# Lab1

## General Remarks

- **Your solution has to compile and run with Java 7.** In other words, you are free to use Java 7 or older but we do not accept solutions that require Java 8.
- Don't use any other 3rd party library except the ones we provided for you (e.g. Bouncy Castle).

## Submission

- You must upload your solution using TUWEL before the submission deadline: **13.11.2014, 18:00 CET**. - please note that the deadline is hard! You are responsible for submitting your solution in time. If you do not submit, you won't get any points!
- Upload your solution as a **ZIP file**. Please submit **only the provided template and your classes**, the `build.xml` file and a `readme.txt` (no compiled class files, no third-party libraries - except the libraries already provided in the template, no svn/git metadata, no hidden files etc.).
- The purpose of `readme.txt` is to **reflect about your solution**. It should contain a short summary of the status of your code so that a tutor can get the information right before the mandatory interview (see below) and can give you some tips for the next assignment.
- Your submission must compile and run in our lab environment. Use and complete the project template provided in TUWEL.
- Test your solution extensively in our lab environment. It'll be worth the time.
- Please make sure that your upload was successful (i.e., you should be able to download your solution - as the tutors will do during the interview).

## Interviews

- After the submission deadline, there will be a mandatory interview (Abgabegespräch). You must register for a time slot for the interviews using TUWEL.
- You can do the interview only if you submitted your solution before the deadline!
- The interview will take place in the DSLab PC room (see General Course Information). During the interview, you will be asked about the solution that you uploaded (i.e., **changes after the deadline will not be taken into account!**). In the interview you need to explain your code, design and architecture in detail.
- Remember that you can do the interview only once!

---

**Important**: Please note that Lab 2 will extend your Lab 1 solution. That means that it will pay to implement your solution in an extensible way (just like you would build 'real' software).

---

# Project Template

In TUWEL you can find a project template that contains everything you need to get started e.g., an Ant script. Ant is a Java-based build tool that significantly eases the development process. If you have not installed Ant yet, download it and follow the instructions. **Note that some Ant versions have a bug regarding input/output handling**. It is recommended to use the same version that is used in the lab (which is 1.8.4).

We provide a template build file ( `build.xml` in the project template) in which you only have to adjust some parameters and class names. Furthermore, there are `.properties` files that contain parameters such as TCP ports. Fill in those parameters as stated in the description within the file (or according to the Lab Port Policy section). The specific sections are marked with `# TODO: REPLACE with real value such as ...` . **Please do not add additional parameters because we might replace those files for testing purposes**.

Put your source into the subdirectory `src/main/java` . To compile your code, simply type `ant` in the directory where the build file is located. Enter `ant run-controller` to start the cloud controller, `ant run-client` to start the client and `ant run-nodeX` (with X being 1 to 4) to start the respective node. Note that it's **absolutely required** that we are able to start your programs with these predefined commands! Also note that build files created by IDE's like NetBeans very often aren't portable, so please use the provided template.

The template contains a skeleton of the project, plus some (very) basic tests. We encourage you to use the Eclipse IDE, since the template already provides the respective `.classpath` and `.project` files (see the Links section to get started). **Furthermore we plan to use Eclipse for the final Lab-Test, so this is a good opportunity to get familiar with it**. Please adhere to the structure of the template and add your implementation to make the test run through. The template uses the factory pattern to instantiate the key components of the framework. Simply follow the `// TODO` blocks in the factory class `test.ComponentFactory` (located in `src/test/java` ) and return your implementation of the respective interfaces.

Please note that there are two different ways for you to start your application. One is through `ant run-*` targets which execute the static main method of your specified starter class, and the other way is indirectly through `test.ComponentFactory` , which gets executed by some tests. For the latter way it is important that you make use of the provided objects ( `Config` ) as these might get mocked by the test classes. (The class `Config` is described in the implementation details below.)

The template includes a `Shell` class which reads user requests from a given `InputStream` ( `System.in` by default). The user commands are transformed into method invocations of a Java object which handles the commands, using the Java reflection mechanism. The **dslab14-shell-example.zip** available in TUWEL provides a simple usage example, and you may also take a look into the implementation of `cli.Shell` to see how the mechanism works. You do not have to use the provided I/O facility if you prefer to implement the I/O handling on your own. In case you want to use the `Shell` , make sure to register your implementation of `client.IClientCli` using `Shell.register()` . The same mechanism applies for the cloud controller and the node. This allows the `Shell` to look up and invoke the appropriate method for each user command.

The template also contains a class `ScenarioTest` . Its purpose is to let you define test scripts (see examples in `src/test/resources` ), consisting of a sequence of commands which are executed. Hence, `ScenarioTest` allows you to automate a sequence of test commands, which you would otherwise have to type manually in the terminal (see below). We advise you to write your own test scenarios, but this is optional - i.e., if you prefer to test manually, you may also do so.

Note that the predefined tests cover only a very small part of the functionality. We advise you to extend the project with your own testing code and testing scripts. Note that we will run **additional** tests (which are not included in the template) during the grading process, i.e., there is no guarantee that you receive all points if the predefined tests execute successfully.

## Test Scenarios (executed by class `ScenarioTest` )

The system you are going to implement is based on commands like `!login` , `!credits` or `!users` , typed on the command line (see below for further details). In order to facilitate testing and relieve you from having to type these commands in multiple terminals over and over, we provide a simple infrastructure that allows to automate your tests. Please refer to Test Scenario.

## Description

In this assignment you will learn

- the basics of TCP and UDP socket communication
- how to develop and synchronize multithreaded programs
- different connection types

## Overview

In this year's first assignment we are going to build a simple client-server system that can be used to evaluate arithmetic expressions. The architecture is as follows: nodes provide the computation power to perform mathematical operations and are handled by a cloud controller.

At any point of communication, the clients only know the address of one particular server. This server provides no computation power at all, but forwards any incoming calculation request to one or more of the available nodes. Due to this task, we will call this server 'cloud controller' in the following.

The cloud controller handles information about every client and every node in the communication process. Clients are limited in the amount of calculations they are allowed to perform. Because it schedules the client's requests, the cloud controller can easily keep track of the user's current limit and block calculation requests where necessary. The same approach is used to balance the upcoming load of the nodes: The node to choose for responding to the next client request is always the one with the lowest usage at that time (assuming that it provides the required mathematical operation). Figure 1 illustrates a simple example with two clients each evaluating an expression ('client1', 'client2'). The 'Cloud Controller' splits the expressions and forwards them to the nodes that provide the specific mathematical operation. The figure also illustrates how the user credits decrease (it is sufficient that you decrease them at the time when the result is returned).
In case of an error (e.g., division by zero), just subtract credits for every operation that has been performed (including the one that failed) and return an appropriate error message.
Note that you must not change the credits if a calculation cannot be performed due to any technical reason e.g., a node crashed in the meantime and the request cannot be handled because at least one operator is not supported.

For the sake of simplicity, there are a couple of definitions that make the evaluation easier:

- The credits are based on the number of operators in an expression. More concretely, the client has to pay 50 credits per operator e.g.: for `10 + 10` the client pays 50; for `10 + 10 * 2` the client pays 100; ...).
- You can assume that the expressions are valid arithmetic expressions. They consist of two or more numbers separated by a single operator `+` , `-` , `*` , `/` . Between the numbers and the operators there is a whitespace so you can split an expression into its parts more easily. Note that expressions like `5 * -2` are valid because 5 and -2 are valid numbers and there is exactly one operator between them.
- Be sure to handle division correctly (round half away from zero) e.g., `7 / 4` results to `2` , `7 / 3` results to `2` whereas `5 / 2` results to `3` and do not forget to handle divisions by 0.

- You do not have to check for numerical overflows. You can assume that there will not be a calculation that produces a value larger than `Integer.MAX_VALUE` or smaller than `Integer.MIN_VALUE` .
- There is no operator precedence. In other words, expressions are evaluated from left to right. Examples:
  `5 * 3 + 10 = 25 because (5 * 3) + 10`
  `10 + 5 * 3 = 45 because (10 + 5) * 3`

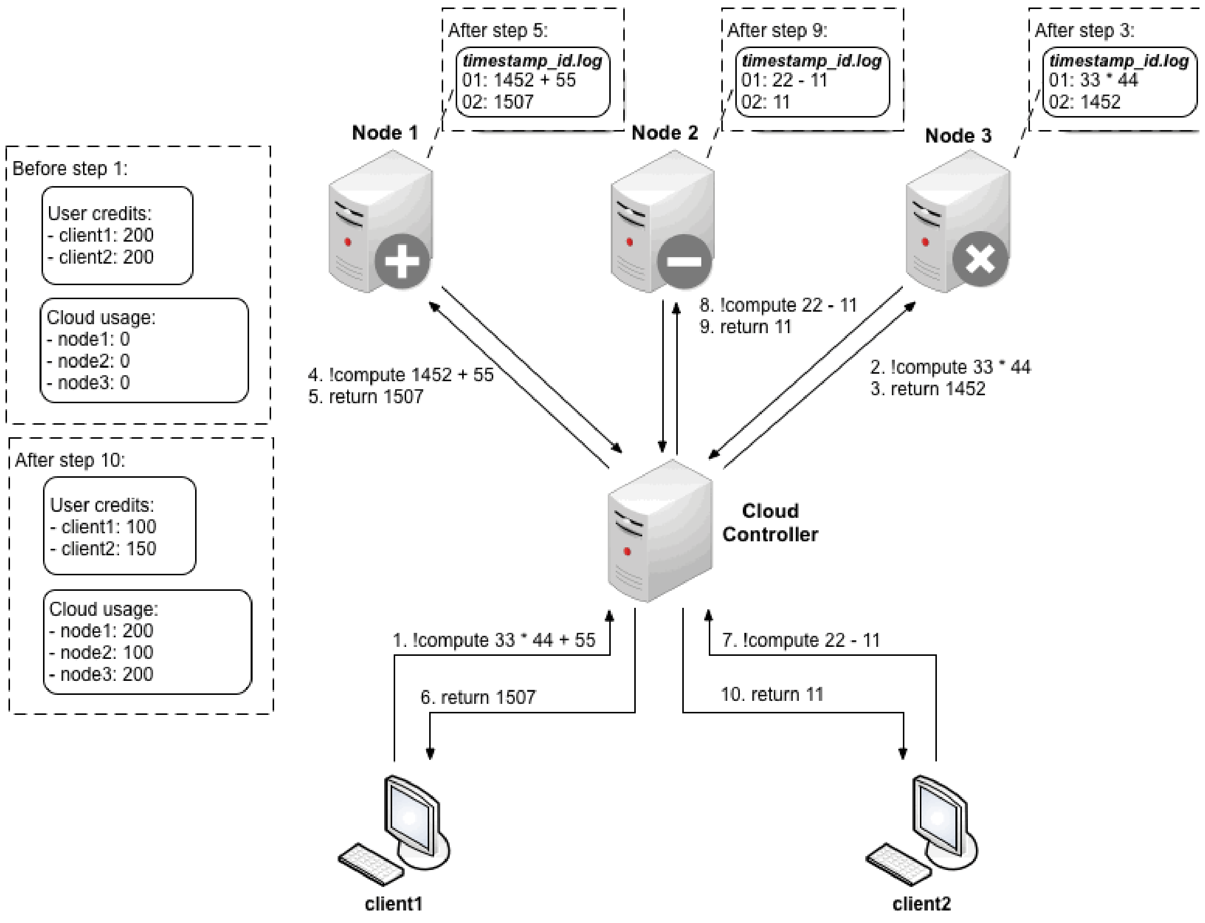

**Figure 1**

To avoid a waste of network resources, there is no connection being held between the cloud controller and a node between two distinct requests. That is, after the node has responded to the cloud controller's request, the connection gets closed again. However, to signal that it is still online and ready to handle requests, a node needs to send UDP messages (so called "isAlive" packets) in a recurring manner - any other communication in this assignment is done using TCP. Figure 2 illustrates this behavior: Imagine that in the example above, 'Node 1' would fail to send isAlive packets to the 'Cloud Controller'. The 'Cloud Controller' will remove 'Node 1' from its list of available nodes.

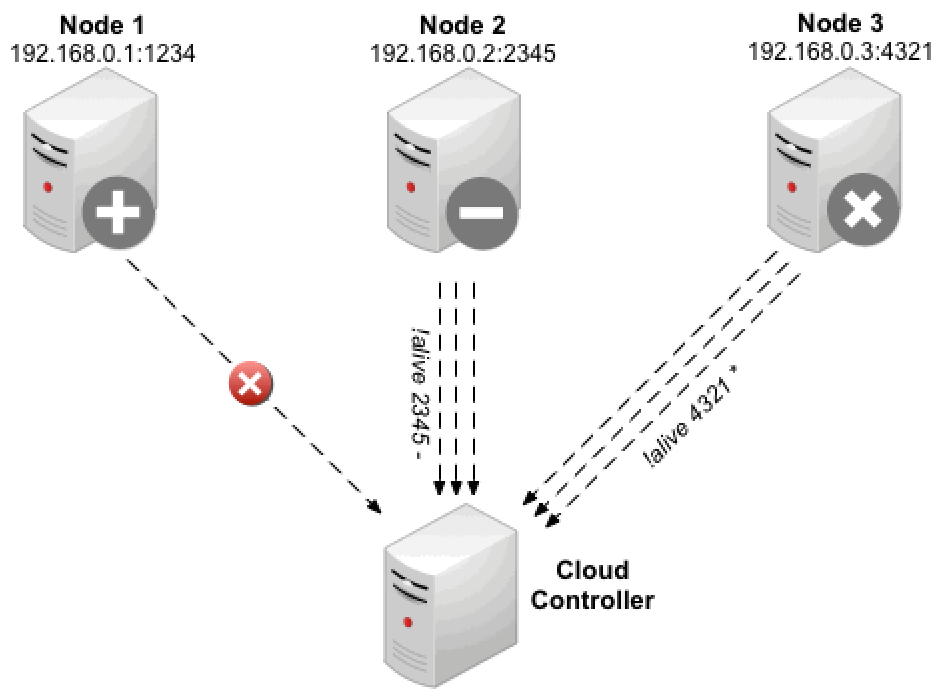

**Figure 2**

Please note that these figures are for illustration purposes and some details have been omitted for the sake of simplicity. You find these details (parameters, return values etc.) below.

When implementing your solution, you should focus particularly on **proper handling of multi-threading and resource utilization**! That is, make sure your code properly closes and cleans up all resources (e.g., sockets, I/O streams), does not utilize excessive CPU resources (avoid "busy waiting"!), and does not leave behind any inaccessible "zombie" threads. Also, make sure that all components of your system can be individually stopped and restarted. In particular, it should be possible to temporarily stop and restart the nodes, and the cloud controller should handle such situations gracefully. If the cloud controller gets restarted, it will lose its state (which is intended), but after some time the state should be recovered (e.g., list of nodes is updated as soon as the `!alive` messages arrive).

In short summary, you should implement the following classes. The description further below discusses the implementation details.

- `client.Client`
- `controller.CloudController`
- `node.Node`

To send requests between the different components (client, cloud controller, node) simply send strings. You are not allowed to use any other types of objects. Especially, you must not use data transfer objects (DTOs) or serialized objects ( `byte[]` ).
In other words, you must not use `ObjectInputStream` / `ObjectOutputStream` or use the Java serialization mechanism in order to transmit objects.

# Cloud Controller

## Configuration Parameters

The cloud controller application reads the following parameters from the `controller.properties` config file:

- `tcp.port` : the port to be used for instantiating a `java.net.ServerSocket` (handling TCP connection requests from clients).
- `udp.port` : the port to be used for instantiating a `java.net.DatagramSocket` (handling UDP isAlive from nodes).
- `node.timeout` : the period in milliseconds each node has to send an isAlive packet (containing the node's TCP port and the supported operations). If no such packet is received within this time, the node is assumed to be offline and is no longer available for handling requests.
- `node.checkPeriod` : specifies the number of delay milliseconds to repeatedly test whether a node has timed-out or not (see `node.timeout` ).

You can assume that the parameters are valid and do not have to verify them.

## Implementation Details

The first thing the cloud controller needs to do on startup is to read the `user.properties` file which must be located in its classpath (the properties file is provided in the template). Each line of a .properties-file stores a single property consisting of key and value. We will use a .properties-file here to store information about each user, more precise, the user's password required for logging in and the particular credits limiting the user in the amount of calculations he/she is allowed to schedule. For instance, an entry in such a .properties-file for user alice with password 12345 and 500 credits looks this:

```
alice.password = 12345
alice.credits = 500
```

The class `util.Config` can be used to read .properties-files from the classpath. Note that you cannot directly obtain a list of users from it. You somehow have to determine them on your own.

When communicating with clients and nodes, the credits of users may change. However, **do not** store these changes back to the .properties-file: The credits of each user shall be reset to the initial value after a restart of the cloud controller.

Next, create a `java.net.ServerSocket` as well as a `java.net.DatagramSocket` instance. We want to concurrently listen for new connections from clients on the `ServerSocket` and wait for incoming isAlive packets on the `DatagramSocket` . Since both relevant methods ( `ServerSocket.accept()` and `DatagramSocket.receive()` ) are blocking operations, you shall spawn a new thread for each in which you call these methods in a loop. This way, the cloud controller is still able to listen for command line inputs.

Since a `java.net.Socket` , which is returned by `ServerSocket.accept()` , provides blocking I/O-operations (via `getInputStream()` and `getOutputStream()` ) and we want to serve multiple clients simultaneously, again each socket connection shall be handled in a separate thread.

Study the java sockets and datagrams tutorial to get familiar with these constructs. We recommend using thread pools (implementations of `java.util.concurrent.ExecutorService` ) for implementing the described behavior. They help to minimize the overhead of creating a thread every time a request is received by reusing already existing thread instances. Java provides some sophisticated implementations that can be easily instantiated by using the static factory methods of `java.util.concurrent.Executors` . Anyway you may also manually instantiate new threads on your own without using these classes. During the interview sessions, you should be able to **explain in detail** which threading strategy you've implemented, how the threads are reused in the pool, whether and how you limit the total number of concurrent threads, etc. and especially **why you have implemented it that way** (what are the alternatives and what are the drawbacks). Help can be found in the Java Concurrency Tutorial.

After loading the user information and initializing all sockets and threads, your cloud controller is able to serve requests.

Concerning the `DatagramSocket` , isAlive messages are the only valid packets that may arrive here. Nodes that did not send such a packet for the specified time ( `node.timeout` ) must be concerned offline. You can either use a thread or a `java.util.Timer` in combination with a `java.util.TimerTask` (be careful with resource handling!) to implement this kind of garbage collector. Do a recurring check every `node.checkPeriod` ms. (If you know a better approach, please feel free to implement it that way. There are a couple of more efficient solutions.)

In case a node goes offline, its usage statistics shall not get lost. It is enough to keep this information in-memory, so the usage statistics can finally get lost after stopping the cloud controller.
Whenever a request is processed successfully, the usage statistics is increased by `50 * (number of result digits)` e.g.: `11 * 22 = 242` and therefore usage is `50 * 3 = 150` . In every other case e.g., division by zero, the usage statistics is not changed.

Concerning TCP communication, different client messages may arrive (described in detail in the client part further below). Some of them require the cloud controller to forward them to one of the available nodes (sometimes the cloud controller may need to adapt the message and add additional information first). In this case, the cloud controller should always choose the least used node that is supposed to be online and is able to handle the request, i.e., the node that has the lowest usage statistics. Keep in mind that even though you listen for isAlive packets, this does not guarantee a node reported to be online did not go down in the meantime.

If the client requests an operation that cannot be handled by any node or the user does not have enough credits to start the computation, return the respective error message.

Again, if the calculation cannot be performed due to any reason which the user is not responsible for, make sure that you do not subtract credits from the users account.

Because the cloud controller is the communication interface for both clients and nodes, the cloud controller can easily keep track of the users' credits and the nodes' usage statistics.

Each computation decreases the credits of a single user and increases the usage of the used node. Naturally, the cloud controller has to count the number of operators in the expression.

Since you've got to manage users and nodes across threads you have to deal with **synchronization** – **make sure your code is thread-safe**. Study the Java Concurrency Tutorial if you are not familiar with threading and/or synchronization.

Finally, the cloud controller accepts the following interactive commands:

- `!nodes`

Prints out some information about each known node, online or offline. A node is known if it has sent at least one isAlive packet since the cloud controller's last startup. The information shall contain the node's IP, TCP port, online status (online/offline) and usage.

E.g.:

```
>: !nodes
1. IP: 127.0.0.1 Port: 10001 offline Usage: 750
2. IP: 127.0.0.2 Port: 10002 online Usage: 200
```

- `!users`

Prints out some information about each user, containing username, login status (online/offline) and credits.

E.g.:

```
>: !users
1. alice online Credits: 500
2. bill offline Credits: 180
```

- `!exit`

Shutdown the cloud controller. Do not forget to logout each logged in user (you do not have to inform them but you must close the connections properly). Note that as long as there is any non-daemon thread alive, the application won't shut down, so you need to stop them. Therefore call `ServerSocket.close()` , which will throw a `java.net.SocketException` in the thread blocked in `ServerSocket.accept()` , and `DatagramSocket.close()` , which will throw a `SocketException` in the thread blocked in `DatagramSocket.receive()` . All other threads currently alive should simply run out. If you are using an `ExecutorService` you have to shut it down (there are a couple of methods including `shutdown()` , `shutdownNow()` , `awaitTermination()` , etc.) and in case of a `Timer` , call `Timer.cancel()` . Anyway you may not call `System.exit()` , instead free all acquired resources orderly.

Further implementation details can be found in the following parts.

# Node

## Configuration Parameters

The node application reads the following parameters from the `nodeX.properties` config file:

- `log.dir` : the directory where the log files are written to (path relative to the project root directory.
  Note for Windows users: please make sure you are using forward slashes (e.g., `logs/node1` ).
- `tcp.port` : the port to be used for instantiating a `ServerSocket` (handling the TCP requests from the cloud controller).
- `controller.host` : the host name (or an IP address) where the cloud controller is running.
- `controller.udp.port` : the UDP port where the cloud controller is listening for node datagrams.
- `node.alive` : the period in ms the node needs to send an isAlive datagram to the cloud controller.
- `node.operators` : a string of supported operators ( `+` , `-` , `*` , `/` ) without whitespace e.g., `node.operators=+` (supports only addition) or `node.operators=*/` (supports multiplication and division).

You can assume that the parameters are valid and do not have to verify them.

## Implementation Details

A node provides up to 4 arithmetic operations ( `node.operators` ). The calculation simply takes two numeric operands and applies the operation `+` , `-` , `*` or `/` , respectively. If the calculation is successful, the resulting number is sent back to the cloud controller. Otherwise, the cloud controller somehow has to be informed about the reason of the failure.

To be able to receive requests from the cloud controller, you have to create a `ServerSocket` again. Blocking I/O-operations should be handled in own threads using exactly the same approach described for the cloud controller part. After a node has completely processed a request and sent the response back to the cloud controller, the respective `Socket` should be closed. For any new request, a new `Socket` needs to be created.

Furthermore, after the request has been handled a log file has to be written to the directory specified by `log.dir` . The name of the file is `<time>_<nodeId>.log` where `<time>` is the current time formatted as follows: `yyyyMMdd_HHmmss.SSS` and `<nodeId>` is the name of the node e.g., `20141001_123015.937_node2.log` .

For the sake of simplicity, you can assume that a node will never finish multiple requests at the same millisecond. However, make sure that your logging facility is thread-safe and concurrent! Hint: When dealing with `Date` , there could be a concurrency issue. Read the documentation thoroughly and you will find out that ThreadLocal is ideally suited for this kind of problem. Use `ThreadLocal` along with `SimpleDateFormat` to format the `Date` properly. During the interview you should be able to explain briefly how this works and discuss alternatives.

The log file has exactly two lines. The first line is the request whereas the second line is either the result of the computation or a string with an error message in case of a failure.

```
5 / 2
3
```

or

```
5 / 0
Error: division by 0
```

From time to time, each node needs to send isAlive packets to the cloud controller to demonstrate it is still online and is ready to handle client requests. The very first packet that arrives works as a registration in the cloud (i.e., the cloud controller then is aware of the new node). Open a `DatagramSocket` on an arbitrary port and send an isAlive-packet every `node.alive` ms. Use the `DatagramSocket.send()` method for this. Making the node's TCP port as well as the supported operations part of the isAlive message is very important so that the cloud controller knows where to forward client requests (example: `!alive 12502 */` ). Again, you can either use a `Thread` or a `Timer` to continually send these datagrams.

The only interactive command the node accepts is `!exit` which shuts down the node. To this, the same rules as for the cloud controller apply.

# Client

## Configuration Parameters

The client application reads the following parameters from the `client.properties` config file):

- `controller.host` : the host name (or an IP address) where the cloud controller is running.
- `controller.tcp.port` : the TCP port where the cloud controller is listening for client connections.

You can assume that the parameters are valid and do not have to verify them.

## Implementation Details

The client communicates with the cloud controller to schedule calculations.

One of the first things to do here is to create a `Socket` and connect to the cloud controller. You will need the `controller.host` and `controller.tcp.port` values for this. Unlike the other components, listening in a separate thread on the client side is not strictly required. Out- and in-bound communication may be blocking and in a single thread. Outgoing messages are sent each time the user enters one of the interactive commands described below. Keep the connection open as long as either the client or the cloud controller shuts down.

## Interactive commands

- `!login <username> <password>`

Log in the user. Before the user hasn't successfully logged in, this is the only command that will be executed by the cloud controller.

E.g.:

```
>: !login alice 23456
Wrong username or password.
>: !login alice 12345
Successfully logged in.
>: !login bill 23456
You are already logged in!
```

- `!credits`

Requests the user's current amount of credits. Requires a successfully logged in user.

E.g.:

```
>: !credits
You have 500 credits left.
```

- `!buy <credits>`

Allows the user to increase his/her amount of credits. Requires a successfully logged in user.

E.g.:

```
>: !credits
You have 500 credits left.
>: !buy 1000
You now have 1500 credits.
```

- `!list`

Gets the list of arithmetic operations that can be used. Requires a successfully logged in user.

E.g.:

```
>: !list
+-*
```

- `!compute <term>`

Sends the given mathematical term to the controller, who returns the result or an appropriate error message in case of a failure.

Eg.:

```
>: !compute 5 + 5
10
```

- `!logout`

Log out the currently logged in user, and drop any state information from memory that the client has associated with this user.

- `!exit`

Shutdown the client: Logout the user if necessary and be sure to release all resources, stop all threads and close any open sockets.

## General

Your implementation should be able to deal with unknown commands or missing arguments (reply with a simple usage message in this case). The cloud controller should also make sure that clients cannot execute any commands different from `!login` or `!exit` before they have actually logged in successfully.

# Test Scenario

The syntax of the test scenarios is very simple, as illustrated by the example below (see `src/test/resources/00_login_test.txt` in the template):

```
* CloudController controller
* Client alice
* Node node1

alice: !credits
> verify("500", T(test.util.Flag).NOT)

alice: !login alice 12345
> verify("success")
alice: !credits
> verify("500")

controller: !users
> verify(".*alice.*online.*bill.*offline.*", T(util.Flag).REGEX)

controller: !exit
alice: !exit
node1: !exit
```

There are four types of commands:

- comments (lines starting with #)
  e.g. `# check whether the credits are updated`
- start instruction (lines starting with a star *)
  e.g. `* Client alice`
- evaluation commands (lines starting with a > are executed directly by the JVM)
  e.g. `> System.out.println("check done")`
- terminal commands (simulates input on a component)
  e.g. `alice: !credits`

The example first starts a cloud controller named "controller", a client named "alice" and a node name "node1" (lines starting with a star, *), simulating three separate "terminal windows". The following lines start with the identifier of the terminal ("alice", "controller" and "node1"), followed by a colon (":"). After the colon, you can define the command that should be executed.
The `ScenarioTest` will simply execute these commands one after the other, which may prove convenient as you develop your solution. You may extend the provided test scenario in `src/test/resources/scenario` . Steps are simple text files that should start with an increasing two-digits number followed by an underscore ( '_' ), e.g., `01_mytest1.txt` , `02_mytest2.txt` etc.
As already mentioned, using the test script feature is optional.

Zuletzt geändert: Donnerstag, 9. Oktober 2014, 11:18