Chapter II: Application Layer

UG3 Computer Communications & Networks (COMN)

Myungjin Lee
myungjin.lee@ed.ac.uk

Slides copyright of Kurose and Ross
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

- aside: applications with P2P architectures have client processes & server processes
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
Addressing processes

• to receive messages, process must have identifier
• host device has unique 32-bit IP address
• Q: does IP address of host on which process runs suffice for identifying the process?
  ▪ A: no, many processes can be running on same host

• identifier includes both IP address and port numbers associated with process on host.
• example port numbers:
  – HTTP server: 80
  – mail server: 25
• to send HTTP message to www.inf.ed.ac.uk web server:
  – IP address: 129.215.33.176
  – port number: 80
• more shortly…
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

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

close
clientSocket

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
Example app: UDP client

Python UDPClent

```
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

code:
```
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)
```

Explanation:
- **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**
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
Illustration of TCP socket in client/server
Client/server socket interaction: TCP

server (running on hostid)

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

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

- read request from connectionSocket

- write reply to connectionSocket

- close connectionSocket

TCP connection setup

client

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

- send request using clientSocket

- read reply from clientSocket

- close clientSocket
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()
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
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

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, ...

timing

• some apps (e.g., Internet telephony, interactive games) require low delay to be “effective”
## Transport service requirements: common apps

<table>
<thead>
<tr>
<th>application</th>
<th>data loss</th>
<th>throughput</th>
<th>time sensitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>file transfer</td>
<td>no loss</td>
<td>elastic</td>
<td>no</td>
</tr>
<tr>
<td>e-mail</td>
<td>no loss</td>
<td>elastic</td>
<td>no</td>
</tr>
<tr>
<td>Web documents</td>
<td>no loss</td>
<td>elastic</td>
<td>no</td>
</tr>
<tr>
<td>real-time audio/video</td>
<td>loss-tolerant</td>
<td>audio: 5kbps-1Mbps</td>
<td>yes, 100’s ms</td>
</tr>
<tr>
<td></td>
<td></td>
<td>video:10kbps-5Mbps</td>
<td>msec</td>
</tr>
<tr>
<td>stored audio/video</td>
<td>loss-tolerant</td>
<td>same as above</td>
<td></td>
</tr>
<tr>
<td>interactive games</td>
<td>loss-tolerant</td>
<td>few kbps up</td>
<td>yes, few secs</td>
</tr>
<tr>
<td>text messaging</td>
<td>no loss</td>
<td>elastic</td>
<td>yes, 100’s ms</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>yes and no</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>application</th>
<th>application layer protocol</th>
<th>underlying transport protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>e-mail</td>
<td>SMTP [RFC 2821]</td>
<td>TCP</td>
</tr>
<tr>
<td>remote terminal access</td>
<td>Telnet [RFC 854]</td>
<td>TCP</td>
</tr>
<tr>
<td>Web</td>
<td>HTTP [RFC 2616]</td>
<td>TCP</td>
</tr>
<tr>
<td>file transfer</td>
<td>FTP [RFC 959]</td>
<td>TCP</td>
</tr>
<tr>
<td>streaming multimedia</td>
<td>HTTP (e.g., YouTube), RTP [RFC 1889]</td>
<td>TCP or UDP</td>
</tr>
<tr>
<td>Internet telephony</td>
<td>SIP, RTP, proprietary (e.g., Skype)</td>
<td>TCP or UDP</td>
</tr>
</tbody>
</table>
Web and HTTP

First, a review…

• web page consists of objects
• object can be HTML file, JPEG image, Java applet, audio file,…
• web page consists of base HTML-file which includes several referenced objects
• each object is addressable by a URL, e.g.,
  
  www.someschool.edu/someDept/pic.gif

  host name       path name
HTTP overview

HTTP: hypertext transfer protocol

- Web’s application layer protocol
- 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
HTTP overview (continued)

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
HTTP connections

**non-persistent HTTP**
- at most one object sent over TCP connection
  - connection then closed
- downloading multiple objects required multiple connections

**persistent HTTP**
- multiple objects can be sent over single TCP connection between client, server
Non-persistent HTTP

suppose user enters URL: www.someSchool.edu/someDepartment/home.index

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

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

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

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

(time)
4. HTTP server closes TCP connection.

5. HTTP client receives response message containing html file, displays html. Parsing html file, finds 10 referenced jpeg objects

6. Steps 1-5 repeated for each of 10 jpeg objects
Non-persistent HTTP: response time

RTT (definition): time 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
Persistent HTTP

non-persistent HTTP issues:
• requires 2 RTTs per object
• OS overhead for each TCP connection
• browsers often open parallel TCP connections to fetch referenced objects

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
• as little as one RTT for all the referenced objects