Networking

CS 4410
Operating Systems

[R. Agarwal, L. Alvisi, A. Bracy, M. George, Kurose, Ross, E. Sirer, R. Van Renesse]

These slides are being posted far before the actual lecture dates and are therefore subject to tweaks until this message is removed.

Basic Network Abstraction

- A process can create “endpoints”
- Each endpoint has a unique address
- A message is a byte array
- Processes can:
  - receive messages on endpoints
  - send messages to endpoints

Network “protocol”

Agreement between processes about the content of messages

Syntax: Layout of bits, bytes, fields, etc.

– message format

Semantics: what fields, messages mean

Examples:

- HTTP “get” requests and responses
  - HTML is part of the format
- IRL: *Excuse me*, *please*, *thank you*, *etc.*

Network Layering

Network abstraction is usually *layered*

- Like Object Oriented-style inheritance
- Also like the hw/sw stack

Application
Presentation
Session
Transport
Network
Link
Physical

Proposed 7-Layer ISO/OSI reference model (1970's)

Application
Transport
Network
Link
Physical

Actual 5-Layer Internet Protocol Stack

OSI Layers

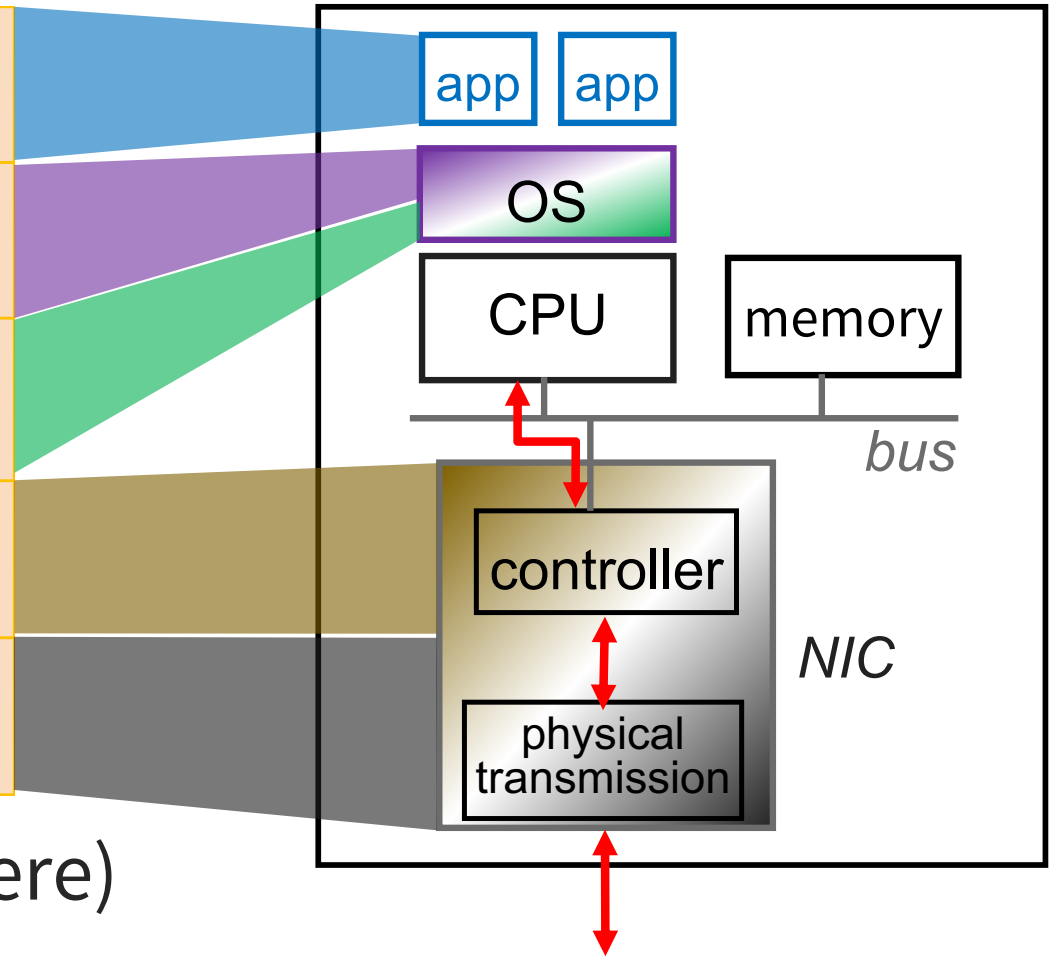
Application	Network-aware applications, clients & servers
Presentation	Translation between network and application formats (*e.g.*, RPC packages, sockets)
Session	Connection management
Transport	Data transfer, reliability, packetization, retransmission. Lets multiple apps share 1 network connection
Network	Path determination across multiple network segments, routing, logical addressing.
Link	Decides whose turn it is to talk, finds physical device on network.
Physical	Exchanges bits on the media (electrical, optical, *etc.*)

Internet Protocol Stack

Application	exchanges **messages**	HTTP, FTP, DNS
Transport	Transports messages; exchanges **segments**	TCP, UDP
Network	Transports segments; exchanges **datagrams**	IP, ICMP (ping)
Link	Transports datagrams; exchanges **frames**	Ethernet, WiFi
Physical	Transports frames; exchanges **bits**	wires, signal encoding

Who does what?

Application	HTTP, FTP, DNS (*these*^ are usually in libraries)
Transport	TCP, UDP
Network	IP, ICMP (ping)
Link	Ethernet, WiFi
Physical	wires, signal encoding



(Hard to draw firm lines here)

- Each host has 1+ Network Interface Cards (NIC)
- Attaches into host's system buses
- Combination of hardware, software, firmware

Layers support **Modularity**

Each layer:

- relies on services from layer below
- exports services to layer above

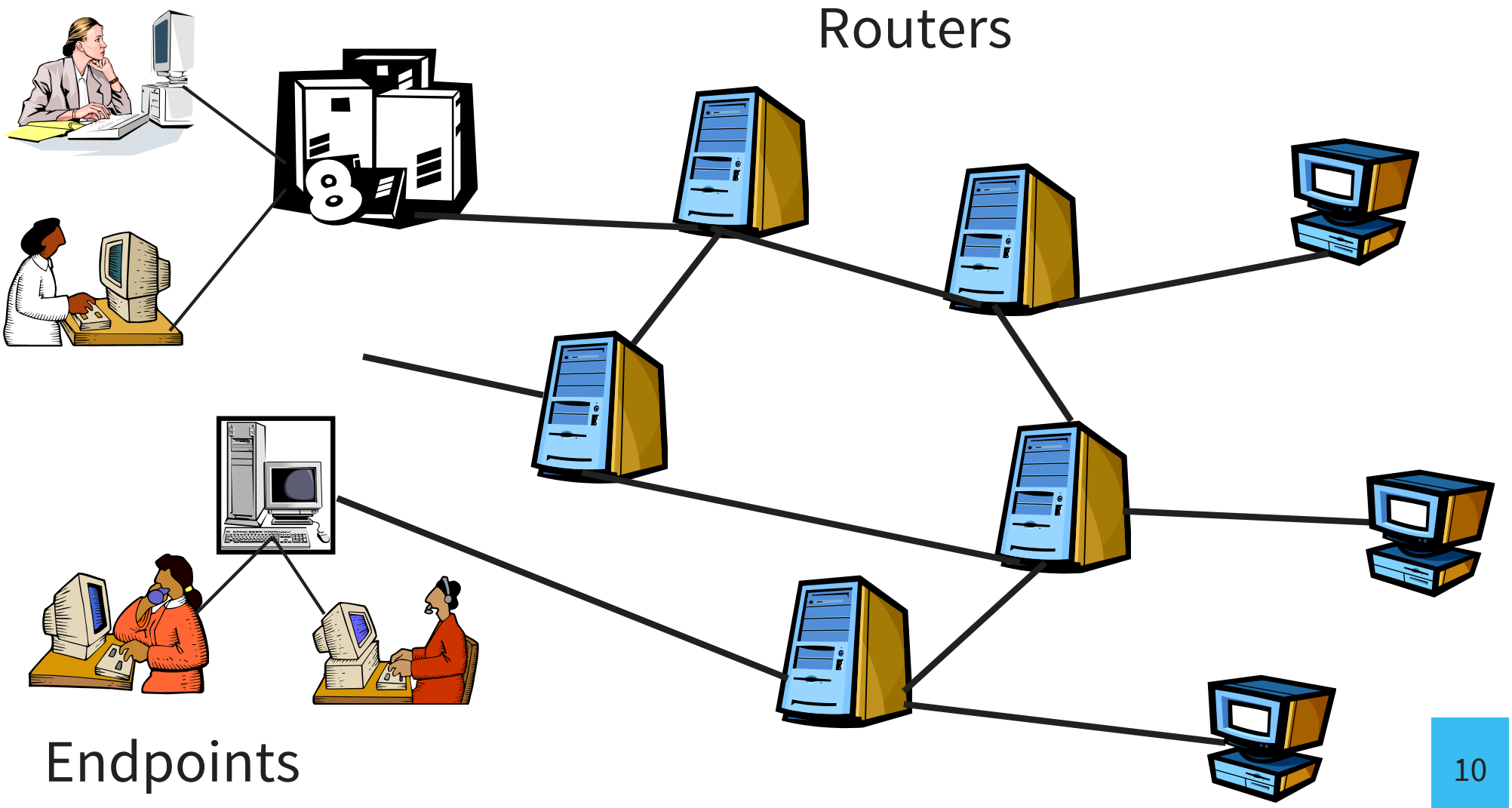
Can identify the relationship between distinct pieces of complex system.

Interfaces between layers:

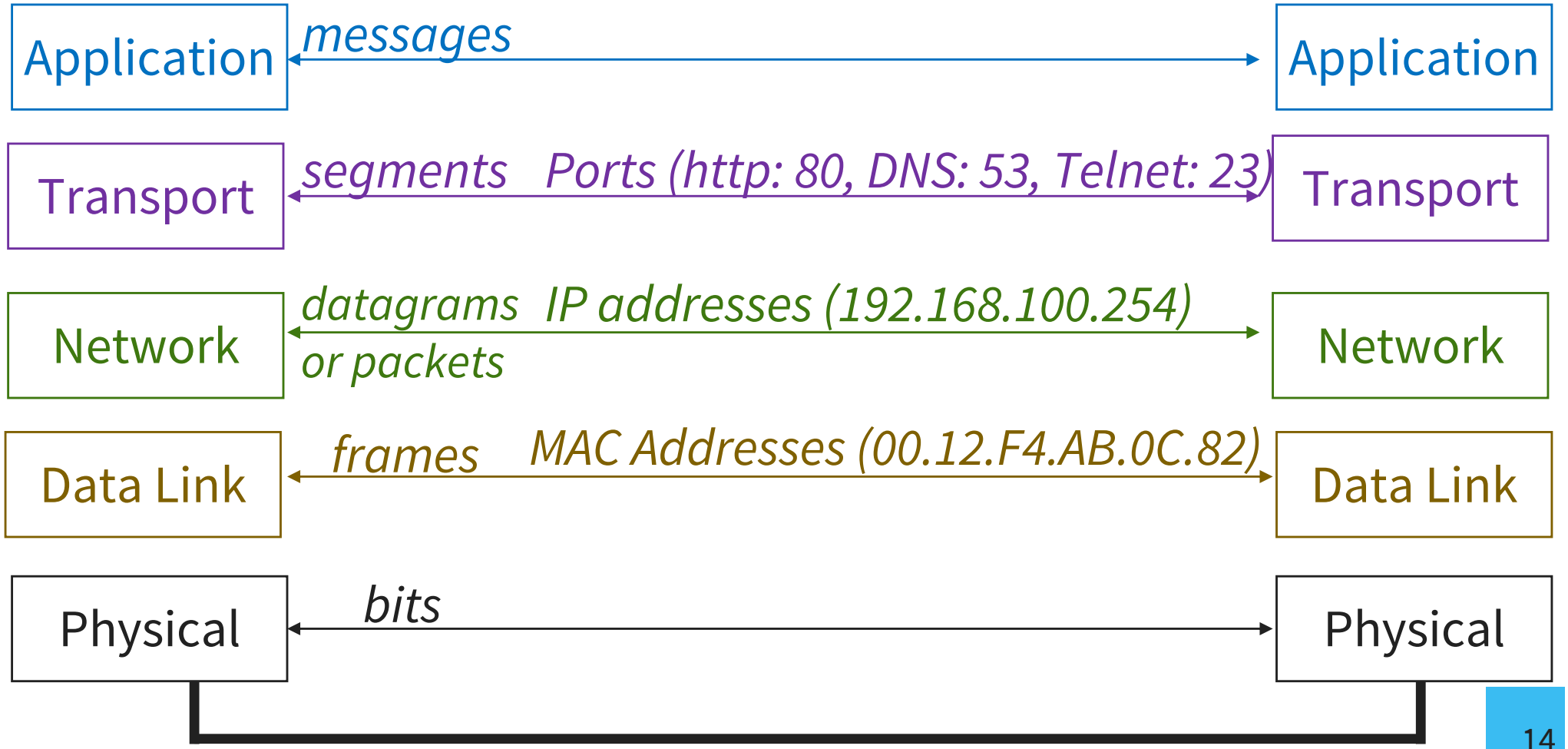
- Hide implementation details
- Ease maintenance, updates
  - change of implementation of layer's service transparent to rest of system

Internet, The Big Picture

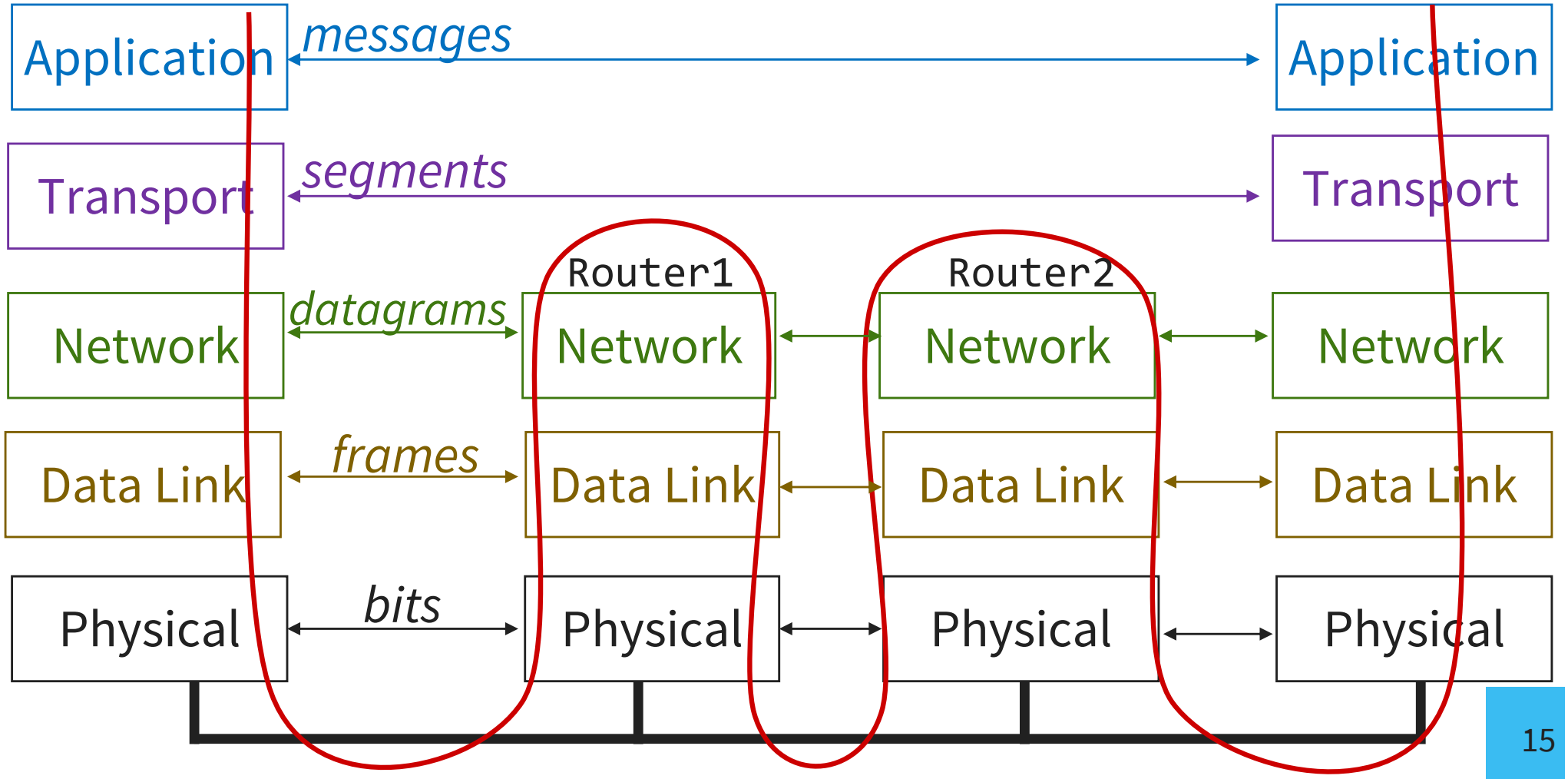
How about an analogy?



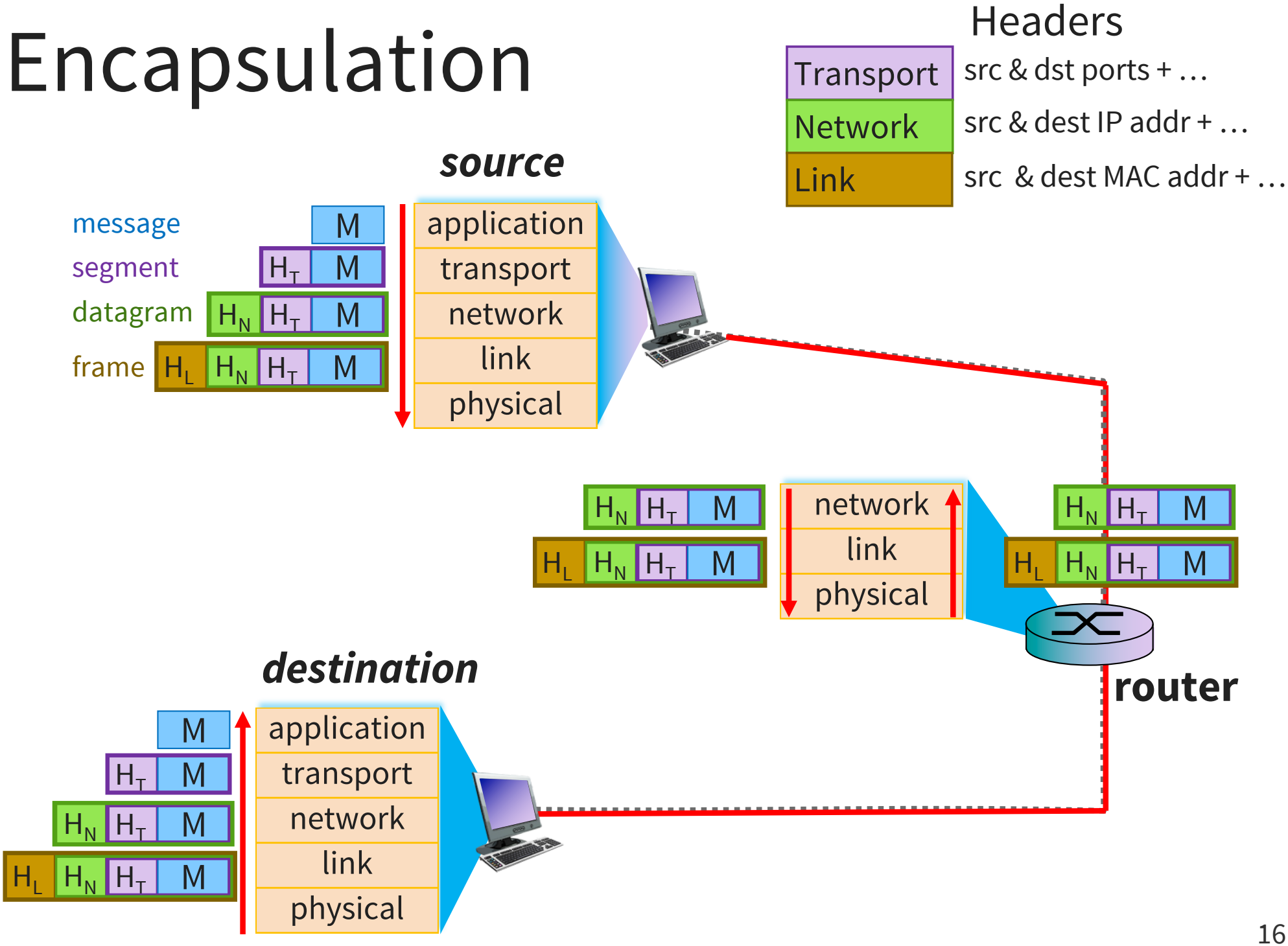
The Big Picture



The Big Picture



Encapsulation



End-to-End Argument

- Occam's Razor for Internet architecture
- Application-specific properties are best provided by the applications, **not the network**
  - Guaranteed, or ordered, packet delivery, duplicate suppression, security, *etc.*

- Internet performs the simplest packet routing and delivery service it can
  - Packets are sent on a best-effort basis
  - Higher-level applications do the rest

End-to-End Example

Should the network guarantee packet delivery?

Consider: a file transfer program (read file from disk, send it, receiver reads packets & writes them to disk)

- **Q:** If network guarantees delivery, wouldn't applications be simpler? (no retransmissions!)
- **A:** no, still need to check that file was written to remote disk intact

A check is necessary if nodes can fail.

→ Applications need to be written to perform their own retransmits

Why burden the network with properties that can, and must, be implemented at the periphery?

Application
Transport
Network
Link
Physical

Application Layer

Several figures in this section come from
"Computer Networking: A Top Down Approach"
by Jim Kurose, Keith Ross

Naming

People

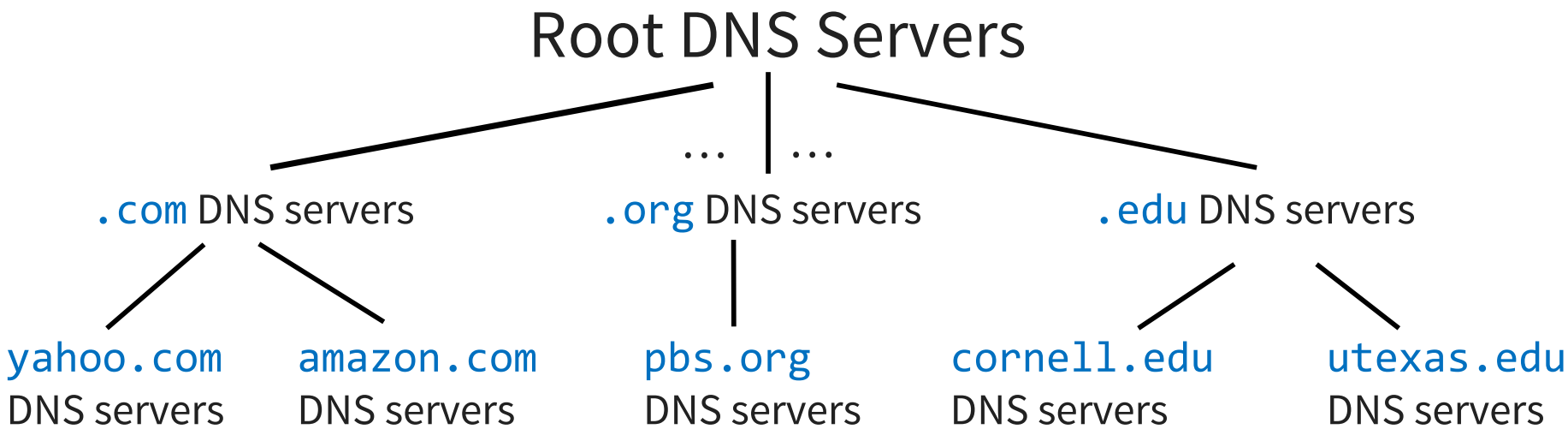
- SSN, NetID, Passport #

Internet Hosts, Routers

1. IP address (32 bit), `151.101.117.67`
   - For now, 32-bit descriptor, like a phone number
   - Longer addresses in the works…
   - Assigned to hosts by their internet service providers
   - **Not physical:** does not identify a single node, can swap machines and reuse the same IP address
   - **Not entirely virtual:** determines how packets get to you, changes when you change your ISP
2. Virtual: "name" `www.cnn.com`
   - Used by humans (no one wants to remember a bunch of #s)

How to convert hostname to IP address?

Domain Name System (DNS)



Distributed, Hierarchical Database

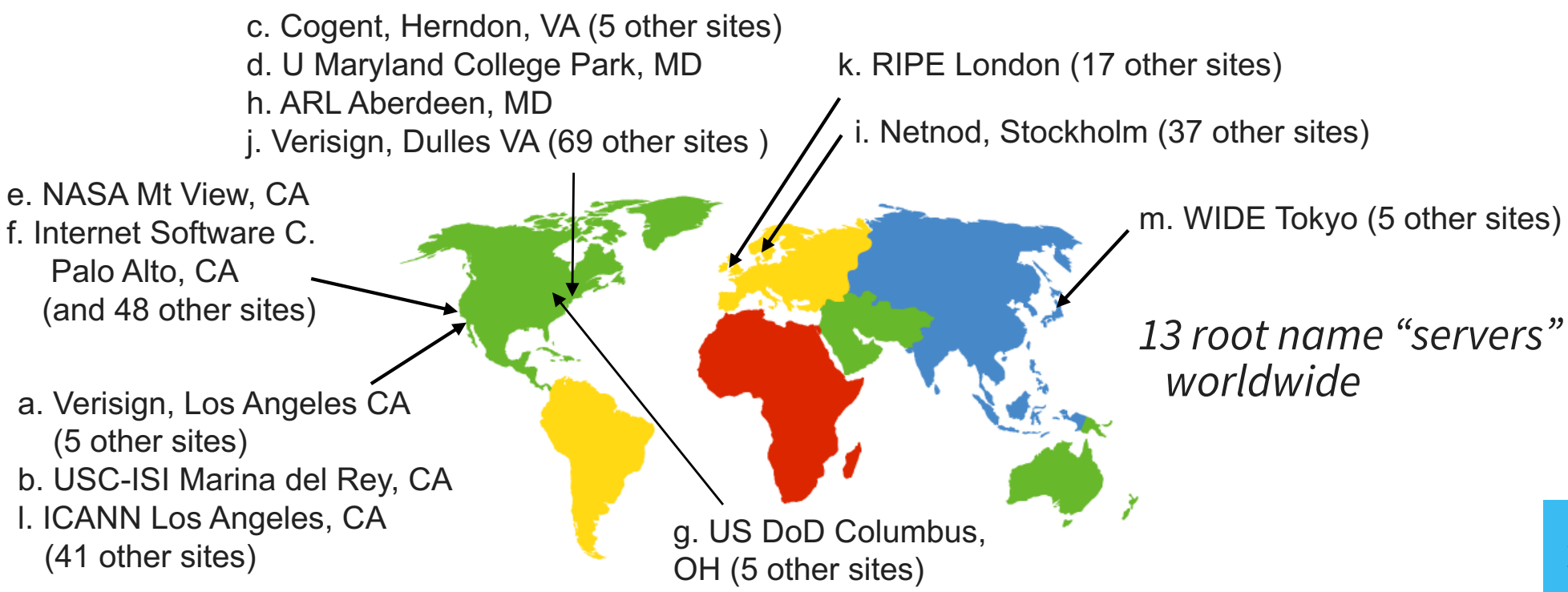
- Application-Layer Protocol: hosts & name servers communicate to resolve names
- Names are separated by dots into components

  Not to be confused with dots in IP addresses (in which the order of least significant to most significant is reversed)
- Components are looked up from the right to the left
- All siblings must have unique names
- Lookup occurs from the top down

DNS: root name servers

Contacted by local name server that can not resolve name

- owned by Internet Corporation for Assigned Names & Numbers (ICANN)
- contacts authoritative name server if name mapping not known
- gets mapping
- returns mapping to local name server



DNS Lookup

1. the client asks its local nameserver
2. the local nameserver asks one of the *root nameservers*
3. the root nameserver replies with the address of the authoritative nameserver
4. the server then queries that nameserver
5. repeat until host is reached, cache result.

Example: Client wants IP addr of www.amazon.com

1. Queries root server to find com DNS server
2. Queries `.com` DNS server to get `amazon.com` DNS server
3. Queries `amazon.com` DNS server to get IP address for `www.amazon.com`

DNS Services

Simple, hierarchical namespace works well

- Can name anything
- Can alias hosts
- Can cache results
- Can share names (replicate web servers by having 1 name correspond to many IP addresses)

Q: Why not centralize?

- Single point of failure
- Traffic volume
- Distant Centralized Database
- Maintenance

A: Does not scale!

What about security? (don't ask!)

Application Layer

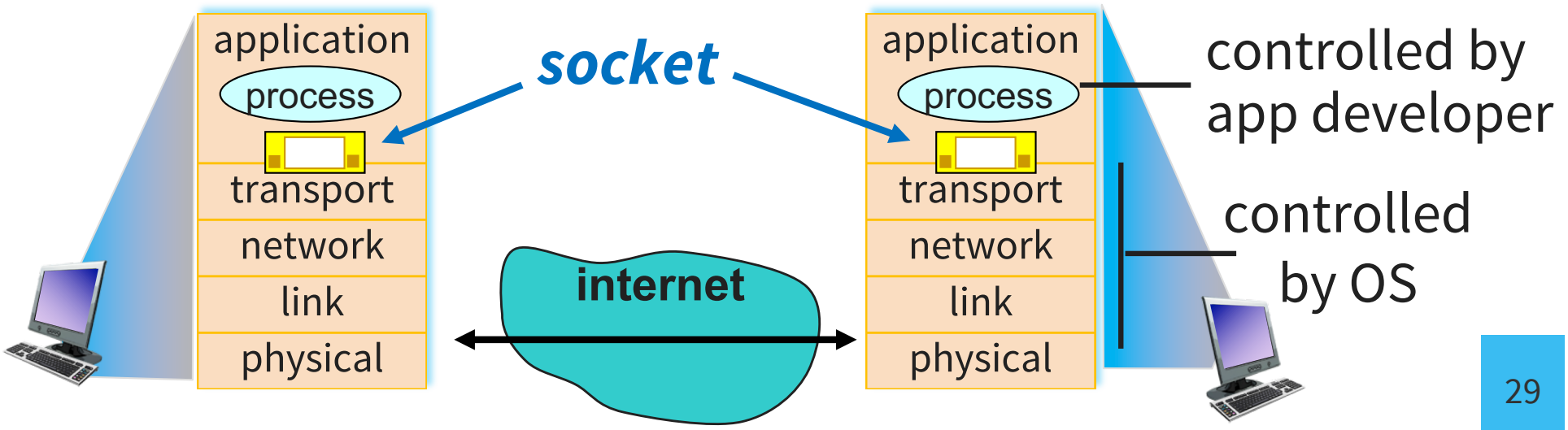
- Network-aware applications
  - Clients & Servers
  - Peer-to-Peer

Sockets

"Door" between application process and end-end-transport protocol

Sending process:

- shoves message out door
- relies on transport infrastructure on other side of door to deliver message to socket at receiving process



Socket programming

Two socket types for two transport services:

- UDP: unreliable datagram
- TCP: reliable, byte stream-oriented

Host could be running many network applications at once. Distinguish them by binding the socket to a **port number**:

- 16 bit unsigned number
- 0-1023 are well-known
  (web server = 80, mail = 25, telnet = 23)
- the rest are up for grabs (see A3)

Application Example

1. Client reads a line of characters (data) from its keyboard and sends data to server
2. Server receives the data and converts characters to uppercase
3. Server sends modified data to client
4. Client receives modified data and displays line on its screen

Socket programming with UDP

No "connection" between client & server

- no handshaking before sending data
- **Sender:** explicitly attaches destination IP address & port # to each packet
- **Receiver:** extracts sender IP address and port # from received packet

Data may be lost, received out-of-order

Application viewpoint: UDP provides unreliable transfer of groups of bytes ("datagrams") between client and server

Client/server socket interaction: UDP

Server (running on serverIP)

create serversocket, bind to port x

read data (and clientAddr) from serversocket

modify data

send modified data to clientAddr via serversocket

Client

create clientsocket

create message

send message to (serverIP, port x) via clientsocket

receive message (and serverAddr) from clientsocket

close clientsocket

Python UDP Client

```
import socket     #include Python’s socket library
serverName = ‘servername’
serverPort = 12000

#create UPD socket
clientSocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

#get user input
message = input('Input lowercase sentence: ‘)

# send with server name + port
clientSocket.sendto(message.encode(), (serverName, serverPort))

# get reply from socket and print it
modifiedMessage, serverAddress = clientSocket.recvfrom(2048)
print(modifiedMessage.decode())

clientSocket.close()
```

Python UDP Server

```
import socket   #include Python’s socket library
serverPort = 12000

#create UPD socket & bind to local port 12000
serverSocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
serverSocket.bind(('', serverPort))
print("The server is ready to receive")

while True:
    # Read from serverSocket into message,
    # getting client’s address (client IP and port)
    message, clientAddress = serverSocket.recvfrom(2048)
    print("received message: "+message.decode())
    modifiedMsg = message.decode().upper()
    print("sending back to client")

    # send uppercase string back to client
    serverSocket.sendto(modifiedMsg.encode(), clientAddress)
```

Socket programming w/ TCP (**A3**)

Client must contact server

Server:

- already running
- server already created "welcoming socket"

Client:

- Creates TCP socket w/ IP address, port # of server
- Client TCP establishes connection to server TCP

- when contacted by client, *server TCP creates new socket* to communicate with that particular client
  - allows server to talk with multiple clients
  - source port #s used to distinguish clients

Application viewpoint: TCP provides reliable, in-order byte-stream transfer between client & server

Client/server socket interaction: TCP

Server (running on hostID)

create welcoming serversocket,
bind to port x

in response to connection request,
create connectionsocket

read data from connectionsocket

modify data

send modified data to clientAddr
via connectionsocket

close connectionsocket

Client

create clientsocket
connect to (hostID, port x)

create message

send message via clientsocket

receive message from clientsocket

close clientsocket

Python TCP Client

```
import socket     #include Python’s socket library
serverName = ‘servername’
serverPort = 12000

#create TCP socket for server on port 12000
clientSocket = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
clientSocket.connect((serverName,serverPort))

#get user input
message = input('Input lowercase sentence: ‘)

# send (no need for server name + port)
clientSocket.send(message.encode())

# get reply from socket and print it
modifiedMessage, serverAddress = clientSocket.recvfrom(1024)
print(modifiedMessage.decode())

clientSocket.close()
```

Python TCP Server

```
import socket   #include Python’s socket library
serverPort = 12000

#create TCP welcoming socket & bind to server port 12000
serverSocket = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
serverSocket.bind(('', serverPort))
#server begins listening for incoming TCP requests
serverSocket.listen(1)
print("The server is ready to receive")

while True:
    # server waits on accept() for incoming requests
    # new socket created on return
    connectionSocket, addr = serverSocket.accept()
    message = connectionSocket.recv(1024).decode()
    print("received message: "+message)
    modifiedMsg = message.upper()

     # send uppercase string back to client
    connectionSocket.send(modifiedMsg.encode())

    # close connection to this client, but not welcoming socket
    connectionSocket.close()
```

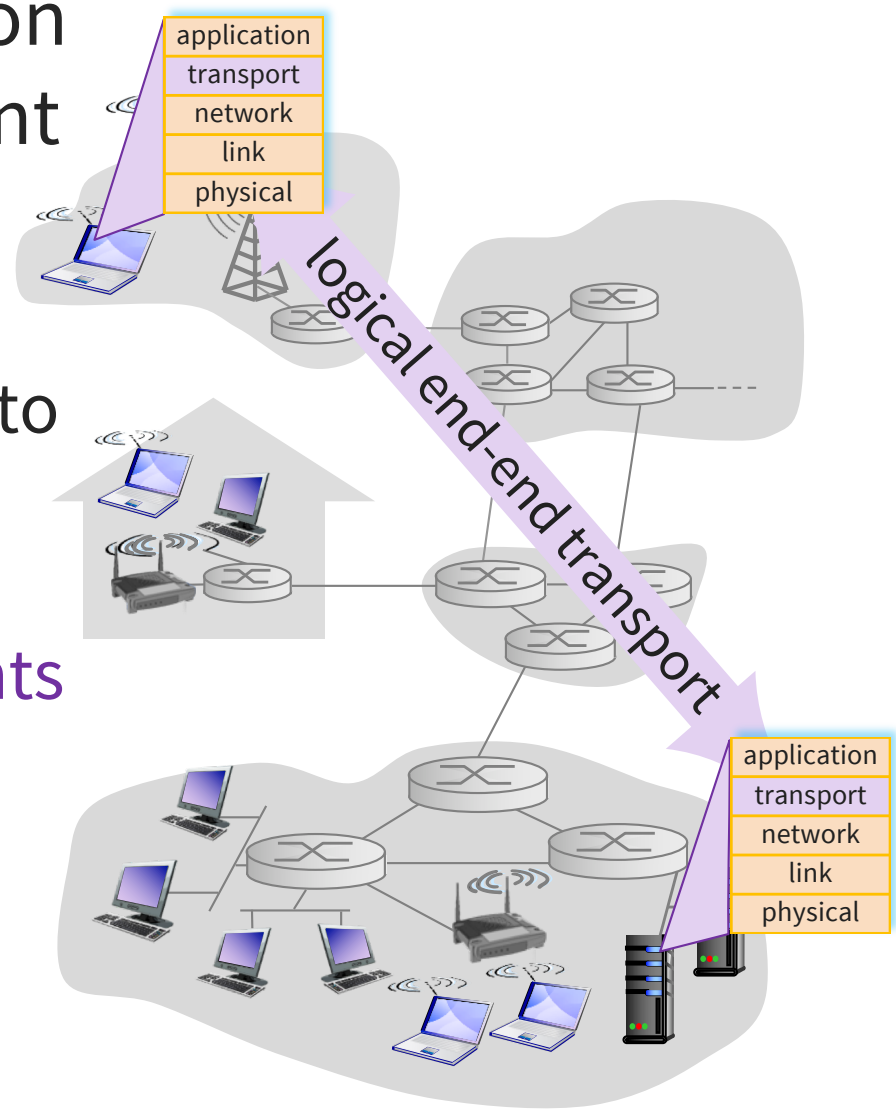
Application
Transport
Network
Link
Physical

Transport Layer: UDP & TCP

Several figures in this section come from
"Computer Networking: A Top Down Approach"
by Jim Kurose, Keith Ross

Transport services and protocols

- Provide **logical** communication between processes on different hosts
- Run in end systems
  - **Sender:** packages messages into segments, passes to network layer
  - **Receiver:** reassembles segments into messages, passes to application layer



App chooses protocol it wants (*e.g.*, TCP or UDP)

Transport services and protocols

User Datagram Protocol (UDP)

- **unreliable, unordered delivery**
- no-frills extension of best-effort IP

"**Unreliable** Datagram Protocol"

Transmission Control Protocol (TCP)

- **reliable, in-order delivery**
- congestion control
- flow control
- connection setup

"**Trusty** Control Protocol"

Services **not** available:

- delay guarantees
- bandwidth guarantees

How to create a segment

Sending application:

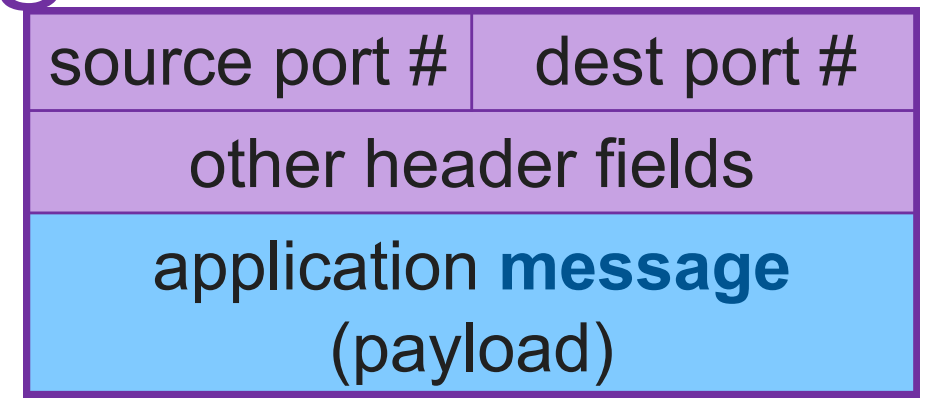
- specifies IP address and destination port
- uses socket bound to a source port

Transport Layer:

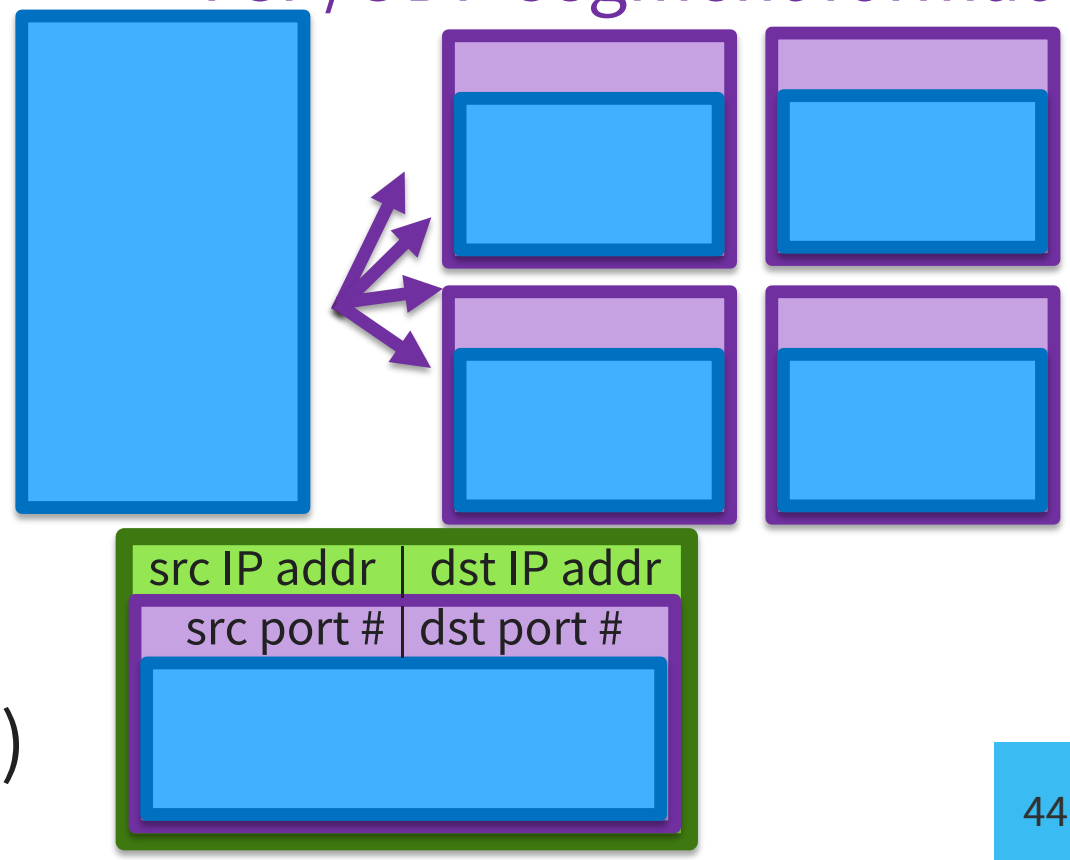
- breaks application message into smaller chunks
- adds transport-layer header to each

Network Layer:

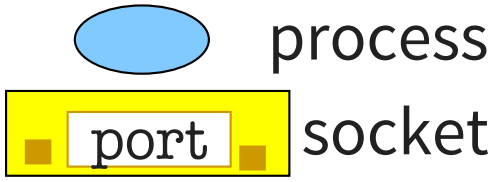
- adds network-layer header (with IP address)



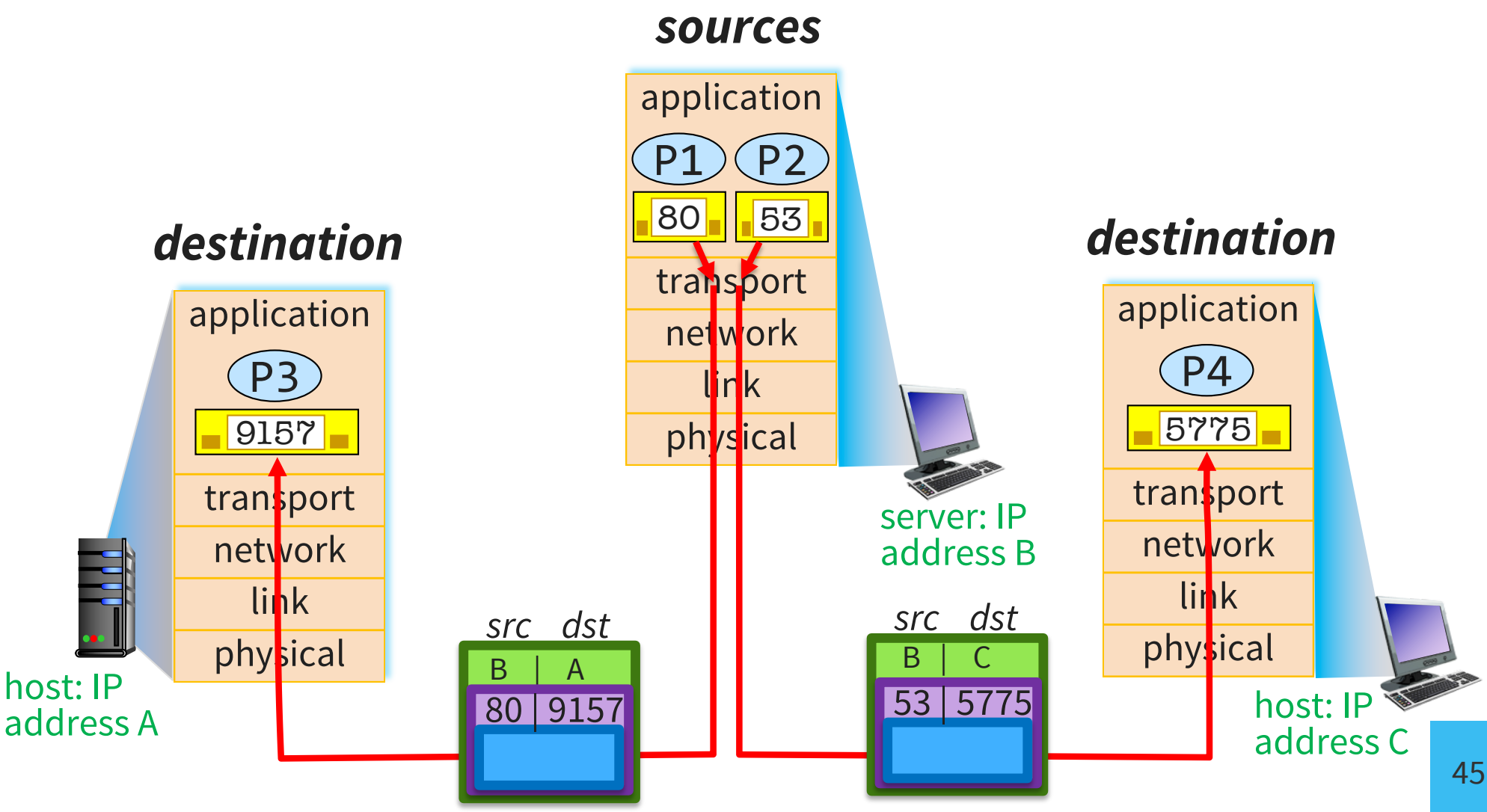
TCP/UDP segment format



Multiplexing at Sender



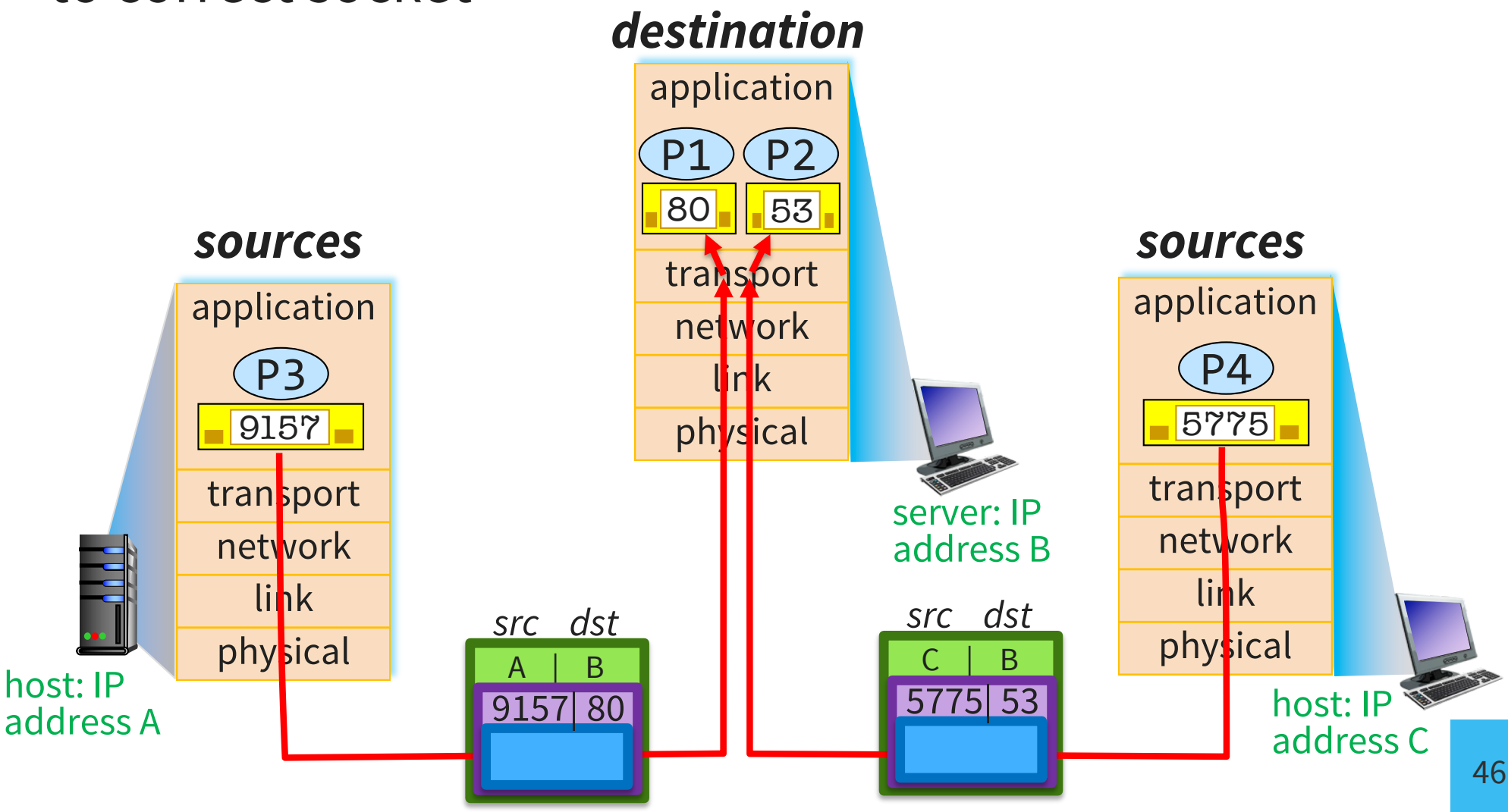
- handles data from multiple sockets
- adds transport header (later used for demultiplexing)



Demultiplexing at Receiver

process
socket

- use header information to deliver received segments to correct socket



User Datagram Protocol (UDP)

- no frills, bare bones transport protocol
- **best effort** service, UDP segments may be:
  - lost
  - delivered out-of-order, duplicated to app
- **connectionless:**
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others
- reliable transfer still possible:
  - **add reliability at application layer**
  - application-specific error recovery!

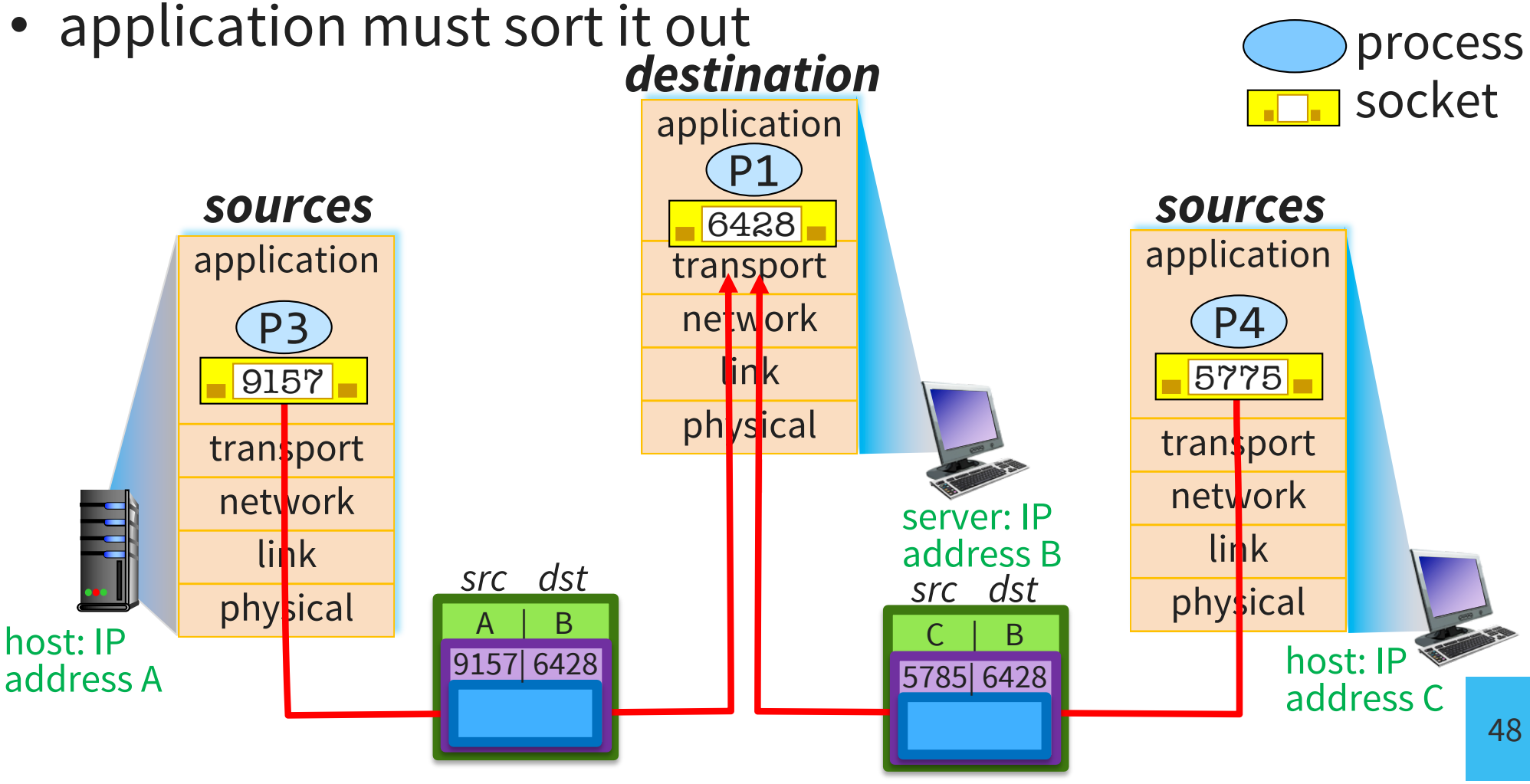
I was gonna tell you guys a joke about UDP…
But you might not get it

I was you guys about UDP might not

Connectionless demux: example

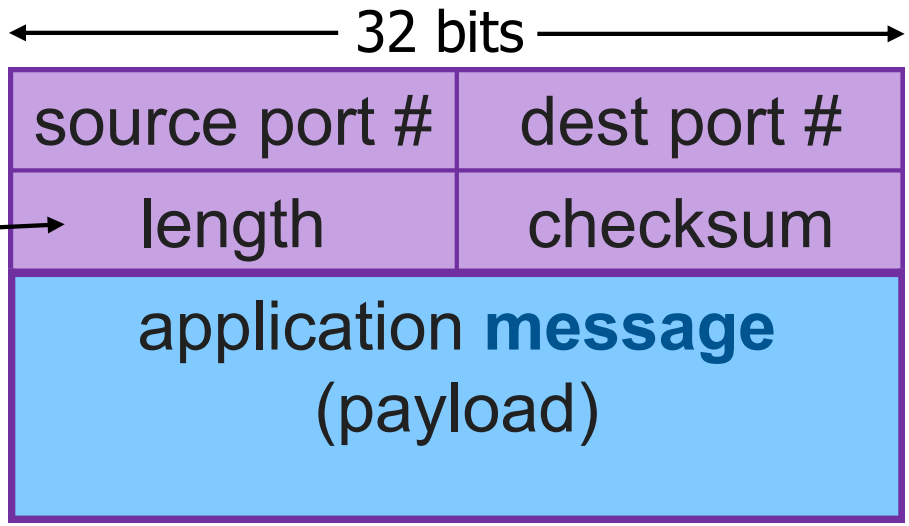
Host receives 2 UDP segments:

- checks dst port, directs segment to socket w/that port
- *different src IP or port* but *same dst port* → *same socket*
- application must sort it out



UDP Segment Format

length (in bytes) of UDP segment, including header



UDP header size: 8 bytes

(IP address will be added when the segment is turned into a datagram/packet at the Network Layer)

UDP Advantages & Disadvantages

Speed:

- no connection establishment (which can add delay)
- no congestion control: UDP can blast away as fast as desired

Simplicity:

- no connection state at sender, receiver
- small header size (8 bytes)

(Possibly) **Extra work for applications**:
Need to handle reordering, duplicate suppression, missing packets

Not all applications will care about these!

Who uses UDP?

Target Users: streaming multimedia apps

- loss tolerant *(occasional packet drop OK)*
- rate sensitive *(want constant, fast speeds)*

UDP is good to build on

Applications & their transport protocols

Application	Application-Layer Protocol	Underlying Transport Protocol
Electronic mail	SMTP	TCP
Remote terminal access	Telnet	TCP
Web	HTTP	TCP
File transfer	FTP	TCP
Remote file server	NFS	Typically UDP
Streaming multimedia	typically proprietary	UDP or TCP
Internet telephony	typically proprietary	UDP or TCP
Network management	SNMP	Typically UDP
Routing protocol	RIP	Typically UDP
Name translation	DNS	Typically UDP

Transmission Control Protocol (TCP)

- Reliable, ordered communication
- Standard, adaptive protocol that delivers good-enough performance and deals well with congestion
- All web traffic travels over TCP/IP

- Why? enough applications demand reliable ordered delivery that they should not have to implement their own protocol

TCP Segment Format

HL: header len

U: urgent data

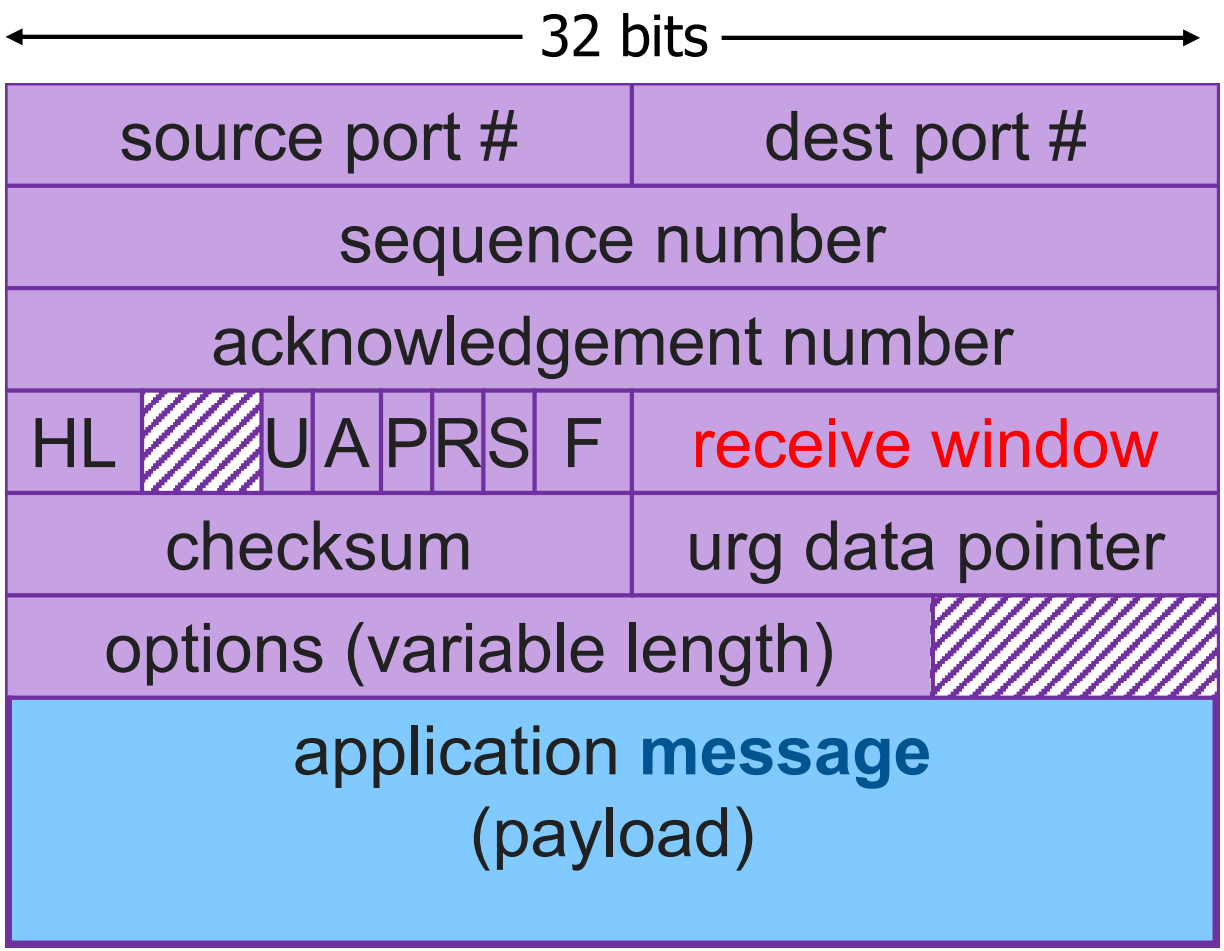
A: ACK # valid

P: push data now

RST, SYN, FIN: connection commands (setup, teardown)

bytes receiver willing to accept

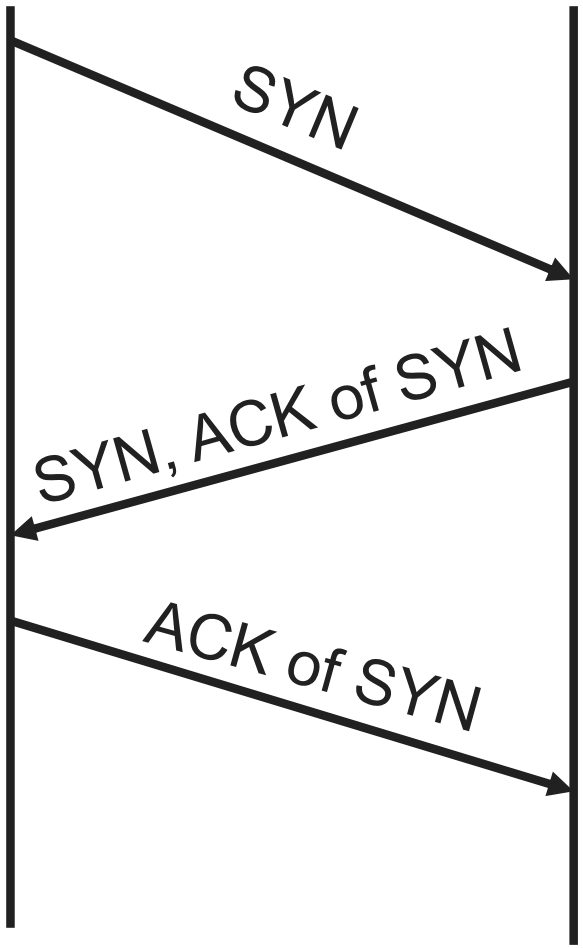
generally not used



TCP header size: 20-60 bytes

(IP address will be added when the segment is turned into a datagram/packet at the Network Layer)

TCP Connections



- TCP is *connection* oriented
- A connection is initiated with a three-way handshake
- Three-way handshake ensures against duplicate SYN packets
- Takes 3 packets, 1.5 RTT (**R**ound **T**rip **T**ime)

SYN = Synchronize
ACK = Acknowledgement

I would tell you a joke about TCP... If only to be acknowledged

TCP Handshakes

3-way handshake establishes common state on both sides of a connection.

Both sides will:

- have seen one packet from the other side
  → know what the first seq# ought to be
- know that the other side is ready to receive

Server will typically create a new socket for the client upon connection.

TCP Sockets

Server host may support many simultaneous TCP sockets

Each socket identified by its own 4-tuple

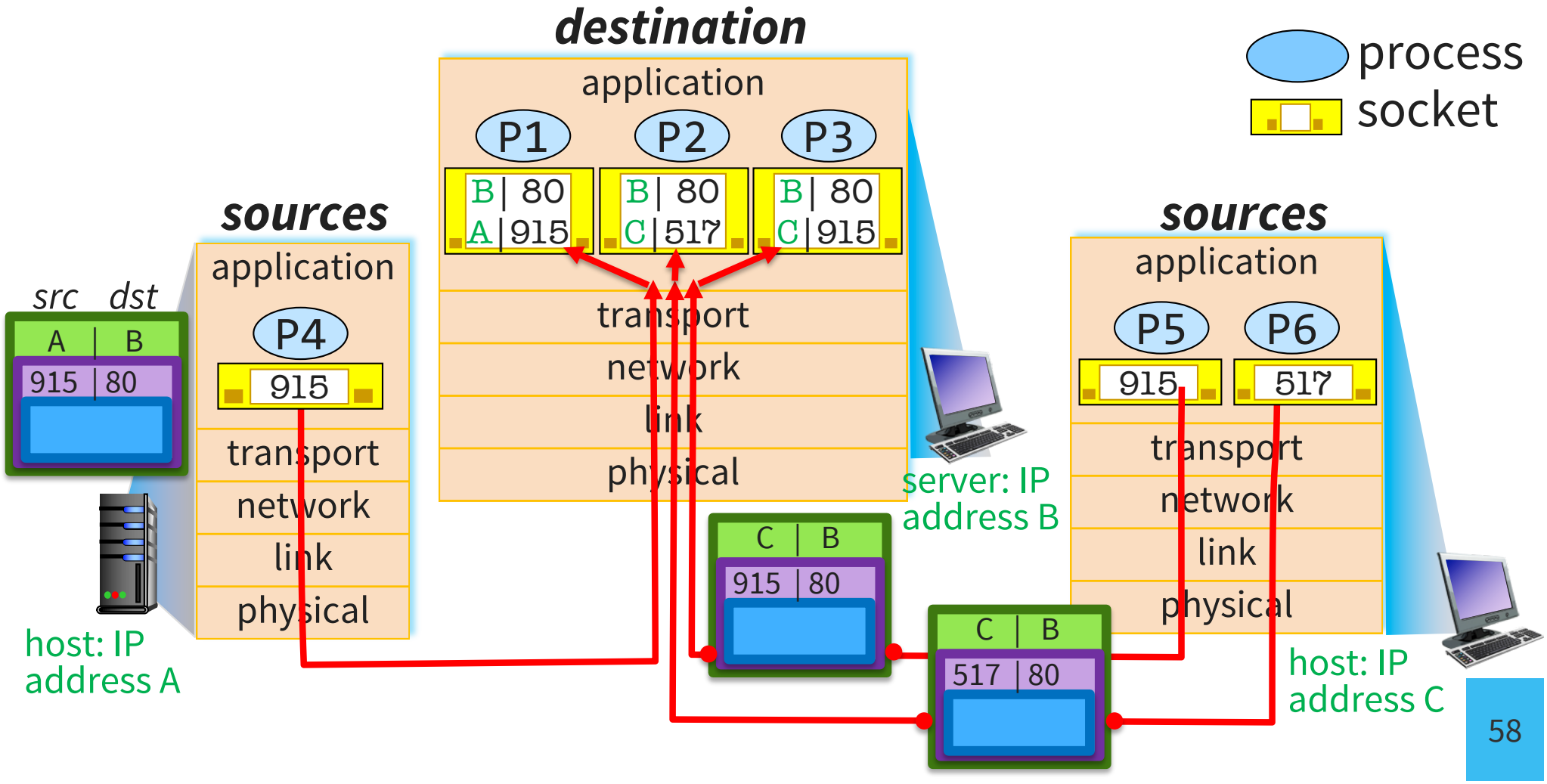
- source IP address
- source port number
- dest IP address
- dest port number

Connection-oriented demux: receiver uses all 4 values to direct segment to appropriate socket

Connection-oriented demux: example

Host receives 3 TCP segments:

- all destined to IP addr B, port 80
- demuxed to different sockets with socket's 4-tuple



TCP Packets

Each packet carries a unique sequence #

- The initial number is chosen randomly
- The SEQ is incremented by the data length

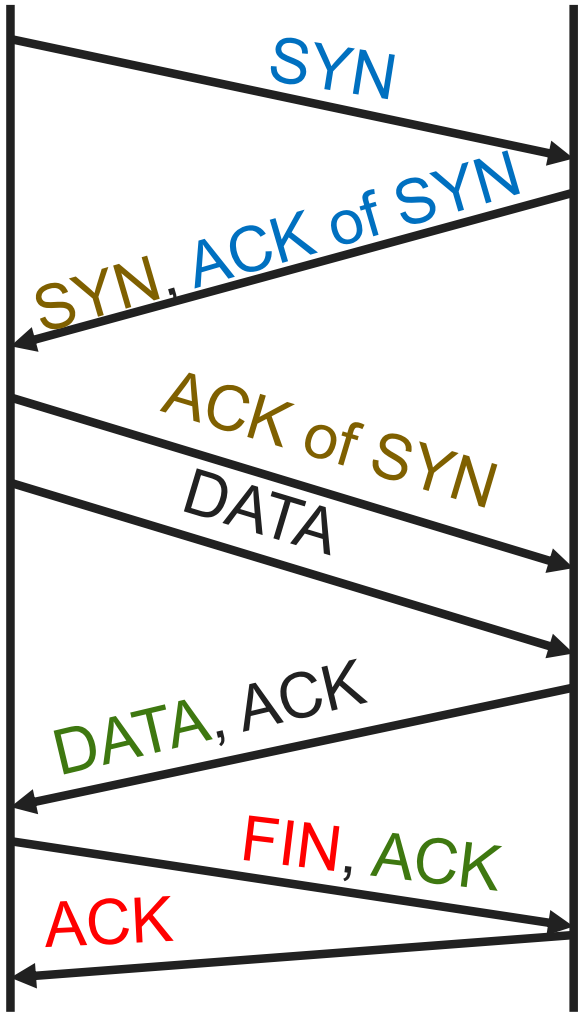
  4410 simplification: just increment by 1

Each packet carries an **ack**nowledgement

- Acknowledge a set of packets by ACK-ing the latest SEQ received

Reliable transport is implemented using these identifiers

TCP Usage Pattern



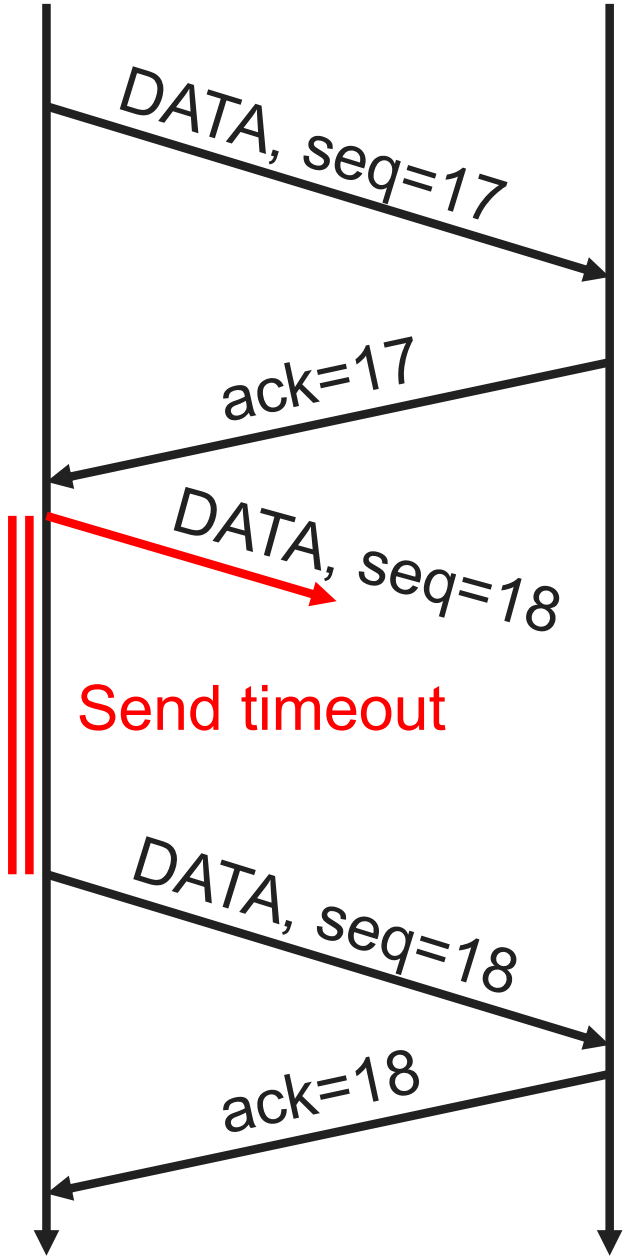
3 round-trips:

1. set up a connection
2. send data & receive a response
3. tear down connection

FINs work (mostly) like SYNs to tear down connection

Need to wait after a FIN for straggling packets

Reliable transport



- Sender-side: TCP keeps a copy of all sent, but unacknowledged packets
- If acknowledgement does not arrive within a “send timeout” period, packet is resent
- Send timeout adjusts to the round-trip delay

Here's a joke about TCP.
Did you get it?
Did you get it?
Did you get it?
Did you get it?

TCP timeouts

What is a good timeout period ?

- Goal: improve throughput without unnecessary transmissions

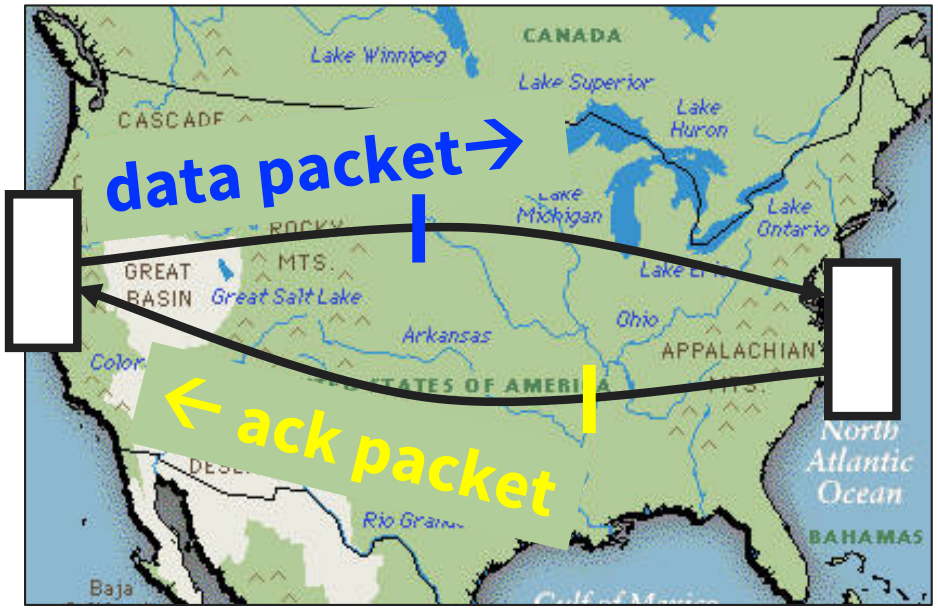
```
NewAverageRTT = (1 - α) OldAverageRTT + α LatestRTT
NewAverageVar = (1 - β) OldAverageVar + β LatestVar
where LatestRTT = (ack_receive_time – send_time),
            LatestVar  = |LatestRTT – AverageRTT|,
            α = 1/8, β = ¼ typically.
Timeout = AverageRTT + 4*AverageVar
```

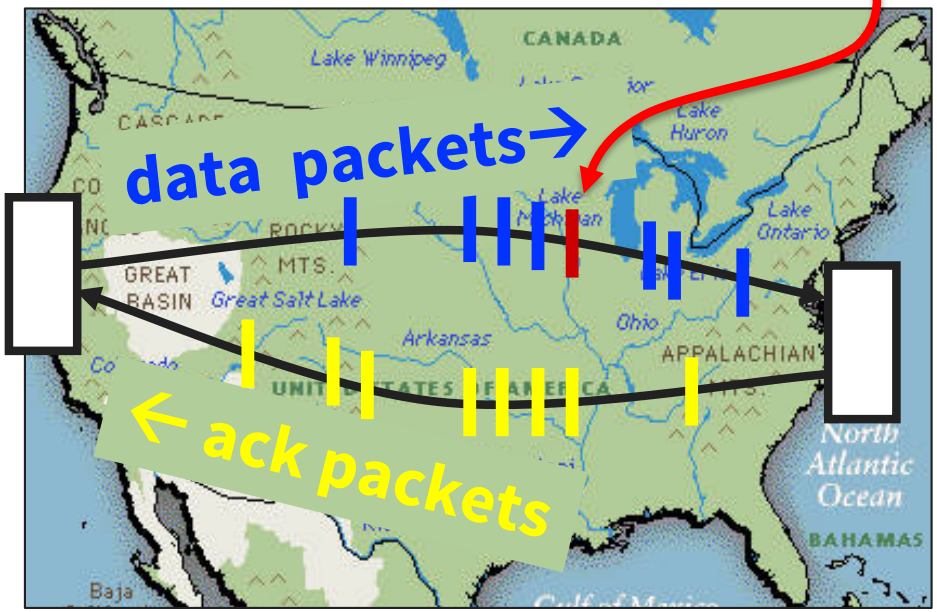
→ Timeout is a function of RTT and variance

Pipelined Protocols

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

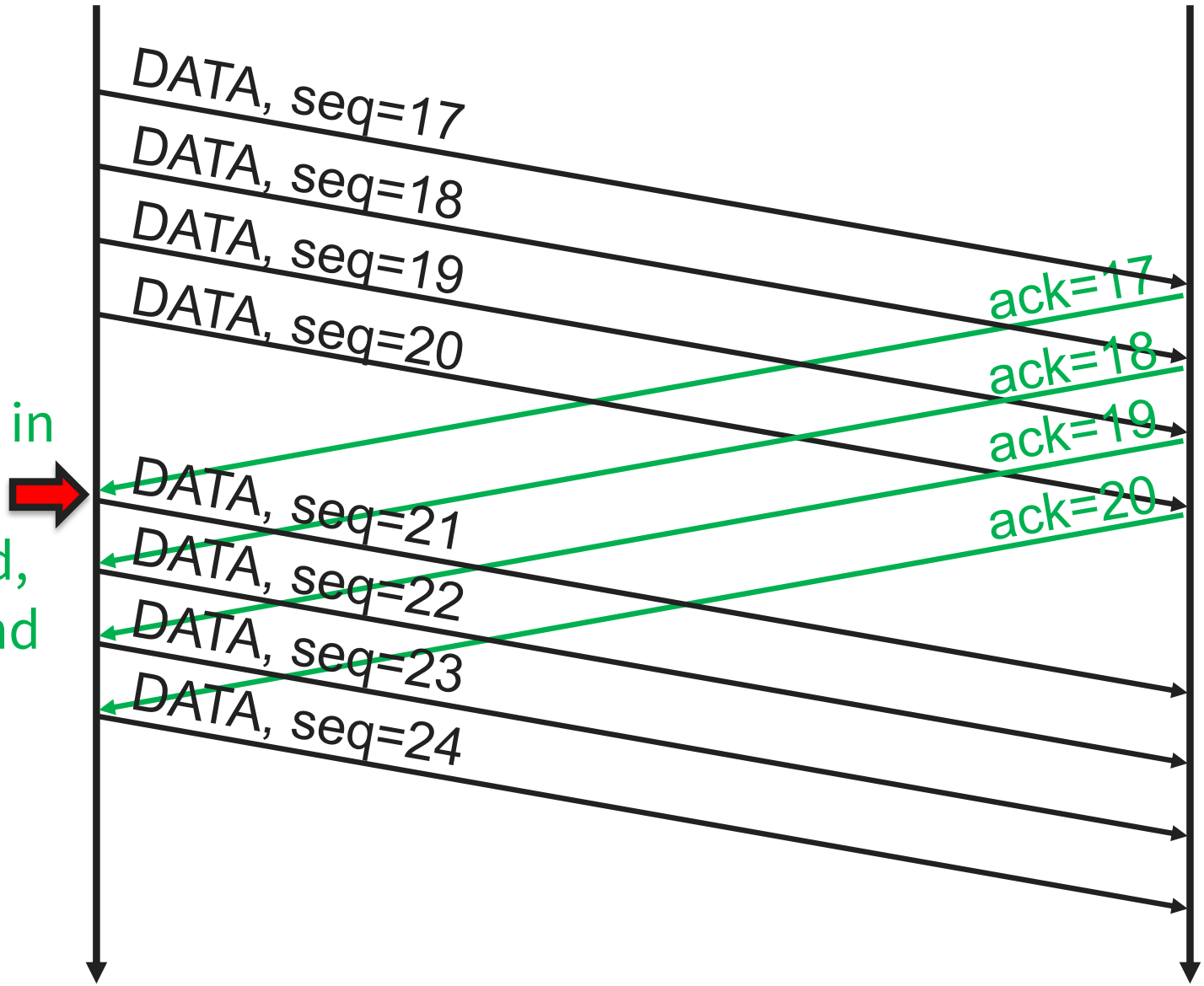
- increases throughput
- need buffering at sender and receiver
- How big should the window be?
- What if a packet in the middle goes missing?





Example: TCP Window Size = 4

When first item in window is acknowledged, sender can send the 5th item.



How much data “fits” in a pipe?

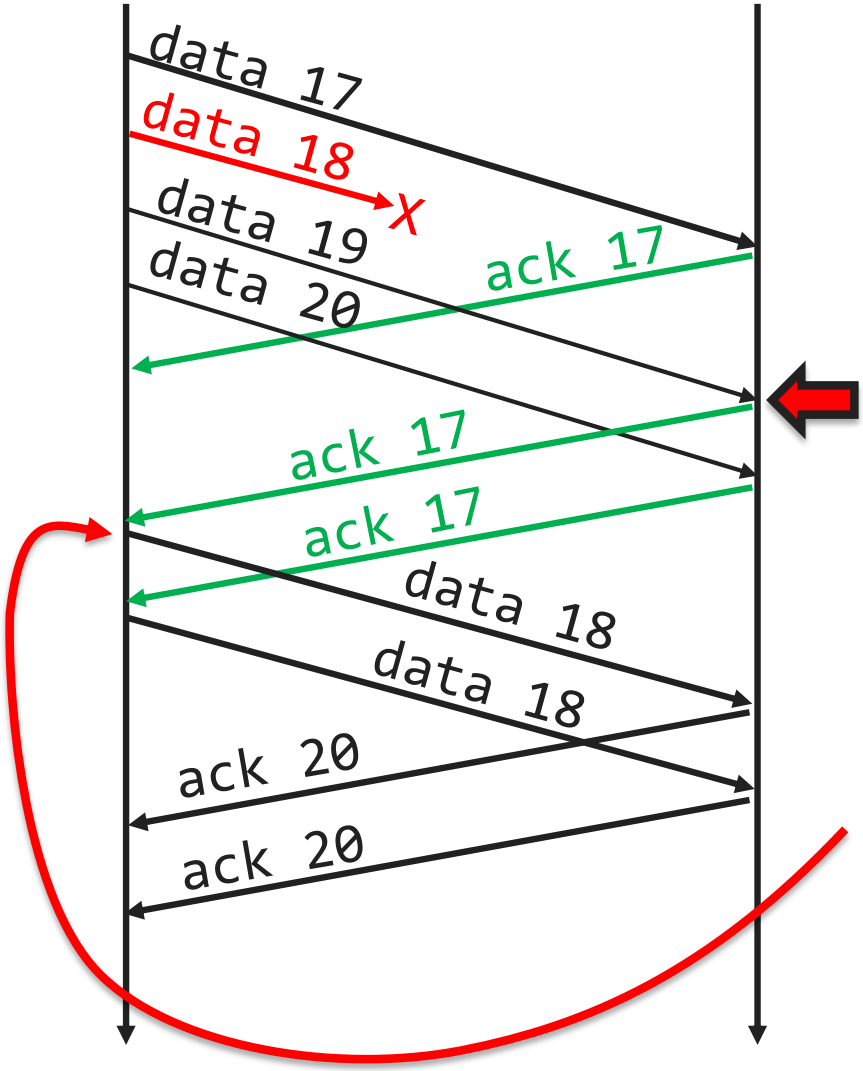
Suppose:

- b/w is `b` bytes / second
- RTT is `r` seconds
- ACK is a small message

→ you can send `b*r` bytes before receiving an ACK for the first byte

(but b/w and RTT are both variable…)

TCP **Fast Retransmit**



Also called **Selective Repeat**

Receiver detects a lost packet (*i.e.*, a missing seq), ACKs the last id it successfully received

Sender can detect the loss without waiting for timeout

TCP Congestion Control

Additive-Increase/Multiplicative-Decrease (**AIMD**):

- window size++ every RTT if no packets dropped
- window size/2 if packet is droppped
  - drop evident from the acknowledgements

→ slowly builds up to max bandwidth, and hover there

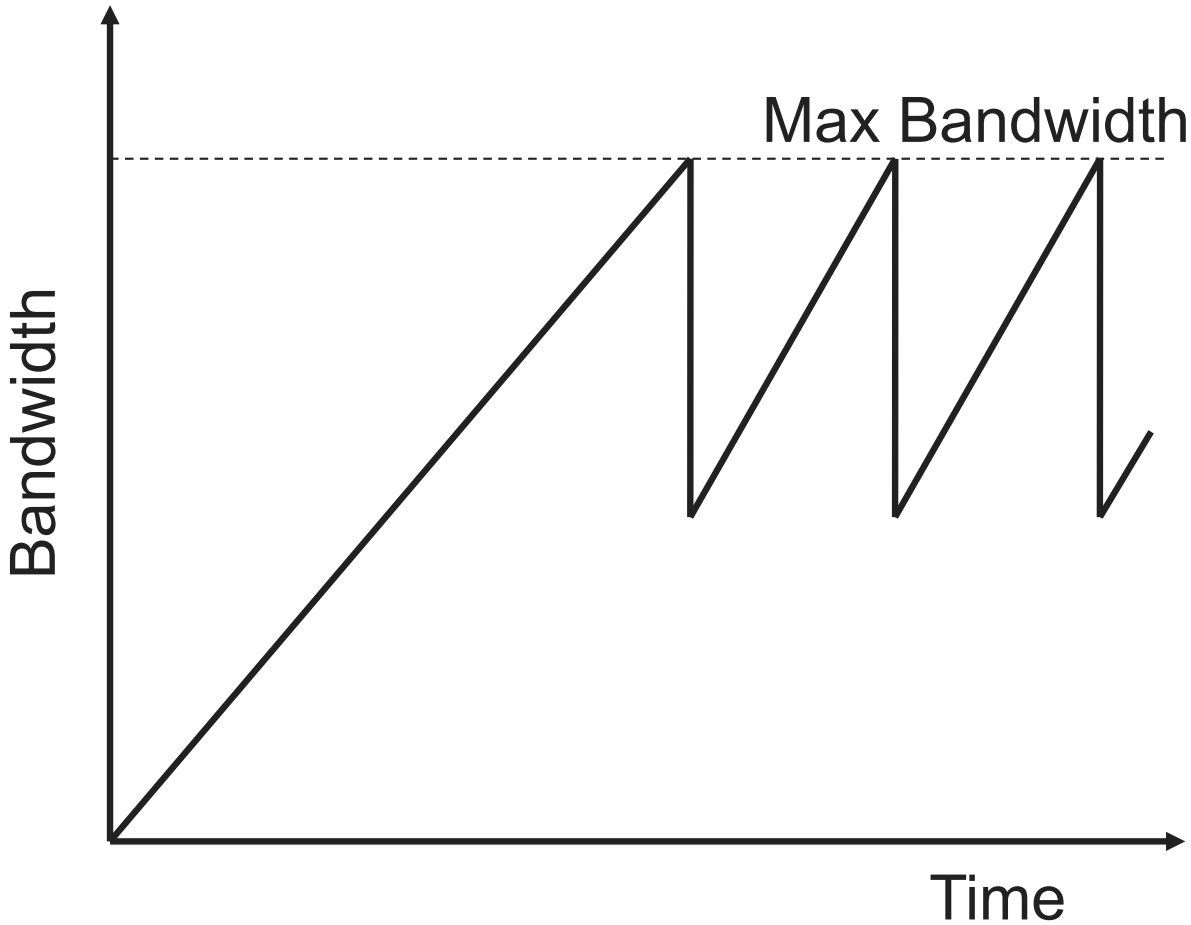
- Does not achieve the max possible
+ Shares bandwidth well with other TCP connections

This linear-increase, exponential backoff in the face of congestion is termed ***TCP-friendliness***

TCP Window Size

- Linear increase
- Exponential backoff

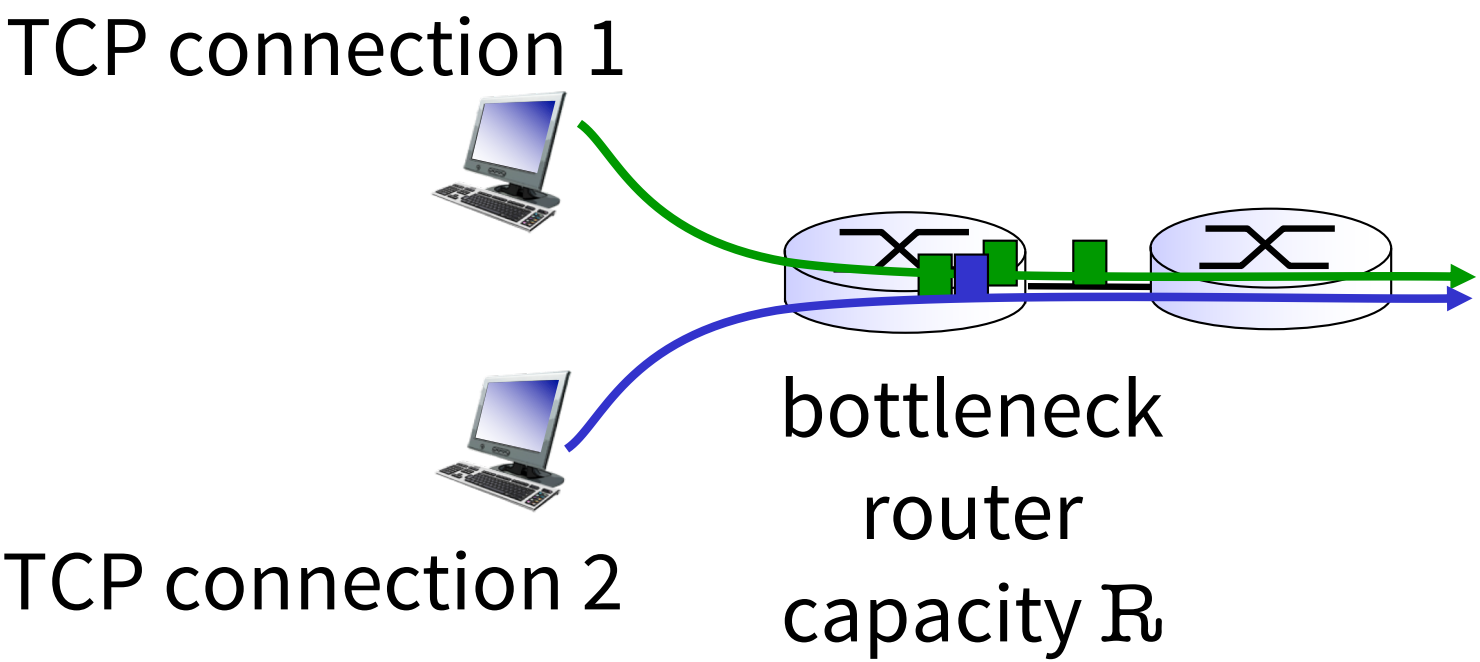
(Assuming no other losses in the network except those due to bandwidth)



<u>Window Sizes:</u>
1,2,3,4,5,6,7,8,9,10,
5,6,7,8,9,10,
5,6,7,8,9,10,
. . .

TCP Fairness

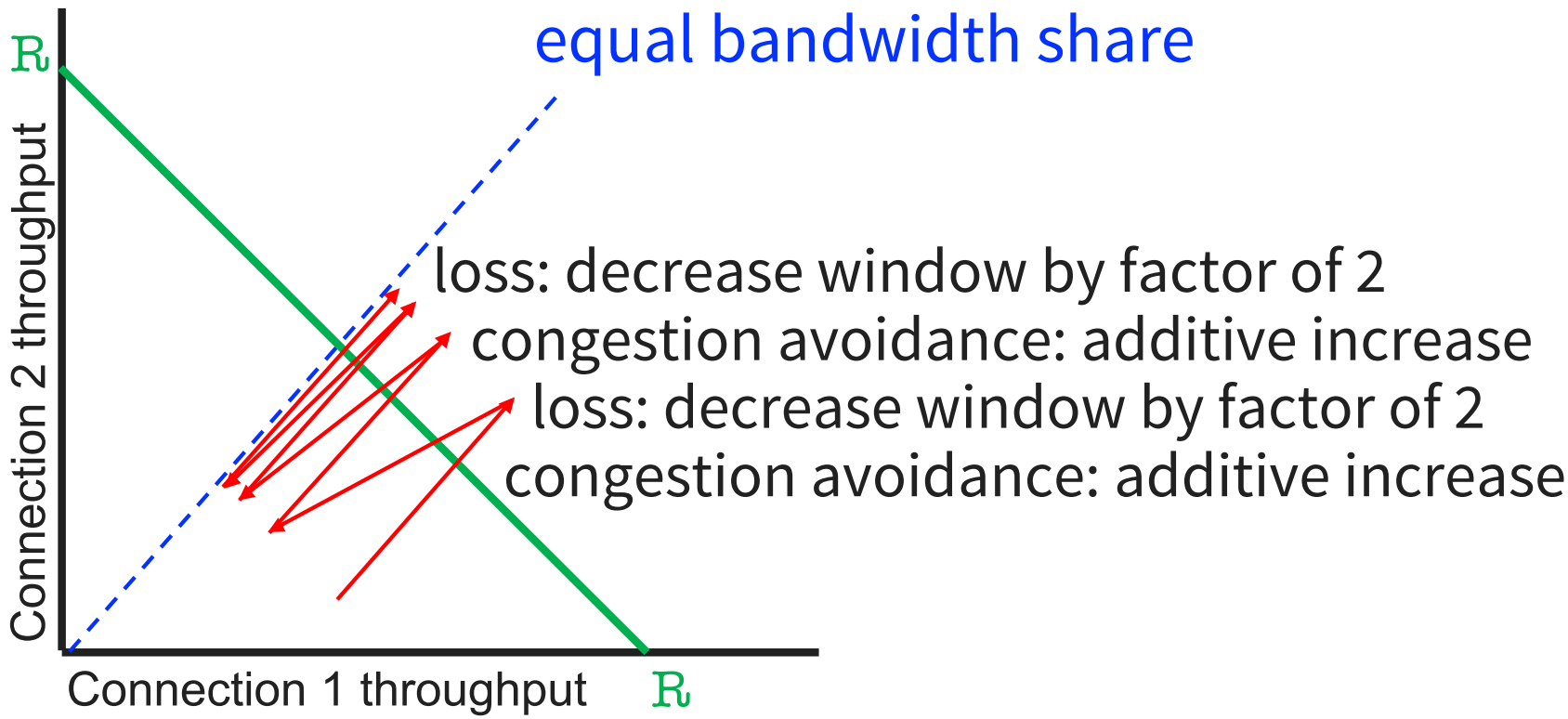
Fairness goal: if k TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/k



Why is TCP fair?

Two competing sessions:

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



TCP Slow Start

(horrible name)

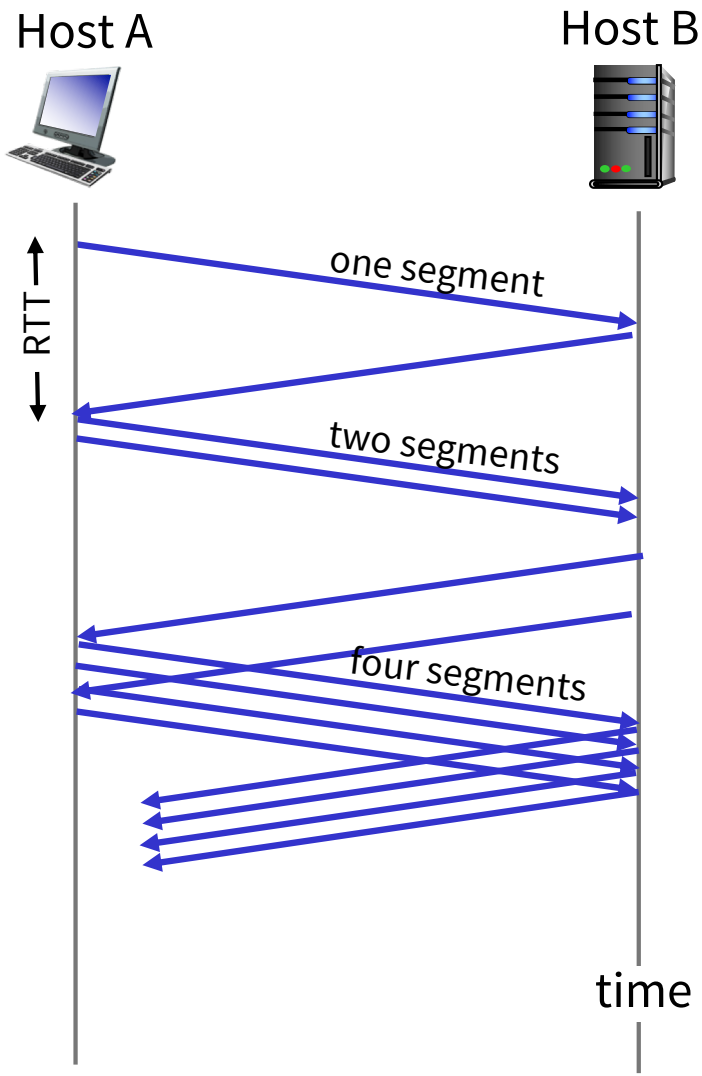
Problem:

- linear increase takes a long time to build up a window size that matches the link bandwidth*delay
- most file transactions are short

→ TCP spends a lot of time with small windows, never reaching large window size

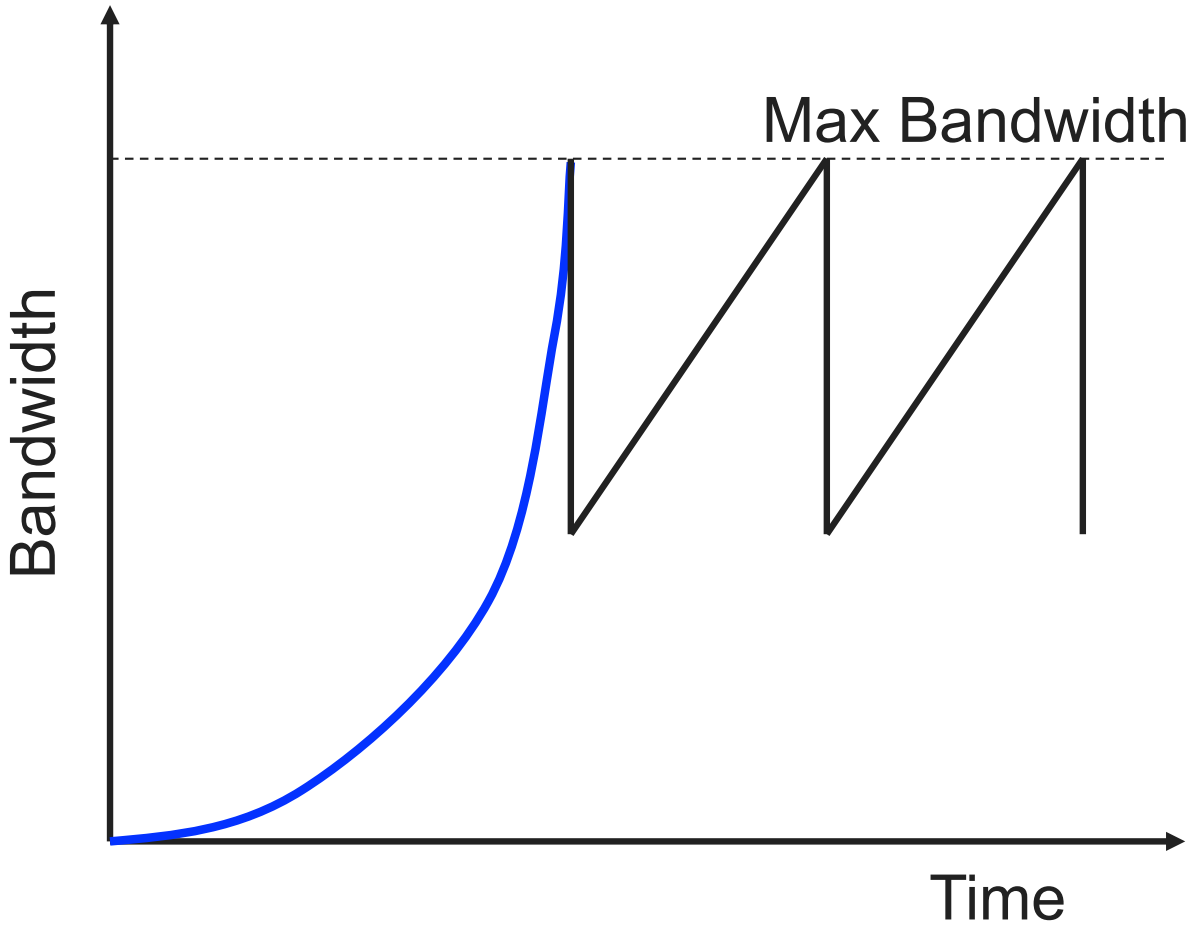
Solution: Allow TCP to increase window size by doubling *until first loss*

Initial rate is **slow** but **ramps up exponentially fast**



TCP Slow Start

- Initial phase: **exponential increase**
- Assuming no other losses in the network except those due to bandwidth



A word about A3

AIMD is a technique independent of TCP

- In A3 you are asked to implement AIMD at the application layer in response to a server's limited buffer size
- In A3, you are not throttling the TCP window size (# of outstanding packets allowed) but **the size of the message itself**
- If you are not clear about this distinction, you will have difficulties with Part 2. ☹

TCP Summary

- Reliable ordered message delivery
  - Connection oriented, 3-way handshake
- Transmission window for better throughput
  - Timeouts based on link parameters
- Congestion control
  - Linear increase, exponential backoff
- Fast adaptation
  - Exponential increase in the initial phase

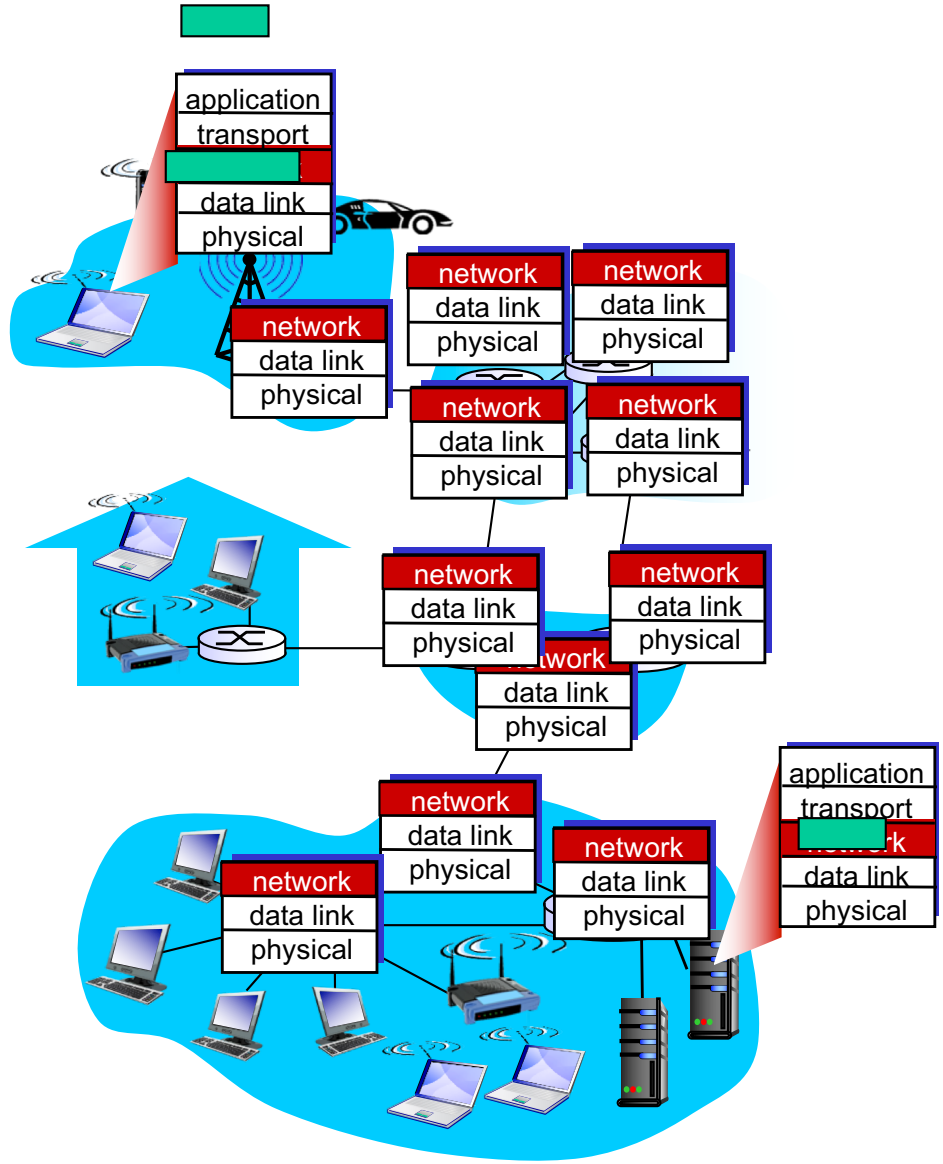
Application
Transport
Network
Link
Physical

Network Layer: Forwarding & Routing

Several figures in this section come from
"Computer Networking: A Top Down Approach"
by Jim Kurose, Keith Ross

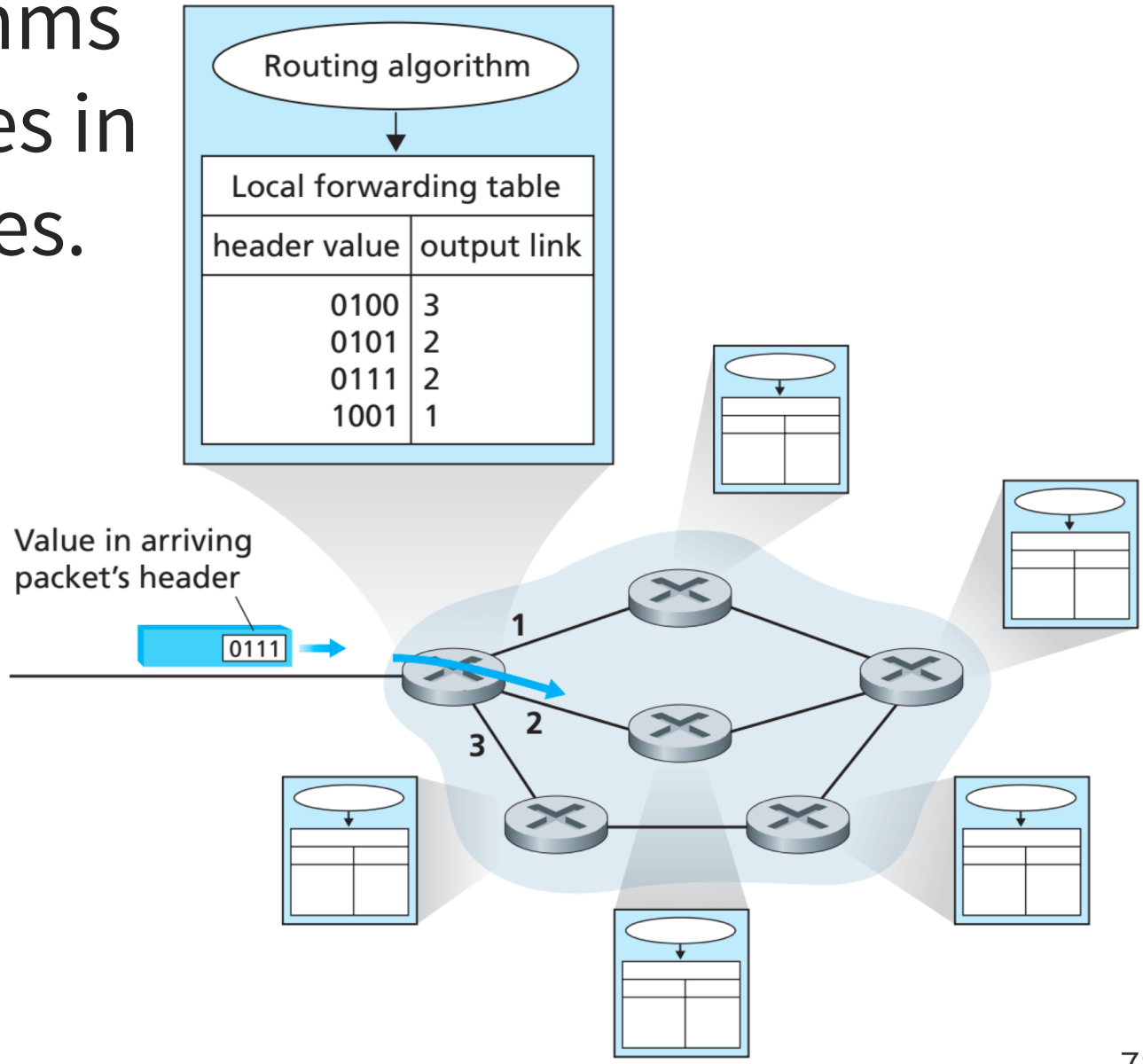
Network layer

- transport segment from sending to receiving host
- on sending side encapsulates segments into datagrams
- on receiving side, delivers segments to transport layer
- network layer protocols in *every* host, router
- router examines header fields in all IP datagrams passing through it

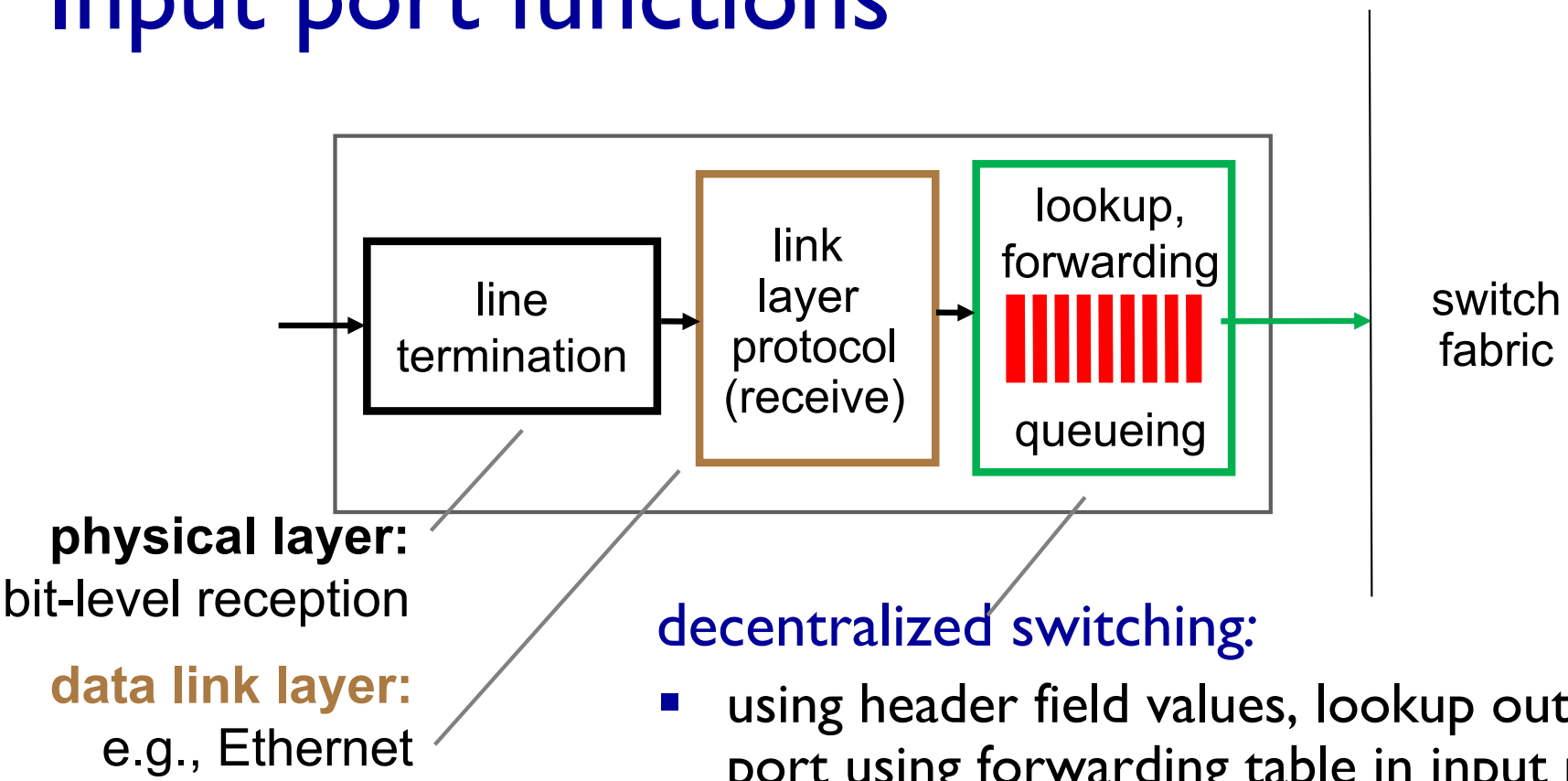


Forwarding

Routing algorithms determine values in forwarding tables.



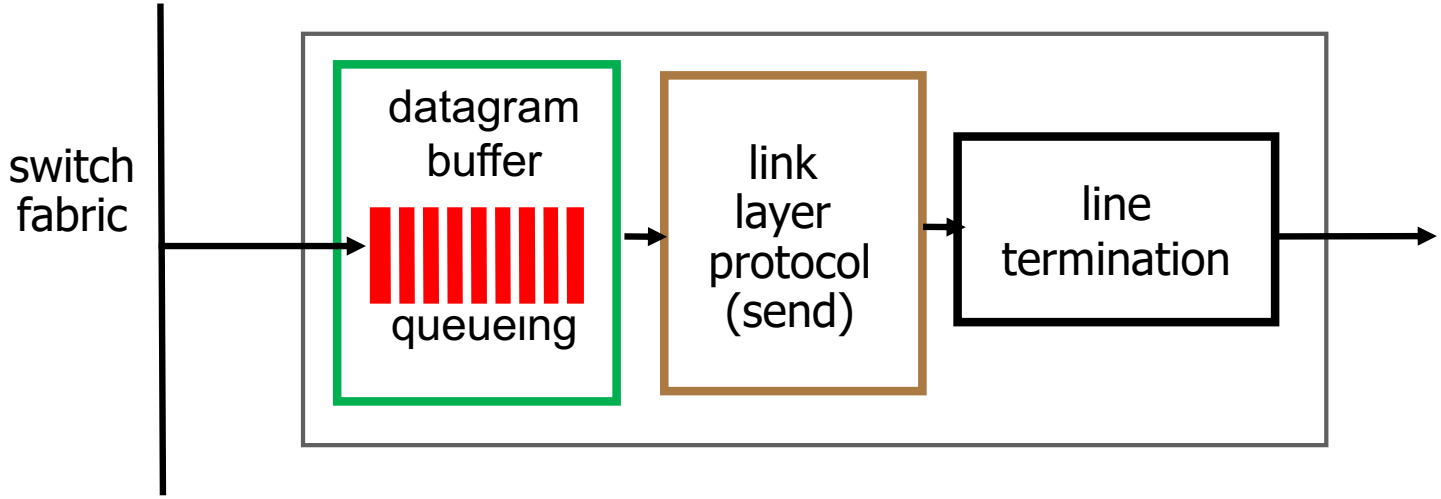
Input port functions



decentralized switching:

- using header field values, lookup output port using forwarding table in input port memory (*"match plus action"*)
- *traditionally*: forward based on destination IP address

Output ports



- *buffering* required when datagrams arrive from fabric faster than the transmission rate

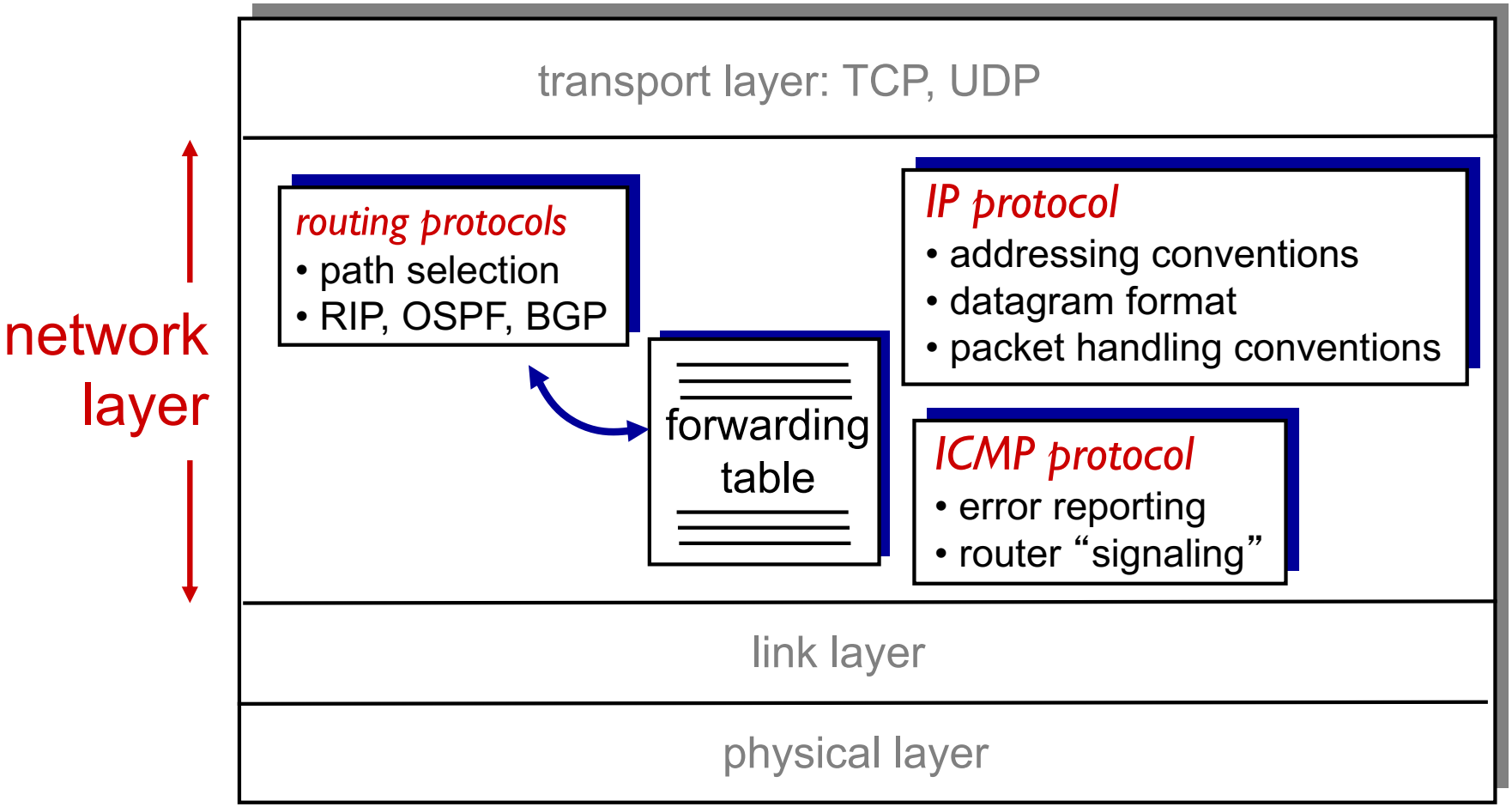
Datagram (packets) can be lost due to congestion, lack of buffers

- *scheduling discipline* chooses among queued datagrams for transmission

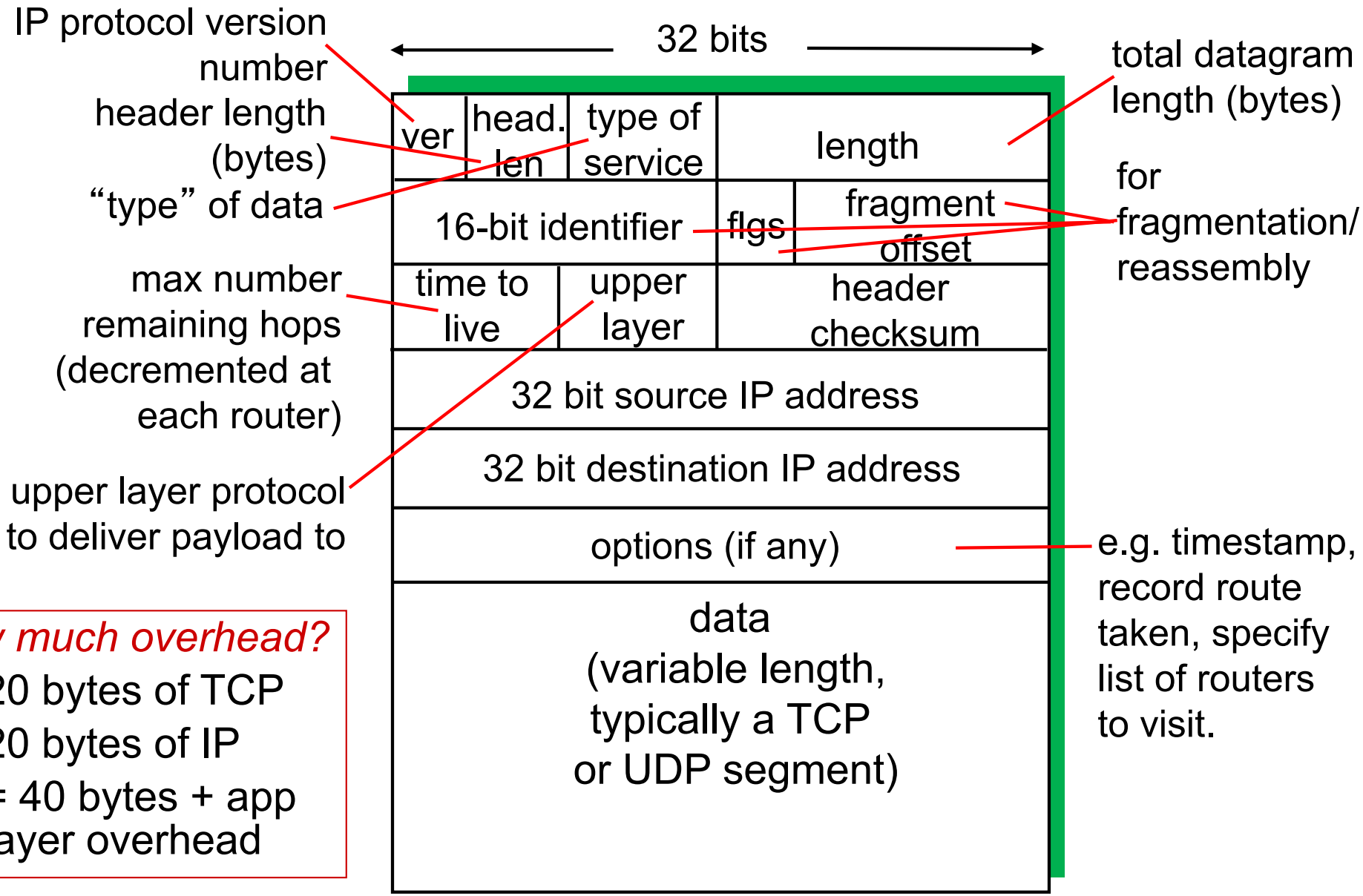
Priority scheduling – who gets best performance, network neutrality

The Internet network layer

host, router network layer functions:



IP datagram format

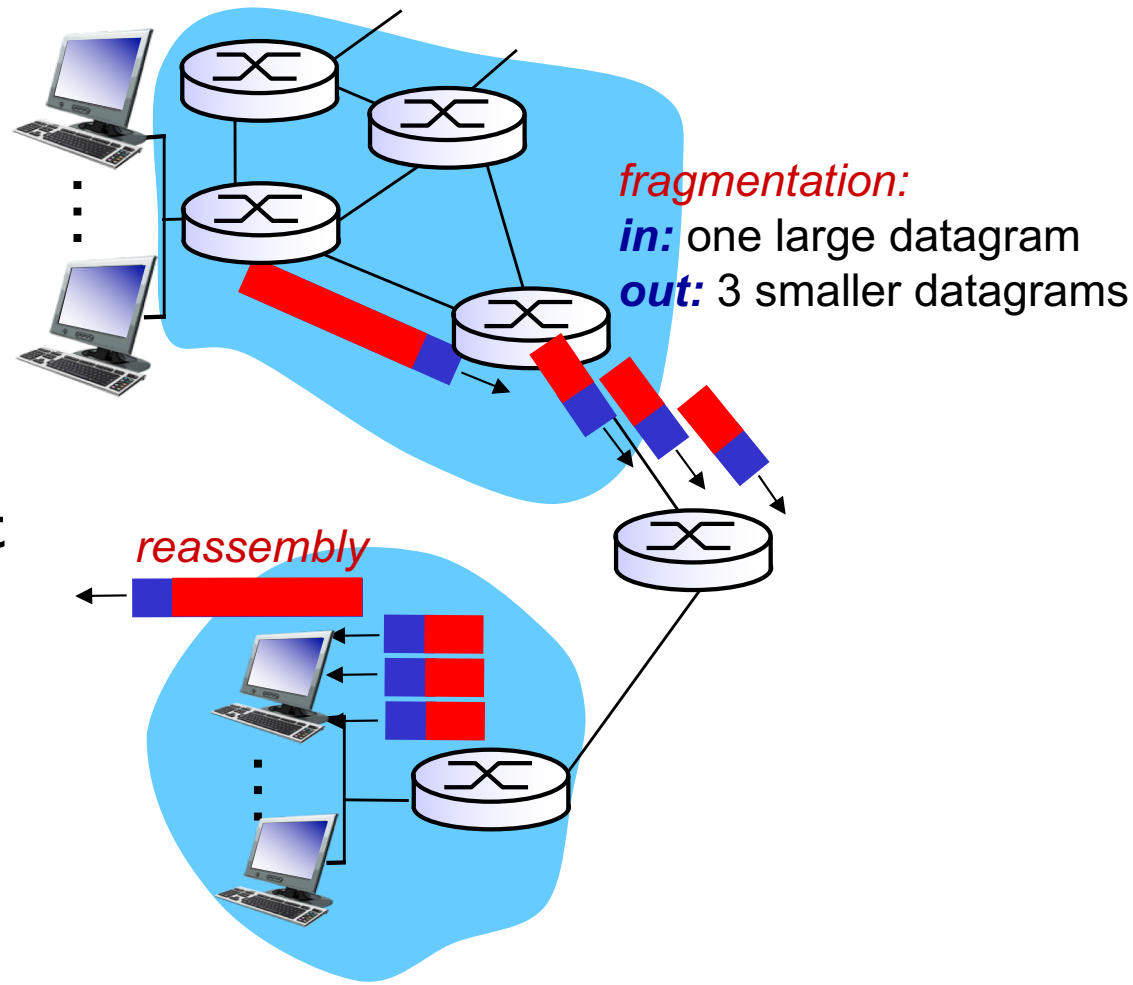


how much overhead?

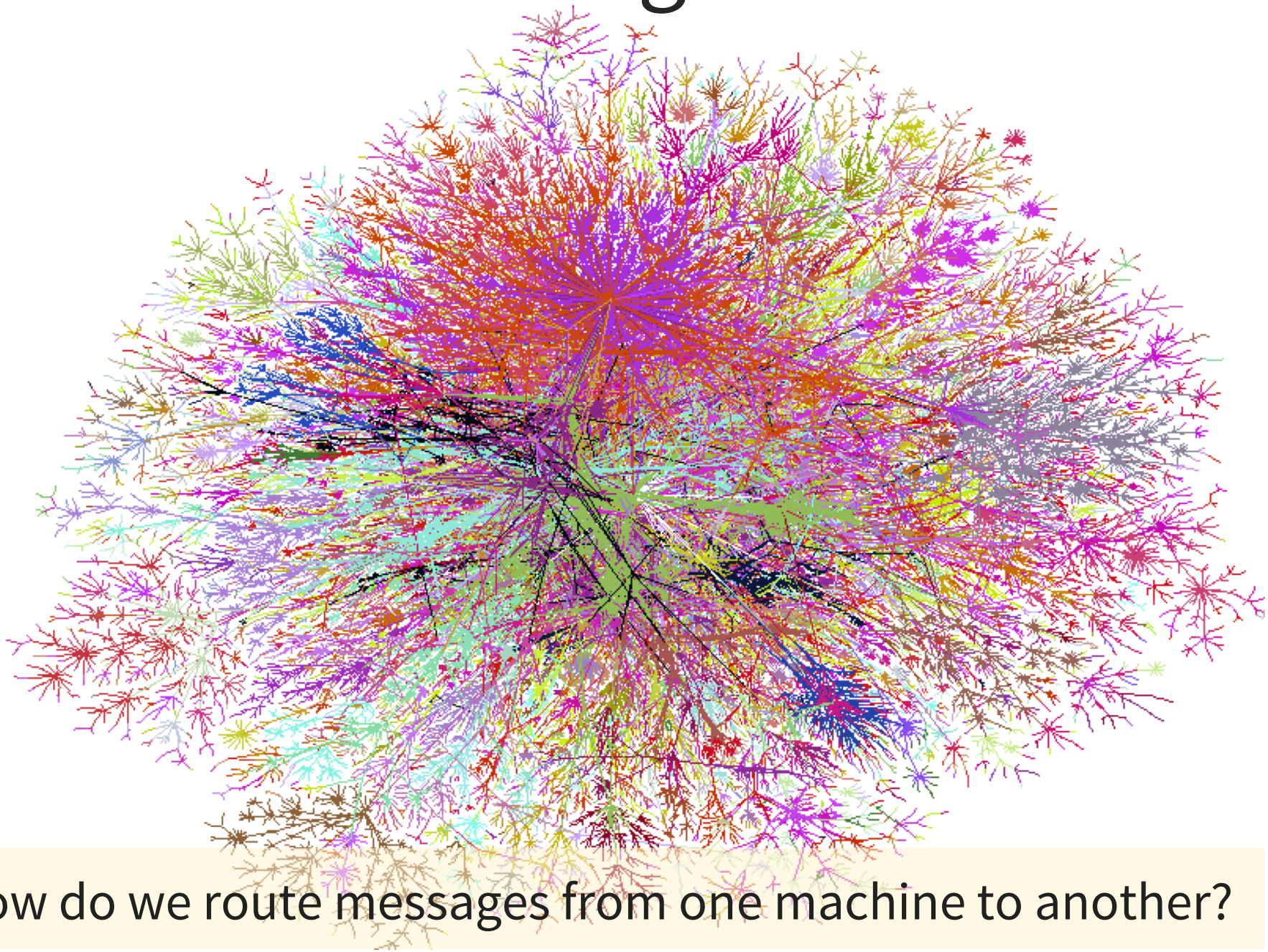
- 20 bytes of TCP
- 20 bytes of IP
- = 40 bytes + app layer overhead

IP fragmentation, reassembly

- network links have MTU (max.transfer size) - largest possible link-level frame
  - different link types, different MTUs
- 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



The Internet is Big….

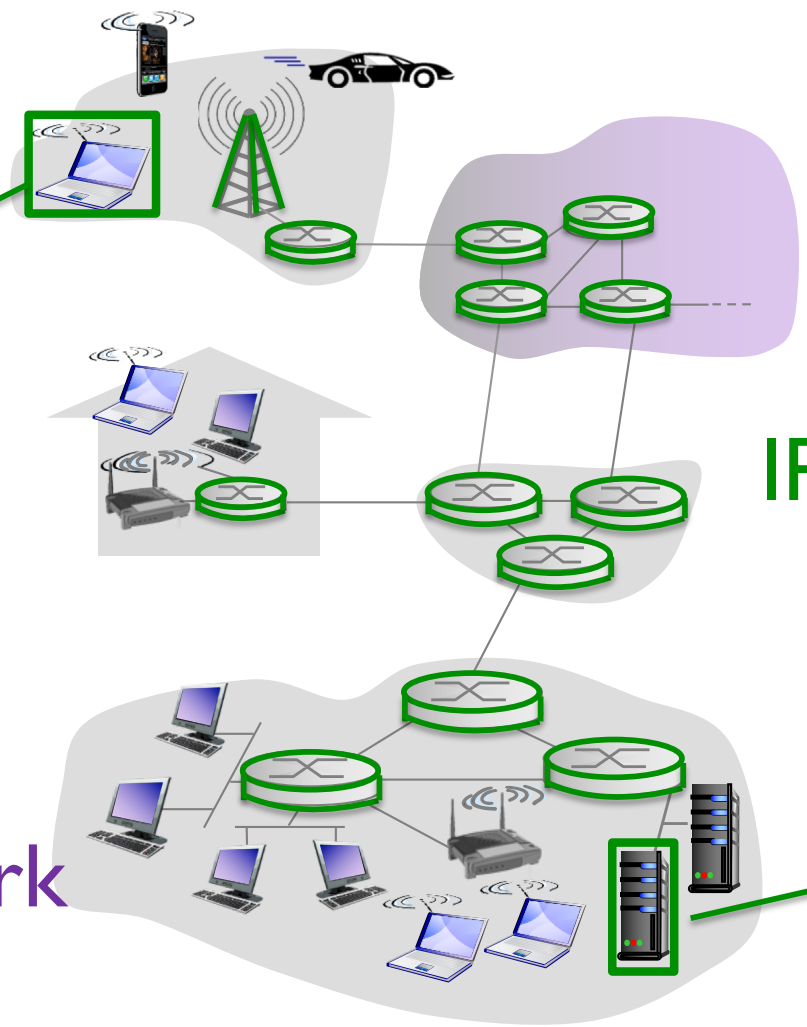


How do we route messages from one machine to another?

Transport Layer vs. Network Layer

Logical communication between *processes* on hosts

Logical communication between *hosts*



IP: best-effort delivery

TCP & UDP: relies on & enhances network layer services

Routing Challenge

Discover and maintain paths through the network between communicating endpoints.

- Metrics of importance
  - Latency
  - Bandwidth
  - Packet Overhead (“Goodput”)
  - Jitter (packet delay variation)
  - Memory space per node
  - Computational overhead per node

Domains

- Wired networks
  - Stable, administered, lots of infrastructure
    - *e.g.*, the Internet
- Wireless networks
  - Wireless, dynamic, self-organizing
  - Infrastructure-based wireless networks
    - A.k.a. cell-based, access-point-based
    - *e.g.*, Cornell's "rover"
  - Infrastructure-less wireless networks
    - A.k.a. ad hoc

Algorithm Classifications

Route discovery, selection and usage

- Reactive vs. **Proactive**
- **Single path** vs. Multipath
- Centralized vs. **Distributed**

Reactive Routing

- Routes discovered on the fly, as needed
  - Discovery often involves network-wide query
  - Used on many wireless ad hoc networks

- Examples
  - Dynamic source routing (DSR)
  - Ad hoc on-demand distance vector (AODV)

Dynamic Source Routing (DSR) Protocol

Route Discovery:

(1) Source sends neighbors RouteRequest

"I'm Source X looking for Dest Y"

- Path to Y generated as neighbors add themselves to the path & pass RREQ to *their* neighbors
- Nodes drop redundant RREQs

(2) Destination sends back a RouteReply

"I'm Dest Y responding to Source X"

- Source X caches path to Y
- future data packets specify path in header

Route Maintenance:

- Broken links reported
- Affected paths removed from caches

Reactive Routing

- Pros
  - Routers require no state
  - State proportional to # of used routes
  - Communication proportional to # of used routes and failure rate
- Cons
  - Route discovery latency is high
  - Jitter (variance of packet interarrival times) is high

Algorithm Classifications

Route discovery, selection and usage

- Reactive vs. **Proactive**
- **Single path** vs. Multipath
- Centralized vs. **Distributed**

Proactive Routing

- Routes are disseminated from each node to all others, periodically
  - Every host has routes available to every other host, regardless of need
  - Used on the internet, some wireless ad hoc networks

Graph Abstraction of the Network

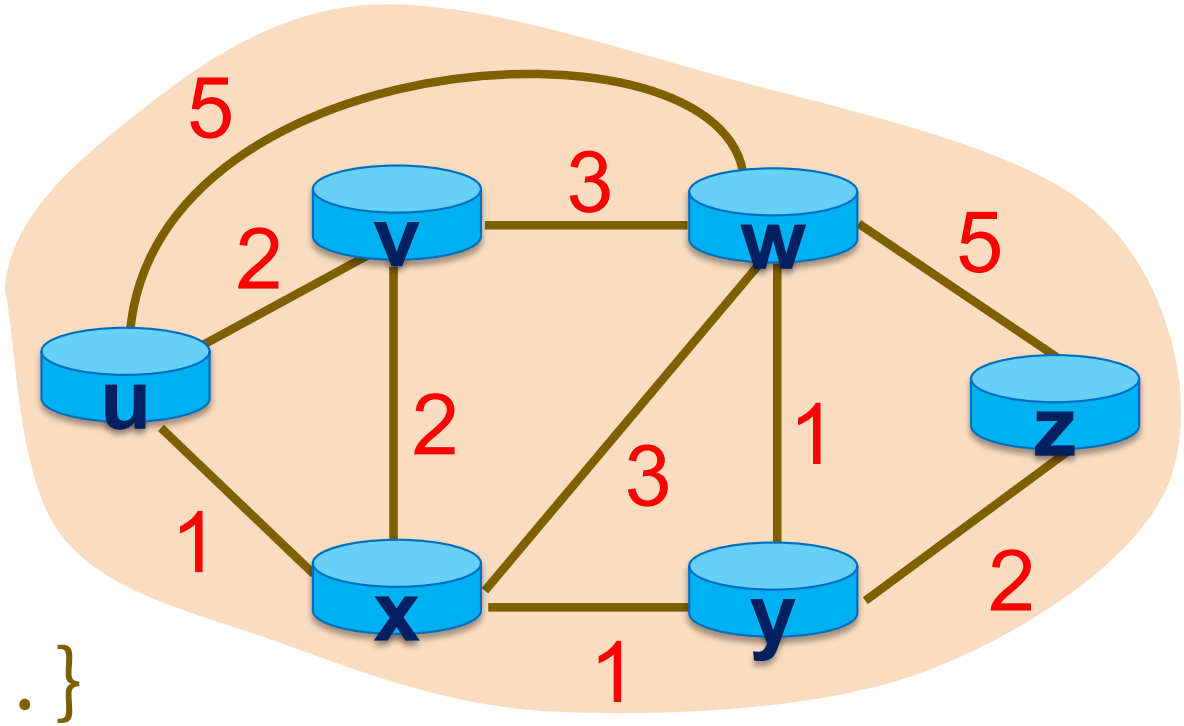
graph G = (V,E)

set of routers

V = { u, v, w, x, y, z }

set of links

E ={ (u,v), (u,x),(u,w)… }



cost of link c(x,x’) e.g., c(w,z) = 5

(cost could always be 1, or inversely related to b/w or congestion)

key question: what is the least-cost path between u and z ?

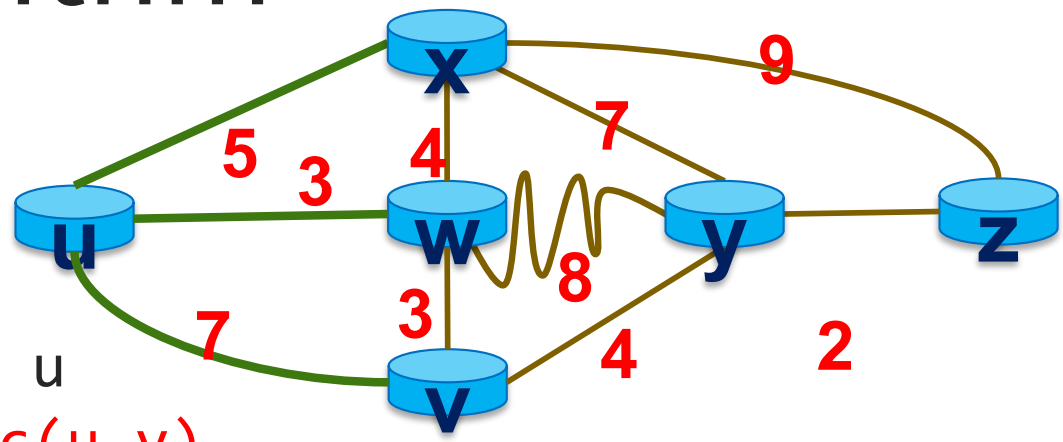
routing algorithm: algorithm that finds that least cost path

Link State (LS) Routing Algorithm

- iterative, centralized
- network topology, all link costs known up front
  - accomplished via “link state broadcast”
  - all nodes have same info
- based on Dijkstra’s (shortest path algorithm)
  - computes least cost paths from one node (‘source”) to all other nodes
- Example: Open Shortest Path First (OSPF) Protocol

```
c(x,y): link cost from node x to y;
        (∞ for non-neighbors)
D(v):   current cost of path from source to v
N':     set of nodes whose least cost path
        definitively known
```

Dijsktra's algorithm



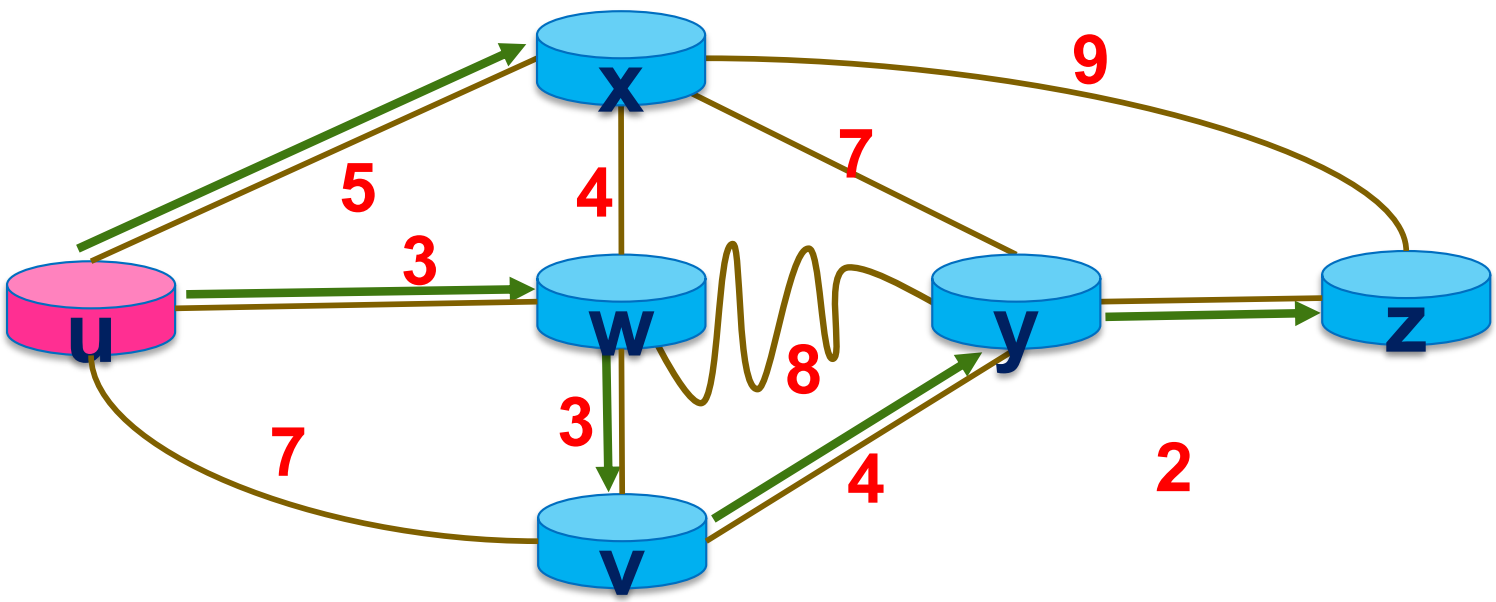

```
Initialization:
  N' = {u}
  for all nodes v
    if v adjacent to u
        then D(v) = c(u,v)
    else D(v) = ∞

 Loop
   find w not in N' such that D(w) is a minimum
   add w to N'
   update D(v) for all v adjacent to w & not in N' :
      D(v) = min( D(v), D(w) + c(w,v) )
   /* new cost to v either: old cost to v or known
    shortest path cost to w plus cost from w to v */
until all nodes in N'
```

Dijsktra's in Action

p(x): predecessor node along path from source to node x

D(v): current cost of path from **source** to v

Step	N'	D(v), p(v)	D(w), p(w)	D(x), p(x)	D(y), p(y)	D(z), p(z)
0	u	7,u	3,u	5,u	∞	∞
1	uw	6,w		5,u	11,w	∞
2	uwx	6,w			11,w	14,x
3	uwxv				10,v	14,x
4	uwxvy					12,y
5	uwxvyz					



Algorithm Classifications

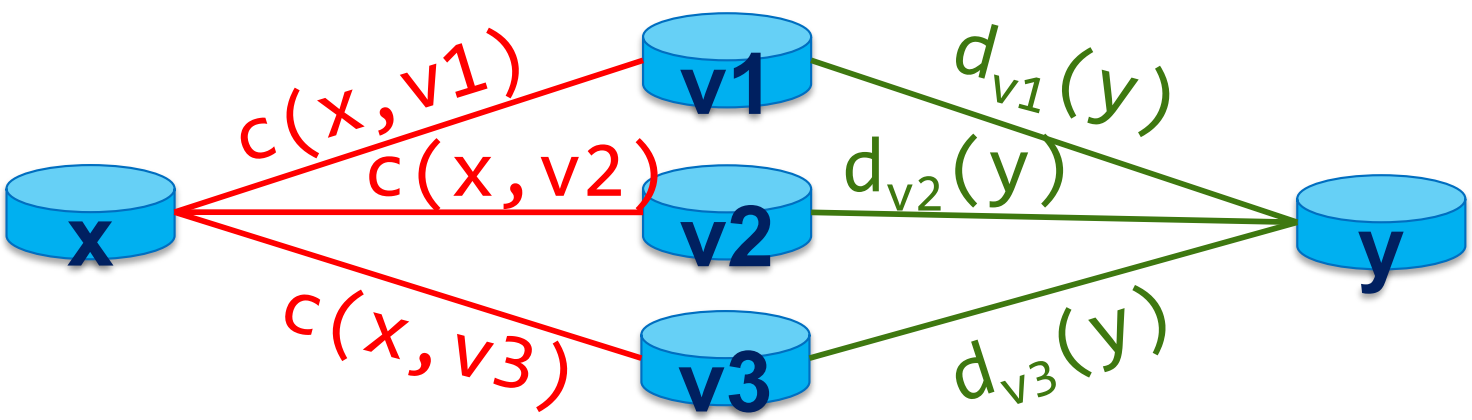
Route discovery, selection and usage

- Reactive vs. **Proactive**
- **Single path** vs. Multipath
- Centralized vs. **Distributed**

Distance Vector (DV) Routing Algorithm

- iterative, asynchronous, distributed
- based on Bellman-Ford (shortest path algorithm)
- Example: Routing Information Protocol (RIP)

```
let
   dx(y) := cost of least-cost path from x to y
then
   dx(y) = min {c(x,v) + dv(y) }
         for all neighbors v of x
```



Bellman Ford Example

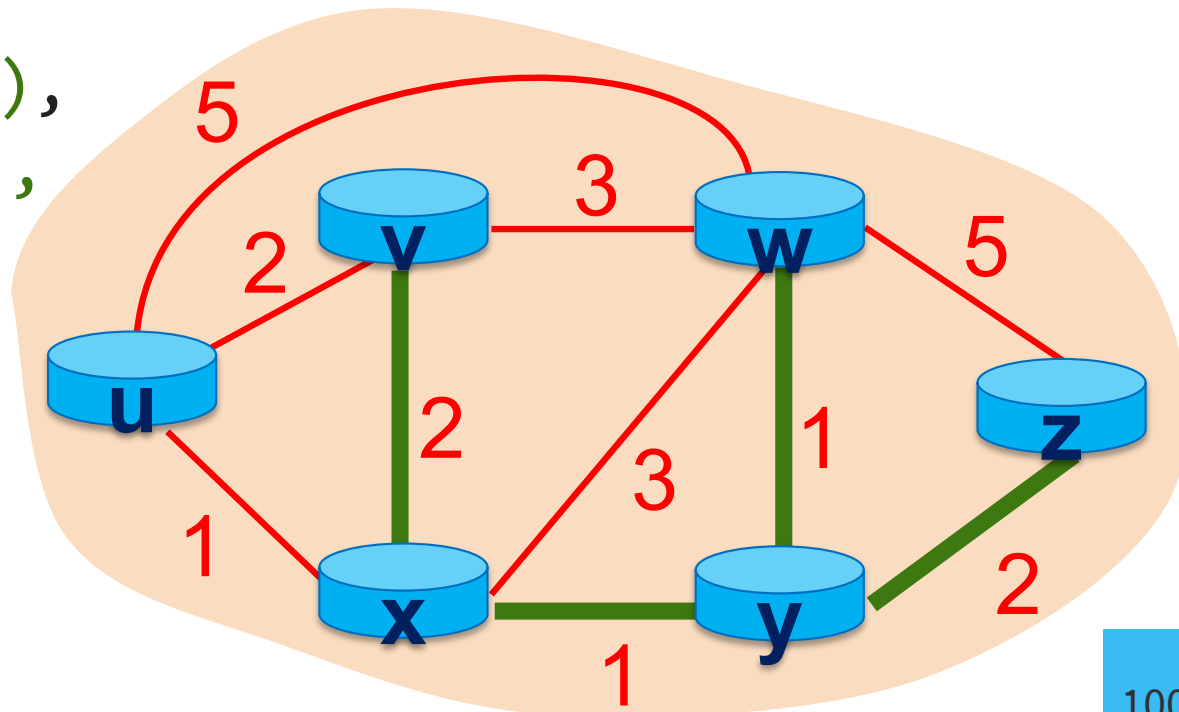
Shortest path from u to z?

Who are u's neighbors? {v, x, w}

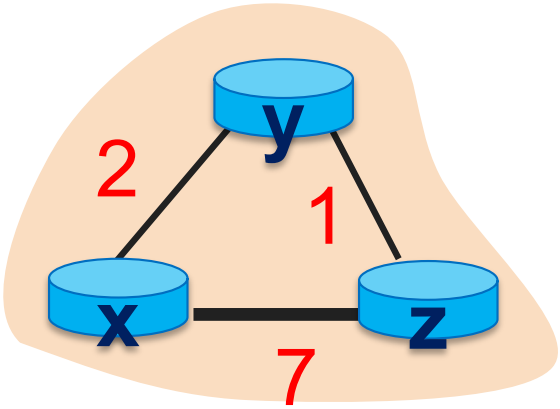
What are *their* shortest paths to z?

$d_v(z) = 5,\ d_x(z) = 3,\ d_w(z) = 3$

```
du(z)=min{c(u,v)+dv(z),
          c(u,x) + dx(z),
          c(u,w) + dw(z)
     = min {2 + 5,
            1 + 3,
            5 + 3}
     = 4
```



DV Algorithm



Each node x:

- knows cost to each neighbor v: c(x,v)
- maintains its neighbors' distance vectors

From time to time (esp. when a change occurs), each node sends its own distance vector estimate to neighbors.

When x receives new DV estimate from neighbor, it updates its own DV using B-F equation.

DV Algorithm In Action

time

X, t=0		cost to		
		x	y	z
from	x	0	2	7
	y	∞	∞	∞
	z	∞	∞	∞

Y, t=0		cost to		
		x	y	z
from	x	∞	∞	∞
	y	2	0	1
	z	∞	∞	∞

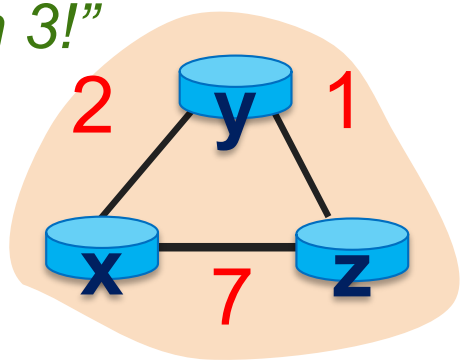
X, t=1		cost to		
		x	y	z
from	x	0	2	**3**
	y	**2**	**0**	**1**
	z	∞	∞	∞

Y sends X its DV

X updates its own DV

"If Y can get to Z in 1,

*then *I* can get to Z in 3!"*



DV Algorithm when costs decrease

time

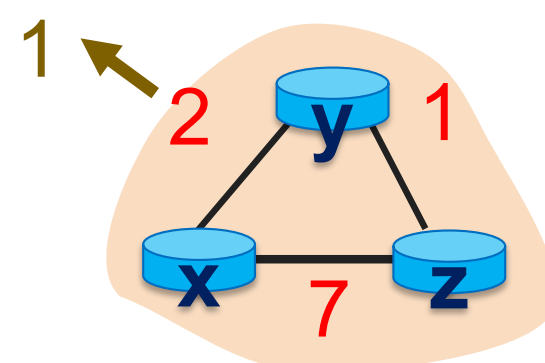
X, t=0		cost to		
		x	y	z
from	x	0	2	3
	y	2	0	1
	z	3	1	0

Y, t=0		cost to		
		x	y	z
from	x	0	2	3
	y	1 ~~2~~	0	1
	z	3	1	0

Y detects link-cost changes 2 → 1 Updates DV, broadcasts

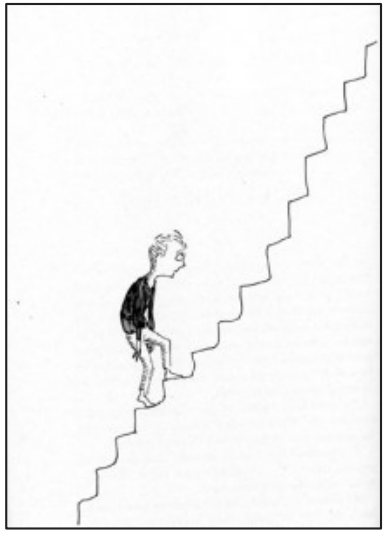
X, t=1		cost to		
		x	y	z
from	x	0	**1**	**2**
	y	**1**	**0**	**1**
	z	3	1	0

X updates its own DV, broadcasts



Counting to Infinity…

What if connections to z are lost?



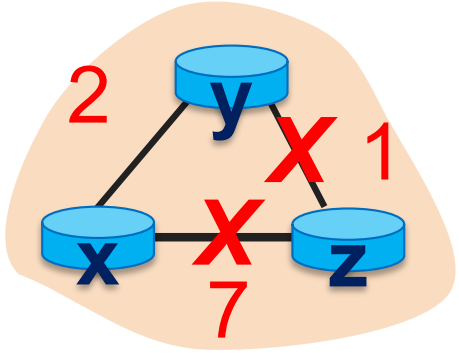
"Well, I can't reach Z anymore, but Y can do that in 1, so I can still get to Z in 3."

X, t=n		*cost to*		
		x	y	z
from	x	0	2	~~z~~ 3
	y	2	0	1
	z	∞	∞	∞

"Well, I can't reach Z anymore, but X can do that in 3, so I can still get to Z in 5."

Y, t=n		*cost to*		
		x	y	z
from	x	0	2	3
	y	2	0	~~1~~ 5
	z	∞	∞	∞

Next: Y sends X its new DV, X updates Y's DV, reruns BF, x → z increases from 3 → 7 … Next…!!



Path Vector (PV) Routing Algorithm

- Distance Vector with paths
- Example: Border Gateway Protocol (BGP)

  "glue that holds the Internet together"

High level:

- Each node x sends its distance vector with the actual path
- Nodes can filter out broken paths

Instead of just shortest path, **BGP** uses other considerations to select which route is best

Why BGP?

- Shortest path algorithms insufficient to handle myriad of operational (e.g., loop handling), economic, and political considerations

- Policy categories (Caesar and Rexford):
  - business relationships
  - traffic engineering
  - scalability (improving stability, aggregation)
  - Security

Routing Gone Wrong

- Pakistan, 2008: "*I'll* take you to youtube!"
  - "How Pakistan knocked YouTube offline"
  - "Insecure routing redirects YouTube to Pakistan"

- China, 2010: "*I'll* take you to .gov and .mil"
  - "How China swallowed 15% of 'Net traffic for 18 minutes"
  - "China Hijacks 15% of Internet Traffic?"

Algorithm Classifications

Route discovery, selection and usage

- Reactive vs. **Proactive**
- **Single path** vs. Multipath
- Centralized vs. **Distributed**

Proactive Routing

- Pros
  - Route discovery latency is very low
- Cons
  - O(N) state in every router
  - Constant background communication

Hybrid Routing

- Proactive & Reactive routing have drawbacks
  - Work best under different network conditions
  - Many parameters to pick to get optimal performance

- Perform hybrid routing
- Some routes are disseminated proactively, others discovered reactively
  - Can outperform reactive *and* proactive across many scenarios
    SHARP [Mobihoc 2003]

Application
Presentation (ish)
Transport
Network
Link
Physical

Remote Procedure Call

Several figures in this section come from
"Distributed Systems: Principles and Paradigms"
by Andrew Tanenbaum & Maarten van Steen

Client/Server Paradigm

Common model for structuring distributed computation

- ***Server:*** program (or collection of programs) that provide some *service*, e.g., file service, name service
  - may exist on one or more nodes
- ***Client:*** a program that uses the service

Typical Pattern:

1. Client first *binds* to the server: locates it in the network & establishes a connection
2. Client sends *requests*: messages that indicate which service is desiredth parameters
3. Server returns *response*

Pros and Cons of Messages

+Very flexible communication

- Want a certain message format? *Go for it!*

—Problems with messages:

- programmer must worry about message formats
- must be packed and unpacked
- server must decode to determined request
- may require special error handling functions

Messages are not a natural programming model for most programmers.

Procedure Call

A more natural way to communicate:

- every language supports it
- semantics are well defined and understood
- natural for programmers to use

Idea: Let clients call servers like they do procedures

Remote Procedure Call (RPC)

Goal: design RPC to look like a local PC

- A model for distributed communication
- Uses computer/language support
- 3 components on each side:
  - user program (client or server)
  - set of *stub* procedures
  - RPC runtime support

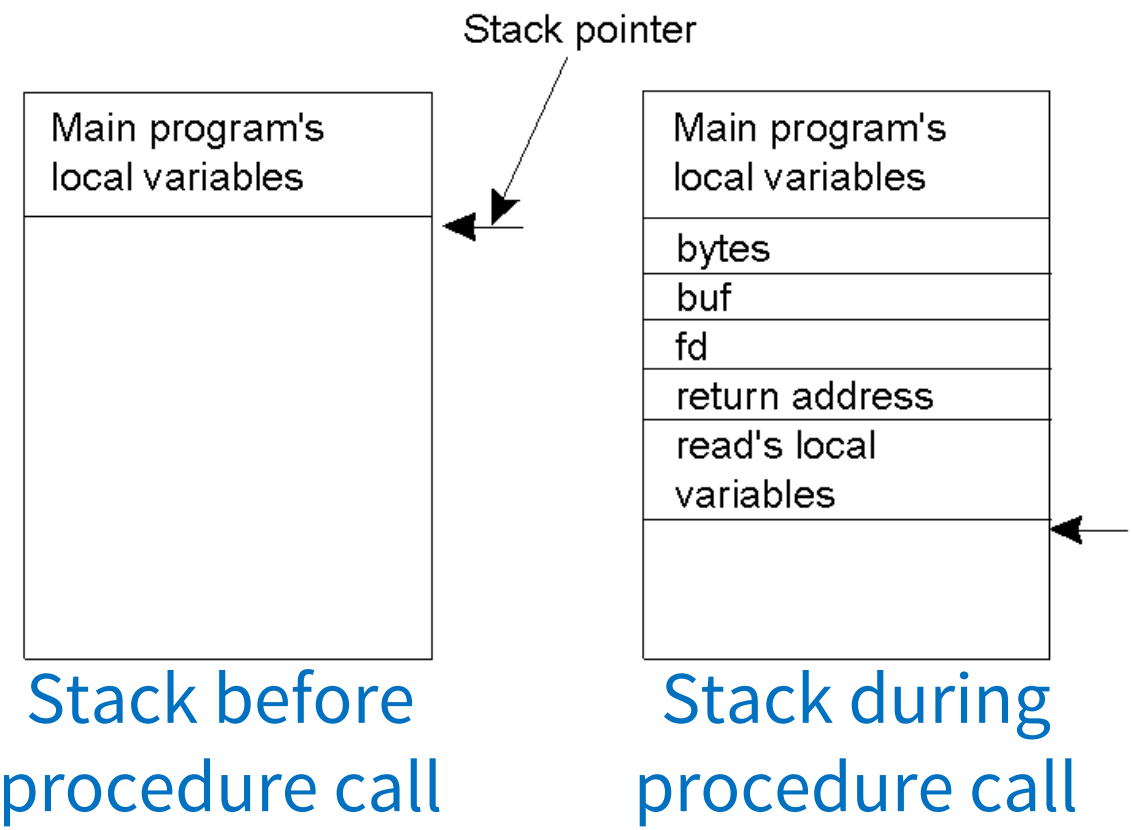
Birrell & Nelson @ Xerox PARC

"Implementing Remote Procedure Calls" (1984)

How does a function call work?

`read(int fd, char* buf, int nbytes)`

- File descriptor
- character array
- how much to read



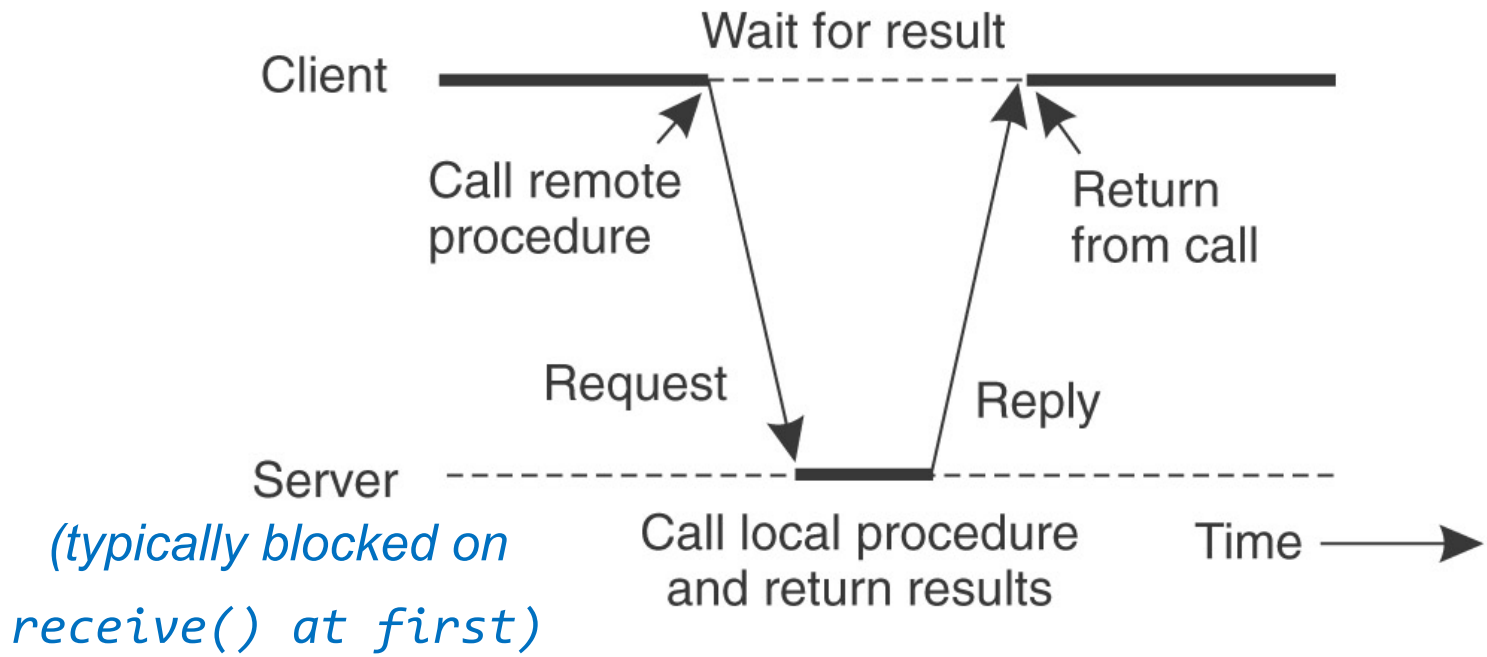
- Linker inserts read implementation into obj file
- Implementation usually invokes a system call

[Tanenbaum & van Steen, Fig 4-5]

How does a RPC work?

Basic idea:

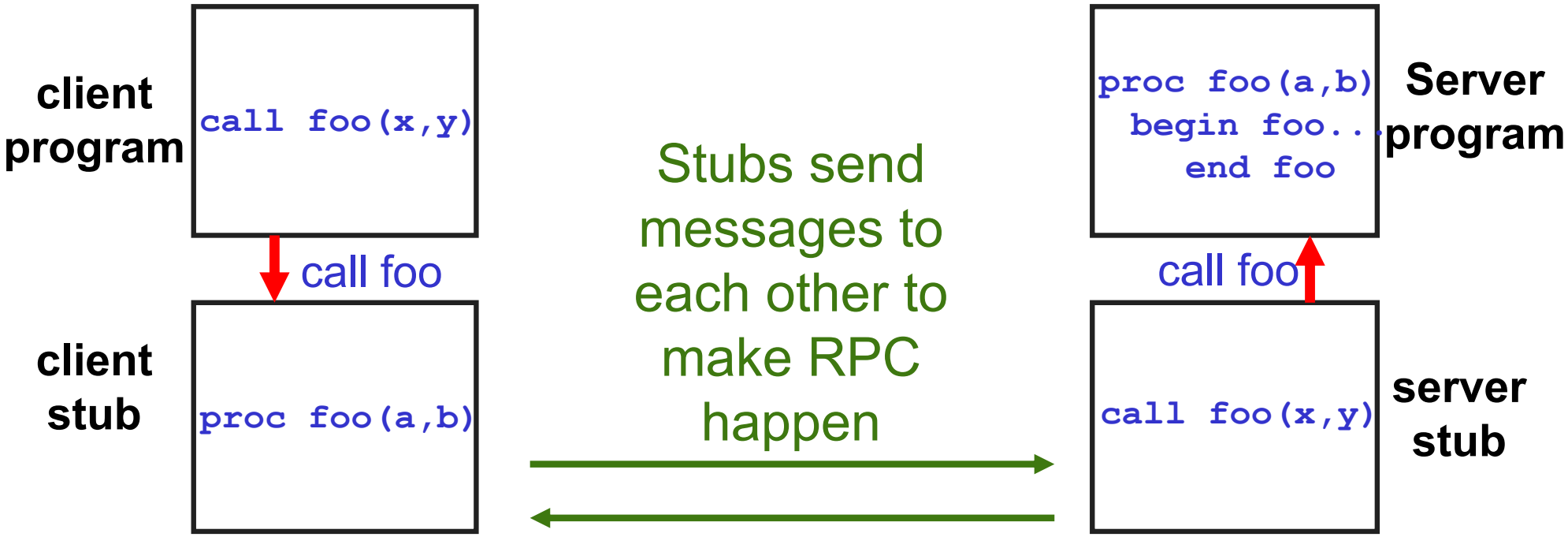
- Server *exports* a set of procedures
- Client calls these procedures, as if they were local functions



- Message passing details hidden from client & server (like system call details are hidden in libraries)

[Tanenbaum & van Steen, Fig 4-6]

RPC Stubs



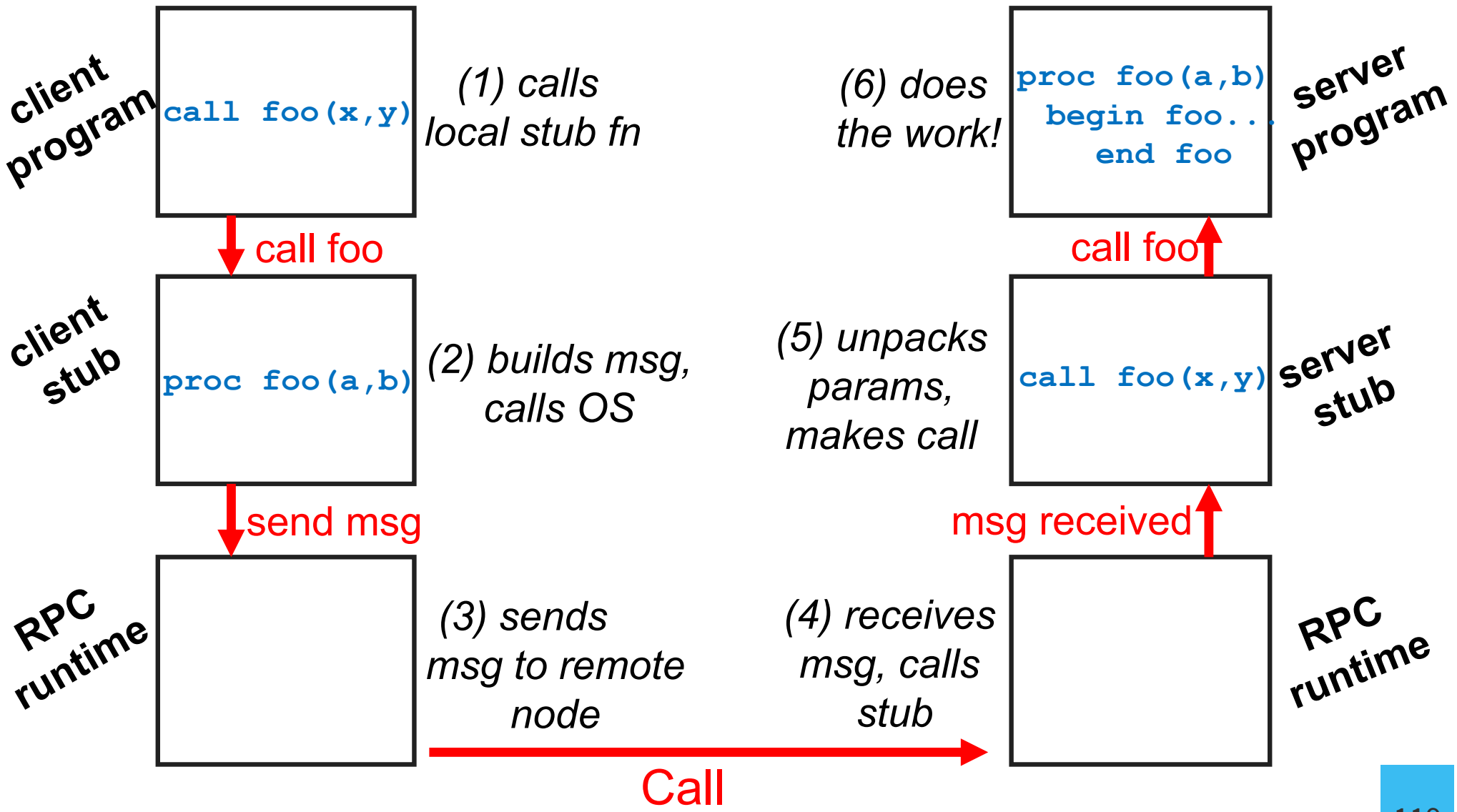
Client-side stub:

- Looks (to the client) like a callable server procedure
- Client program thinks it is calling the server

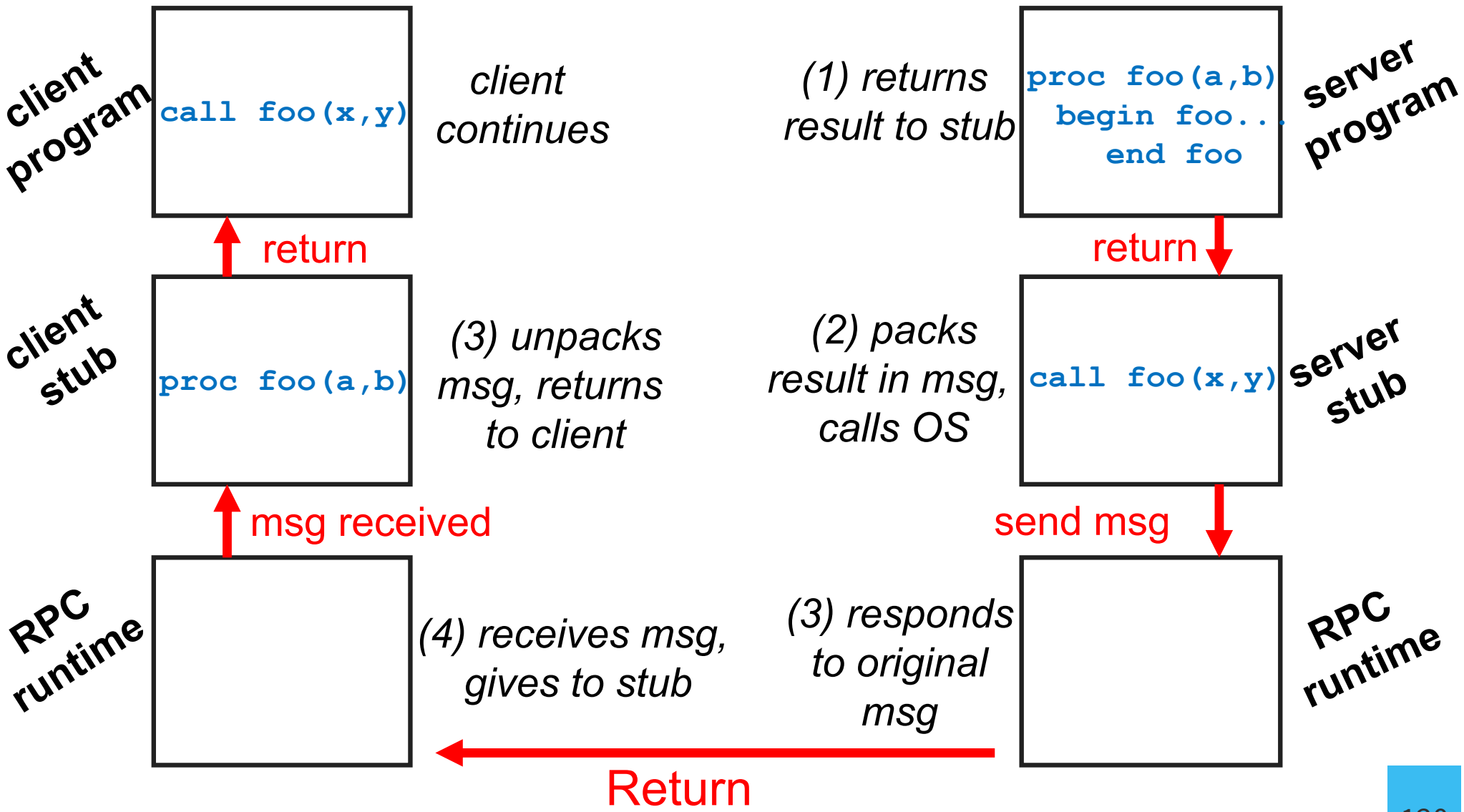
Server-side stub:

- Server program thinks it is called by the client
- `foo` actually called by the server stub

RPC Call Structure



RPC Return Structure

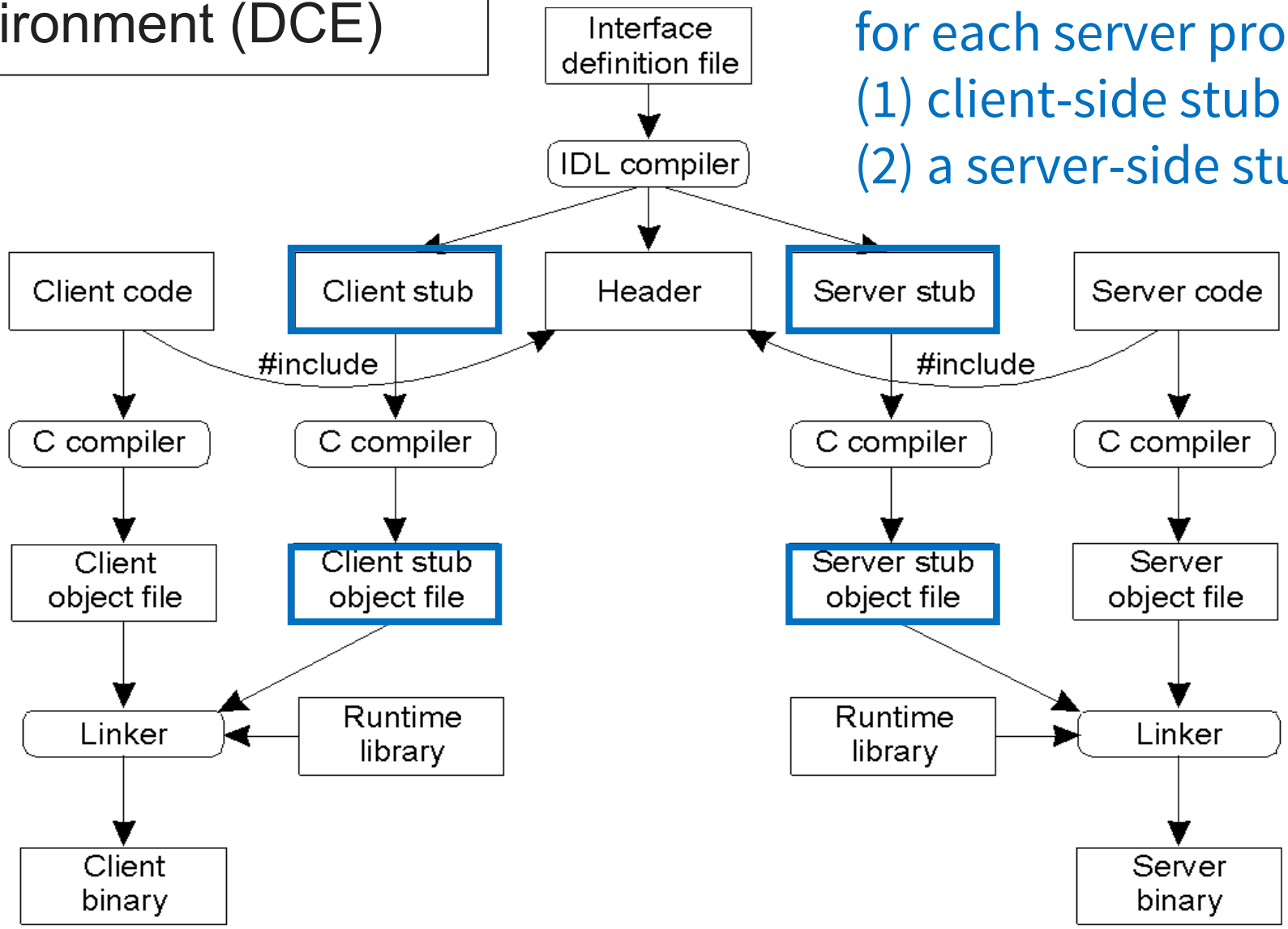


Example RPC system:

Distributed Computing Environment (DCE)

Stub compiler

- reads IDL
- produces 2 stub procedures for each server procedure
  (1) client-side stub
  (2) a server-side stub

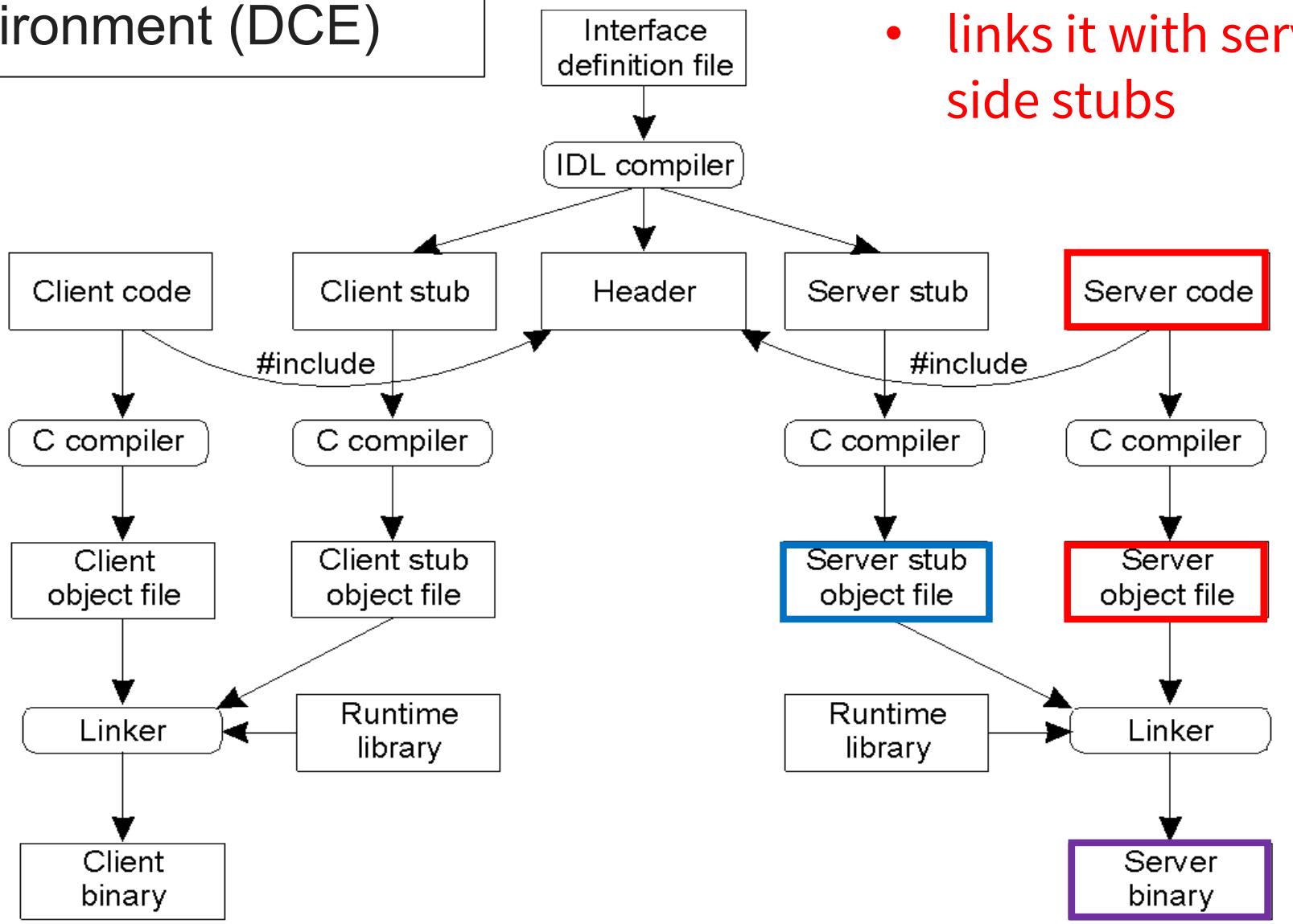


Example RPC system:

Distributed Computing Environment (DCE)

Server writer:

- writes server
- links it with server-side stubs



Binding: Connecting Client & Server

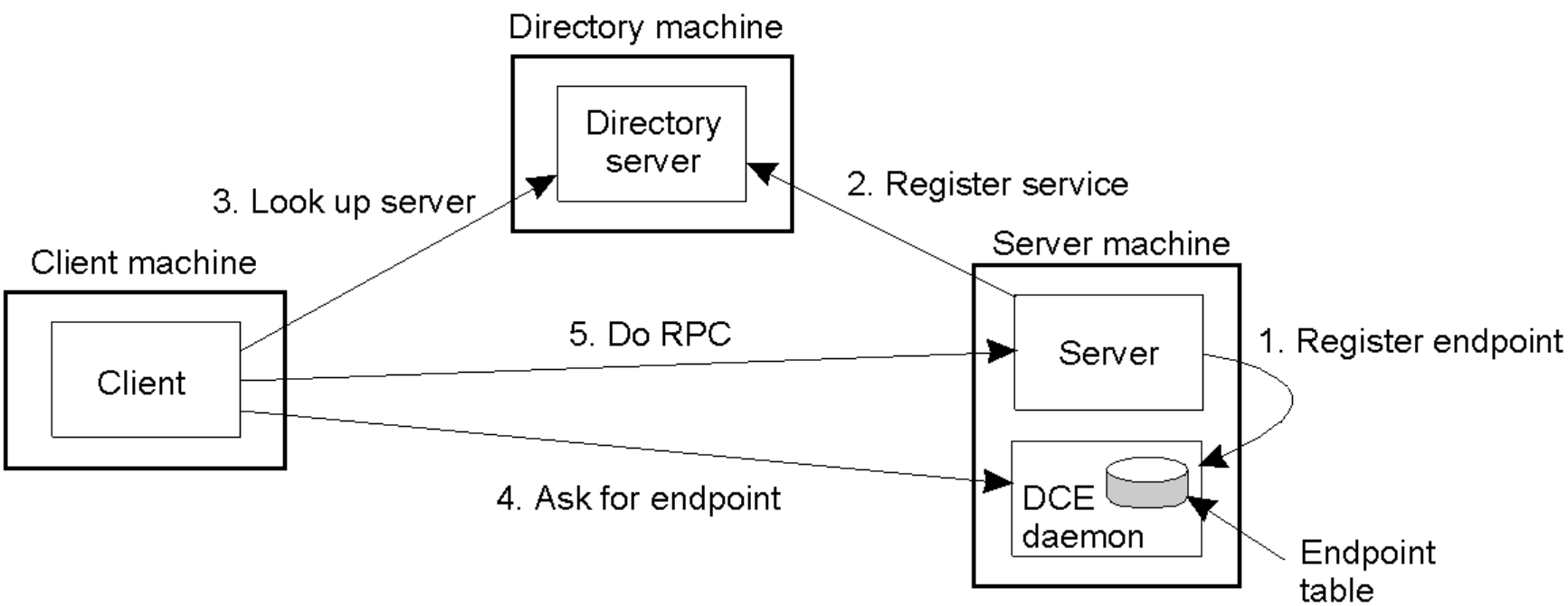
Server *exports* its interface:

- identifying itself to a network name server
- telling the local runtime its dispatcher address

Client *imports* the server. RPC runtime:

- looks up the server through the name service
- contacts requested server to set up a connection

Import and *export* are explicit calls in the code



RPC Concerns

- Parameter Passing
- Failure Cases
- Performance

Your function call has been secretly replaced with a remote function call. Is this okay?

RPC Marshalling

Packing parameters into a message packet

- RPC stubs call type-specific procedures to marshall (or unmarshall) all of the parameters to the call

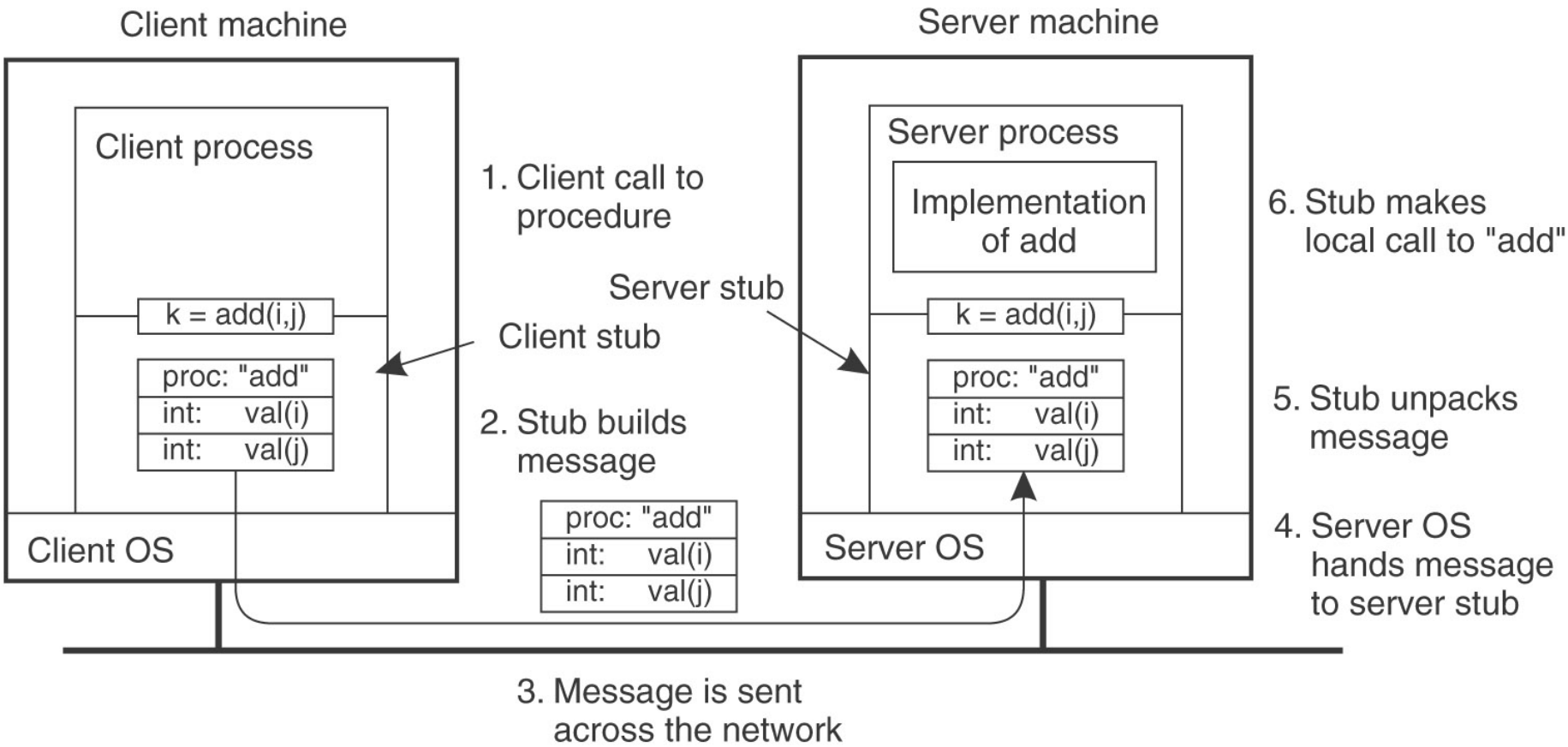
On Call:

- Client stub marshalls parameters into the call packet
- Server stub unmarshalls parameters to call server's fn

On return:

- Server stub marshalls return values into return packet
- Client stub unmarshalls return values, returns to client

Parameter Passing



What could go wrong?

[Tanenbaum & van Steen, Fig 4-7]

RPC Concerns

- Parameter Passing
  - Data Representation
  - Passing Pointers
  - Global Variables
- Failure Cases
- Performance

Data Representation

Data representation?

ASCII vs. Unicode, structure alignment, n-bit machines, floating-point representations, endianness

→Server program defines interface using an *interface definition language* (IDL)

For all client-callable functions, IDL specifies:

- names
- parameters
- types

Passing Pointers

- Forbid pointers? (breaks transparency)
- Have server call client and ask it to modify when needed (breaks transparency)
- Have stubs replace call-by-reference semantics with Copy/Restore
  - Optimization: if stub knows that a reference is exclusively input/output copy only on call/return
  - Only works for simple arrays & structures
    - Union types? YUCK
    - Multi-linked structures? YUCK
    - Raw pointers? YUCK

RPC Concerns

- Parameter Passing
- Failure Cases
- Performance

RPC Failure Cases

Function call failure cases:

- Called fn crashes → so does the caller

RPC Failure cases:

- server fine, client crashes? (orphans)
- client fine, server crashes?
  - Client just hangs?
  - Stub supports a timeout, error after n tries?
  - Client deals w/failure (breaks transparency)

Aside: Idempotency

Multiple calls yields the same result

What's idempotent?

- read block 50

What's not?

- appending a file
- most I/O

How many times will a function be executed?

A calls B. B never responds… Should A resend or not?

2 Possibilities:

(1) B never got the call:

- Resend → B executes the procedure *once*
- Don't resend → B executes the procedure *zero times*

(2) B performed the call then crashed:

- Resend → B executes the procedure *twice*
- Don't resend → B executes the procedure *once*

Can we even promise transparency?

What semantics will RPC support?

A calls B. B responds… What does A assume about how many times the function was executed?

Exactly once:

- system guarantees local semantics
- at best expensive, at worst, impossible

At-least-once:

+ easy: no response? A re-sends
− only works for idempotent functions
− server operations must be stateless

At-most-once:

− requires server to detect duplicate packets
+ works for non-idempotent functions

RPC Concerns

- Parameter Passing
- Failure Cases
- Performance
  - Remote is not cheap
  - Lack of parallelism (on both sides)
  - Lack of streaming (for passing data)

RPC Concluding Remarks

RPC:

- Common model for distributed application communication
- ***language support*** for distributed programming
- relies on a *stub compiler* & IDL server description
- commonly used, *even on a single node*, for communication between applications running in different address spaces (most RPCs *are* intra-node!)

"Distributed objects are different from local objects, and keeping that difference visible will keep the programmer from forgetting the difference and making mistakes."
–Jim Waldo+, "A Note on Distributed Computing" (1994)