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+, Safari 10.1+, 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.

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.
* 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.
* All team members have completed the introductory course on SQLite databases and would like to use the acquired skills in practice.

2. Goals analysis

*[For each goal there must be overview of ways to solve it and analysis of their applicability, pros and cons regarding the system to develop. The overview could mention external libraries, frameworks, algorithms, known architectural solutions (architectural patterns), suggest some modifications of them. For each external element provide reference to corresponded publications/documentation.*

*The rest of this section including subsections are project-specific and provided just as shortened examples.]*

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 user, who can do the following: ...
* Teacher, who can manage student accounts and edit levels and inherits abilities from Student.
* Administrator is a teacher, who can add and remove simulator units.

There are two main approaches to support authorization:

* Protection rings [ref], that…
* Access control list [ref], that…

Authentication can be implemented either via login/password pair stored in local DB and appropriate login form, or via external service like Google or Facebook.

It is easier to delegate it to ready-to-use external service, so let’s use Google for authentication. *(In fact, the* *solution is questionable at least. First, there are usually other user-related data besides login/password that must be stored on the application level anyway. Second, integration with an external general-purposes service sometimes could be more complex than writing your own project-specific solution).*

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.

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 be responsive, as the support of the mobile browsers or small screens is not required. Therefore, there is no need to use any UI-framework, and the best option to build a layout is to use only plain HTML5 + CSS3.

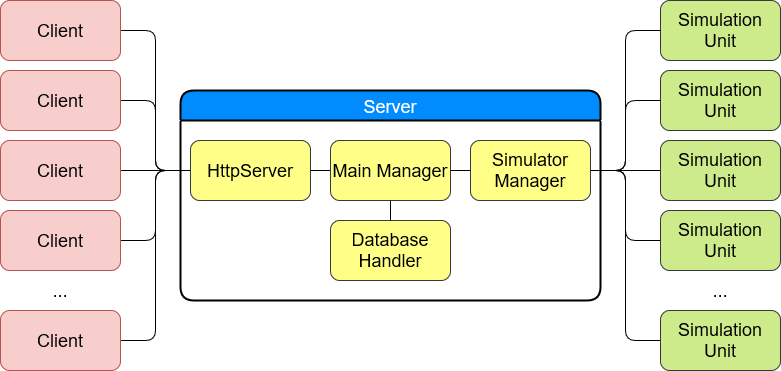
The web pages of the application must dynamically update its data and regularly interact with the server via AJAX. What is important, this behavior must be absolutely the same in any supported web browser. Therefore, it seems to be the best option to use some JavaScript library or framework like jQuery,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

3. Solution description

*[Provide overall description of the system’s architecture].*

3.1. Modules and subsystems

*[Identify top-level modules/subsystems. Pictures are welcome. For each module provide clear description of its functions/responsibilities and interactions with other modules (contact in other words). Also, for each module identify architectural decisions like key frameworks, libraries, languages, architectural patterns, algorithms. References to the section 4 are OK]*

**

*Fig.1 — System structure*

The system consists of three main components (fig. 1):

* Server — the core part of the system which manages users, levels and simulations.
* Clients — users’ computers that connect to the server to interact with it (i.e. submit solutions).
* Simulation units — the entities that execute the simulations of levels. The simulation units can be located either on the same computer as the server or on separate machines.

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.

3.2. Deployment

*[Provide deployment view, i.e. which processes/code parts running on separate devices and how they interact with each other. For distributed software by “devices” separate computers are assumed first of all. Also, this section could show functional distribution between CPU and GPU, interactions with external sensors or other specialized hardware. Pictures are welcome.]*

The system is deployed as shown in 3.1 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.

3.3. …

4. Key architectural elements

*[Identify and define key system-wide (i.e. affecting multiple modules from section 3) behavioral elements like protocols, algorithms, computational mechanisms. Usually such elements are corresponded to at least one goal from the section 1. Each element must be described in terms of modules identified in section 3 and their behavior. Pictures are welcome.*

*This section could be placed before section 3 if necessary.*

*The rest part of this section including subsections are examples]*

4.1. [ex.]Authentication protocol

…

4.2. [ex.]Authorization

…

4.3. [ex.]Fault-tolerant network interaction

…

4.4. [ex.]Modules API

All the modules’ external APIs should follow Command-Query Separation [<reference>] pattern excluding …, where atomic types should be used and, thus, they must violate this pattern.

All the modules excluding … must implement their APIs in two forms: 1) regular Java API in terms of interfaces, and 2) RESTful API using Jetty [<reference>]as embedded servlet container.

Regular Java API must be used in most situations due to better performance and reliability. RESTful APIs should be used mostly for automatic testing purposes.

4.5. [ex.]External plug-in API

For module … the external API for 3rd party video processing plugins must be implemented in reliable way. A plug-in must be run as a separate process restricted by RAM and CPU usage by … So, the main application shall tolerate the situations where plug-in killed due to critical error (segmentation fault or so), overusing memory or CPU, spawning threads.

Interaction with plug-in on lower level must be implemented via STDIN/STDOUT/STDERR. On higher level, the sandbox for plug-ins must be implemented (see 3.NN) and provide the following operations: …

To transmit objects through the raw stream JSON serialization/deserialization must be used. Jackson [<reference>] library is suggested for this. There is the requirement to possess bandwidth of … MB/s for plugins, so Jackson must be checked first to match this.

4.6. …

5. Platform

*[Enumerate all the hardware and external software decisions that come from all the previous section.*

*All the rest is just an example.]*

Server platform: x86-64/amd64, Linux 2.6

Client platform: ARM, Android; Web browser (Edge, Chrome, Firefox)

Language: Java on server and Android client, JavaScript on Web-based client

Server frameworks and libraries: Tomcat as servlet container, Spring (Boot, Security, MVC), PostgreSQL as DBMS, Hibernate for interactions with DB, Jackson for JSON(de-)serialization, JMeter for load testing.

Web client frameworks and libraries: ReactJS