Study Point Exercise-2 – HTTP, Security and Multi-threaded Servers and Clients

You can earn a maximum of **5** points for this exercise as outlined below:

- 1 point for the exercise "Get HTTP Request Headers on the Server" + " Get/Post-parameters" from the HTTP-exercises
- 1 point for a minimum of one of the exercises from "Securing Java Web Applications-1"
- 1 point for the Exercise: Exam-preparation-1_ProducerConsumer.pdf
- 1 point for the "Simple Echo Server – Preparing for CA-1" (see next pages)
- 1 point for exercise: "Getting Started with Digital Ocean.pdf" (In the folder Instructions and Software). To get this study point you must provide a remote URL+PORT to your server.

If you hand in via mail you can get the additional "attendance point" if your score is **three** our above. If it's not, you should probably have attended the class to get help ;-)

When to hand in

If you hand in via mail send a mail as described below, no later than **Friday, September 2th. 15.00**

If you attend the class, you can demo your solution up until **15.45**

<u>These deadlines are hard. The tutors are paid until 16.00 and are **not** expected to continue after that</u>

How to hand in:

Either demonstrate what you did in the class or send a mail to iwantstudypoints@gmail.com including the following:

Topic: Study Point Exercise-2

Content:
First line should be your full name,
Second line: The link to your repository for the first two exercises
Third line: the link to your Git-hub repository for : Exam-preparation-1_ProducerConsumer.pdf.
Fourth line: the link to your Git-hub repository for the Echo Server Exercise
Fifth line: Instructions (ready to cut and paste) in how to connect to the server via Telnet

Simple Echo Server – Preparing for CA-1

This exercise will guide you through the steps in creating a very simple Echo server + client which will echo any string sent from a client back to (all) clients upper cased (hello world HELLO WORLD). During the exercise you will be guided through a number of steps/hints, all *required* *for CA-1.*

While going through this exercise you will be introduced to:

- *What is a jar(and war) file*
- *Maven plugins*
- *How to run your applications without NetBeans*
- *The importance of log-information, and how to include it in for the Chat Server*
- *How to start your server from a Test Suite and how to test "asynchronous" results, and how to test with Maven*
- *Handling blocking calls on the client*
- *Handling blocking calls on the server*
- *How to upload your server to DigitalOcean*

The task is ***not to complete the exercise as fast as possible****, but to* ***understand*** *the hints you are given throughout the exercise.*

Remember; this is an individual exercise. It will give you more or less all the individual skills necessary to be a great member of a team that has to implement the Chat system. You should actually require the completion of this exercise from all members in your group before you start on CA-1.

Part-1

Fork or clone this project https://github.com/Lars-m/EchoFriday.git.

This is a Maven project, and is in NO WAY tied to NetBeans. So you can open it with any IDE like Eclipse, IntelliJ and obviously NetBeans that can handle Maven Projects. If it complains about something like " invalid target release: 1.8" go to your project properties and select JDK 1.8 as your Java Platform.

This project is an almost identical solution to the `EchoServer` we have implemented in the class + a very simple client with a blocking receive method (make sure you understand what blocking means here).

a) How to Start the Server and Client from within NetBeans and from a normal terminal.

My guess is that up until now most of the code you have designed during your first year with us has been executed from within NetBeans. But here comes the big surprise, that's not how it works in real life ;-)

Obviously we are going to provide our Server and Clients as Jar files that can be executed on any platform with a Java Runtime. On most systems you can start a jar-file, by double clicking it. Our server however, requires start-up arguments (ip and port) so that won't work.

First however, let's execute the Server and Client from within NetBeans the "usual way".

Right click the project node, and select *properties* and *run*. In this dialog select:

- **Main class**: server.EchoServer
- **Arguments**: localhost 7777

Rebuild and run the project and verify that the server was started.

Now right-click `EchoClient.java` and select run. Verify that NetBeans can execute your client code (obviously in a separate thread) while still running the server. This is cool, since you can execute the server normally, and *debug* the client or the other way around completely from within NetBeans.

Before you continue you should spend some time with the code, and familiarize yourself with what it does.

How to run an executable jar-file

Now let's execute the code as an end-user, that is; execute the generated jar-file.

Navigate into the projects *target-folder* and find the generated Echo-0.8.jar file. Copy this into a folder somewhere else (to simulate that we have deployed the server - eventually it needs to go to DigitalOcean) and start it like:

```
java -jar Echo-0.8.jar localhost 9999
```

This is how you start an executable jar-file from the command-line or a script-file (the values in green, are the values we provide to the `main()` function).

This will fail, with the message "no main manifest attribute in Echo-08.jar". The problem is that java has no clue what so ever about which of the files contains the main to execute. Actually, the jar-file is not "executable".

What is a jar-file

Before we solve this problem you should, if you have never done it before, figure out what a jar-file is.

To do that, unzip the jar-file (yes it's a zip-achieve ;-) to somewhere on your system and investigate all its content. Google *jar-files* and make sure you understand what you find (what is a *.class file etc.). Also open the MANIFEST.MF file in a text-editor and notice that it has no information about "the class with the `main()` function".

Creating an Executable jar-file

Now let's see whether we can change the jar-file to be *executable*, that is; it has information about which file to execute when double clicked or started as we did above.

For a Maven project we need a plugin to do that. Copy and Paste the code below into the `plugins` section of your projects pom-file.

```
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-jar-plugin</artifactId>
```

```
              <version>3.0.2</version>
              <configuration>
                 <archive>
                    <manifest>
                       <addClasspath>true</addClasspath>
                       <mainClass>server.EchoServer</mainClass>
                     </manifest>
                  </archive>
              </configuration>
           </plugin>
```

Now do a "Clean and build" and repeat the steps from the section "How to run an executable jar-file". This time it should work.

Again, unpack the jar-file and investigate the MANIFEST to see why this can be executed.

Before you continue, read a little about Maven, Maven-plugins and the Pom-file.

Executing a file with Maven (if you feel you have time, and want more Maven ;-)

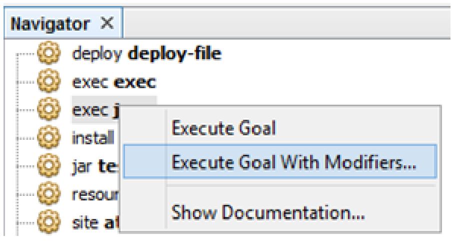
If you want to execute you code with Maven, this is obviously also possible.

Uncomment the `exec-maven-plugin` in your pom-file.

Execute with Maven from within NetBeans:

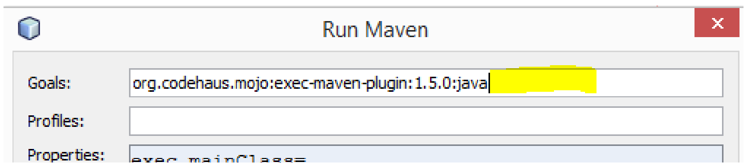
Select the top project node in the projects-tab and notice all the available goals in the Navigator window.

Right-click the exec java goal and select "Execute Goal with Modifiers"



In the "Run Maven" dialog type (in the area marked with yellow):

- `@server` If you want to start the server
- `@client` If you want to start the client



Goals: org.codehaus.mojo:exec-maven-plugin:1.5.0:java

Profiles:

Properties: exec.mainClass=

Investigate the uncommented plugin to understand why this works.

Execute with Maven from a terminal window

Open a terminal (console-window on Windows), navigate to the root of your project and type:

`mvn exec:java@server` This should start your server (unless you already have an instance running)

`mvn exec:java@client` This should start your client

Part-2 Provide the server with a log-file

When a server goes into production, it's important to have access to run-time log-information. When you develop locally, all exceptions are visible in NetBeans, but in production we need a strategy to get this information.
The same is true when we start to deploy our Tomcat applications. The first question you should ask yourself having deployed a server is, "how do I access runtime output and debug information?".
Our simple `EchoServer` (and more important, the Chat-server) obviously also needs a Log-file.
Java ships with a default Logging system, which unfortunately is a bit complex. For this exercise however, you can use the simple Logger strategy given in the code below which uses a single log-level hierarchy for logging:

Implement a new class `Log` and paste in the code below:

```
public class Log {
  public static String LOG_NAME= "myLog";
  public static void setLogFile(String logFile, String logName) {
    try {
      LOG_NAME = logName;
      Logger logger = Logger.getLogger(LOG_NAME);
      FileHandler fileTxt = new FileHandler(logFile);
      fileTxt.setFormatter(new java.util.logging.SimpleFormatter());
      logger.addHandler(fileTxt);
    } catch (IOException | SecurityException ex) {
      Logger.getLogger(LOG_NAME).log(Level.SEVERE, null, ex);
    }
  }
  public static void closeLogger() {
    for (Handler h : Logger.getLogger(LOG_NAME).getHandlers()) {
      System.out.println("Closing logger");
      h.close();
    }
  }
}
```

In you main code, add the following code around the code that starts the server:

```
try {
  Log.setLogFile("logFile.txt", "ServerLog");
  //Start the server here
}
finally{
  Log.closeLogger();
}
```

Now you can log everything like this:

Whenever you want to do a: `System.out.println("Starting the Server");`

Replace with this (note the Level.**INFO**)

```
Logger.getLogger(Log.LOG_NAME).log(Level.INFO, "Starting the Server");
```

Whenever you want to report an exception, you should do it like this (note the Level.**SEVERE**):

```
catch (IOException ex) {
   Logger.getLogger(Log.LOG_NAME).log(Level.SEVERE, null, ex);
}
```

Implement the Log-class in your project. Add the try – finally block as sketched above, add a log message indicating that the server is started (with Level.INFO) and finally replace all exception code as above.

Execute the code, and observe the generated log-file, and its content.

Part-3 Testing the Server

Testing is an important discipline for all serious companies, and therefore also for this semester. For this exercise we are going to use JUnit to perform an integration test involving both the client and the server. Compared to the code we did in the class, the server has been slightly changed with a `stopServer()` method, which means we can start and stop the server from within a JUnit-test as sketched below (this is an integration test, and not a simple unit-test, but JUnit is perfect for that as well).

1) Create a new test class `ClientServerIntegration`**`Test`**[1] and paste in the code below. <u>Make sure you understand why the server must be started in this way.</u>

```
@BeforeClass
  public static void setUpClass() {
    new Thread(new Runnable(){
      @Override
      public void run() {
        String[] args = new String[2];
        args[0] = "localhost";
        args[1] = "7777";
        EchoServer.main(args);
      }
    }).start();
  }

  @AfterClass
  public static void tearDownClass() {
    EchoServer.stopServer();
  }

  @Test
  public void send() throws IOException{
    EchoClient client = new EchoClient();
    client.connect("localhost",7777);
    client.send("Hello");
    assertEquals("HELLO", client.receive());
    client.stop();
  }
```

Execute the test, and verify that it "goes green" (by now, you might want to comment out or remove the initial "test code" in the try-catch block in the clients main).

This test will be a lot more challenging when you have solved the Blocking problems on the Client, because the test result will be delivered asynchronously.

1 In order for the Maven Surfire plugin to find your Test Cases, they must be named like xxxTest.java

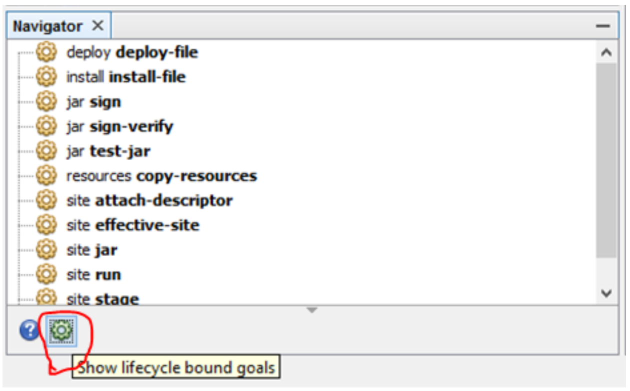
2) Add a new test that should verify that the client (and server) closes the connection when the method `stop()` is called. You might need to provide a public method `Boolean isStoped()` on the client to actually test this.

3) As you continue try to add more tests to verify what you do (not always easy). A great deal of the study points you can earn for the Chat-CA, comes from your testing efforts.

Test with Maven

<u>From inside NetBeans:</u>

Select the top project node in the projects-tab and press the button to show "lifecycle bound goals" as sketched below.



This should make all the lifecycle goals visible, inclusive "surefire test".

Right click *surefire test*, and verify that all tests are executed.

<u>From a terminal window</u>

Open a terminal (console-window on Windows), navigate to the root of your project and type:

`mvn test` Verify that all test are executed

You might feel that the way results are presented here, is a bit "boring" compared to the gui-test runner you are used to.

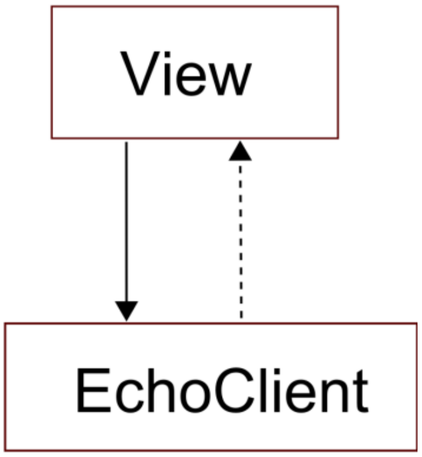
Remember however, that GUI's are for humans to read and Maven are a way for us to automate the full development cycle. Eventually our goal will be to have a system where we can push our changes to git, have "something" that will detect this and; compile, build, test and perhaps even deploy if everything was fine (tests were "green"). If not, the "something" should react by sending a mail or ....

This requires that we can do all the build steps from code, which is what we can do with Maven.

Part-4 Solving the Blocking Problems – Client Side

We are using a blocking API (as opposed to for example Node.js) for our Network communication. This gives a number of challenges both for the client and the server.

a) Provide the client with a GUI



Rewrite the client so that it can be used by a GUI. Obviously this should be done the "right way" so we keep model and presentation separated.

This is pretty simple for the send(..) method since the View can create an EchoClient instance and just call the send(..) method.

For the receive method it’s more complex. The problem is that receive() blocks, so calling the method will block the GUI until we (if ever) receive data from the server.

<u>This problem is two-fold:</u>

First, the view cannot call the EchoClient for information about received data. It is the EchoClient that should `notify` the View whenever it receives data. The EchoClient however, should not have any knowledge about any concrete View’s. This call for the **Observer Pattern:** http://en.wikipedia.org/wiki/Observer_pattern

Secondly, we still have the problem with the blocking call, whenever we try to read from the network socket. This problem calls for **Threads**.

b) Implement an observer design in your client

Solve the first problem by implementing an observer based solution in the client.

You can do this in two ways, both introduced in exercises presented last week. Either implement everything by yourself, inspired by the observer pattern (https://en.wikipedia.org/wiki/Observer_pattern) or use the library interface + class:

- java.util.Observer (http://docs.oracle.com/javase/7/docs/api/java/util/Observer.html)
- java.util.Observable (http://docs.oracle.com/javase/7/docs/api/java/util/Observable.html)

<u>Solve this problem in two steps.</u>

- First use the Observer patter to implement a "general" module that can be used by ALL clients
- Then IDENTIFY the blocking problem, and solve it using threads

c) Update the test code to verify that the data we get via the Observers notify(..) method is the correct value

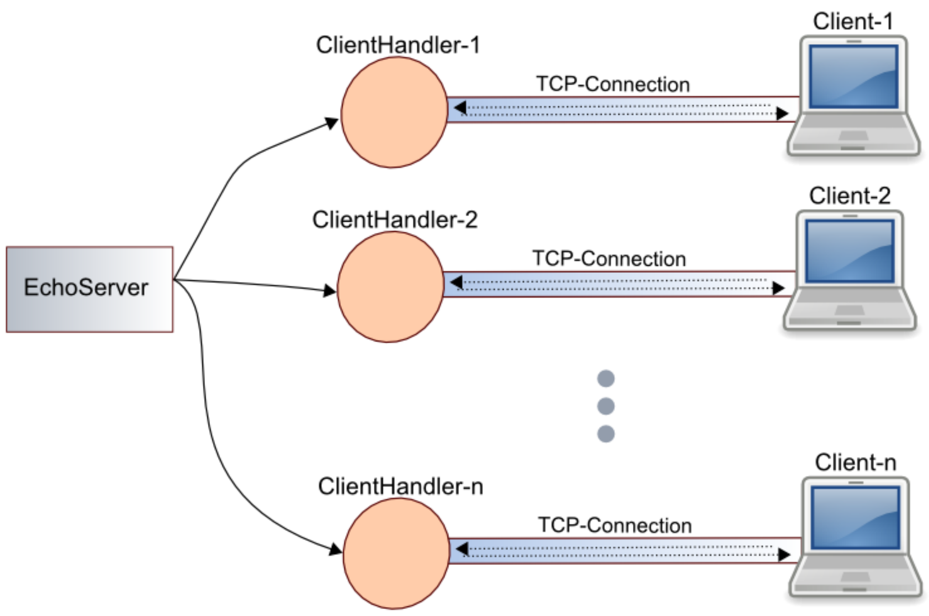
This is probably more challenging than what you would think at first. Again the problem is two-fold. The first problem is that you don't know when the notify method is called (that depends on the server), and the second is that it's called on its own thread. This is why testing the chat-system is rewarded with relatively many study-points. Question related to this MUST be asked on the Q&A system.

Part-5 Solving the Blocking Problems – Server Side

a) Change the **startClient.bat** script to start the GUI, instead of as originally, the EchoClient. Start the server and two or more instances of the client.

Notice how <u>only</u> the first client is being served (until it closes). The problem (again) is due to the blocking calls in the handleClint(..) method which never allow the outer loop to come back to the accept() method to serve a new client.

Again, this problem calls for Threads. What we would like is a design as sketched below where the Echo Server will spawn a new thread (look at it as a virtual CPU) for each incoming connection.



b) Move the handleClient() method (remove the static declaration) into a new class called ClientHandler which should extend Thread (refer to the illustration on the previous page).

c) Change the method to become a constructor (that takes a Socket as the argument).

d) Move everything below the PrintWriter declaration into a method run() (remember the class extends Thread)

e) Move the declaration of input, writer and the socket up to the top of the class to make them visible to both the constructor and the run() method.

f) Replace the original call (in EchoServer) to handle Client with a declaration of a new ClientHandler and call its start method.

g) Update the test-code to verify that the server has the ability to handle many clients "simultaneously".

Final additions to prepare you for the design of the Chat system

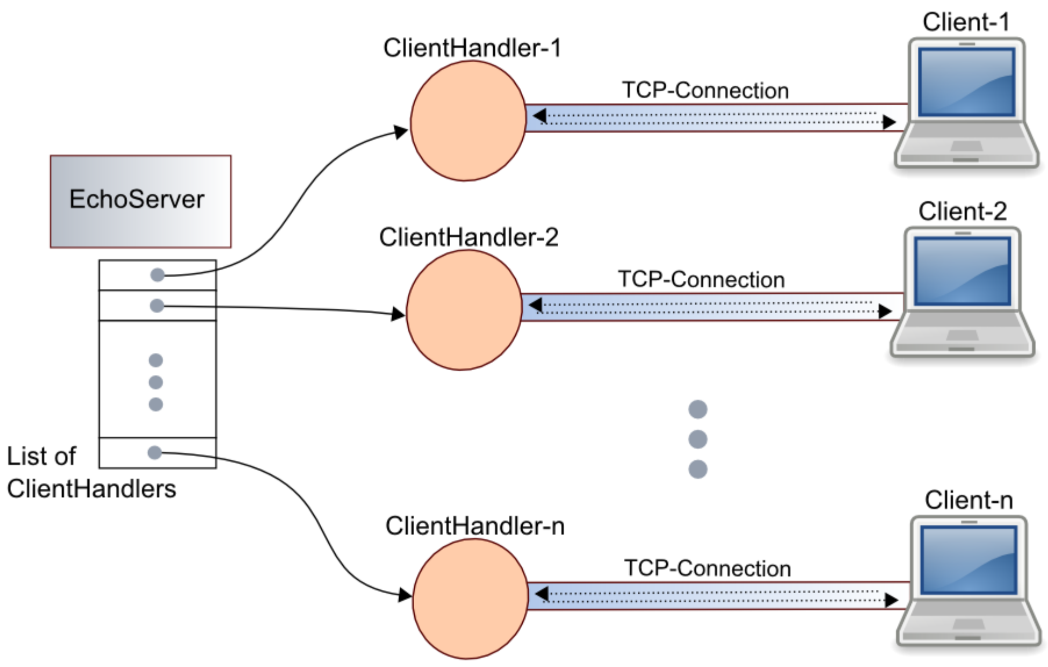
So far the server only echoes back a message to the client that sent the message. For a chat system a message must be delivered to either all, or specific clients other than the one that actually sent the message.

In this last part you should change the EchoServer and ClientHandler so that a message is delivered (upper cased) to **ALL** connected clients.

You could do this by following the steps given below:

- Add a list of ClientHandlers to the EchoServer and add all clients to the list when they connect.
- Provide the EchoServer with a removeHandler(ClientHandler ch) method, called by ClientHandlers when a connection is closed.
- Provide each ClientHandler with a send(String message) which should output the string to the handlers output stream (that is, send it to its actual client).
- Provide the EchoServer with a send(String msg) method which should iterate through all handlers and call their send(..) method with the argument upper cased.

This design is reflected in this figure



Upload the Simple Echo Server to DigitalOcean

Expect to use at least an hour on this exercise, but it will save you a lot of work when you have to deploy your chat-server.

Follow the steps in "To install a Java application server on your virtual machine" in the document ***Getting Started with Digital Ocean.pdf*** (located in the Instruction & Software folder), and install your server on your Digital Ocean virtual machine.

Verify that you can access the server using the client implemented in part-4.