Java for Interfaces and Networks

Project Report

By *Özgun Mirtchev*

Prof. Franziska Klügl & Dr. Owe Köckemann

Örebro University

January 23, 2017

Contents

1 Introduction

In this project a multi-agent simulation will be implemented. The agents will be clients that will be connecting to a running server with the simulation running, which will in turn draw it on a GUI. The model that was implemented in this program was the BOIDS flocking simulation. The different sections will guide the reader through some brief explanation of how the program works. In the Simulation-section one will be able to find information about the most important classes and will contain class diagrams and a sequence diagram over the whole process. In The Model-section, there will be some explanation on how the BOIDS-algorithm was implemented and various parameters that was used to get a good looking simulation of the agents. Lastly, there is a Discussion-section, where the writer will express their toughts on how the work was done, how it could be done better and which mistakes was done and how they were changed to make it better.

2 Simulation

This section will only contain a brief summary of what the most important classes do, namely Server (and its inner classes ClientThread, SimulationRunner) and Client. For more information on the other small classes, please refer to the JavaDocs.

2.1 Server

Creating the server, it was decided to keep the GUI-part out in the ServerGUI class, to create some abstraction. Most of the simulation work is done on the server, the GUI only displays the boids which are contained on the server. It was important to think about how to design the system. If we want to send and receive a boid from the server to the client and vice versa, we need one or two processes which handle this operation. For this reason, two threadpools was created with the ExecutorService class. One thread pool is handling the threads on the server, such as the listener for incoming client connections and the other for the simulation, which is running and synchronizing the connected clients while updating the flock of boids on the server, from the received boids. The other threadpool, will be running the client threads when invoked.

Whenever a client is connected, the created socket from the ServerSocket class is used to create a clientThread class, which implements Callable. This thread is added to a list of other Callables, which is invoked continuously in the simulation thread, with a fixed delay. The method invokeAll for the list of Callables returns a list of the what each Callable has produced, synchronously so each connected client is updated at the same time. The returned list contains the updated boids, from the client threads and is iterated through, to update the flock list on the server. A class diagram can be seen in figure 2.1.

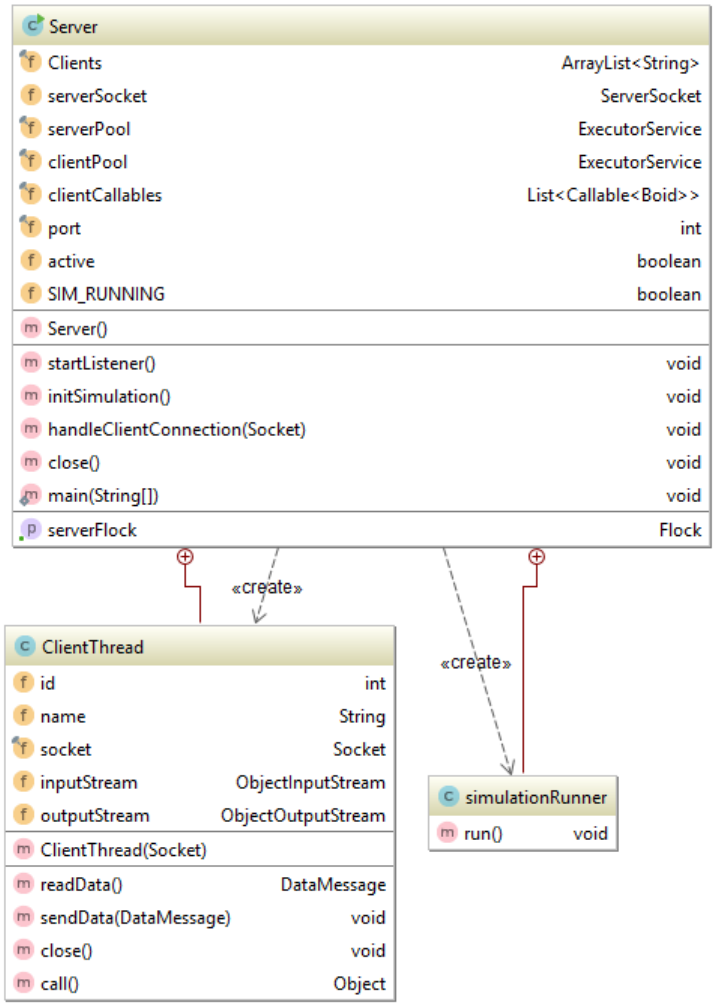

Figure 2.1: Server class diagram

2.2 Client

The client is a simple class. When starting the process, all the connections and streams to the server is established. When this is complete, a thread is started for listening to incoming objects from the server to the client. Before the client receives a boid, it receives the entire flock-list from the server. This is so that the client knows where it is compared to all other clients when it activates the update-method. When the values are updated, it then sent back to the server and yet again awaits a new object to do the same process over again. A class diagram can be seen in figure 2.2

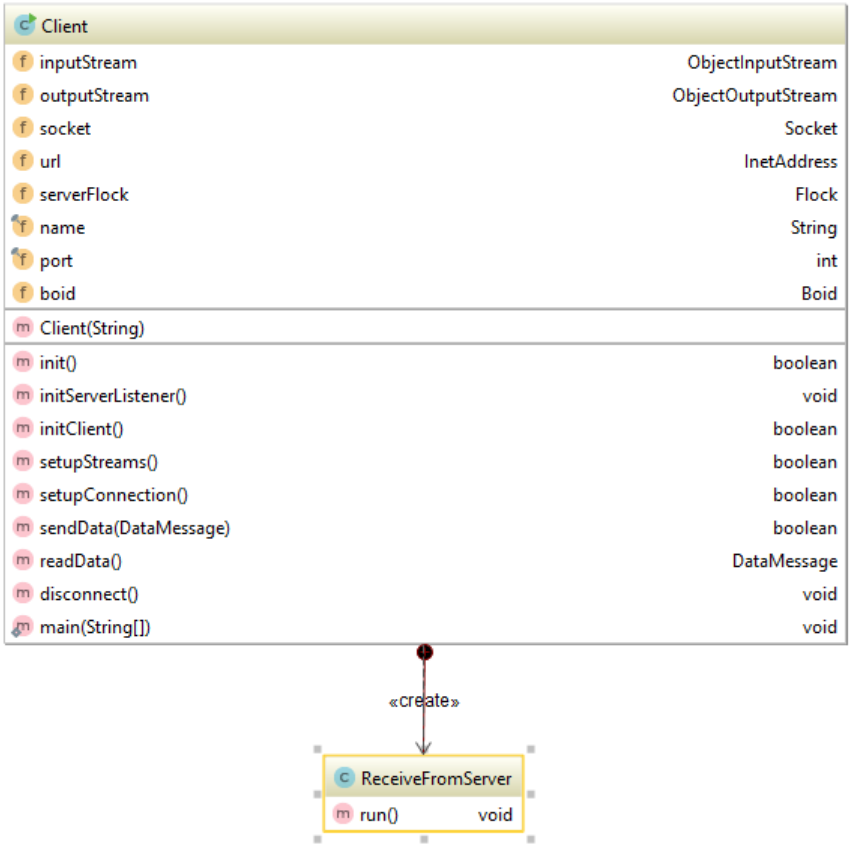

Figure 2.2: Client class diagram

2.3 Starting the process

The program is started from the main-method in the ServerGUI class. When the program is started, it first makes sure to create a connection to the server, before it creates the window. This is because a connection is immediately established when starting the program. If the connection fails, while starting it won't open a window. If the connection fails while running, it will shut down the GUI. When the connection is successful, a window is created as seen in figure 2.3.

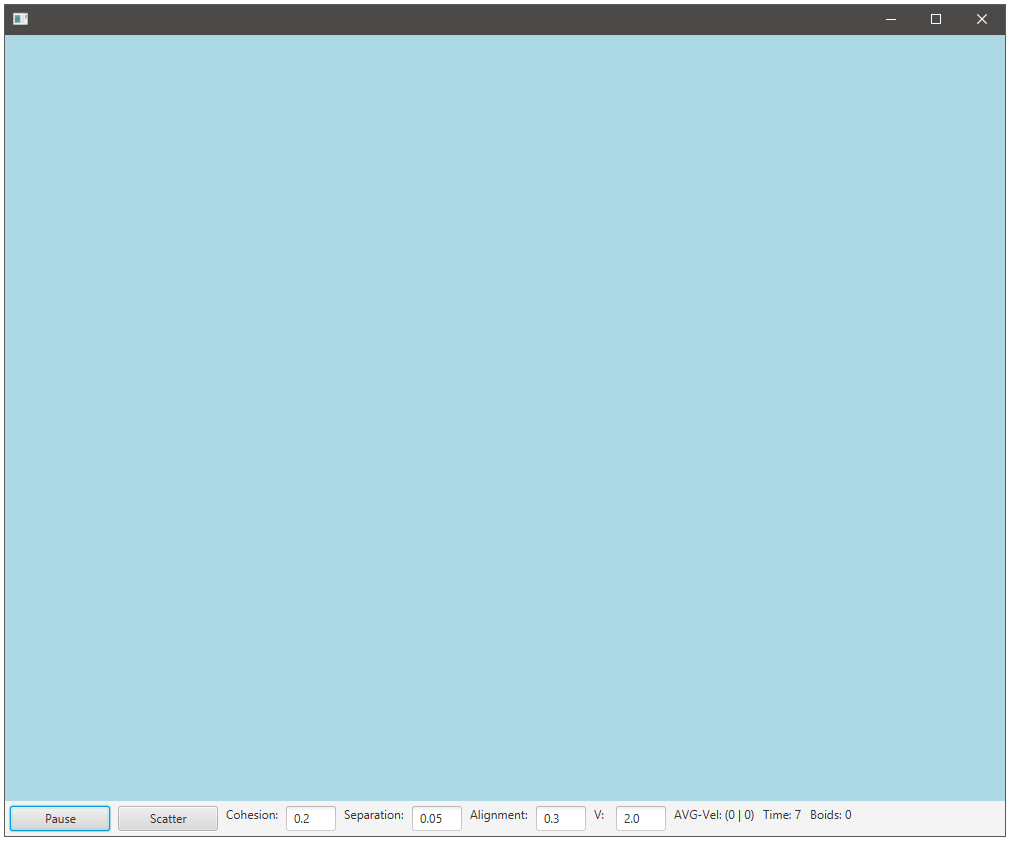

Figure 2.3: Program started

The GUI is simple, with a few buttons. One pause button to pause the simulation thread on the server and a scatter button, specifically for the boids. There are also some Textfields to be able to change the weights for the rules of the boids, which will be explained in the next section. At this point, the server is running and waiting for clients to connect. Starting a client-process can be done from the main-method of the Client class. When a client connects, it is registered at the server and added to the flock-container at the server. Since the GUI is listening to this container, it will draw the client-boid to the scene as seen in figure 2.4.

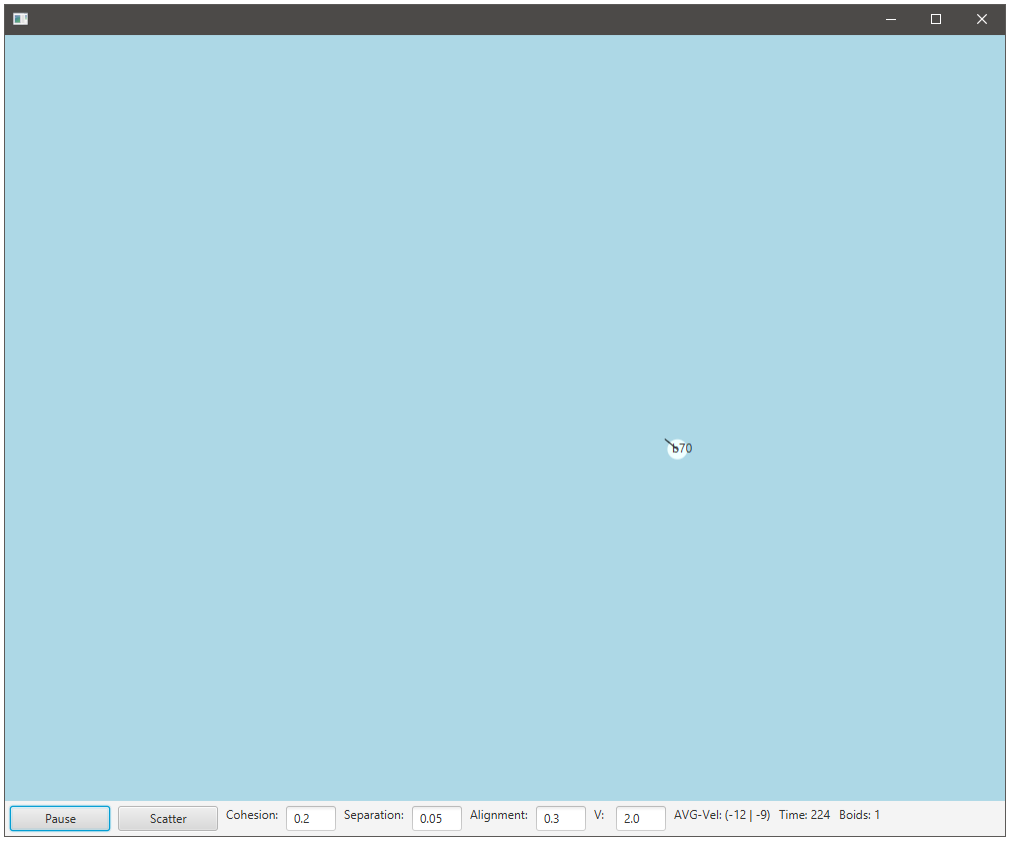

Figure 2.4: Client connected

As can be seen, this boid is of a Circle figure from JavaFX. It has the client ID on top of it, to reduce cluttering when they are grouped up, and a velocity vector drawn out, to show its heading. When more clients connect, they will show up randomly on the window and if they get close enough to another boid, they will move together as seen in figure 2.5.

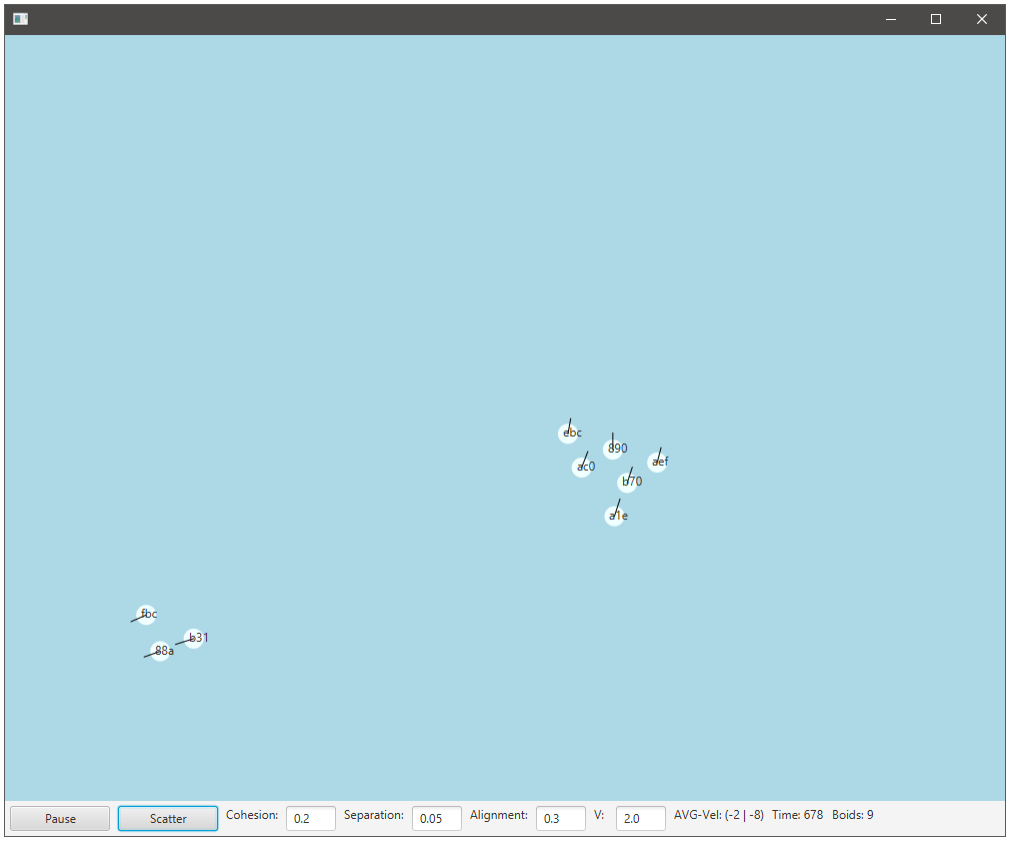

Figure 2.5: Clients moving together

2.4 Server-Client Sequence diagram

To be able to get a better view of how all the (major) classes integrate and which processes are called for each thread, a small sequence diagram was created seen in figure 2.6. It gives a better look of how the program runs but there were several things left out to make sure it fits on the paper, such as conditions and alternative processes if one fails. For this reason, the diagram was created with the mind that the program will be running as smooth as it can, without any errors and interuptions.

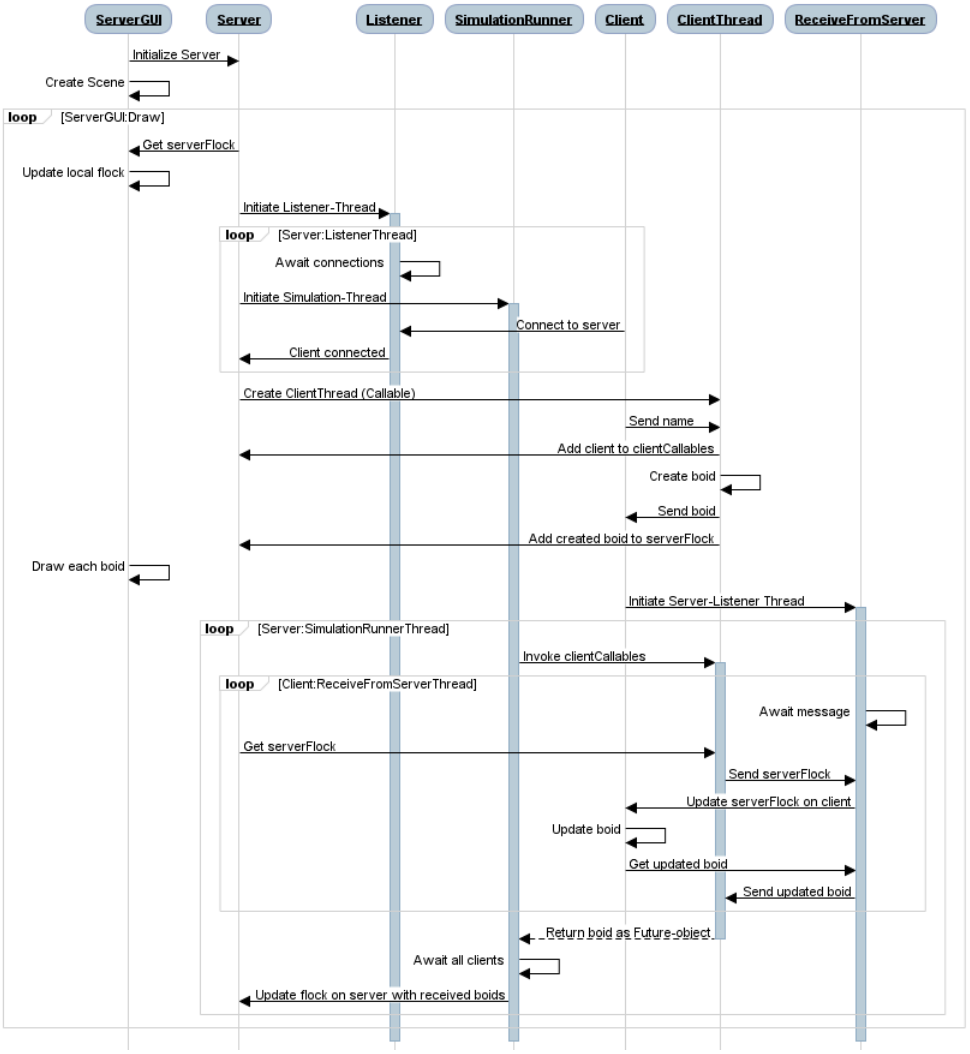

Figure 2.6: Server-Client Sequence Diagram

3 The Model

The chosen model was the boids flocking simulation. The implementation was mostly based on the algorithm written by Craig Reynolds. By using this model we want our agents to be able to behave like a flock, without running into each other and also have the same average speed, like a school of fish swimming in the ocean. A custom-made Vector class, Vector2, was also created, which enables a better overview of what really is going on when the vectors are computed. Also it makes it easier to change things internally if it is needed.

The algorithm is made by using three important rules:

3.1 Rules

Cohesion

With this rule each boid looks around a smaller radius, set by a constant, and averages the position of every other boid in range to create a center of mass. This makes the boids turn the direction towards the center of all the boids. Without this, the boids will not flock together tightly.

Separation

This rule takes care of the separation between the boids. It prevents the boids from running over each other, and not move too close to one another. Without this rule, they would all stack on each other.

Alignment

To enable the boids to move around together, every boid in range needs to average the velocity among them. Otherwise, the boids will try to separate themselves because of their individual velocity. If the Cohesion is enabled while this one is disabled, they will still stick around together for a bit, but eventually they will separate from the flock.

3.2 Weights

To ameliorate some issues with the rules, they have a weight added to them, so that it will be possible to decide how much of each rule really contributes to the vector changes. The parameters will vary of course, depending on how many boids there are around and what value the other weights has. It is important have a good balance, and I'm sure there are better ways to calibrate the weights but overall running some tests and by deciding visually, the following weight values was decided:

- **Cohesion** was set to 0.2 to prevent the flock from trying to get into each other too much. It alleviated some of the problems that some boids wanted to get into the flock very badly but as soon as they got in with a high velocity the separation rule kicked in and also produced a high value to prevent it from going into the others in the flock and caused a more ping-pong result. The alignment weight is also helping with slowing the boid down before the separation rule is applied.
- **Separation** was set to 0.05 to prevent the flock from being too jerky, since they tried to separate too much from each other.

- **Alignment** was set to 0.3 to give the flock a more natural and soft turn-around when joining other boids.

4 Discussion

Overall, I think my design is fine for this task, however there are still some bugs present and other improvements I'd like to do, had I more time. It took quite a long time for me to learn how JavaFX works, and I tried to use FXML-files as well for my implementation, however, it mostly made it a bit more confusing and produced more work than what was necessary. I'd like to have seen a better implementation in terms of architecture as well, such as MVC. Right now, it's not very pretty code and it is quite messy, with some spaghetti-code. Had I known from the start what I was heading into and how I would build, I think it would've made it easier to see what to do ahead and easier to implement the classes.

Before I knew about ExecutorService, I had to synchronize threads by settings booleans everywhere, on the server as well as the client. It was not very pretty. However, when I learned how to use the class, I was able to execute the threads to make them return the needed values, synchronously. Using the Future class also made it a lot easier to handle the returned values from the threads. I wish I had known it before I had finished implementing the server, it would've saved a lot of time and confusion when reading the code. Since it was awhile ago I had worked with socket connections as well, it took some time to relearn how it works and also to make it work in Java. Fortunately, it isn't very difficult once you've started.

Upon starting the program, it was mentioned that it was making sure that the server connection was established at first, otherwise it is shut down. It is debatable whether it is really necessary to shut down the GUI entirely, but because of lack of time, there wasn't a better way to do it at this time. Perhaps a better way would be to set a timeout value for the GUI, which was still periodically checking if the server was up or not, and connect automatically if the server happen to be running.

Testing the program, was done to some extent. I tried using JUnit test-unit for my Vector-class however, it took a bit of time to set it up for me so I didn't continue with it. I relied mostly on print-outs and logs to control that the values that was received and sent, was correct. Perhaps not the most conventional way to control variable values... I did have plans to use assert to make sure some containers had consistent values, however it was also skipped to save time so that I don't need to remove them later.

Overall it was an interesting project. It helped me learn a lot more about multi-agent systems and it was fun to implement the boids model.

Thank you.