STARL: TOWARD A WEB INTERFACE FOR DISTRIBUTED ROBOTICS

BY

SHUTING LI

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Bachelor of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2016

Urbana, Illinois

Adviser:

Sayan Mitra

ABSTRACT

Most first-time users find it complicated to use the StarL programming framework, especially when they have little experience with Java. The major challenges for programming distributed robotic applications are (1) the learning curve for Java,(2) setting up the StarL development environment (3) learning curve for effectively using the Java functions in StarL. We therefore introduce the StarL web interface that provides a more user-friendly access to the StarL programming framework while emphasizing more on the StarL high-level coordination of distributed robots. The StarL web interface enables researchers to implement their applications on distributed robots in the StarL high-level language, run the project and then plot the experiment data for analyzing the robot's traces. The main contribution of this thesis is the user-friendly interface with syntax highlighting and data visualization of the robots' traces obtained through simulation. A Formation example application will illustrate the many aspects of the StarL web interface.

To my parents, for their love and support.

ACKNOWLEDGMENTS

I owe my gratitude to all those people who have provided generous help and because of whom my research experience has been one that I will cherish forever.

My deepest gratitude is to my adviser, Prof. Sayan Mitra. I have been fortunate to have an adviser who gave me the freedom to explore on my own and the guidance to meet high standards. His insightful comments and constructive criticisms at different stages of my research helped me finish this thesis.

The project lead, Yixiao Lin, has been always there to listen and give advice. I am deeply grateful to him for the long discussions that helped me sort out the technical details of my work. I am also thankful to him for encouraging the use of correct grammar and consistent notation in my writings and for teaching me how to express ideas.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1 Motivations for simulator for distributed robots

In the past few years, we are witnessing a large growth in the robotics industry. In the effort to make robotic systems available to more users, combining robotic systems with easier access has become a recent trend. Faculty and researchers at Georgia Tech developed “robotarium”[1], which allows researchers and students from anywhere to upload their own programming code and watch it compile on the robots in real-time via streamed video and receive scientific data results. Researcher from Commotion Laboratory at UCLA also conducts remote experiments on multiple mobile robots [2]. Such robot systems are generally able to provide users with a programming environment including compiler, simulator and debugger. In addition to that, importance in providing an easier access and stable environment on which the robotic system runs also need to be addressed. Relative work has been done on providing a distributed virtual environment and applies it to mobile robot teleoperations over the Internet[3]. One such attempt is to use a virtual reality world generator to mimic the system of an actual robotic system settings. However our goal is to build the interface with more intuitive instructions and representations while at the same time maintain the powerful features when developing distributed robotic system applications.

1.2 Introduction to the StarL web interface

With the ideas of building a distributed robotic system with easy access and a user-friendly interface in mind, we hence introduce the StarL web interface, a programming web interface for the StarL high-level language.

The StarL open-source programming framework[4] provides a Java-based framework that is comprehensive for distributed robotic system development. While the StarL web interface requires less knowledge of Java, with the simpler web interface and StarL high-level language, users are able to use high level language to develop their own applications and research on verification problems.

1.3 Contributions of this thesis

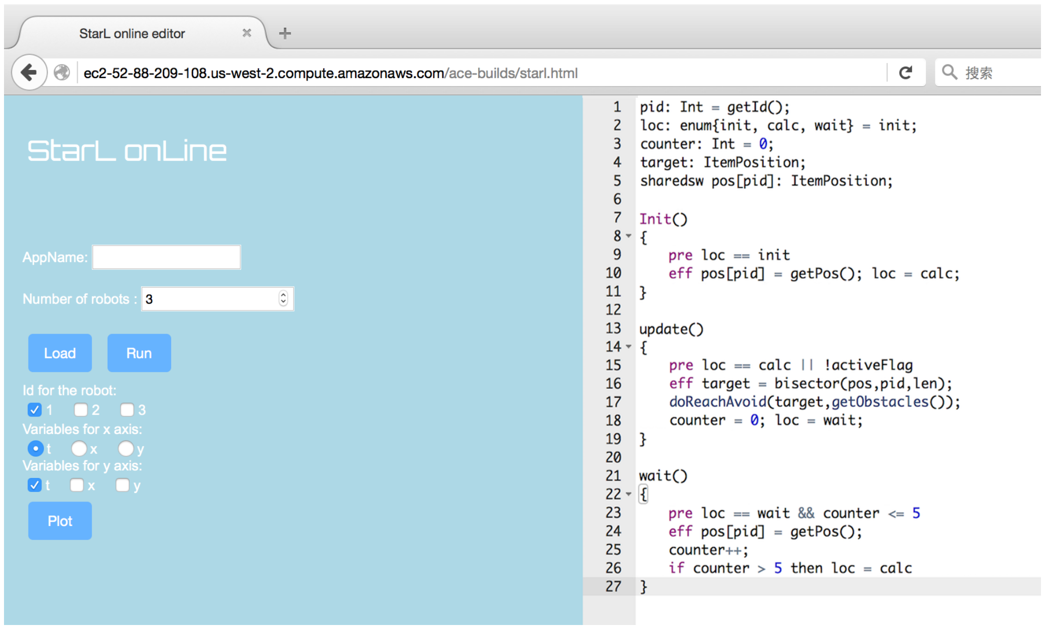
This paper will provide a description on how to use the StarL web interface with an example application, introduce several features that could improve the user experience, such as syntax highlighting, error notifications, file system and data visualization.

In chapter 2, we will introduce the StarL web interface with the Formation application, and mainly focus on the StarL high-level language and the standard flow of applications. Then in chapter 3, the architecture of the StarL web interface will be demonstrated with further explanation of the file system, logging and visualization. Lastly, we will summarize the accomplishments of the StarL web interface to conclude the paper and show the future work in chapter 4.

CHAPTER 2

FLOW OF THE STARL WEB INTERFACE

Without the StarL web interface, users who are not familiar with Java have to download the Java Environment first, and they need to learn the Java programming language. To shorten the learning curve for implementing applications using the StarL framework, the StarL web interface, shown in Figure 2.1, can handle editing coding scripts, compilation and data visualization. The StarL web interface is constructed on Apache servers as an integrated development environment providing a coding editor, build tools and data visualization. Here we use the Formation application as an example to explain the flow of the StarL web interface.


StarL online editor
ec2-52-88-209-108.us-west-2.compute.amazonaws.com/ace-builds/starl.html
搜索
StarL onLine
Load
Run
Plot

AppName:

Number of robots : 3

Id for the robot: [✓ 1] 2 3

Variables for x axis: [✓ t] x y

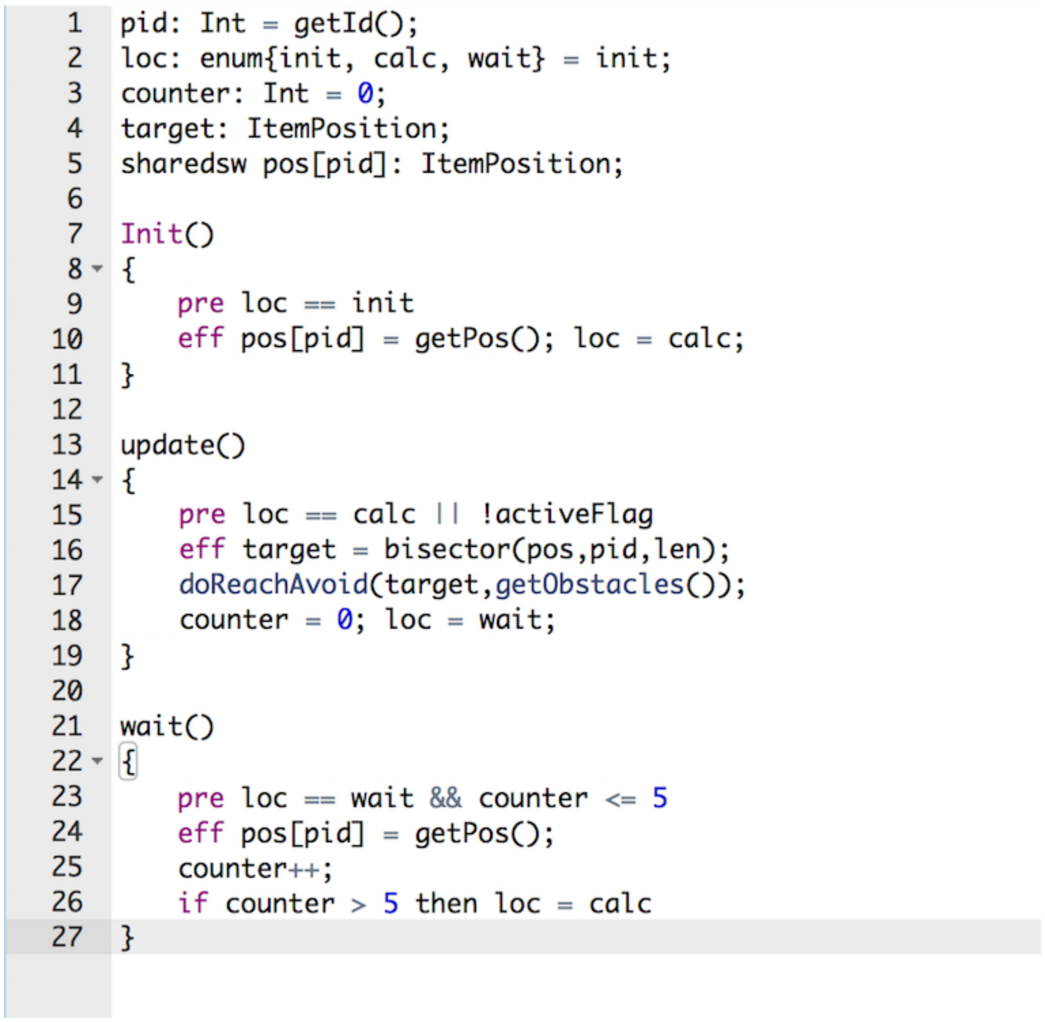
Variables for y axis: [✓ t] x y

```
pid: Int = getId();
loc: enum{init, calc, wait} = init;
counter: Int = 0;
target: ItemPosition;
sharedsw pos[pid]: ItemPosition;

Init()
{
    pre loc == init
    eff pos[pid] = getPos(); loc = calc;
}

update()
{
    pre loc == calc || !activeFlag
    eff target = bisector(pos,pid,len);
    doReachAvoid(target,getObstacles());
    counter = 0; loc = wait;
}

wait()
{
    pre loc == wait && counter <= 5
    eff pos[pid] = getPos();
    counter++;
    if counter > 5 then loc = calc
}
```

Figure 2.1: A screenshot of the StarL web interface

2.1 Example: Formation application

The formation application aims to form a regular polygon among the robots. The following code will take the random position as initial position and then update the position of robot accordingly, as stated in [5].



```
 1  pid: Int = getId();
 2  loc: enum{init, calc, wait} = init;
 3  counter: Int = 0;
 4  target: ItemPosition;
 5  sharedsw pos[pid]: ItemPosition;
 6
 7  Init()
 8  {
 9      pre loc == init
10      eff pos[pid] = getPos(); loc = calc;
11  }
12
13  update()
14  {
15      pre loc == calc || !activeFlag
16      eff target = bisector(pos,pid,len);
17      doReachAvoid(target,getObstacles());
18      counter = 0; loc = wait;
19  }
20
21  wait()
22  {
23      pre loc == wait && counter <= 5
24      eff pos[pid] = getPos();
25      counter++;
26      if counter > 5 then loc = calc
27  }
```

Figure 2.2: The Formation application code showan in the editor of the StarL web interface

The code for the formation application is shown in the Figure 2.2, written in StarL high-level language. There are three states in total: init, update and wait. The first 11 lines are for initialization. In line 10, the function reads the current positions of three robots and goes to the next stage. Then, the update state will calculate the next target from the built-in function, bisector in line 16. The path will be generated by the doReachAvoid function in line

17, which will take the target and obstacles as input, calculate a path that can reach the target while avoiding the obstacles. The wait state will wait until it receives the done signal from the simulation. The code will check the condition in line 23. It will update the positions in line 24 and go back to the update function in line 26.

2.2 StarL high-level language

When users first open the StarL web interface, it will automatically open an example code with detailed comments, as shown in the figure 2.1, which can provide information on how to implement project using the StarL high-level language. The editor will also be able to show highlighting and detect syntax error.

The syntax highlighting in the browser is provided by the Ace editor. Among all the user-friendly editors, such as Sublime, Vim, Brackets, Ace editor is a lightweight, open-source editor in JavaScript that can be easily embedded into web interface. For the StarL high-level language used in the web interface, we create a new mode with a set of highlighting rules.

Keyword	agent, MW, SW, init, exit, pre, eff, else, if, shared, atomic, failed, done
Data type	float, int, boolean, Itemposition
Constant	true, false
Built-in function	getObstacles, getId, getPos, doReachAvoid, size,

Table 2.1: Table for different types of highlighting

We will highlight the keyword, data type, constant and built-in function with four different color, as shown in Table 2.1. The *agent* will be used for application name. *MW* and *SW* stands for multi-writer and single-writer. *init* and *exit*, are for initialization state and exit state respectively. Conditional statements: *pre* and *eff* will be used together for the precondition and effect; *if* and *else* are also permitted. Date status can be determined as two categories: *shared* and *atomic*. The current state of the robot's motion will be saved in *failed*, *active* and *done*. As stated in [6], the *failed* status indicated the robot has hit the obstacles.The *active* status means the ReachAvoid primitive is in grogress. When the robot has reached the target point, *done* will be true. There are four data types: *float*, *int*,*double*, *string*, *boolean* and *Itemposition* in StarL high-level language for users to declare. Constants are *true*, *false* and numbers. Built-in function can provide convenient use for coding. We can use the helper functions provided by StarL: *getObstacles*, *getId*, *getPos*, *doReachAvoid*, and *size* . The primitive *getObstacles* will return the positions of obstacles. The *getId* will return the index of current robot. From *getPos*, we can get the current position of the robot. The primitive *doReachAvoid* can calculate a path to the target position avoiding the obstacles with the target and obstacles inputs. *size* will return the size of data array. The web interface also provides a proper syntax highlighting to give users a hint of what types of variable they are using and where there is possibility that an error happens. Figure 2.2 serve as an example for highlighting.

2.3 Flow of StarL web interface

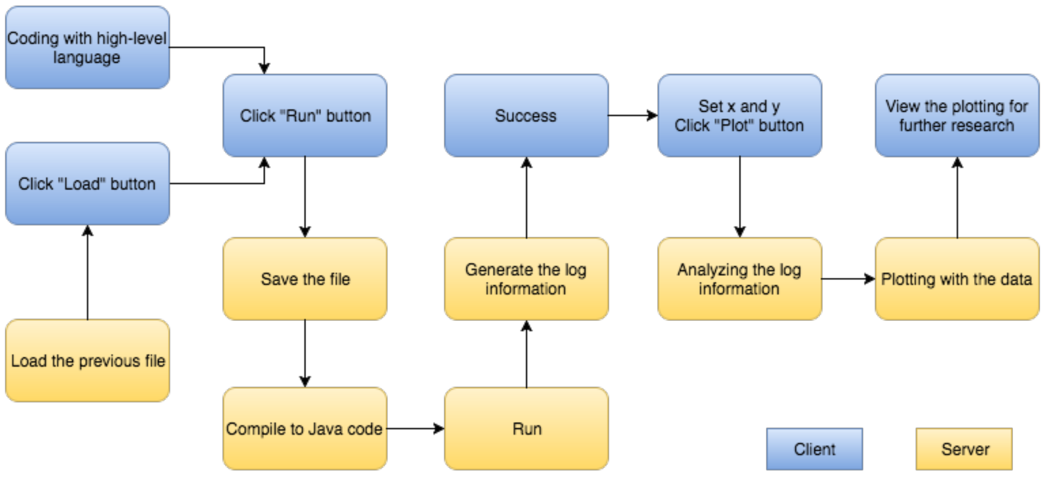

Coding with high-level language
Click "Run" button
Success
Set x and y Click "Plot" button
View the plotting for further research
Click "Load" button
Save the file
Generate the log information
Analyzing the log information
Plotting with the data
Load the previous file
Compile to Java code
Run
Client
Server

Figure 2.3: The flow chart for the StarL web interface

The web interface will provide an easier access for StarL so that users can analyze their app easily by using merely several buttons. The flow of the StarL web interface is simple, as shown in Figure 2.3.

1. Users can write their own scripts. Or they can load the previous ap-

plication by typing the name of the project in the text-box and click the "Load" button, which will evoke the load function to read the file through the file system at the server side.

2. After programming, click the "Run" button. The server will save the file the user is currently editing to the StarL server under the user-specified file name in the text-box.

3. If compilation failed, an error message will appear beside the run button. Otherwise, a success message will be shown.

4. Then the server will compile the StarL high-level code into scripts in Java. StarL will run the Java scripts and generate the log file before the server analyzes the log file and outputs the data file.

5. After the file is successfully compiled and run, the user can then choose how to plot the results. First, choose the robotID(s). Then, choose the variables for the x and y axes. Finally, click "Plot" button.

6. After the Gnuplot function at the server side is evoked, the interface will display the plotting. Users can freely change the setting for graphs and research further with different plotting. For example, shown in 2.4, is a y vs x plot of iRobot0.

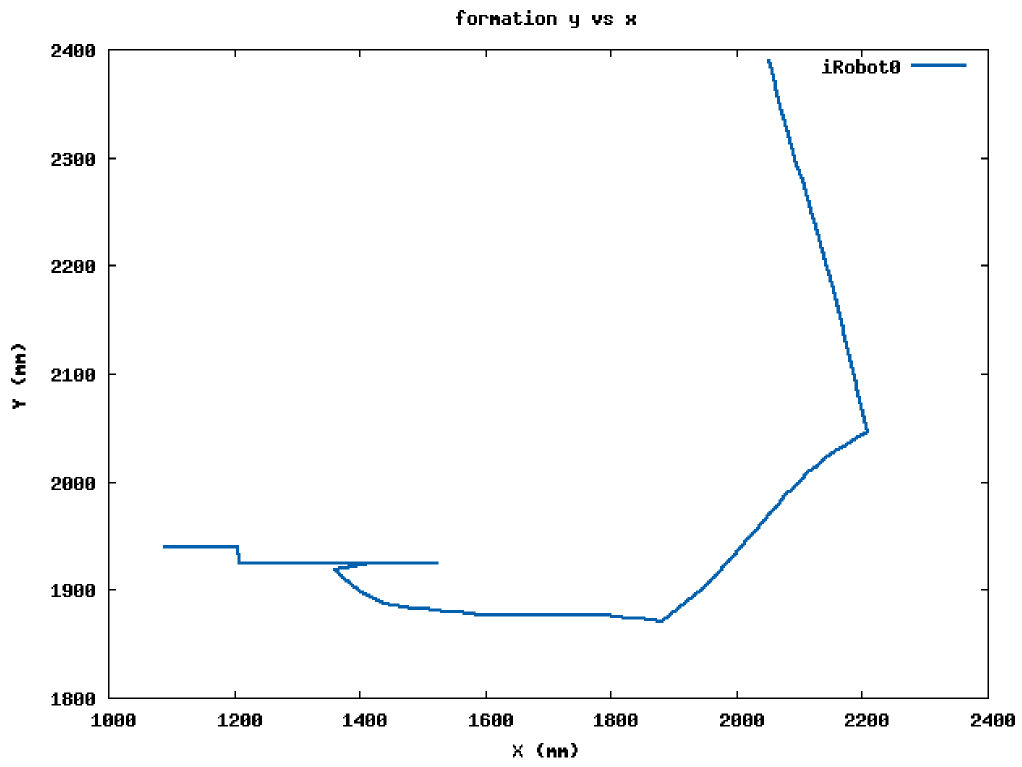
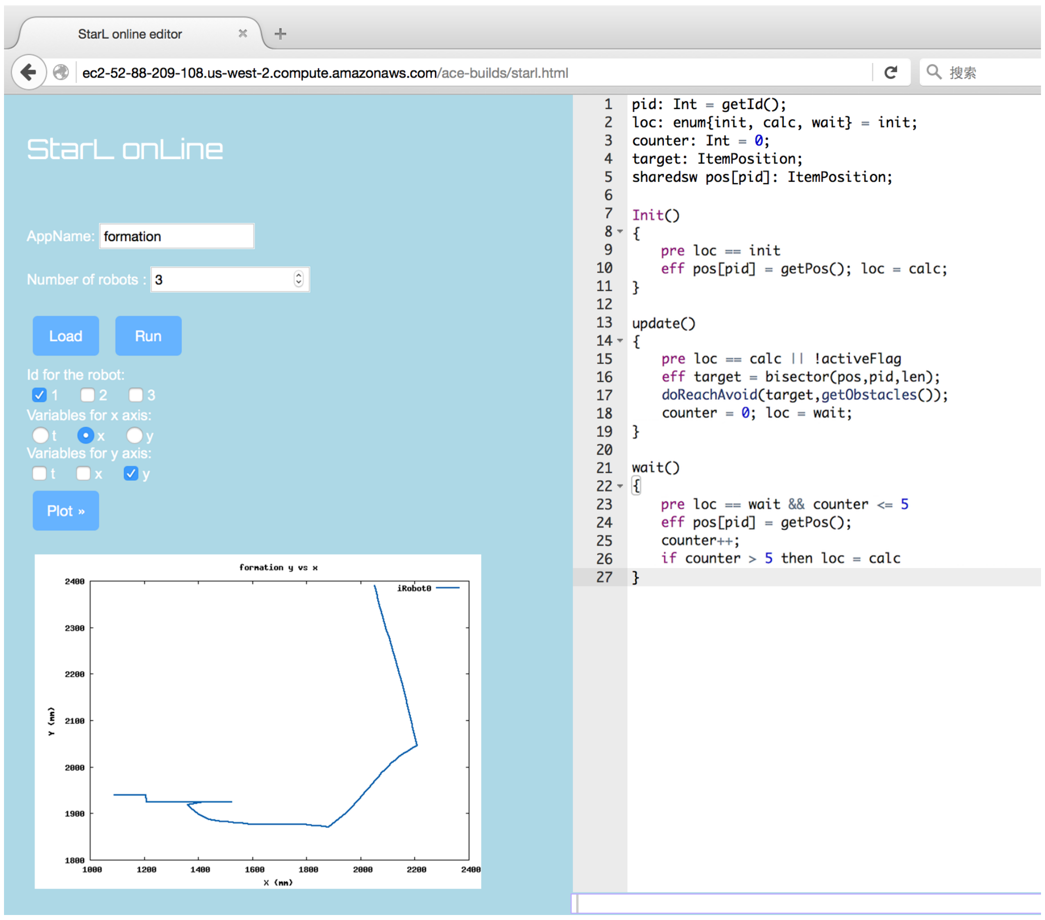


Figure 2.4: Plot from formation application

2.4 Deployment of different components

Interface deployment follows the rule of providing intuitive meaning. The interface is divided into two parts with the control on the left and editor on the right. For the control panel, a text input cell is place to specify the application name for loading or running a previous project. Underneath it a “load” and a “run” button are placed to load the code or save, compile and run the current scripts. A few check boxes are then place below to select for diverse plotting mode. With the check boxes, users can choose the set of data for a certain robot or a group of robots to be shown in the graph. An additional button named “Plot” is placed beneath it to plot the patterns. The data visualization will shown under the ”Plot” button, as shown in Figure 2.5. Users can right-click on the plot and save the image.



AppName: formation

Number of robots : 3

Id for the robot: [✓ 1] 2 3

Variables for x axis: t [✓ x] y

Variables for y axis: t x [✓ y]

```
pid: Int = getId();
loc: enum{init, calc, wait} = init;
counter: Int = 0;
target: ItemPosition;
sharedsw pos[pid]: ItemPosition;

Init()
{
    pre loc == init
    eff pos[pid] = getPos(); loc = calc;
}

update()
{
    pre loc == calc || !activeFlag
    eff target = bisector(pos,pid,len);
    doReachAvoid(target,getObstacles());
    counter = 0; loc = wait;
}

wait()
{
    pre loc == wait && counter <= 5
    eff pos[pid] = getPos();
    counter++;
    if counter > 5 then loc = calc
}
```

Figure 2.5: The result of the formation application

CHAPTER 3

ARCHITECTURE FOR WEB INTERFACE

The architecture the for StarL web interface can be divided into two sides: the client side and the server side. With the constant communication between the client and server, the StarL web interface can therefore provide useful features such as file system, StarL simulation and data visualization. The Apache HTTP server[7], a public-domain web server which is developed as an open-source software, is deployed on the Amazon Web Service EC2 (Elastic Cloud Computing) [8]. The Apache web server is fast, reliable and secure. It delivers contents that can be accessed through the Internet. Apache is robust in handling large volumes of traffic on a single server. Apache modules encapsulate a wide number of functionalities including cryptographic protocols like SSL, server-side programming languages like PHP, and load balancing across multiple servers to handle large amounts of traffic. The StarL web interface mainly relies on PHP for the file system and commands on a bash terminal.

3.1 Mapping between the interface and the StarL

1. First, on the client side, the web interface will send the contents of the editor to the server side. The server side will save all the contents in the file system as StarL high-level code.

2. Then the-high level code will be compiled to Java code through the compiler. Also, in the setting of project, for example, the number of robots will be used to generate the main.java file.

3. The StarL core component will then generate a simulation with these java files. The log information generated will be used to plot some graphs with Gnuplot. The graph will be sent to the client side and

displayed in the interface.

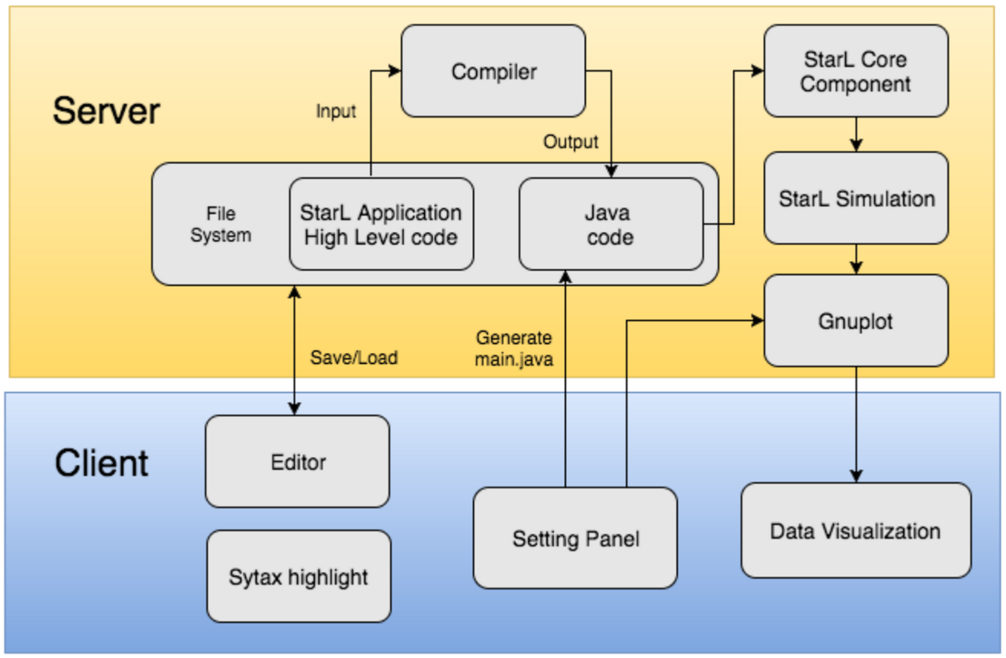


Figure 3.1: The structure of the StarL web interface

3.2 File system

During the research on different area with distributed robots, users might implement multiple projects using the StarL web interface. The file system on the server side allows the users to save or load a certain project or switch between different projects.

Provided with only the name of the application, for example, myApplication, the file will be saved in the current folder of the working path with the name of myApplication. If users would like to save the file in a new folder, use newFolder/myApplication. The folder will be automatically created and the file will be saved in the new folder. Since the ACE editor is using HTML, we need to send XMLHttpRequest to evoke the PHP function. The Apache server is used in supporting PHP scripts. In the PHP scripts, use the filepath and the file contents from user inputs to save the file. The saveTo function in JavaScript on the client side will save the script in the current editor to the specific path on the server side. First, it will create the XMLHttpRequest object in line 2. Then it can get the contents of current editor in line 5. The path and the contents will be marked in line 6. Send the XMLHttpRequest in line 11. The function in line 15 will check the status of the process and display the return data.

```
 1 function saveTo(fn){
 2     var hr = new XMLHttpRequest();
 3     var url = "saveTo.php";
 4     var ln = env.editor.getSession().getValue();
 5     var vars = "path="+fn+"&contents="+ln;
 6     hr.open("POST", url, true);
 7     hr.setRequestHeader("Content-type", "application/x-www-form-
      urlencoded");
 8     hr.onreadystatechange = function() {
 9       if(hr.readyState == 4 && hr.status == 200) {
10         var return_data = hr.responseText;
11       document.getElementById("status").innerHTML = return_data;
12       }
13     }
14     hr.send(vars);
15     document.getElementById("status").innerHTML = "processing...
      ";
16 }
```

The script in PHP file on the server side will save the file with contents and file path. The function will open the file to get existing content in line 350. Change the contents of the file in line 352 with the $file_put_contents$ function. Then the Savedto will be sent back to the client side and displayed in the interface.

```
1     <?
2      if($_POST['path'])
3         {
4         $file = $_POST['path'];
5         $current = $_POST['contents'];
6         file_put_contents($file, $current);
7         echo Savedto;
8     }
9     ?>
```

With the XMLHttpRequest and Php code above, we can have access to the filesystem, and therefore provide a better editing experience for users.

3.3 Logging information

Table 3.1: An example of logging information for iRobot.

RobotID	Timestamp	Type	X	Y	Z	Orientation
iRobot0	660	POSITION	5	-7	0	8.25
iRobot0	760	POSITION	5	-5	0	16.5

Table 3.2: An example of logging information for quadcopter.

RobotID	Timestamp	Type	X	Y	Z	Roll	Pitch	Yaw	Force
quadcopter0	530	POSITION	4519	1334	167	110.0	0	0	464
quadcopter0	605	POSITION	4519	1334	204	110.0	0	0	504

Logging file will be generated by the StarL after each run, which contains certain formatted message with meaningful data. Tables 3.1 and 3.2 are examples of logging information generated by StarL core component. We have two types of built-in models: iRobot and quad-copter. IRobot is a ground robot, which is similar to the Roomba, the robotic vaccum cleaner. The quad-copter is the standard research-used quad-copter. In table 3.1, logging for iRobot contains the robotID, timestamp, information type, coordinates x, y, z and the orientation in degree. In table 3.2, logging for quad-copter contains the robotID, timestamp, information type, coordinates x, y, z and the orientation in three dimensions(roll, pitch and yaw) and the force to lift the quadcopter. The analyzing function in Python will deal with the logging file and then output a data file that is readable for Gnuplot.

The Python program for extracting the data out of the logging files will be called when the client side send a XMLRequest to evoke the PHP file. Then, a command "python readFiles.py filePath" will give the path to the file for Python program so that it can locate the file and analyze the data. With the input file, readFile programme will extract the data from each different robots and then save the data in separate files so that plotting programme can easily read the data from different robots

3.4 Plotting

To generate the graphs in the web interface for further analysis, we employ Gnuplot, a command-line driven plotting tool so that we can send bash commands by PHP files to plot graphs. It is designed for scientists and researchers to visualize mass data [9].

After the server finished running the project and analyzing the logging files, Gnuplot can read the output data file and generate a graph with corresponding variables for the x, y axes. Since the Gnuplot is command-line driven, we can send commands to a terminal by using a similar method as stated in Section 3.2. The project will use the same function for XMLHttpRequest but with different PHP scripts. We will first use a Python function to write the Gnuplot scripts with the settings from the front-end. Users can change the number of robots they want to simulate by settings in control panel. In the web interface, users get to choose the variable they would like to explore.

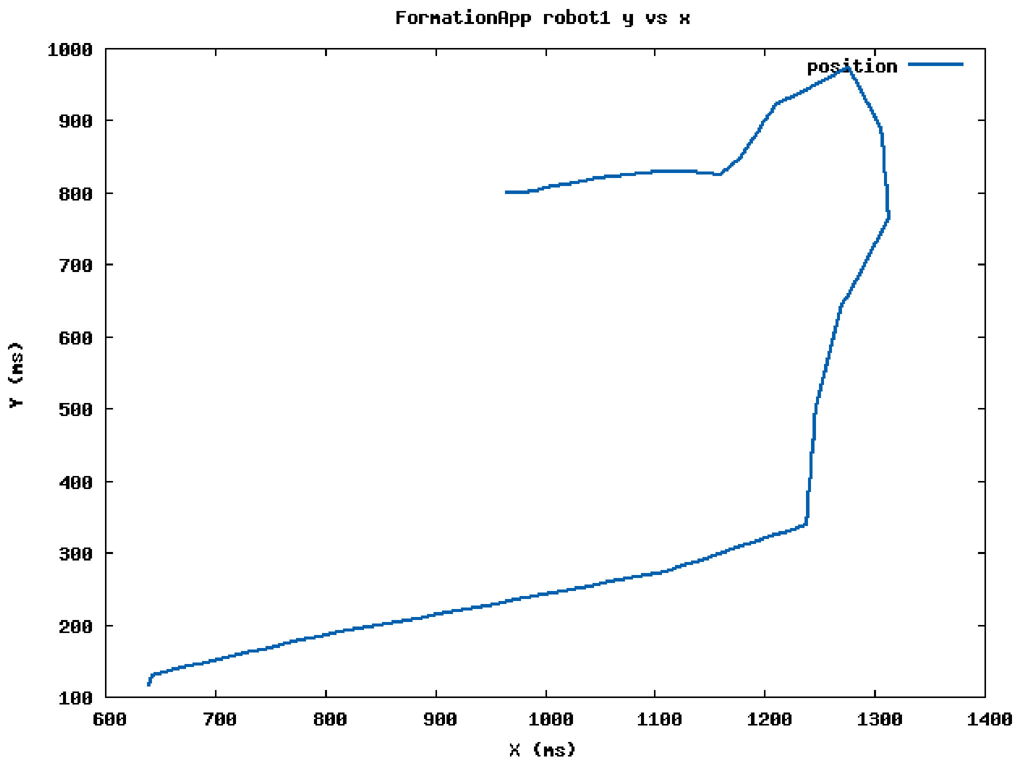


Figure 3.2: y vs x with iRobot

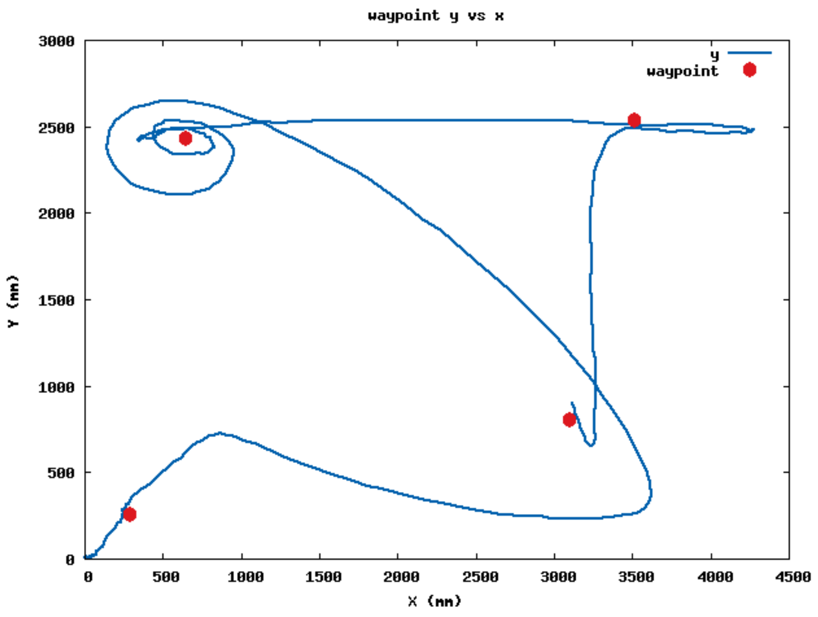


Figure 3.3: y vs x with quad-copter

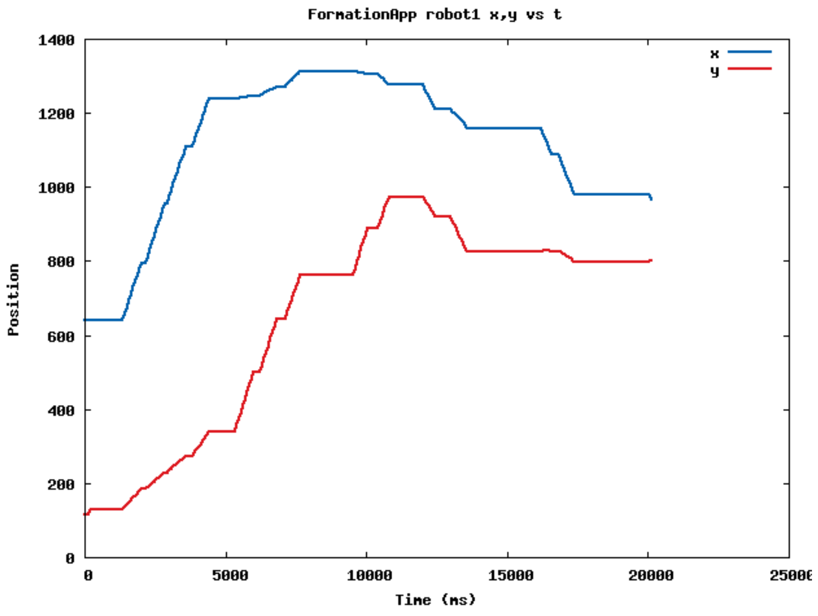


Figure 3.4: x, y vs t with iRobot

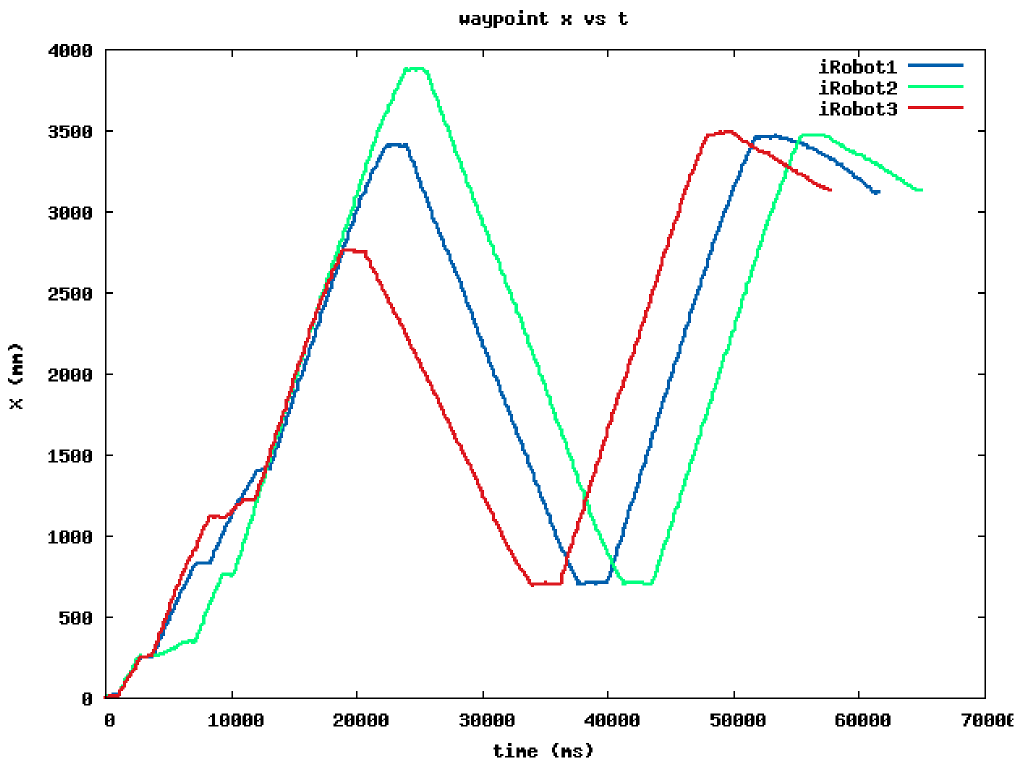


Figure 3.5: Multiple robots

The user can choose how many robots to visualize in a graph. If we plot y vs. x graph, it will show a graph of the trace and any obstacles in the current map, as shown in Figure 3.2. Compared to the trace of iRobot (Figure 3.2) and quad-copter (Figure 3.3), we can find that quad-copter can easily get overshot since the speed is faster than iRobot. If we choose x vs. t or y vs. t, users can estimate the speed or acceleration through the graph, as shown Figure 3.4. The data visualization also supports plotting for multiple robots, for example, Figure 3.5. We check all the boxes for the robots we want to explore, and the plot will use different colors for various robots and provide a color legend so that users can compare the behaviors among different machines.

CHAPTER 4

CONCLUSION

The StarL web interface is intuitive, easy to use yet powerful and provides the functionality and reliability from StarL. The features such as the file system, logging analysis, and plotting data support the convenient operation in the interface. The syntax highlighting, buttons and check boxes on the client side also adds to the ease of use. Users who focus on the applications of distributed robots can hence save a lot of time on simulation and analysis of the robots' behaviors since the StarL web interface shortens the learning curve of programming. In the future, we will further develop the data visualization with real-time animation of the simulation using WebGL. Syntax check will also be a direction to provide a more helpful tool for programming distributed robot applications.

For more information about StarL web interface click on the following link: https://github.com/shutingli/StarLWebInterface/tree/master for further information.

REFERENCES

[1] D. Pickem, E. Squires, and M. Egerstedt, "The robotarium: An open, remote-access, multi-robot laboratory."

[2] Y. Cao, T.-W. Chen, M. D. Harris, A. B. Kahng, M. Lewis, and A. Stechert, "A remote robotics laboratory on the internet," *Proc. INET-95, Honolulu*, 1995.

[3] M. Matijasevic, K. P. Valavanis, D. Gracanin, and I. Lovrek, "Application of a multi-user distributed virtual environment framework to mobile robot teleoperation over the internet," *Machine Intelligence & Robotic Control*, vol. 1, no. 1, pp. 11–26, 1999.

[4] Y. Lin and S. Mitra, "Starl: Towards a unified framework for programming, simulating and verifying distributed robotic systems," in *Proceedings of the 16th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems 2015 CD-ROM.* ACM, 2015, p. 9.

[5] Y. Lin, S. Mitra, and S. Li, "Porting code across simple mobile robots," *Submitted for review of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2016, submitted for review.

[6] Y. Lin, G. Ritwika, and S. Mitra, "Starl framework for programming,simulating and verifying distributed robotic applications," in *Submitted for review of the ACM Transactions on Embedded Computing Systems(TECS).* ACM, 2016.

[7] R. T. Fielding and G. Kaiser, "The apache http server project," *Internet Computing, IEEE*, vol. 1, no. 4, pp. 88–90, 1997.

[8] E. Amazon, "Amazon web services," *Available in: http://aws. amazon. com/es/ec2/(November 2012)*, 2015.

[9] J. Racine, "Gnuplot 4.0: A portable interactive plotting utility," *Journal of Applied Econometrics*, vol. 21, no. 1, pp. 133–141, 2006.