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 define their robot’s behavior using a programming language.*

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

*1.2. Performance*

*● Calculation time in a 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 for a week without a restart.*

*1.4. Extensibility*

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

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

1.3. Architectural goals

* Programming language for solution scripts: The programming language for students’ solutions must be easy to integrate into the system and suitable for educational purposes.
* Web browser support: Users must be able to access the system using various modern web browsers: Mozilla Firefox 52+, Google Chrome 69+.
* OS support: 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 the ability to dynamically add or remove simulator units.
* Server should try to load simulator units equally or as close as possible.
* Server should have persistent storage of accounts, levels and solutions.
* Language extensibility: Possibility of future support of other languages for solutions.
* Client extensibility: 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 the client-side part of the project.
* It seems preferable to support only screens with a width of at least 1080 pixels because the project, in this case, is much easier to develop and potential losses are not significant (the client-application is supposed to be used mostly for programming, so it is expected to be launched on desktop devices).

2. Goals analysis

The client application can be implemented in a number of different ways. These are as follows:

* Web interface
* Native desktop application with GUI
* Cross-platform application with GUI
* Console application

While web interface may have some disadvantages (e.g. it can behave incorrectly in some web browsers, it has lower performance and limited capabilities compared to native apps which, for instance, have access to the local storage), it seems to be the best solution for the project for the following reasons:

* Easy to implement
* Works on any platform
* Does not require download and installation
* Easy to use

2.1. Security

Due to the 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 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 a local database. It is easy to use, easy to create and because of storing hashes instead of plaintext passwords possible database leaks are not so critical.

Another important area is code execution as we are going to execute solutions on remote simulators. An administrator should create a whitelist of permitted imports that are needed to solve tasks and can’t be used for malicious purposes. The simulator should recognize if an 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.

Python has some advantages over other options: it is popular, has the easiest syntax among these languages and is widely considered to be the best educational programming language.

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

Groovy analysis:

Pros:

* Easy to learn basics
* Can be simply integrated into Java application
* Dynamic compilation
* Supports Java Class Library

Cons:

* Slower than Java
* Less popular than Java and Python
* Static compilation should be used where it is possible to avoid performance issues

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 the 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 the 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:

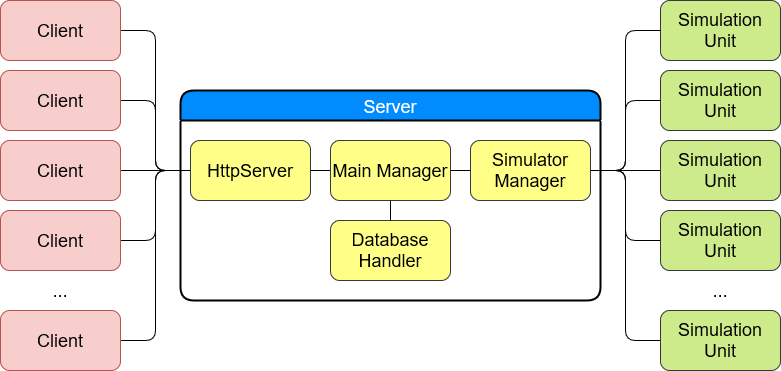
* Server — the core part of the system which manages users, levels and simulations. Implementation language is Java which has a high degree of platform independence. The server provides an API for clients to interact with it.
* Clients — users’ computers that connect to the server to interact with it (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 a 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 the database. It is required for data that 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.

3.2.1 Simulation Units

Simulation Units are reactive, they only respond to incoming requests.

Requests can start new simulation, get current status and number of running tasks, delete a level, create new level or upload a resource file.

If it’s requested to start new simulation, it will conduct the simulation and respond with simulation result or error message.

If it is requested to manage levels and resources it will respond with success or error message.

3.2.2 Server

Server is reactive for Users and active for Simulation Units, Clients make HTTP requests to the Server, Server processes request and makes request to the Simulation Unit if needed, and returns the suitable result to the Client.

Simulator Manager is capable of monitoring Simulation Units. It will keep track of current number of running tasks and online/offline status. When Main Manager requests Simulator Manager to run a simulation then Simulator Manager selects a Simulation Unit with the least number of running tasks to conduct this simulation. If no Simulation Units are online and connected when trying to run a simulation then Main Manager is notified. If Simulator Manager is waiting for Simulation Unit to complete the simulation but Simulation Unit fails and doesn’t respond then Simulation Manager marks this unit offline and tries to retry this simulation with another Simulation Unit. When level is uploaded on the server, it is uploaded on all of the Simulation Units.

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 the server receives a password it calculates its hash and compares it to the one stored in the DB.

It is possible to perform Man In The Middle attack to leak username and password. The most common solution is using an encrypted channel for data transferring.

4.2. Authorization

The type of account will be stored alongside the username, password hash, and some additional information. This type indicates if the 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 the 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 the MainManager interface:

* Lobby — an entity related to one level simulation. It contains information about the chosen level as well as the players participating in this lobby.
* Level — an object associated with any level. Provides information useful for the client (e.g. its name or difficulty but not its source code).
* Player — represents any user that can send solutions for levels.
* SimulationResult — the result of a simulation for a given user.
* Solution — an object that contains all simulation results for a player at a given level.
* CompileResult — an object that includes information about the results of a player’s code compilation and possibly its execution.

|  |  |  |
| --- | --- | --- |
| **Interface** | **Method** | **Description** |
| MainManager | login | Logs a user in the system if their username exists in the database.  Parameters:   * username — user unique name. |
| logout | Logs a user out of the system if their username exists in the database.  Parameters:   * username — user unique name. |
| getLobbies | Returns a list of available lobbies. The list of lobbies must be sorted by date of creation (the newest one - in the head). |
| getLevels | Returns a list of created levels. |
| getSolutions | Returns a list of all the solutions of specified user.  Parameters:   * username — user unique name. |
| joinLobby | Places a user in the lobby gotten by specified lobby id.  Parameters:   * username — user unique name. * lobbyID — lobby unique id. |
| createLobby | Creates a new lobby by level id and places a user in it.  Parameters:   * username — user unique name. * levelID — level's unique id. * playersAmount — the number of players in the lobby. |
| leaveLobby | Removes a user from a lobby.  Parameters:   * username — user unique name. * lobbyID — lobby unique id. |
| returnToLobby | If a player has already submitted a solution and left the lobby, returns them to that lobby.  Parameters:   * username — user unique name. * lobbyID — lobby unique id. |
| submit | 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.  Parameters:   * username — user unique name. * lobbyID — lobby unique id. * code — the code to compile. |
| editSubmittedCode | Cancels the submission of the compiled code and returns the code itself. In case the user hasn't submitted any code yet, returns null.  Parameters:   * username — user unique name. * lobbyID — lobby unique id. |
| isSimulationFinished | Returns whether the simulation has already been finished.  Parameters:   * lobbyID — lobby unique id. |
| getSimulationResult | Returns the simulation result or null in case it hasn't been processed yet.  Parameters:   * username — user unique name. * lobbyID — lobby unique id. |
| Lobby | getId | Returns the lobby's unique id. |
| getHostAvatarAddress | Returns the address of host's avatar icon. |
| getHostName | Returns the name of the host of this lobby. |
| getCurrentPlayersAmount | Returns the number of players present in the lobby. |
| getAcceptablePlayersAmount | Returns the highest allowed number of players for this lobby. |
| getPlayers | Returns the list of players. |
| getLevel | Returns the level. |
| Level | getId | Returns the unique id of the level. |
| getIconAddress | Returns the address of the level's icon. |
| getName | Returns the level's name. |
| getDifficulty | Returns the level's difficulty. |
| getType | Returns the level's type. |
| getDescription | Returns the level's description. |
| getRules | Returns the level's rules. |
| getGoal | Returns the level's goal. |
| getMinPlayers | Returns lowest possible number of players for this level. |
| getMaxPlayers | Returns highest possible number of players for this level. |
| Player | getAvatarAddress | Returns the address of player's avatar icon. |
| getName | Returns the player's unique name. |
| isSubmitted | Returns whether the player has submitted their solution. |
| SimulationResult | getId | Returns the unique id of simulation result. |
| isSuccessful | Returns whether the user robot has reached the goal.  Parameters:   * username — user unique name. |
| getDate | Returns the date of simulation result. |
| getLog | Returns user simulation log.  Parameters:   * username — user unique name. |
| Solution | getLevel | Returns the level. |
| getSimulationResults | Returns the list of simulation results. All the simulation results must be sorted by date (the newest one - in the head). |
| CompileResult | isCompiled | Returns whether the compilation has been successful. |
| isSimulated | Returns whether the simulation has been processed. |
| getMessage | Returns the compile message. |

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. The Main Manager interacts with the Simulation Manager via the SimulatorManager interface:

|  |  |  |
| --- | --- | --- |
| **Interface** | **Method** | **Description** |
| SimulatorManager | addSimulator | Adds a new simulator unit in simulator pool.  Parameters:   * url — location of new simulator unit. |
| removeSimulator | Removes a simulator unit from the simulator pool.  Parameters:   * url — location of new simulator unit. |
| getSimulatorsList | Returns a list of all available simulator units. |
| runSimulation | Runs simulation on an available simulation unit.  Parameters:   * levelId — filename of the level. * lobbyId — id of lobby, used to define id of simulation. * solutions — map of players and their solutions. |

4.4.4. Main Manager - Database Handler API

Main Manager - Database Handler API is a regular Java API, whose function is to organize long-term storage of user accounts, solutions and levels. The Database Handler opens the existing database upon opening or creates a new one if no database is found. Methods that retrieve data from database do not change the database. The Main Manager interacts with the Database Handler via the DatabaseHandler interface:

|  |  |  |
| --- | --- | --- |
| **Interface** | **Method** | **Description** |
| DatabaseHandler | getPlayerByName | Returns a player by their name if that player is present in the database; otherwise returns a null object.  Parameters:   * name — the name of the player. |
| savePlayer | Saves a player to the database.  Parameters:   * player — the player to be saved. |
| getSolutionsOfPlayer | Returns a list of all solutions generated for a given player.  Parameters:   * player — the player whose solutions are to be found in the database. |
| saveSolutionForPlayer | Puts a solution for a given player into the database.  Parameters:   * player — the player associated with the saved solution. * solution — the solution to be saved. |
| removePlayer | Removes a player from the database.  Parameters:   * player — the player to be removed. |

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+), 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: SQLite as DBMS, Gson for JSON (de-)serialization.

Web client frameworks and libraries: jQuery, CodeMirror.