Simple Client/Server Application (15 points)

Within this assignment a simple Client / Server application using plain Java Sockets[1] should be implemented. This will be evaluated on the 5th April!

The server should offer a simple computation service. The three binary operations **+** (addition), **-** (subtraction) and ***** (multiplication) as well as the unary operation **lucas** (Lucas Numbers[2]) should be supported for integers. A client chooses one of the operations, picks one or two arguments (depending on the service) and submits a request to the server. On the server side, the request is processed and the result is returned.

Assignments:

1. Protocol
   Define the protocol to be used for the communication between the client and the server.
   a. Start by defining the set of all required messages
   b. Describe the format of all those messages in JSON[3]
   c. Illustrate the behavior of the server by sketching a transition system. (Hint: simply draw a graph where vertices are states and edges are transitions between states. Label transitions with the message type which has to be received / send or socket events (<open>, <close>) and conditions in which the corresponding transitions shall be triggered. Mark the initial state and eventual terminal states)
   d. Illustrate the behavior of the client using the same means. **(for all: 3.5 points)**

2. Protocol (extended)
   Extend the protocol of the first task by a user authentication mechanism – hence, users have to authenticate themselves before the service is offered. The authentication only consists by stating the name (very secure). If the name is known, the service is offered, otherwise denied.
   (1.5 points)

3. Implementation
   Implement the following components using Java (TCP) Sockets (the following architecture is just a suggestion – components may be implemented differently as long as the same features are supported):
   a. Implement a utility class *Protocol* defining a constant for a server port number as well as two methods *request* and *replay*. The first should accept a socket connected to the server as an argument and uses the associated input / output streams to realize the client side of the communication. Additional parameters specify the operation to be conducted (+,-,*,lucas) and the operands to be passed. The replay method simply accepts a socket connected to the client and conducts the server's end of the communication protocol.
   b. Implement the Server, which is opening a new socket and waiting for client connections. In case a client is connecting, the client's request shall be processed

[1] http://docs.oracle.com/javase/tutorial/networking/sockets/
[2] https://en.wikipedia.org/wiki/Lucas_number
[3] http://json.org/

according to your protocol by forwarding it to the *replay* method of the protocol class.

c. Implement a Client component which is opening a connection to a given server (IP) and requesting one of the offered services. Encapsulate the entire request handling within a single method which is accepting the address of the server, the username, the operation and the operands as arguments and returning the computed result. Handle potential errors/exceptions within this method. Internally, the *request* method should be utilized. **(for all: 5 points)**

For all implementations, ensure proper exception and resource handling (e.g. close all streams + consider the possibility of IOExceptions and authentication errors).

4. <u>Multithreaded Server</u>

Based on your implementation conduct the following steps:

a. Demonstrate that the server is only capable of handling one request at a time.
b. Create a second, multithreaded server implementation by extending your original server. Within the new version, each request should be processed within its own thread. Hence, whenever accept() returns a server socket, a new request handler should be created, which is processing the request within its own thread.
c. Demonstrate that your new server can handle multiple clients simultaneously.

(a,b and c: 3 points)

d. Implement a proper remotely triggered server-shutdown. The server should stop accepting new requests and finishing all already started requests. Also describe the necessary protocol modifications. (You must not use System.exit(…); *Note:* when closing a server socket, any thread currently blocked by an invocation of the accept() method will throw a SocketException – you may exploit this fact.) **(2 point)**

Tip: Let your request be handled using an Extension of the Runnable interface and use an ExecutorService (see Executors[4]) to maintain all threads. For the shutdown, close the server socket first and wait until all jobs within the executor are completed.

[4] http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Executors.html