Design Notes
Assignment 1

Majid Ghaderi

Disclaimer

These notes are based on my own implementation. I do not claim that my implementation is the simplest or the best. Feel free to use or disregard any of these suggestions as you wish.

Send and Receive Operations

A critical aspect of the client design is to avoid deadlocks. If the send and receive operations are not designed properly, deadlock could happen. These operations should be implemented according to server expectations. Recall how the server is implemented:

```
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 server compresses the received data on the fly. It does not, for example, save it in a temp file and then compress and send the file back. The client side should operate similarly.

Consider the client design presented in Program 1. This design could cause deadlock when

compressing large files. The client keeps sending data to the server before reading anything from the socket. The server keeps sending compressed data back to the client, but since client does not read anything from its socket, the socket buffer quickly fills up, which results in blocking the server when it calls `write()` on its socket output stream. Consequently, the server will not read anything from its socket input, which eventually leads to blocking the client. At that point, both the client and server are blocked leading to a deadlock.

Program 1 `Serial Design`

```
1: while not end of inFile do
2:     Read from the inFile
3:     Write to the socket output
4: end while
5: Shutdown socket output stream
6: while not end of socket input stream do
7:     Read from the socket input
8:     Write to the outFile
9: end while
10: Clean up (e.g., close the streams and socket)
```

Potential Solutions

There are at least two solutions that can be implemented to avoid deadlock:

1. **Interleaved Design:** One solution is to have read and write operations interleaved to avoid filling socket buffers. That is, rather than sending the entire file first and then reading from the socket, the client can write a chunk of data and then try to read a compressed chunk from the socket. This is very much similar to how the server is working. However, this approach requires taking care of properly closing the streams and ensuring that all data coming to the client is read. Perhaps the biggest disadvantage of this approach is its performance penalty. Network IO is often the bottleneck in networked applications. Thus, the read and write operations should be parallelized as much as possible in order to maximize the utilization of the input and output streams of the socket.

2. **Parallel Design:** In this approach, sending and receiving operations are performed using separate threads. Therefore, they cannot block each other. It is a very clean design and one that I have implemented in my own code. A thread is in charge of reading the input file and sending it to the server chunk by chunk. Another thread is in charge of reading compressed data from the socket chunk by chunk and writing it to the output file. Note that both threads work with the same socket: one reads from it, while the other writes to it. This is perfectly fine in Java as input and output streams of the socket are independent. With parallel design, you need to properly manage the execution threads, *e.g.*, wait for the

threads to finish before closing the socket, which can be easily achieved using the method `join()` in Java's `Thread` class. See Program 2 for a high-level design.

Program 2 `Parallel Design`

1: Establish a TCP connection with `server`
2: Start the sending and receiving threads
3: Join the sending and receiving threads
4: Clean up (*e.g.*, close the streams and socket)

Testing Your Code

By running the client and server on separate machines (separated over the Internet) and setting a small buffer size, you should see an interleaved sequence of read/write operations (look for R/W messages).