Programming Assignment 1 - UDP Chat

Implementation details | Information sources | Program execution | Examination criteria

1.Purpose with assignment

We want you straight away to have a look at the fundamentals of distributed systems.

- Use basic communication methods to better understand.
- Practice the use of terminology concerning failure model.
- Understand the benefits and drawbacks with datagrams.

This assignment touches on topics from lectures: introduction to distributed systems, characterization of distributed systems and system models.

2.Description of the assignment

In this assignment, you will implement a simple chat server that handles a set of chat clients. The clients and server communicate via UDP packets (“datagrams”) over an IP network. In other words, seen from the perspective of the OSI session and transport layers the architecture is connectionless. The chat server should be able to handle an arbitrary number of clients.

Requirements on the implementation

- A list of participants should be available.
- Crashed clients should be marked "disconnected".
- It should be possible to join and leave the chat room.
- Broadcast and private communication between participants should be possible
- Everyone in the chat room should be alerted when someone enters/leave the chat room.

Requirements on the report:

- A section should discuss which failure model you have chosen and why. Is there a fault that you cannot handle? Why?
- You should discuss the confidentiality of the messages in your chat application. Are your messages secure?
- You should discuss the integrity of the messages in your chat application. Could someone tamper with your message? Why?
- You should add the list of your commandos, also a description, how to send a broadcast, a private message or how to change name.

3.Implementation details

This section contains an outline of the theoretical background necessary to complete the exercise. For further details, refer to the course literature and the numerous sources on the Internet.

3.1. UDP packet communication

A UDP packet, or datagram, has invocation style *maybe*; it is a one-shot message sent between two computers on an IP network. The UDP protocol provides no delivery guarantees, meaning that messages may be lost during transmission. Normally on a LAN the message loss rate is extremely low, but for this assignment we have simulate a 30% message loss rate.

You must implement either at-least-once or at-most-once invocation style using UDP. You will be asked to describe your individual solution. You are advised to discuss your invocation design with your supervisor before attempting to implement it.

A UDP message is transmitted between two sockets, which are logical communication endpoints bound to a specific port, denoted by a port number, on a computer. You can set up your chat server on any machine and any port with port in the range 25000-25050. Do a check that the port is open too!

`[IP of your computer]`, and use port number `25000-25050`

3.2. Server requests

A client should be able to send 5 different types of requests to the server. The requests are listed below:

Connection request

A connection request should be sent to the server during client initialization. It effectively acts as a “subscription request”, announcing the client’s presence to the server so that the server can send chat messages to the client as needed. A connection request should contain the name of the user making the request. The server either accepts or denies the request (e.g., a request could be denied if the user name was already taken, since the communication protocol will likely depends on unique user names) and sends an appropriate response message to the client.

Broadcast message

A broadcast message should be distributed to all clients, preceded with the name of the user sending the message. If user Alfred broadcasts message “Hello world!”, all other clients should see the following line (or similar) in their chat window:

```
Alfred: Hello world!
```

Private message

A private message should be sent to a single user. The request must therefore contain the name of the user to whom the message should be sent. If Alfred wants to send a private message to Bruce, he types

```
/tell Bruce, DotA tonight?
```

On Bruce's screen, this should be displayed as

```
Alfred: DotA tonight?
```

The senders screen should also present this information like

```
Bruce: DotA tonight?
```

Disconnect

There should be a way to leave the chat in a controlled way.

```
/leave Bye folks!
```

List

There should be a way to list participants in the chat.

```
/list
```

3.3. Marshalling and unmarshalling

The contents of a UDP message are simply a sequence of bytes. Thus, you need to convert, or marshal, any information that you want to transmit over UDP into such a representation. Similarly, the receiver of a UDP message must "unpack", or unmarshal, the byte sequence into the preferred local representation. For marshalling purposes, the method `getBytes()` in the `String` class can be used to convert a string into a series of bytes. For unmarshalling, it is convenient to use the `String` constructor:

```
String(byte[] bytes, int offset, int length)
```

This converts the length bytes in byte array bytes starting at offset into a String object. How you format the different messages such that the server can distinguish between the different request types is up to you. A very useful method in the `String` class is `split()` , which separates a text string into several new strings based on a pattern. You will likely find this useful when parsing messages – you can find the documentation for all String methods in the Java API docs (http://java.sun.com/javase/6/docs/api/).

3.4. Handling multiple clients

The server must keep track of all connected clients; their names, IP addresses and port numbers. The code framework (see next paragraph) contains code for maintaining a collection of users. The `ClientConnection` class is used to store user details, and the `Server` object has a `Vector m_connectedClients` of `ClientConnection` objects that holds all connected clients. The Server member methods `addClient()` , `sendPrivateMessage()` and `broadcast()` have been completed in the code framework and can be used to maintain the connected users and send messages to them (although you need to supply the actual message sending code in the `sendMessage()` method in the `ClientConnection` class). The framework does not have any code for disconnecting clients; you have to implement these yourselves.

3.5. Existing code framework

From the ZIP file two source folders can be extracted. `Client` contains all classes related to the client application, and `Server` contains the server-related classes. These files are not complete – all code dealing with the network communication remains to be filled in. However, the code structure and a simple graphical interface for the client have been finished for you.

4.Information sources

You are encouraged to search for information by yourselves. However, the sources listed below are good starting points.

Coulouris et al., chapter 4, in particular sections 4.2-4.2.3, which contain information about sockets and UDP communication. For more information about how to use sockets and UDP in Java, see http://docs.oracle.com/javase/tutorial/networking/TOC.html in particular the section on datagram communication:
http://docs.oracle.com/javase/tutorial/networking/datagrams/clientServer.html

For general details about the Java API, see http://java.oracle.com , more specifically http://docs.oracle.com/javase/7/docs/api/ .

5.Program execution

To run the programs use Eclipse "run configuration..." or, open two command shells. In the first shell, enter the directory with your implementation files and type:

```
javac UDPChat/Server/*.java UDPChat/Client/*.java
```

to compile the source files. Then, to start the server, type:

```
java UDPChat.Server <portnumber>
```

The portnumber argument should be replaced with an integer larger than 1024 that you want to use to listen for client requests. Initially, the portnumber argument is not used, since socket creation code is not part of the framework. You can use the ports 25000-25050, but make sure first that the port is open.

Now, with the server running, enter the same directory in the other shell window and type

```
java UDPChat.Client <hostname/IP> <portnumber> <username>
```

Where `hostname` is the name of the host where you started your server, such as localhost or ubuntu2.iki.his.se. The `portnumber` must match the portnumber you used to start the server, and the `username` can be anything you like as long as it is not rude or immature.

To connect several clients to the same server, you can either start more shell windows, or use a '&' on the command line to start several clients in the background from the same shell. Another effective way of testing your code is to have several group members run clients from their own accounts.

When handing in your assignment you are asked to give both the source code and a runnable file for both the client and the server. In Java these runnable files are called JAR-files. To produce a JAR-file (JAva Archive file) issue the following command:

```
jar cvfm MyJarName.jar manifest.txt *.class
```

The `manifest.txt` need to have the following line:

```
Main-Class: <your main class name, e.g., ServerMain>
```

To test if your JAR-file work as expected try to run it with the following command:

```
java -jar MyServerJarFile.jar <portnumber>
java -jar MyClientJarFile.jar <hostname/IP> <portnumber> <username>
```

6.Examination criteria

6.1. Implementation

The implementation should correctly handle all three types of server requests. It should be possible to connect a reasonable number of clients. The invocation style should be at-least-once or at-most-once.

6.2. Documentation

Each student turns in a very short (max 1-2 pages) report (in PDF format) describing their solution. The report should contain

- A short overview of the solution, detailing parts that you think are important including the theoretical parts on failure models and invocation solutions.
- A reflection of what you have learned in the exercise and feedback on the exercise itself.

You may use either Swedish or English for your report and code. Grammar, good document layout and spell checking is a required. Badly written report will be discarded without comments. The same goes for source code with sloppy indentation or commenting.

6.3. Releasing (submitting) your code and documentation

Hand in using SCIO / Assignments.

1. Attach your PDF-report named YOUR_LOGIN_udp_report.pdf
2. Attach your server-JAR named YOUR_LOGIN_udp_server.jar
3. Attach your client-JAR named YOUR_LOGIN_udp_client.jar
4. Attach your project(s) named YOUR_LOGIN_udp.zip

Happy coding!

This assignment was created by Sanny Syberfelt and further developed by Marcus Brohede. The assignment is maintained by András Márki.