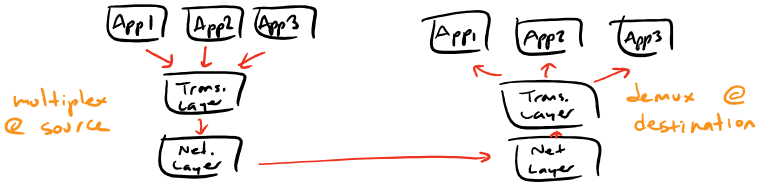
Chapter 3 - Transport Layer

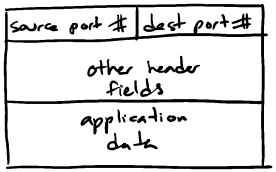
Wednesday, February 10, 2016 9:24 AM

→ Between application layer & network layer
→ provides logical communication between processes → network layer provides communication between hosts
→ can provide reliable communication even when underlying network protocol is unreliable (IP)

→ convert application-layer messages to and from transport-layer segments
↳ direct messages sent from host-to-host to the right processes using multiplexing & demultiplexing → passing through sockets

MULTIPLEXING / DEMULTIPLEXING





← takes data from sockets & adds header information to chunks to create segments to pass to network layer

port #s are 16-bits (0-65535), 0-1023 are restricted for well-known application protocols

UDP

→ minimal, connectionless transport protocol
↳ no handshaking, each UDP segment handled independently

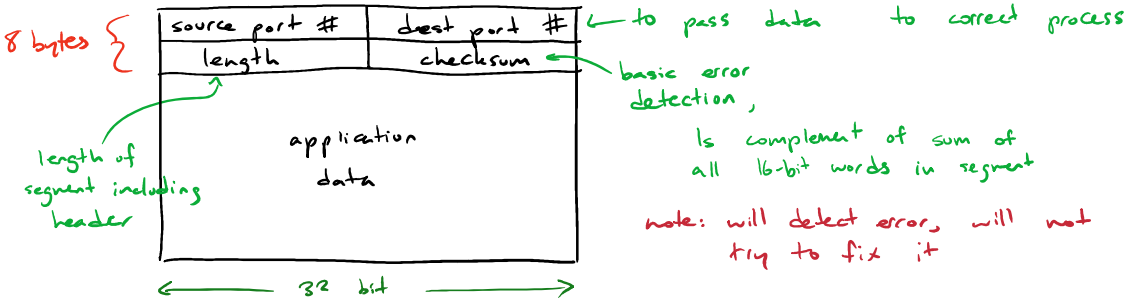
→ basically talking directly with IP except for some basic integrity checking & muxing/demuxing
→ no guarantee of reliable delivery
↳ must be built into the application itself

BENIFITS

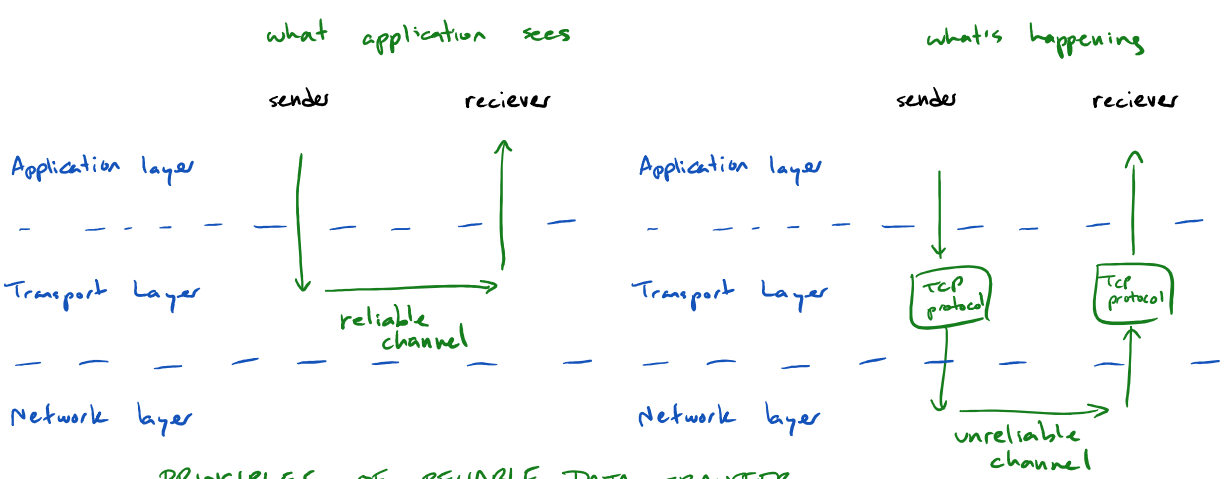
→ precise application-level control of what is sent & when
→ no connection establishment delay → why DNS uses UDP
→ no state/buffer information needed

↳ can handle many active clients
↳ TCP needs states for reliable data
→ small packet overhead
→ no congestion control → good for the app, bad for network

SEGMENT HEADER



RELIABLE DATA TRANSFER
→ no data is corrupted or lost and all are delivered in order



PRINCIPLES OF RELIABLE DATA TRANSFER
retransmission → bad/missing packets retransmitted
error detection → checksum
reciever feedback → acknowledgements (positive & negative)

ARQ → reliable transfer protocols based on retransmission
Automatic Repeat reQuest

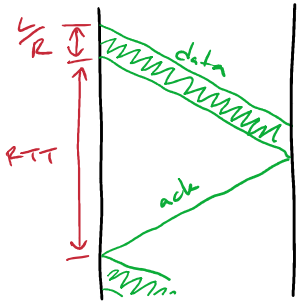
STOP & WAIT
→ sends packet, waits for acknowledgement, sends another, if recieves NAK, resends

TIMEOUT → if doesn't recieve ACK, will resend after timeout

ACK & NAK should also have checksum for reliability
↳ if garbled ACK or NAK, resend packet to be sure
↳ causes duplicate packets, so ACKs & NAKs must also have sequence numbers associated with packet

PIPELINING → sending a certain amount of packets one-after-another before the previous have been acknowledged
→ causes buffering

STOP & WAIT

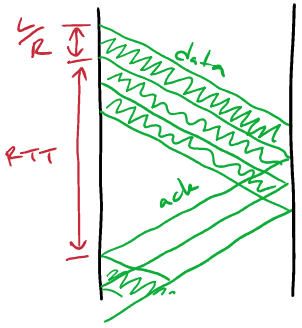


utilization

$$U_{sender} = \frac{L/R}{RTT + L/R}$$

fraction of time sender is busy doing work over entire time process for one sent packet

PIPELINING



$$U_{sender} = \frac{3L/R}{RTT + L/R}$$

← pipelining with 3 triples our utilization factor

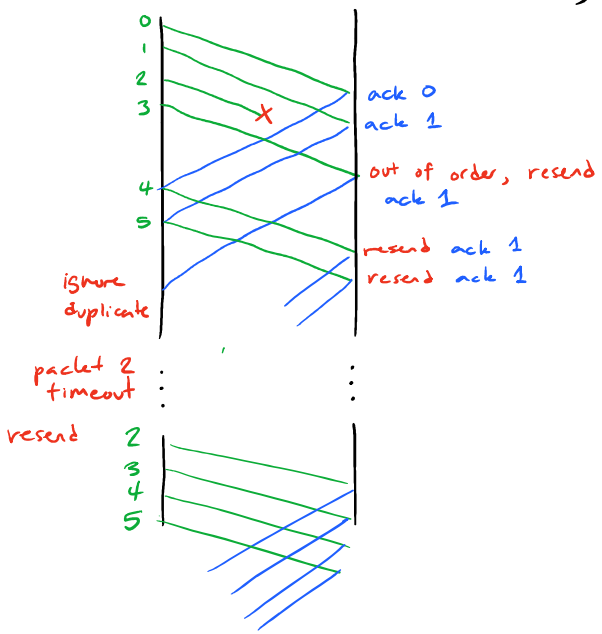
requirements:
- must increase range of sequence numbers
- sender/reciever have to buffer multiple packets
- more complicated loss/ack techniques
  ↳ Go-Back N, selective repeat

GO-BACK-N (GBN)
→ sender sends N packets w/o waiting for an ACK
"window size"
→ reciever will send ACK for all packets recieved in order
↳ "cumulative acknowledgement" → has recieved all packets up to & including #
→ if packet arrives out-of-order, it will be discarded & ACK for most recently recieved packet in order will be resent

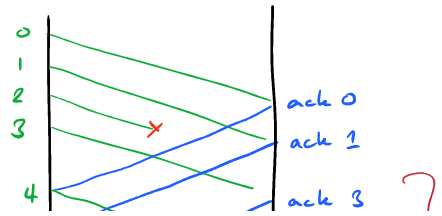
→ still has a timer like stop & wait, but upon timeout will resend unacknowledged packet and resend packets above unacknowledged

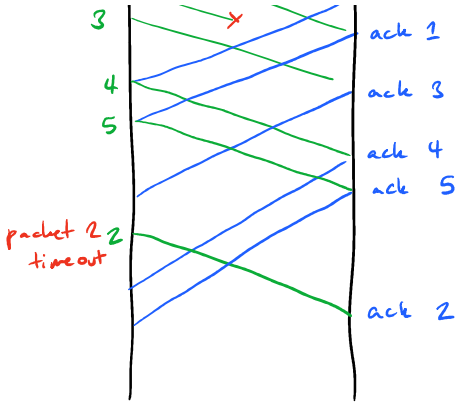
PROS
- reciever doesn't buffer any out-of-order packets
- only need to use on timer, for oldest unacknowledged packet

CONS
- wasteful b/c retransmitting all N packets when only one fails
  → bad performance if window size large and if bandwidth delay large



SELECTIVE REPEAT (SR)
→ reciever individually acknowledges correctly recieved packets
→ sender only retransmits packets lost or corrupted
→ out of order packets are buffered until missing packets recieved
→ each packet has own timer & timeout

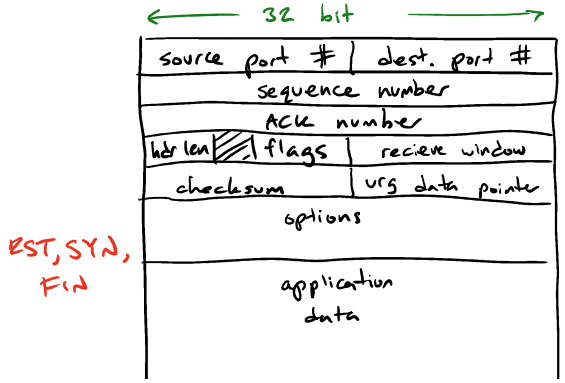




NOTE: window size $\leq \frac{1}{2}$ sequence number space for SR protocols

↳ otherwise sequence number rolls over and reciever doesn't know if recieving a retransmission or a new packet

TCP
- reliable data transfer
  ↳ flow control, sequence numbers, acknowledgements, timers, congestion control
  - flow control → don't overwhelm reciever
  - sequence numbers, acknowledgements, timers → reliability
  - congestion control → ↳ equal link bandwidth for each connection
- connection-oriented
  ↳ 3-way handshake, → create connection, → acknowledge/send request
  ↳ maintain connection throughout session
- uses cumulative acks



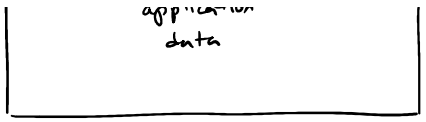
RST, SYN, FIN

20 bytes

no field that explicitly says what the entire length of the TCP segment is

↳ minimum header size, if options length = ∅

shared w/ UDP:
- destination/source port #
- length (although only header length w/ TCP)
- checksum

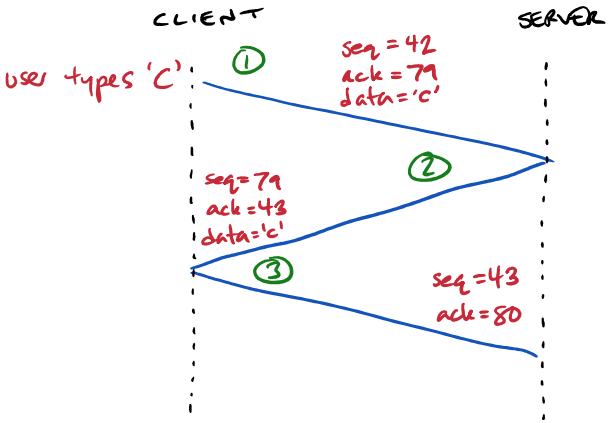


- destination/source port #
- length (although only header length w/ TCP)
- checksum

SEQUENCE NUMBERS → identify each byte to be delivered
→ segments are given the sequence number of the first byte in their data

ACKNOWLEDGEMENT NUMBERS → contain the sequence # of the next byte waiting to be recieved

example using Telnet



① 42 and 79 were chosen randomly in handshake
↳ represents first segment sent from server
↳ acking because that's what client is waiting for
↳ represents first segment sent from client

② acknowledges it has recieved segment 42
↳ because data 'c' was only one byte it tells client it now expects 42+1, byte 43,
and also echos back 'c'
↳ this is the first segment being sent by the server, so the sequence # is 79, which was decided on handshake

③ acknowledges it recieved echo
↳ because it recieved segment 79 which was one byte, it expects 80, so that's its ACK #
↳ because it sent segment 42, which was one byte long, it's new segment # is 43
↳ even though it contains no data

RECEIVE WINDOW → used for flow control
→ indicates # of bytes willing to accept

HEADER LENGTH → specifies length of TCP header

OPTIONS FIELD → optional & variable length, typically not used

FLAGS → 6 bits, ACK, SYN, FIN, RST, PSH, URG

URGENT DATA POINTER → not really used (not used in practice)
→ identifies location of urgent data

TCP CONNECTION VARIABLES

recieve buffer
send buffer
recieve window
last byte read
congestion window → constraint on rate sender can send traffic into network

amount of unacknowledged data may not exceed minimum of cwnd or rwnd

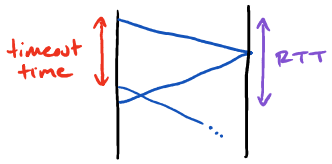
$$LastByteSent - LastByteAcked \leq \min\{cwnd, rwnd\}$$

$$\text{sender rate} \approx \frac{cwnd}{RTT}$$

can control rate by adjusting cwnd

RTT ESTIMATIONS & TIMEOUT

↳ timeout must be longer than RTT
otherwise it will send unnecessary retransmissions
↳ if too long, slow reactions to loss
↳ RTT is variable, so timeouts are computed and changed



To estimate RTT:

↳ TCP records RTT for one unacknowledged segment at a time called SampleRTT

↳ to smooth it out, an exponential

moving average is kept track of and updated like so

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

↑ note: this is an assignment called with every new value of SampleRTT, it is not an equation

$$\alpha = \frac{1}{8} = 0.875$$ ← recommended value

↳ the variability/deviation is also kept track of as DevRTT

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

$$\beta = \frac{1}{4} = 0.25$$ ← recommended

↳ using EstimatedRTT and DevRTT we keep a running value for the timeout interval as such

$$TimeoutInterval = EstimatedRTT + 4 * DevRTT$$

↑ initially set to 1 second ↑ safety margin

TCP'S RELIABLE DATA TRANSFER

keys: error detection
recieve feedback
retransmission

→ TCP only uses one timer even if there are multiple transmitted unacknowledged segments
↳ timers require considerable overhead

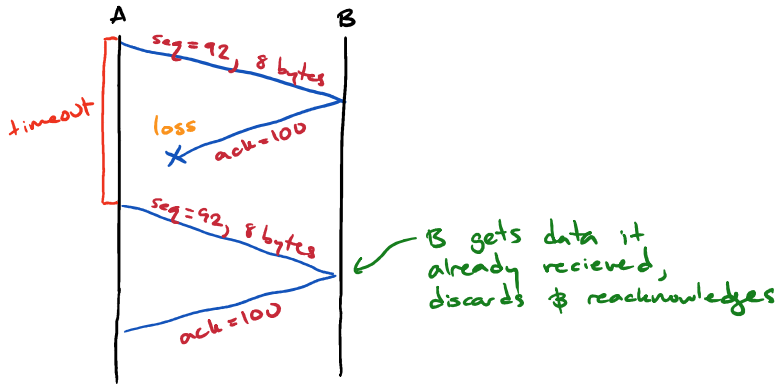
simplified model:
TCP responds only to three major events
Data recieved from application above

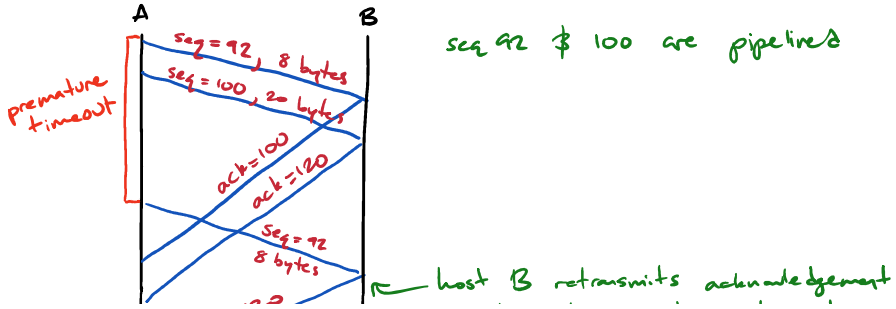
↳ create TCP segment w/ sequence # NextSeqNum
↳ start timer if not already running
↳ pass segment to IP
↳ set NextSeqNum = NextSeqNum + length(data)

Timer timeout
↳ retransmit unacknowledged segment w/ smallest sequence #
↳ start timer
↳ double timeout interval

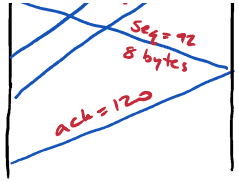
Acknowledgement recieved
↳ if incoming ACK number > sequence # oldest unacknowledged byte
then the ACK is acknowledging a previously unacknowledged segment
else
then the ACK is about something it already acknowledged, so ignore it

RETRANSMISSION SCENARIOS

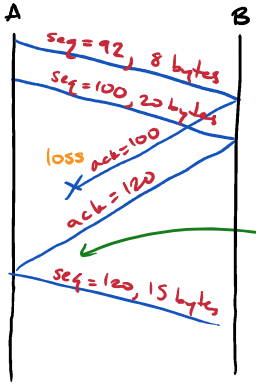




seq 92 & 100 are pipelined



← host B retransmits acknowledgement for largest segment, not just seq + len(data)



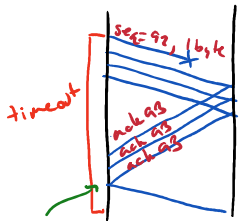
doesn't matter ack=100 was lost b/c from cumulative acknowledgement, A knows 100 was recieved

OTHER TCP BEHAVIOURS

FAST RETRANSMIT

↳ when segment is lost, the long timeout is a delay
↳ the sender can detect packet loss by duplicate acks

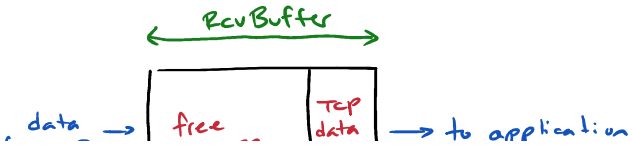
if 3 duplicate ACKs a loss is probable, retransmit before timeout

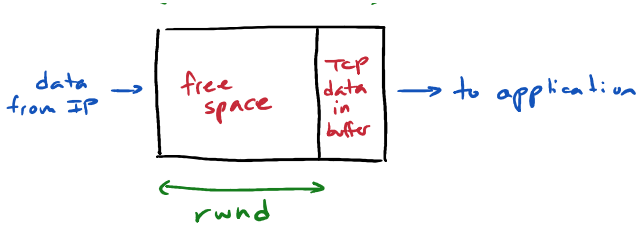


does not wait for timeout

FLOW CONTROL

↳ avoids overwhelming reciever / overflow reciave buffer
↳ in TCP both sides specify reciave window, rwnd
↳ how much free buffer space available at reciever (# bytes willing to accept)





available at reciever
(# bytes willing to accept)

in order to not overflow:

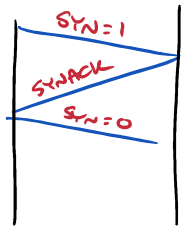
$$LastByteRcvd - LastByteRead \leq RcvBuffer$$

recieve window is amount of spare room

$$rwnd = RcvBuffer - (LastByteRcvd - LastByteRead)$$

CONNECTION MANAGEMENT

↳ three-way handshake to establish connection
↳ agree on connection parameters



CONGESTION CONTROL

↳ end-end congestion control (used by TCP)
↳ TCP recieves no information about congestion state from IP

TCP keeps track of congestion window

$$LastByteSent - LastByteAcked \leq \min\{cwnd, rwnd\}$$

amount of unacknowledged data at sender

$$\text{TCP sender rate} \simeq \frac{cwnd}{\quad}$$

So TCP can control sending rate by modifying cwnd "self-clocking"

data at server

TCP sender rate $\simeq \frac{cwnd}{RTT}$

So TCP can control sending rate by modifying cwnd "self-clocking"

↳ it will increase cwnd until loss event happens, timeout or 3 duplicate ACKs

slow start

cwnd = 1, double cwnd every RTT

continue slow st. until

ssthresh exceeded, then increase only linearly

proceed until loss

if timeout cwnd = 1, ssthresh = $\frac{1}{2}$ cwnd

if 3 dup acks cwnd = $\frac{1}{2}$ cwnd, ssthresh = $\frac{1}{2}$ cwnd