

The Application Layer: Sockets Recap & Introduction to the Transport Layer

Smith College, CSC 249
February 21, 2008

slides mostly from J.F Kurose and K.W. Ross,
copyright 1996-2007

Overview

- The Socket API
 - ❖ Defined for UNIX
 - ❖ Used by most operating systems
- A simple Java Web server
- Python sockets examples

2

Socket API Overview

- In UNIX, all devices are file abstractions
 - ❖ Open, close, read, write
- Sockets are simply one more file abstraction
- Sockets are useful for sending data from one host to another
- Most operating systems use "The Socket API" as defined in/for UNIX

3

Socket API Overview

- Socket Programming Procedures
 - ❖ Socket()
 - ❖ Bind()
 - ❖ Listen()
 - ❖ Accept()
 - ❖ Connect()
 - ❖ Along with send and receive procedures
 - ❖ Close()
- And for DNS...
 - ❖ getHostByName
 - ❖ getServByName
 - ❖ getProtoByName

4

Sockets

- The API is used for communicating between a Client and a Server
- Client
 - ❖ Active participant in communication
 - ❖ Initiates conversations and sends data
- Server
 - ❖ Passively listens and waits for data
- Socket
 - ❖ Protocol to use?
 - ❖ Protocol address of the other machine
 - ❖ Client or server?

5

Connection-Oriented

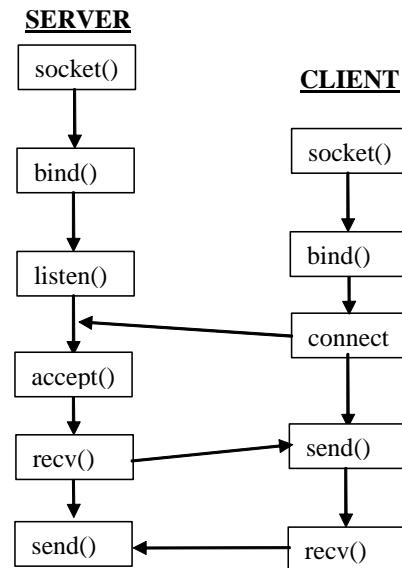
- The message is only transferred after a connection has been made
- Both parties know that a message will be communicated
- No need to tell destinations (IP address and port number) in subsequent messages
- → TCP

6

Connectionless

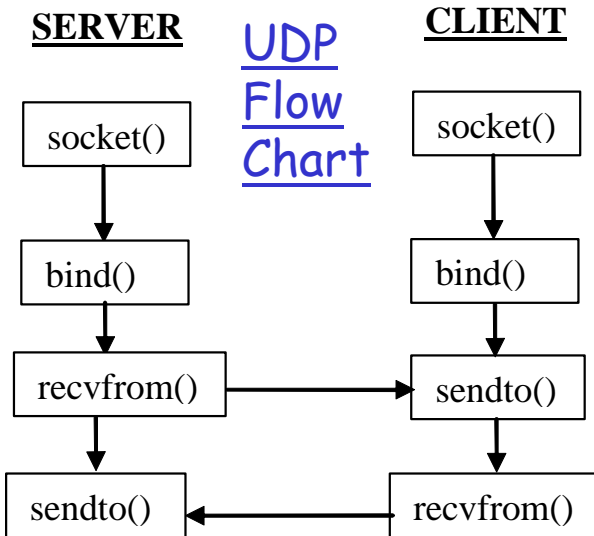
- Send Individual Messages
- Socket has to be told where to send the message every time
- Can get excessive if a large number of messages need to be sent between the same hosts
- → UDP

7



TCP
Flow
Chart

8



9

Procedures: Socket()

- descriptor = **socket**(protoFamily, type, protocol)
 - ❖ Creates a socket and returns an integer descriptor
 - ❖ ProtoFamily - refers to Family of protocols that this protocol belongs to, for TCP/IP use PF_INET
 - ❖ Type - SOCK_STREAM, SOCK_DGRAM
 - SOCK_STREAM - Connection Oriented
 - SOCK_DGRAM - Connectionless Message Transmission

10

Close()

- The socket is no longer going to be used
- **Close**(sock)
 - ❖ Sock - the descriptor
- Note: For a connection oriented socket, connection is terminated before socket is closed

11

Bind()

- **Bind**(socket, localAddr, addrLen)
 - ❖ Call after socket() has been called
 - ❖ Used to assign the port at which the client/server will be waiting for connections/messages
 - The port number is part of the addr struct
 - ❖ Socket - descriptor
 - ❖ localAddr - socket address structure → including the port number
 - ❖ addrLen - length of the address

12

Listen() - Server Procedure

- ❑ Listen(socket, queuesize)
 - ❖ Called at server
 - ❖ socket - descriptor at server
 - ❖ queueSize - buffering of requests

- ❑ This procedure tells the server to leave a socket running, in passive mode, at this port

13

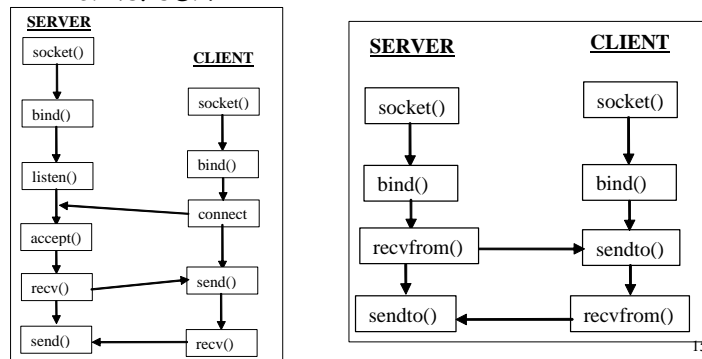
Server Recap so far...

- ❑ All servers begin by calling socket() to create a socket and bind() to specify a protocol port number
- ❑ UDP: the server is now ready to accept messages
- ❑ TCP: additional steps to become ready are
 - ❖ Server calls listen() to place the socket in passive mode
 - ❖ Server calls accept() to accept a connection request

14

Client Recap so far...

- ❑ What does the client do with and for its sockets? → Order of events?
 - ❖ TCP vs. UDP?



15

Accept() - Server Procedure

- ❑ Newsock = accept(socket, caddr, caddrlen)
 - ❖ Accept() fills the fields of the struct caddr with the address of the client that formed the connection
 - ❖ Accept() creates a new socket for this connection and returns the descriptor of this new socket
 - ❖ The server's original "listen()" socket remains unchanged
- ❑ A request has come to the server
 - ❖ → The phone is ringing
- ❑ Accept picks up the connections (only TCP)

16

Connect() - Client Procedure

- **Connect**(socket, saddr, saddrlen)
 - ❖ Arguments 'socket' is the descriptor of a socket on the client's computer to use for the connection
 - ❖ 'saddr' and len specify the server's info
 - ❖ With TCP, this initiates the connection to the specified server
- This is used to make the "phone call"
- Two uses
 - ❖ Connection-oriented transport - make the call
 - ❖ Possible use - Connectionless - identify the server to send the many, independent messages

17

Send() and Sendto()

- Used to send packets from one host to another
 - ❖ **Send**(socket, data, length, flags)
 - Socket - descriptor
 - Data - pointer to buffer in memory with the data
 - Length - of data to be send
 - Flags - for debugging, not general use (typ = 0)
- **Sendto**() is used with an unconnected socket
 - ❖ **Sendto** (socket, data, length, flags, destAddress, addressLen)

18

Recv() and Recvfrom()

- Used to receive messages in a connection oriented communication
 - ❖ **Recv**(socket, buffer, length, flags)
 - Buffer - memory location/structure to store the data
 - Length - the length of buffer
- **Recvfrom**() is used in connectionless communication
 - ❖ **Recvfrom**(socket, buffer, flags, sndraddr, saddrlen)
 - Sndraddr - sender's address
 - Saddrlen - length of sender's address

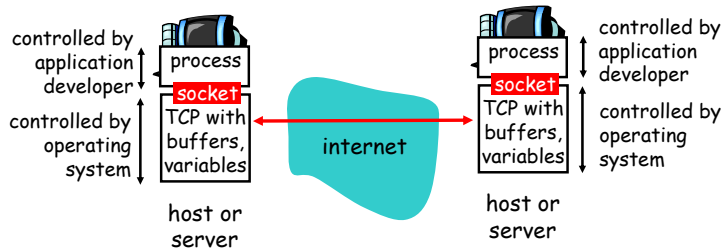
19

Chapter 2: Application layer

- 2.1 Principles of network applications
- 2.2 Web and HTTP
- 2.3 FTP
- 2.4 Electronic Mail
 - ❖ SMTP, POP3, IMAP
- 2.5 DNS
- 2.6 P2P file sharing
- 2.7 **Socket programming with TCP**
- 2.8 Socket programming with UDP

20

Socket programming



21

Socket programming *with TCP*

What is the order of steps for using sockets with TCP - clients and servers together...

- Server process must be running *first*
- Then the client can create a socket, which causes...
 - ❖ DNS lookup for server IP address
 - ❖ TCP to establish connection between the client and server
 - Which causes the server process to create a new, dedicated socket for this specific client process
- Client creates message - as a byte stream
- Client sends the message into its socket
- TCP takes over and delivers the message
 - ❖ Guarantees delivery
 - ❖ With bytes delivered in the original order
- Server process performs its application duties and sends a response message through its socket...

22

TCP vs. UDP

- Where and when are IP addresses and port numbers used in TCP vs. UDP sockets?

23

A Simple Java Web Server

24

A Simple Java Web Server

The Web server does the following:

- ❑ handles one HTTP request
- ❑ accepts the request
- ❑ parses the request header
- ❑ obtains requested file from server's file system
- ❑ creates HTTP response message:
 - ❖ header lines + file
- ❑ Sends the response to the client

On the program print out, identify:

- ❑ Application code segments
 - ❖ Protocol specific code segments
- ❑ Code associated with sockets
 - ❖ Function of each such line

25

A Simple Java Web Server

```
// The following code is different from that in TCPServer.java:
// Read the first line of the HTTP request message - this is
// expected to be of the form GET file_name HTTP/1.1
requestMessageLine = inFromClient.readLine();

// Parse this line (hoping to find "GET ...")
StringTokenizer tokenizedLine =
    new StringTokenizer(requestMessageLine);
if (tokenizedLine.nextToken().equals("GET")){
    fileName = tokenizedLine.nextToken();
    if (fileName.startsWith("/") == true )
        fileName = fileName.substring(1);
// Attach a stream "infile" to the file "filename"
File file = new File(fileName);
int numofBytes = (int) file.length();
FileInputStream inFile = new FileInputStream (fileName);
```

27

A Simple Java Web Server

```
// Create socket before server program receives request for
// TCP connection from client → listening at port number
// 6789, waiting for a request from any client
ServerSocket listenSocket = new ServerSocket(6789);

// Once the request arrives, .accept() creates a new
// object (socket) for the specific, requesting client
Socket connectionSocket = listenSocket.accept();

// Two input streams are created next:
BufferedReader inFromClient =
    new BufferedReader(new InputStreamReader(
        connectionSocket.getInputStream()));
DataOutputStream outToClient =
    new DataOutputStream(connectionSocket.getOutputStream());
```

26

A Simple Java Web Server

```
// Determine the size of the file and construct an array of bytes
// of that size. And then read from the stream "infile" to the byte
// array "fileInBytes". This conversion to bytes is required
// because the output stream "outToClient" must be a byte stream
byte[] fileInBytes = new byte[numofBytes];
inFile.read(fileInBytes);

// Construct the HTTP response message:
// Send the HTTP header lines into the DataOutputStream outToClient
outToClient.writeBytes("HTTP/1.0 200 Document Follows\r\n");
if (fileName.endsWith(".jpg"))
    outToClient.writeBytes("Content-Type: image/jpeg\r\n");
if (fileName.endsWith(".gif"))
    outToClient.writeBytes("Content-Type: image/gif\r\n");
outToClient.writeBytes("Content-Length: " + numofBytes + "\r\n");
outToClient.writeBytes("\r\n");
```

28

A Simple Java Web Server

```
// Send the requested file fileInBytes to the TCP send buffer. TCP
// concatenates the file fileInBytes to the header lines and sends
// it all off to the client
    outToClient.write(fileInBytes, 0, numOfBytes);

// After serving one request for one file " this is a simple Web server
// our WebServer closes the socket
    connectionSocket.close();
}
else System.out.println("Bad Request Message");
}
}
```

29

```
# client.py
from socket import *
# Set the socket parameters
host = "localhost"
port = 21567
buf = 1024
addr = (host,port)

# Create socket
UDPSock = socket(AF_INET,SOCK_DGRAM)
def _msg = "===Enter message to send to server===";
print "\n",def _msg

# Send messages
while (1):
    data = raw_input('>> ')
    if not data:
        break
    else:
        if(UDPSock.sendto(data,addr)):
            print "Sending message '",data,"' ....."

# Close socket
UDPSock.close()
```

31

Python Sockets Examples

- Step through the following examples of Python socket code on the print outs, and identify what is happening where

```
# server.py
from socket import *

# Set the socket parameters
host = "localhost"
port = 21567
buf = 1024
addr = (host,port)
# Create socket and bind to address
UDPSock = socket(AF_INET,SOCK_DGRAM)
UDPSock.bind(addr)

# Receive messages
while 1:
    data,addr = UDPSock.recvfrom(buf)
    if not data:
        print "Client has exited!"
        break
    else:
        print "\nReceived message '", data,"'"

# Close socket
UDPSock.close()
```

30

32

```

# Echo client program
# From the python.org library reference
import socket

HOST = 'EGR102-L.sciencead.smith.edu' # The
    remote host
PORT = 50007 # The same port as used by the
    server
s = socket.socket(socket.AF_INET,
    socket.SOCK_STREAM)
s.connect((HOST, PORT))
s.send('Hello, world')
data = s.recv(1024)
s.close()
print 'Received', repr(data)

```

33

```

# Echo server program
# From the python.org library reference
import socket

HOST = '' # Symbolic name meaning the local host
PORT = 50007 # Arbitrary non-privileged port
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))
s.listen(1)
conn, addr = s.accept()
print 'Connected by', addr
while 1:
    data = conn.recv(1024)
    if not data: break
    conn.send(data)
conn.close()

```

34

Chapter 2: Summary

Our study of network apps now complete

- Application architectures
 - ❖ client-server
 - ❖ P2P
 - ❖ hybrid
- application service requirements:
 - ❖ reliability, bandwidth, delay
- Internet transport service model
 - ❖ Connection-oriented, reliable: TCP
 - ❖ Connectionless - unreliable, datagrams: UDP
- specific protocols:
 - ❖ HTTP
 - ❖ FTP
 - ❖ SMTP, POP, IMAP
 - ❖ DNS
- socket programming

35

Chapter 2: Summary

Learned about *protocols*

- typical request/reply message exchange:
 - ❖ client requests info or service
 - ❖ server responds with data, status code
- Message formats:
 - ❖ Defined by protocols
 - ❖ headers: fields giving info about the following data from the application
 - ❖ data: info being communicated to/from the application client/server processes

Themes

- Control vs. data msgs
 - ❖ in-band, out-of-band
- centralized vs. decentralized architecture
- stateless vs. stateful
- reliable vs. unreliable msg transfer
 - ❖ = Connection-oriented vs. connectionless
- "complexity at network edge"
 - ❖ e.g, DNS

36

Onto Chapter 3...

Chapter 3: Transport Layer

Our goals:

- understand principles behind transport layer services:
 - ❖ multiplexing/demultiplexing
 - ❖ reliable data transfer
 - ❖ flow control
 - ❖ congestion control
- learn about transport layer protocols in the Internet:
 - ❖ UDP: connectionless transport
 - ❖ TCP: connection-oriented transport
 - ❖ TCP congestion control

37

38

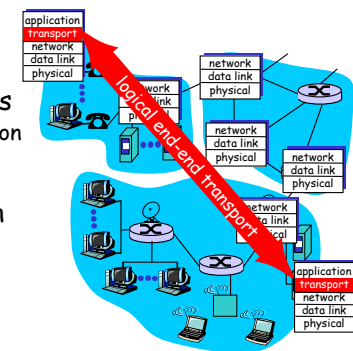
Chapter 3 outline

- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
 - ❖ segment structure
 - ❖ reliable data transfer
 - ❖ flow control
 - ❖ connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

39

Transport services and protocols

- provide *logical communication* between application processes running on different hosts
 - ❖ *Not* a physical communication or path that specifies routers, for example
- transport protocols run in end systems
 - ❖ sending side: breaks application messages into **segments**, passes them to network layer
 - ❖ receiving side: reassembles segments into messages, passes to application layer



40

Transport vs. network layer

- ❑ *transport layer*: logical communication between processes
 - ❖ relies upon network layer services
- ❑ *network layer*: logical communication between hosts

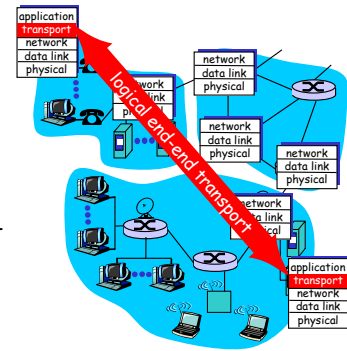
Household analogy:
12 kids sending letters to 12 kids

- ❑ processes = kids
- ❑ application messages = letters in envelopes
- ❑ hosts = houses
- ❑ network-layer protocol = postal service → delivery to the house
- ❑ transport protocol = delivery to each child

41

Internet transport-layer protocols

- ❑ reliable, in-order delivery (TCP)
 - ❖ congestion control
 - ❖ flow control
 - ❖ connection setup
- ❑ unreliable, unordered delivery: UDP
 - ❖ no-frills extension of "best-effort" IP
- ❑ services not available:
 - ❖ delay guarantees
 - ❖ bandwidth guarantees



42

Chapter 3 outline

- ❑ 3.1 Transport-layer services
- ❑ **3.2 Multiplexing and demultiplexing**
- ❑ 3.3 Connectionless transport: UDP
- ❑ 3.4 Principles of reliable data transfer
- ❑ 3.5 Connection-oriented transport: TCP
 - ❖ segment structure
 - ❖ reliable data transfer
 - ❖ flow control
 - ❖ connection management
- ❑ 3.6 Principles of congestion control
- ❑ 3.7 TCP congestion control

43

Multiplexing/demultiplexing

- ❑ **Multiplexer**
 - ❖ Selects input from one of many input lines and directs the information to a single output line
 - ❖ Many sockets to one network connection
- ❑ **Demultiplexer**
 - ❖ Direct a single input to one of many possible output lines
 - ❖ Single network connection to many sockets (processes)

44

Multiplexing/demultiplexing

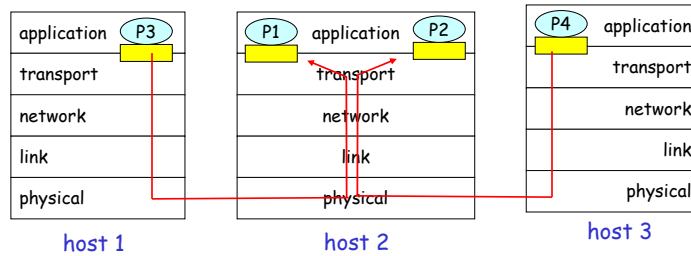
Demultiplexing at rcv host:

delivering received segments to correct socket

Multiplexing at send host:

gathering data from multiple sockets, enveloping data with header (later used for demultiplexing)

■ = socket ○ = process



45

Summary of Chapter 3 Intro...

- Overview of transport layer services
 - ❖ Comparison to network layer services
- Multiplexing and demultiplexing

46