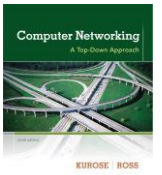
Chapter 2
Application Layer



Computer Networking: A Top Down Approach
6th edition
Jim Kurose, Keith Ross
Addison-Wesley
March 2012

A note on the use of these ppt slides:
We're making these slides freely available to all (faculty, students, readers). They're in PowerPoint form so you see the animations; and can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a *lot* of work on our part. In return for use, we only ask the following:
- If you use these slides (e.g., in a class) that you mention their source (after all, we'd like people to use our book!)
- If you post any slides on a www site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

Thanks and enjoy! JFK/KWR

The course notes are adapted for Bucknell's CSCI 363
Xiannong Meng
Spring 2016

Chapter 2: outline

2.1 principles of network applications
- app architectures
- app requirements

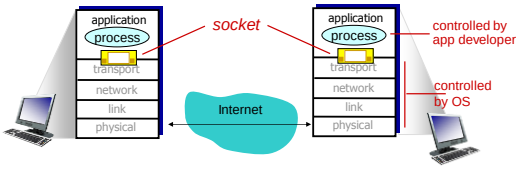
2.2 Web and HTTP
2.3 FTP
2.4 electronic mail
- SMTP, POP3, IMAP

2.5 DNS
2.6 P2P applications
2.7 socket programming with UDP and TCP

Socket programming

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

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



Socket programming

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 the data to the server.
2. The server receives the data and converts characters to uppercase.
3. The server sends the modified data to the client.
4. The client receives the modified data and displays the line on its screen.

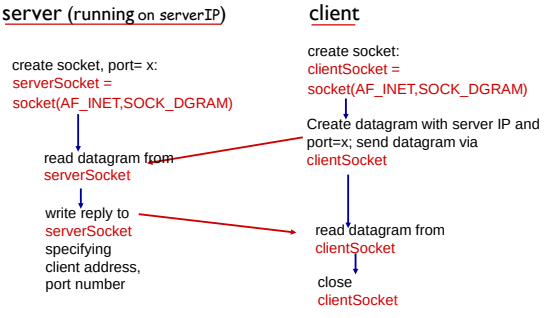
Socket programming *with UDP*

UDP: no "connection" between client & server
- no handshaking before sending data
- sender explicitly attaches IP destination address and port # to each packet
- rcvr 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



Example app: UDP client

Python UDPClient

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

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

Example app: UDP server

Python UDPServer

```
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)
```

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

UDP server and client in C

- Exact the same functionality can be implemented in C.
- Since the C language is a more primitive, more complicated code is needed.
- See examples/socket.
- Basic flow:
  - Create a socket
  - Associate the server host information with a socket address
  - Bind the client socket to a local address
  - Send the message to the server (read from client)
  - Read the echoed message from the server (send back to the client)

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

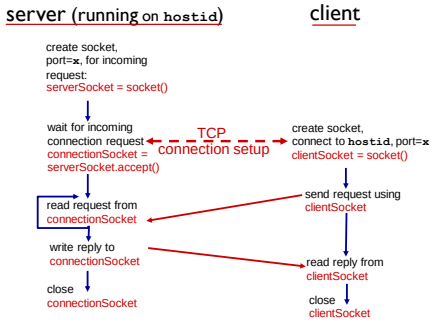
client contacts server by:
- Creating TCP socket, specifying IP address, port number of server process
- *when client creates socket:* client TCP establishes connection to server TCP

- 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 (more in Chap 3)

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

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

create TCP socket for server, remote port 12000
No need to attach server name, port

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)

TCP server and client in C

- Exact the same functionality can be implemented in C.
- Since the C language is a more primitive, more complicated code is needed.
- See examples/socket.
- Basic flow:
  - Create a socket
  - Associate the server host information with a socket address
  - Bind the client socket to a local address
  - Send the message to the server (read from client)
  - Read the echoed message from the server (send back to the client)