# COMP 431
# Internet Services & Protocols

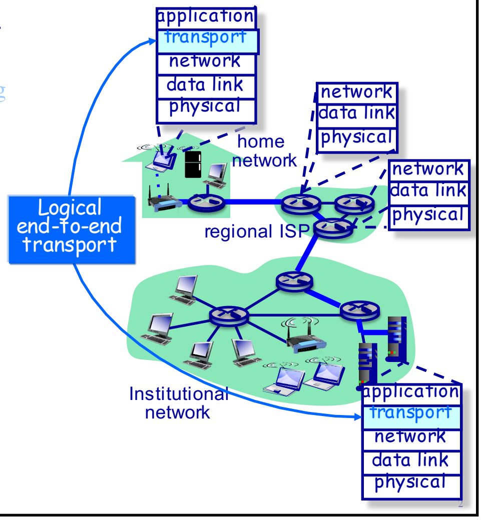
## The Transport Layer
Pipelined Transport Protocols

*Jasleen Kaur*

February 25, 2020

# Transport Layer Protocols & Services
## Outline

- Fundamental transport layer services
  - » Multiplexing/Demultiplexing
  - » Error detection
  - » Reliable data delivery
  - » Pipelining
  - » Flow control
  - » Congestion control
- Service implementation in Internet transport protocols
  - » UDP
  - » TCP



# Transport Protocol Performance

## Performance of RDT3.0

- Can an end-system make efficient use of a network under RDT 3.0?
- Consider a 1 Gbps link with 15 *ms* end-to-end propagation delay
- How busy is the network under RDT 3.0?

$$\textit{utilization} = \frac{\textit{time network busy}}{\textit{observation interval}} = \frac{\textit{time to transmit a packet}}{\textit{packet generation time}}$$

- How long does it take to transmit a 1,000 byte packet?

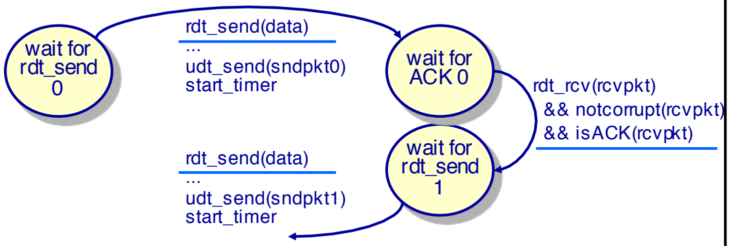
$$\begin{matrix}\textit{transmission}\\ \textit{time}\end{matrix} = \frac{1\ \textit{kB packet} \times 8\ \textit{b/byte}}{10^{9}\ \textit{bps}} = 8\ \mu s$$

- How fast can an end-system generate packets?

15miliseconds + 8microseconds for packet 0

# Transport Protocol Performance

## Performance of RDT3.0



- How fast can an end-system generate packets?
  - » Packet transmission time = 8 $\mu s$
  - » Propagation delay to receiver = 15 *ms*
  - » ACK generation/transmission time ≈ 8 $\mu s$
  - » Propagation time for ACK to return to sender = 15 *ms*
- Best case: 1 packet every 30.016 *ms*

# Transport Protocol Performance

## Performance of RDT3.0

- How busy is the network under RDT 3.0?

$$utilization = \frac{time\ network\ busy}{observation\ interval} = \frac{time\ to\ transmit\ a\ packet}{packet\ generation\ time}$$

$$= \frac{8\ \mu s}{30.016\ ms} = 0.027\%$$

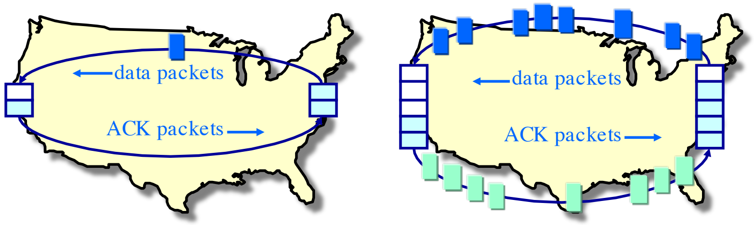
- Is this good?
  - » 1,000 byte packet every 30 *ms* results in (maximum) throughput of 266 *kbps* over a 1 Gbps link!
    (266,000 *bps* over a 1,000,000,000 *bps* link)

***Network protocols limit the use of physical resources!***

# Improving Transport Protocol Performance

## Pipelining data transmissions

- Performance can be improved by allowing the sender to have multiple unacknowledged packets "in flight"



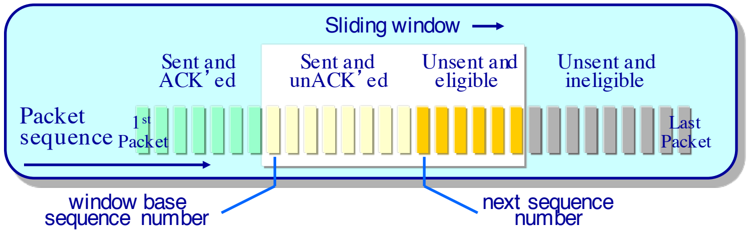
Stop-and-Wait protocol

Pipelined protocol

- Issues?
  - » The range of sequence numbers must be increased
  - » More packets must be buffered at sender and receiver

# Pipelined Protocols

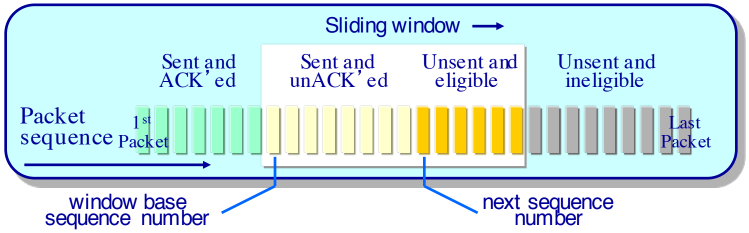
## "Go-Back-*n*" protocols



- Packet header contains a $k$-bit sequence number
- A "window" of up to $N \le 2^k$ consecutive, unacknowledged packets allowed to be in-flight
  - » Up to $N$ packets may be buffered at the sender
  - » Window advances as ACKs are received
- Receiver generates "cumulative ACKs"
  - » ACKs contain the sequence number of the last in-order packet received

# Pipelined Protocols

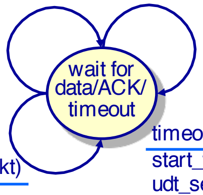
## "Go-Back-*n*" protocols



- Receiver protocol
  - » Use cumulative ACKs — ACK packet $n$ only if all packets numbered less than $n$ have been received
  - » If losses occur, sender may receive duplicate ACKs
- Sender protocol
  - » A timer is set for each (or just the oldest) in-flight packet
  - » On timeout for packet $n$, retransmit packet $n$ and **all** higher number packets in the current window

# Go-Back-*n* Protocol

## Sender extended FSM



```
rdt_send(data)
```

```
if (nextseqnum < base+N) {
   compute chksum
   make_pkt(sndpkt[nextseqnum],nextseqnum,data,chksum)
   udt_send(sndpkt[nextseqnum])
   if (base == nextseqnum) start_timer
   nextseqnum  += 1
 }
else
   refuse_data(data)
```

```
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
```

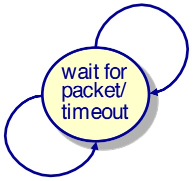
```
base := getacknum(rcvpkt) + 1
if (base == nextseqnum)
  stop_timer
else
  start_timer
```

```
timeout
```

```
start_timer
udt_send(sndpkt[base])
udt_send(sndpkt[base+1])
...
udt_send(sndpkt[nextseqnum–1])
```

# Go-Back-*n* Protocol

## Receiver extended FSM



```
default
```

```
udt_send(sndpkt)
```

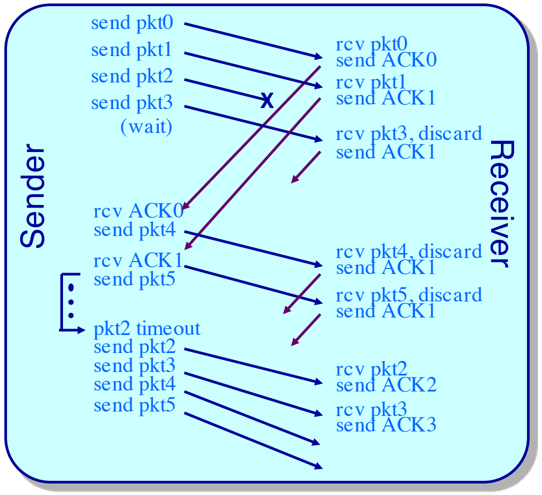
```
rdt_rcv(rcvpkt)
  && notcorrupt(rcvpkt)
  && has_seqnum(rcvpkt,expectedSeqNum)
```

```
extract(rcvpkt,data)
deliver_data(data)
make_pkt(sndpkt,ACK,expectedSeqNum)
expectedseqnum  += 1
udt_send(sndpkt)
```

- In-order packets processed, out-of-order packets discarded
  - » Sender will eventually timeout and retransmit out-of-order packets
  - » Thus the receiver need not buffer any packets
- Always send ACK for correctly-received packet with highest *in-order* sequence number
  - » May generate duplicate ACKs
  - » But minimal state — need only remember *expectedseqnum*

# Go-Back-*n* Protocol

## Execution example



Sender

send pkt0
send pkt1
send pkt2
send pkt3
(wait)

rcv ACK0
send pkt4
rcv ACK1
send pkt5

pkt2 timeout
send pkt2
send pkt3
send pkt4
send pkt5

Receiver

rcv pkt0
send ACK0
rcv pkt1
send ACK1
rcv pkt3, discard
send ACK1

rcv pkt4, discard
send ACK1
rcv pkt5, discard
send ACK1

rcv pkt2
send ACK2
rcv pkt3
send ACK3

- Assume a window size of 4 packets
- Receiver ignores out-of-order packets
- Sender retransmits only on timeout
  - » (Duplicate ACKs now have no effect)

# Transport Protocol Performance

## Performance of Go-Back-*n* protocols

- Can an end-system make more efficient use of a network under a Go-Back-*n* protocol?
- Consider again transmitting 1,000 byte packets on a 1 Gbps link with 15 *ms* end-to-end propagation delay

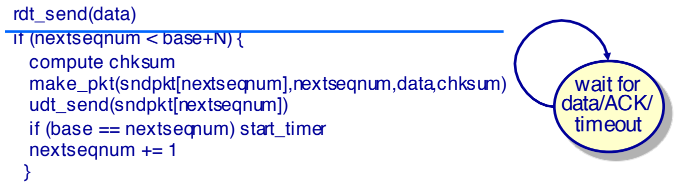
$$utilization = \frac{time\ to\ transmit\ a\ packet}{packet\ generation\ time}$$

$$transmission\ time = \frac{1\ kB\ packet \times 8\ b/byte}{10^9\ bps} = 8\ \mu s$$

- How fast can an end-system transmit packets?
  - » Depends on the window size!

# Transport Protocol Performance

## Performance of Go-Back-*n* protocols



```
rdt_send(data)
if (nextseqnum < base+N) {
   compute chksum
   make_pkt(sndpkt[nextseqnum],nextseqnum,data,chksum)
   udt_send(sndpkt[nextseqnum])
   if (base == nextseqnum) start_timer
   nextseqnum += 1
  }
```

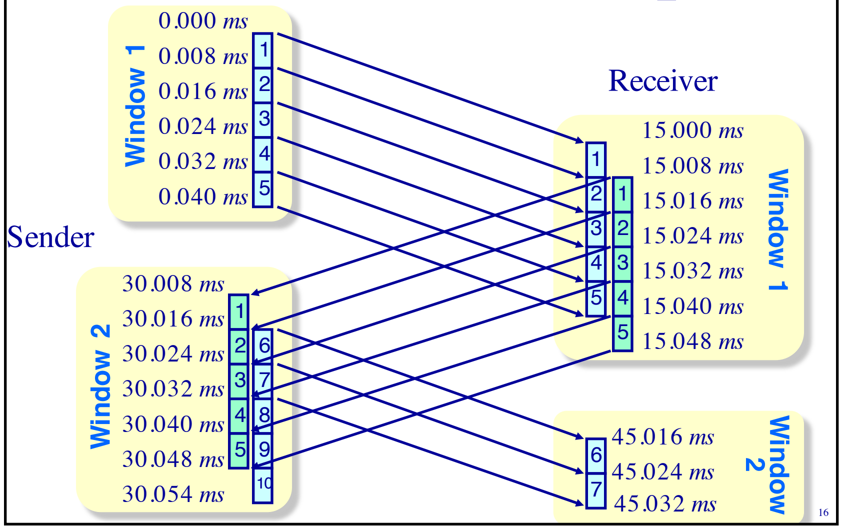
- How fast can an end-system transmit packets?
  - *N* packets can be sent before the sender must wait for an ACK
- *N* packets sent every 30.016 *ms*
  - Packet generation/transmission time = 8 *μs*
  - Round-trip-time to receiver = 30 *ms*
  - ACK generation/transmission time ≈ 8 *μs*

# Transport Protocol Performance

## Performance of Go-Back-*n* protocols

*x* Data packet with sequence number *x*

*x* ACK packet with sequence number *x*



# Transport Protocol Performance

## Performance of Go-Back-*n* protocols

- Performance with a window size of $N = 64$ packets:

$$utilization = \frac{\textit{time to transmit N packets}}{\textit{time to receipt of first ACK}}$$

$$= \frac{512\ \mu s}{30.016\ ms} = 1.7\%$$

A 64x improvement!

- Is this good?
  - » 64 1,000 byte packets every 30 *ms* results in (maximum) throughput of 17 *Mbps* over a 1 Gbps link!

# Pipelined Protocols

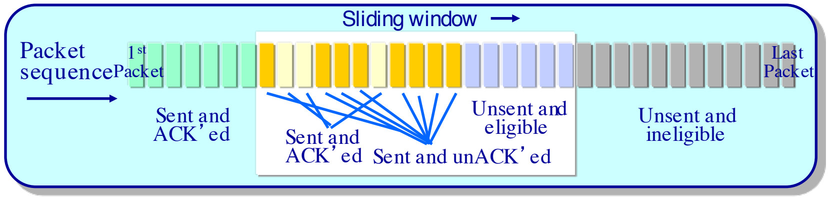
## “Selective Repeat” protocols

- Receiver *individually* acknowledges all correctly received packets
  - » Buffers packets as needed for eventual in-order delivery to upper layer
- Sender only resends packets for which an ACK has not been received
  - » Sender maintains a timer for each unACK’ed packet
- Sender window is the same as before
  - » *N* consecutive sequence numbers
    (Limits the sequence numbers of sent, unACK’ed packets)

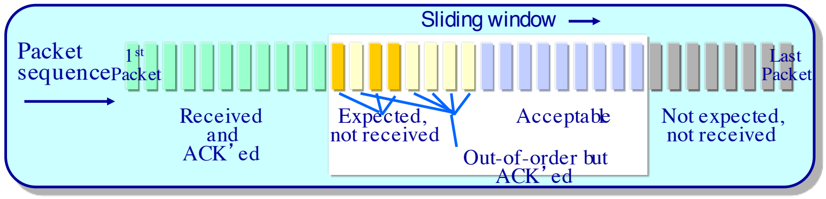
# Selective Repeat Protocols

## Sender and receiver windows

- Sender's view of sequence number space

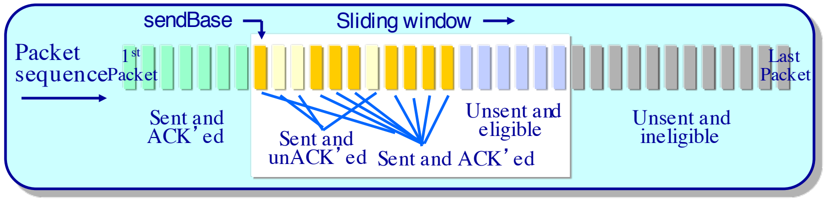


- Receiver's view of sequence number space



# Selective Repeat Protocols

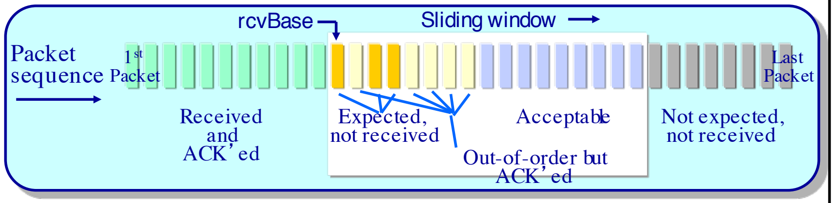
## Sender state machine



- Call from above:
  - » If next available sequence number is within window, send the packet and start a timer for it
- Timeout for packet *n*:
  - » Resend packet *n*, restart timer for packet *n*
- ACK received for packet with sequence number *n*:
  - » If *n* in [*sendBase*, *sendBase*+*N*–1] then mark packet *n* as received
  - » If *n* == *sendBase*, advance *sendBase* to next highest unACKed sequence number and move the window forward by that amount

# Selective Repeat Protocols

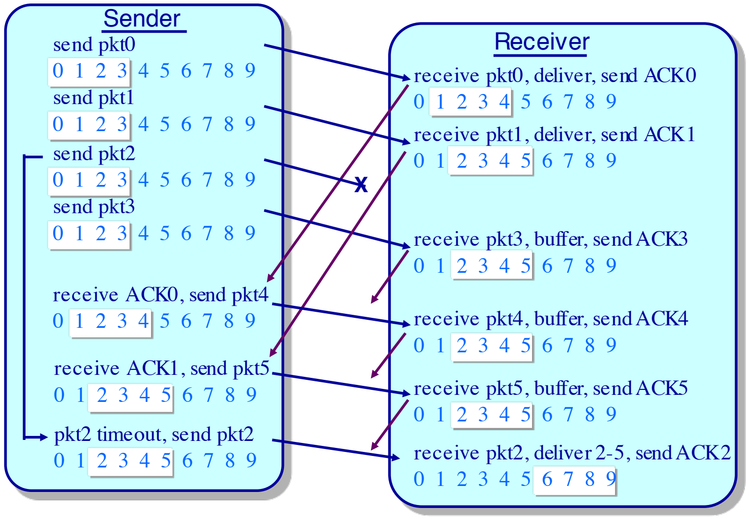
## Receiver state machine



- Packet $n$ in [$rcvbase$, $rcvbase+N-1$] correctly received:
  - » Send an ACK for packet $n$
  - » If packet $n$ is out-of-order then buffer
  - » If $n == rcvBase$, deliver packet $n$, and all other buffered consecutive in-order packets, to application, and advance the window by the number of delivered packets
- Packet $n$ in [$rcvbase-N$, $rcvbase-1$] received:
  - » Send an ACK for packet $n$
- Otherwise discard packet (without ACK'ing)

# Selective Repeat Protocols

## Execution example



Sender

send pkt0
0 1 2 3 4 5 6 7 8 9

send pkt1
0 1 2 3 4 5 6 7 8 9

send pkt2
0 1 2 3 4 5 6 7 8 9

send pkt3
0 1 2 3 4 5 6 7 8 9

receive ACK0, send pkt4
0 1 2 3 4 5 6 7 8 9

receive ACK1, send pkt5
0 1 2 3 4 5 6 7 8 9

pkt2 timeout, send pkt2
0 1 2 3 4 5 6 7 8 9

Receiver

receive pkt0, deliver, send ACK0
0 1 2 3 4 5 6 7 8 9

receive pkt1, deliver, send ACK1
0 1 2 3 4 5 6 7 8 9

receive pkt3, buffer, send ACK3
0 1 2 3 4 5 6 7 8 9

receive pkt4, buffer, send ACK4
0 1 2 3 4 5 6 7 8 9

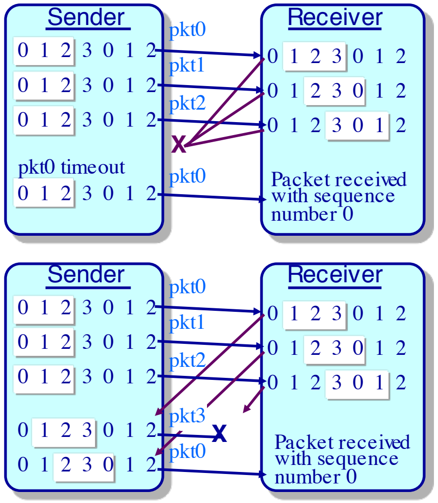
receive pkt5, buffer, send ACK5
0 1 2 3 4 5 6 7 8 9

receive pkt2, deliver 2-5, send ACK2
0 1 2 3 4 5 6 7 8 9

# Selective Repeat Protocols

## Window state ambiguity

- How many sequence numbers do we need?
  - » As many as the largest number of packets that can be in flight?
- If the sequence number space is close to the window size then the receiver can get confused



Sender
0 1 2 3 0 1 2
0 1 2 3 0 1 2
0 1 2 3 0 1 2
pkt0 timeout
0 1 2 3 0 1 2
pkt0
pkt1
pkt2
X
pkt0
Receiver
0 1 2 3 0 1 2
0 1 2 3 0 1 2
0 1 2 3 0 1 2
Packet received with sequence number 0
Sender
0 1 2 3 0 1 2
0 1 2 3 0 1 2
0 1 2 3 0 1 2
0 1 2 3 0 1 2
0 1 2 3 0 1 2
pkt0
pkt1
pkt2
pkt3
X
pkt0
Receiver
0 1 2 3 0 1 2
0 1 2 3 0 1 2
0 1 2 3 0 1 2
Packet received with sequence number 0