Software architecture document for project “Robotics”

1. Authors

V. Chernikov

O. Markelov

S. Morozov

I. Shatalov

1. Goals and limitations

1.1. Key functional requirements

* Users must be able to upload their solutions and receive simulation results.
* Teachers must be able to edit levels and create new ones.
* Previous solutions should be stored until they are obsolete.
* Users should have accounts with associated previous solutions and results.

1.2. Non-functional requirements

*1.1. Environment*

*● The system should run smoothly on machines with 4 GB RAM and a quad-core x86-64 processor.*

*● The application can be used on any device that has a JVM (Java Virtual Machine) installed and that is connected to the internet.*

*● A student can write scripts for the robot in Groovy programming language.*

*● A teacher can create a new level describing it in Java/Groovy programming language.*

*1.2. Performance*

*● Calculation time in standard situation should be under 2 seconds. Only exceptionally may take up to 10 seconds.*

*● One server should be enough for at least 60 students.*

*1.3. Reliability*

*● The server should work during a week without restart.*

*1.4. Extensibility*

*● The client application not dependent on JVM can be created using the same server API.*

*● The amount of levels calculated at the same time can be increased by adding extra computers to the system.*

1.3. Architectural goals

* There must be multi-user remote access with Web-UI.
* Users must be able to access the system using various modern web browsers: Mozilla Firefox 52+, Google Chrome 69+, Microsoft Edge 17+.
* Up to 200 simultaneous simulations must be supported that is virtually impossible by a single simulation unit.
* The server side must operate on Windows and Linux.
* Users should not be able to run malicious code on the server or simulator units.
* Simulator units should be monitored, if one of them is down server should avoid requesting simulations on it.
* Privileged users should have ability to dynamically add or remove simulator units.
* Server should try to load simulator units equally or as close as possible.
* Server should have a persistent storage of accounts, levels and solutions.
* Possibility of future support of other languages for solutions.
* Possibility of working with custom clients other than the default client.

1.4. Additional goals, restrictions and preferences

* All team members are currently studying Java on the Object-oriented programming course, therefore it is better to use this language if it is possible.
* All team members have completed the introductory course on SQLite databases and would like to use the acquired skills in practice.
* One of the team members is familiar with HTML5 + CSS3 + JS and wants to improve his jQuery and AJAX knowledge, therefore web application seems to be the best option for client-side part of the project.
* As the client-application is supposed to be used mostly for programming, it seems enough to support only screens with a width of at least 1080 pixels.

2. Goals analysis

2.1. Security

Due to multi-user nature of the system, there must be session support, each session must be authenticated and each operation must be authorized.

There are 3 main user roles in the system:

* Student is a user, who can do the following: log in/out the system; create new lobbies or join existing ones; write a programming code for the robot, send it to the server and get a report; watch his previous solutions.
* Teacher is a user, who inherits all the abilities from Student in addition to the ability to manage student accounts and edit levels as well.
* Administrator is a Teacher, who can add and remove simulator units.

We want administrators to be able to modify, delete and create accounts, so we are going to store login and password hash in local database.

Another important area is code execution as we are going to execute solutions on remote simulators. Administrator should create a whitelist of permitted imports that are needed to solve tasks and can’t be used for malicious purposes. Simulator should recognize if illegal package is imported and abort execution.

2.2. Programming language for solution scripts

There are a number of languages suitable for writing scripts in them. Their list includes the following:

* JavaScript
* Python
* Groovy
* Perl
* etc.

It is also possible to create our own language specifically for this project, but that is a much harder thing to do compared to using an existing one. Moreover, using an existing language is much more useful for students.

The preferred programming language for the project is Java, and Groovy can be easily integrated with Java, therefore Groovy seems to be the best option for that purpose. It would be good to store language’s name alongside the solution, so we would be able to integrate other languages later.

2.3. OS support

The server side of the project should run both on Windows and Linux machines. Although it is possible to achieve by writing separate code each of these systems in a language such as C++, it would be hard both to develop and maintain the software because any change in the system would result in changes in several places.

Another solution is to use a virtual machine. The best option seems to be the Java Virtual Machine which runs both on Windows and Linux (as well as many other operating systems).

2.4. Web browser support

Despite the fact that web application should have a cross-browser user interface, there is no goal to make it pixel-perfect. Also there is no goal to make a layout of the pages responsive, as the support of the mobile browsers or small screens is not required. Therefore, there is no need to use any UI-framework or library, and the best option to build a layout is to use only plain HTML5 + CSS3.

The web pages of the application must dynamically update its data and regularly interact with the server via AJAX. What is important, this behavior must be absolutely the same in any supported web browser. Therefore, it seems to be the best option to use some JavaScript library or framework like jQuery, React, Angular, etc.

2.5. Client extensibility

It should be possible to use a custom client to work with the system. That can be done if the default client and the server use an API to interact with each other. The API therefore can be used by other applications without any complications.

2.6. Language extensibility

As the system is expected to be able to work with solutions in multiple languages in future, it should be possible to easily add support for new languages.

3. Solution description

3.1. Modules and subsystems

The system consists of three main components:

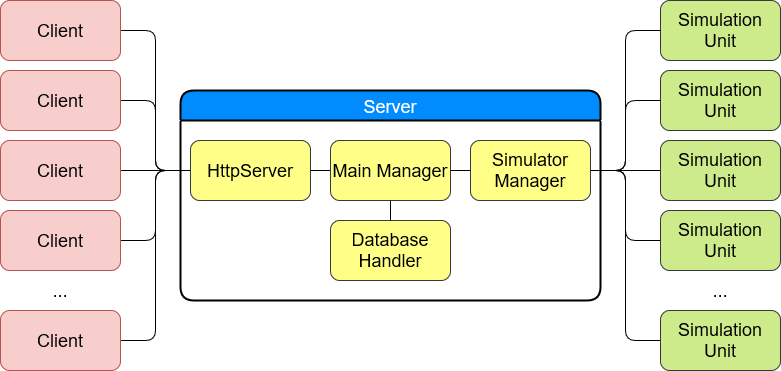
* Server — the core part of the system which manages users, levels and simulations. Implementation language is Java which has a high degree of platform independency. The server provides an API for clients to interact with it.
* Clients — users’ computers that connect to the server to interact with it (i.e. submit solutions). They use web browsers to interact with the server via AJAX. All the Web pages provide a static design layout (with fixed width of 1040 pixels), made with plain HTML5 + CSS3. Also JavaScript + jQuery are used for interacting with the user.
* Simulation unit — the entity that executes the simulations of levels.

The server can be divided into the following subsystems:

* HTTP server which is used to interact with clients using HTTP requests and to call the Main Manager according to the requests.
* Database Handler — the part which saves and loads data to/from database. It is required for data which is meant to be stored for a long period of time and survive the whole system restarts (i.e. students’ accounts, information about their solution attempts, levels, etc.).
* Simulation Manager is used to interact with Simulation Units. It sends the data for each simulation to the most suitable Simulation Unit. Once a simulation is finished, it collects the information about its outcome from the corresponding Simulation Unit.
* Main Manager — the part which connects the aforementioned subsystems with each other and puts the whole system together.

The students’ solutions are scripts written in Groovy programming language.

3.2. Deployment

**

*Fig.1 — System structure*

The system is deployed as shown in fig.1, its main components are Server and one or more Simulation Units. Server and Simulation Unit are separate applications that exchange data using HTTP REST API, therefore they can be deployed on different machines. Both Server and Simulation Unit are multithreaded for better performance, so there is no point in running several Simulation Units on one machine. Server is capable of utilizing several Simulation Units at once.

Simulation Units are reactive, they only respond to incoming requests. Requests can either start new simulation or get current status of Simulation Unit (number of running simulations and some additional information).

Server is reactive for Users and active for Simulation Units, Clients make HTTP requests to the Server, Server processes request, makes request to the Simulation Unit if needed, and returns suitable result to the Client. Also Server requests current status of each connected Simulation Unit in the background to find best candidate for the next simulation and detect if any of Simulation Units are down.

4. Key architectural elements

4.1. Authentication protocol

For authentication we will use Password Authentication Protocol as it is simple yet effective. On the landing page user will be asked to type username and password that will be sent to the server for checking and then response shall be received.

Password and username are sent in plaintext, but we may switch to using https in the future. When server receives password it calculates its hash and compares to the one stored in the DB.

4.2. Authorization

Type of account will be stored alongside the username, password hash and some additional information. This type indicates if user is a student, teacher or administrator. According to this type user is granted permissions.

4.3. Fault-tolerant network interaction

Simulator Manager will periodically check status of simulation units. If unit is offline it is marked as inactive and next simulations should be sent to different simulation units.

4.4. Modules API

4.4.1. Client - HTTP Server RESTful API

|  |  |  |
| --- | --- | --- |
| **Object** | **Method** | **Description** |
| sign | login | Logs a user in the system if his username exists in the database.  Parameters:  - username – user unique name.  - password – user password. |
| logout | Logs a user out of the system if his username exists in the database.  Parameters:  - username – user unique name.  - password – user password. |
| lobbies | get | Returns list of available lobbies.  Notes:  The list of lobbies is sorted by date of creation (the newest one - in the head). |
| levels | get | Returns list of created levels. |
| solutions | get | Returns list of all the solutions of specified user.  Parameters:  - username – user unique name. |
| lobby | join | Places a user in the lobby gotten by specified 'lobbyID'.  Parameters:  - username – user unique name.  - lobbyID – lobby's unique id. |
| create | Creates a new lobby by 'levelID' and places a user in it.  Parameters:  - userName – user unique name.  - levelID – level unique id.  - playersAmount – amount of players. |
| leave | Removes a user from the lobby gotten by specified 'lobbyID'.  Parameters:  - userName – user unique name.  - lobbyID – lobby's unique id. |
| return | Returns the lobby gotten by specified 'lobbyID'.  Parameters:  - userName – user unique name.  - lobbyID – lobby's unique id. |
| submit | Compiles the specified code and returns compile result.  Parameters:  - userName – user unique name.  - lobbyID – lobby's unique id.  - code – a code to compile.  Notes:  If the compilation is successful, the code is being saved for the future simulation. |
| code | edit | Cancels the submission of the lately compiled code and returns the code itself.  Parameters:  - userName – user unique name.  - lobbyID – lobby's unique id. |
| simulationResult | isReady | Returns whether the simulation has already been finished.  Parameters:  - lobbyID – lobby's unique id. |
| get | Returns the simulation result or null in case it hasn't been processed yet.  Parameters:  - userName – user unique name.  - lobbyID – lobby's unique id. |

4.4.2. HTTP Server - Main Manager API

HTTP Server - Main Manager API is a regular Java API, whose methods are identical to Client - RESTful API described in the previous paragraph. There are also several interfaces for objects, used in MainManager interface.

public interface MainManager {

*/\*\**

*\* Logs a user in the system if his username exists in the database.*

*\**

*\* @param username user unique name.*

*\* @return whether the user is found in the database and successfully logged in.*

*\*/*

boolean login(String username, String password);

*/\*\**

*\* Logs a user out of the system if his username exists in the database.*

*\**

*\* @param username user unique name.*

*\* @return whether the user is found in the database and successfully logged out.*

*\*/*

boolean logout(String username);

*/\*\**

*\* Returns list of available lobbies.*

*\**

*\* The list of lobbies must be sorted by date of creation (the newest one - in the head).*

*\**

*\* @return list of available lobbies.*

*\*/*

List<Lobby> getLobbies();

*/\*\**

*\* Returns list of created levels.*

*\**

*\* @return list of created levels.*

*\*/*

List<Level> getLevels();

*/\*\**

*\* Returns list of all the solutions of specified user.*

*\**

*\* @param userName user unique name.*

*\* @return list of all the solutions of specified user.*

*\*/*

List<Solution> getSolutions(String userName);

*/\*\**

*\* Places a user in the lobby gotten by specified 'lobbyID'.*

*\**

*\* A host-user must be in the head of the list.*

*\**

*\* @param userName user unique name.*

*\* @param lobbyID lobby's unique id.*

*\* @return lobby which user was placed in.*

*\*/*

Lobby joinLobby(String userName, int lobbyID);

*/\*\**

*\* Creates a new lobby by 'levelID' and places a user in it.*

*\**

*\* A host-user must be in the head of the list.*

*\**

*\* @param userName user unique name.*

*\* @param levelID level's unique id.*

*\* @param playersAmount amount of players.*

*\* @return the created lobby.*

*\*/*

Lobby createLobby(String userName, int levelID, int playersAmount);

*/\*\**

*\* Removes a user from the lobby gotten by specified 'lobbyID'.*

*\**

*\* @param userName user unique name.*

*\* @param lobbyID lobby's unique id.*

*\* @return whether the user has been successfully removed from the lobby.*

*\*/*

boolean leaveLobby(String userName, int lobbyID);

*/\*\**

*\* Returns the lobby gotten by specified 'lobbyID'.*

*\**

*\* A host-user must be in the head of the list.*

*\**

*\* @param userName user unique name.*

*\* @param lobbyID lobby's unique id.*

*\* @return the lobby.*

*\*/*

Lobby returnToLobby(String userName, int lobbyID);

*/\*\**

*\* Compiles the specified code and returns compile result.*

*\**

*\* If the compilation is successful, the code is being saved for the future simulation.*

*\**

*\* (A simulation itself starts automatically when all the users successfully submitted the code)*

*\**

*\* @param lobbyId lobby's unique id.*

*\* @param username user unique name.*

*\* @param code a code to compile.*

*\* @return the result of compilation.*

*\*/*

CompileResult submit(String username, String code, int lobbyId);

*/\*\**

*\* Cancels the submission of the lately compiled code and returns the code itself. In case the*

*\* user hasn't submitted any code yet, returns null.*

*\**

*\* @param username user unique name.*

*\* @param lobbyId whether the submission was successfully cancelled.*

*\* @return the earlier submitted code.*

*\*/*

String editSubmittedCode(String username, int lobbyId);

*/\*\**

*\* Returns whether the simulation has already been finished.*

*\**

*\* @param lobbyId lobby's unique id.*

*\* @return whether the simulation has already been finished.*

*\*/*

boolean isSimulationFinished(int lobbyId);

*/\*\**

*\* Returns the simulation result or null in case it hasn't been processed yet.*

*\**

*\* @param username user unique name.*

*\* @param lobbyId lobby's unique id.*

*\* @return simulation result or null in case it hasn't been processed yet.*

*\*/*

SimulationResult getSimulationResult(String username, int lobbyId);

}

public interface Level {

*/\*\**

*\* Returns the unique id of the level.*

*\**

*\* @return the unique id of the level.*

*\*/*

int getId();

*/\*\**

*\* Returns the address of the level's icon.*

*\**

*\* @return the address of the level's icon.*

*\*/*

String getIconAddress();

*/\*\**

*\* Returns the level's name.*

*\**

*\* @return the level's name.*

*\*/*

String getName();

*/\*\**

*\* Returns the level's difficulty.*

*\**

*\* @return the level's difficulty.*

*\*/*

String getDifficulty();

*/\*\**

*\* Returns the level's type.*

*\**

*\* @return the level's type.*

*\*/*

String getType();

*/\*\**

*\* Returns the level's description.*

*\**

*\* @return the level's description.*

*\*/*

String getDescription();

*/\*\**

*\* Returns the level's rules.*

*\**

*\* @return the level's rules.*

*\*/*

String getRules();

*/\*\**

*\* Returns the level's goal.*

*\**

*\* @return the level's goal.*

*\*/*

String getGoal();

*/\*\**

*\* Returns minimal level's amount of players.*

*\**

*\* @return minimal level's amount of players.*

*\*/*

int getMinPlayers();

*/\*\**

*\* Returns maximal level's amount of players.*

*\**

*\* @return maximal level's amount of players.*

*\*/*

int getMaxPlayers();

}

public interface Solution {

*/\*\**

*\* Returns the level.*

*\**

*\* @return the level.*

*\*/*

Level getLevel();

*/\*\**

*\* Returns the list of simulation results.*

*\**

*\* All the simulation results must be sorted by date (the newest one - in the head).*

*\**

*\* @return the list of simulation results.*

*\*/*

List<SimulationResult> getSimulationResults();

}

public interface Lobby {

*/\*\**

*\* Returns the lobby's unique id.*

*\**

*\* @return the lobby's unique id.*

*\*/*

int getId();

*/\*\**

*\* Returns the address of host's avatar icon.*

*\**

*\* @return the address of host's avatar icon.*

*\*/*

String getHostAvatarAddress();

*/\*\**

*\* Returns the host's name.*

*\**

*\* @return the host's name.*

*\*/*

String getHostName();

*/\*\**

*\* Returns the amount of players present in the lobby.*

*\**

*\* @return the amount of players present in the lobby.*

*\*/*

int getCurrentPlayersAmount();

*/\*\**

*\* Returns the maximal amount of players for this lobby.*

*\**

*\* @return the maximal amount of players for this lobby.*

*\*/*

int getAcceptablePlayersAmount();

*/\*\**

*\* Returns the list of players.*

*\**

*\* @return the list of players.*

*\*/*

List<Player> getPlayers();

*/\*\**

*\* Returns the level.*

*\**

*\* @return the level.*

*\*/*

Level getLevel();

}

public interface Player {

*/\*\**

*\* Returns the address of player's avatar icon.*

*\**

*\* @return the address of player's avatar icon.*

*\*/*

String getAvatarAddress();

*/\*\**

*\* Returns the player's unique name.*

*\**

*\* @return the player's unique name.*

*\*/*

String getName();

*/\*\**

*\* Returns whether the player has submitted his solution.*

*\**

*\* @return whether the player has submitted his solution.*

*\*/*

boolean isSubmitted();

}

public interface CompileResult {

*/\*\**

*\* Returns whether the compilation has been successful.*

*\**

*\* @return whether the compilation has been successful.*

*\*/*

boolean isCompiled();

*/\*\**

*\* Returns whether the simulation has been processed.*

*\**

*\* @return whether the simulation has been processed.*

*\*/*

boolean isSimulated();

*/\*\**

*\* Returns the compile message.*

*\**

*\* @return the compile message.*

*\*/*

String getMessage();

}

public interface SimulationResult {

*/\*\**

*\* Returns the unique id of simulation result.*

*\**

*\* @return the unique id of simulation result.*

*\*/*

int getId();

*/\*\**

*\* Returns whether the user robot has reached the goal.*

*\**

*\* @param username user unique name.*

*\* @return whether the user robot has reached the goal.*

*\*/*

boolean isSuccessful(String username);

*/\*\**

*\* Returns the date of simulation result.*

*\**

*\* @return the date of simulation result.*

*\*/*

Date getDate();

*/\*\**

*\* Returns user simulation log.*

*\**

*\* @param username user unique name.*

*\* @return user simulation log.*

*\*/*

String getLog(String username);

}

4.4.3. Main Manager - Simulator Manager API

Main Manager - Simulator Manager API is a regular Java API, whose function is to control available simulator units and run tasks on them.

public interface SimulatorManager {

*/\*\**

*\* Add new simulator unit in simulator pool.*

*\**

*\* @param url location of new simulator unit.*

*\* @return true if SU was added, false otherwise.*

*\*/*

boolean addSimulator(String url);

*/\*\**

*\* Remove simulator unit from the simulator pool.*

*\**

*\* @param url location simulator unit to remove.*

*\* @return true if SU was added, false otherwise.*

*\*/*

boolean removeSimulator(String url);

*/\*\**

*\* Get list of all available SU.*

*\**

*\* @return list of simulator units.*

*\*/*

ArrayList<String> getSimulatorsList();

*/\*\**

*\* Run simulation on available simulation unit.*

*\**

*\* @param levelId filename of the level.*

*\* @param lobbyId id of lobby, used to define id of simulation.*

*\* @param solutions map of players and their solutions.*

*\* @return result of the simulation.*

*\*/*

SimulationResult runSimulation(String levelId, int lobbyId, Map<Player, String> solutions);

}

4.5. External plug-in API

To transmit objects through the raw stream JSON (de-)serialization must be used. Gson [<https://github.com/google/gson>] library is suggested for this.

5. Platform

Client platform: Web browser (Mozilla Firefox 52+, Google Chrome 69+, Microsoft Edge 17+), display with a width of at least 1080 pixels.

Language: Java on server, HTML5 + CSS3 + JavaScript on Web-based client, Groovy for solutions.

Server frameworks and libraries: MySQL as DBMS, Gson for JSON (de-)serialization.

Web client frameworks and libraries: jQuery.