CMPS 405: OPERATING SYSTEMS

Sockets Network Programming

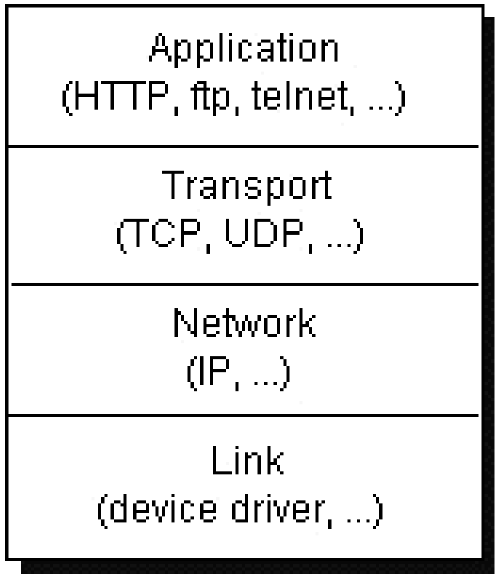
This material is based on the operating system books by Silberschatz and Stallings and other Linux and system programming books.

Objectives

- Motivations
- TCP vs. UDP transport protocols
- Sockets and ports
- Application protocols and the client/server model
- Networking in Java
- Stream-based communications in Java (TCP)
- Multithreaded servers
- The DNS application protocol
- Self study:
  - Packet-based communications in Java (UDP)

Motivation

- Computers running on the Internet communicate to each other using either the Transmission Control Protocol (TCP) or the User Datagram Protocol (UDP)



- IP is the basic protocol used on the Internet. It is responsible to make it possible to transfer packets from one computer to another, given their IP addresses.

TCP vs. UDP transport protocols : TCP

- ❖ **Definition:** TCP (Transmission Control Protocol) is a connection-based protocol that provides a reliable flow of data between two computers.

- ❖ **Connection-based:** After you make a connection between two programs using TCP, you have a steadily open connection on which you can send data without having to specify who you want the data to be sent to. This works very similar to reading or writing on a pipeline, or on a regular file.

- ❖ TCP guarantees that data sent from one end of the connection actually gets to the other end and in the same order it was sent. Otherwise, an error is reported.

TCP vs. UDP transport protocols : TCP

- The Hypertext Transfer Protocol (HTTP), File Transfer Protocol (FTP), and Telnet are all examples of applications that require a reliable communication channel.
- The order in which the data is sent and received over the network is critical to the success of these applications.

TCP vs. UDP transport protocols : UDP

- **Definition:** UDP (User Datagram Protocol) is a protocol that sends independent packets of data, called datagrams, from one computer to another with no guarantees about arrival. UDP is not connection-based like TCP.

- UDP is used for services that send small amounts of data between programs that do not require a long-time connection, or that send only little amount of data at a time.
  - The '**talk**' program uses the UDP protocol.
  - Another example of a service that doesn't need the guarantee of a reliable channel is the **ping** command. The purpose of the ping command is to test the communication between two programs over the network.

TCP vs. UDP transport protocols

- There are various other protocols used in conjunction with IP, such as ARP, RARP, ICMP, SNMP and others.

Sockets and Ports

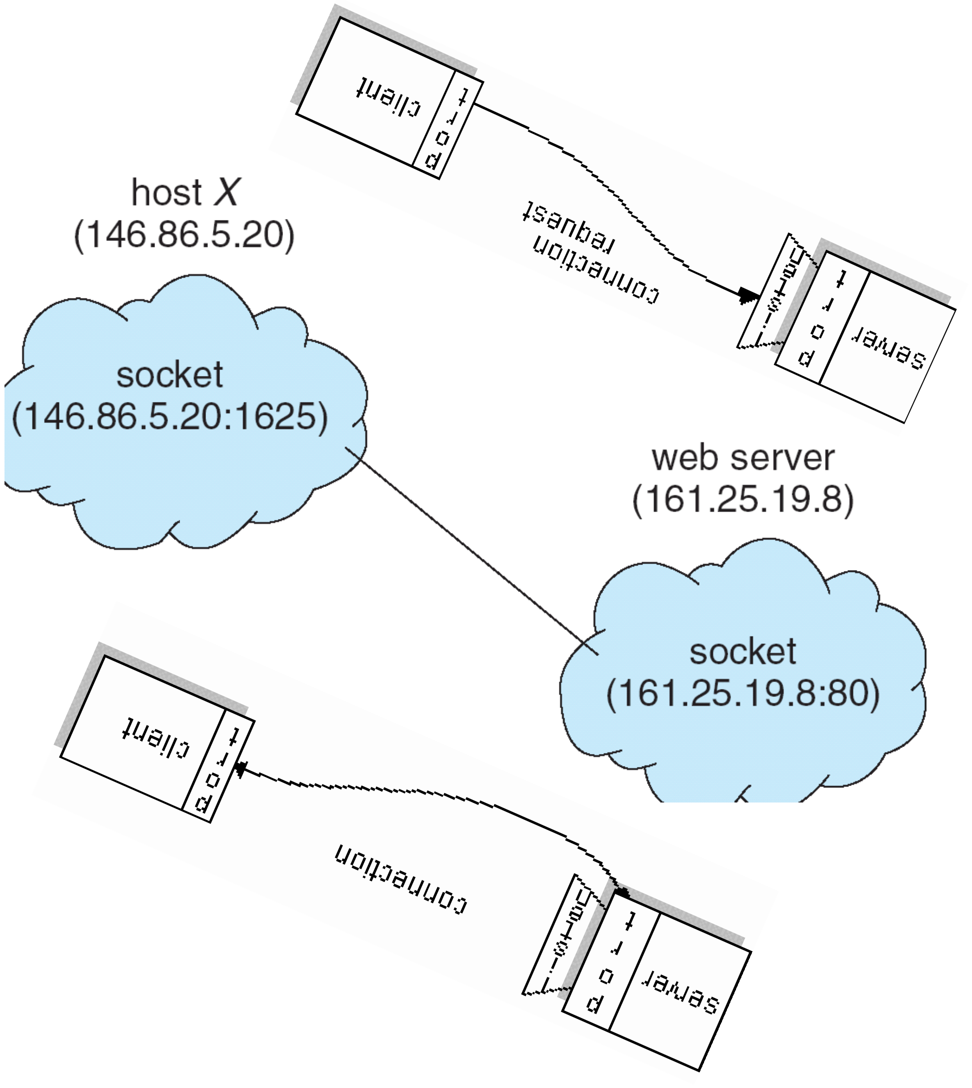
- **Definition:** A *socket* is one endpoint of a two-way communication link between two programs running on the network. A socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent.

- Concatenation of IP address and port Number

32 bits

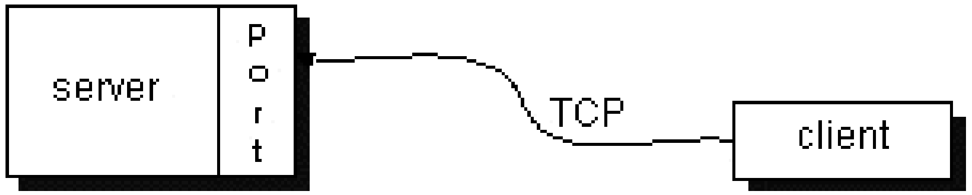
16 bits

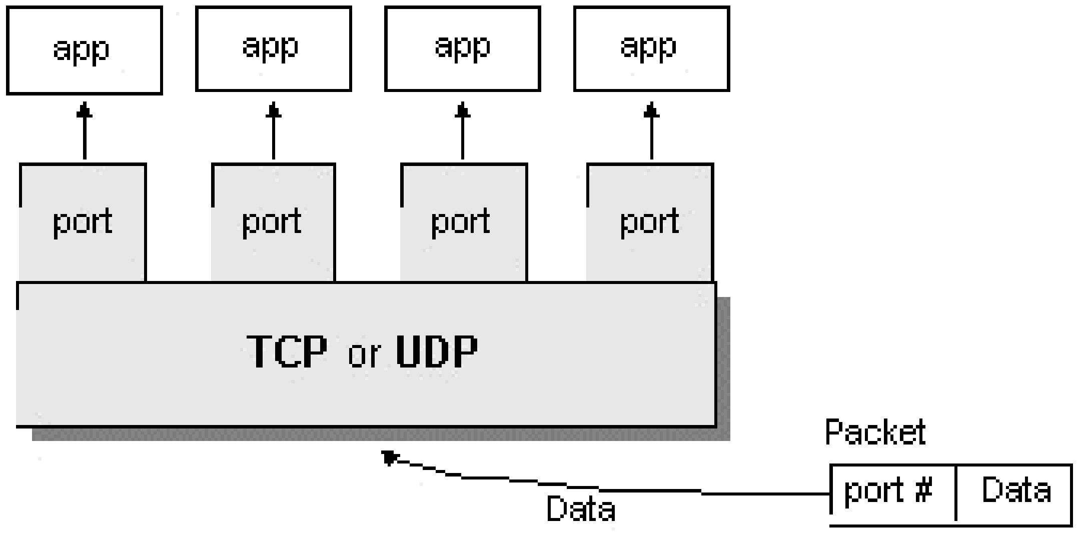
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**



Sockets and Ports

- **Definition:** The TCP and UDP protocols use ports to map incoming data to a particular process running on a computer.





Sockets and Ports

- Port numbers could be any number between 1 and 65535.
- Port numbers ranging from 0 - 1023 are restricted; they are reserved for use by well-known services.
- Use port numbers above 1024.
- Reserved port numbers are specified in the file /etc/services on any decent Unix machine.

Port No	Desc	Port No	Desc	Port No	Desc
13/tcp	daytime	20/tcp	ftp-data	23/tcp	telnet
13/udp	daytime	21/tcp	ftp	25/tcp	smtp
15/tcp	netstat				

Application Protocols: clients & servers

- **Internet Clients**
  - **FTP**
    - Ftp is a Client program used to transfer files across the Internet.
    - The Client connects to an FTPD Server and allows a user to scan a tree-structure of files and directories on the remote machine, retrieve and transmit files, etc.
    - It uses two connections - one to exchange commands, and another to exchange data (the files themselves).

- **Internet Servers**
  - **FTPD**
    - The server that normally accepts connections from ftp Clients.
    - It works off the well-known ftp port number **21**.

Application Protocols: clients & servers

- **Internet Clients**
  - **TELNET**
    - Telnet is a Client program that enables working on a remote machine, using the keyboard and screen of the local one.
    - It allows connecting from one type of machine to a completely different type.

- **Internet Servers**
  - **TELNETD**
    - The Server that talks to the Telnet client.
    - Uses the well-known Telnet port number **23**.

Networking in Java

- The networking package is java.net
- Two type of supported communications are:

1. **Stream-based communications (TCP)**
   - Applications view networking as streams of data
   - Connection-based protocol
   - Uses TCP (Transmission Control Protocol)

2. **Packet-based communications (UDP)**
   - Individual packets transmitted
   - Connectionless service
   - Uses UDP (User Datagram Protocol)

Stream-based communications in Java (TCP)

- The client establishes a connection with a service on a server with pre-known address and port number.

- The client does none, or any combination of the following one or multiple times then exits:

  - **Requests a service option**

  - **Receives a response of the requested service option**

  - **Perform a specific process due to the received response**

Client

Stream-based communications in Java (TCP)

- **Requesting a service option**

  - The client SW receive command from the user, translate it to a formatted message based on the application protocol specifications, and finally send that message to the server. (FTP service, DNS service).

  - In a simpler form, the client SW is only triggered by the user without any other follow up interaction with the user instead the client SW interacts with the server. (SMTP service)

  - In the simplest form, even the client SW does not send any message to the server. (time and date service, log service)

Client

Stream-based communications in Java (TCP)

❖ **Receiving the response of the requested service option**

- Receive messages from the server, translate them into human-readable form, and show them to the user. (FTP service, DNS service)
- In the a simpler form, the client SW receives the message(s) from the server and does not need to present it to the user. (DHCP service)
- In the simplest form, even the client SW does not receive any message(s) from the server. (log service, archive service)

Client

Stream-based communications in Java (TCP)

- The server ensures that the service is available to clients.
  - The availability is usually at all times, but it can also be bounded.

- The server accepts connection requests from clients.

- The server does none, or any combination of the following one or multiple times then closes the connection with the client:

  - **Receives a service request option**

  - **Perform a specific process due to the received response**

  - **Sends the response of the requested service option**

Server

Stream-based communications in Java (TCP)

- To implement a network application protocol you need to implement the two sides of the communication based on the specifications of the application protocol.

  - **The server side** **Server**

    Service

    The service logic at the Server side

  - **The client side** **Client**

    Service

    The service logic at the Client side

Implementing The Server-side of The Service

- Register an available port and a maximum number of clients (optional) to mark the service at the server side.

  - Create a ServerSocket object

```
ServerSocket server=null;
int port;
server = new ServerSocket(port);
```

Server

Implementing The Server-side of The Service

- Implement the availability of the service.
  - If you want the service to be available at all times then an infinite loop should host the actual code of the service code.

- Block to accept connections from clients
  - A connected client is represented by a Socket object.

- Execute the actual server service logic for this client

```
while (true) {
    Socket client = server.accept();
    /*Code to execute the actual service logic for
    this client*/
}
```

Implementing The Server-side of The Service

- The actual server service logic can be executed by:
  - A small segment of code that can be embedded, or

    ```
    //…… directly code the logic of the service
    ```

  - Calling a method that codes this logic, or

    ```
    service(client); /*code the service in a method and call
    it*/
    ```

  - Creating an object from a class that codes this logic, or

    ```
    new Service(client); /*code the service in a class and
    create an object of*/
    ```

  - Creating and starting a thread of a class that codes this logic.

    ```
    new Service(client).start(); //code a threaded service
    ```

Service
The service logic at the Server side
Server

Implementing The Server-side of The Service

- The server-side logic of the service might or might not require the creation of one or two network streams to exchange messages.
  - If the service requires only receiving from the client then an input stream is sufficient.

```
Scanner fromClient = null;
fromClient = new Scanner(client.getInputStream());
```

  - If the service requires only sending to the client, then an output stream is sufficient.

```
Formatter toClient = null;
toClient = new Formatter(client.getOutputStream());
```

  - If the service might require both sending and receiving, then both streams are essentially needed.

Server

Implementing The Server-side of The Service

- Implement the actual server-side logic of the service which might involve the exchange of messages and/or performing predefined actions/procedures/methods.
  - The logic depends on the specifications in the application protocol to be implemented which defines all possible types of interaction.
- Close opened network streams if any then close the connection with the client.
  - `fromClient.close();`
  - `toClient.close();`
  - `client.close();`

Implementing The Client-side of The Service

- Connect to a registered service on a known address and port number.

  - Create a Socket object

    ```
    Socket server = null;
    server = new Socket("localhost", 5563);
    ```

  - Execute the actual client service logic for this client

    ```
    /*Code to execute the actual service logic at
    client side*/
    ```

Client

Implementing The Client-side of The Service

Service

The service logic at the Client side

- The actual client service logic can be executed by:
  - A small segment of code that can be embedded, or

```
/*…… directly code the logic of the service at the client side */
```

  - Calling a method that codes this logic, or

```
service(server); /*code the service in a method and call it*/
```

  - Creating an object from a class that codes this logic, or

```
new Service(server); /*code the service in a class and create an object of*/
```

Client

Implementing The Client-side of The Service

- The client-side logic of the service might or might not require the creation of one or two network streams to exchange messages.
  - If the service requires only receiving from the server, then an input stream is sufficient.

```
Scanner fromServer = null;
fromServer = new Scanner(new (server.getInputStream());
```

  - If the service requires only sending to the server, then an output stream is sufficient.

```
Formatter toServer = null;
toServer = new Formatter(server.getOutputStream());
```

  - If the service might require both sending and receiving, then both streams are essentially needed.

Client

Implementing The Client-side of The Service

- Implement the actual client-side logic of the service which might involve the exchange of messages and/or performing predefined actions/procedures/methods.
    - The logic depends on the specifications in the application protocol to be implemented which defines all possible types of interaction.

- Close opened network streams if any then close the connection with the server.

```
fromServer.close();
toServer.close();
server.close();
```

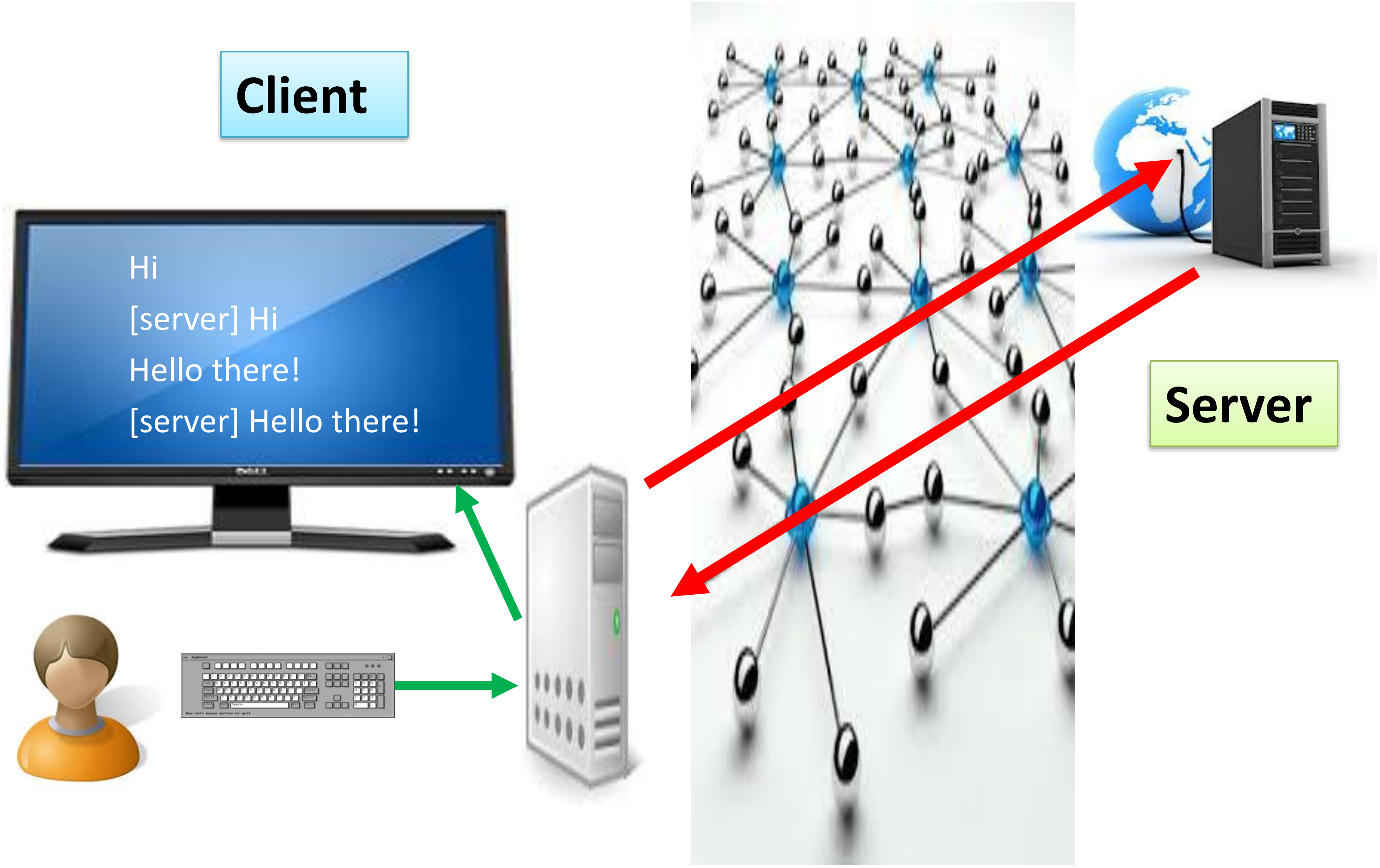
Client

Examples of Stream-based Communication in Java

- **Echo service**
  - The server writes back to the client what ever it receives from the client as long as the client did not send a special predefined message resulting in the server terminating the connection with the client.

- **DNS service**
  - The server resolves a domain name or IP address provided by the client and write it back to the client.

Echo service



Hi

[server] Hi

Hello there!

[server] Hello there!

Practice

- **Example:**
  - Implementing a stream-based Echo service.
  - In-class demo.
  - Check the code on BB.

Java API

ServerSocket class

Constructor and Description
ServerSocket()Creates an unbound server socket.
ServerSocket(int port)Creates a server socket, bound to the specified port.
ServerSocket(int port, int backlog)Creates a server socket and binds it to the specified local port number, with the specified backlog.
ServerSocket(int port, int backlog, InetAddress bindAddr)Create a server with the specified port, listen backlog, and local IP address to bind to.

Java API

ServerSocket class

Modifier and Type	Method and Description
Socket	accept()Listens for a connection to be made to this socket and accepts it.
InetAddress	getInetAddress()Returns the local address of this server socket.
int	getLocalPort()Returns the port number on which this socket is listening.
void	setPerformancePreferences(int connectionTime, int latency, int bandwidth)Sets performance preferences for this ServerSocket.
void	setSoTimeout(int timeout)Enable/disable SO_TIMEOUT with the specified timeout, in milliseconds.
void	close()Closes this socket.
boolean	isClosed()Returns the closed state of the ServerSocket.

Java API

Socket class

Modifier	Constructor and Description
protected	Socket()Creates an unconnected socket, with the system-default type of SocketImpl.
protected	Socket(InetAddress address, int port)Creates a stream socket and connects it to the specified port number at the specified IP address.
protected	Socket(String host, int port)Creates a stream socket and connects it to the specified port number on the named host.
protected	Socket(InetAddress address, int port, InetAddress localAddr, int localPort)Creates a socket and connects it to the specified remote address on the specified remote port.
protected	Socket(String host, int port, InetAddress localAddr, int localPort)Creates a socket and connects it to the specified remote host on the specified remote port.

Java API

Socket class

InetAddress	getInetAddress()Returns the address to which the socket is connected.
InetAddress	getLocalAddress()Gets the local address to which the socket is bound.
int	getLocalPort()Returns the local port number to which this socket is bound.
int	getPort()Returns the remote port number to which this socket is connected.
InputStream	getInputStream()Returns an input stream for this socket.
OutputStream	getOutputStream()Returns an output stream for this socket.
void	setPerformancePreferences(int connectionTime, int latency, int bandwidth)Sets performance preferences for this socket.
void	close()Closes this socket.
boolean	isClosed()Returns the closed state of the socket.

Java API

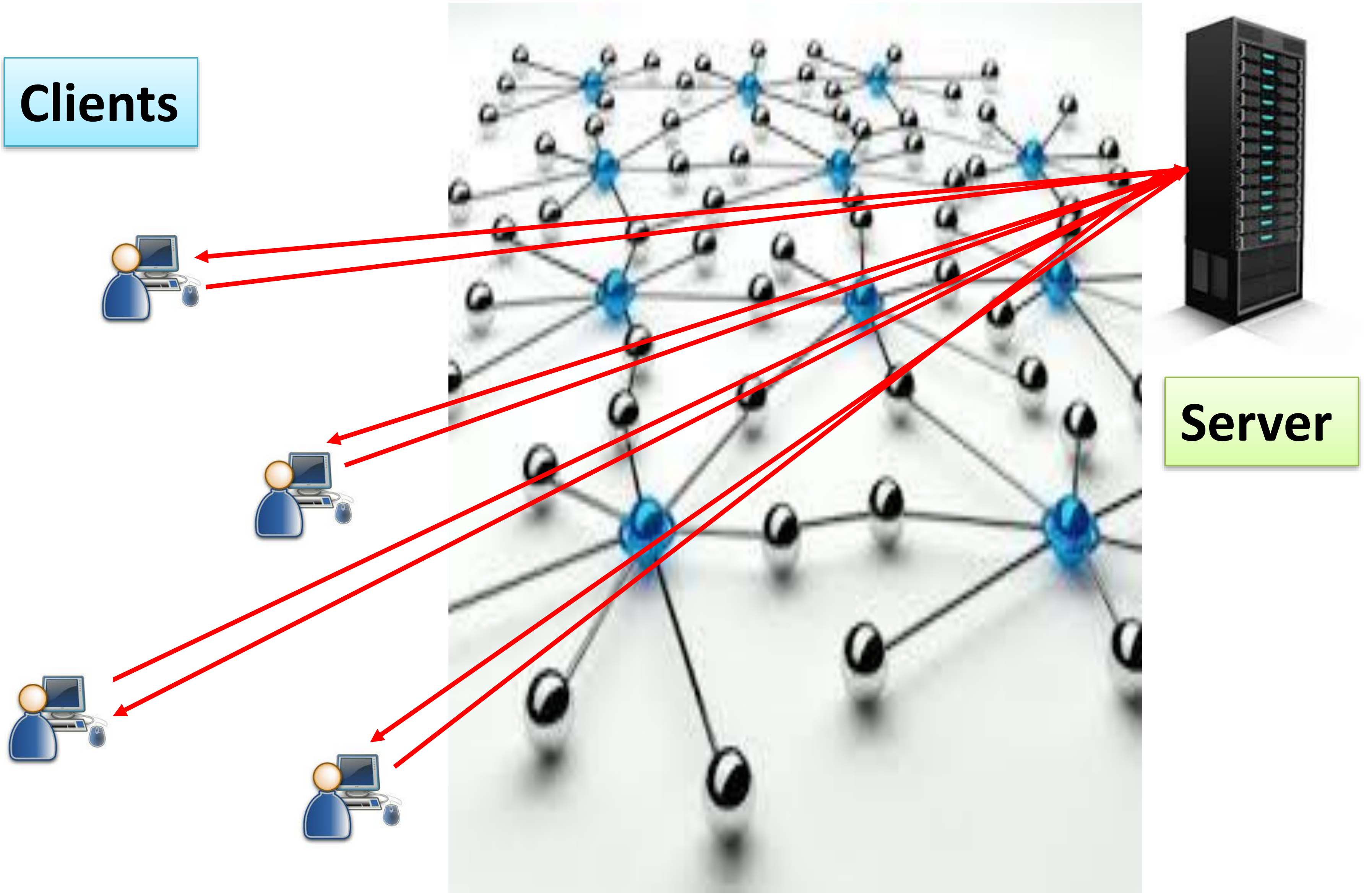
InetAddress class

String	getHostAddress()Returns the IP address string in textual presentation.
String	getHostName()Gets the host name for this IP address.
static InetAddress	getLocalHost()Returns the address of the local host.
boolean	equals(Object obj)Compares this object against the specified object.
byte[]	getAddress()Returns the raw IP address of this InetAddress object.
static InetAddress[]	getAllByName(String host)Given the name of a host, returns an array of its IP addresses, based on the configured name service on the system.
static InetAddress	getByName(String host)Determines the IP address of a host, given the host's name.

Objectives

- ❖ Motivations
- ❖ TCP vs. UDP transport protocols
- ❖ Sockets and ports
- ❖ Application protocols and the client/server model
- ❖ Networking in Java
- ❖ Stream-based communications in Java (TCP)
- ❖ Multithreaded servers
- ❖ The DNS application protocol
- ❖ Self study:
  - ➢ Packet-based communications in Java (UDP)

Serving multiple clients simultaneously



Multithreaded Service

- Most of the time the goal of the server is to serve multiple clients simultaneously.

- One way to do so is by implementing the service to be multithreaded service as we mentioned before.
  - Code the service in a class that extends Thread class or implements Runnable interface.

  - Create and start a thread of service whenever a client connects to the server.

```
new Service(client).start();
```

Practice

- **Example:**
  - Changing the code of the echo service to be multithreaded.
  - In-class demo.
  - Check the code on BB.

Examples of Stream-based Communication in Java

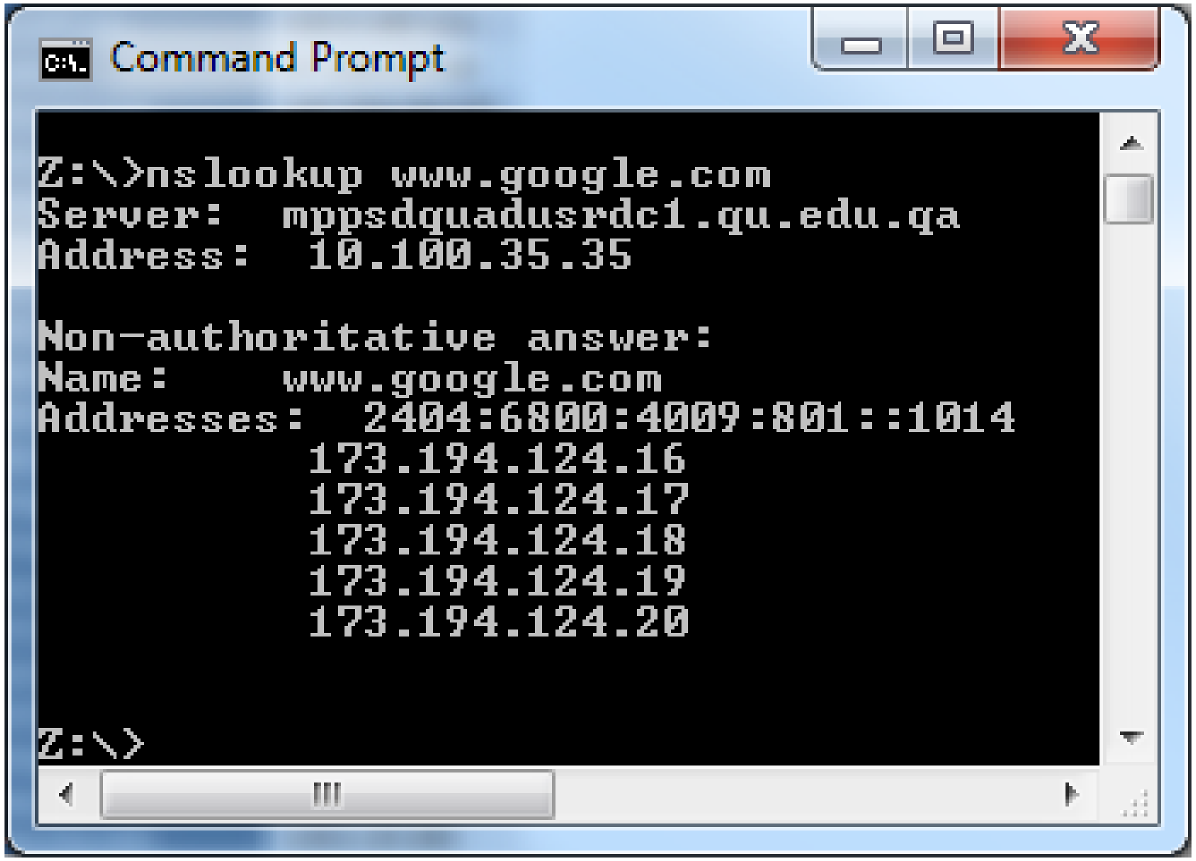
- ❖ **Echo service**
  - ➢ The server writes back to the client what ever it receives from the client as long as the client did not send a special predefined message resulting in the server terminating the connection with the client.

- ❖ **DNS service**
  - ➢ This service resolves a domain name to an IP address(s). The client sends domain name to the server and the server replies to the client by sending the corresponding IP(s) if any or a string saying “DN cannot be resolved”.

DNS Service

Domain Name	IP Address
cdns2.qatar.net.qa	212.77.192.60
www.yahoo.com	87.248.122.122
java.sun.com	192.9.162.55
www.google.com	173.194.36.52
www.google.com	173.194.36.48
www.google.com	173.194.36.50
www.google.com	173.194.36.51
www.google.com	173.194.36.49
www.qu.edu.qa	86.36.68.18
upm.edu.my	119.40.119.1
uum.edu.my	103.5.180.122
yu.edu.jo	87.236.233.10
www.sun.com	137.254.16.113
www.oracle.com	2.23.241.55
www.gm.com	2.22.9.175
www.motorola.com	23.14.215.224
www.nokia.com	2.22.75.80
www.intel.com	212.77.199.203
www.intel.com	212.77.199.210
www.apple.com	2.22.77.15
www.honda.com	164.109.25.248
www.gmail.com	173.194.36.54
www.gmail.com	173.194.36.53
www.hotmail.com	94.245.116.13
www.hotmail.com	94.245.116.11
www.toyota.com	212.77.199.224
www.toyota.com	212.77.199.203
www.gmc.com	2.22.247.241
www.mit.edu	18.9.22.169
www.cmu.edu	128.2.10.162

Use **nslookup** command in Windows to find corresponding IP(s) for a given domain name.

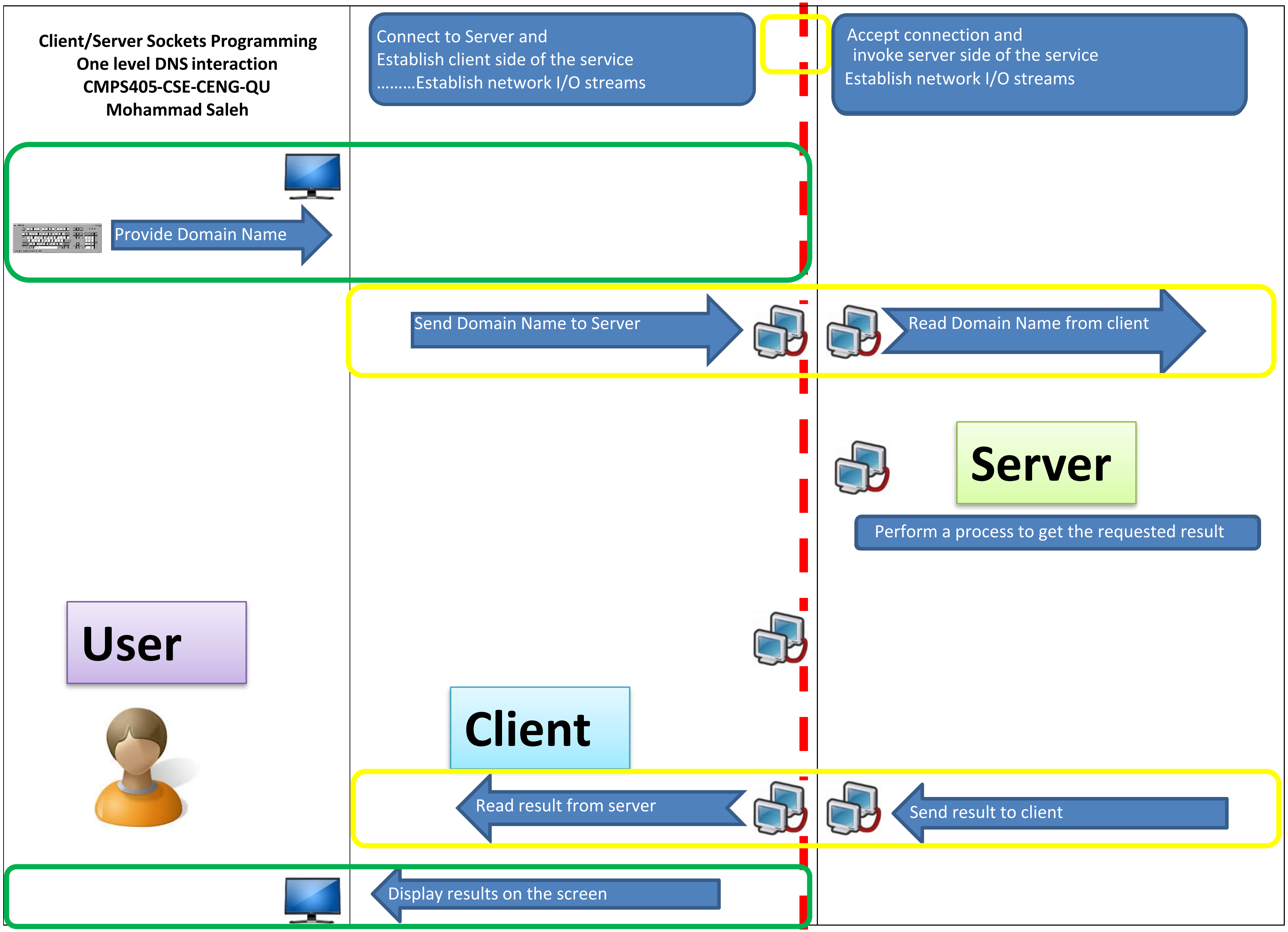


```
Z:\>nslookup www.google.com
Server:  mppsdquadusrdc1.qu.edu.qa
Address:  10.100.35.35

Non-authoritative answer:
Name:    www.google.com
Addresses:  2404:6800:4009:801::1014
          173.194.124.16
          173.194.124.17
          173.194.124.18
          173.194.124.19
          173.194.124.20


Z:\>
```

Client/Server Sockets Programming
One level DNS interaction
CMPS405-CSE-CENG-QU
Mohammad Saleh


Connect to Server and
Establish client side of the service
.........Establish network I/O streams
Accept connection and
invoke server side of the service
Establish network I/O streams
Provide Domain Name
Send Domain Name to Server
Read Domain Name from client
Server
Perform a process to get the requested result
User
Client
Read result from server
Send result to client
Display results on the screen

DNS Service

Practice

- **Example:**
  - Implementing a multithreaded DNS service.
  - In-class demo.
  - Check the code on BB.

Objectives

- ❖ Motivations
- ❖ TCP vs. UDP transport protocols
- ❖ Sockets and ports
- ❖ Application protocols and the client/server model
- ❖ Networking in Java
- ❖ Stream-based communications in Java (TCP)
- ❖ Multithreaded servers
- ❖ The DNS application protocol
- ❖ Self study:
  - ➢ Packet-based communications in Java (UDP)

Packet-based communications in Java (UDP)

- The **DatagramSocket** constructor takes an integer port-number argument binds the server to a port where it can receive packets from clients.
  - Clients sending packets to this Server specify the same port number in the packets they send.

- The server side of the application uses two objects of from the class **DatagramPacket** one to send and the other to receive.

Packet-based communications in Java (UDP)

- **DatagramSocket** methods:
  - ***receive*** waits for a packet to arrive at the Server.
    - Blocks until a packet arrives, then stores the packet in its **DatagramPacket** argument.
  - ***getAddress*** returns an **InetAddress** object containing the IP address of the computer from which the packet was sent.
  - ***getPort*** returns an integer specifying the port number through which the client computer sent the packet.
  - ***getLength*** returns an integer representing the number of bytes of data received.
  - ***getData*** returns a byte array containing the data.

Practice

- **Example:**
  - Implementing a UDP Echo application protocol.
  - In-class demo.
  - Check the code on BB.

Datagram Echo Server

```
public class Server {
    DatagramSocket server;
    public Server() {
    try{
     server = new DatagramSocket(5000);
    }
    catch(SocketException socketException){ System.exit(1);}
    while(true){
     try{
       byte[] data = new byte[100];
       DatagramPacket packet = new DatagramPacket(data, data.length);
       server.receive(packet);
       System.out.printf("Received packet from client %s:%s with length %s
                         with the body:\n%s\n",
                         packet.getAddress(),packet.getPort(),
                         packet.getLength(),
                         new String(packet.getData(),0,packet.getLength()));
       DatagramPacket sendPacket = new DatagramPacket(packet.getData(),
                                                      packet.getLength(),
                                                      packet.getAddress(),
                                                      packet.getPort());
       server.send(sendPacket);
    }
    catch(Exception e){}
    }
    public static void main(String[] args){ new Server(); }
}
```

Datagrams Echo Client

```java
public class Client {
  private DatagramSocket client;
  private Scanner kb;
  public Client() {
    try{
       client = new DatagramSocket();
       kb = new Scanner(System.in);
       while (true) {
         //on next slide
       }
    }
    catch(IOException e){System.exit(1);}
  }
  public static void main(String[] args){new Client();}
}
```

Datagrams Echo Client

```
while (true) {
   String message = kb.nextLine();
   byte[] data = message.getBytes();
   DatagramPacket packet = new DatagramPacket(data,
                                             data.length,
                                             InetAddress.getLocalHost(),
                                             5000);
   client.send(packet);
   if (message == null || message.equals("bye"))
      break;
   byte[] receivedData = new byte[100];
   DatagramPacket receivedPacket = new DatagramPacket(receivedData,
                                                    receivedData.length);
   client.receive(receivedPacket);
   System.out.printf("Received packet from server %s:%s with length %s
                     with the body:\n%s\n",
                     receivedPacket.getAddress(),receivedPacket.getPort(),
                     receivedPacket.getLength(),
                     New String(receivedPacket.getData(),0,
                                receivedPacket.getLength()));
}
```

Java API

DatagramSocket class

Modifier	Constructor and Description
protected	DatagramSocket()Constructs a datagram socket and binds it to any available port on the local host machine.
protected	DatagramSocket(int port)Constructs a datagram socket and binds it to the specified port on the local host machine.
protected	DatagramSocket(int port, InetAddress laddr)Creates a datagram socket, bound to the specified local address.

- The methods are similar to that of the **ServerSocket** and **Socket** classes.

Java API

DatagramPacket class

Constructor and Description
DatagramPacket(byte[] buf, int length)Constructs a DatagramPacket for receiving packets of length length.
DatagramPacket(byte[] buf, int length, InetAddress address, int port)Constructs a datagram packet for sending packets of length length to the specified port number on the specified host.

Modifier and Type	Method and Description
InetAddress	getAddress()Returns the IP address of the machine to which this datagram is being sent or from which the datagram was received.
byte[]	getData()Returns the data buffer.
int	getLength()Returns the length of the data to be sent or the length of the data received.
int	getPort()Returns the port number on the remote host to which this datagram is being sent or from which the datagram was received.
void	setAddress(InetAddress iaddr)Sets the IP address of the machine to which this datagram is being sent.
void	setData(byte[] buf)Set the data buffer for this packet.
void	setLength(int length)Set the length for this packet.
void	setPort(int iport)Sets the port number on the remote host to which this datagram is being sent.

Practice

- Check the examples on BB.
  - Study the programs.
  - Execute them and try to reason their execution results.
  - Try the commented variations in these programs.
  - Produce your own copy of each program with modifications aiming at deeper understanding of the subject.
- Implement your own network application protocol.