Computer Networks: Sockets

Slides courtesy Kurose & Ross

Agenda

- Computer networks, primarily from an application perspective
- Protocol layering
- Client-server architecture
- End-to-end principle
- TCP
- Socket programming

Why Networking?

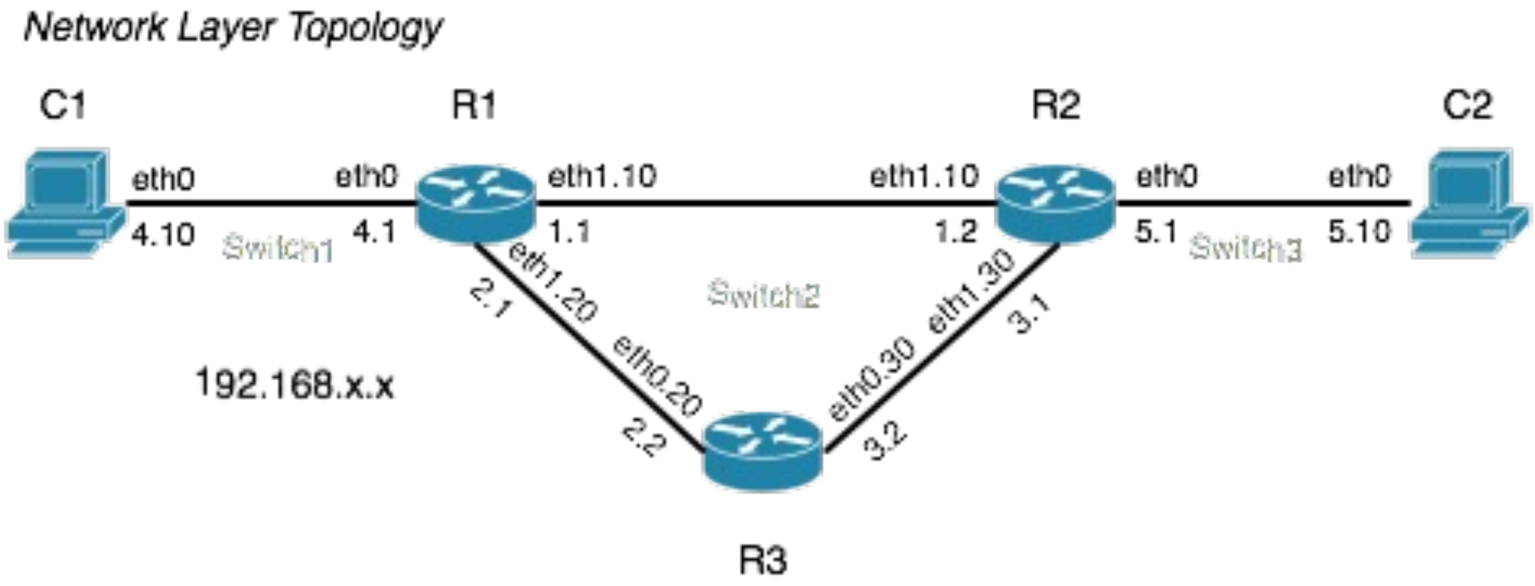
- All communication takes place over computer networks
- Networking affects how we design distributed systems:
  - Architecture
  - Performance
  - Reliability and Resiliency

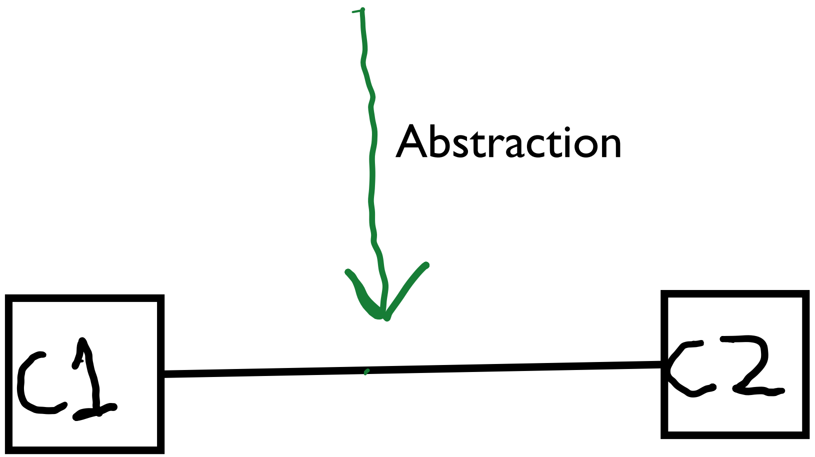
Networking Goals

- Reliable delivery of data (packets)
- Low latency delivery of data
- Utilize physical networking bandwidth
- Share network bandwidth among multiple agents

Network Elements

- Links:
  - Wired or wireless
- Hosts or end-points:
  - Servers/clients
- Packets:
  - Units of data transmission
- Switches, Routers, Middleboxes:
  - Receive, process, forward packets





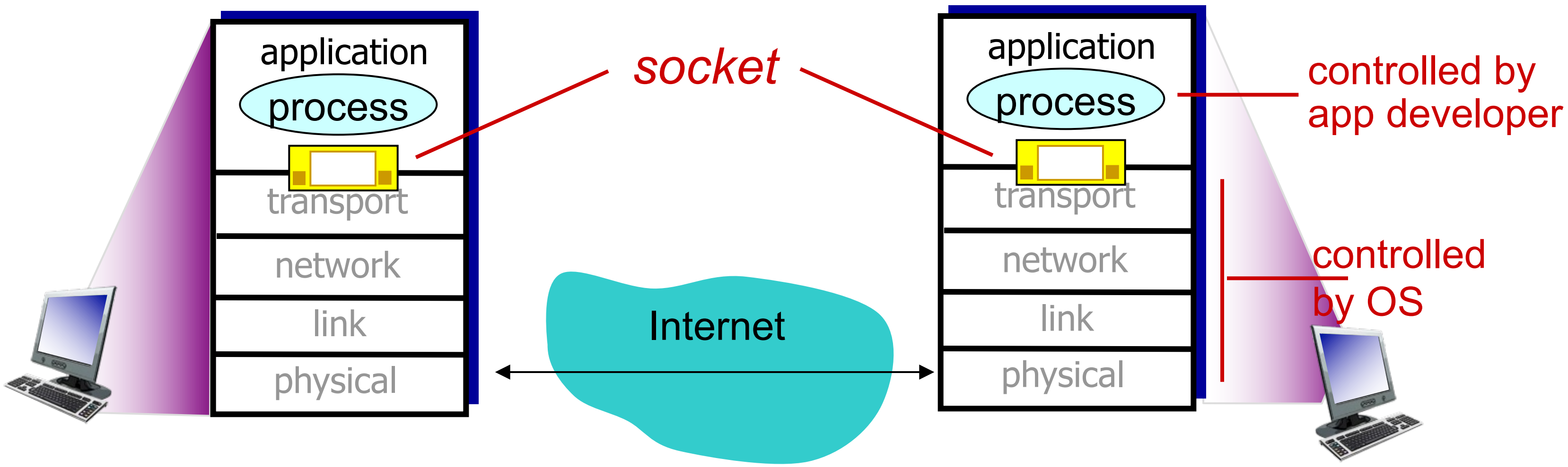
Abstraction

Network debugging

- Check port availability (netstat –plant) . <1024 are privileged
- TCP client: nc . For quick testing if server is working correctly
- Many wrappers. https://docs.python.org/3/library/socketserver.html
- Be careful about data byte order and encoding.
  - Sending "bits on the wire". How are they interpreted by the receiver?
- Common issues:
  - Sending data before recipient is ready
  - Blocking operations

Sockets

- process sends/receives messages to/from its socket
- socket analogous to door
  - sending process shoves message out door
  - sending process relies on transport infrastructure on other side of door to deliver message to socket at receiving process



Socket programming *with UDP*

UDP: no "connection" between client & server

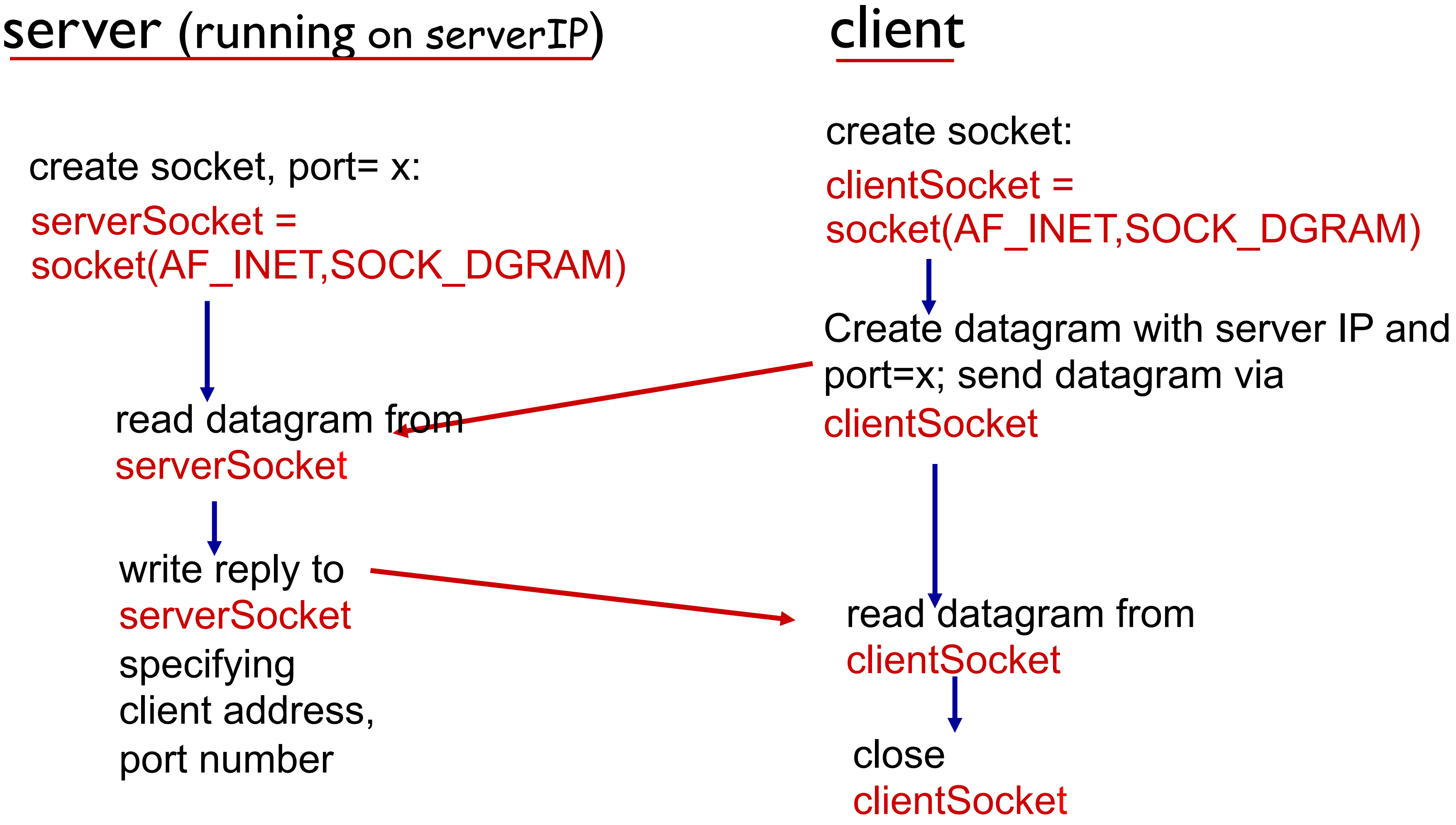
- no handshaking before sending data
- sender explicitly attaches IP destination address and port # to each packet
- receiver extracts sender IP address and port# from received packet

UDP: transmitted data may be lost or 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 socket, port= x:

serverSocket =
socket(AF_INET,SOCK_DGRAM)

read datagram from
serverSocket

write reply to
serverSocket
specifying
client address,
port number

client

create socket:

clientSocket =
socket(AF_INET,SOCK_DGRAM)

Create datagram with server IP and
port=x; send datagram via
clientSocket

read datagram from
clientSocket

close
clientSocket

Example app: UDP client

Python UDPClient

include Python's socket library

create UDP socket for server

get user keyboard input

Attach server name, port to message; send into socket

read reply characters from socket into string

print out received string and close socket

```
from socket import *
serverName = ‘hostname’
serverPort = 12000
clientSocket = socket(socket.AF_INET,
                              socket.SOCK_DGRAM)
message = raw_input(’Input lowercase sentence:’)
clientSocket.sendto(message,(serverName, serverPort))
modifiedMessage, serverAddress =
                              clientSocket.recvfrom(2048)
print modifiedMessage
clientSocket.close()
```

Example app: UDP server

Python UDPServer

create UDP socket

bind socket to local port number 12000

loop forever

Read from UDP socket into message, getting client’s address (client IP and port)

send upper case string back to this client

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind((", serverPort))
print “The server is ready to receive”
while 1:
    message, clientAddress = serverSocket.recvfrom(2048)
    modifiedMessage = message.upper()
    serverSocket.sendto(modifiedMessage, clientAddress)
```

Socket programming *with TCP*

client must contact server

- server process must first be running
- server must have created socket (door) that welcomes client's contact

client contacts server by:

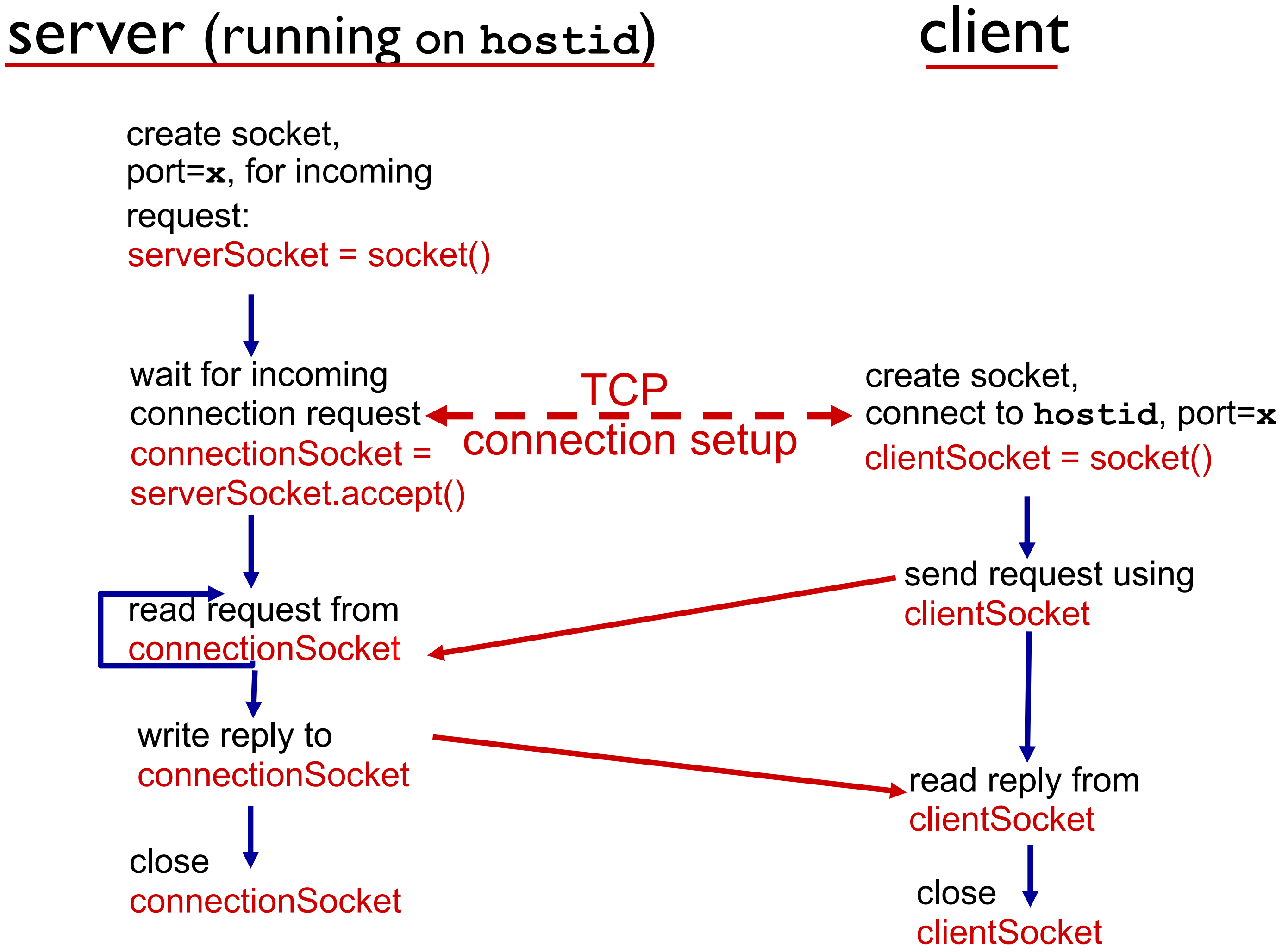
- Creating TCP socket, specifying IP address, port number of server process
- *when client creates socket:* client TCP establishes connection to server TCP

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

application viewpoint:

TCP provides reliable, in-order byte-stream transfer ("pipe") between client and server

Client/server socket interaction: TCP



server (running on `hostid`)

client

create socket,
port=**`x`**, for incoming
request:
serverSocket = socket()

wait for incoming
connection request
connectionSocket =
serverSocket.accept()

TCP
connection setup

create socket,
connect to **`hostid`**, port=**`x`**
clientSocket = socket()

send request using
clientSocket

read request from
connectionSocket

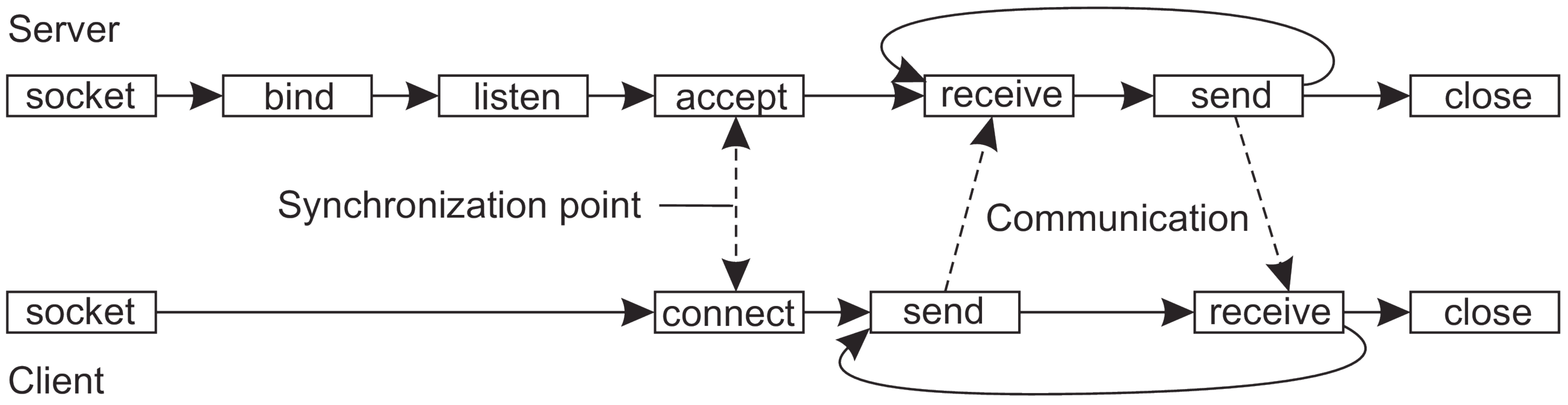
write reply to
connectionSocket

read reply from
clientSocket

close
connectionSocket

close
clientSocket

TCP Connection Flow



- Some operations (**accept** and **receive**) are **Blocking**.
- These calls wont return and the server won’t execute the next program instruction.
  - Unless some client connects or sends data.
  - Or the server process is signalled/killed/interrupted

Example app:TCP client

Python TCPClient

create TCP socket for server, remote port 12000

No need to attach server name, port

```
from socket import *
serverName = ’servername’
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName,serverPort))
sentence = raw_input(‘Input lowercase sentence:’)
clientSocket.send(sentence)
modifiedSentence = clientSocket.recv(1024)
print ‘From Server:’, modifiedSentence
clientSocket.close()
```

Example app: TCP server

Python TCPServer

```
 from socket import *
serverPort = 12000
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind((‘’,serverPort))
serverSocket.listen(1)
print ‘The server is ready to receive’
while 1:
    connectionSocket, addr = serverSocket.accept()

    sentence = connectionSocket.recv(1024)
    capitalizedSentence = sentence.upper()
    connectionSocket.send(capitalizedSentence)
    connectionSocket.close()
```

create TCP welcoming socket

server begins listening for incoming TCP requests

loop forever

server waits on accept() for incoming requests, new socket created on return

read bytes from socket (but not address as in UDP)

close connection to this client (but *not* welcoming socket)

Socket Example

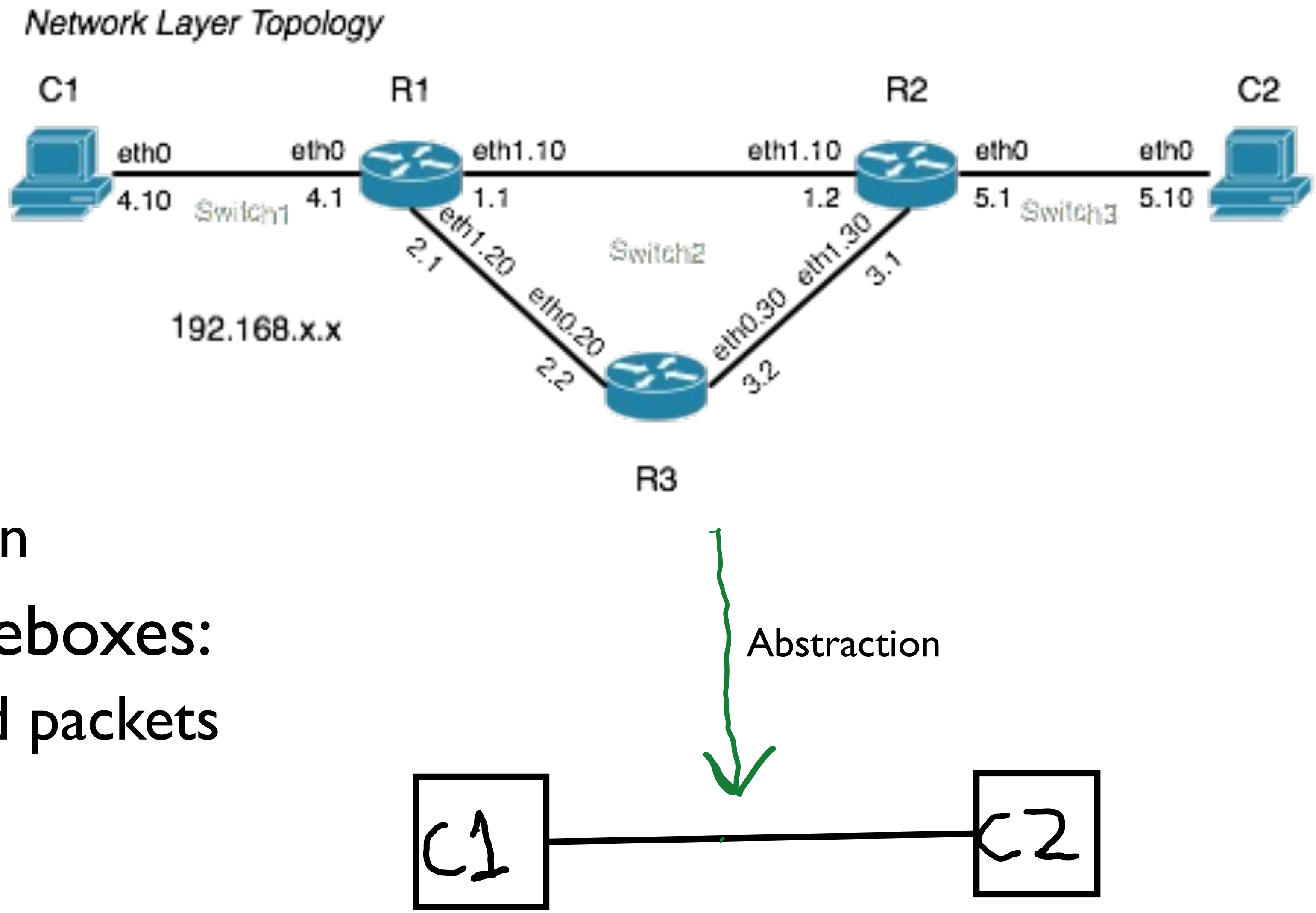
```
# An example script to connect to Google using socket
# programming in Python
import socket # for socket
import sys

try:
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    print "Socket successfully created"
except socket.error as err:
    print "socket creation failed with error %s" %(err)

# default port for socket
port = 80

try:
    host_ip = socket.gethostbyname('www.google.com')
except socket.gaierror:

    # this means could not resolve the host
    print "there was an error resolving the host"
    sys.exit()

# connecting to the server
s.connect((host_ip, port))

print "the socket has successfully connected to google \
on port == %s" %(host_ip)
```

Common Pitfalls

- Note that ”client” and ”server” are not permanent classifications of processes.
  - In the previous examples, they are sender and receiver
  - Because TCP is connection oriented, many network services happen to
  - In most distributed systems, a process is going to act as both sender and receiver at different points in time.
- Careful with binary data! Serialize to string where possible
  - Payload = pickle.dumps(pyobj) ; socket.send(payload)
  - Network byte order may be different than host (little vs. big endian)
- Designing distributed systems often comes down to identifying communication message formats and protocols ahead of time
  - Who is sending what, and to whom?
  - How to parse and react to messages of a certain type?
  - Show me your data structures….

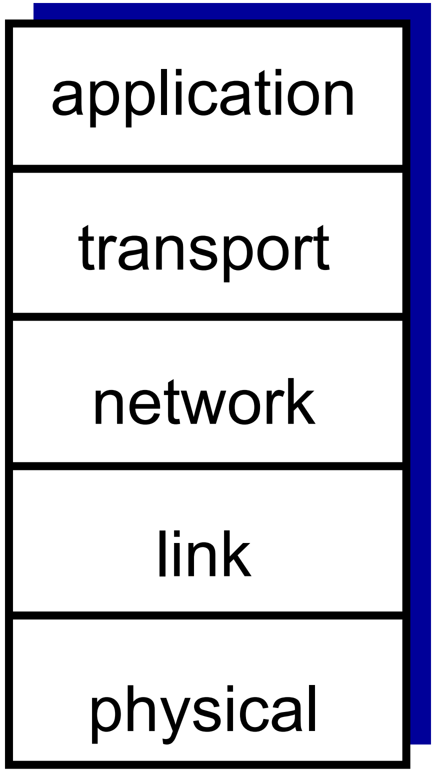
Network Elements

- Links:
  - Wired or wireless
- Hosts or end-points:
  - Servers/clients
- Packets:
  - Units of data transmission
- Switches, Routers, Middleboxes:
  - Receive, process, forward packets

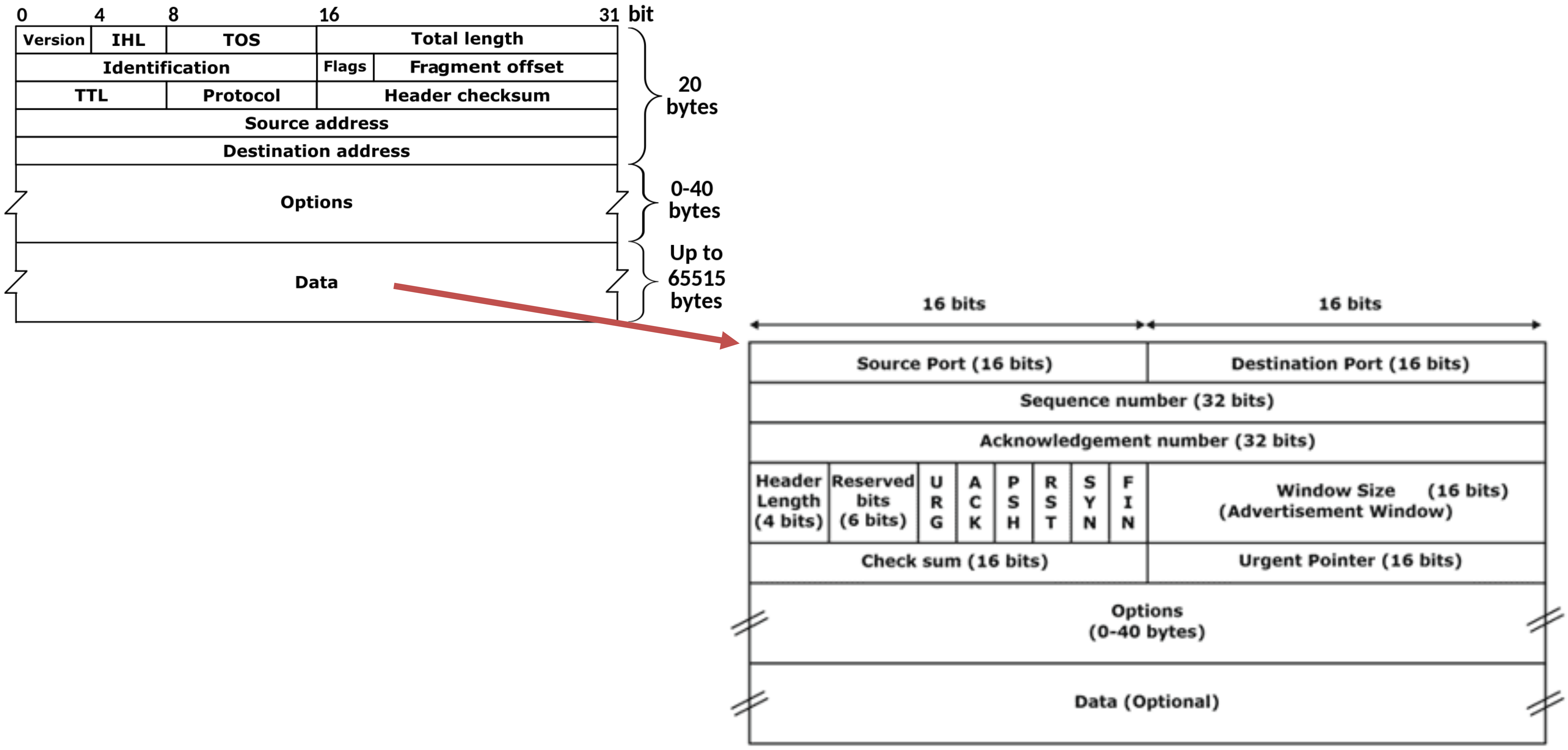


Internet protocol stack

- *application:* supporting network applications
  - FTP, SMTP, HTTP
- *transport:* process-process data transfer
  - TCP, UDP
- *network:* routing of datagrams from source to destination
  - IP, routing protocols
- *link:* data transfer between neighboring network elements
  - Ethernet, 802.111 (WiFi), PPP
- *physical:* bits "on the wire"

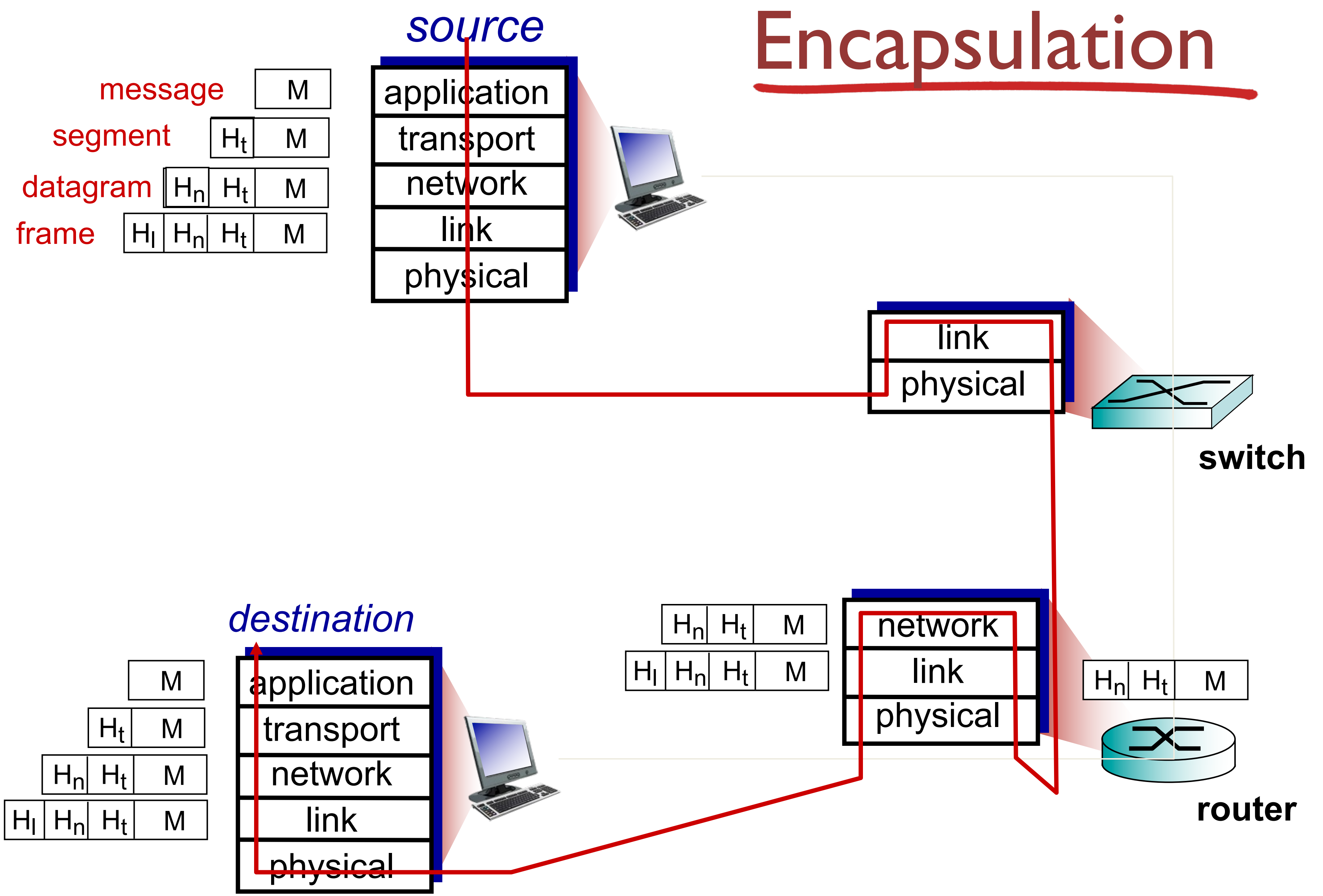


TCP and IP Headers



TCP Header

Encapsulation



source

message M

segment H_t M

datagram H_n H_t M

frame H_l H_n H_t M

application

transport

network

link

physical

link

physical

switch

destination

M

H_t M

H_n H_t M

H_l H_n H_t M

application

transport

network

link

physical

H_n H_t M

H_l H_n H_t M

network

link

physical

H_n H_t M

router

App-layer protocol defines

- types of messages exchanged,
  - e.g., request, response
- message syntax:
  - what fields in messages & how fields are delineated
- message semantics
  - meaning of information in fields
- rules for when and how processes send & respond to messages

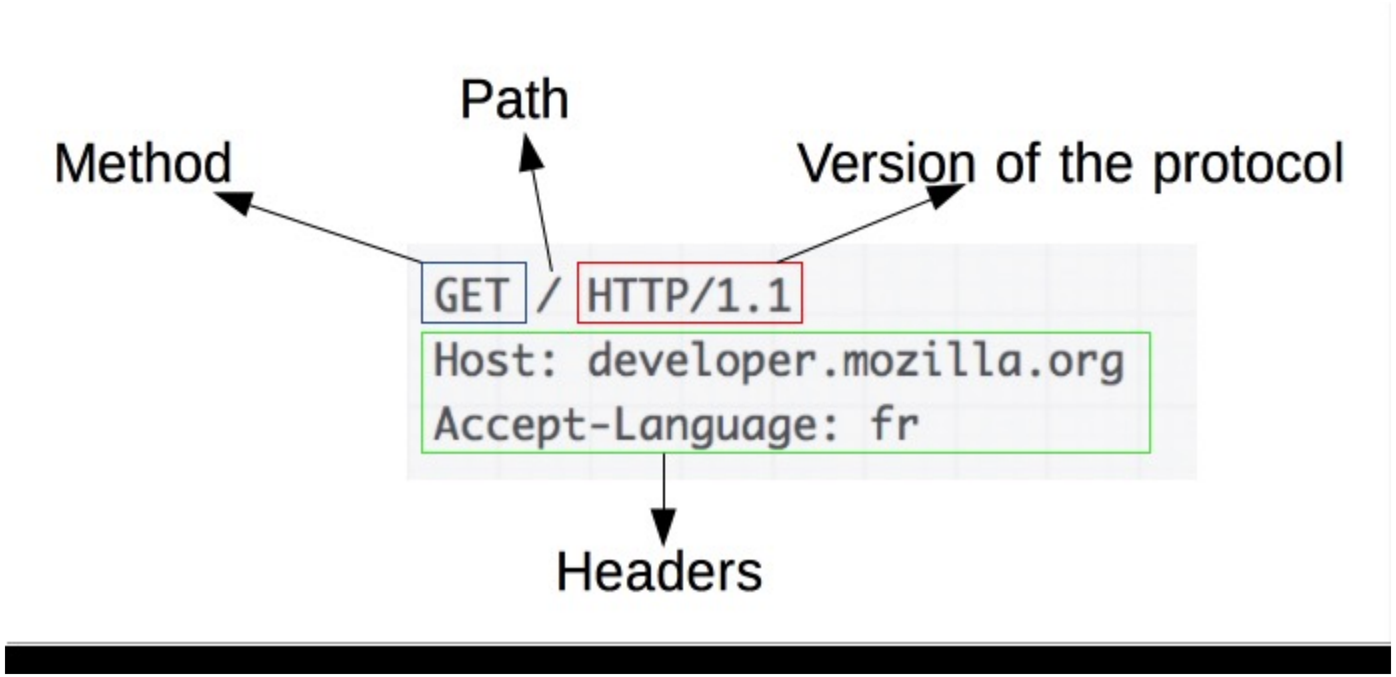
open protocols:

- defined in RFCs
- allows for interoperability
- e.g., HTTP, SMTP

proprietary protocols:

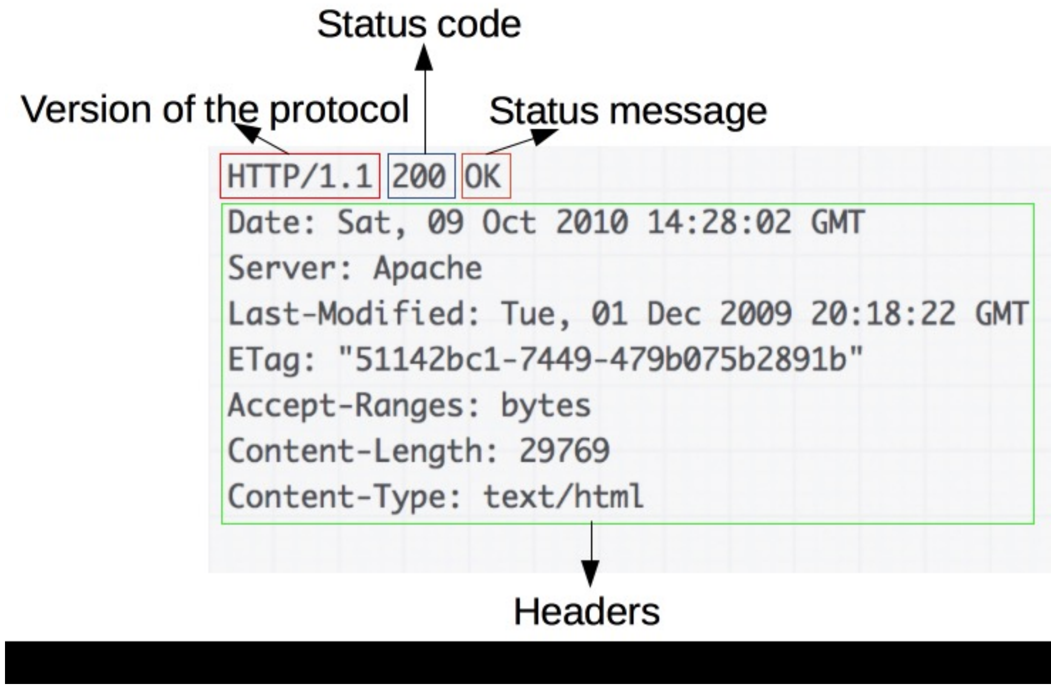
- e.g., Skype

HTTP Header Example



```
GET / HTTP/1.1
Host: developer.mozilla.org
Accept-Language: fr
```

Request



```
HTTP/1.1 200 OK
Date: Sat, 09 Oct 2010 14:28:02 GMT
Server: Apache
Last-Modified: Tue, 01 Dec 2009 20:18:22 GMT
ETag: "51142bc1-7449-479b075b2891b"
Accept-Ranges: bytes
Content-Length: 29769
Content-Type: text/html
```

Response

HTTP overview

uses TCP:

- client initiates TCP connection (creates socket) to server, port 80
- server accepts TCP connection from client
- HTTP messages (application-layer protocol messages) exchanged between browser (HTTP client) and Web server (HTTP server)
- TCP connection closed

HTTP is "stateless"

- server maintains no information about past client requests

aside

protocols that maintain "state" are complex!

- past history (state) must be maintained
- if server/client crashes, their views of "state" may be inconsistent, must be reconciled

What transport service does an app need?

data integrity

- some apps (e.g., file transfer, web transactions) require 100% reliable data transfer
- other apps (e.g., audio) can tolerate some loss

timing

- some apps (e.g., Internet telephony, interactive games) require low delay to be "effective"

throughput

- some apps (e.g., multimedia) require minimum amount of throughput to be "effective"
- other apps ("elastic apps") make use of whatever throughput they get

security

- encryption, data integrity, …

Principle Of End-To-End System Design

"END-TO-END ARGUMENTS IN SYSTEM DESIGN" J.H. Saltzer, D.P. Reed and D.D. Clark

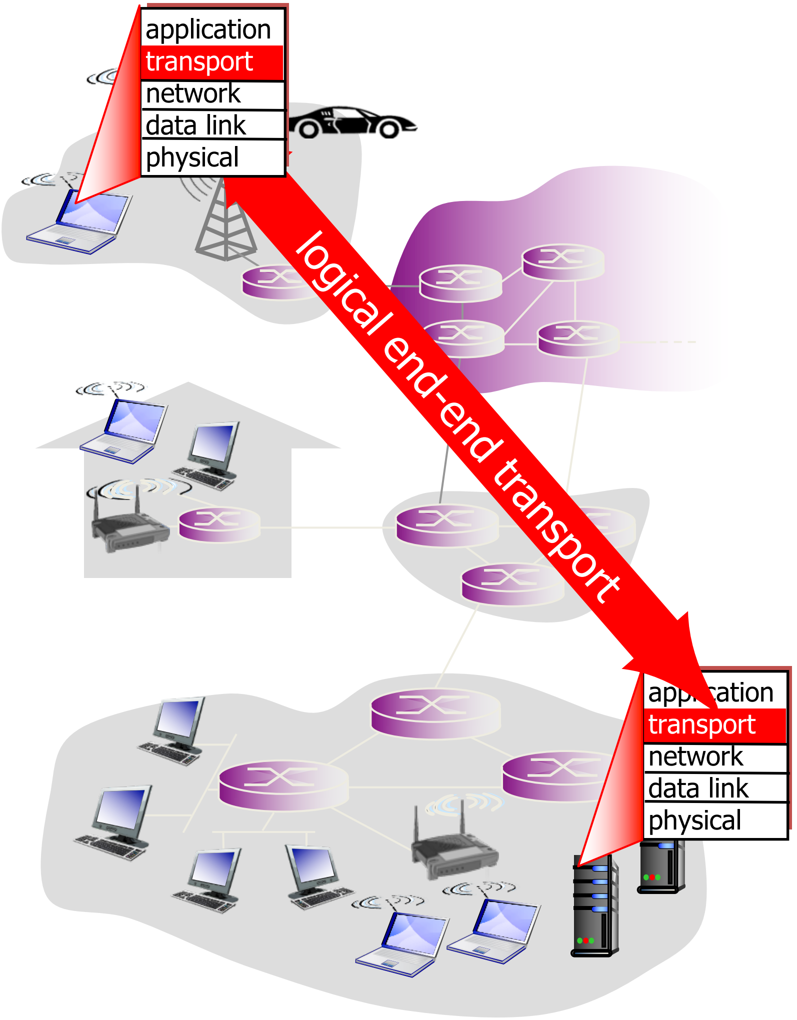
- *Where to implement functionality in a distributed system?*
  - Especially relevant in networking
- Example: Copy a file across the network reliably
  - Option 1 : Copy file, and then verify contents using checksums
  - Option 2 : Build a perfectly reliable network, routers, etc.
- Even with a perfectly reliable network, things can go wrong
  - Need application level verification anyway

Principle Of End-To-End System Design (2/2)

- It is better to implement functionality at the "ends" of the network (aka the hosts)
  - Enables effective layering
  - Better to implement functionality at higher layers of abstraction
- Also useful in non-network settings like operating systems
  - Implementing system calls in hardware is not a great idea

Transport services and protocols

- provide *logical communication* between app processes running on different hosts
- transport protocols run in end systems
  - send side: breaks app messages into *segments*, passes to network layer
  - rcv side: reassembles segments into messages, passes to app layer
- more than one transport protocol available to apps
  - Internet: TCP and UDP



Internet transport protocols services

TCP service:

- *reliable transport* between sending and receiving process
- *flow control:* sender won't overwhelm receiver
- *congestion control:* throttle sender when network overloaded
- *does not provide:* timing, minimum throughput guarantee, security
- *connection-oriented:* setup required between client and server processes

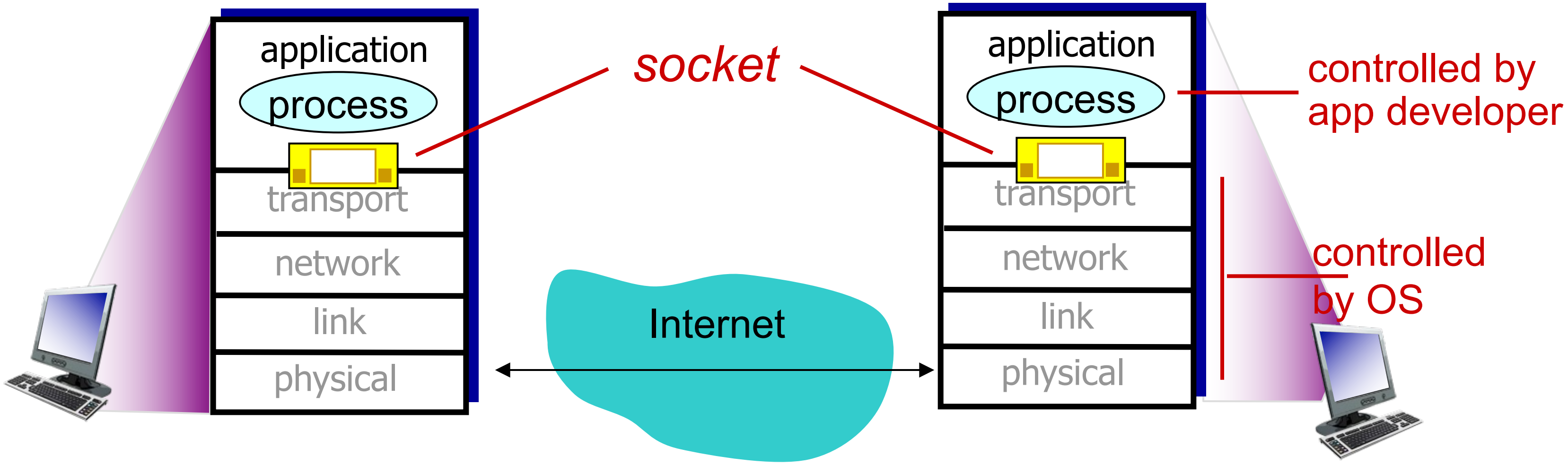
UDP service:

- *unreliable data transfer* between sending and receiving process
- *does not provide:* reliability, flow control, congestion control, timing, throughput guarantee, security, orconnection setup,

Q: why bother? Why is there a UDP?

Sockets

- process sends/receives messages to/from its socket
- socket analogous to door
  - sending process shoves message out door
  - sending process relies on transport infrastructure on other side of door to deliver message to socket at receiving process



Transport vs. network layer

- *network layer:* logical communication between hosts
- *transport layer:* logical communication between processes
  - relies on, enhances, network layer services

household analogy:

12 kids in Ann's house sending letters to 12 kids in Bill's house:

- hosts = houses
- processes = kids
- app messages = letters in envelopes
- transport protocol = Ann and Bill who demux to in-house siblings
- network-layer protocol = postal service

Transport vs. network layer

- *network layer:* logical communication between hosts
- *transport layer:* logical communication between processes
  - relies on, enhances, network layer services

household analogy:

12 kids in Ann's house sending letters to 12 kids in Bill's house:

- hosts = houses
- processes = kids
- app messages = letters in envelopes
- transport protocol = Ann and Bill who demux to in-house siblings
- network-layer protocol = postal service

UDP: User Datagram Protocol [RFC 768]

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

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

Internet transport protocols services

TCP service:

- *reliable transport* between sending and receiving process
- *flow control:* sender won't overwhelm receiver
- *congestion control:* throttle sender when network overloaded
- *does not provide:* timing, minimum throughput guarantee, security
- *connection-oriented:* setup required between client and server processes

UDP service:

- *unreliable data transfer* between sending and receiving process
- *does not provide:* reliability, flow control, congestion control, timing, throughput guarantee, security, orconnection setup,

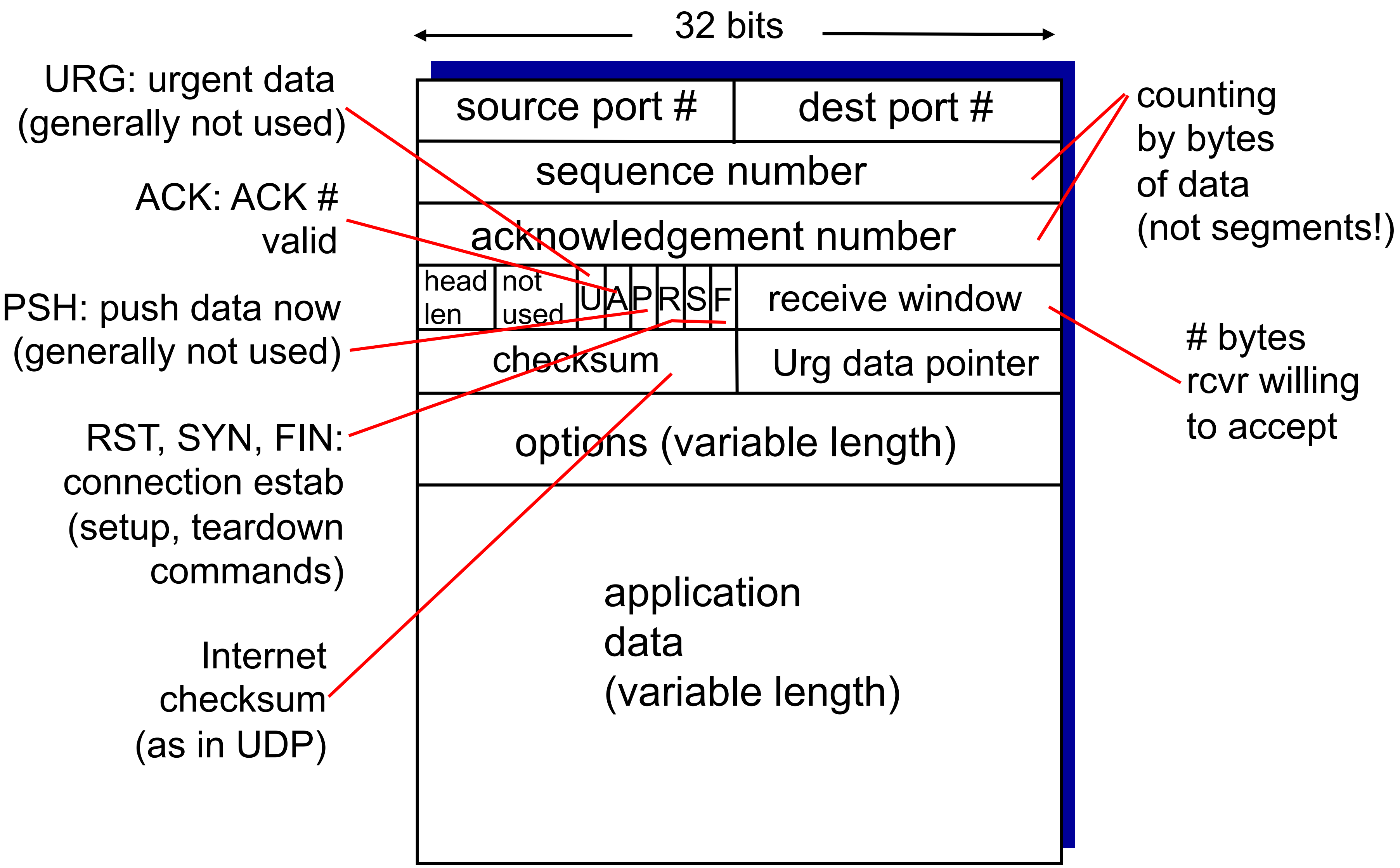
Q: why bother? Why is there a UDP?

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

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte steam:*
  - no "message boundaries"
- pipelined:
  - TCP congestion and flow control set window size

- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- connection-oriented:
  - handshaking (exchange of control msgs) inits sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

TCP segment structure



32 bits

source port #

dest port #

sequence number

acknowledgement number

head len

not used

U

A

P

R

S

F

receive window

checksum

Urg data pointer

options (variable length)

application data (variable length)

URG: urgent data (generally not used)

ACK: ACK # valid

PSH: push data now (generally not used)

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

Internet checksum (as in UDP)

counting by bytes of data (not segments!)

bytes rcvr willing to accept

TCP seq. numbers, ACKs

sequence numbers:

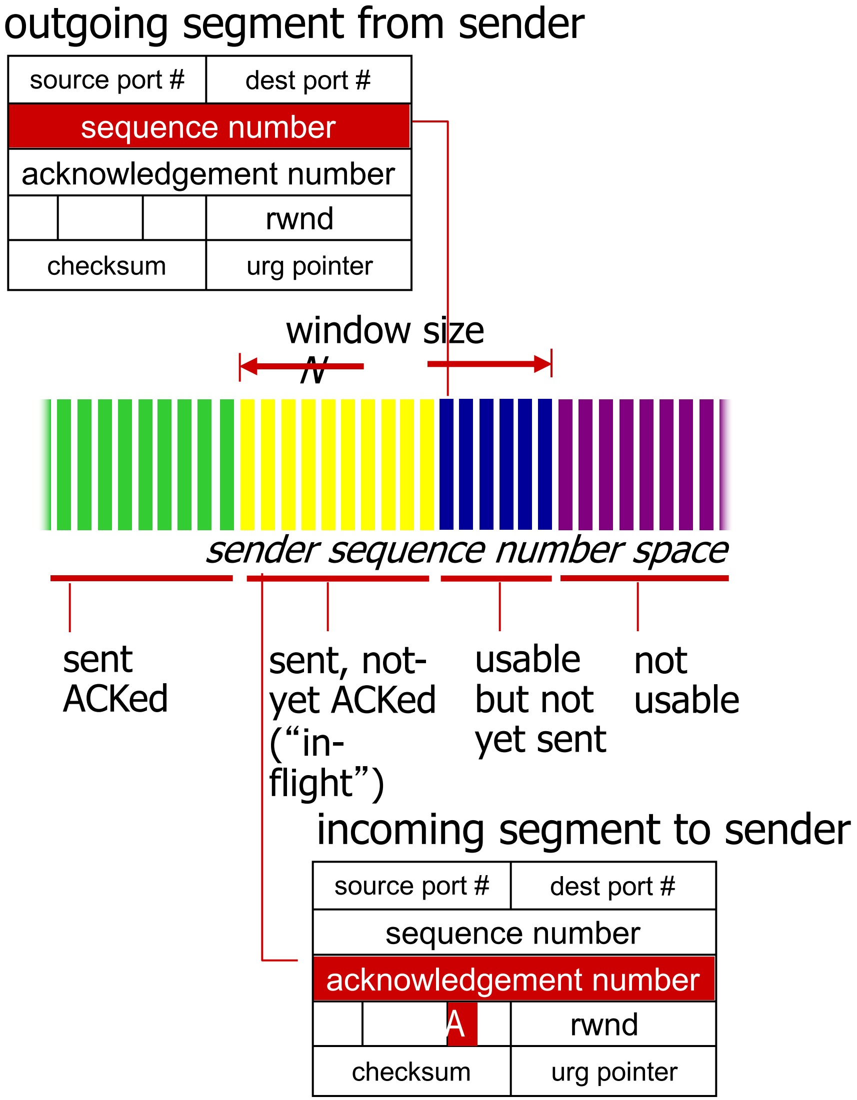
- byte stream “number” of first byte in segment’s data

acknowledgements:

- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments

- A: TCP spec doesn’t say, - up to implementor



outgoing segment from sender

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

window size
N

sender sequence number space

sent ACKed

sent, not-yet ACKed (“in-flight”)

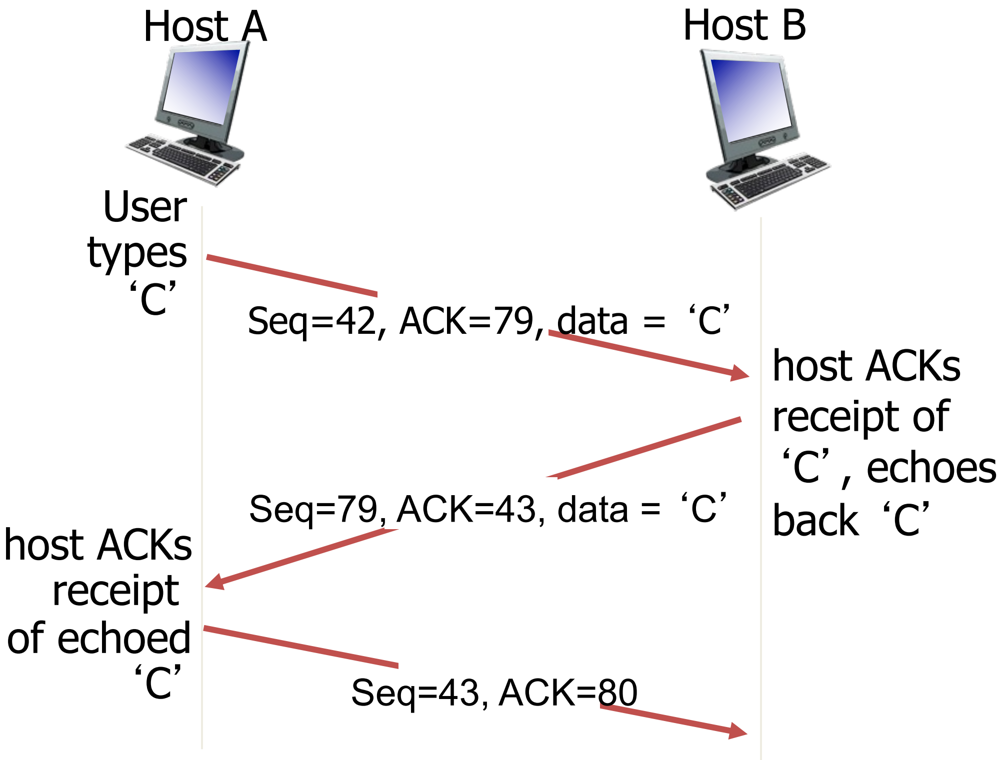
usable but not yet sent

not usable

incoming segment to sender

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

TCP seq. numbers, ACKs



simple telnet scenario

TCP sender events:

data rcvd from app:

- create segment with seq #
- seq # is byte-stream number of first data byte in segment
- start timer if not already running
  - think of timer as for oldest unacked segment
  - expiration interval: `TimeOutInterval`

timeout:

- retransmit segment that caused timeout
- restart timer

ack rcvd:

- if ack acknowledges previously unacked segments
  - update what is known to be ACKed
  - start timer if there are still unacked segments

Approaches towards congestion control

two broad approaches towards congestion control:

end-end congestion control:

- no explicit feedback from network
- congestion inferred from end-system observed loss, delay
- approach taken by TCP

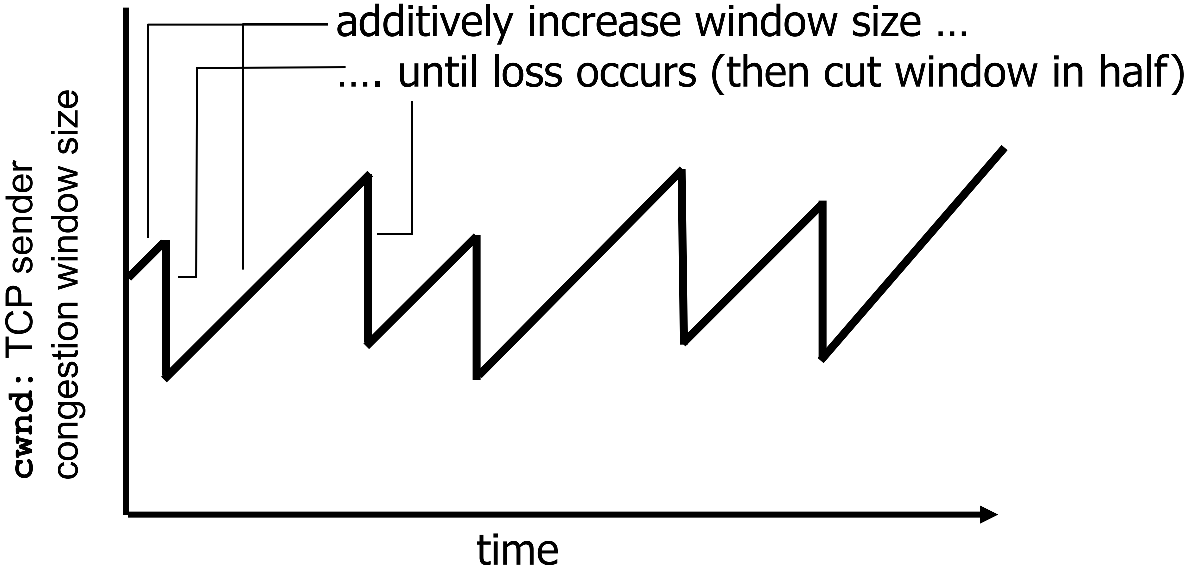
network-assisted congestion control:

- routers provide feedback to end systems
  - single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
  - explicit rate for sender to send at

TCP congestion control: additive increase multiplicative decrease

- *approach:* sender increases transmission rate (window size), probing for usable bandwidth, until loss occurs
  - *additive increase:* increase **`cwnd`** by 1 MSS every RTT until loss detected
  - *multiplicative decrease:* cut **`cwnd`** in half after loss

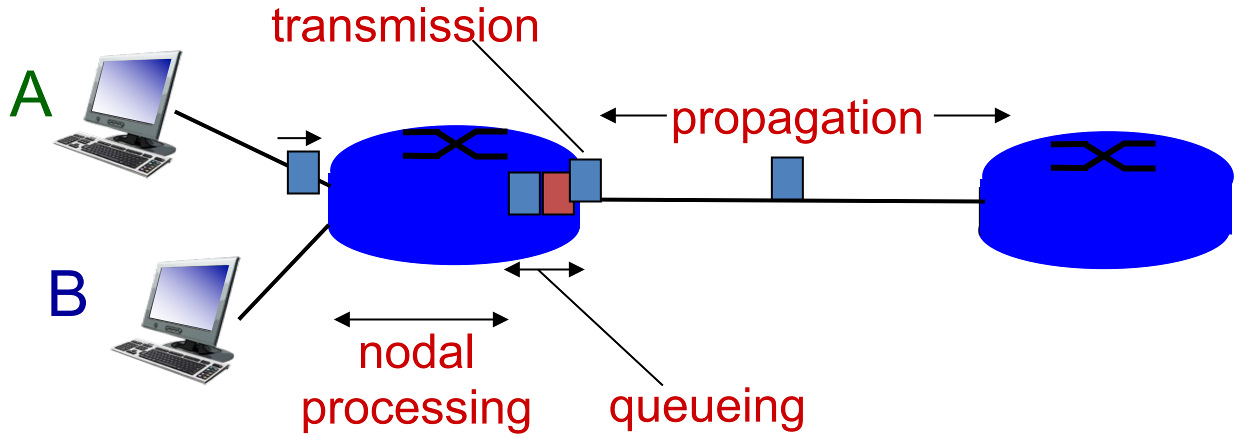
AIMD saw tooth behavior: probing for bandwidth



TCP Performance

- Bandwidth = 1/RTT*(sqrt(2/3)*packet-loss-probability)

Four sources of packet delay



$$d_{\text{nodal}} = d_{\text{proc}} + d_{\text{queue}} + d_{\text{trans}} + d_{\text{prop}}$$

d_{proc}: nodal processing

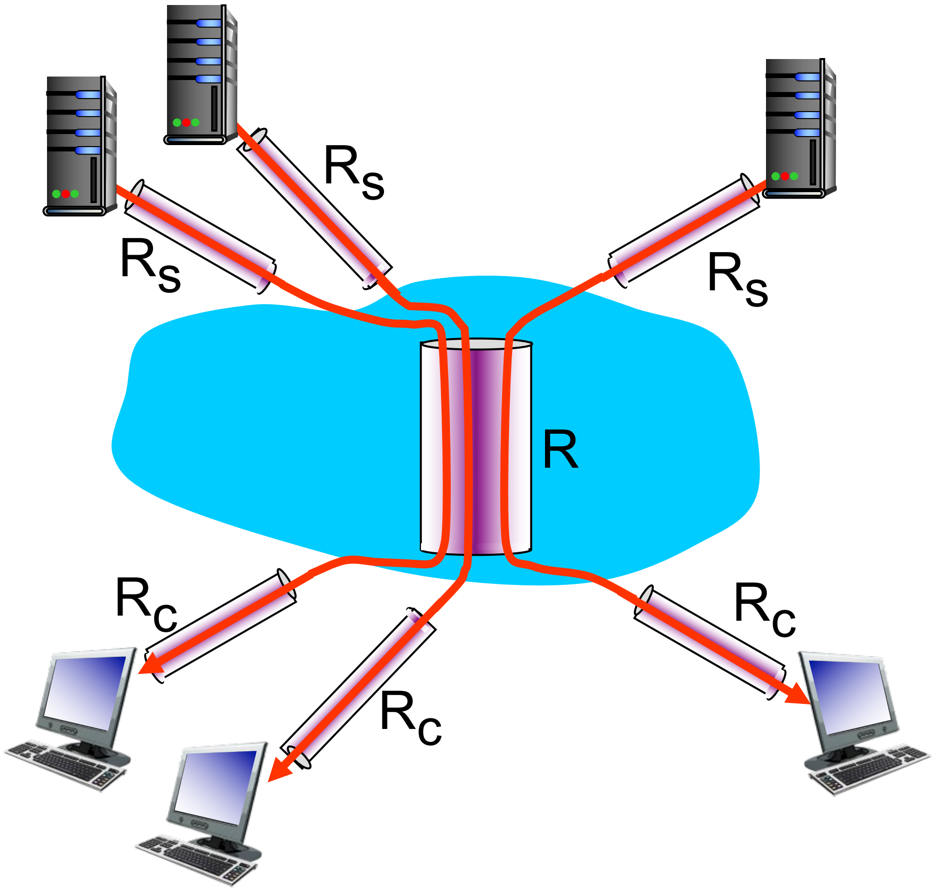
- check bit errors
- determine output link
- typically < msec

d_{queue}: queueing delay

- time waiting at output link for transmission
- depends on congestion level of router

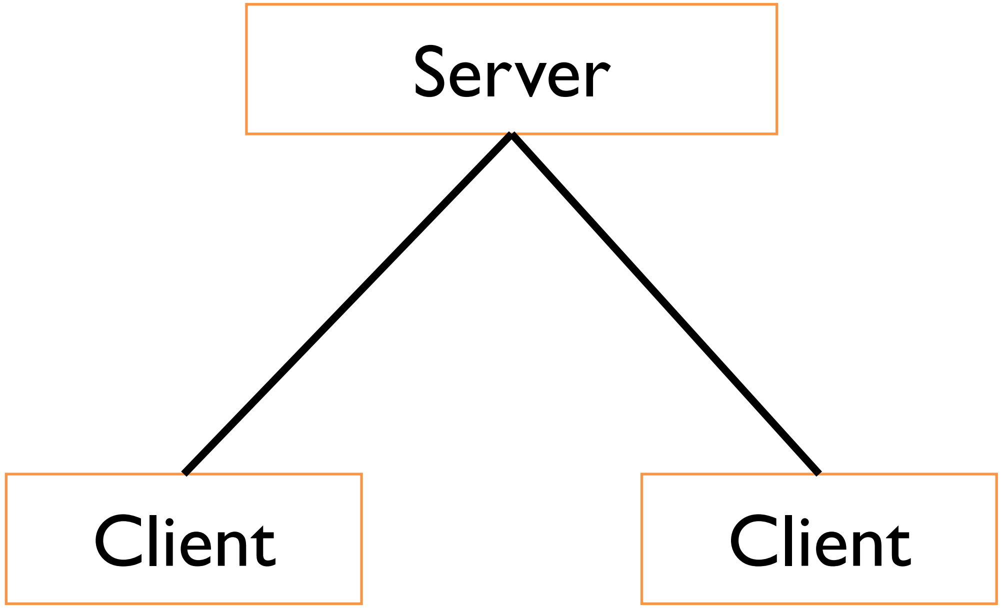
Throughput: Internet scenario

- per-connection end-end throughput: min(R_c,R_s,R/10)
- in practice: R_c or R_s is often bottleneck



10 connections (fairly) share backbone bottleneck link R bits/sec

Client-server architecture



Server:

- always-on host
- permanent IP address
- data centers for scaling

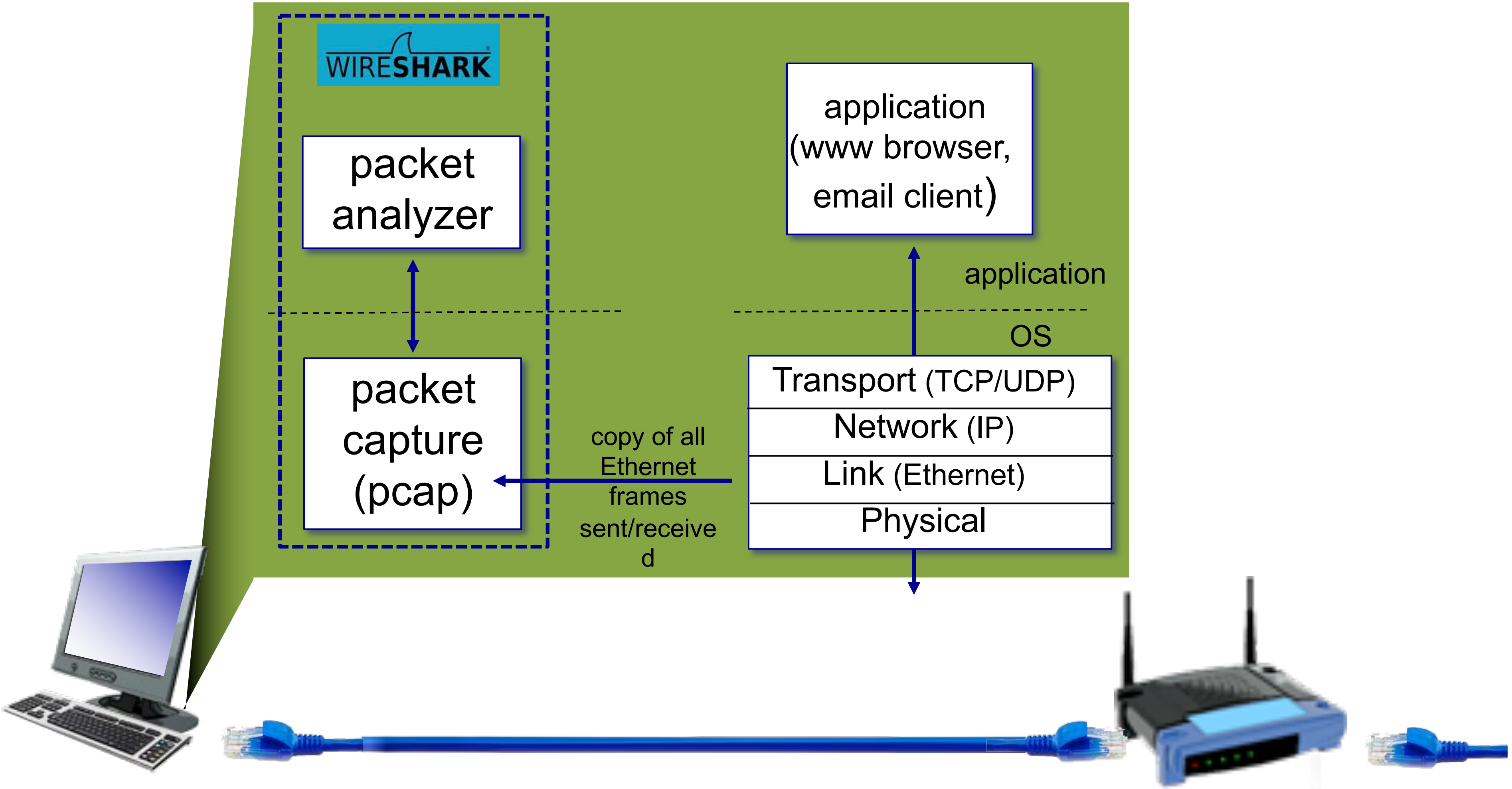
Clients:

- communicate with server
- may be intermittently connected
- may have dynamic IP addresses
- do not communicate directly with each other

Higher Level Networking

- Client/server code abstracted out (python's twisted framework)
- Message queues: Kafka, ZeroMQ, etc
- Durability of messages (can persist on disk)
- Message lifetimes (time to live)
- Filtering, queueing policies
- Batching policies
- Delivery policies (at most once, at least once, etc)

Debugging Networks: Packet Capture



Separation of Concerns

- Break problem into separate parts
- Solve each problem independently
- Encapsulate data across layers
- Protocol: Rules for communication within same layer
- Service: Abstraction provided to layer above
- API: Concrete way of using that service
- Layering+Encapsulation Example