Course project

Kim Hammar

7 dec 2015

Revision History

Revision	**Date**	**Author(s)**	**Description**
0.1	07.12.15	KH	First draft
0.2	13.12.15	KH	Error correction

Contents

1 Abstract

Course project in a course on Network programming in Java [1], carried out at Royal Institute of Technology, Stockholm.

2 Task specification

You are "hired" by JEM inc (Java Enterprise Microsystems Inc.) to design and develop the distributed application software (clients and servers) for the NOG (Nordic Olympic Games) event.

The NOG information system should allow storing, retrieving and updating personal information about NOG participants. The system should also be able to provide statistical information about participants. The system is to be developed in two version(1) a single-user version; (2) a multi-user version. You should also develop a NOG virtual meeting place. The NOG virtual meeting place is Internet based software which offers remotely located users to communicate and share information represented as textual, image or audio files.

2.1 Sub-assignment 1. A Single-User Information System for NOG

Develop a distributed application in Java that allows storing, retrieving and updating information about participants of NOG. The application should concist of a client with a user interface and a server. In this assignment we assume a single user semantics for the application, i.e It's not required to support coherency of multiple copies of participant records which may be cached by multiple client at the same time.

2.2 Sub-assignment 2. A Multi-User Information System for NOG

Develop a multi-user application, similar to the solution developed in sub-assignment 1. In this version a multi-user semantic is required. Many users can fetch the participants-data at the same time and when one user updates his local-cache of the data, the change need to be replicated among all clients connected in order to prevent them from using stale data.

2.3 Sub-assignment 3. Chat Rooms for NOG

Develop a distributed "building of chat rooms".

3 Platform

The platform used for the development-process, benchmarks and tests is a computer running Xubuntu 14.04 LTS, CPU: Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz

Java version: 1.7.0_79 (OpenJDK version 7 update 79).

4 Software and technologies used

- Java Remote Method Invocation (java.rmi package)
- Java Persistence API (javax.persistence package)
- JDBC (java.sql package)
- Java Swing (javax.swing package)
- Java Socket (java.net package)
- PostgreSQL 9.3.9
- Apache JMeter
- NetBeans IDE 8.0.2
- Javadoc

5 The application

5.1 Sub-assignment 1. A Single-User Information System for NOG

Distributed java application that allows storing and updating information about participants in NOG.

The server is using the HTTP protocol and only supports GET and PUT methods (PUT For storing and GET for fetching). The server is multi-threaded but doesnt provide any multi-user semantics in the likes of keeping consistent data among clients (hence the minimized protocol of GET/PUT).

The client is a simple http-client atached with a GUI developed with Java Swing.

5.1.1 Protocols used

- HTTP
  - PUT/GET methods.

5.1.2 GUI

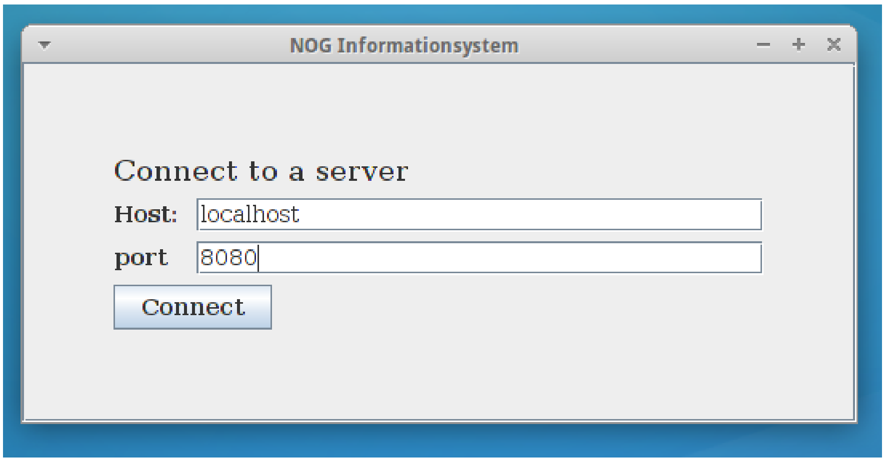

Figure 1: Screenshot of the startframe of the application

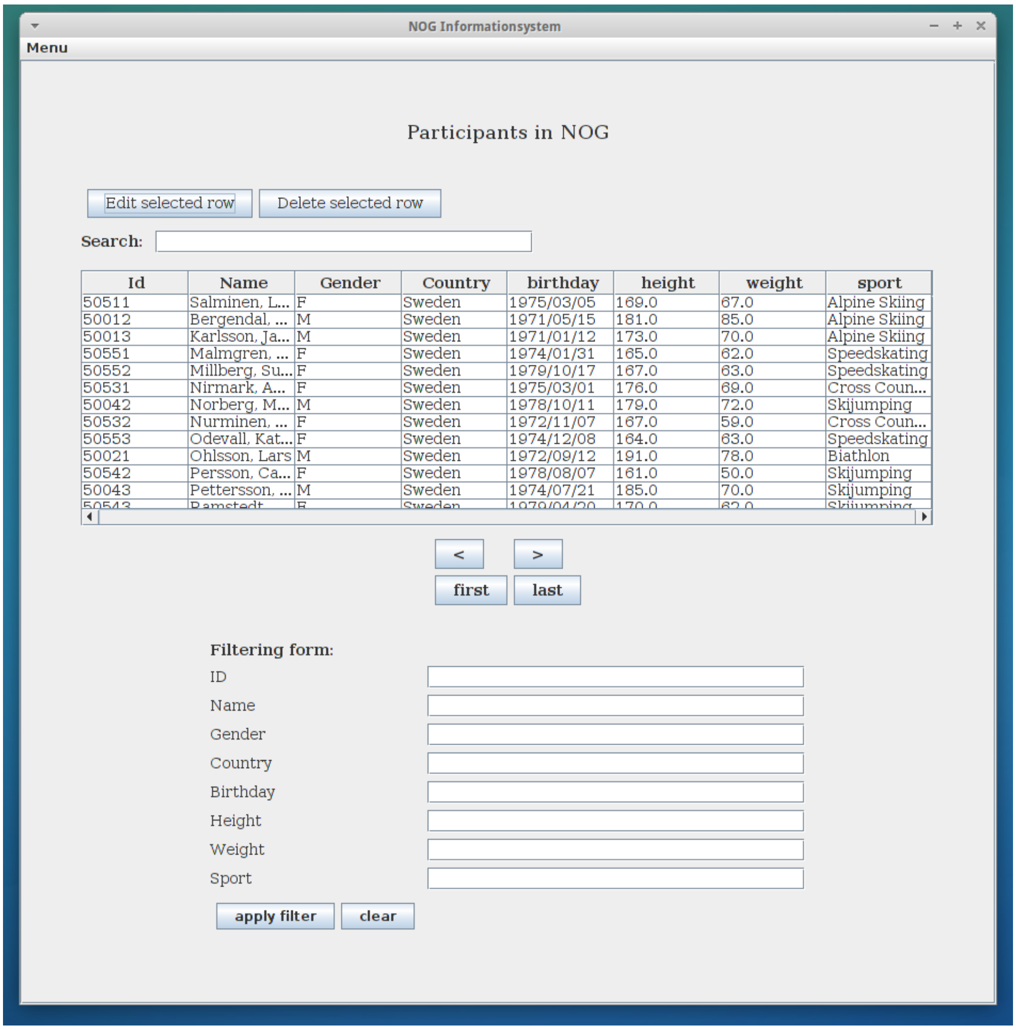

Figure 2: Screenshot of the mainPanel of the application

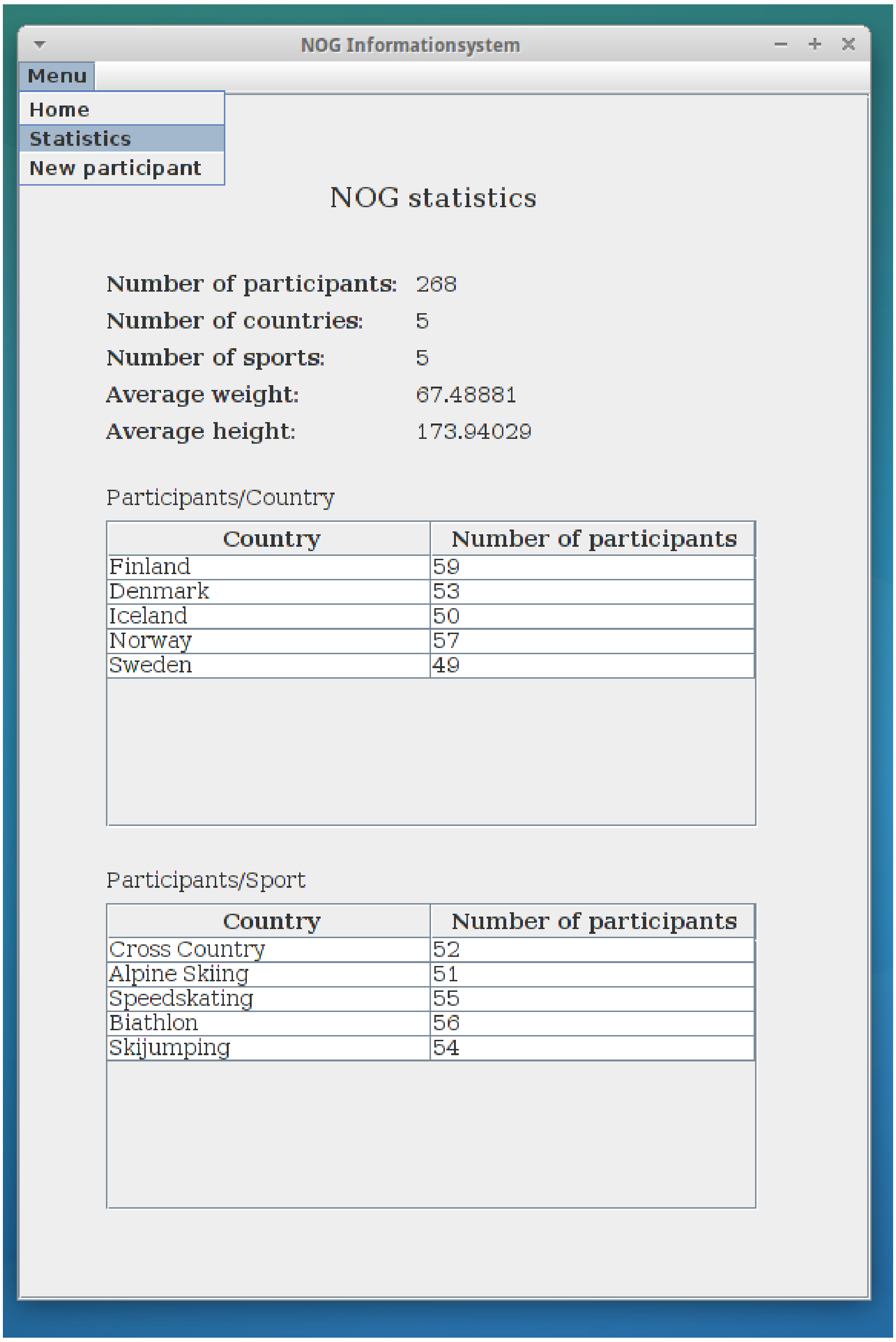

Figure 3: Screenshot of the statisticsPanel of the application

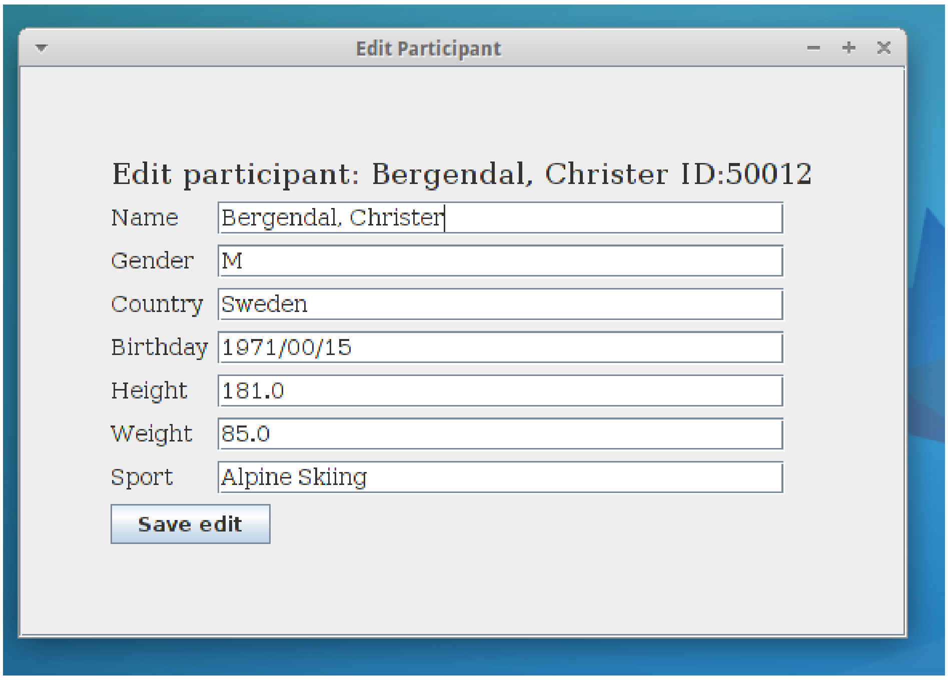

Figure 4: Screenshot of the EditPanel of the application

5.1.3 Architecture

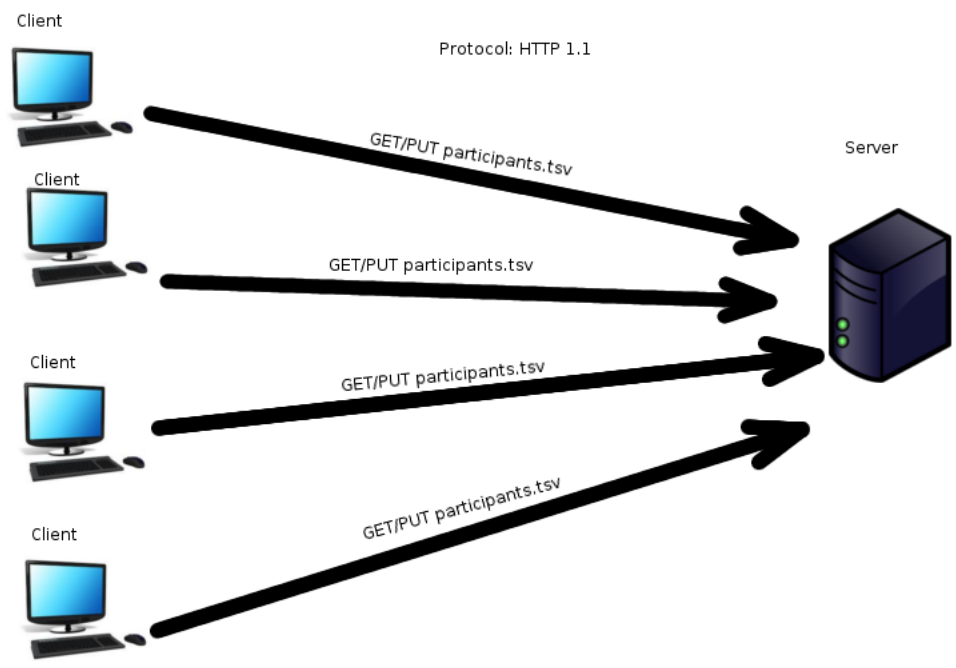

Figure 5: Application architecture (client-server).

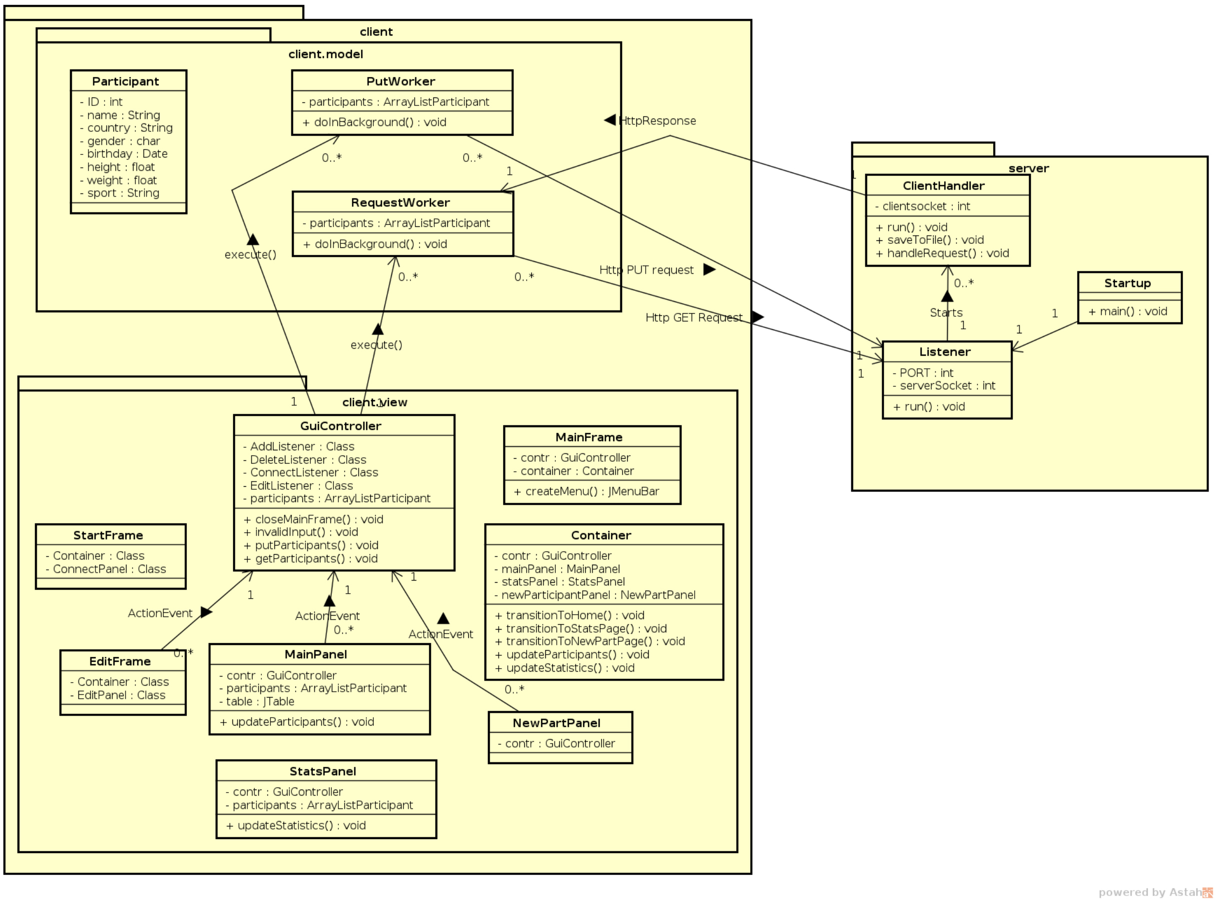

Figure 6: Class diagram for the application.

5.1.4 Load-testing

These tests have been done on my local machine so It is'nt a real proof of how the application would hold under huge loads in production but we can still see some interesting results. My main purpose with this load test was to see how well the multi-threaded semantics is working in reality. Since the test-environment is on a machine with seven cores, we can expect that the througput would be higher when there is multiple clients sending requests concurrently.

All load-tests for this application was done with Apache JMeter.

Without latency simulation

No. threads	**No. requests**	**Throughput/sec**	**(KB/sec)**
1	100	990.099	17579.091
2	100	1470.588	26110.122
4	100	1886.792	33499.779
10	100	2631.579	46723.376
TOTAL	400	1486.989	26401.313

Table 2: Load-test for http-server

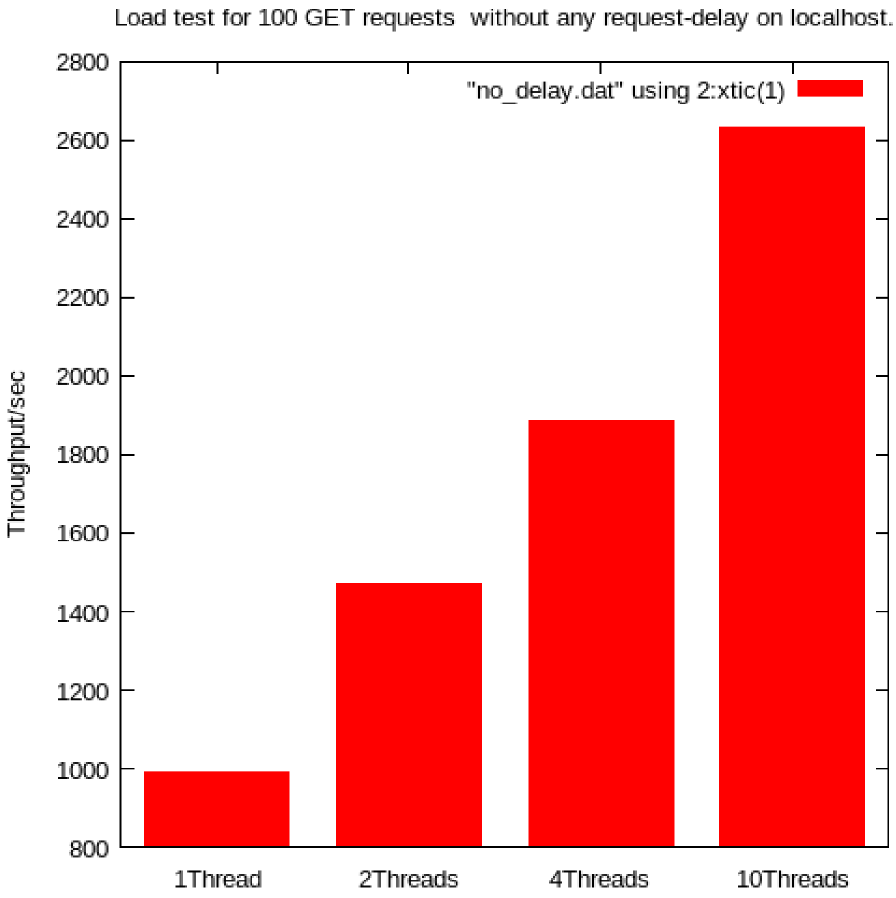

Figure 7: Throughput/sec for different number of threads

The result show that 10 threads gave more than doubled the througput compared to 1 thread. But it was'nt the result I expected.
I expected a linear growth in throughput with respect to number of threads until we reach ≈ 7 threads (which is the maximum number of threads that can run in parallell on the test-machine) where the througput would stabilize around some value.

With latency simulation
Since the server was running on my local machine it barely was'nt any latency between the requests at all, I figured that was the reason the tests did'nt match my expectations. To simulate network-latency that might occur outside of the test-environment I added a 200 millisecond delay at the server while handling the requests and re-did the tests to see what effect it gave.

No. threads	**No. requests**	**Throughput/sec**	**(KB/sec)**
1	100	4.946	87.817
2	100	9.870	175.253
4	100	19.685	349.505
10	100	49.189	873.334
TOTAL	400	10.672	189.486

Table 3: Load-test for http-server with latency simulation

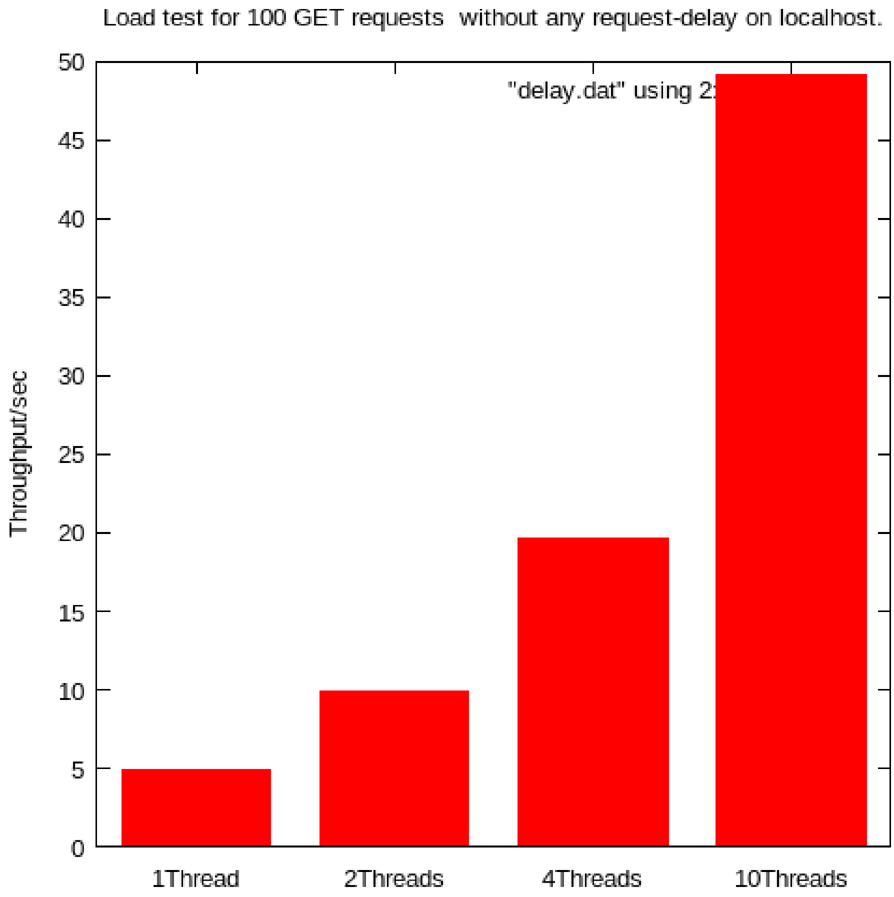

Figure 8: Throughput/sec for different number of threads

As can be seen from the benchmark-results with the latency-simulation, the througput is much lower, obviously.

What's interesting here is the througput gains we get by having multiple threads issuing the requests in parallell, we can see that 10 threads issuing GET-requests in semi-parallell give ≈ 10 times higher througput/sec than 1 thread.

5.1.5 How to run the application

The application consists of two parts: client and server, to run it:

1. build and compile the application - with your IDE or from terminal:

   ```
   mvn install
   ```

2. run the server - Startup.java contains the main-method of the server. Run it from inside your IDE or from terminal with:

```
java Startup
```

3. run the client - GuiController.java contains the main-method of the client. Run it from inside your IDE or from terminal with:

```
java GuiController
```

Note: by default the server will start listening on port 8080, if you want it to listen on another port you can give a portnumber upon initialization as a command-line argument.

5.2 Sub-assignment 2. A Multi-User Information System for NOG

A multi-user distributed application that allows storing, fetching and updating (delete, edit, add) participants data. The server is developed with Java RMI and the persistence layer is developed with JPA (Java Persistence API). The underlying database is PostgreSQL.

The server is multithreaded (with java rmi) and provides functionality to let many user at a time update and have local copies of the data where every update will be replicated among every other user that is connected.

The client is a simple Java RMI client with a Java Swing GUI.

5.2.1 Protocols used

Custom developed protocol for remote method calls between server and client.
Server-interface:

- getParticipants
- putParticipants
- addParticipant
- deleteParticipant
- editParticipant

Client-interface:

- updateParticipants

5.2.2 GUI

Same GUI as for sub-assigment 1.

5.2.3 Architecture

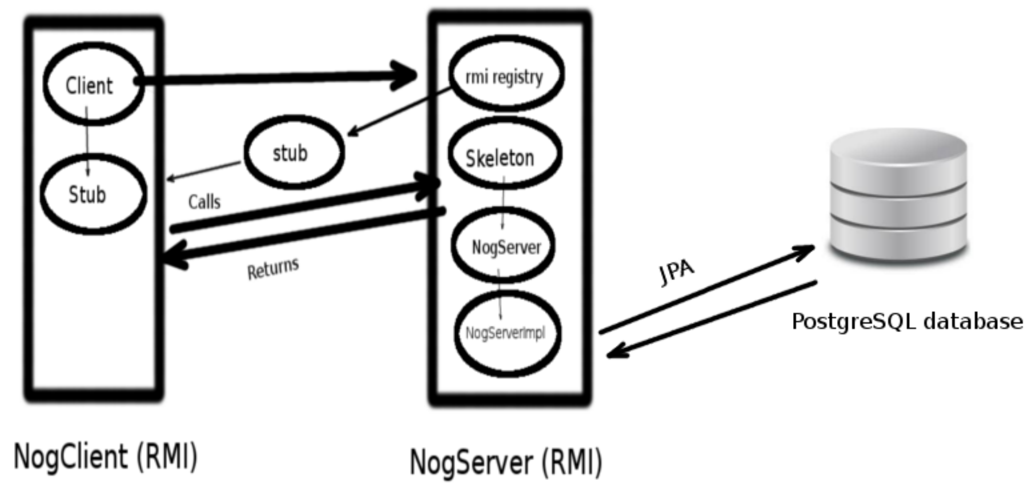

Figure 9: Application architecture (Three-tier-architecture).

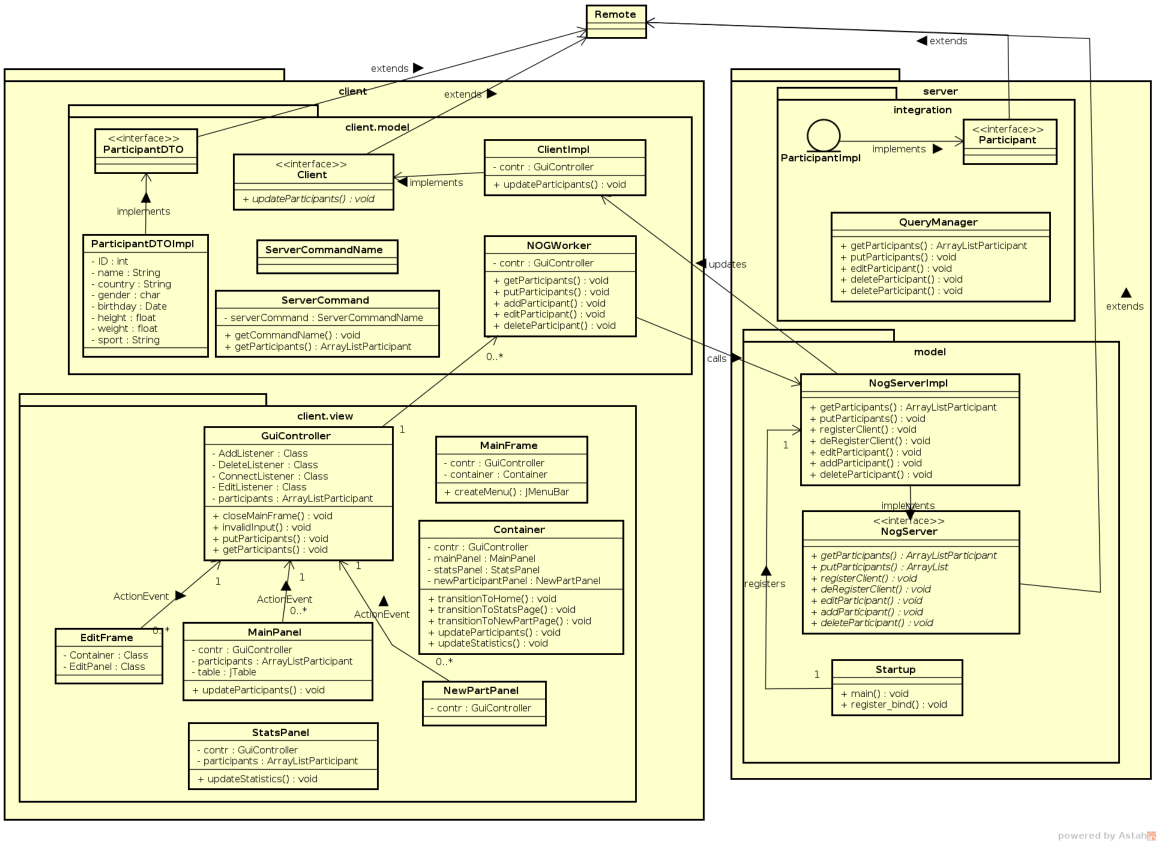

Figure 10: Class diagram over the application architecture.

5.2.4 Load-testing

The loadtesting done for this assigment is of another nature than the tests done for sub-assignment 1. Here i have created a custom LoadTesting class for issuing remote method-calls to the RMI-server all method-calls is done single-threaded. An important note here is that all of these functions contains database interaction, so besides the Java RMI server these tests also depend on the database-layer which is in PostgreSQL. Just like for the loadtesting done for sub-assignment, the number of calls done for each method is 100.

Method	No. calls	Time (s)
getParticipants	100	0.926
addParticipant	100	0.293
deleteParticipant	100	0.110
deleteParticipant	100	0.211

Table 4: Performance-test for rmi-server

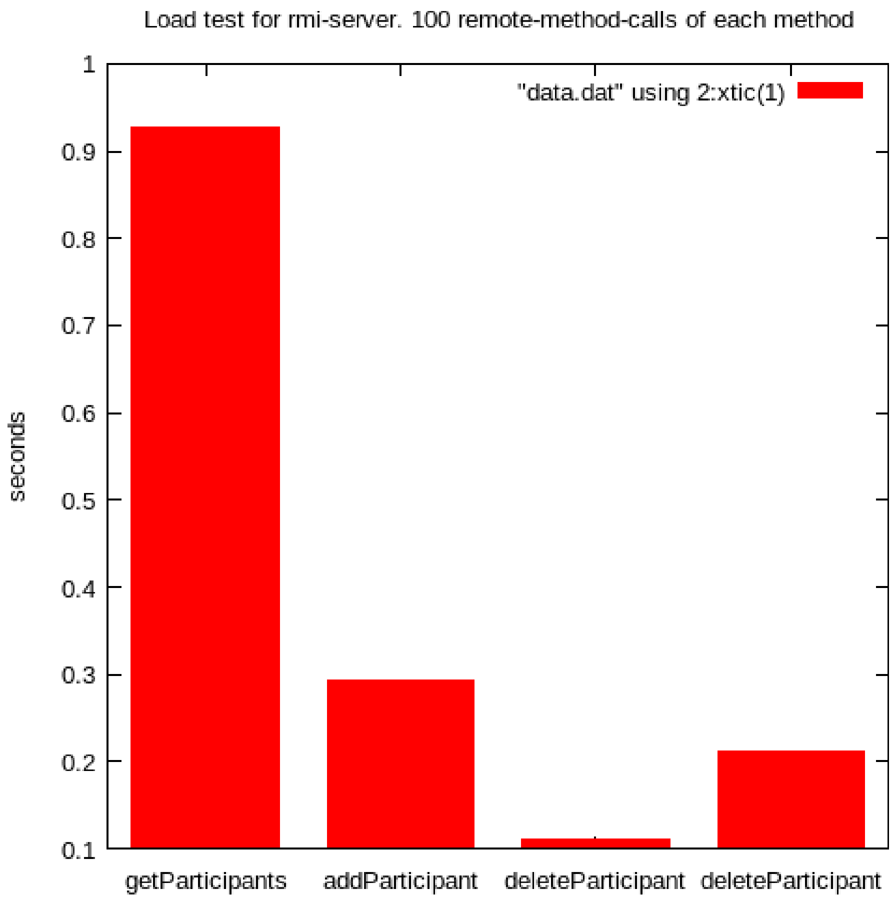

Figure 11: Benchmark results.

The result is not very suprising, *getParticipant* is by far the method that takes the most time and *deleteParticipant* takes the least.

This was not a sophisticated load-test but we can still see that the rmi-server is alot slower than the http-server in sub-assignment 1, if we convert the data in the table above we can get that the rmi-server can handle 100 calls for getParticipants in $\approx$ 0.92 seconds. In comparison with the load-test result from sub-assignment 1 (the single-threaded version) which could handle $\approx$ 990 GET-requestst per second, the rmi-server is way slower. The fact that the rmi-server is slower than the http-server is not suprising since the http-server simply reads from a tsv file while the rmi-server goes through many more steps: conversion from relational data to object-data with the ORM, compile psql-commands down to sql etc. but I did'nt expect the differencies to be this big.

5.2.5 How to run the application

The application consists of two parts: client and server, to run it:

1. build and compile the application - with your IDE or from terminal:

   ```
   mvn install
   ```

2. run the server - Startup.java contains the main-method of the server. Run it from inside your IDE or from terminal with:

   ```
   java Startup
   ```

3. run the client - GuiController.java contains the main-method of the client. Run it from inside your IDE or from terminal with:

   ```
   java GuiController
   ```

Note: This application uses rmi registry, if it is not already started when you run the server the server will start it. You can also explictly start rmiregistry before you run the server.

5.3 Sub-assignment 3. Chat Rooms for NOG

Distributed chat application for NOG users. The server is deveoped with java RMI and provides functionality for users to create/destroy/add chat-rooms, to Direct-Message other users and to block users.

The client is developed with java-rmi and is attached with a GUI developed with java Swing.

5.3.1 Protocols used

Custom developed protocol for remote method calls between server and client.
Server-interface:

- getClients

- registerClient
- deRegisterClient
- getChatRooms
- addChatRoom
- sendMessage
- joinChat
- destroyChatRoom
- leaveChatRoom
- privateChatRoom

Client-interface:

- updateClients
- updateChatRooms
- updateChat
- chatRoomDestroyed
- BlockClient
- unBlockClient
- getBlockedList

5.3.2 GUI

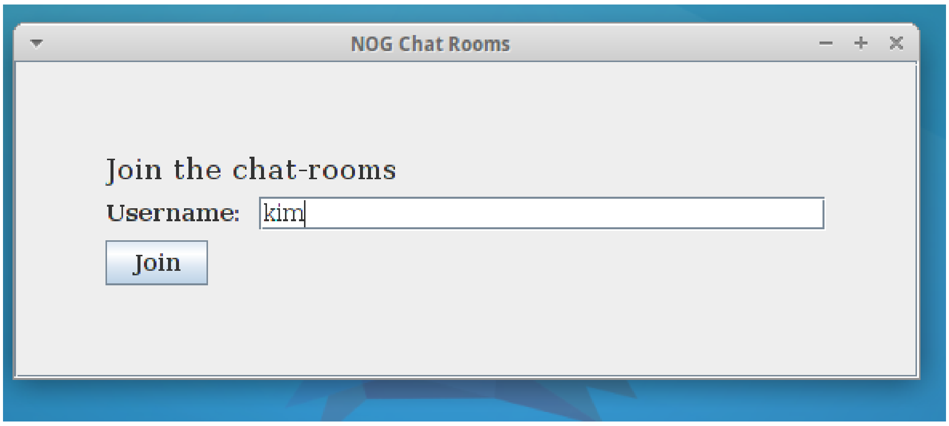

Figure 12: Screenshot of the startframe of the application

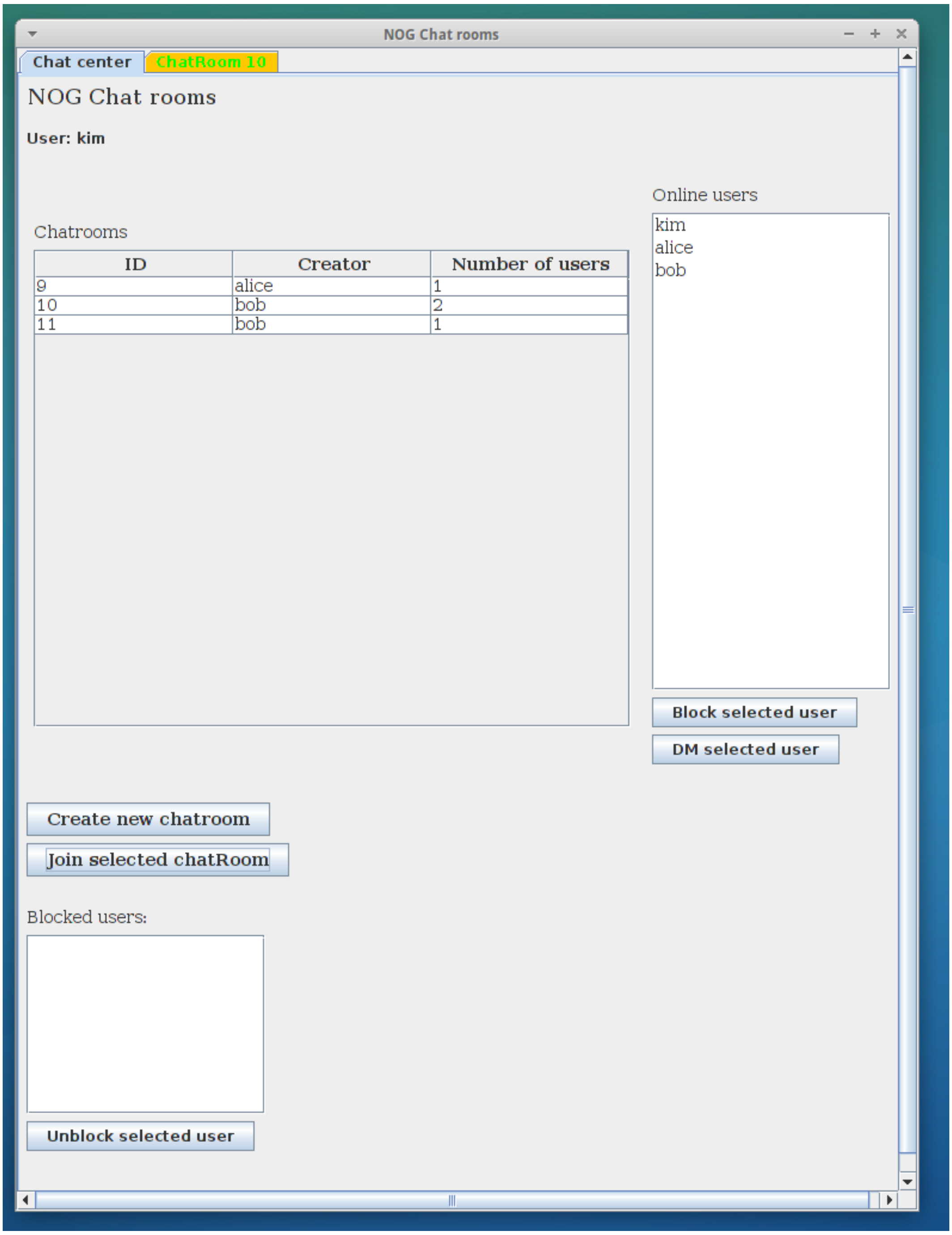

Figure 13: Screenshot of the MainPanel of the application

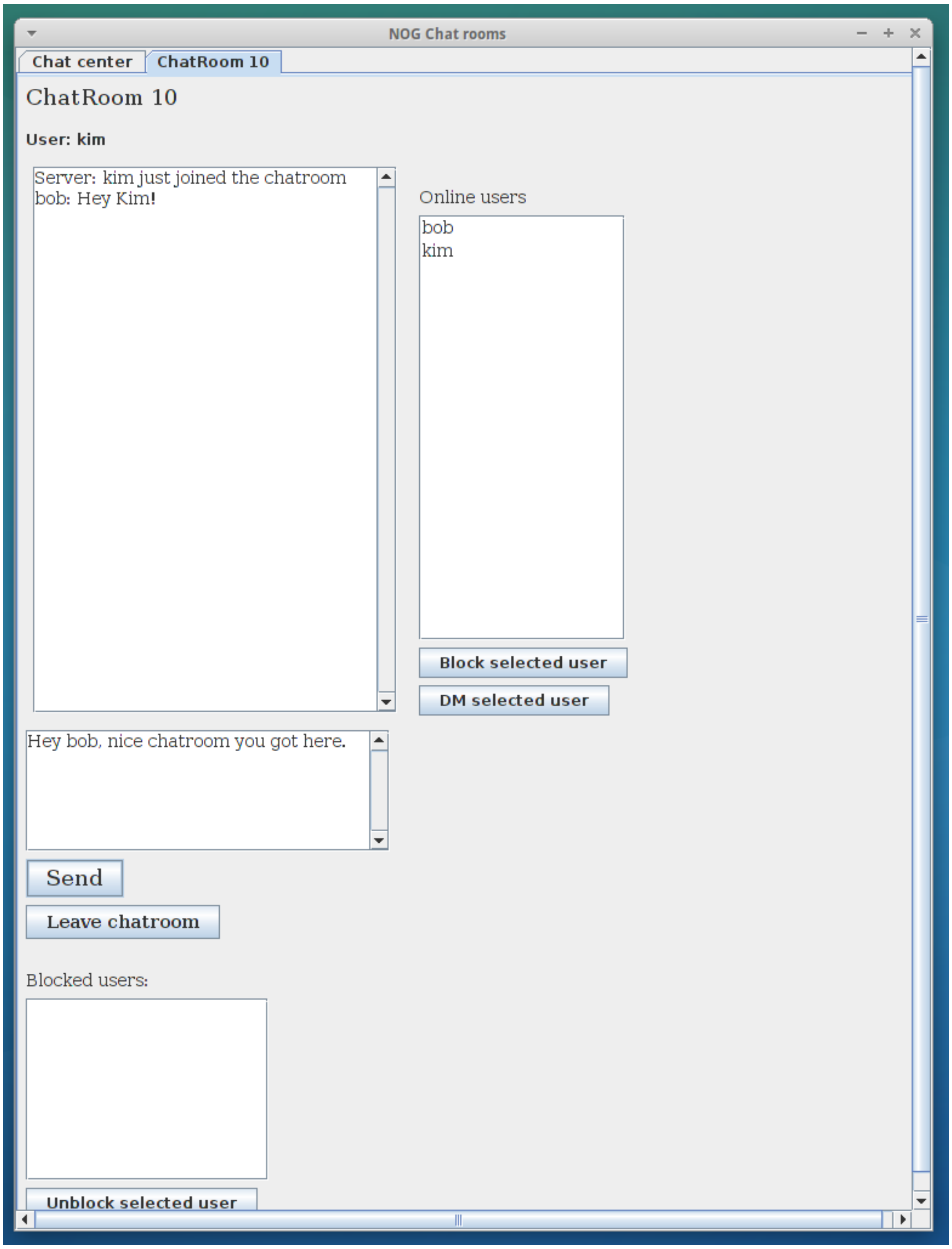

Figure 14: Screenshot of the MainPanel of the application

5.3.3 Architecture

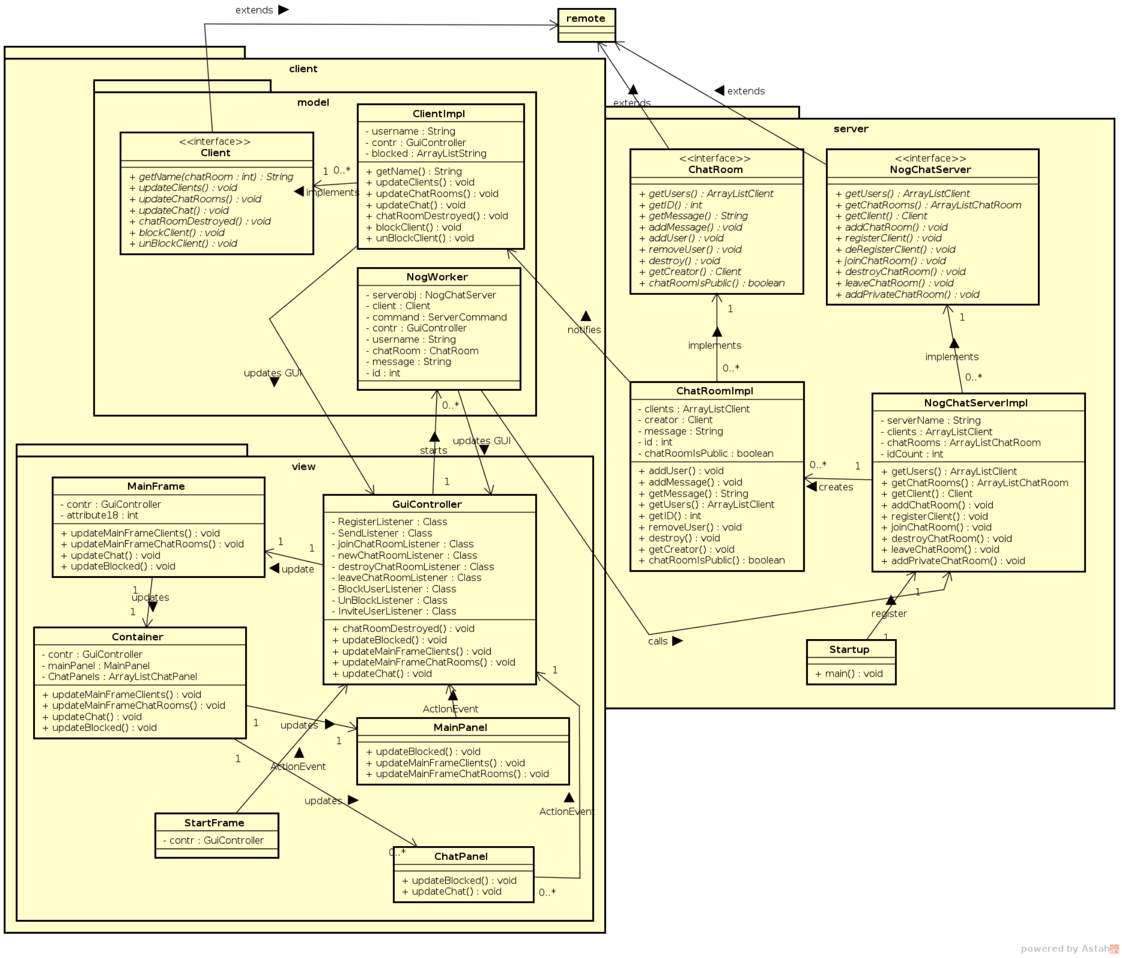

Figure 15: Class diagram over the application architecture.

5.3.4 How to run the application

The application consists of two parts: client and server, to run it:

1. build and compile the application - with your IDE or from terminal:

   ```
   mvn install
   ```

2. run the server - Startup.java contains the main-method of the server. Run it from inside your IDE or from terminal with:

   ```
   java Startup
   ```

3. run the client - GuiController.java contains the main-method of the client. Run it from inside your IDE or from terminal with:

```
java GuiController
```

Note: This application uses rmi registry, if it is not already started when you run the server the server will start it. You can also explictly start rmiregistry before you run the server.

6 Documentation

Sub-assignment 1. Api-docs

Sub-assignment 2. Api-docs

Sub-assignment 3. Api-docs

Note: the docs can be found in /apidocs (relative to each project file-structure)

References

[1] Royal Institute of Technology. Network programming in java. https://www.kth.se/social/course/ID2212/, 2015. [Online; accessed 7-Dec-2015].