Multiplatform Application Development
2nd Course

Programming of Services and Processes

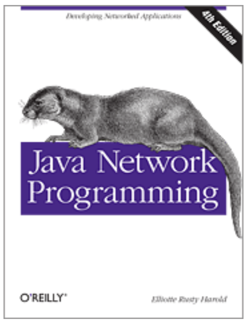
UNIT 6: Sockets

Professor: Empar Carrasquer

1. OBJECTIVES

- ✓ Study the types of sockets.
- ✓ Know the *java.net* package features.
- ✓ Implement programs that communicate using Java Sockets.
- ✓ Create simple client/server applications.

2. BIBLIOGRAPHY



- ➢ Book: **Java Network Programming, 4th Edition**

  Developing Networked Applications

  - ✓ 3rd Edition Free Download:

    http://it-ebooks.info/book/909/

- ➢ ORACLE JAVA TUTORIALS:

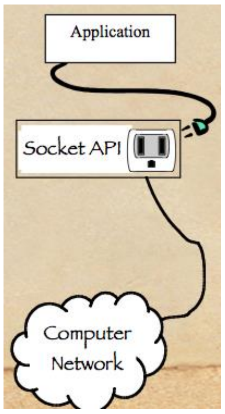
  http://docs.oracle.com/javase/tutorial/networking/sockets/

3. SOCKETS

3.1 INTRODUCTION

A **socket** is an abstract object that represents the end-points of a communication channel.

Did you know? The socket term comes from the electricity/phone socket metaphor.



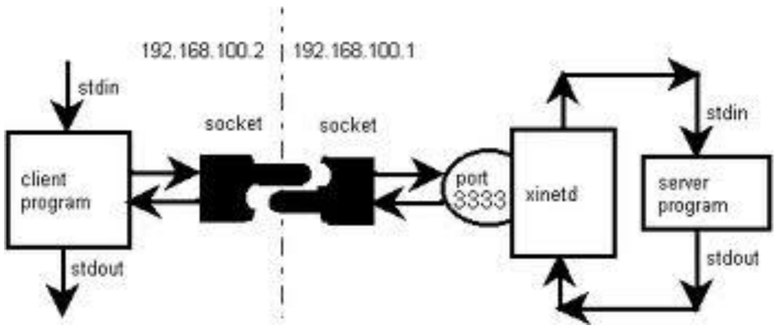
A socket on one computer that talks to a socket on another computer creates a **communication channel**. A programmer can use that channel to send data between the two machines. When you send data, each layer of the TCP/IP stack adds appropriate header information to wrap your data. These headers help the stack get your data to its destination. The good news is that the programming language hides all of this and the data is provided on streams (like in an I/O operation)

Origins.

Sockets were first introduced in the v.4.2 Unix BSD, in the year 1981. It became the standard interface for connecting to the Internet and since then all modern operating systems (Unix, Linux, Mac OS X, Android, Windows, etc.) have some implementation of the Berkeley socket interface.

Socket-based communication is **independent of the programming language** used for implementing it. That means that a socket program written in Java language can communicate to a program written in non-Java (say C, C++, C#, Python) socket program.

How do sockets work? Example of TCP socket:



A **server** (program) runs on a specific computer and has a **ServerSocket** that is bound to a specific local port. The server **listens** to the socket for a **client** to make a connection request. If everything goes well, the server accepts the connection.

Upon acceptance, server returns **a new Socket connected to client**. Server needs a new socket so that it can continue to listen to the original socket for incoming connection requests while serving the connected client on new Socket.

3.2 TYPES OF SOCKETS

Generally speaking, there are two types of sockets depending upon the transport protocol they use and how they work: **Stream Sockets** (TCP) and **Datagram Sockets** (UDP)

Both TCP and UDP sockets play the same role, but they do it differently. Both receive transport protocol packets and pass their contents to the Presentation Layer.

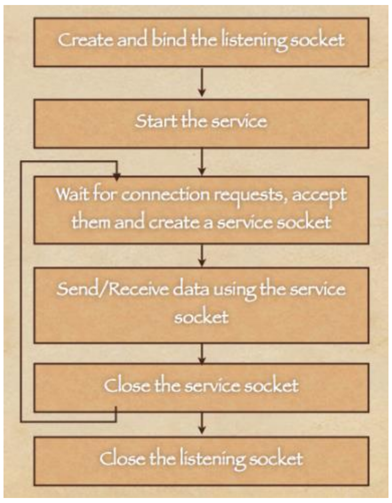
3.2.1 Stream Sockets

They are connection-oriented and make use of TCP transport protocol. TCP divides messages into packets (datagrams) and reassembles them in the correct sequence at the receiving end. It also handles requesting retransmission of missing packets. With TCP, the upper-level layers have much less to worry about. TCP is reliable.

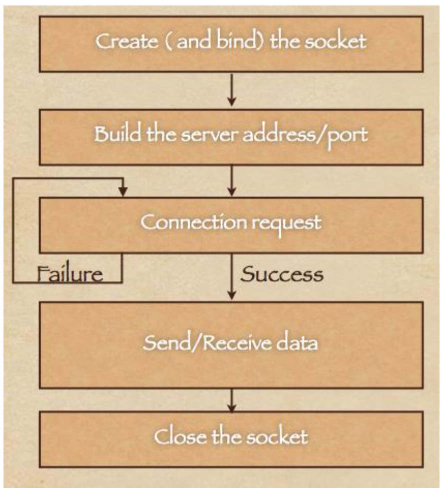
The communication using stream sockets implies: **connection**, **dialogue** and **disconnection**. A **server process** creates a socket and waits for a client process request for connection.

When a **client process** wants to start a communication, it creates its socket connected to the server and the communication channel is established. The communication terminates when client or server closes its socket.

Server Process


Create and bind the listening socket
Start the service
Wait for connection requests, accept them and create a service socket
Send/Receive data using the service socket
Close the service socket
Close the listening socket

Client Process

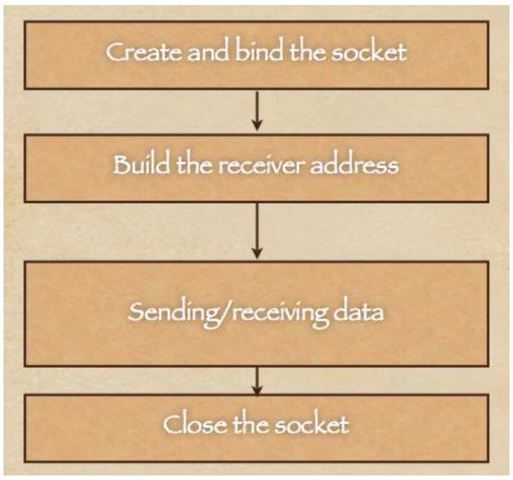


3.2.2 Datagram Sockets

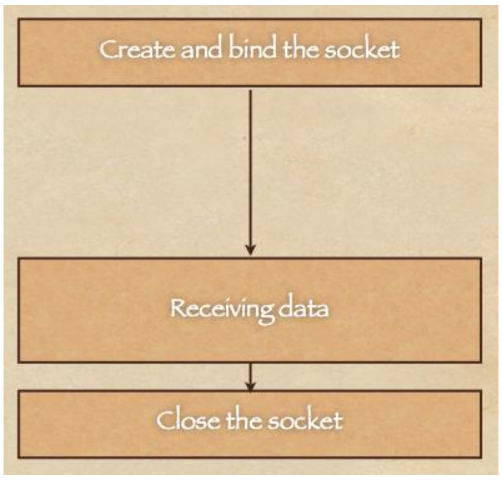
They are **connectionless** (the same socket can be used to send data to different receivers) and make use of UDP protocol. UDP doesn't provide any data order or retransmission requests features. It simply passes packets along. The upper layers have to make sure that the message is complete and assembled in correct sequence.

Datagram sockets make no distinction between server and client process. The steps to Send and/or Receive data are the following:

Send/Receive data



Receive data



In general, UDP has lower performance overhead on your application, but only if your application doesn't exchange lots of data all at once and doesn't have to reassemble lots of datagrams to complete a message. Otherwise, TCP is the simplest and probably most efficient choice.

4. Search and answer.
Search Internet and find out the type of socket (TCP/UDP and port) that the following well-known common services use: dhcp, ftp, telnet, pop, smtp, login, http, https, ssh.

4. PROGRAMMING JAVA SOCKETS

As we know, a socket is an endpoint of a two-way communication link between two programs running on the network. Socket is bound to a port number so that the Transport layer can identify the application where the data must be sent.

Java provides a set of classes, defined in a package called **java.net**, that simplify the complexity involved in the development of network client and server applications. The key classes, interfaces, and exceptions are:

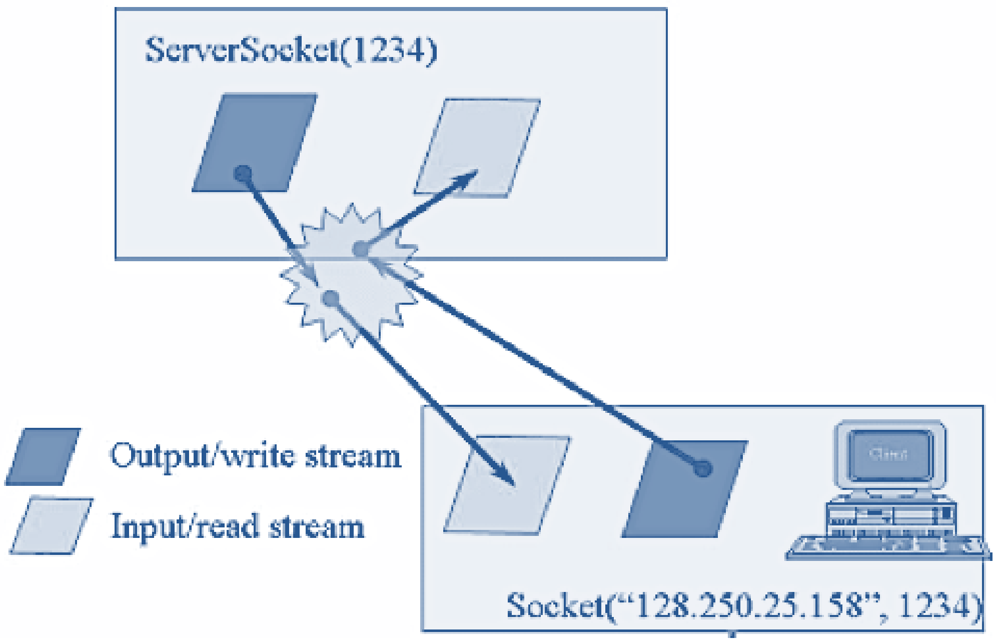
Classes		**Interfaces**	**Exceptions**
ContentHandler	*ServerSocket*	ContentHandlerFactory	BindException
DatagramPacket	*Socket*	FileNameMap	ConnectException
DatagramSocket	SocketImpl	SocketImplFactory	MalformedURLException
DatagramSocketImJpl	URL	URLStreamHandlerFactory	NoRouteToHostException
HttpURLConnection	URLConnection		ProtocolException
InetSocketAddress	URLEncoder		SocketException
MulticastSocket	URLStreamHandler		UnknownHostException
			UnknownServiceException
http://docs.oracle.com/javase/7/docs/api/java/net/package-summary.html			

4.1 TCP SOCKET

The two key classes from the *java.net* package used in creation of server and client programs are: ***ServerSocket*** and ***Socket***.

A server program creates a specific type of socket, *ServerSocket*, which is used to listen for client requests. When there is a connection request, the server program creates a *new socket* through which it will exchange data with the client using *input and output streams.*

The socket abstraction is very similar to the file concept: developers have to open a socket, perform I/O, and close it.



4.1.1 Implementing a simple client/server

The steps for a simple **socket server** program are:

1. Create the Server Socket: `ServerSocket s = new ServerSocket()`

   If the port is being used by any other application, an exception is thrown.
2. Bind to host and port: `s.bind(…)`
3. Wait for the client request connection: `Socket cs = s.accept()`
   - *accept()* method blocks the server until a client request connection is received.
   - When a client requests to connect, *accept()* returns a *new Socket* object connected to the client.
4. Get I/O streams from the new created Socket object to communicate to the client.
5. Perform communication from/to client (depending on the protocol programmed in the server)

   Receive from client: `read` from the input stream.

   Send to client: `write` to the output stream.
6. Close streams.
7. Close sockets.

The Java code is:

```java
private static final int PORT = 4444;
private static final String MACHINE = "localhost";

public static void main(String[] args) {
    try {
        System.out.println("CREATING SERVER SOCKET");
        ServerSocket serverSocket = new ServerSocket();

        System.out.println("BINDING TO SOCKET [" + MACHINE + "," + PORT + "]");
        InetSocketAddress sockAddr = new InetSocketAddress(MACHINE, PORT);
        serverSocket.bind(sockAddr);

        System.out.println("ACCEPTING CLIENT CONNECTIONS...");
        Socket clientSocket = serverSocket.accept();
        System.out.println(".......  CLIENT CONNECTION ESTABLISHED ON PORT: " + clientSocket.getPort());

        // I/O streams creation
        BufferedReader br = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
        BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(clientSocket.getOutputStream()));

        // read message from client
        String message = br.readLine();
        System.out.println(".......  MESSAGE FROM CLIENT RECEIVED: " + message);

        // send message to client
        message = ">>>>  Thank you. Message received ok! <<<<";
        bw.write(message);
        bw.newLine();
        bw.flush();

        // Closing client and server socket connection
        br.close();
        bw.close();
        clientSocket.close();
        serverSocket.close();
        System.out.println("SERVER FINISHED. CONNECTION CLOSED");

    } catch (IOException ex) {
        System.out.println("\nCONNECTION ERROR: " + ex.getMessage());
    }
}
```

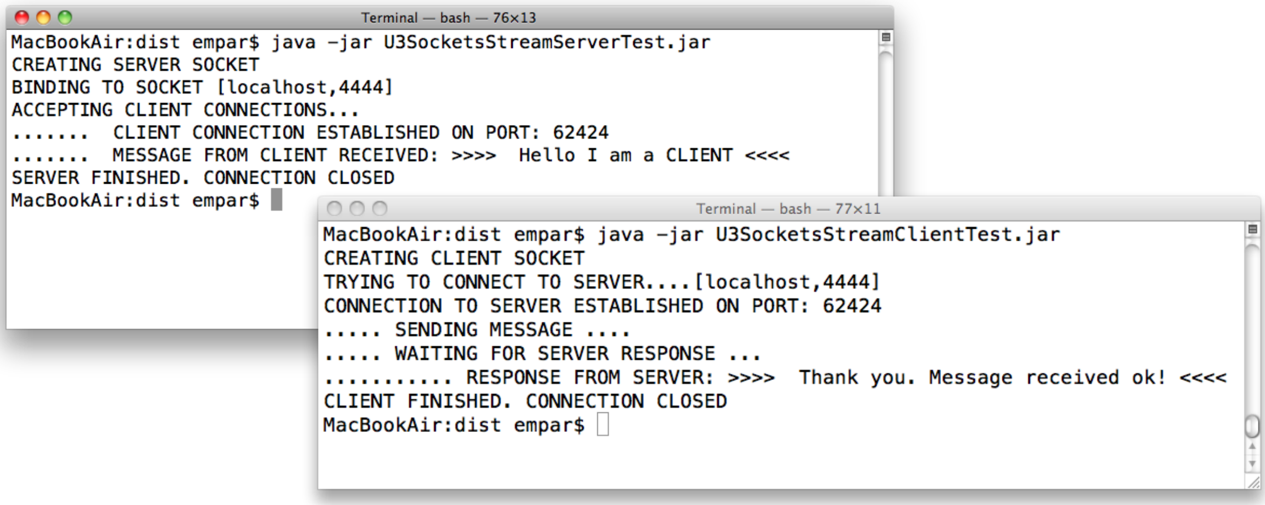
The steps for a simple **client socket** program are:

1. Create the client socket: `Socket client = new Socket();`
2. Connect to the server (host address and port): `client.connect(addr);`
3. Get I/O Streams to communicate to server.
4. Perform communication through streams: `read(),write(..)`
5. Close streams.
6. Close the socket.

The Java code is:

```java
private static final int PORT = 4444;
private static final String MACHINE = "localhost";

public static void main(String[] args) {
    try {
        System.out.println("CREATING CLIENT SOCKET");
        Socket clientSocket = new Socket();

        System.out.println("TRYING TO CONNECT TO SERVER...." + "[" + MACHINE + "," + PORT + "]");
        InetSocketAddress socketAddr = new InetSocketAddress(MACHINE, PORT);
        clientSocket.connect(socketAddr);

        System.out.println("CONNECTION TO SERVER ESTABLISHED ON PORT: " + clientSocket.getLocalPort());
         // I/O streams creation
        BufferedReader br = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
        BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(clientSocket.getOutputStream()));

        System.out.println("..... SENDING MESSAGE ....");
        bw.write(">>>>  Hello I am a CLIENT <<<<");
        bw.newLine();
        bw.flush();

        System.out.println("..... WAITING FOR SERVER RESPONSE ...");
        String message = br.readLine();

        System.out.println("........... RESPONSE FROM SERVER: " + message);

        // closing connection to server
        br.close();
        bw.close();
        clientSocket.close();
        System.out.println("CLIENT FINISHED. CONNECTION CLOSED");

    }
    catch (IOException ex) {
        System.out.println("\nCONNECTION ERROR: " + ex.getMessage());

    }
}
```

To **test** this simple client/server socket **start** the **server** program from a terminal system window and then **start** the **client** program from another terminal window. The output is:



```
MacBookAir:dist empar$ java -jar U3SocketsStreamServerTest.jar
CREATING SERVER SOCKET
BINDING TO SOCKET [localhost,4444]
ACCEPTING CLIENT CONNECTIONS...
.......  CLIENT CONNECTION ESTABLISHED ON PORT: 62424
.......  MESSAGE FROM CLIENT RECEIVED: >>>>  Hello I am a CLIENT <<<<
SERVER FINISHED. CONNECTION CLOSED
MacBookAir:dist empar$
```

```
MacBookAir:dist empar$ java -jar U3SocketsStreamClientTest.jar
CREATING CLIENT SOCKET
TRYING TO CONNECT TO SERVER....[localhost,4444]
CONNECTION TO SERVER ESTABLISHED ON PORT: 62424
..... SENDING MESSAGE ....
..... WAITING FOR SERVER RESPONSE ...
........... RESPONSE FROM SERVER: >>>>  Thank you. Message received ok! <<<<
CLIENT FINISHED. CONNECTION CLOSED
MacBookAir:dist empar$
```

Data Transmission recommendations:

- ✓ To transmit text only data → you can use *InputStream* and *OutputStream*
- ✓ To transmit binary data → you must turn the streams into *DataInputStream* and *DataOutputStream*.
- ✓ To transmit serialized objects → use *ObjectInputStream* and *ObjectOutputStream* instead.

5. Practice example: Knock, Knock!

- Read carefully the Knock, knock client/server socket example at: http://docs.oracle.com/javase/tutorial/networking/sockets/clientServer.html
- Then implement and test the client/server programs in your computer (test by starting multiple clients):
  a. Implement a first version of the server accepting multiple client connections sequentially.
  b. Implement a second version of the server that accepts simultaneous client connections using threads.
- Now work in pairs: one of you executes *n* client programs and the other executes the server program.
- Finally, test the server/client in the classroom. Decide one of you that will be the server and the others will be clients.

For your knowledge:

- ➢ The client program can run on any computer in the network (LAN, WAN, or Internet) as long as there is no firewall between them that blocks the communication**.**
- ➢ It is not possible that multiple server programs to run on the same port: port numbers are a mutually exclusive resource so they cannot be shared among different processes at the same time.

4.2 UDP SOCKET

As we know, TCP guarantees the delivery of packets and preserves their order on destination, with a performance cost, of course. However, sometimes these features are not required (for example, in audio and video streaming, loss of packets can be tolerated) and it would be better to use a lighter transport protocol: the UDP protocol that uses *datagram packets*.

Datagram packets are used to implement a *connectionless* packet delivery service supported by the UDP protocol. Each message is transferred from source machine to destination based on information contained within that packet. That means, **each packet needs to have destination address and each packet might be routed differently**, and might arrive in any order. **Packet delivery is not guaranteed**.

The Datagram packet format is:

| Msg | length | Host | serverPort |

Java supports datagram communication through the following classes:

- DatagramPacket
- DatagramSocket

- The class **DatagramPacket** contains several **constructors** that can be used for creating packet object. Two of them are:

  `DatagramPacket(byte[] buf, int length)`

  Constructs a DatagramPacket for **receiving packets** of length **length**.

  `DatagramPacket(byte[] buf, int length, InetAddress address, int port);`

  ➢ This constructor creates a datagram packet for **sending packets** to a remote host of length ***length*** to the specified ***port*** number on the specified ***address***. The ***message*** to be transmitted is indicated in the first argument: ***buf*** .

  The key **methods** of DatagramPacket class are:

  `byte[] getData()` → Returns the data buffer.

  `int getLength()` → Returns the length of the data to be sent or received.

  `void setData(byte[] buf)` → Sets the data buffer for this packet.

  `void setLength(int length)` → Sets the length for this packet.

- The class **DatagramSocket** supports various methods that can be used for transmitting or receiving a datagram over the network. The two key methods are:

  `void send(DatagramPacket p)` → **Sends** a *DatagramPacket* from this socket.

  ➢ The DatagramPacket includes the following information: the data to be sent, its length, the IP address of the remote host, and the port number on the remote host.

  `void receive(DatagramPacket p)` → **Receives** *DatagramPacket* from this socket.

  ➢ This method blocks until a datagram is received. The DatagramPacket received contains the data and also the sender IP address and port on sender's machine.

  The **message** is **truncated** if its length is longer than the packet length.

4.2.1 Implementing a simple client/server

A **simple UDP server program** that waits for client's requests and then sends back a response:

```
private static final int PORT = 1234;
private static final String MACHINE = "localhost";

public static void main(String[] args) {
    try {
        System.out.println("CREATING DATAGRAM SOCKET ON " + MACHINE + " Port: " + PORT);
        InetSocketAddress sockAddr = new InetSocketAddress(MACHINE, PORT);
        DatagramSocket datagramSocket = new DatagramSocket(sockAddr);

        System.out.println("WAITING FOR A MESSAGE ...");
        byte[] message = new byte[40];
        DatagramPacket datagramPacket = new DatagramPacket(message, message.length);

        datagramSocket.receive(datagramPacket); // blocks until message is received

        // Message received: get sender's address and port information
        InetAddress senderAddr = datagramPacket.getAddress();
        int senderPort = datagramPacket.getPort();

        System.out.println(".... MESSAGE FROM ["
                + senderAddr.getHostAddress() + ","
                + senderPort + "] RECEIVED: "
                + new String(message));

        System.out.println(".... SENDING RESPONSE TO SENDER...");
        String s = ">>>>Message received OK<<<<";
        byte[] messageBack = s.getBytes();
        DatagramPacket datagramPacketBack = new DatagramPacket(messageBack, messageBack.length,
                                                        senderAddr, senderPort);
        datagramSocket.send(datagramPacketBack);

        System.out.println("RESPONSE SENT.  CLOSING SOCKET");
        datagramSocket.close();
        System.out.println("SERVER FINISHED");

    } catch (SocketException ex) {
        System.out.println("nSOCKET ERROR: " + ex.getMessage());
    } catch (IOException ex) {
        System.out.println("nI/O ERROR: " + ex.getMessage());
    }
}
```

The **UDP simple client**:

```
private static final int TOPORT = 1234;
private static final String TOMACHINE = "localhost";

public static void main(String[] args) {
    try {
        System.out.println("CREATING DATAGRAM SOCKET");
        // binds socket to any available port choosedd by the kernel on the local machine
        DatagramSocket datagramSocket = new DatagramSocket();

        System.out.println(".... SENDING MESSAGE ...");
        String message = ">>>>  Hello I am an UDP CLIENT <<<<";
        DatagramPacket datagramPacket = new DatagramPacket(message.getBytes(),
                                                  message.getBytes().length,
                                                  InetAddress.getByName(TOMACHINE),
                                                  TOPORT);
        datagramSocket.send(datagramPacket);
        System.out.println(".... MESSAGE SENT.");

        System.out.println(".... WAITING FOR RESPONSE...");
        byte[] response = new byte[40];
        datagramPacket = new DatagramPacket(response, response.length);
        datagramSocket.receive(datagramPacket); // blocks until message is received

        System.out.println(".... MESSAGE RECEIVED: " + new String(response));

        System.out.println("CLOSING SOCKET");
        datagramSocket.close();

        System.out.println("CLIENT FINISHED");

    } catch (SocketException ex) {
        System.out.println("nSOCKET ERROR: " + ex.getMessage());
    }
    catch (IOException ex){
        System.out.println("nI/O ERROR: " + ex.getMessage());
    }

}
```

6. Think over.

Modify previous the UDP client/server programs to include the following:

- Ask always the user the desired Port and IP. Run programs and test they work fine.
- In the server program, change the *message* buffer array size to store only 20 bytes. Execute server and client and watch what happens. Is there any error? Is the message received complete? Why?
- Now, imagine you want to send a message of 80000 bytes length. Modify the client program to fill and send a string with 80000 'E' chars. Modify server to accept this so long message. Run both client and server. Explain what happens.

7. Learning more.

- Search into the Oracle online documentation more information about Socket, ServerSocket, DatagramSocket and DatagramPacket classes.
- Find out what methods might be interesting and useful.
- Write examples that make use of these new methods in your client/server applications.

5. COMMUNICATION MODELS

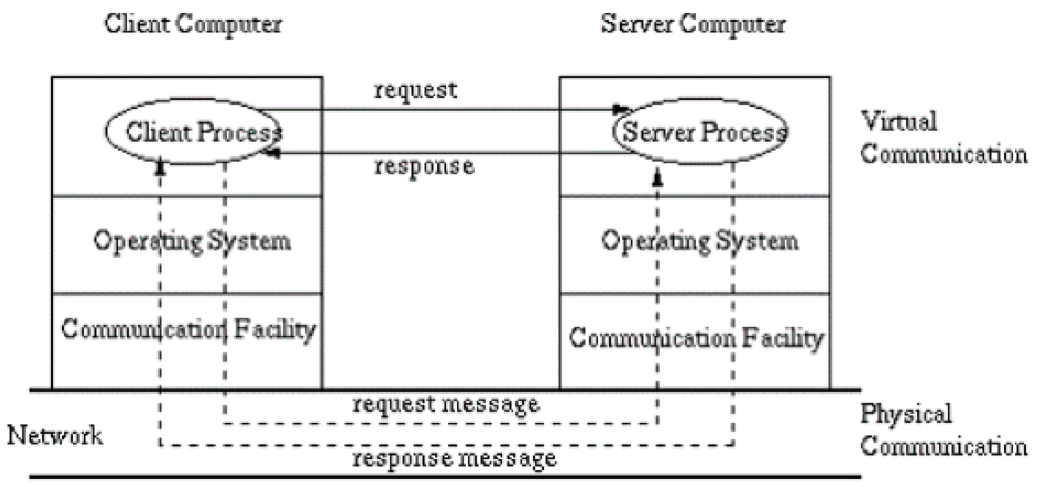
Practically all modern distributed network applications are built on *sockets* and the *IP stack*. However, there are other high-level issues to take into account when developing distributed applications. Therefore, we choose a different ***communication model*** depending on the purpose of the application.

A ***communication model*** specifies how the various elements of a distributed application communicate with each other.

The *client-server* model, the *workgroup* model, and *peer-to-peer* are currently the most used communication models.

5.1 CLIENT-SERVER

Most of the Net Applications use the Client-Server architecture. These terms refer to the two processes (or two applications), which will be communicating with each other to exchange some information. One of the two processes acts as a **client** process and another process acts as a **server**.



The client-server model

Client process:

This is the process that usually makes a request for information. After getting the response this process may terminate or may do some other processing.

> *For example: Internet Browser works as a client application, which sends a request to Web Server to get one HTML web page.*

Server process:

This is the process that takes requests from clients. After getting a request from the client, this process will do the required processing to gather the requested information and will send it to the client. Server process is always alert and ready to serve incoming requests.

> *For example: Web Server keeps waiting for requests from Internet Browsers and as soon as it gets any request from a browser, it picks up a requested HTML page and sends it back to that Browser.*

The client-server model is ***simple*** to implement (using for example TCP or UDP sockets), ***modular***, ***extensible*** and ***flexible***. Modular and extensible mean that the client-server concept is not limited to a single client and a single server - the concept can be extended to a myriad of configurations, allowing for great flexibility in the types of networks that can be built.

8. Activity.

- Make a list of the client-server applications you usually use in your mobile phone/computer.

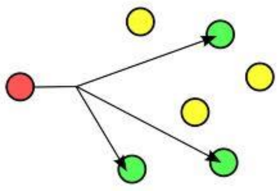
- Choose one of them and describe the communication process.

5.2 GROUP COMMUNICATION

Client-server communication involves two parties: the client and the server. Sometimes, however, communication involves multiple processes, not only two. A solution is to perform message passing operations to each receiver.

With ***group communication*** a message can be sent to multiple receivers in one operation, called ***multicast***.

To receive a multicast message, the receivers' sockets in the group must use the same multicast IP (224.0.0.0 to 239.255.255.255).



For example: Modern on-line games like Call of Duty use group communication mechanisms to transmit information to all players of the same game. As data is replicated, performance and reliability increase, especially when server crashes.

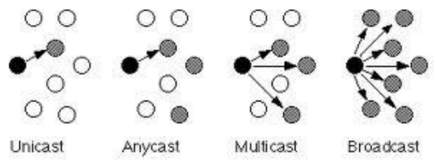
Did you know?

Today, the majority of Internet applications rely on point-to-point transmission. The use of point-to-multipoint transmission has traditionally been limited to local area network applications.

Over the past few years the Internet has seen a rise in the number of new applications that rely on multicast transmission. Multicast IP conserves bandwidth by forcing the network to do packet replication only when necessary, and offers an attractive alternative to unicast transmission for live stock quotes, multiparty video-conferencing, and shared whiteboard applications (among others)

9. Multicast messages.

a. Search Internet for information about the 4 fundamental types of IP addresses and explain the basics of each one:



b. Which IP protocol, TCP or UDP, is used in multicasting?

c. What type of router do we need to transmit multicast messages?

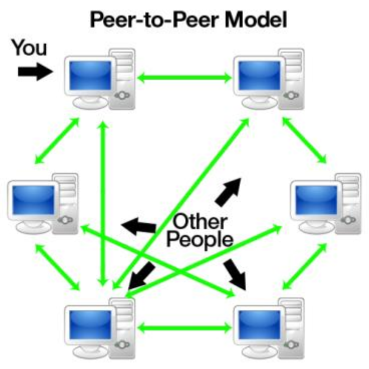
d. How is multicast implemented in Java?

e. Find examples of types of applications that make use of multicasting.

5.3 PEER-TO-PEER

Peer-to-peer (**P2P**) computing or networking is a distributed application architecture that partitions tasks or workloads between peers.

Peers are equally privileged, equipotent participants in the application. They are said to form a **peer-to-peer** network of nodes. Each node is a computer on the network which acts and communicates with other Peers to make a portion of their resources, such as processing power, disk storage or network bandwidth, directly available to other network participants, without the need for central coordination by servers or stable hosts.



The most commonly known application is ***file sharing***, which popularized the technology, like the famous ***Torrents*** or the ***Emule*** Project.

Peers are both *suppliers* and *consumers* of resources, in contrast to the traditional client–server model where the consumption and supply of resources is always divided.

Hybrid models

Hybrid models are a combination of *peer-to-peer* and *client-server* models. A common hybrid model is to have a central server that helps peers find each other. There are a variety of hybrid models; Spotify is an example of a hybrid model.

Currently, hybrid models have better performance than either pure unstructured networks or pure structured networks because certain functions, such as searching, do require a centralized functionality but benefit from the decentralized aggregation of nodes provided by unstructured networks.

10. Learning more.

a. What was the origin of the P2P technology? What was the first peer-to-peer network?

b. Search Internet and find information about the technology behind the famous music service *Spotify*. Describe its main features, pros and cons.

c. Compare *Spotify* to iTunes Radio or another free music service you like. Which one do you think is better? Why?

6. EXERCISES