Sockets

A socket is an abstraction that represents an endpoint for communications between processes across the network. Sockets are used to support to the transportation protocols.

- Sockets can be used for an active communication between two specific computers on a network, where they establish a connection-oriented service. This is called TCP mode.

- Sockets can also be used for connectionless communication. When the socket is used for a connectionless data transmission, the data is sent out, networked with a specified direction, but with no established connection with the intended recipient this is called UDP mode. A socket for UDP mode can also receive data without been connected to a source. The socket just waits for some data to turn up.

Java provides the classes necessary to implement sockets on the machines where your program executes. These classes are defined in the package **java.net**.

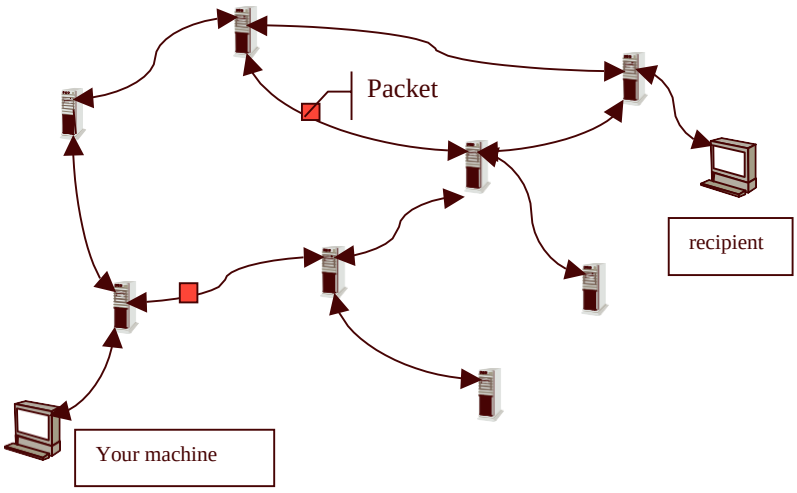
Every socket has two characteristics that identify it, the IP address of the computer the socket resides in, and a port number. There are often several network-based services provided by a single computer. These could include FTP, HTTP, electronic mail and others. Each of the services is assigned a well-known Port, i.e., an address inside the computer that everyone knows and everyone who wishes to use the service connects to the port for that service.

Styles of Communication

There are two types of connection that can be used to be talk to a network. Connection oriented protocols such as the **Transmission Control Protocol (TCP)** and connectionless protocols such as the **User Datagram Protocol (UDP)**

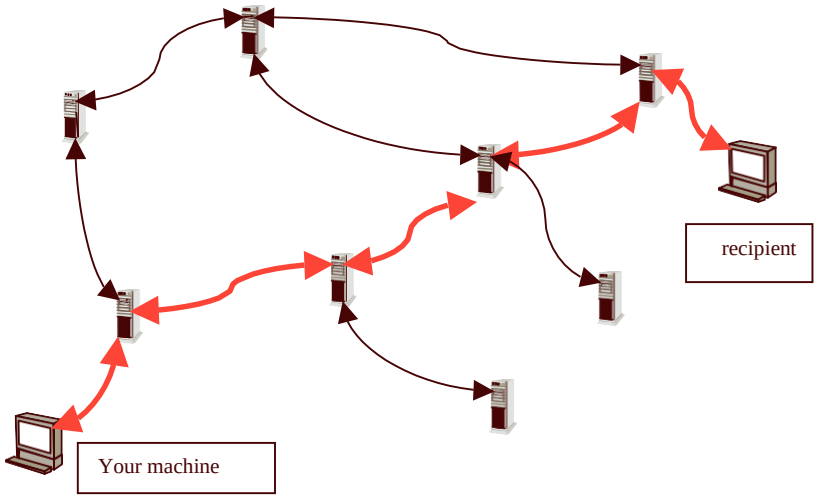
UDP - Dataram like a letter service

There is no active connection to the recipient machine. Your computer can package a piece of data into a packet, put a label on it that identifies the recipient and then send the packet to a machine on the network's designated router. The router will then try to send the packet to another router (preferably the router nearest to the recipient machine that will then hopefully send the data directly to the recipient) Note that when several packets are transmitted each may traverse a different path across the network. There is no guarantee that a packet will actually end up at its destination nor that packets will arrive in the same order as they were transmitted. There is no automatic confirmation to the originator that a packet has arrived at its destination or that the packet has not been corrupted in the transmission process. This type of service is best used for single-shot communications.



TCP – Stream like a telephone service

A connection must first be established. Once this has been successfully achieved a continuous conversation may occur. Packets are guaranteed to be delivered error free and in the correct sequence. The stream socket service allows the conversation between two processes to look like each process is writing to a standard stream. This stream has error checking transparently implemented. All possible network errors are corrected before a process receives any data.

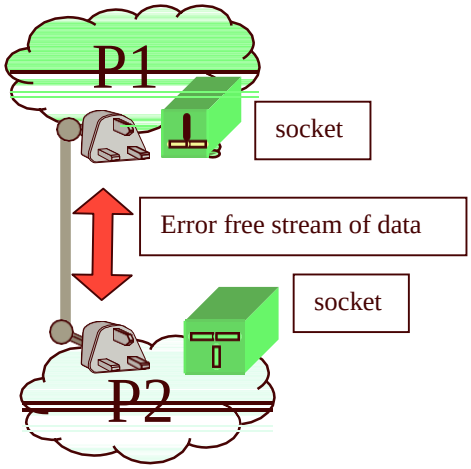


Programming using Stream Sockets

Stream sockets are the endpoints of a communications link between two processes. The streams connecting the two sockets are bi-directional perfect error free communication streams. The lower level TCP protocol performs all error recovery.

Sockets are appropriate for applications that require guaranteed delivery of information.

- Terminal sessions
- Remote login
- File Transfer applications
- Mail applications
- Hypertext applications.



Stream sockets are slower than datagram sockets because each packet from the sender must be confirmed when arrived at the receiver as error free and in the correct sequence, otherwise the sender will re-transmit the packet. There is also an initiation phase where the connection is established so that communication cannot start immediately.

If two processes wish to communicate they must each create a socket. Once created the socket must then be bound to a specific network address. One process must initiate the connection (this process is called the client) and the other process must wait for a connection request (this process is called the server). This delegation of responsibilities is highlighted in the diagrams below.

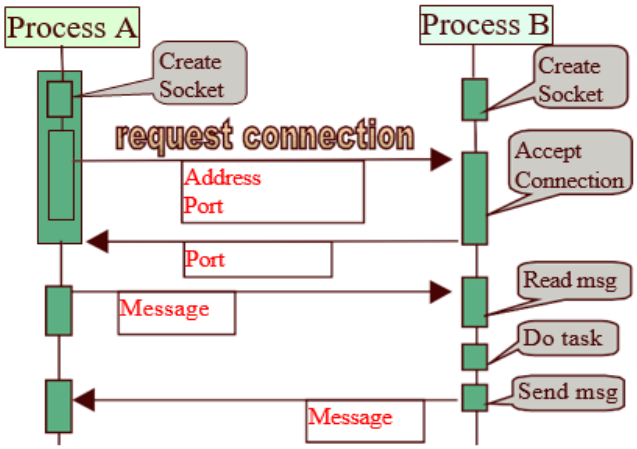
Class Client	Collaborators • Server
Responsibility • Implements a task • Requests connection from other process. • Sends message to other process • Receives messages from other process	

Class Server	Collaborators • Client
Responsibility • Implement a task • Waits for connection from another process. • Receives messages from other process • Sends messages to other process	

The client starts by trying to establish a connection with the listening socket at a server. If the server accepts the communication request, then the connection is established and communication can start. If a connection cannot be established, then the client process must recover from this situation. It can do this either by trying to re-connect to the same server or by choosing to connect to a different server that offers the same services.

Once communication has ended the sockets must be closed.

The client process needs to know the Internet address and the port number of the server process in order to initiate the connection. When the client knows the servers real address it sends a connection request packet containing its own Internet address and port number to the server process.



If the server process accepts the connection-request from the client it creates a new socket bound to a new port number for use by the client and sends this information back to the client as an acknowledgement.

Once the client received a positive acknowledgement from the server, it re-binds its socket to the new port supplied by the server. At this point, a connection has been established. Both process now know the Internet address and port number of the process at the other end of the connection and therefore communication is established.

Programming the Client Side

Java encapsulates all the low-level functionality of binding an address/port to a socket in a class, which is inherited by all Socket classes. The Socket class itself acts as a wrapper to hide all the functionality associated with the establishment of a connection which then happens automatically as a Socket is created.

Therefore, an object of the **Socket** class represents a client socket. This class has several constructors to allow the programmer to create a client socket as follows:

- Specifying the address of the server in an InetAddress structure and a port number

```
InetAddress address = InetAddress.getByName("localhost");
Socket server = new Socket(address,9090)
```

- Specifying the address of the server as a String along with a port number.

```
Socket server = new Socket("localhost",9090);
```

See Java Documentation for others constructor options.

Once a socket is created you can access the Streams associated with the Socket class. Remember that every socket is bi-directional and so it has both an input stream and an output stream. The functions to do this are

- getInputStream()
- getOutputStream()

These streams behave just like any other stream in Java. Therefore, if you want the client to interact with the server using character strings you should wrap the socket output stream in a **DataInputStream** and **DataOutputStream** available from the **java.io** package, respectively. For example:

```
DataInputStream inFromServer = new DataInputStream(server.getInputStream());
String text = InFromServer.readUTF();
```

```
DataOutputStream outToServer = new DataOutputStream(server.getOutputStream());
outToServer.writeUTF("some text");
```

If using binary objects, then you could wrap the input stream in an **ObjectInputStream** and the output stream in an **ObjectOutputStream**.

Programming the Server Side

The server code is a slightly more complicated than the client code. Most servers' use a **ServerSocket** to bound a fixed port number known to the client's initial entry point to the server. However, if a client continued to use that socket then no other processes could contact the server until the first had completed its communication. This is because a socket can only form one half of a single communication link.

Therefore, once a server has accepted a connection on its well-known port it creates another **Socket** (bound to another unused port) for subsequent communication with the client. This releases the socket connected to the well-known port and thus allows other processes to connect to the server. The connection to a socket listening on the servers' well-known port is analogous to the main switchboard of a company whereby callers are diverted onto extension lines in order that new calls may be accepted on the main number.

The constructor of the **ServerSocket** class must specify the port number. A normal **Socket** is returned to the client process as a result of connecting to the initial **ServerSocket**. The server code typically waits infinitely for connections. Therefore, the server side may be written:

```
ServerSocket server = new ServerSocket(9090);

while (true) {
       Socket client = server.accept();

       //some more code
}
```

A client can connect to a **ServerSocket** once its **accept()** function has been called. This function returns the new Socket as its result. A **ServerSocket** cannot accept another connection until the **accept()** function is called again. The server can send messages to the client using the normal **Socket** class in the same way described previously for the client side (e.g., the **DataInputStream** and **DataOutputStream**).

Handling IOExceptions

Note: both **Socket** and **ServerSocket** constructors throw an **IOException** if an error occurs. Therefore they must be enclosed in a **try .. catch** block of code.

Client Side

```
try
{
    Socket server = new
    Socket("localhost",9090);

     //some more code
}
catch (IOException e)
{
    // handle error
}
```

Server Side

```
try
{
    ServerSocket server = new
    ServerSocket(9090);

     //some more code
 }
 catch (IOException e)
 {
    // handle error
 }
```

Alternatively, you can simply add the IOException at the definition of the method and not catch the IOException as follows:

```
public static void main(String[] args) throws IOException {
       //some more code here
}
```

Implement ex8.java Java Program

Write a Java program that consists of two classes: 1) **ex8client.java** and 2) **ex8server.java**. The former class will contain the implementation of the client side in which the user should be prompt to type in a message and send it to the server. The latter class will contain the implementation of the server side that will receive client's message and will convert it from lowercase to uppercase. Then the server will send back to the client the converted message. Use Stream Sockets to implement your program.

Programming using Datagram Sockets

Datagrams can be generated using two java classes also available in the **java.net** package:

- The **DatagramSocket** class
- The **DatagramPacket** class

The DatagramSocket class

This class is a socket used for sending and receiving datagram packets. Since when sending packets, the relative information (e.g., address and port number) are contained in the packets themselves the constructors are not concerned with the destination for the data at all when implementing the client side. You can create a datagram socket object with the default constructor:

```
DatagramSocket client = new DatagramSocket();
```

A socket used to receive a datagram in the server side needs to have a specific fixed port address. Because the machine sending the data includes the receiving port number as part of the packet header information. To create a receiving datagram socket you can use the following constructor which takes an int as parameter, e.g., the port number.

```
DatagramSocket server = new DatagramSocket(portNumber);
```

To send a datagram you call the **send()** method of the DatagramSocket, and pass the DatagramPacket object (which encapsulates the datagram) to be sent as an argument. To receive a datagram packet you call the corresponding **receive()** method of the DatagramSocket. Again both of these methods may generate IOExceptions and therefore must be enclosed in a **try .. catch** block of code.

Client Side

```
try
{
      socket.send(packet);
}
catch (IOException e)
{
      // handle error

}
```

Server Side

```
try
{
      socket.receive(packet);
}
catch (IOException e)
{
      // handle error

}
```

Alternatively, you can simply add the IOException at the definition of the method and not catch the IOException as follows:

```
public static void main(String[] args) throws IOException {
       //some more code here
}
```

The **receive()** method blocks until a packet is actually received. Once you are finished with a DatagramSocket you must close it using the **close()** method.

The DatagramPacket Class

The DatagramPacket class encapsulates a datagram packet. A datagram packet is an uninterpreted sequence of bytes therefore a byte array must be created containing the information you wish to send.

There are two DatagramPacket constructors:

- one for packets that you receive which must specify two pieces of data: 1) a reference to a data buffer which is a byte array and the length. Note that the array is initially empty and the data from the received data will be stored in. The length of the packet which at maximum must be the length of the byte array (typically is set to1024)

```
byte[] data = new byte[length];
DatagramPacket receivedPacket = new DatagramPacket(data,length);
```

- one for packets that you intend to send which in addition to the data and length of the byte array must also specify the destination address and portNumber. Note that the byte array, i.e., data, should contain the data and should not be empty as above.

.

```
DatagramPacket sendPacket = new DatagramPacket(data,       //byte[]
                                               length,     //int
                                               destination,//InetAddress
                                               portNumber) //int
```

As it is common in Java this class contains already implemented methods, e.g., **getData()** that can be useful. Refer to the Java Documentation for more information.

To obtain the data held in a packet call the **getData()** function.

```
byte data = packet.getData();
```

You can also get and set methods for the Internet address and the port number using **getAddress()** and **getPort()**, respectively, or **setAddress(InetAddress a)** and **setPort(int p)**, respectively.

Data Conversion from byte arrays to other data types.

The simplest conversion is between the String class and byte arrays because:

- The String class has a built-in constructor to enable Strings to be created from byte arrays
- It has a function to convert Strings to byte arrays, e.g., **getBytes()**

In Java the base class Object has a function **toString()** (which is overridden in all the standard Java classes) to return a String representation of the object. All user-defined classes should also define this function. This then enables the programmer to transmit a packet containing the String representation of the object as follows:

- Get String representation of object
- Convert String to byte array
- Store in DatagramPacket
- Send packet (**client** is the instance of the DatagramSocket)

```
String message = "Some message";
byte[] data = message.getBytes();
InetAddress addr = InetAddress.getByName("localhost");
DatagramPacket sendPacket = new DatagramPacket(data, data.length, addr, 9090);
client.send(sendPacket);
```

Of course, the receiving program must then do the following:

- Create an empty byte array
- Receive the packet (**server** is the instance of the DatagramSocket)
- Fill the byte array with the data of the packet
- Convert packet to String. Parse String representation and create new object this may be achieved by adding an object constructor whose parameter is a byte array. This constructor converts the byte array into a String and then calls the constructor which parses the String. In this way, all conversion is done internally inside the object.

```
byte[] data = new byte[1024];
DatagramPacket receivedPacket = new DatagramPacket(data, data.length);
server.receive(receivedPacket);
data = receivedPacket.getData();
String message = new String(data);
```

This would be the recommended procedure for simple datagram packet structures.

Implement ex9.java Java Program

Write a Java program that consists of two classes: 1) **ex9client.java** and 2) **ex9server.java**. The former class will contain the implementation of the client side in which the user should be prompt to type in a message and send it to the server. The latter class will contain the implementation of the server side that will receive client's message and will convert it from lowercase to uppercase. Then the server will send back to the client the converted message. Use Datagram Sockets to implement your program.