Name	: Md. Nazmul Hasan
ID	: IT-17005
Lab No	: 02
Name of the Lab	: Socket Programming Lab

Objectives:

i) Learn Socket programming.
ii) A UDP datagram.
iii) Learn about a datagram socket and stream socket.

Socket Programming Lab

1. Briefly explain the term IPC in terms of TCP/IP communication.

Answer: In computer science, inter-process communication or inter-process communication (IPC) refers specifically to the mechanisms an operating system provides to allow the processes to manage shared data. Typically, applications can use IPC, categorized as clients and servers, where the client requests data and the server responds to client requests. When used in conjunction with IP, they are referred to as TCP/IP and UDP/IP. The transport protocol is usually responsible, among other things, for providing more granular addressing (targeting a specific process on a machine), data integrity (in the case of TCP, not so in UDP)... In essence, it provides transparent transfer of data between end points.

2. What is the maximum size of a UDP datagram? What are the implications of using a packet-based protocol as opposed to a stream protocol for transfer of large files.

Answer: This field specifies the length in bytes of the UDP header and UDP data. The minimum length is 8 bytes, the length of the header. The field size sets a theoretical limit of 65,535 bytes (8 byte header + 65,527 bytes of data) for a UDP datagram. However the actual limit for the data length, which is imposed by the underlying IPv4 protocol, is 65,507 bytes (65,535 − 8 byte UDP header − 20 byte IP header).

In telecommunications, packet switching is a method of grouping data that is transmitted over a digital network into packets. Packets are made of a header and a payload. Data in the header is used by networking hardware to direct the packet to its destination, where the payload is extracted and used by application software. Packet switching is the primary basis for data communications in computer networks worldwide.

The routing and transferring of data by means of addressed packets so that a channel is occupied during the transmission of the packet only, and upon completion of the transmission the channel is made available for the transfer of other traffic.

Just like other data that's sent over the Internet, audio and video data is broken down into data packets. Each packet contains a small piece of the file, and an audio or video player in the browser on the client device takes the flow of data packets and interprets them as video or audio.

Sending video over the Internet, as opposed to sending text and still images, requires a faster method of transporting data than TCP/IP, which prioritizes reliability over speed. The above reasons are the implications of using a packet-based protocol as opposed to a stream protocol for transfer of large files.

3. TCP is a reliable transport protocol, briefly explain. What techniques are used to provide this reliability?

Answer: The Transmission Control Protocol (TCP) is one of the main protocols of the Internet protocol suite. It originated in the initial network implementation in which it complemented the Internet Protocol (IP). Major internet applications such as the World Wide Web, email, remote administration, and file transfer rely on TCP, which is part of the Transport Layer of the TCP/IP suite. SSL/TLS often runs on top of TCP.
TCP is connection-oriented, and a connection between client and server is established before data can be sent. TCP is a reliable stream delivery service which guarantees that all bytes received will be identical and in the same order as those sent. Since packet transfer by many networks is not reliable, TCP achieves this using a technique known as positive acknowledgement with re-transmission. This requires the receiver to respond with an acknowledgement message as it receives the data. The sender keeps a record of each packet it sends and maintains a timer from when the packet was sent. The sender re-transmits a packet if the timer expires before receiving the acknowledgement. The timer is needed in case a packet gets lost or corrupted.

4. Why are the htons(), htonl(), ntohs(), ntohl() functions used?

Answer: Different computers use different byte orderings internally for their multi-byte integers (i.e. any integer that's larger than a char.) The upshot of this is that if you send() a two-byte short int from an Intel box to a Mac (before they became Intel boxes, too, I mean), what one computer thinks is the number 1, the other will think is the number 256, and vice-versa.
The way to get around this problem is for everyone to put aside their differences and agree that Motorola and IBM had it right, and Intel did it the weird way, and so we all convert our byte orderings to "big-endian" before sending them out. So these functions convert from your native byte order to network byte order and back again.
Anyway, the way these functions work is that you first decide if you're converting from host (your machine's) byte order or from network byte order. If "host", the first letter of the function you're going to call is "h". Otherwise it's "n" for "network". The middle of the function name is always "to" because you're converting from one "to" another, and the penultimate letter shows what you're converting to. The last letter is the size of the data, "s" for short, or "l" for long. Thus:
htons( ) - host to network short
htonl( ) - host to network long
ntohs( ) - network to host short
ntohl( ) - network to host long

5. What is the difference between a datagram socket and a stream socket? Which transport protocols do they correspond to?

Answer: The difference between SOCK_STREAM and SOCK_DGRAM is in the semantics of consuming data out of the socket.

Stream socket allows for reading arbitrary number of bytes, but still preserving byte sequence. In other words, a sender might write 4K of data to the socket, and the receiver can consume that data byte by byte. The other way around is true too - sender can write several small messages to the socket that the receiver can consume in one read. Stream socket does not preserve message boundaries.

Datagram socket, on the other hand, does preserve these boundaries - one write by the sender always corresponds to one read by the receiver (even if receiver's buffer given to read(2) or recv(2) is smaller then that message).

So if your application protocol has small messages with known upper bound on message size you are better off with SOCK_DGRAM since that's easier to manage.

If your protocol calls for arbitrary long message payloads, or is just an unstructured stream (like raw audio or something), then pick SOCK_STREAM and do the required buffering.

Performance should be the same since both types just go through local in-kernel memory, just the buffer management is different.

2.2.2 Exercise- 2(Time protocol): Your task in this exercise is to implement the Time protocol, as defined in RFC868 (http://www.faqs.org/ rfcs/rfc868.html). You should write both a UDP and TCP server, as well as clients that access the services. The client should display the result back to the user in a "nice" way. This means that you should not just dump the raw 32-bit data you get from the server on the command line, but present it in a humanly readable fashion. The deliverables should be called **TimeUdpClient.java**, **TimeUdpServer.java**, **TimeTcpClient.java** and **TimeTcpServer.java**, if you have chosen "Java" as a language for the exercises. Otherwise, use the appropriate file extension.

Source Code:

*1. **TimeTcpClient.java** ->*

```
import java.io.*;
import java.net.*;
public class TimeTcpClient {
public static void main(String[] args) {
try{
Socket socket = new Socket("localhost",6666);
DataOutputStream dout=new DataOutputStream(socket.getOutputStream());
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
while(true){
   System.out.println("Say something to server: ");
   String s = br.readLine();
   dout.writeUTF(s);
   if(s.equalsIgnoreCase("exit"))
      break;
}
socket.close();
}catch(Exception e){System.out.println(e);}
```

```
}
}
```

Console Output:

```
run:
Say something to server:
Hi! My name is Ruhan
Say something to server:
```

---

2. TimeTcpServer.java ->

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

try{
ServerSocket ss=new ServerSocket(6666);
Socket s=ss.accept();//establishes connection
System.out.println("Client is connected with the server.");
DataInputStream dis=new DataInputStream(s.getInputStream());
while(true){
   String  str=(String)dis.readUTF();
   System.out.println("Message = "+str);
   if(str.equalsIgnoreCase("exit"))
      break;
}
ss.close();
}catch(Exception e){System.out.println(e);}
}
}
```

Console Output:

```
run:
Client is connected with the server.
Message = Hi! My name is Ruhan
```

---

3. TimeUdpClient.java ->

```
import java.net.*;
public class TimeUdpClient{
  public static void main(String[] args) throws Exception {
    DatagramSocket ds = new DatagramSocket();
    String str = "Hi! This is Ruhan.";
    InetAddress ip = InetAddress.getByName("127.0.0.1");
    DatagramPacket dp = new DatagramPacket(str.getBytes(), str.length(), ip, 3000);
    ds.send(dp);
      System.out.println("The message has been succesfully sent.");
```

```
    ds.close();
  }
}
```

Console Output:

```
run:
The message has been succesfully sent.
BUILD SUCCESSFUL (total time: 0 seconds)
```

***4. TimeUdpServer.java* ->**

```
import java.net.*;
public class TimeUdpServer{
  public static void main(String[] args) throws Exception {
    DatagramSocket ds = new DatagramSocket(3000);
    byte[] buf = new byte[1024];
    DatagramPacket dp = new DatagramPacket(buf, 1024);
    ds.receive(dp);
    String str = new String(dp.getData(), 0, dp.getLength());
    System.out.println(str);
    ds.close();
  }
}
```

Console Output:

```
run:
Hi! This is Ruhan.
BUILD SUCCESSFUL (total time: 5 seconds)
```