Networking

CS 4410
Operating Systems

Cornell CIS
COMPUTING AND INFORMATION SCIENCE

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

- Introduction
- Application Layer
- Transport Layer
- Network Layer
- Remote Procedure Calls

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

Example:

- HTTP “get” requests and responses

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

Application
Transport
Network
Link
Physical

Actual 5-Layer Internet Protocol Stack

OSI Layers

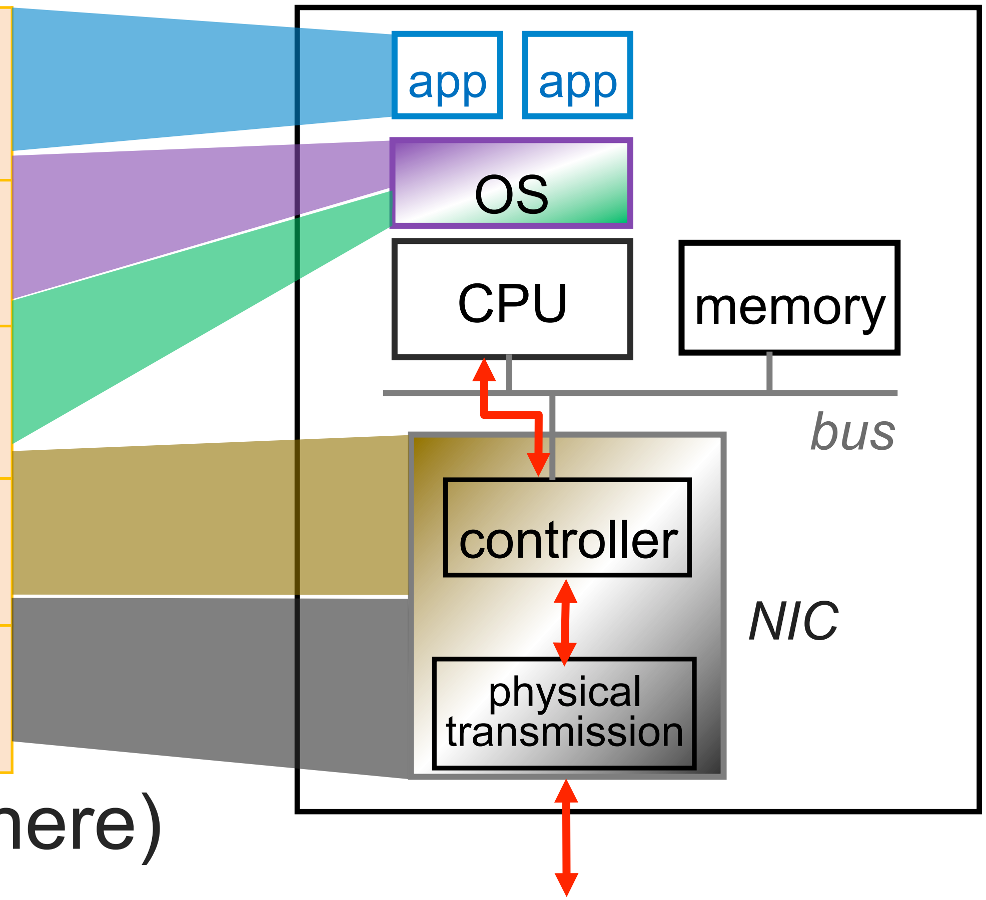
Application	Network-aware applications, clients & servers
Presentation	Translation between network and application formats (*e.g.*, RPC packages, sockets)
Session	Connection management
Transport	Data transfer, reliability, packetization, retransmission. Lets multiple apps share 1 network connection
Network	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

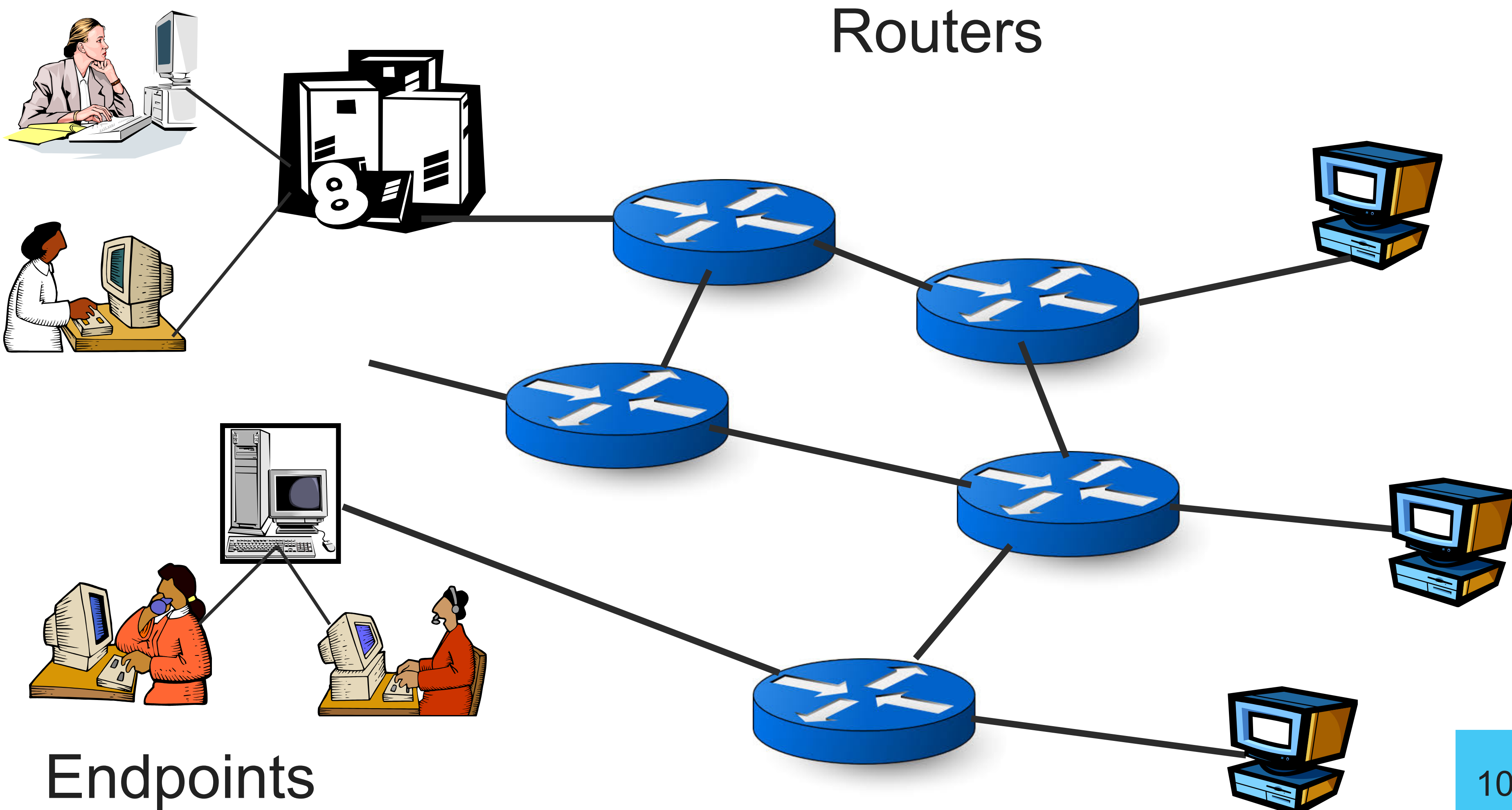
Can identify the relationship between distinct pieces of complex system.

Interfaces between layers:

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

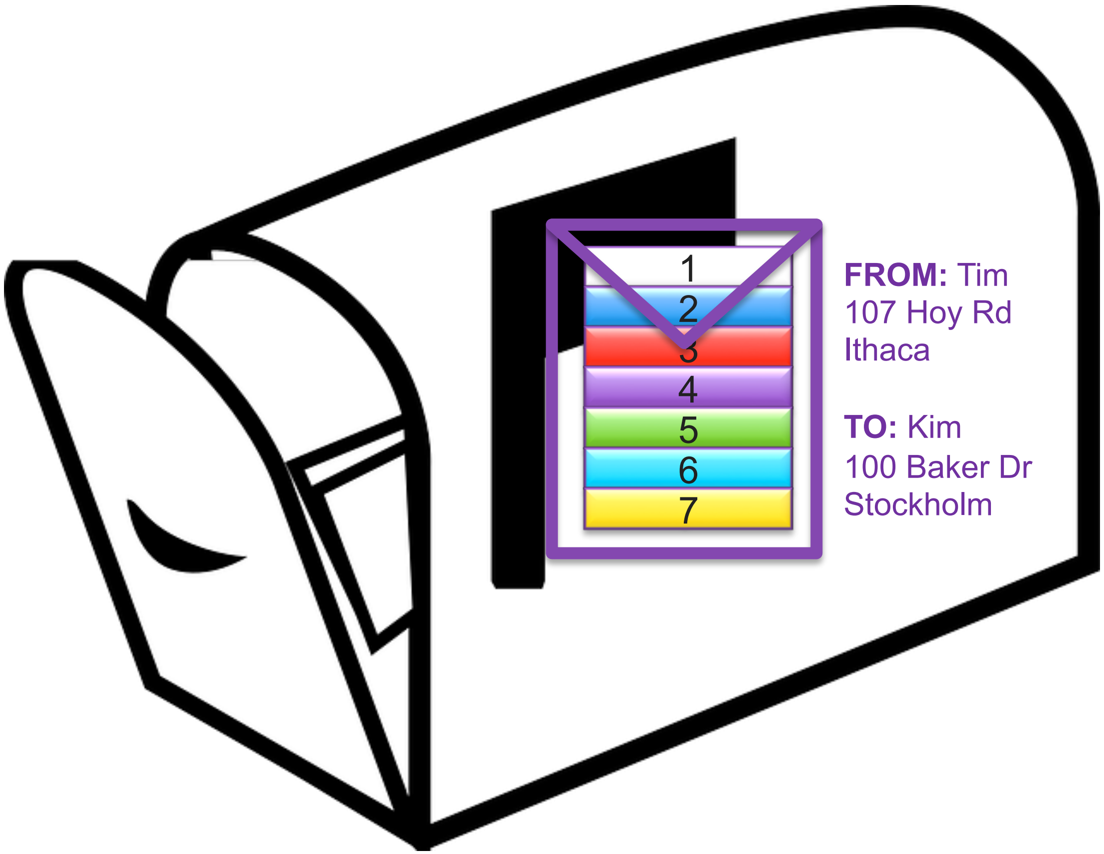
Internet, The Big Picture

How about an analogy?



Application Layer

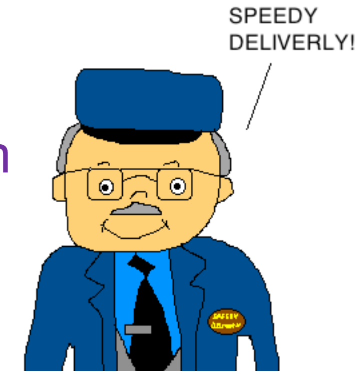
Tim → Kim

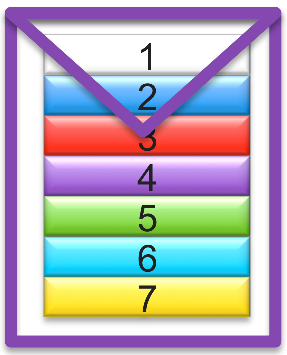


Analogy by Shubham Dubey on Quora

Transport Layer

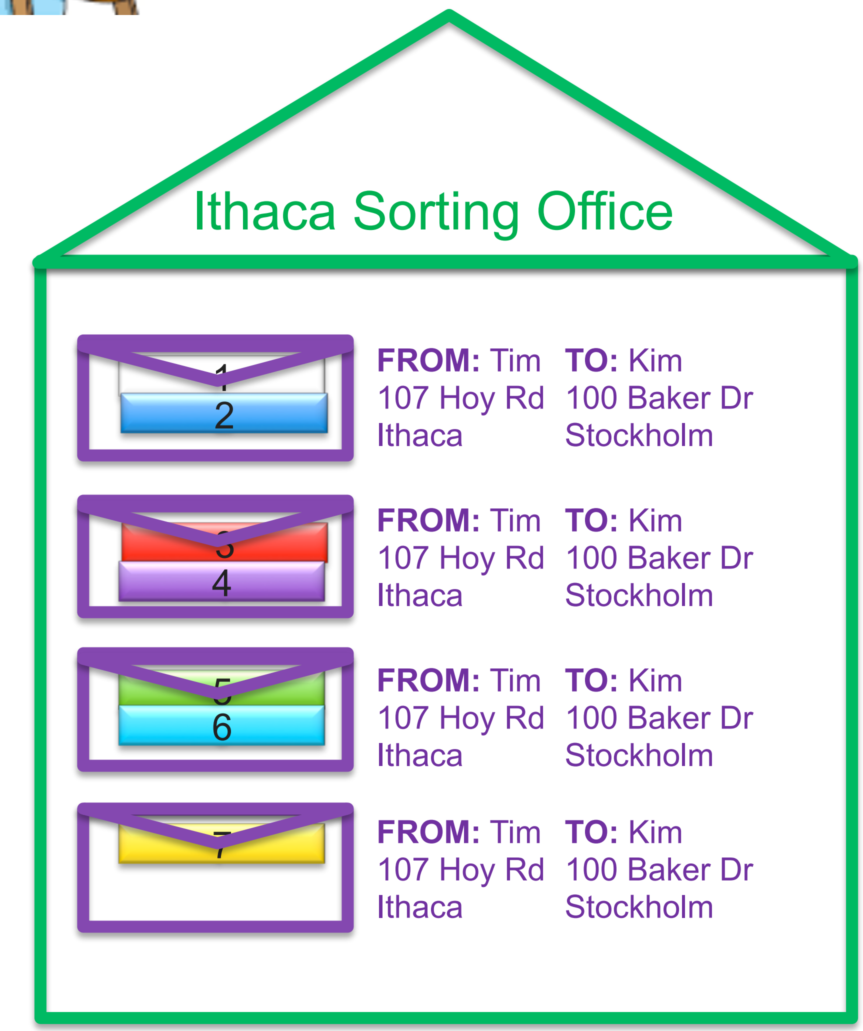
Ithaca Postman → Stockholm Postman





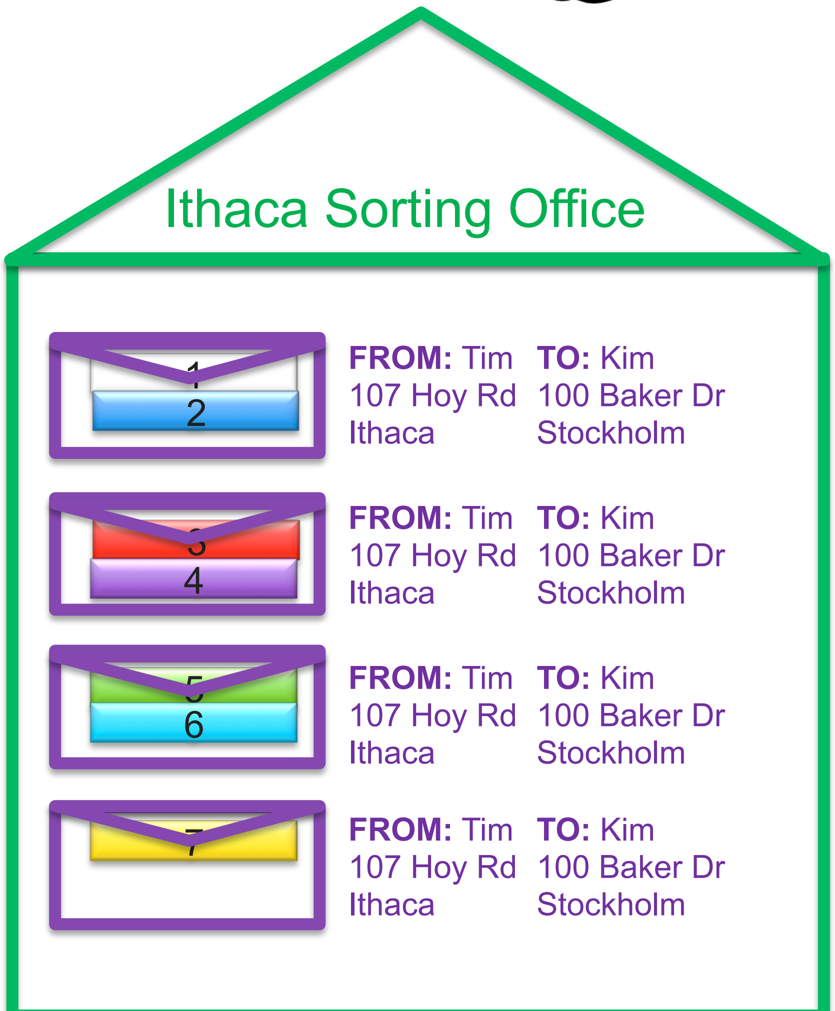
FROM: Tim
107 Hoy Rd
Ithaca

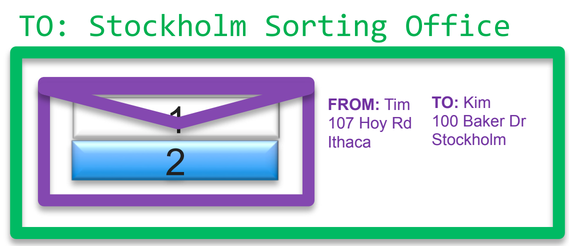
TO: Kim
100 Baker Dr
Stockholm

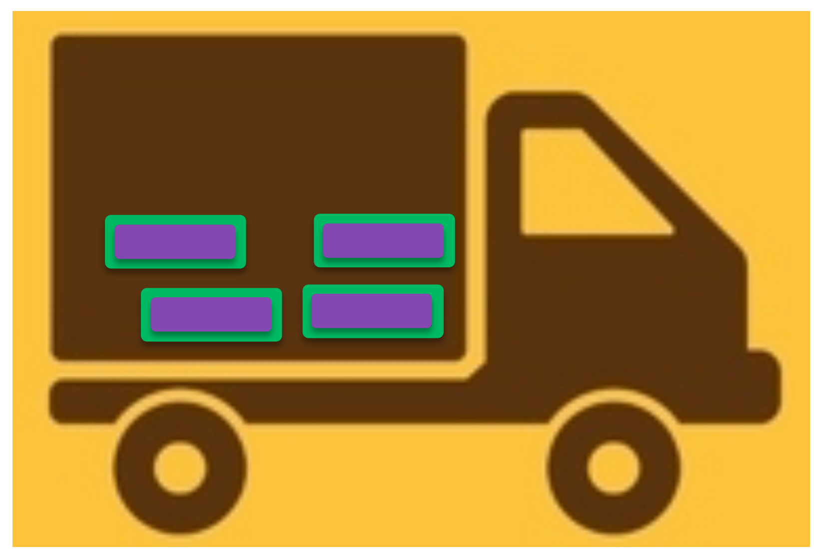


Network Layer

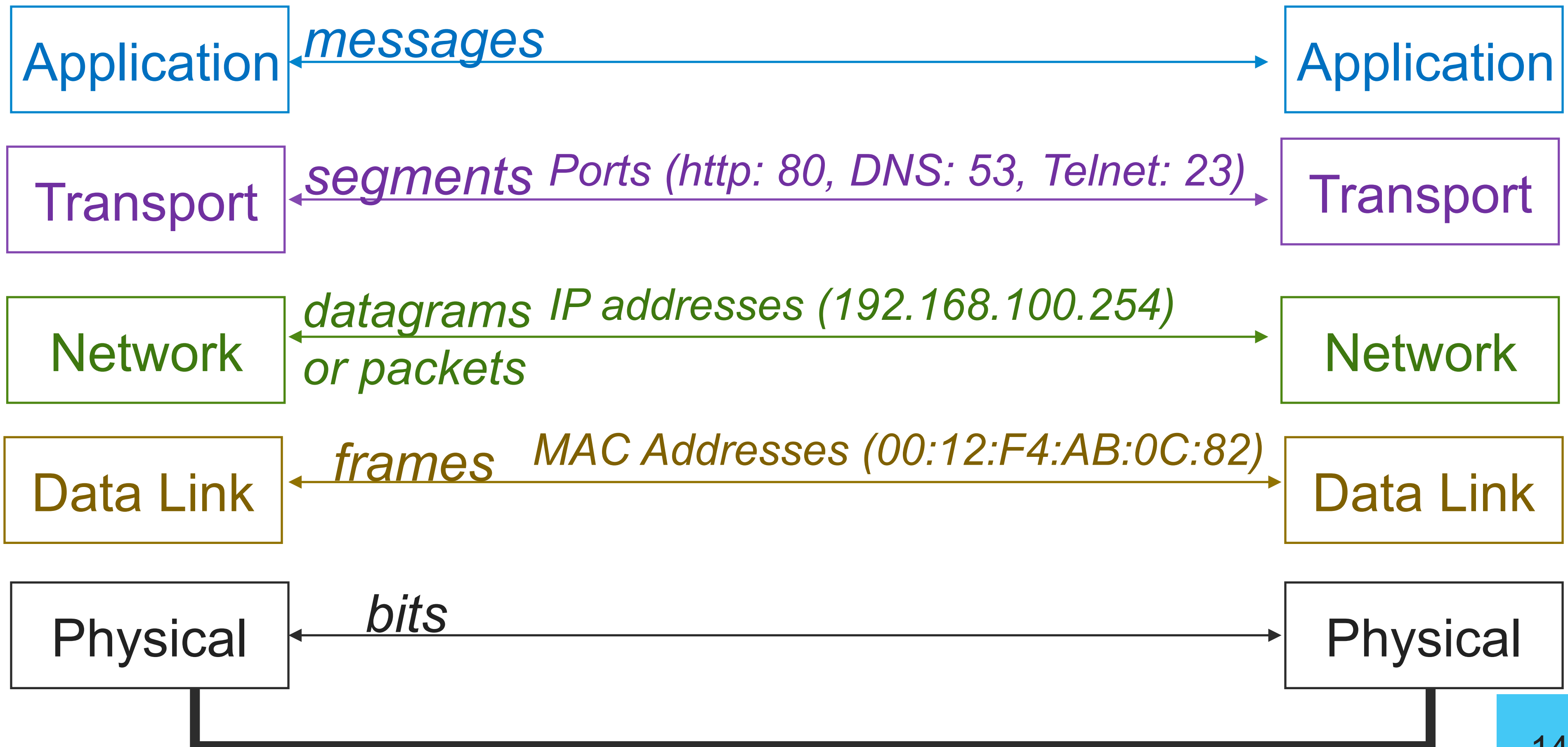
Ithaca Sorting Office → Stockholm Sorting Office



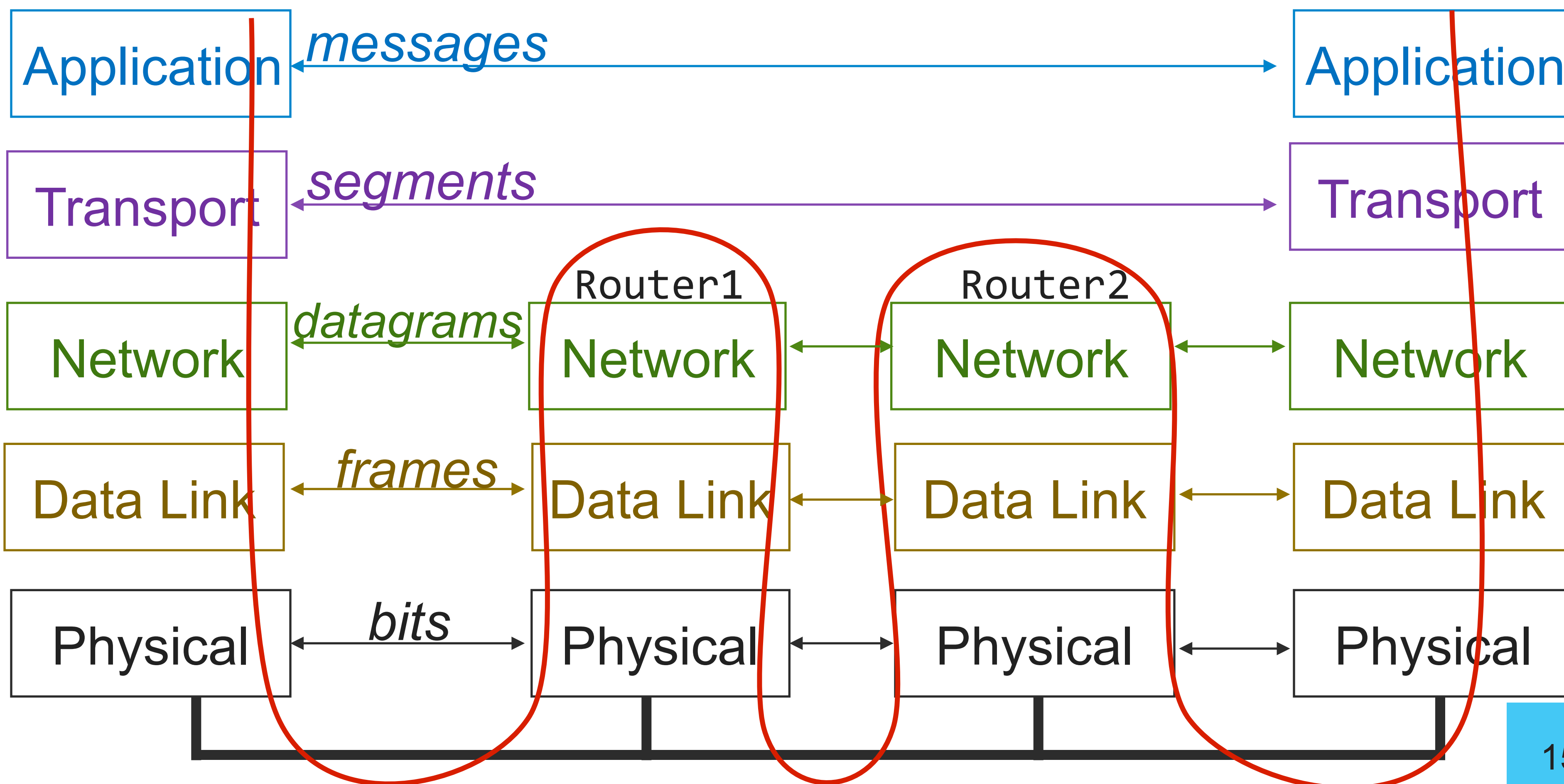




The Big Picture

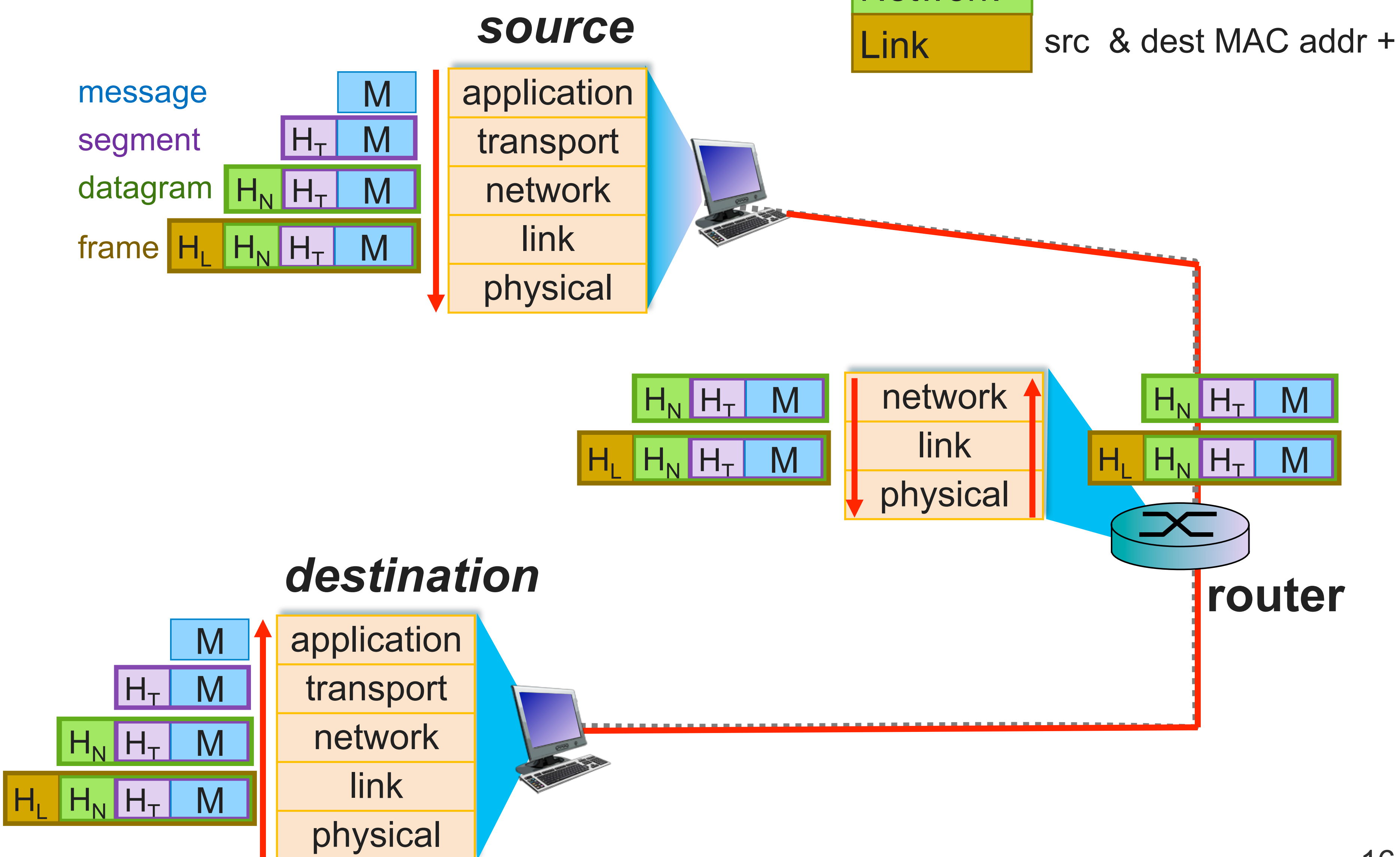


The Big Picture



Encapsulation

Headers	
Transport	src & dst ports + …
Network	src & dest IP addr + …
Link	src & dest MAC addr +



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?

Some issues…

- How do endpoints find each other?
- What does a message look like?
- Can messages be lost? large? jumbled?

The Missing Layers

Presentation

translation between network & application formats (e.g., RPC packages, sockets). Allows communicating applications to interpret the meaning of data exchanged:

- data conversion
- character code translation
- compression
- encryption

Session

synchronization of data exchange:

- supports checkpointing and recovery schemes
- establish, manage, and tear down connections

Need these services?
Put them in your application.

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

Internet Overview

- Every host is assigned, and identified by, an IP address
- Each packet contains a header that specifies the destination address
- The network routes the packets from the source to the destination

Naming

People

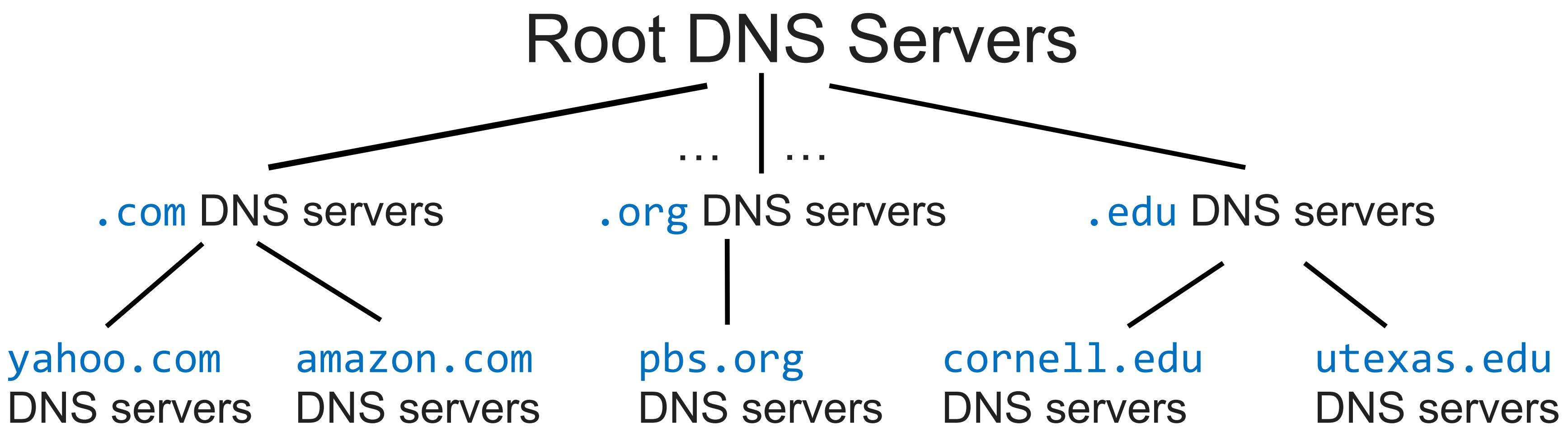
- SSN, NetID, Passport #

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

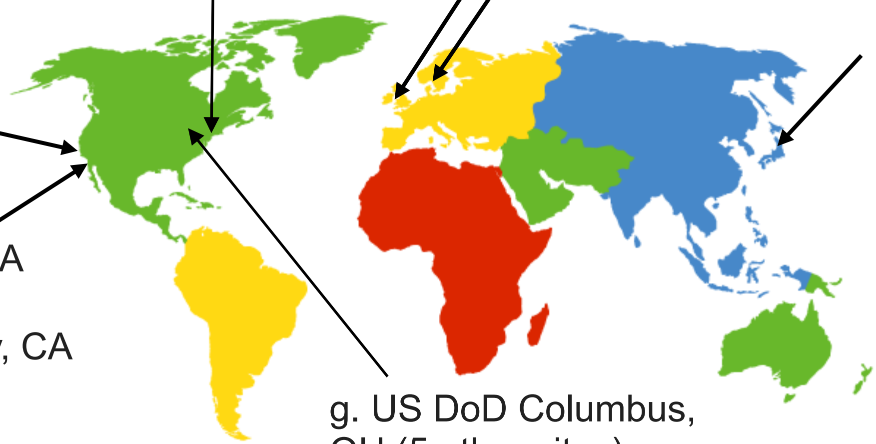
  Not to be confused with dots in IP addresses (in which the order of least significant to most significant is reversed)

- Components resolved from right to left
- All siblings must have unique names
- Lookup occurs from the top down

DNS: root name servers

Contacted by local name server that cannot 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



13 root name "servers" worldwide

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

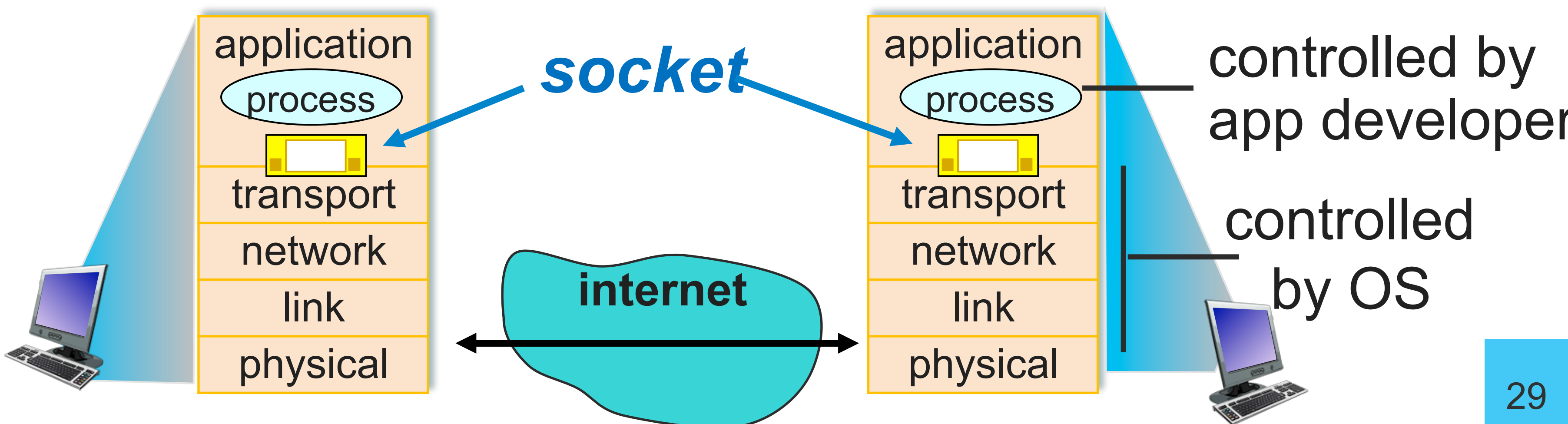
- Network-aware applications
  - Clients & Servers
  - Peer-to-Peer

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

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 clientsock

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()
```

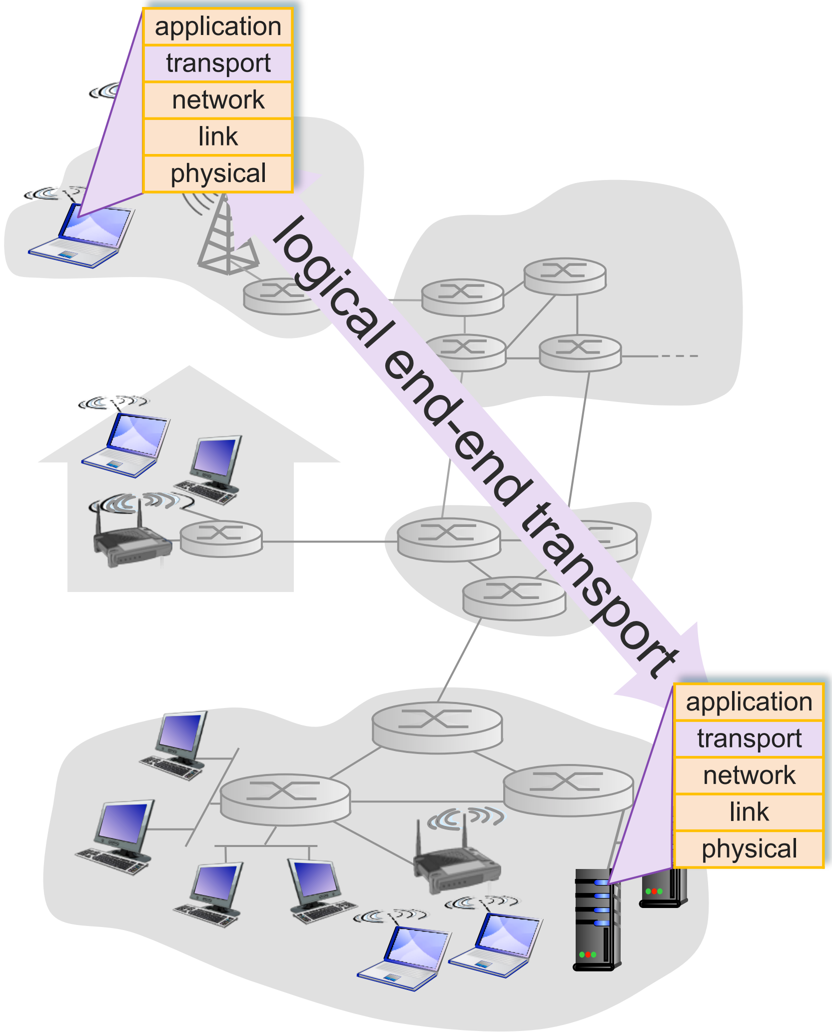
Application
Transport
Network
Link
Physical

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

Transport Layer Analogy

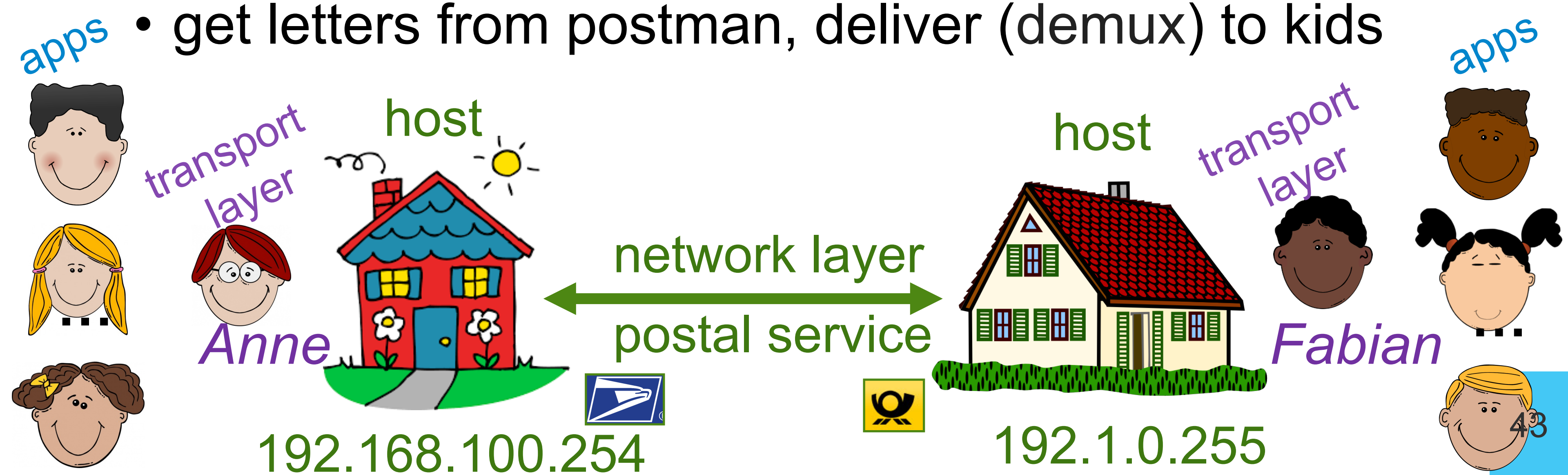
2 houses (**hosts**), each has 12 kid siblings

Kids: (applications)

- write letters (messages) to cousins

Parents: (transport layer protocol)

- gather the letters (multiplexing)
- put them in addressed envelopes (segments)
- give them to the postman (network layer)
- get letters from postman, deliver (demux) to kids

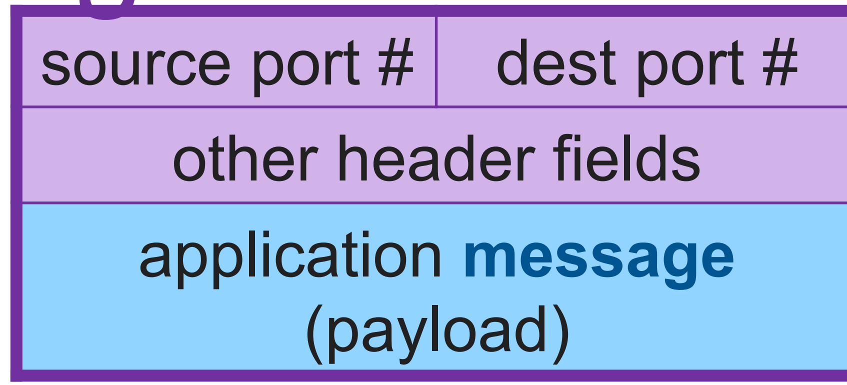


How to create a segment

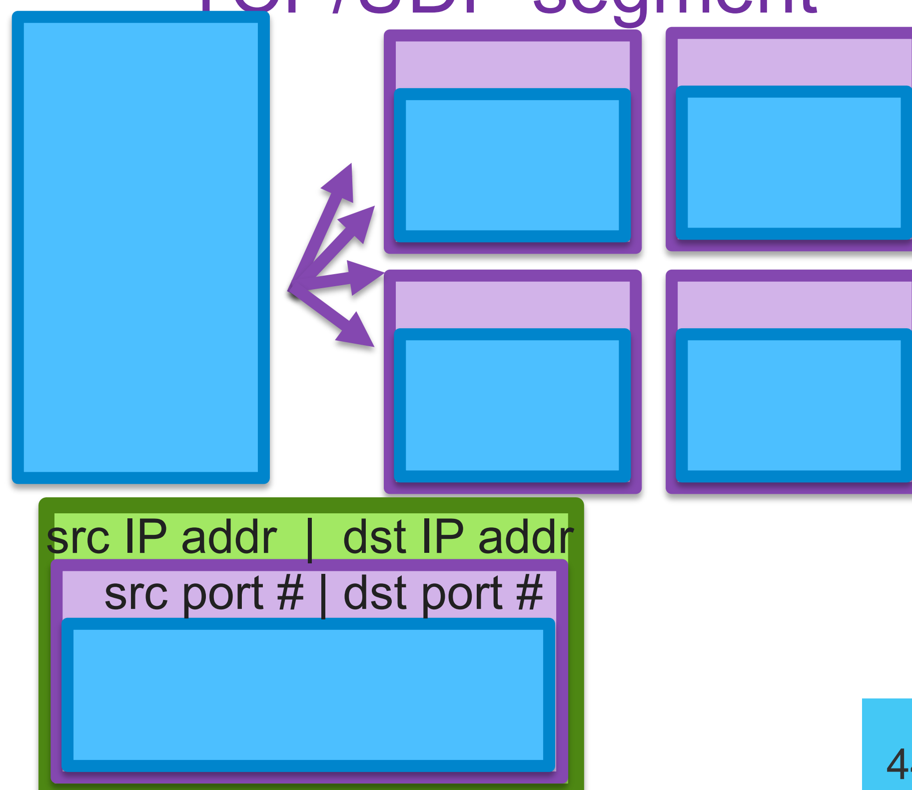
Sending application:
- specifies IP address and destination port
- uses socket bound to a source port

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

Network Layer:
- adds network-layer header (with IP address)

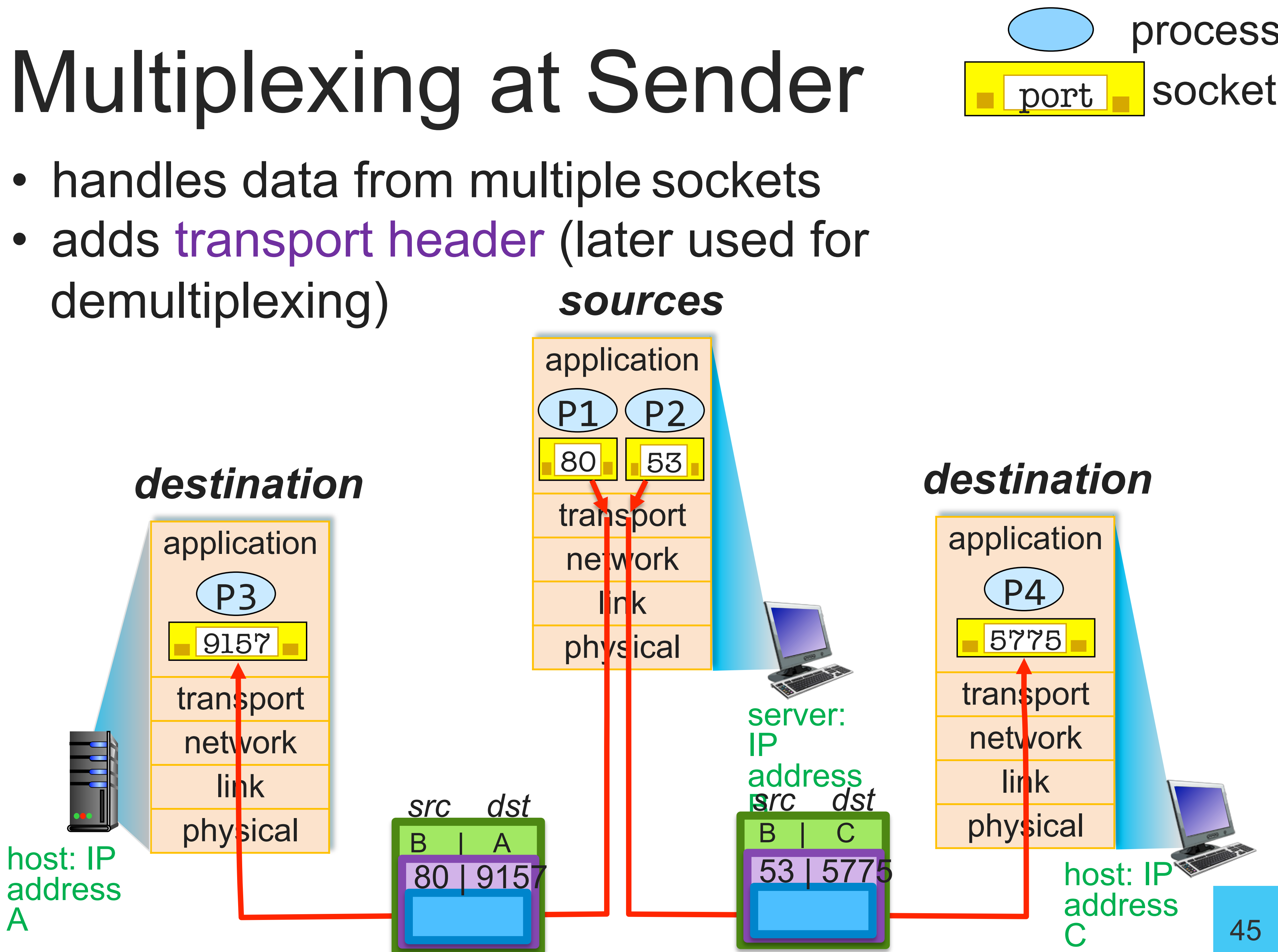


TCP/UDP segment

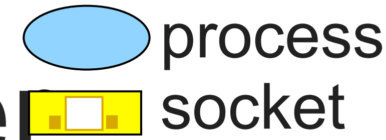


Multiplexing at Sender

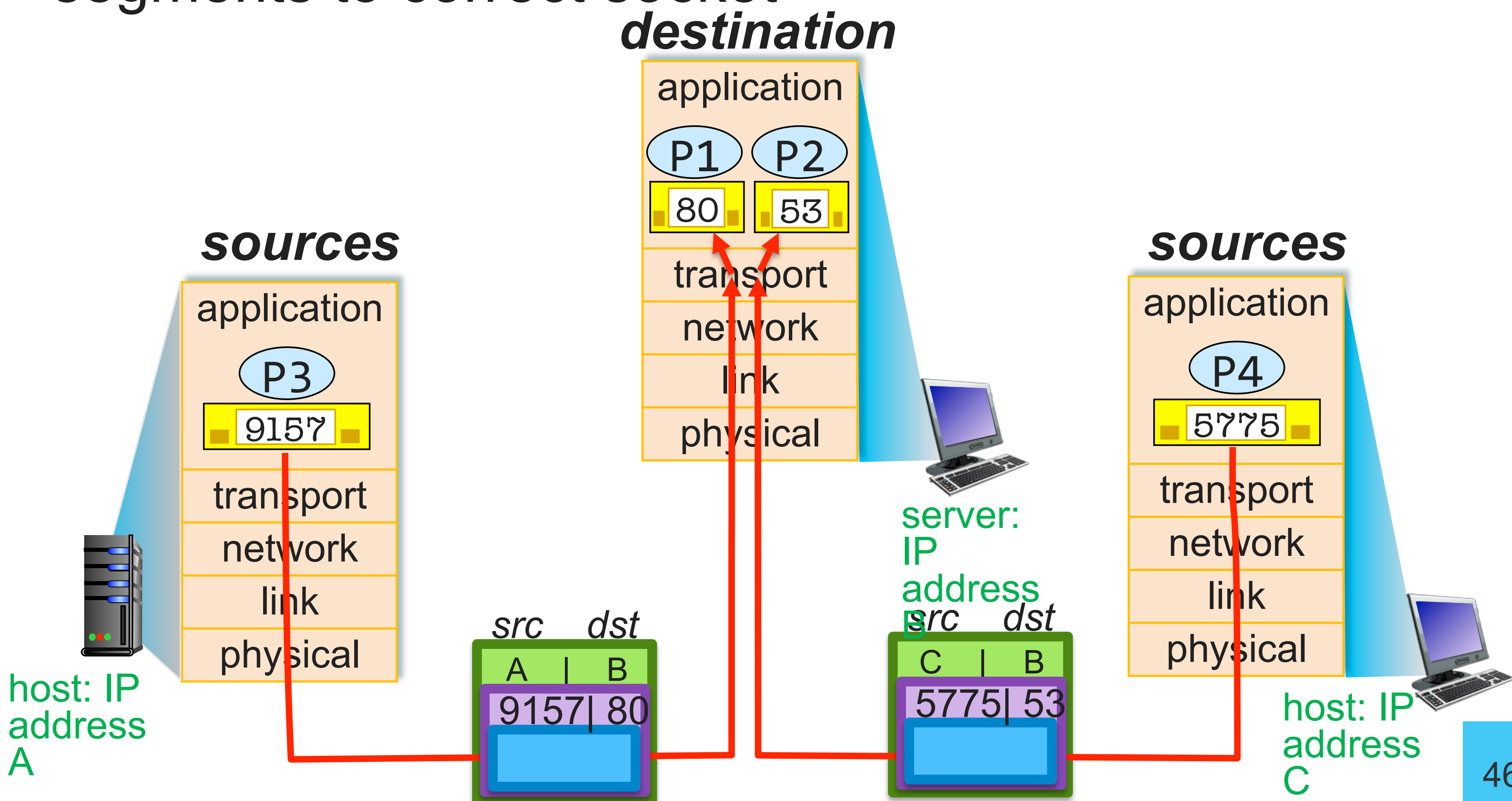
- handles data from multiple sockets
- adds transport header (later used for demultiplexing)



Demultiplexing at Receiver



- 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!

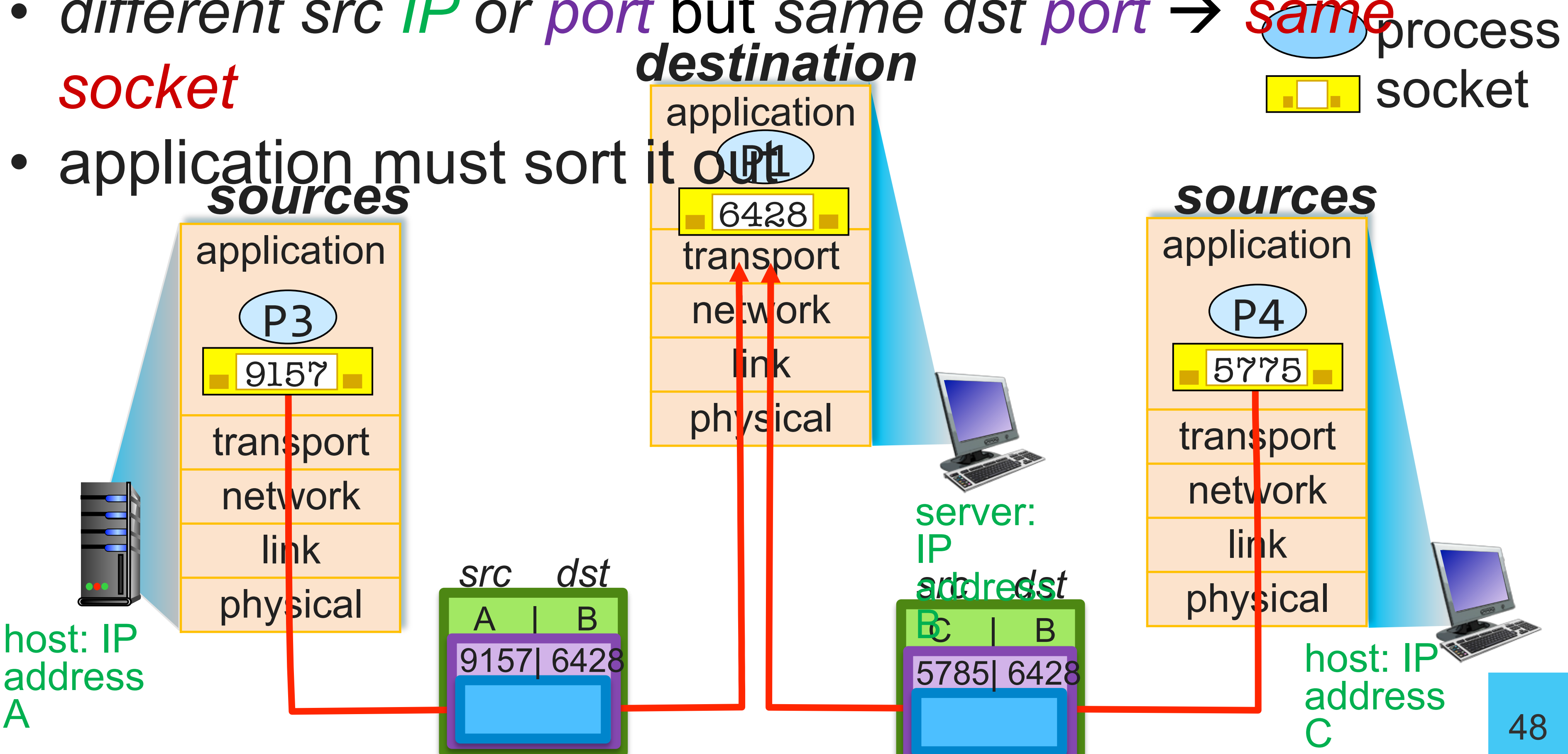
I was gonna tell you guys a joke about UDP…
But you might not get it

I was you guys about UDP might not

Connectionless demux: example

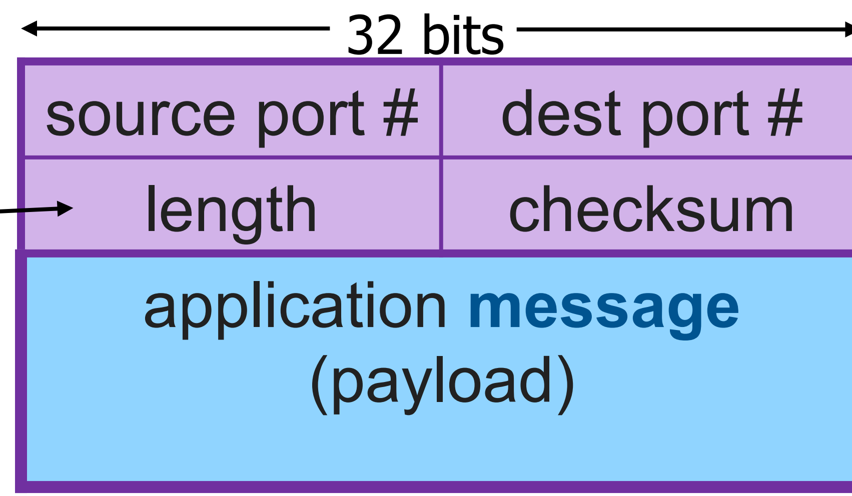
Host receives 2 UDP segments:

- 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

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

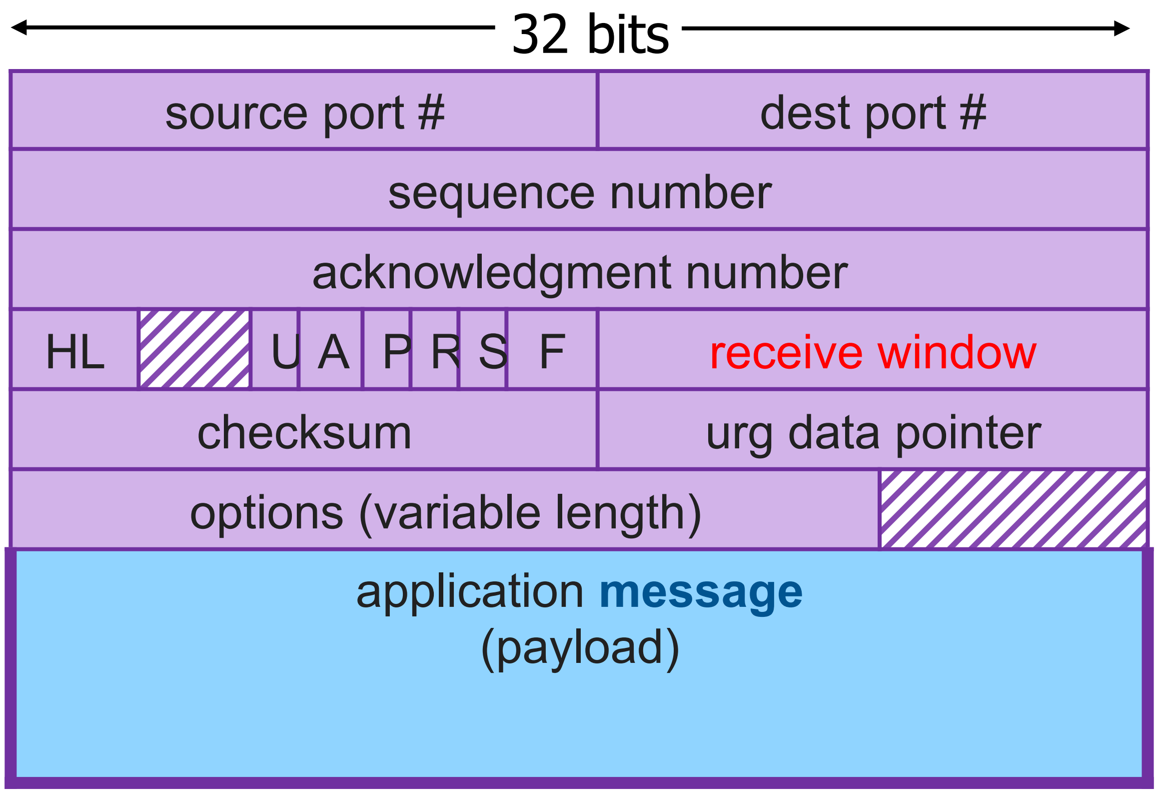
U: urgent data

A: ACK # valid

P: push data now

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

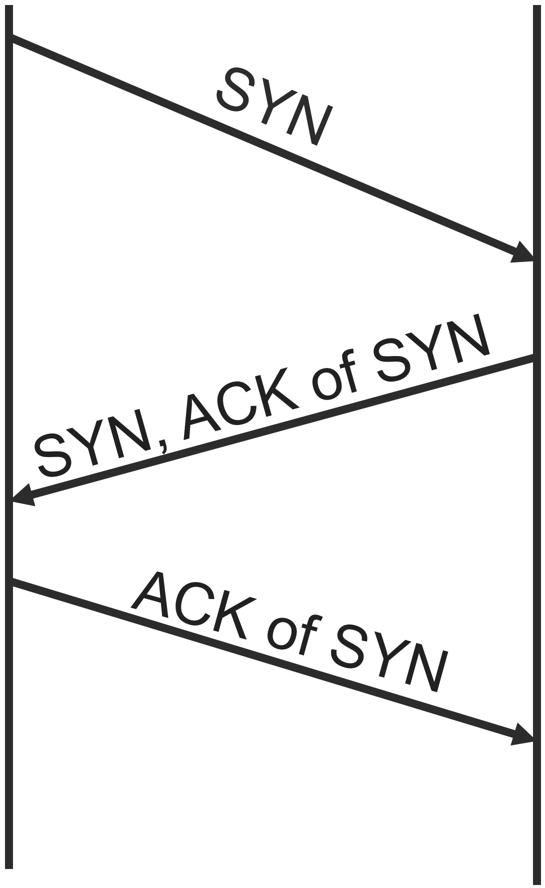
bytes receiver willing to accept



TCP header size: 20-60 bytes

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

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 = Acknowledgment

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

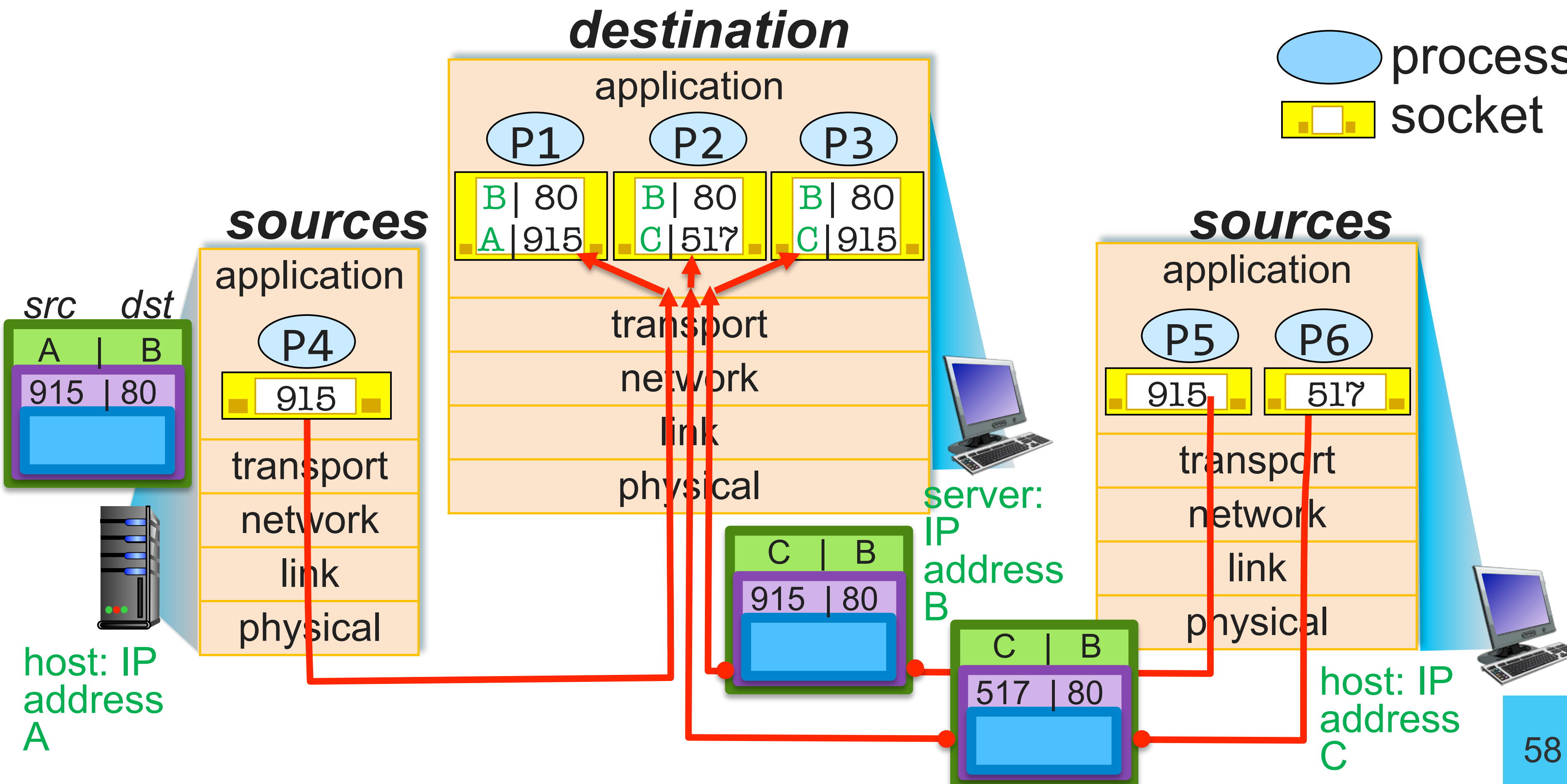
- source IP address
- source port number
- dest IP address
- dest port number

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

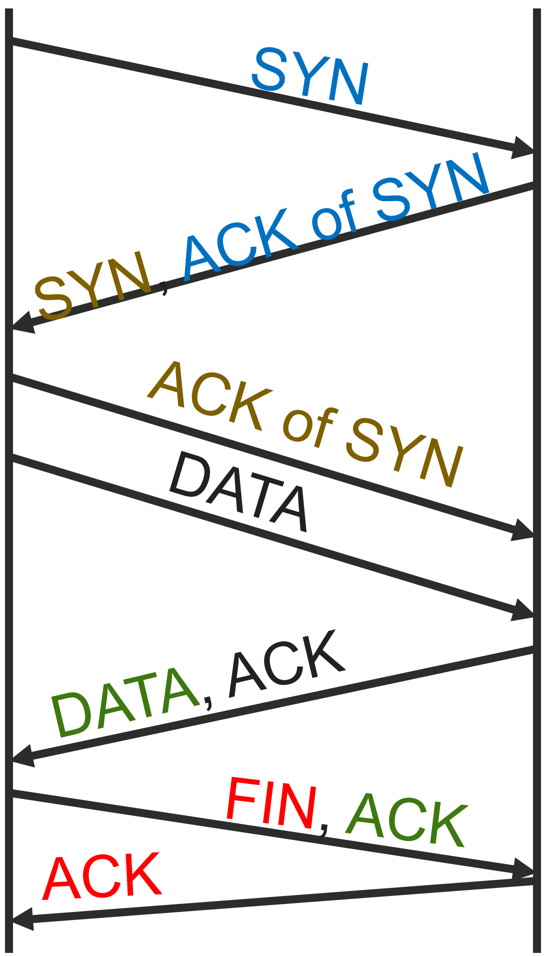
  4410 simplification: just increment by 1

Each packet carries an **ack**nowledgment

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

Reliable transport is implemented using these identifiers

TCP Usage Pattern



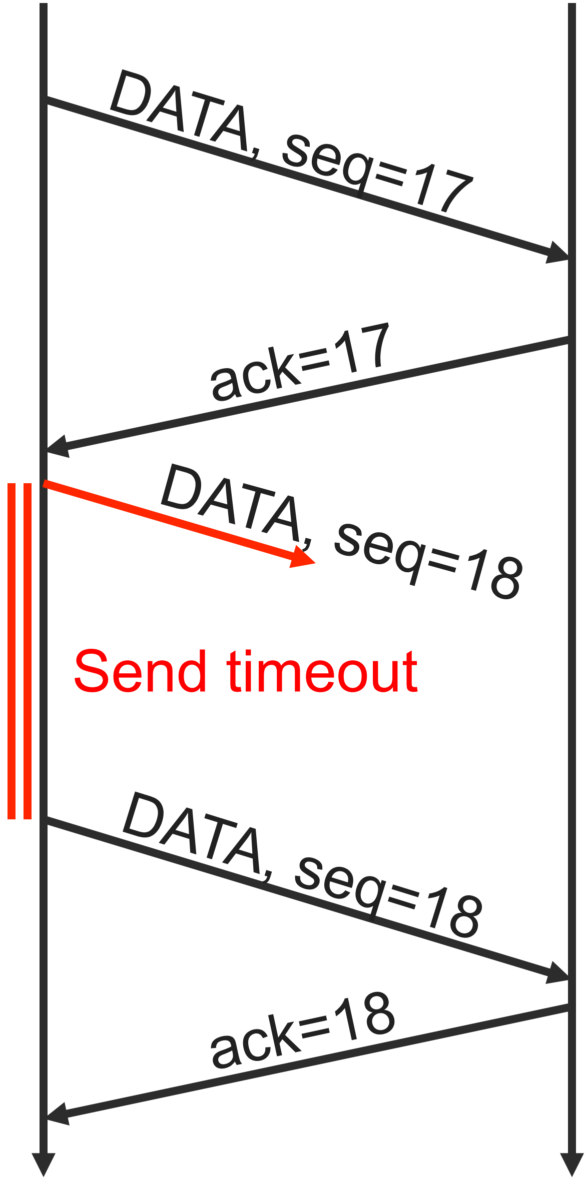
3 round-trips:

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 acknowledgment 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?

How long does it take to send a segment?

- S: size of segment in bytes
- L: one-way latency in seconds
- B: bandwidth in bytes per second

- Then the time between the start of sending and the completion of receiving is about L + S/B seconds (ignoring headers)
- And another L seconds (total: 2L + S/B) before the acknowledgment is received by the sender
  - assuming ack segments are small
- The resulting end-to-end throughput (without pipelining) would be about S / (2L + S/B) bytes/second

TCP timeouts

What is a good timeout period ?

– Goal: improve throughput without unnecessary transmissions

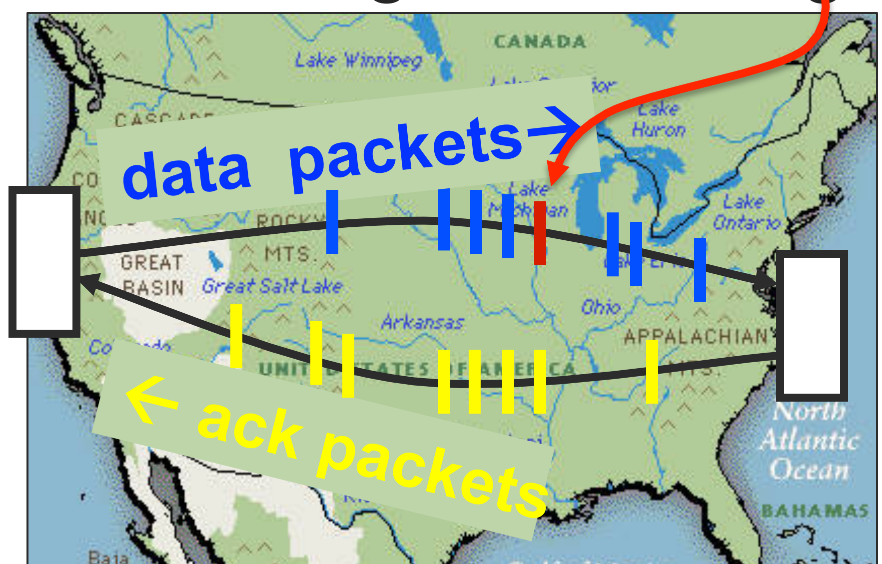
```
NewAverageRTT = (1 - α) OldAverageRTT + α LatestRTT
NewAverageVar = (1 - β) OldAverageVar + β LatestVar
where LatestRTT = (ack_receive_time – send_time),
           LatestVar  = |LatestRTT – AverageRTT|,
           α = 1/8, β = ¼ typically.
Timeout = AverageRTT + 4*AverageVar
```

→ 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?

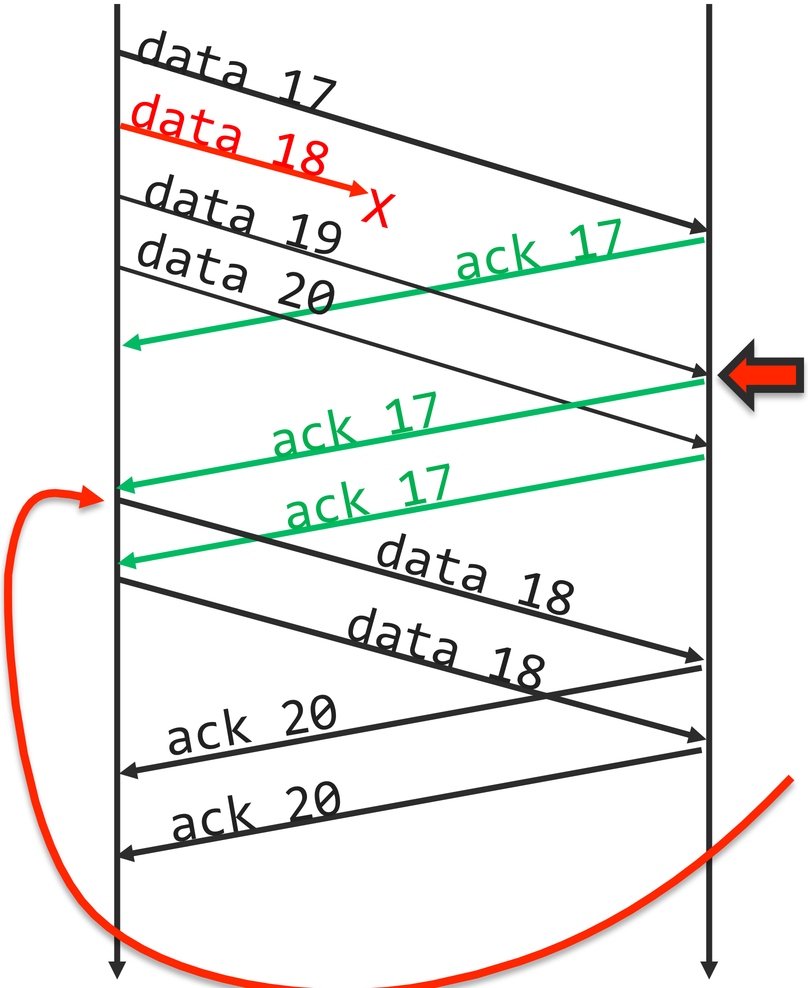
Suppose:

- 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…)

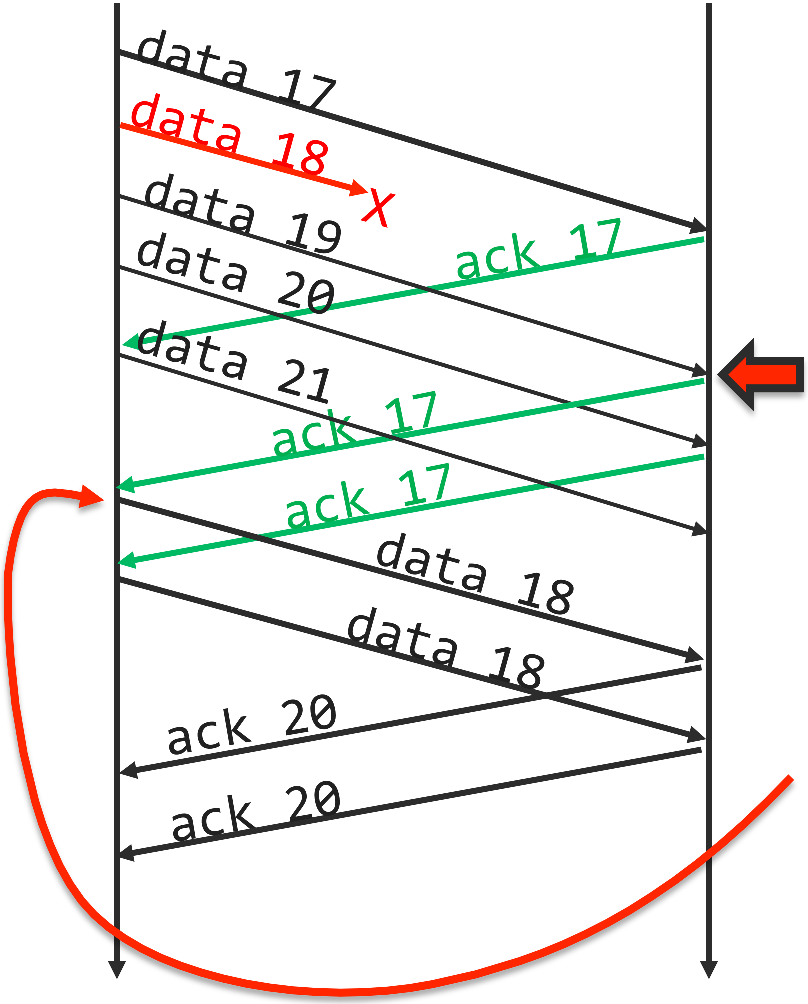
TCP **Fast Retransmit**



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 **Fast Retransmit**



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 acknowledgments

→ slowly builds up to max bandwidth, and hover there

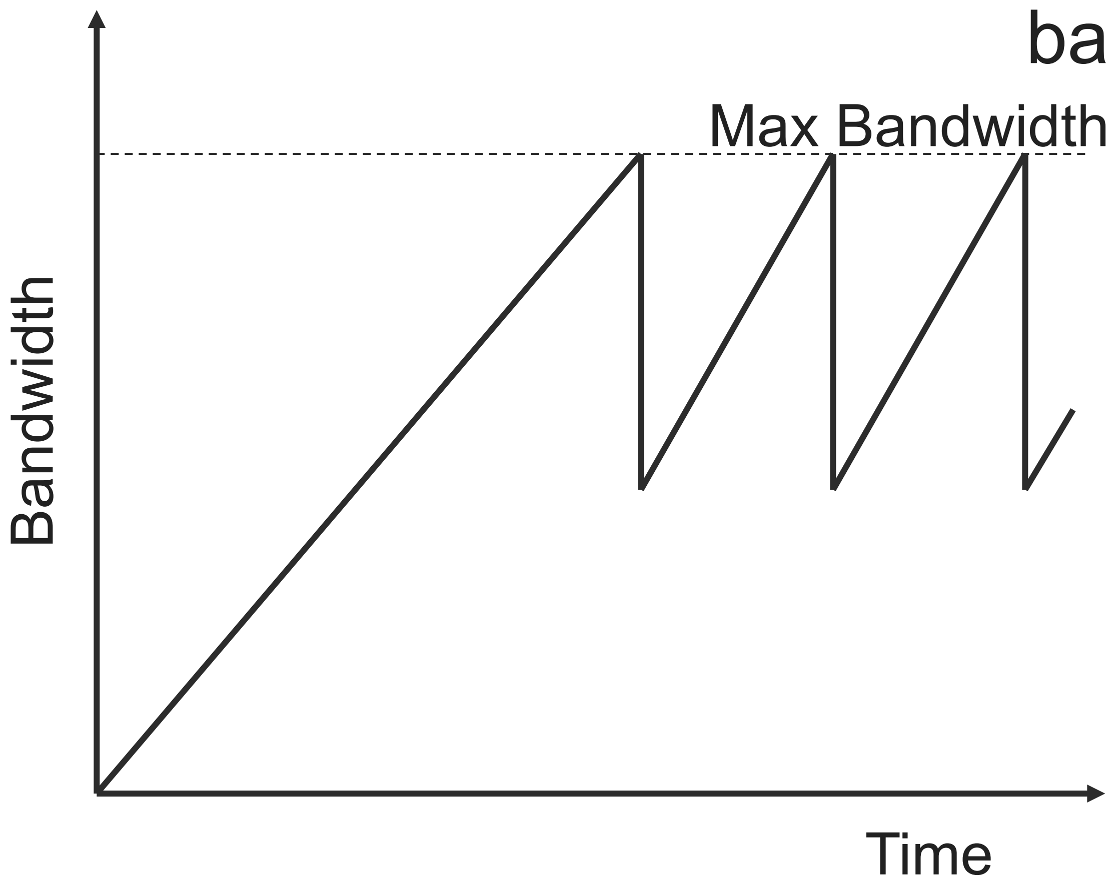
- – Does not achieve the max possible
- \+ Shares bandwidth well with other TCP connections

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)

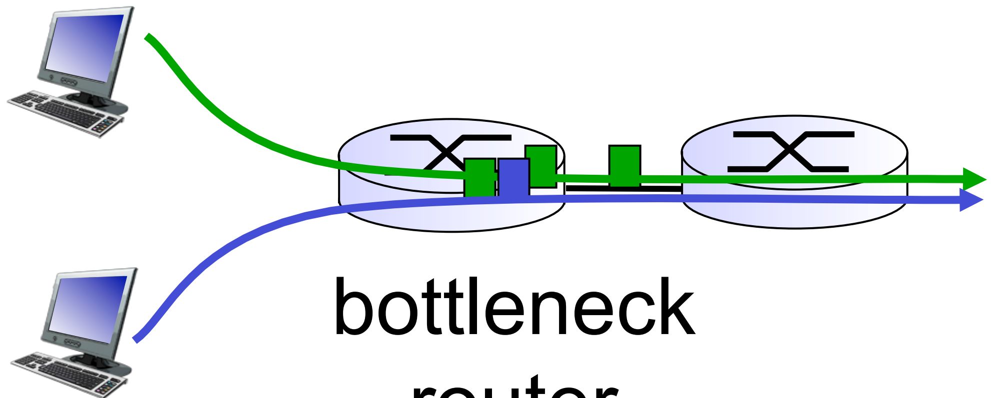


<u>Window Sizes:</u>
1,2,3,4,5,6,7,8,9,
10,5,6,7,8,9,10,
5,6,7,8,9,10,
. . .

TCP Fairness

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

TCP connection 1



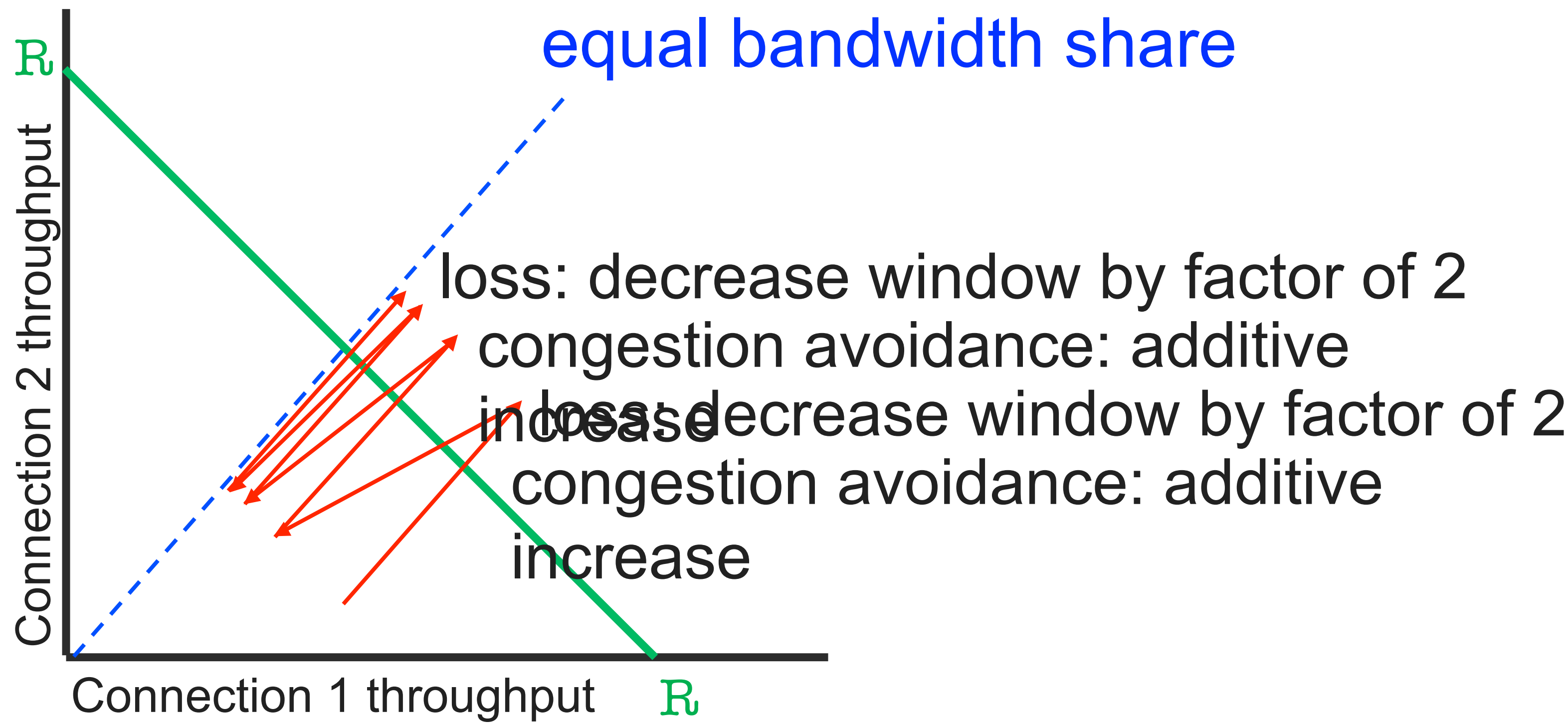
TCP connection 2

bottleneck router capacity `R`

Why is TCP fair?

Two competing sessions:

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



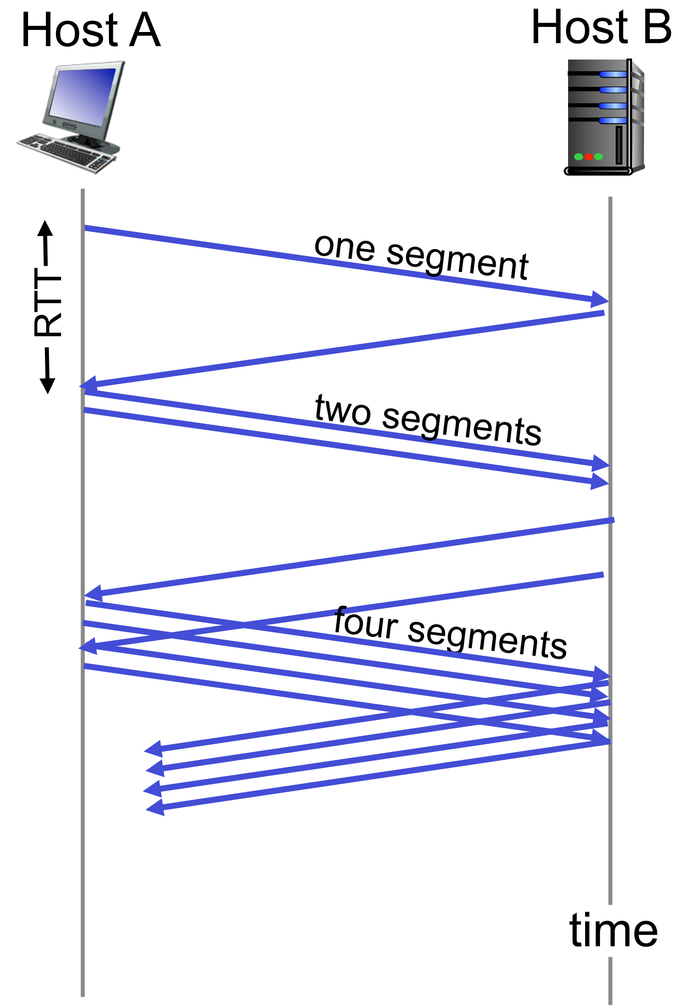
TCP Slow Start

(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

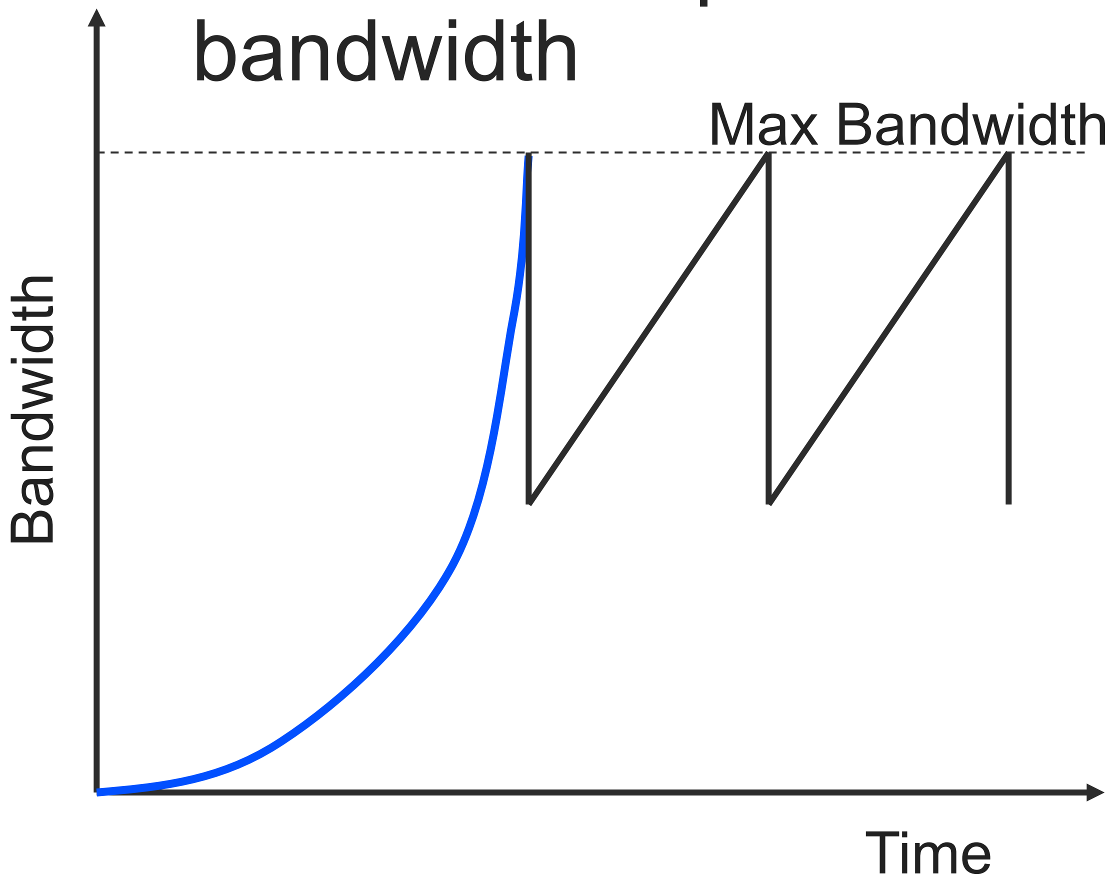


Solution: Allow TCP to increase window size by doubling *until first loss*

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



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 Layer
Transport Layer
Network Layer
Link Layer
Physical Layer

Link Layer:
Local Area Networking (LAN) and Ethernet

Link Layer

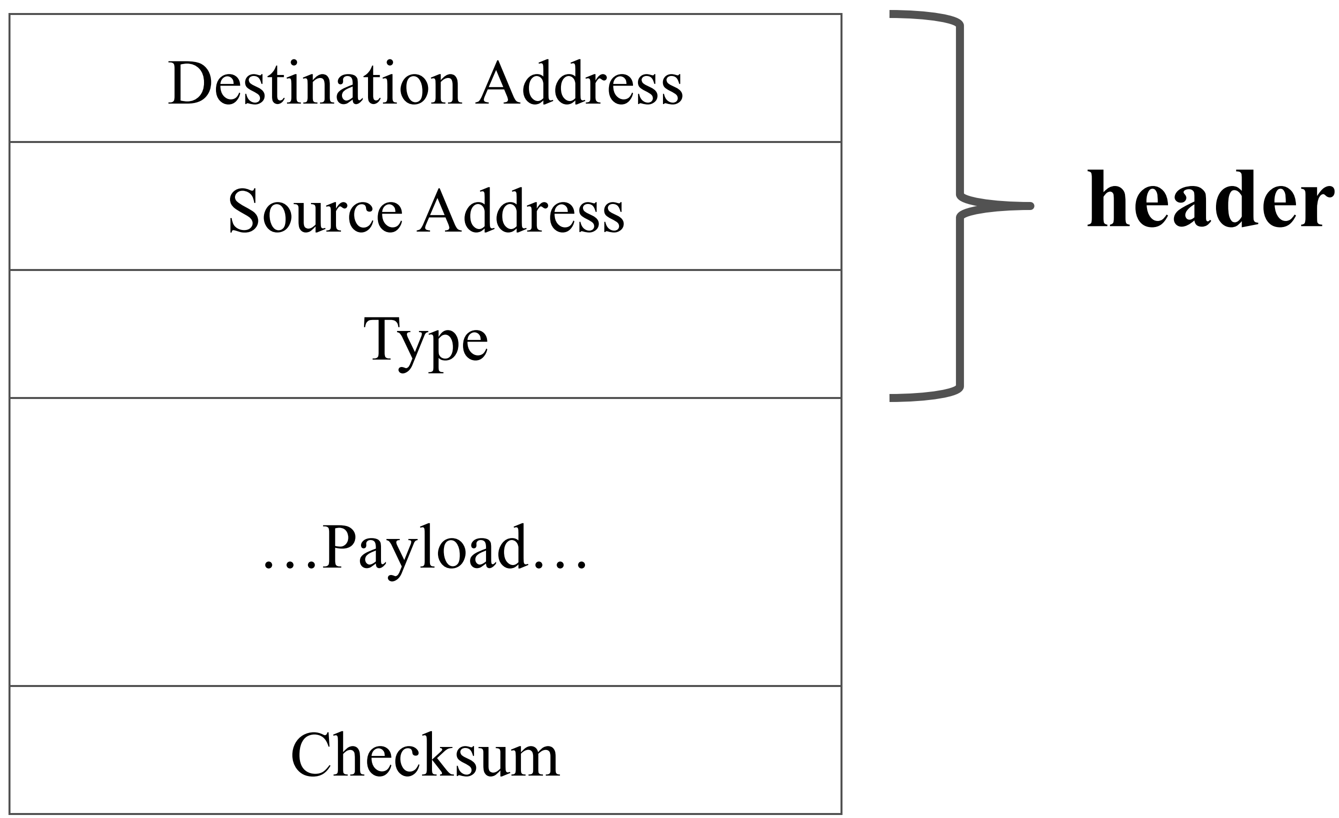
- Each host has one or more *NICs*
  - *Network Interface Cards*
    - Ethernet, 802.11, etc.
- Each NIC has a *MAC address*
  - *Media Access Control* address
  - Ethernet example: b8:e3:56:15:6a:72
  - Unique to network instance
    - often even globally unique
- Messages are *packets* or *frames*

Example: Ethernet

- 1976, Metcalfe & Boggs at Xerox
  - Later at 3COM
- Based on the Aloha network in Hawaii
- Named after the "*luminiferous ether*"
- Centered around a broadcast bus
- Simple link-level protocol, scales pretty well
- Tremendously successful
- Still in widespread use
  - many orders of magnitude increase in bandwidth since early versions

Ethernet basics

An Ethernet packet



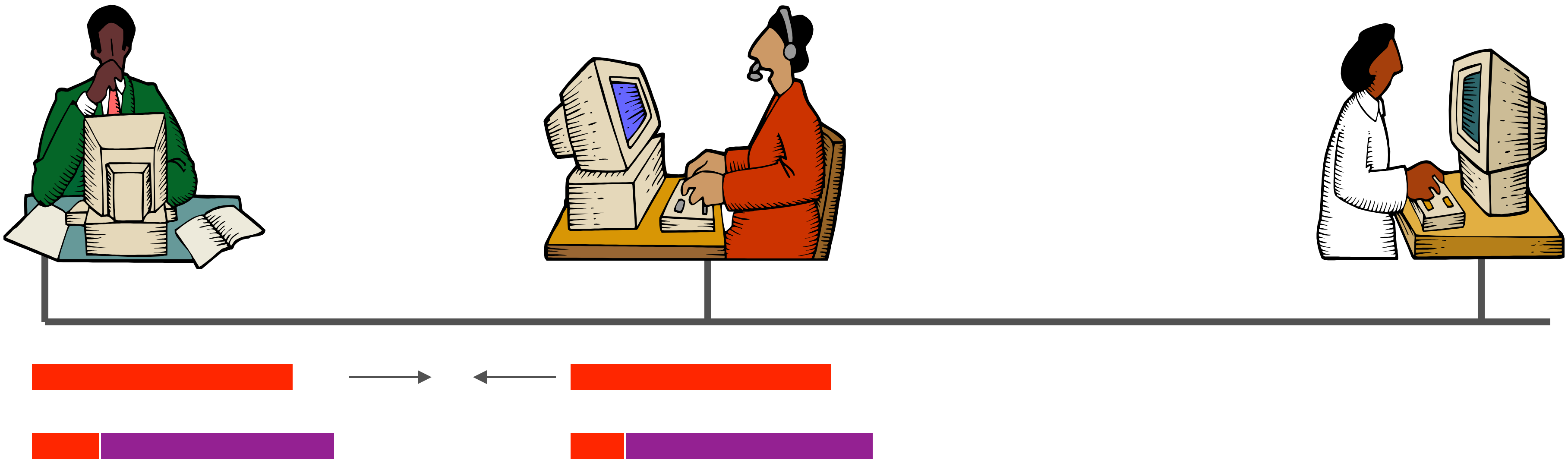
“CSMA/CD”

- Carrier sense
  - Listen before you speak
- Multiple access
  - Multiple hosts can access the network
- Collision detect
  - Detect and respond to cases where two hosts collide

Sending packets

- Carrier sense, broadcast if ether is available

Collisions



- What happens if two people decide to transmit simultaneously ?

Collision Detection & Retransmission

- The hosts involved in the collision stop data transmission, sleep for a while, and attempt to retransmit
- How long they sleep is determined by how many collisions have occurred before
- They abort after 16 retries, hence no guarantee that a packet will get to its destination

CRC Checksum

(Cyclic Redundancy Check)

- Basically a hash function on the packet
- Added to the end of a packet
- Used to detect malformed packets, e.g. electrical interference, noise

Ethernet Features

- Completely distributed
  - No central arbiter
- Inexpensive
  - No state in the network
  - No arbiter
  - Cheap physical links (twisted pair of wires)

Ethernet Problems

- The endpoints are trusted to follow the collision-detect and retransmit protocol
  - Certification process tries to assure compliance
  - Not everyone always backs off exponentially
- Hosts are trusted to only listen to packets destined for them
  - But the data is available for all to see
    - All packets are broadcast on the wire
    - Can place Ethernet card in promiscuous mode and listen

Switched Ethernet

- Today's Ethernet deployments are much faster
- In wired settings, *Switched Ethernet* has become the norm
  - All hosts connect to a switch
  - Each p2p connection is a mini Ethernet set-up
  - More secure, no possibility of snooping
  - Switches organize into a spanning tree
- Not to be confused with Ethernet *Hub*
  - A hub simply connects the wires

Wireless

- 802.11 protocols inherit many of the Ethernet concepts
- Full compatibility with Ethernet interface
  - Same address and packet formats

Lessons for LAN design

- Best-effort delivery simplifies network design
- A simple, distributed protocol can tolerate failures and be easy to administer

Network Layer

Application Layer
Transport Layer
Network Layer
Link Layer
Physical Layer

Network Layer

- There are lots of Local Area Networks
  - each with their own
    - address format and allocation scheme
    - packet format
    - LAN-level protocols, reliability guarantees
- Wouldn't it be nice to tie them all together?
  - Nodes with multiple NICs can provide the glue!
  - Standardize address and packet formats
- This gives rise to an "Internetwork"
  - aka WAN (wide-area network)

Internetworking Origins

- Expensive supercomputers scattered throughout the US
- Researchers scattered differently throughout the US
- Needed a way to connect researchers to expensive machinery

Internetworking Origins

- Department of Defense initiated studies on how to build a resilient global network (60s, 70s)
  - How do you coordinate a nuclear attack?
- Interoperability and dynamic routing are a must
  - Along with a lot of other properties
- Result: *Internet (orig. ARPAnet, then NSFnet)*
- A **complex** system with **simple** components

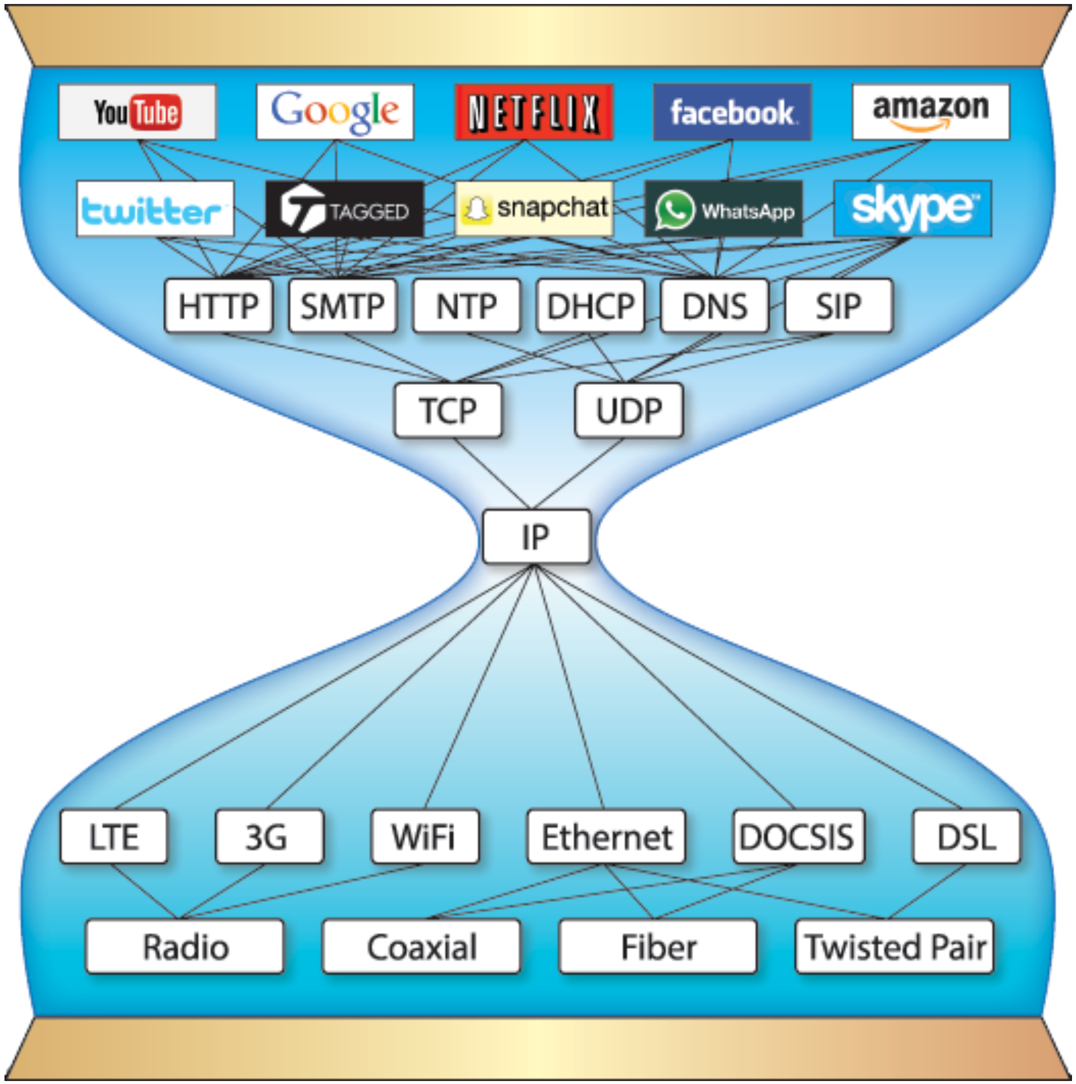
Internet Overview

- Every host is assigned, and identified by, an <u>IP address</u>
- Messages are called <u>datagrams</u>
  - the term *packet* is probably more common though…
- Each datagram contains a <u>header</u> that specifies the destination address
- The network <u>routes</u> datagrams from the source to the destination

IP

- Internetworking protocol
  - Network layer
- Common address format
- Common packet format for the Internet
  - Specifies what packets look like
  - *Fragments* long packets into shorter packets
  - *Reassembles* fragments into original shape
- IPv4 vs IPv6
  - IPv4 is what most people use
  - IPv6 more scalable and clears up some of the messy parts

IP: Narrow Waist



from: http://if-we.clients.labzero.com/code/posts/what-title-ii-means-for-tcp/

IP Addressing

- Every (active) NIC has an IP address
  - IPv4: 32-bit descriptor, e.g. 128.84.12.43
  - IPv6: 128-bit descriptor (but only 64 bits “functional”)
  - Will use IPv4 unless specified otherwise…
- Each Internet Service Provider (ISP) owns a set of IP addresses
- ISPs assign IP addresses to NICs
- IP addresses can be re-used
- Same NIC may have different IP addresses over time

IP “subnetting”

- An IP address consists of a prefix of size *n* and a suffix of size 32 – *n*
  - Either specified by a number, e.g., 128.84.32.00/24
  - Or a “netmask”, e.g., 255.255.255.0 (in case n = 24)
- A “subnet” is identified by a prefix and has 2^{32-n} addresses
  - Suffix of “all zeroes” or “all ones” reserved for broadcast
  - Big subnets have a short prefix and a long suffix
  - Small subnets have a long prefix and a short suffix

Addressing & DHCP

"I just got here. My physical address is 1a:34:2c:9a:de:cc. What's my IP?"

"Your IP is 128.84.96.89 for the next 24 hours"

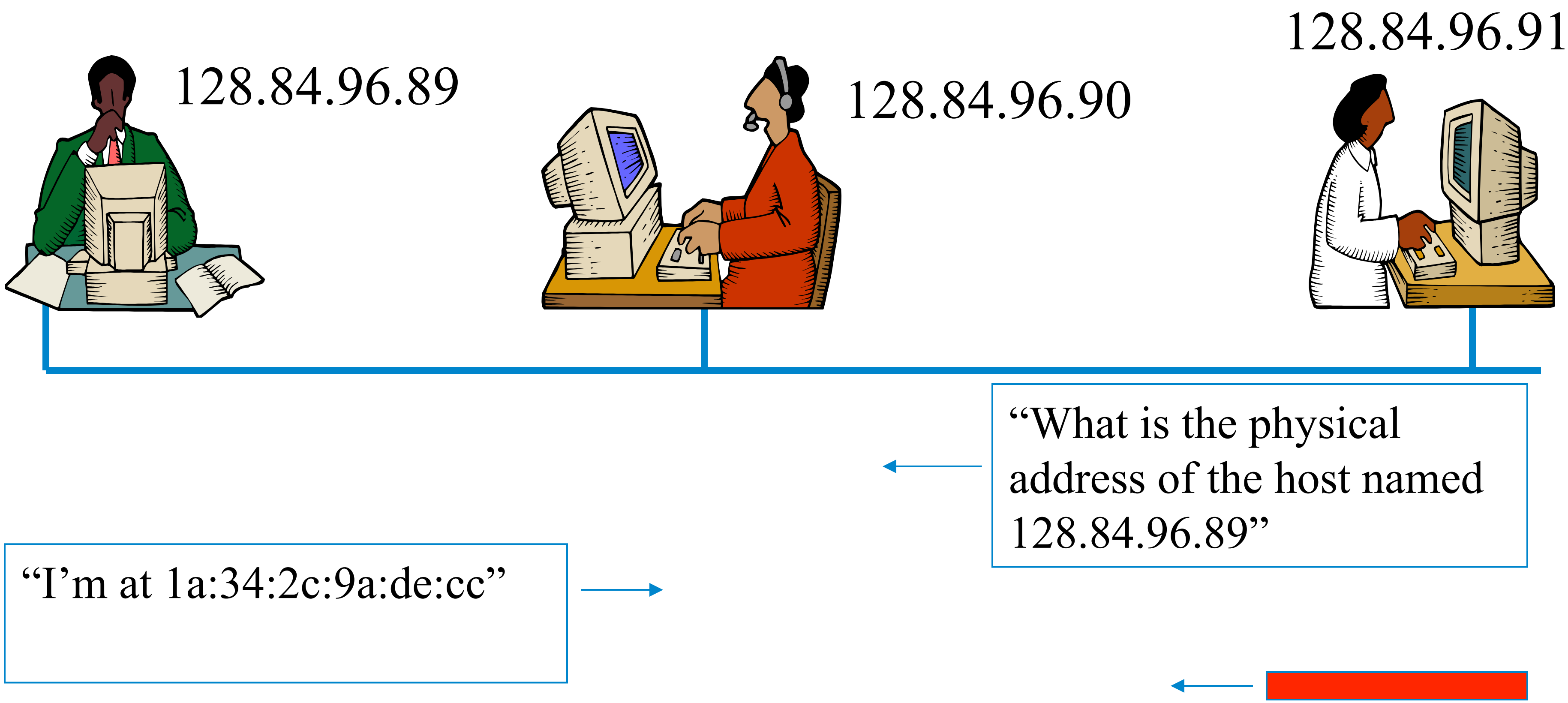
DHCP is used to discover IP addresses (and more)

DHCP = Dynamic Host Configuration Protocol

DHCP

- Each LAN (usually) runs a DHCP server
  - you probably run one at home inside your “router box”
- DHCP server maintains
  - the IP subnet that it owns (say, 128.84.245.00/24)
  - a map of IP address <-> MAC address
    - possibly with a timeout (called a “lease”)
- When a NIC comes up, it broadcasts a DHCPDISCOVER message
  - if MAC address in the map, respond with corresponding IP address
  - if not, but an IP address is unmapped and thus available, map that IP address and respond with that
- DHCP also returns the netmask
- Note: NICs can also be statically configured and don’t need DHCP

Addressing & ARP

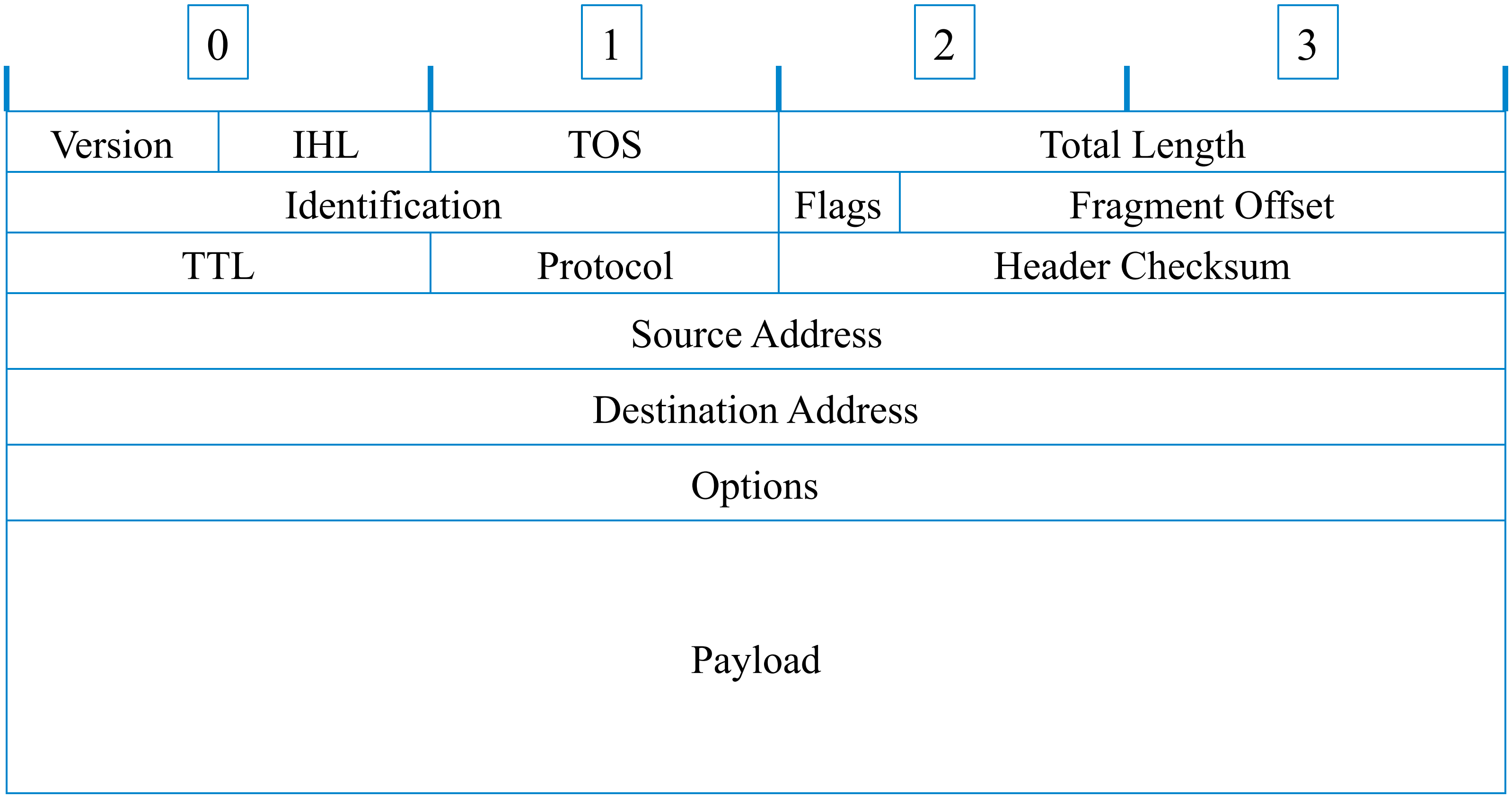


- ARP is used to discover MAC addresses *on same subnet*
  - ARP = Address Resolution Protocol

Scale?

- ARP and DHCP only scale to single subnet
- Need more to scale to the Internet!

IPv4 packet layout



IP Header Fields

- Version (4 bits): 4 or 6
- IHL (4 bits): Internet Header Length in 32-bit words
  - usually 5 unless options are present
- TOS (1 byte): type of service (not used much)
- Total Length (2 bytes): length of packet in bytes
- Id (2 bytes), Flags (3 bits), Fragment Offset (13 bits)
  - used for fragmentation/reassembly. Stay tuned
- TTL (1 byte): Time To Live. Decremented at each hop
- Protocol (1 byte): TCP, UDP, ICMP, …
- Header Checksum (2 bytes): to detect corrupted headers

IP Fragmentation

- Networks have different maximum packet sizes
  - “MTU”: Maximum Transmission Unit
    - Big packets are sometimes desirable – less overhead
    - Huge packets are not desirable – reduced response time for others
- High-level protocols could try to figure out the minimum MTU along the network path, but
    - Inefficient for links with large MTUs
    - The route can change underneath
- Consequently, IP can transparently fragment and reassemble packets

IP Fragmentation Mechanics

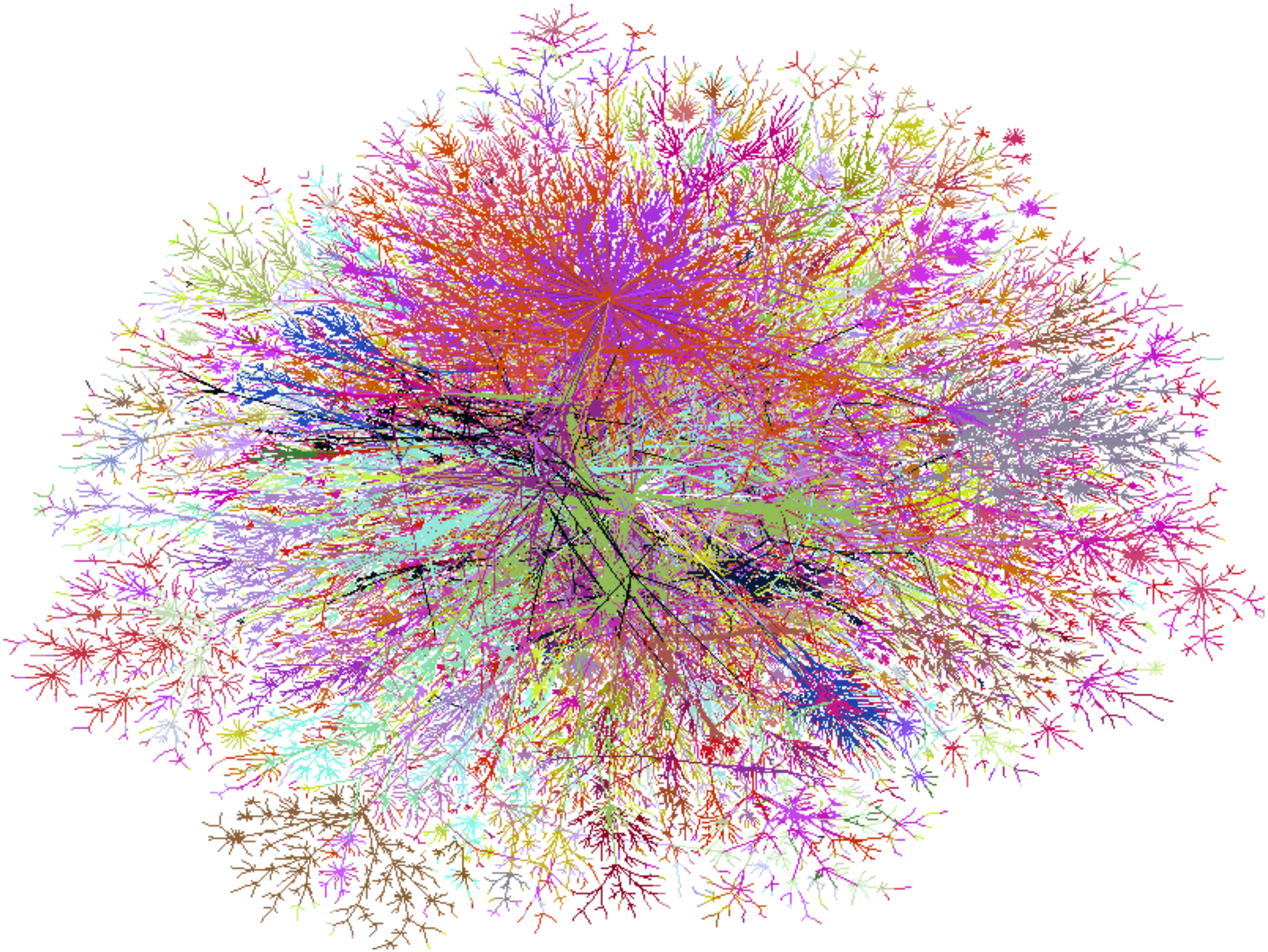
- Source assigns each datagram an “identification”
- At each hop, IP can divide a long datagram into N smaller datagrams
- Sets the More Fragments bit except on the last packet
- Receiving end puts the fragments together based on *Identification* and *More Fragments* and *Fragment Offset (times 8)*

IP Options (not well supported)

- Source Routing: The source specifies the set of hosts that the packet should traverse
- Record Route: If this option appears in a packet, every router along a path attaches its own IP address to the packet
- Timestamp: Every router along the route attaches a timestamp to the packet
- Security: Packets are marked with user info, and the security classification of the person on whose behalf they travel on the network
  - Most of these options pose security holes and are generally not implemented

Routing

The Internet is Big…



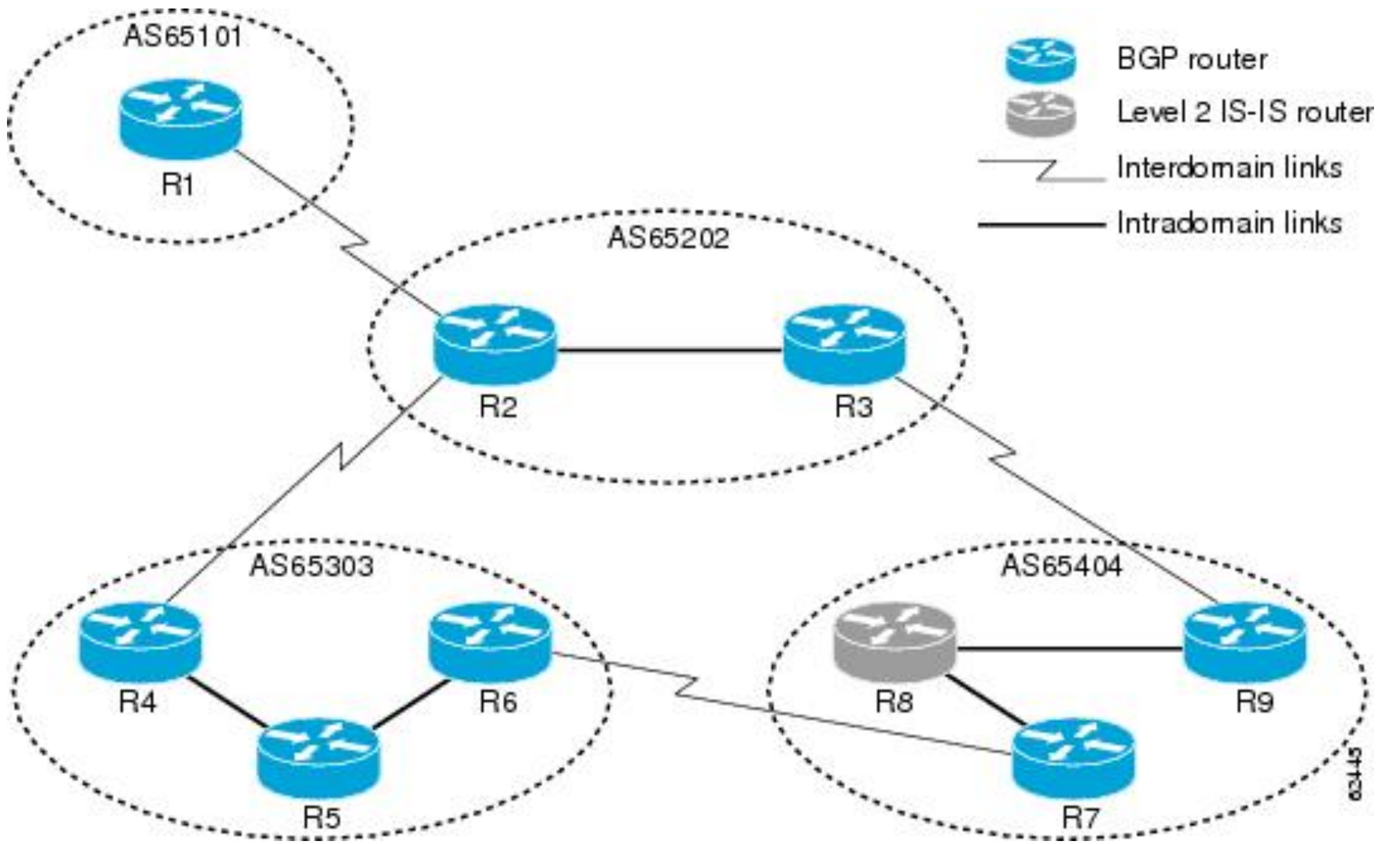
Routing

- How do we route messages from one machine to another?
- Subject to
  - churn
  - efficiency
  - reliability
  - economical considerations
  - political considerations

Internet Protocol (IP)

- The Internet is subdivided into disjoint Autonomous Systems (AS)

Graph of subgraphs



Autonomous Systems

- Each AS is a routing domain in its own right
  - has a private IP network
  - runs its own routing protocols
  - may have multiple IP subnets
    - each with their own IP prefix
  - has a unique “AS number”
- ASs are organized in a graph
  - routing between ASs using BGP (Border Gateway Protocol)

Thus routing is hierarchical!

Three steps:

1. A packet is first routed to an “edge router” at the source AS---using the internal routing protocol used by the source AS
2. Next the packet is routed to an edge router at the destination AS---determined by the destination address prefix---using BGP
3. The destination AS’s edge router then forwards the packet to its ultimate destination---determined by the address suffix---using the internal routing protocol used by the destination AS

Internet Routing, observations

- There are no longer special “government” routers that route between ASs. Instead, each AS has one or more “edge routers” that are connected by interdomain links.
- Two types:
  - **Transit AS**: forwards packets coming from one AS to another AS
  - **Stub AS**: has only “upstream” links and does not do any forwarding

What’s an ISP?

- An ISP (Internet Service Provider) is simply an AS (or collection of ASs) that provides, to its customers (which may be people or other ASs), access to the “The Internet”
- Provides one or more PoPs (Points of Presence) for its customers.

Routers (Layer-3 Switches)

- Connects multiple LANs (subnets)
- Two classes:
  - Edge or Border router: Resides at the edge of an AS, and has two faces
    - one faces outside to connect to one or more per edge router in other ASs
    - one faces inside, connecting to zero or more other routers within the same AS
  - Interior router:
    - has no connections to routers in other ASs

Routing Table

- Maps IP address to interface or port and to MAC address
- Longest Prefix Matching
- Your laptop/phone has a routing table too!

Address	IF or Port	MAC
128.84.216/23	en0	`c4:2c:03:28:a1:39`
127/8	lo0	`127.0.0.1`
128.84.216.36/32	en0	`74:ea:3a:ef:60:03`
128.84.216.80/32	en0	`20:aa:4b:38:03:24`
128.84.217.255/32	en0	`ff:ff:ff:ff:ff:ff`

Router Function

often implemented in hardware

```
for ever:
    receive IP packet p
    if isLocal(p.dest): return localDelivery(p)
    if --p.TTL == 0: return dropPacket(p)
    matches = { }
    for each entry e in routing table:
        if p.dest & e.netmask == e.address & e.netmask:
            matches.add(e)
    bestmatch = matches.maxarg(e.netmask)
    forward p to bestmatch.port/bestmatch.MAC
```

Routing Loops?

- In steady state, there should be no routing loops
- But steady state is rare. If routing tables are not in sync, routing loops can occur.
- To avoid problems, IP packets maintain a maximum hop count (TTL) that is decreased on every hop until 0 is reached, at which point a packet is dropped.

How are these routing tables constructed?

- For end-hosts, mostly DHCP and ARP as discussed before
- For routers, using a “routing protocol”
  - take Prof. Agarwal’s networking course!

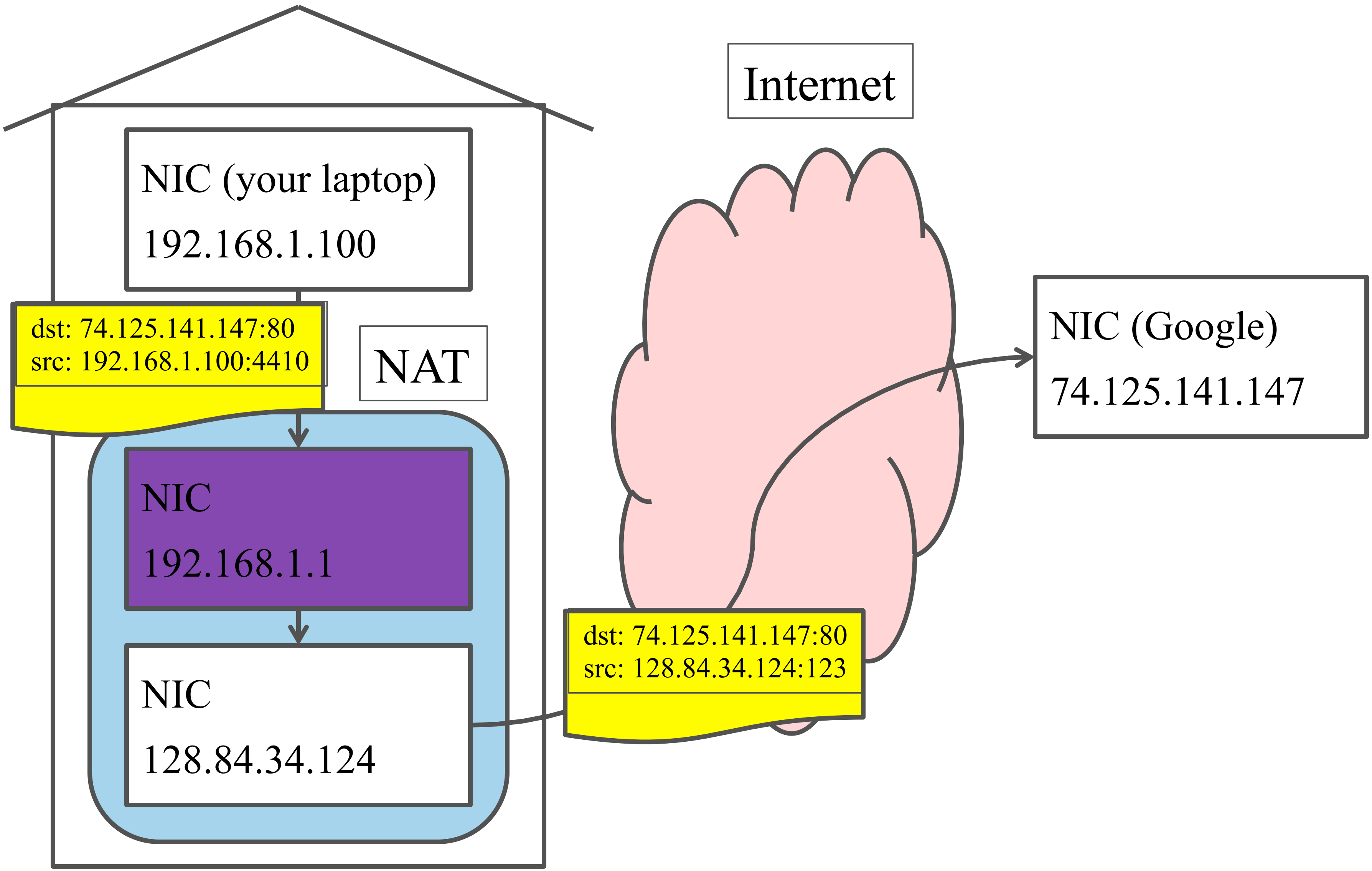
Network Address Translation

- IPv6 adoption is very slow, and IPv4 addresses have run out
- NAT allows entire sites to use a single globally routable IPv4 address for a collection of machines
  - exploits the sparsely used 16-bit TCP/UDP port number space
- A “NAT box” keeps a table that maps global TCP/IP addresses into local ones
- Overwrites the local source address with the globally addressable address

“Private” IP addresses

- The IPv4 addresses 10.x.x.x and 192.168.x.x are freely available for anybody to use
- Many machines have the IP address 192.168.0.100, for example

From your laptop to Google…



Vice versa: punching holes or “game ports”

- When an external host tries to send a message to one of your machines in your house, it first arrives at the NAT box
  - Because you advertise your global IP address
- How does the NAT box know which of your machines to forward the message to?
- Answer: a table. It is indexed by the destination TCP or UDP port in the message

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 *connects* to the server: locates it in the network & establishes a connection
2. Client sends *requests*: messages that indicate which service is desired and the 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

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

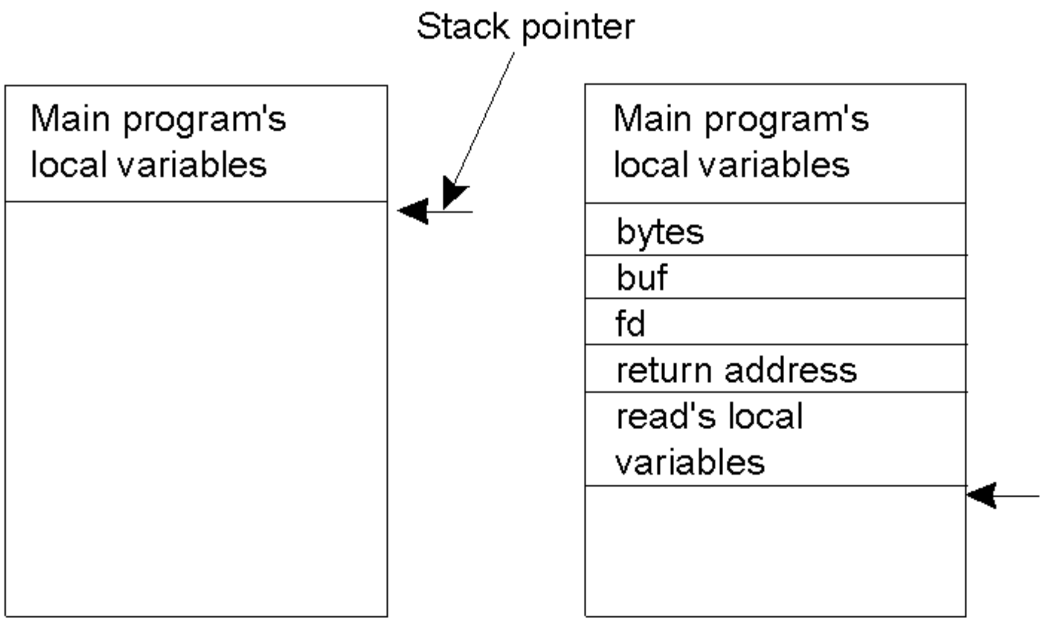
Birrell & Nelson @ Xerox PARC

"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



Stack before procedure call

Stack during procedure call

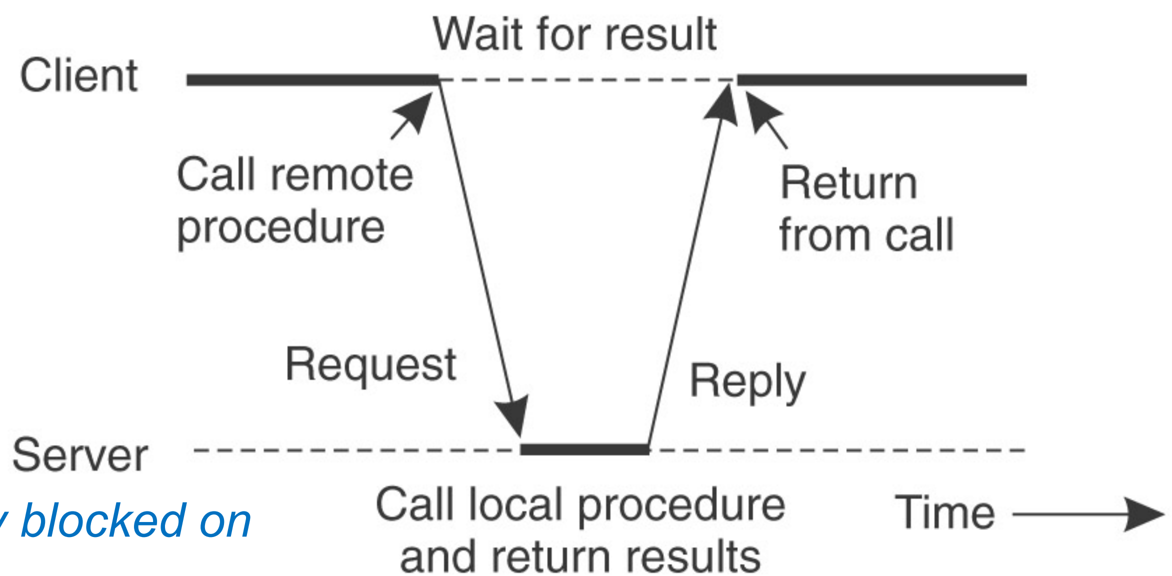
- Linker inserts read implementation into obj file
- Implementation usually invokes a system call

[Tanenbaum & van Steen, Fig

How does a RPC work?

Basic idea:

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



(typically blocked on `receive() at first`)

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

[Tanenbaum & van Steen, Fig

RPC Stubs

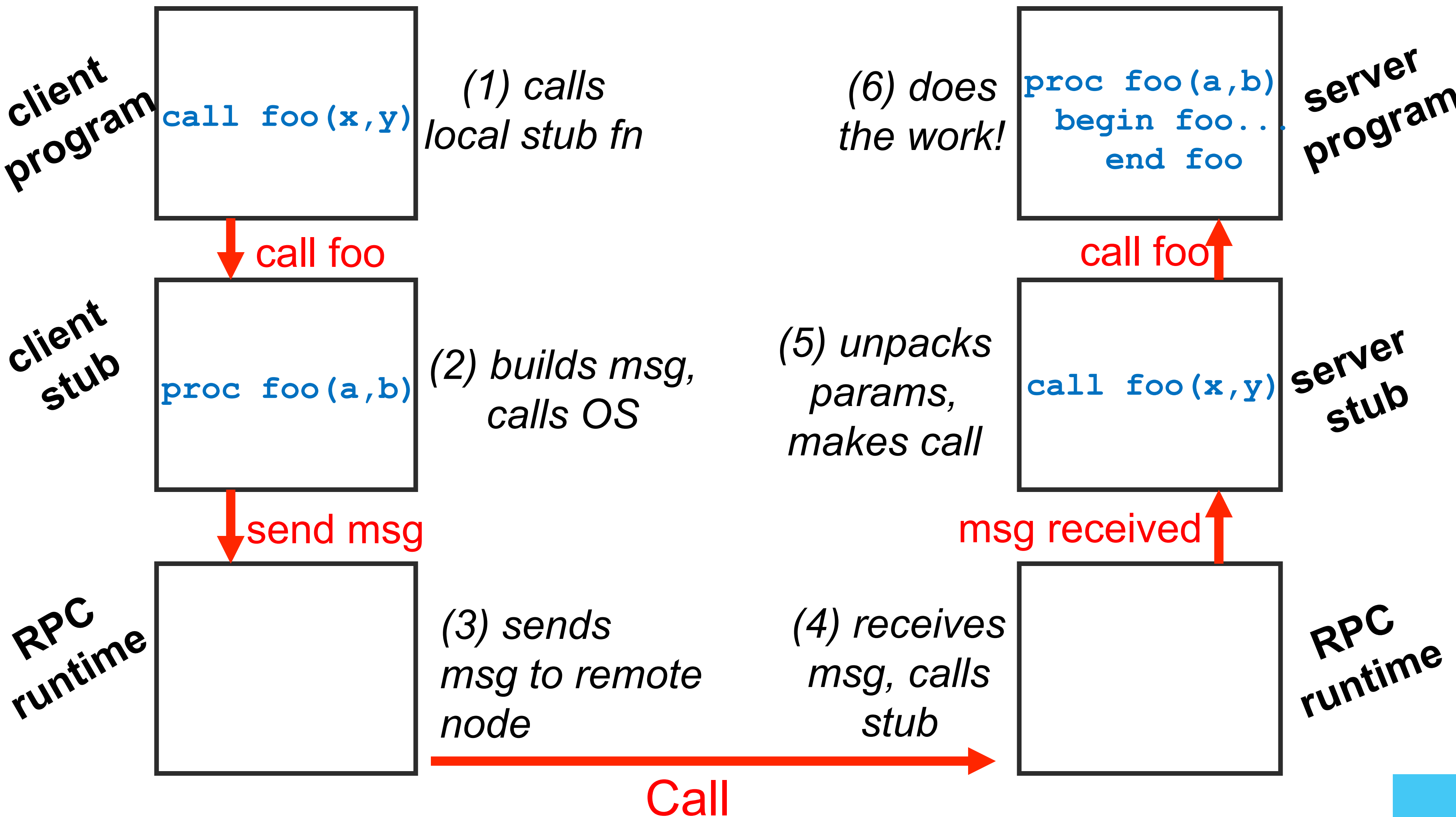
Client-side stub:

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

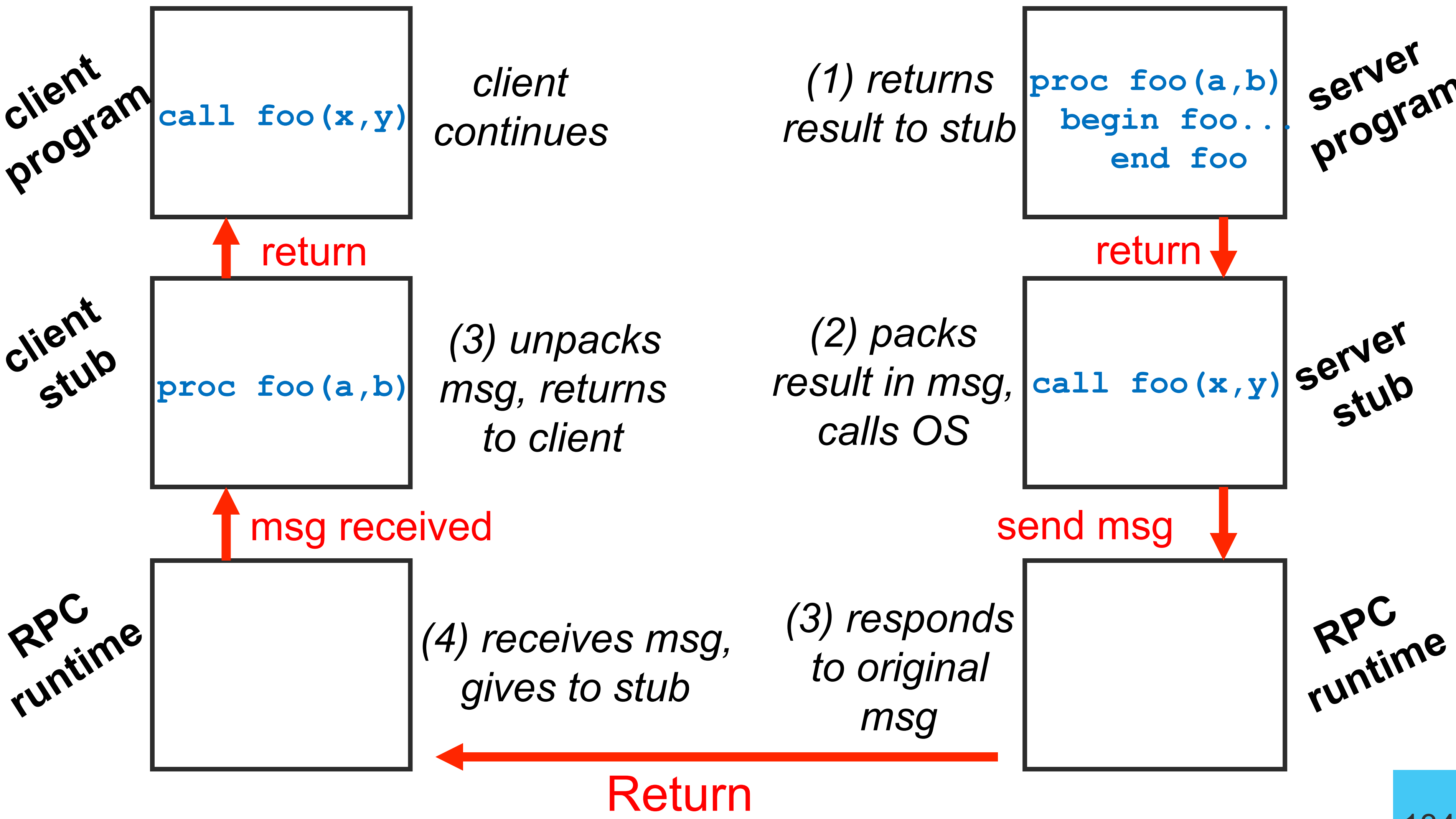
Server-side stub:

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

RPC Call Structure

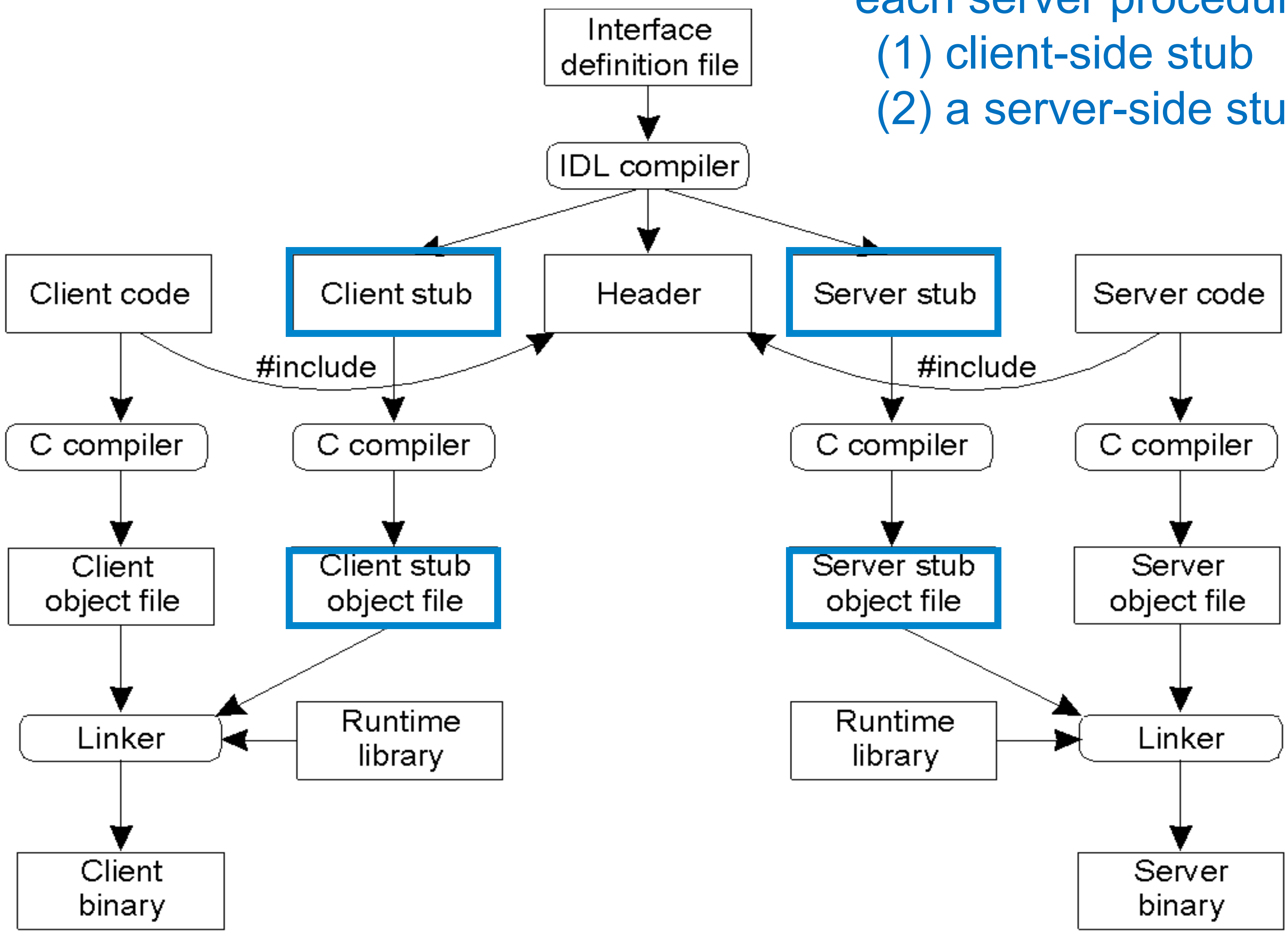


RPC Return Structure



Example RPC system: *Stub compiler*

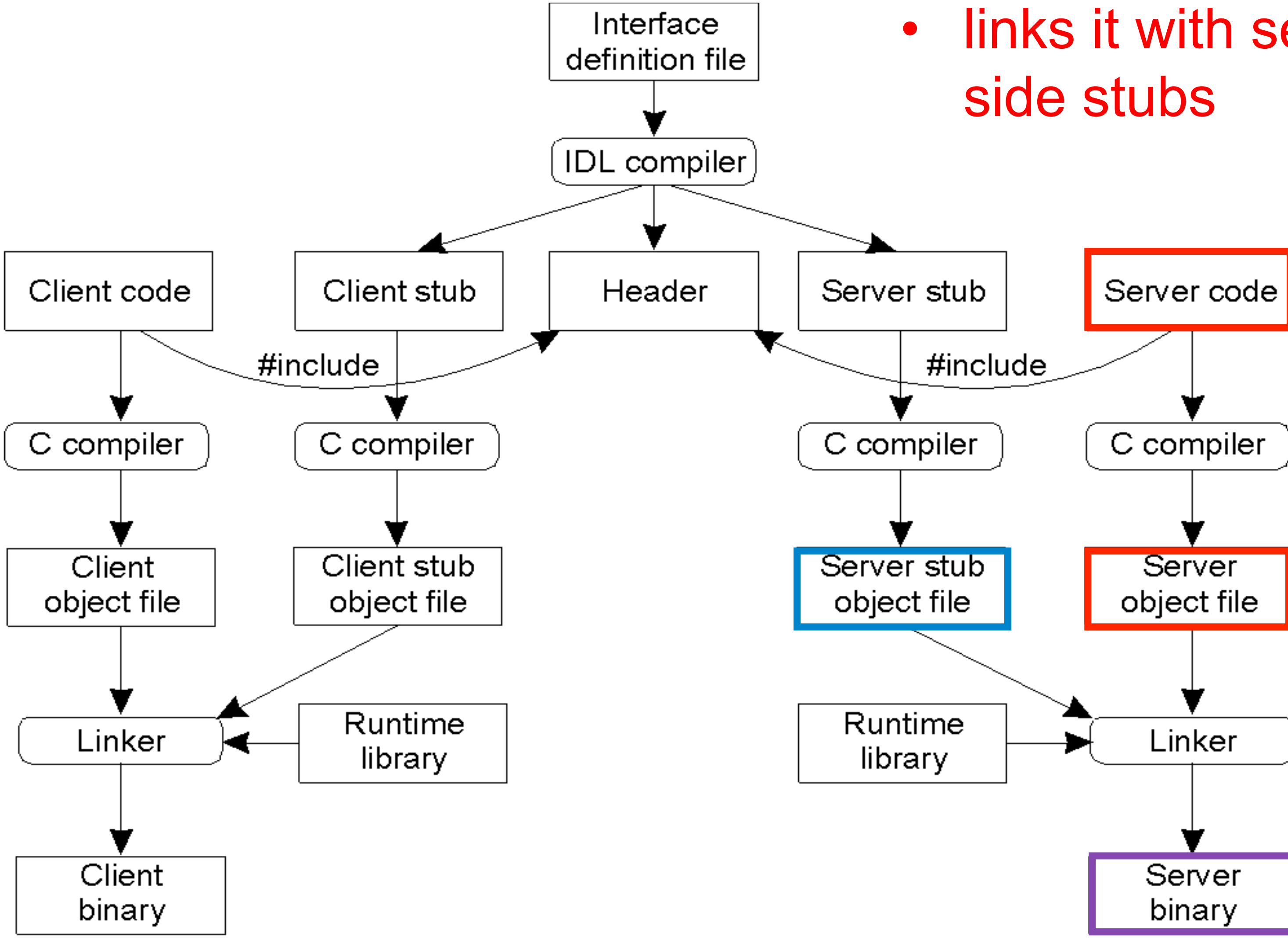
- reads IDL
- produces 2 stub procedures for each server procedure:
  (1) client-side stub
  (2) a server-side stub



Example RPC system:

Server writer:

- writes server
- links it with server-side stubs



Binding: Connecting Client & Server

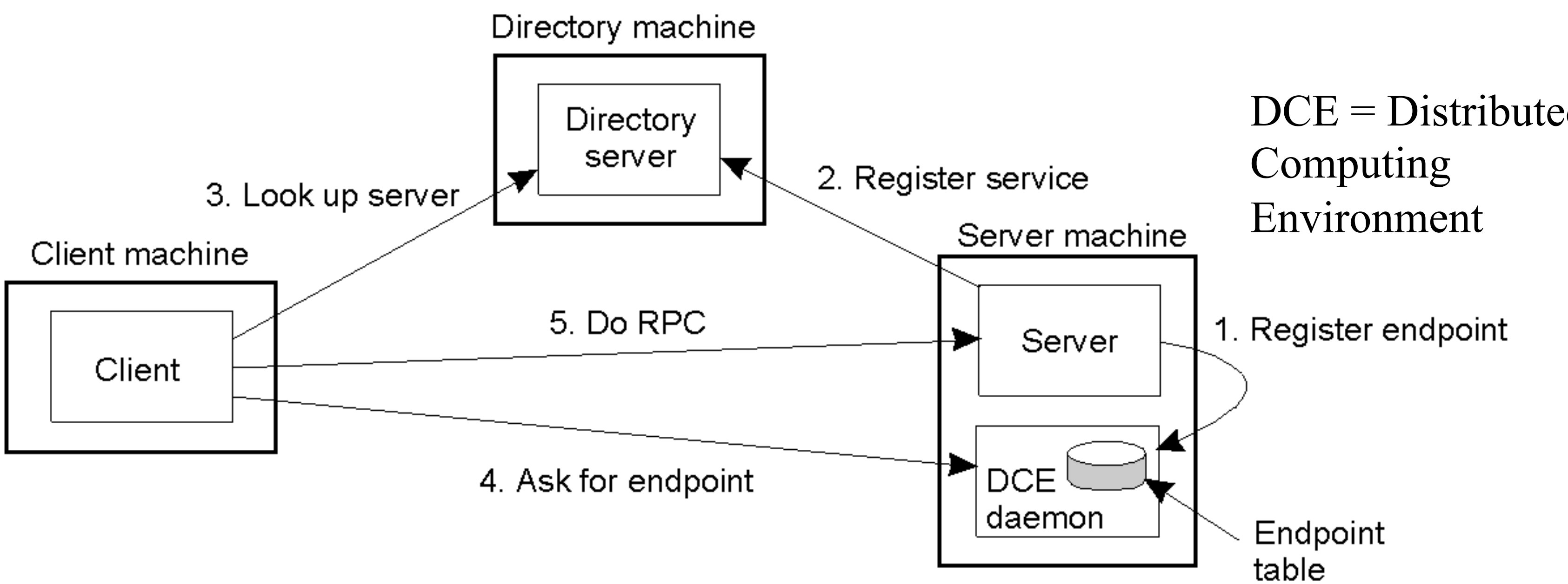
Server *exports* its interface:

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

Client *imports* the interface. 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



DCE = Distributed Computing Environment

RPC Concerns

- Parameter Passing
- Failure Cases
- Performance

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

RPC Marshaling

Packing parameters into a message packet

- RPC stubs call type-specific procedures to marshal (or unmarshal) all of the parameters to the call

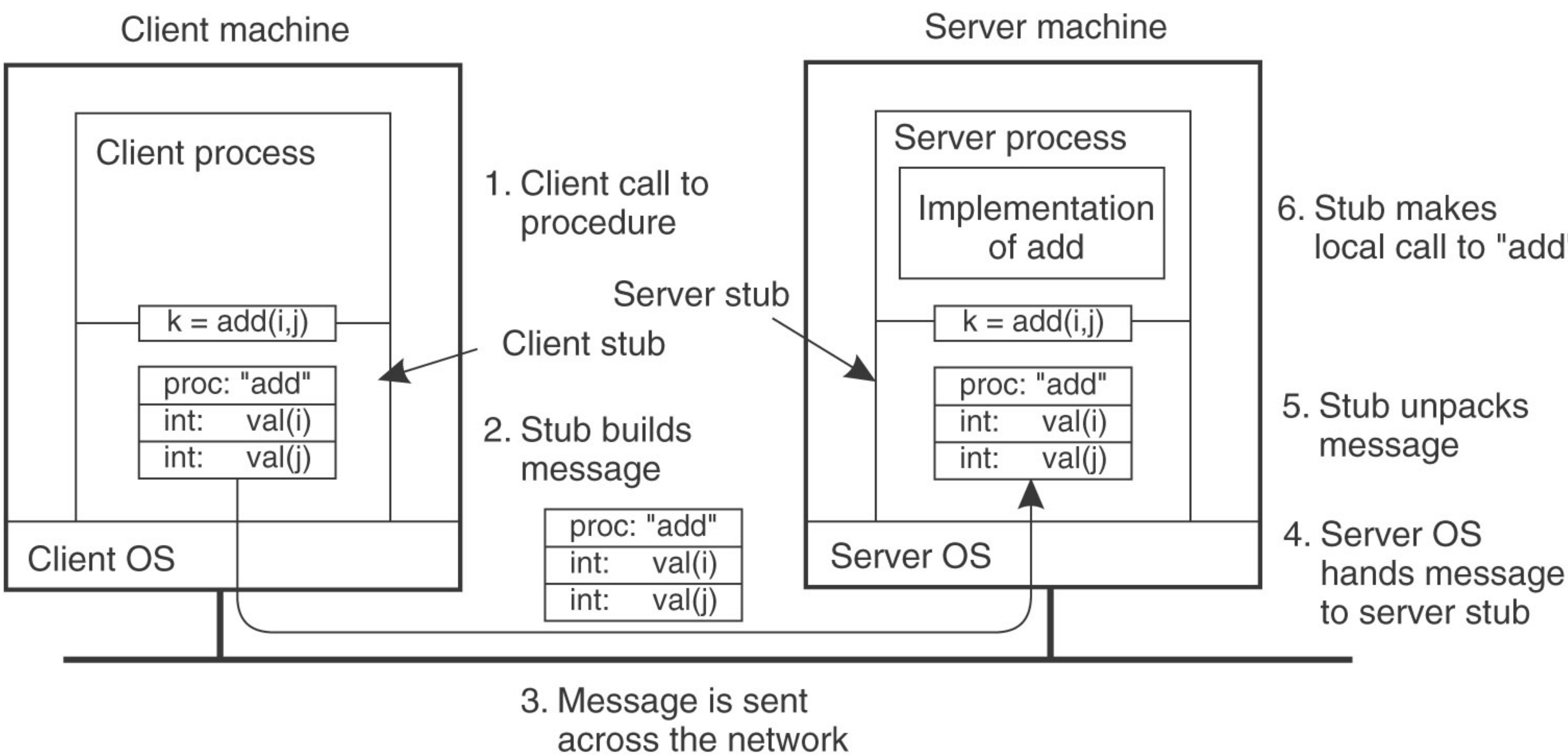
On Call:

- Client stub marshals parameters into the call packet
- Server stub unmarshals parameters to call server's fn

On return:

- Server stub marshals return values into return packet
- Client stub unmarshals return values, returns to client

Parameter Passing



What could go wrong?

[Tanenbaum & van Steen, Fig

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, endian-ness

→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 to a file

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)