Distributed Systems Programming

A.Y. 2021/22

Laboratory 4

In this laboratory activity, you are invited to practice with the TCP/IP socket API, the de-facto standard API for accessing network services from layers 4-3-2 in the Internet.

The activity is comprehensive of the following tasks:

- implementation of a TCP/IP socket server application (in Java);
- implementation of a TCP/IP socket client application (in Java).

The tool that is recommended for the development of the solution is:

- *Eclipse IDE for Enterprise Java Developers* for the development of the TCP/IP socket server and client.

Context of the activity

The *Converter* service is a concurrent TCP server, listening to the TCP port number 2001, which can perform the media type conversion of image files. The media types supported by this service are three: PNG, JPEG, and GIF. The server is able to establish TCP connections with multiple clients; each request is managed by a different thread.

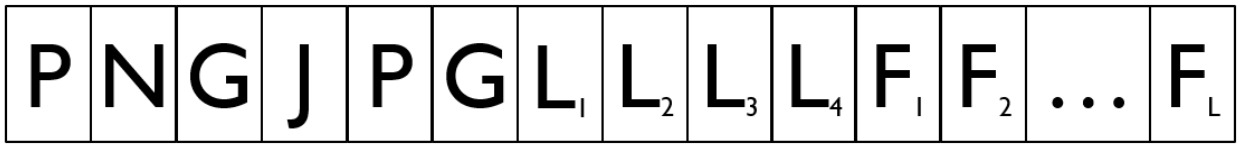
The *ConversionRequest* client is an application that must interact with the *Converter* service to perform the conversion of an image file. This application receives three parameters from the command line:

1. the original media type of the image file;
2. the target media type to which the image file must be converted;
3. the path of the image file in the local file system of the *ConversionRequest* client.

After checking that the file exists, the client establishes a TCP connection with the *Converter* service. On this connection, the following protocol is followed. When the TCP connection is successfully established, the client sends to the server:

- three ASCII characters (1 byte for each character), representing the original media type of the image file that must be converted. The allowed sequences of ASCII characters are PNG, JPG and GIF;
- three ASCII characters (1 byte for each character), representing the media type to which the image file must be converted. The allowed sequences of ASCII characters are PNG, JPG and GIF;
- a 4-byte 2's complement integer number in network byte order, representing the length in bytes of the image file that must be converted;
- the bytes of the image file that must be converted.

The following picture illustrates an example of a client request to convert a file image from PNG to JPEG media type. The bytes L_1, L_2, L_3, L_4 represent the file length; the bytes F_1, F_2, …, F_L represent the content of the file to be converted.



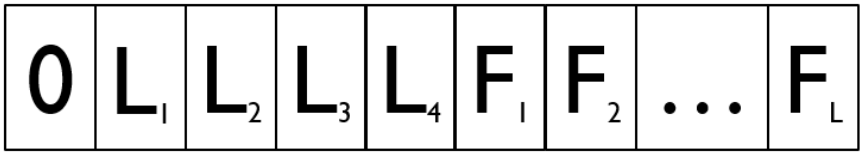
After the server has successfully received all these pieces of information from the client, it converts the file to the requested media type. Then, it sends to the client:

- the ASCII character '0' (1 byte) representing the successful outcome of the receiving and conversion operations;
- a 4-byte 2's complement integer number in network byte order, representing the length in bytes of the converted image file;
- the bytes of the converted image file.

Finally, the server starts the procedure for gracefully closing the connection.

After having received the file and having saved it locally, the client completes the closing of the TCP connection.

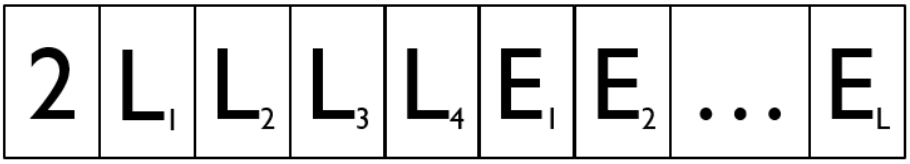
The following picture illustrates an example of a server response in the case that all the operations are successful. The bytes L_1, L_2, L_3, L_4 represent the length of the converted file; the bytes F_1, F_2, …, F_L represent the contents of the converted image file.



Instead, in case there are issues in receiving the message sent by the client or in converting the image file, the server sends to the client:

- an ASCII character (1 byte) representing a number greater than 0 in case of unsuccessful outcome for the receiving or conversion operation (more specifically, the ASCII character is '1' for a wrong request, '2' for internal error of the server);
- a 4-byte integer number in network byte order, representing the length in bytes of a string describing the error that occurred;
- the bytes of an ASCII string describing the error that occurred.

The following picture illustrates an example of a server response in the case that the receiving or conversion operation has failed because of an internal error of the server. The bytes L_1, L_2, L_3, L_4 represent the length of the error message; the bytes E_1, E_2, …, E_L represent the content of the error message.



How to test your client and server

Try the conversion of files of different types and check that the received converted file is correct (i.e., it can be opened by a viewer without errors). Try a conversion with a file of large size (e.g., 10MB) and check that your protocol implementation is still correct, and efficient in transferring the files in terms of transfer time. Run more clients concurrently and check how execution time varies.

Test interoperability by testing your client and server against the reference client and server provided with the lab material. In order to execute the reference client and server, you must use the following commands in their local directories:

- java -jar client.jar <original_type> <target_type> <image_path>
- java -jar server.jar

The images to be converted must be put in the "image" folder of the directory where the client.jar is executed.

Test and improve the robustness of your client and server

Both the client and the server should use timeouts when waiting for input from the peer, in order to avoid deadlocks.

Try your client and server under erroneous conditions and fix any robustness problems:

- Try to connect the client to a non-reachable address.
- Try to stop a client before it has completed its operation (by pressing ^C in its window) and check that the server recovers correctly from this error.
- Try to stop the server (by pressing ^C in its window) while a client is connected and check that the client manages the error.

How to experience this laboratory activity

You are invited to complete both the tasks required to fully carry out this laboratory activity (i.e., development of the TCP server and client). However, you can reuse the Java code that performs the image conversion from the Java *Converter* service, published as solution for the second Laboratory activity.

For the implementation of concurrency in the server-side, you can use the *ExecutorService* framework provided by the JDK, since it simplifies the execution of tasks in asynchronous mode (https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutorService.html).

In order to avoid synchronization problems (e.g., race conditions) between different threads in the server, it is suggested to avoid write operations on shared variables.