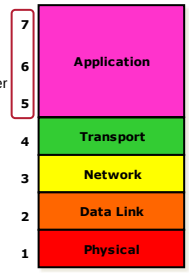
Internet Technology

05. Transport Layer

Paul Krzyzanowski
Rutgers University
Spring 2016

Transport Layer

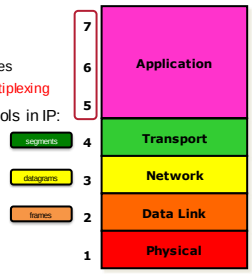
- Transport Layer
  - Provides logical communication channels between apps
- Transport layer managed by end systems
  - Routers are unaware; they provide network layer services
- Multiple transport protocols available
  - Under IP: TCP, UDP, SCTP, and more



Internet Protocol Layers

Transport Layer

- Network Layer
  - Logical connection between hosts
- Transport Layer
  - Logical connection between processes
  - Transport layer multiplexing & demultiplexing
- Most common transport-layer protocols in IP: TCP & UDP
  - UDP: unreliable data transfer
  - TCP
    - Reliable data transfer
    - In-order delivery
    - Flow control
    - Congestion control



Internet Protocol Layers

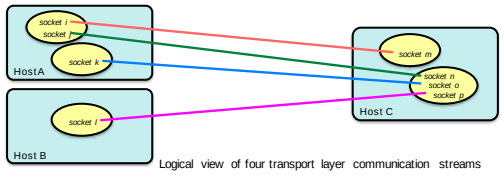
Today, we'll discuss

- Transport layer multiplexing/demultiplexing
- Reliable data transfer

Transport Layer Multiplexing & Demultiplexing

Transport Layer Multiplexing

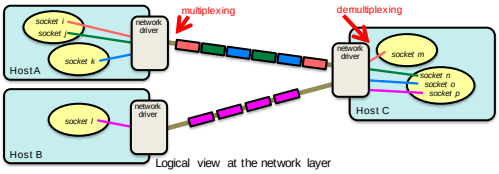
- Problem:
  Multiple communication channels over one network link
  - This is a problem whenever a protocol at one layer is used by multiple protocols or or communication sessions at a higher layer



Logical view of four transport layer communication streams

Transport Layer Multiplexing

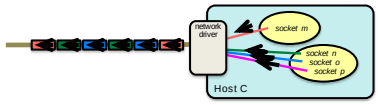
- Problem:
  Multiple communication channels over one network link
  - This is a problem whenever a protocol at one level is used by multiple protocols or or communication session at one
- Need to identify which segment belongs to which channel



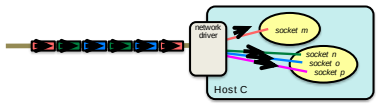
Logical view at the network layer

Multiplexing & Demultiplexing

Multiplexing



Demultiplexing



How is it done?

- Transport layer protocols in IP have port numbers
  - 16 bit integers (0 .. 65535)
  - IP header (network layer) has source address, destination address
  - TCP/UDP headers (transport layer) have source port, destination port
- Each socket is uniquely identified in the operating system
- Before a socket can be used, it is created & named
  - *socket* system call creates a unique socket
  - *bind* system call associates a local address with the socket
    - With an address of INADDR_ANY, the socket is associated with ALL local interfaces
    - With a port of 0, the OS assigns a random unused port number to the socket

UDP multiplexing & demultiplexing

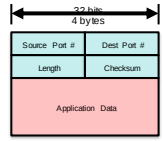
- A UDP socket is identified by its port number
- All UDP segments addressed to a specific port # will be delivered to the socket identified by that port number
  - A socket will request data via recv(), recvfrom(), or recvmsg() system calls
  - OS looks for a UDP socket with a matching destination port: hash table of socket structures; hash key created from UDP destination port
- Limited demultiplexing
  - Segments addressed to the same *(host, port)* from different processes or different systems will be delivered to the same socket!
  - The receiver can get the source address & port to know how to address reply messages

Why use UDP?

- Control the timing of data
  - A UDP segment is passed to the network layer immediately for transmission
  - TCP uses congestion control to delay transmission
- Preserve message boundaries
  - With TCP, multiple small messages may be consolidated into one TCP segment
- No connection setup
  - TCP requires a three-way handshake to establish a connection
- No state to keep track of
  - Less memory, easier fault recovery, simple load balancing
- Less network overhead
  - 8-byte header instead of TCP's 20-byte header

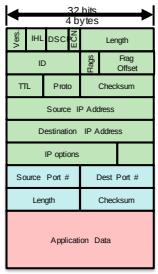
UDP Structure

- Defined in RFC 768
- Eight byte header



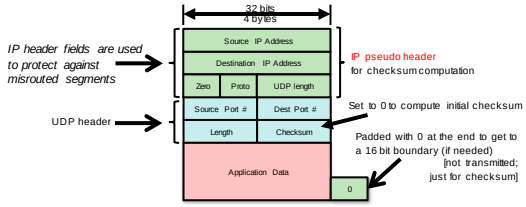
UDP Structure in context

Eight byte header within a 20 byte IP header



UDP Checksum

- IP does not guarantee error-free packet delivery
- The UDP header contains a 16-bit checksum
  - Checks for data corruption
- Checksum is generated by the sender and validated only by the receiver only: segments with bad checksums are simply dropped



UDP Checksum Calculation

- Sender
  - Iterate over 16-bit words in the Pseudo header + UDP segment
  - UDP checksum field = 0
  - Create a one's complement checksum
    - Add two 16-bit values. If overflow, add 1 to the result
    - Do this for all the data you need to checksum
    - Invert the bits of the result to get the checksum value
- Receiver
  - Perform the same one's complement sum on all data *including* the checksum field
  - The result should be all 1s (**0xffff**)

The same checksum calculation is used for the IP header, UDP header, & TCP header

One's Complement Checksum Example

- How to compute a One's complement
  - Sum the numbers
  - Add any overflow carry to the result
- Create checksum for:

```
0110 1011 0000 1010
1011 0001 1100 1000
1100 0000 1111 0101
```

```
  0110 1011 0000 1010
+ 1011 0001 1100 1000
= 10001 1100 1101 0010
+                    1
= 0001 1100 1101 0011
+ 1100 0000 1111 0101
= 1101 1101 1100 1000
```

- Then invert the bits

```
~ 1101 1101 1100 1000
= 0010 0010 0011 0111   ← checksum
```


One's Complement Checksum Example

- Validate
  - Sum the numbers, including the checksum

```
  0110 1011 0000 1010
+ 1011 0001 1100 1000
+ 1100 0000 1111 0101
+ 0010 0010 0011 0111   ← add the checksum
```

```
  0110 1011 0000 1010
+ 1011 0001 1100 1000
= 10001 1100 1101 0010
+                    1
= 0001 1100 1101 0011
+ 1100 0000 1111 0101
= 1101 1101 1100 1000
+ 0010 0010 0011 0111
= 1111 1111 1111 1111   ← add checksum
```

- A result of all 1's (= -0) means the transmission was good

TCP multiplexing & demultiplexing

- Every TCP socket is identified by:

  (source address, destination address, source port, destination port)

- A TCP socket has a state:
  - LISTEN: the socket is used only for accepting connections
  - ESTABLISHED: the socket is connected
  - Other states that we'll ignore for now:
    - Connection setup:
      - SYN_SENT: trying to establish a connection
      - SYN_RCVD: received a connection request
    - Connection teardown:
      - FIN_WAIT_1: socket has been closed by the local application; no acknowledgement from remote
      - FIN_WAIT_2: socket has been closed by the local application; remote acknowledged the closing
      - CLOSING: socket has been closed by the local & remote apps; remote has not acknowledged close
      - TIME_WAIT: connection closed; waiting to be sure that the remote side received the last ACK
- Let's look at an example

Server: Create a new socket

```
svr = socket(AF_INET, SOCK_STREAM, 0);
```

Address family: Internet (IPv4)
Type: "stream" – connection-oriented (TCP)

Create a new socket at the server: it has no addresses so far

Client (135.10.10.1)

Local Addr	Local Port	Remote Addr	Rem Port	State

Server (192.115.8)

svr →

Local Addr	Local Port	Remote Addr	Rem Port	State

N.B.: We refer to a socket table here for convenience but it is just a logical construct. The actual implementation is operating-system specific but this data is generally stored in a list of socket buffer structures. On Linux, for example, the kernel function tcp_v4_lookup will search for either a listening or an established socket with specific addresses and ports (see net/ipv4/tcp_ipv4.c, around line 507)

Server: Bind – assign a local address

```
bind(svr);
```

Assign a local address (INADDR_ANY) and port (1234) to the socket

Client (135.10.10.1)

Local Addr	Local Port	Remote Addr	Rem Port	State

Server (192.115.8)

svr →

Local Addr	Local Port	Remote Addr	Rem Port	State
0.0.0.0	1234			

Server: Make it a listening socket

```
listen(svr, 10);
```

Set the state of the socket to *listen*. This socket can *only* be used to accept connections

Client (135.10.10.1)

Local Addr	Local Port	Remote Addr	Rem Port	State

Server (192.115.8)

svr →

Local Addr	Local Port	Remote Addr	Rem Port	State
0.0.0.0	1234			LISTEN

Server: Wait for a connection

```
snew = accept(svr);
```

Wait for an incoming connection on this socket

Client (135.10.10.1)

Local Addr	Local Port	Remote Addr	Rem Port	State

Server (192.115.8)

svr →

Local Addr	Local Port	Remote Addr	Rem Port	State
0.0.0.0	1234			LISTEN

Client: Create a new socket

```
s = socket();
```

Create an new socket at the client: no addresses so far

Client (135.10.10.1)

Local Addr	Local Port	Remote Addr	Rem Port	State

← s

Server (192.115.8)

svr →

Local Addr	Local Port	Remote Addr	Rem Port	State
0.0.0.0	1234			LISTEN

Client: Assign a local address & port

```
bind(s);
```

Assign any local address (INADDR_ANY) and have the OS pick a port (port=0)

Client (135.10.10.1)

Local Addr	Local Port	Remote Addr	Rem Port	State
0.0.0.0	7801			

← s

Server (192.115.8)

svr →

Local Addr	Local Port	Remote Addr	Rem Port	State
0.0.0.0	1234			LISTEN

Client: Connect to the server

```
connect(s, dest_addr);
```

Connect to address 192.11.5.8, port 1234

Client (135.10.10.1)

Local Addr	Local Port	Remote Addr	Rem Port	State	
0.0.0.0	7801				← s

connection

Server (192.115.8)

	Local Addr	Local Port	Remote Addr	Rem Port	State
svr →	0.0.0.0	1234			LISTEN
snew →	192.11.5.8	1234	135.10.10.1	7801	SYN_RCVD

- Send a connection establishment request to address 192.11.5.8, port 1234 (TCP segment to port 1234 with a connection setup bit set; we'll look at the exact handshake later)
- On the server, search the table for a LISTEN socket where
  - packet's destination addr == table's local addr (0.0.0.0 matches any incoming addr)
  - packet's destination port == table's local port
- Create a new socket for the connection

Client: Complete the connection

```
connect(s, dest_addr);
```

Server acknowledges the connection; Client fills in the entry

Client (135.10.10.1)

Local Addr	Local Port	Remote Addr	Rem Port	State	
0.0.0.0	7801	192.11.5.8	1234	ESTABLISHED	← s

Server (192.115.8)

	Local Addr	Local Port	Remote Addr	Rem Port	State
svr →	0.0.0.0	1234			LISTEN
snew →	192.11.5.8	1234	135.10.10.1	7801	ESTABLISHED

Now we can talk!

Communicate

Client-to-server communication

– Server finds socket by searching for a TCP socket with these properties:
1. Status == ESTABLISHED
2. IP src addr == remote addr
3. TCP src port = remote port
4. IP dest addr == local addr
5. TCP dest port == local port

Client (135.10.10.1)

Local Addr	Local Port	Remote Addr	Rem Port	State	
0.0.0.0	7801	192.11.5.8	1234	ESTABLISHED	← s

Server (192.115.8)

	Local Addr	Local Port	Remote Addr	Rem Port	State
svr →	0.0.0.0	1234			LISTEN
snew →	192.11.5.8	1234	135.10.10.1	7801	ESTABLISHED

Communicate

Server-to-client communication

– Client finds socket by searching for a TCP socket with these properties:
1. Status == ESTABLISHED
2. IP src addr == remote addr
3. TCP src port = remote port
4. IP dest addr == local addr
5. TCP dest port == local port

Client (135.10.10.1)

Local Addr	Local Port	Remote Addr	Rem Port	State	
0.0.0.0	7801	192.11.5.8	1234	ESTABLISHED	← s

Server (192.115.8)

	Local Addr	Local Port	Remote Addr	Rem Port	State
svr →	0.0.0.0	1234			LISTEN
snew →	192.11.5.8	1234	135.10.10.1	7801	ESTABLISHED

Two clients sharing the same port

Different source address disambiguates the sessions

Client (135.10.10.2)

Local Addr	Local Port	Remote Addr	Rem Port	State
0.0.0.0	7801	**192.11.5.8**	1234	ESTABLISHED

Client (135.10.10.1)

Local Addr	Local Port	Remote Addr	Rem Port	State
0.0.0.0	7801	**192.11.5.8**	1234	ESTABLISHED

Server (192.115.8)

	Local Addr	Local Port	Remote Addr	Rem Port	State
svr →	0.0.0.0	1234			LISTEN
snew1 →	192.11.5.8	1234	**135.10.10.1**	7801	ESTABLISHED
snew2 →	192.11.5.8	1234	**135.10.10.1**	7801	ESTABLISHED

Two endpoints sharing the same address

The OS will not allow two sockets to share the same port on one client

Client (135.10.10.1)

Local Addr	Local Port	Remote Addr	Rem Port	State
0.0.0.0	**7801**	192.11.5.8	1234	ESTABLISHED
0.0.0.0	**7802**	192.11.5.8	1234	ESTABLISHED

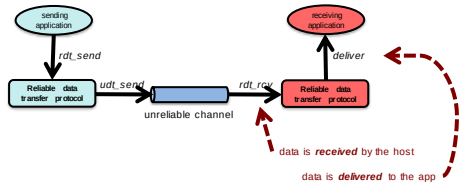
Server (192.115.8)

	Local Addr	Local Port	Remote Addr	Rem Port	State
svr →	0.0.0.0	1234			LISTEN
snew1 →	192.11.5.8	1234	135.10.10.1	**7801**	ESTABLISHED
snew2 →	192.11.5.8	1234	135.10.10.1	**7802**	ESTABLISHED

Reliable Data Transfer

Reliable Data Transfer (RDT) Goal

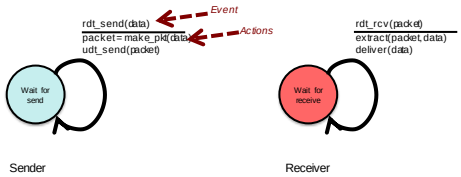
Develop a protocol for transmitting data reliably over an unreliable network



RDT over a reliable channel

- Assume the channel is reliable
- Trivial – nothing to do!

Here's the finite state machine (FSM):



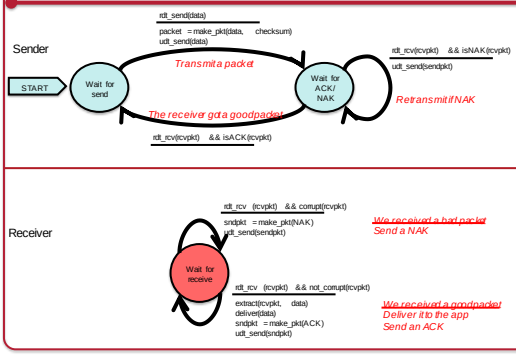
RDT over a channel with bit errors

- All packets are received
- Some might be corrupt
- Approach
  - Acknowledge each packet
    - Positive acknowledgement (ACK): *"I got it; looks good!"*
    - Negative acknowledgement (NAK): *"Please repeat"*
  - Sender retransmits a packet if it receives a NAK
  - ARQ (**A**utomatic **R**epeat re**Q**uest)
    - Set of protocols that use acknowledgements & retransmission

We need to support three capabilities

1. Error detection
   - How do we know if the packet is corrupt?
   - Use a checksum (error detecting code)
2. Receiver feedback
   - The receiver will acknowledge each packet with an ACK or NAK
3. Retransmission
   - If a sender gets a NAK, the packet will be retransmitted

RDT over a channel with bit errors



Stop-and-wait

- The sender cannot send any data until it receives an ACK for the previously sent packet
- This type of protocol is a stop-and-wait protocol

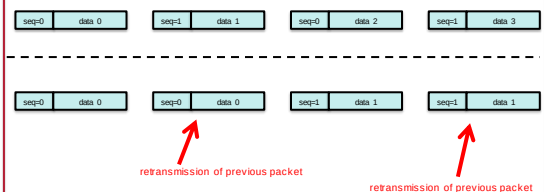
What about a corrupted ACK/NAK message?

- The sender does not know whether the last packet was received correctly or not
- We can
  - Have the sender send a "please repeat" in response to a corrupt ACK/NAK
    - But what if that gets corrupted?
  - Add a robust error correcting code
    - Works for a channel that does not lose data
  - Resend the data in response to a corrupted ACK/NAK
    - Duplicate packets may be received
    - Receiver needs to distinguish between new data & a retransmission
    - Use a **sequence number**. Here, we only need a 1-bit number.

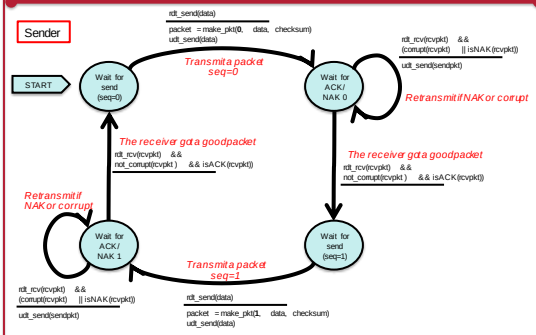
A 1-bit sequence number

Sequence bit flip-flops between consecutive messages

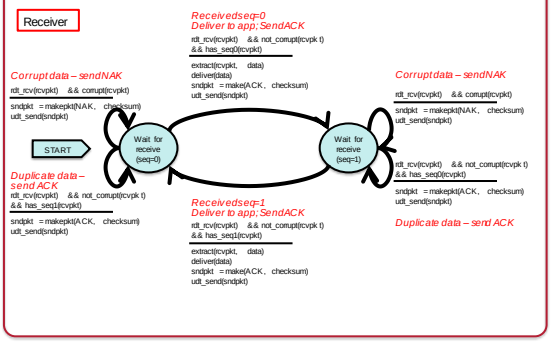
Alternating bit protocol



RDT over a channel with bit errors



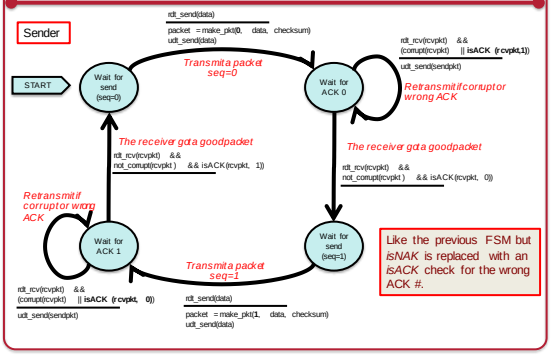
RDT over a channel with bit errors



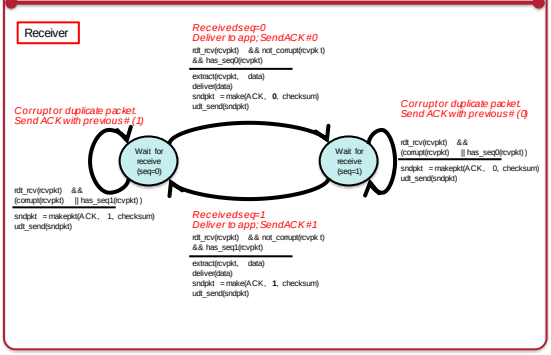
RDT over a channel with bit errors

- If a corrupted packet is received
  - Send a NAK
- If a duplicate packet is received
  - Send an ACK since we already processed the packet
- We can get rid of NAKs
  - Send an ACK for the last correctly received packet
  - If a sender receives duplicate ACKs, it knows that the previous packet has not been received correctly
  - Modify protocol: add sequence numbers to ACKs

RDT over a channel with bit errors – no NAK



RDT over a channel with bit errors – no NAK



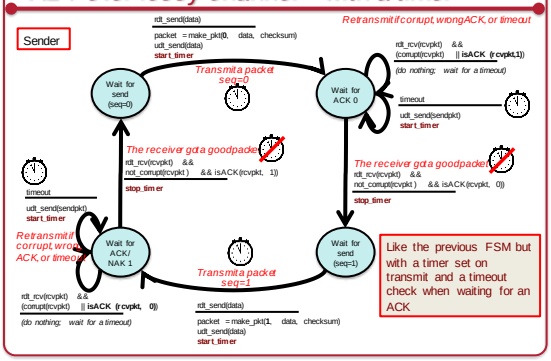
RDT over a lossy channel

- We considered only bit errors
  - Packets were always delivered
- How do we detect & deal with packet loss?

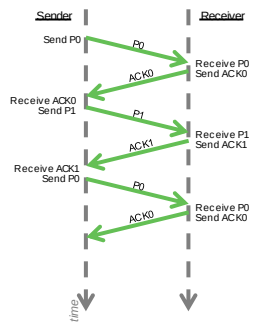
Dealing with packet loss

- Burden of detection & recovery is on sender
- If sender's packet is lost OR receiver's ACK is lost
  - Sender will not get a reply from the receiver
- Approach
  - Introduce a **countdown timer**. Set the timer at transmit
  - If time-out and no reply, retransmit
  - How long to wait? Maximum round-trip delay?
    - Long wait until we initiate error recovery
    - Pick a "likely loss" time
    - Retransmit if no response within that time
    - Introduces possibility of **duplicate packets**
      - But we already know how to deal with them

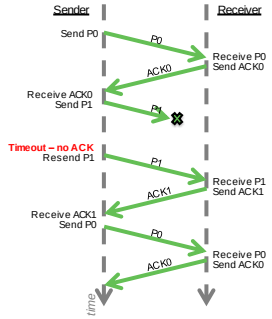
RDT over lossy channel – with a timer



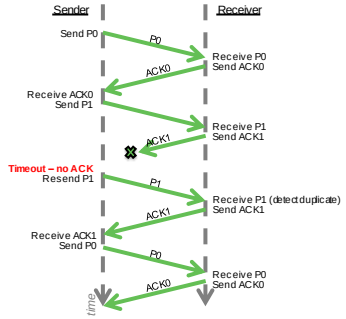
RDT – Alternating Bit Protocol: no loss



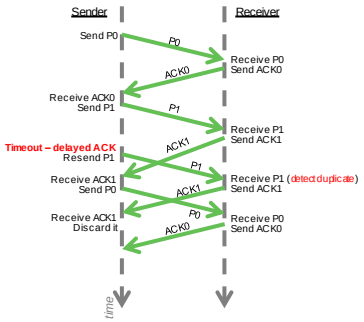
RDT – Alternating Bit Protocol: lost Packet



RDT – Alternating Bit Protocol: lost ACK



RDT – Alternating Bit Protocol: early timeout



Network utilization with stop-and-wait

- A **stop-and-wait** protocol gives us horrible **network utilization**
- Consider
  - Cross-country link ⇒ Round-trip propagation delay (RTT) ≈ 30 ms
  - Assume 1 Gbps link (ignore router delays), $R = 10^9$ bits/second
  - Assume 1,000-byte packets (L = 8,000 bits)
  - Time to transmit the packet: $d_{trans} = L \div R = 8{,}000 \div 10^9 = 8\ \mu s$
- With a stop-and-wait protocol
  - one-way delay = $d_{trans} + d_{prop}$ = (30 ms ÷ 2) + 8 µs = 15.008 ms
  - Assume ACK packets are tiny; one-way delay for ACK packet = 15 ms
    - ACK is received at 15.008 + 15 = 30.008 ms
  - Next packet can be sent (15.008 + 15) = 30.008 ms after the first one
  - Utilization = fraction of time sender is sending bits into the channel

$$U = \frac{L/R}{RTT + (L/R)} = \frac{0.008}{30.008} = 0.00027 = 0.027\%$$

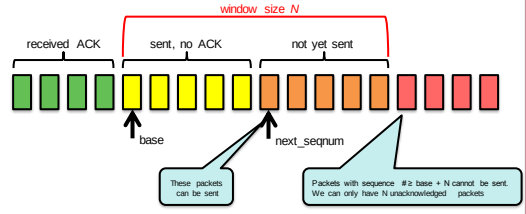
The sender can transmit 1,000 bytes in 30.008 ms: 267 kbps on a 1 Gbps link!

Improve Network Utilization: Pipelining

- Don't wait for an acknowledgement before sending the next packet
- But then we need to
  1. Increase the range of sequence numbers
     - Each in-transit packet needs a unique number
  2. Hold on to unacknowledged packets at sender
  3. Hold on to out-of-sequence packets at receiver
- Two approaches for pipelined error recovery
  - **Go-Back-N**
  - **Selective Repeat**

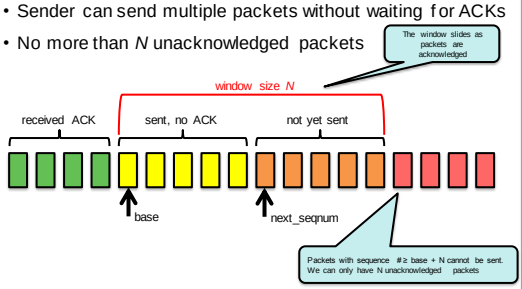
Go-Back-N (GBN)

- Sender can send multiple packets without waiting for ACKs
- No more than N unacknowledged packets



Go-Back-N (GBN)

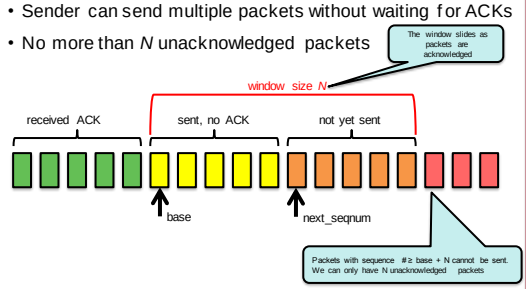
- Sender can send multiple packets without waiting for ACKs
- No more than *N* unacknowledged packets



GBN = Sliding Window Protocol

Go-Back-N (GBN)

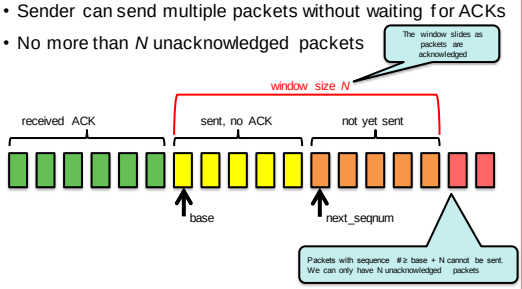
- Sender can send multiple packets without waiting for ACKs
- No more than *N* unacknowledged packets



GBN = Sliding Window Protocol

Go-Back-N (GBN)

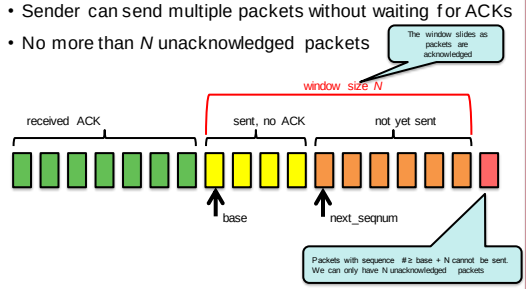
- Sender can send multiple packets without waiting for ACKs
- No more than *N* unacknowledged packets



GBN = Sliding Window Protocol

Go-Back-N (GBN)

- Sender can send multiple packets without waiting for ACKs
- No more than *N* unacknowledged packets



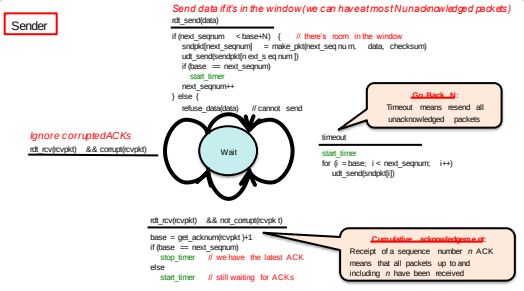
GBN = Sliding Window Protocol

Sequence numbers

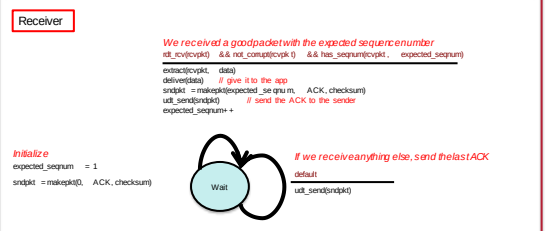
A sequence number will take up a fixed #, *k*, of bits in the header

- Range of sequence numbers is 0 .. 2^k-1
- Modulo 2^k arithmetic: 2^k-1 increments to 0

Extended FSM for a GBN sender



Extended FSM for a GBN receiver



The receiver discards out-of-order packets

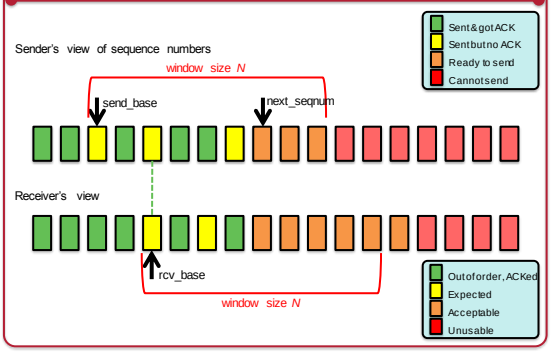
If packet *n* is lost and *n+1* arrives, the receiver does not buffer packet *n+1*. The sender will retransmit all unacknowledged packets (go back N).

The receiver has to only keep track of the next sequence number.

Selective Repeat

- Problem with Go-Back-N
  - With a large window size and large delays, many packets can be in the pipeline
  - A single error can cause GBN to retransmit many packets (all that are unacknowledged)
  - If *P(channel error)* increases, the pipeline can become filled with excess retransmissions
- Selective Repeat Protocol
  - Retransmit only those packets that were lost or corrupted
  - Receiver must acknowledge each correctly received packet
    - Even if it is out of order
    - Out of order packets must be buffered
  - Window size *N* = limit of number of outstanding packets
    - But some packets in the window may be acknowledged
    - The window slides when the earliest packet in the window is acknowledged

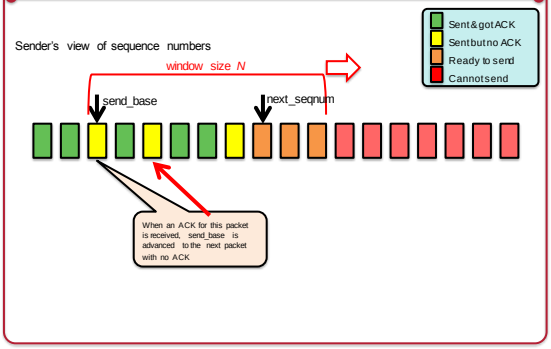
Selective Repeat Windows



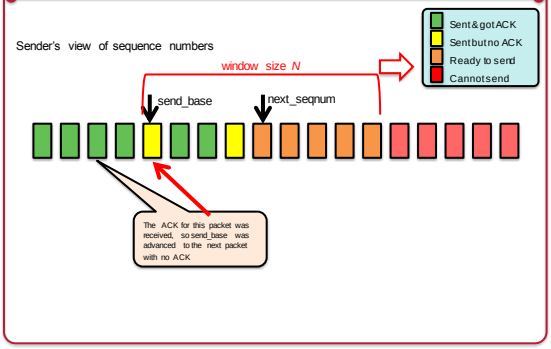
Selective Repeat: sender operation

- Send requests from application
  - Check next available sequence #
  - If no room in window, reject (or buffer)
  - Else send the packet (with sequence #)
- Timeout
  - Each packet has its own timer
  - Retransmit only the specific packet on timeout
- ACK received
  - If packet is within window
    - Mark packet as received
    - If sequence # == send_base
      advance the base (start of window) to the next unacknowledged sequence number

Out-of-order ACKs



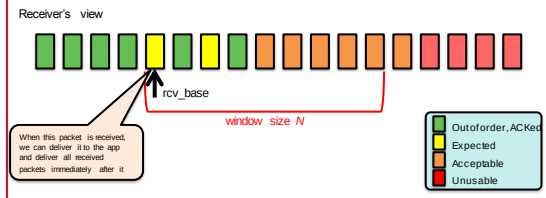
Out-of-order ACKs



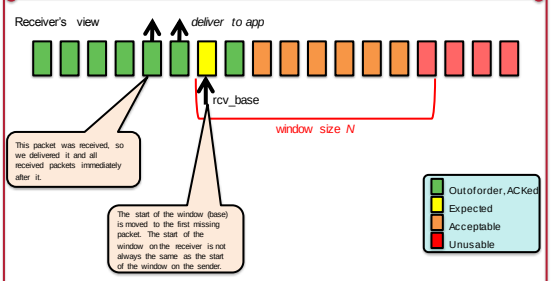
Selective Repeat: receiver operation

- Good packet with seq # in [rcv_base, rcv_base+*N*-1]
  - Packet is within the receiver's window
  - Send ACK for that sequence #
  - If sequence # == rcv_base
    - Deliver packet to app and deliver all successive packets that have been received
    - Adjust start of window (rcv_base)
- Good packet with seq # in [rcv_base-*N*, rcv_base-1]
  - Packet is within the *before* receiver's window
  - We already saw it – but send ACK anyway
- Anything else
  - Ignore the packet

Selective Repeat: receiving packets



Selective Repeat: receiving packets



The end