50.012 Networks

Lecture 2: Application Layer Overview & Socket Programming

+ Lab 1 Introduction

2021 Term 6

Assoc. Prof. CHEN Binbin

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

Learning objectives today

- Concepts in network application layer
  - client-server paradigm
  - peer-to-peer paradigm
  - Application layer protocols
  - transport-layer service models
- Creating network applications
  - socket programming

Some network apps

- E-mail, remote login
- Web, search
- text messaging
- P2P file sharing
- multi-user network games
- streaming stored video (YouTube, Netflix, Kankan)
- voice over IP (e.g., Skype)
- real-time video conferencing
- social networking
- Dropbox, GoogleDoc
- AR/VR
- IoT
- …

Creating a network app

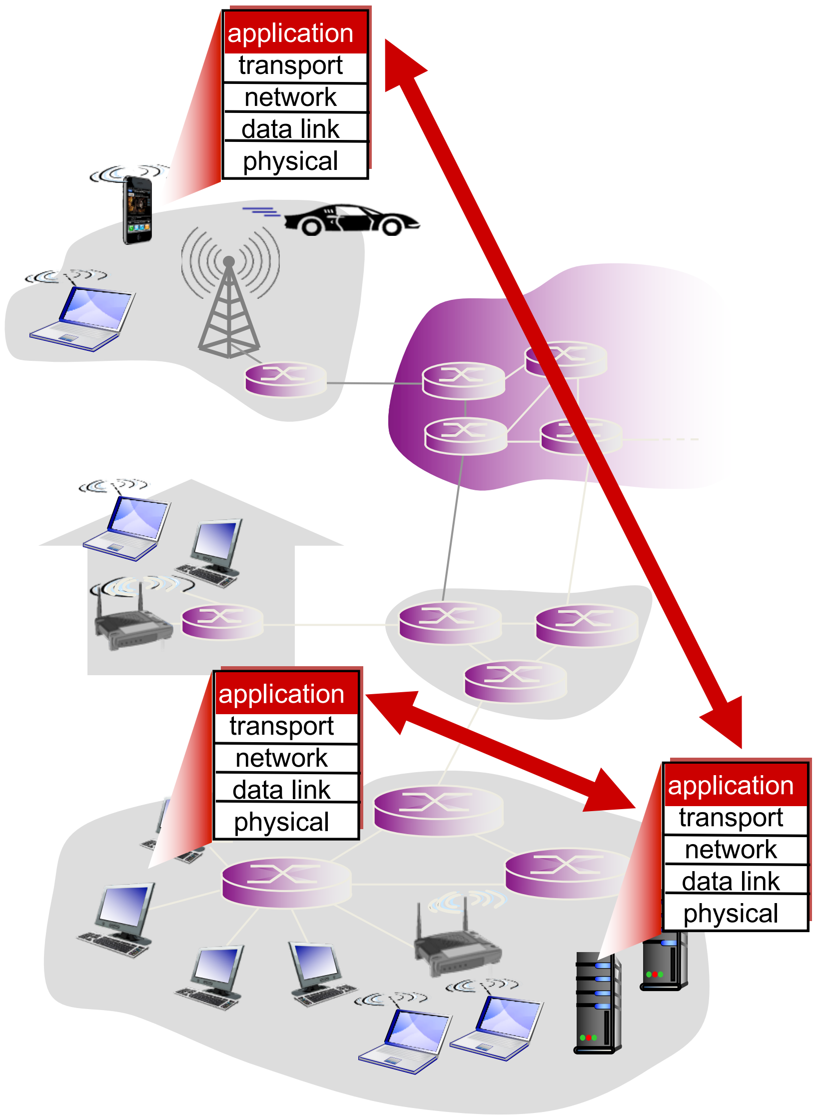
write programs that:

- run on (different) *end systems*
- communicate over network
- e.g., web server software communicates with browser software

for network-core devices

- they do not need to run user-supplied applications

Running applications on end systems allows for rapid app development, propagation

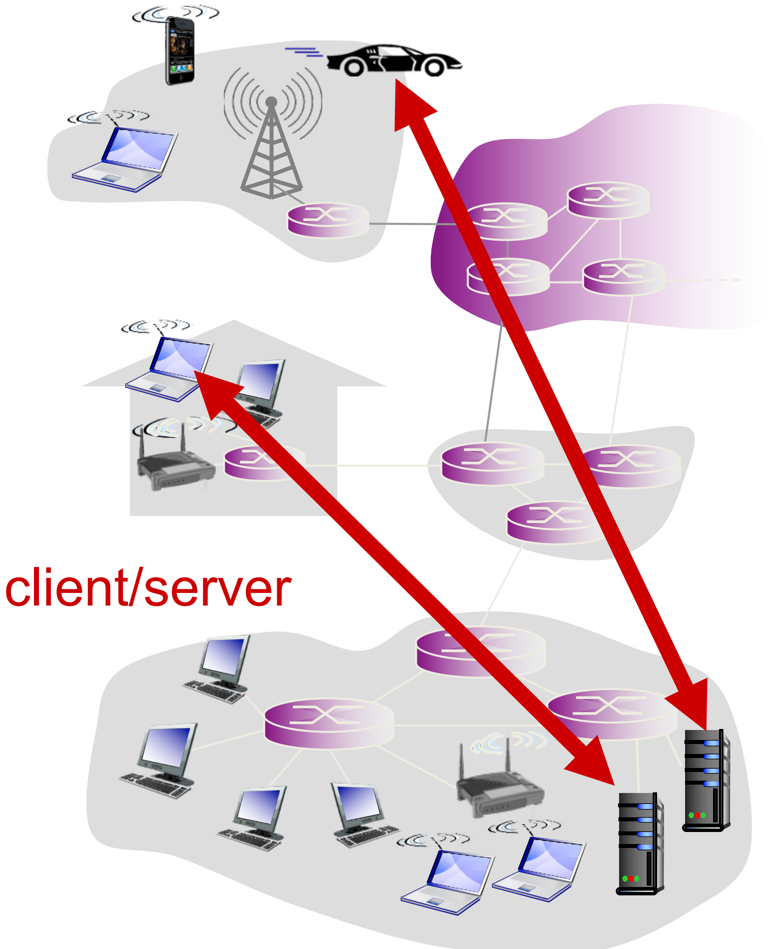


Application architectures

possible structure of applications:

- client-server
- peer-to-peer (P2P)

Client-server architecture



server:

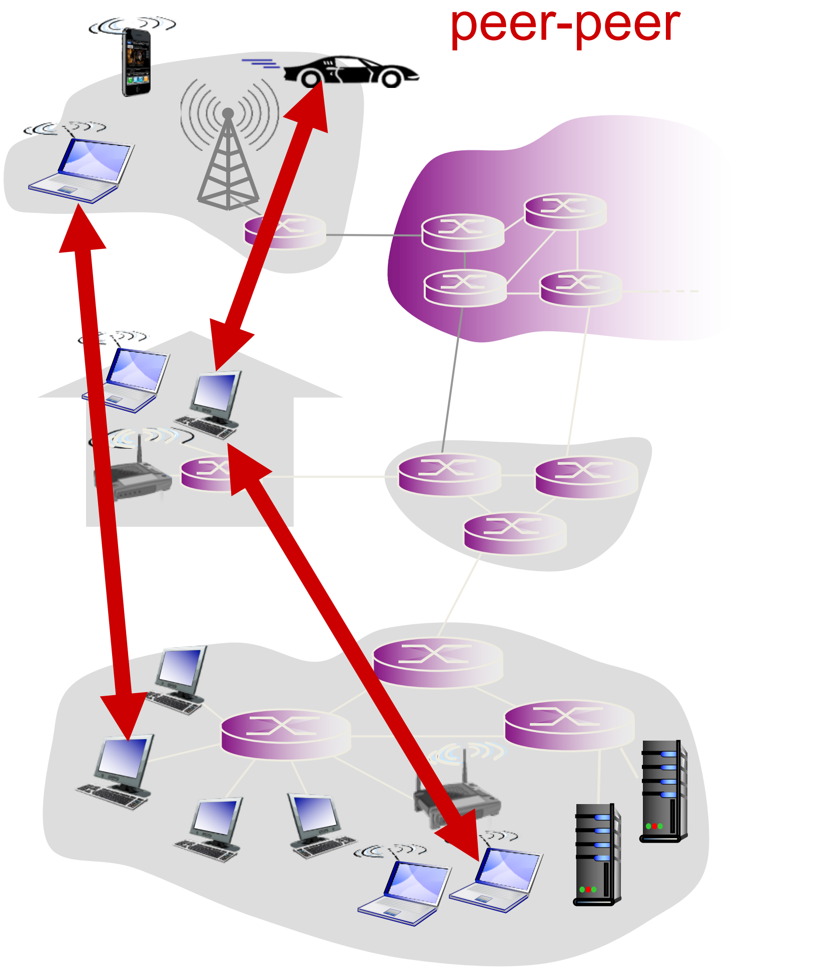
- always-on host
- permanent IP address
- data centers for scaling

clients:

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

P2P architecture

- *no* always-on server
- arbitrary end systems directly communicate
- peers request service from other peers, provide service in return to other peers
  - *self scalability* – new peers bring new service capacity, as well as new service demands
- peers are intermittently connected and change IP addresses
  - complex management



Processes communicating

process: program running within a host

- within same host, two processes communicate using inter-process communication (defined by OS)
- processes in different hosts communicate by exchanging messages

clients, servers

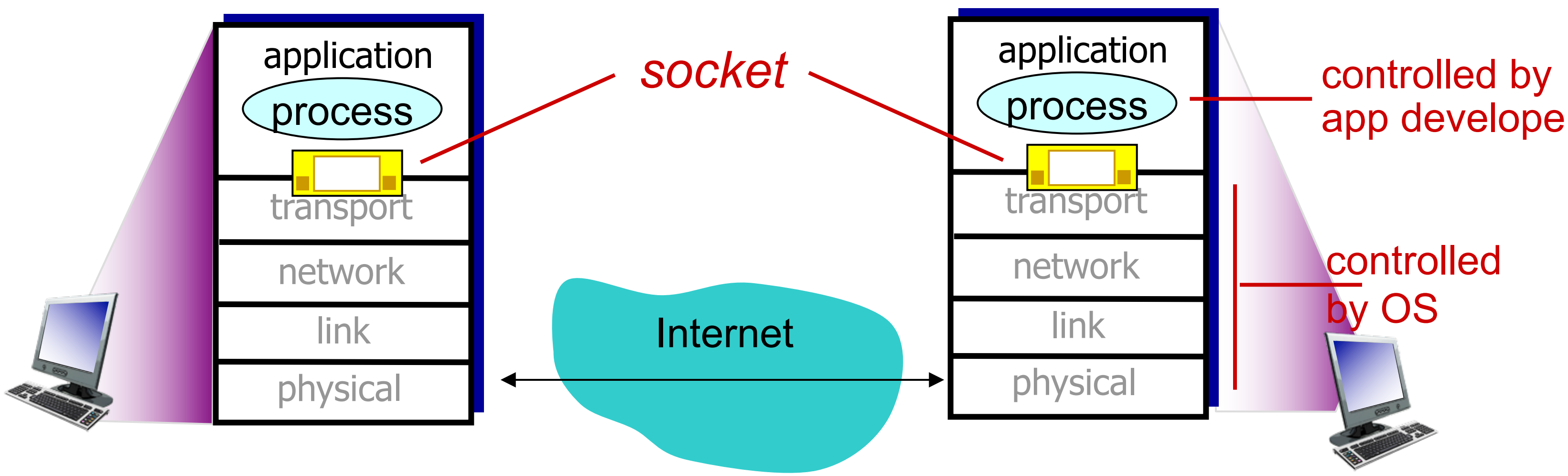
client process: process that initiates communication

server process: process that waits to be contacted

- Note: applications with P2P architectures also have client processes & server processes

Sockets

- process sends/receives messages to/from its socket
- socket analogous to door (*a door between a factory's manufacturing area & its packaging area*)
  - sending process shoves message out of the door
  - sending process relies on transport infrastructure on other side of door to deliver message to socket at receiving process



Addressing processes

- to receive messages, process must have *identifier*
- host device has unique 32-bit IP address
- *<u>Q:</u>* does IP address of host on which process runs suffice for identifying the process?

- *identifier* includes (type of transport protocol), IP address, and port numbers associated with process on host.
- example port numbers:
  - HTTP server: 80
  - mail server: 25
- to send HTTP message to gaia.cs.umass.edu web server:
  - IP address: 128.119.245.12
  - port number: 80
- more shortly…

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:

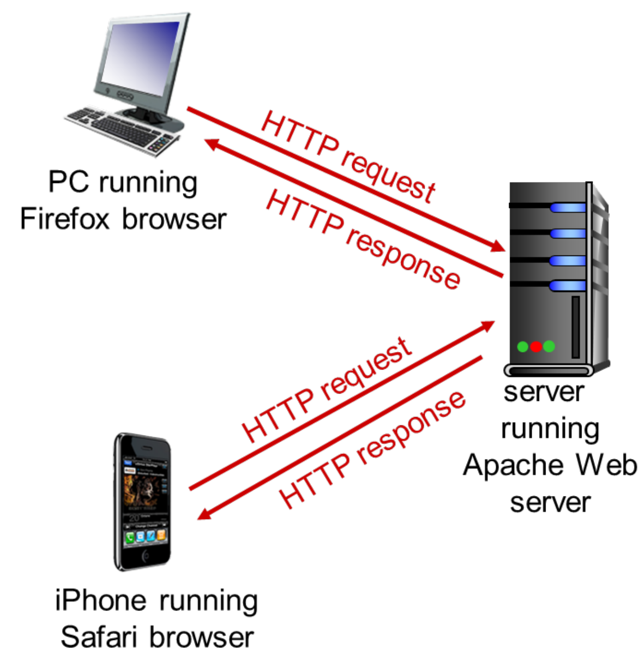
- defined in RFCs
- allows for interoperability
- e.g., HTTP, SMTP

proprietary protocols:

- e.g., Skype

A brief review of HTTP

- HTTP: hypertext transfer protocol
- HTTP is Web's application layer protocol
- Client/server model
  - Request
  - Response



HTTP/0.9 -> HTTP/1.0 -> HTTP/1.1 -> HTTP/2 -> HTTP/3

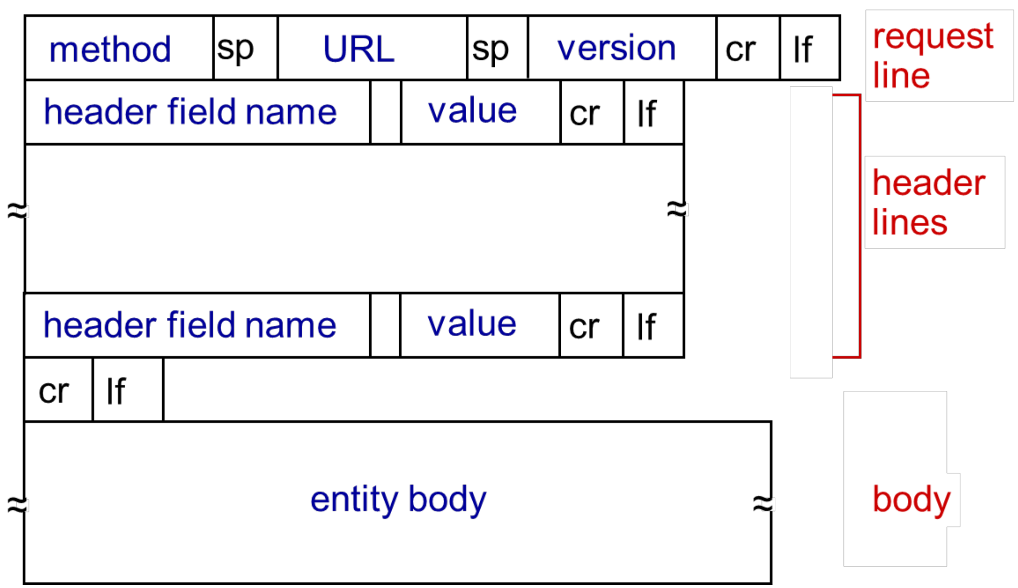
HTTP/3 relies on QUIC as the underlying transport

HTTP REQUEST

- HTTP GET is a common HTTP method (verb): the client requests the server to send it a resource identified by an URL
  - ASCII (human-readable format)
  - Other HTTP "verbs": PUT, POST, DELETE, HEAD, OPTIONS, CONNECT, . . .

- Example:

```
GET /index.html HTTP/1.1\r\n
Host: www-net.cs.umass.edu\r\n
User-Agent: Firefox/3.6.10\r\n
Accept: text/html,application/xhtml+xml\r\n
Accept-Language: en-us,en;q=0.5\r\n
Accept-Encoding: gzip,deflate\r\n
Accept-Charset: ISO-8859-1,utf-8;q=0.7\r\n
Keep-Alive: 115\r\n
Connection: keep-alive\r\n
\r\n
```



HTTP RESPONSE

- Status line: first line of the HTTP response, includes:
  - A numeric status code
  - Sentence with reason

status line (protocol status code status phrase)

header lines

data, e.g., requested HTML file

```
HTTP/1.1 200 OK\r\n
Date: Sun, 26 Sep 2010 20:09:20 GMT\r\n
Server: Apache/2.0.52 (CentOS)\r\n
Last-Modified: Tue, 30 Oct 2007 17:00:02
   GMT\r\n
ETag: "17dc6-a5c-bf716880"\r\n
Accept-Ranges: bytes\r\n
Content-Length: 2652\r\n
Keep-Alive: timeout=10, max=100\r\n
Connection: Keep-Alive\r\n
Content-Type: text/html; charset=ISO-8859-
   1\r\n
\r\n
data data data data data ...
```

- Common status codes:

200 OK
- request succeeded, requested object later in this msg

301 Moved Permanently
- requested object moved, new location specified later in this msg (Location:)

400 Bad Request
- request msg not understood by server

404 Not Found
- requested document not found on this server

505 HTTP Version Not Supported

What transport service does an app need?

process addressing

- A must

reliable data transfer

- 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
- Also consider the requirements of receiver & network…

security

Transport service requirements: common apps

application	data loss	throughput	time sensitive
file transfer	no loss	elastic	no
e-mail	no loss	elastic	no
Web documents	no loss	elastic	Yes and no
real-time audio/video	loss-tolerant	audio: 5kbps-1Mbps video:10kbps-5Mbps	100's msec
stored audio/video	loss-tolerant	same as above	few secs
interactive games	loss-tolerant	few kbps up	100's msec
text messaging	no loss	elastic	yes and no

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, or connection setup,

Q: why bother? Why is there a UDP?

Internet apps: application, transport protocols

application	application layer protocol	underlying transport protocol
e-mail	SMTP [RFC 2821]	TCP
remote terminal access	Telnet [RFC 854]	TCP
Web	HTTP/1.1 [RFC 2616]	TCP (*Note: HTTP/3 is on QUIC*)
file transfer	FTP [RFC 959]	TCP
streaming multimedia	HTTP, RTP [RFC 1889]	TCP or UDP
Internet telephony	SIP, RTP, proprietary (e.g., Skype)	TCP or UDP
domain name lookup	DNS	UDP

Securing TCP

TCP & UDP

- no encryption
- cleartext passwds sent into socket traverse Internet in cleartext

Secure Sockets Layer (SSL) / Transport Layer Security (TLS)

- provides encrypted TCP connection
- data integrity
- end-point authentication

SSL and TLS protocols

Protocol	Published	Status
SSL 1.0	Unpublished	Unpublished
SSL 2.0	1995	Deprecated in 2011 (RFC 6176)
SSL 3.0	1996	Deprecated in 2015 (RFC 7568)
TLS 1.0	1999	Deprecation planned in 2020[11]
TLS 1.1	2006	Deprecation planned in 2020[11]
TLS 1.2	2008	
TLS 1.3	2018	

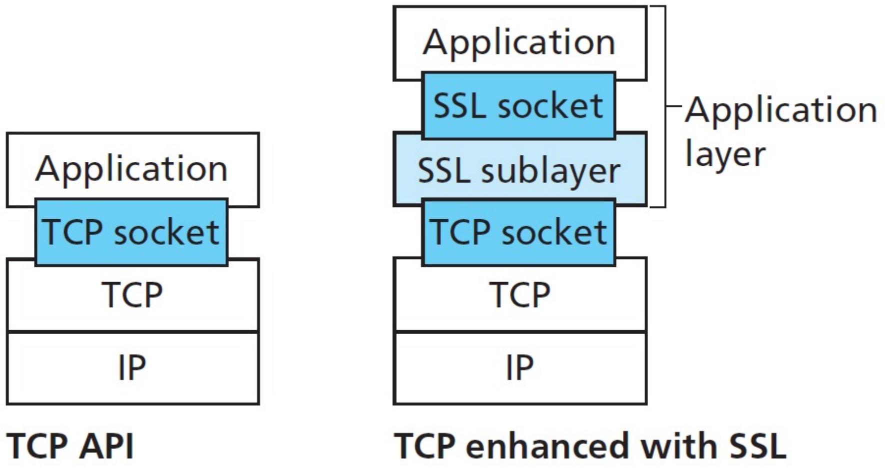
Securing TCP

TLS is at app layer

- apps use SSL libraries, that “talk” to TCP

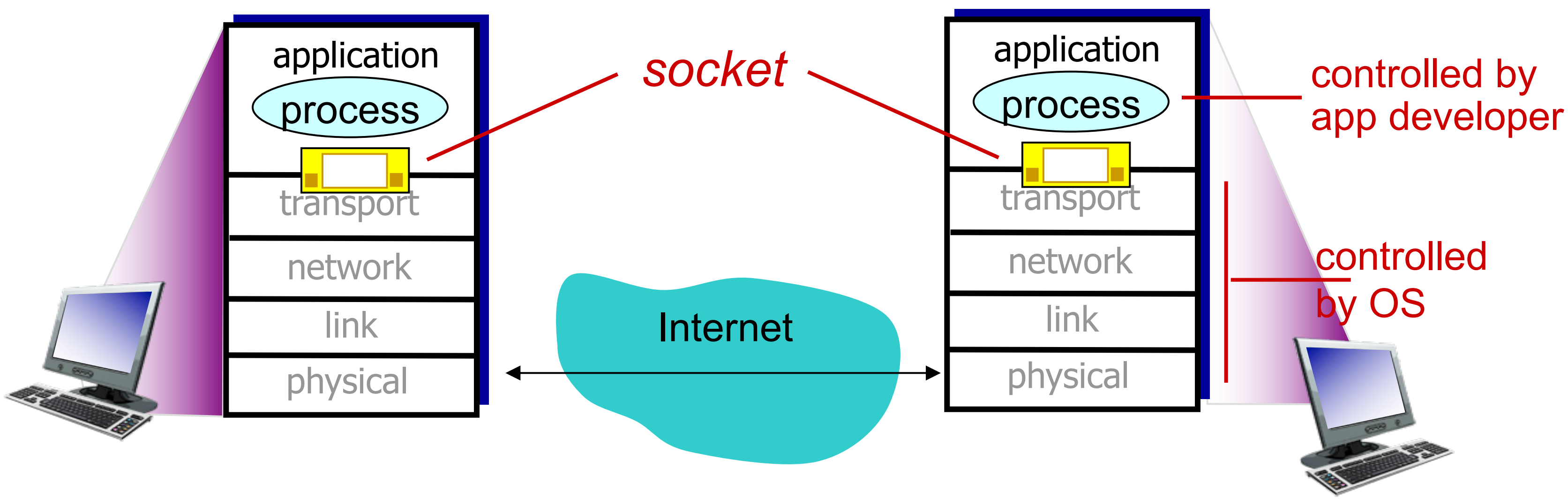
TLS socket API

- cleartext passwords sent into socket traverse Internet encrypted



Socket programming

goal: learn how to build client/server applications that communicate using sockets



Socket programming

Two socket types for two transport services:

– *UDP (User Datagram Protocol):* unreliable datagram

– *TCP (Transmission Control Protocol):* reliable, byte stream-oriented

We will use a simple network app as 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*

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

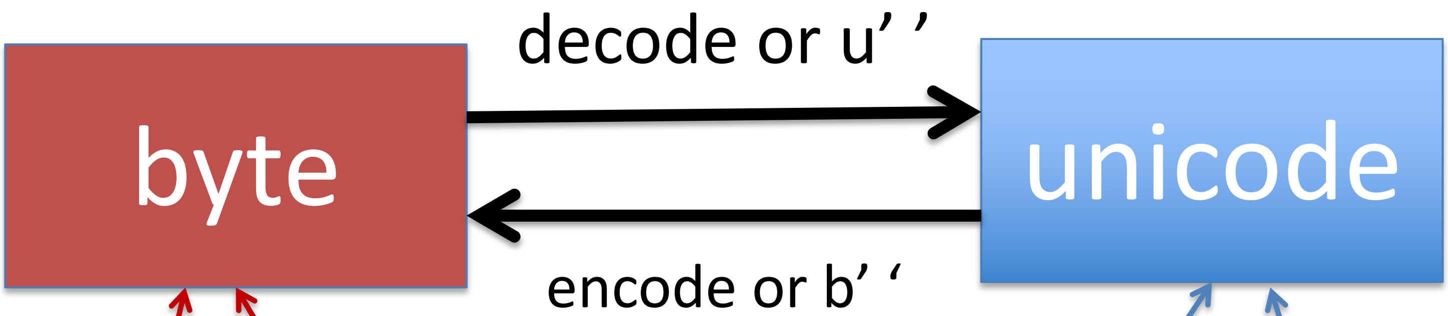
Attach server name, port to message; send into socket

read reply characters from socket into string

print out received string and close socket

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

Conversion between byte & unicode str in Python 3



clientSocket.sendto(message.encode(),
(serverName, serverPort))

message = raw_input(’Input lowercase sentence:’)

modifiedMessage, serverAddress =
clientSocket.recvfrom(2048)

print(modifiedMessage.decode())

https://nedbatchelder.com/text/unipain/unipain.html#1

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 True:
    message, clientAddress = serverSocket.recvfrom(2048)
    modifiedMessage = message.decode().upper()
    serverSocket.sendto(modifiedMessage.encode(),
                        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:

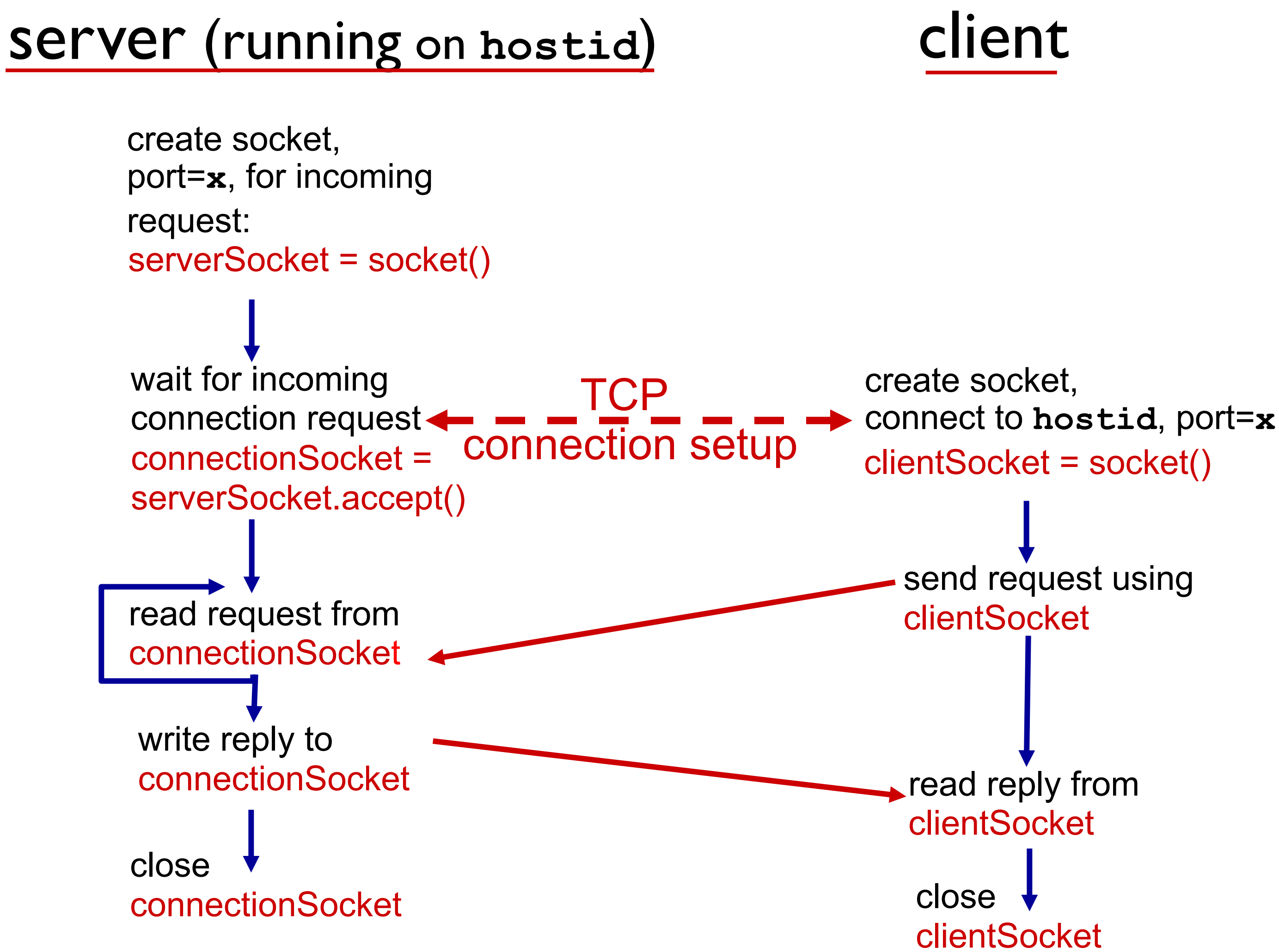
- Creating TCP socket, specifying IP address, port number of server process
- *when client creates socket:* client TCP establishes connection to server TCP

- 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



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.encode())
modifiedSentence = clientSocket.recv(1024)
print(‘From Server:’, modifiedSentence.decode())
clientSocket.close()
```

Example app: TCP server

Python TCPServer

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

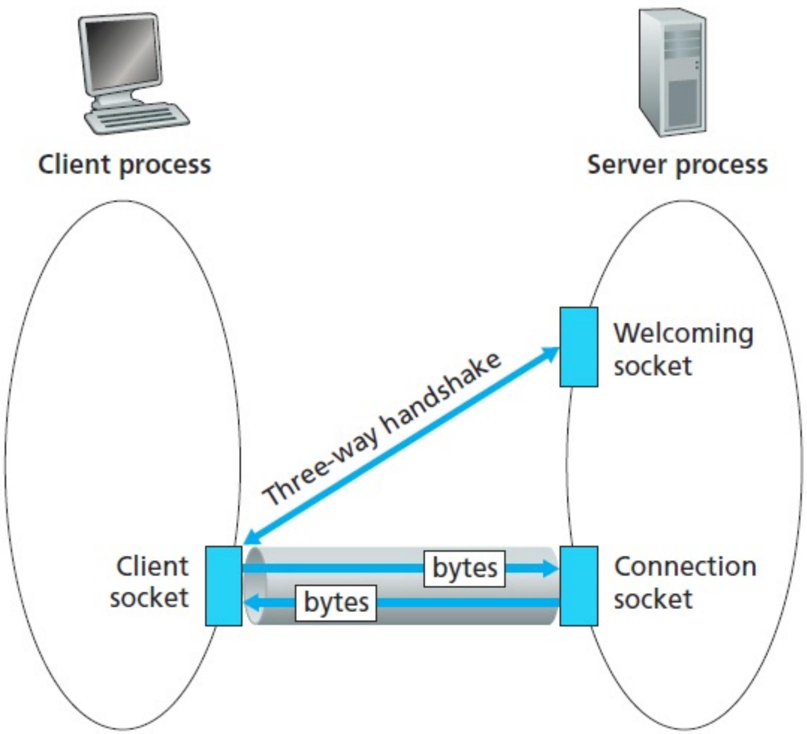
read bytes from socket (but not address as in UDP)

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

```
 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 True:
    connectionSocket, addr = serverSocket.accept()

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

TCP sockets



Client IP: open, client port: open
Server IP: Y.Y.Y.Y Server port: 80

Client IP: X.X.X.X, client port: a
Server IP: Y.Y.Y.Y Server port: 80

Comparison: UDP vs. TCP socket

- Socket establishment

UDP client

```
clientSocket = socket(AF_INET,
                      SOCK_DGRAM)
```

TCP client

```
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName,serverPort))
```

UDP server

```
serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind(('', serverPort))
```

TCP server

```
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind((‘’,serverPort))
serverSocket.listen(1)
```

Comparison: UDP vs. TCP socket

- Use of socket

UDP client

```
clientSocket.sendto(message.encode(),
                    (serverName, serverPort))
modifiedMessage, serverAddress =
                    clientSocket.recvfrom(2048)
```

TCP client

```
clientSocket.send(sentence.encode())
modifiedSentence = clientSocket.recv(1024)
```

UDP server

```
message, clientAddress =
serverSocket.recvfrom(2048)
serverSocket.sendto(message, clientAddress)
```

TCP server

```
connectionSocket, addr = serverSocket.accept()
message = connectionSocket.recv(1024)
connectionSocket.send(message)
```

Lab1: A Simple Web Proxy Server

- Hand-out: 17th September (Friday)
  - Materials will be released before the lab starts
- Hand-in: 28th September 23:59 (next next Tue)

- Late policy: no late submission is allowed
  - You can submit multiple times, so do submit a working version early to avoid last-minute rush (you can submit a partially working version too, and explain which features work)
  - Let me know if you cannot submit for the second time (I may forget to change the default setting)
  - We will mark your last submission

How will we use the 2 hours on Friday morning?

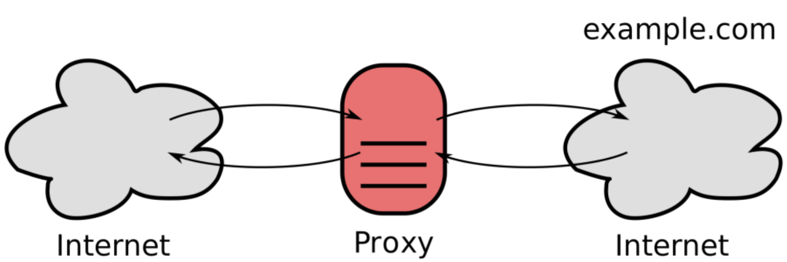
- Opening: 9am. I will introduce some necessary background and the lab requirements
- Rendezvous Point (RP): 10:30am
- Self-paced between the opening and the RP. You may ask questions over the MS Teams chat, and you are encouraged to help answer others' questions (but do not share your solution directly!)
- We will discuss common questions together during the RP

HTTP Proxies

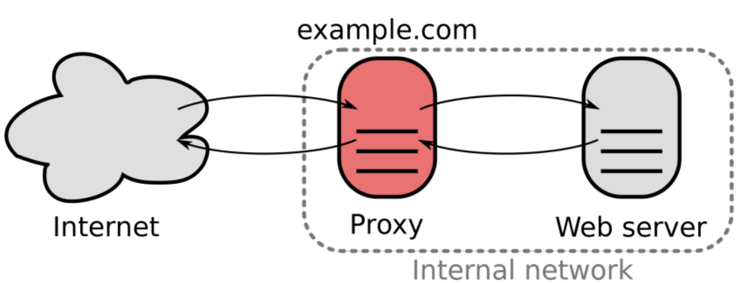
- Proxy: an entity authorized to act on behalf of another
  - An intermediate server that is performing requests for us
- A caching proxy keeps copies of resources for the client
  - E.g., results of HTTP GET queries
  - Results of non-idempotent operations (e.g., POST) are not cached
- These cached results are served to subsequent queries
  - These clients do not have to be the same as original clients
  - As long as GET was requesting the same resource

Types of Proxies

- Forward (Open) Proxies
  - Content accelerators: by reducing delay and load on outgoing connections
  - Content filters / access control (or flip it: bypass filtering)
  - Content logging and eavesdropping (or flip it: accessing services anonymously)
- Reverse Proxies can also cache queries in front of servers
  - Application firewall
  - TLS acceleration
  - Distribute the load, A/B testing, and multivariate testing
  - Accelerators: Cache / compression



An open proxy



A reverse proxy

QUESTIONS?