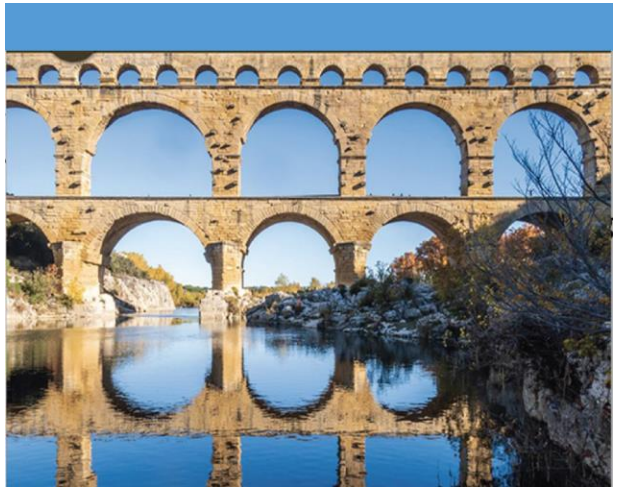
Application Layer: Overview (7 of 7)

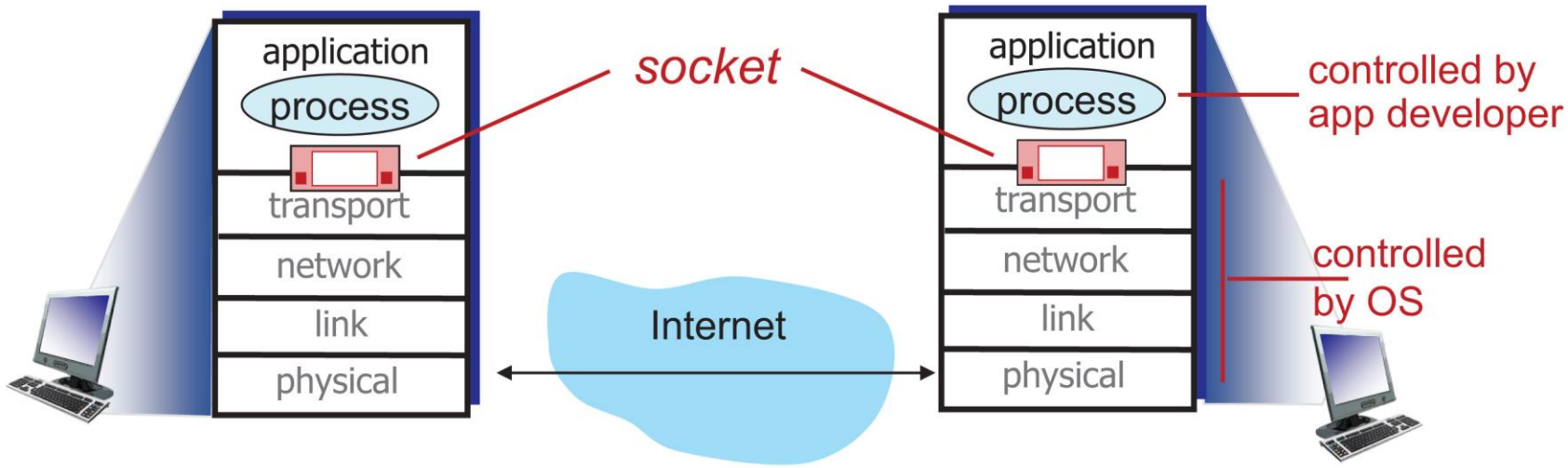
- Principles of network applications
- Web and HTTP
- E-mail, SMTP, IMAP
- The Domain Name System DNS
- socket programming with UDP and TCP



Socket Programming (1 of 2)

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

socket: door between application process and end-end-transport protocol



Socket Programming (2 of 2)

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 its screen

Socket Programming with UDP

UDP: no "connection" between client and 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 processes

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 serverIP address
And port=x; send datagram via
clientSocket

read datagram from
clientSocket

close
clientSocket

Example App: UDP Client

Python UDP Client

include Python's socket library →

```
from socket import *
serverName = 'hostname'
serverPort = 12000
```

create UDP socket →

```
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 data (bytes) from socket →

```
modifiedMessage, serverAddress = clientSocket.recvfrom(2048)
```

print out received string and close socket →

```
print(modifiedMessage.decode())
clientSocket.close()
```

Note: this code update (2023) to Python 3

Example App: UDP Server

Python UDP Server

create UDP socket →

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET, SOCK_DGRAM)
```

bind socket to local port number 12000 →

```
serverSocket.bind(('', serverPort)) print('The server is ready to receive')
```

loop forever →

```
while True:
```

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

```
message, clientAddress = serverSocket.recvfrom(2048)
modifiedMessage = message.decode().upper()
```

send upper case string back to this client →

```
serverSocket.sendto(modifiedMessage.encode(), clientAddress)
```

Note: this code update (2023) to Python 3

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
  - client source port # and IP address used to distinguish clients (more in Chap 3)

Application viewpoint

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

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

Example App: TCP Client

Python TCP Client

create TCP socket for server, remote port 12000 →

```
from socket import *
serverName = 'servername'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName,serverPort))
sentence = input('Input lowercase sentence:')
clientSocket.send(sentence.encode())
```

No need to attach server name, port →

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

Note: this code update (2023) to Python 3

Example App: TCP Server

Python TCP Server

create TCP welcoming socket →

```
from socket import *
serverPort = 12000
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 receive')
```

loop forever →

```
while True:
```

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

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

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

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

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

```
connectionSocket.close()
```

Note: this code update (2023) to Python 3

Chapter 2: Summary (1 of 2)

our study of network application layer is now complete!

- application architectures
  - client-server
  - P2P
- application service requirements:
  - reliability, bandwidth, delay
- Internet transport service model
  - connection-oriented, reliable: TCP
  - unreliable, datagrams: UDP
- specific protocols:
  - HTTP
  - SMTP, IMAP
  - DNS
- socket programming:

  TCP, UDP sockets

Chapter 2: Summary (2 of 2)

Most importantly: learned about **protocols!**

Important themes:

- typical client/server request/reply message exchange
- centralized vs decentralized
- stateless vs stateful
- scalability
- reliable vs unreliable message transfer
- "complexity at network edge"