Lecture 3

Socket Programming

Outline

Part 1

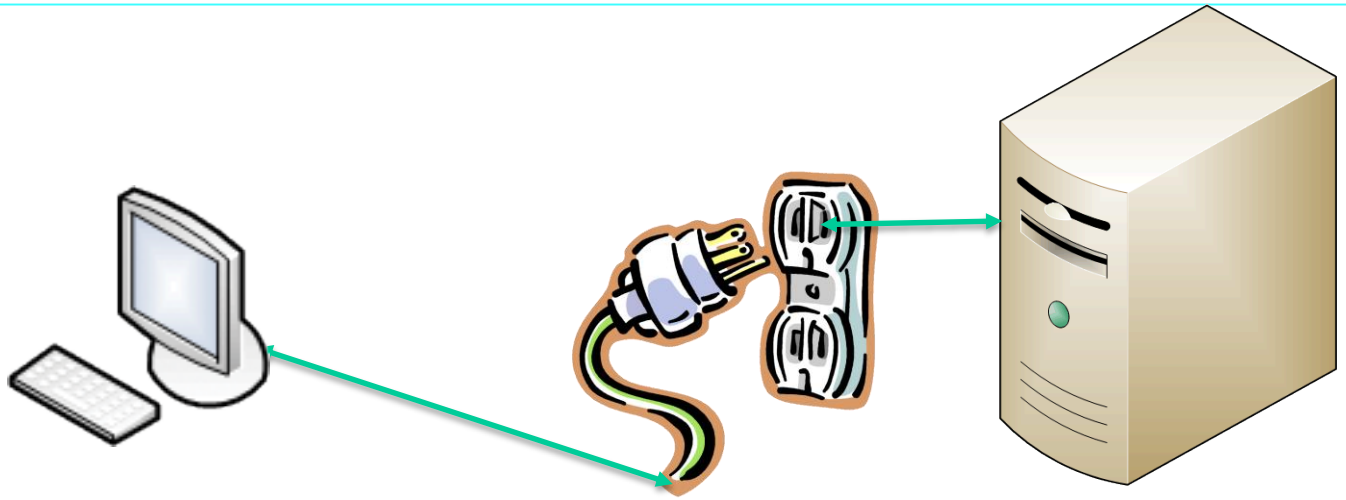
- Socket programming
- TCP socket programming: Socket() function
- UDP socket programming: DatagramSocket() function

Motivation of Socket Programming

- When facilities for Inter Process Communication (IPC) and networking were added to Unix, the idea was to make the interface to IPC similar to that of file I/O
  - In Unix, a process has a set of I/O descriptors that one reads from and writes to
  - These descriptors may refer to files, devices, or communication channels (sockets)
  - The lifetime of a descriptor is made up of three phases:
    - Creation (open socket)
    - Reading and writing (receive and send to socket)
    - Destruction (close socket).
- Berkeley sockets application programming interface (API) also known as the BSD (Berkeley Software Distribution) socket API
- Directly control TCP or UDP packets
  - UDP: a size limit of 64 KBytes on datagrams
  - TCP: no limit.
- Many programming languages support socket programming: Java, C++, etc.
  - Use Java in examples
- Used for developing new Internet protocols, and new network technologies
- Not recommended for web-based applications
  - Scripts, such as JavaScipt, ASP, and PHP, are better and faster for quick development

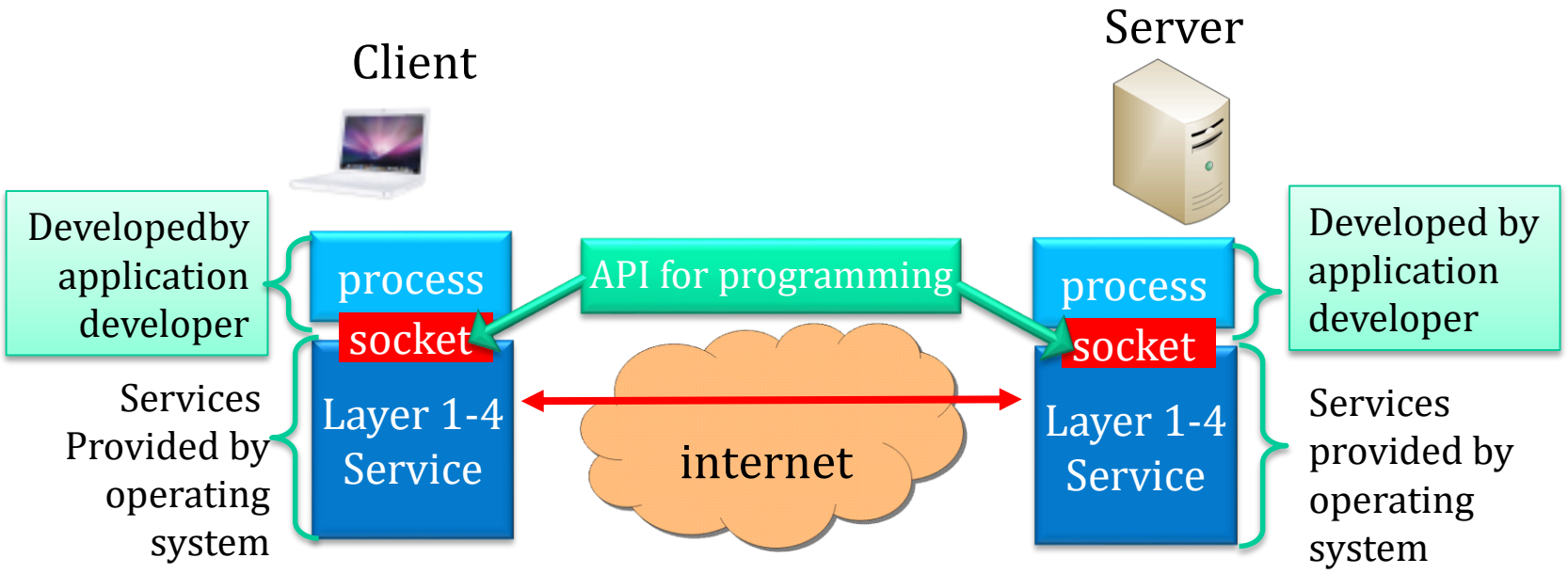
Socket

- A server has a socket that is bound to a specific port number, such as port 80 for http service, and waits for a connection request from a client.
- A client makes a connection request based on the host name of the server and the port number.
- Upon acceptance, the server gets a new socket bound to the port.
- If the connection is accepted, a socket is successfully created and the client can use the socket to communicate with the server.



Socket Programming

- A socket provides an API between an application process and end-to-end transport protocol (UDP or TCP) plus the Layer 1-3 services provided by the operating system
- A programmer can simply use the API to obtain Layer 1-4 services from the OS and quickly develop an application code



Outline

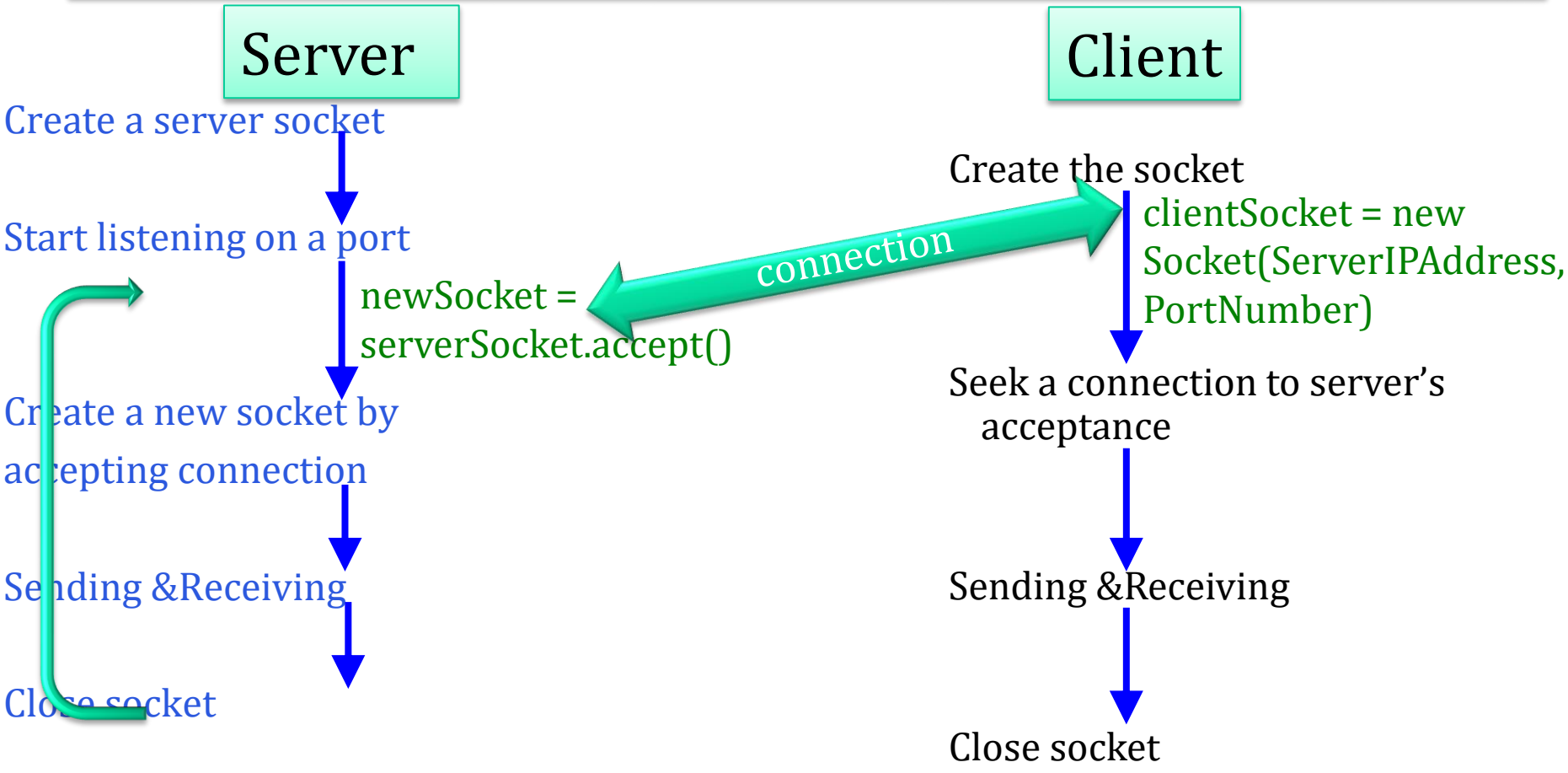
Part 1

- Socket programming
- TCP socket programming: Socket() function
- UDP socket programming: DatagramSocket() function
- IPv6 socket programming

TCP Socket

- TCP provides reliable transport
- A server socket process must first open a socket with a port number (e.g., port 80 for http) that is waiting for a connection request from a client
- Client must contact the server using the server's IP address with a waiting socket using the same port number (e.g., port 80 for http)
- When the client creates a socket, the client establishes a connection to the server TCP socket
  - A port number (>1024) is assigned to the client as the client's source port number
  - TCP is connection-oriented
- When contacted by multiple clients, the server can be programmed to use multiple threads to create new sockets to communicate with clients
  - Client source port numbers are used to distinguish clients

Client/server TCP socket interaction



4 tuples for a TCP socket

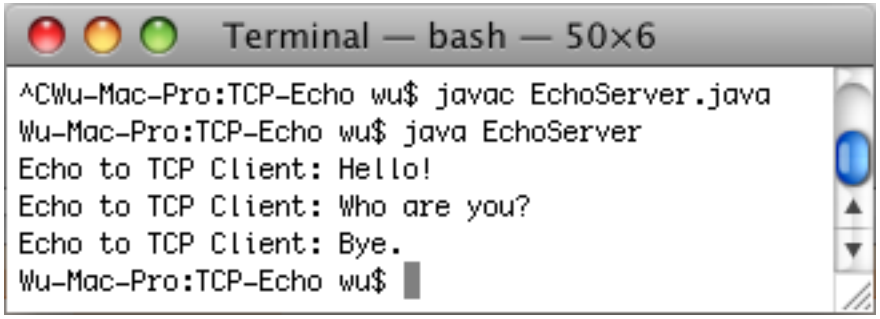
- A TCP socket contains:
  - Source IP address
  - Source port number
  - Destination IP address
  - Destination port number
- Since a TCP socket contains both client and server addressing information, a socket can be treated as an I/O device, such as a hard drive or printer

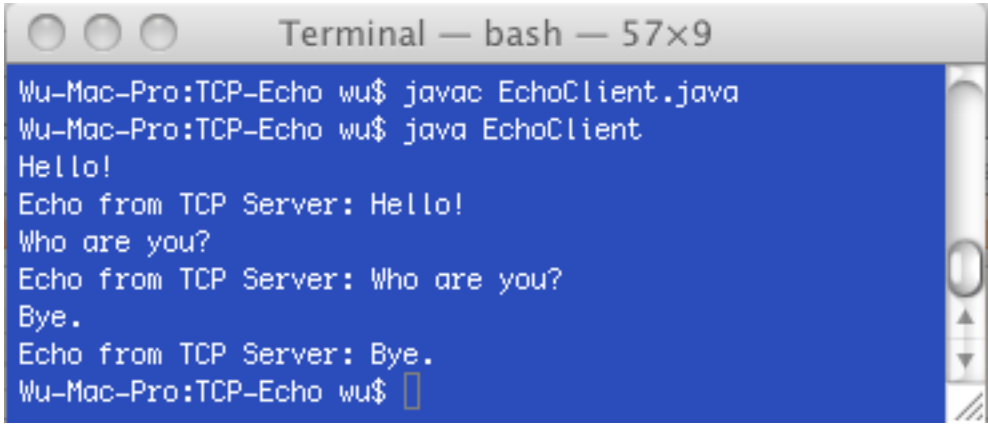
Example 1

- A TCP server echoes the client's message
- Run the TCP client/server codes
  - Two codes:
    - EchoClient.java
    - EchoServer.java
- The two codes are run in a single physical computer by using a loopback IP address 127.0.0.1
- The anatomy of both codes will be illustrated step-by-step
- Extension:
  - Replace the IP address 127.0.0.1 with your neighbor's IP address
  - Change the port number to 4000

Run codes

- Compile both codes
- Open two terminal (shell) windows and execute as following:
- Run EchoServer code first:
- Then client side: Use “Bye.” to close socket





TCP Socket Programming

- Client side
  - java.net.Socket
    - Specifies the remote server's hostname/IP address and port number
      - Socket(String host, int port)
      - Socket(InetAddress address, int port)
- Server side
  - java.net.ServerSocket
    - A server socket waits for and accepts connections from a client over the network.
      - ServerSocket(int port)
    - Accept a connection
      - ServerSocket.accept()

Open a client socket in Java

- Open a socket in a client:

```
Socket clientSocket;
clientSocket= new Socket(”Hostname", PortNumber);
```

- It is necessary to handle exceptions
- Socket clientSocket;

```
try {
          clientSocket= new Socket(”Hostname”,
PortNumber);
    }
catch (IOExceptione) {
    System.out.println(e);
    }
```

Open a server socket in Java

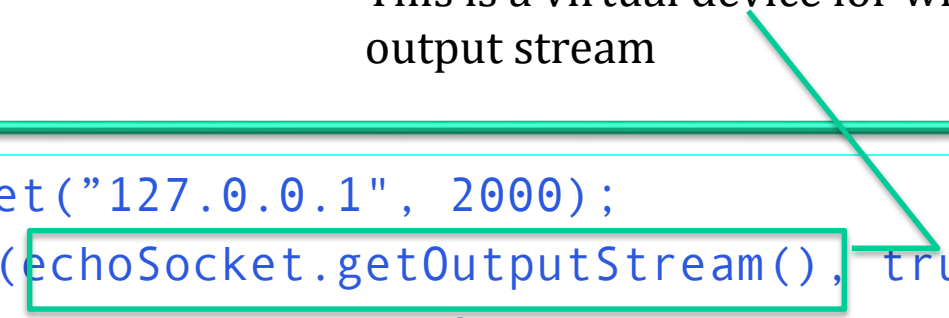
- Open a socket in a server:

```
ServerSocket serverSocket = null;
try {
     serverSocket = new ServerSocket(PortNumber);
      }
catch (IOException e) {
      System.out.println(e);
               }
Socket clientSocket = null;
try {
clientSocket = serverSocket.accept();
}
catch (IOException e) {
System.out.println("Accept failed: PortNumber" + e);
      System.exit(-1);
}
```

Java System.exit

- If running java program from a batch file/shell script, then a script can test the return code on the next line.
- Typically a program indicates a successful termination with a 0, and various kinds of abnormal terminations with a none-zero number
- For example,
  - System.exit(1) indicates that the command line arguments are invalid
  - System.exit(-1) indicates that the port number may be in use

Create Socket




```
echoSocket = new Socket("127.0.0.1", 2000);
out = new PrintWriter(echoSocket.getOutputStream(), true);
```

- The first statement in this sequence creates a new Socket object and names it echoSocket.
  - The Socket constructor used here requires the hostname/IP address and the port number of the server
  - The example code uses the localhost 127.0.0.1
  - It can be an IP address of another computer such as 131.204.1.5
- The second argument is the port number. Port number 2000 is the port on which the Echo server listens
- The second statement obtains the socket's output stream and opens a PrintWriter on it
- Although using System.out to write to the console is still permissible under Java, its use is recommended mostly for debugging purposes or for sample programs

Use PrintWriter to send a stream

- Recommended method of writing to the console when using Java is through a PrintWriter stream.
  - PrintWriter is one of the character-based classes
  - Using a character-based class for console output makes it easier to internationalize the java program.

```
PrintWriter(OutputStream out, boolean flushOnNewline)
```

  - outputStream is an object of type OutputStream
  - flushOnNewline controls whether Java flushes the output stream every time a newline ('\\n') character is output
    - If flushOnNewline is true, flushing automatically takes place
    - If false, flushing is not automatic
- PrintWriter supports the print( ) and println( )
- To write to the console by using a PrintWriter, specify System.out for the output stream and flush the stream after each newline.
- For example, this line of code creates a PrintWriter that is connected to the console output:

```
PrintWriterpwline = new PrintWriter(System.out, true);
```

Replace by a socket stream
`echoSocket.getOutputStream()`

Use InputStreamReader in efficient way

- An InputStreamReader converts from byte streams to character streams using a specified charset
- Wrap an InputStreamReader within a BufferedReader
  - Without buffering, each invocation of read() or readLine() could cause bytes to be read from the file/device, converted into characters, and then returned, which can be very inefficient
- The socket's input stream is passed to a BufferedReader
- For example:

```
in = new BufferedReader(newInputStreamReader
  (echoSocket.getInputStream()));
```

Send/Receive streams summary

- The example uses readers and writers so that Unicode characters can be used over the socket for International communication
- To send data through the socket to the server, EchoClient simply writes to the PrintWriter
- To obtain the server's response, EchoClient reads from the BufferedReader

A loop for user input and echo

- A loop reads a line at a time from the standard input stream (keyboard) and immediately sends it to the server by writing it to the PrintWriter connected to the socket:

```
    String userInput;
    while ((userInput = stdIn.readLine()) != null) {
out.println(userInput);
System.out.println("echo: " + in.readLine());
    }
```

- The last statement in the while loop reads a line of information from the BufferedReader connected to the socket
  - The readLine method waits until the server echoes the information back to EchoClient
  - When readline returns, EchoClient prints the information to the standard output

TCP Server code (1)

- The server program begins by creating a new ServerSocket object to listen on a specific port
- When writing a server code, choose a port that is not already dedicated to other services
- ServerSocket is a java.net class that provides a system-independent implementation of the server side of a client/server socket connection
- The constructor for ServerSocket produces an exception if it cannot listen on the specified port (for example, the port is already being used)
  - In this case, the Server must exit

TCP Server code

- If the server successfully binds to its port, then the ServerSocket object is successfully created, then the server continues to the next step, by waiting and accepting a connection from a client
- The accept method waits until a client requests a connection to the port of this server
- When a connection is successfully established, the server's accept method returns a new socket object that is bound to the same server port and designates its remote address and remote port for the client as part of the TCP socket (4 tuples)
- The server can communicate with the client over this new Socket
- "Bye." causes the server to close the socket, but the client will not close the socket.
- "Control D" will close both client and server sockets

Close socket

- The while loop continues until the user types an end-of-input character
  - the end-of-input character: Control D in Unix
- A well-behaved program always closes:
  - The readers and writers connected to the socket and to the standard input stream
  - The socket connection to the server
  - The order here is important: close any streams connected to a socket before you close the socket itself.

```
out.close();
in.close();
stdIn.close();
echoSocket.close();
```

TCP Client code (1)

Code name: EchoClient.java

```
import java.io.*;
import java.net.*;
public class EchoClient {
   public static void main(String[] args) throws IOException {
        try {
                Socket echoSocket = null;
                PrintWriter out = null;
                BufferedReader in = null;
                try {
                        echoSocket = new Socket("127.0.0.1", 2000);
                        out = new
   PrintWriter(echoSocket.getOutputStream(), true);
                        in = new BufferedReader(new
   InputStreamReader(echoSocket.getInputStream()));
                }
                catch (UnknownHostException e) {
                System.err.println("Do not know about host:
   127.0.0.1."+e);
                System.exit(1);
                }
```

TCP Client Code (2)

```
                    BufferedReader stdIn = new BufferedReader(new
    InputStreamReader(System.in));
                    String userInput;
                    while ((userInput = stdIn.readLine()) != null) {
                              out.println(userInput);
                              System.out.println("Echo from TCP Server: " + in.readLine());
                    if (userInput.equals("Bye."))
                              break;
                    }
                    out.close();
                    in.close();
                    stdIn.close();
                    echoSocket.close();
          }
          catch (IOException e) {
                    System.err.println("Could not get I/O for the connection to: localhost."
    + e);
                    System.exit(1);
          }
    }
}
```

TCP server code (1)

Code name: EchoServer.java

```
import java.net.*;
import java.io.*;
public class EchoServer {
  public static void main(String[] args) throws IOException {
   ServerSocket serverSocket = null;
   try {
        serverSocket = new ServerSocket(2000);
   }
   catch (IOException e) {
       System.out.println("Could not listen on port: 2000" + e);
       System.exit(-1);
   }
   Socket clientSocket = null;
   try {
        clientSocket = serverSocket.accept();
   }
   catch (IOException e) {
       System.out.println("Accept failed: 2000" + e);
       System.exit(-1);
   }
```

TCP server code (2)

```
    PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);
    BufferedReader in = new BufferedReader(new
    InputStreamReader(clientSocket.getInputStream()));
    String inputLine, outputLine;
    while ((inputLine = in.readLine()) != null) {
          outputLine = inputLine;
          out.println(outputLine);
          System.out.println("Echo to TCP Client: " + outputLine);
          if (outputLine.equals("Bye."))
          break;
    }
    out.close();
     in.close();
     clientSocket.close();
     serverSocket.close();
  }
}
```

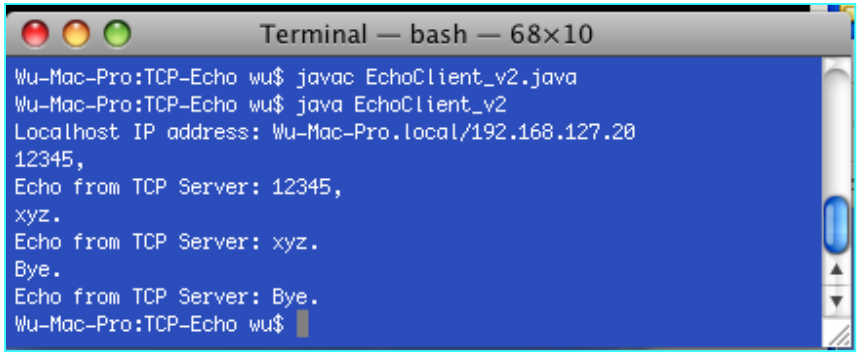
Exercise 1

- Replace the IP address 127.0.0.1 with your neighbor's IP address
- Change the port number to 4000
- Run the client/server codes

Example 2: Get Localhost IP address by a Java method

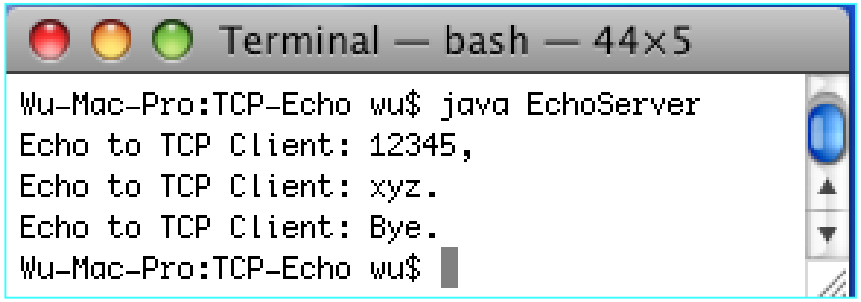
Client version 2

EchoClient_v2.java



Server (no change)

EchoServer.java



EchoClient_v2.java (1)

```
import java.io.*;
import java.net.*;
import java.lang.*;
public class EchoClient_v2 {
    public static void main(String[] args) throws IOException {
          try {
                    Socket echoSocket = null;
                    PrintWriter out = null;
                    BufferedReader in = null;
                    InetAddress localHost = InetAddress.getLocalHost();

                    System.out.println("Localhost IP address: " + localHost);
                    try {
                              echoSocket = new Socket(localHost, 2000);
                              out = new PrintWriter(echoSocket.getOutputStream(), true);
                              in = new BufferedReader(new
    InputStreamReader(echoSocket.getInputStream()));
                    } catch (UnknownHostException e) {
                              System.err.println("Do not know about host." + e);

                              System.exit(1);
                    }
```

Obtain the localhost IP address

EchoClient_v2.java (2)

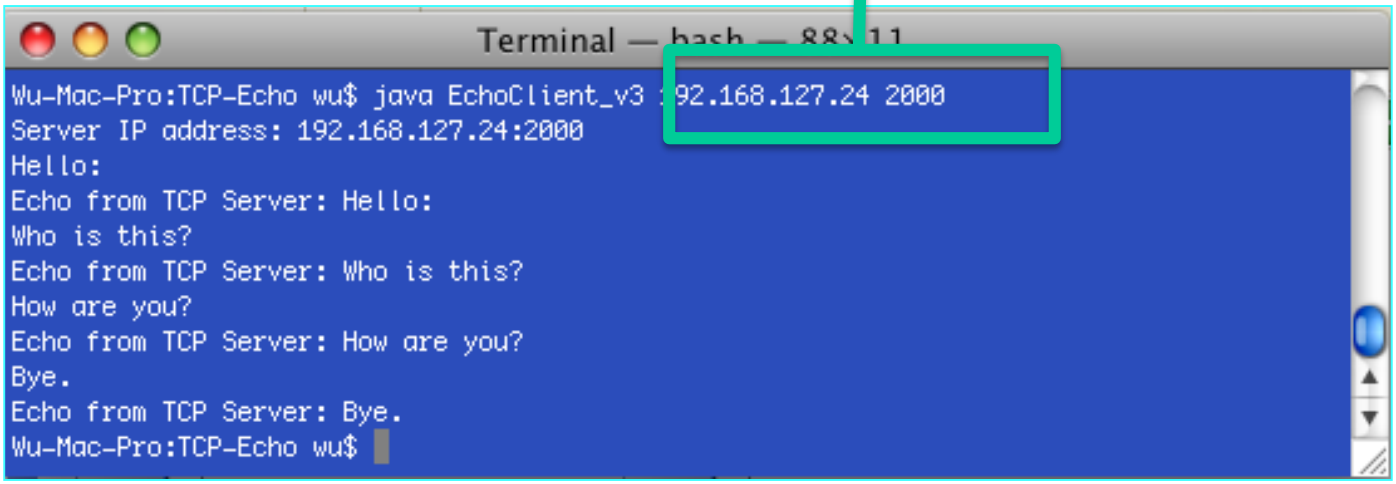
```
                BufferedReader stdIn = new BufferedReader(new
    InputStreamReader(System.in));
                String userInput;
                while ((userInput = stdIn.readLine()) != null) {
                        out.println(userInput);
                        System.out.println("Echo from TCP Server: " +
    in.readLine());
                        if (userInput.equals("Bye."))
                        break;
                }
                out.close();
                in.close();
                stdIn.close();
                echoSocket.close();
        }
        catch (IOException e) {
                System.err.println("Could not get I/O for the connection to
    localhost."+e);
                System.exit(1);
        }
    }
}
```

Example 3: TCP Socket between UNIX and XP

EchoServer.java

EchoClient_v3.java

Server's IP address and port number



EchoClient_v3.java (1)

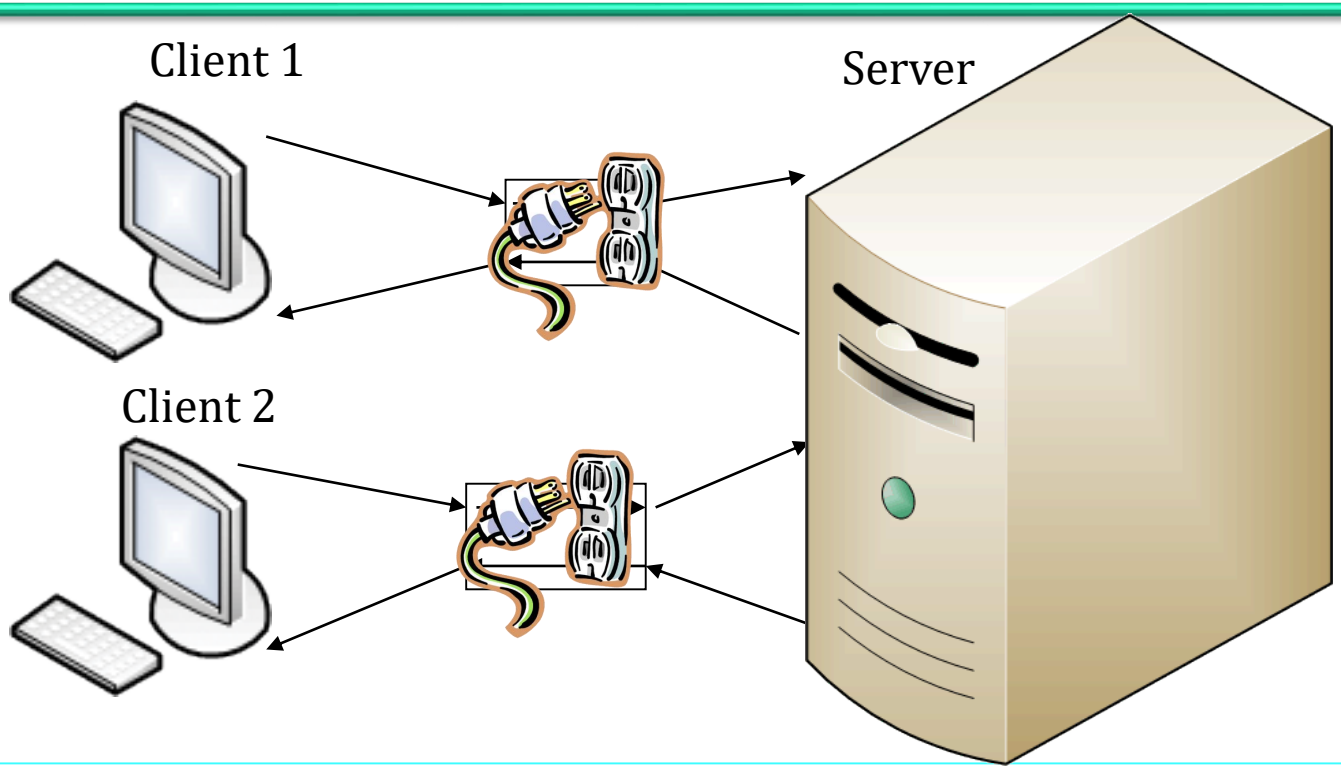
```
import java.io.*;
import java.net.*;
import java.lang.*;
//Please enter the following to run
//java EchoClient_v3 <Server IP> <Server Port>; use a space to separate them and hit
    enter
public class EchoClient_v3 {
    public static void main(String[] args) throws IOException {
          try {
                    Socket echoSocket = null;
                    PrintWriter out = null;
                    BufferedReader in = null;
                    String echoServer = null;
                    int echoServPort;          /* Echo server port */
          String echoServIP;     /* IP address of server */
          echoServIP = args[0]; /* First arg: server IP Address */
          echoServPort = Integer.parseInt(args[1]);        /* Second arg: string to echo
    */
                    System.out.println("Server IP address: " + echoServIP + ":" +
    echoServPort);
```

Obtain server's IP address and port number from command line

EchoClient_v3.java (2)

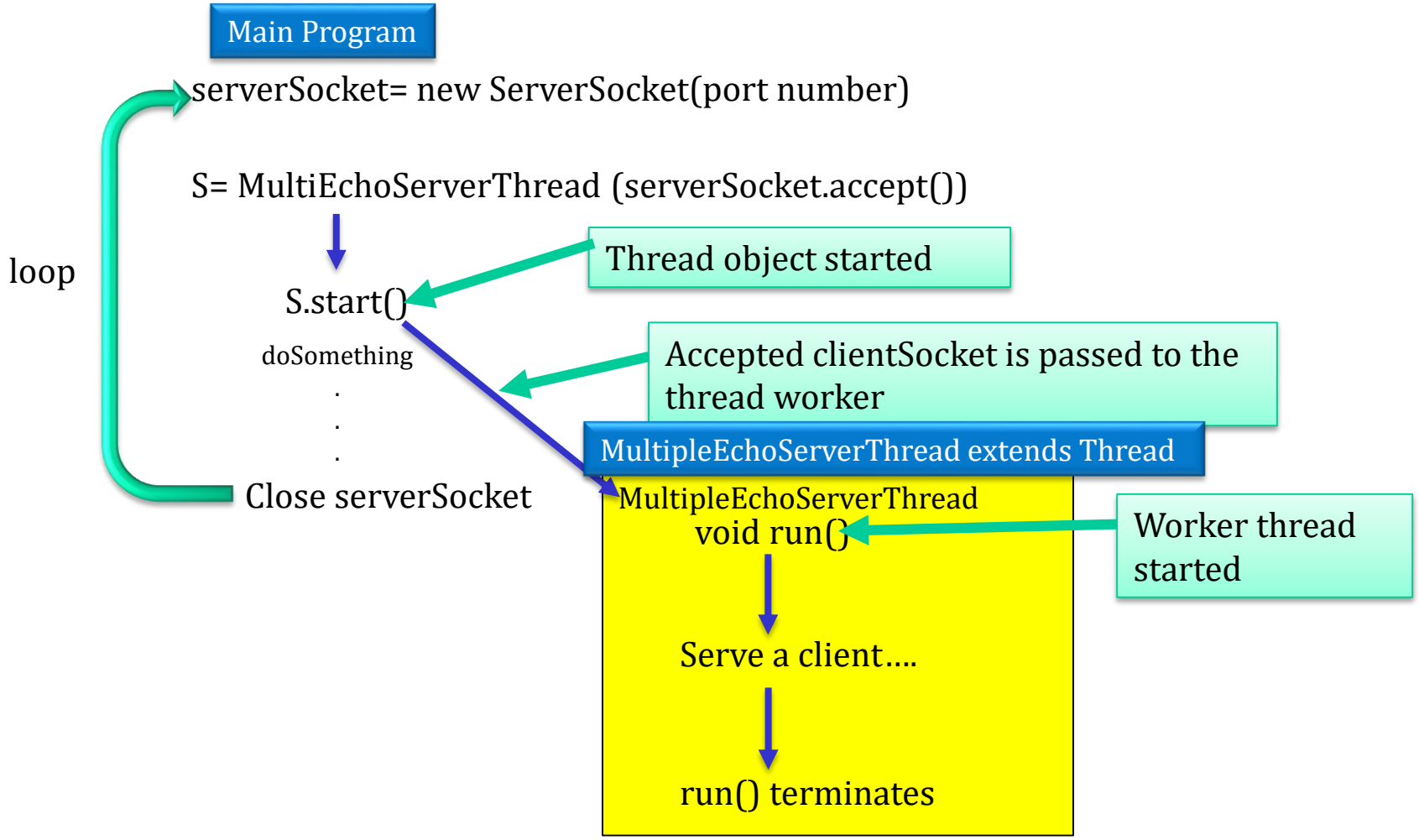
```
                        try {
                                echoSocket = new Socket(echoServIP, echoServPort);
                                out = new PrintWriter(echoSocket.getOutputStream(), true);
                                in = new BufferedReader(new InputStreamReader(echoSocket.getInputStream()));
                        }
                        catch (UnknownHostException e) {
                                System.err.println("Do not know about echoServer: " + echoServIP +e);
                                System.exit(1);
                        }
                        BufferedReader stdIn = new BufferedReader(new InputStreamReader(System.in));
                        String userInput;
                        while ((userInput = stdIn.readLine()) != null) {
                                out.println(userInput);
                                System.out.println("Echo from TCP Server: " + in.readLine());
                        if (userInput.equals("Bye."))
                        break;
                        }
                        out.close();
                        in.close();
                        stdIn.close();
                        echoSocket.close();
                }

                catch (IOException e) {
                        System.err.println("Could not get I/O for the connection to echoServer: "+e);
                        System.exit(1);
                }
        }
}
```

Example 3: Multiple threads



- Server supports many connections simultaneously
- Allows each client to receive independent service from the server

Multiple threads flow chart



Multiple Threads in Java TCP Programming

```
        /* Main program to start multiple threads */
boolean listening = true;
serverSocket = new ServerSocket(portNumber);
// start new thread in run method
while (listening)
        new MultiEchoServerThread(serverSocket.accept()).start();
serverSocket.close();

Class MultiEchoServerThread extends Thread {
        MultiEchoServerThread (...) {   // constructor
        this.clientSocket = clientSocket;
}
        public void run() {
                /* Exchange information with client*/
        }
}
```

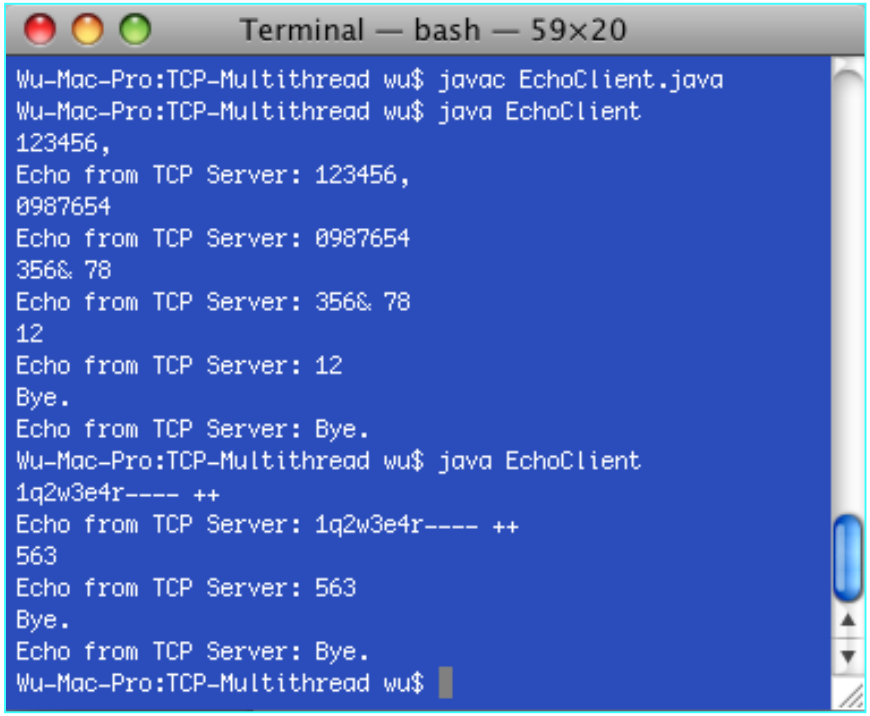
One server serves 2 clients simultaneously

- Server side:
  - Server codes:
    - MultiEchoServer.java (main program)
    - MultiEchoServerThread.java
  - Starts first
  - Always waiting for clients
  - Never quits
- Watch the port number changes when a client reconnects

Client 1

- Client 1:
  - Opens a shell in the same computer as the server and runs EchoClient
  - Connects to TCP server
  - Uses “Bye.” to disconnect
  - Reconnects to TCP server and uses a different port number

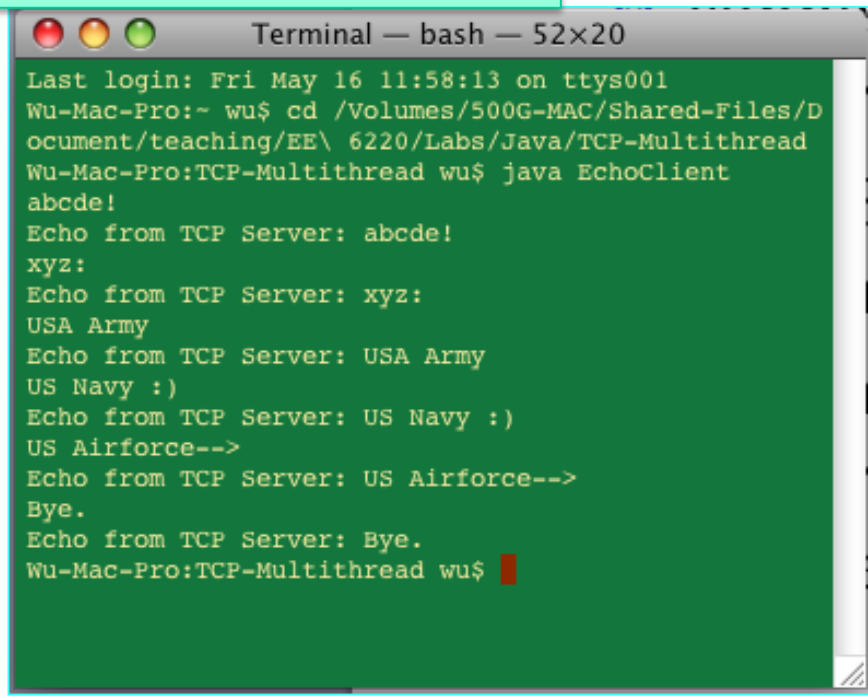
Code name: EchoClient.java



Client 2

- Open a shell in the same computer as the server and run EchoClient
- Connects to the TCP Server while Client 1 is connected
- Uses “Bye.” to disconnect

Code name: EchoClient.java



Server threads

- Server main thread waits for client requests
  - After accept() method, main thread creates a new worker thread to handle this specific socket connection
  - Main thread returns to accept new connections
- Worker thread handles this request, and provides logical independent service for each client so that information can be exchanged

Server main program

MultiEchoServer.java

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

public class MultiEchoServer {
   public static void main(String[] args) throws IOException {
ServerSocketserverSocket = null;
boolean listening = true;


        try {
serverSocket = new ServerSocket(2000);
        }
    catch (IOExceptione) {
System.out.println("Could not listen on port: 2000");
        System.exit(-1);
   }
        while (listening)
        new MultiEchoServerThread(serverSocket.accept()).start();
serverSocket.close();
   }
}
```

Server codes: Mutithreads(1)

MultiEchoServerThread.java

```
import java.net.*;
import java.io.*;
public class MultiEchoServerThread extends Thread {
   private Socket clientSocket = null;

   public MultiEchoServerThread(SocketclientSocket) {
this.clientSocket = clientSocket;
   }


   public void run(){
        String inputLine, outputLine;
        try {
PrintWriter out = new PrintWriter( clientSocket.getOutputStream(), true);
BufferedReader in = new BufferedReader(newInputStreamReader(
   clientSocket.getInputStream()));
```

Server codes: Mutithreads(2)

```
                while ((inputLine = in.readLine()) != null) {
outputLine = inputLine;
out.println(outputLine);
System.out.println("TCP Port Number: " + clientSocket.getPort());
System.out.println("Echo to TCP Client: " + outputLine);

                        if (outputLine.equals("Bye."))
                break;
                }
out.close();
in.close();
clientSocket.close();
        }
         catch (IOExceptione) {
e.printStackTrace();
         }

    }
}
```

Exercise 2

- Modify the client's TCP code to use multiple physical computers

Outline

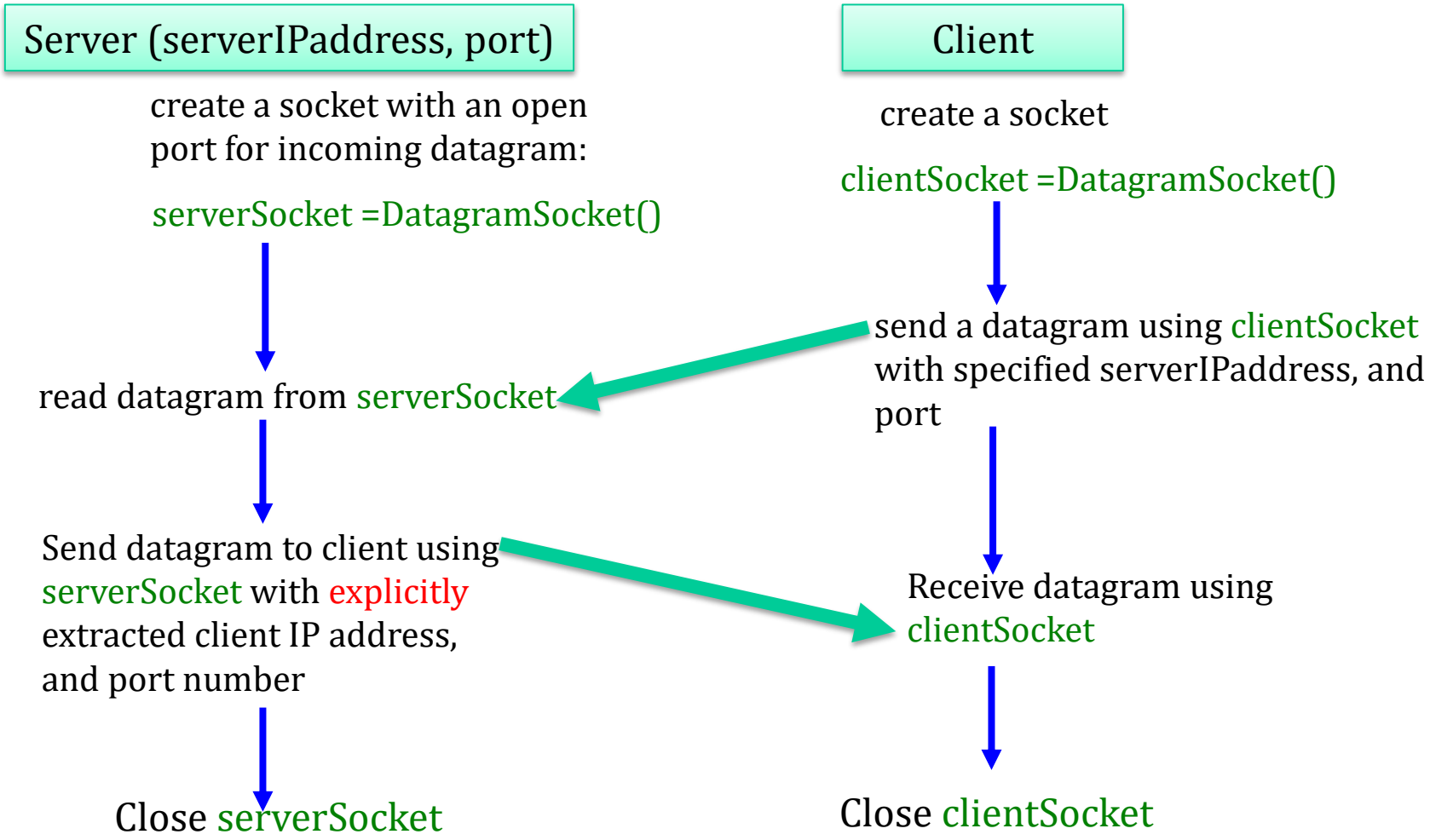
Part 1

- Socket programming
- TCP socket programming: Socket() function
- UDP socket programming: DatagramSocket() function

2 tuple for a UDP socket

- A UDP socket contains:
  - Server IP address
  - Server port number
- In contrast to a TCP socket, a UDP socket has no connection establishment
  - UDP is connectionless
- The UDP server must obtain the client IP address in the code explicitly because it uses a 2-tuple UDP socket
  - A major difference between UDP and TCP
- Sender explicitly specifies IP address and port of destination for each datagram

Client/server UDP socket interaction



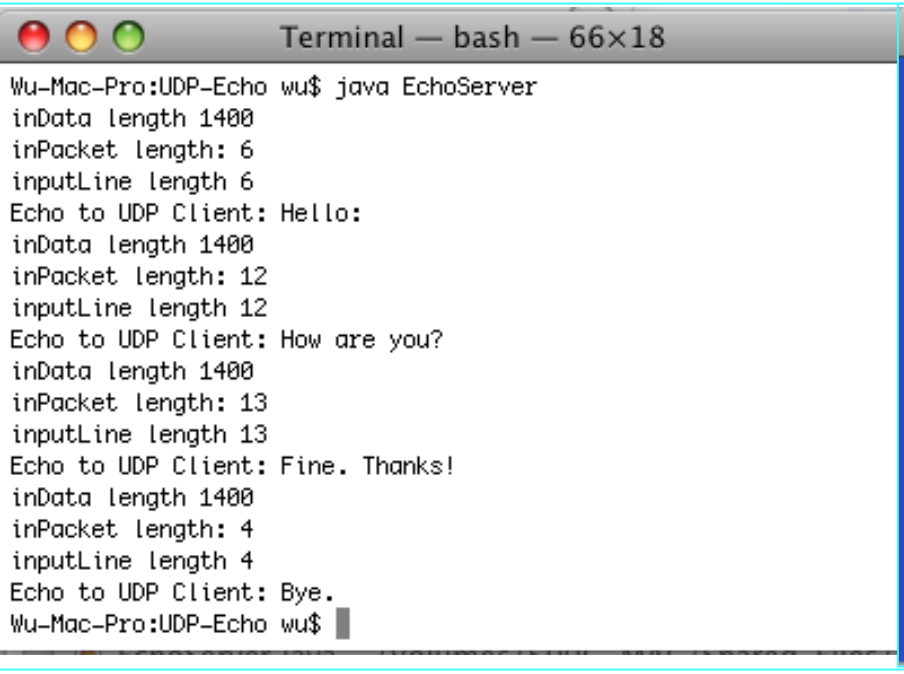
Example 4

- Simple Echo using UDP socket
- Server code: EchoServer.java
- Client code: EchoClient.java
- Both codes are run in one physical computer using loop back address 127.0.0.1

Run codes

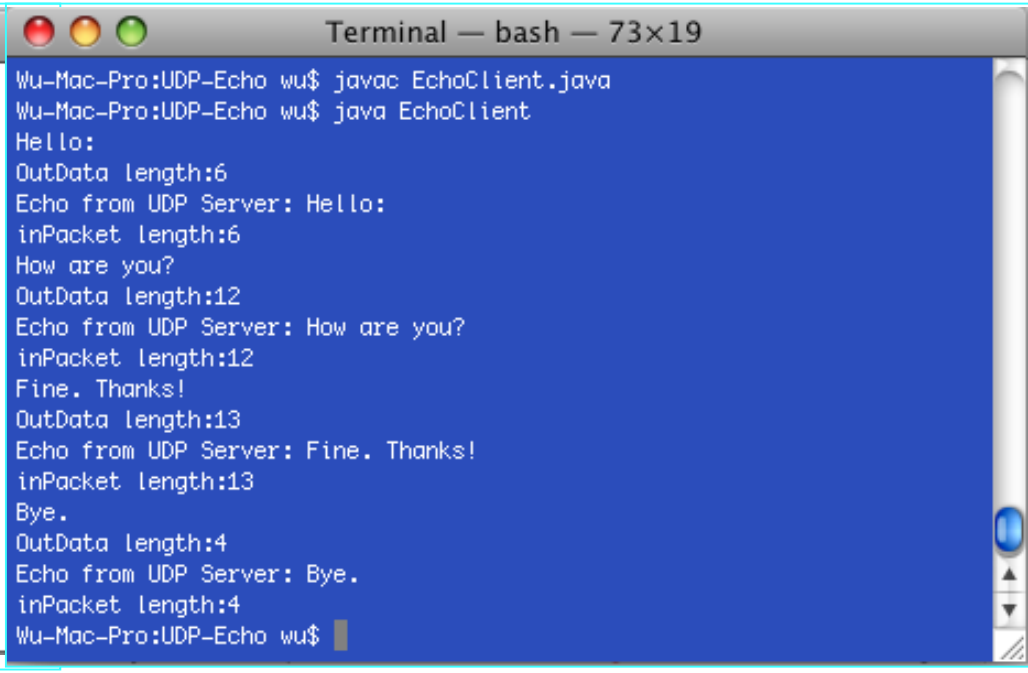
Server side

Code name: EchoServer.java



Client side

Code name: EchoClient.java



UDP techniques - Send

Create the object of datagram

```
   byte[] outData = new byte[1400];
   byte[] inData = new byte[1400];
outData = userInput.getBytes();
DatagramPacket outPacket = new DatagramPacket(outData,
   outData.length, destinationIPAddress, 2000);
echoSocket.send(outPacket);
```

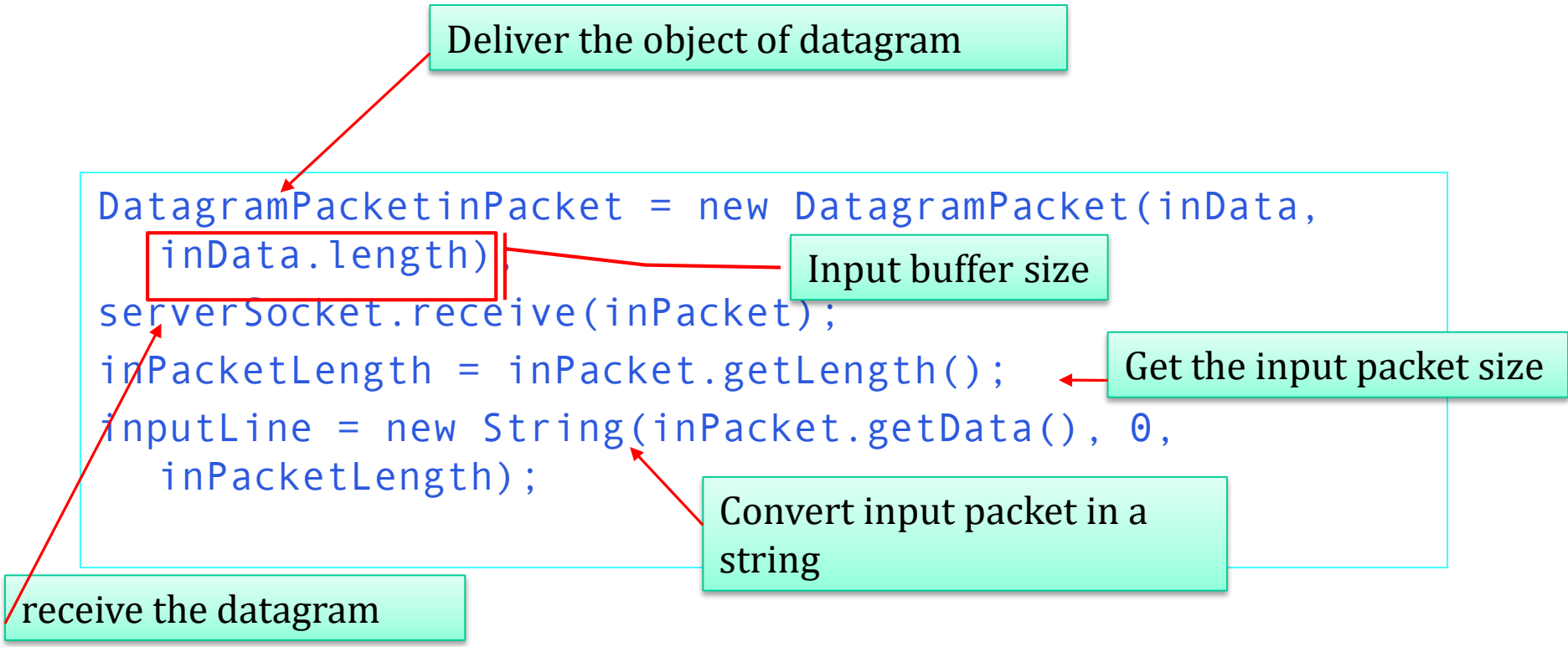
Put data in output buffer

UDP server port number

Datagram length

Deliver the datagram

UDP techniques - Receive



UDP server code (1)

```
import java.net.*;
import java.io.*;
public class EchoServer {
   public static void main(String[] args) throws IOException {
        DatagramSocket serverSocket = null;
        try {
            serverSocket = new DatagramSocket(2000);
        } catch (IOException e) {
            System.out.println("Could not listen on port: 2000" +
   e);
            System.exit(-1);
        }
       byte[] inData = new byte[1400];
       byte[] outData  = new byte[1400];
       int inPacketLength;
       String inputLine;
```

Create a UDP socket

UDP server code (2)

```
                    while (true) {
                        DatagramPacket inPacket = new DatagramPacket(inData, inData.length);
                        serverSocket.receive(inPacket);
                System.out.println("inData length " + inData.length);
                        inPacketLength = inPacket.getLength();
                        System.out.println("inPacket length: " + inPacketLength);
                inputLine = new String(inPacket.getData(), 0, inPacketLength);
                System.out.println("inputLine length " + inputLine.length());
                InetAddress clientIPAddress = inPacket.getAddress();
                        int port = inPacket.getPort();
                outData = inputLine.getBytes();
                DatagramPacket outPacket = new DatagramPacket(outData, outData.length,
    clientIPAddress, port);
                        serverSocket.send(outPacket);
                System.out.println("Echo to UDP Client: " + inputLine);
                if (inputLine.equals("Bye."))
                    break;
            }
            serverSocket.close();
        }
    }
```

Receive a datagram using the open socket

Obtain client IP address and port number

Send a datagram to the client using the IP address and port number

UDP client code (1)

```
import java.io.*;
import java.net.*;
public class EchoClient {
   public static void main(String[] args) throws IOException {

        BufferedReader stdIn = new BufferedReader(new
   InputStreamReader(System.in));
        DatagramSocket echoSocket = new DatagramSocket();
       InetAddress serverIPAddress = InetAddress.getLocalHost();
       byte[] outData = new byte[1400];
       byte[] inData = new byte[1400];
        String userInput;
        int inPacketLength;
```

Create a UDP socket

UDP client code (2)

```
                    while ((userInput = stdIn.readLine()) != null) {
                outData = userInput.getBytes();
                DatagramPacket outPacket = new DatagramPacket(outData,
outData.length, serverIPAddress, 2000);
                echoSocket.send(outPacket);
                System.out.println("OutData length:" + outData.length);
                DatagramPacket inPacket = new DatagramPacket(inData,
inData.length);
                echoSocket.receive(inPacket);
                inPacketLength = inPacket.getLength();
                String echoString = new String(inPacket.getData(), 0,
inPacketLength);
                System.out.println("Echo from UDP Server: " + echoString);
                System.out.println("inPacket length:" + inPacketLength);
        if (userInput.equals("Bye."))
                break;
         }
        stdIn.close();
        echoSocket.close();
    }
}
```

Send a datagram to the server using the IP address and port number

Receive a datagram from the server using the socket

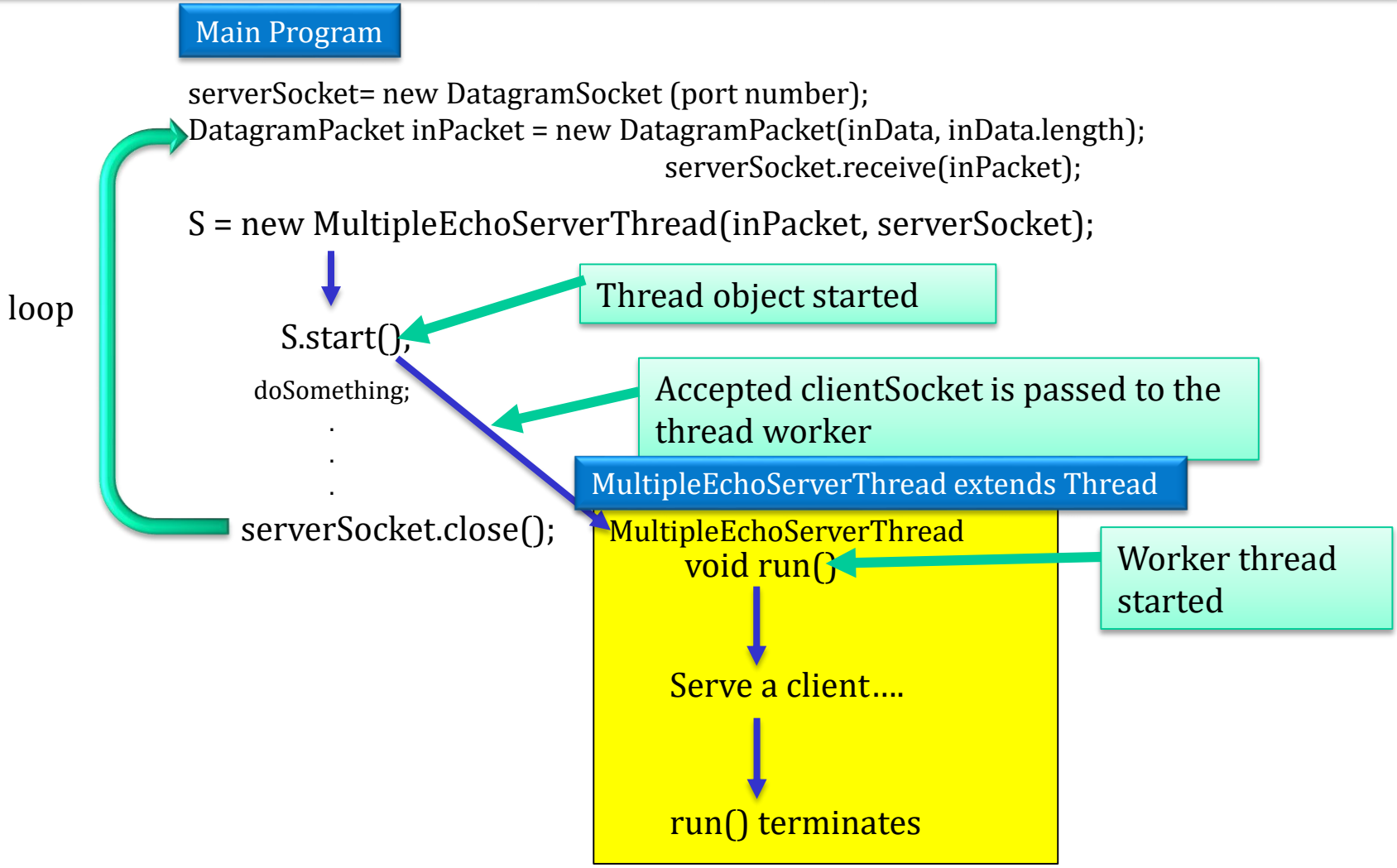
Exercise 3

- Modify codes for UDP communication with your neighbor's computer

Multi-thread UDP socket programming

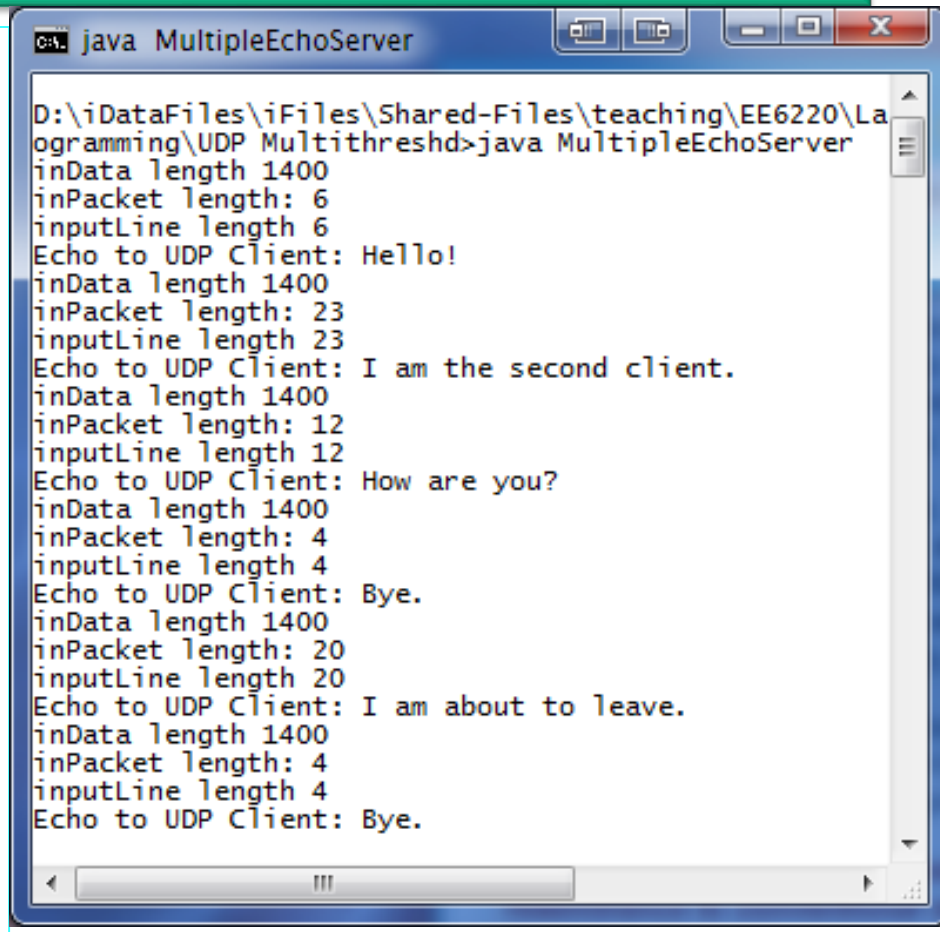
- DatagramSocket in UDP
- Server main thread waits for client requests
  - After receiving a datagram, the main program creates a new worker thread to handle this specific socket
  - Main thread returns to wait for new client
- Worker thread handles this request, and provides logical independent service for each client so that information can be exchanged
- The main program will keep listening for new socket requests from other clients using an infinite loop according to the always true flag while serving existing clients

Multiple threads UDP flow chart



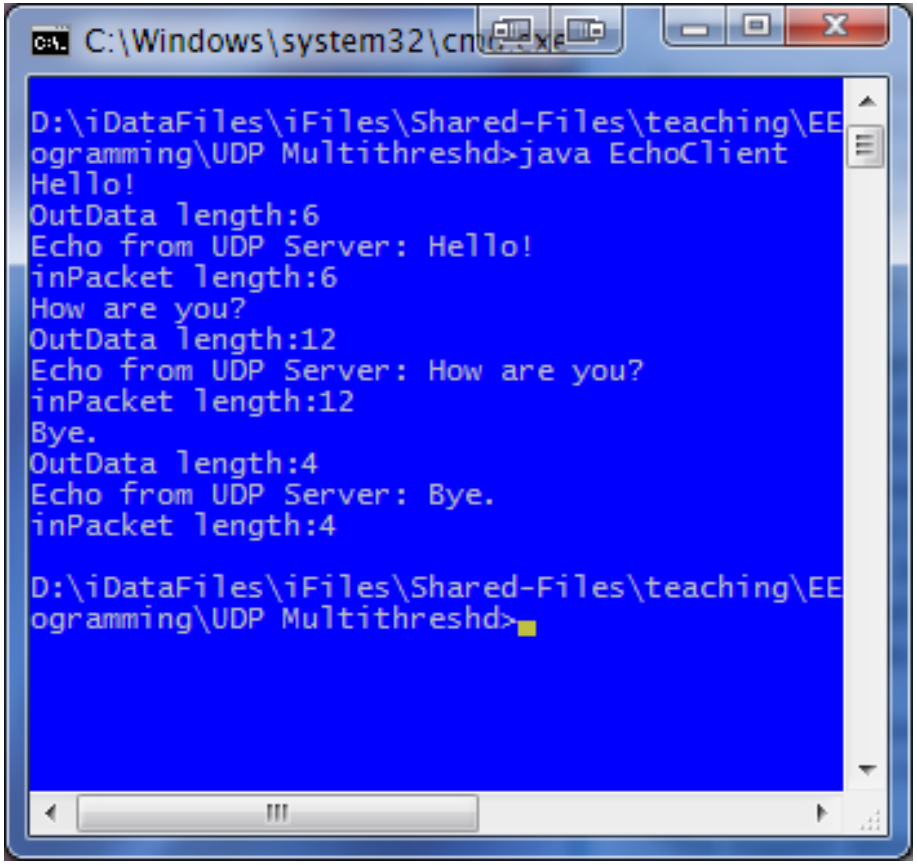
UDP server serves 2 clients simultaneously

- Server side:
  - Server codes:
    - MultiEchoServer.java (main program)
    - MultiEchoServerThread.java
  - Starts first
  - Always waiting for clients
  - Never quits



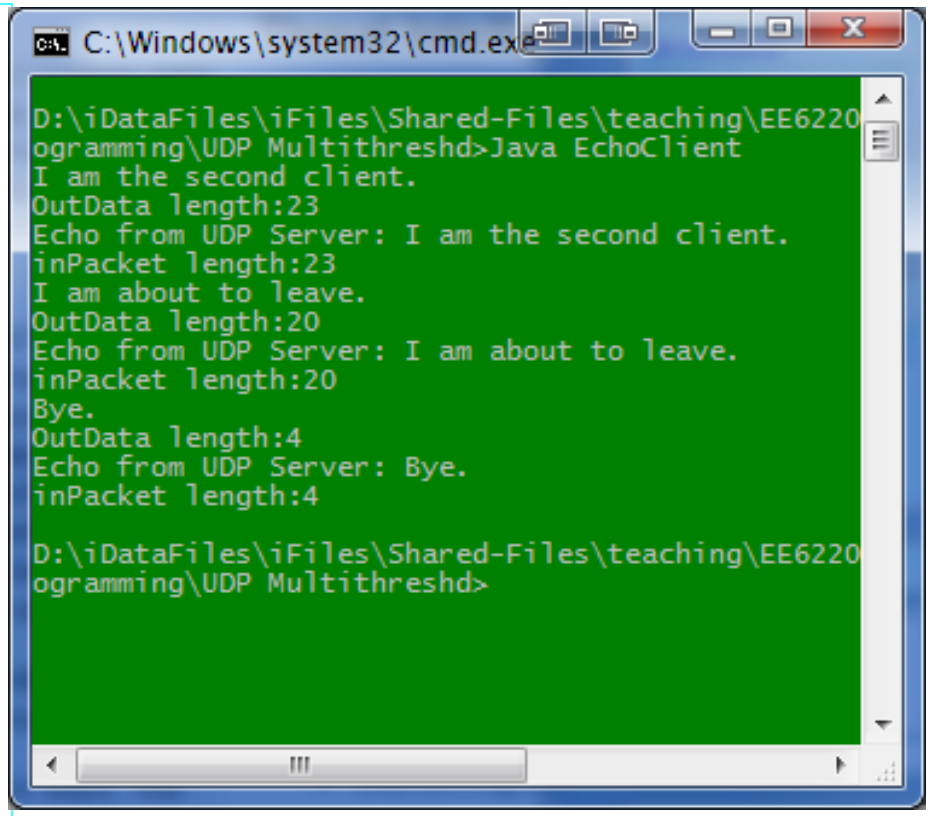
UDP Client 1

- Client 1:
  - Open a shell in the same computer as the server and run EchoClient
  - Connects to UDP server
  - Client 1 types in a message, receives the echo and displays it
  - Uses “Bye.” to disconnect



UDP Client 2

- Open a shell in the same computer as the server and run EchoClient
- Connects to UDP Server while Client 1 is connected
- Uses “Bye.” to disconnect



Server main program

MultiEchoServer.java

```
import java.net.*;
import java.io.*;  public class MultipleEchoServer {
    public static void main(String[] args) throws IOException {
          DatagramSocket serverSocket = null;
          boolean listening = true;
          try {
          serverSocket = new DatagramSocket(4000);
          }
          catch (IOException e) {
          System.out.println("Could not listen on port: 2000" + e);
          System.exit(-1);
          }
          byte[] inData = new byte[1400];
          while (listening) {
                    DatagramPacket inPacket = new DatagramPacket(inData, inData.length);
                    serverSocket.receive(inPacket);
                    System.out.println("inData length " + inData.length);
                    new MultipleEchoServerThread(inPacket, serverSocket).start();
          }
          serverSocket.close();
    }
}
```

Server codes: Mutithreads(1)

MultiEchoServerThread.java

```
private DatagramPacket inPacket = null;
private DatagramSocket serverSocket = null;

public MultipleEchoServerThread(DatagramPacket inPacket, DatagramSocket
serverSocket) {
      this.inPacket = inPacket;
      this.serverSocket = serverSocket;
}

public void run(){
      byte[] outData  = new byte[1400];
      int inPacketLength;
      String inputLine;
      inPacketLength = inPacket.getLength();
      System.out.println("inPacket length: " + inPacketLength);
      inputLine = new String(inPacket.getData(), 0, inPacketLength);
      System.out.println("inputLine length " + inputLine.length());
```

Server codes: Mutithreads(2)

```
                InetAddress clientIPAddress = inPacket.getAddress();
        int port = inPacket.getPort();
        outData = inputLine.getBytes();
        DatagramPacket outPacket = new DatagramPacket(outData,
    outData.length, clientIPAddress, port);
        try {
        serverSocket.send(outPacket);
        } catch (IOException e) {
        System.out.println("send error" + e);
        }
        System.out.println("Echo to UDP Client: " + inputLine);
    }
}
```

Exercise 4

- Modify codes for Multithread UDP communications with your neighbors' computers
- Modify the server code to display client's IP address and port number in each echo in the server's console

References

- https://docs.oracle.com/javase/tutorial/networking/sockets/index.html
- More examples: http://cs.lmu.edu/~ray/notes/javanetexamples/
- Python socket programming: https://pymotw.com/2/socket/tcp.html
- C socket programming: http://www.cs.rpi.edu/~moorthy/Courses/os98/Pgms/socket.html