ECSE-4670: Computer Communication Networks (CCN)

Chapter 3a: Transport Layer

Shivkumar Kalyanaraman: shivkuma@ecse.rpi.edu
Biplab Sikdar: sikdab@rpi.edu

Adapted in part from J. Kurose (Umass)

Chapter Goals

- Understand principles behind transport layer services:
  - multiplexing/demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- Instantiation and implementation in the Internet

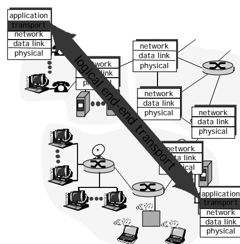
Chapter Overview

- Transport layer services
- Multiplexing/demultiplexing
- Connectionless transport: UDP
- Principles of reliable data transfer
- Connection-oriented transport: TCP
  - reliable transfer
  - flow control
  - connection management
- Principles of congestion control
- TCP congestion control

Transport services and protocols

- Provide *logical communication* between app' processes running on different hosts
- Transport protocols run in end systems
- Transport vs. network layer services:
  - *network layer:* data transfer between end systems
  - *transport layer:* data transfer between processes
  - Relies on, enhances, network layer services

Transport Services and Protocols



Transport-layer protocols

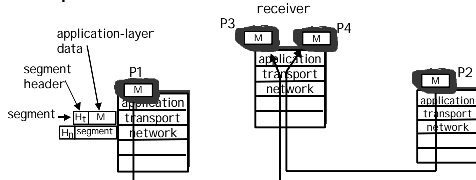
Internet transport services:

- Reliable, in-order unicast delivery (TCP)
  - congestion
  - flow control
  - connection setup
- Unreliable ("best-effort"), unordered unicast or multicast delivery: UDP
- Services not available:
  - real-time
  - bandwidth guarantees
  - reliable multicast

Multiplexing / demultiplexing

Recall: *segment* - unit of data exchanged between transport layer entities
- aka TPDU: transport protocol data unit

Demultiplexing: delivering received segments to correct app layer processes

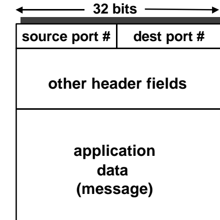


Multiplexing / demultiplexing

Multiplexing:

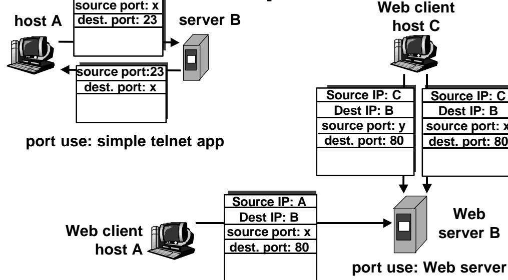
gathering data from multiple app processes, enveloping data with header (later used for demultiplexing)

multiplexing/demultiplexing:
- based on sender, receiver port numbers, IP addresses
  - source, dest port #s in each segment
  - recall: well-known port numbers for specific applications



TCP/UDP segment format

Multiplexing/demultiplexing: examples



UDP: User Datagram Protocol [RFC 768]

- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out of order to app
- *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

UDP: User Datagram Protocol [RFC 768]

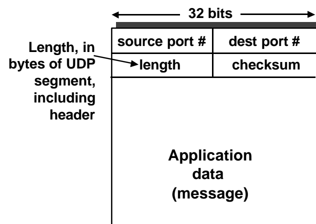
Why is there a UDP?

- no connection establishment (which can add delay)
- simple: no connection state at sender, receiver
- small segment header
- no congestion control: UDP can blast away as fast as desired.
  - May not be a good idea, though!

UDP: more

- Often used for streaming multimedia apps
  - loss tolerant
  - rate sensitive
- Other UDP uses (why?):
  - DNS
  - SNMP
- Reliable transfer over UDP: add reliability at application layer
  - application-specific error recovery!

UDP: more

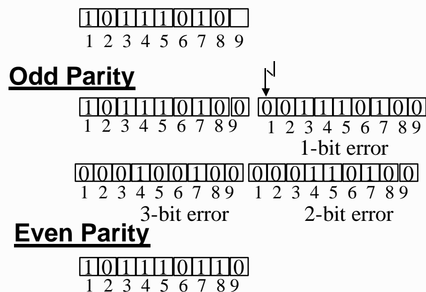


UDP segment format

Error Detection and Correction

- Single bit-errors vs Burst Errors
  110101 → 100101 vs 100001
- n-bit codeword = m message bits + r check bits
- Hamming Distance = # of different bits
  1010101
  1001010
  0011111 ⇒ Hamming distance = 5
- Distance d code = minimum Hamming distance *between any two code words* written in the code
- To detect d-bit errors, distance d+1 code required
- To correct d-bit errors, distance 2d+1 code required

Parity Checks



Parity is a distance 2 code => can detect 1-bit errors

UDP checksum

Goal: detect "errors" (e.g., flipped bits) in transmitted segment

Sender:

- treat segment contents as sequence of 16-bit integers
- checksum: addition (1's complement sum) of segment contents
- sender puts checksum value into UDP checksum field
- In reality some IP header fields are included w/ the UDP segment for checksumming.

UDP checksum

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?*
    - More in chap 5 on stronger error detection methods

UDP Checksum Example

- Consider three 16-bit words:
  0110011001100110
  0101010101010101
  0000111100001111
- (1's complement) sum of first two 16-bit words is:
  1011101110111011
- Adding the third word to the above sum gives:
  1100101011001010
- 1's complement of this sum => invert 0's and 1's

0011010100110101 (this is the *checksum field*)

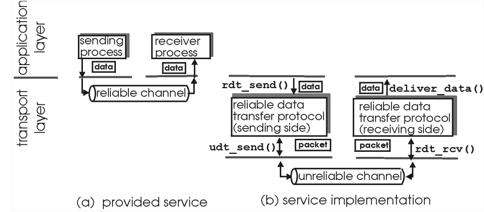
- If no errors, sum of all four 16-bit words (incl. Checksum) will be all 1s, I.e., 1111111111111111

UDP Servers

- Most UDP servers are ***"iterative"*** => a single server process receives and handles incoming requests on a ***"well-known"*** port.
- Can filter client requests based on incoming IP/port addresses or wild card filters
- Port numbers may be reused, but packet is delivered to at most one end-point.
- Queues to hold requests if server busy

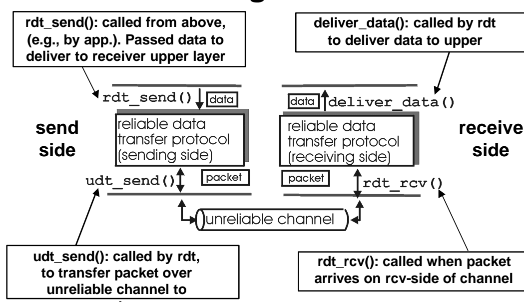
Principles of Reliable Data Transfer

- 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



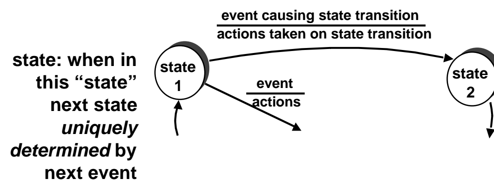
Reliable Data Transfer: Getting Started

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

Reliable Data Transfer: Getting Started

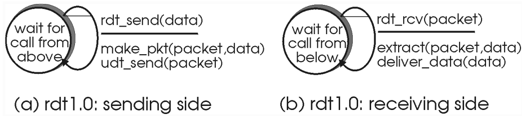
state: when in this "state" next state ***uniquely determined*** by next event



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

Rdt1.0: Reliable Transfer over a Reliable Channel (cont.)



(a) rdt1.0: sending side (b) rdt1.0: receiving side

Rdt2.0: Channel with Bit Errors

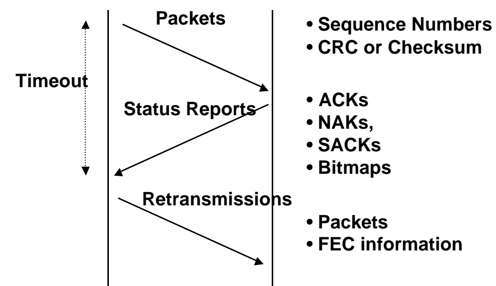
- Underlying channel may flip bits in packet
  - recall: UDP checksum to detect bit errors
- *The* question: how to recover from errors:
  - *acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK

Rdt2.0: Channel with Bit Errors (cont.)

- *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
  - receiver feedback: control msgs (ACK,NAK) rcvr->sender

Temporal Redundancy Model

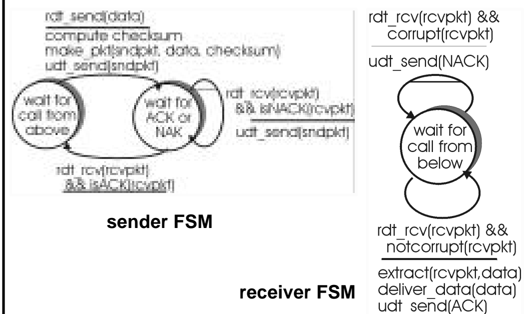


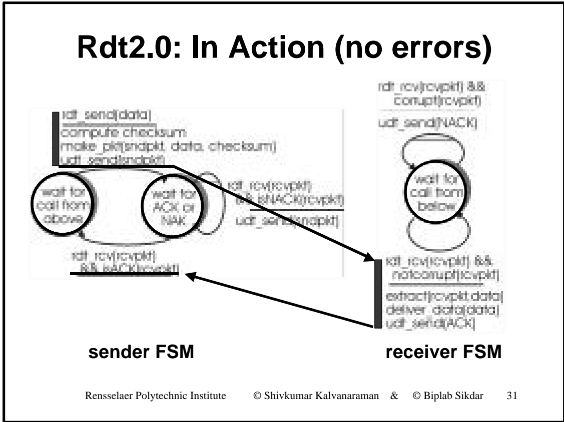
Reliability Models

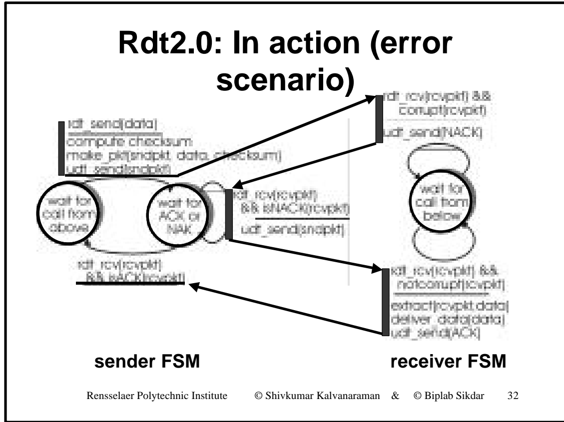
- Reliability => requires ***redundancy*** to recover from uncertain loss or other failure modes.

- Two types of redundancy:
  - Spatial redundancy: independent backup copies
    - Forward error correction (FEC) codes
    - Problem: requires huge *overhead*, since the FEC is also part of the packet(s) it cannot recover from erasure of all packets
  - Temporal redundancy: retransmit if packets lost/error
    - Lazy: trades off *response time* for reliability
    - Design of status reports and retransmission optimization important

Rdt2.0: FSM Specification







Reliability mechanisms learnt so far…

- **Mechanisms:**
  - Checksum in pkts: detects pkt corruption
  - ACK: "packet correctly received"
  - NAK: "packet *in*correctly received"
  - [aka: *stop-and-wait Automatic Repeat reQuest (ARQ)* protocols]
- **Reliability capabilities achieved:**
  - An *error-free* channel
  - A *forward* channel which has *bit*-errors

Rdt2.0 has a Fatal Flaw!

What happens if ACK/NAK corrupted

- Reverse channel bit-errors
- Sender doesn't know what happened at receiver!

What to do?

- Sender ACKs/NAKs receiver's ACK/NAK? *Problem: What if sender ACK/NAK lost?*
- Retransmit packet.

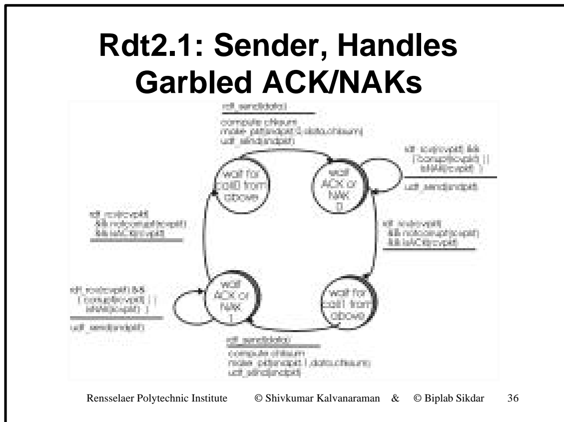
Problem: if original pkt correctly received, this is a duplicate.

Rdt2.0 has a Fatal Flaw! (cont.)

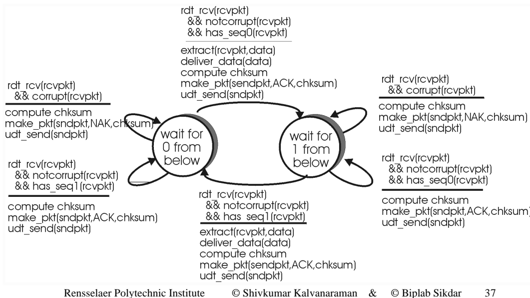
Handling duplicates, garbled ACK/NAKs:

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

stop and wait
Sender sends one packet, then waits for receiver response



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 0 or 1 seq. #**

Rdt2.1: Discussion

Receiver:

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

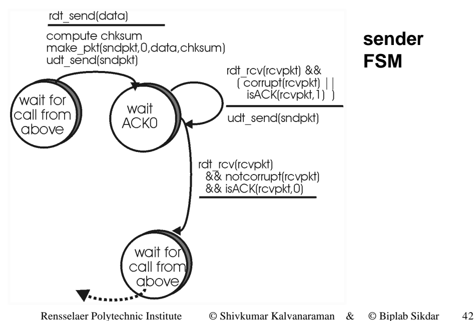
Reliability mechanisms learnt so far…

- **Mechanisms:**
  - **Checksum: detects corruption *in pkts & acks***
  - **ACK: "packet correctly received"**
  - **NAK: "packet *in*correctly received"**
  - **Sequence number: identifies packet or ack**
  - **1-bit sequence number used *only* in forward channel [aka: *alternating-bit* protocols]**
- **Reliability capabilities achieved:**
  - **An *error-free* channel**
  - **A *forward & reverse* channel with *bit*-errors**
  - **Detects *duplicates* of packets/acks/naks**

Rdt2.2: a NAK-free Protocol

- **Same functionality as rdt2.1, using ACKs only.**
  - ***Why bother ?***
- **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: a NAK-free Protocol



sender FSM

Mechanisms learnt so far…

- **Mechanisms:**
  - Checksum: detects corruption *in pkts & acks*
  - ACK: "packet correctly received"
  - Duplicate ACK: "packet *in*correctly received"
  - Sequence number: identifies packet or ack
    - 1-bit sequence number used *both in forward & reverse* channel
- **Reliability capabilities achieved:**
  - An *error-free* channel
  - A *forward & reverse* channel with *bit*-errors
  - Detects *duplicates* of packets/acks
  - *NAKs eliminated*

Rdt3.0: Channels with Errors *and* Loss

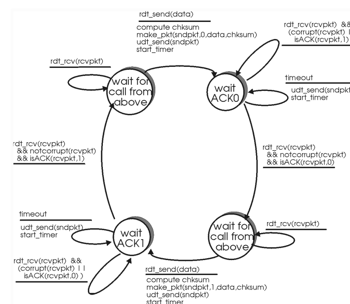
New assumption: underlying channel can also lose packets (data or ACKs)
- What's the difference, anyway?
- *Checksum, seq. #, ACKs, retransmissions* will help, but not enough

Q: how to deal with loss?
- Sender waits until certain data or ACK lost, then retransmits
- Yuck: drawbacks?

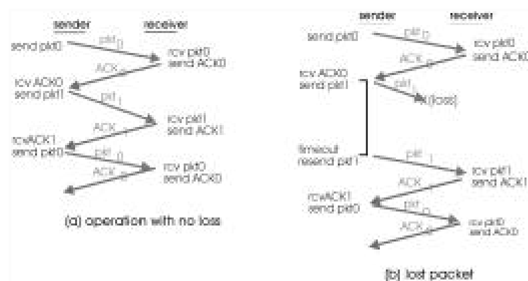
Rdt3.0: Channels with Errors *and* Loss

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

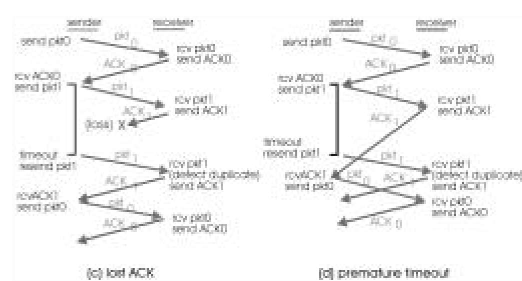
Rdt3.0 Sender



Rdt3.0 in Action



Rdt3.0 in Action



Mechanisms learnt so far…

- <u>Mechanisms:</u>
  - Checksum: detects corruption *in pkts & acks*
  - ACK: "packet correctly received"
  - <u>Duplicate</u> ACK: "packet *in*correctly received"
  - Sequence number: identifies packet or ack
    - 1-bit sequence number used *both in forward & reverse* channel
  - Timeout only at sender
- <u>Reliability capabilities achieved:</u>
  - An *error-free* channel
  - A *forward & reverse* channel with *bit*-errors
  - Detects *duplicates* of packets/acks
  - *NAKs eliminated*
  - A *forward & reverse* channel with *packet*-errors (loss)

Performance of Rdt3.0

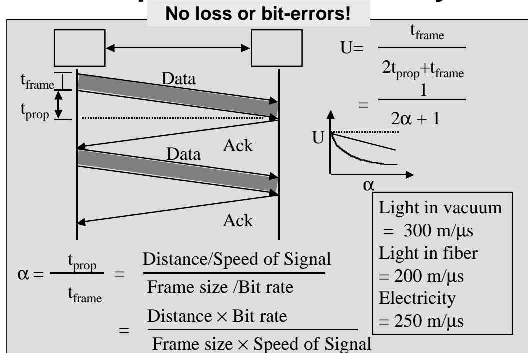
- rdt3.0 works, but performance stinks!
- example: 1 Gbps link, 15 ms e2e prop. delay, 1KB packet:

$$T_{transmit} = \frac{8kb/pkt}{10^{**}9\ b/sec} = 8 \text{ microsec}$$

$$\text{Utilization} = U = \frac{\text{fraction of time}}{\text{sender busy sending}} = \frac{8 \text{ microsec}}{30.016 \text{ msec}} = 0.00015$$

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

Stop and Wait Efficiency



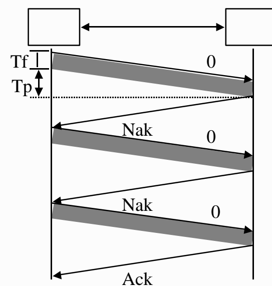
Utilization: More Examples

- Satellite Link:
  - Propagation Delay t_{prop} = 270 ms
    Frame Size = 4000 bits = 500 bytes
    Data rate = 56 kbps ⇒ t_{frame} = 4/56 = 71 ms
    $\alpha = t_{prop}/t_{frame}$ = 270/71 = 3.8
    U = 1/(2α+1) = <u>*0.12*</u> (too low !!)
- Short Link (eg: LAN):
  - 1 km = 5 μs,
    Rate=10 Mbps,
    Frame=500 bytes ⇒ t_{frame}= 4k/10M= 400 μs
    $\alpha=t_{prop}/t_{frame}$=5/400=0.012 ⇒ U=1/(2α+1)= <u>*0.98*</u> (great!)

Note: no loss or bit-errors!

Stop-and-Wait ARQ: w/ loss

- P=Probability of bit-error
- $\alpha = T_p/T_f$
- $U=T_f/[N_r(T_f+2T_p)] = 1/[N_r(1+2\alpha)]$
- $N_r=\Sigma\ i\ P^{i-1}(1-P) = 1/(1-P)$
- $U=(1-P)/(1+2\alpha)$



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
  - Also called "sliding window" protocols

Pipelined protocols

- **Two generic forms of pipelined protocols:**
  1. ***go-Back-N***
  2. ***selective repeat***

A.k.a "<u>sliding window</u>" protocols

Sliding Window Protocols: Efficiency

t_{frame}

t_{prop}

Data

Ack

$$U = \frac{N t_{frame}}{2t_{prop} + t_{frame}} = \begin{cases} \dfrac{N}{2\alpha+1} \\ 1 \text{ if } N > 2\alpha+1 \end{cases}$$

Note: no loss or bit-errors!

Go-Back-N

Sender:

- **k-bit seq # in pkt header**
  - **Allows *upto $N = 2^k - 1$ packets in-flight*, unacked**
- **"Window": *limit* on # of consecutive unacked pkts**
  - **In GBN, window = N**

send_base, nextseqnum, window size N

already ack'ed | usable, not yet sent | sent, not yet ack'ed | not usable

Go-Back-N

- **ACK(n): ACKs all pkts up to, including seq # n - "cumulative ACK"**
  - **Sender may receive duplicate ACKs (see receiver)**
  - ***Robust* to losses on the reverse channel**
  - **Can pinpoint the *first packet lost*, but *cannot identify blocks* of lost packets in window**
- **One timer for oldest-in-flight pkt**
- ***Timeout* => retransmit pkt "base" and all higher seq # pkts in window**

GBN: 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 = nextseqnum + 1
   }
else
  refuse_data(data)
```

WAIT

```
rdt_rcv(rcv_pkt) && notcorrupt(rcvpkt)
______________
base = getacknum(rvcpkt)+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))
```


GBN: Receiver Extended FSM

```
default
______________
udt_send(sndpkt)
```

WAIT

```
rdt_rcv(rcvpkt) &&
  notcorrupt(rcvpkt)  &&
  hasseqnum(rcvpkt,expectedseqnum)
______________
extract(rcvpkt,data)
deliver_data(data)
make_pkt(sndpkt,ACK,expectedseqnum)
udt_send(sndpkt)
```

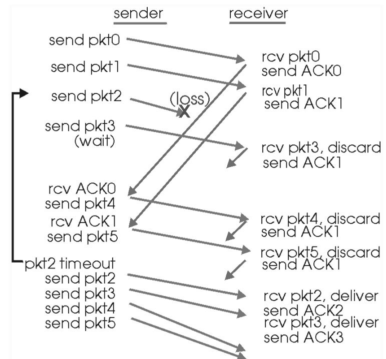
<u>Receiver:</u>

- **ACK-only: always send ACK for correctly-received pkt with highest *in-order* seq #**
  - **may generate duplicate ACKs**
  - **need only remember `expectedseqnum`**
  - **A.k.a. "receiver window"**

GBN: Receiver Extended FSM

- **out-of-order pkt:**
  - **discard (don't buffer) -> no receiver buffering!**
  - **ACK pkt with highest in-order seq #**

GBN in action



Mechanisms learnt so far…

- **Checksum: detects corruption *in pkts & acks***
- **ACK: "packet correctly received"**
  - **<u>Duplicate</u> ACK: "packet *in*correctly received"**
  - **Cumulative ACK: acks all pkts upto & incl. seq #**
- **Sequence number: identifies packet or ack**
  - **1-bit sequence number used *both in forward & reverse* channels**
  - **k-bit sequence number in *both forward & reverse* channels**
- **Timeout only *at sender.***
  - ***One timer for entire window***
- **Window: *sender and receiver side. <u>Limits</u>* on what can be sent (or expected to be received).**
- **Buffering at sender only**

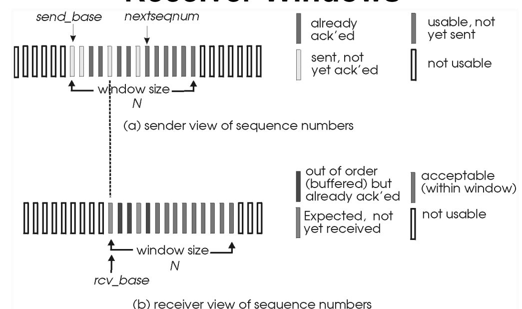
Capabilities learnt so far…

- **<u>Reliability capabilities achieved:</u>**
  - **An *error-free* channel**
  - **A *forward & reverse* channel with *bit*-errors**
  - **Detects *duplicates* of packets/acks**
  - ***NAKs eliminated***
  - **A *forward & reverse* channel with *packet*-errors (loss)**
  - ***Pipelining* efficiency.**
    - **Go-back-N: Entire outstanding window retransmitted if pkt loss/error**

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 #

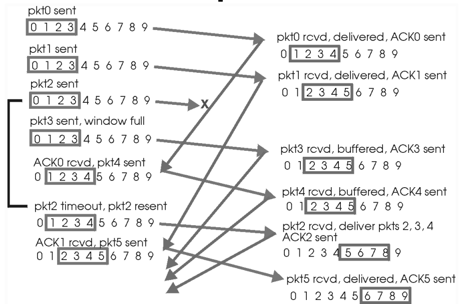
Selective repeat
Receiver

Pkt n in [rcvbase, rcvbase+N-1]
- send ACK(n): "selective ack"
- 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)

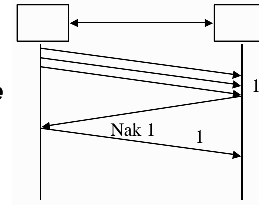
otherwise:
- ignore

Selective repeat in action



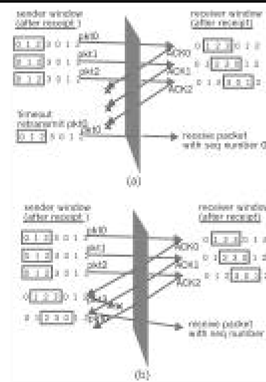
Selective Repeat: Performance

- **Error Free:**
  U = 1 ***if N>2a+1***
  N/(2α+1) ***otherwise***



- **With Errors:**
  $N_r = \Sigma\ i\ P^{i-1}(1-P)$
  $= 1/(1-P)$
- **U= (1-P)** ***if N>(1+2a)***
  N(1-P)/(1+2α) ***otherwise***

Selective Repeat: Dilemma



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?
A: sequence # space >= 2*window

Reliability Mechanisms: Summary

- Checksum: detects corruption *in pkts & acks*
- ACK: "packet correctly received"
  - Duplicate ACK: "packet *in*correctly received"
  - Cumulative ACK: acks all pkts *upto & incl.* seq # (*GBN*)
  - Selective ACK: *acks pkt "n" only (selective repeat)*
- Sequence number: identifies packet or ack
  - 1-bit sequence number used *both in forward & reverse* channels
  - k-bit sequence number in *both forward & reverse* channels.
    - Let $N = 2^k - 1$ = sequence number space size

Reliability Mechanisms: Summary

- Timeout only *at sender.*
  - *One timer for entire window* (go-back-N)
  - *One timer per pkt* (selective repeat)
- Window: *sender and receiver side.*
  - ***Limits*** on what can be sent (or expected to be received).
  - Window size (W) upto N –1 (Go-back-N)
  - Window size (W) upto N/2 (Selective Repeat)
- Buffering
  - *Only at sender* (Go-back-N)
  - *Out-of-order buffering* at *sender & receiver* (Selective Repeat)

Reliability capabilities: Summary

- Reliability capabilities achieved:
  - An *error-free* channel
  - A *forward & reverse* channel with *bit*-errors
  - Detects *duplicates* of packets/acks
  - *NAKs eliminated*
  - A *forward & reverse* channel with *packet*-errors (loss)
  - *Pipelining* efficiency.
    - Go-back-N: Entire outstanding window retransmitted if pkt loss/error
    - Selective Repeat: only lost packets retransmitted
      - performance penalty if ACKs lost (because acks non-cumulative) & more complexity