Faculty of Science and Technology
Department of Computer Science

RoboMind

A platform for on-the-fly programming and inspection of behavior-based robot programs

—

Alexander Svendsen

INF-3981 Master's Thesis in Computer Science, June 2014

Abstract

Lego Mindstorms is a popular tool used by universities for educational and demonstrational purposes. Lego Mindstorms is a set of buildable and programmable robotic kits, made by Lego. It allows for a high level of participation from the audience, while being easily programmable. However, for demonstrational and recruitment purposes, it is not without shortcomings. When there is a limited time to talk to people, it is difficult to explain and change a running Lego Mindstorm program. Typically the program is explained by looking at the robot's behavior and base the explanation on that. Modifying an already running program involves multiple slow steps that disrupts what the robot was previously doing.

A more interactive approach would be to allow the audience to see what is happening inside the *"robot's mind"* while it is running. Where changes could be made on-the-fly without disrupting what the robot was previously doing.

This thesis introduces *RoboMind*, a platform for on-the-fly programming and inspection of behavior-based robot programs. The idea behind RoboMind is to provide users with an interface where they can both visually inspect and modify a robots information and behaviors at run-time. Through RoboMind, users can add or edit existing behavior modules on-the-fly without disrupting what the robot was previously doing. The interface allows users to inspect which behaviors are running in addition to the collected sensor samples from the robot's run-time.

RoboMind has been thoroughly tested and developed for usage with the latest generation of the Lego Mindstorms, the EV3. RoboMind offers a fully featured Python-programming environment for run-time modification of running EV3 programs with the help of the ev3-python library made by the author.

Acknowledgments

I would like to thank my advisor John Markus Bjørndalen for his continuous constructive feedback and invaluable support during this project. Your help has been invaluable.

Special thanks goes to my friend Simon Engstrøm Nistad. He has helped me a lot trough this thesis and I am grateful for these five awesome years.

I would like to thank my friends, Simen Lomås Johannesen, Steffen Hageland, Tom Arne Pedersen and Ida Jaklin Johansen. Without their continuous support through these five long years at the university, I would not have made it. I cannot thank you people enough, and I really look forwarding to working with some of you as colleagues in the near future.

Finally, I would like to thank my friends and family for their support and encouraging words through these cumbersome years of study.

Contents

Abstract **iii**

Acknowledgments **v**

1 Introduction **1**

1.1 Problem Definition 2

1.2 Contributions 3

1.3 Limitations 3

2 Background **5**

2.1 Lego Mindstorm 5

2.1.1 EV3 7

2.2 leJOS 8

2.3 Behavior-based systems 9

2.3.1 Subsumption 10

2.3.2 Fuzzy behavior 11

3 RoboMind **13**

3.1 Setup 13

3.2 Connect and configure 14

3.3 Control and Program 18

3.4 Responsive Design 20

4 Architecture 21

4.1 ev3-python library 21

4.2 RoboMind 23

5 Design and Implementation 27

5.1 Robo 27

5.1.1 Communication 29

5.1.2 Technologies 30

5.2 ev3-python 31

5.2.1 Demonstrational use of the ev3-python API 31

5.2.2 Technologies 35

5.3 RoboMind 35

5.3.1 Web server 36

5.3.2 WebSocket Server 36

5.3.3 Brick Manger 37

5.3.4 Code Manager 37

5.4 Web interface 38

5.5 Behavior library 39

5.5.1 Subsumption 39

5.5.2 Fuzzy behavior 40

6 Evaluation **41**

6.1 Experiments 41

6.1.1 Experiment environment 41

6.1.2 Robo 42

6.1.3 ev3-python 45

6.1.4 RoboMind Master Server 46

6.1.5 Web Interface 48

6.2 Issues found 49

6.3 Improvements 50

6.3.1 Robo 50

6.3.2 RoboMind master server 50

6.3.3 Web interface 50

7 Related Work **51**

7.1 nxt-python 51

7.2 Programming on the device 52

7.3 The Player/Stage Project 53

7.4 REAL 54

8 Conclusion and Future Work **57**

8.1 Contributions 57

8.2 Concluding remarks 58

8.3 Future Work 58

References **59**

Appendices **63**

Appendix A Behavior Example **63**

A.1 Behavior: AvoidColliding 64

A.2 Behavior: DriveAround 64

List of Figures

2.1 The programmable intelligent brick of each generation 6

2.2 Shows a short program written in the default programming environment of the EV3 7

2.3 Main components in the EV3 education set, without the small motor 8

3.1 Screenshot of the initial view in RoboMind 14

3.2 Screenshot of the initial view when the user has connected to an EV3 15

3.3 Screenshot of the web interface when multiple sensors have been opened 16

3.4 Screenshot of the web interface when multiple behavior modules have been added and are running 17

3.5 Screenshot of the web interface on a mobile device 20

4.1 Architecture of the ev3-python library 22

4.2 Architecture of RoboMind 25

5.1 Design of Robo 28

5.2 Design of ev3-python library 32

5.3 Design of RoboMind, without EV3s connected 35

6.1 Distribution of roundtrip time during a Ping request over USB 43

6.2 Shows a request's improved roundtrip time 44

6.3 Average roundtrip time for requests during different workloads on Robo. Measurements done through the USB channel. . . . 46

6.4 Time taken before 1000 streaming packages is received on sensor samples from Robo through the different communication channels. Also shows the variations in time when 1-4 sensors is opened for streaming. 47

Chapter 1

Introduction

Different robotic platforms has in the last decades been a popular tool used by universities for educational and demonstrational purposes[16, 15, 12, 17]. At the University of Tromsø, a wide variety of different robots has been used in multiple recruitment fairs and school visits. One such robotic platform is the Lego Mindstorms[2]. Lego Mindstorms is a set of buildable and programmable robotic kits, made by Lego. It allows for a high level of participation from the audience, while being easily programmable. However, for demonstrational and recruitment purposes, it is not without shortcomings.

When there is a limited time to talk to people, it is difficult to explain and change a running Lego Mindstorm program. Typically the program is explained by looking at the robot's behavior and base the explanation on that. Modifying an already running program involves multiple slow steps that disrupts what the robot was previously doing. Usually the steps involves stopping the running program, uploading the modified program and restarting the program on the robot.

A more interactive approach would be to allow the audience to see what is happening inside the *"robot's mind"* while it is running. The mind could detect any changes in its settings, or source code, which are immediately applied and used in the running robot. Furthermore, the mind could provide an environment where programming can be done in a language not only intended for the Lego Mindstorms.

This thesis introduces *RoboMind*, a platform for on-the-fly programming and inspection of behavior-based robot programs. The idea behind RoboMind is to provide users with an interface where they can both visually inspect and modify a robots information and behaviors at run-time. Through RoboMind, users can add or edit existing behavior modules on-the-fly without disrupting what the robot was previously doing. The interface allows users to inspect which behaviors are running in addition to the collected sensor samples from the robot's run-time.

RoboMind has been thoroughly tested and developed for usage with the latest generation of the Lego Mindstorms, the EV3. However, RoboMind can be used with other robotic platforms by making library additions in the source code. RoboMind offers a fully featured Python-programming environment for run-time modification of running EV3 programs with the help of the ev3-python library made by the author.

1.1 Problem Definition

From the problem definition of this thesis:

> *Develop a platform and architecture for on-the-fly inspection and modification of robot programs, focusing on the behavior system for the robots. The platform should allow on-the-fly visual inspection of the robot's activities and behaviors as well as run-time modification of the robot code. The platform should also be easy to deploy and use, both in the lab and when visiting schools and recruitment fairs.*

The required programming should be offered in the Python programming language. The required robotic platform for this thesis is the third generation of Lego Mindstorms, the EV3. It launched September 2013, which means it is still in its early life span. Unfortunately, it was discovered that most of the development environments on the EV3 are still in early alpha and beta stages and suffers from bugs. There also did not exist a stable enough environment to develop Python applications directly on the EV3, back in the beginning of this thesis. This meant that other alternative solutions on the EV3 must be made. The solution chosen was to implement a control program on the EV3 in one of the more stable development environments. Where a library implemented in Python is offered to the user that is able to control the program on the EV3.

1.2 Contributions

The contributions of this thesis are:

- **Robo:** A server control program, runnable on the Lego Mindstorm EV3 with the help of the leJOS framework. Robo provides a language independent control interface, accessible through network. Applications using the interface have direct control of the provided actions in an EV3.
- **ev3-python:** A library made in Python for controlling Robo. It implements Robo's control interface to give users programmable control of an EV3.
- **RoboMind:** A platform for on-the-fly inspection and modification of behavior-based robot programs. It uses Robo and the ev3-python library to give users a platform where they can inspect sensor information and modify behaviors on-the-fly for an EV3.

1.3 Limitations

Because of the limited timeframe for this thesis, the following limitations were made:

- **Security:** There has been no focus on removing the potential security risks in the system. Through the RoboMind platform, users are allowed to program code that will be executed directly on the server. Users could through this method potentially write malicious code and hack RoboMind. A potential solution could be to run the RoboMind platform in a virtual machine, where there is no concern if the system was compromised. Alternatively, use a sandbox version of Python, which isolates the potential unsecure code.

Chapter 2

Background

This chapter will first present some background related material to the Lego Mindstorms, a robotic platform by Lego. It will further present leJOS, the selected programming environment used on the latest generation of Mindstorms. The chapter is concluded by presenting the behavior driven programming environments offered to the user.

2.1 Lego Mindstorm

Lego Mindstorms[2] is a set of buildable and programmable robotic kits, made by Lego[1]. It gives the users the possibility of building whatever robots are possible through the different kits. The kits provides ordinary Lego pieces, a brain (also called the intelligent brick, or brick for short), and a set of modular motors and sensors. The motors and sensors are directly connected to the brick, which in turn control and collect data from them. What the brick does with the connected modules depends entirely on the program written by the user. The motors come in a variety of sizes and offers mechanical movements through rotations. Motors can pick up data such as if they are stalled, moving, or how many degrees it has rotated since the start. The sensors read sample data about the real world and offer it to the brick. What the sensor can read depends entirely on its type, e.g., touch, light, temperature, gyro, and distance. How this data is represented also depends on the sensor type.

[1] `http://www.lego.com/`

Different Lego Mindstorms kits provide different parts and may vary a lot depending on the generation and intended use. The different kits can be placed into one out of three generations[5]. Each generation stays around for about 5-7 years, before a new one is launched. These are:

- **RCX** (Robotics Invention System). It launched January 1998. Four sets came out. One basic kit, one adapted for educational use, and two upgraded versions of the basic set.
- **NXT** (as in *next* generation). It launched August 2006. Three sets came out. One basic set, an upgrade version of the basic set and a kit adapted for educational use.
- **EV3** (as in *evolution 3*). It launched September 2013. Currently two kits are out, 31313 (the ordinary kit) and the educational core kit.

The programmable intelligent brick of each generation can be seen in figure 2.1. The bricks represent the biggest change between each generation. Motors and sensors can to some degree be used between generations. Generally, the later generation support reuse of motors and sensors from the older ones. Differences within generations are generally quite small, most variations are found in the Lego blocks and the included set of sensors.

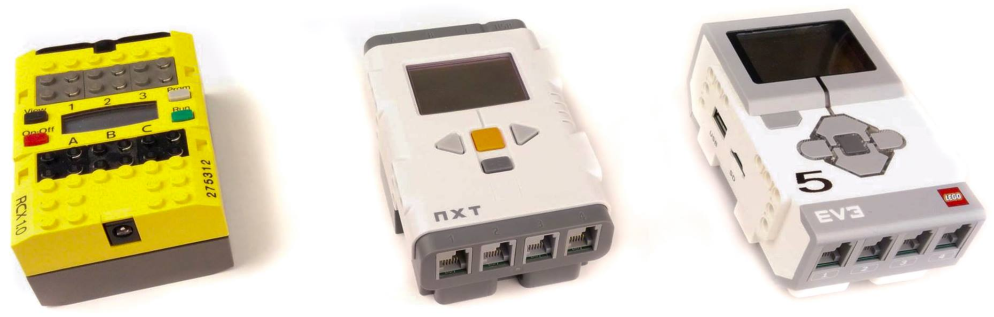


Figure 2.1: The programmable intelligent brick of each generation. Left RCX, Middle NXT, Right EV3

Programming software is included with each Lego Mindstorm kit. The software enables the users to write programs on their own computer, which can later be transferred and run on the intelligent brick. Programming is done through a graphical user interface. Actions on the brick are represented through drag and drop blocks. The user drags and drops these blocks together to make a program. A short program can be seen in figure 2.2 together with a description. Unfortunately, the brick does not by default allow any other programming options. To be able to use a general-purpose programming language like Java or C#, the firmware or operating system on the brick must be exchanged.

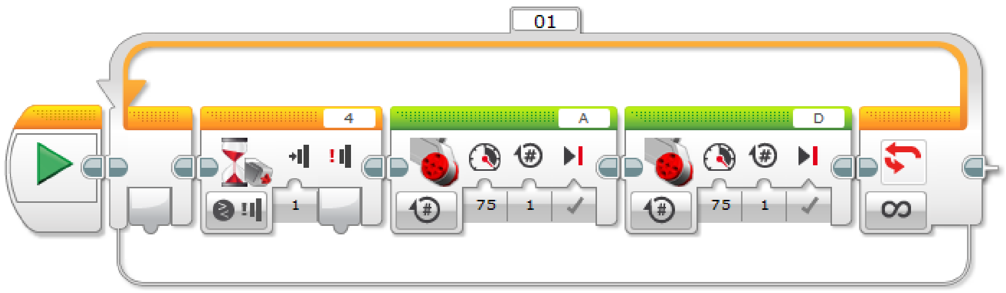


Figure 2.2: Shows a short program written in the default programming environment of the EV3. It is running an infinity loop around all the blocks. The first action inside the loop waits for the touch sensor on port 4 to be pressed. When pressed, the connected motor on port A rotates 360 degrees. Then the connected motor on port D rotates 360 degrees. Then the loop starts at the beginning again, waiting for the button to be pressed. This continues until the program exits.

2.1.1 EV3

For this thesis only the intelligent brick from the third generation is used (the EV3), as it was the required platform. Only the educational kit for the EV3 was available when writing this thesis, but the solution offered should work for both kits, as the intelligent bricks are the same. The kit contains one programmable brick, two large motors, one small motor, one ultrasonic sensor, two touch sensors, one gyro sensor and one color sensor. The set can be seen in figure 2.3, without the small motor.

The biggest upgrade in EV3 kits is the intelligent brick. In general, it is a much more powerful machine, with an extended set of functions. The specs of the EV3 compared the previous generation (the NXT) can be seen in table 2.1. The EV3 also offers the possibility of booting a secondary OS installed on the SD-card. This makes replacing the default environment on the EV3 extremely easy, as it only requires a bootable SD-card. It also gives

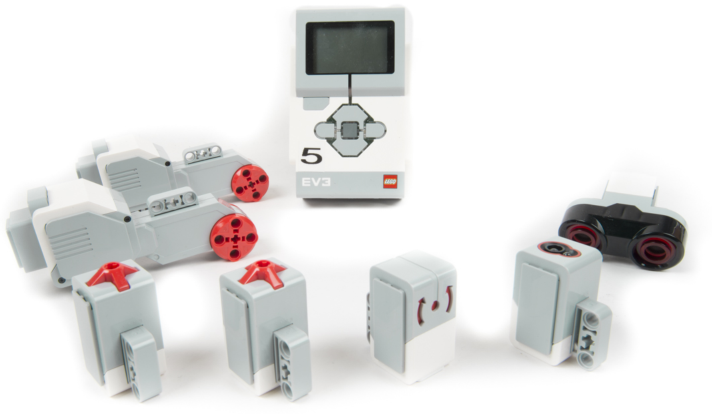


Figure 2.3: Main components in the EV3 education set, without the small motor

the possibility of booting the old default environment, as it is never removed. Previous generations of Lego Mindstorms required flashing new firmware into the brick, replacing the old one to allow other programming enviorments[23].

Even though the EV3 still is in its early lifespan, there already exists multiple open source environments to accommodate other programming languages. However, most of these are in early alpha and beta stages and suffers from bugs.

2.2 leJOS

One of the more stable and advanced enviorments available for the EV3 is leJOS[4]. leJOS is a Java programming environment and offers a release for each generations of Lego Mindstoms. It utilizes the improved spec on the EV3 to offer a fully features JVM (java virtual machine) over a Linux operating system. It is now possible to directly control and configure the brick through SSH. This is an improvement from previous releases as they only offered a self-written minified JVM and operating system to accommodate the lower specs[1].

It launched its first beta release 18-april-2014 and have continuously been updated throughout the period of this thesis. leJOS also offers a vigilant

	EV3	NXT
main processor	300 MHZ	48 MHZ
main memory	64 MB RAM 16 MB Flash	64 KB RAM 256 KB Flash
display	LCD 178 x 128 pixel	LCD 100 x 64 pixel
usb	1 port (480mb/s)	n/a
wifi	through external usb dongel	n/a
sd-card	micro-sd cards up to 32GB	n/a
bluetooth	yes	yes
operation system	linux	proprietary

Table 2.1: A comparison between the NXT and EV3 brick in specs. Table based on numbers gotten from [21].

community through its forums, where developers put out news and help their community by answering questions[3].

2.3 Behavior-based systems

Behavior-based systems[7] is an control type in robotics. Behavior-based systems focuses on making small behaviors that together through collaboration can complete complex tasks, without an internal representation of the world. A single behavior is a control block that is responsible for a particular situation. The actual movement of a robot is determined via the interaction between behaviors. The behaviors are often reactive, in the sense that it contain no internal representation of the world, it only reacts to information read from a robots sensors. In other words, the robot has no idea where in the room it is, but its behaviors can react to a wall by seeing it through the robots sensors and try to avoid colliding whit it.

Reactive behavior-based systems allows for a simple, but efficient design of robot programs. Through RoboMind, the reactive behavior-based architecture subsumption is offered to the users in its web interface. It is one of the earliest examples of behavior-based robots and is the typical example used in the field.

The underlying library also provides an implementation of the fuzzy-behavior architecture. It is not used in the RoboMind platform, as it is a much harder architecture to provide useful feedback back to the user interface. In fuzzy behavior, some behavior can half run, almost run, not run at all, etc. raising the difficulty of providing an easy to understand interface over what is running with the architecture.

2.3.1 Subsumption

Subsumption is an architecture proposed by Rodney Brooks in 1986[8]. It is an architecture where the complete behavior is decomposed into sub-behaviors in a bottom-up fashion. Each sub-behavior runs in parallel and has direct access to the sensors and actuators[2]. Typically, a sub-behavior has a sense-act function: they sense something and act upon it. Sub-behaviors are organized in a hierarchical arrangement, where higher-level behaviors subsumes (take advantage, or use) the lower behaviors to provide their behavior. The lower levels provides the more basic and fundamental behaviors. The higher levels provides the more complex behaviors by controlling and using the lower level behaviors. An example is given by Brooks in [8] where the lowest behavior provides the ability *"avoid objects"* and the second lowest behavior provides the ability *"wander"*. For the robot to be able to *"wander"* the robot first be able to *"avoid objects"*, in other words the second behavior subsumes the first behavior to be able to provide its ability. Higher-level behaviors also has the ability to access and inhibit the lower behaviors, however the reverse is not possible.

Subsumption emphasizes on an iterative develop and testing process, where layers one at a time is completed and thoroughly tested, before moving on to the next higher level behavior. Completed behaviors is never revisited as they were perfected the first time.

An important fact in subsumption is that there is no centralization of control. All sub-behaviors run asynchronously and in parallel to provide the complete behavior[24].

[2]Actuator is typically a motor. Something that makes movements possible.

2.3.2 Fuzzy behavior

Fuzzy behavior[20] share many similarities to subsumption. It also decompose a complete behavior into sub-behaviors with sense-act functions, all running in parallel. The difference can be found in how fuzzy behavior coordinates the execution of the sub-behaviors. Instead of making the higher levels responsible for controlling the lower levels outputs, fuzzy behavior removes the hierarchy. It utilizes fuzzy logic to decide what behaviors should run and fuses the active behaviors control outputs. Fuzzy logic is an approach where there exists multiple degrees of truth, not just true or false. For example the statement *"the water is cold"* could be 70% true. This can be used in behaviors to decide to what degree different behaviors should run. The logic can be expressed in the form:

IF context IS something THEN behavior

The rule expresses that the given behavior should be activated based on the strength given by the truth-value of the context. Multiple behaviors can utilize the same context to different degrees to allow running at the same time. By utilizing fuzzy logic, a system gains the advantage of expressing partial and parallel activation of behaviors, giving smooth transition between them. The weighted outputs from the fuzzy logic can then be fused together to provide the correct control outputs to actuators.

Chapter 3

RoboMind

This chapter will show how the platform RoboMind can be used. First, some setup details will be explained before presenting how the platform can be used in a step-by-step fashion. Footage of the same steps with the running robot can be found with the delivered source code and at [22].

3.1 Setup

The EV3s requires some setup before they can be used in the system. As described in chapter 2, leJOS is used instead of the default enviorment on the EV3s. With the help of the leJOS enviorment the author has built a control program called *Robo*, that allows external systems to connect and control an EV3. It is therefore required that leJOS and the program Robo is up and running on the EV3s before they can be used in the system. The installation instructions can be found with the source code. When the EV3s are properly configured, the RoboMind platform can start controlling them through the ev3-python library.

Full external control over the program Robo is provided through the ev3-python library built by the author. The library provides users with an extensive API over the available features offered through the program Robo. RoboMind uses this library to be able offer the same features in its interface. The last setup required is how the ev3-python library should connect with the EV3s. A connection can be established through one out of three channels: USB, Bluetooth or Wi-Fi. Through USB or Bluetooth the RoboMind system must be in close proximity with the EV3. Through Wi-Fi, the EV3 and the RoboMind platform can be located anywhere, as long as the EV3 is accessible. Wi-Fi demands some configurations on the EV3, such as password authentication. The router must also allow external access if the EV3s is not located on the same network as RoboMind.

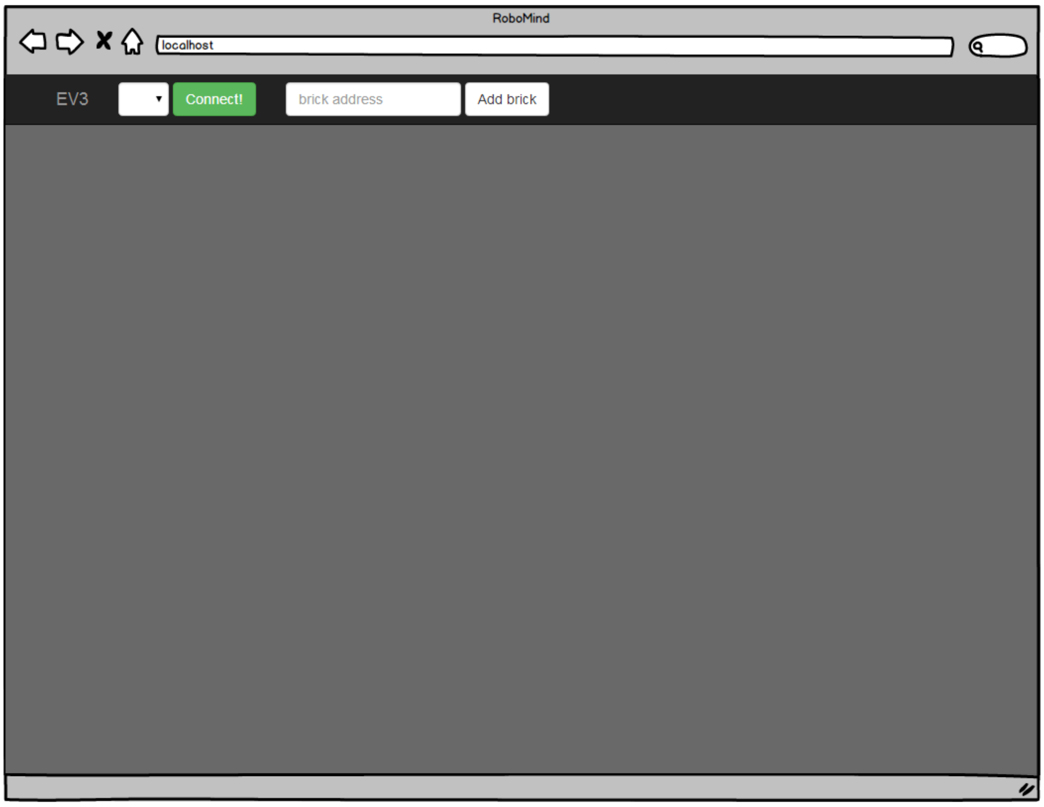


Figure 3.1: Screenshot of the initial view in RoboMind

3.2 Connect and configure

RoboMind gives users control over its resources through a web interface. The web interface can be accessed from anywhere through a user's web browser, although the user should have the robot in sight when using the system. In theory, by adding a web camera that shows where the robots are, users could easily use the RoboMind from anywhere.

The initial web view can be seen in figure 3.1. It shows an empty web page with a navigation bar on top. On the navigation bar, there are a couple

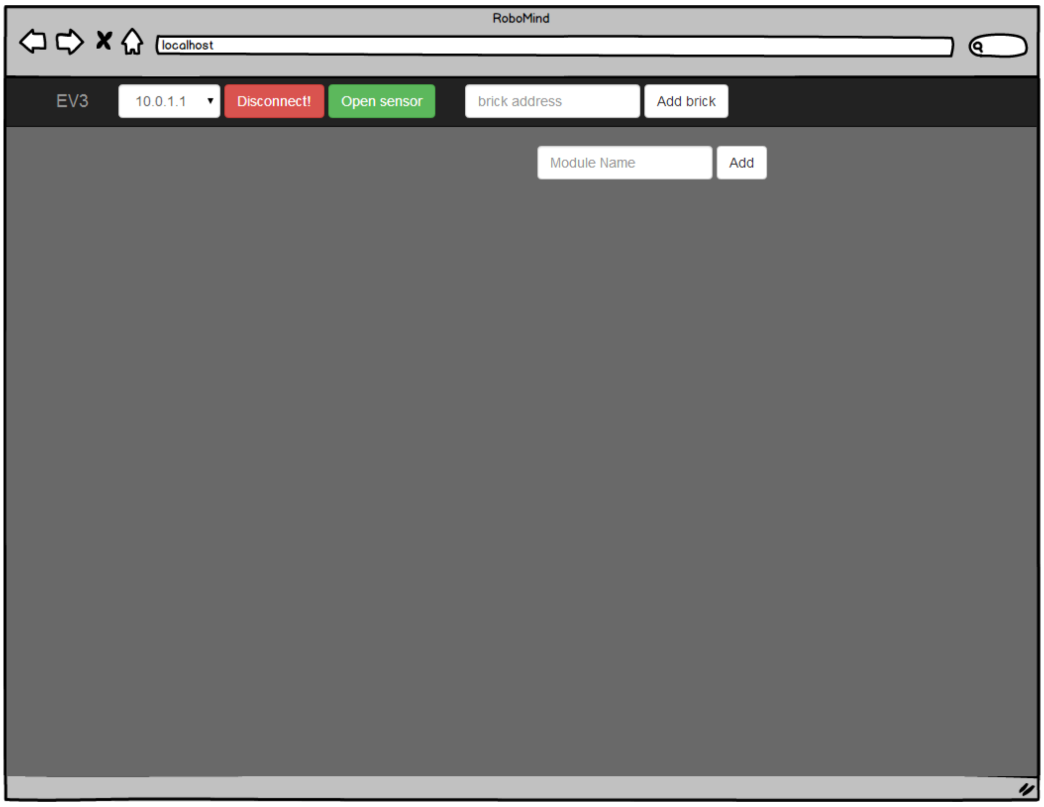
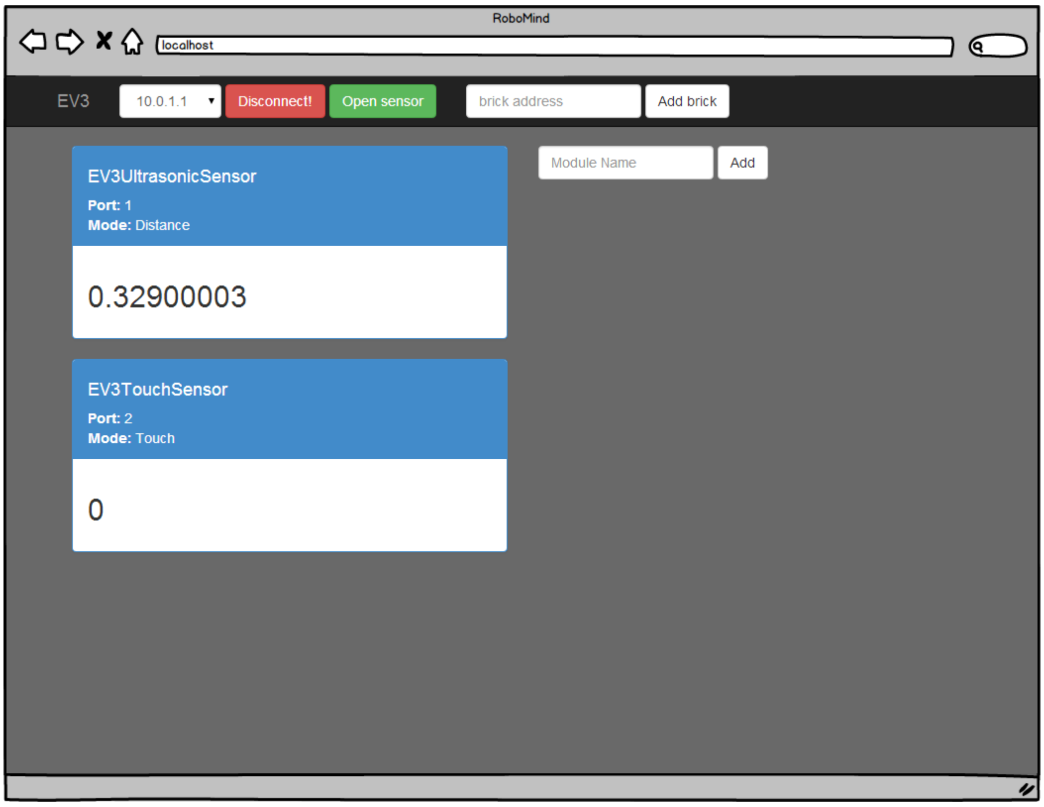


Figure 3.2: Screenshot of the initial view when the user has connected to an EV3

of buttons to allow users to add EV3s to the system and connect to one of them. The web interface can only have one EV3 connected at a time, since there is not enough room for the information from multiple connected EV3s. If a user wants to view the information from multiple EV3s at the same time, new browser pages can be used with new instances of the web interface.

With an EV3 connected the web interface will be updated with the latest stored information about that EV3 (see figure 3.2 for the typical view when connected to an EV3 without any information stored). A user has now full control over the different aspects RoboMind offers towards an EV3. These are mainly: Opening of a new sensor, or start editing behavior modules for the EV3. Unfortunately opening of new sensors is left as a task to the user. Automatic sensor discovery is rather limited on the EV3. Many of the sensors utilize the same drivers and offer little or no help when trying to figure out what type is connected. Automatic sensor discovery is offered to a limited degree in the underlying library, but it is too limited to be directly used in RoboMind. Because of this users are imposed the task of opening sensors



EV3UltrasonicSensor
Port: 1
Mode: Distance

0.32900003

EV3TouchSensor
Port: 2
Mode: Touch

0

Figure 3.3: Screenshot of the web interface when multiple sensors have been opened

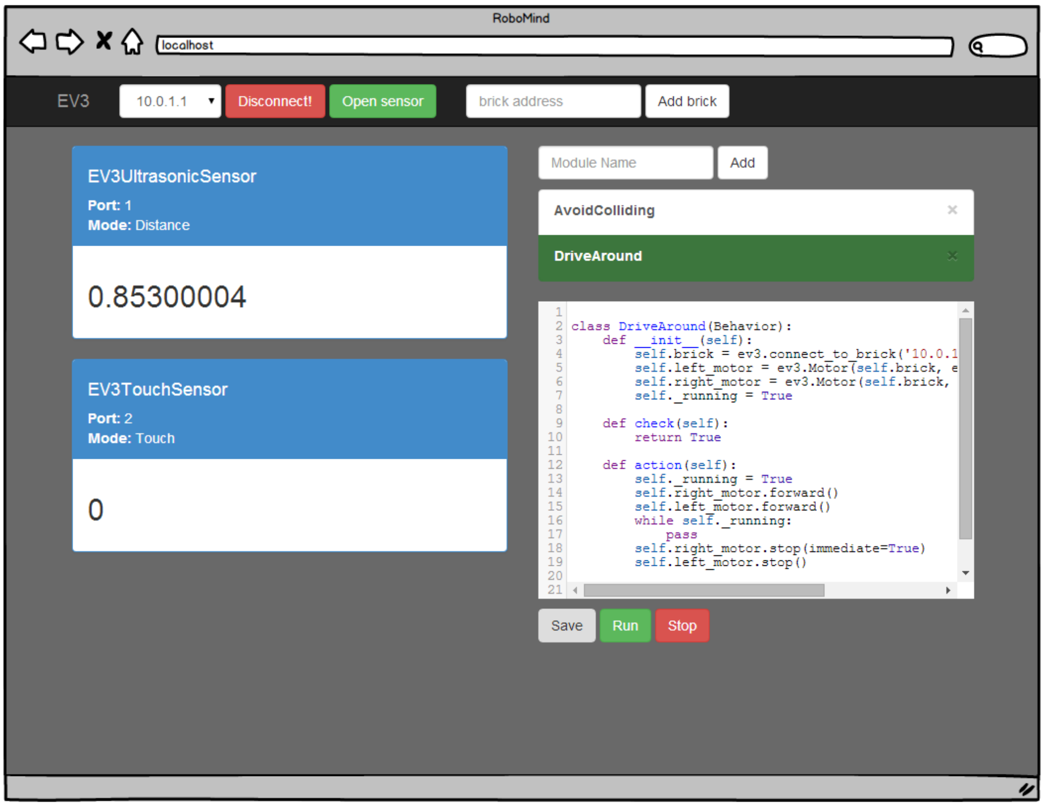
themselves. This can be done through the green button in the navigation bar. Adding new behavior modules is done by writing module names in the input field underneath the navigation bar.

A change on a connected EV3 is always stored in the memory of RoboMind. New clients connecting to an EV3 will always receive the latest stored information about it. This applies to both sensors and behavior modules. Information is also synchronized between web interfaces. Therefore, if a user updates a behavior module, the change will be stored in memory and get pushed to all the other connected clients. When an EV3 gets disconnected from the RoboMind platform all information about the EV3 is removed and connected web interfaces will receive an error notification. The main reason behind removing all the data and not offer a persistent storage of the information is because an EV3 may have changed its identifier the next time it connects. Meaning the platform has no sure way to know if a new or old EV3 has connected. Furthermore, when an EV3 gets disconnected, it is probably because the Robo program has stopped running. Meaning all state on the

EV3 is lost and information such as which sensors are connected needs to be reset. These things could be mended, but since the EV3s offers a lot of tinkering in its design and is meant to be rebuilt a lot, it made little sense to persistently store the state.

Figure 3.3 shows the view when multiple sensors has been opened. The squares represent sensors. The blue part contains the information's such as which sensor this is, what mode it is in and what port the sensor is connected through. The white part contains the last updated sample received on that sensor. As a step to minimize communication, only a change in the samples will be sendt from the RoboMind platform to the web interfaces.

If the user wants control over a different EV3, the user must first disconnect from the currently connected EV3. This can be done by pressing the red button on the navigation bar.



EV3UltrasonicSensor
Port: 1
Mode: Distance

0.85300004

EV3TouchSensor
Port: 2
Mode: Touch

0

```

class DriveAround(Behavior):
    def __init__(self):
        self.brick = ev3.connect_to_brick('10.0.1
        self.left_motor = ev3.Motor(self.brick, e
        self.right_motor = ev3.Motor(self.brick,
        self._running = True

    def check(self):
        return True

    def action(self):
        self._running = True
        self.right_motor.forward()
        self.left_motor.forward()
        while self._running:
            pass
        self.right_motor.stop(immediate=True)
        self.left_motor.stop()

```

Figure 3.4: Screenshot of the web interface when multiple behavior modules have been added and are running

3.3 Control and Program

The last part offered through the web interface is the behavior programming. Figure 3.4 shows a view where two behavior modules are running. Users can add new behaviors by writing names in the input box just underneath the navigation bar. Underneath the input box each behavior is listed. Behaviors can be deleted by clicking the close button. Underneath the behavior list is the code view for the selected behavior. The selected behavior can be changed by simply clicking on the different behaviors in the behaviors list. The selected behavior will have a dark blue or dark green color depending on the setting. Underneath the code view, there are three buttons: Save, Run and Stop. The save button saves all the changes in behaviors to the server. Run starts running the behaviors on the server. The currently running behavior will always have a green background color. Stop stops all the behaviors on the server. By pressing Run again, the behaviors will resume where it left of.

Even though it may look like the behavior modules are executed in the web interface, they are in fact transferred to RoboMind and executed. The web interface only gets updated by RoboMind telling it what is running. Scheduling of behaviors follows the subsumption architecture (explained in chapter 2) with a minor tweak, to make it easier to use. Instead of running all behaviors in parallel, RoboMind schedules behaviors one at a time. Behaviors utilize a priority queue, where the behavior list represents the priority of the behaviors. The behaviors further up have more priority. Scheduling is done in a non-preemptive fashion, where the behaviors themselves must release control before a next one can run. When a behavior with a higher priority wants to run, the currently active behavior receives a suppress signal. It should then by itself release the control and let the behavior with a higher priority run.

Behavior modules must be programmed in the programming language Python[1]. A behavior module must implement the interface shown in listing 3.1 to be executable.

A behavior module must be implemented in a class that contains the three methods: check, action and suppress. The class should contain everything related to that specific behavior. In other words, it should contain all the necessary code that makes the desired behavior possible.

The method check should check the conditions of the behavior, should it run

[1]`www.python.org`

Listing 3.1: Behavior module interface

```
class SomeName(Behavior):
    """
    Should contain a specific behavior belonging to a robot.
    Each Behavior must define the three following functions:
    """
    def check(self):
        """
        Does this behavior want to start running?
        Should then return true.
        """
    def action(self):
        """
        What should this behavior do while it has control?
        Should stop when suppress is called.
        """
    def suppress(self):
        """
        Should release control by stopping action(),
        so other behaviors can start running.
        """
```

or not? Typically, it accesses the available sensors and checks if the values correspond to a setting when the behavior should run.

Action contains the action code of the behavior. Typically, it accesses the actuators on the EV3 and does an action, like rotating the wheels forward. The behavior has control as long as it is inside the action method and new behaviors will not be scheduled before the method exits.

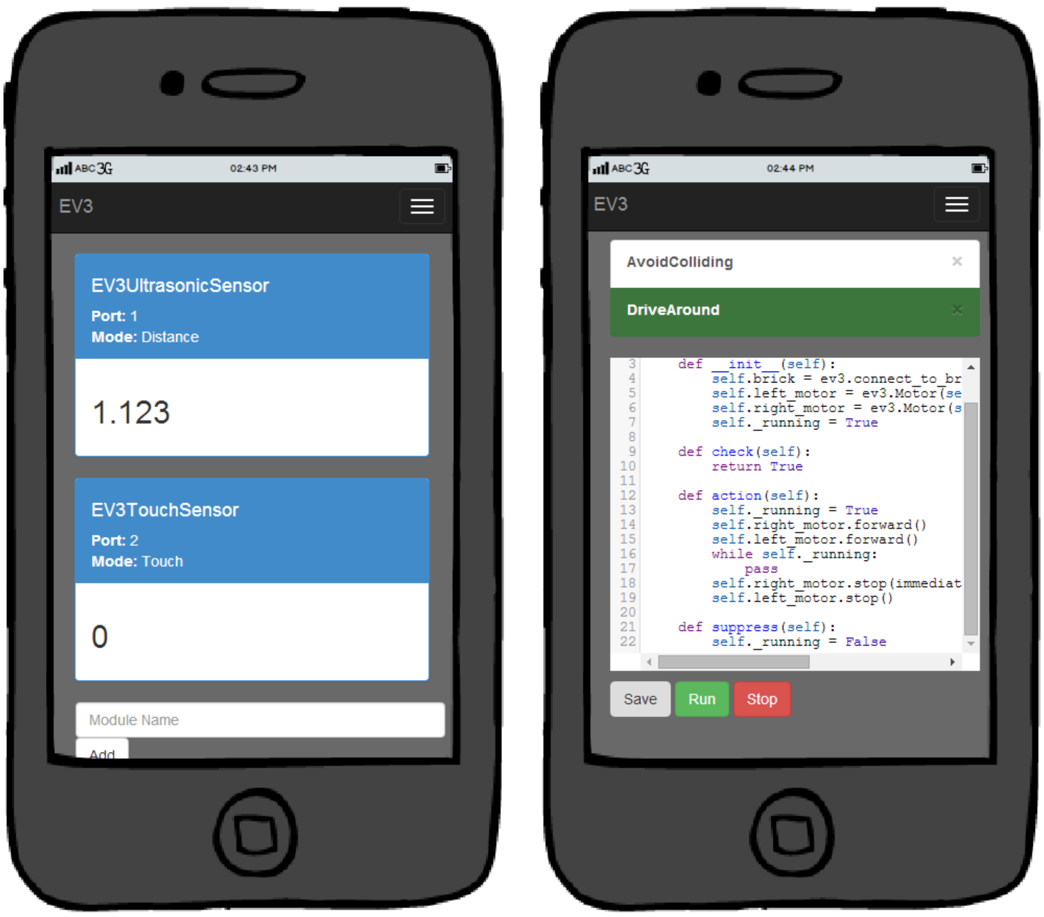
Suppress is called when the behavior should no longer run. This method should stop whatever action the behavior has started and make the behavior exit the action method. When a behavior exits the action method, the behavior releases its control and allows new behaviors to be scheduled instead.

Mistakes found in the code are caught and displayed to the user in the web interface. The code will not run without fixing the mistakes.

The behaviors modules used in the screenshots and in the video can be seen in appendix A for a more complete look at a whole example.

3.4 Responsive Design

The web interface support a wide range of different devices through the use of responsive web design. It automatically resizes and relocate the different components within the web interface when a device with a different screen size is used[9]. Figure 3.5 shows how the web interface typically will look on a mobile device.



```
    def __init__(self):
        self.brick = ev3.connect_to_br
        self.left_motor = ev3.Motor(se
        self.right_motor = ev3.Motor(s
        self._running = True

    def check(self):
        return True

    def action(self):
        self._running = True
        self.right_motor.forward()
        self.left_motor.forward()
        while self._running:
            pass
        self.right_motor.stop(immediat
        self.left_motor.stop()

    def suppress(self):
        self._running = False
```

Figure 3.5: Screenshot of the web interface on a mobile device. A look at the responsive design

Chapter 4

Architecture

RoboMind is a client-server system for on-the-fly programming and inspection of EV3s. It provides users with a web interface for inspecting and controlling the available resources at RoboMind. RoboMind utilizes the ev3-python library made by the author to provide users with its services towards EV3s. This chapter presents the architecture of RoboMind and the different components it uses. It will first present the architecture between the program *Robo* running on the EV3s and the ev3-python library, before moving on to the general architecture of RoboMind.

4.1 ev3-python library

The library built enables a user to connect and control an EV3 with the use of the programming language Python, hence the name ev3-python. Since Python is an interpreted language, it means that a user through ev3-python can easily experiment interactively with connected EV3s.

Figure 4.1 shows the architecture of the ev3-python library connected to a single EV3. The architecture follows the typical client-server model[18], where the EV3 acts as the server and the ev3-python library act as the client. External access to the available resources at the EV3 is provided through the program *Robo*. Robo is a server application running in Linux-operating system of the EV3 with the help of the leJOS framework. External applications can connect to Robo through three different connection channels: USB, Bluetooth or Wi-Fi. When connected an application can start control-

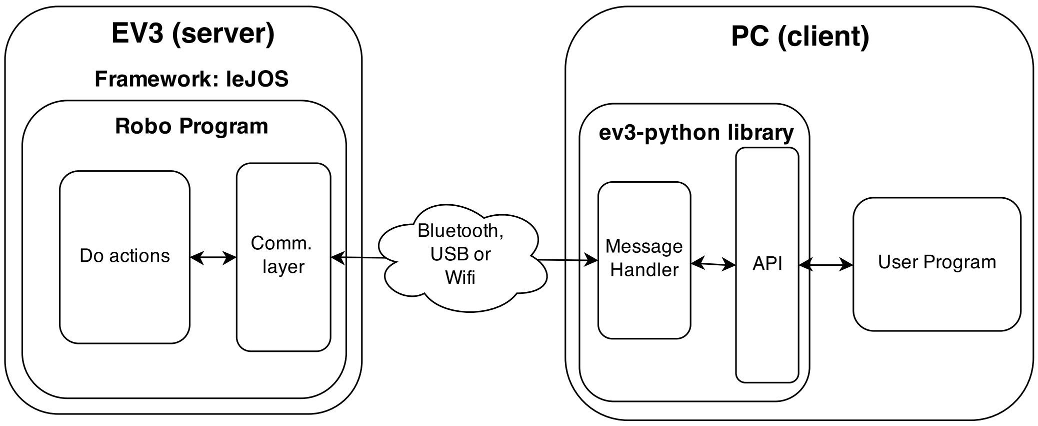


Figure 4.1: Architecture of the ev3-python library

ling the EV3 by sending control messages to Robo in the JSON[1] format. The ev3-python library implements the message API of Robo in its underlying layers to provide users with a usable programming interface. In theory, any application that implements the message API of Robo can directly control the program without the need of the ev3-python library.

The architecture between the ev3-python library and an EV3 can be thought of as a connect-control type instead of a connect-program type. Programming is offered by using the interface, but in reality, the user is not programming on the EV3. The connected EV3 only follows the commands given to it by the server program Robo. The architecture only gives the users an illusion of programming directly on the EV3 with the Python language. This is done as there did not back in the beginning of this thesis exists a good enough enviorment to directly program on the EV3 with the Python programming language. However, by using a program such as Robo the same result is provided: to be able to program the EV3 with the Python language and modify the robot programs and runtime. Robo also provides a platform where the computational need is removed from the less powerful EV3 to a more capable computer. Which can be better depending on the application.

The EV3 is chosen as the server since it allows for less cumbersome configuration when redeploying the system. If the roles was reversed and the EV3 was the one who should initialize the connection, the EV3 would need to be reconfigured each time the ev3-python library switched location. Either through parameter configuration inside the EV3 program, or using the six available buttons on the EV3. By giving the EV3, the server role in this

[1]http://www.json.org/

relationship the configuration need is given to the ev3-python library, where it is easier to handle.

External applications are required to stay connected to Robo for the duration they intend to use the EV3. As long as an external application stays connected, Robo stores its state for quick access later. Some operations such as opening of sensors can take up to 5 seconds, keeping opened sensors in memory saves a lot of time when accessing their operations later. The saved state is deleted when the connection is shutdown. In other words Robo resets itself between each connection. Since Robo is a state full server application, it means inconsistency issues may arise if multiple connected applications can make changes in its state. To remove the inconsistency issues, only one connection is allowed at a time. This also increase the efficiency of Robo, which is already running on very limited resources.

Lastly, streaming services is also offered by Robo and must be activated by a request. Robo offers streaming of sensor samples and sensor discovery, while the usual request-response communication can happen as normal. In other words, Robo can automatically push information on newly connected sensors and new sensor samples continuously to the client.

4.2 RoboMind

Figure 4.2 shows the architecture of RoboMind. It consists of three main components: Multiple EV3s and multiple web browser clients connected to a single Master. Users can connect to the master through their web browsers to gain access to the web interface in a typical client-server fashion. This is done by using the REST interface[10] at the master's web server. Through the web interface users have direct control over the master and its resources, the EV3s. The web interface enables users to add, inspect, control and program EV3s at the master, as shown in chapter 3. Since the master provides users with a web interface accessible from a web server, users can access the system from anywhere as long as they have access to the internet, although users should have in sight the EV3 they want to control through the system.

From a user's perspective, the master may seem transparent, as the web interface is built to give the illusion of direct access to EV3s. When in reality all communication passes through the master. Nothing is directly run in the web interface. It only provides a view for the user over the available resources at the master. Everything is transferred from the web interface in the form of JSON messages to the master. This also applies to the code written in the web interface. In fact, the master is the one who runs the code written in the web interface. The master continuously provide feedback to web interface as to what part of the code is running and programming flaws found. The web interface updates it view based on these feedbacks to give the illusion of code running directly in the web interface. Changes made on the EV3s or in the behavior code will be stored in the master's memory until told otherwise. The information is saved until either a user deletes it, or the EV3 gets disconnected entirely from the system.

In theory, users could through their web interfaces directly connect and control EV3s without the need of the master as a middleware. There is two main reasons why this is not done. The first is the limited resources typically available in the web page. For security reasons, JavaScript running from a web page is not granted full access to the resources of a machine[14]. Because of this, it is difficult to provide a full programming environment for Python. Meaning it will be even more difficult to fully allow usage of the ev3-python library in a web page. Secondly, only one connection is allowed to the program Robo. Through the master, this limitation is removed by sharing the known and updated information about the EV3s. Updates in the stored information are propagated to all interested web interfaces as a step to give a consistent view of the current state in an EV3. In other words if a user makes a change on an EV3, say in what sensors are connected, all users who are interested in the same EV3 will receive an updated view when changed

The architecture of RoboMind is module based. It is possible to switch out the components with different ones as long as they follow the same API, or use the components in other systems by making minor changes in the code. Additions in the supported robots could be allowed by extending the set of libraries used in RoboMind.

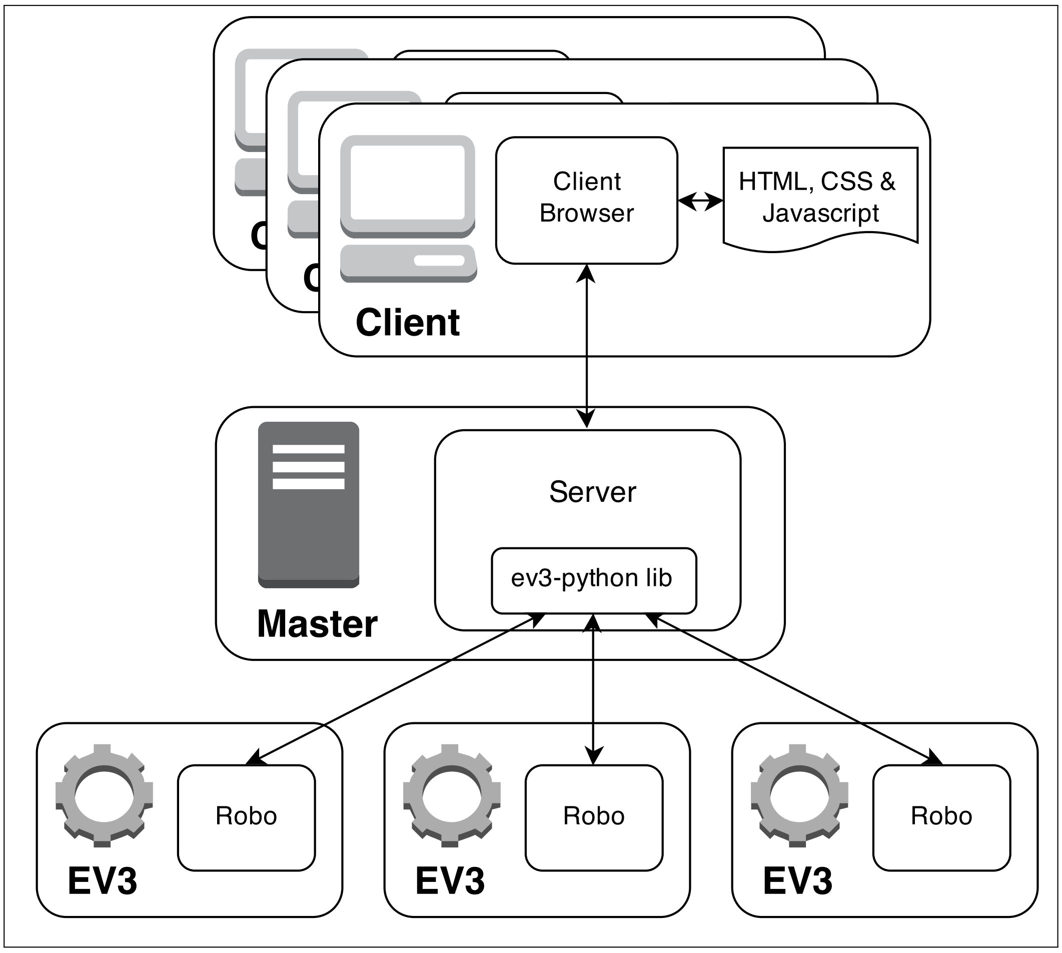


Figure 4.2: Architecture of RoboMind

Chapter 5

Design and Implementation

This chapter presents the design and implementation of the different components in the RoboMind platform. It will present the components in a bottom-up fashion.

5.1 Robo

Robo is a socket-based server program that is runnable on the Lego Mindstorm EV3s. It allows connected clients to control of a wide variety of actions on a Lego Mindstorm EV3 through its programming language independent JSON interface.

Clients can connect to Robo through either Bluetooth, USB, or Wi-Fi with the use of the appropriate socket. Sockets can either follow the Transmission Control Protocol (TCP)[1] for communication over Wi-Fi or USB, or the RFCOMM protocol[2] for communication over Bluetooth. Both protocols allows for a reliable communication exchange between the client and Robo.

A connected client is able to control Robo by the exchange of JSON messages. The communication is typically done in request-response pairs, where each request from the client will receive a response from Robo. There is one request-response pair for each action the client wants Robo to execute.

[1]`http://www.ietf.org/rfc/rfc793.txt`

[2]`http://developer.bluetooth.org/TechnologyOverview/Pages/RFCOMM.aspx`

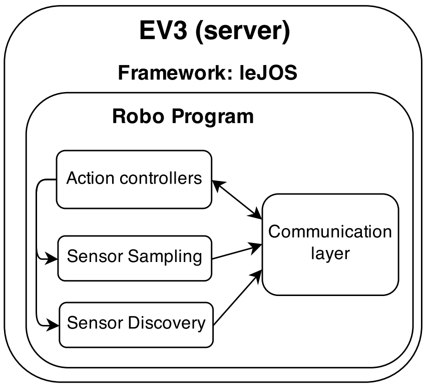


Figure 5.1: Design of Robo

Figure 5.1 shows the design of the Robo program. Messages received from a client will be received in the communication layer of Robo. The messages will be parsed and depending on the message invoke one of the action controllers. Action controllers can control things like sensors, motors, etc. After the action is completed, a response is sent back to the client, telling it if everything went as expected or not. Streaming services is also offered by Robo and must be activated by a request. Robo offers streaming of sensor samples and sensor discovery. Both of these services runs in separate threads.

The sensor discovery thread automatically pushes new sensor information to the client. The information contains the most accurate description of sensors connected to the EV3. It will automatically push data when sensors has been connected or disconnected. When a sensor is connected, it will push data such as the name of the sensor and what port it is connected to. However, in some cases it is impossible to automatically tell what sensor is connected. In these cases, only the sensor type is sent. Note that the streaming service does not open a connected sensor, it only tells client what has connected and where. The leJOS framework provides the automatic sensor discovery.

The sensor sampling thread automatically pushes fetched sample data on the opened sensors. It continuously loops through the opened sensor and fetches the samples, which are pushed to the client. The service will wait for sensors to connect if there are not any available.

As explained in chapter 4, Robo only allows one connected client at a time. The client is also required to stay connected to Robo for the duration it want access to the EV3. This is done as a step to increase the speed of some operations. Operations such as opening of sensor can take a lot of time, and is intended to stay opened while the program runs. Robo treats new connected clients as new program starts. Meaning Robo does not have anything pre-saved between each connection. New clients would have no clue if a previous client left motors preset with a too slow or too high speed. Robo simply treats each disconnect as a new beginning and refreshes everything in its memory before allow a new client to connect. This is also Robo's fault tolerance. Non-intended errors causes Robo do disconnect its client, which refreshes its state.

To ease the use of Robo a few extra services is provided. When Robo is running and no client is connected, Robo broadcast its unconnected state by blinking the EV3's led lights in an orange color. When a client is connected, the lights stays green. Robo also broadcasts its hostname on the networks so clients who does not know its address can easier find it. Broadcasting by Bluetooth is automatically provided by the EV3, while multicast[3] is provided in Robo through the leJOS framework.

5.1.1 Communication

For a request to be valid it must follow the specified JSON interface seen in listing 5.1. The interface shows the minimal JSON message required to be a valid request. There may be additional required attribute-value pairs depending on what type of request is sent.

Listing 5.1: JSON request interface

```
{
  "cla" : "", // type of command, motor, sensor, etc.
  "cmd" : "", // the command
  "seq" : 0,  // sequence number
  //... may be extra data depending on the request
}
```

A response from Robo will always look like the interface given in listing 5.2. There is no other attribute-value pairs, but some of the pairs may be omitted if there is no data on them. This is also how a streaming package from Robo will look.

[3]http://www.tldp.org/HOWTO/Multicast-HOWTO-2.html

Listing 5.2: JSON response interface

```
{
  "msg" : "", // Response message / Type of response
  "data" : 0, // Data from motors and such
  "seq" : 0,  // Sequence number. The same as in request
              // If this is a request packages

  "sample_string" : "" //When string is needed for the data

  //Sensor fields
  "sample" : [], //Sample array
  "samples" : [ [], [], [], [] ], //Array of all samples
}
```

The sequence number is Robo's method to allow multiple data streams running on the same socket. Each request sent to Robo must contain a sequence number. The same sequence number is used in the response to that request. Streaming packages do not use the sequence number. By using a sequence number Robo allows for both a request-response and streaming communication over the same socket. It is up to the clients to use the sequence number the correct way, by either increasing it, or using a pseudo-random number. As long as it is unique enough for the client to know which response belongs to what request.

5.1.2 Technologies

Robo is implemented using the Java programming language[4] and the framework leJOS. The leJOS framework is used to control the actions of an EV3. Robo is mainly implemented in Java because it is the only language supported by the leJOS framework.

For the JSON communication, the Gson[5] library is used. Gson is a Java library for deserialization and serialization of JSON to and from Java objects. It ease the use of JSON in Java.

[4]http://www.java.com/
[5]https://code.google.com/p/google-gson/

5.2 ev3-python

ev3-python is a Python library that gives users programmable control of an EV3 through the Robo program. It has implemented Robo's control interface in its lower layers to be able to provide programmable control to the user.

The Python programming language is used since it is a dynamic, interactive language and easily allows users to experiment interactive with connected EV3s. It also is a very productive language and familiar to the author.

Figure 5.2 shows the design of ev3-python library. Basic usage consists of using the API to connect and control an EV3. All communication between the ev3-python library and the EV3 will first pass through the asynchronous message handler. The asynchronous message handler runs in its own thread and is tasked with the sending and receiving of all messages between the ev3-python library and the program Robo. The asynchronous message handler keeps track of what requests are waiting for which response and can block their thread until the response is received. Any action that wants a response from the EV3 before continuing, like opening of sensors, will be blocking calls until the response is received. Other actions that does not require a response, like drive forward can be blocking calls or immediate calls. Immediate calls does not wait for a response and proceeds with its execution. Blocking is implemented by using the sequence number in the messages to block requests threads until their sequence number is received and their response can be provided.

The user also has the option of starting a subscription on the two different data streams. In this case, the API will send a message to Robo telling it to start the streaming services. Streaming messages received will be parsed continuously in the asynchronous message handler, who will in turn send these to the subscription module. The subscription module is responsible for making callbacks back to the user's program when a streaming message of the different types is received. The callbacks are executed in their own separate thread as to not block the rest of the API.

5.2.1 Demonstrational use of the ev3-python API

The use of the library is designed around the connected EV3, the brick object. Other external components (e.g. sensors and motors) transfer their commands by uses the brick object as the communication point. The brick

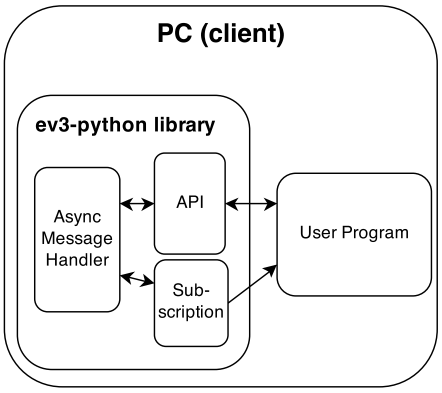


Figure 5.2: Design of ev3-python library

object and the opened components on that brick object is stored in memory, where reuse of the same connection addresses causes reloading of the same objects. In other words if a new instances of the brick object is created where the same connection address as a previous connected brick is used, the same brick will be loaded from memory. This allows for an easier design in the RoboMind platform, where behaviors can easily share the same brick object by using the same address.

The following paragraphs shows how the library can be used by explaining short code snippets.

Connect to brick and read sensor samples

Listing 5.3 shows how the library can be used to connect to an EV3 and open two different kinds of color sensors, the Hi-tech and EV3 versions.

After the sensors has been opened, the colorID mode is extracted into separate objects. Each sensor in the library has their own set of multiple modes. Different modes represent entirely new states in the sensor and even the data size may very between modes. Activation of modes can also take a long time. Modes are extracted into their own object for this very reason, to make sure users know what type of mode they are using.

The rest of the code shows how color samples can be extracted from the mode in a loop.

Listing 5.3: Color sensor reading

```
import ev3

brick = ev3.connect_to_brick(address='10.0.1.1')

color_sensor1 = ev3.EV3ColorSensor(brick, port=1)
color_sensor2 = ev3.HiTechnicColorSensor(brick, port=2)

colorid_1 = color_sensor1.get_selected_mode()
# same as above
colorid_2 = color_sensor2.get_color_id_mode()

for _ in range(0, 4):
   print colorid_1.get_color_id() # prints color name
   print colorid_2.get_color_id() # prints color name
```

Motor control

Listing 5.4 shows how the library can be used to control connected motors. The code shows how two motors is opened on port *"A"* and *"B"*. Then a couple of actions on the motor is invoked, the first one leads to the motor running forward. When the program terminates the motors will stop.

Subscription example

Listing 5.5 show how the library can be used to subscribe on events. Subscription is handled a little differently than the rest of the components in the library. Subscription is first initialized by itself where the proper configuration is done, before the brick object uses it. This is done because the subscription require a lot of configuration to work properly. This way the subscription object can be used in multiple bricks without requiring new subscription configuration for each new brick. The code shows how subscription can be activated on the brick and how a callback function can be registered on the streams received. The code will print samples recived for four seconds before terminating.

Listing 5.4: Motor control

```python
import ev3
import time

brick = ev3.connect_to_brick(address='10.0.1.1')
a = ev3.Motor(brick, port='A')
b = ev3.Motor(brick, port='B')

a.forward()
b.forward()

time.sleep(2) # Let the motors dive forward 2 sec
print "Moving?", a.is_moving(), b.is_moving()
print a.get_position(), b.get_position()
```

Listing 5.5: Subscription example

```python
import ev3
import time

brick = ev3.connect_to_brick(address='10.0.1.1')
color_sensor = ev3.EV3ColorSensor(brick, 1)


def print_samples(samples):
  print samples

sub = ev3.Subscription(
 subscribe_on_sensor_changes=False,
 subscribe_on_sample_data=True)

sub.subscribe_on_samples(print_samples)

brick.set_subscription(sub)
time.sleep(4) # stream for 4 seconds
```

5.2.2 Technologies

For Bluetooth communication the Pybluez[6] library is used. It is required to have Pybluez installed to use Bluetooth communication in the ev3-python library. However, it is not required if the Bluetooth part of the library is not used.

5.3 RoboMind

Figure 5.3 shows the design of the RoboMind platform without connected EV3s. It shows the master server and a web client connected to it. At the master server, there are four main components: the Web Server, the WebSocket Server, the Brick Manager and the Code Manager.

The master server is implemented in Python, while the web interface is implemented in HTML, CSS and JavaScript. Do note that the RoboMind platform is only a prototype and there are still room for improvements.

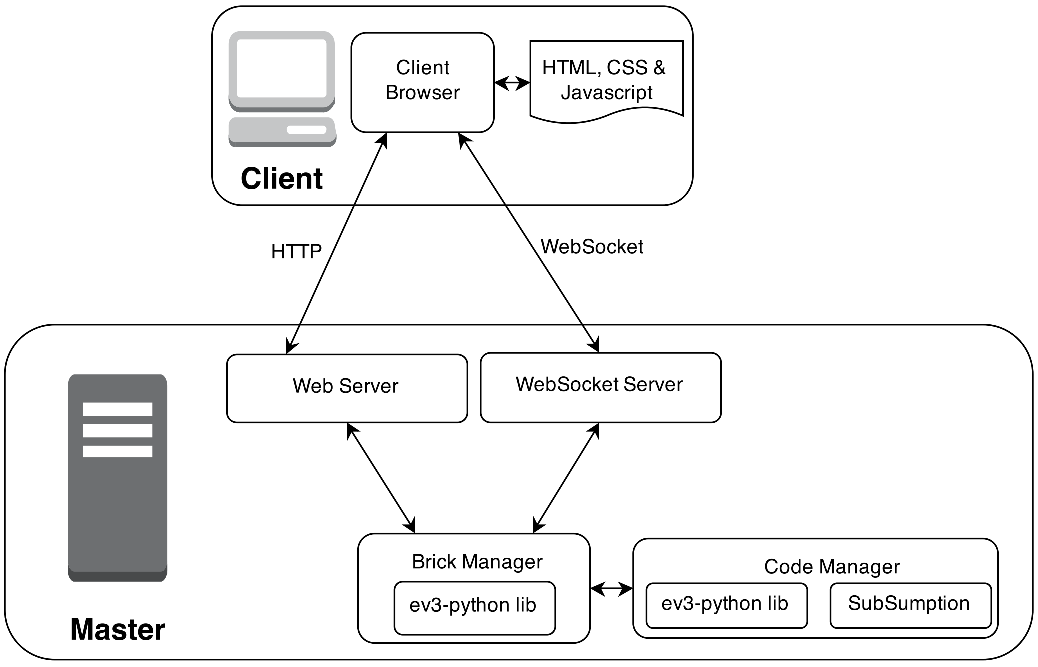


Figure 5.3: Design of RoboMind, without EV3s connected

[6] https://code.google.com/p/pybluez/

5.3.1 Web server

The web server is responsible for delivering the static web interface files to the clients through its REST API. It is also responsible for providing some additional information between the clients and the brick manager. Such as what EV3s are available for the client and making the RoboMind platform connect to new EV3s. The rest of the communication on the platform requires that the client established a stable connection to RoboMind by the use of WebSockets[7].

Technologies

The web server is implemented using the Flask[8] micro framework. Flask is a web micro framework that ease the implementation of web applications. Flask comes with its own RESTfull request dispatcher, meaning it delivers most of the static files automatically without intervention.

For the extra communication needs between the clients and web server there is provided a JOSN-RPC[9] interface on top of Flask. By using JSON-RPC, adding new communication options between the client and web server can easily be done, as well as removing some.

5.3.2 WebSocket Server

When clients' wants to register their interest in an EV3, they connect to the WebSocket Server and specifies which EV3 they are interested in. Communication about that EV3 will now be pushed between the client and the RoboMind platform over the connection WebSocket. This involves data such as behavior code, sensor samples, etc.

WebSockets is used since it easily allows pushing of data between the master and the web clients. Meaning updates in information will happen more instantly than if the usual polling mechanisms was used. WebSockets also provides less the latency and requires less resources then the traditional pulling mechanisms[19].

[7]`http://www.websocket.org/aboutwebsocket.html`
[8]`http://flask.pocoo.org/`
[9]`http://www.jsonrpc.org/specification`

Technologies

The SimpleWebSocketServer[10] library is used to provide the WebSocket server. SimpleWebSocketServer is an open source Python library that easily allows for implementing WebSocket logic. It takes care of implementing the Web-Socket protocol in its underlying layers.

5.3.3 Brick Manger

The Brick Manager is mainly responsible for monitoring which clients is connected to which brick. It can be thought of as a gateway module who ensure that the communication is sent to the right places.

The brick manager is responsible for directing behavior code to the code manager and to ensure the all clients has the latest updated code. The brick manager also sends out updates on what is running from the code manager to the clients.

The brick manager also ensures that the received data streams from the EV3s is sent to the interested clients. As a minor tweak to lessen the communication between the master server and its clients, only an update in the sensor samples will be sent out. If there are no changes in the received sensor samples the brick manager will not send the same data to clients who has already received an identical message.

5.3.4 Code Manager

The code manager is responsible for running the behavior code. It uses the Subsumption library to continuously run and update the behaviors added by the clients. Behaviors sent to the code manger is evaluated and extracted by using Python's *"exec"* function. The Subsumption controller is implemented by using two threads. One thread run the *"action"* method for the active behavior. The other thread continuously loops through all the behaviors and checks who one wants to be active. If a thread with a higher priority than the one currently running wants to become active it calls the *"suppress"* method on the currently active behavior. The Subsumption controller then updates the active behavior to the new one.

[10]`https://github.com/opiate/SimpleWebSocketServer`

When a client updates a behavior, it is stored in the code manager and sent to all other clients. If the behavior updated is the currently running behavior, it is switched out and the *"suppress"* method for the old behavior is called. This will lead to the running behavior monetarily stopping to allow for a change.

Behaviors added to the code manager has full access to the ev3-python library and Python interpreter on the master server. This means the behavior modules can potentially be huge security risks, but it is simply ignored in this prototype.

5.4 Web interface

The use of the web interface was shown back in chapter 3. The web interface is a single-page web app executed in the client's web browser. The JavaScript[11] code is responsible for updating different parts of the web view based on the received information from the RoboMind master server. To structure the JavaScript code and make the code more modular the framework Backbone[12] is used. Backbone provides a loosely-MVC (Model, View, Controller) structure to the web application. It is loosely because the View behaves as the controller and the view.

Technologies

Typically JavaScript files is loaded by inserting a *script* tags into the HTML file. This works great for small applications when there are few JavaScript files. However, when there are many JavaScript files, some depending on others to work, inserting a *script* tag for each file and in the right order can become rather difficult. This is typically the case when it comes to backbone applications as there is often one file per module and some are depending on being loaded in the right order. RequireJS[13] is used as the module loader in this web application to remove these issues. RequireJS is a JavaScript file loader that support nested dependencies. It also provides a tool to optimize the JavaScript application by minimizing all the files into one single file, minimizing some of the file loading latency. RequireJS also removes the naming conflicts in models by splitting different modules into

[11]`http://www.w3schools.com/js/`

their own namespaces.

Backbone requires that the Underscore[14] library is included in the project to work properly. It is required since Backbone uses many of its functionalities. Underscore is also the template engine used by backbone to load HTML file templates into the web view.

JQuery[15] is also a library required by Backbone. It is a small library that easier allow things like HTML manipulation and event handling.

The code editor in the web interface is provided by the library CodeMirror[16]. CodeMirror is a text editor implemented in JavaScript for editing code in a magnitude of different programming languages.

To easier implement the front-end design the bootstrap framework[17] was used. Bootstrap is a front-end framework that provides multiple HTML, CSS and JavaScript design components for building web applications. If used properly the framework automatically provides a responsive web design. The web view shown in chapter 3 has been built by using the components in the framework as shown in its examples.

5.5 Behavior library

5.5.1 Subsumption

Implementation of the Subsumption library has been inspired by a solution found in the leJOS framework. However, since leJOS is a Java framework it cannot directly be used as in the RoboMind platform, which is implemented in Python.

[13] `http://requirejs.org/`
[14] `http://underscorejs.org/`
[15] `http://jquery.com/`
[16] `http://codemirror.net/`
[17] `http://getbootstrap.com/`

5.5.2 Fuzzy behavior

Fuzzy behavior was provided as a means to more easily allow parallel behaviors and to prove that the ev3-python library provided could be used in other behavior architectures. Unfortunately running parallel behaviors, and provide sensible feedback to a use interface, is rather difficult within the limited timeframe of this thesis, so it was dropped. The library is available with the source code. The design of the fuzzy behavior code was inspired by examples of how Pyro[18] (Python Robotics) library used it.

With the source, code there is provided an example robot who uses the fuzzy behavior. The robot drives around and at the same time tries to avoid colliding with objects. The closer it gets to an object the more it tries to avoid it by turning.

[18] `http://pyrorobotics.com/?page=PyroModuleBehaviorBasedControl`

Chapter 6

Evaluation

This chapter begins presenting the test and experiments done on different components in the system. Then the chapter moves on to a discussion on what can be improved and issues found during the development.

6.1 Experiments

6.1.1 Experiment environment

The experiments has been run on the author's own computer. The computer has the following configurations:

- **Operating System:** Windows 7
- **Memory:** 16GB RAM
- **CPU:** Intel(R) Core(TM) i7-3820 3.60GHZ, Quad Core

The configuration for the used EV3 can be seen in table 2.1. The EV3 uses leJOS version 0.6.0-alpha. The RoboMind platform has been developed on version 2.7.3 of Python.

The RoboMind platform has been tested on both windows and Linux and works as expected on both. The tests presented in this chapter and the use case presented in chapter 3 shows that the prototype works as designed. The latency between the Robo and the ev3-library shows that it is possible to exchange a couple of hundred messages within one second.

The following subsections contains the information about the different tests used and what the results was.

6.1.2 Robo

CPU and Memory Usage

Performance measurements of the program Robo has been done with the Linux program *top*. The result of the measurements done at different workloads can be seen in table 6.1.

	CPU	Memory
Idle	3%	293%
Multiple-requests	70%	293%
Streaming	87%(max available)	293%

Table 6.1: Robo's performance during different workloads.

Table 6.1 shows that during idle times the CPU load is relative low around 3%, while it quickly rises when given work. During requests, the CPU stays around 70%. This is the measured CPU load while Robo receives *Ping* requests. Robo is only answering these requests with a *Pong* response message. Other requests may demand more CPU depending on the implementation. The application is most CPU intensive when streaming is turned on, where it takes all the available CPU resources it can get.

Table 6.1 shows that the memory usage stays on 293% at all times. The number is above 100% since top also takes into account swap memory. The Results shows that Robo request all available memory it can get during its runtime and never releases any of it. This also applies to other programs like the menu, which is also running as a leJOS application on the EV3.

Time spent on requests

As a step to find where the time went into each request, some measurements has been done in the Robo program. The different actions in Robo was measured during a Ping request. The result was then compared to the roundtrip time for the whole requests. It was discovered that 42% of the roundtrip time was spent on parsing and marshaling of the JSON messages. The time used to complete a Ping action on Robo was so small it could not be measured. The result can be seen in figure 6.1. The rest of the roundtrip time is lost in the communication latency and in the message receiver at the ev3-python library.

One can conclude from the results that JSON is a very inefficient message format to use on the less powerful EV3. A possible improvement could be to switch out the JSON message format for a less computational demanding format type, like a raw binary format.

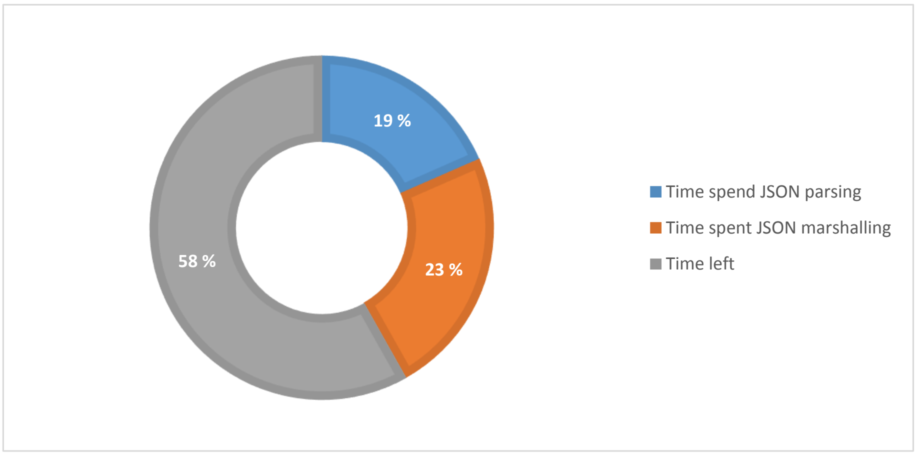


Figure 6.1: Distribution of time during a Ping request over USB. Shows that 19% of the time is spent on parsing the JSON request and 23% of the time is spent on marshalling the JSON response.

Streaming improvement

When measuring a request's roundtrip time during different workloads in Robo, some performance issues was discovered. During streaming of sensor samples, the roundtrip time of the Ping requests went up with much more than expected. When looking into it, it was discovered that the streaming service had added a minor sleep function as a step to lessen the performance needed while streaming was running. When removing the sleep function the roundtrip time improved. The performance cost with the sleep function is in other words greater than without it.

It appears that a minor sleep functions that only sleeps for a couple of milliseconds adds more performance costs that are gained. The roundtrip times with the old and new solution can be seen in figure 6.2.

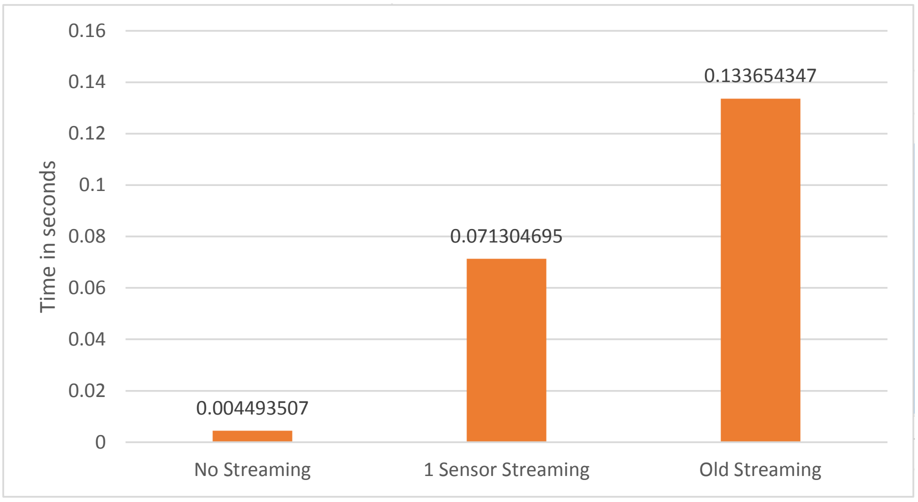


Figure 6.2: Shows a request's improved roundtrip time. First column shows the roundtrip time of a request while Robo is not streaming. Second column shows the roundtrip time while Robo is streaming and the sleep function is not there. Third column shows the roundtrip while Robo is streaming and the sleep function is still there. Both streaming tests only streams sensor samples from a single sensor. All communication goes through the USB channel in this test

6.1.3 ev3-python

CPU and Memory Usage

Measurements of the ev3-python library has been done with the tool psutil[1] and windows resource monitor[2]. The result of the measurement at different workloads can be seen in table 6.3. Two performance measurement tools are used since psutil only measure CPU use on one core, while windows resource monitor measure average used over all the cores.

	CPU psutil	CPU Reosurce Monitor	Memory psutil
Multiple-requests	4%	1%	0.071%
Streaming	6%	1%	0.063%

Table 6.2: The ev3-python library's performance during different workloads.

Table 6.3 shows CPU use during different workloads with the ev3-python library. During multiple ping requests to Robo, the CPU use is relative low. Psutil reports CPU usage of 4% while windows resource monitor reports CPU usage of 1%. When the ev3-python library has more intense workloads like when it is constantly receiving streaming messages of sensor samples, psutil report 6% CPU usage while the windows resource manager reports an overall 1%.

Memory usage is low in all cases, under 1% independent of workloads.

Latency

Figure 6.3 shows the roundtrip latency between the ev3-python library and Robo under different workloads on Robo. Measurements is done by sending 1000 Ping request to Robo and calculating an average across these requests. The results shows that during no streaming the latency between Robo and the ev3-python library is on average 0.004 seconds, which means a couple of hundred requests per second can be made. Of course, this is during an optimal request where there is minimal delay in executing it on the EV3.

[1] `https://code.google.com/p/psutil/`

[2] `http://www.7tutorials.com/how-use-resource-monitor-windows-7`

Streaming slows down the roundtrip time of requests. However, it seems the amount of sensors data streamed has no effect on the request's roundtrip time.

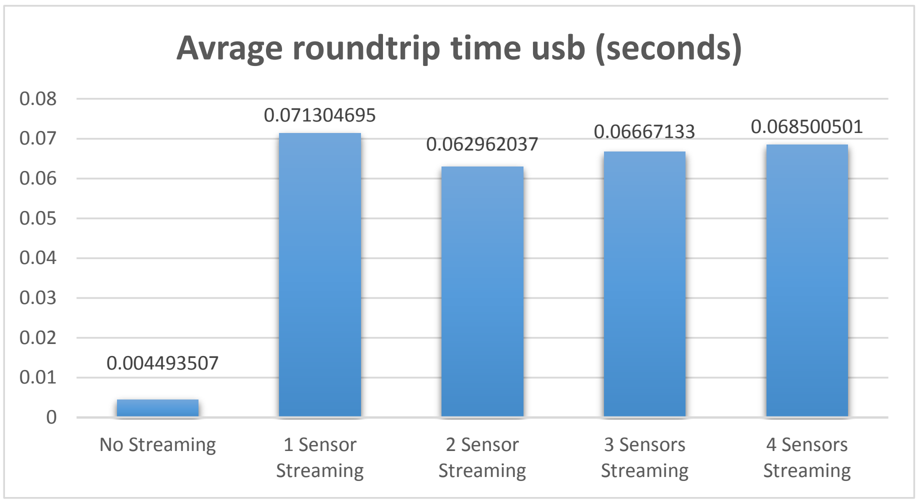


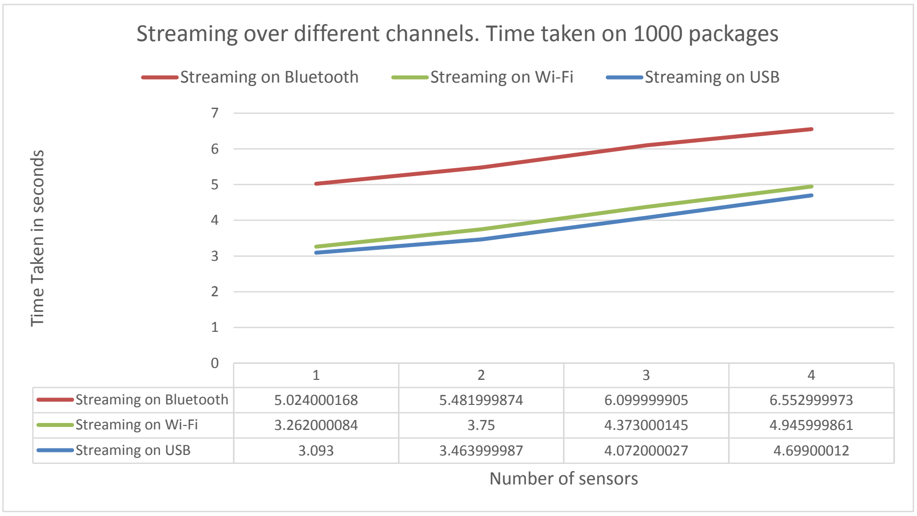
Figure 6.3: Average roundtrip time for requests during different workloads on Robo. Measurements done through the USB channel. Measured in seconds.

Figure 6.4 shows the time taken to receive 1000 packages on the sensor-sampling stream from Robo through the three different channels. The figure shows that more sensor opened leads to a longer time. The figure also shows that the quickest transfer times goes in the order: USB, Wi-Fi, and Bluetooth.

6.1.4 RoboMind Master Server

CPU and Memory Usage

Windows resource monitor and psutil was also used to measure CPU and memory usage on the RoboMind master server. The measured CPU and



	1	2	3	4
Streaming on Bluetooth	5.024000168	5.481999874	6.099999905	6.552999973
Streaming on Wi-Fi	3.262000084	3.75	4.373000145	4.945999861
Streaming on USB	3.093	3.463999987	4.072000027	4.69900012

Figure 6.4: Time taken before 1000 streaming packages is received on sensor samples from Robo through the different communication channels. Also shows the variations in time when 1-4 sensors is opened for streaming.

memory usage can be seen in table 6.3. The table shows the measured performance when there is only one client connected during different workloads. While the server is only pushing updated sensor data to the client, the CPU and memory usage is low.

The more code added to the system the more memory it takes, but it is still small. When the system runs behavior code, the load on one CPU core spikes to the max. This is logical since the behavior controller continuously loops through the running behaviors checking which behavior wants to run. It in other words act as a busy loop. This could be improved by allowing the system to sleep for a short duration after it has been confirmed which behavior should run. However, this would lead to a lesser degree of precision when to switch between active behaviors.

	CPU psutil	CPU Reosurce Monitor	Memory psutil
Streaming	7%	1%	0.098%
Running Code	100%	12%	0.099%

Table 6.3: Performance on the RoboMind master server during different workloads.

Flask

Performance tests of Flask can be seen in table 6.4. The tool apache benchmark[3] has been used to test the performance of the Flask server in a local network. The numbers shows that Flask still has a great performance even when not properly deployed on a production server. The numbers could be improved by properly deploying it.

Flask	
Transfer rate	293.62 kbytes/sec
Request pr second(mean)	1953.54
Time pr request(mean)	5.119 ms
Time pr request(concurrent)	0.512ms

Table 6.4: Flask performance. 10 000 requests, 10 connections concurrent, 2 bytes served for each request, a "OK" message.

6.1.5 Web Interface

The web interface has been tested in the Chrome browser[4] and measured with the tool chrome tool: Task Manager. The measured performance can be seen in table 6.5. The measured performance goes up when the web interface is connected directly to the RoboMind server through the WebSocket.

It has been rather difficult measuring the roundtrip latency from web interface to the RoboMind master server through the WebSocket, since both only pushes data to each other without expecting a response. It has not been possible to test the latency this within the limited period of this thesis.

[3]`http://httpd.apache.org/docs/2.2/programs/ab.html`
[4]`https://www.google.com/intl/en/chrome/browser`

	CPU	Memory
Idle, not connected	0%	32100KB
Connected Streaming	3%	35500KB
Running behaviors	3%	35500KB

Table 6.5: Performance of the web interface during different workloads.

6.2 Issues found

Program's Startup Time

The startup time for programs with the leJOS environment is quite slow on the EV3. Even a small and non-computational heavy programs demands at least 20 seconds of idle time before it starts running. Non-surprisingly, this has been discussed before in the leJOS community, where the old NXT leJOS environment had an almost instant load time. The developers of leJOS has responded in [1] that no real effort has been made to optimize program load so far. Previously leJOS used a custom-built JVM, with less functionality to be able to cope with the small performance of the NXT. However, on the EV3 a fully featured JVM and Java system is used, which demands additional cost for more functionality, slowing down the load time of programs.

The first couple of requests to Robo can also sometimes have a slower response time then they normally should have. It seems that it takes a couple of seconds after the Robo has started before everything needed by the program is loaded into memory. After the first couple of requests, everything works in the speed it normally should do during the runtime.

Bugs in leJOS

Since leJOS is still in early development, a couple of bugs has been found in the framework. The developers of the framework have updated some of the bugs. The author has fixed others. One such bug found was in the Bluetooth communication, where it did not allow the connection to be properly shutdown. This meant that no new Bluetooth connections could be established after the first one.

6.3 Improvements

6.3.1 Robo

As already discussed, an improvement that could be made in Robo to switch out the JSON message format for a less computational demanding format type, like a raw binary format.

6.3.2 RoboMind master server

The performance of the RoboMind master server could potentially be better if it was actually deployed on a production server. However, within the timeframe of this thesis this has not been possible.

Furthermore, the WebSocket implementation could be exchanged for another long polling mechanism to improve the number of supported browsers, since WebSockets are generally only supported by the latest versions of browsers.

6.3.3 Web interface

Unfortunately, there was no time to use RequireJS optimization where all the JavaScript files is bundled into a single optimized file. The roundtrip latency from the web server could be improved by doing so.

Chapter 7

Related Work

7.1 nxt-python

nxt-python[6] is a Python library for the Lego Mindstorms NXT robots. It lets users use the programming language Python for programmable control of the older generation of Mindstorms, the NXT. It provides an extensive API over the available features on the NXT, together with support for third party sensors and motors.

The library works by implementing Lego's command API, the *Bluetooth Development Kit*[1] in order to control an NXT. In other words, programs using the library is not executed directly on the NXT, but controls it from an external machine. It does this by sending the different binary commands specified in the Bluetooth development kit in order to control an NXTs actions. The library connects to available NXTs through either Bluetooth or USB, but it depends on the user having the proper drivers and libraries installed. Lego has yet to publicly release such a *development kit* for the EV3s.

nxt-python share many similarities to the ev3-python library made in this thesis. Both of them uses a connect-control scheme in its architecture, where there is no direct programming on the device, only a transfer of commands. The use of the libraries also shares similarities. Both uses a brick object to represent a connected device. Other external components (e.g. sensors and motors) transfer their commands by uses the brick object as the communi-

[1]`www.lego.com/en-gb/mindstorms/downloads/nxt/nxt-bdk` (Last accessed 25-May-2014)

cation point.

This is where the similarities stops as the libraries works in entirely different ways. ev3-python works by controlling the program Robo on the EV3 by sending JSON commands. nxt-python works by using the Bluetooth development API to control the firmware on the NXT with binary commands, without running a program in it. nxt-python also doesn't offer any streaming support like the ev3-python library provides.

Unfortunately, the nxt-python library cannot be used to directly connect and control the EV3s. It is however possible to use the nxt-python library in RoboMind as an addition or replacement to the ev3-python library. This would allow RoboMind to use NXTs in the system instead or as an addition to the EV3s.

7.2 Programming on the device

There exists multiple programming environments for the Lego Mindstorm EV3 device. Some of these includes: ROBOTC (for C users), MonoBrick (for .NET users), leJOS (for Java users) and the default-programming environment that is included in every Lego Mindstorm kit. Unfortunately, all of these offers a static programming environment, meaning they must be compiled and uploaded before they can be run on a Lego Mindstorm device. None of these programming environments offers a dynamic setting where code can be added to an already running program.

As for dynamic programming environments for the EV3 there dosn't exists many. One found was the the python-ev3 library[2]. Back in the beginning of this thesis, python-ev3 were far from complete and littered with bugs. It was also built with the help of an outdated, unstable version of leJOS. It was unusable as a direct programming environment for the EV3. The hope is that more dynamic programming environment, as the python-ev3 will be fully completed in the future. It would replace the need of programs such as Robo to offer on-the-fly commanding of EV3s, with real dynamic on-the-fly programming on the actual device. However, programs such as Robo do provide a platform where the computational need is removed from the less powerful EV3 to a more capable computer. Which can be better depending on the application.

[2]`www.github.com/topikachu/python-ev3` (Last accessed 22-May-2014)

In this thesis, leJOS is used as the programming environment to make the controllable server program Robo, but any of the other programming environments listed could be used instead. leJOS was chosen as it seemed the more stable and feature rich framework at the start of this thesis.

7.3 The Player/Stage Project

The Player/Stage project[11] describes two development software tools usable in a multitude of different robotic platforms. The project presents the tools *Player*, a robot device server and *Stage*, a simulator for development of robot programs.

Player is the software tool provided for running on robots. Player is a server program that provides external application with direct access to the robot's sensors and actuator through its interface. External application connects to Player over a TCP socket and can configure the device on-the-fly. Player is runnable on a magnitude of different robotic platforms and can be configured to support new hardware platforms as the need arises.

Robo uses a similar design to Player. Both programs are robot device servers and both provide language independent interface accesable through the TCP network protocol, where Robo also allows for Bluetooth. Differences can be found in how data is transferred between external applications and the programs. Player treats sensors and actuators as files, where clients must open them with the proper access to either read or write to them. Robo does not do anything similar and only provides a useable interface to the components. Furthermore, Player allows multiple clients to be connected at the same time where new commands may overwrite each other. The developers argue that if multiple clients are connected and commands overlap it was probably the developer's intention and solving it would be more of hindrance then a solution. Robo only allows one connected client at all times to solve these problems and make it a more fool proof application.

Player could in theory be used instead of Robo in this thesis, but it would need to be largely reconfigured to work on the Lego Mindstorm platform. A daunting task within the timeframe of this thesis. Furthermore, Robo offers some streaming services towards its clients for easier access to continuous information. No such services is mentioned from Player.

Stage is a simulator application that provides a development platform for robot controllers. Stage implements the Player interface to allow users to first develop the controllers virtually before using them on real world robots running Player. It offers a full visualization of a 2D virtual world where robot controllers can be thoroughly tested before deployed in the real world.

RoboMind has never been developed with the intention of simulating running robots. It is made for real world robot usage, where programs can be developed on-the-fly and be updated while the robots runs. Stage is an application where development is done in simulations, where code is thoroughly tested and compiled before run in the real world. Stage doesn't offer any live updating of already running code. Furthermore, there is no guarantee from the developers of Stage that robots will works the same way they did in the simulations.

7.4 REAL

REAL (Remotely Accessible Laboratory)[13] is a system that provides users with remote access to a programmable robot. The objective behind REAL is to remove users need for expensive equipment to test their robotic code and algorithms, by providing remote access to a free robotic platform.

Access to REAL is granted through a user's web browser. Only one user is allowed control over the robot at a time, so users must first go through an access page. When a user is granted control of the robot a magnitude of possibilities opens up. The robot can now be directly controlled either in a point and click fashion, or the user can supply their own runnable code to the robot. Togheter with the control interface two live video streams is provided over the running robot.

Code written must follow the manufacturer C specific interface to work on the robot used in REAL. Code must be written on the users own device before transferring it as files to the server at REAL. REAL compiles the code and allows users to specify when to start executing it on the robot. After the execution the users has the option of downloading the log files from robot, where information such as program flow and sampled sensor values can be inspected.

REAL shows how the addition of web cams to a system may aid students and researchers with access to expensive equipment they could not afford themselves. The same idea can be applied to RoboMind to provide the same access. REAL also shows how direct programming on a robot may be offered through a web interface, where code is shipped and compiled at the server before used on the robot. It does not however, provide any real time access to sensor information and program flow. It does neither provide any tweaking in already running programs. Programs must be entirely stopped before recompiled and restarted.

Chapter 8

Conclusion and Future Work

This chapter concludes the work of this thesis and presents the contributions, concluding remarks and future work.

8.1 Contributions

This thesis describes the architecture, design, implementation and evaluation of RoboMind, a platform for on-the-fly programming and inspection of behavior-based robot programs. Through RoboMind, users can add or edit existing behavior modules on-the-fly without disrupting what the robot was previously doing. The interface allows users to inspect which behaviors are running in addition to the collected sensor samples from the robot's run-time.

RoboMind has been thoroughly tested and developed for usage with the latest generation of the Lego Mindstorms, the EV3. The tests confirm that RoboMind works as intended. RoboMind offers a fully featured Python-programming environment for run-time modification of running EV3 programs. This is provided with the help of the ev3-python library, which controls the EV3s through the control program Robo, both is implemented by the author.

8.2 Concluding remarks

In this thesis, a lot of effort went into figuring out the quirks of leJOS. leJOS allows for a wide variety of different actions through its extensive API, but using it for development can be a cumbersome process. Programming is done on an external computer, where a program must first be compiled before it can be sent over to the EV3. This takes time, even with the proper scripts it requires at least two seconds of idle time before it was ready to run on the EV3. Including external jar libraries was even harder, where many not so obvious configurations must be made. The programs also have a slow starting time, where even small and non-computational heavy programs demands at least 20 seconds of idle time before it actually starts running. Furthermore, in the leJOS environment there exists bugs. It is not a stable environment, which means some is spent on fixing these bugs to be able to use all the available features in the API.

8.3 Future Work

The prototype works as designed, but there is always room for improvements. The following features was could further improve the system and was not implemented because of the limited timeframe of this thesis:

- More can be done on Robo's API to allow for more functionality. Potentially redesign the control interface to use a less computational demanding communication format.
- A more descriptive and feature rich web interface. There are a lot of room for improvements here as only some basic features are in place. The web interface should be more intuitive for new users, so it can automatically be taken in use without explaining all the features. Furthermore, more features in the web interface could be implemented. Like persistent saving of code and settings and a better editor for behavior programming.

References

[1] Forum discussion on the alpha-0.6.0 release. `http://www.lejos.org/forum/viewtopic.php?f=18&t=5822&start=15`. [Online; accessed 22-Feb-2014].

[2] Lego mindstorms homepage. `http://www.lego.com/en-us/mindstorms/?domainredir=mindstorms.lego.com`. [Online; accessed 08-May-2014].

[3] The lejos forum page. `http://www.lejos.org/forum`. [Online; accessed 08-May-2014].

[4] The lejos homepage. `http://www.lejos.org/`. [Online; accessed 08-May-2014].

[5] Summary of mindstorms history. `http://www.lego.com/en-us/mindstorms/gettingstarted/historypage/`. [Online; accessed 07-May-2014].

[6] nxt-python homepage. `https://code.google.com/p/nxt-python/`, 2012. [Online; accessed 22-May-2014].

[7] R.C. Arkin. *Behavior-based Robotics*. Bradford book. MIT Press, 1998.

[8] R.A. Brooks. A robust layered control system for a mobile robot. *Robotics and Automation, IEEE Journal of*, 2(1):14–23, Mar 1986.

[9] Jay Bryant and Mike Jones. Responsive web design. In *Pro HTML5 Performance*, pages 37–49. Apress, 2012.

[10] Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. *ACM Trans. Internet Technol.*, 2(2):115–150, May 2002.

[11] B. Gerkey, R. Vaughan, and A. Howard. The player/stage project: Tools for multi-robot and distributed sensor systems. In *11th International Conference on Advanced Robotics (ICAR 2003)*, Coimbra, Portugal, June 2003.

[12] L.M. Grabowski and P. Brazier. Robots, recruitment, and retention: Broadening participation through cs0. In *Frontiers in Education Conference (FIE), 2011*, pages F4H–1–F4H–5, Oct 2011.

[13] E. Guimaraes, A. Maffeis, J. Pereira, B. Russo, E. Cardozo, M. Bergerman, and M.F. Magalhaes. Real: a virtual laboratory for mobile robot experiments. *Education, IEEE Transactions on*, 46(1):37–42, Feb 2003.

[14] O. Hallaraker and Giovanni Vigna. Detecting malicious javascript code in mozilla. In *Engineering of Complex Computer Systems, 2005. ICECCS 2005. Proceedings. 10th IEEE International Conference on*, pages 85–94, June 2005.

[15] T. Karp, R. Gale, L.A. Lowe, V. Medina, and E. Beutlich. Generation nxt: Building young engineers with legos. *Education, IEEE Transactions on*, 53(1):80–87, Feb 2010.

[16] Jennifer S. Kay. Robots as recruitment tools in computer science: The new frontier or simply bait and switch? 2010.

[17] Tom Lauwers, Emily Hamner, and Illah Nourbakhsh. A strategy for collaborative outreach: Lessons from the csbots project. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, SIGCSE '10, pages 315–319, New York, NY, USA, 2010. ACM.

[18] Haroon Shakirat Oluwatosin. Client-server model. *IOSR Journal of Computer Engineering (IOSR-JCE).*

[19] Frank Greco Peter Lubbers and Kaazing Corporation. Html5 web sockets: A quantum leap in scalability for the web. http://www.websocket.org/quantum.html. [Online; accessed 28-May-2014].

[20] Alessandro Saffiotti. Fuzzy logic in autonomous robotics: behavior coordination. In *Sixth IEEE Intl. Conference on Fuzzy Systems (FuzzIEEE'97*, pages 573–578, 1997.

[21] Xander Soldaat. Comparing the nxt and ev3 bricks. `http://botbench.com/blog/2013/01/08/`

comparing-the-nxt-and-ev3-bricks/. [Online; accessed 07-May-2014].

[22] Alexander Svendsen. Robomind: A youtube movie. http://youtu.be/76abq0vTG5A. [Online; accessed 31-May-2014].

[23] Laurens Valk. Ev3 and nxt: Difference and compatibility. http://robotsquare.com/2013/07/16/ev3-nxt-compatibility/. [Online; accessed 07-May-2014].

[24] M. Wilson and B. Dupuis. In *From Bricks to Brains: The Embodied Cognitive Science of LEGO Robots*, AU Press Series, pages 199–221. AU Press, 2010.

Appendix A

Behavior Example

The next sections shows how the behaviors used as the example in chapter 3 works. The first behavior has the biggest priority and always gets to run if it wants to. It only wants to run when the ultrasonic sensor in the front of the robot has discovered something in close proximity (less than 0.25 cm). The second behavior always wants to run if it got the chance. The Lego robot used for this example can be seen in figure A.1

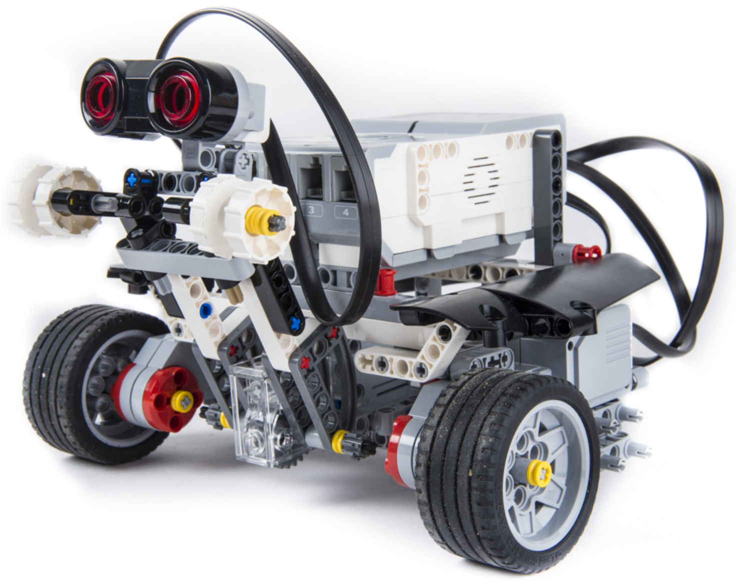


Figure A.1: Picture of the robot wanderer *used in this behavior example*

A.1 Behavior: AvoidColliding

This behavior wants control when the robot is near colliding with something in front of it. When the behavior has control, it starts turning 90 degrees to the left. It then releases control so other behaviors can start running. However, if the robot is again close to colliding with something this behavior will again regain control. The behavior module can be seen in figure A.1.

Listing A.1: AvoidColliding Behavior Code

```
class AvoidColliding(Behavior):
    def __init__(self):
        self._brick = ev3.connect_to_brick('10.0.1.1')
        self.left_motor = ev3.Motor(self._brick, ev3.
           MOTOR_PORTS.PORT_D)
        self.right_motor = ev3.Motor(self._brick, ev3.
           MOTOR_PORTS.PORT_A)
        self.ultrasonic = ev3.EV3UltrasonicSensor(self.
           _brick, ev3.SENSOR_PORTS.PORT_1).
           get_distance_mode()

    def check(self):
        distance = self.ultrasonic.fetch_sample()[0]
        if distance < 0.25 and distance != -1:
            return True
        return False

    def action(self):
        self.left_motor.rotate(600, immediate_return=True)
        self.right_motor.rotate(-600)

    def suppress(self):
        pass
```

A.2 Behavior: DriveAround

This behavior simply drives forward all the time. It wants to run all the time, but has less priority then the AvoidColliding behavior. Meaning it always release control when AvoidColliding behavior wants to take over. Before releasing control, it will always set the robot back to a stable state. Meaning it will stop the motors before handing over control. The behavior module can be seen in figure A.2.

Listing A.2: DriveAround Behavior Code

```
class DriveAround(Behavior):
    def __init__(self):
        self.brick = ev3.connect_to_brick('10.0.1.1')
        self.left_motor = ev3.Motor(self.brick, ev3.
           MOTOR_PORTS.PORT_D)
        self.right_motor = ev3.Motor(self.brick, ev3.
           MOTOR_PORTS.PORT_A)
        self._running = True

    def check(self):
        return True

    def action(self):
        self._running = True
        self.right_motor.forward()
        self.left_motor.forward()
        while self._running:
            pass
        self.right_motor.stop(immediate=True)
        self.left_motor.stop()

    def suppress(self):
        self._running = False
```