Cloud Databases

Milestone 1: Introduction to socket programming

The objective of this assignment is to implement a simple client program that is able to establish a TCP connection to a given server and exchange text messages with it. The client should provide a command line-based interface that captures the user's input and controls the interaction with the server. Besides connection establishment and tear down, the user must be able to pass messages to the server. These messages are in turn echoed back to the client where they are displayed to the user. A sequence of interactions is shown below.

Echoclient *and Echoserver*

Overview

```
EchoClient> connect clouddatabases.msrg.in.tum.de 5551
EchoClient> Connection to MSRG Echo server established: /131.159.52.23 / 5551
EchoClient> send hello world
EchoClient> hello world
EchoClient> disconnect
EchoClient> Connection terminated: 131.159.52.23 / 5551
EchoClient> send hello again
EchoClient> Error! Not connected!
EchoClient> quit
EchoClient> Application exit!
```

The assignment serves to refresh or establish the basic knowledge of TCP-based network programming using Java stream sockets, mostly from the client's perspective. This embodies concepts such as client/server architecture, network streams, and message serialization.

Learning objectives

From the software development perspective, in this assignment you will learn to:

- Understand the client/server paradigm
- Get exposed to socket-based programming, mostly the client socket API
- Differentiate between client-side application, client-side communication stub, messages, basic notions of protocol, and server

- Use Java tools for logging (java.util.Logging) and building (Maven)

We will build on these concepts in subsequent assignments.

Detailed assignment description

Provided infrastructure

Together with this assignment handout, we provide you with an already deployed echo server. The server is running at one of our servers (clouddatabases.msrg.in.tum.de) and listens on port 5551. Once a connection is established, the server will continuously parse bytes (ISO-8859-1 encoded chars) from the network stream until it encounters a carriage return and linefeed (\r\n). This mirrors loosely the convention of telnet (see RFC318). These chars serve as a message delimiter and invoke the server to pass the afore-received message back to the connected client.

Assigned development tasks

In this assignment, the following components need to be developed:

- Client program comprised of
  - Application logic: command-line shell to interact with the server
  - Communication logic: socket-based communication by reading and writing from/to a buffer
- Logging capability to log client/server interactions
- Gracefully handle failures and exceptions

Client program

Develop the client program which consists of the application logic, a simple command-line-based user interface (CLI), and the communication logic for interacting with the server based on TCP stream sockets in Java. For the former, you should use standard in and out streams. (i.e., System.in and System.out). Once the program is started, the CLI should print out the prompt “EchoClient>” in every line and provide the commands listed in the following table.

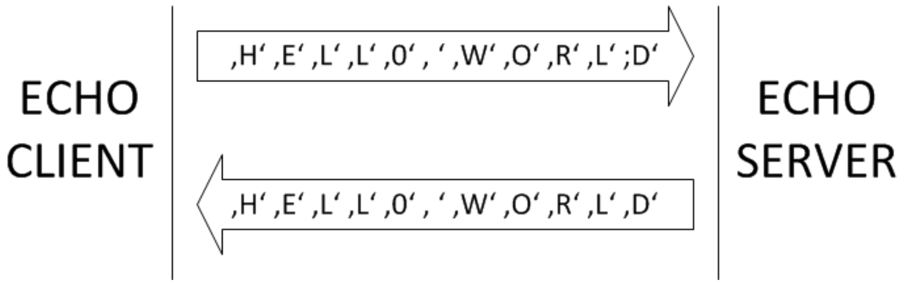
Command	**Informal description**	**Parameters**	**Shell output**
`connect <address> <port>`	Tries to establish a TCP- connection to the echo server based on the given server address and the port number of the echo service.	**address**: Hostname or IP address of the echo server. **port**: The port of the echo service on	**server reply**: Once the connection is established, the echo server will reply with a confirmation message. This message should be displayed to the user.

		the respective server.	(*Note, if the connection establishment failed, the client application should provide a useful error message to the user*);
`disconnect`	Tries to disconnect from the connected server.	-	**status report**: Once the client got disconnected from the server, it should provide a suitable notification to the user. (*Note that the connection might also be lost due to a connection error or a breakdown of the server*);
`send <message>`	Sends a text message to the echo server according to the communication protocol.	**message**: Sequence of ISO-8859-1 coded characters that correspond to the application-specific protocol. (see more detailed description below)	**server reply:** Once the echo server receives the message it will send back the same message to the client. This message should be displayed to the user in a new line. *(Note the situation when the client is not connected to a server)*;
`logLevel <level>`	Sets the logger to the specified log level	**level**: One of the following log levels: (SEVERE | WARNING | … | FINEST)	**confirmation message**: loglevel set from <previous> to <new>
`help`		-	**help text**: Shows the intended usage of the client application and describes its set of commands.
`quit`	Tears down the active connection to the server and exits the program execution.	-	**status report**: Notifies the user about the imminent program shutdown.
`<anything else>`	Any unrecognized input in the context of this application.	`<any>`	**error message**: Unknown command prints the help text.

The communication logic of the client program involves mainly stream sockets (see Java API [Link]). Methods that execute the commands issued by the user are part of the application logic of the client program. Try to decouple the two program components as much as possible. The communication logic should eventually be implemented as a library that provides well defined functionalities (i.e., methods for connection handling, interaction with the server, message passing, etc.). An interface provides a way for applications to get access to these functionalities. Hence, the client application should use this library by calling the respective methods in its interface. This library will be needed in subsequent milestones.

Communication protocol and application-specific messages

The following definitions should help you to develop the communication logic.


‚H' ‚E' ‚L' ‚L' ‚0' ‚ ' ‚W' ‚O' ‚R' ‚L' ;D'
ECHO CLIENT
ECHO SERVER
‚H' ‚E' ‚L' ‚L' ‚0' ‚ ' ‚W' ‚O' ‚R' ‚L' ‚D'

Message

A message in the context of this assignment consists of a sequence of ASCII characters. The carriage-return (i.e., '\r\n') serves as a message delimiter. This means that the server stops parsing the current message once it comes across a carriage return. The maximum message size that is handled by the server is 128 Kbyte.

Protocol

Generally speaking, a protocol is a set of rules and message formats, which describe the communication between different processes to fulfil a specific task by exchanging messages according to the specified rules. The definition of a protocol incorporates two important steps; (1) The specification of the message formats and (2) the order in which messages are exchanged. The communication protocol for this assignment is quite simple. The client sends a text message, as defined above, to the echo server. The echo server in turn parses the message and replies with a message that contains the same content.

Marshalling and unmarshalling

Marshalling refers to the transforming of data elements into a representation that enables the transmission of the message content (e.g., bytes) over a communication network. The inverse operation, unmarshalling, is needed to restore the data elements after they have been transmitted.

This assignment focuses on the message exchange via sockets. In order to understand the principles of network programming, we insist on a low-level communication interface. Hence, your client methods for message sending and reception must only rely on writing bytes to or reading bytes from the socket, respectively. You are not allowed to use the Java-specific object serialization methods like `InputStreamReader / -Writer or ObjectStreamReader / -Writer`. The method signatures should look like the following:

- `void send(byte[]);`
- `byte[] receive();`

Imagine that the server is implemented in a language other than Java and does not understand the Java object serialization format.

Logging

In addition to the actual implementation, we would like to encourage you to get familiar with and use the java.util.Logging. Although there are many, e.g., log4j, log4j2, sl4j, …, we just use the one included in Java. Logging is a useful mechanism to keep track of the program behavior during development and even in production. Besides any other logging information, you consider as useful, log at least all the messages that are sent to and all incoming replies received from the server. Logging should be dynamically controllable (ALL | CONFIG | FINE | FINEST | INFO| OFF | SEVERE | WARNING) by the command `logLevel`. (see table above).

Graceful failure handling

A vital prerequisite for the client is its ability to handle failures gracefully. Please make sure that your program is robust to any kind of wrong or unintended user input (e.g., wrong/unknown commands, number and format of parameters, etc.). In addition to that, consider problems that might occur in communication, i.e., handle exceptions properly and watch out for a controlled breakdown of the connection and the data stream. In addition, error messages also print out status messages to the shell if it makes sense (e.g., if the client is connected or disconnected from the server).

General Considerations

- Document your program properly using JavaDoc comments.
- Stick to the common Java coding conventions ([Link]).
- Pay attention to a good program design (e.g., decoupling of UI and program logic.)

Suggested Development plan

- First of all, clone *Milestone 1* from your respective Git repository from GitLab
- Then, try to set up a socket connection to the server.
- Start with passing single chars to the server and steadily extend the program in order to send messages as defined above.
- Finally, implement the command-line interface and handle errors.

Deliverables & code submission

By the **deadline** (see Moodle), you must hand in your software artifacts that implement all the coding requirements and include all necessary libraries and the build script.

Submission instructions

You must commit and push all the deliverables to GitLab by the deadline. After the deadline, your GitLab repository's role will change from ***Developer*** to ***Reporter***. You will be able to pull the code from GitLab, but you won't be able to push new changes to the repository.

Marking guidelines and marking scheme

All the code you submit must be compatible with the build scripts, interfaces, and test cases that we propose with the respective assignment. In addition, your code must build and execute on *current java version: 1.8,* without any further interference and provide the specified functionality.

The way we test your submission is as follows:

- *We pull your source code*
- *Run "mvn package" and expect in the target directory the jar "echo-client.jar"*
- *We start a separate echo server bound to a different ip and port*
- *Finally, we invoke "java –jar target/echo-client.jar" and use the command line interface*

Additional Resources

- Milestone 1 GitLab's repository: https://gitlab.lrz.de/cdb-21-22/milestone1
- Integrated Development Environment Eclipse: http://www.eclipse.org/
- Integrated Development Environment IntelliJ IDEA: IntelliJ IDEA: The Java IDE for Professional Developers by JetBrains
- Java SE API: Socket (Java Platform SE 8)
- JUnit: http://www.junit.org/
- Maven build tool: Maven – Welcome to Apache Maven
- Echo server: Our echo server implementation is now available under
  - IP: clouddatabases.msrg.in.tum.de
  - Port: 5551