2020/2021

Mini-Project

Submitted in fulfillment of the requirements for the

Electrical and Communication Engineering Course FROM THE LEBANESE UNIVERSITY

FACULTY OF ENGINEERING – BRANCH III

Prepared By:

Ahmad Srour	**5352**
Ahmad Said	**5613**

TCP Chat Application

Supervised by:

Dr. Mohamad Aoude

Table of Contents

ABSTRACT

In today's world computer has become an integral part of the business sector for professional activities as well as for personal activities. As technologies have, evolved networking appeared and slowly from initial wired network technology, we moved to the wireless network technology. Now if we will think then we can know that networking affects everything.

Computer Network is an interconnection between computers or we can say computer network is group of computers linked to each other, which enables one computer to communicate with another computer. It acts as basis of communication in Information Technology (IT). It is system of connected computing devices and shares information and resources between them. The devices in network are connected by communication links (wired/wireless) and share data by Data Communication System.

Most of the connection on the internet use TCP protocol, in the project we will explain and implement a simple 'server-client' application, based on TCP socket using JAVA. The same analogy can be used to create any other types of network applications like FTP or HTTP.

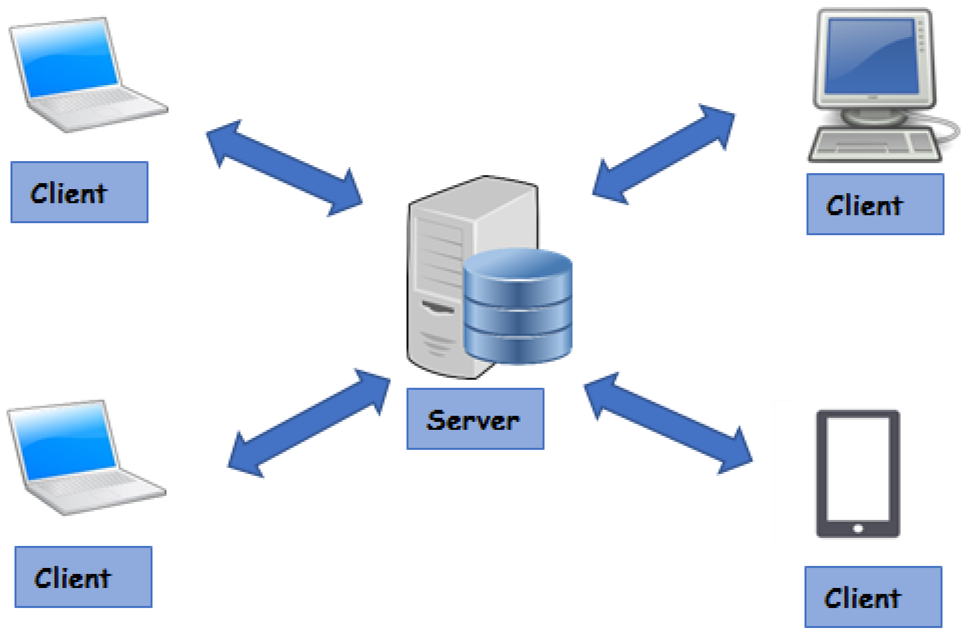
Network Programming

Network Programming involves writing programs that communicate with other programs across a computer network.

There are many issues that arise when doing network programming which do not appear when doing single program applications. However, JAVA makes networking applications simple due to the easy-to-use libraries. In general, applications that have components running on different machines are known as distributed applications, and usually they consist of client/server relationships.

Server-Client Model

A server is an application that provides a "service" to various clients who request the service.



There are many client/server scenarios in real life:

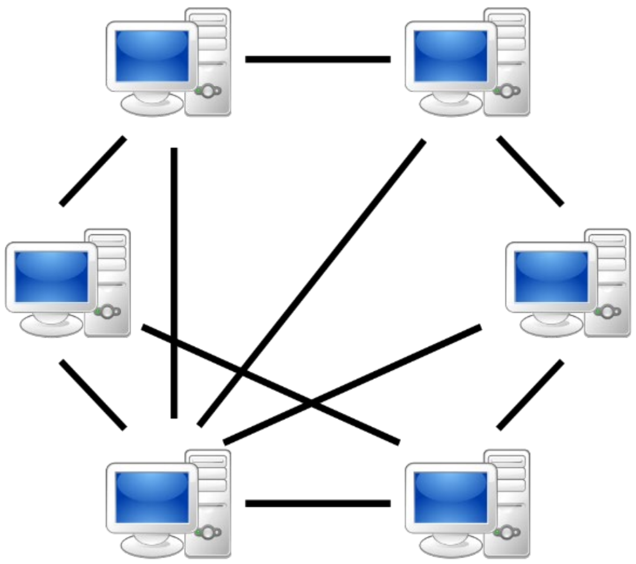
- Bank tellers (server) provide a service for the account owners (client)
- Waitresses (server) provide a service for customers (client)
- Travel agents (server) provide a service for people wishing to go on vacation (client)

In some cases, servers themselves may become clients at various times.

E.g., travel agents will become clients when they phone the airline to make a reservation or contact a hotel to book a room.

Peer-to-Peer Model

In the general networking scenario, everybody can be either a client or a server at any time. This is known as peer-to-peer computing. In terms of writing java applications, it is similar to having many applications communicating among one another. A famous application of peer-to-peer network is torrent.



TCP vs UDP

The Transmission Control Protocol (TCP) is a transport protocol that is used on top of IP to ensure reliable transmission of packets. TCP includes mechanisms to solve many of the problems that arise from packet-based messaging, such as lost packets, out of order packets, duplicate packets, and corrupted packets.

The User Datagram Protocol (UDP) is a lightweight data transport protocol that works on top of IP. UDP provides a mechanism to detect corrupt data in packets, but it does not attempt to solve other problems that arise with packets, such as lost or out of order packets.

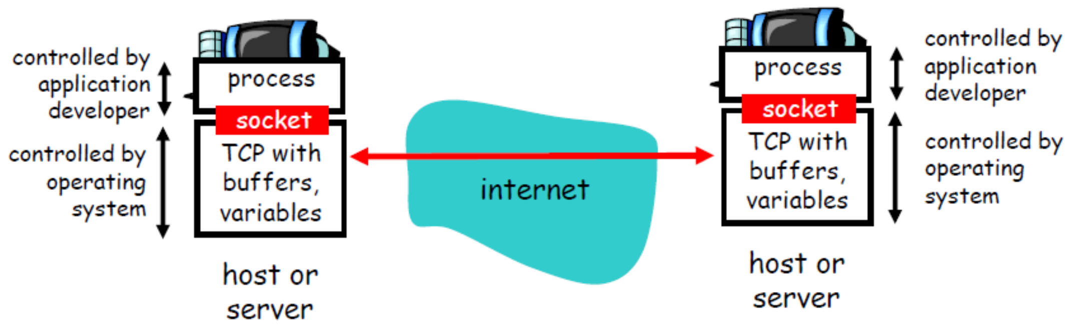
Feature	TCP	UDP
Connection status	Requires an established connection to transmit data (connection should be closed once transmission is complete)	Connectionless protocol with no requirements for opening, maintaining, or terminating a connection
Data sequencing	Able to sequence	Unable to sequence
Guaranteed delivery	Can guarantee delivery of data to the destination router	Cannot guarantee delivery of data to the destination
Retransmission of data	Retransmission of lost packets is possible	No retransmission of lost packets
Error checking	Extensive error checking and acknowledgment of data	Basic error checking mechanism using checksums
Method of transfer	Data is read as a byte stream; messages are transmitted to segment boundaries	UDP packets with defined boundaries; sent individually and checked for integrity on arrival
Speed	Slower than UDP	Faster than TCP
Broadcasting	Does not support Broadcasting	Does support Broadcasting
Optimal use	Used by HTTPS, HTTP, SMTP, POP, FTP, etc	Video conferencing, streaming, DNS, VoIP, etc

Network Socket

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 to. An endpoint is a combination of an IP address and a port number.

The core of a network application consists of a pair of programs – a client program and a server program. When these programs are executed, a client and a server process are created, and these two processes communicate with each other by reading from and writing to sockets.


controlled by application developer
controlled by operating system
process
socket
TCP with buffers, variables
host or server
internet
process
socket
TCP with buffers, variables
host or server
controlled by application developer
controlled by operating system

General workflow

- Server side:
  - server process must first be running
  - server must have created socket (door) that welcomes client's contact
- Client side:
  - creating client-local TCP socket
  - specifying IP address, port number of server process
  - When client creates socket: client TCP establishes, connection to server TCP
  - When contacted by client, server TCP creates new socket for server process to communicate with client
  - allows server to talk with multiple clients

Server Sockets

In Java, the basic life cycle of a server program is:

1. A new ServerSocket is created on a particular port using a ServerSocket( ) constructor.
2. The ServerSocket listens for incoming connection attempts on that port using its accept( ) method. accept( ) blocks until a client attempts to make a connection, at which point accept( ) returns a Socket object connecting the client and the server.
3. Depending on the type of server, either the Socket's getInputStream( ) method, getOutputStream( ) method, or both are called to get input and output streams that communicate with the client.
4. The server and the client interact according to an agreed-upon protocol until it is time to close the connection.
5. The server, the client, or both close the connection.
6. The server returns to step 2 and waits for the next connection.

1. Creation

The ServerSocket constructor requires a port number (1024–65535, for non-reserved ones) as an argument. For example:

```
ServerSocketserverSocket= new ServerSocket(1234);
```

2. Waiting State

We put the server into a waiting state for any client connection. For example:

```
Socket s = serverSock.accept();
```

The returned Socket object is called "service socket" and represents the connection established (as on the client side).

3. Set up input/output stream

Methods *getInputStream()* and *getOutputStream()* of class Socket are used to get references to streams associated with the socket returned in step 2.

These streams will be used for communication with the client that has just made connection. For a non-GUI application, we can wrap a Scanner object around the lnputStream object returned by method getlnputStream, in order to obtain string-orientated input just as we would do with input from the standard input stream, System.in). For example:

```
Scanner input= new Scanner(l ink.getlnputStream());
```

Similarly, we can wrap a *PrintWriter* object around the *OutputStream* object returned by method *getOutputStream()* . Supplying the *PrintWriter* constructor with a second argument of true will cause the output buffer to be flushed for every call of *println* (which is usually desirable). For example:

```
PrintWriter output = new PrintWriter(link.getOutputStream(),true);
```

4. Send and receive data

Having set up our *Scanner* and *PrintWriter* objects, sending and receiving data is very straightforward. We simply use method *nextLine*() for receiving data and method *println()* for sending data, just as we might do for console I/O. For example:

```
output.println("Awaiting data…");

String input = input.nextLine();
```

Client Socket

1. Establish a connection to the server.

We create a Socket object, supplying its constructor with the following two arguments:

- ❖ the server's IP address (of type InetAddress);
- ❖ the appropriate port number for the service.

A local port is chosen to locally attach the socket, The three way handshake is negotiated with the server. The socket representing the established connection is returned. For simplicity's sake, we shall place client and server on the same host, which will allow us to retrieve the IP address by calling static method getLocalHostof class InetAddress. For example:

```
Socket link = new Socket(InetAddress.getLocalHost(),1234);
```

2. Set up input/output stream

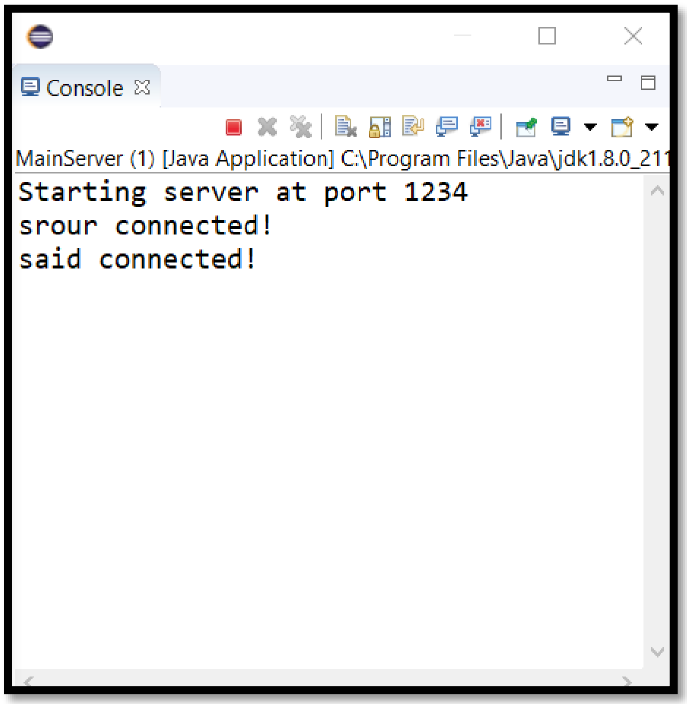
These are set up in exactly the same way as the server streams were set up (by calling methods *getInputStream()* and *getOutputStream()* of the Socket object that was created in step 2).

3. Send and receive data

The Scanner object at the client end will receive messages sent by the PrintWriter object at the server end, while the PrintWriter object at the client end will send messages that are received by the Scanner object at the server end (using methods nextLine and println respectively).

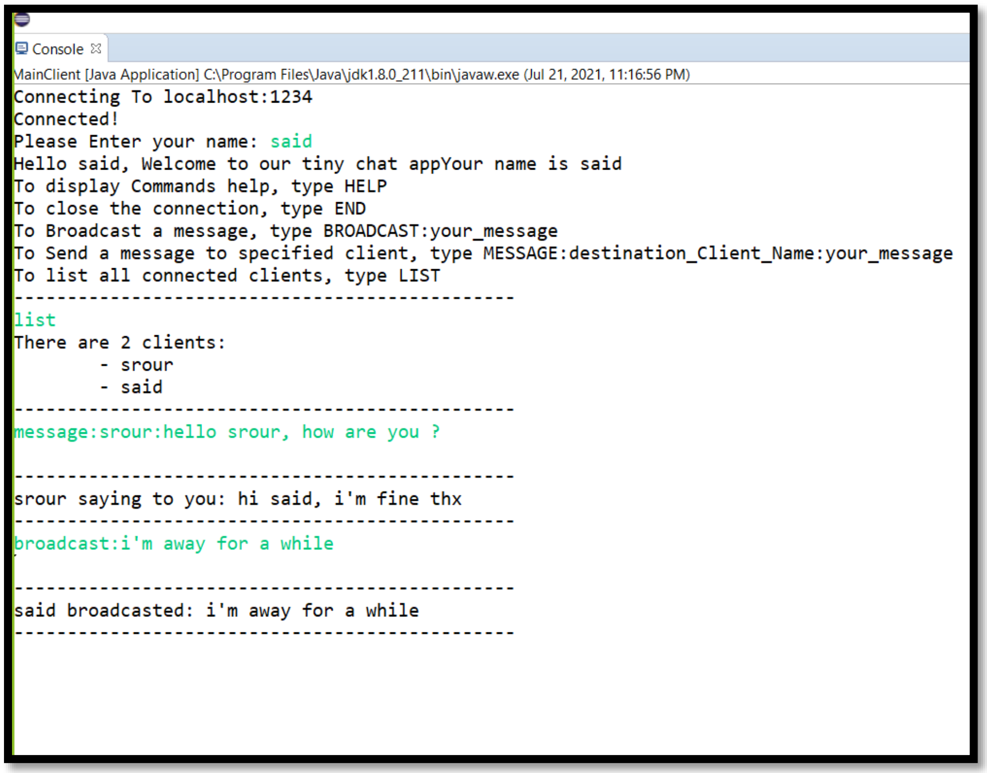
Sketch of the program

Server Console



```
MainServer (1) [Java Application] C:\Program Files\Java\jdk1.8.0_211
Starting server at port 1234
srour connected!
said connected!
```

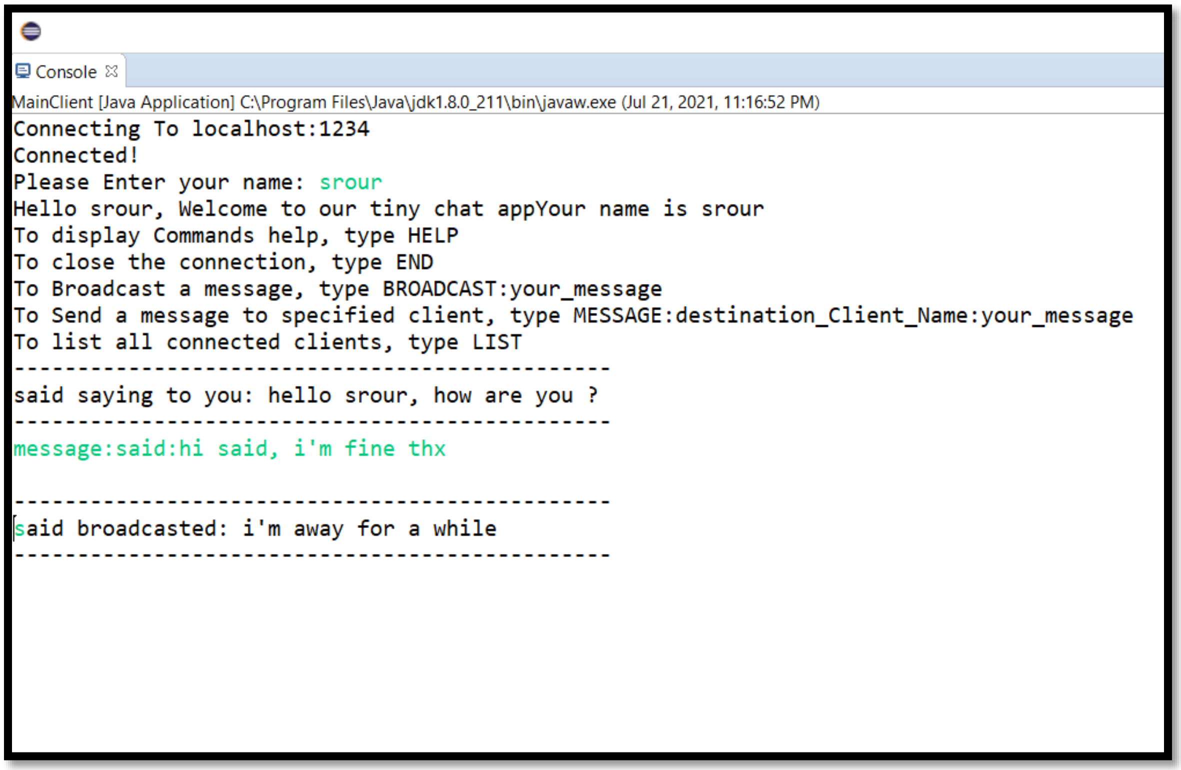
Clients connections



```
MainClient [Java Application] C:\Program Files\Java\jdk1.8.0_211\bin\javaw.exe (Jul 21, 2021, 11:16:56 PM)
Connecting To localhost:1234
Connected!
Please Enter your name: said
Hello said, Welcome to our tiny chat appYour name is said
To display Commands help, type HELP
To close the connection, type END
To Broadcast a message, type BROADCAST:your_message
To Send a message to specified client, type MESSAGE:destination_Client_Name:your_message
To list all connected clients, type LIST
-----------------------------------------------
list
There are 2 clients:
        - srour
        - said
-----------------------------------------------
message:srour:hello srour, how are you ?

-----------------------------------------------
srour saying to you: hi said, i'm fine thx
-----------------------------------------------
broadcast:i'm away for a while

-----------------------------------------------
said broadcasted: i'm away for a while
-----------------------------------------------
```


Console
MainClient [Java Application] C:\Program Files\Java\jdk1.8.0_211\bin\javaw.exe (Jul 21, 2021, 11:16:52 PM)

```
Connecting To localhost:1234
Connected!
Please Enter your name: srour
Hello srour, Welcome to our tiny chat appYour name is srour
To display Commands help, type HELP
To close the connection, type END
To Broadcast a message, type BROADCAST:your_message
To Send a message to specified client, type MESSAGE:destination_Client_Name:your_message
To list all connected clients, type LIST
-----------------------------------------------
said saying to you: hello srour, how are you ?
-----------------------------------------------
message:said:hi said, i'm fine thx

-----------------------------------------------
said broadcasted: i'm away for a while
-----------------------------------------------
```

Conclusion

TCP used for organizing data in a way that ensures the secure transmission between the server and client. It is the most common protocol in networks that use the Internet Protocol (IP). Unlike UDP, TCP provides reliable message delivery, therefore it is suitable for our chat application. Future work would be implementing a chat application to send files like photos or videos…