CS419: Computer Networks

Lecture 2: Jan 31, 2004

Sockets Programming

Slides stolen from Rohan Murty / Hitesh Ballini slides from 2004 CS519
Their slides were adapted from Prof. Matthews’ slides from 2003
Goal: learn how to build client/server application that communicate using sockets

Socket API

- introduced in BSD4.1 UNIX, 1981
- Sockets are explicitly created, used, released by applications
- client/server paradigm
- two types of transport service via socket API:
  - unreliable datagram
  - reliable, byte stream-oriented
Sockets

**Socket**: a door between application process and end-end-transport protocol (UCP or TCP)
Languages and Platforms

Socket API is available for many languages on many platforms:

- C, Java, Perl, Python,…
- *nix, Windows,…

Socket Programs written in any language and running on any platform can communicate with each other!

Writing communicating programs in different languages is a good exercise
Decisions

• Before you go to write socket code, decide
  – Do you want a TCP-style reliable, full duplex, connection oriented channel? Or do you want a UDP-style, unreliable, message oriented channel?
  – Will the code you are writing be the client or the server?
    • Client: you assume that there is a process already running on another machines that you need to connect to.
    • Server: you will just start up and wait to be contacted
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 client-local 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 client
- Frees up incoming port
- allows server to talk with multiple clients

TCP provides reliable, in-order transfer of bytes (“pipe”) between client and server
Pseudo code TCP client

- Create socket, connectSocket
- Do an active connect specifying the IP address and port number of server
- Read and Write Data Into connectSocket to Communicate with server
- Close connectSocket
Pseudo code TCP server

- Create socket (serverSocket)
- Bind socket to a specific port where clients can contact you
- Register with the kernel your willingness to listen that on socket for client to contact you
- Loop
  - Accept new connection (connectSocket)
  - Read and Write Data Into connectSocket to Communicate with client
  - Close connectSocket
- End Loop
- Close serverSocket
Client/server socket interaction: TCP (Java)

Server (running on **hostid**)  

- create socket, port=x, for incoming request:  
  ```java
  welcomeSocket = ServerSocket()
  ```

- wait for incoming connection request  
  ```java
  connectionSocket = welcomeSocket.accept()
  ```

- read request from `connectionSocket`
- write reply to `connectionSocket`
- close `connectionSocket`

Client

- create socket, connect to **hostid**, port=x  
  ```java
  clientSocket = Socket()
  ```

- send request using `clientSocket`

- read reply from `clientSocket`
- close `clientSocket`
import java.io.*;
import java.net.*;
class TCPClient {
    public static void main(String argv[]) throws Exception {
        Socket clientSocket = new Socket("boo.cs.cornell.edu", 6789);
        DataOutputStream outToServer =
            new DataOutputStream(clientSocket.getOutputStream());
        BufferedReader inFromServer =
            new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
        outToServer.writeBytes(stuff_to_write);
        stuff_to_read = inFromServer.readLine();
        clientSocket.close();
    }
}
import java.io.*;
import java.net.*;

class TCPServer {
    public static void main(String argv[]) throws Exception {
        ServerSocket listen_socket = new ServerSocket(6789);
        while(true) {
            Socket client_socket = listen_socket.accept();
            Connection c = new Connection(client_socket);
        }
    }
}
class Connection extends Thread {
    while(true) {
        BufferedReader inFromClient =
            new BufferedReader(new
                             InputStreamReader(connectionSocket.getInputStream()));

        DataOutputStream  outToClient = new
                             DataOutputStream (connectionSocket.getOutputStream());

        inputString = inFromClient.readLine();
        .......
        outToClient.writeBytes(outputString);
    }
}
Socket programming with UDP

UDP: very different mindset than TCP

- no connection just independent messages sent
- no handshaking
- sender explicitly attaches IP address and port of destination
- server must extract IP address, port of sender from received datagram to know who to respond to

UDP: transmitted data may be received out of order, or lost

**application viewpoint**

*UDP provides unreliable transfer of groups of bytes ("datagrams") between client and server*
Pseudo code UDP server

- Create socket
- Bind socket to a specific port where clients can contact you
- Loop
  - (Receive UDP Message from client x)+
  - (Send UDP Reply to client x)*
- Close Socket
Pseudo code UDP client

• Create socket

• Loop
  (Send Message To Well-known port of server)+
  (Receive Message From Server)

• Close Socket
Example: Java client (UDP)

```java
import java.io.*;
import java.net.*;

class UDPClient {
    public static void main(String args[]) throws Exception {
        BufferedReader inFromUser =
            new BufferedReader(new InputStreamReader(System.in));
        DatagramSocket clientSocket = new DatagramSocket();
        InetAddress IPAddress = InetAddress.getByName("hostname");

        byte[] sendData = new byte[1024];
        byte[] receiveData = new byte[1024];

        String sentence = inFromUser.readLine();
        sendData = sentence.getBytes();

        // Create input stream
        // Create client socket
        // Translate hostname to IP address using DNS

        // Send data
        DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length, IPAddress, 7777);
        clientSocket.send(sendPacket);

        // Receive data
        DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);
        clientSocket.receive(receivePacket);
        String receivedSentence = new String(receiveData);
        System.out.println(receivedSentence);
    }
}
```
Example: Java client (UDP), cont.

```java
DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length, IPAddress, 9876);
clientSocket.send(sendPacket);

DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);
clientSocket.receive(receivePacket);
String modifiedSentence = new String(receivePacket.getData());
System.out.println("FROM SERVER:" + modifiedSentence);
clientSocket.close();
```
Example: Java server (UDP)

```java
import java.io.*;
import java.net.*;

class UDPServer {
    public static void main(String args[]) throws Exception {
        DatagramSocket serverSocket = new DatagramSocket(9876);
        byte[] receiveData = new byte[1024];
        byte[] sendData  = new byte[1024];

        while(true)
        {
            DatagramPacket receivePacket =
                new DatagramPacket(receiveData, receiveData.length);
            serverSocket.receive(receivePacket);
        }
    }
}
```
Example: Java server (UDP), cont

```
String sentence = new String(receivePacket.getData());
InetAddress IPAddress = receivePacket.getAddress();
int port = receivePacket.getPort();

String capitalizedSentence = sentence.toUpperCase();
sendData = capitalizedSentence.getBytes();
DatagramPacket sendPacket =
    new DatagramPacket(sendData, sendData.length, IPAddress, port);
serverSocket.send(sendPacket);
```
Client/server socket interaction: UDP

**Server** (running on **hostid**)

- create socket, \( \text{port=\text{x}} \), for incoming request:
  - serverSocket = DatagramSocket()
- read request from serverSocket
- write reply to serverSocket
  - write reply to serverSocket
  - specifying client host address, port number

**Client**

- create socket, \( \text{clientSocket} = \text{DatagramSocket()} \)
- Create, address (**hostid**, \( \text{port=\text{x}}, \) send datagram request using clientSocket
- read reply from clientSocket
- close clientSocket
C/C++ Sockets
Byte ordering

- Big Endian byte-order

The byte order for the TCP/IP protocol suite is big endian.
Byte-Order Transformation

Host byte order

16-bit

htons

16-bit

nths

32-bit

htonl

32-bit

ntohl

Network byte order

u_short  htons ( u_short  host_short );

u_short  ntohs ( u_short  network_short );

u_long   htonl ( u_long   host_long );

u_long   ntohl ( u_long   network_long );
Socket address structure

struct sockaddr_in
{
    u_char sin_len;
    u_short sin_family;
    u_short sin_port;
    struct in_addr sin_addr;
    char sin_zero [8];
};
Some Definitions

• Internet Address Structure

```c
struct in_addr {
    in_addr_t s_addr;
};
```

in_addr_t is defined as a long on linux machines, implying 32 bit addresses!
Address Transformation

```c
int inet_aton ( const char *strptr , struct in_addr *addrptr ) ;
char *inet_ntoa ( struct in_addr inaddr ) ;
```
Socket Types

Application program

Stream socket interface
TCP

Datagram socket interface
UDP

Raw socket interface

IP

Physical and data link layers
**Server**

1. Create transport endpoint for incoming connection request: `socket()`

2. Assign transport endpoint an address: `bind()`

3. Announce willing to accept connections: `listen()`

4. Block and Wait for incoming request: `accept()`

5. Wait for a packet to arrive: `read()`

6. Formulate reply (if any) and send: `write()`

7. Release transport endpoint: `close()`

**Client**

1. Create transport endpoint: `socket()`

2. Assign transport endpoint an address (optional): `bind()`

3. Determine address of server: `gethostbyname()`

4. Connect to server: `connect()`

4. Formulate message and send: `write()`

5. Wait for packet to arrive: `read()`

6. Release transport endpoint: `close()`
Connectionless Service (UDP)

**Server**

1. Create transport endpoint: `socket()`

2. Assign transport endpoint an address: `bind()`

3. Wait for a packet to arrive: `recvfrom()`

4. Formulate reply (if any) and send: `sendto()`

5. Release transport endpoint: `close()`

**Client**

1. Create transport endpoint: `socket()`

2. Assign transport endpoint an address (optional): `bind()`

3. Determine address of server: `gethostbyname()`

4. Formulate message and send: `sendto()`

5. Wait for packet to arrive: `recvfrom()`

6. Release transport endpoint: `close()`
Procedures That Implement The Socket API

Creating and Deleting Sockets

• `fd=socket(protofamily, type, protocol)`
  Creates a new socket. Returns a file descriptor (fd). Must specify:
  • the protocol family (e.g. TCP/IP)
  • the type of service (e.g. STREAM or DGRAM)
  • the protocol (e.g. TCP or UDP)

• `close(fd)`
  Deletes socket.
  For connected STREAM sockets, sends EOF to close connection.
Procedures That Implement The Socket API
Putting Servers “on the Air”

• **bind**(fd,laddress,laddresslen)
  Used by server to establish port to listen on. When server has >1 IP addr, can specify “IF_ANY”, or a specific one

• **listen** (fd, queuesize)
  Used by connection-oriented servers only, to put server “on the air”
  Queuesize parameter: how many pending connections can be waiting
• \( \text{afd} = \text{accept} \ (\text{lfd}, \text{caddress}, \text{caddreslen}) \)
  Used by connection-oriented servers to accept one new connection

  • There must already be a listening socket (lfd)
  • Returns afd, a new socket for the new connection, and
  • The address of the caller (e.g. for security, log keeping, etc.)
Procedures That Implement The Socket API

How Clients Communicate with Servers?

• `connect (fd, saddress, saddreslen)`
  Used by connection-oriented clients to connect to server
  • There must already be a socket bound to a connection-oriented service on the fd
  • There must already be a listening socket on the server
  • You pass in the address (IP address, and port number) of the server.

Used by connectionless clients to specify a “default send to address”
  • Subsequent “sends” don’t have to specify a destination address
Procedures That Implement The Socket API

How Clients Communicate with Servers? (TCP)

• int `write` (fd, data, length)
  Used to send data
  • `write` is the “normal” write function; can be used with both files and sockets

• int `read` (fd, data, length)
  Used to receive data... parameters are similar!

NOTE: both functions can return a value less than the length
Procedures That Implement The Socket API

How Clients Communicate with Servers (UDP)

- **int sendto** (fd, data, length, flags, destaddress, addresslen)
  Used to send data.
  - Connectionless socket, so we need to specify the dest address

- **int recvfrom** (fd, data, length, flags, srcaddress, addresslen)
  Used to receive data… parameters are similar, but in reverse
Concurrent Server: TCP (C/C++)

Server (running on hostid)

- create socket, port=x, for incoming request:
  - socket(), bind(), listen()
- wait for incoming connection request
  - accept()
- read and process
  - read()
  - reply
  - write()
- close

TCP connection setup

Client

- create socket, connect to hostid, port=x
  - socket(), connect()
- send request
  - write()
- read reply from
  - read()
- close

- close()
Non-blocking I/O

- By default, `accept()`, `recv()`, etc block until there’s input
- What if you want to do something else while you’re waiting?
- We can set a socket to not block (i.e. if there’s no input an error will be returned)
- … or, we can tell the kernel to let us know when a socket is ready, and deal with it only then
non-blocking/select

• The host uses `select()` to get the kernel to tell it when the peer has sent a message that can be `recv()`’d

• Can specify multiple sockets on which to wait
  -- `select` returns when one or more sockets are ready
  -- operation can time out!
Java vs C

• Java hides more of the details
  – new ServerSocket of Java = socket, bind and listen of C
  – new Socket hides the getByName (or gethostbyname) of C; Unable to hide this in the UDP case though
  – Socket API first in C for BSD; more options and choices exposed by the interface than in Java?
PROJECT 1 : BASIC SOCKETS

AIM: Write a program (referred to as the **IP box**) that opens four sockets, two TCP and two UDP

2 TCP SOCKETS:

1. A **receive-config** socket: IP BOX acts as a Server (must be bound to a port you have to find, and the interface IP address)

2. A **send-config** socket: IP BOX acts as a receiver
• 2 UDP SOCKETS

• App -- acts as the interface between the IP layer and the application

• Iface – represents the network interface

• Both must be bound to an used port and the interface address
IP BOX OPERATION

- Send-config sockets connects to the Test Box and sends a "ready-to-test" command

- The Test Box then connects to recv-config socket and send a ‘\n’ terminated command which must be echoed

- The Test Box then sends UDP packets to app and iface sockets which must be echoed (Note: If the Test Box does not receive your echo, it retransmits the packet)
(cont …)

• On receiving both the echoes, the Test Box sends a “send-stat” command to the send-config socket

• The IP box sends a “list-of-stats”

• The Test Box then sends an exit message (during final test, this will have a 40 character hex string representing a hashed timestamp, which your program must RECORD!)