Assignment 1
Computer Science 441
Due: 23:55, Friday October 7, 2022
Instructor: M. Ghaderi

1 Objective

This assignment aims to achieve several objectives at the same time. The first objective is to learn the basic structure of a client-server application. The second objective is to learn the general form and requirements of programming assignments in this course. The third objective is to help you refresh your Java programming skills and setup your development environment for the rest of this course. Thus, while simple and straightforward, the expectation is that every student will try and complete this assignment.

2 Specification

2.1 Overview

In this assignment, you will implement a program called `GzipClient`. You have to write code to establish a TCP connection with a remote server, read the content of a given file from the local file system, send it to the server for GZIP compression, receive the compressed file content and save it to a new file in the local file system. While the server used in this assignment provides GZIP compression, the client implementation is in fact independent of that and can be used to obtain other services (*e.g.*, encryption) by changing the server implementation.

2.2 Protocol

The protocol that the client and server use to communicate in this assignment is a custom protocol (*i.e.*, designed by the instructor). The high-level behavior of the sender side and client side of the protocol is presented in Programs 1 and 2, respectively:

Program 1 `GzipServer`

```
1: while not shutdown do
2:     Listen for connection requests
3:     Accept a new connection request from a client
4:     while not end of input stream do
5:         Read data from the socket input stream
6:         Write compressed data to the socket output stream
7:     end while
8:     Close the output stream
9:     Close the client socket
10: end while
```

Program 2 `GzipClient`

Input Parameters: `server`, `inFile`, `outFile`

1: Establish a TCP connection with `server`
2: Open the file specified by `inFile` for reading
3: Create the output file with name `outFile`
4: Read the input file and write to the socket
5: Read from the socket and write to the output file
6: Clean up (*e.g.*, close the streams and socket)

2.3 Implementation

Let us start by discussing the server implementation, as the client implementation should comply with the server implementation. The following code snippet shows the actual implementation of the read and write operations in the server.

```
Socket socket = serverSocket.accept();
InputStream input = socket.getInputStream();
GZIPOutputStream output = new GZIPOutputStream(socket.getOutputStream(), true);

int readBytes = 0;
byte[] buff = new byte[MAX_BUFF_SIZE];

// while not EOF from client side
while ( (readBytes = input.read(buff)) != -1) {
  output.write(buff, 0, readBytes);
  output.flush(); // not needed, but a good practice
}

output.close(); // NEEDED to finalize compression
socket.close(); // non persistent server
```

The important thing to notice is that the server does not know how much data it has to read from the client. In this implementation, the server keeps reading from its socket input stream by repeatedly calling the `read()` method until it returns `-1` (indicating the end of file was reached or the connection was closed). Thus, at the client side, you have to signal to the server that the transmission is complete by closing the socket output stream at the client side. This will cause subsequent calls to `read()` method to eventually return `-1`. Unfortunately, the way Java sockets are implemented, closing the input or output stream of a socket closes the socket as well. To keep the socket open, *e.g.*, to be able to read the compressed data coming from the server, while closing its output stream, the client side program should use `shutdownOutput()` method when it has transmitted the entire file.

The client implementation should have a similar structure. Specifically, as in the server, all read and write operations should operate over a chunk of data (*i.e.*, using a buffer). The size of the buffer is a design decision but a multiple of TCP packet size is recommended for performance reasons (*e.g.*, 32 KByte). While a program can use the version of read and write methods that operate over a single byte, its performance will suffer due to the overhead of single byte input and output operations. An important aspect of your design is how to implement Steps 4 and 5 (see Program 2) to avoid *deadlock*, *i.e.*, a situation where the client and server are blocked by each other. To this end, you must consider implementing sending and receiving operations of the client in separate threads that run independently from each other. *Refer to the assignment notes and tutorials for more detail on this.*

2.4 Design Requirements

Your client design should be based on the following principles:

- Block IO: Read and write operations must use a buffer to perform file and socket IO. As discussed earlier, this is to avoid poor performance.
- Parallel IO: Read and write operations on the socket must be implemented in separate threads, namely one thread to read from the socket and one tread to write to the socket. As discussed earlier, this is to avoid deadlock without any performance penalty.

3 Software Interfaces

3.1 Method Signatures

The required method signatures for class `GzipClient` are provided to you in the source file `GzipClient.java`. There are two methods that you need to implement, namely the constructor `GzipClient()` and method `gzip()`. A brief description of `gzip()` is provided below. For more information, refer to the Javadoc documentation provided in the source file.

- `void gzip(String inFile, String outFile)`
  This is the main method that reads the local file and communicates with the remote server to create a GZIP compressed version of the file. The parameter `inFile` specifies the name of the input file, while the parameter `outFile` specifies the name of the compressed file to be created.

3.2 Exception Handling

Your implementation should include exception handling code to deal with all checked exceptions in your program. Print exception messages (*i.e.*, the stack trace) to the standard system output.

3.3 Running Your Code

A driver class named `GzipDriver` is provided on D2L to demonstrate how we are going to run your code. Read the inline documentation in the source file `GzipDriver.java` for detailed information on how to use the driver class. Moreover, you can download the server program that implements the server side of the protocol from D2L. The server program is provided in a jar file named `gzipserver.jar`. The server comes with a `README` file, which includes instructions on how to run it.

3.4 Console Output

Add print statements (using `System.out.println`) in your code to print the following information on the console:

1. When reading from the socket: `println("R " + numBytes)`
2. When writing to the socket: `println("W " + numBytes)`

where `numBytes` denotes the number of bytes read from or written to the socket. Do not directly print anything else to the console beyond exception messages and the above read/write statements. For debugging purposes, you can use the global `logger` object defined in the `GzipDriver` class, whose level can be set using the command line option `-v`. Refer to the source file `GzipDriver.java` for more information. The logger can be used to write messages to console during code development.

3.5 Design Document

Prepare and submit a design document to describe how you implemented Steps 4 and 5 in Program 2. Follow the design document formatting requirements described on D2L.

Restrictions

- You are not allowed to modify the method signatures provided to you. However, you can (and should) implement additional methods and classes as needed in your implementation.
- You have to write your own code for communicating with the server. Ask the instructor if you are in doubt about any specific Java classes that you want to use in your program.