ECE5650: Network Programming

Socket Programming with TCP
Socket Programming with UDP
Build Web Server

Socket programming

Goal: learn how to build client/server application that communicate using sockets

Socket API
- introduced in BSD4.1 UNIX, 1981, as a special type of file, representing one end of comm
- explicitly created, used, released by apps
- client/server paradigm
- two types of transport service via socket API:
  - unreliable datagram
  - reliable, byte stream-oriented

socket

a host-local, application-created, OS-controlled interface (a “door”) into which application process can both send and receive messages to/from another application process
**Operations on a socket:**

- **Client Socket**
  - connect to a remote machine
  - send and receive data
  - close a connection

- **Server Socket**
  - Bind to a port
  - listen for incoming data
  - accept connections from remote machines on the bound port
  - send and receive data
  - Close a connection

**Berkeley Sockets**

Connection-oriented communication pattern using sockets.
Socket-programming using TCP

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

TCP service: reliable transfer of bytes from one process to another

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
- allows server to talk with multiple clients
- source port numbers used to distinguish clients (more in Chap 3)

TCP provides reliable, in-order transfer of bytes ("pipe") between client and server
Stream jargon (in Java)

- A stream is a sequence of characters that flow into or out of a process.
- An input stream is attached to some input source for the process, e.g., keyboard or socket.
- An output stream is attached to an output source, e.g., monitor or socket.

High-level steps when programming TCP sockets:
1) Open a socket.
2) Open an input or output stream to the socket.
3) Read from and write to the stream.
4) Close the streams.
5) Close the socket.

High-level steps when programming UDP sockets (steps 2&4 used in TCP not needed):
1) Open a socket.
2) Read from and write to the socket.
3) Close the socket.

Socket programming with TCP

Example client-server app:
1) client reads line from standard input (inFromUser stream), sends to server via socket (outToServer stream)
2) server reads line from socket
3) server converts line to uppercase, sends back to client
4) client reads, prints modified line from socket (inFromServer stream)
Client/server socket interaction: TCP

Server (running on hostid)  Client

create socket, port=x, for incoming request: welcomeSocket = ServerSocket();

wait for incoming connection request connectionSocket = welcomeSocket.accept();

read request from connectionSocket
write reply to connectionSocket
close connectionSocket

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

send request using clientSocket
read reply from clientSocket
close clientSocket

Example: Java client (TCP)

```java
import java.io.*;
import java.net.*;
class TCPClient {
    public static void main(String argv[]) throws Exception {
        String sentence;
        String modifiedSentence;
        BufferedReader inFromUser =
            new BufferedReader(new InputStreamReader(System.in));
        Socket clientSocket = new Socket("hostname", 6789);
        DataOutputStream outToServer =
            new DataOutputStream(clientSocket.getOutputStream());
        BufferedReader inFromServer =
            new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
        String sentence = inFromUser.readLine();
        modifiedSentence = sentence.toUpperCase();
        outToServer.writeUTF(modifiedSentence);
        System.out.println(inFromServer.readLine());
    }
}
```
Example: Java client (TCP), cont.

```java
BufferedReader inFromServer =
    new BufferedReader(new
       InputStreamReader(clientSocket.getInputStream()));

sentence = inFromUser.readLine();
outToServer.writeBytes(sentence + "\n");
modifiedSentence = inFromServer.readLine();
System.out.println("FROM SERVER: " + modifiedSentence);
clientSocket.close();
```

Example: Java server (TCP)

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

class TCPServer {
    public static void main(String argv[]) throws Exception {
        String clientSentence;
        String capitalizedSentence;
        ServerSocket welcomeSocket = new ServerSocket(6789);
        while(true) {
            Socket connectionSocket = welcomeSocket.accept();
            BufferedReader inFromClient =
                new BufferedReader(new
                    InputStreamReader(connectionSocket.getInputStream()));
            }
    }
```
Example: Java server (TCP), cont

```java
DataOutputStream outToClient = new DataOutputStream(connectionSocket.getOutputStream());
clientSentence = inFromClient.readLine();
capitalizedSentence = clientSentence.toUpperCase() + '\n';
outToClient.writeBytes(capitalizedSentence);
```

End of while loop, loop back and wait for another client connection

Socket programming with UDP

UDP: no "connection" between client and server
- no handshaking
- sender explicitly attaches IP address and port of destination to each packet
- server must extract IP address, port of sender from received packet

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

UDP provides unreliable transfer of groups of bytes ("datagrams") between client and server
**Client/server socket interaction: UDP**

**Server (running on hostid)**
- create socket, port=x, for incoming request: `serverSocket = DatagramSocket()`
- read request from `serverSocket`
- write reply to `serverSocket` specifying client host address, port number

**Client**
- create socket, `clientSocket = DatagramSocket()`
- Create, address (hostid, port=x), send datagram request using `clientSocket`
- read reply from `clientSocket`
- close `clientSocket`

**Example: Java client (UDP)**

- Input: receives packet (TCP received "byte stream")
- Output: sends packet (TCP sent "byte stream")

Client process

Client UDP socket

Input: sends packet (TCP sent "byte stream")

Output: receives packet (TCP received "byte stream")

Keyboard

Monitor
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();
        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 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);

            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);
        }
    }
}
```

Example: Java server (UDP), cont

```java
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);
```
User-Defined Socket and ServerSocket

class SSLServerSocket extends ServerSocket {
   ...
   public Socket accept() throws IOException {
      SSLSocket s = new SSLSocket(…) {}
   }
   ...
}
class SSLSocket extends java.net.Socket {
   ...
   public SSLSocket(…) {
      super();
      ...
   }
}

Multithreaded Server:
Serving Multiple Clients Concurrently

Multithreaded Server:
Serving Multiple Clients Concurrently

Client 1 Process

Client 2 Process

Internet/Switch

Server Process

Multithreaded Server:
Serving Multiple Clients Concurrently

Multithreaded Server:
Serving Multiple Clients Concurrently

Multithreaded Server:
Serving Multiple Clients Concurrently

Multithreaded Server:
Serving Multiple Clients Concurrently

Multithreaded Server:
Serving Multiple Clients Concurrently
Multithreaded Process

- A thread of control is a sequence of instructions being executed within the context of a process. It has its own program counter and stack.
- A traditional process has a single thread of control.
- A MT process has two or more threads within the same context. They share the same set of open files, child processes, timers, etc.
- Each thread has its own id.

Java Thread Basics

- Create a thread by extending Thread

```java
public class MyEx1 {  
    public static void main() {  
        Foo t;  
        t = new Foo();  
        t.start();  
    }  
}  

class Foo extends Thread {  
    public void run() { … }  
}
```

- Create a thread by implementing Runnable interface

```java
public class MyEx2 {  
    public static void main() {  
        Thread t = new Thread(new Bar());  
        t.start();  
    }  
}  

class Bar implements Runnable {  
    public void run() { … }  
}
```
### Java Thread Basic

- **Thread** `thr = new Thread("Thread Name");`
- **Thread** `thr = new Thread(myRunnable, "Name");`

thr.stop(); // the calling thread will exit
thr.yield(); // the calling thread will yield control of CPU
thr.join(); // wait for a thread to exit

How to synchronize the execution of multiple threads?
→ leave for your own exploration

---

### A Tiny Multithreaded Web Server

// Support HTTP protocol: `GET /path/filename`
// java.TinyHttpd 1234

import java.net.*;
import java.io.*;
import java.util.*;
public class TinyHttpd {
    public static void main( String argv[] ) throws IOException {
        ServerSocket ss = new ServerSocket(Integer.parseInt( argv[0] ));
        while ( true ) {
            Socket sk=ss.accept();
            new TinyHttpdConnection(sk).start();
        }
    }
}
class TinyHttpdConnection extends Thread {
    Socket client;
    TinyHttpdConnection (Socket client) throws SocketException {
        this.client = client;
        setPriority( NORM_PRIORITY -1 );
    }
    public void run() {
        try {
            BufferedReader in = new BufferedReader(
                new InputStreamReader( 
                    client.getInputStream(), “8859_1” ));
            OutputStream out = client.getOutputStream();
            PrintWriter pout = new PrintWriter( 
                new OutputStreamWriter(out, “8859_1”), true );
            String request = in.readLine();
            System.out.println( “Request “+ request );
            StringTokenizer st = new StringTokenizer( request );
            if ( (st.countToken() >=2 ) && st.nextToken().equals(“Get”) ) {
                if ( (request = st.nextToken()).startsWith(“/”) )
                    request = request.substring( 1 );
                if ( request.endsWith(“/”)) || request.equals(“”))
                    request = request + “index.html”;
                try {
                    FileInputStream fis = new FileInputStream( request );
                    byte [] data = new byte[ fis.available() ];
                    fis.read( data );
                    out.write( data );
                    out.flush();
                } catch (FileNotFoundException e) { … }
                else { … }
                client.close();
            } catch ( IOException e) { … }
        } catch ( IOException e) { … }
    }
}
Secured web server

// HTTP protocol: GET /path/filename [options]
// java.TinyHttpd

public class TinyHttpd {
    public static void main(String argv[]) throws IOException {
        System.setSecurityManager(
            new TinyHttpdSecurityManager());
        ServerSocket ss = new ServerSocket(
            Integer.parseInt(argv[0]));
        while (true)
            new TinyHttpdConnection(ss.accept()).start();
    }
}

Summary

- Definition of a Socket:
  - (IP + port) makes a means for programs to network
  - controlled mainly by OS
- TCP sockets: 5 Steps needed to read/write from/to:
  - open socket, define input/output stream, read/write to streams, close streams, close socket
- UDP sockets: 3 Steps needed to read/write from/to:
  - open socket, read/write to sockets, close socket
- TCP Socket:
  - TCP Server must be up before client establishes a client socket because handshaking is required.
  - IP address and port of server needed for client to establish a socket.
  - Client Socket port is determined by OS.
- UDP Socket
  - UDP Server may not be up before client establishes a client socket because handshaking is not required.
  - Programmer can specify a port for client socket otherwise it is determined by the OS.