Distributed Systems Seminars: 2. TCP Sockets

All things must come to the soul
from its roots, from where it is planted.
Saint Teresa of Avila

The Story

If we want to develop a great and fast solution, it's clear we should try out the most basic and low-level technology! This approach will allow us to be as flexible and as fast as we want! I'm pretty sure there's nothing that can go wrong and you will be the master of bits and bytes in no time!

Today's goal is to try creating a chat client using the basic TCP sockets and a predefined protocol.

Warning Note

This task is **individual**, thus you have to complete it **on your own**. Whenever identical (or suspiciously similar) solutions are detected, the first one to publish is assumed the author and others get **0 points for the task**. **Note**, that this **does not mean** that you have to be an author of all code in your repository: this is simply not the way how modern IT industry works. You are allowed (**and even encouraged**) to use third-party libraries and solutions from the Internet. However, you should **clearly indicate** where the code originated (it's sufficient to write a comment above the foreign code with URL and the amount of code you borrowed).

This material is **not meant to provide all necessary information** to succeed in the course. It may give you sufficient knowledge to satisfy the minimal criteria, but you should take this document more as a roadmap that simplifies discovering necessary information.

The task will be evaluated manually, thus **provide a meaningful README** file that explains how to run your solution and what to type in order to verify all tasks you solved. **If examiner fails to run your code, you will get 0 points for the task.**

Even though this course **neither focus nor evaluates** the quality of your code, you ought to pay some attention to these properties of your solution: smart design and clean code always pay off on their own.

The Task

1. Create a branch "Seminar02" from your master's or "Seminar01" code and perform the following tasks in this branch **(1 point)**;
2. Establish a socket connection to the server **XX.XX.XX.XX** and gracefully shutdown it at the end of an application (**+1 point**);
3. Implement "ping", "echo" and "exit" commands following the protocol, defined below. (**+1 point**);
4. Implement "login" and "list" commands following the protocol, defined below. (**+1 point**);
5. Implement message sending ("msg" command) and receiving following the protocol, defined below **(+1 point**);
6. **BONUS**: Implement live user list updates (i.e., printing when user logins or logs out) (**+1 point**);
7. **BONUS**: Implement file transfer (and receive) using "file" command. **(+1 point**);

The Communication Protocol

The communication protocol, described below, works in the command-response mode. This means that client is **active** (initiates data exchange) and server is **reactive** (only responds to client requests).

Every command has to start with the **integer** that holds the length of the following message (in bytes). Thus, client has to prepare the message first (i.e., command ID followed by parameters), send message size

to the socket stream and, afterwards, send the message content. When the server managed to parse message content, it will send response in the similar manner (i.e., response size followed by the encoded response). For example, the first command in the table below (ping) has id 1 and no content. Thus, you should send 1 (message size, as int, (which is 4 bytes)) and "1" as byte afterwards (thus, complete transfer is: "01 00 00 00 01" in hex). Server response will be formatted similarly: 1 (message size, as integer) and 2 as a byte (thus, you will receive "01 00 00 00 02").

If the server does not respond within a second or sends truncated responses to your echo commands, you should ensure you are specifying content length correctly. Similarly, if the server responds with the garbage (some unexpected message of unexpected length), this may mean that you did not read previous message completely. Whenever you face this problem, you may try checking request before sending (when it's already in a form of a byte array) and/or read response byte-by-byte.

Command Name	ID (byte)	Content	Response	Comment
Ping	1	-	**2** (byte)	Allows testing connection.
Echo	3	**String** encoded in a byte array (use getBytes).	**String** encoded in a byte array.	Allows testing message size encoding.
Login	5	**String[2]** : login (**String**) password (**String**)	**6** (byte) – new user, registration ok; **7** (byte) – login ok; **Errors:** **100, 101, 102, 110, 111**	After login succeeds, user should be able to use the following commands (not sooner). Encode-decode such content as **written here**.
List	10	-	**String[]** (array with active user names). **Errors:** **100, 101, 102, 112**	In order to detect whether there's an error in response or the valid message, check the response length: errors will always have message length 1.
Msg	15	**String[2] :** receiver login (**String**), message (**String**)	**16** (byte) – message sent. **Errors:** **100, 101, 102, 112**	Receiver should be registered on the server. Otherwise you will get an error.
File	20	**Object[3] :** receiver login (**String**), filename, no path (**String**), file content (**byte[]**)	**21** (byte) – file sent. **Errors:** **100, 101, 102, 112**	Don't forget to send only the name of the file (no path) as that will simplify the receiving! Also, check a file size.
Receive Msg	25	-	**String[] :** login, text **26** (byte) – no messages. **Errors:** **100, 101, 102, 112**	You should invoke this command periodically in order to ensure there are no messages waiting for you to receive.
Receive File	30	-	**Object[3] :** receiver login (**String**), filename, no path (**String**), file content (**byte[]**) **31** (byte) – no files waiting **Errors:** **100, 101, 102, 112**	You'd better be ready that someone sends you the filename with some path and you should retrieve only the name out of it. (there's always some smart guy that fucks up)
Error Codes				
Server Error	-	-	100	Internal Server Error occurred. Usually that means that it's not your fault (or you just sent something THAT bad that server was not expecting).
Wrong Size	-	-	101	The transfer size is below 0 or above 10MB.
Serialization	-	-	102	Server failed to deserialize content.
Unknown	-	-	103	Server did not understand the

Command				command.
Incorrect Parameters	-	-	104	Incorrect number or content of parameters. Server expected something different (usually the stuff described above)
Wrong Password	-	-	110	The specified password is incorrect (this login was used before with another password).
Not Logged In	-	-	112	This command requires login first.
Sending Failed	-	-	113	Failed to send the message or file. Either receiver does not exist or his pending message quota exceeded.

Implementation Hints

The first hint I have for you: don't try developing a nice and sound solution, unless you want to invest a lot of time into this task. At first, try solving a few basic tasks, and postpone formatting and code refactoring for later. Even though it sounds a bit contradicting to everything you should have learned/heard about software development before, sometimes you should be able to understand that time constrains are more important than the code quality. Writing a nice, flexible and maintainable code usually costs quite a lot of time. The lesson you should learn today is that all those levels of abstraction (e.g., protocols, libraries, etc.) are there for some reason, as managing bits and bytes can get quite complicated and buggy really fast. Therefore, don't give up if you don't manage to complete this task as fast as you expect, following tasks should be simpler.

So, let's get directly to the code now.

2. Establish a socket connection

First of all, ensure that you're not hardcoding server address and port: they may (and will) be changed for testing your solution. The best approach is to retrieve server address from the command line. In any case, **don't forget to describe how to configure and run your application in README file**! You should be able to open a client socket and handle exceptions properly (check how your application will be verified below). Also, think how and when to properly close the opened socket. If you have an app that opens connection and reports that connection is correctly opened or opening failed due to some reason, you can be sure that you already got some points for this seminar.

3. Implementing "ping", "echo", and "exit"

I would recommend you implementing "exit" as soon as possible: that's the best place to write all your cleanup/shutdown code.

We discussed how to build a simple command-line interface previous time, thus today we will focus on the new stuff. Both commands ("ping" and "echo") are designed for communication testing purposes, thus it's a great idea to implement them first. First of all, I recommend you to implement some method that has the following (or similar) signature:

```
byte[] executeCommand(byte[] command);
```

Looking at the protocol you should be able to see, that it operates in a request-response manner. Additionally, all messages ought to be prepended with the length of the message. If you encapsulate this logic inside this method, you can immediately forget about that all and focus on more high-level stuff in the remaining code. The method, mentioned above, should create input and output streams from the appropriate streams of the opened socket, **send provided byte array length as int**, send byte array itself

and await for the response (in the similar manner: int that indicates the response size and the response itself).

If you have implemented such method, it should be pretty easy to implement ping command (just pass in `new byte[]{1}` and ensure that you get back `{2}`. If that wasn't the case, be sure to handle errors properly (at least by throwing exception and/or printing the error description/explanation to console).

Similarly, it's pretty easy to implement the echo command: invoke `getBytes` on the string, entered by a user, create another array with the size +1 (to be able to prepend string bytes all with the echo command id (3)), assign echo id to the first element of the new array and copy string's bytes to the new array. For that, you can either write your own simple cycle or use this handy util. After you exchange messages with a server, check for errors, convert returned bytes to string and print response. We're done here!

4. Implement "login" and "list".

Starting with the remaining commands, we have slightly more completed communication: we should convert some complex objects to byte array and back before sending them to the server. Sounds like a great idea for another utility method! I believe that you can define this one on your own. In any case, something really similar to what you need is already defined in the link mentioned in protocol and a few lines above. Additionally to that, ensure you check user input (that some login and password are entered, etc.) and handle properly server errors: with this commands user can indeed cause server to report an error instead of a successful response.

5. Implement sending and receiving messages

Well, sending a message hardly differs from commands discussed above. However, receiving is another story. As the protocol is defined in the request-response manner, there's no way to "subscribe" for new messages and receive them when they are available. Instead, your application should periodically send request for new messages to the server and analyze response. In order to do that, consider using Timer class. Regarding check interval, you are more or less free to pick any, but consider a few facts: human eye does not notice delay shorter than 200ms and delays around second are tolerable for communication. Try setting up your timer within these numbers. Another thing that you should understand when you are using Timer (or another asynchronous processing tool) is that the code you write in a timer task is about to be executed in another thread and may conflict with the code from the main execution thread. For example, if your timer code decides to check for new messages while you are sending user message from the main thread, some shit is going to happen and you will be screwed up at some random points of time. In order to avoid that, you have to ensure that there's **at any point of time a single thread using the socket**. There are a few ways to achieve that: 1) Synchronize access to the socket 2) open a separate socket only for receiving messages (don't forget to login there as well).

Finally, if your code supports sending/receiving messages, you're free to chat with other lucky guys who reached this point, but **don't forget to mention that your application support this in README file and clearly identify how to test this.**

6. BONUS Tasks

Well, if you reached this point and still have some time and desire to continue, there are a few tasks that you can try solving. Both of the bonus tasks are pretty easy and do not need any new concepts and ideas: in order to implement live user updates you should store somewhere list of users returned by the "list" command, periodically invoke "list" and compare the response to the last set of users you had.

Finally, exchanging files is quite similar to exchanging messages, just that you need to check the file size before sending anything (server will just refuse to handle too big messages and stop processing anything you send after that) and ensure you are able to convert file to byte array and back easily.

Looks like that's it, there's nothing more to entertain you… until the next seminar!

Evaluation Process

This section briefly indicates how your task will be evaluated.

1. There is the "Seminar02" branch in your repository: **+1 point**;
2. Application in branch "Seminar02" runs successfully when the server is running and fails with exception or prints a connection error message when server is not available: **+1 point**;
3. Application allows entering ping and echo <any text> commands. Both commands work properly when server is running and fail when server is turned off. Also, both commands print server response (if any), application gracefully shuts down when user types "exit": **+1 point**;
4. Application allows logging on (and properly handles login error messages) and allows listing currently logged in users: **+1 point**;
5. Application allows sending a messages to another user (using "msg" command) and is able to receive a message from the remote user (when logged in): **+1 point**;
6. Application prints when some new user logs in to the server and logs out of the server: **+1 point**;
7. Application allows sending and receiving files using "file" command: **+1 point**;
   Total: 7 points.