Transport Protocols

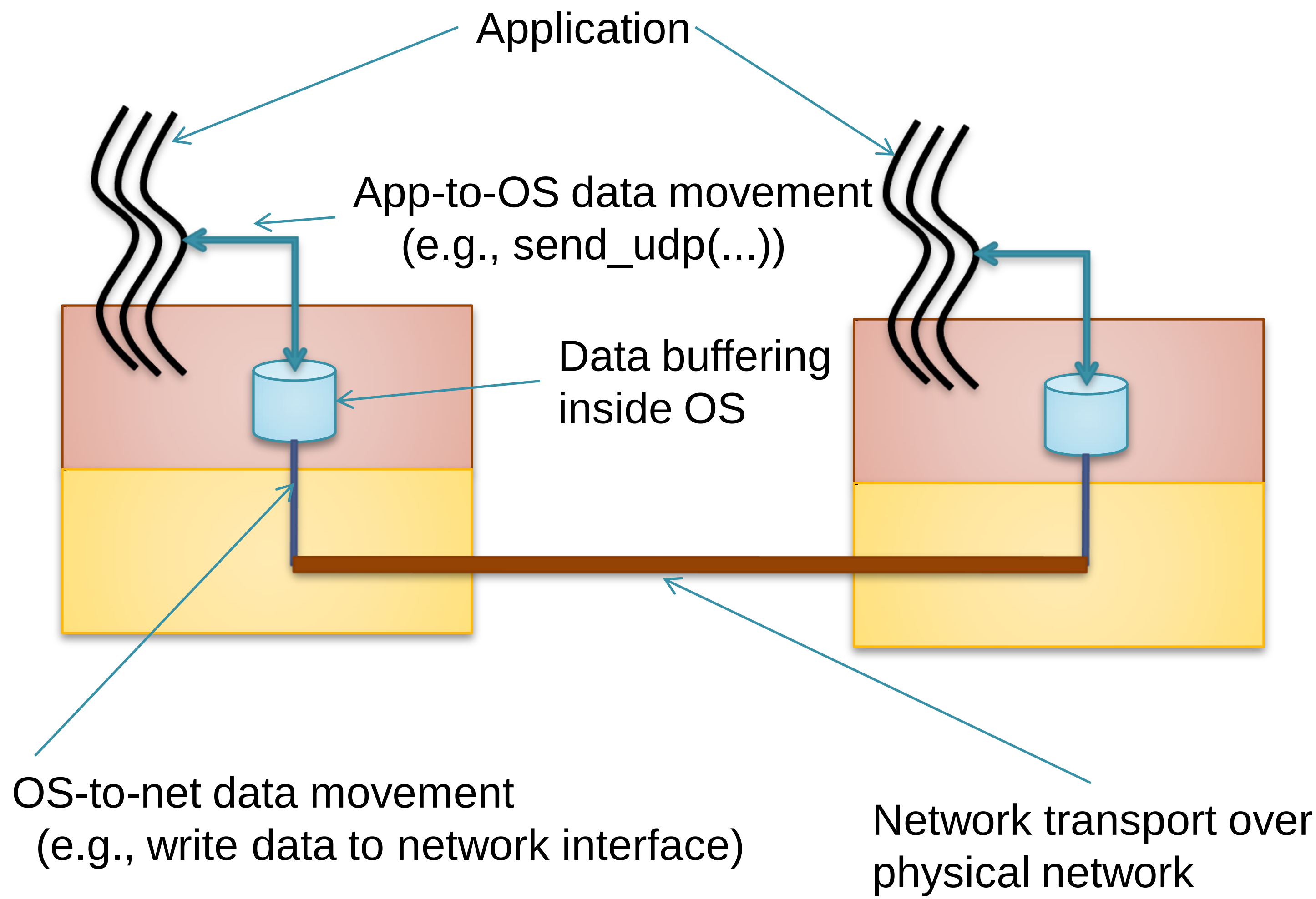
Introduction

- Protocols – what are they used for?
  - Transferring application information among machines ("transporting" protocols)
  - Naming and discovering resources and machines ("networking" protocols)
- "Transporting" protocols might be "end-to-end" protocols
- "Networking" protocols might be "hop-by-hop" protocols

“Transporting” protocols

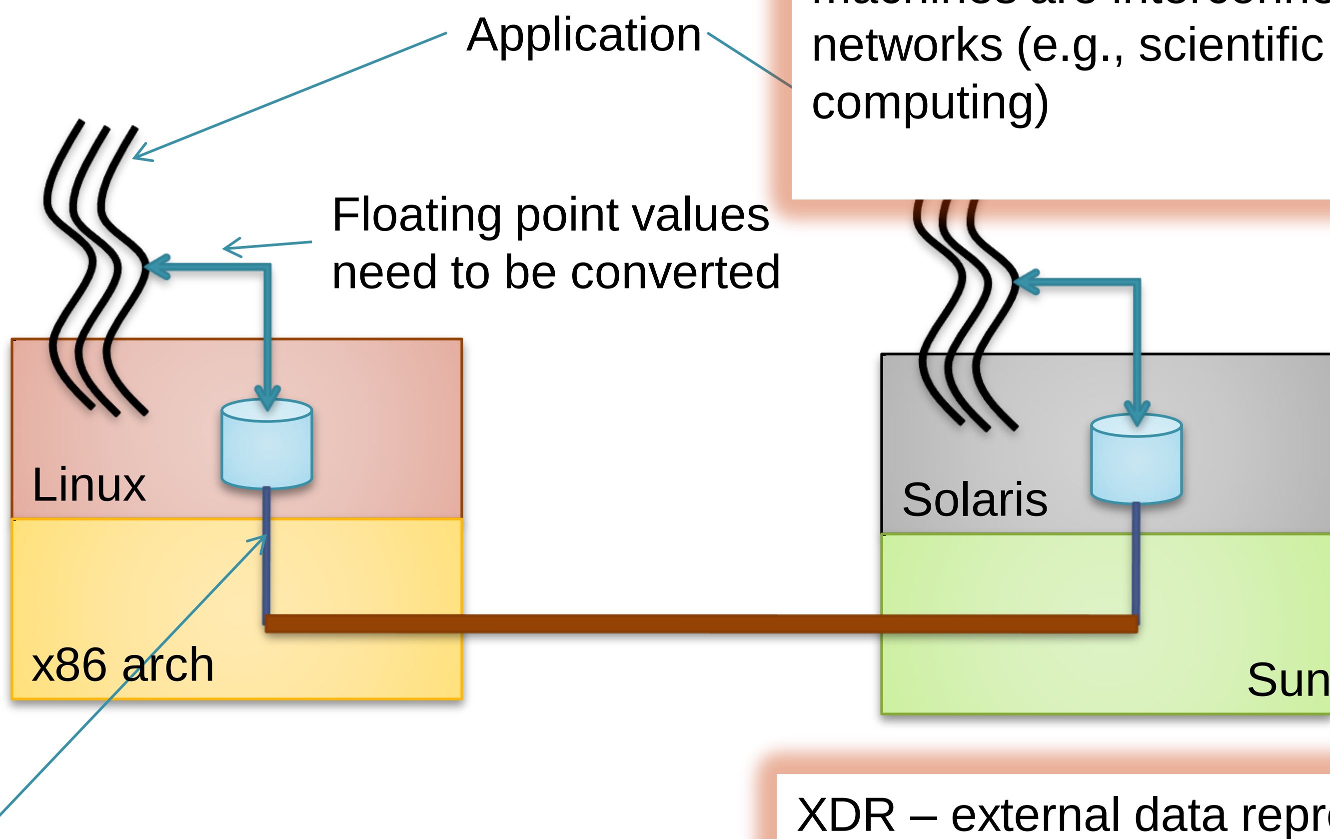
- “Transporting” protocols are responsible for pushing data from *A* to *B*
- Do ***two*** major functions:
  - ***Manipulate data*** – moving data to/from net, error detection, buffering, encryption, moving data to/from apps, presentation formatting, framing
  - ***Control transfer process*** – flow control, congestion control, multiplexing, time-stamping, and detect network transmission problems

Manipulating Data in Protocols



Presentation Formatting

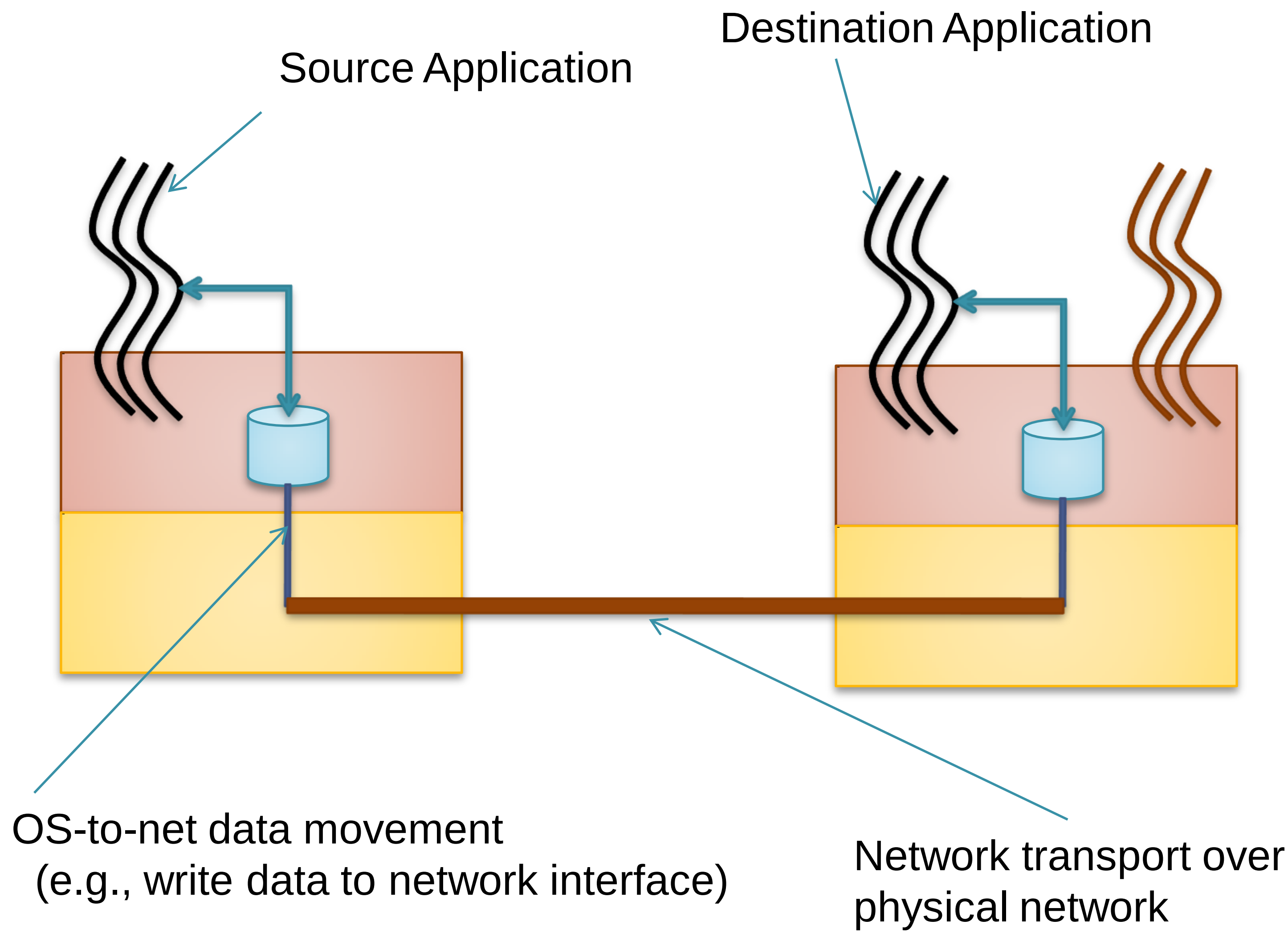
Needed when heterogeneous machines are interconnected by networks (e.g., scientific computing)



Simplest formatting – host-to-network byte ordering; happens on all network programs

XDR – external data representation
NetCDF – network common data format

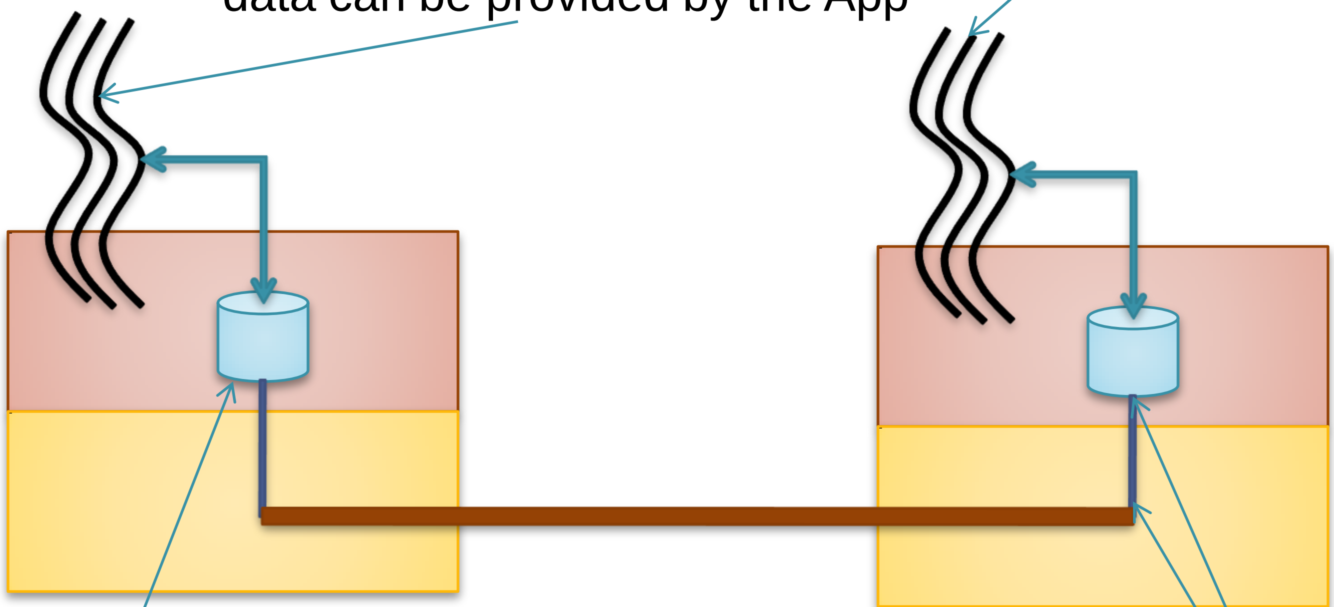
Multiplexing among Apps



Error Detection

Source application
can be involved in "fixing"
data loss as well. Newer
data can be provided by the App

Destination Application



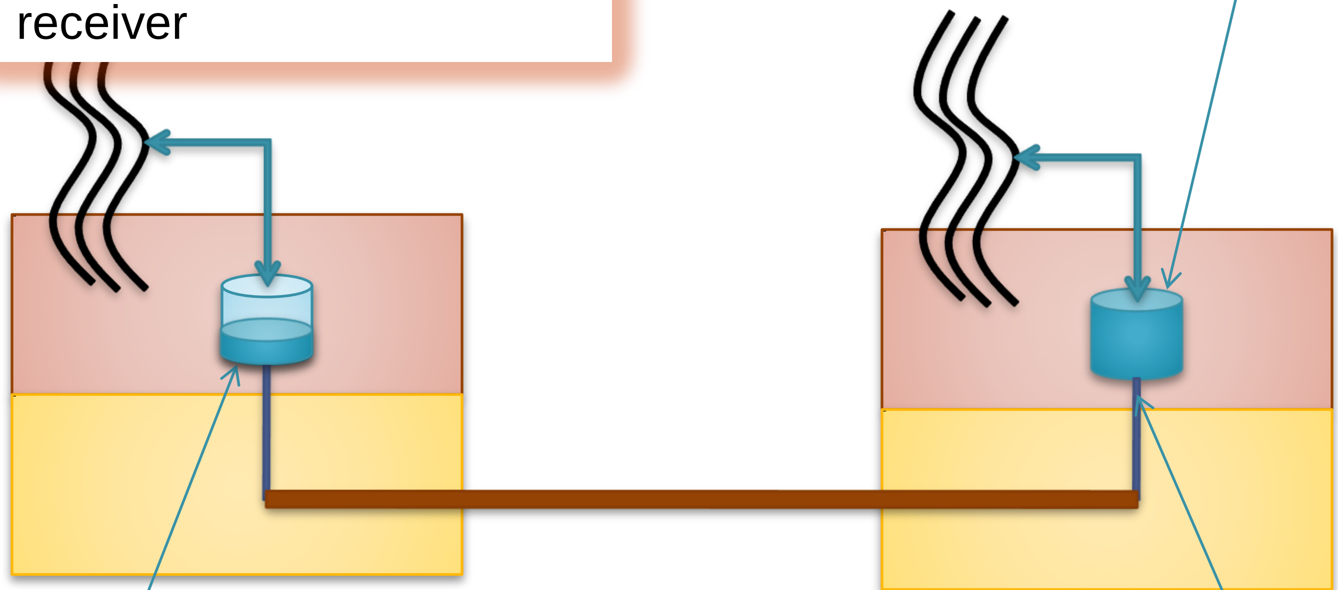
Data held by OS could be re-transmitted
to "fix" network errors (without the
App knowing about the network data loss)

Error detection can be
carried out by network hardware
and/or OS

Flow Control

Prevent fast transmitter from
overrunning a slow receiver

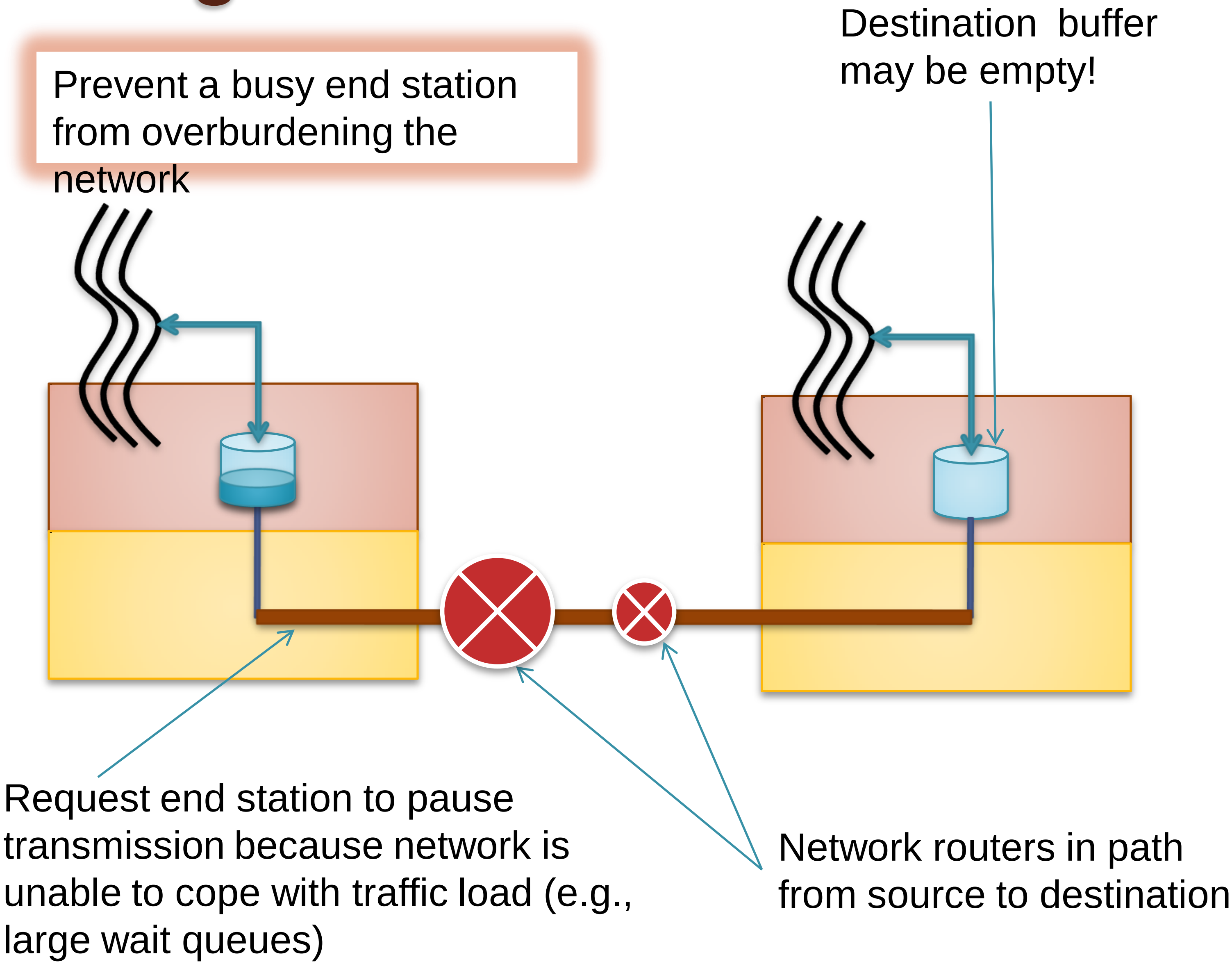
Destination buffer full – may be destination application is not actively empting the buffer or slow



Transmission pauses waiting for space at the receiver

Send flow control message to the source to stop transmit

Congestion Control



Other “Transporting” Protocol Issues

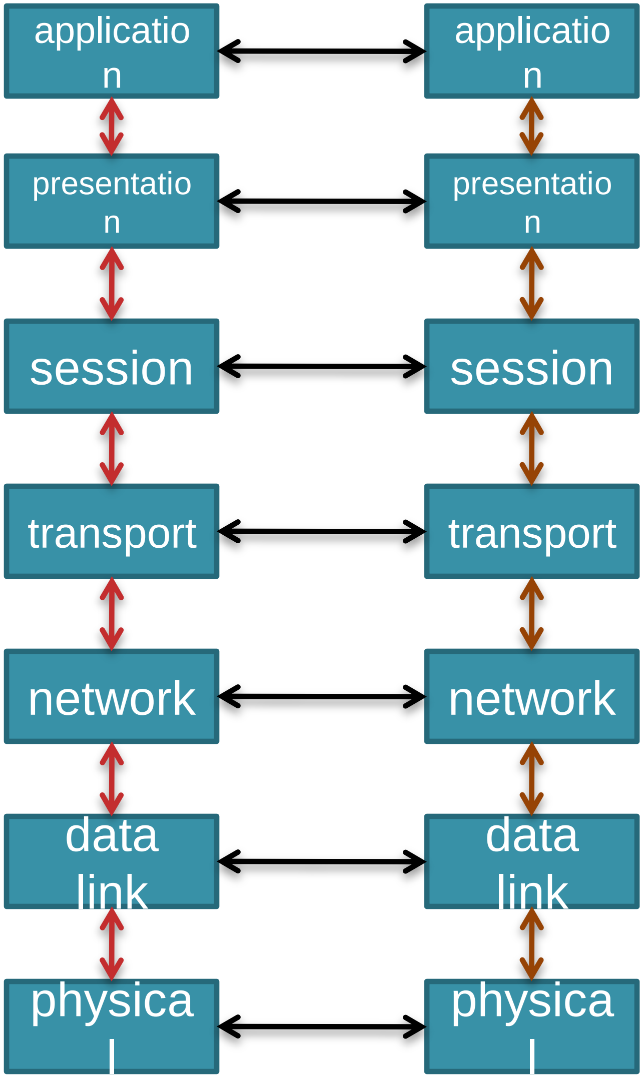
- Addressing
  - Mostly done by Networking protocols
  - Multiplexing part is done by transporting protocols
- Connection establishment
  - Sender/receiver synchronization
  - State establishment at both ends
- Connection release
  - Gracefully tearing down the state

Implementation of “Transporting” protocols

- “Transporting” protocols are implemented in a ***layered*** manner
- Primary motivation – complexity management

The concept of Layering

- **Physical** – transmits bits across a link
- **Data link** – deals with checksum; errors; access control
- **Network** – route computation; packet fragmentation; network interconnection
- **Transport** – lost packets, packet reordering, congestion
- **Session** – not much use; multimedia session control?
- **Presentation** – data representation for network

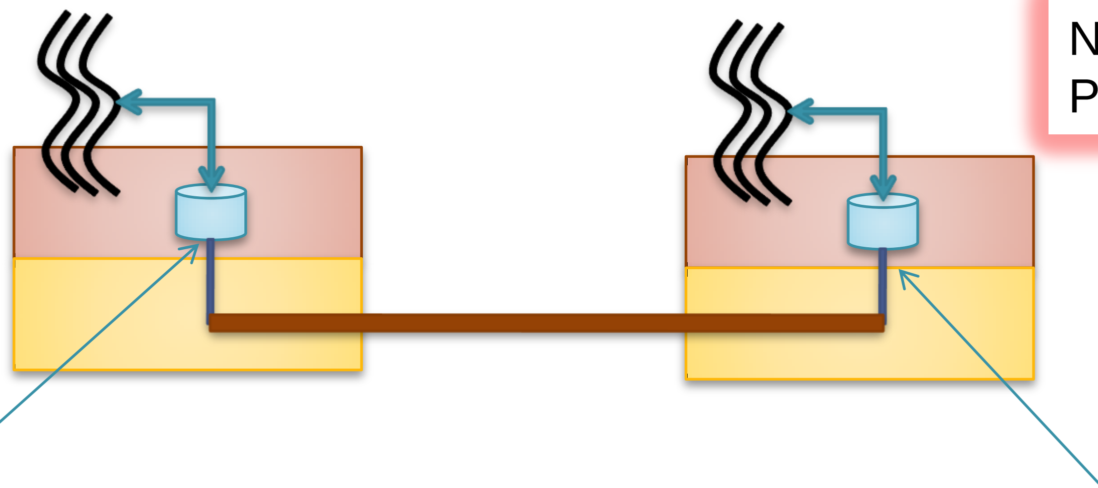


Drawbacks of Layered Implementation

- Efficiency – could lead to multiple copying
- Framing – data can be divided/joined as it goes through a layered stack

Simple Data Transfer Protocol

No reliability in Protocol



```
while (1) {
   get_net_layer(&buf);
   pkt.payload = buf;
   put_phy_layer(&pkt);
}
```

```
while (1) {
   get_phy_layer(&pkt);
   buf = pkt.payload;
   put_net_layer(&buf);
}
```

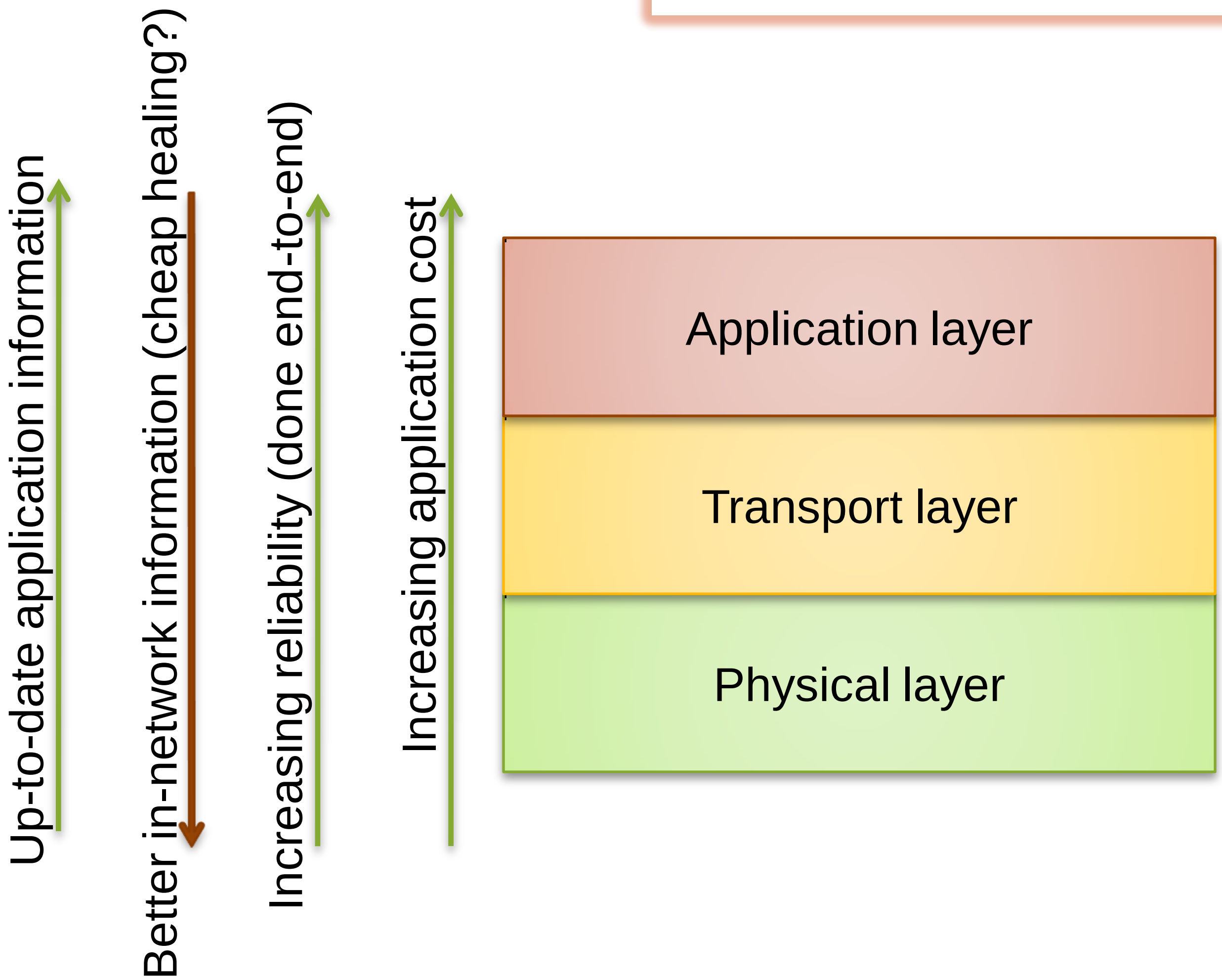
Buffer might be necessary even when retransmission is not done. For example, buffering can help in framing the data. That is, data injected by the application is usually buffered and “packetized” based on some condition.

Reliable Data Transfer Protocols

- Objective:
  - Reliably transmit data between two nodes
- Reliable data transfer dealt with in different layers of the protocol stack
  - physical layer – could be doing reliable transmit
  - transport layer – could be redoing reliable data transmit
  - application layer – could be doing it instead of the transport layer

Reliable Data Transfer...

Layer not relevant to the discussion are not shown here!



Implementing Reliable Data Transfer

- OS protocols provide reliability "buffer" to "buffer"
- ***Concern 1***: prevent loss at buffer (buffer overruns)
- ***Concern II***: recover from data loss in networks
- ***Concern III***: detect and correct corrupted data
- ***Concern IV***: deal with out-of-order data reception

Simple Data Transfer Protocol

```
// sender

while (1) {
    get_net_layer(&buf);
    pkt.payload = buf;
    put_phy_layer(&pkt);
}
```

```
// receiver

while (1) {
    get_phy_layer(&pkt);
    buf = pkt.payload;
    put_net_layer(&buf);
}
```

Unrestricted Simplex Protocol

- SDTP shown above assumes:
    - infinite buffers (no buffer overruns)
    - loss-less channel (no packet loss)
    - error-free channel (no packet corruption)
    - in-order data transfer (no packet reordering)
- SDTP is very idealistic – may work in local area environments!

Simple Data Transfer Version 2

- Adds flow control:
  - Receiver has finite buffer space and may not keep up with the sender

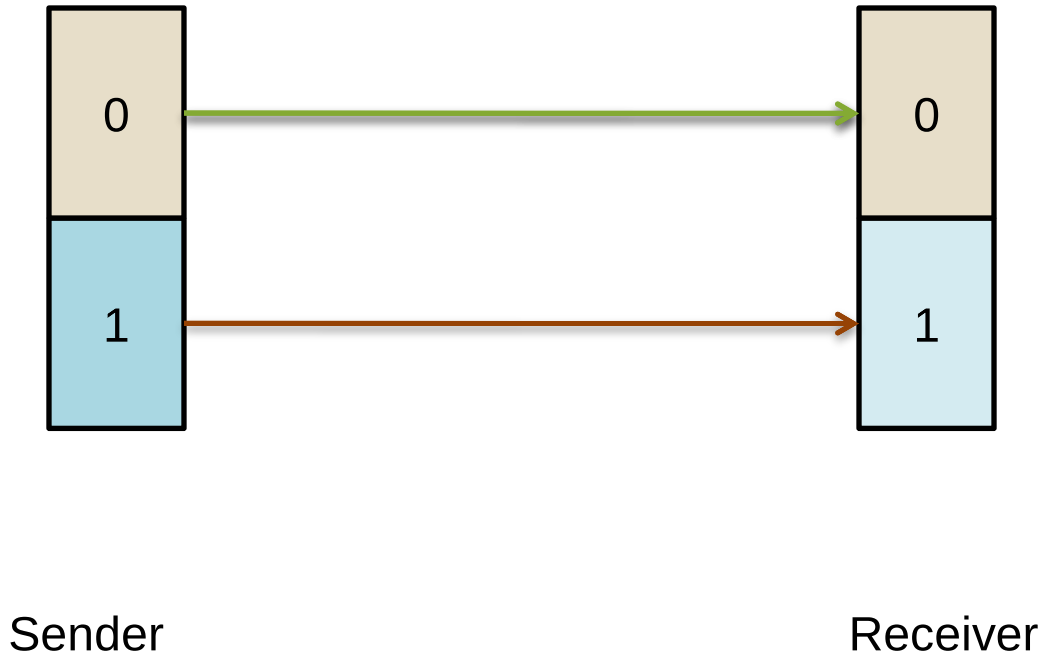
```
// sender loop

bool dst_buf_full = FALSE
while (1) {
   if (dst_buf_full == FALSE)
       get_net_layer(&buf);
       pkt.payload = buf;
       put_phy_layer(&pkt);
   get_phy_layer(&ack);
   dst_buf_full = ack.buf_state;
   // some timed trigger to
   // iterate
}

send_data() {
   // write data to network buffer
}
```

```
// receiver
int bspace = BUF_SIZE;

while (1) {
   get_phy_layer(&pkt);
   buf = pkt.payload;
   bspace--;
   put_net_layer(&buf);
   if (bspace <= 0)
       ack.buf_state = FALSE;
   else
       ack.buf_state = TRUE;
   put_phy_layer(&ack);
}

receive_data() {
    // read data from networ
    // buffer
    // increment bspace counter
}
```

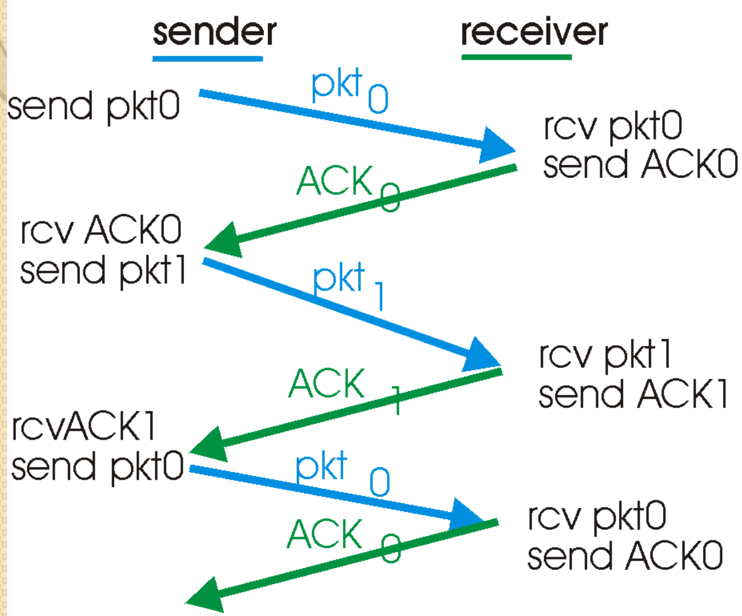
Sliding Window Protocols

- Generalize the protocols to:
  - Duplex
  - Piggyback ACK
  - Set windows at sender and receiver to denote valid frames
- Three variants:
  - A One-Bit Sliding Window Protocol
  - A Protocol Using Go Back N
  - A Protocol Using Selective Repeat

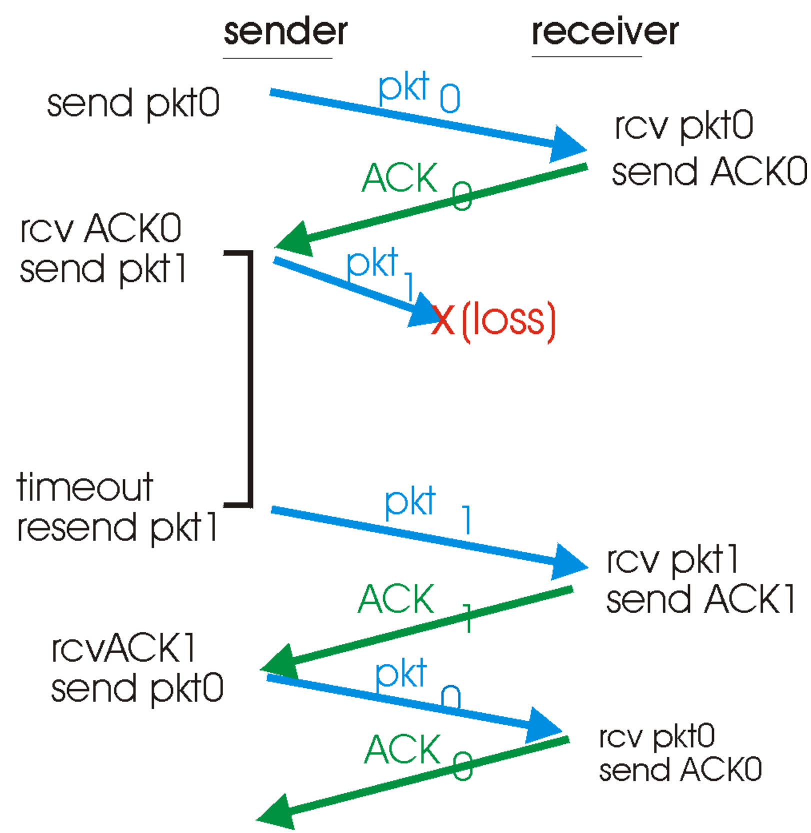
One-bit Sliding Window



Protocol in action

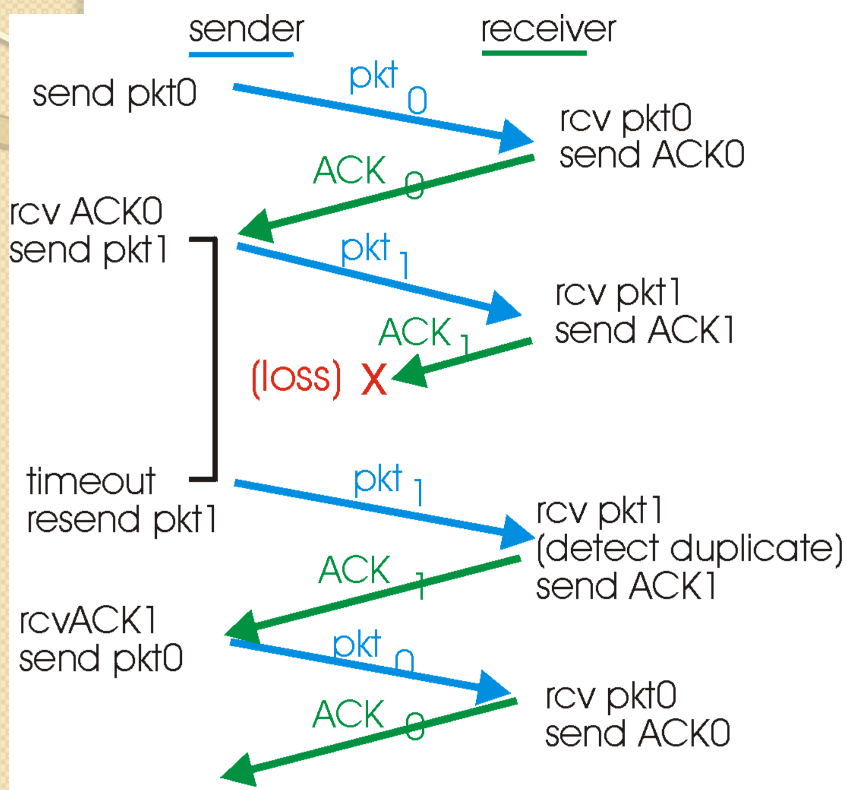


(a) operation with no loss

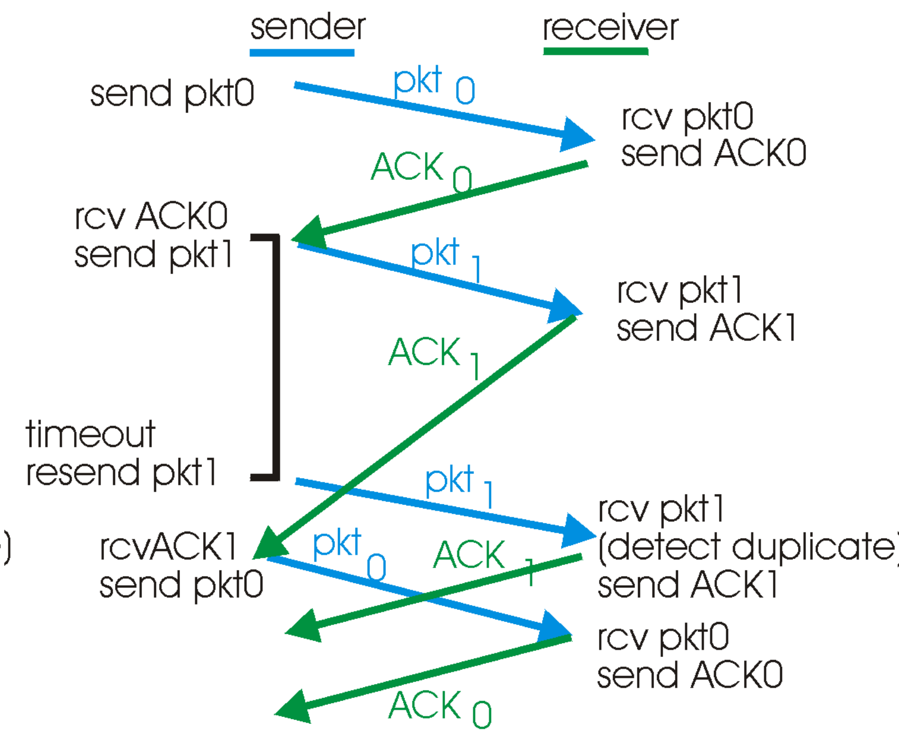


(b) lost packet

Protocol in action



(c) lost ACK



(d) premature timeout

Performance

Example: 1.0 Gbps link, 15 ms e-e prop. delay, 1KB packet:

$$T_{transmit} = \frac{L \text{ (packet length in bits)}}{R \text{ (transmission rate, bps)}} = \frac{8\text{kb/pkt}}{10^{**}9 \text{ b/sec}} = 8 \text{ microsec}$$

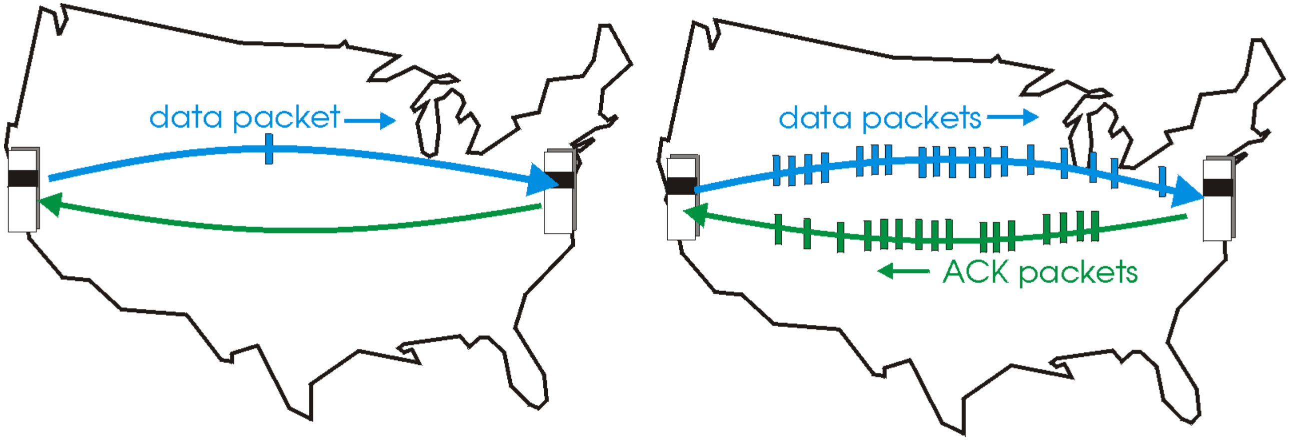
$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

- U_{sender}: utilization – fraction of time sender busy sending
- 1KB pkt every 30 msec -> 33kB/sec thruput over 1 Gbps link
- network protocol limits use of physical resources!

Pipelined protocols

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

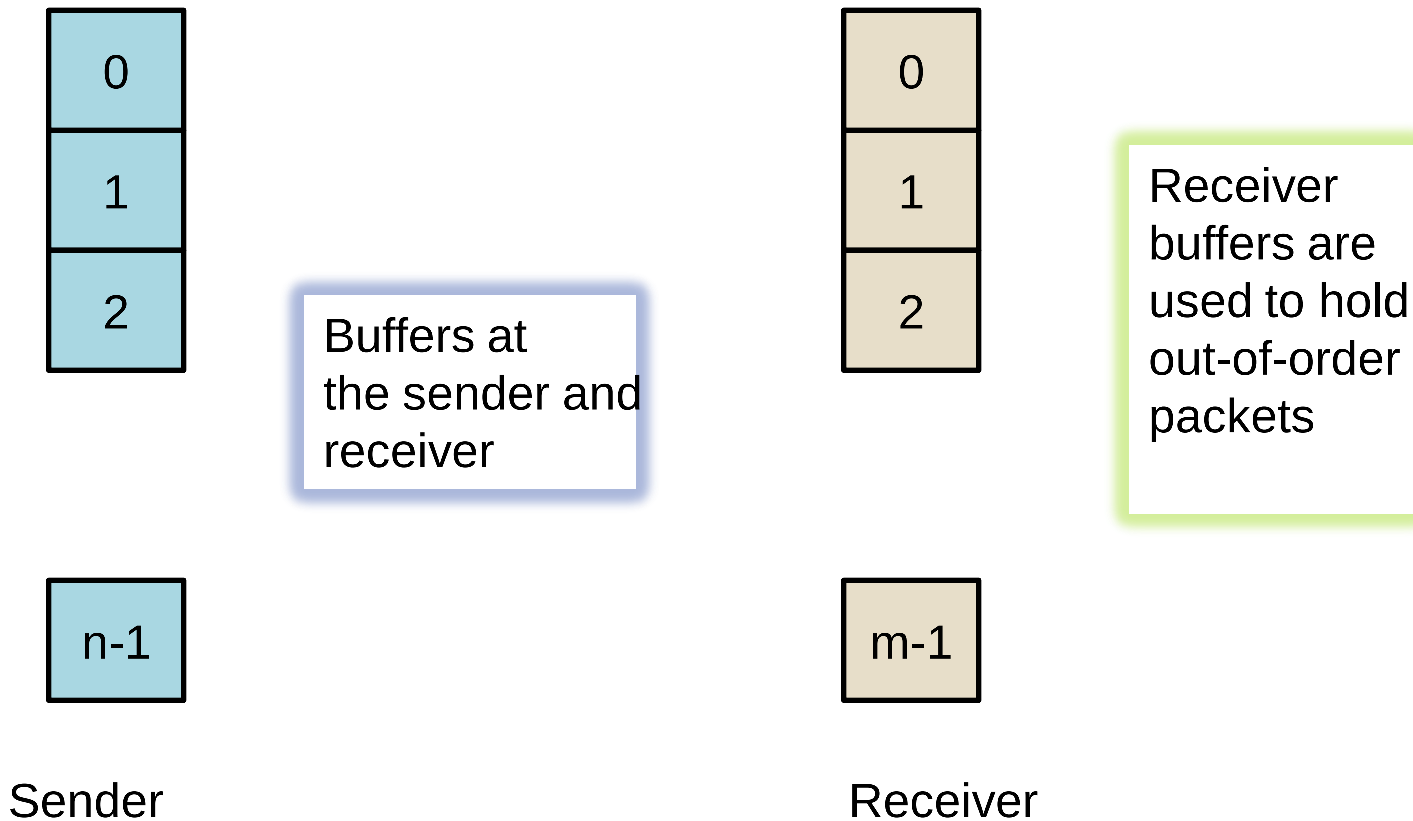
- range of sequence numbers must be increased
- buffering at sender and/or receiver



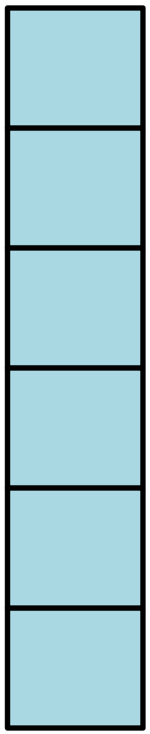
(a) a stop-and-wait protocol in operation (b) a pipelined protocol in operation

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

Pipelined Protocols

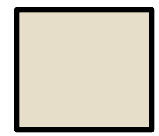


Go-Back-N



Sender

Sender buffers
Un ACKd packets



Receiver

Receiver
does not
buffer
out-of-order
packets

Sends “cumulative”
ACKs

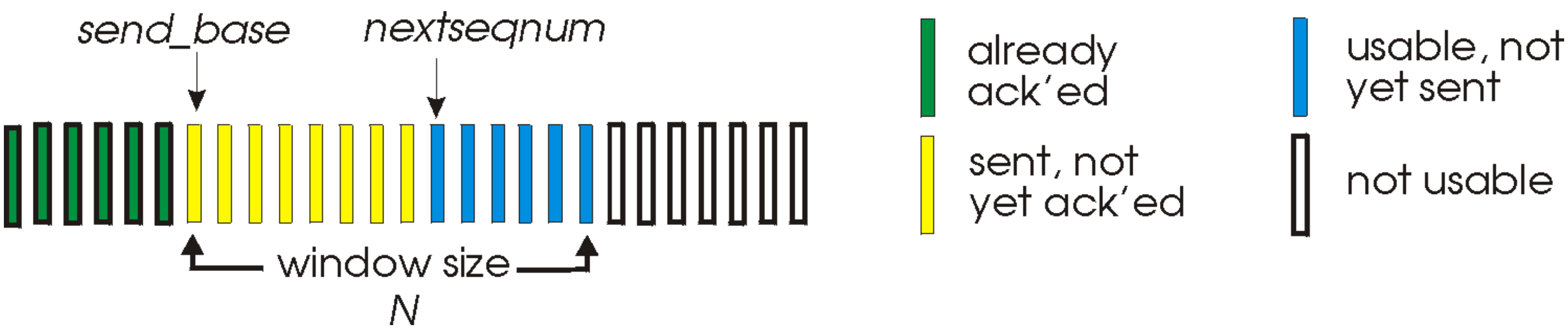
Packet loss causes retransmission
of all Un ACKd packets.

Can be costly with large window
sizes

Go-Back-N

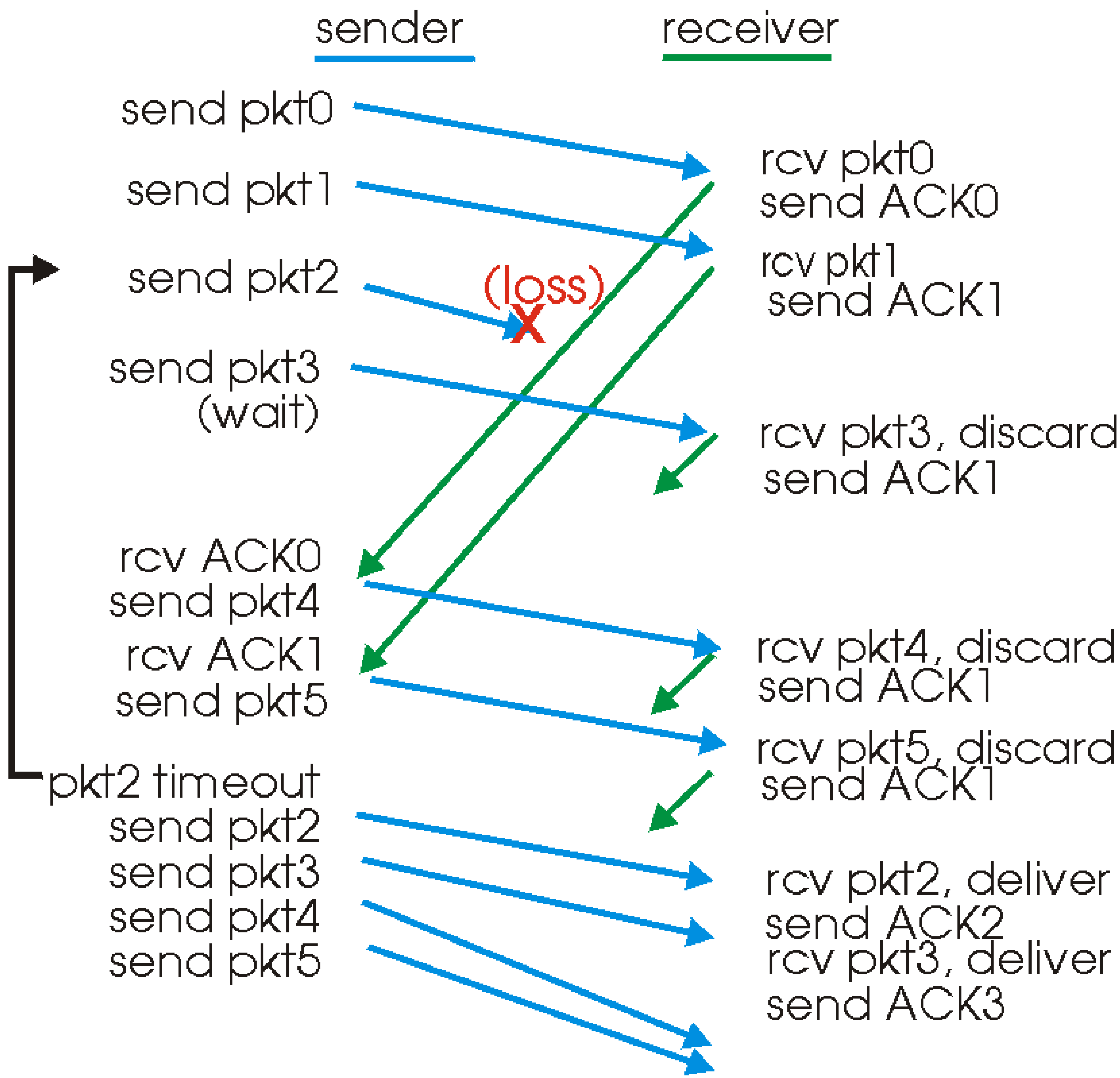
Sender:

- 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
  - timer for each in-flight pkt
- *timeout(n):* retransmit pkt n and all higher seq # pkts in window

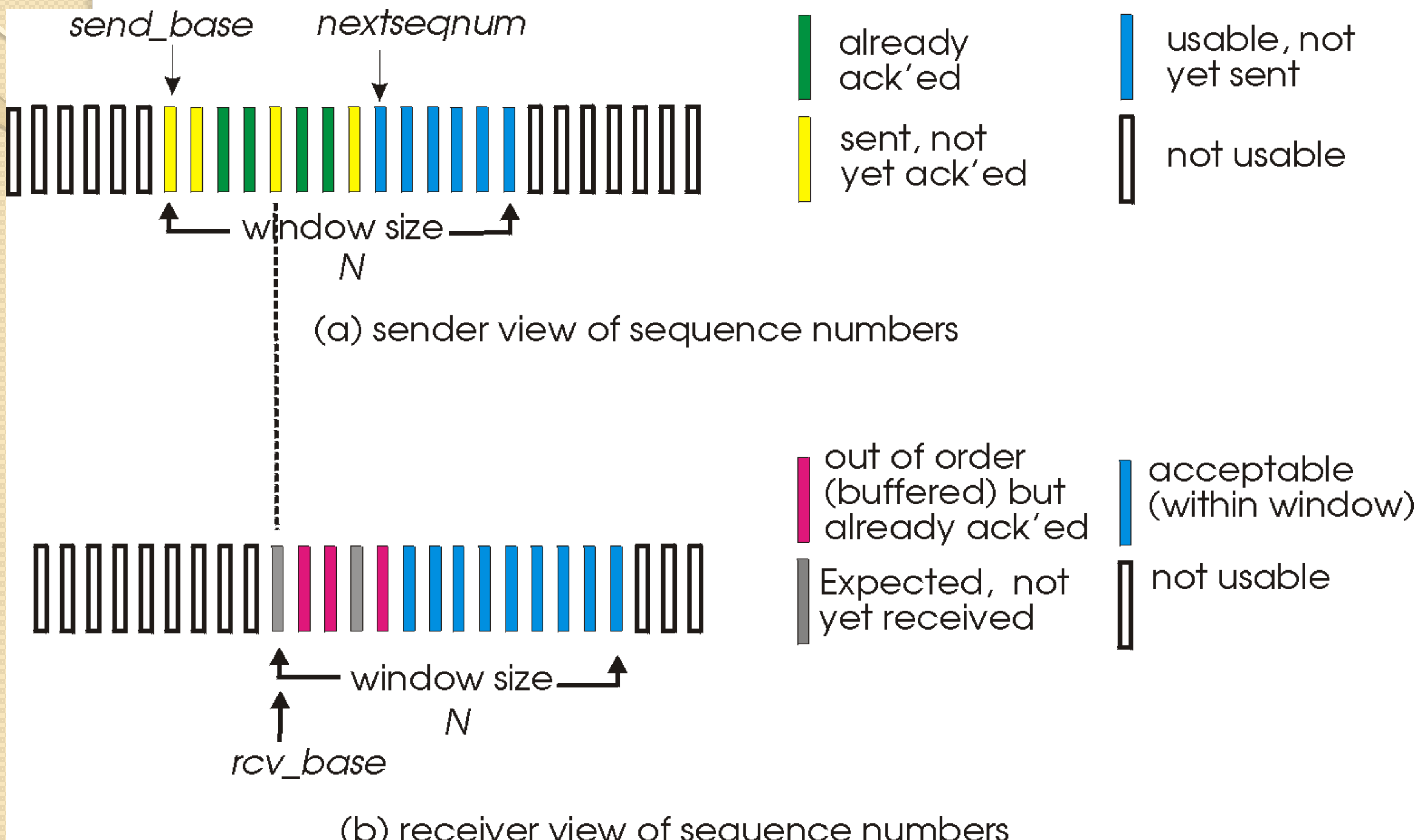
GBN in action



Selective Repeat

- receiver *individually* acknowledges all correctly received pkts
  - buffers pkts, as needed, for eventual in-order delivery to upper layer
- sender only resends pkts for which ACK not received
  - sender timer for each unACKed pkt
- sender window
  - N consecutive seq #'s
  - again 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

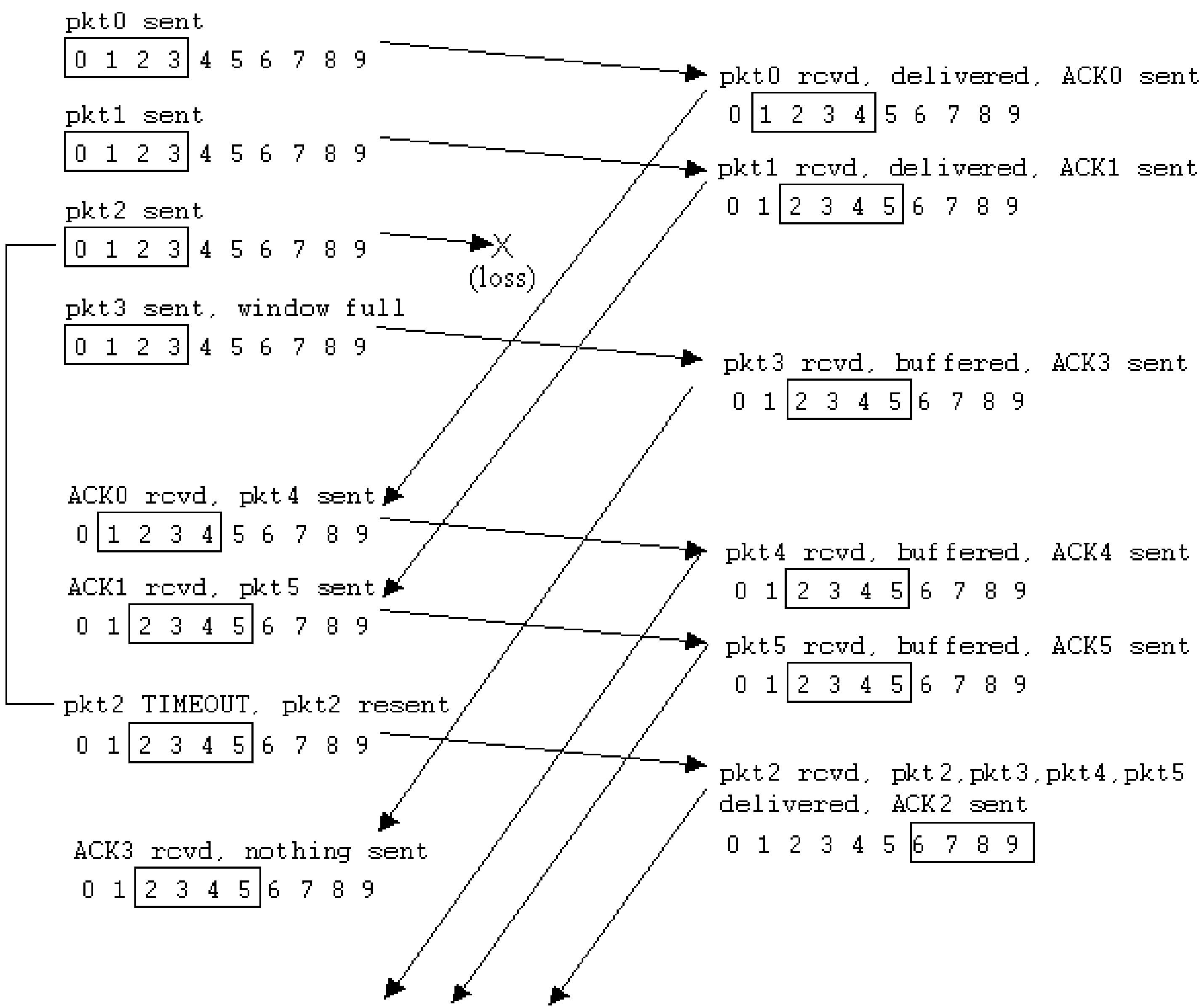
pkt n in [rcvbase-N,rcvbase-1]

- ACK(n)

otherwise:

- ignore

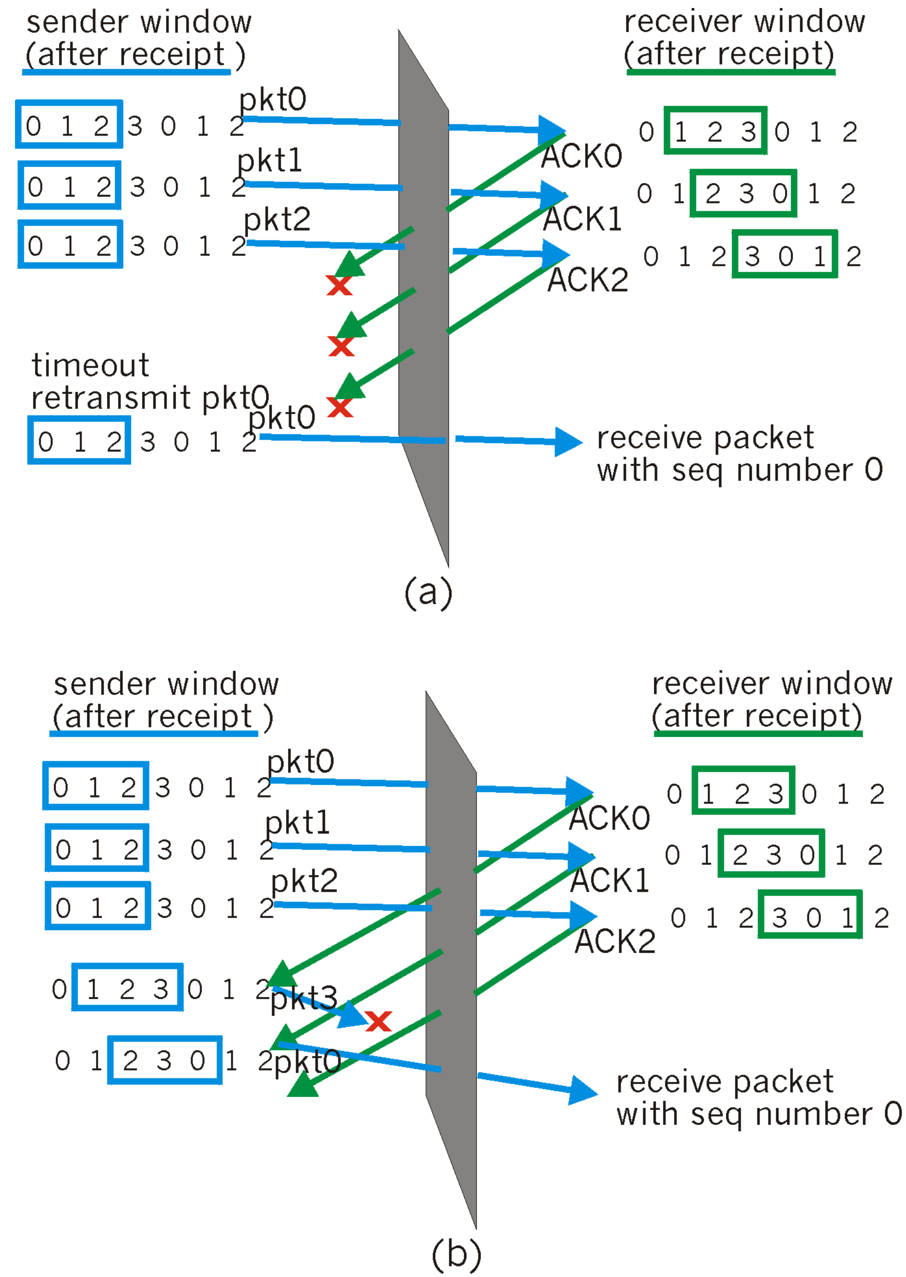
Selective repeat in action



Selective repeat: dilemma

Example:

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



- receiver sees no difference in two scenarios!
- incorrectly passes duplicate data as new in (a)

Q: what relationship between seq # size and window size?