

CSC3002F Networks Assignment 2019

Socket programming project

Department of Computer Science
University of Cape Town, South Africa

February 20, 2019

This assignment is on networked applications, where you are required to develop a client-server chat application in groups of three students. This document describes the context/requirements (Section 1), what to submit (Section 2), and some basic information about socket programming (Section A).

1. Chat App

There are a range of chat applications with a variety of features. Some are intended for group communication, such as the IRC, whereas others, such as Facebook Chat, are typically designed for a 1:1 private chat. A few others, such as Skype and WhatsApp, are typically used for both 1:1 and group chat. There are also others that are ideal for broadcast messaging in group communication (Vula chat is one example of this). All these ‘chat apps’ are supported by different protocols and can be distinguished based on their features and interaction interfaces.

1.1. Application Design

In this assignment, you are required to design and implement a client-server chat application that makes use of TCP sockets. You do need to implement an application layer protocol to support the client-server architecture, as well as implement a server that can manage the interaction between clients. You should implement a chat client that allows one or more *pairs of users* to exchange messages in real-time. As this is a network application, the different clients and the server should be able to run on different hosts. The client interface can be in terminal, with appropriate user menus. A GUI will be nice to have, but is not a requirement.

An important step in protocol design involves defining the framework of communication for the intended application. This entails specifying requirements and constraints such as: whether real-time interaction is expected; reliability (ie if we need

to verify/check that every message is delivered correctly); bandwidth or connectivity limitations (eg whether we assume limited/expensive bandwidth or not); and authentication/confidentiality issues, among others. One communication constraint in many developing countries is bandwidth. In this assignment, you are to design a chat application that takes into consideration at least two constraints; limited bandwidth, and any other constraint of your choosing. With respect to bandwidth, you need to implement the application such that a recipient has the option of accepting or declining the receiving of a file.

1.2. Protocol Design and Specification

The protocol design includes specification for the pattern of communication, message structure, as well communications rules. In this assignment, the pattern of communication will be client-server-based, meaning that a server will be responsible for the overall control and coordination of communication between parties. The pattern of communication also specifies if transmission mode for messages is unicast, multicast, or broadcast. Note that each of the transmission modes may be used in different aspects of the protocol.

Messages in an application protocol could be either text, consisting of readable character strings (e.g. 'connection-request'), or binary format, where messages are blocks of structured binary data (eg. '1000001' could be used to mean 'connection-request'). Text-based protocols have the advantage of being human readable, hence provide for easier understanding, monitoring and testing. You are required to use text-based messaging in this application protocol design.

Specification of protocol messages involves defining the types and structure of messages. Three types of messages can be defined; commands, data transfer, and control. Command messages define the different stages of communication between parties, such as the initiation or termination of communication. Data transfer messages are used to carry the data that is exchanged between parties, and such data could be fragmented into several messages. Control messages manage the dialogue between parties, including such aspects as message acknowledgements, and retransmission requests.

The message structure constitutes at least the header and body. The header, whose structure must be known to the receiver, may contain fields that describe the actual data in the message. Some of the fields/information contained in the header might include the message type, the command, recipient information, and sequence information. The header generally has a fixed size and contains clues that should help the receiver to understand the rest of the message.

The last aspect of the protocol design will be the communication rules that specify the sequence of messages at every stage of communication. This requires clearly specifying messages and reactions for every communication scenario. You will need to represent such rules with sequence diagrams.

2. What you need to submit

You will be required to submit the following:

1. Your code with proper inline documentation (comments)
2. A report (4 pages maximum) on the design and functionality of your chat application.
3. In addition, the report needs to include:
 - a) A list of features with an explanation for their inclusion
 - b) A protocol specification, detailing the message formats and structure. You are required to include sequence diagram(s).
 - c) Screenshots of the chat application revealing its features. To personalise the screenshot(s), you have to mention all three student numbers of your group in a chat of which you take the screenshot.
4. Oral presentation to be scheduled with the TAs and Tutors (oral to be done on day after submission deadline)

A. Multi-threaded Client/Server ApplicationsSockets Programming in Java

A.1. What is a socket?

A socket is the one end-point of a two-way communication link between two programs running over the network. Running over the network means that the programs run on different computers, usually referred as the local and the remote computers. However one can run the two programs on the same computer. Such communicating programs constitutes a client/server application. The server implements a dedicated logic, called **service**. The clients connect to the server to get served, for example, to obtain some data or to ask for the computation of some data. Different client/server applications implement different kind of services.

To distinguish different services, a numbering convention was proposed. This convention uses integer numbers, called port numbers, to denote the services. A server implementing a service assigns a specific port number to the entry point of the service. There are no specific physical entry points for the services in a computer. The port numbers for services are stored in configuration files and are used by the computer software to create network connections.

A socket is a complex data structure that contains an internet address and a port number. A socket, however, is referenced by its descriptor, like a file which is referenced by a file descriptor. That is why, the sockets are accessed via an application programming interface (API) similar to the file input/output API. This makes the programming of network applications very simple. The two-way communication link between the two

programs running on different computers is done by reading from and writing to the sockets created on these computers. The data read from a socket is the data wrote into the other socket of the link. And reciprocally, the the data wrote into a socket in the data read from the other socket of the link. These two sockets are created and linked during the connection creation phase. The link between two sockets is like a pipe that is implemented using a stack of protocols. This linking of the sockets involves that internally a socket has a much more complex data structure, or more precisely, a collaboration of data structures. Thus, a socket data structure is more than just an internet address and a port number. You have to imagine a socket as a data structure that contains at least the internet address and the port number on the local computer, and the internet address and the port number on the remote computer.

A.2. How is a network connection created?

A network connection is initiated by a client program when it creates a socket for the communication with the server. To create the socket in Java, the client calls the `Socket` constructor and passes the server address and the the specific server port number to it. At this stage the server must be started on the machine having the specified address and listening for connections on its specific port number.

The server uses a specific port dedicated only to listening for connection requests from clients. It can not use this specific port for data communication with the clients because the server must be able to accept the client connection at any instant. So, its specific port is dedicated only to listening for new connection requests. The server side socket associated with specific port is called server socket. When a connection request arrives on this socket from the client side, the client and the server establish a connection. This connection is established as follows:

1. When the server receives a connection request on its specific server port, it creates a new socket for it and binds a port number to it.
2. It sends the new port number to the client to inform it that the connection is established.
3. The server goes on now by listening on two ports:
 - it waits for new incoming connection requests on its specific port, and
 - it reads and writes messages on established connection (on new port) with the accepted client.

The server communicates with the client by reading from and writing to the new port. If other connection requests arrive, the server accepts them in the similar way creating a new port for each new connection. Thus, at any instant, the server must be able to communicate simultaneously with many clients and to wait on the same time for incoming requests on its specific server port. The communication with each client is done via the sockets created for each communication.

The `java.net` package in the Java development environment provides the class `Socket` that implements the client side and the class `serverSocket` class that implements the server side sockets. The client and the server must agree on a protocol. They must agree on the language of the information transferred back and forth through the socket. There are two communication protocols:

- stream communication protocol
- datagram communication protocol

The stream communication protocol is known as TCP (transfer control protocol). TCP is a connection-oriented protocol. In order to communicate over the TCP protocol, a connection must first be established between two sockets. While one of the sockets listens for a connection request (server), the other asks for a connection (client). Once the two sockets are connected, they can be used to transmit and/or to receive data. When we say two sockets are connected we mean the fact that the server accepted a connection. As it was explained above the server creates a new local socket for the new connection. The process of the new local socket creation, however, is transparent for the client.

The datagram communication protocol, known as UDP (user datagram protocol), is a connectionless protocol. No connection is established before sending the data. The data are sent in a packet called datagram. The datagram is sent like a request for establishing a connection. However, the datagram contains not only the addresses, it contains the user data also. Once it arrives to the destination the user data are read by the remote application and no connection is established. This protocol requires that each time a datagram is sent, the local socket and the remote socket addresses must also be sent in the datagram. These addresses are sent in each datagram.

The **java.net** package in the Java development environment provides the class `DatagramSocket` for programming datagram communications. UDP is an unreliable protocol. There is no guarantee that the datagrams will be delivered in a good order to the destination socket. For, example, a long text, split in several pages and sent one page per datagram, can be received in a different page order. On the other side, TCP is a reliable protocol. TCP guarantee that the pages will be received in the order in which they are sent. When programming TCP and UDP based applications in Java, different types of sockets are used. These sockets are implemented in different classes. The classes `ServerSocket` and `Socket` implement TCP based sockets and the class `DatagramSocket` implements UDP based sockets as follows:

- Stream socket to listen for client requests (TCP): the class `ServerSocket`. Stream socket (TCP): the class `Socket`.
- Datagram socket (UDP): the class `DatagramSocket`.

This document shows how to program TCP based client/server applications. The UDP oriented programming is not covered in document.

A.2.1. Opening a socket

The client side When programming a client, a socket must be opened like below:

```
Socket MyClient ;  
MyClient = new Socket("MachineName", PortNumber);
```

This code, however, must be put in a try/catch block to catch the IOException:

```
Socket MyClient ;  
try {  
    MyClient = new Socket("MachineName", PortNumber);  
}  
catch ( IOException e ) {  
    System . out . p r i n t l n ( e ) ;  
}
```

where:

- **MachineName** is the machine name to open a connection to and
- **PortNumber** is the port number on which the server to connect to is listening.

When selecting a port number, one has to keep in mind that the port numbers in the range from 0 to 1023 are reserved for standard services, such as email, FTP, HTTP, etc. For our service (the chat server) the port number should be chosen greater than 1023.

The server side: When programming a server, a server socket must be created first, like below:

```
ServerSocket MyService ;  
try {  
    MyServerice = new ServerSocket(PortNumber);  
}  
catch ( IOException e ) {  
    System . out . p r i n t l n ( e ) ;  
}
```

The server socket is dedicated to listen to and accept connections from clients. After accepting a request from a client the server creates a client socket to communicate (to send/receive data) with the client, like below :

```
Socket clientSocket = null ;
```

```

try {
    serviceSocket = MyService . accept ( );
}
catch ( IOException e ) {
    System . out . p r i n t l n ( e ) ;
}

```

Now the server can send/receive data to/from the clients. Since the sockets are like the file descriptors the send/receive operations are implemented like read/write file operations on the in- put/output streams.

A.2.2. Creating an input stream

On the client side, you can use the `DataInputStream` class to create an input stream to receive responses from the server:

```

DataInputStream input ;
try {
    input = new DataInputStream ( MyClient . getInputStream ( ) ) ;
}
catch ( IOException e ) {
    System . out . p r i n t l n ( e ) ;
}

```

The class `DataInputStream` allows you to read lines of text and Java primitive data types in a portable way. It has several read methods such as `read`, `readChar`, `readInt`, `readDouble`, and `readLine`. One has to use whichever function depending on the type of data to receive from the server.

On the server side, the `DataInputStream` is used to receive inputs from the client:

```

DataInputStream input ;
try {
    input = new DataInputStream( serviceSocket . getInputStream ( ));
}
catch ( IOException e ) {
    System . out . p r i n t l n ( e ) ;
}

```

A.2.3. Create an output stream

On the client side, an output stream must be created to send the data to the server socket using the class **`PrintStream`** or **`DataOutputStream`** of **`java.io`** package:

```
PrintStream output ;
try {
    output = new PrintStream ( MyClient . getOutputStream ( ) ) ;
}
catch ( IOException e ) {
    System . out . p r i n t l n ( e ) ;
}
```

The class **PrintStream** implements the methods for displaying Java primitive data types values, like **write** and **println** methods. Also, one may want to use the **DataOutputStream**:

```
DataOutputStream output ;
try {
    output = new DataOutputStream ( MyClient . getOutputStream ( ) ) ;
}
catch ( IOException e ) {
    System . out . p r i n t l n ( e ) ;
}
```

The class `DataOutputStream` allows you to write Java primitive data types; many of its methods write a single Java primitive type to the output stream.

On the server side, one can use the class `PrintStream` to send data to the client.

```
PrintStream output ;
try {
    output = new PrintStream(serviceSocket.getOutputStream());
}
catch ( IOException e ) {
    System . out . p r i n t l n ( e ) ;
}
```

A.2.4. Closing sockets

Closing a socket is like closing a file. You have to close a socket when you do not need it any more. The output and the input streams must be closed as well but before closing the socket.

On the client side you have to close the input and the output streams and the socket like below:

```
try {
    output . close ();
    input . close ();
    MyClient . close ();
}
catch ( IOException e ) {
    System . out . p r i n t l n ( e ) ;
}
```

On the server you have to close the input and output streams and the two sockets as follows:

```
try {
    output . close ();
    input . close ();
    serviceSocket . close ();
    MyService . close ();
}
catch ( IOException e ) {
    System . out . p r i n t l n ( e ) ;
}
```

Usually, on the server side you need to close only the client socket after the client gets served. The server socket is kept open as long as the server is running. A new client can connect to the server on the server socket to establish a new connection, that is, a new client socket¹.

¹Credits: these notes in the appendix are based on <http://www.ase.md/aursu/ClientServer-Threads.html> and chapter 12, section 4 of Introduction to Programming Using Java, 6th ed. (v6.0.3, January 2014), by David J. Eck.