Computer Networks - Transport Layer Part 4

Kelly Chen - kjc8rx

13 October 2021

1 rdt3.0

1.1 Channels with Errors and Loss

New channel assumption: The channel can not only corrupt the data and corrupt the acknowledgements, but also lose it.

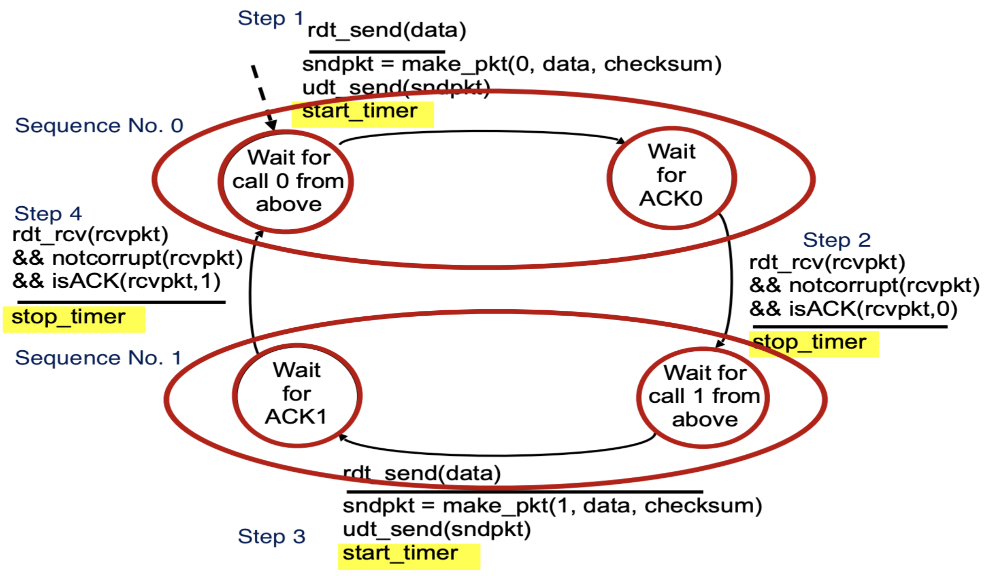
- Checksum, sequence numbers, ACKs, re-transmissions will be of help... but not quite enough

Q: How do humans handle lost sender-to-receiver words in conversation?
Timer: Wait a certain amount of time before we ask ”What did you say?”

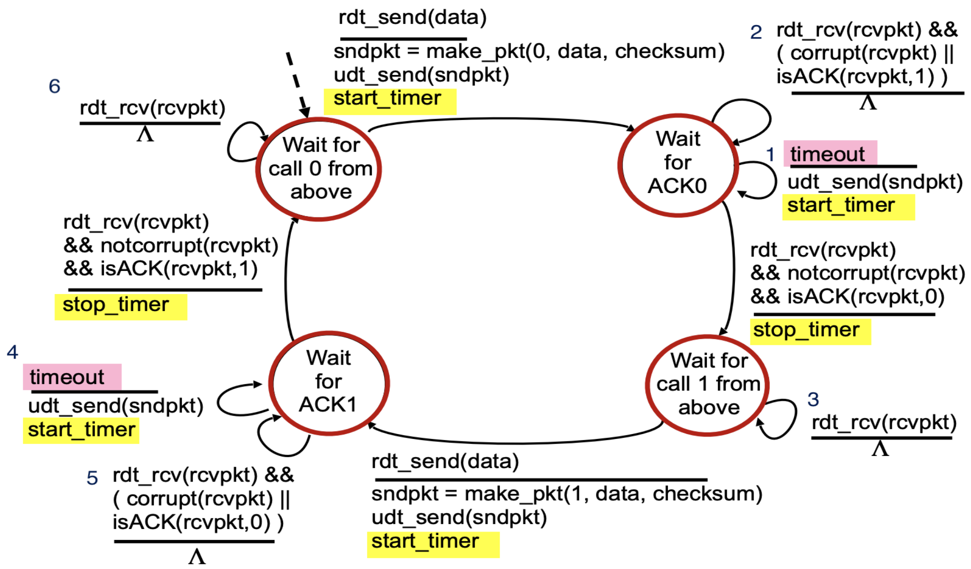
Approach: Sender waits ”reasonable” amount of time for ACK. If no ACK is received (ACK is delayed or lost), retransmit. This could result in a duplicate if ACK was delayed, but we may look at the sequence number. Receiver must specify a sequence number of the packet being ACKed. We will have a countdown timer to interrupt after a ”reasonable” amount of time.

1.2 rdt3.0 Sender

State machine for sequence numbers 0 and 1:


Step 1
rdt_send(data)
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
start_timer
Sequence No. 0
Wait for call 0 from above
Wait for ACK0
Step 4
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,1)
stop_timer
Step 2
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,0)
stop_timer
Sequence No. 1
Wait for ACK1
Wait for call 1 from above
rdt_send(data)
sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)
Step 3
start_timer

State machine with corrupt states:


rdt_send(data)
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
start_timer
2 rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt,1))
Λ
6
rdt_rcv(rcvpkt)
Λ
Wait for call 0 from above
Wait for ACK0
1 timeout
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,1)
stop_timer
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,0)
stop_timer
4 timeout
udt_send(sndpkt)
start_timer
Wait for ACK1
Wait for call 1 from above
3 rdt_rcv(rcvpkt)
Λ
rdt_send(data)
sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)
start_timer
5 rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt,0))
Λ

Explanations of the above state machine:

1. Didn't receive ACK for pkt 0 in time, resend pkt and restart timer
2. Receive pkt that was corrupt, or ACK for another state (ACK for 1), don't do anything and wait for timer to expire. The timer will do fthe reset. We treat corrupted acknowledgements as lost

3. Duplicate in-flight packet
4. Wait for an ACK, do send again, and start timer again
5. Corrupt or ACK for 0, in this case do nothing
6. In waiting, if we receive something in this state it must have been a duplicate from something else in the pipeline from before because we are waiting for an rdt send, so do nothing

1.3 Performance of rdt3.0 (stop-and-wait)

We are sending and waiting for acknowledgement. We measure performance with utilization (U_{sender}), or the fraction of time sender busy sending.

Example: 1 Gbps link, 15 ms prop. delay, 8000 bit packet

- Data transmit time:

$$D_{\text{trans}} = \frac{L}{R} = \frac{8000bits}{10^9 bits/sec} = 8microsecs$$

1.4 Stop-and-Wait Operation

Utilization = round trip time + transmit time / how much data we are sending

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

Utilization is really low, less than one percent. We can use pipelining.

1.5 Pipelined Protocols Operation

Pipelining: Sender allows multiple, "in-flight", yet-to-be-acknowledged packets

- Range of sequence numbers must be increased
- Buffering at sender and/or receiver

1.6 Pipelining: Increased Utilization

$$U_{\text{sender}} = \frac{3L/R}{RTT + L/R} = \frac{.0024}{30.008} = 0.00081$$

2 Go-Back-N

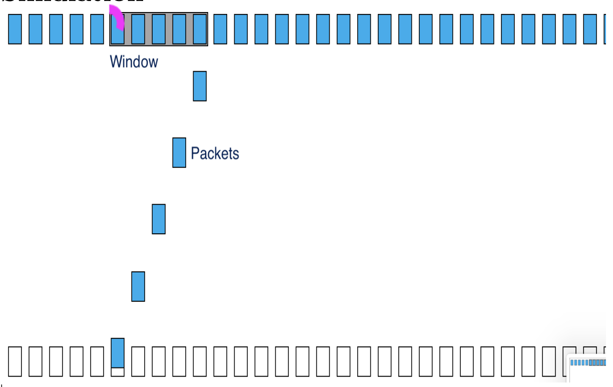
2.1 Go-Back-N Sender

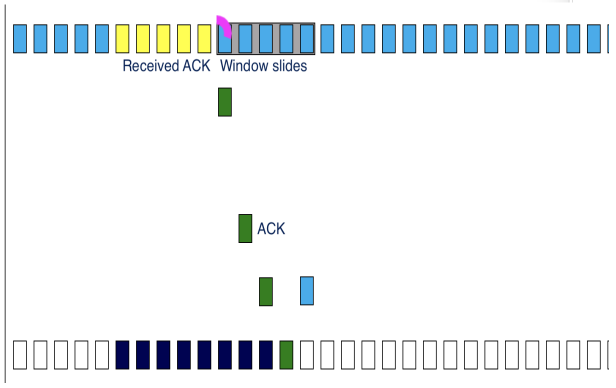
Window: How many packets we're going to send at a particular time. Window size is what we are going to pick, and is dynamic. If the receiver has a lot of space in its buffer, then we'll increase the window size. Otherwise, we shrink the window size.

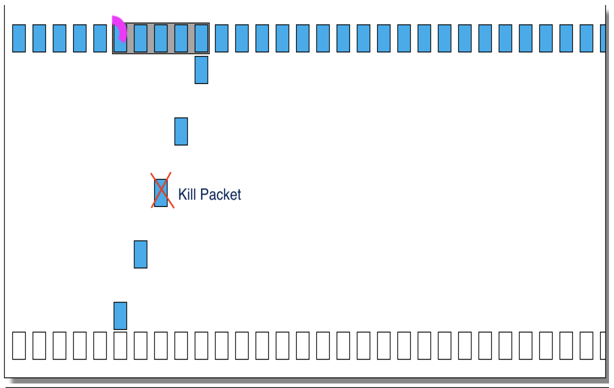
Cumulative ACK: Receive n packets close to each other, we will send a cumulative ACK for all packets up to sequence number n. We will just ACK the highest sequence number.

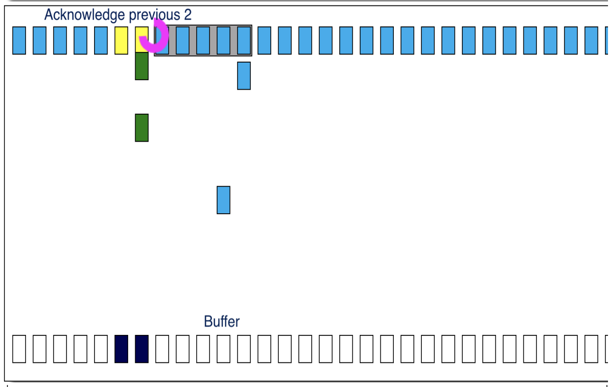
- On receiving ACK(n): move window forward to begin at n+1
- Timer for oldest in-flight packet
- Timeout(n): re-transmit packet n and all higher sequence number packets in window

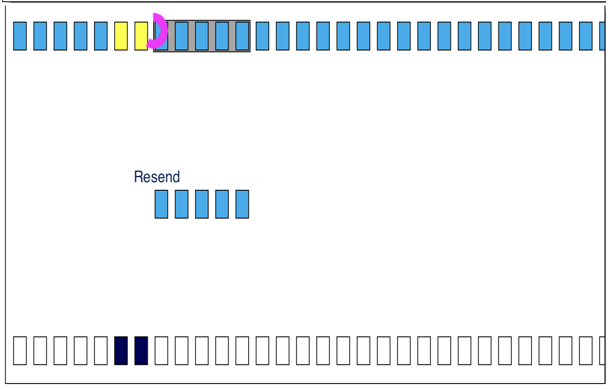
Simulation

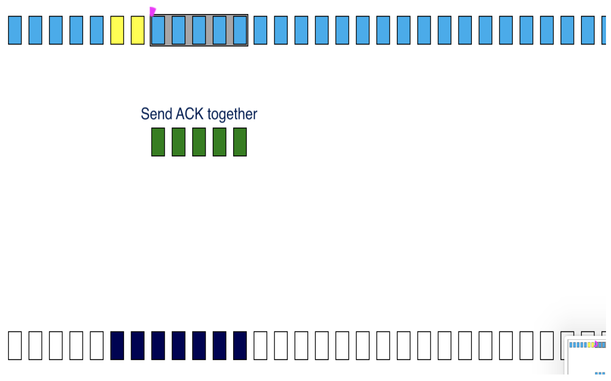





Kill Packet


Acknowledge previous 2
Buffer


Resend



2.2 Go-Back-N Receiver

We are only going to acknowledge a sequence number if we've received all the previous ones. We have the implementation decision to buffer or not buffer.

ACK-only: always send ACK for correctly-received packet so far, with highest in-order sequence number. We may generate duplicate ACKs.

If received out-of-order packet: generate ACK for the highest received in-order sequence number, but none of the received out-of-order packets.

3 Selective Repeat

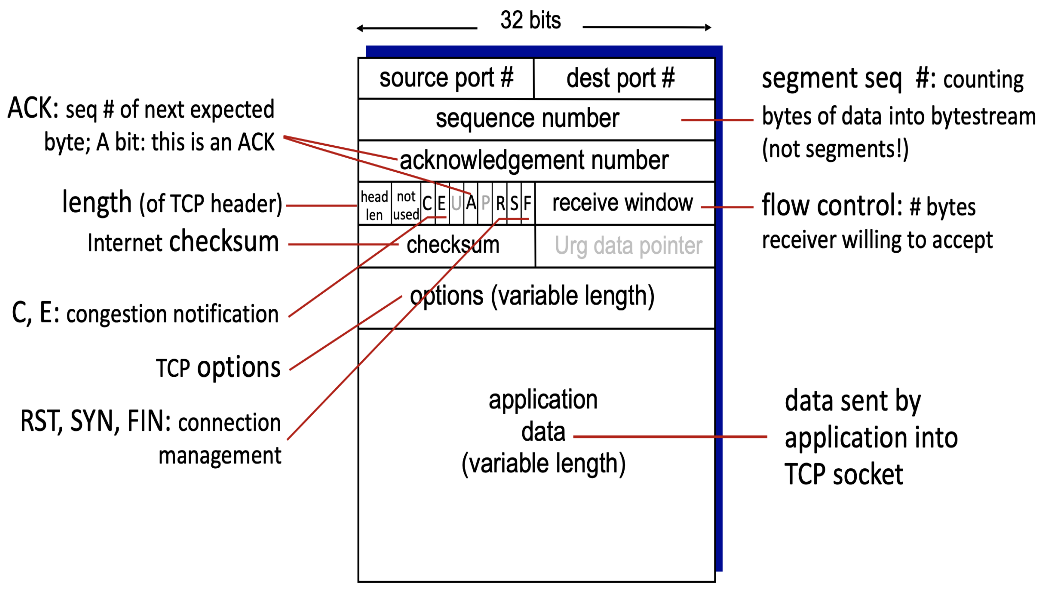
For selective repeat, we individually acknowledge each packet (instead of cumulative ACK) and buffer each packet as needed. The sender is going to have timeouts and retransmits for unacknowledged packets. Each packet is going to have a timer of its own. Timer is associated with packet instead of window.

4 TCP

- Point-to-point:
  - One sender, one receiver
- Reliable, in-order byte stream:
  - No ”message boundaries”
- Full duplex data:
  - Bi-directional data flow in same connection
  - MSS: maximum segment size
- Cumulative ACKs

- Pipelining:
  - TCP congestion and flow control set window size
- Connection-oriented:
  - Handshaking (exchange of control messages) initializes sender, receiver state before data exchange
- Flow controlled:
  - Sender will not overwhelm receiver

4.1 TCP Segment Structure


32 bits
source port #
dest port #
sequence number
acknowledgement number
head len
not used
C E U A P R S F
receive window
checksum
Urg data pointer
options (variable length)
application data (variable length)
ACK: seq # of next expected byte; A bit: this is an ACK
length (of TCP header)
Internet checksum
C, E: congestion notification
TCP options
RST, SYN, FIN: connection management
segment seq #: counting bytes of data into bytestream (not segments!)
flow control: # bytes receiver willing to accept
data sent by application into TCP socket

- Segment seq #: Random 32-bit number used to avoid session hijacking
- ACK: Flag in TCP packet header that indicates it's an ACK.
- Checksum: Not very effective
- TCP options: Variable length header has header length field, option to set SYN, FIN, and ACK flags
  - For nmap scans, does SYN scan and set SYN flag, starting TCP handshake.
- Receive window: Negotiate how many packets to send
- C, E: Flags for congestion control

- Urgent: Rarely used, pointer to location in data field. If urgent flag in set, go to data segment instead of parsing through whole thing

4.2 TCP Sequence numbers, ACKs

Packet assigned sequence number dependent on packet's position in sequence. For each packet we ACK, we're also going to have an acknowledgement number and set the ACK flag. TCP uses cumulative ACKs. We always acknowledge the next number in the sequence.

Q: How do we handle out-of-order segments? A: Up to the TCP implementation, choice to buffer or drop.

Telnet:

- Not encrypted
- Can be hacked with session hijacking
- Man in the middle -¿ session hijack
- Acknowledge the next byte in the session. Looks at the sending sequence number, and acknowledge the next byte in the sequence

4.3 TCP Round Trip Time, Timeout

Q: How to set TCP timeout value?

- Longer than RTT, but RTT varies
- Too short: premature timeout, unnecessary re-transmissions
- Too long: slow reaction to segment loss

A: Set it to the RTT.

Q: how to estimate RTT?

- SampleRTT: Measured time from segment transmission until ACK receipt
  - Ignore retransmissions
- Do a bunch of measurements over time and average them
- SampleRTT will vary, want estimated RTT "smoother"
  - Average several recent measurements, not just current SampleRTT

$$EstimatedRTT = (1 - \alpha) * EstimatedRTT + \alpha * SampleRTT$$

- Exponential weighted moving average (EWMA)
- Influence of past sample decreases exponentially fast
- Typical value:

$$\alpha = 0.125$$

- Timeout interval: EstimatedRTT plus "safety margin"
  - Large variation in EstimatedRTT: want a larger safety margin

$$TimeoutInterval = EstimatedRTT + 4 * DevRTT$$

- DevRTT: EWMA of SampleRTT deviation from EstimatedRTT:

$$DevRTT = (1 - \beta) * DevRTT + \beta * |SampleRTT - EstimatedRTT|$$

- Typical value:

$$\beta = 0.25$$

4.4 TCP Sender (Simplified)

If we received data from the application, we create a sequence with a sequence number and start a timer if it's not running. The timer could be thought as for the oldest unACKed segment. We may get a timeout event. If we transmit a segment and it calls timeout, then we restart the timer. If we receive an ACK, we acknowledge the previously unACKed segments, and update what is known to be ACKed. We start a timer if there are still unACKed segments.

4.5 TCP Flow Control

Q: What happens if network layer delivers data faster than application layer removes data from socket buffers?
A: We get buffer overflow.

We can have a receive window byte that says this is how many bytes the receiver can accept. This gives us full control.The receiver controls the sender so the sender won't overflow the buffer by transmitting too much information too fast. We'll advertise with the rwnd number free buffer space. It is the amount of remaining space in the buffer.

4.6 TCP Connection Management

The receiver window is exchanged early on in the "handshake".

Before exchanging data, sender/receiver "handshake":

- Agree to establish connection (each knowing the other willing to establish connection)
- Agree on connection parameters(e.g., starting sequence numbers)

5 Agreeing to Establish a Connection

2-way handshake:

- Does not work
- Client chooses sequence number x, request to establish a connection with server, and the server acknowledges x

Q: Will 2-way handshakes always work in network?
A: There will be variable network delays. If we get a timeout before the connection is established, and then client sends again, might think the connection is closed. There is also a duplicates and message reordering problem. We can't "see" the other side.

5.1 TCP 3-way Handshake

To optimize, we have both client and server acknowledge they've establish a connection. Previously, the client did not acknowledge.

6 Closing a TCP Connection

Closing the connection happens with the FIN bit, so we set the FIN flag on the responding packet. The server responds with FIN ACK and closes its connection.