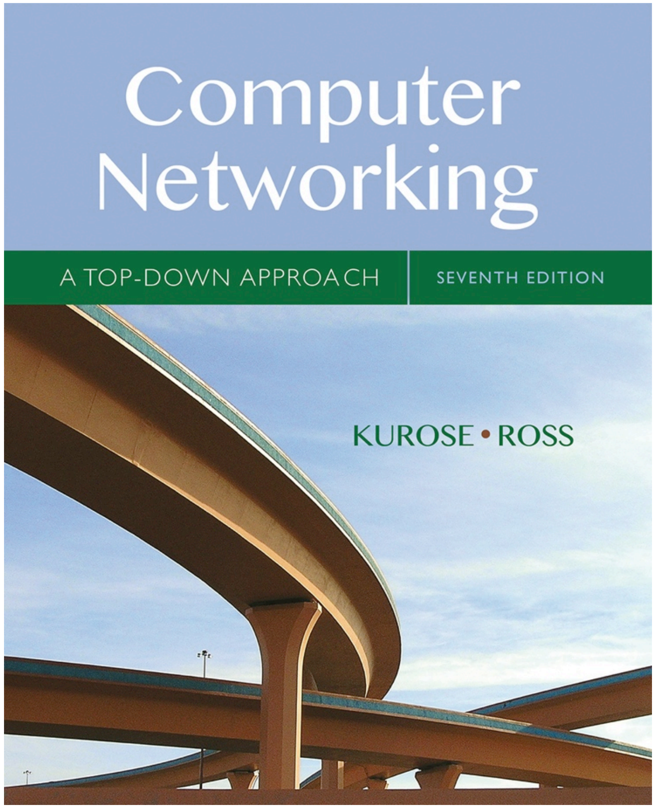
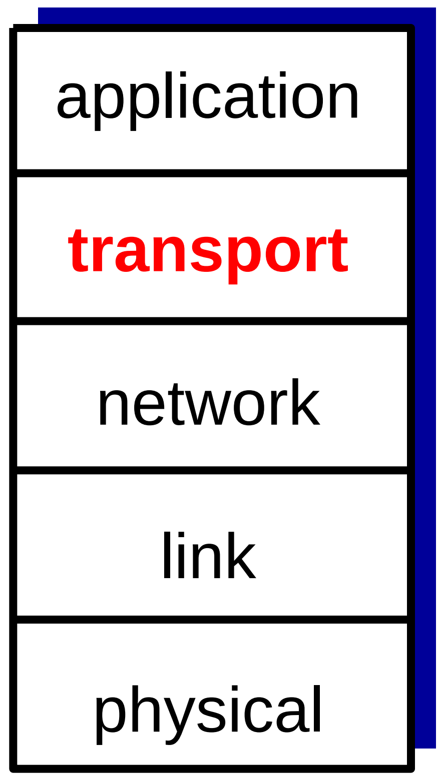
Introduction to TCP/IP II



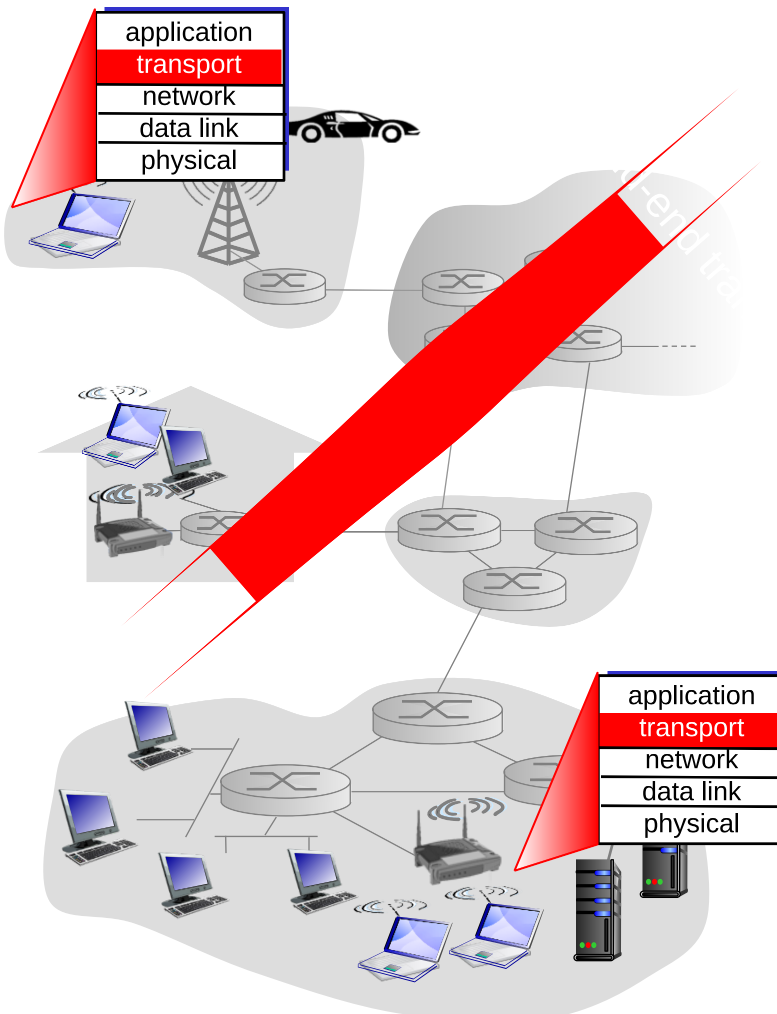
7th edition
Jim Kurose, Keith Ross
Pearson/Addison Wesley
April 2016

- essentially adapted from Kurose and Ross


application
transport
network
link
physical

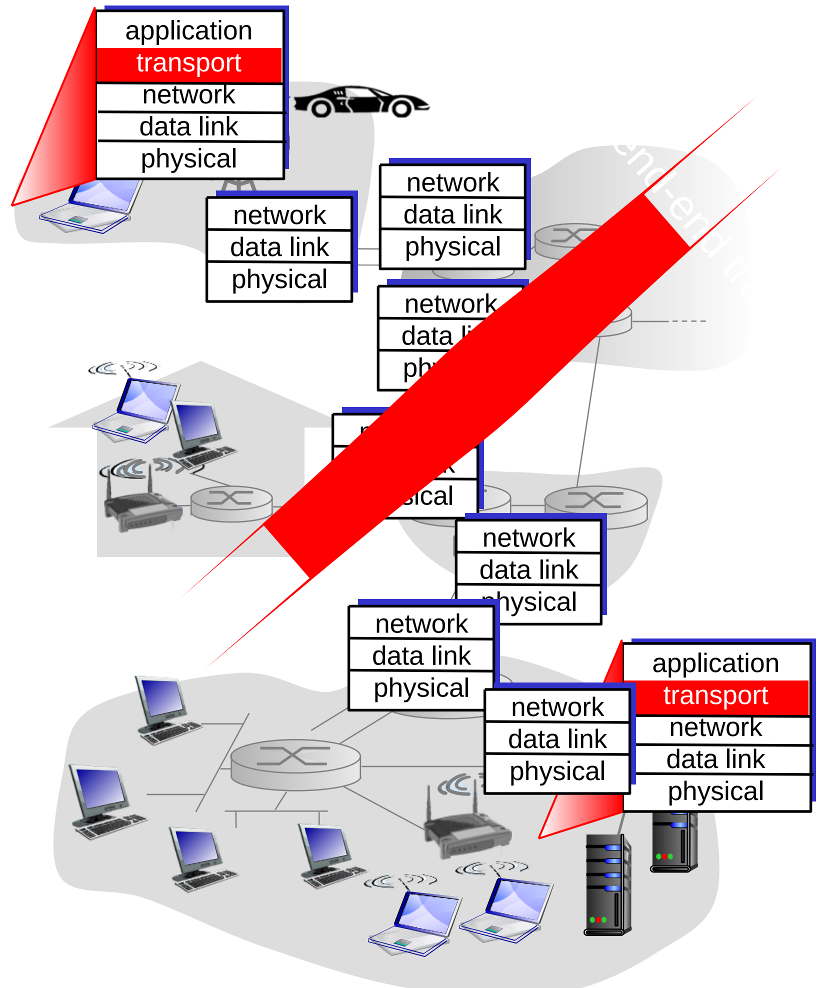
Transport services and protocols

- provide *logical communication* between app processes running on different hosts
- transport protocols run in end systems
  - send side: breaks app messages into *segments*, passes to network layer
  - rcv side: reassembles segments into messages, passes to app layer



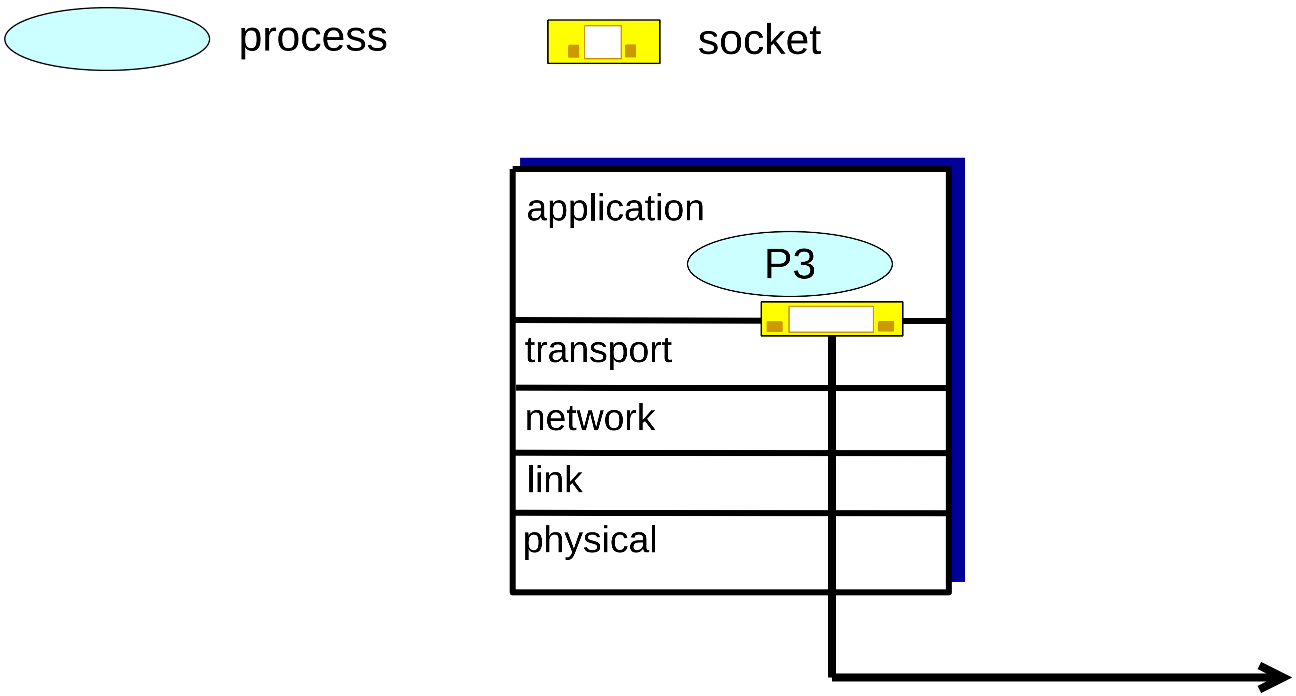
Internet transport-layer protocols

- reliable, in-order delivery (TCP)
  - congestion control
  - flow control
  - connection setup
- unreliable, unordered delivery: UDP
  - no-frills extension of "best-effort" IP
- services not available:
  - delay guarantees



Sockets

- Application processes sends messages to (or receives messages from) transport layer through socket.
- socket programming is for this purpose: socket.send(M) and M=socket.recv()



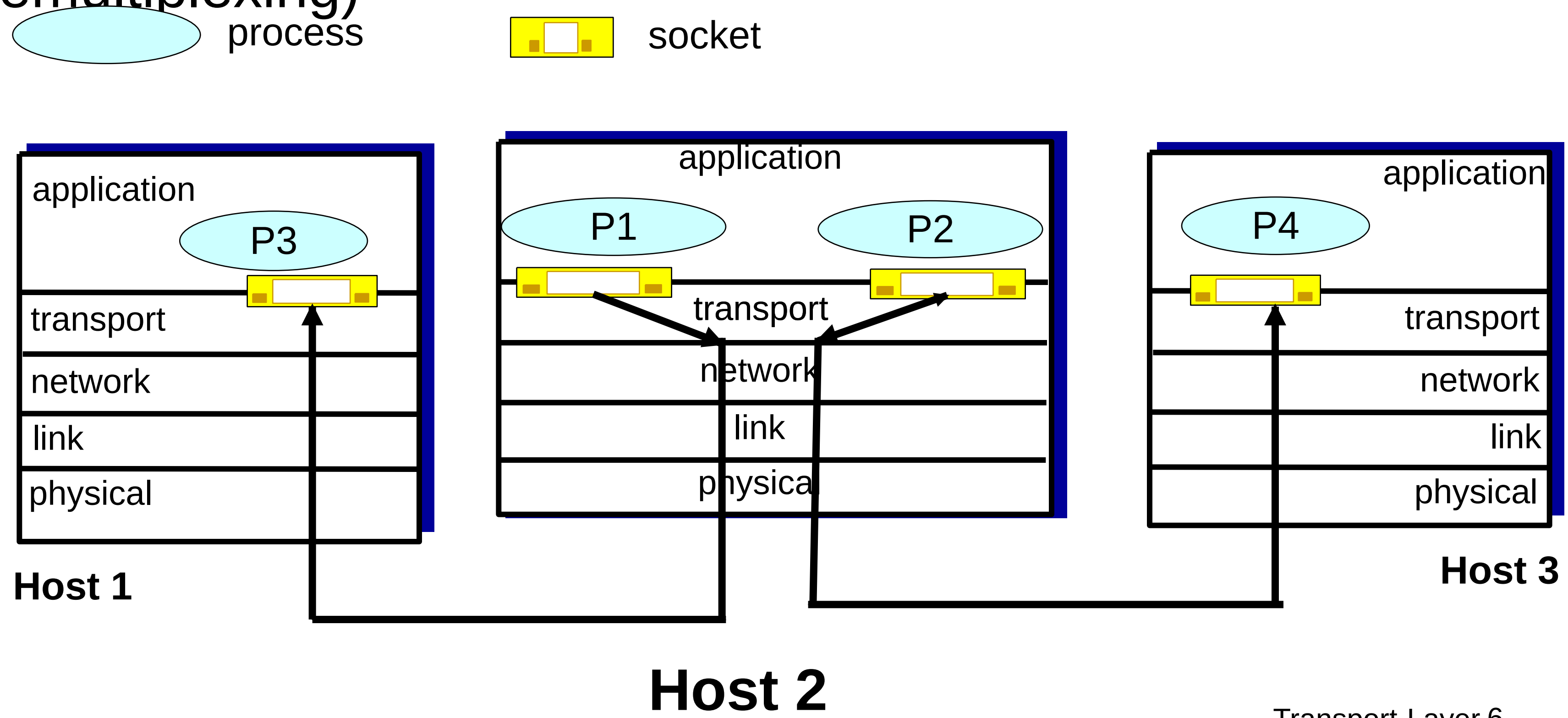
Multiplexing/demultiplexing

multiplexing at sender:

handle data from multiple sockets, add transport header (later used for demultiplexing)

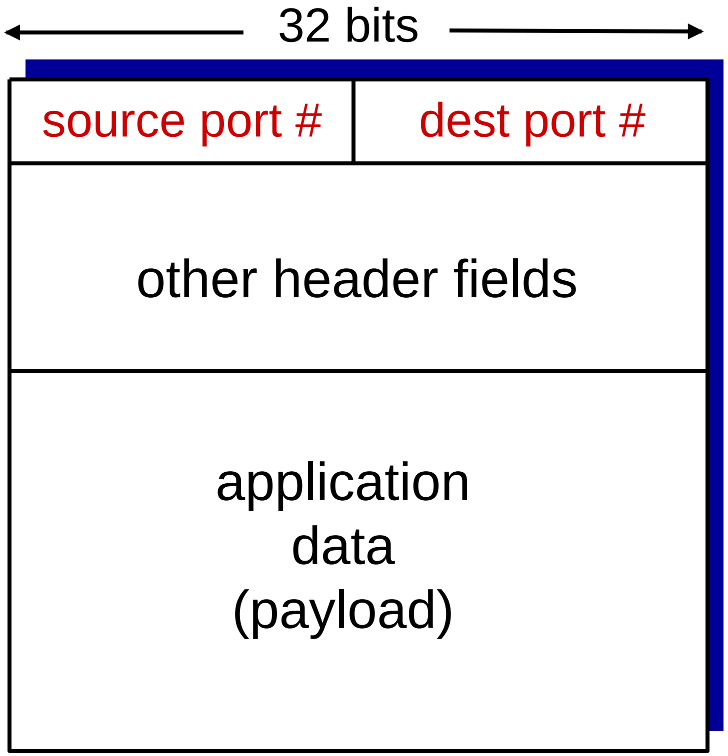
demultiplexing at receiver:

use header info to deliver received segments to correct socket



How demultiplexing works

- host receives IP datagrams
  - each datagram has source IP address, destination IP address
  - each datagram carries one transport-layer segment
  - each segment has source, destination port number
- host uses *IP addresses & port numbers* to direct



TCP/UDP segment format

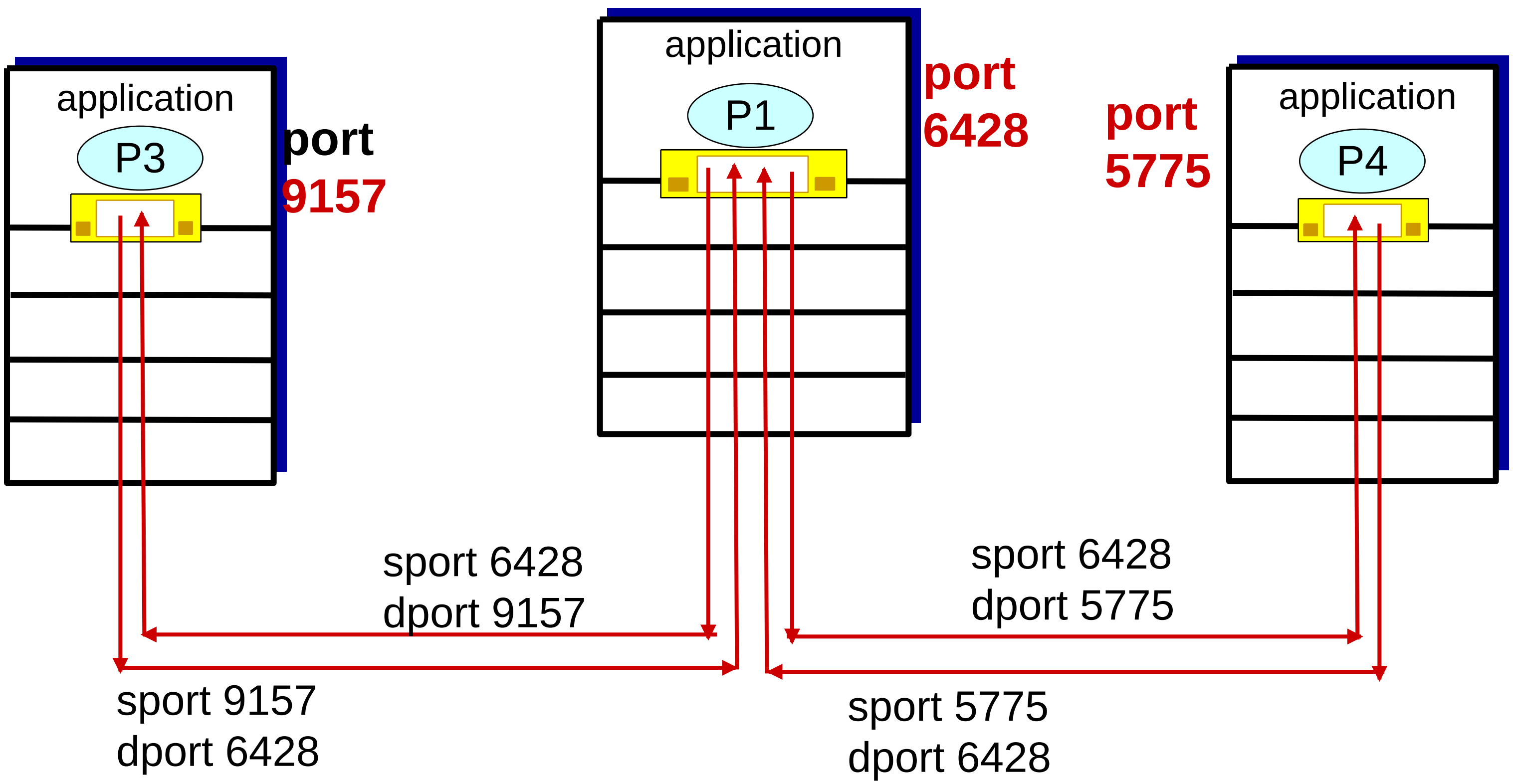
UDP demultiplexing

- 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

- a UDP server (e.g., DNS) with two clients is an example

UDP demux: example

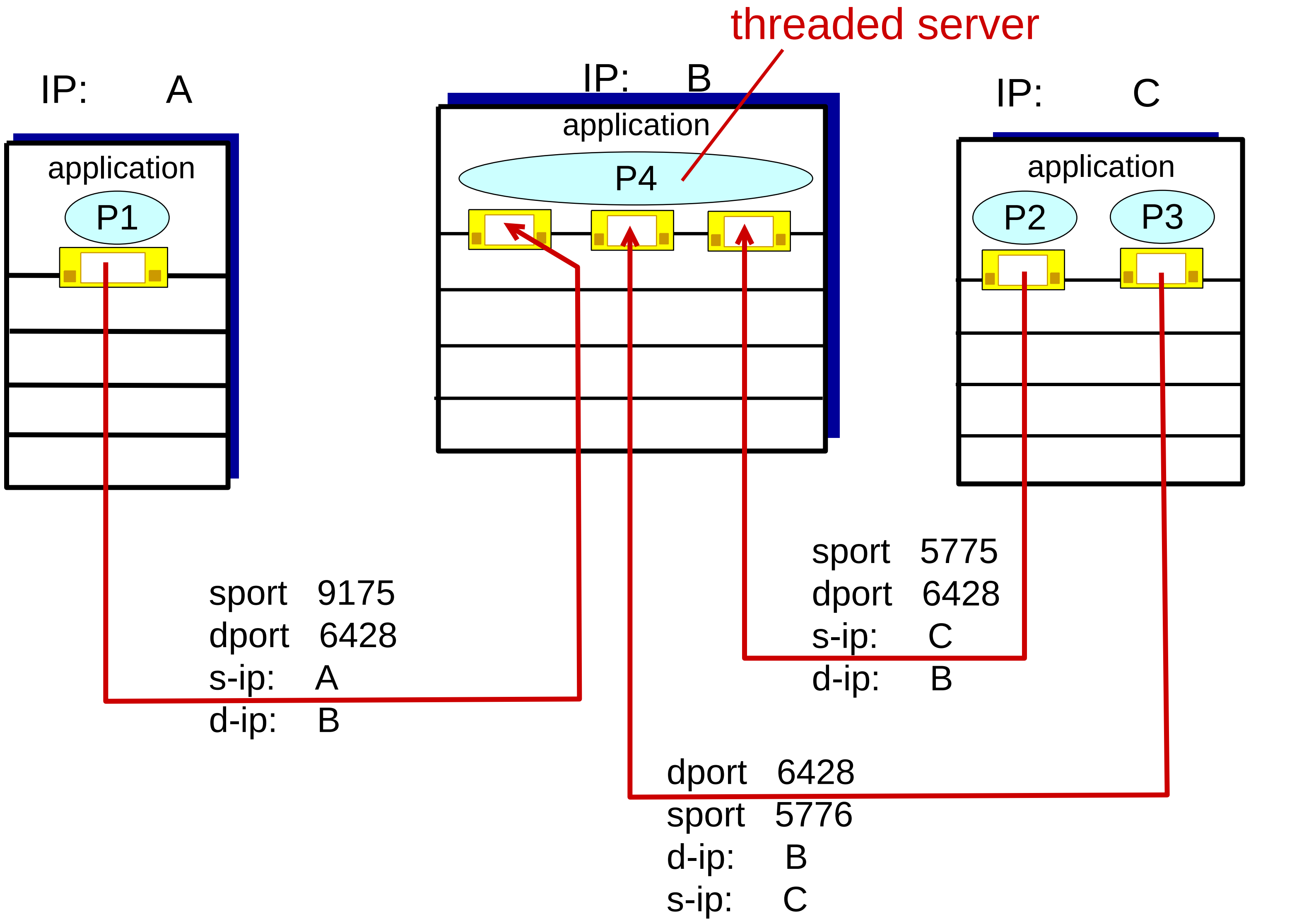


TCP demux

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

TCP demux: example

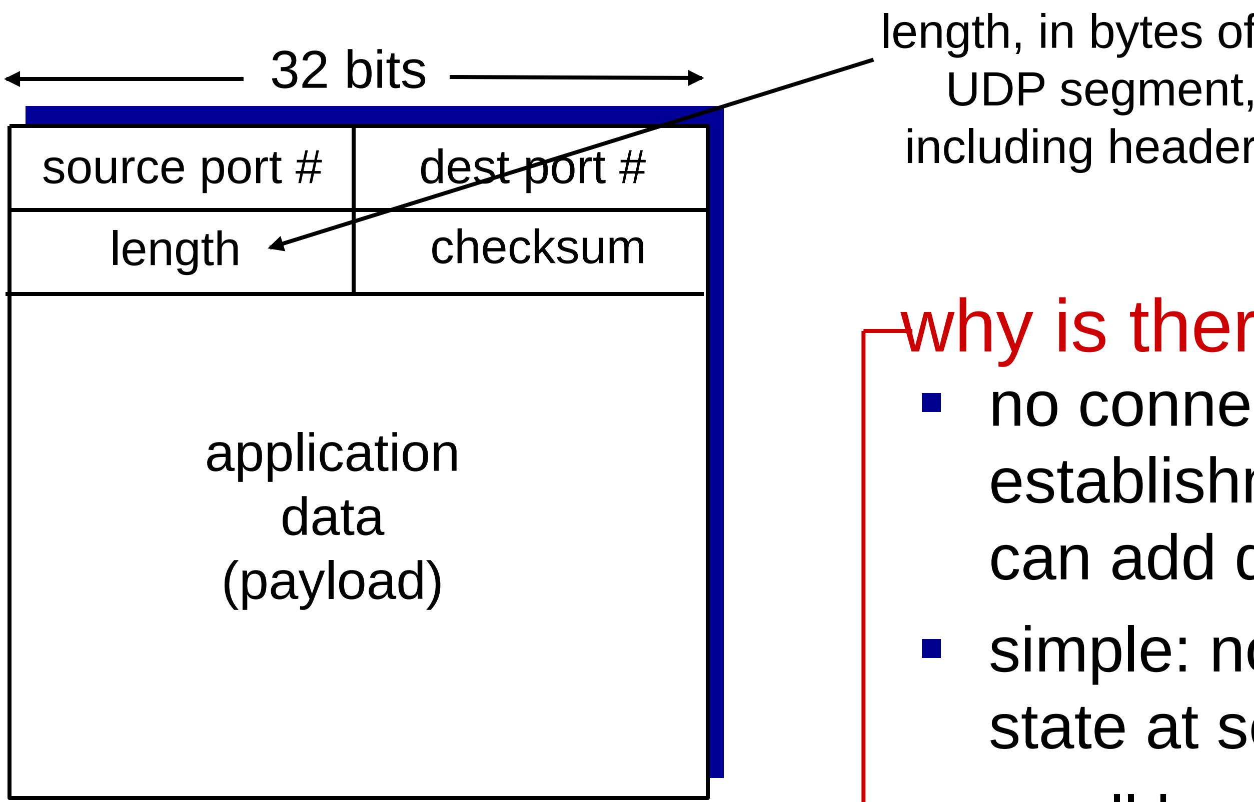


UDP: User Datagram Protocol [RFC 768]

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

- UDP use:
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
  - SNMP
- reliable transfer over UDP:
  - add reliability at application layer
  - application-specific error recovery!

UDP: segment header



UDP segment format

why is there a UDP?

- no connection establishment (which can add delay)
- simple: no connection state at sender, receiver
- small header size
- no congestion control: UDP can blast away as fast as desired

UDP checksum

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

sender:

- treat segment contents, including header fields, as sequence of 16-bit integers
- checksum: addition (one's complement sum) of segment contents
- sender puts checksum value into UDP checksum field

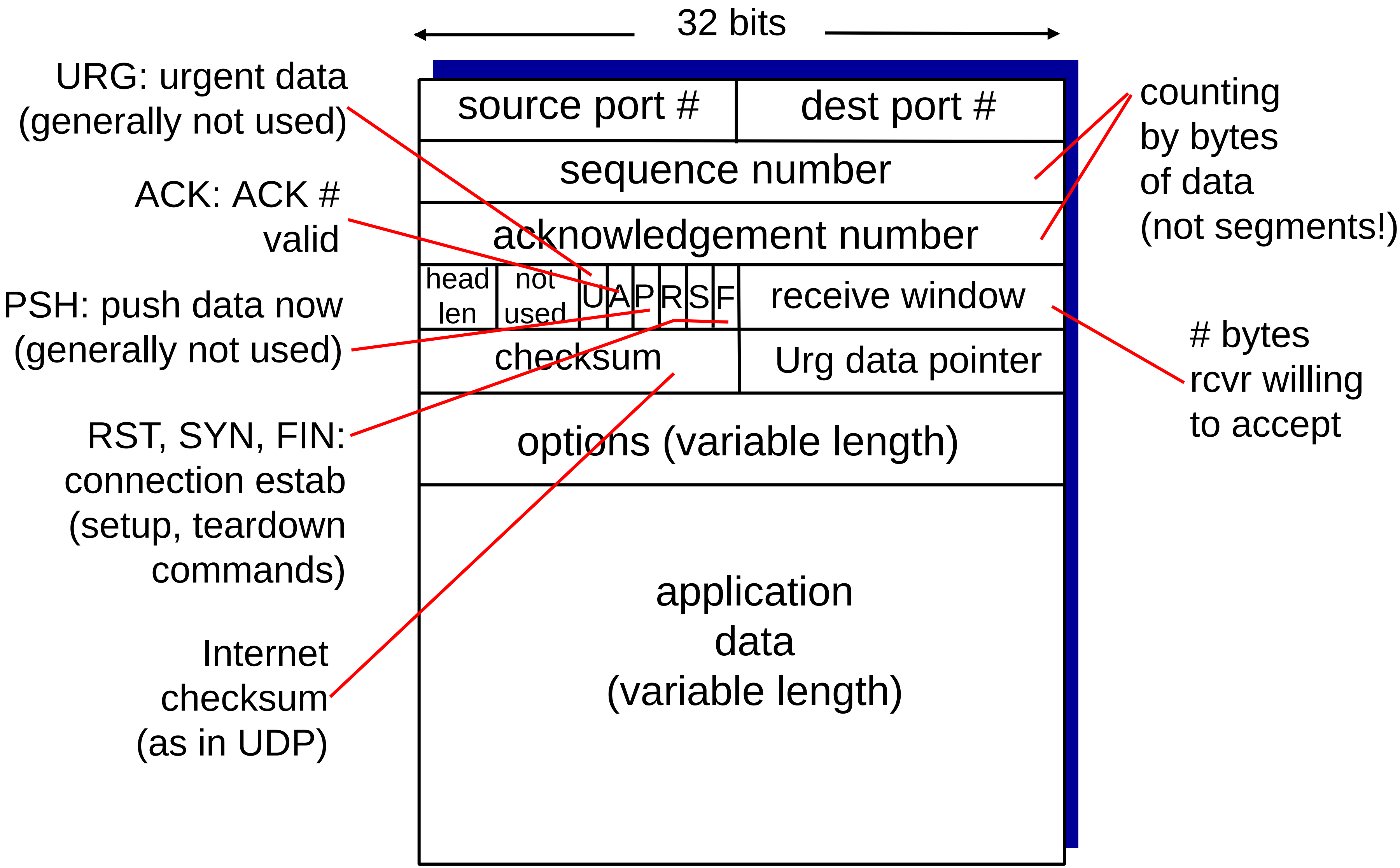
receiver:

- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected. *But maybe errors*

TCP: Overview RFCs: 793,1122,1323, 2018, 2581

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte steam:*
  - no “message boundaries”
- pipelined:
  - TCP congestion and flow control set window size
- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- connection-oriented:
  - handshaking (exchange of control msgs) inits sender, receiver state before data exchange
- flow controlled:

TCP segment structure



seq. # and ACK

sequence numbers:

- Initial seq # can be any
- Seq # (**next** packet) =seq# (**current** packet)+ #databytes (**current** packet)
- nex seq# >current seq #.
- seq# can be used to recover the packet order.

Acknowledge number:

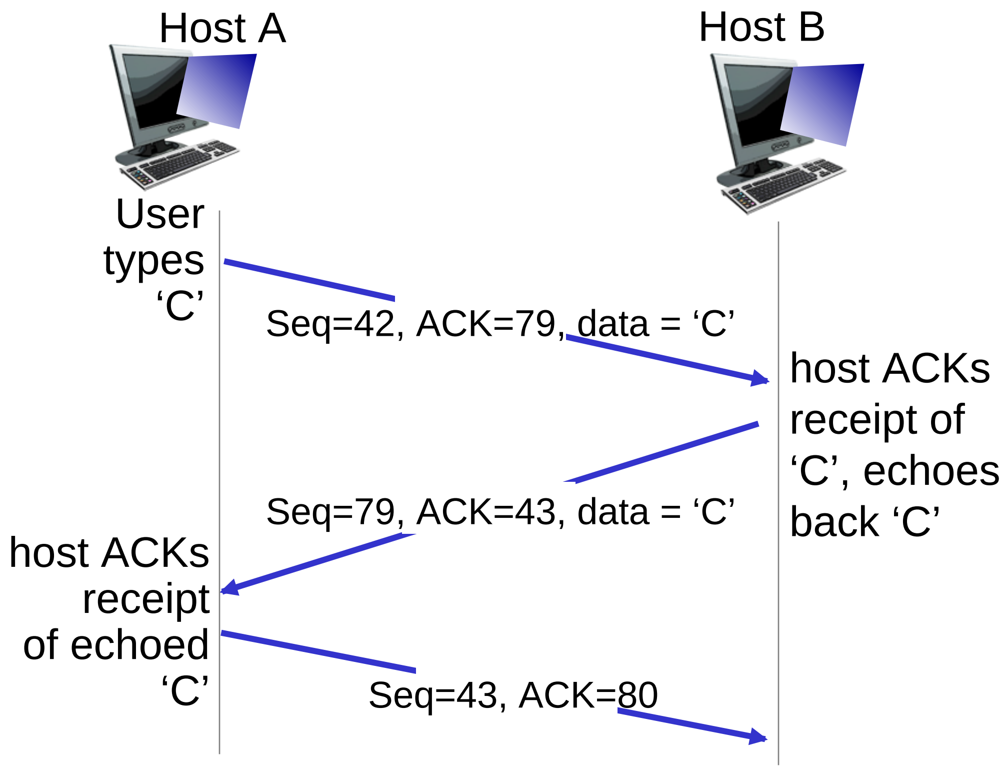
ack_num=seq# of next packet expected from other side

ack_num=502: please send your packet with seq#=502.

means packets with seq#<502 have been received.

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

TCP seq. numbers, ACKs

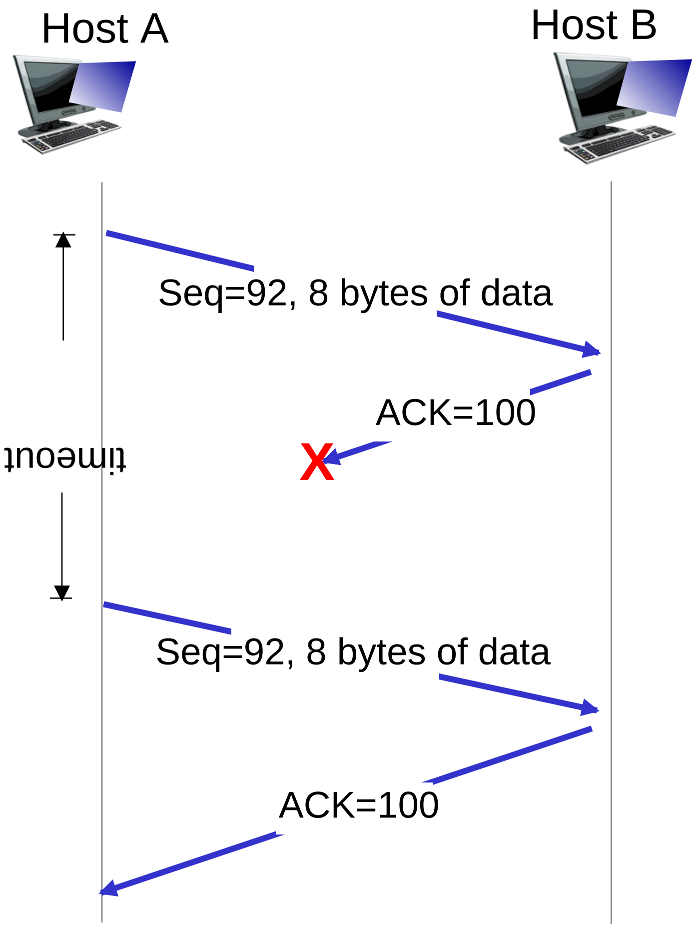


simple telnet scenario

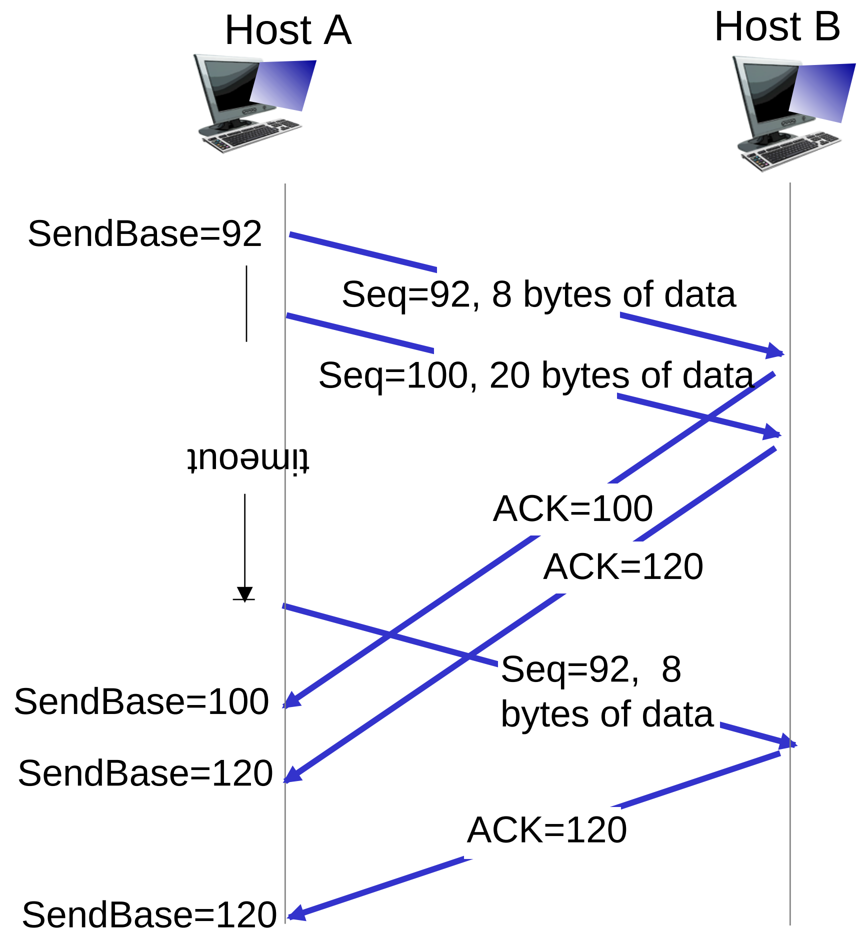
timeout

- If a segment is not acked, then the packet might be lost and so sender needs to retransmit it.
- Sender needs to set a *timeout (*e.g., 5s*)*. After waiting for this length of time, he needs to retransmit the segment.

TCP: retransmission scenarios

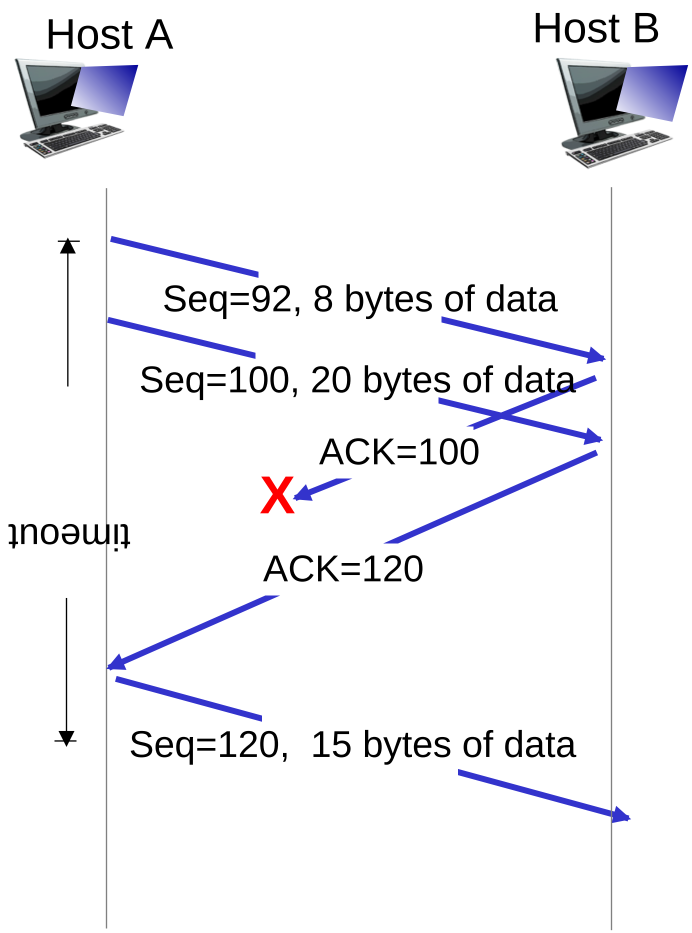


lost ACK scenario



premature timeout

TCP: retransmission scenarios

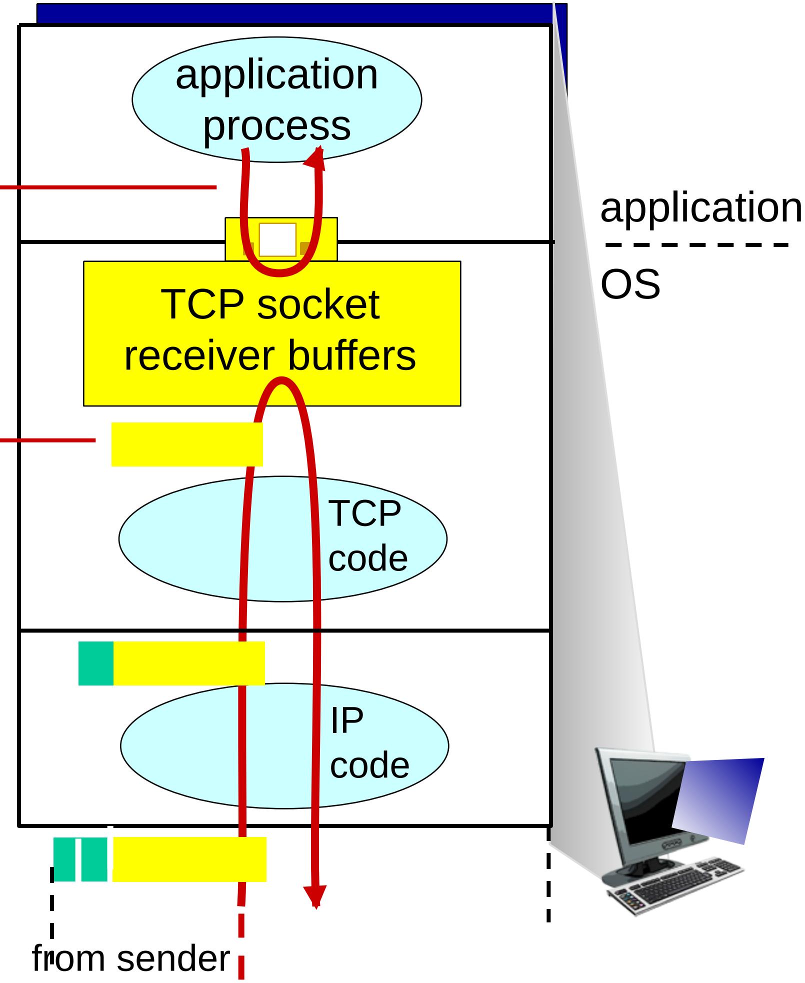


cumulative ACK

TCP flow control

application may remove data from TCP socket buffers ….

… slower than TCP receiver is delivering (sender is sending)



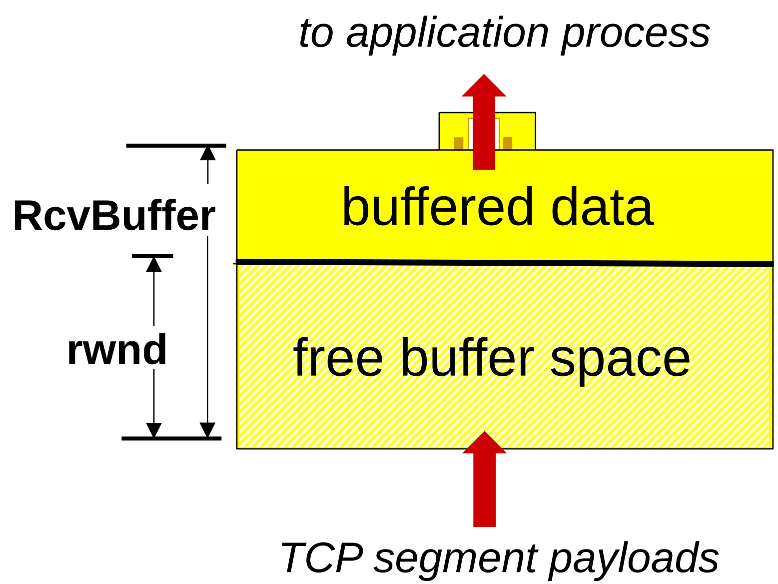
receiver protocol stack

flow control

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

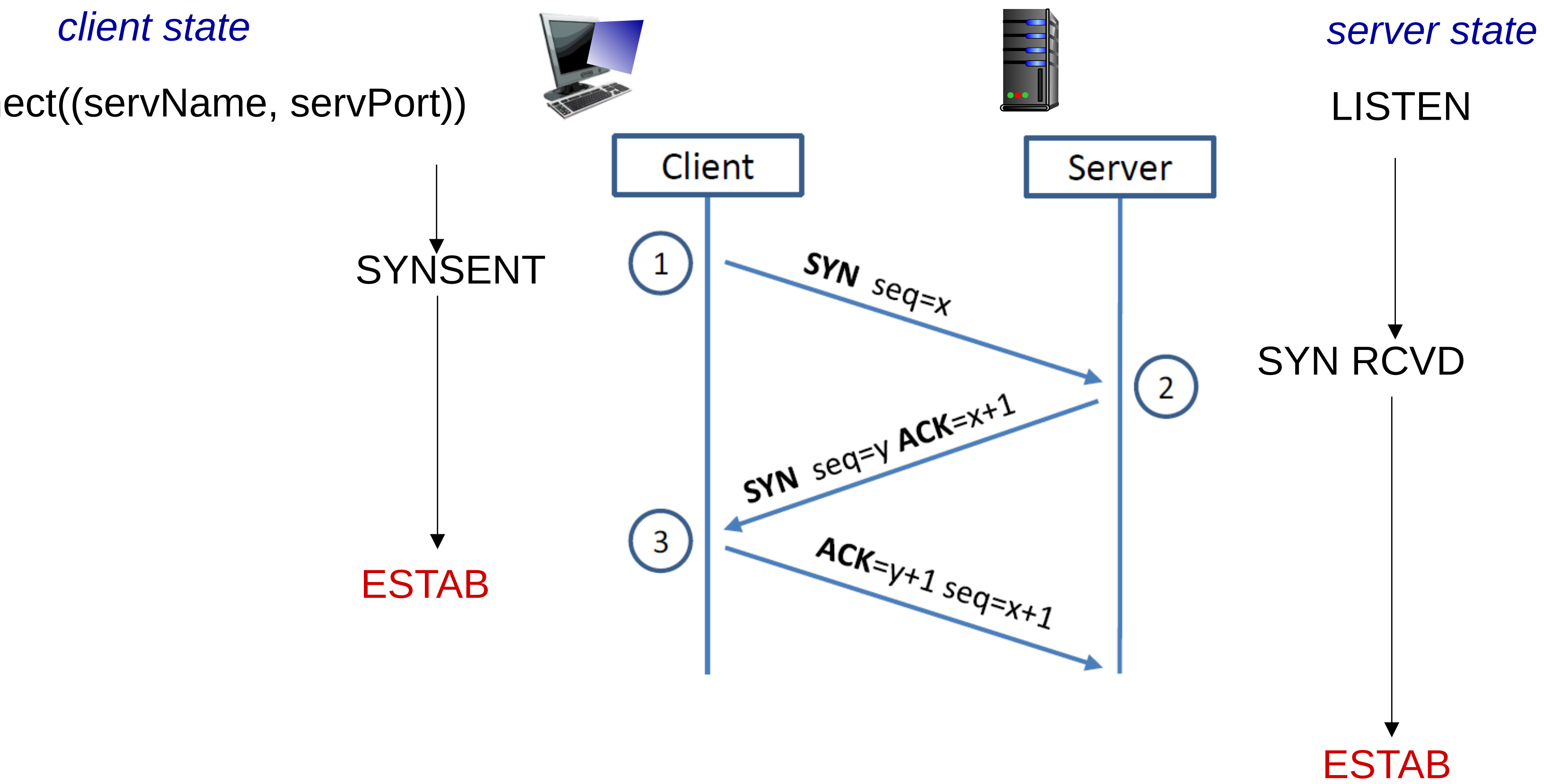
TCP flow control

- receiver puts free buffer size **rwnd** in TCP header of receiver-to-sender segments
  - **RcvBuffer=4096bytes** (typical default)
- If sender receives a packet with small rwnd, it reduces the sending speed.
- This guarantees receive buffer will not overflow



receiver-side buffering

TCP 3-way Handshake Protocol



x is random chosen by client
y is random chosen by server

TCP: closing a connection

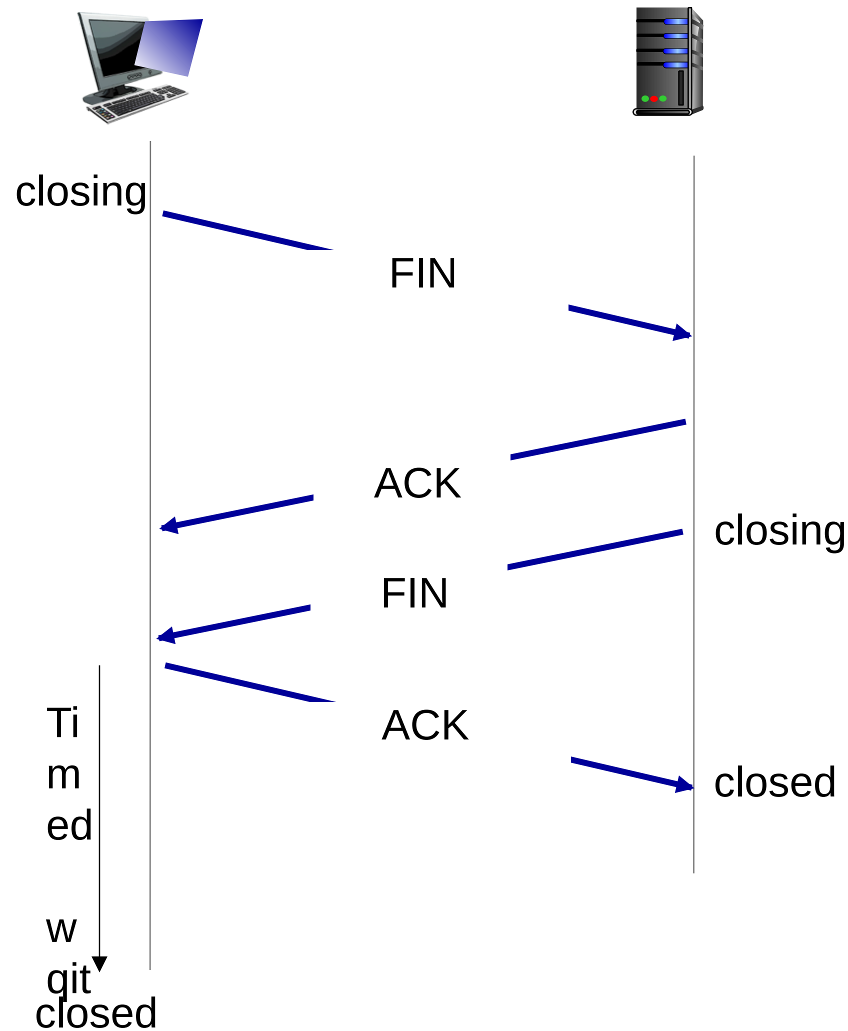
client closes socket:
close(fd);

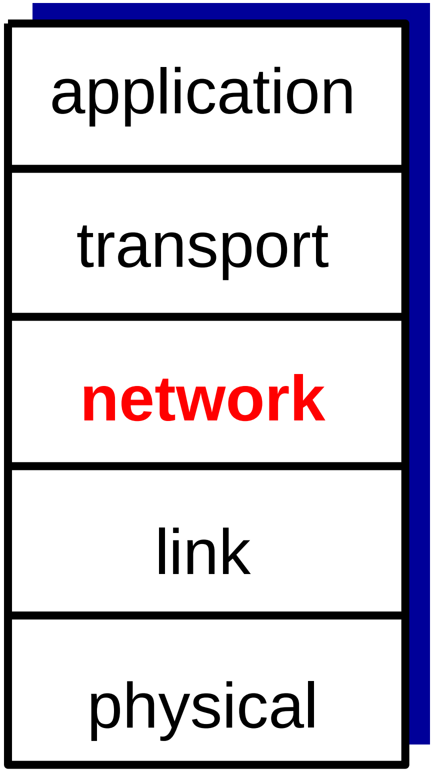
Step 1: client sends FIN segment to server to close C-> S direction

Step 2: server receives FIN, replies with ACK and also FIN to close the S->C direction.

Step 3: client receives FIN, replies with ACK and enters timed-wait state.

Step 4: server receives FIN and enter closed state.




application
transport
network
link
physical

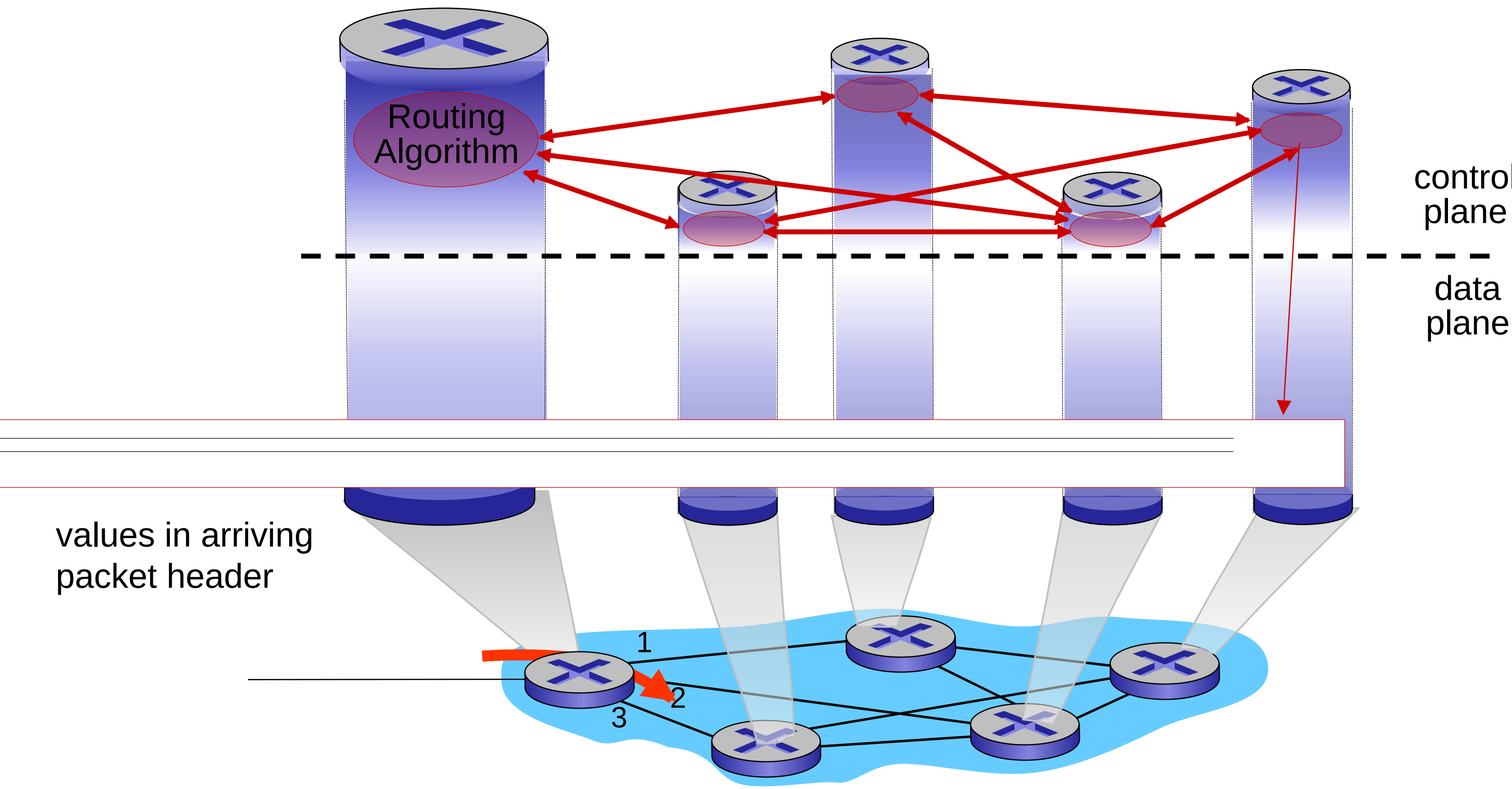
Two network-layer functions

network-layer functions:

- *forwarding:* move packets from router's input to appropriate router output
- *routing:* determine route taken by packets from source to destination
  - *routing algorithms*

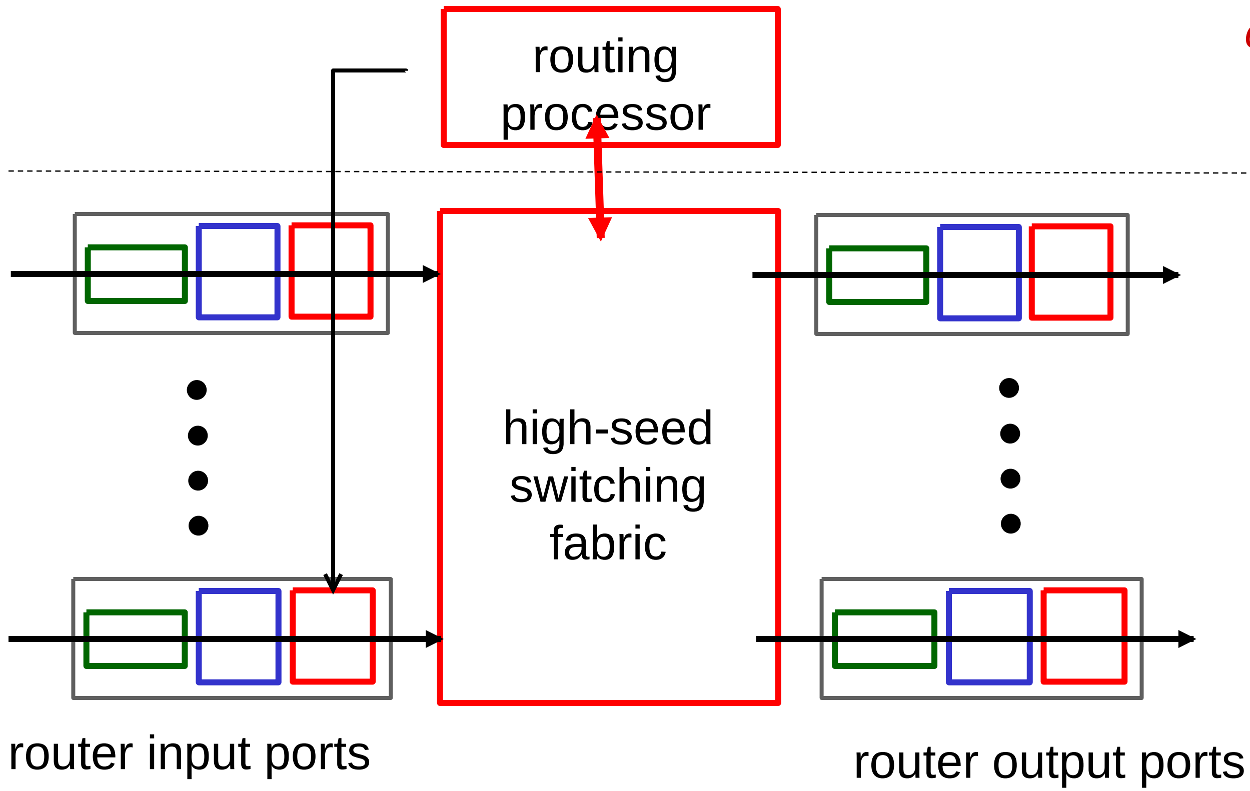
Per-router control plane

Individual routing algorithm components *in each and every router* interact in the control plane



Router architecture overview

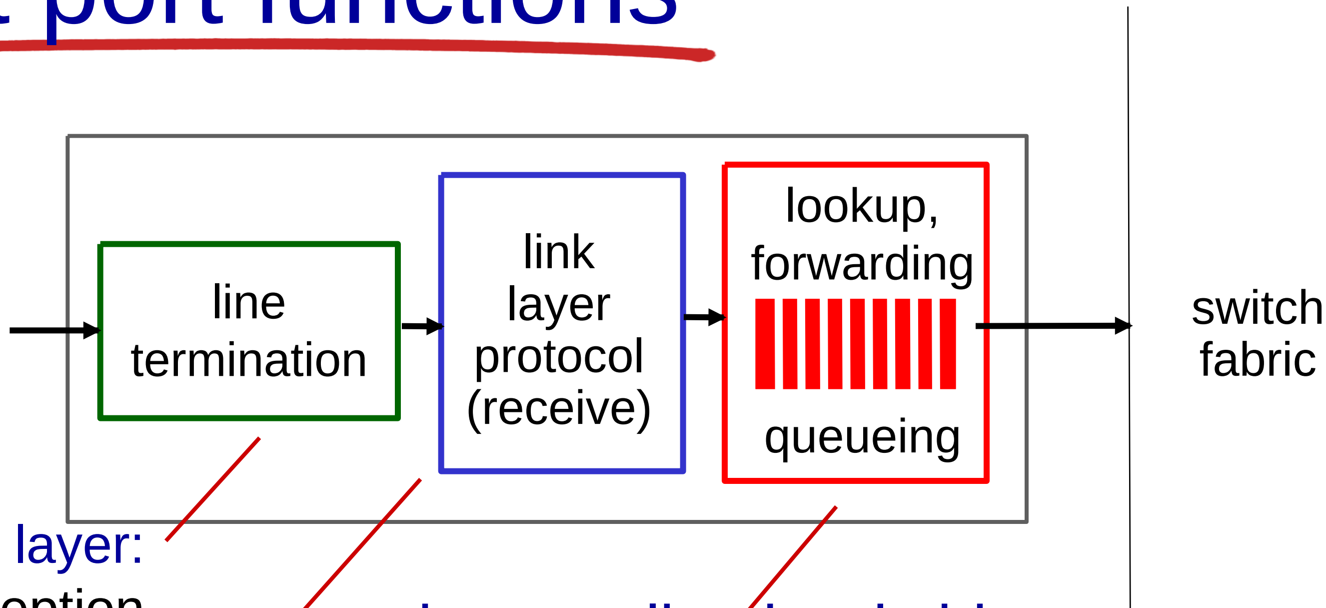
- high-level view of generic router architecture:



routing, management
control plane (software)
operates in millisecond
time frame

forwarding data plane
(hardware) operttes in
nanosecond
timeframe

Input port functions



physical layer:
bit-level reception

data link layer:
e.g., Ethernet
(chapter 5)

decentralized switching:

- *destination-based forwarding:* forward based only on destination IP address (traditional)
- *generalized forwarding:* forward based on any set of header field values
- queuing: if datagrams arrive faster than forwarding rate into switch fabric
  - if queue is full, the arriving packet is dropped.

Destination-based forwarding

forwarding table

Destination IP Address Range	Link Interface
11001000 00010111 00010000 00000000 through **11001000 00010111 00010111 11111111**	0
11001000 00010111 00011000 00000000 through **11001000 00010111 00011000 11111111**	1
11001000 00010111 00011001 00000000 through **11001000 00010111 00011111 11111111**	2
otherwise	3

Longest prefix matching

longest prefix matching

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

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

examples:

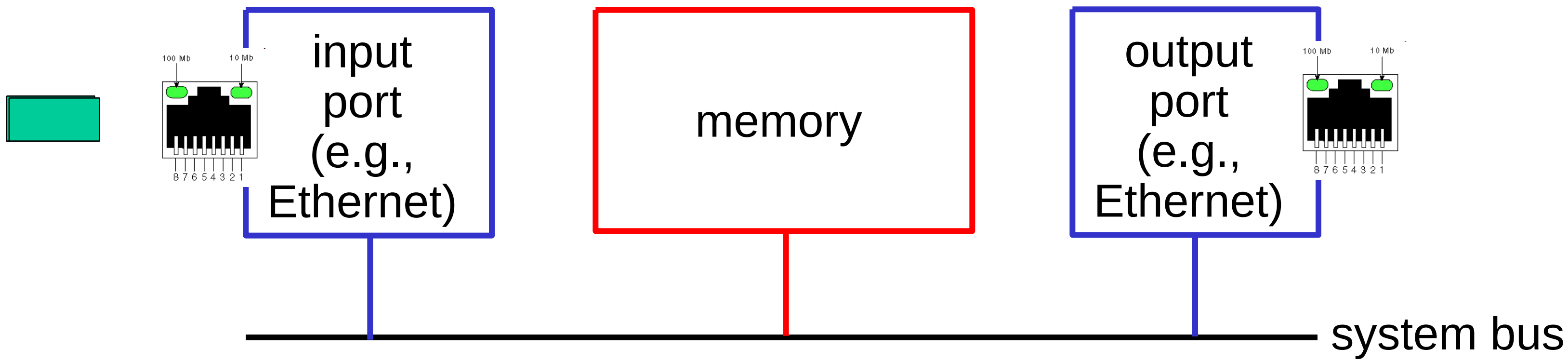
DA: 11001000 00010111 00010110 10100001 which interface?

DA: 11001000 00010111 00011000 10101010 which interface?

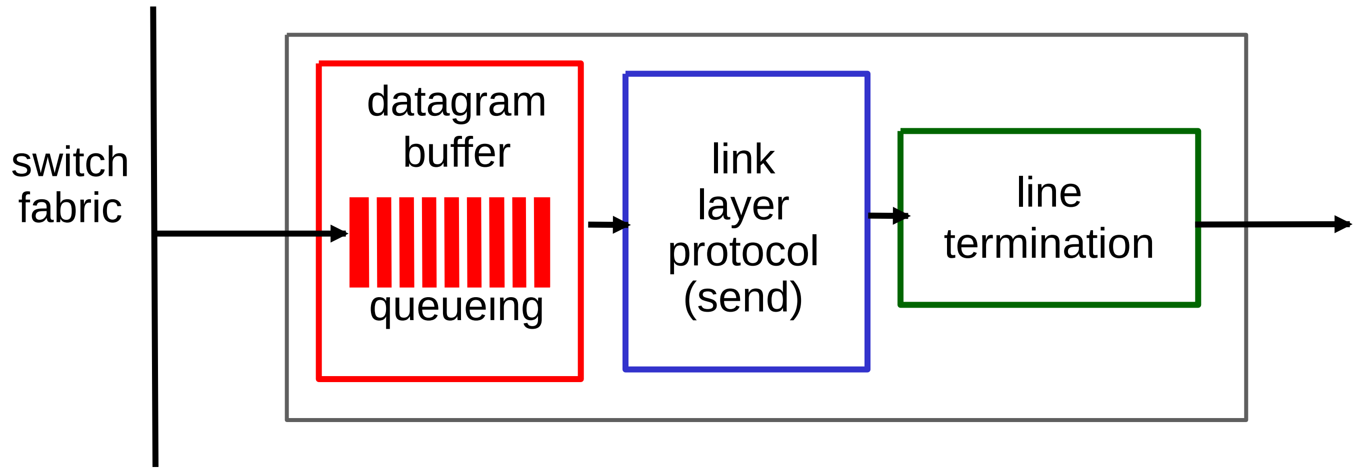
Switching via memory

first generation routers:

- traditional computers with switching under direct control of CPU
- packet copied to system's memory

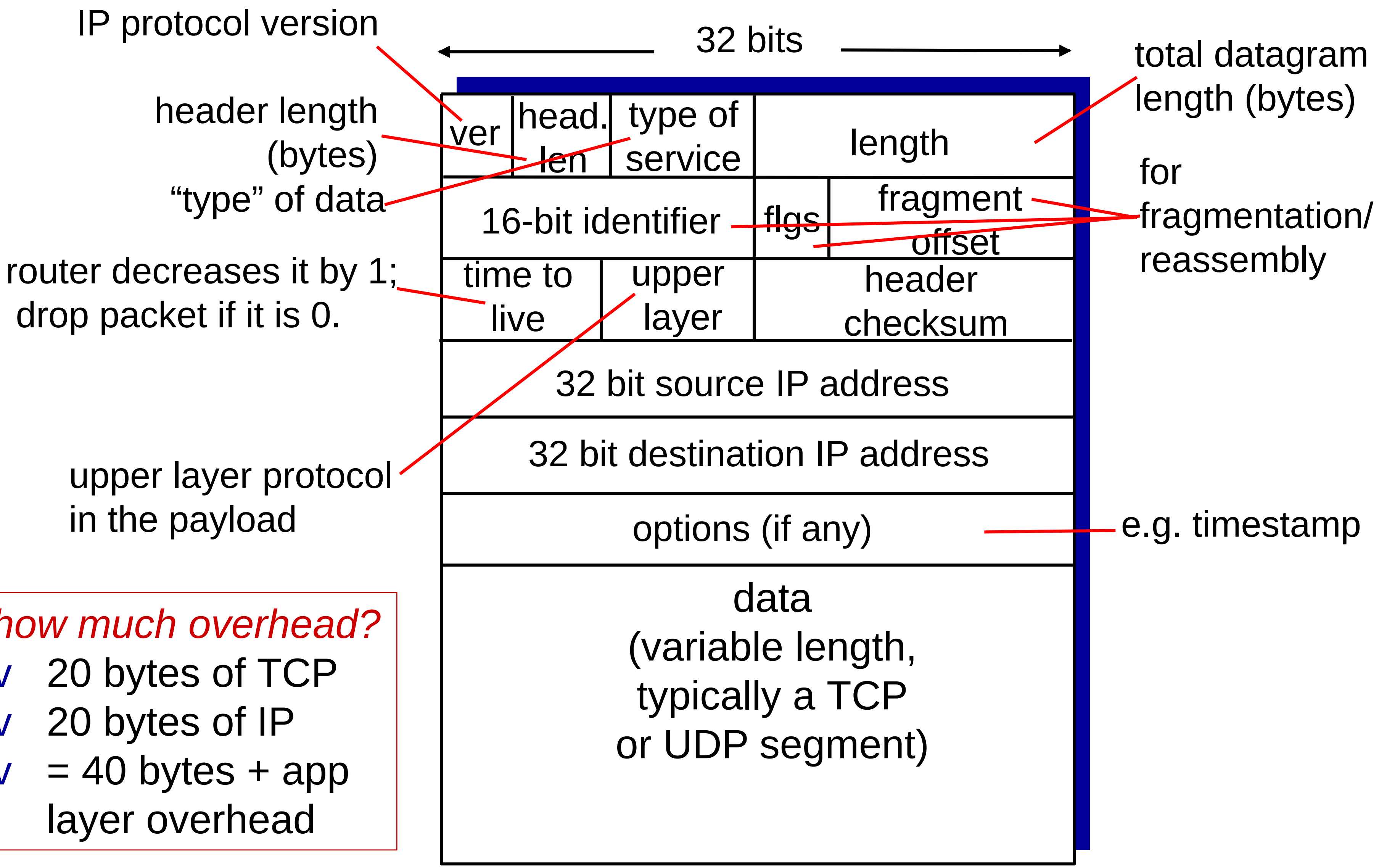


Output ports



- *buffering* (or a queue) required when datagrams via switching is faster than the outgoing transmission
- Packet will be dropped if the queue is full.

IP datagram format

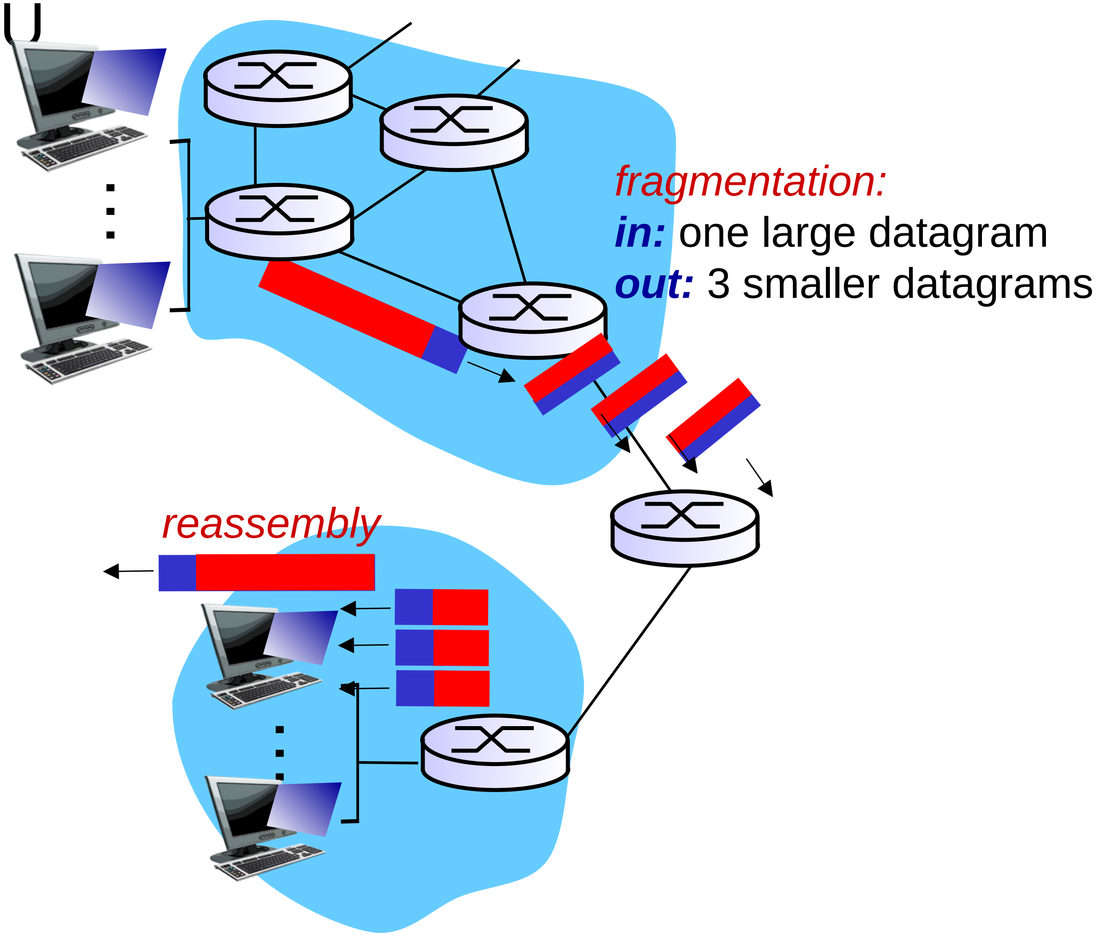


how much overhead?

- 20 bytes of TCP
- 20 bytes of IP
- = 40 bytes + app layer overhead

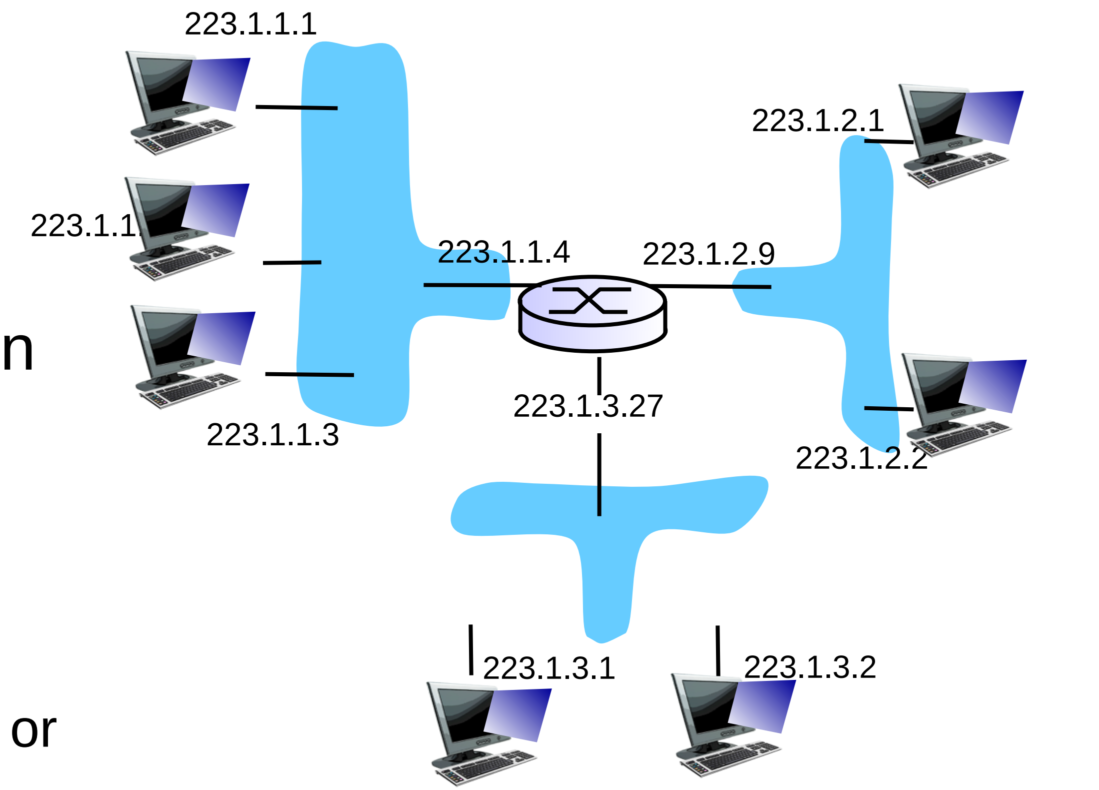
IP fragmentation, reassembly

- network links have MTU (max transfer size)
  - vary on link types
- large IP datagram divided ("fragmented") within net
  - one datagram becomes several datagrams
  - "reassembled" only at final destination
  - IP header bits used to identify, order related fragments



IP addressing: introduction

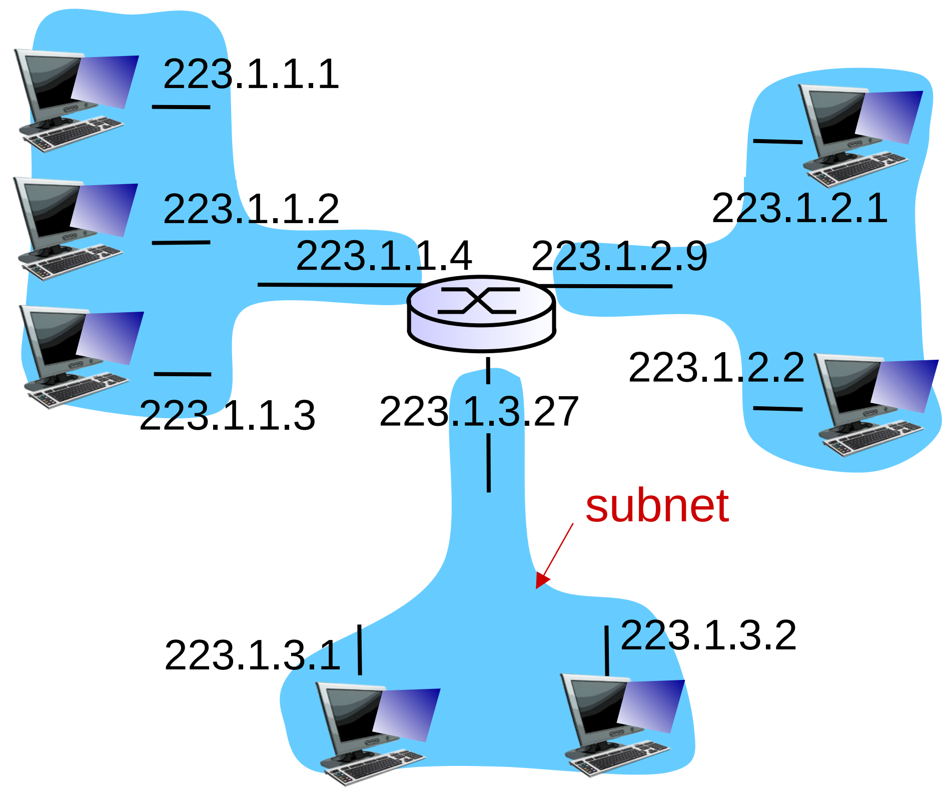
- *IP address:* 32-bit identifier for host, router *interface*
- *interface:* connection between host/router and physical channel
  - router's typically have multiple interfaces
  - host typically has one or two interfaces (e.g., wired Ethernet, wireless 802.11)
- *IP addresses associated with each interface*



$$223.1.1.1 = \underbrace{11011111}_{223}\ \underbrace{00000001}_{1}\ \underbrace{00000001}_{1}\ \underbrace{00000001}_{1}$$

Subnets

- IP address:
  - subnet part - high order bits
  - host part - low order bits
- *what's a subnet ?*
  - device interfaces with same subnet part of IP address
  - can physically reach each other *without intervening router*

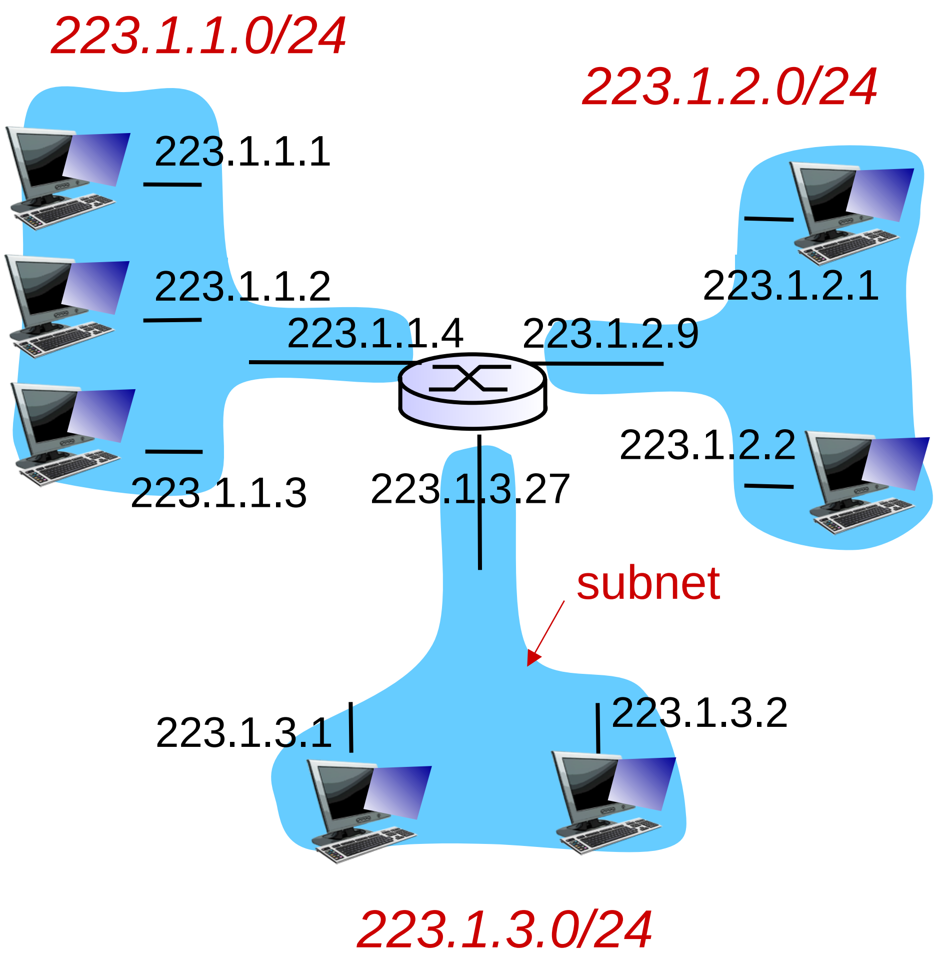


network consisting of 3 subnets

Subnets

recipe

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

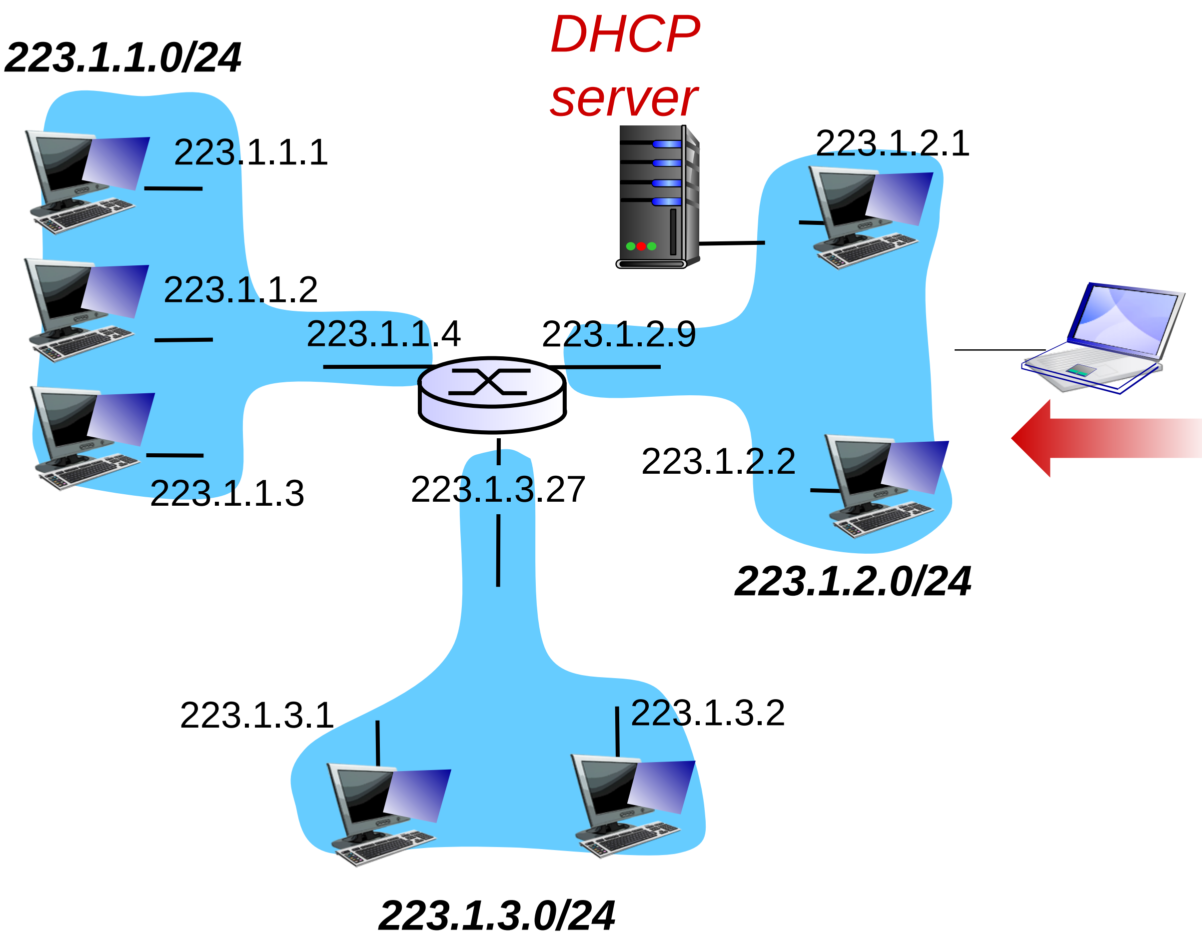


subnet mask: /24

DHCP: Dynamic Host Configuration Protocol

goal: allow host to *dynamically* obtain its IP address from network server when it joins network

DHCP client-server scenario



arriving DHCP client needs address in this network

DHCP client-server ~~scenario~~