Application Layer

1 Application Layer

Network Architecture:

- Client-server architecture
- P2P Architecture

Processes and Socket programming

- TCP
- UDP

2 Creating a network app

Write programs that:

- Run on (different) end systems
- Communicate over network

No need to write software for network-core devices

- Network-core devices do not run user applications
- Applications on end systems allow for rapid app development, propagation

3 Application Architectures

3.1 Client-Server Architecture

Server:

- Always-on host
- Fixed (static) IP address
- Data centres for scaling

Clients

- Communicate with server
- May be intermittently connected
- May have dynamic IP addresses
- Do not communicate directly with each other

3.2 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 service capacity, as well as new service demands
- Peers are intermittently connected and change IP addresses

3.3 Hybrid

- Often there is a hybrid architecture, an example of this might be video calling, the initial connection and authentication might be handled by a server, but the actual video will be sent via P2P

4 Processes communicating

Definition: Process

A program running within a host

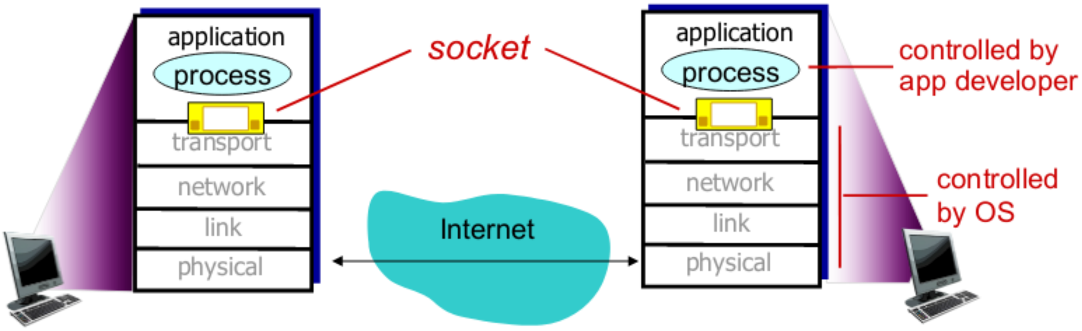
Definition: Socket

A software mechanism that allows a process to create and send messages into, and receive messages from the network

- Within same host, two processes communicate using inter-process communication (defined by OS)
- Processes in different hosts communicate by exchanging messages
- A process is analogous to a house, and its socket is analogous to its door
- Interface between application layer and transport layer

5 Sockets

- Process sends/receives messages to/from its socket
- A socket shoves message out of the door
- Sending process relies on transport infrastructure on the other side of the door to deliver message to a socket at the receiving process



6 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

Security:

- Encryption, data integrity, ...

Timing:

- Some apps (e.g. internet telephony, interactive games) require low delay to be ”effective”

7 Internet Transport Protocols Service

TCP Service:

- **Connection-Oriented**: Setup required between client and server process
- **Reliable transport** between sending and receiving processes
- **Flow control**: Sender won't overwhelm receiver
- **Full-duplex connection**: Connection can send messages to each other at the same time

UDP service:

- **Unreliable data transfer** between sending and receiving processes
- **Does not provide** reliability, flow control, timing, security or connection setup

8 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

Definition: Open protocols

Defined in Request For Comments (RFC) Allow for interoperability e.g. HTTP, SMTP

9 HTTP Overview

Definition: HTTP (Hypertext transfer protocol)

Web's application layer protocol

HTTP 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

Important: HTTP

HTTP is stateless - server maintains no information about past client requests

Definition: Client/Server Model

Client - Browser that requests, receives, (using HTTP Protocol) and ”displays” web objects
Server: Web server sends (using HTTP protocol) objects in response to requests

10 HTTP Connections

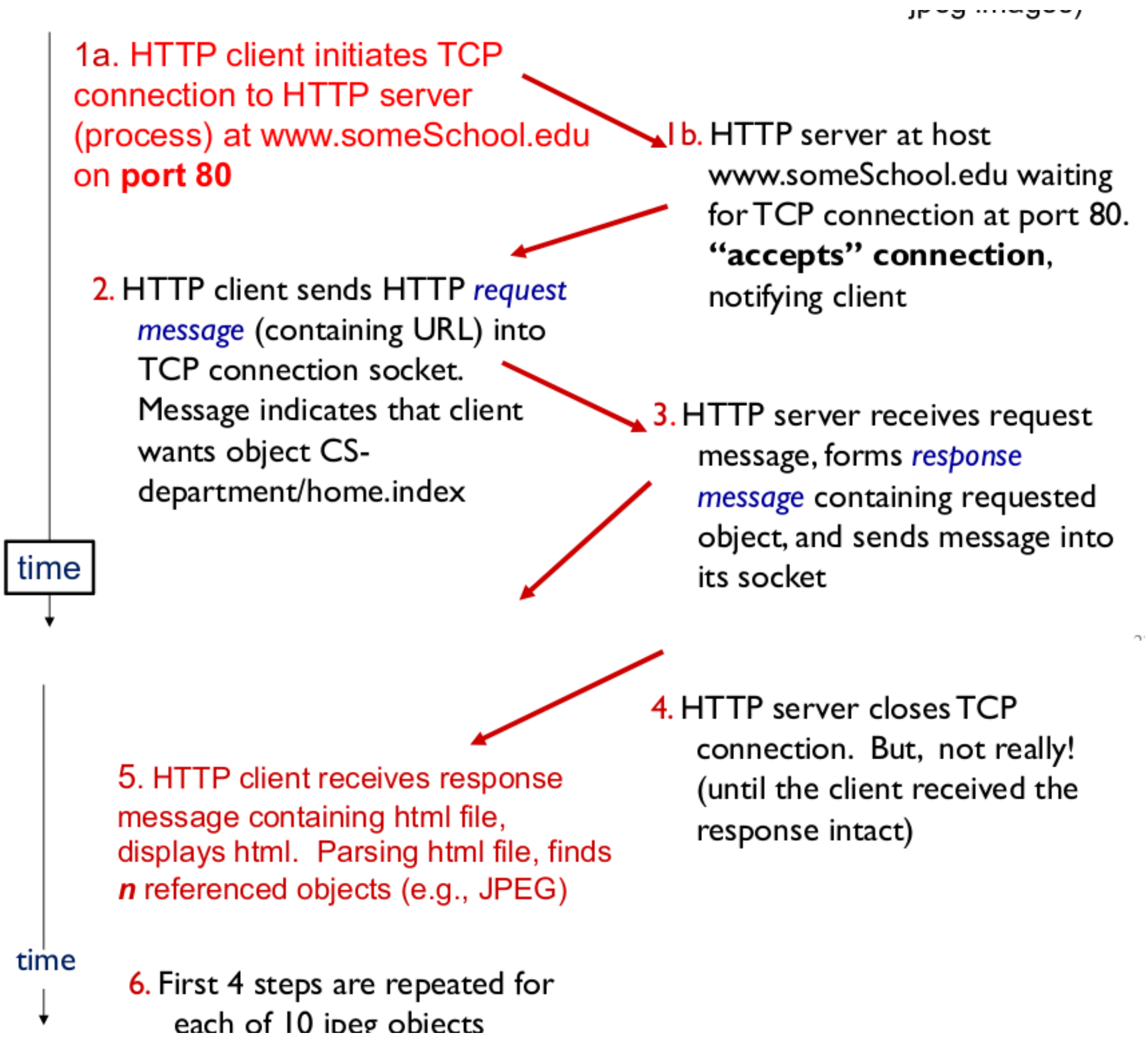
Non persistent HTTP

- At most one object sent over TCP connection, connection then closed

Persistent HTTP

- Multiple objects can be sent over single TCP connection, between client, server

10.1 Non-persistent HTTP


1a. HTTP client initiates TCP connection to HTTP server (process) at www.someSchool.edu on **port 80**

1b. HTTP server at host www.someSchool.edu waiting for TCP connection at port 80. **"accepts" connection**, notifying client

2. HTTP client sends HTTP *request message* (containing URL) into TCP connection socket. Message indicates that client wants object CS-department/home.index

3. HTTP server receives request message, forms *response message* containing requested object, and sends message into its socket

4. HTTP server closes TCP connection. But, not really! (until the client received the response intact)

5. HTTP client receives response message containing html file, displays html. Parsing html file, finds ***n*** referenced objects (e.g., JPEG)

6. First 4 steps are repeated for each of 10 jpeg objects

10.1.1 Response Time

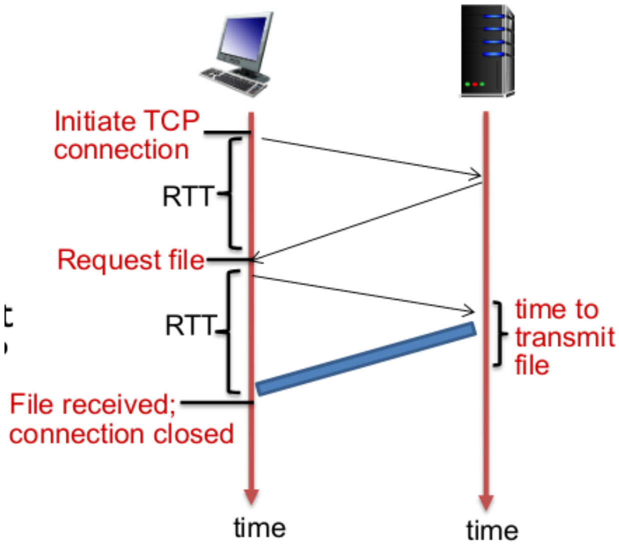
Definition: RTT (Round trip time)

The for a small packet to travel from client to server and back

HTTP Response time:

- One RTT to initiate TCP connection
- One RTT for HTTP request and first few bytes of HTTP response to return
- File transmission time

- Non-persistent HTTP response time = 2RTT + file transmission time
  Incurred for each file



10.2 Persistent HTTP

- Server leaves connection open after sending response
- Subsequent HTTP messages between same client/server sent over open connection
- Client sends requests as soon as it encounters a referenced object
- Takes as little as one RTT + file transmission time total
  - Assuming connections to server already established
  - Assuming all files requested in parallel

11 Socket Programming

Goal: learn how to build client/server applications that communicate using sockets
Socket: door between application process and end-end-transport protocol

Two socket types for two transport services:

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

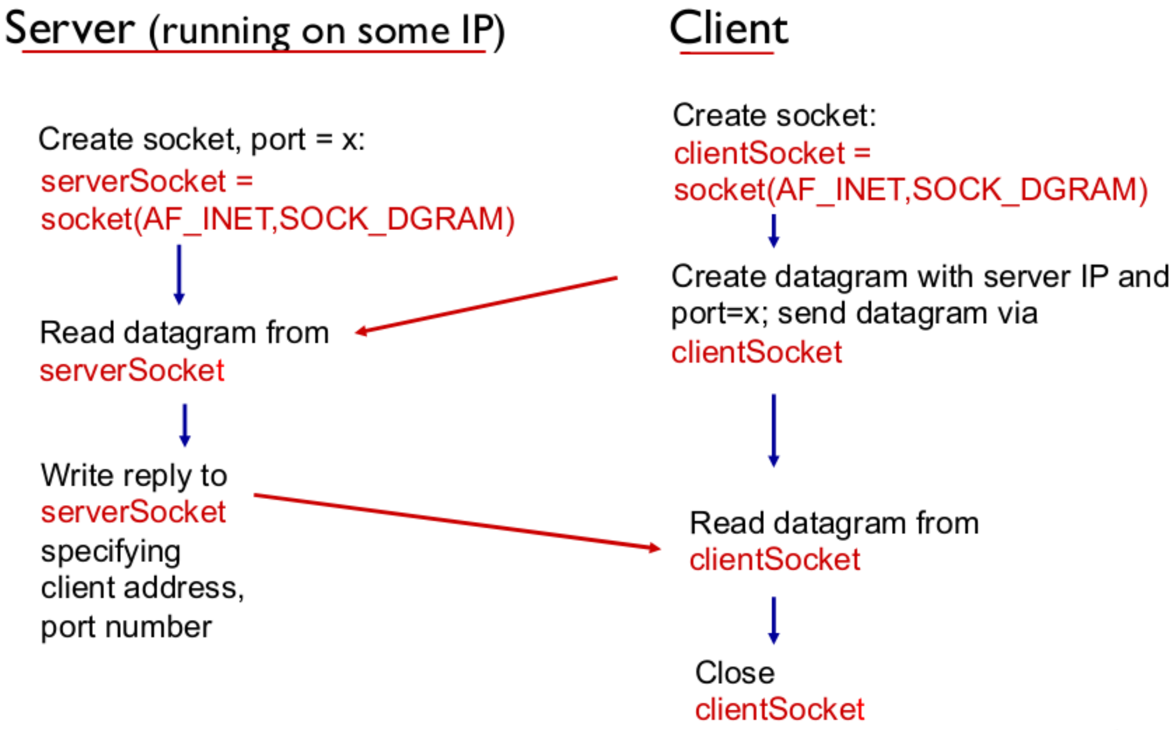
Application example:

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 the screen

11.1 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

11.1.1 Client/server socket interaction: UDP



Server (running on some IP)

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

11.1.2 Example app: UDP client

```
# include python's socket library
from socket import *
# If server IP is empty then local system
serverName =''
# choose an unreserved port
serverPort=12000
# create UDP socket for server
clientSocket=socket(AF_INET, SOCK_DGRAM)
# get user keyboard input
message=input('Input lowercase sentence: ')
# Attach server name, port to message; send into socket
clientSocket.sendto(message.encode(),(serverName,ServerPort))
# Read reply characters from socket into string
modifiedMessage, serverAddress = clientSocket.recvfrom(2048)
# Print out recived string and close socket
print(modifiedMessage.decode())
clientSocket.close()
```

11.1.3 Example app: UDP server

```
from socket import *
serverPort = 12000
# Create UDP socket
serverSocket = socket(AF_INET, SOCK_DGRAM)
# Bind socket to local port number 12000
serverSocket.bind(('', serverPort))
print('The serve is ready to receive')
# while True
        # Read from UDP socket into message, getting client's address (client IP and port)
        message, clientAddress = serverSocket.recvfrom(2048)
        # Send upper case string back to this client
        modifiedMessage = message.decode().upper()
        serverSocket.sendto(modifiedMessage.encode)
```

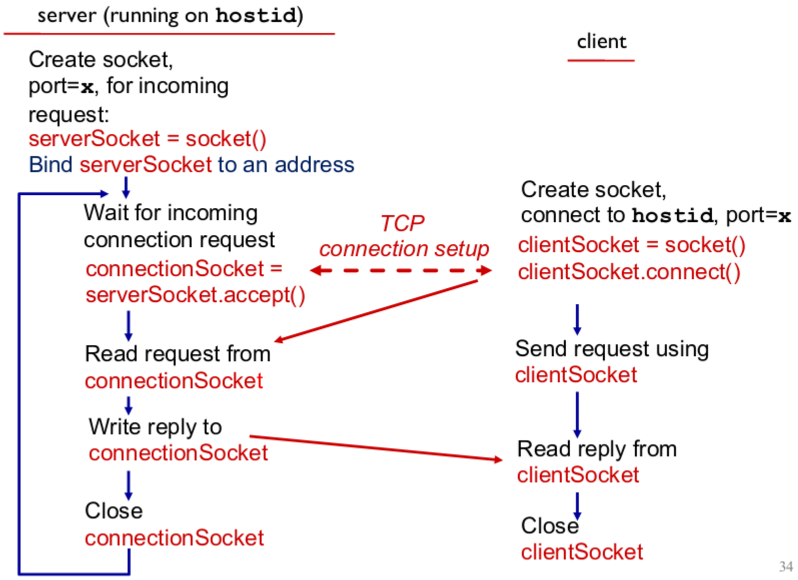
12 Socket Programming with TCP

- Client must contact server
- Server process must first be running
- Server must have created socket that welcomes client's contact
- Client contacts server by
  - Creating TCP socket, specifying IP address, port number of server process
- When client establishes socket: client TCP establishes connection to server TCP
- When contacted by the client:
  - Server TCP create 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

12.1 Client/server socket interaction: TCP



12.2 Example app: TCP Client

```
from socket import *
serverName=''
serverPort=12000
clientSocket=socket(AF_INET,SOCK_STREAM)
clientSocket.connect((serverName, serverPort))
sentence=input('Input lowercase sentence: ')
clientSocket.send(sentence.encode())
modifiedSentence=clientSocket.recv(1024)
print('From Server: ', modifiedSentence.decode())
clientSocket.close()
```

12.3 Example app: TCP server

```
from socket import *
serverPort=12000
# Create TCP welcoming socket
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(('',serverPort))
# Server begins listening for incoming TCP requests
serverSocket.listen(1)
print('The server is ready to revieve')
while True:
        # Server waits on accept() for incoming requests, new socket created on return
        connectionSocket, addr=serverSocket.accept()
        # Read bytes from socket (not address as in UDP)
        sentence=connectionSocket.recv(1024).decode()
        capsSentence=sentence.upper()
        connectionSocket.send(capsSentence)
        # Close connection to this client (but not welcoming socket)
        connectionSocket.close()
```