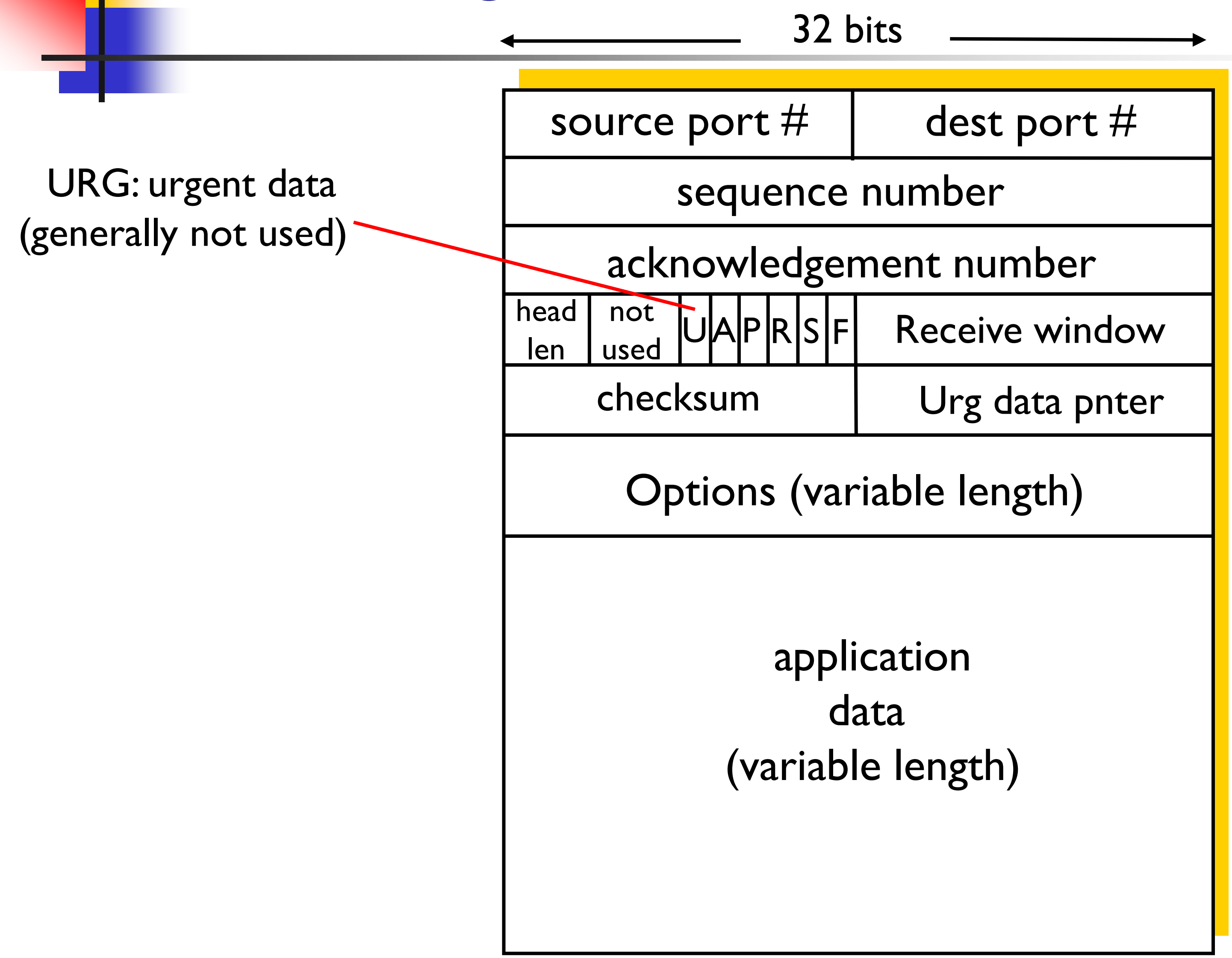



header length field: the length of the TCP header in 32-bit words

used when a sender and receiver negotiate the maximum segment size (MSS)

flow control : # bytes rcvr willing to accept

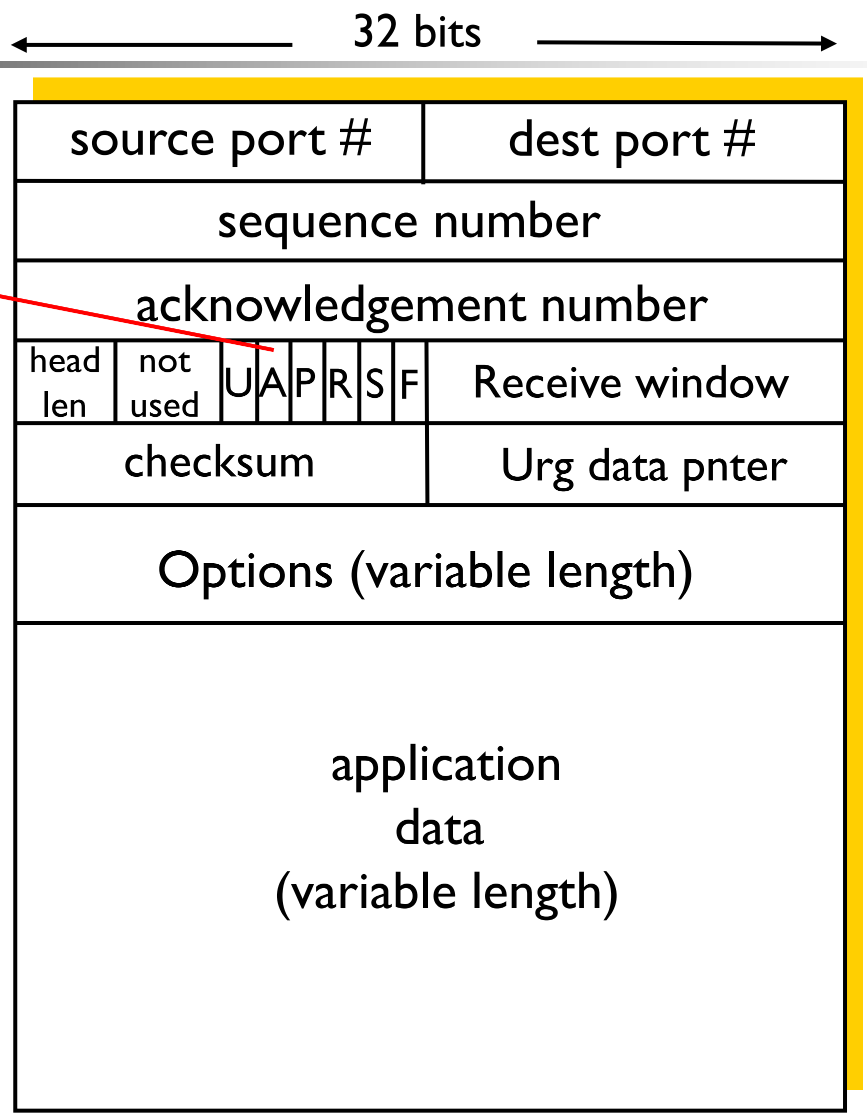
# TCP Segment Structure



TCP segment: header fields and a data field

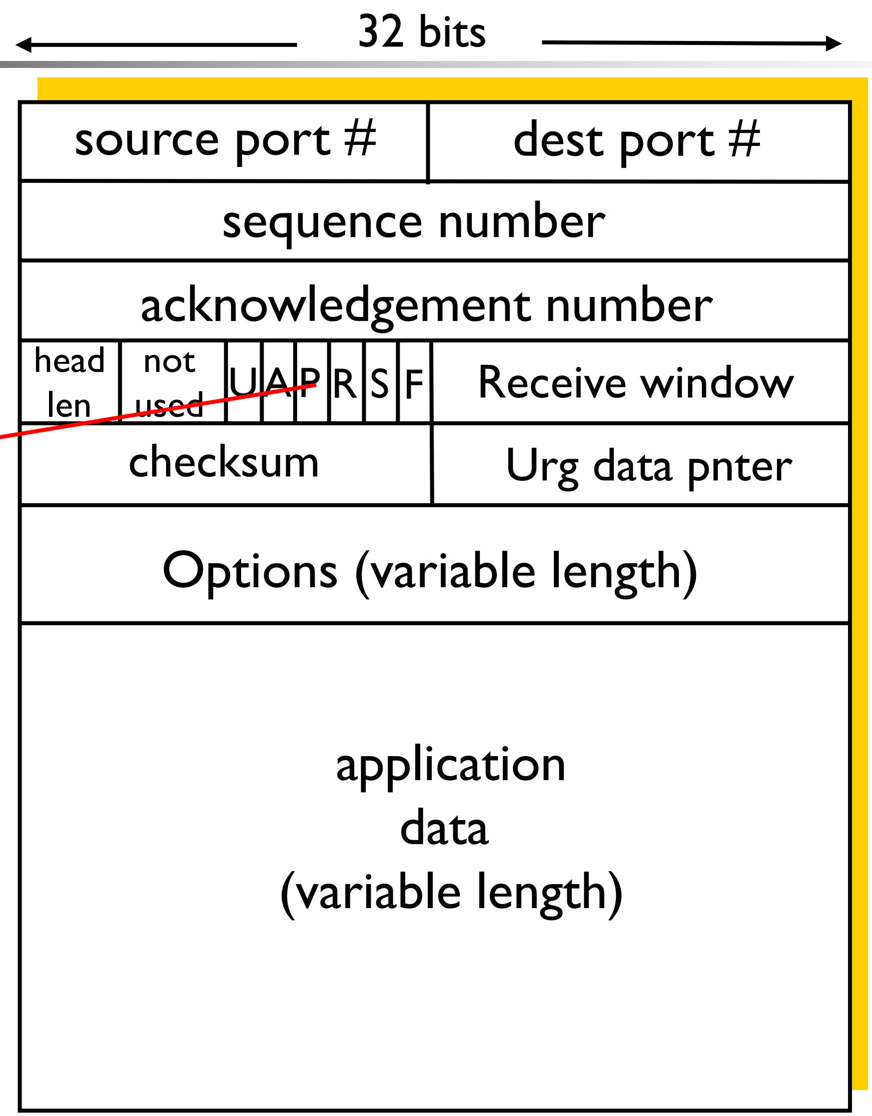
# TCP Segment Structure

ACK bit: indicate that the value carried in the acknowledgment field is valid;



TCP segment: header fields and a data field

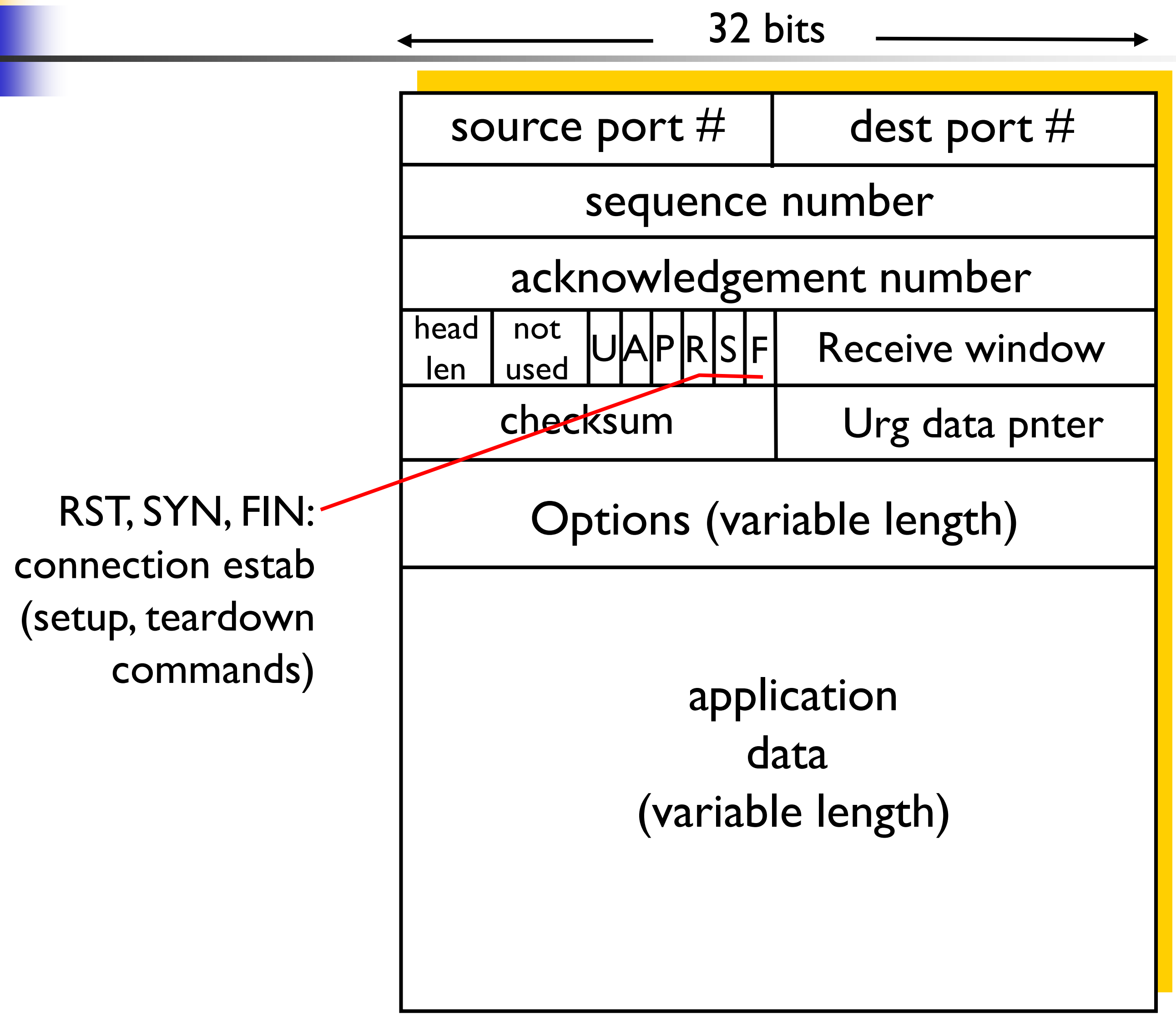
# TCP Segment Structure




PSH: push data now (generally not used)

TCP segment: header fields and a data field

# TCP Segment Structure



# TCP Segment Structure