Information Systems Institute

Distributed Systems Group (DSG)
UE Verteilte Systeme WS 2015/16 (184.167)

Lab 1

Submission Deadline: 05.11.2015, 18:00

Contents

1 General Remarks

- We suggest reading the following tutorials before you start implementing:
  - Networking Basics[1]: Short explanation of networking basics like TCP, UDP and ports.
  - Java IO Tutorial[2]: It's absolutely necessary to be familiar with I/O and streams to do sockets operations!
  - Java TCP Sockets Tutorial[3]: A very useful introduction to (TCP) sockets programming.
  - Java Datagrams Tutorial[4]: Provides all the information you need to send and receive datagram packets using UDP.
  - Java Concurrency Tutorial[5]: A tutorial about concurrency that covers threads, thread-pools and synchronization.
- Group work is NOT allowed in this lab. You have to work alone. Discussions with colleagues (e.g., in the forum) are allowed but the code has to be written alone.
- Be sure to check the Hints & Tricky[6] parts section for questions!
- Please make sure you have read the General Course Information[7] thoroughly! It contains important information about the course.
- **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).

[1]`http://java.sun.com/docs/books/tutorial/networking/overview/networking.html`
[2]`http://java.sun.com/docs/books/tutorial/essential/io/index.html`
[3]`http://java.sun.com/docs/books/tutorial/networking/sockets/index.html`
[4]`http://java.sun.com/docs/books/tutorial/networking/datagrams/index.html`
[5]`http://java.sun.com/docs/books/tutorial/essential/concurrency/index.html`
[6]`https://tuwel.tuwien.ac.at/mod/book/view.php?id=226077&chapterid=65`
[7]`https://tuwel.tuwien.ac.at/mod/book/view.php?id=226077`

- Please make sure you properly handle exceptions[8].
- Check the provided FAQ[9], where we discuss a lot of known issues and problems from the last years!

2 Submission Guide

2.1 Submission

- You must upload your solution using TUWEL before the submission deadline: **05.11.2015, 18:00**. - 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).

2.2 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[10]). 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 off to implement your solution in an extensible way (just like you would build 'real' software). Furthermore, note that you can only continue with Lab 2, if you received at least 7 points for this lab.**

[8] `http://www.javatpoint.com/exception-handling-in-java`
[9] `https://tuwel.tuwien.ac.at/mod/book/view.php?id=226077&chapterid=62`
[10] `https://tuwel.tuwien.ac.at/mod/book/view.php?id=226077&chapterid=59`

3 Project Template

In TUWEL you can find a project template that contains everything you need to get started e.g., an Ant script. Ant[11] 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[12]. **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 1.9.2).

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[13] 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-server` to start the chatserver, and `ant run-client` to start the client. 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[14], since the template already provides the respective `.classpath` and `.project` files (see the Links[15] section to get started). **Furthermore we will 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[16] 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 `dslab15-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 chatserver. 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

[11] http://ant.apache.org/
[12] http://ant.apache.org/manual/install.html#installing
[13] https://tuwel.tuwien.ac.at/mod/book/view.php?id=226077&chapterid=66
[14] http://www.eclipse.org/downloads/
[15] https://tuwel.tuwien.ac.at/mod/book/view.php?id=226077&chapterid=64
[16] http://en.wikipedia.org/wiki/Mock_object

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.

3.1 Test Scenarios (executed by `ScenarioTest`)

The system you are going to implement is based on commands like `!login`, `!list` 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 Section 6 (Test Scenario).

4 Lab Description

In this assignment you will learn

- the basics of TCP and UDP socket communication
- how to program multithreaded
- different connection types

4.1 Overview

In this assignment you have to build a simple chat application, which is composed of two components: some **clients** and a **chatserver**. The centralized chatserver is used by all chat clients to communicate with each other. The server is responsible for receiving and processing requests from multiple clients simultaneously. Furthermore, some client requests will be transmitted using TCP and some using UDP.

The **chatserver** application is responsible for:

- accepting new TCP connections from clients.
- reading requests from clients on the TCP connections and optionally responding to them.
- reading requests from clients on the UDP socket and responding to them.

The **client** application is responsible for:

- reading commands from standard input and transmitting them to the server using TCP or UDP (depending on type of the request).
- reading messages from the server on the TCP connection and writing them to standard output.
- reading UDP packets from the server and writing their content to standard output.
- sending/receiving TCP requests to/from other clients

The client first needs to send the `login` request using the TCP connection with the chatserver, before it is allowed to send further requests. The login request requires a correct username and password. This information helps the server to track which users are online and which are offline, and to map client TCP connections to users. When users want to go offline and receive no further chat messages from other users, they send the `logout` request to the chatserver (again using the TCP connection).

There are two types of chat messages that can be sent to other users: `public` and `private` messages. When users send a public chat message, all online users (except the sender) have to receive this message. On the other hand, private chat messages are sent privately only to one user. Therefore, the sender of the private message specifies the receiver of the message, performs a `lookup` to distinguish the private chat address of the respective user, establishes a new TCP connection with the user's private chat address and delivers the message to the user. Both types of messages are sent and received using a TCP connection,

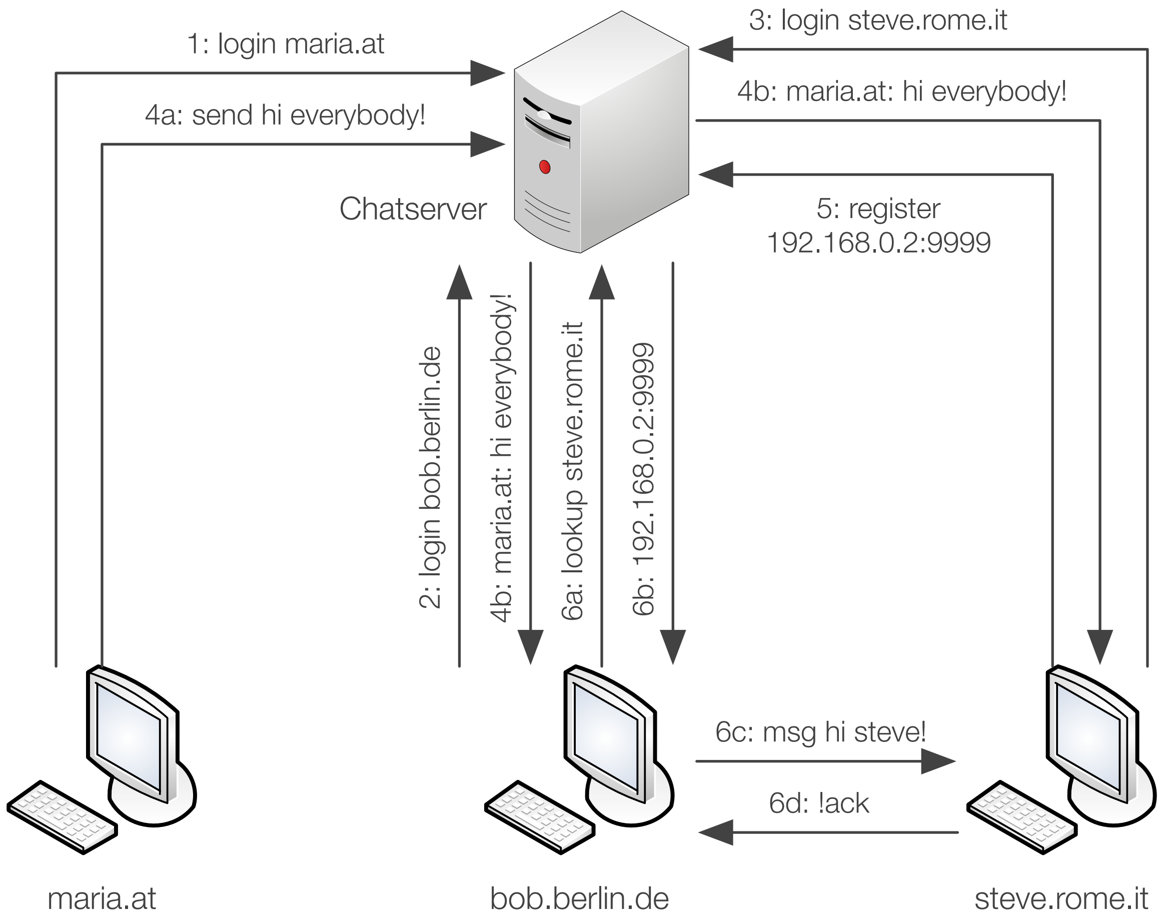

Figure 1: Chat: Overview

because TCP guarantees the order and delivery of the messages, which is primary concern for any chat application.

There is one request that is sent using UDP: `list`. The list request asks the server which users are currently online. The server has to respond with a list of all online users. The server's response is transmitted back to the client using UDP.

4.2 Chatserver

4.2.1 Arguments

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

- `tcp.port`: the port to be used for instantiating a `java.net.ServerSocket` (handling TCP requests from clients).
- `udp.port`: the port to be used for instantiating a `java.net.DatagramSocket` (handling UDP requests from clients).

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

4.2.2 Implementation Details

The first thing the chatserver 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. For instance, an entry in such a `.properties`-file for user 'maria.at' with password '12345' looks like this:

```
maria.at.password = 12345
```

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.

The next step for the server is to create a `java.net.ServerSocket` as well as a `java.net.DatagramSocket` instance. We want to concurrently listen for new connections on the `ServerSocket` and wait for incoming packets on the `DatagramSocket`. The `ServerSocket` listens for incoming TCP connections, whereas the `DatagramSocket` listens for incoming UDP packets. Since both relevant methods (`ServerSocket.accept()` and `DatagramSocket.receive()`) are blocking operations, you need to spawn a new thread for each, in which you call these methods in a loop. Furthermore, since `Socket`s returned by `ServerSocket.accept()` provide blocking I/O-operations (via `getInputStream()` and `getOutputStream()`) and we want to serve multiple clients simultaneously, each client connection should also be handled in a separate thread. To maximize concurrency and performance of our server, each incoming `DatagramPacket` (filled by `DatagramSocket.receive()`) is also processed in a separate thread. Study the Java sockets[17] and datagrams[18] tutorial to get familiar with these constructs.

For implementing multithreading in the server, we recommend using thread pools (implementations of `java.util.concurrent.ExecutorService`). 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 easily be 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[19].

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

Clients communicate with the server using both TCP and UDP. The UDP communication part is easy: the server gets a `DatagramPacket` with the request (`!list`) as the packet's data and responds with a `DatagramPacket` containing the response. You can use the `DatagramPacket.getSocketAddress()` method to find out who sent the request and where to send the response to. The UDP communication is totally anonymous and public, meaning that the chatserver doesn't know whom it is replying. The TCP communication, on the other hand, requires that clients log in first before they can send any public or private messages. After a successful log in, the user's state is set as online, and the chatserver responds with the `"Successfully logged in."` message. The chatserver should also save the TCP connection on which the user is logged in to be able to send messages to this user later on. When the chatserver receives the log out request from the user, it should set the user's state as offline, respond with `"Successfully logged out"` message, and close the TCP connection with the client. If the server receives an unknown request, it should reply with an error message using the same channel of communication (TCP or UDP) from where the request originated.

Since our chat application also allows private messages, users can `register` their private addresses (`IP:port`) at the server. Based on this information, users can perform a `lookup` for a specific username at the server and receive the private address of the respective user. This address will then be used by the

[17] http://java.sun.com/docs/books/tutorial/networking/sockets/index.html
[18] http://java.sun.com/docs/books/tutorial/networking/datagrams/index.html
[19] http://java.sun.com/docs/books/tutorial/essential/concurrency/index.html

user to establish a private connection and send a private message. For this lab, you can decide how the chatserver stores the users's private address details (username, IP, port) by for example using a simple in-memory storage. However, please keep in mind that we will extend the storage and lookup mechanism in the next lab.

Since you've got to manage users across threads you have to deal with synchronization – make sure your code is thread-safe! Therefore, you should not only put the `synchronized` keyword blindly on every method, because this makes the execution of the method serial which defeats the purpose of making the server multi-threaded. Instead, think about the minimal subset of your code that needs to be thread-safe! Study the Java Concurrency Tutorial[20] if you are not familiar with threading and/or synchronization.

Finally, the chatserver accepts the following interactive commands:

- `!users`

  Prints out the username (in alphabetical order) and login status (online/offline) about each known user.

  E.g.:

  ```
  >: !users
  1. bob.berlin.de offline
  2. maria.at online
  ```

- `!exit`

  Shutdown the chatserver. 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 properly. Therefore, call `ServerSocket.close()`, which throws a `SocketException` in the thread blocked by `ServerSocket.accept()`, and `DatagramSocket.close()`, which throws a `SocketException` in the thread blocked by `DatagramSocket.receive()`. For the threads responsible for communicating with clients you can use `Socket.close()`, which also throws a `SocketException`. If you are using a `java.util.concurrent.ExecutorService` you have to call its `shutdown()` method. Anyway you may not call `System.exit()`. Instead you must free all acquired resources and exit normally.

4.3 Client

4.3.1 Arguments

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

- `chatserver.host`: the host name (or an IP address) where the chatserver is running.
- `chatserver.tcp.port`: the TCP port where the chatserver is listening for client connections.
- `chatserver.udp.port`: the UDP port where the chatserver is listening for client requests.

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

4.3.2 Implementation Details

The main task of your client is to read user requests from standard input (`System.in`), process them if necessary and send them to the chatserver's TCP or UDP port (depending on type of the request). To open the TCP connection with the chatserver, initialize a `java.net.Socket` instance with chatserver's host and TCP port. From there you can use `InputStream` and `OutputStream` (returned by `getInputStream()` and `getOutputStream()`) to communicate with the server. Because reading from the `InputStream` is a

[20] http://java.sun.com/docs/books/tutorial/essential/concurrency/index.html

blocking I/O operation, you will require new threads to read responses from the chatserver. To communicate with the chatserver using UDP, initialize a `java.net.DatagramSocket` with an empty constructor to use any available port on the system. To send a request, create a `java.net.DatagramPacket` with the request as a byte array (use the `String.getBytes()` method), chatserver's host and UDP port. The sending of `DatagramPackets` is handled by the `DatagramSocket.send()` method. To read the chatserver's UDP responses, use the `DatagramSocket.receive()` method with newly created `DatagramPackets`. The `DatagramSocket.receive()` method is also a blocking I/O operation and will require a new 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 chatserver doesn't shut down. For all commands, except for `!list`, the client needs to be logged-in, in order to send any public or private message. When sending a public message, the message is sent to all the other clients connected to the chatserver, where users are currently online (i.e., logged in). This means that the clients should be listening for messages from the chatserver, on the TCP connection established with the chatserver at login. Private messages allow clients to directly communicate with other clients. This means that the message is not sent to the server, but instead a new TCP connection to the recipient client is established, whose address needs to be discovered through an implicit `!lookup` command issued by the client as shown in Figure 2.

Finally, the client accepts the following interactive commands:

- `!login <username> <password>`

  Log in user. This must be the first message to be sent over a TCP connection.

  E.g.:

  ```
  >: !login bob.berlin.de 23456
  Wrong username or password.
  >: !login bob.berlin.de 12345
  Successfully logged in.
  ```

- `!logout`

  Log out currently logged in user. This command is sent over the TCP connection and requires the user to be logged in.

  E.g.:

  ```
  >: !logout
  Successfully logged out.
  ```

- `!send <message>`

  Send a public message (the message may contain spaces). This command is sent over the TCP connection and requires the user to be logged in.

  E.g.:

  ```
  # bob.berlin.de is logged in
  >: !send this is a public test message

  # output on other connected clients, where users are online
  bob.berlin.de: this is a public test message
  ```

- `!register <IP:port>`

  Registers the private address (`IP:port`) that can be used by another user to establish a private connection, at the server. Furthermore, the client creates a new `ServerSocket` for the given port and listens for incoming connections from other clients.

  E.g.:

  ```
  >: !register 192.186.0.3:8888
  Successfully registered address for bill.de.
  ```

- `!lookup <username>`

  Performs a lookup of the given username and returns the address (`IP:port`) that has to be used to establish a private connection.

  E.g.:

  ```
  >: !lookup steve.rome.it
  192.186.0.2:9999
  ```

- `!msg <receiver> <message>`

  Sends a private message to the receiver (the message may contain spaces). The receiver parameter specifies the username of the message receiver. To send a private message the client has to resolve the address of the specified user, by using the previously introduced `lookup` mechanism. After resolving the address, the client establishes a new TCP connection with the receiver, sends the message and waits for an acknowledgement. Once the receiver successfully received the message the client automatically replies with an acknowledgement (`!ack`) and both ends close the connection. See Figure 2, which illustrates the overall process.

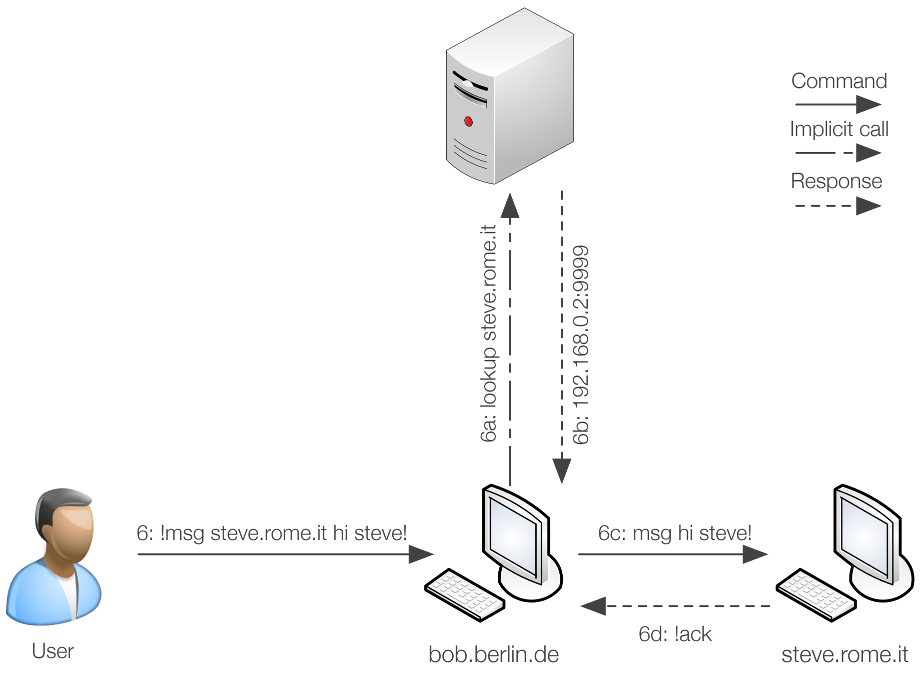

Figure 2: Sending private messages with `!msg` command

  E.g.:

  ```
  >: !msg bill.de this is a private test message
  Wrong username or user not reachable.
  >: !msg steve.rome.it this is a private test message
  steve.rom.it replied with !ack.
  ```

- `!list`

  List all online users in alphabetical order. This command is sent over the UDP connection and doesn't require the user to be logged in.

  E.g.:

  ```
  >: !list
  Online users:
  * bob.berlin.de
  * steve.rome.it
  ```

- `!lastMsg`

  Prints the last received **public** message. In order to do so the client only saves the last received public message.

  ```
  >: !lastMsg
  No message received!

  # After bob.berlin.de sends a public message
  >: !lastMsg
  bob.berlin.de: this is a public test message
  ```

- `!exit`

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

5 Further Reading Suggestions

- APIs:
  - IO: IO Package API[21]
  - Concurrency: Thread API[22], Runnable API[23], ExecutorService API[24], Executors API[25]
  - Java TCP Sockets: ServerSocket API[26], Socket API[27]
  - Java Datagrams: DatagramSocket API[28], DatagramPacket API[29]
- Tutorials:
  - JavaInsel Sockets Tutorial - Section 21.6[30], 21.7[31]: German tutorial for using TCP sockets.
  - JavaInsel Datagrams Tutorial - Section 11.11[32]: German tutorial for using datagram sockets (note that this chapter is from edition 7 because it has been removed in newer versions).

6 Test Scenario

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

```
* Chatserver chatserver
* Client maria.at

# check whether '!send' is successful
maria.at: !send test
> verify("success", T(test.util.Flag).NOT)

maria.at: !login maria.at 12345
> verify("success")

chatserver: !users
```

[21] http://java.sun.com/javase/7/docs/api/index.html?java/io/package-summary.html
[22] http://java.sun.com/javase/7/docs/api/index.html?java/lang/Thread.html
[23] http://java.sun.com/javase/7/docs/api/index.html?java/lang/Runnable.html
[24] http://java.sun.com/javase/7/docs/api/index.html?java/util/concurrent/ExecutorService.html
[25] http://java.sun.com/javase/7/docs/api/index.html?java/util/concurrent/Executors.html
[26] http://java.sun.com/javase/7/docs/api/index.html?java/net/ServerSocket.html
[27] http://java.sun.com/javase/7/docs/api/index.html?java/net/Socket.html
[28] http://java.sun.com/javase/7/docs/api/index.html?java/net/DatagramSocket.html
[29] http://java.sun.com/javase/7/docs/api/index.html?java/net/DatagramPacket.html
[30] http://openbook.galileocomputing.de/javainsel9/javainsel_21_006.htm#mjcd64e398cec5737d9a288a4b4df04e2b
[31] http://openbook.galileocomputing.de/javainsel9/javainsel_21_007.htm#mj1ba27dc5fdf53f527163767f188e1d2e
[32] http://openbook.galileocomputing.de/java7/1507_11_011.html#dodtp497f87ed-dd23-48d4-80c7-7e11b3ec99d6

```
> verify(".*bill.*offline.*maria.*online.*", T(util.Flag).REGEX)

chatserver: !exit
maria.at: !exit
```

There are four types of commands:

- comments (lines starting with #)

  e.g. `# check wether '!send' is successful`

- start instruction (lines starting with a star *)

  e.g. `* Client maria.at`

- 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. `maria.at: !login`

The example first starts a chatserver named "chatserver" and a client named "maria.at" (lines starting with a star, *), simulating two separate "terminal windows". The following lines start with the identifier of the terminal ("maria.at" and "chatserver"), 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.