Transport Services and Protocols

↳ Provides logical comm. b/w app processes running on diff hosts

↳ during comm the location of processes isn't necessary → no Physical comm.

↳ so we belive the devices are connected intermediately and have direct comm. path

Sender side

↳ breaks app mssg into segments

↳ assign header to segments

↳ passes to network layer

receiver side

↳ reassembles segments into mssgs

↳ passes to app layer

always implemented on host

Application Layer

↳ web browser requests to send a mssg

Transport LAYER

↳ breaks mssg into segments

↳ each segment is given a header

Network LAYER

↳ puts network layer header on segments

↳ the segments are routed through network

Destination Side TRANSPORT LAYER

↳ segments are reassembled into mssgs

↳ then passed to application layer

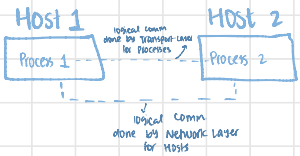
Transport Layer

↳ logical comm. b/w processes

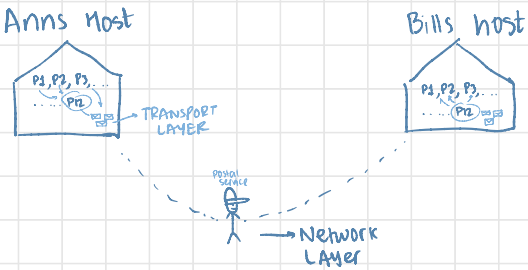
VS

Network Layer

↳ logical comm. b/w hosts



> **household analogy:**
>
> *12 kids in Ann's house sending letters to 12 kids in Bill's house:*
>
> - hosts = houses
> - processes = kids
> - app messages = letters in envelopes
> - transport protocol = Ann and Bill who demux to in-house siblings
> - network-layer protocol = postal service



Transport Layer Protocols → Limitations

↳ TCP

↳ UDP

↳ delay guarantees

↳ bandwidth guaranteess

Transmission Control Protocol (TCP)

↳ reliable services

↳ in order delivery → data received in order

↳ congestion control → limits sender to a specific rate to reduce traffic and blocks

↳ flow control → data is sent at a rate which receiver can process

↳ connection setup → handshaking

↳ connection oriented transport

PROS

↳ faster as less delay → due to fewer packet loss

in transmission: Send next ACK

in retransmission: send cumulative ACK

User Datagram Protocol (UDP)

Connectionless Protocol

↳ unreliable → no guarantee whether you receive data or not

↳ unordered delivery

↳ best effort protocol

↳ connection less transport

PROS

↳ fastr

↳ more efficient

in retransmission: Send Prev ACK

in transmission: send current ACK

A

Multiplexing at sender

↳ handle data from multiple sockets

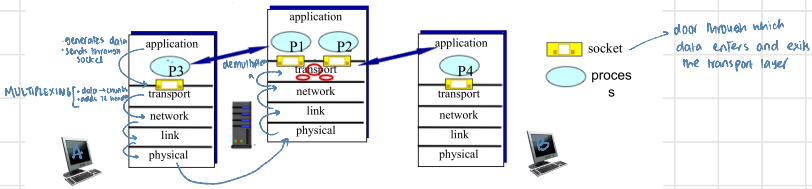
↳ add transport layer header

↳ send to network layer

B

Demultiplexing → distribute among multiple processes

↳ use header to deliver received segments to correct socket



↳ each host can have multiple sockets and run multiple processes

How demultiplexing works

↳ host receives IP datagram

↳ each datagram has

↳ source IP address

↳ destination IP address

} to identify host

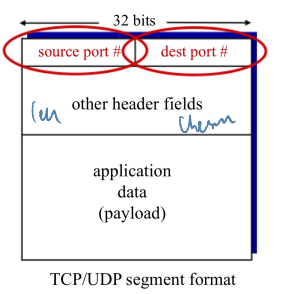
↳ each datagram carries

↳ 1 transport layer segment

↳ each segment has

↳ source port no.

↳ destination port no.



TCP/UDP segment format

↳ host uses

IP address + port no. → identification of each process

to direct segment to correct socket

Connectionless demultiplexing

→ how UDP does demultiplexing

- *recall:* created socket has host-local port #:

```
DatagramSocket mySocket1
= new DatagramSocket(12534);
```

(create datagram; port)

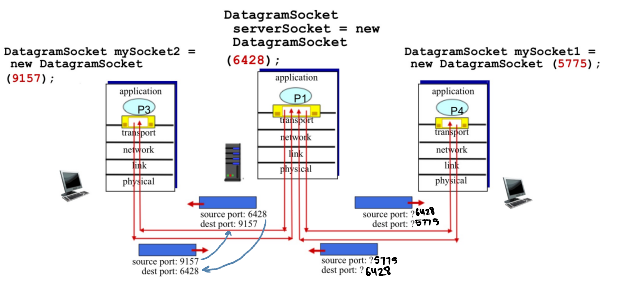
- *recall:* when creating datagram to send into UDP socket, must specify
  - destination IP address
  - destination port #

→ YOU NEED For identification

- when host receives UDP segment:
  - checks destination port # in segment
  - directs UDP segment to socket with that port #

→ IP datagrams with *same dest. port #*, but different source IP addresses and/or source port numbers will be directed to *same socket* at dest

↳ same destination IP
↳ diff source IP
↳ same socket



know source by looking at TCP segment in header containing sourceIP, source port

Connection oriented Demultiplexing

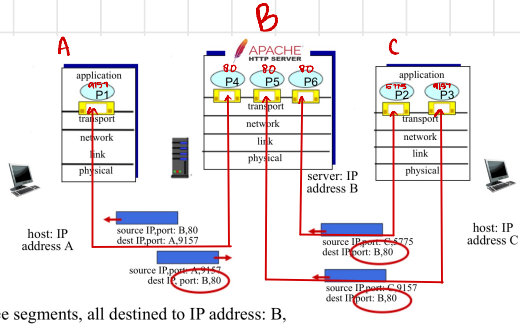
→ HOW TCP does demultiplexing

- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number

YOU NEED for identification

- demux: receiver uses all four values to direct segment to appropriate socket
- server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- web servers have different sockets for each connecting client
  - non-persistent HTTP will have different socket for each request

↳ same destination IP
↳ diff source IP
↳ diff socket, same port



Three segments, all destined to IP address: B, dest port: 80 are demultiplexed to *different* sockets

Summary

- Multiplexing, demultiplexing: based on segment, datagram header field values
- **UDP:** demultiplexing using destination port number (only)
- **TCP:** demultiplexing using 4-tuple: source and destination IP addresses, and port numbers
- Multiplexing/demultiplexing happen at *all* layers

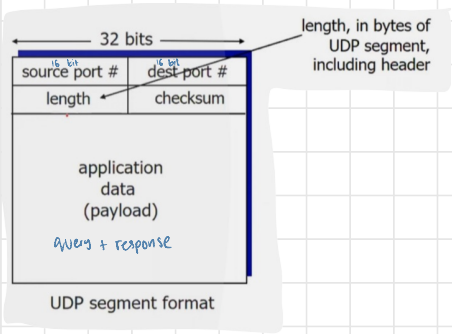
Connectionless Transport: UDP

User Datagram Protocol (UDP)

- no frills protocol → segments may be lost
- bare bones (no fancy extra features)
- unordered delivery
- best effort service → send and hope for the best
- not reliable → no guarantee whether you receive data or not

UDP use:

- streaming multimedia apps (online Gaming, Streaming Video, Video Conferencing) (loss tolerant, rate sensitive)
- DNS
- SNMP



UDP segment format

UDP is a connectionless protocol

- no handshaking b/w UDP sender and receiver
- each UDP segment handled independently

How to make UDP reliable

- add reliability at application layer
- add congestion control at application layer

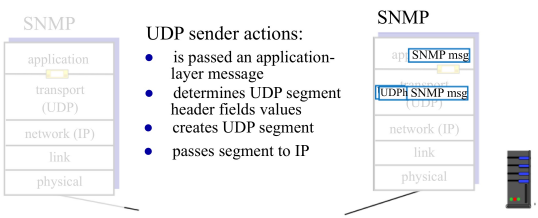
↓

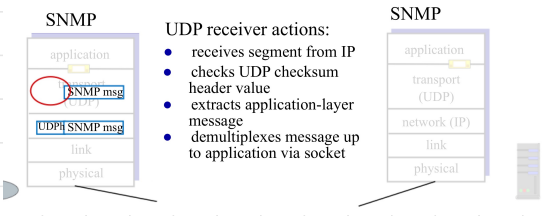
CDN: more work for application developer → add reliability, debug error

PROS

- no delay, faster → as no connection establishment which adds RTT delay
- no congestion control → can be as fast as desired
- simple: no connection state at sender/receiver
- small header size → 20 bytes → good for sending huge amount of data → TCP header is 20
- helps with reliability → checksum
- can function when network service is compromised

UDP: Transport Layer Actions





UDP checksum

↳ a method to detect errors in transmitted segment

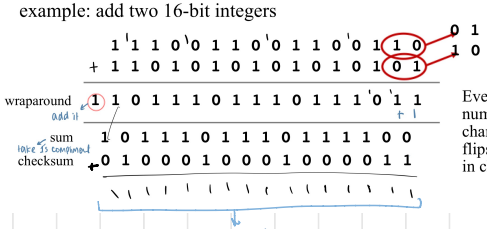
sender:

- treat contents of UDP segment (including UDP header fields and IP addresses) as sequence of 16-bit integers
- checksum: addition (one's complement sum) of segment content
- checksum value put into UDP checksum field

receiver:

- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - Not equal - error detected
  - Equal - no error detected. *But maybe errors nonetheless?* More later ….

example: add two 16-bit integers

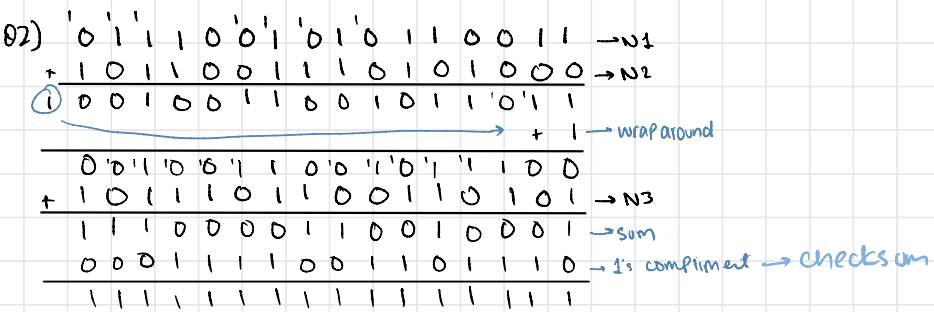


Even though numbers have changed (bit flips), *no* change in checksum!

if all 1's : no change

if any "0" : change → error

Q2)



hence no change

Question 2: **[10 Marks]**

Checksum is an error-detecting code used in many Internet standard protocols, including IP, TCP, and UDP. You have to generate Checksum for the following data blocks, which are transmitted using UDP.

N1	N2	N3
0111001010110011	1011001110101000	1011101100110101

Q2)

```
  0 1 1 1 0 0 1 0 1 0 1 1 0 0 1 1   → N1
+ 1 0 1 1 0 0 1 1 1 0 1 0 1 0 0 0   → N2
(1) 0 0 1 0 0 1 1 0 0 1 0 1 1 0 1 1
                                + 1   → wrap around
  0 0 1 0 0 1 1 0 0 1 0 1 1 0 0 0
+ 1 0 1 1 1 0 1 1 0 0 1 1 0 1 0 1   → N3
  1 1 1 0 0 0 0 1 1 0 0 1 0 0 0 1   → sum
  0 0 0 1 1 1 1 0 0 1 1 0 1 1 1 0   → 1's compliment
```

CHECKSUM

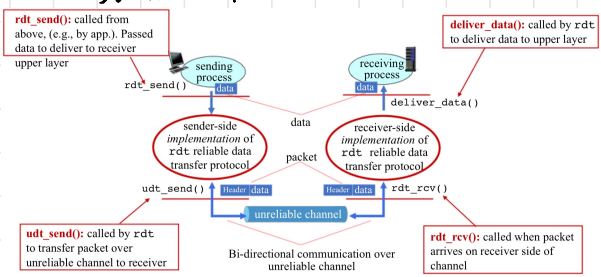
Principles of reliable data transfer protocols (rdt)

↳ important in → application, transport, link LAYER

a) provided service

reliable service *abstraction*

b) service implementation



AL generate data → data

called by application layer ← call rdt_send() → sends data to TL

divide in chunks, add header → packet

call udt_send() → transfers packet from unreliable channel

called by rdt ← call rdt_receive() → receives packet

call deliver_data() → delivers data to upper layer

Reliable Data transfer (rdt)

↳ we'll:

- ↳ incrementally develop sender, receiver side rdt (reliable data transfer protocol)
- ↳ unidirectional (→ OR ←) data transfer
- ↳ bidirectional (↔) control info transfer
- ↳ use FSM (finite state machine) to specify sender, receiver

data info: video

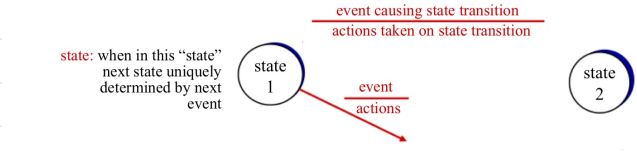
control info: OK / NOT OK msgs on video being received

rdt Protocols

Timing seq diagrams

1. rdt 1.0
2. rdt 2.0 — error
3. rdt 2.1 — error
4. rdt 2.2 — error + lost
5. rdt 3.0

2–5: stop and wait protocols



Transp

rdt 1.0 : reliable transfer over reliable channel

↳ reliable transfer

↳ reliable channel

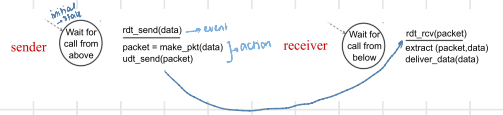
↳ no bit errors

↳ no loss of packets

↳ seperate FSMs for sender, receiver:

↳ sends data into underlying channel

↳ receives data from underlying channel



rdt 2.0: channel with bit errors

↳ underlying channel may flip bits in packet 101 → 001

↳ checksum to detect errors

PROS

↳ error detection

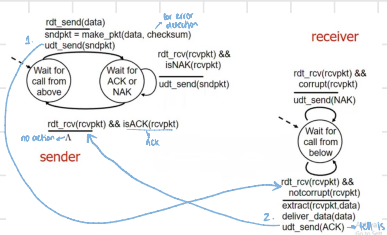
↳ feedback : control mssg from r to s (ACK/NAK)

↳ HOW TO recover from errors

1. Acknowledgements (ACKs) : receiver explicitly tells sender that packet received OK → 0
2. Negative Acknowledgements (NAKs): receiver explicitly tells sender that packet had ERRORs → 1
   ↳ sender retransmits packet on receipt of NAK

↳ FSM specifications

operation with no errors



1 → no event

Λ → no action

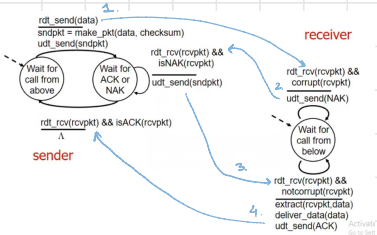
Note: "state" of receiver (did the receiver get my message correctly?) isn't known to sender unless somehow communicated from receiver to sender

- that's why we need a protocol!

Sending: 2 states

receiver: 1 state

corrupted packet senario



Stop and wait Protocol

↳ sender sends one packet then waits for receiver response

↳ has low response time → as waiting

FLAW

↳ what happens if ACK/NAK corrupted → ACK/NAK flipped

↳ sender doesn't know what happened to receiver

↳ can't just transmit a duplicate

↓ solution 1

Handling duplicates

↳ sender retransmits current pkt if ACK/NAK corrupted

↳ sender adds sequence number to each pkt → Packet

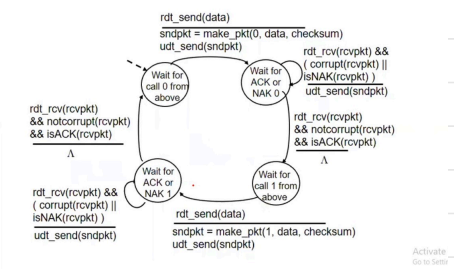
↳ If receiver receives pkt of same seq no.

↳ it will discard one and deliver the other

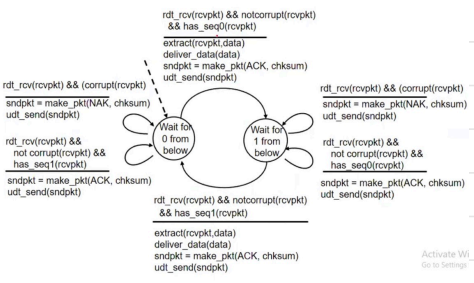
↓

rdt 2.1:

rdt2.1: sender, handles garbled ACK/NAKs



rdt2.1: receiver, handles garbled ACK/NAKs



↳ discussion

Sender

↳ seq no. added to pkt → 2 seq no; (0,1) will suffice

↳ check if ACK/NAK corrupted

↳ twice as many states

↳ as state must remember whether 'expected' pkt should have seq no. of 0 or 1

receiver

↳ checks if received packet is duplicate

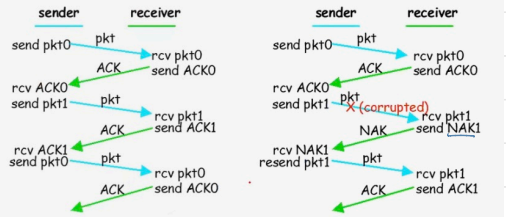
↳ state indicates whether 0 or 1 is expected pkt seq no.

↳ receiver can't know if its last ACK/NAK received OK at sender

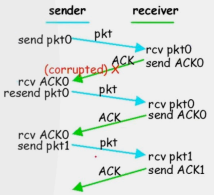
Sending: 4 states

receiver: 2 state

rdt 2.1 in action



rdt 2.1 in action (cont)



if Pkt corrupted → send R NAK1

if Ack corrupted → S resend pkt 1

solution 2

rdt 2.2: a NAK free protocol

↳same functionality as rdt 2.1, using ACKs only

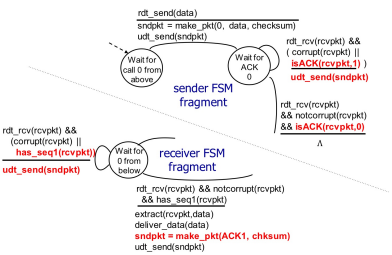
↳instead of NAK, receiver sends ACK for last Pkt received OK → sender, will get duplicate ACK and will know he here Pkt has been received and retransmit next Pkt

↳receiver must explicitly include seq no. of Pkt being ACKed

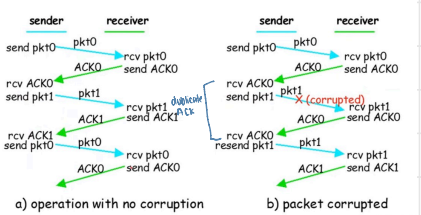
↳duplicate ACK at sender results in same actions as NAK:

↳retransmits current Pkt

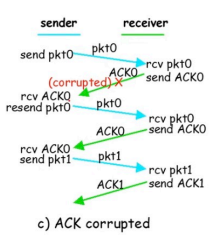
TCP uses this approach to be NAK free



rdt 2.2 in action



rdt 2.2 in action (cont)



if Pkt corrupted → send R last ACK

if Ack corrupted → S resend pkt 1

rdt3.0 : channels with errors and loss

↳ new channel assumption

↳ underlying channel can also lose packets (data, ACKs)

↳ checksum, seq no., ACKs, retransmission will be of help but not quite enough

↳ How do humans handle lost sender to rec words in conversation

↓ approach

use timer

↳ sender waits reasonable amount of time for ACK

↳ retransmits if no ACK received in this time

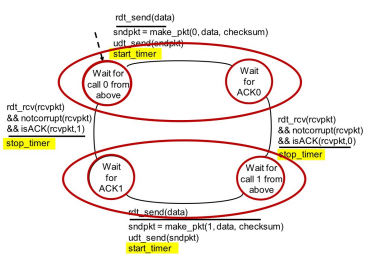
↳ if pkt (or ACK) just delayed (not lost):

↳ retransmission will be duplicate → but seq no. already handles this 2.1, 2.2

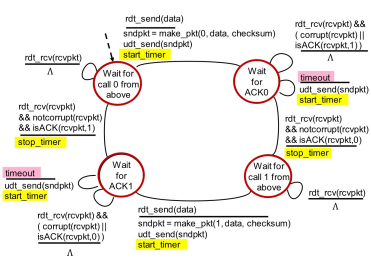
↳ receiver must specify seq no of pkt being ACKed

↳ use countdown timer to interrupt after "reasonable" amount of time

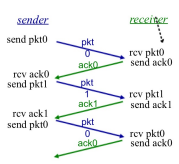
t3.0 sender



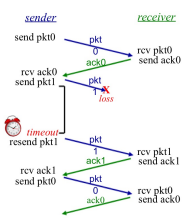
s.0 sender



rdt3.0 in action

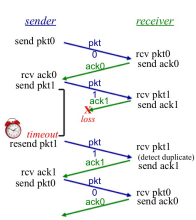


(a) no loss

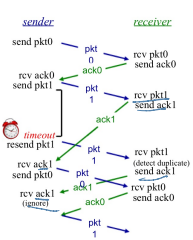


(b) packet loss

rdt3.0 in action



(c) ACK loss



(d) premature timeout/ delayed ACK

if Pkt corrupted → sender (R) plow ACK

if Ack corrupted → sender (S) resend pkt 1

if Pkt loss → after timeout (S) resend Pkt 1

if ACK loss → after timeout (S) resend Pkt 1

if delayed ACK → receiver (R) delay duplicate send ack; sender (S) ignore duplicate

↳ Performance of rdt3.0

↳ correct but slow performance → as stop and wait protocol

↳ has capacity to transfer more data but protocol limits it down

conversions

1 byte = 8 bit

1 Gb = 10^9

1 micro s = 10^{-6}

1 milli s = 10^{-3}

a) link: 1 Gbps (10^9), prop delay: 15ms, packet: 8000 bit

$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ micro s} = 0.008 \text{ ms}$$

D_{trans}: time to transmit packet into channel; 8 micro s = 10^{-6}; 0.008 ms = 10^{-3}

$$d_{trans} = \frac{L}{R}$$ L: packet length bits; R: link transmission rate bps

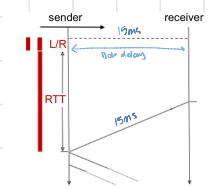
$$d_{prop} = \frac{d}{s}$$ d: length of physical link; s: propagation speed 2×10^8 m/s

$$RTT = 2 \times \text{prop delay} = 2(15) = 30 \text{ msec}$$

RTT: Round Trip Time
- time from when sender sent pkt
- receive received receiver sent pkt

↳ U sender: utilization - fraction of time sender busy sending

$$U_{sender} = \frac{L/R}{RTT + L/R} = \frac{0.008}{30 + 0.008} = \frac{0.008}{30.008} = 0.00027 \text{ ms}$$



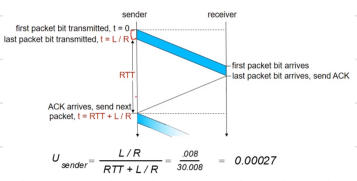
- 1KB pkt every 30ms

$$\text{throughput} = \frac{1 \text{ KB}}{30 \text{ ms}} = 33 \text{ KB/s}$$

$$U = \frac{N}{RTT + \text{Packet size}}$$ N: window size; RTT: 2 × Prop Delay

$$\frac{10{,}000}{5 \times 10^9} = 50$$

↳ stop and wait operation → low performance (not good approach)



Part c) A point-to-point satellite transmission link of 5 Gbps and 30ms propagation delay connecting two computers uses a stop-and-wait ARQ strategy. All packets consist of 10,000 bits and average round trip time RTT is 50ms.

i) Calculate the link efficiency (utilization) when only one packet is sent.

ii) Calculate the new link efficiency (utilization) when pipelining is applied, now sender send three packets back-to-back.

Part c)

i) Answer:

$$\text{Usender} = \frac{L/R}{RTT+L/R} = \frac{10000/5\text{Gbps}}{50\text{ms}+10000/5\text{Gbps}} = 0.0000399$$

ii) Answer:

$$\text{Usender} = \frac{3L/R}{RTT+L/R} = \frac{3 \times 10000/5\text{Gbps}}{50\text{ms}+10000/5\text{Gbps}} = 0.0001199$$

b) How the channel utilization may be increased?

Solution:-

The problem is the low utilization of the system which could be solved by pipelining protocol

Question 4: CLO-01 **[5x2=10 points]**

Write the difference, in maximum 30 words each (answer will not be marked after 30 words)

a) (In Forwarding table computing/learning) process at switches versus routers

Solution:-

routers: compute tables using routing algorithms, IP addresses
switches: learn forwarding table using flooding, learning, MAC addresses

b) intra-AS routing versus intra-domain routing

Solution:-

No difference.

c) Time-to-Live field of IPv4 Header versus Hop-Limit field of IPv6 Header

Solution:-

No difference, same function, just different naming convention.

d) Flow control mechanism in link layer versus flow control in transport layer

Solution:-

On the link layer, pacing between adjacent sending and receiving nodes on a link is handled, while on transport layer, overflow at the final destination node is avoided.

e) Connectionless link versus Unreliable link

Solution:-

Connection-less: no handshaking between sending and receiving NICs
Unreliable: receiving NIC doesn't send ACKs or NAKs to sending NIC

solves low performance

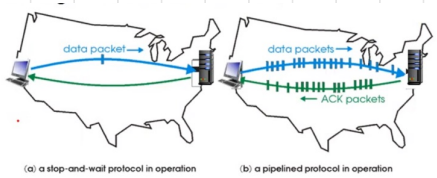
good approach

↳ Pipelined Protocols Operation

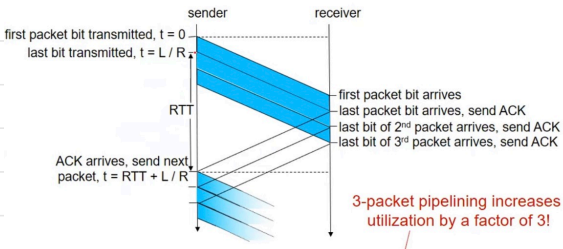
↳ Pipelining: sender allows multiple, 'in-flight', yet to be acknowledged packets

↳ range of seq no. must be increased → k bit

↳ buffering at sender / receiver (addition)



Pipelining: increased utilization



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

Generic forms of Pipeline Protocols

1. GO-Back-N
2. Selective Repeat

Pipelined protocols: overview

Go-back-N:

- sender can have up to N unacked packets in pipeline
- receiver only sends *cumulative ack*
  - doesn't ack packet if there's a gap
- sender has timer for oldest unacked packet
  - when timer expires, retransmit *all* unacked packets

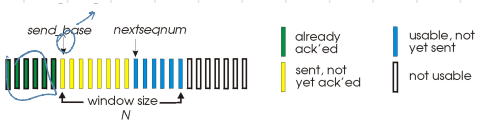
Selective Repeat:

- sender can have up to N unack'ed packets in pipeline
- rcvr sends *individual ack* for each packet
- sender maintains timer for each unacked packet
  - when timer expires, retransmit only that unacked packet

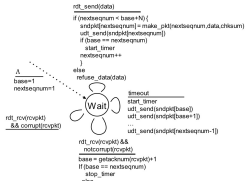
1. Go-Back-N: sender

retransmitt PKt from first out of order

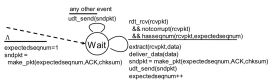
↳ Sender can have upto N unacked → sender window
PKts in pipeline
↳ k-bit seq no in Pkt header



Go-Back-N: sender extended FSM



Go-Back-N: receiver extended FSM



Cumulative ACK:

↳ ACK(n): ACKs all packets up to, including seq no n on receiving
↳ ACK(n): move window forward to begin at n + 1
↳ timer for oldest in-flight packet
↳ timeout(n): retransmit packet n and all higher seq no. packets in window

↳ will only receive ack of 0,1 even though 3 was also sent
↳ retransmit 2,3

Lec18

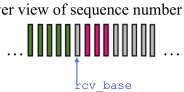
Go Back-N: receiver (GBN)

↳ ACK only: always send ACK for correctly-received packet so far,
with highest in-order seq no
↳ may generate duplicate ACKs
↳ need only remember expected seqnum → rcv_base
↳ on receipt of out-of-order packet: → 3 received but 2 not so not in order hence discard 3
↳ can discard (don't buffer) or buffer → an implementation decision
↳ re-ACK Pkt with highest in-order seq no

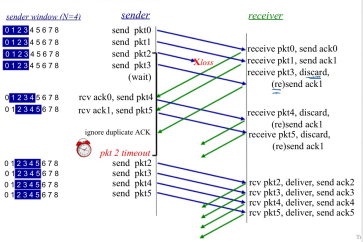
0,1 ✗ 3
send Ack

sends ack of last acked value

Receiver view of sequence number space:



Go-Back-N in action



If Pkt loss → discard, resend last Ack after timeout, send
R
S
ignore duplicate ACK

CON

↳ Cumulative ack
↳ will have to resend from first out of order
↳ retransmission high
↳ more bandwidth used

2. Selective Repeat (SR)

only retransmit pkt that is lost

0, 1, 2, 3, 4, 5 (2 = lost; 3, 4, 5 = buffer)

↳ sender can have upto N unacked → sender window

PKTs in pipeline

↳ receiver individually acknowledge all correctly received packets

↳ buffers packets → as needed, for eventual in-order delivery to upper layer

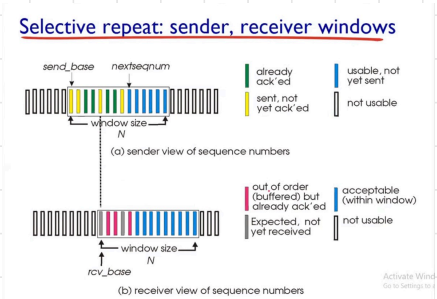
↳ sender times out / retransmits individually for unACKed packets

↳ sender maintains times for each unACKed pkt

↳ sender window

↳ N consecutive seq no.'s

↳ limits seq no.'s of sent, unACKed packets



Selective repeat: sender and receiver

sender

data from above:
- if next available seq # in window, send packet

timeout(*n*):
- resend packet *n*, restart timer

ACK(*n*) in [sendbase,sendbase+N]:
- mark packet *n* as received
- if n smallest unACKed packet, advance window base to next unACKed seq #

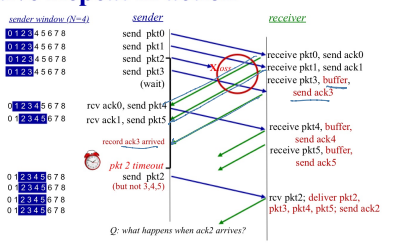
receiver

packet *n* in [rcvbase, rcvbase+N-1]
- send ACK(*n*)
- out-of-order: buffer
- in-order: deliver (also deliver buffered, in-order packets), advance window to next not-yet-received packet

packet *n* in [rcvbase-N,rcvbase-1]
- ACK(*n*)

otherwise:
- ignore

Selective Repeat in action



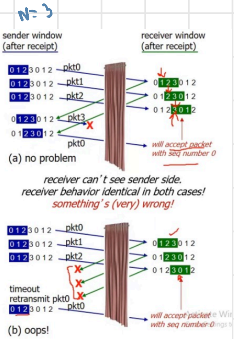
if Pkt loss → buffer, send ack cumulat after time out 5 → send pkt2 → send ack2, deliver pkt 2,3,4,5

Selective repeat: dilemma

example:
- seq #'s: 0, 1, 2, 3
- window size=3
- receiver sees no difference in two scenarios!
- duplicate data accepted as new in (b)

Q: what relationship between seq # size and window size to avoid problem in (b)?

N = 3



CON

↳ old PKT 0 is resent

↳ receiver thinks it is new PKT

↳ it is accepted

SOLUTION

↳ rs b/w seq no and window size

↳ seq no and window size should have a diff of >1

a) window = 3, seq no = 4 → diff of 1

b) window = 3, seq no = 5 → diff >1 → then the problem is avoided

TCP: Overview

connection oriented

- point to point → host to host comm. only
  - one sender, one receiver
- reliable, in-order byte stream:
  - no "msg boundaries"
- full duplex data
  - bidirectional data flow
  - MSS: max segment size
- cumulative ACKs
- pipelining
  - TCP congestion and flow control set window size
- connection-oriented
  - handshaking (exchng of control msgs) initializes sender, receiver state before data exchange
- flow controlled
  - sender will not overwhelm receiver

Sequence no

- first byte in segment data

outgoing segment from sender

source port	dest port
sequence number	
acknowledgement number	
	rwnd
checksum	urg pointer

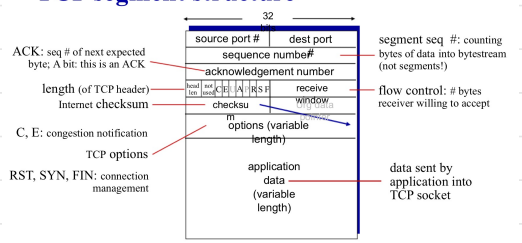
Acknowledgements

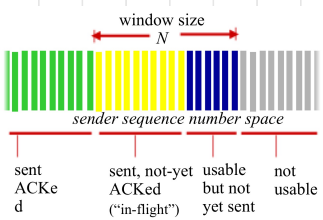
- seq no. of next byte expected from other side
- cumulative ACK

outgoing segment from receiver

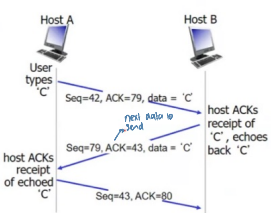
source port	dest port
sequence number	
acknowledgement number	
A	rwnd
checksum	urg pointer

TCP segment structure





TCP seq. numbers, ACKs



simple telnet scenario

↓

user bom techniques

GBN, SR

Q4) UDP header:

Source Port	Destination Port	User datagram length	
0019	D36A	001C	001C

0019: 0000 1001 → 16+8+1 = 25

001C: 0001 1100 → 16+8+4 = 28

IP: 202.28.33.21

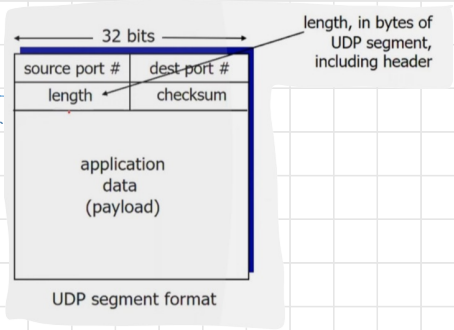
a) Source Port number: 25

b) Socked address of the sender end: 202.28.33. (IP)

c) User datagram length: 28

d) length of data: 28 − 8 = 20 bytes (UDP header length always 8 bytes)

e) Packet direction: server to client → 25 is SMTP



8) TCP header

Source Port	Destination Port	Sequence no.		Ack no.		
0006	0080	0000	0001	0000	0000	3FF0

0080: 0000 0000 1000 0000

3FF0: 0011 1111 1111 0000

Window size	Checksum	Urgent pointer				
0000	0004	FFFF	0100	0000	0010	0000

1) Source Port number 6

2) Destination Port 128

3) Sequence Number 1

4) Ack Number 0

5) TCP header length 3

6) Ack bit 1

7) FIN bit 0

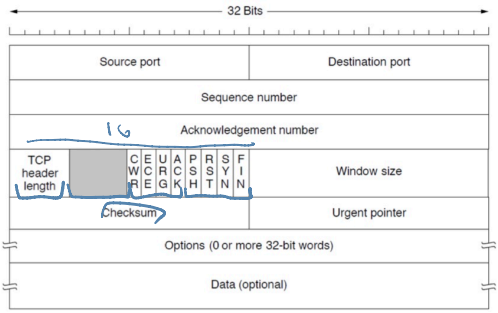
8) SYN bit 0

9) Window size 0

10) checksum 4

X = 4 Y = 6 Z = 8

21k 4638



lec 19

TCP round trip time, timeout

Q) How to set TCP timeout value?

↳ Time out value shouldn't be

↳ too short → Premature timeout → unnecessary retransmissions

↳ too long → slow reaction to segment lost

↳ longer than RTT

↳ but RTT varies

round Trip Time
↳ time from when
- sender sent Pkt
- receiver received
- receiver sent Ack

Q) How to estimate RTT?

SampleRTT

↳ a variable used to measure time

from segment transmission until Ack receipt

↳ ignore retransmission

↳ it will vary → as depends on network congestion

↳ so calculate estimatedRTT using

SampleRTT to make it smoother

avg of multiple SampleRTT

typical value 0.125

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

* large variation in EstimatedRTT → want a larger safety margin

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

deviation of SampleRTT from EstimatedRTT

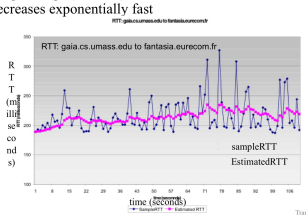
typically 0.25

→ Put new values

safety margin

$$TimeoutInterval = EstimatedRTT + 4*DevRTT$$

- exponential weighted moving average (EWMA)
- influence of past sample decreases exponentially fast
- typical value: □ = 0.125



Q) ERTT = 30ms
SRTT = 15ms
$\alpha = 0.125$

$ERTT = (1-0.125) + 30 + 0.125(15)$

TCP reliable data transfer

- TCP creates rdt service on top of IP's unreliable service
  - pipelined segments
  - cumulative acks
  - single retransmission timer
- retransmissions triggered by:
  - timeout events
  - duplicate acks

let's initially consider simplified TCP sender:
- ignore duplicate acks
- ignore flow control, congestion control

TCP sender

↳ event: data received from application

↳ create segment with seq no.

↳ start timer if not already running

↳ assume timer as oldest ACKed segment

↳ expiration interval: TimeOutInterval

↳ event: timeout

↳ retransmit segment that caused timeout

↳ restart timer

↳ event: ACK received

↳ if ACK acknowledges previously unACKed segment

↳ update what is known to be ACKed

↳ start timer if there are still unACKed segments

TCP Receiver: ACK generation [RFC 5681]

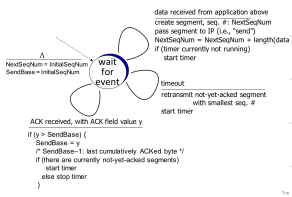
Event at receiver	TCP receiver action

↳ when retransmission

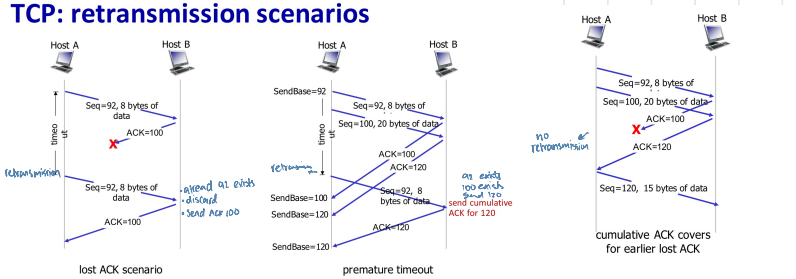
↳ Timeout is doubled

send ACK of next value

TCP sender (simplified)



TCP: retransmission scenarios



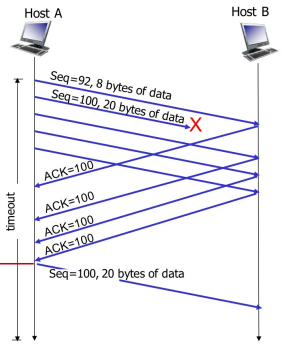
Transport Layer: 3

TCP fast retransmit

↳ if sender receives 3 additional ACKs, for same data

↳ resend unACKed segment with smallest seq no.

↳ its most likely that unACKed segment got lost, so don't wait for timeout



Receipt of three duplicate ACKs indicates 3 segments received after a missing segment – lost segment is likely. So retransmit!

PROS

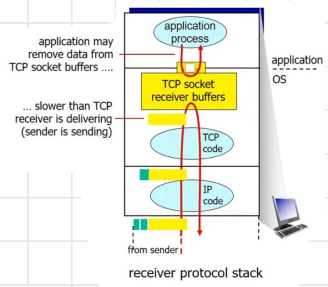
↳ saves time, as time out period relatively long

↳ Pkt not received

Sent 3 additional duplicate ACK

↳ than it retransmits before timeout

TCP flow control

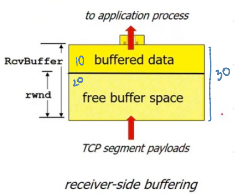


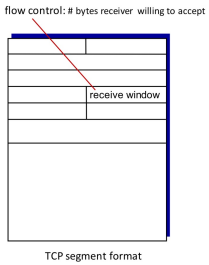
↳ receiver controls sender, so sender won't overflow receivers buffer by transmitting too much, too fast

↳ TCP receiver 'advises' free buffer space in rwnd field in TCP header
  ↳ RcvBuffer size set via socket options → default 4096 bytes; → auto adjusted by many OS

↳ sender limits amount of unACKed data to received rwnd → last Byte sent - last Byte ACK <= receiving window

↳ guarantees receive buffer will not overflow





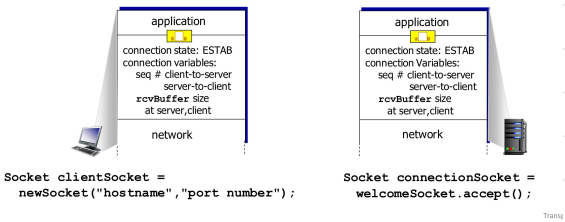
↳ what happens if network layer delivers data faster than application layer removes data from socket buffers

TCP connection management

↳ before exchanging data sender/receiver handshake

↳ agree to establish connection

↳ agree on connection parameters → e.g. starting seq no.



Agreeing to establish connection

↳ 2-way handshake

↳ 3-way handshake

2-way handshake

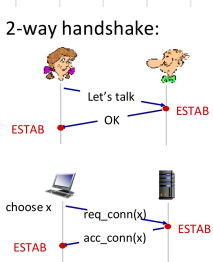
CONS

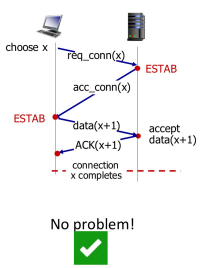
↳ variable delays

↳ retransmitted messages → e.g. req_conn(x) due to msg loss

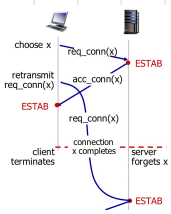
↳ message reordering

↳ can't see other side

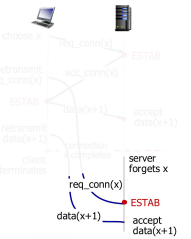




No problem!



Problem: half open connection! (no client)

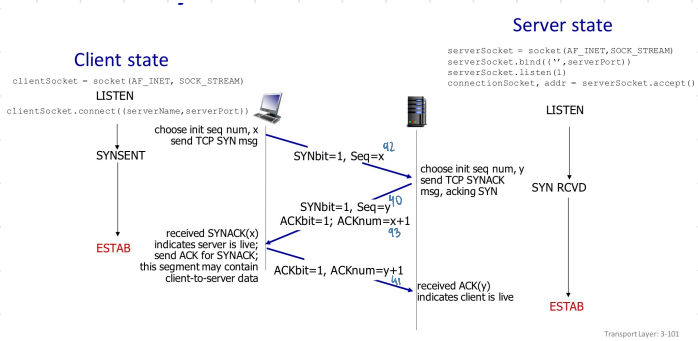


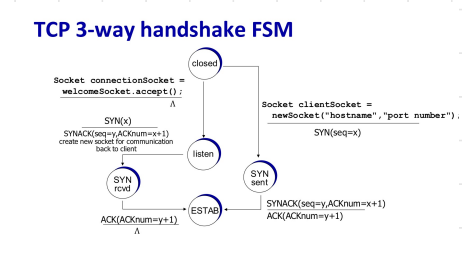
Problem: dup data accepted!

3-way handshake

↳ chooses ISN → client initial seq no

↳ sends TCP SYN msg → to open connection





Closing TCP connection

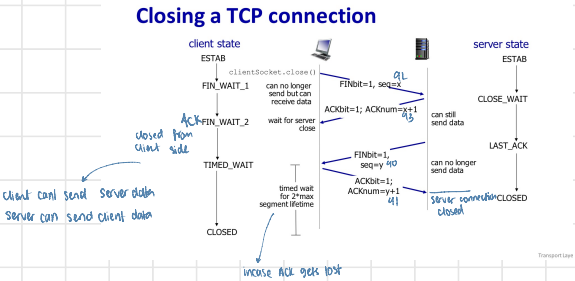
↳ client, server each close their side of connection

↳ send TCP segment with FIN bit = 1 → to close connection

↳ respond to received FIN with ACK

↳ on receiving FIN, ACK can be combined with own FIN

↳ simultaneous FIN exchanges can be handled



Question 4: Because of the connection-oriented nature of TCP, a connection setup phase is required at the beginning of each session, as well as a connection tear-down phase at the end of the session. Enumerate the events below in the order they occur as host A opens a TCP connection to host B, transmits data and then closes the connection. Write a 1 next to the event that occurs first and continue like that until all occurring events are enumerated (the first event has been enumerated for you). You may assume that no segments are lost. Also indicate at which host the event happens. Please note that there might be events listed below that are not a part of the above data transfer and hence should not be enumerated. **[10 points]**

Event	Host	Order
Send an ACK segment	A	8
Do the rest of the data exchange	A, B	4
Close the connection	A	11
Send an ACK segment	B	6
Send a FIN segment	A	5
Send a SYN segment	A	1
Send a FIN segment	B	7
Send a RST segment		
Send a SYN-ACK segment	B	2
Enter the TIME-WAIT state	A	9
Send an ACK+DATA segment	A	3
Close the connection	B	10

Question 5: The Transmission Control Protocol uses a method called congestion control to regulate the traffic

Principles of congestion control

congestion

↳ too many senders,
sending too much data
too fast for network to handle

Manifestations
↳ long delays → queueing in router buffers
↳ packet loss → buffer overflow at routers

flow control

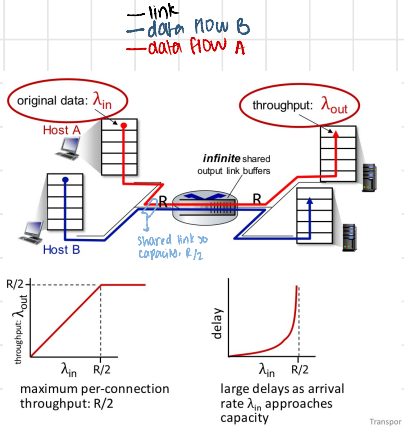
↳ 1 sender, too fast for 1 receiver

Causes/costs of congestion

Scenario 1: simple scenario

↳ 2 senders
↳ 2 receivers
↳ 1 router, infinite buffers → no overflow
↳ input, output link capacity: R
↳ 2 flows
↳ no retransmissions needed → all data is being sent

Q) what happens as arrival rate λ_{in} approaches R/2 ?



maximum per-connection throughput: R/2

large delays as arrival rate λ_{in} approaches capacity

Transpor

Scenario 2

↳ 1 router, finite buffers → can overflow

↳ sender retransmits lost/timed-out packet → pkts may be lost

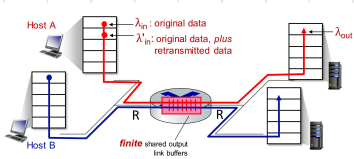
↳ application layer input = application layer output

$$\lambda_{in} = \lambda_{out}$$

↳ transport layer input includes retransmissions

$$\lambda'_{in} \geq \lambda_{in}$$

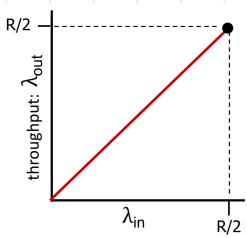
λ'_{in}: og data + retransmitted data; λ_{in}: OG data



↳ 1 hop

↳ Idealization: Perfect knowledge

↳ sender only sends when router buffers avl

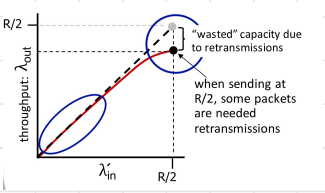


↳ Idealization: Perfect knowledge some

↳ packets can be lost (dropped at router) due to full buffers

↳ sender knows when packet has been dropped

↳ only resends if packet known to be lost (timeout)



↳ realistic scenario: Un-needed duplicates

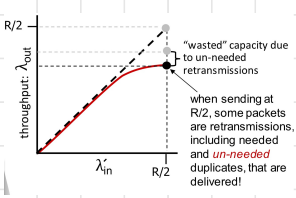
↳ packets can be lost/dropped at route due to full buffers

↳ this req retransmission

↳ but senders time can time out prematurely

↳ sending 2 copies, both of which are delivered

- when timeout
- Send another copy
- rcv gets 2 copies



↳ costs of congestion

↳ more retransmission for given receiver throughput → more retransmissions

↳ un-needed retransmission → link carries multiple copies of a packet

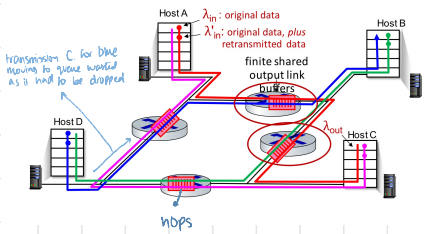
↳ decreasing throughput

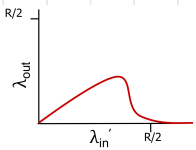
Scenario 3

↳ 4 senders
↳ multi hop paths
↳ timeout / retransmit

Q) what happens if λ_{in} and λ'_{in} increase?
↳ as red λ_{in} increases, λ'_{in} increases → red capacity ↑ ↳ red takes more space in $R/2$
↳ all arriving blue pkts at upper queue are dropped → blue ↓
↳ blue throughput → 0

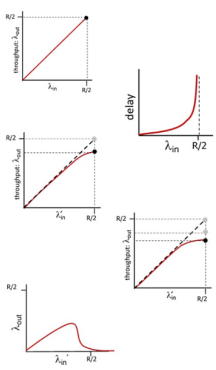
↳ another cost of congestion
↳ when pkt dropped
↳ any upstream transmission capacity and buffering used for that packet was wasted





Causes/costs of congestion: insights

- throughput can never exceed capacity
- delay increases as capacity approached
- loss/retransmission decreases effective throughput
- un-needed duplicates further decreases effective throughput
- upstream transmission capacity / buffering wasted for packets lost downstream



Transport Layer: 3-11

Throughput: rate at which bits are being sent from sender to receiver

Question # 3: **[CL0-3]** **[7.5 + 7.5 + 5 = 20 Points]**

Part a) Consider there is only one router, with finite buffer capacity. The input and output link capacity of router is R. Two flows are attached to the router, each flow sharing half of the router capacity and each flow have:

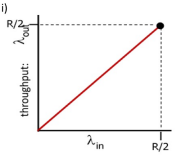
i) The perfect knowledge of the router buffer capacity.
ii) Have partial knowledge, and knows only if the packet is dropped at the router (retransmission needed for lost packet).
iii) Have no knowledge about router buffer, nor about any dropped packet (guess only by acknowledgement packets).

Plot the graphs for each of the above case for only one flow, showing transport-layer input on x-axis and maximum achievable receiver throughput at y-axis.

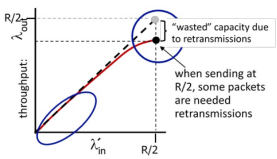
Question # 3: **[CL0-2]** **[7.5 + 7.5 + 5 = 20 Points]**

Part a) Answer:

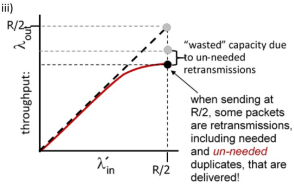
i)



ii)



iii)



Approaches towards congestion control

End-end congestion control

↳ no explicit feedback from network

↳ congestion inferred from observed loss, delays

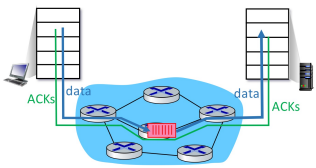
↳ approach taken by TCP

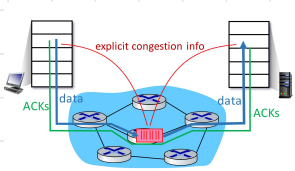
Network Assisted congestion control

↳ routers provide direct feedback to end systems (sending/receiving hosts) with flow passing through the congested router

↳ may indicate congestion by

1. single bit indicating congestion
   ↳ TCP ECN, ATM, DECbit protocols
2. explicitly set sending rate for sender





TCP congestion control: AIMD

Approach:

↳ senders can increase sending rate (cwnd) (window size) until packet loss (congestion) occurs,

↳ then decrease sending rate on loss event

Active increase

↳ increase sending rate by 1 MMS every RTT, until loss detected

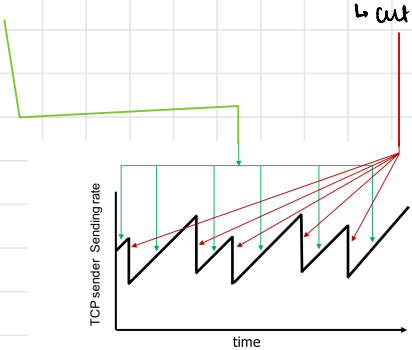
Multiplicative decrease

(cut sending rate at each loss event)

↳ sending rate (cwnd → congestion window) is

↳ cut in half on loss detected by triple duplicate ACK (TCP Reno)

↳ cut to 1 MSS (max segment size) when loss detected by timeout (TCP Tahoe)



Additive Increase Multiplicative Decrease

AIMD

↳ a distributed, asynchronous algorithm

↳ that

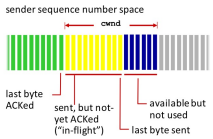
↳ Optimises congested flow rates network wide

↳ has desirable stability properties

↳ by Probing for bandwidth

TCP Congestion Control : Details

TCP congestion control: details



TCP sending behavior:

- *roughly:* send `cwnd` bytes, wait RTT for ACKS, then send more bytes

$$\text{TCP rate} \approx \frac{\text{cwnd}}{\text{RTT}} \text{ bytes/sec}$$

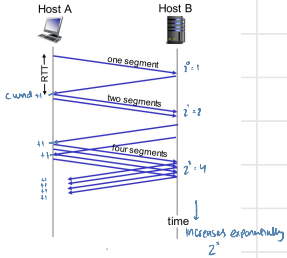
- TCP sender limits transmission: `LastByteSent- LastByteAcked ≤ cwnd`
- `cwnd` is dynamically adjusted in response to observed network congestion (implementing TCP congestion control)

ssthresh = $\frac{1}{2}$ cwnd before loss event

TCP Slow Start

TCP slow start

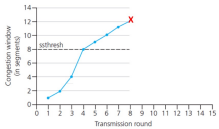
- when connection begins, increase rate exponentially until first loss event:
  - initially **cwnd** = 1 MSS
  - double **cwnd** every RTT
  - done by incrementing **cwnd** for every ACK received
- *summary:* initial rate is slow, but ramps up exponentially fast



TCP: from slow start to congestion avoidance

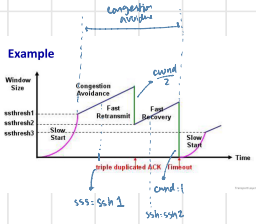
Q: when should the exponential increase switch to linear?

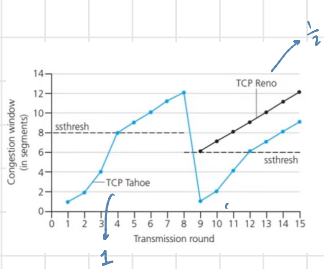
A: when **cwnd** gets to 1/2 of its value before timeout.



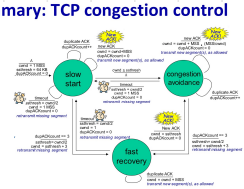
Implementation:

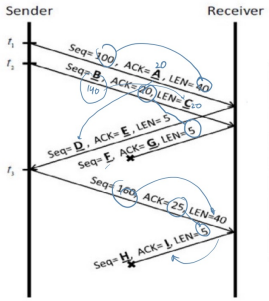
- variable **ssthresh**
- on loss event, **ssthresh** is set to 1/2 of **cwnd** just before loss event





Summary: TCP congestion control





A	20
B	100+40 140
C	20
D	20
E	100+40 140
F	20+5 : 25
G	140+20 160
H	25+5
I	160+40 : 200

Solution:

B = 140; D = 20; E = 140; F = 25; H = 30; I = 200

2. Examine the TCP throughput graph depicted in Figure 1, with the sender's TCP window size represented on the y-axis. Identify the labeled points from A to F and address the following questions: **[0.5*3=1.5 points]**
   a. Identify the region where TCP slow-start is operating. A,D
   b. Identify the region where TCP congestion-avoidance is operating E
   c. Identify the operation at point B and point C.

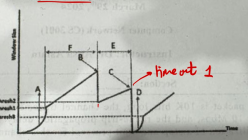
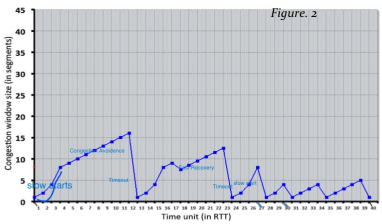


Figure 1 3 duplicate

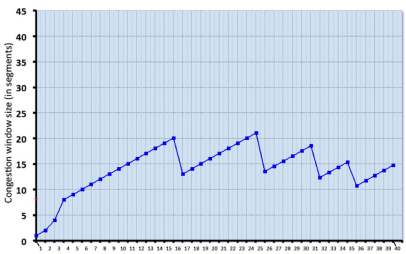
Solution:-

Question 5: Consider the figure. 2, which plots the evolution of TCP's congestion window at the beginning of each time unit (where the unit of time is equal to the RTT). In the abstract model for this problem, TCP sends a "flight" of packets of size *cwnd* at the beginning of each time unit. The result of sending that flight of packets is that either *(i)* all packets are ACKed at the end of the time unit, *(ii)* there is a timeout for the first packet, or *(iii)* there is a triple duplicate ACK for the first packet. In this problem, you are asked to reconstruct the sequence of events (ACKs, losses) that results in the evolution of TCP's *cwnd* shown below.

Figure. 2



(a) Give the time unit(s) (in RTT) at which TCP is in slow start.
Solution:
The times where TCP is in slow start are: 1,2,3,13,14,15,24,25,26,28,29,31,35,40.
(b) Give the time unit(s) (in RTT) at which TCP is in congestion avoidance.
Solution:
The times where TCP is in congestion avoidance are:
4,5,6,7,8,9,10,11,12,16,17,19,20,21,22,23,27,30,32,33,34,36,37,38,39
(c) Give the time unit(s) (in RTT) at which TCP is in fast recovery.
Solution:
The times where TCP is in fast recovery are: 18.
(d) Give the time unit(s) (in RTT) at which packets are lost via timeout.
Solution:
The times where TCP has a loss by timeout are: 12,23,27,30,34,39.
(e) Give the time unit(s) (in RTT) at which packets are lost via *triple ACK*.
Solution:
The times where TCP has a loss by triple duplicate ACK are: 17
(f) Give the time unit(s) (in RTT) at which the value of *ssthresh* changes (hint: if it changes between t=3 and t=4, use t=4 in your answer)
Solution:
The times where the ssthresh changes are: 18,24,28,31,40.



Answer the following questions.
The initial value of *cwnd* is 1 and the initial value of *ssthresh* (Little + sign) is 8.

1. Give the times at which TCP is in slow start. 1,2,3

2. Give the times at which TCP is in congestion avoidance.

4,5,6,7,8,9,10,11,12,13,14,15,16,18,19,20,21,22,23,24,25,27,28,29,30,31,33,34,35,37,38,39,40

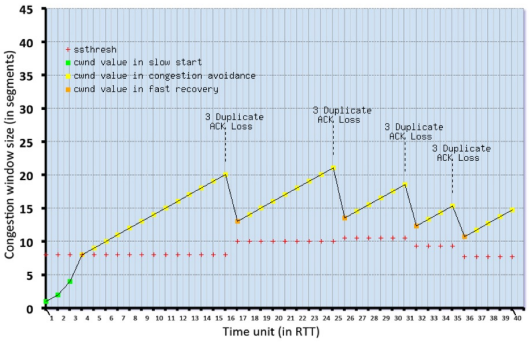
3. Give the times at which TCP is in fast recovery. 17,26,32,36

4. Give the times at which packets are lost via *triple ACK*. 16,25,31,35

5. Give the times at which the value of *ssthresh* changes (if it changes between t=3 and t=4, use t=4 in your answer) 17,26,32,36

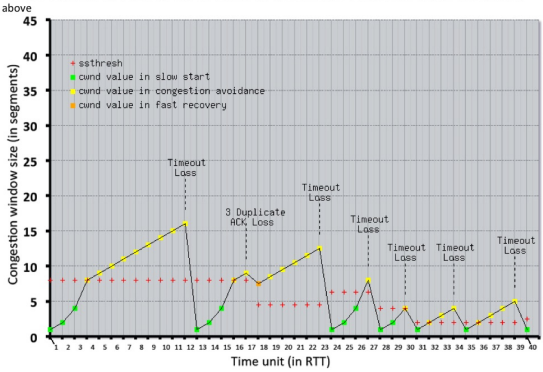
The complete solution is shown in the figure below:

- For intervals of time when TCP is in slow start, the plotted value of *cwnd* is shown as a green square
- For intervals of time when TCP is in congestion avoidance, the plotted value of *cwnd* is shown as a yellow square
- For intervals of time when TCP is in fast reccovery, the plotted value of *cwnd* is shown as an orange square
- The values for *ssthresh* are shown following a change as a red plus sign
- A flight of packets experiencing a loss has the loss type (which determines the next value of *cwnd*) labeled above



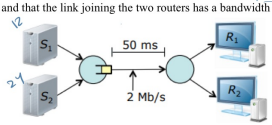
The complete solution is shown in the figure below:

- For intervals of time when TCP is in slow start, the plotted value of cwnd is shown as a green square
- For intervals of time when TCP is in congestion avoidance, the plotted value of cwnd is shown as a yellow square
- For intervals of time when TCP is in fast recovery, the plotted value of cwnd is shown as an orange square
- The values for ssthresh are shown following a change as a red plus sign
- A flight of packets experiencing a loss has the loss type (which determines the next value of cwnd) labeled above



Question 3 [10 Points]

The following diagram shows two TCP senders at left and the corresponding receivers at right. Both senders use TCP Reno. Assume that the MSS is 1 KB, that the one-way propagation delay for both connections is 50 ms and that the link joining the two routers has a bandwidth of 2 Mb/s.



(a) Assume that the link buffer overflows whenever **cwnd1+ cwnd2 ≥ 36 KB** and that at time 0, cwnd1=12 KB and cwnd2=24 KB. Approximately, what are the values of cwnd1 and cwnd2 one RTT later? Assume that all packet losses are detected by a triple duplicate ack.

Solution [6 points]

Cwnd = 36 KB when buffer overflows and congestion is detected. Therefore, ssthreshold would be updated and taken half.

New_ssthreshold1 = 12/2 KB = 6 KB
New_cwnd1 = New_ssthreshold1 + 3MSS
New_cwnd1 = 9 KB

New_ssthreshold2 = 24/2 KB = 12 KB
New_cwnd2 = New_ssthreshold2 + 3MSS
New_cwnd2 = 15 KB

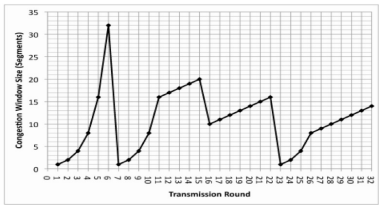


Figure 2

a) **Slow Start:** identify the intervals of time when TCP slow start is operating.

1-6, 7-11, 23-26

b) **Congestion Avoidance:** identify the intervals of time when TCP congestion avoidance is operating.

11-15, 16-22, 26-32

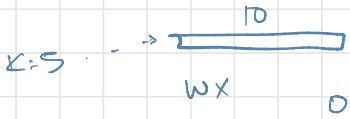
c) **Fast Retransmission**: identify the intervals of time when TCP fast retransmission is used.

15-16

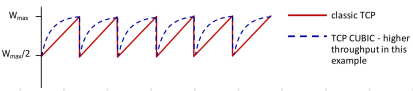
d) **Fast Recovery:** identify the intervals of time when TCP fast recovery is operating.

15-16

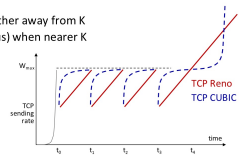
TCP CUBIC



- Is there a better way than AIMD to "probe" for usable bandwidth?
- Insight/intuition:
  - W_{max}: sending rate at which congestion loss was detected
  - congestion state of bottleneck link probably (?) hasn't changed much
  - after cutting rate/window in half on loss, initially ramp to to W_{max} *faster*, but then approach W_{max} more *slowly*



- K: point in time when TCP window size will reach W_{max}
  - K itself is tuneable
- increase W as a function of the *cube* of the distance between current time and K
  - larger increases when further away from K
  - smaller increases (cautious) when nearer K
- TCP CUBIC default in Linux, most popular TCP for popular Web servers

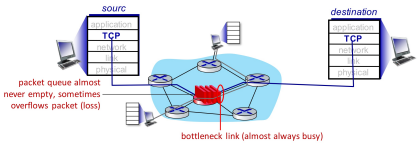


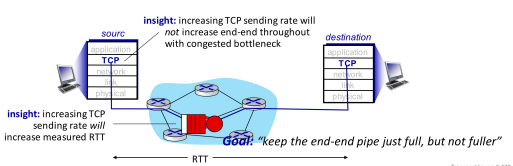
TCP and the congested "bottleneck link"

↳ TCP (CUBIC) increase TCP's sending rate until Pkt loss occurs at some routers output → the bottleneck link

↳ understanding congestion

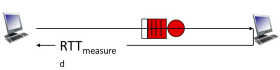
↳ useful to focus on congested bottleneck link





Delay-based TCP congestion control

Keeping sender-to-receiver pipe "just full enough, but no fuller": keep bottleneck link busy transmitting, but avoid high delays/buffering



$$\text{measured throughput} = \frac{\text{\# bytes sent in last RTT interval}}{RTT_{measure}}$$

d

Delay-based approach:

- RTT_{min} - minimum observed RTT (uncongested path)
- uncongested throughput with congestion window `cwnd` is cwnd/RTT_{min}

```
if measured throughput "very close" to uncongested throughput
    increase cwnd linearly          /* since path not congested */
else if measured throughput "far below" uncongested throughout
    decrease cwnd linearly          /* since path is congested */
```

- congestion control without inducing/forcing loss
- maximizing throughout ("keeping the just pipe full… ") while keeping delay low ("…but not fuller")
- a number of deployed TCPs take a delay-based approach
  - BBR deployed on Google's (internal) backbone network

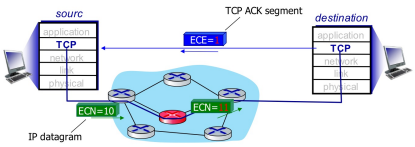
Throughput

↳ rate at which bits are being sent from sender to receiver

Explicit congestion notification (ECN)

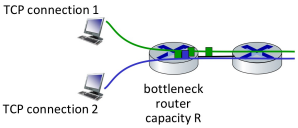
TCP deployments often implement *network-assisted* congestion control:

- two bits in IP header (ToS field) marked *by network router* to indicate congestion
  - *policy* to determine marking chosen by network operator
- congestion indication carried to destination
- destination sets ECE bit on ACK segment to notify sender of congestion
- involves both IP (IP header ECN bit marking) and TCP (TCP header C,E bit marking)



TCP Fairness

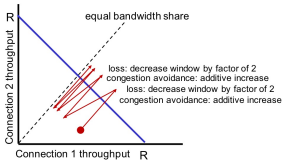
Fairness goal: if *K* TCP sessions share same bottleneck link of bandwidth *R*, each should have average rate of *R/K*



Q: is TCP Fair?

Example: two competing TCP sessions:

- additive increase gives slope of 1, as throughout increases
- multiplicative decrease decreases throughput proportionally



Is TCP fair?

A: Yes, under idealized assumptions:

- same RTT
- fixed number of sessions only in congestion avoidance

Fairness: must all network apps be "fair"?

Fairness and UDP

- multimedia apps often do not use TCP
  - do not want rate throttled by congestion control
- instead use UDP:
  - send audio/video at constant rate, tolerate packet loss
- there is no "Internet police" policing use of congestion control

Fairness, parallel TCP connections

- application can open *multiple* parallel connections between two hosts
- web browsers do this , e.g., link of rate R with 9 existing connections:
  - new app asks for 1 TCP, gets rate R/10
  - new app asks for 11 TCPs, gets R/2

Evolving Transport Layer Functionality

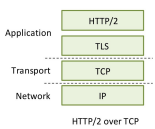
- TCP, UDP: principal transport protocols for 40 years
- different "flavors" of TCP developed, for specific scenarios:

Scenario	Challenges
Long, fat pipes (large data transfers)	Many packets "in flight"; loss shuts down pipeline
Wireless networks	Loss due to noisy wireless links, mobility; TCP treat this as congestion loss
Long-delay links	Extremely long RTTs
Data center networks	Latency sensitive
Background traffic flows	Low priority, "background" TCP flows

- moving transport–layer functions to application layer, on top of UDP
  - HTTP/3: QUIC

QUIC: Quick UDP Internet Connections

- application-layer protocol, on top of UDP
  - increase performance of HTTP
  - deployed on many Google servers, apps (Chrome, mobile YouTube app)



HTTP/2 over TCP

adopts approaches we've studied in this chapter for connection establishment, error control, congestion control

- **error and congestion control:** "Readers familiar with TCP's loss detection and congestion control will find algorithms here that parallel well-known TCP ones." [from QUIC specification]
- **connection establishment:** reliability, congestion control, authentication, encryption, state established in one RTT

- multiple application-level "streams" multiplexed over single QUIC connection
  - separate reliable data transfer, security
  - common congestion control

QUIC: Connection establishment

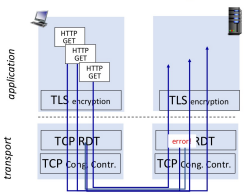
TCP (reliability, congestion control state) + TLS (authentication, crypto state)

- 2 serial handshakes

QUIC: reliability, congestion control, authentication, crypto state

- 1 handshake

QUIC: streams: parallelism, no HOL blocking



(a) HTTP 1.1

Chapter 3: summary

- principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- instantiation, implementation in the Internet
  - UDP
  - TCP

Up next:

- leaving the network "edge" (application, transport layers)
- into the network "core"
- two network-layer chapters:
  - data plane
  - control plane

NETWORK LAYER : DATA PLANE

CHP 4

4.1 Network layer services and protocol

- transport segment from sending to receiving host
  - sender
    - encapsulates segments into datagrams
    - passes to link layer
  - receiver
    - delivers segments to transport layer protocol

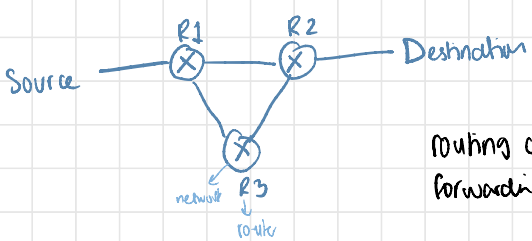
- router
  - examines header files in all IP datagrams
  - moves datagrams from input to output ports to transfer

Two key network layer functions

1. FORWARDING
   - move packets from routers input link to appropriate router output link
2. ROUTING
   - determine route taken by packets from source to destination
   - routing algorithms
     - has forwarding table

analogy: taking a trip

- *forwarding:* process of getting through single interchange
- *routing:* process of planning trip from source to destination

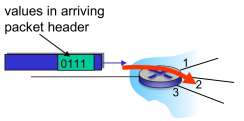


routing chooses which path
forwarding forwards the packet

Data Plane

↳ local, per-route function

↳ determines how data arrives on router input port and is forwarded to router output port



Control Plane

↳ network wide logic

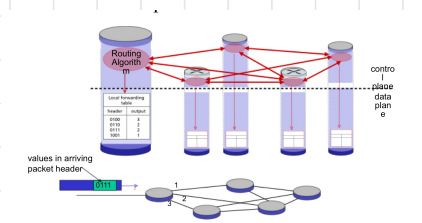
↳ determines how datagram is routed among routes along end-end paths from source to destination host

↳ 2 control plane approaches

1. Traditional Routing algos
   ↳ implemented in routers
2. Software defined networking (SDN)
   ↳ implemented in remote servers

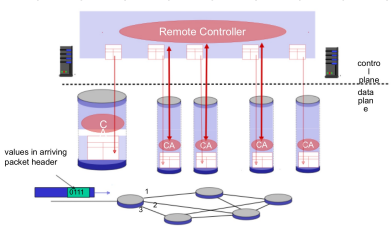
Per Router control plane

↳ individual routing algo components in each and every router interact in the control plane



SDN Control Plane

↳ remote controller computes, installs forwarding tables in routers



Question

Question 3: SDN provides a programmable control plane, however, it network infrastructure needs changes as we studied in chapter # 5 of our textbook. Now, answer the following questions. **[10 points]**

a) How the data plane is different while using SDN? Explain.

Two fold: i) data plane is based on SDN controlled switches and ii) uses flow table to forward traffic.

Figure. 1

a) Show in steps, how data packets entering one interface of a router are forwarded to an output interface of the same router.
   Solution:
   Step 1: IP address is extracted from the IP packet header.
   Step 2: IP prefix matched with forwarding table and corresponding output interface is read.
   Step 3: The IP packets is sent to the output interface queue for further processing.

b) How routing helps forwarding? Explain.
   Solution:
   The data packets using forwarding tables to traverse down the shortest paths which are calculated by routing algorithms.

4.1.2

Network service model

for individual datagrams

- guaranteed delivery
- with <40 msec delay

for a flow of datagrams

- in order datagram delivery
- guaranted min bandwidth to flow
- restrictions on changes in inter-packet spacing

best effort service model

- NO guarantees on
  - successfull datagram delivery to destination
  - timing/order of delivery
  - bandwidth avl to end-end flow

eg internet

Network-layer service model

Network Architecture	Service Model	Quality of Service (QoS) Guarantees ?			
		Bandwidth	Loss	Order	Timing
Internet	best effort	none	no	no	no
ATM	Constant Bit Rate	Constant rate	yes	yes	yes
ATM	Available Bit Rate	Guaranteed min	no	yes	no
Internet	Intserv Guaranteed (RFC 1633)	yes	yes	yes	yes
Internet	Diffserv (RFC 2475)	possible	possibly	possibly	no

pros best effort service

1. Simplicity of mechanism
   - allows internet to be widely deployed adopted
2. Sufficient provisioning of bandwidth
   - allows performance of real time applications

   e.g interactive voice, video
3. replicated application layer distributed services
   - connects close to clients networks

   allowing services to be provided from multiple locations
4. congestion control of "elastic" services helps

4.2.1

Destination Based Forwarding

forwarding table

Destination Address Range	Link Interface
11001000 00010111 00010000 00000000 through	0
11001000 00010111 00010000 00000100 through 11001000 00010111 00010000 00000111	3
11001000 00010111 00011000 11111111	
11001000 00010111 00011001 00000000 through 11001000 00010111 00011111 11111111	2
otherwise	3

Q: but what happens if ranges don't divide up so nicely?

Longest Prefix Matching

longest prefix match

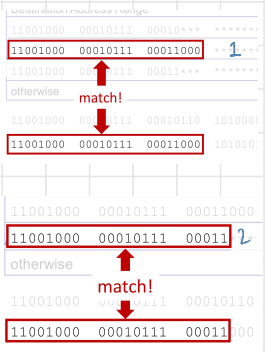
when looking for forwarding table entry for given destination address, use *longest* address prefix that matches destination address.

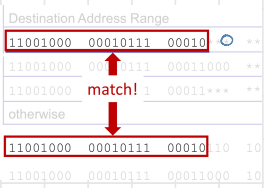
Destination Address Range	Link interface
11001000 00010111 00010*** ********	0
11001000 00010111 00011000 ********	1
11001000 00010111 00011*** ********	2
otherwise	3

examples:

11001000 00010111 00010110 10100001 which interface? 0

11001000 00010111 00011000 10101010 which interface? 1





4.2.5

Packet Scheduling

↳ decides which packet to send next on link
- ↳ FCFS
- ↳ Priority
- ↳ Round Robin
- ↳ Weighted fair queueing

1. FCFS

↳ first come first serve

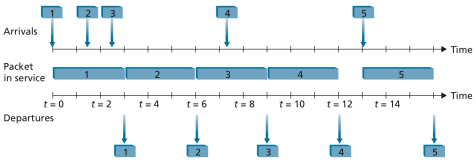
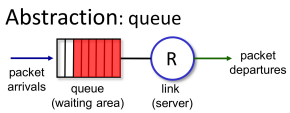


Figure 4.12 ♦ The FIFO queue in operation

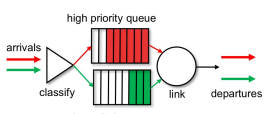


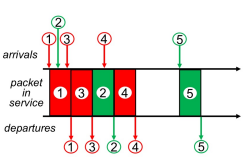
→ every one allocated same time

2. Priority Scheduling

↳ arriving traffic classified, queued by class
- ↳ any header files can be used for classification

↳ sends packet from highest priority queue that has buffered packets
- ↳ FCFS with priority class





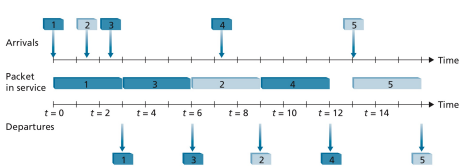


Figure 4.14 ♦ The priority queue in operation

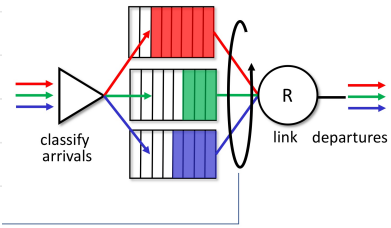
→ every one allocated same time

→ 3 had more priority than 2 and 3 and 2 were born arrived

3. Round Robin Scheduling (RR)

↳ arriving traffic classified,
queued by class
↳ any header files can be used for classification

↳ server cyclically scans class queues,
sending 1 complete packet from each class
in turn



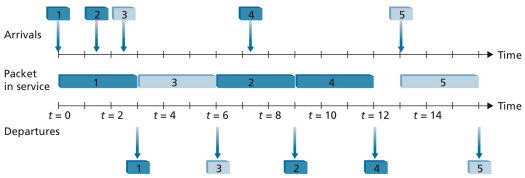


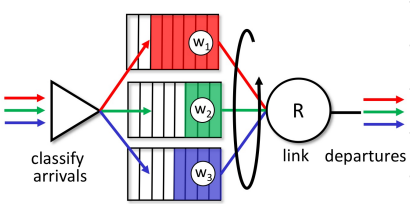
Figure 4.15 ♦ The two-class robin queue in operation

class A: 1, 2, 4
class B: 3, 5
picks alternate classes

weighted Fair Queuing (WFQ)

↳ generalized Round Robin
↳ each class i has weight w
and gets weight amount of service in each cycle
↳ min bandwidth guarantee → Per traffic class

$$\frac{w_i}{\sum_j w_j}$$

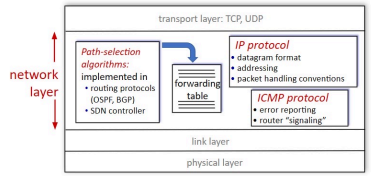


4.3

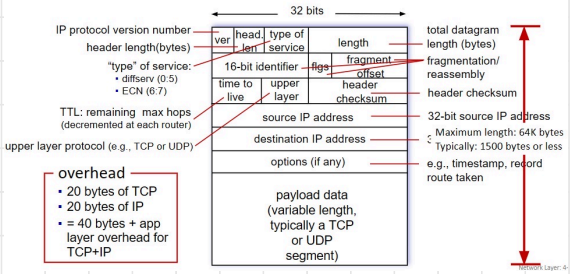
Network Layer: Internet

↳ host functions

↳ router network layer functions



IP Datagram format



↳ needs 3 things to transfer data S → D

1. IP protocol
   - ↳ what format of datagram
   - ↳ how to handle Pkt
2. Path selection Algo / Routing Protocol
   - ↳ gets shortest path
3. ICMP Protocol
   - ↳ tells errors

∴ UDP header = 8 bytes

∴ TCP header = 20 bytes

∴ Network Layer header OR IP = 20 bytes

overhead: TCP + IP + application Layer data

20 + 20

4.4

IP ADDRESSING

IP address - IPv4

↳ 32 bit identifier

↳ associated with each host/router interface

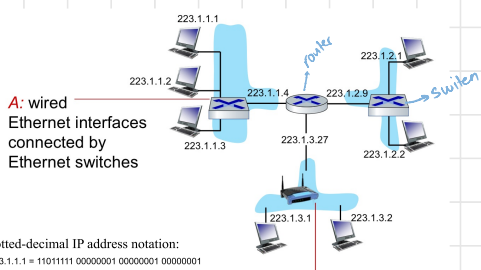
interface

↳ connection b/w host/router and physical link

↳ routers have multiple interfaces

↳ host have 1 or 2 interfaces
- ↳ wired Ethernet
- ↳ wireless

↳ every interface has an IP address



Subnet

↳ device interfaces
  that physically reach eachother
  w/o passing through an intervening router

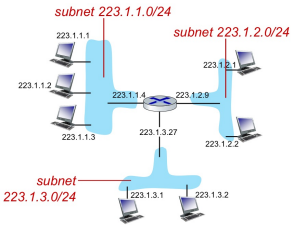
IP address structure

↳ subnet part - high order eg 223.1.1.1
  ↳ devices in same subnet
    have common high order bits
↳ host part - low order eg 223.1.1.1
  ↳ remaining lower bits

Subnets

Recipe for defining subnets:

- detach each interface from its host or router, creating "islands" of isolated networks
- each isolated network is called a *subnet*

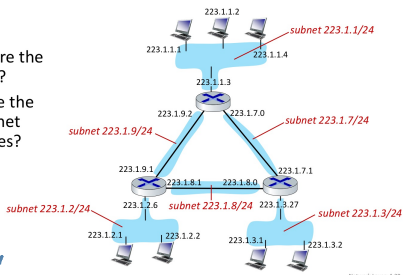


subnet mask: /24
(high-order 24 bits: subnet part of IP address)

↓ 3 subnets

Subnets

- where are the subnets?
- what are the /24 subnet addresses?

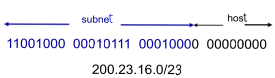


↓ 6 subnets

CIDR → Classless Inter Domain Routing

↳ subnet portion of address of arbitrary length
↳ same address format

a.b.c.d/x → no of bits in subnet portion of address



200.23.16.0/23

IPv6: motivation

- initial motivation
  - 32 bit IPv4 address space would be completely allocated
- additional motivation
  - speed processing: 40 byte fixed length header
  - enable diff network layer treatment of flows

IPv6 doesn't have compared to IPv4

- no checksum → to speed processing at routes
- no fragmentation/reassembly
- no options → all as upper layer, next header protocol at router

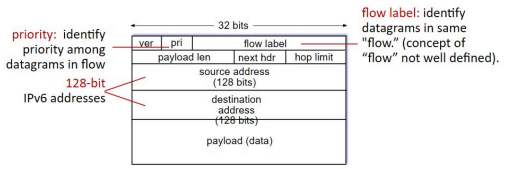
Transition from IPv4 to IPv6

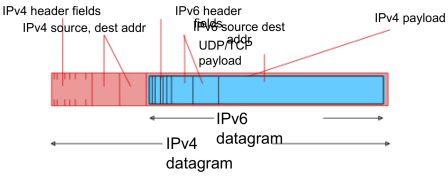
- not all routes can be upgraded simultaneously
  - no 'flag' days

Tunneling

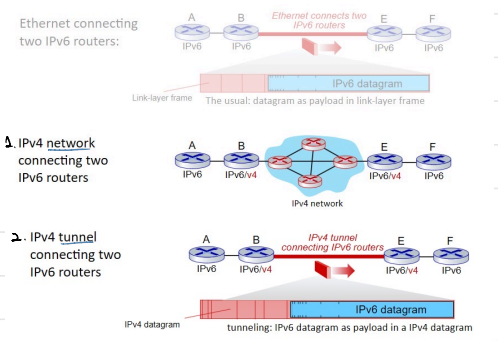
- allows network to operate with mixed IPv4 and IPv6 routes
- IPv6 datagram carried as payload in IPv4 datagram among IPv4 routes → packet within a packet

IPv6 datagram format





Tunneling and encapsulation



Tunneling