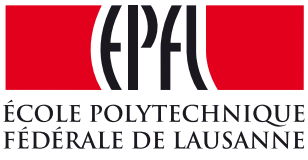
Design of an intuitive and responsive remote control interface for robots

The Universal Robot Controller

Version 0.2.0 15.01.2016



ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE
Master Thesis Project by La Spada Luca

Ijspeert Auke Jan, Master Project supervisor
Crespi Alessandro, Assistant
Estier Thomas, External expert
Lausanne, EPFL, 2015-2016

Abstract

Nowadays, connected devices and particularly smartphones and tablets are everywhere. Why not use them to control robots, instead of developing a new controller for each robot or reuse a controller that is not adapted for your brand new robot? To fulfill this idea, we have created an Android application that is flexible enough to be compatible with various types of robots. More precisely, it can connect to robots using TCP/IP, SSH, XHR, or WebSocket, and can easily communicate with robots that use ROS. Furthermore, the complexity of GUIs are separated in three parts: interfaces which describe the layout, widgets which are graphic components (e.g. battery, joystick), and drivers which make the link between interfaces, widgets and the robot. Moreover, drivers for robots can be stored online, and be downloaded easily in your mobile device, as well as interfaces. The drivers, widgets, and interfaces are created using web technologies (HTML5, CSS, JavaScript, SVG), which offer a lot of freedom and flexibility. The application has been evaluated on three different robots (ROVéo Mini, AmphiBot III, Absolem). To illustrate the potential of the proposed solution, we have created a GUI for each of robots that uses different kind of widgets such as a virtual joystick, a battery status, an attitude indicator.

Contents

1 Introduction

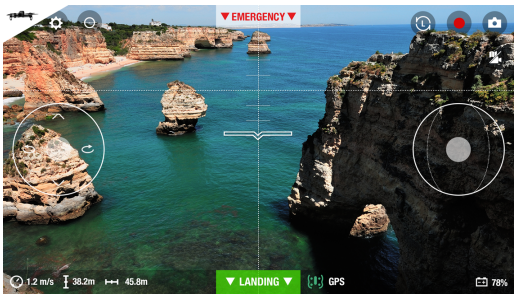
1.1 Motivation

In robotics, remote-controllers are quite important to enable direct piloting for all robots from non-autonomous robots to fully autonomous robots. However, during the last half-century, robots controllers have not evolved; they are always based on the same type of components: joysticks and buttons. Why not bring them a step further by taking in account the two following two ascertainment:

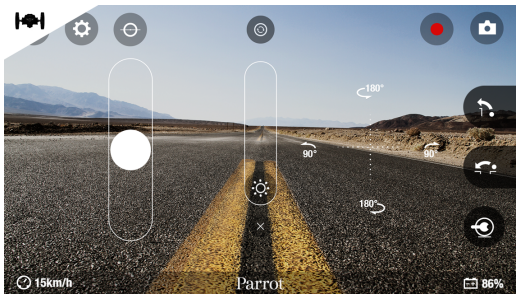
- Nowadays most of the people in developed countries possess a smartphone or a tablet (referred as mobile devices in this paper). These devices offer powerful and flexible development environments to produce high-quality applications in any domain (games, books, music, photography, social, etc.). Moreover, they are easily transportable.

- Each robotics laboratory tends to create a specific remote-controller for their prototypal robots. Consequently, it brings additional charges and complexity to the project.

Hence, why not developing a unique *versatile* remote-controller for robots that offer the possibility to create personalized graphical user interface (GUI) for each of their robots. The GUI will be used to remote-control robots, and should have access to sensors data and information of the robots. As an example, the Parrot [1] FreeFlight application can be used to control different ***Parrot*** robots (Figure 1.1).



(a) The interface of a flying robot (Source: [52]).



(b) The interface of a terrestrial robot (Source: [53]).

Figure 1.1 – Two different interfaces of the FreeFlight application of Parrot.

1.2 Goal

The project therefore aims at developing a *versatile* remote-controller application for robots and design some intuitive and responsive GUIs. The biggest feature of the application is to be able to get new GUIs for robots from an external server. In this context, the project is subdivided in three parts.

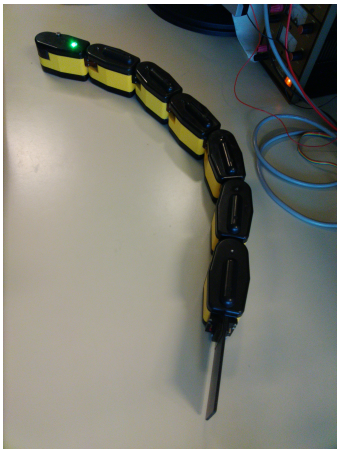
- Developing a mobile application to remote-control robots which name is Universal Robot Controller (URC).
- Developing a repository server to store plugins (e.g. personalized GUI) used by the URC. We refer to the server as Universal Robot Repository (URR). This part is mainly developed in the appendix B.4 and B.5, because it is a technical part.
- Designing intuitive and responsive GUIs for robots.

As proof of concept, the URC should be at least be able to handle 3 kinds of robots.

- The AmphiBot III by the Biorobotics Laboratory [2] of the Swiss Federal Institute of Technology in Lausanne (Figure 1.2c).
- The ROVéo Mini by Rovenso [3], a Swiss startup (Figure 1.2a).
- The Absolem by Bluebotics [4] which uses Robot Operating System [5] (ROS) and has been developed for the European project NIFTi [6] (Figure 1.2b).

(a) The ROVéo Mini

(b) The Absolem



(c) The AmphiBot III

Figure 1.2 – The robots used for the proof of concept.

2 Universal Robot Controller

In the next sections, we will first detail which technologies are used and why we chose them. Then, we describe the architecture of the project. Finally, we will detail the protocols that are used to communicate with our robots.

2.1 Technology Choices

Mobile devices can support 3 types of applications (Figure 2.1). Native applications which are written in the platform languages (e.g. Java for Android, Objective-C or Swift for iOS) and can use all the device hardware (e.g. camera, GPS). They are downloaded from the application shop. Web applications, which are not real applications but websites, are hosted on a web server. In this case, the access to device hardware is limited. Finally, we have hybrid applications which are an amalgamation of native and web applications. Concretely, it is a website embedded in a native applications, thus it has access to all device hardware and can be downloaded from the application shop.

Our choice fell on a hybrid applications because web technologies (HTML5, JavaScript, CSS, SVG, etc.) offers more possibilities over natives to create responsive and intuitive GUIs. Moreover, we cannot use web applications, because firstly, when we are connected to a robot, we do not necessarily have access to the Internet, and secondly, not all robots have an embedded web server to deliver webpages.

Hybrid applications can be easily deployed on various types of platform (e.g. Android, Windows Phone, iOS), but in order to simplify the development, we choose to concentrate our effort on Android, hence all subsequent Java code presented in this paper refers to Android Java code. The reason is that Android has 80% of the market share [8] (Figure 2.2). We also decided to do not use frameworks such as Cordova [9] and PhoneGap [10] which allows to create mobile applications compatible with the various mobile operating system, because using them increase the overall complexity of the project and reduce the control over the application. Nevertheless, the application should be easily convertible to these

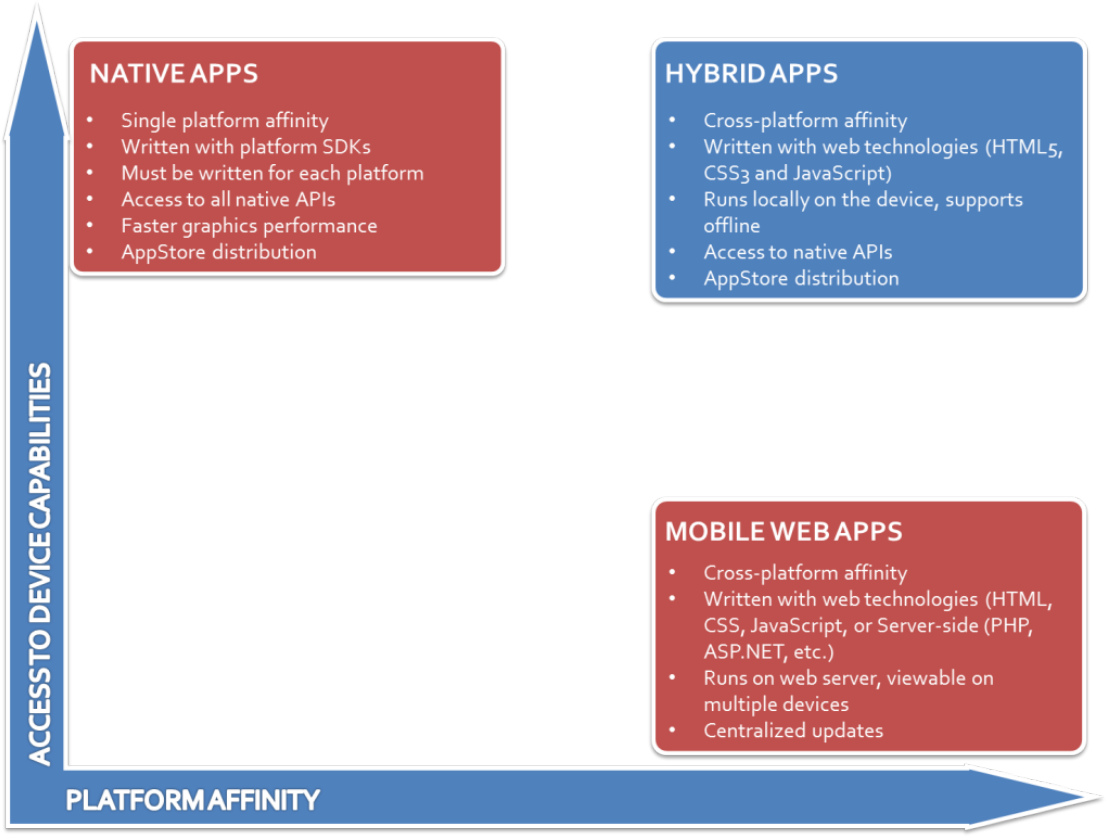


Figure 2.1 – Resume of the differences between native, web, and hybrid applications (Source: [7]).

technologies, or more generally to an iOS or a Windows Phone application.

2.2 Architecture

The architecture of the project has evolved as well from an inside view (the website) than an outside view (the whole application). In order to offer a better understanding of some choices that have been made, the next sections will describe how the project has grown.

2.2.1 Outside view

2.2.1.1 Web application ? Let me laugh !

The first idea, before even considering native or hybrid applications, was simply doing a website that communicates directly with robots using the default communication protocols of JavaScript: XMLHttpRequest [11] (XHR) and WebSocket [12] (Figure 2.3). However, this solution as two main drawbacks. First, as mentioned in the section 2.1, the website

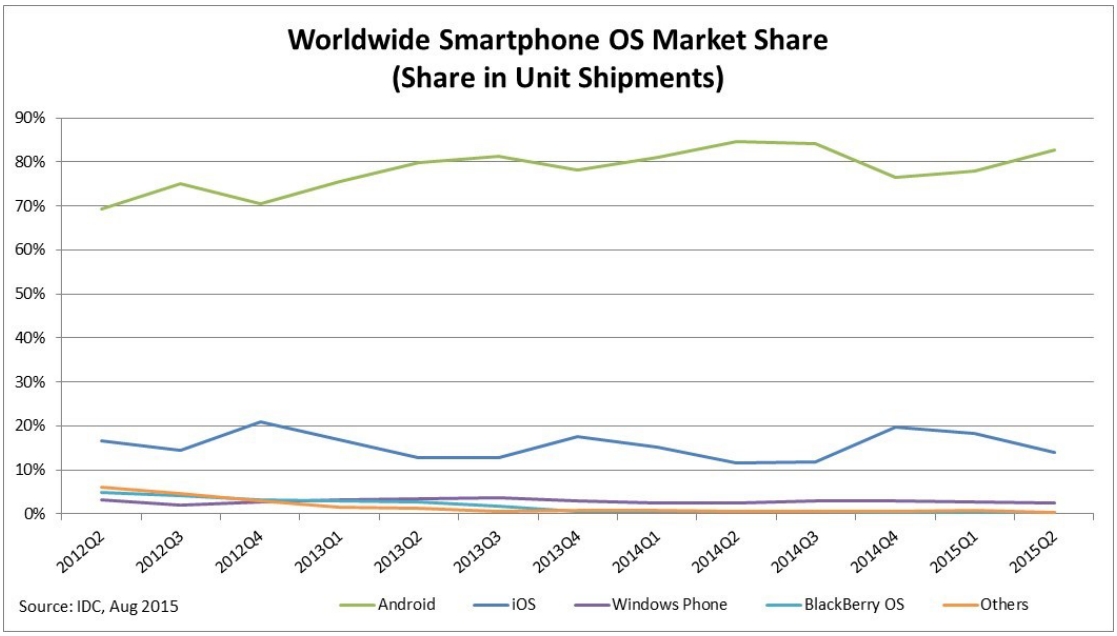


Figure 2.2 – Mobile OS market share from 2012 to 2015 (Source: [8]).

should be hosted somewhere. The three available possibilities are on the Internet, in the robot or on your Android device by adding a local web server. Unfortunately, we do not have necessarily access to the Internet when we are connected to the robot, robots do not incorporate perforce a web server and hacking your Android device to add a local web server is not user friendly. The second issue is that XHR and WebSocket are designed for communicating between websites and web servers, hence in our case their utilities are very limited, because not all robots offer XHR or WebSocket API.

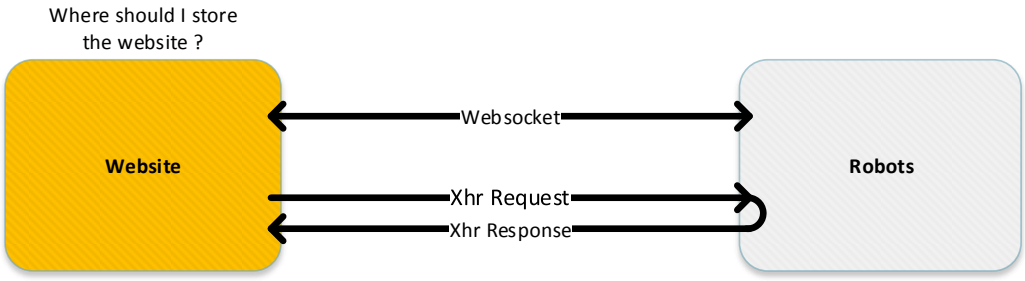


Figure 2.3 – The first architecture of the project where we can only do connection with robots using XHR and WebSocket protocols.

2.2.1.2 WebView ? The Solution !

As stated in section 2.1, to overcome the need to use a web server, we choose to embed our website in a native application that we called Universal Remote Controller (URC) (Figure 2.4). The component that permits to do that is called `WebView`. An implementation of the `WebView` component is available for each popular platform such as iOS [14], Android [13], Windows Phone [15]. Furthermore, this component also provides other useful features.

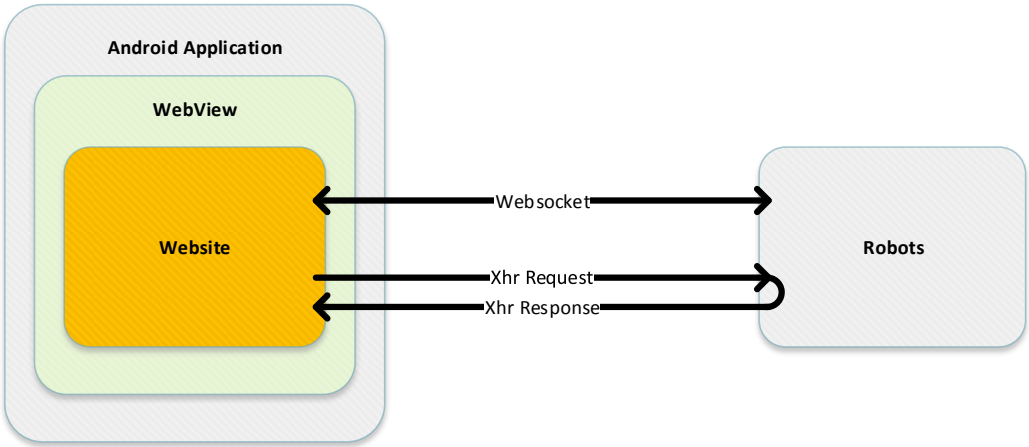


Figure 2.4 – The second architecture which uses Android `WebView` component.

2.2.1.3 @JavascriptInterface ? Bring me further !

Among others, `WebView` allows to inject Java objects into JavaScript context (Listing 2.1). In other words, we can call Java methods from JavaScript. Using this powerful feature, we can increase the connectivity of the URC by developing bridge classes between Java and JavaScript. For instance, we can use the Java Socket[1] [17] to add TCP and UDP connection, or JSch [16] for SSH (Figure 2.5).

[1] Android does not support DHCPv6 [63]

```
class JsObject {
    @JavascriptInterface // Annotation to say, here a function for
    ↪ JavaScript
    public String toString() { return "injectedObject"; }
}
webView.addJavascriptInterface(new JsObject(), "injectedObject");
webView.loadData("", "text/html", null);
webView.loadUrl("javascript:alert(injectedObject.toString())");
```

Listing 2.1 – An example of how we inject Java objects into JavaScript context. Moreover, this example demonstrates that it is also possible to call JavaScript method from Java using the `loadUrl` method. Unfortunately, this callback is very slow 500ms, hence for real-time processing, it should be avoided (Source: [55]).

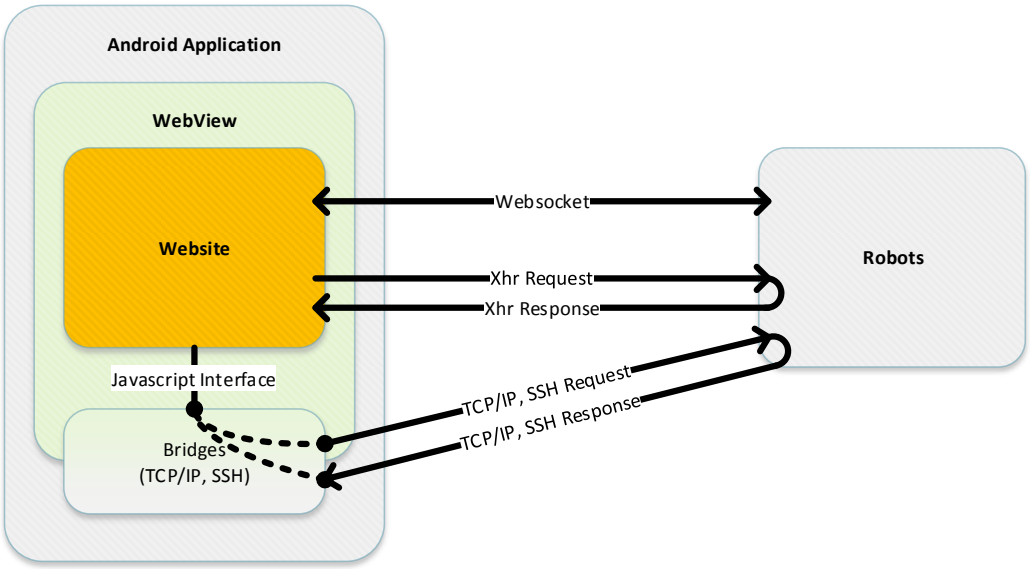


Figure 2.5 – The third architecture which adds additional network protocols: TCP/IP and SSH.

However, there are still a few limitations. First, we can only use primitive as types of arguments. Secondly, the returned value must be a primitive or `void`. These restrictions can be mitigated using some tricks. For instance, by returning stringified JSON [18], or by encoding array of bytes into Base64 [19].

2.2.2 Inside view - Core concepts

The main concept, that we want to enforce in the URC, is reusability. For example, the battery widget used in the GUI of *Mr. Foo* for the robot *Bar* is magnificent ? Take it and add it to your own GUI easily. To achieve this goal, we have to separate the complexity of our GUIs in several components: widgets, interfaces and drivers.

Widgets are graphic components such as batteries, joysticks, buttons, etc. A widget module is composed by at least a JavaScript file, but you can benefit from other web technologies such as HTML5, SVG or CSS.

Interfaces describe the layout of the GUI; how our widgets will be positioned on it. It is important to note that interfaces are layouts. They do not contain any information about widgets aside from identifiers used by drivers. An interface module is composed by at least one HTML file, but you can naturally use additional CSS files. Interfaces are not unique, you can develop several interfaces for the same robot.

Drivers are the core system of our GUIs, they make the link between widgets, interfaces, and the outside world (i.e. robots). For example, if you interact with a joystick widget, drivers will send movement to the robot. On the other hand, drivers can ask for information to the robot to stuff widgets. For instance, drivers make a request for the state of the battery and then set the response value in our battery widget. Drivers are also in charge of loading the widgets on the interface thanks to the identifiers. Notice that we have to develop one driver per robot, because each robot has its own protocol of communication, see the section 2.3 for more details.

More concisely, a GUI is composed by one interface, one driver, and from 1 to n widgets (Figure 2.6).

2.2.3 Inside view - Interactions

2.2.3.1 First sketch

The first structure of the website was really simple (Figure 2.6), a *menu* webpage which routes you to the *settings* webpage where you can set the IP and the port used by your robot, and the *GUI* webpage where it is initialized the connection with your robot and the GUI displayed. However, this implementation is too limiting; you cannot change GUI.

2.2.3.2 Switchable GUI

The ability to change GUI easily is an important feature, however if we can also download them from a repository server, it would add more flexibility to the URC. In order, to add these functionalities, the website has to be modified. First, by adding an external

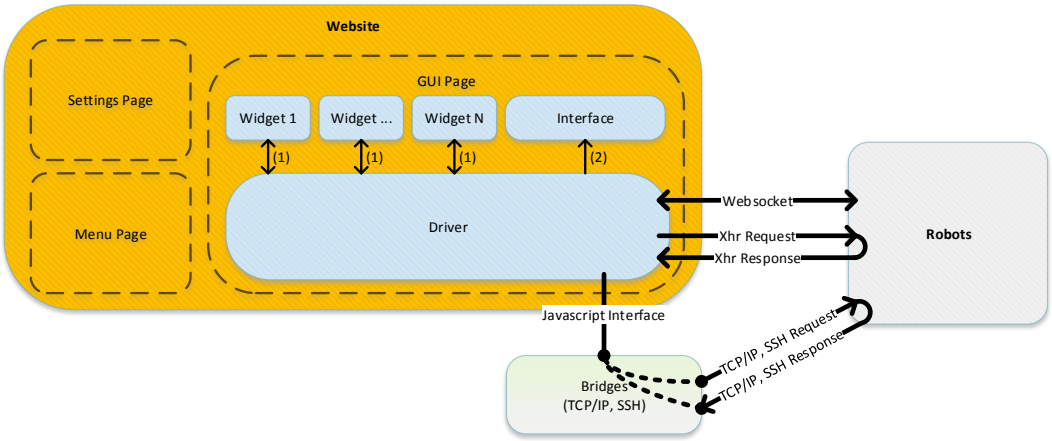


Figure 2.6 – The first architecture of the website where the driver is the brain of the GUI. It interacts with:

(1) Widgets by sending (e.g. battery state) or retrieving (e.g. joystick movement) data.

(2) Interface by injecting the design of widgets.

server (URR) to store our widgets, interfaces and drivers. Secondly, by extending our website with a *drivers manager* and a *interfaces manager* webpage. Together, they have the responsibility to download GUIs from the URR and store them on a local database (IndexedDB [20]). Finally, through the *settings* webpage, we can choose which GUI we want to be displayed when loading the *GUI* webpage (Figure 2.7).

2.2.3.3 Switchable settings webpage

Not implemented, but should be in future

Actually, there is still something that could be improved. The *settings* webpage permits to set only one IP and one port for robots. However, we can imagine that a GUI need more input settings than that. For instance, the robot and his camera have each a different IP. To overcome this problem, the URC should have a *settings* webpage adapted for each GUI (Figure 2.8).

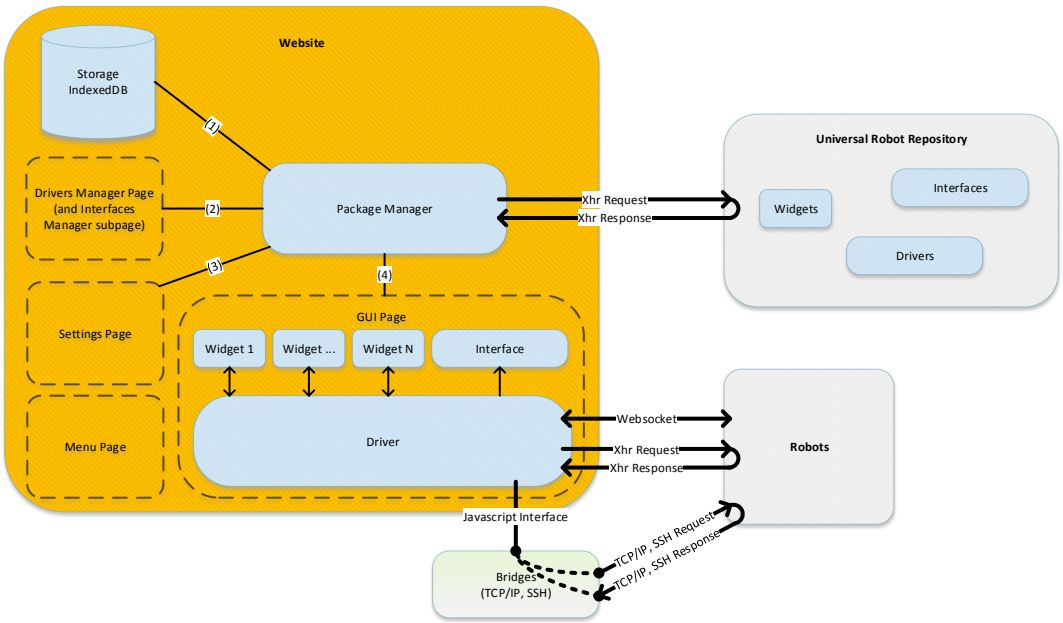


Figure 2.7 – The second architecture of the website which adds the changeable GUI feature. The package manager is the center of this dynamic system. It offers method to list, download, load, and remove GUIs from a local database **(1)**. This library is used by:

(2) *Drivers manager* page which allows to download, and remove drivers.

(3) *Settings* page to set up the robot configuration (driver, interface, IP, and port).

(4) *GUI* page which to load the GUI (driver, widgets, interface) using the settings you defined.

2.3 Robots communication protocols

2.3.1 AmphiBot III

The AmphiBot III communicates within the industrial, scientific and medical (ISM) radio bands 866-868 MHz, hence we cannot communicate directly with it through standard WiFi. The solution is to add a USB dongle to a computer which will do the bridge between the ISM radio and the network (Figure 2.9).

2.3.2 ROVéo Mini

The ROVéo Mini provides an access-point from which we can have access to the robot. Once connected to the WiFi of the ROVéo Mini, we have to do a SSH connection with

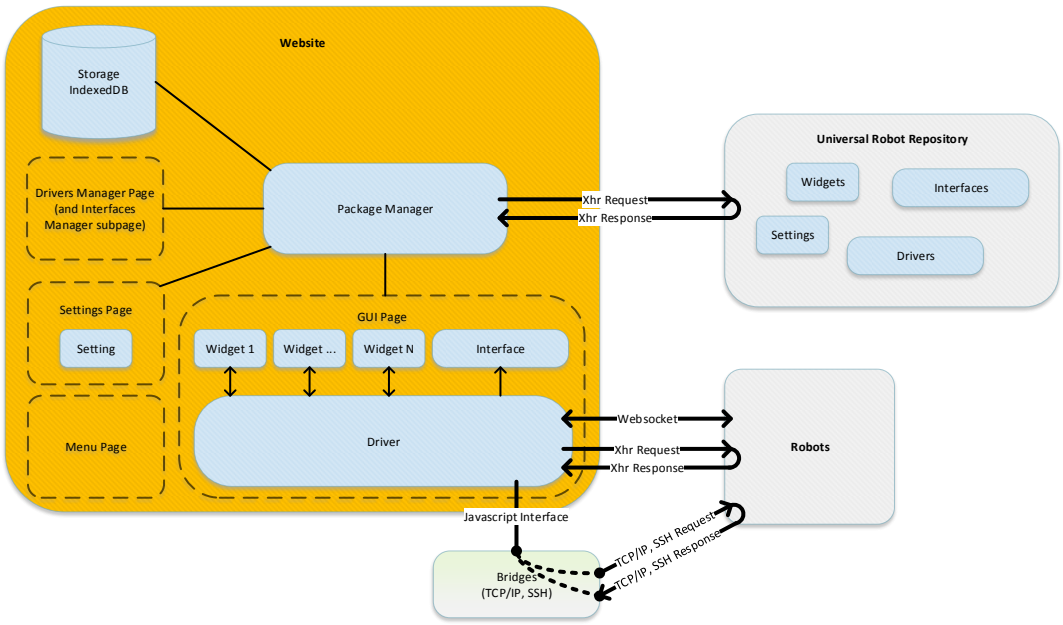


Figure 2.8 – The last iteration of the website with the feature of personalized *settings* webpage. This modification adds a new component called settings, which take care of modify a subpart of the *settings* webpage. For instance, if a robot needs to set several IPs, or needs some additional information.

the robot, and then we have to do a Telnet connection with the Arduino Yún [24] which is the brain of the ROVéo Mini. At this point, the commands can be sent through the Telnet connection (Figure 2.10).

2.3.3 Absolem

The Absolem is a robot that uses ROS, In order to communicate easily with this type of robots, the robot has to have installed the node Rosbridge Suite [21]. At this point, our controller has to connect to the access-point provided by the robot and uses the JavaScript library roslibjs [22] (Figure 2.11). Notice that it exists a Java version called jrosbridge [23].

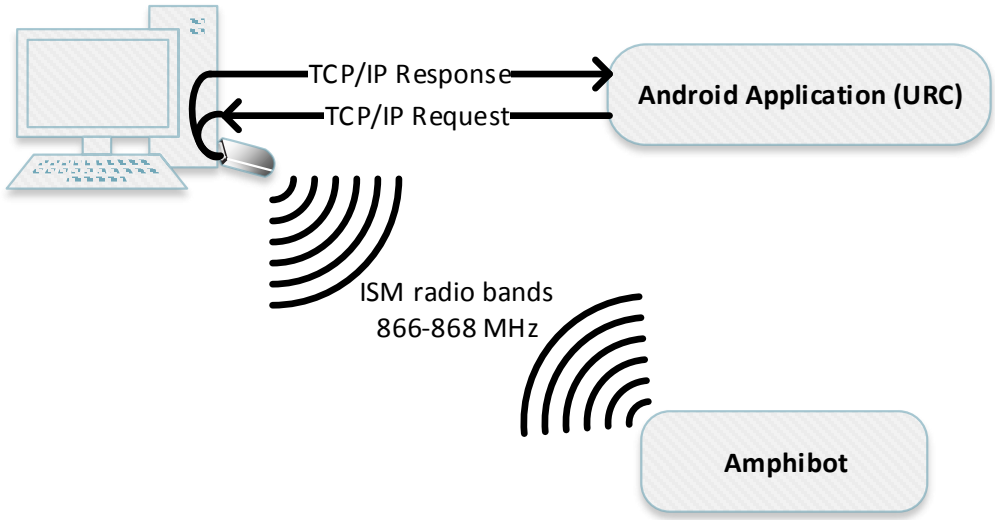


Figure 2.9 – Communication schema of the AmphiBot III with the URC.

Figure 2.10 – Communication schema of the ROVéo Mini with the URC.

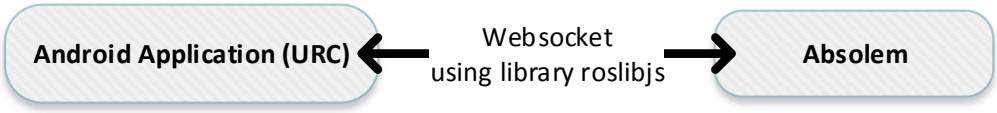


Figure 2.11 – Communication schema of the Absolem with the URC.

3 Graphical user interfaces & Controllers

GUIs and controllers used to be two distinct elements. GUIs are displayed on a screen, whereas controllers are something that we can hold and interact physically such as gamepads, joysticks, etc. However with the advent of touch screens, this border between GUIs and controllers is becoming thinner. For instance, if you play a game on your smartphone, the controller is incorporated in the screen through the GUI using a virtual joystick. In the following sections, we will see what is the state of the art of robotics enterprises in the field of interfaces and controllers for remote-control. Then we will see how we could represent some elements on the GUI such as GPS, batteries, range finders, etc.

3.1 State of the art

The subject is less extensive than what it seems. Most of the enterprises do the same kinds of things, thus the section will be separated by the type of hardware the enterprises use, which are mainly laptop control unit (LCU), mobile devices, and a gamepad with an external mobile device.

3.1.1 Laptop Control Unit

SenseFly [25] (drones company) and Qinetiq [26] (defense technology company) propose to control their robots with a LCU, which are basically one laptop coupled with a gamepad from the video game industry (Figure 3.1). Qinetiq does not show any GUI for their LCU, however SenseFly does (Figure 3.2). It proposes mainly: video feedback, ultrasonic feedback, satellite view, battery state, WiFi signal quality, and other data sensors such as temperatures. Most of their representations use commonly used ideas. The same that you can find in your smartphone or your computer. Their representation of ultrasonic feedback to know if they are some obstacles is quite interesting.

(a) SenseFly LCU (Source: [56]). (b) Qinetiq LCU (Source: [57]).

Figure 3.1 – Laptop Control Unit (LCU).

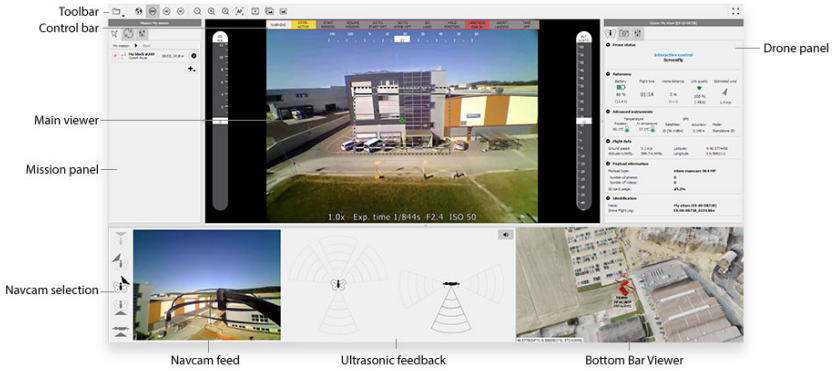
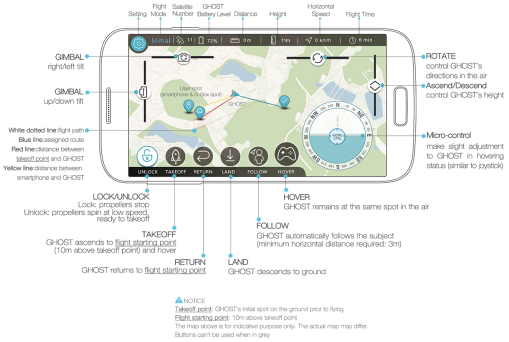


Figure 3.2 – Interface of SenseFly LCU (Source: [58]).

3.1.2 Mobile devices

Ehang [27] (drone company), Parrot [1] (unmanned aerial vehicle (UAV) and unmanned ground vehicle (UGV) company), and iRobot [29] (domestic and military robots company) have designed applications for mobile devices (Figure 3.3 and Figure 3.4). Parrot has the particularity to use the same application to control their unmanned vehicles whether they are UAVs or UGVs. Notice that all GUIs use some kind of virtual joysticks, either a circle that you can move in all directions or two joysticks that you can only move in one direction, one for the X axis and the other for the Y axis. The GUI of iRobot shows the state of the robot using a 3D model, whereas Parrot and Ehang offers several shortcuts to do manipulation such as landing, take off.

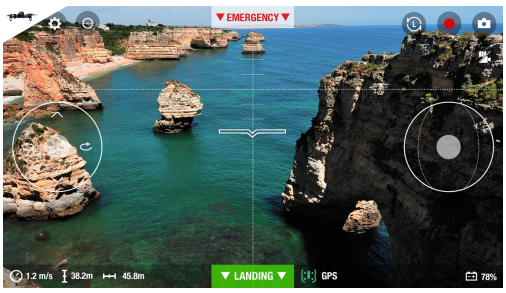


(a) Ehang interface (Source: [59]).

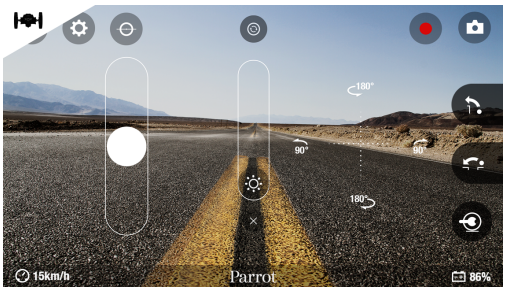
(b) iRobot interface (Source: [60]).

Figure 3.3 – The interface of the application used by Ehang and iRobot.

(a) Flight interface 1 (Source: [51]).



(b) Flight interface 2 (Source: [52]).



(c) Terrestrial interface (Source: [53]).

Figure 3.4 – Interfaces of Parrot application.

3.1.3 Gamepad with mobile device

3D Robotics [30] (drone company) and DJI [31] (drone company) manufacturers two similar controllers based on a gamepad and a facultative slot for a mobile device which enables the interface functionality (Figure 3.5). Their interfaces are quite similar, with a video stream and information about the drone such as speed, height, battery state, WiFi signal quality, GPS, etc. Moreover, their gamepad does not offer anything revolutionary. Finally, Parrot sell the Skycontroller [28] which is an extension for their mobile device application that provides more precision and controls for the Bebop Drone (Figure 3.6).

(a) DJI controller (Source: [61]).

(b) 3D Robotics controller (Source: [62]).

Figure 3.5 – Gamepad controllers with a slot for an external mobile device.

Figure 3.6 – Parrot Skycontroller (Source: [54]).

3.2 How can we represent...

GUIs of our state of the art section put forward several elements that could appear as widgets for the URC. In this section, we will discuss how these elements can be represented. Notice that these are mainly ideas, and a concrete implementation for the URC applications does not necessarily exist.

3.2.1 GPS

The GPS position of the robots can be displayed accurately using the unfriendly tuple (latitude, longitude) either in decimal or in sexagesimal. However, by displaying the position on a map the user has a better view of where is its robot (Figure 3.7).

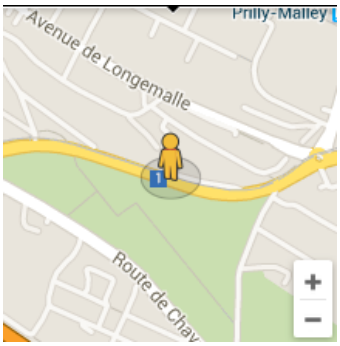


Figure 3.7 – GPS positioning on a map (Source: [32]).

3.2.2 Compass

The first idea to represent the compass like a common compasses (Figure 3.8), but there is a drawback. If we do not hold our Android device parallel to the ground, the representation loses its meaning. Another idea would be to encrust a 3D arrow that shows the direction of the North on the camera stream.

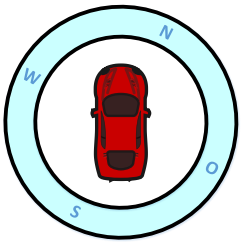


Figure 3.8 – Compass design, we could also rotate the car instead of the ring.

3.2.3 Movement

There are several ways to move robots: virtual joystick, an external peripheral, or using the Android device motion sensor (i.e. gyroscope, accelerometer, etc.), but it exists another way less widespread where you indicate vectors with your finger in the direction where the robot should move.

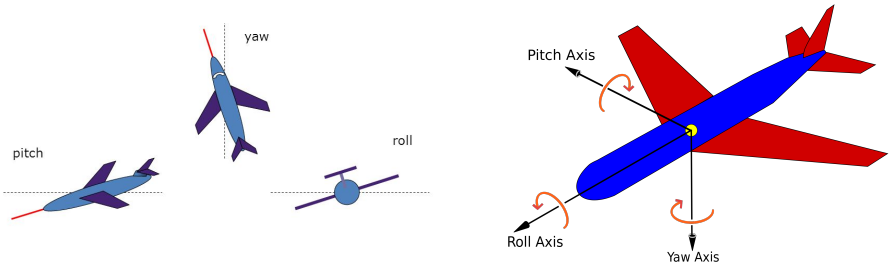
3.2.4 Camera

The video stream is generally displayed in the background of the interface. Some camera can stream 360 degrees video, thus in order such type of video, we should be able to move in its vision (for instance by using a virtual joystick or the Android device motion sensor).

It would be interesting to be able to switch between various types of cameras, such as thermographic, or scotopic vision if the robot supports them.

3.2.5 Roll, pitch, yaw

Roll, pitch and yaw could be interesting information for robots that work on an unpredictable terrain. To represent these axes, we could display the robot on a 3D environment with its axis X pointing to the North (Figure 3.9b). It is also possible to show the 3 axis independently by separating each axis (Figure 3.9a).



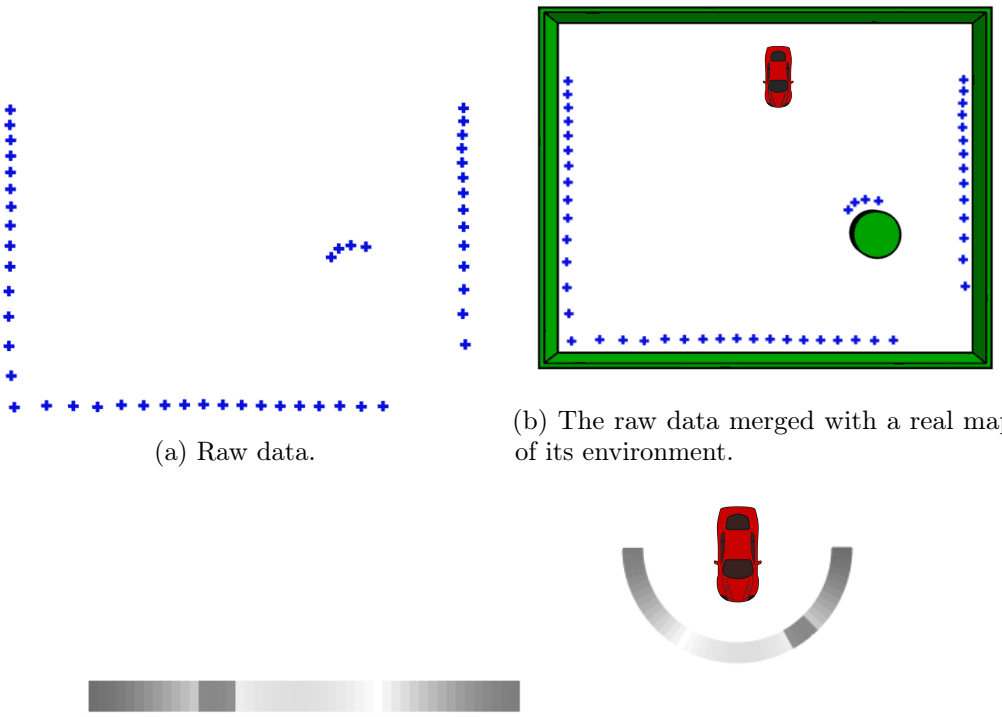
(a) Separated (Source: [33]). (b) Together (Source: [34]).

Figure 3.9 – Two ways of representing roll, pitch and yaw.

3.2.6 Range Finder

We could imagine representing the range finder data in several ways (examples are based on a laser range finder).

- Displaying the raw data in an a graph (Figure 3.10a).
- Displaying the interpolated raw data with a map of the robot environment (Figure 3.10b).
- Displaying the data like a depth map from the point of view of the robot (Figure 3.10c).
- Displaying the data like a depth map, but on an arc. It could be merge with the compass (Figure 3.10d).



(a) Raw data.

(b) The raw data merged with a real map of its environment.

(c) Depth map: black is near, white is far.

(d) Circular depth map: black is near, white is far.

Figure 3.10 – Laser range finder representation (Source: [35]).

3.2.7 Others

More common information, such as altitude, acceleration, speed, batteries, WiFi signal quality, always uses the same paradigm in every electronic device. Nevertheless, if we have something a little different, we could stray from the beaten path. For instance, the AmphiBot III has one battery for each unit of its corpse. Consequently, the widget could show each battery independently (Figure 3.11).

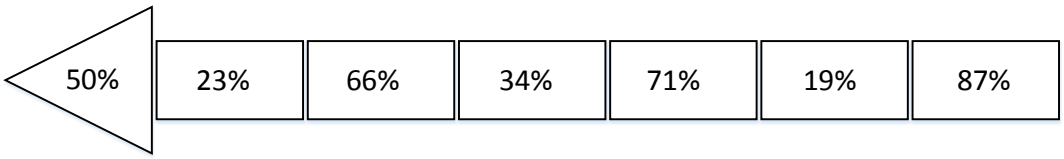


Figure 3.11 – A rudimentary widget for displaying the battery states of the AmphiBot III

4 Results

4.1 Benchmark

4.1.1 AmphiBot III

The default controller of the AmphiBot III is composed by 1 joystick, 4 buttons, and a screen (Figure 4.1). The buttons are mainly used for the configurations, whereas the screen displays the speed and the direction of the robot. Concretely, you control the AmphiBot III only with the joystick.

Figure 4.1 – The controller of the Biorobotics Laboratory for the AmphiBot III.

The GUI created for the AmphiBot III for the URC is rudimentary (Figure 4.2). It allows to send movement update to the robot using a virtual joystick, and display the speed and the direction. The interface is smooth and reactive, but we had to create a TCP bridge that is write-only (i.e. does not wait on the response) to speed up the time needed to update the registries of the robot. Moreover, we had set a limit number of updates

per second, otherwise the robot is flooded by requests, which creates a shift between the robot movement and the virtual joystick.

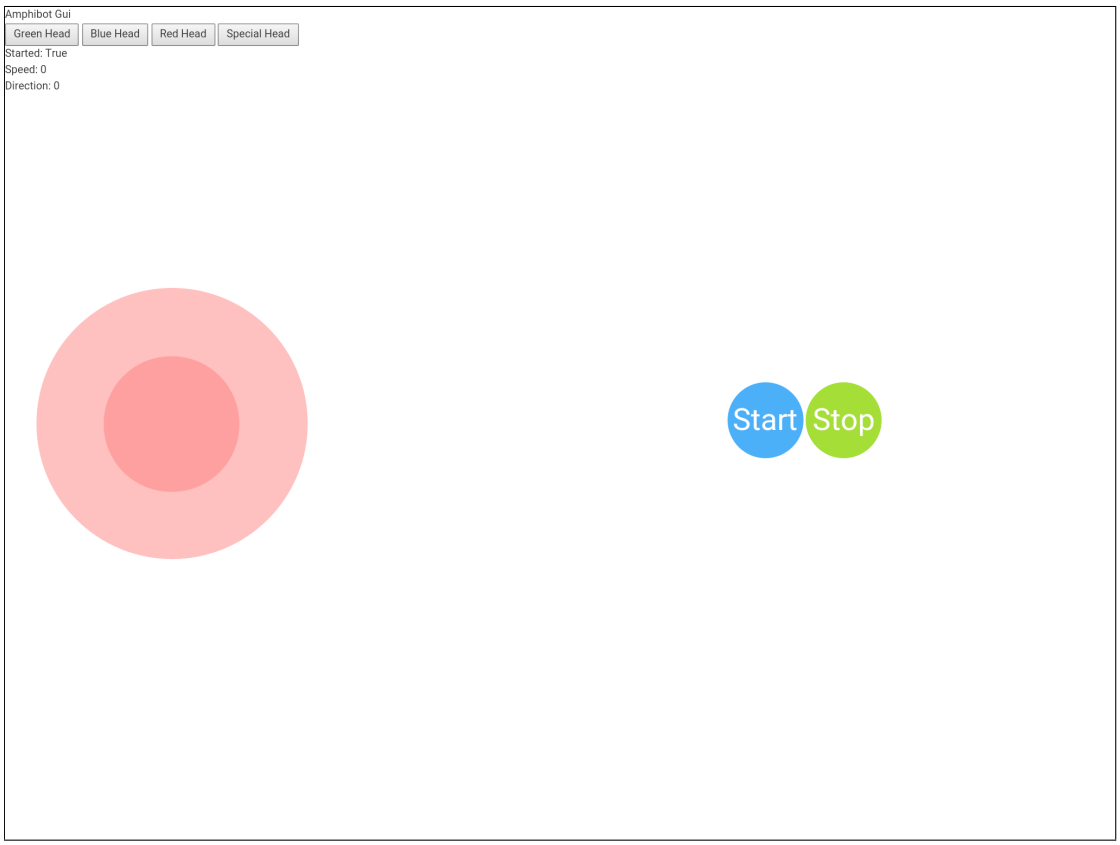


Figure 4.2 – The URC GUI created for the AmphiBot III.

The original controller has one big advantage over its URC counterpart: it can directly connect by radio to the robot. The Biorobotics Laboratory uses the same controller for AmphiBot II, Salamandra robotica I and Salamandra robotica II. Furthermore, these robots are also compatible with the same USB dongle used by AmphiBot III, thus we can easily develop drivers for them.

The data that the AmphiBot III provides are quite limited, but even so we can display the battery state and the position of the output axis of each module. We did not have the time to implement it, but it can be done.

4.1.2 ROVéo Mini

The ROVéo Mini is controlled using the Skycontroller, briefly presented in section 3.1.3. The components used to interact with the robot are: 1 joystick to sent the speed and the direction to the robot, 1 button to activate or deactivate the autostairs mode, and a screen that displays its video stream. Notice that the electronic of the ROVéo Mini is

made of two parts: an Arduino Yún card and the motherboard of a Parrot Bepop drone and its video camera. The Bepop provides the video stream and inertial measurement unit (IMU), which are only accessible using the Java API of the drone, whereas the Arduino is used to control the speed and the direction of the robot, and to retrieve data such as the battery state, and the front/rear angles of the bogy.

The GUI of ROVéo Mini created for the URC is quite similar to the AmphiBot III, but we retrieve in addition the battery state (Figure 4.3).

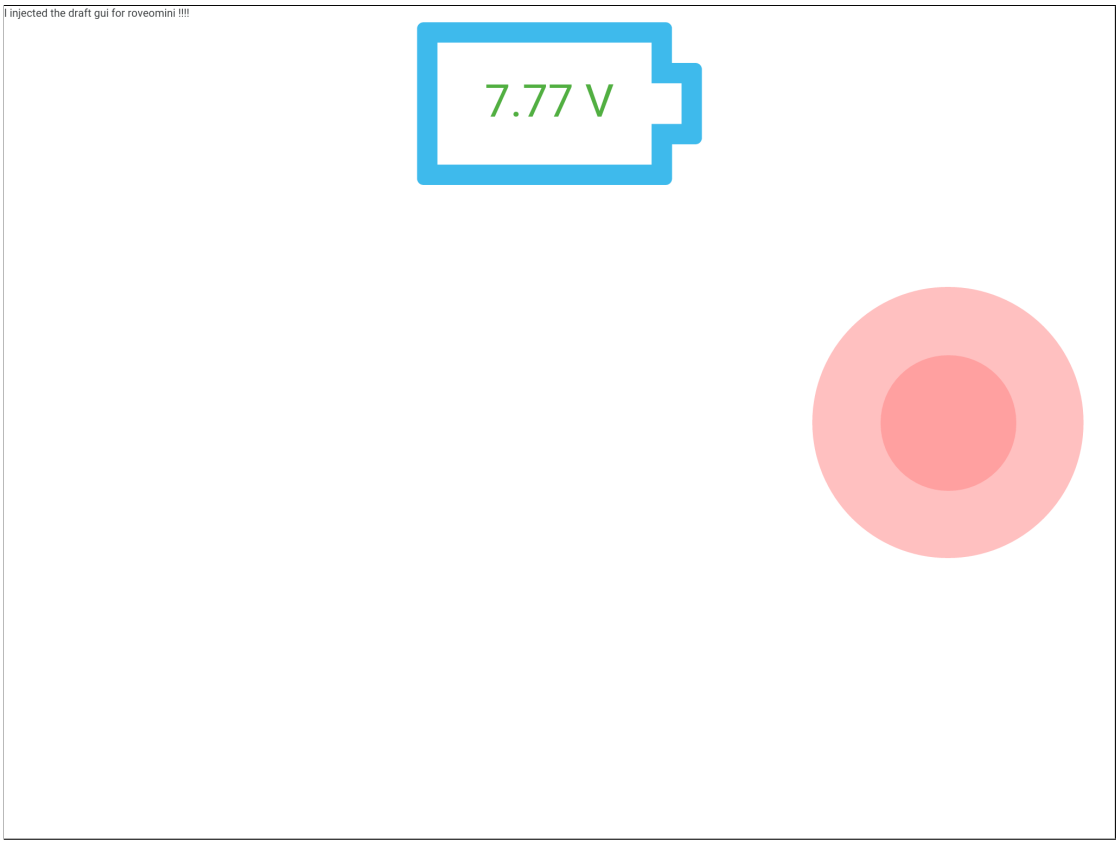


Figure 4.3 – The URC GUI created for the ROVéo Mini.

The original controller of the ROVéo Mini has one big advantage over the URC version: it can access to the Java API of the drone, and thus can retrieve the IMU and the video stream easily, whereas the URC version cannot, because the robot does not provide these data for external device. Moreover, the autostairs mode is not available on the URC version, because it uses the IMU values. Notwithstanding, using the Skycontroller is quite overkilled and it is not easy to carry with you compared to a simple mobile device.

The Rovenso team is currently working on making the IMU and the video stream accessible from an external device. Once it is done, adding the video stream and the autostairs mode to the GUI will be effortlessly.

4.1.3 Absolem

The controller of Absolem robot is a Logitech F710 gamepad (Figure 4.4).

Figure 4.4 – The controller of the Absolem.

The GUI prototype developed for the Absolem allows you to move it using a virtual joystick, to read its IMU, and to display images of its 360 degrees video camera (Figure 4.5). The Absolem does not provide any video stream, but instead he pushes still images one by one in order to emaluate a video stream. This method is not adapted for streaming in HTML5, because refreshing the image leads to poor performances.

It was planned to create a widget to display the laser range finder, and another to *move* in the 360 degrees video camera using the device motion sensor, but the time was lacking.

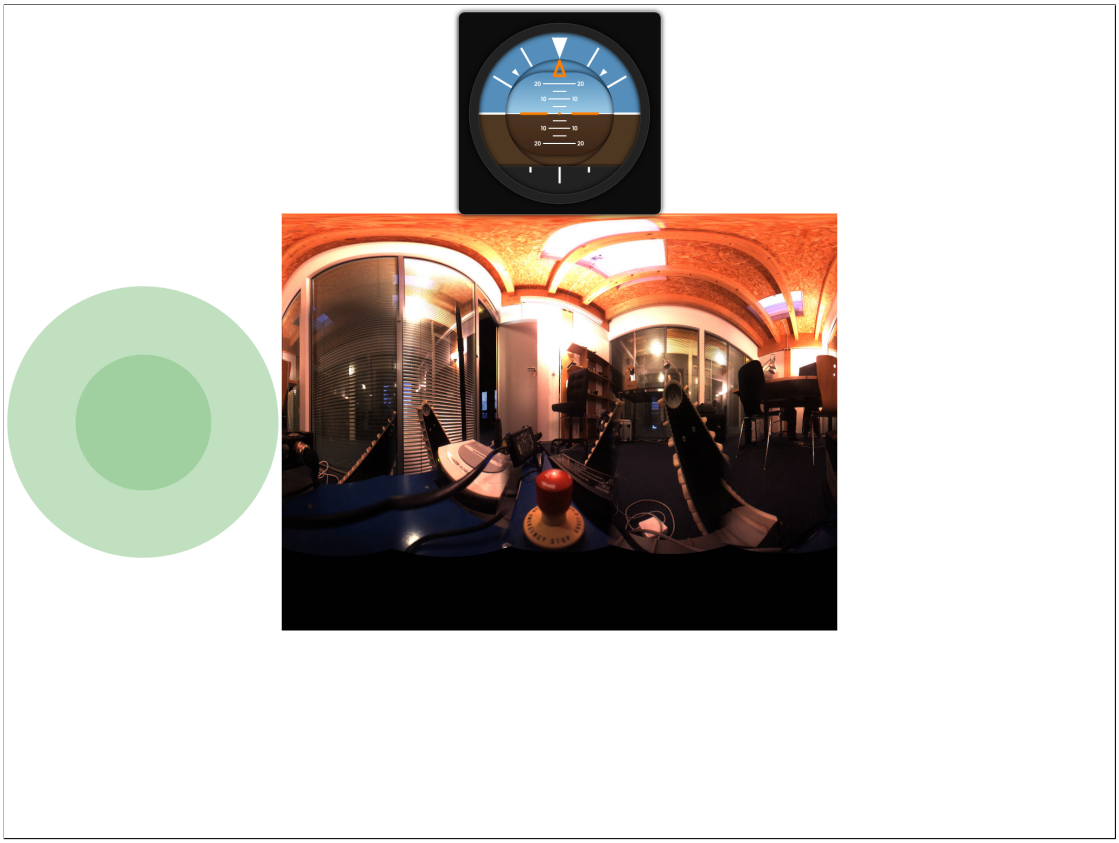


Figure 4.5 – The URC GUI created for the Absolem.

4.2 Conclusion

This resulting project allows to control a great variety of robots and offers a new way to create interfaces and controllers for robots through web technologies. The only limit is your imagination. Moreover, the reusability of widgets is a huge feature that reduce significantly the time of development once the developer base is large enough. Furthermore, the fact that is possible to have several interfaces for the same driver allows you to adapt them depending on their use. For instance, you can adapt the layout for left and right handed. Another idea is to create an interface specifically for debugging, that gives you access to more data about the robot than the default interface. What is more, as the project is mainly based on web technologies, it is usable on other platforms with some concessions (Figure 4.6), because you cannot do TCP/IP and SSH connections, and you need access to a webserver that serves the embedded website of the URC. Finally, being compatible with ROS offer big prospect of usage, because that provides compatibility with various types of robots out of the box.

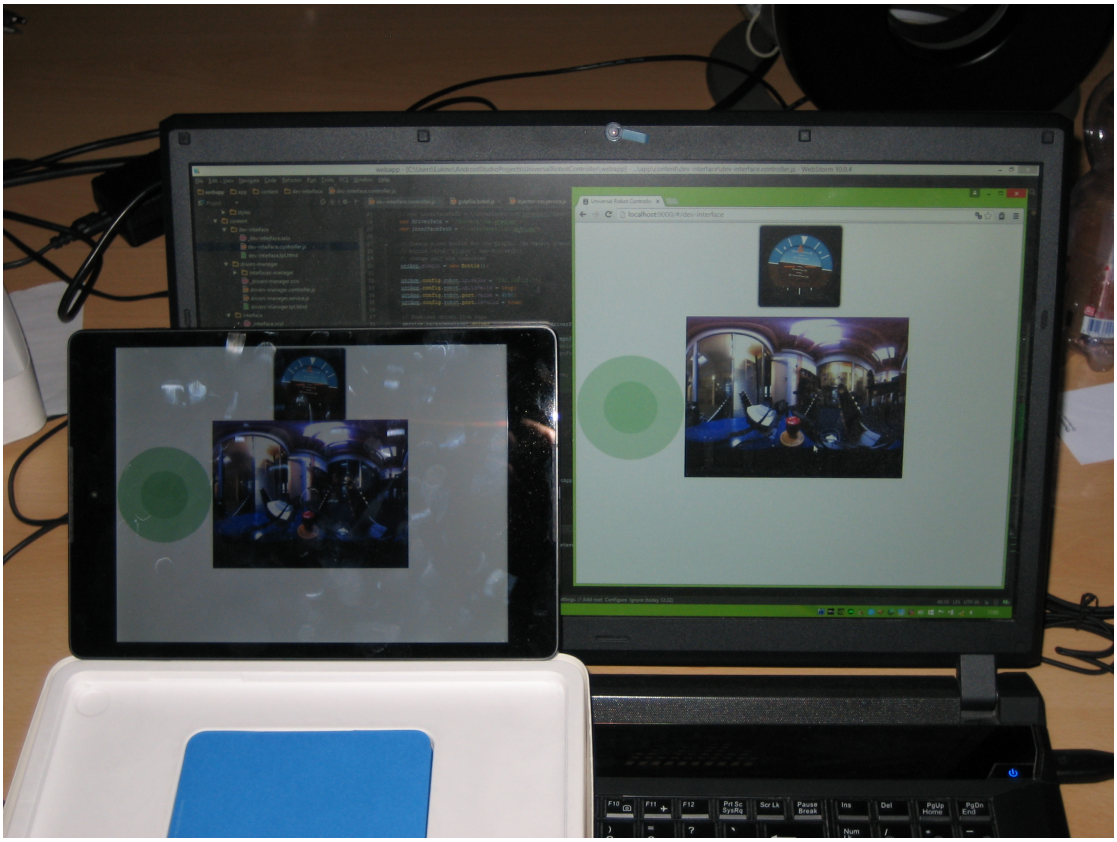


Figure 4.6 – At this stage of development, using it on iOS or a PC requires a webserver that serves the embedded website of the URC.

The freedom offered by the application comes with a great responsibility, because if developers code widgets and drivers without taking in consideration optimization and

quality of code, the GUI may be subject to jerkiness.

The GUIs created during this project are prototypes that shows the potential of the URC. However, the tool to create intuitive and responsive GUIs is already provided by the CSS technology.

The application is subject to many improvements. First, as stated in section 2.2.3.3, the *settings* webpage is not enough flexible, because you can only set one IP and one port. For instance, the Absolem driver need two IP addresses: one for the robot and one to retrieve the images, hence making this webpage customizable according to the drivers would be interesting. Secondly, we could add the possibility to push new drivers, interfaces, and widgets to the URR without needing to copy them manually on the repertory of the server. Finally, in order to increase the user base, it should be adapted for iOS.

A User Manual

This appendix chapter focuses on how we use the URC application (Figure A.1, Figure A.2, Figure A.3, Figure A.4, Figure A.5, and Figure A.6). It is important to use the back button for coming back to the preceding page when you navigate in the application.

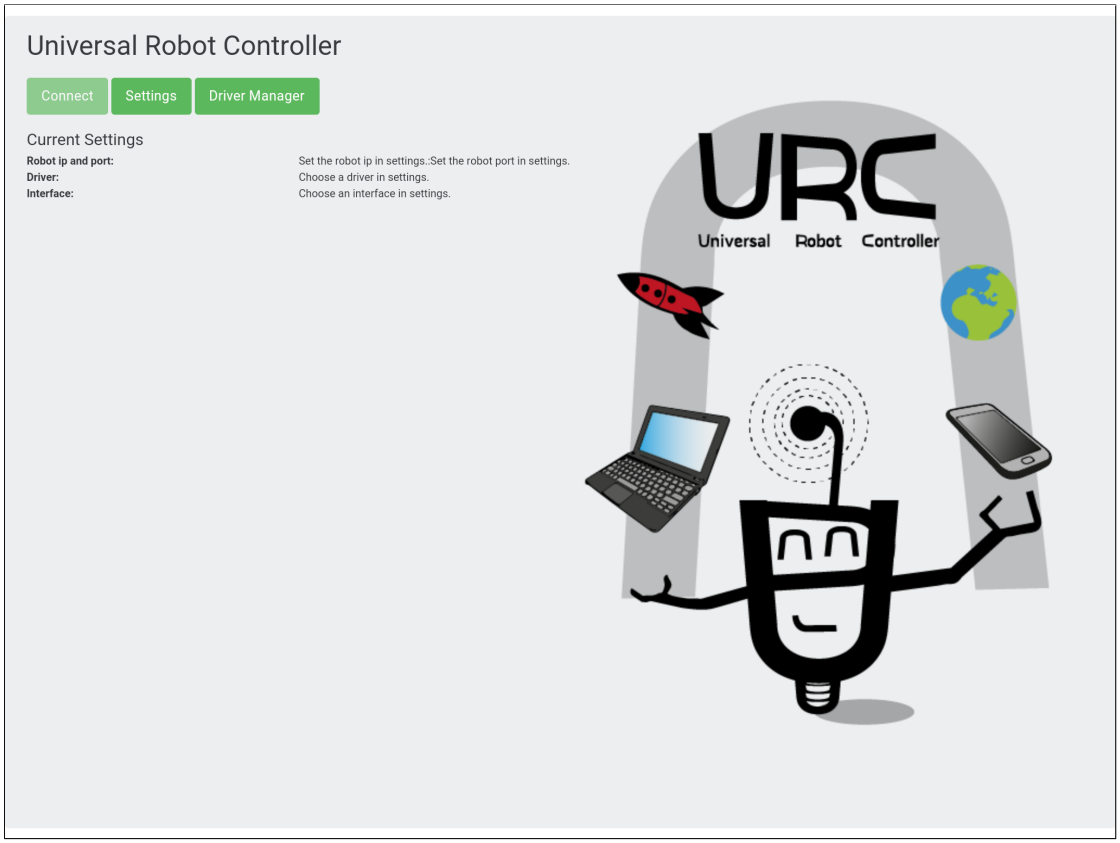


Figure A.1 – The *menu* page of the application. The `Current Settings` table shows the settings used to connect to the robot. You can configure these settings by using the *settings* page. However, you have to first download drivers by opening the *drivers manager* page. Note that the button `Connect` is disabled as long as there are undefined settings.

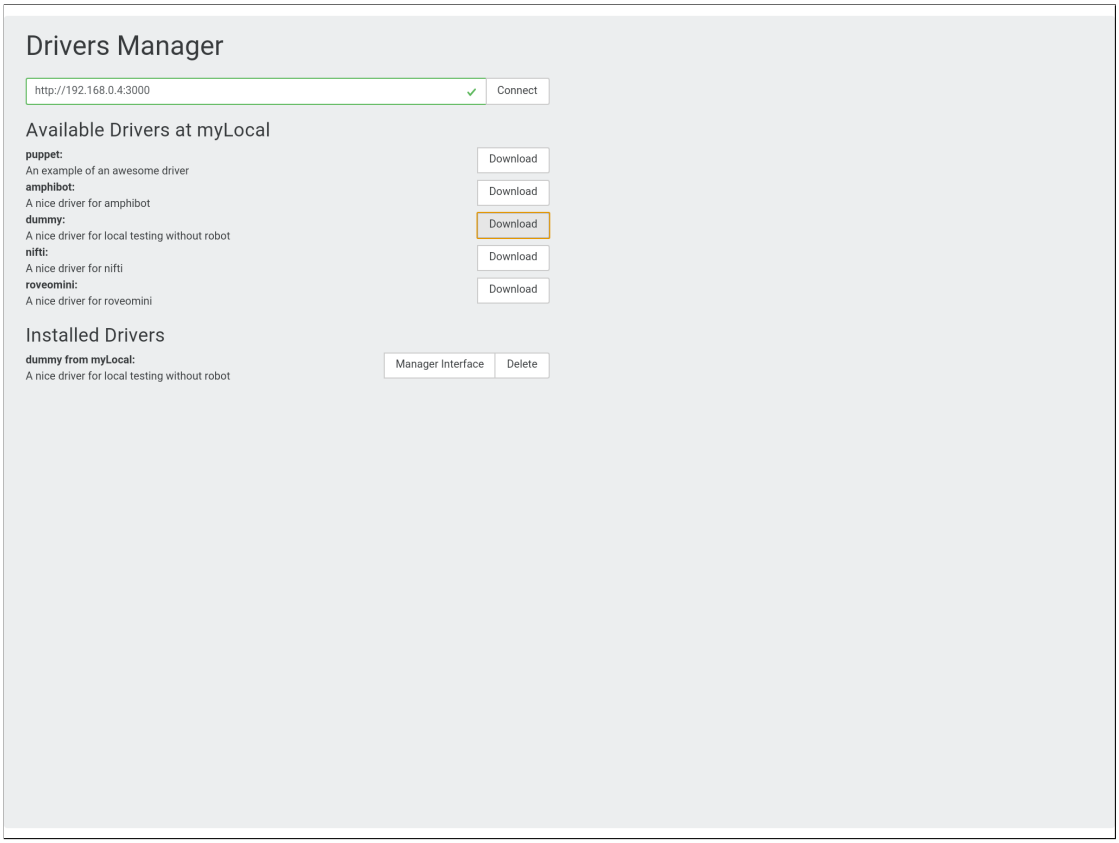


Figure A.2 – The *drivers manager* page. Once connected to a repository the list of available drivers will appear. Simply click on `download` to add it to the application. Each driver comes with a default interface, but you can manage interfaces in the *interfaces manager* you can access with the `Manage interfaces` button.

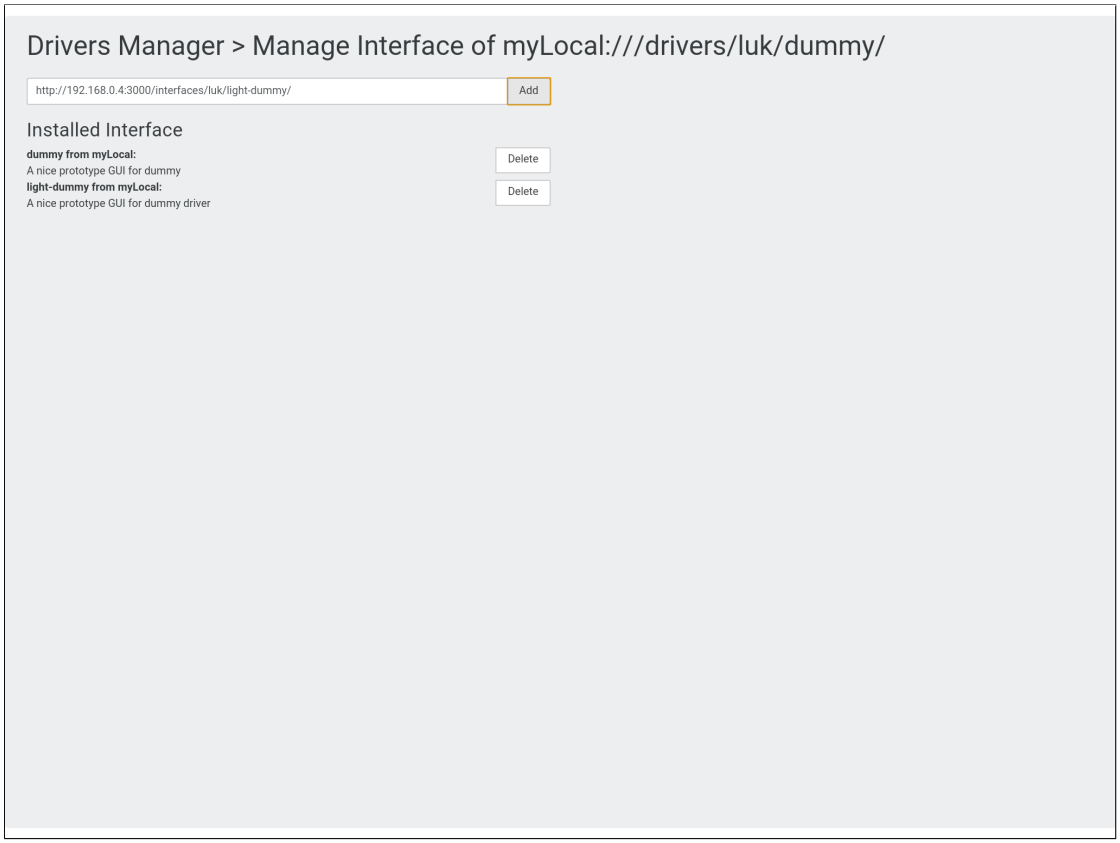


Figure A.3 – The *interfaces manager* page. To add a new interface you have to put the exact directory path of the interface as shown in the figure. Once added, it will directly appear in the installed interface list, where you can easily remove an interface if it is not needed anymore.

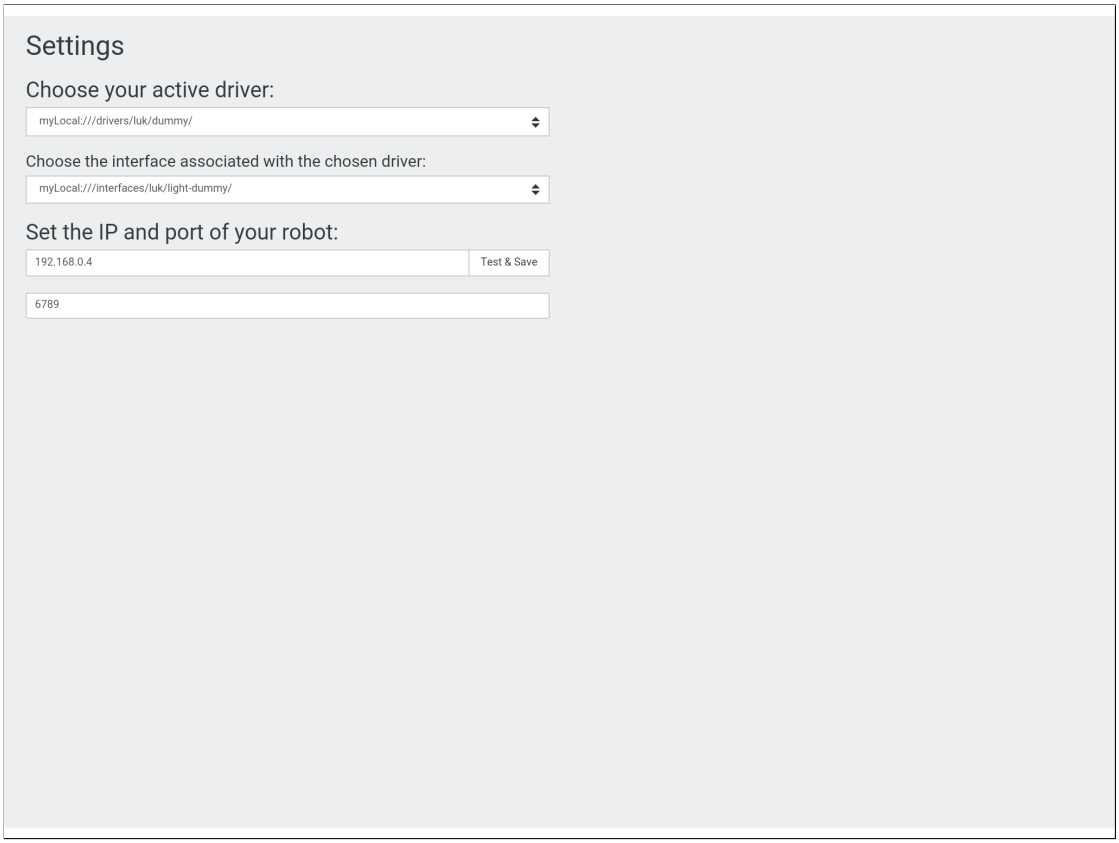


Figure A.4 – The *settings* page. Once you have downloaded a driver, you can go to the *settings* page. This page allows you to choose which driver and interface you want to use. Furthermore, you can set the IP and the port of your robot. The button `Test & Save` will save the IP if the ping is successful. The driver, the interface and the port are automatically saved when modified.

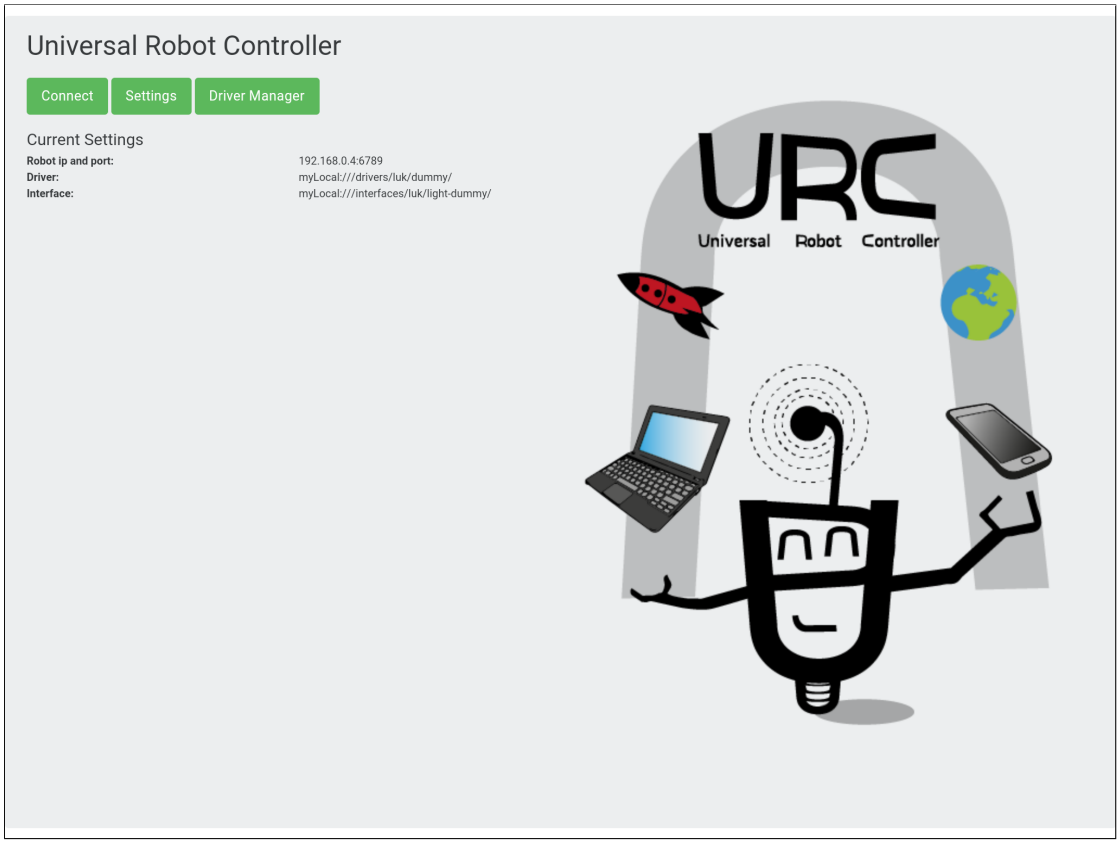


Figure A.5 – The *menu* page with settings set. Once the settings are set, they will appear in the table `Current Settings`. You can now connect to the robot using the button `Connect` which redirects to *GUI* page.

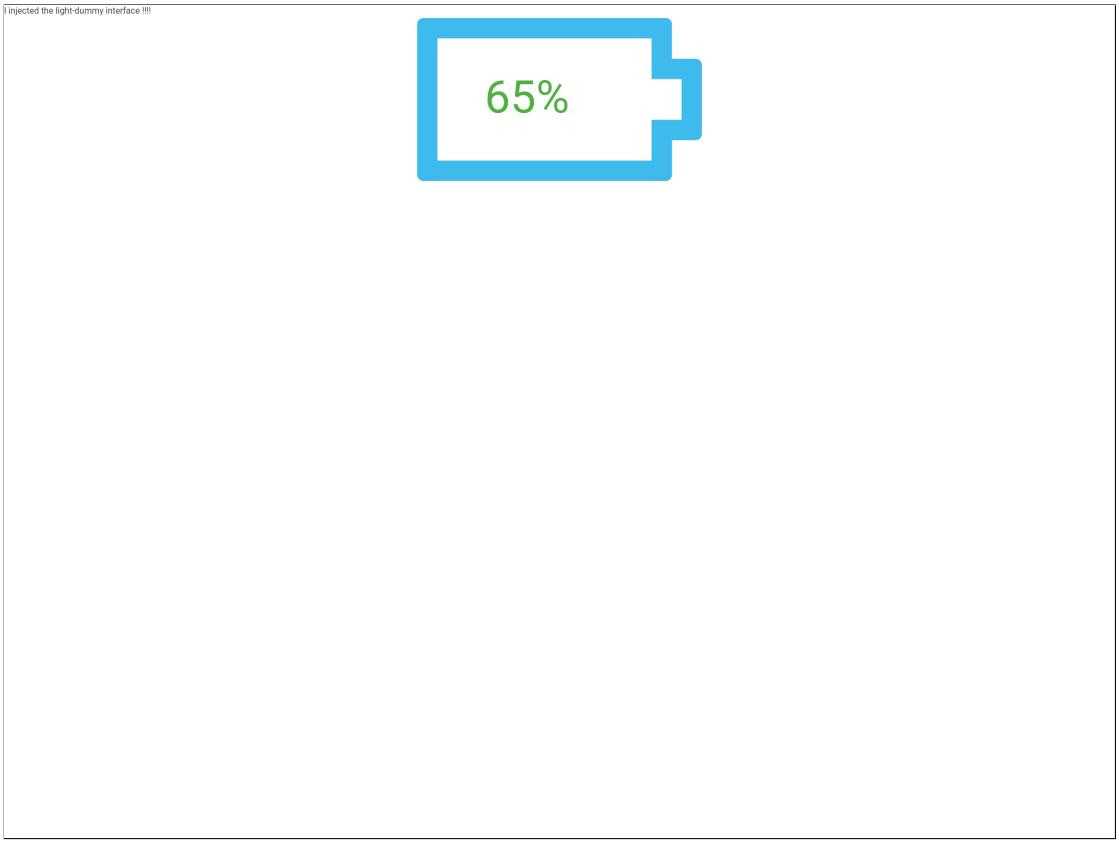


Figure A.6 – The *GUI* page. Once connected to your robot successfully you should see the interface. Please remember to use the back button to disconnect to the robot safely.

B Programmer documentation

This appendix details the URC and URR projects in all its facets such as codes, libraries, structure, and naming convention. The objective is to be able to exploit the URC and the URR at the end. Make sure before to read the chapter 2. All the development was done with a tablet Nexus 9 using Android 6.0 (but should work with Android 5.0 or higher). Make sure that your `WebView` is always up to date.

To have access to the project sources of URC and URR, please ask to the supervisors that were in charge of this project.

B.1 Prerequisite

The embedded website of the URC is coded using web technologies (i.e. JavaScript, CSS (SASS), HTML, SVG) and uses Gulp 4 .0 [36] as the build system. It uses the architectural pattern model-view-controller [37] (MVC), thus having some knowledge about MVC is useful. The JavaScript code uses a lot of promises [38], hence it is mandatory to know how they work. Finally, for the Android part, knowing Java is a plus.

B.1.1 Libraries

B.1.1.1 JavaScript

The website uses several JavaScript libraries, which some of them are not widespread, hence it is recommended understanding them before doing modifications to the project.

- Stateman, a tiny foundation that provides nested state-based routing for complex web applications [39].
- BottleJS, a powerful dependency injection micro container [40].

- localForage, a fast and simple storage library for JavaScript [41].
- Bluebird, a full featured promise library with unmatched performance [42].
- ExpressJS, a fast, unopinionated, minimalist web framework for node [43].
- jQuery, a fast, small, and feature-rich JavaScript library. [44].

B.1.1.2 Java

The Android application uses only two external dependencies, it is not mandatory to master them, but always useful to know what they do.

- JSch, Java Secure Channel [16].
- Rapidoid, the No-bullshit Web Framework for Java [45]

B.2 Naming convention

Most of the files of the URC and the URR use personalized extension in function of what is their usage (Figure B.1).

Type	Extension	Description
Controller	controller.js	Controller from MVC, each webpage has one.
Service	service.js	Service from MVC, used to share reusable code.
Driver	driver.js	The main file of a driver has this name.
Widget	widget.js	The main file of a widget has this name.
Interface	interface.html	The main file of a interface has this name.
Others	js	Only namespace.js and entry.js have this extension.

Figure B.1 – The name convention used for the project files.

B.3 URC

B.3.1 Android application

The complexity of the Android application is mainly concentrated in the bridges used by the website (i.e. `JsSSHBridge` and `JsTCPBridge`) which can be interpreted as managers for SSH and TCP/IP connections. Each bridge uses a corresponding client (i.e. `SSHClient` and `TCPClient`). These two bridges will be injected, as explained in the section 2.2.1.3, into JavaScript context.

`TCPClient` is blocking, in the sense that when you call a send function, the code will block until it receives a response. The delay in favorable situations is between 1-7ms.

`SSHClient` is not blocking, thus executing and reading are done totally asynchronously. The read part is a bit unconventional, and will be explained in detail. The SSH connection is made using Jsch. When you do a connection with Jsch you have access to two streams, one input stream and one output stream. Typically, we will execute command by writing in the output stream our commands. On the other hand, the input stream contains all the terminal lines (i.e. not only the answer of our commands). Then, there are several possibilities to push these lines to JavaScript, but all use a Java thread that will read the input stream continuously (`CharStreamConsumer`).

The first idea is to add the functions `readLine` and `readUntil` to our `JsTCPBridge` and call them from JavaScript. The biggest problem comes from the fact that JavaScript is single-threaded, thus calling continuously the read method will have negative impact on the performances.

The second idea is to do JavaScript callback using `WebView.loadUrl`. The main problem is that we can only `loadUrl` functions that are in the global context of JavaScript, and having global variables and functions is a bad practice.

Finally, the chosen solution is to use WebWorker [46], which allows to have a semblance of multithreading in JavaScript. However, they are very limited, because you cannot have access to external objects when you are in a WebWorker, but you can do XHR. Thus, the idea is to make available the input stream through a web API. For instance, `http://localhost/ssh/" + id + "/readline"`. The web server was created using Rapidoid.

B.3.2 Website

The website is designed to be as modular as possible. We use BottleJS to achieve this goal, which somewhat also do the namespacing [47] of the application. The initialization is done in the file *namespace.js*. In the next sections, we will briefly describe the structure of the website.

B.3.2.1 External libraries

The project never uses the external libraries as global variables. For each library you want to add, you must encapsulate it in a BottleJS services. All these libraries are stored in the folder *app/libs*. In this folder, we also put the *java-bridge* folder which contains the BottleJS service for the injected objects `JsTCPBridge` and `JsSSHBridge`.

B.3.2.2 Services

The *services* folder contain mainly helper services. Most of them are used to reduce the complexity of the website. They are mostly used to manage GUIs (i.e. download, load, remove). The next paragraphs will explain the role of each subfolder of the *services* folder.

B.3.2.2.1 Request The files contained in this folder are mainly used as wrappers. Concretely, we have a wrapper for XHRs in order to use them as promises instead of working with unfriendly and troublesome callbacks. Then, we have a wrapper for localForage in order to add some logs and simplify a little bit his usage. Finally, we have a wrapper to manage settings such as the robot IP, the robot port, the actual driver used, or the selected interface.

B.3.2.2.2 Package manager The services that are included in the *package-manager* folder are used for the interaction between the URC and the URR. Concretely, when you want to download a new GUI, you will use the driver service. It allows to download automatically the default interface and the needed widgets and store them as a unique object in the localForage database under the key that has the following syntax: `repoName:///path/to/driver/on/server`. The driver can be removed by removing its entry in the database. By cascading, it will also delete its widgets and its interfaces. Nevertheless, each downloaded driver is in its own ecosystem, thus drivers deletion does not affect each other. We can add and delete interfaces independently, but widgets cannot.

B.3.2.2.3 Injectors The injector services are used to inject files into a webpage, and remove them if we do not need them anymore. We have three types of injection: HTML, CSS, JavaScript. The JavaScript injector injects Blob representation of JavaScript files into `script` tags using `window.URL.createObjectURL` [48]. The CSS injector works like the JavaScript one, but with `link` tags, and the HTML injector will concretely replace a `div` with the content of the HTML file to inject. These injectors are used by the package manager when we load a GUI.

B.3.2.2.4 Others Finally, we have three free electrons. First, a ping service in JavaScript. Secondly, a URL service that is used to normalize URLs and to create local identifier for drivers, widgets and interfaces. These local identifiers are used as key in the localForage storage. And thirdly, a router service which role is to change the webpage content when we switch to another webpage using Stateman library.

B.3.2.3 Webpages and Routing

All the webpages are stored in the folder *content*. The structure of this folder is mirror of how webpages can be accessed on the website, but without taking in account the *menu* webpage, because we want to avoid a useless encapsulation of all the other webpages in it. For instance, if from the webpage *Foo* we can access to the webpages *Bar* and *Baz*, the *content* folder will contains a folder called *Foo* that contains two subfolders: *Bar* and *Baz*.

B.3.2.4 Databases

The website uses two databases. One to store drivers only, and another to store the settings. It is important to store the drivers in a local database because you cannot be sure to have access to the Internet when you are connected to the robot WiFi.

B.3.2.5 Entry point

The entry point of the application is defined in the file *entry.js*. Its unique goal is to activate Stateman and load the settings stored in the database.

B.4 URR

The URR is a little server created with ExpressJS that stores all the drivers, widgets, and interfaces. Each driver, widget, and interface is described by a JSON file called *urcpm.json* (for URC package manager). Be careful of the confusion, even if the name is the same, the structure of *urcpm.json* is not exactly the same for drivers, widgets, and interfaces (Figure B.2). This strange choice of having the same file name with different content comes from the fact that initially each *urcpm.json* should have the same content, but the project advancing, we had to make some concessions for each component and the *urcpm.json* name was not change. In the near future, it should be refactorize in three distinct filenames (e.g. *urcpm-driver.json*, *urcpm-widget.json*, *urcpm-interface.json*).

Key	Type	Description
name	text	Name of the driver/widget/interface
description	text	Description of the driver/widget/interface
type	text	Take the value: driver, widget, or interface
main	text	Relative path to the main JavaScript file of widgets and drivers, OR to the HTML file that set up the layout for interfaces
dependencies	array	An array of relative path to JavaScript file that need to be injected

(a) Common entries of *urcpm.json* for drivers, interfaces, and widgets.

Key	Type	Description
css	dictionnary	Key-value pair of (resource name, resource relative path)
widgets	array	List of absolute path to used widgets
interfaces	array	List of absolute path to default interfaces, cannot be empty

(b) Specific entries of *urcpm.json* for drivers.

Key	Type	Description
css	array	List of relative path to CSS files
resources	dictionnary	Key-value pair of (resource name, resource relative path)

(c) Specific entries of *urcpm.json* for widgets.

Key	Type	Description
css	array	List of relative path to CSS files

(d) Specific entries of *urcpm.json* for interfaces.

Figure B.2 – Description of the *urcpm.json* for each component.

The web server has exactly the same structure as the *app* folder of the URR project, for instance if your server is running and you navigate to `http://localhost:8000/widgets/luk/walle/urcpm.json` you will access the file *app/widgets/luk/walle/urcpm.json*. Moreover, we set up a very simplistic web API (Figure B.3).

Path	Description
/	Give the name of the repository and list all drivers, widgets, and interfaces path
/drivers	List all drivers path
/widgets	List all widgets path
/interfaces	List all interfaces path

Figure B.3 – Web API of the URR.

Finally, *urcpm.json* can take the form of a object called `urcof` (for URC object file). It is semantically the same. However paths are replaced by Blob or text content of the path files (Figure B.4).

Key	Urcof representation
main	Blob
dependencies	List of Blob
css	List of Blob
resources	Key-value pair of (resource name, text content of the resource)
widgets	Key-value pair (widget absolute path, widget urcof)
interfaces	Key-value pair (interface absolute path, interface urcof)

Figure B.4 – The conversion used to transform the key of a *urcpm.json* into a `urcof`.

B.5 How all those things work ?

In order to describe how the elements presented in the section B.3 and section B.4 interacts, we designed UML sequences of some fundamental use cases. It is supposed that you read the section A, which presents the URC application. Finally, even if some simplifications have been done this chapter will give you a good view of how things work.

B.5.1 Connection to a repository

When the user connects to a repository from the *drivers manager* webpage, the website will ask the URR for the name of the repository and its driver list. Then the webpage is updated with the received information (Figure B.5).

B.5.2 Download a driver

Downloading a driver is the most complex part of the project. It works in two phases which are done sequential for drivers, widgets, and interfaces. The first phase is to download the files from the URR using the paths contained in the *urcpm.json*. Then we store the files as Blob or text content in the `urcof` object (Figure B.6). Notice that the `urcof` of a driver will contain a `urcof` for all used widgets and all interfaces.

B.5.3 Remove a driver

Removing a driver is extremely simple, because you just have to remove it from the local database of drivers (Figure B.7).

B.5.4 Add an interface

Adding a new interface is straightforward, you just have to add the `urcof` of the interface to the `urcof` of the driver stored in the database (Figure B.8). Then, you can select the

new interface in the *settings* webpage.

B.5.5 Remove an interface

Similarly as adding an interface, to remove an interface of a driver you just have to remove its `urcof` of the driver `urcof` stored in the database (Figure B.9).

B.5.6 Connect to a robot

When you have correctly filled all the settings you can connect to the robot. The connection will use the settings selected to choose the correct driver, interface, and widgets to load (inject) (Figure B.10).

B.5.7 Disconnect from a robot

When you disconnect from a robot, the *GUI* webpage will take care of cleaning all the files injected in the DOM. However your driver must have defined a `onLeave` function that clears all kinds of timeouts or intervals.

B.5.8 The Dev Interface button

There is a special button available for developers in the *menu* webpage of the website, called `Dev Interface` (should be disable/hidden in production). This button work like the `Connect` button. However it will download the driver and the interface on the fly each time using hardcoded values for the robot IP, the robot port, the driver path, the interface path, and the repository address (see the file *dev-interface.service.js*). This functionality is very useful when developing a new driver, interface or widget.

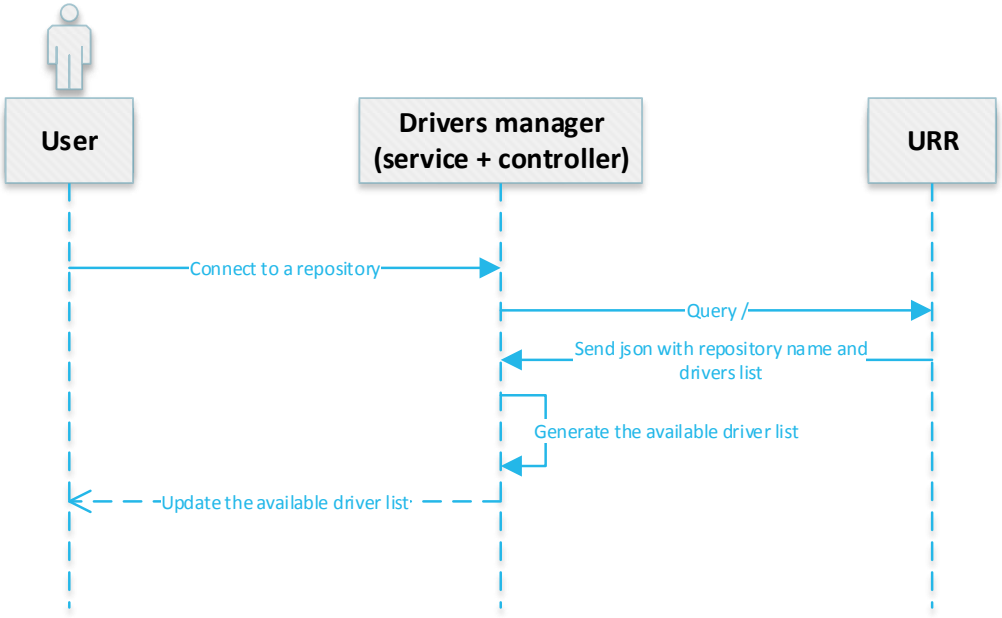


Figure B.5 – The UML sequence representing the connection to a repository in the *drivers manager* webpage.

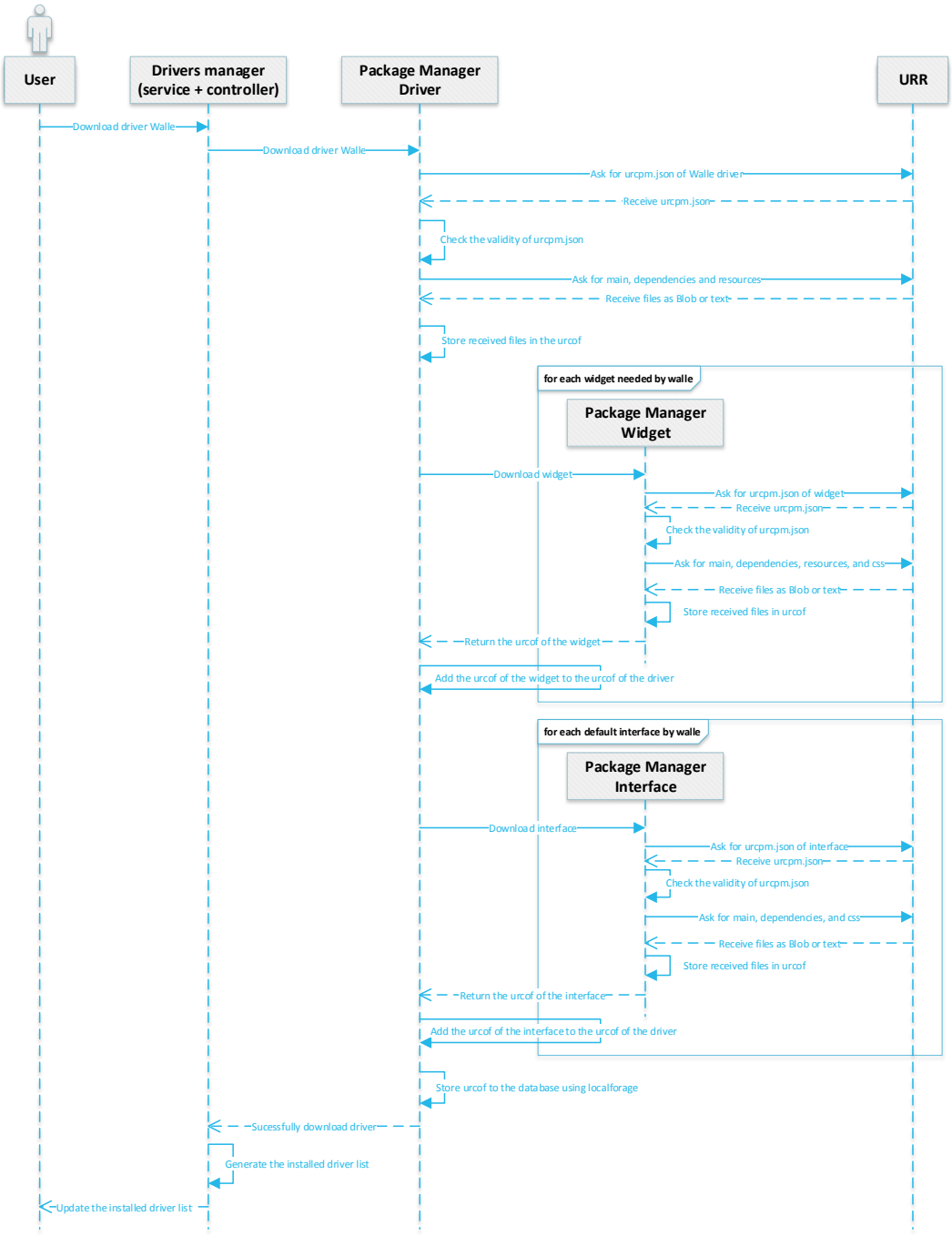


Figure B.6 – The UML sequence representing the download of a driver from a repository.

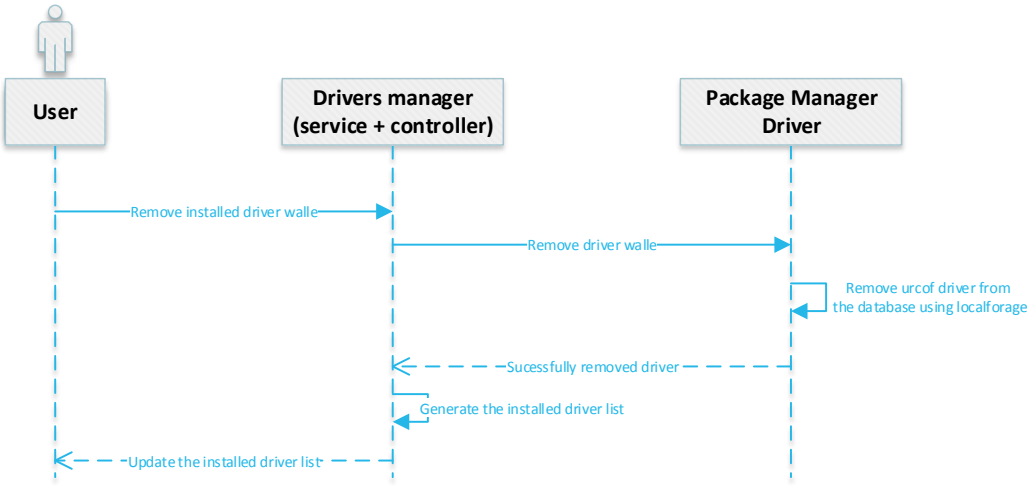


Figure B.7 – The UML sequence representing the remotion of a driver from the URC.

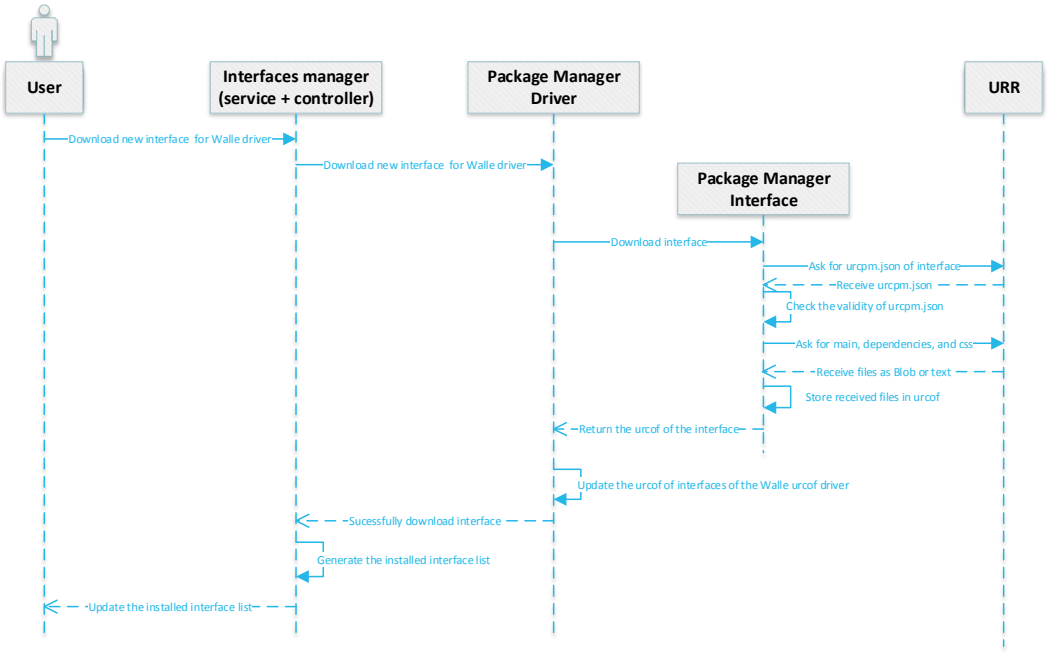


Figure B.8 – The UML sequence representing the addition of an interface to a driver.

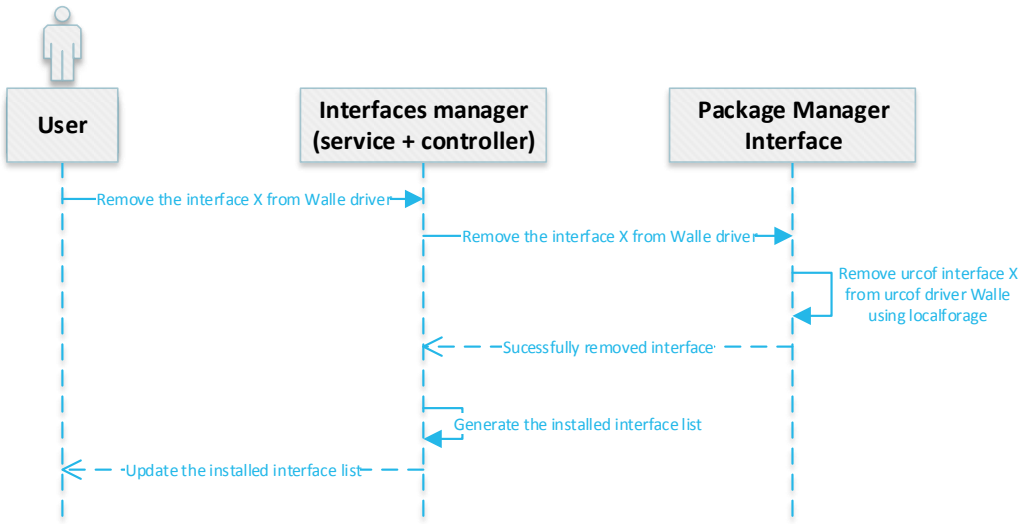


Figure B.9 – The UML sequence representing the remotion of an interface from a driver.

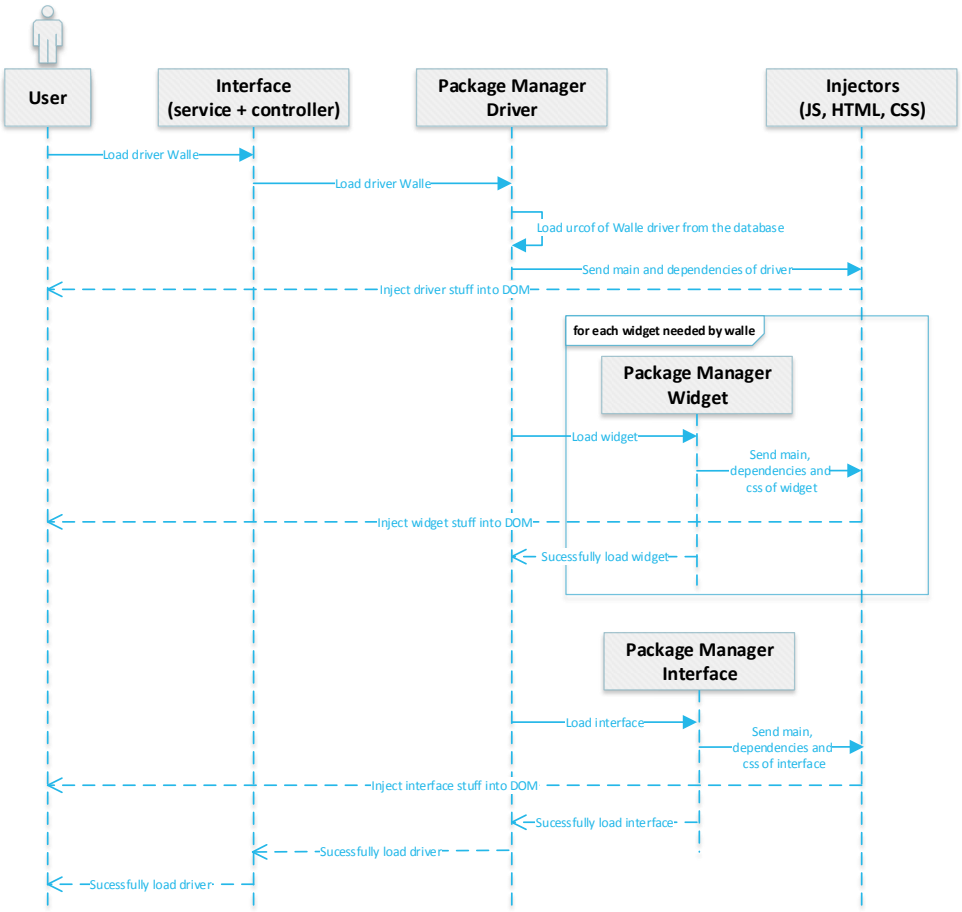


Figure B.10 – The UML sequence representing the connection to a robot.

B.6 Add a new GUI to the URR

This section will explain how to add new drivers, interfaces, and widgets, but there are a few elements to take into account. First, you should not need to modify the URC to add a new component. You only have to add files in the URR. Secondly, URC and URR are two distinct project, nevertheless you have access to the libraries available in URC, but you do not have autocompletion between the projects (i.e. your integrated development environment will warn you about unresolved variables or functions).

We will use a concrete example to explain how to add new widgets, drivers, and interfaces. The final goal is to have a little example that shows batteries charging at the speed of 1% by second and when it reaches 100%, it starts again from 0. The tick of charging will be received from a local server which emulates a robot. The driver will be called *puppet*.

B.6.1 Create the puppet driver skeleton

1. Create a folder called *puppet* in the *app/driver/* folder. This folder will contain only files strictly related to the driver, but nothing about widgets and interfaces.

2. Create an empty file *puppet.driver.js* in the *puppet* folder.

3. Create an empty file *urcpm.json* in the *puppet* folder.

4. Fill the *puppet.driver.js* with the listing B.1.

5. Fill the *urcpm.json* with the listing B.2.

B.6.2 Create the battery widget

1. Create a folder called *battery-svg* in the *app/widgets/* folder.

2. Create an empty file *battery-svg.widget.js* in the *battery-svg* folder.

3. Create an empty file *urcpm.json* in the *battery-svg* folder.

4. Fill the *battery-svg.widget.js* with the listing B.3.

5. Fill the *urcpm.json* with the listing B.4.

6. Create the file *battery.svg* with the content of the listing B.5. This SVG represents our battery.

```
(function(urcApp) {
  'use strict';

  urcApp.plugin.factory('driver', function(c) {
    return new DriverService(); // The name of the function is totally
↪  arbitrary
  });

  function DriverService() {
    var self = {
      activate: activate,
      onLeave: onLeave
    };

    // Function called when we connect with the robot
    function activate() {}

    // Function called when we cancel our connection with the robot
    function onLeave() {}

    return self;
  }
})(urcApp);
```

Listing B.1 – Content of the *puppet.driver.js*.

B.6.3 Create the interface

1. Create a folder called *puppet* in the *app/interfaces/* folder.

2. Create an empty file *puppet.interface.html* in the *puppet* folder.

3. Create an empty file *puppet.css* in the *puppet* folder.

4. Create an empty file *urcpm.json* in the *puppet* folder.

5. Fill the *puppet.interface.html* with the listing B.6.

6. Fill the *puppet.css* with the listing B.7.

7. Fill the *urcpm.json* with the listing B.8.

```
{
  "name": "puppet",
  "description": "An example of an awesome driver",
  "type": "driver",
  "main": "puppet.driver.js",
  "dependencies": [],
  "widgets": [],
  "interfaces": []
}
```

Listing B.2 – Content of the *urcpm.json* of the puppet driver.

B.6.4 Create our emulated robot

The server that emulates a robot is coded in Java. It is a simplistic code (Listing B.9) where each time we ask for the current battery percentage we create a new TCP/IP connection. ***Never do something like that in production code.***

B.6.5 Complete the driver skeleton

First, we have to update the `interfaces` and `widgets` field of the *urcpm.json* of the puppet driver by adding the puppet interface and our SVG battery (Listing B.10). Then, we have to write the driver, and the first thing to do is to inject in the `DriverService` all the needed dependencies. In our case, we will need to inject `TCPBridge` to communicate with our robot, `batterysvg` our widget, `urcApp.config.robot` that stores the robot information (IP and port), and `$` the jQuery plugin. Then we will just add a function that will inject and update our battery, in our case `activateBattery`. Remember to clear the timeouts and intervals in the `onLeave` function. The final code is shown in the listing B.11.

B.6.6 Try the puppet driver

Now you can launch the robot emulator and download the driver puppet in the URC application. Set the settings correctly and then go to the *GUI* webpage. If everything went well, you should see the same image as in the figure B.11.

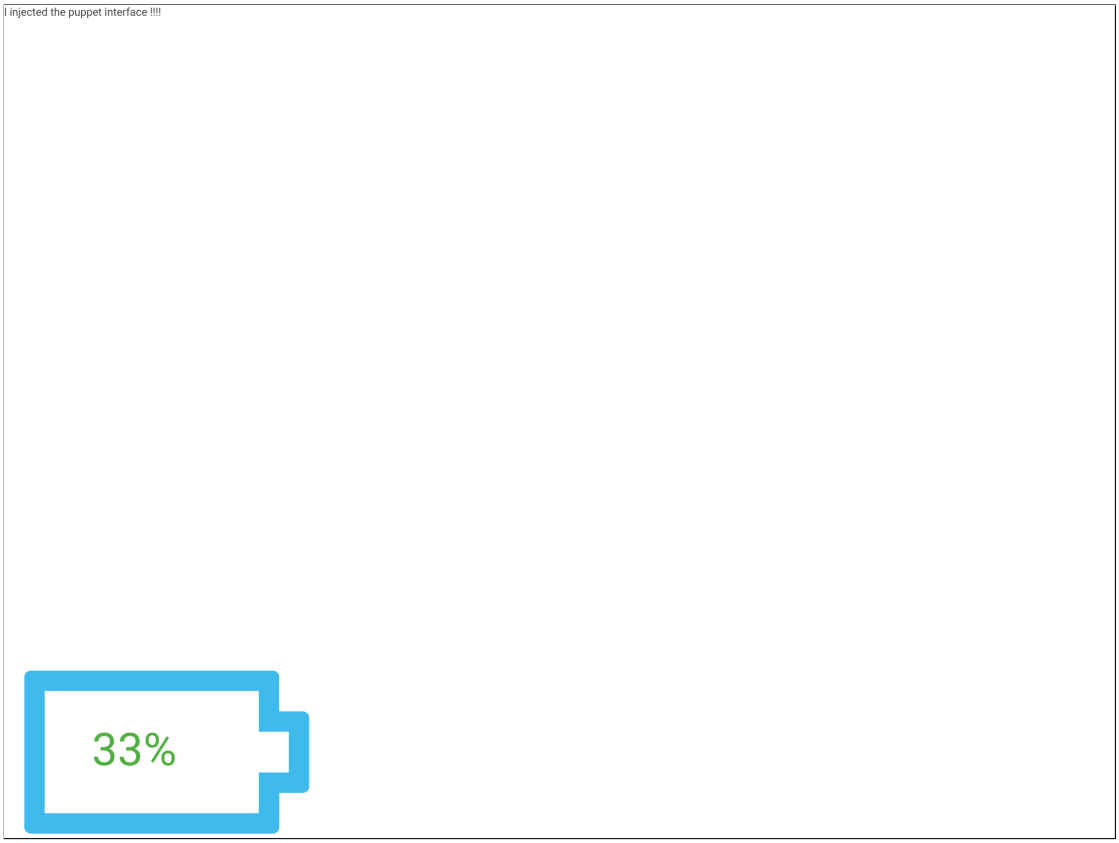


Figure B.11 – The image you should see when you open the *GUI* webpage.

```
(function(urcApp) {
  'use strict';

  urcApp.plugin.factory('widgets.batterysvg', function(c) {
    // The third argument need the path of the current widget
    return new WidgetService(urcApp.libs.svg, urcApp.libs.stampit,
↪   urcApp.config.driver.urcof.widgets['/widgets/battery-svg/'].resources);
  });

  function WidgetService(SVG, stampit, resources) {
    var self = {
      newBattery: newBattery
    };

    var batteryStamp = stampit.methods({
      setLevel: function(level) {
        this.draw.select('#Layer_1 text').each(function() {
          this.text(function(add) {
            add.tspan(level + "%");
          });
        });
      }
    }).refs({
      idElement: null,
      draw: null
    }).init(function() {
      this.draw = SVG(this.idElement);
      this.draw.svg(resources['batterysvg']);
    });

    // Create a battery as a stamp
    function newBattery(idElement) {
      return batteryStamp({idElement: idElement});
    }

    return self;
  }

})(urcApp);
```

Listing B.3 – The content of *battery-svg.widget.js* where `urcApp.libs.svg` is the svg.js [49] library, and `urcApp .libs.stampit` is the Stampit [50] library.

```
{
  "name": "battery-svg",
  "description": "A nice widget battery that use an svg",
  "type": "widget",
  "main": "battery-svg.widget.js",
  "dependencies": [],
  "css": [],
  "resources": {
    "batterysvg": "battery.svg"
  }
}
```

Listing B.4 – Content of the *urcpm.json* of the battery-svg widget.

```
<?xml version="1.0" encoding="utf-8"?>
<svg version="1.1"
    xmlns="http://www.w3.org/2000/svg"
    xmlns:xlink="http://www.w3.org/1999/xlink"
    x="0px" y="0px"
    viewBox="-17 269 448.1 256"
    enable-background="new -17 269 448.1 256"
    xml:space="preserve">
<g id="Layer_2">
    <path fill="#3EBAEC"
        d="M420.9,333H384v-54c0-5.5-4.3-10-9.9-10h-381c-5.6,0-10.1,4.5-10
        .1,10v236c0,5.5,4.5,10,10.1,10h381.1c5.5,0,9.9-4.5,9.9-10v-54H421
        c5.6,0,10.1-4.5,10.1-10V343C431,337.5,426.5,333,420.9,333z M399,4
        29h-14.8H352v32v32H15V301h337v32v32h32.2H399V429z"/>
</g>
<g id="Layer_1">
    <rect x="39" y="366" fill="none" width="296" height="98"/>
    <text id="value" transform="matrix(1 0 0 1 89.1885 417.1191)"
        fill="#51AF41" font-family="'MyriadPro-Regular'"
        font-size="72px">
            XXX %
    </text>
</g>
</svg>
```

Listing B.5 – Content of the *battery.svg*.

```
I injected the puppet interface !!!!
<div id="battery-div"></div>
```

Listing B.6 – Content of the *puppet.interface.html*.

```
// The battery will be in the bottom right-side of the page
#battery-div {
  position: absolute;
  bottom: 0px;
}
```

Listing B.7 – Content of *puppet.css*.

```
{
  "name": "puppet",
  "type": "interface",
  "main": "puppet.interface.html",
  "description": "A nice interface for puppet driver",
  "css": [ "puppet.css"]
}
```

Listing B.8 – Content of *urcpm.json* of puppet interface.

```
import java.io.BufferedReader;
import java.io.DataOutputStream;
import java.io.InputStreamReader;
import java.net.ServerSocket;
import java.net.Socket;

class Main {
  public static void main(String argv[]) throws Exception {
    String clientLine;
    Long send;
    ServerSocket servSocket = new ServerSocket(6789);

    System.out.println("Server launch");

    while (true) {
      Socket connectionSocket = servSocket.accept();
      BufferedReader inFromClient = new BufferedReader(new
↪  InputStreamReader(connectionSocket.getInputStream()));
      DataOutputStream outToClient = new
↪  DataOutputStream(connectionSocket.getOutputStream());
      clientLine = inFromClient.readLine();
      System.out.println("Received: " + clientLine);

      send = (System.currentTimeMillis() / 1000) % 100;
      System.out.println("Send: " + send);
      outToClient.writeBytes(send + "\n");
      outToClient.flush();
    }
 }
}
```

Listing B.9 – Code of our emulated robot for the puppet example.

```
{
  "name": "puppet",
  "description": "An example of an awesome driver",
  "type": "driver",
  "main": "puppet.driver.js",
  "dependencies": [],
  "widgets": [ "/widgets/battery-svg/"],
  "interfaces": ["/interfaces/puppet"]
}
```

Listing B.10 – Content of the updated version of *urcpm.json* for the puppet driver.

```
(function(urcApp) {
  'use strict';

  urcApp.plugin.factory('driver', function(c) {
    var battery = c.widgets.batterysvg;
    return new DriverService(urcApp.libs.jbridge.TCPBridge, battery,
↪  urcApp.config.robot, urcApp.libs.jquery);
  });

  function DriverService(TCPBridge, battery, robot, $) {
    var self = { activate: activate, onLeave: onLeave };
    var rafId;

    function activate() {
      activateBattery();
    }

    function activateBattery() {
      var bat;
      var batId = '#battery-div';
      var $bat = $(batId);
      // If the div exists in the chosen interface
      // inject battery, and start rendering
      if ($bat.length > 0) {
        bat = battery.newBattery(batId.substr(1)); // We don't want the #
        renderLoop();
      }

      function renderLoop() {
        var idTCP = TCPBridge.newClient(robot.ip.value, robot.port.value,
↪  2000);
        bat.setLevel(JSON.parse(TCPBridge.sendString(idTCP, 'Give me the
↪  battery %')).data); // The sendString is blocking
        TCPBridge.closeClient(idTCP);
        rafId = window.requestAnimationFrame(renderLoop);
      }
    }

    function onLeave() {
      window.cancelAnimationFrame(rafId);
    }

    return self;
  }
})(urcApp);
```

Listing B.11 – Content of the *puppet.driver.js*.

References

[1] http://www.parrot.com/. Consulted from September 2015 to January 2016.

[2] http://biorob.epfl.ch/. Consulted from September 2015 to January 2016.

[3] http://www.rovenso.com/. Consulted from September 2015 to January 2016.

[4] http://www.bluebotics.com/. Consulted from September 2015 to January 2016.

[5] http://www.ros.org/. Consulted from September 2015 to January 2016.

[6] http://www.nifti.eu/news/the-new-nifti-robot-platform. Consulted from September 2015 to January 2016.

[7] http://yises.com/blog/wp-content/uploads/2015/02/app_nativevshybrid.png. Consulted from September 2015 to January 2016.

[8] http://www.idc.com/prodserv/smartphone-os-market-share.jsp. Consulted from September 2015 to January 2016.

[9] https://cordova.apache.org/. Consulted from September 2015 to January 2016.

[10] http://phonegap.com/. Consulted from September 2015 to January 2016.

[11] https://xhr.spec.whatwg.org/. Consulted from September 2015 to January 2016.

[12] https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API. Consulted from September 2015 to January 2016.

[13] http://developer.android.com/reference/android/webkit/WebView.html/. Consulted from September 2015 to January 2016.

[14] https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIWebView_Class/. Consulted from September 2015 to January 2016.

[15] https://msdn.microsoft.com/library/windows/apps/windows.ui.xaml.controls.webview.aspx. Consulted from September 2015 to January 2016.

[16] http://www.jcraft.com/jsch/. Consulted from September 2015 to January 2016.

[17] http://docs.oracle.com/javase/7/docs/api/java/net/Socket.html. Consulted from September 2015 to January 2016.

[18] http://www.json.org/. Consulted from September 2015 to January 2016.

[19] https://tools.ietf.org/html/rfc4648. Consulted from September 2015 to January 2016.

[20] https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API. Consulted from September 2015 to January 2016.

[21] https://github.com/RobotWebTools/rosbridge_suite. Consulted from September 2015 to January 2016.

[22] https://github.com/RobotWebTools/roslibjs. Consulted from September 2015 to January 2016.

[23] https://github.com/WPI-RAIL/jrosbridge. Consulted from September 2015 to January 2016.

[24] https://www.arduino.cc/en/Main/ArduinoBoardYun. Consulted from September 2015 to January 2016.

[25] https://www.sensefly.com/home.html. Consulted from September 2015 to January 2016.

[26] https://www.qinetiq.com/Pages/default.aspx. Consulted from September 2015 to January 2016.

[27] http://www.ehang.com/. Consulted from September 2015 to January 2016.

[28] http://www.parrot.com/fr/produits/skycontroller/. Consulted from September 2015 to January 2016.

[29] http://www.irobot.fr/. Consulted from September 2015 to January 2016.

[30] https://3drobotics.com/. Consulted from September 2015 to January 2016.

[31] http://www.dji.com/. Consulted from September 2015 to January 2016.

[32] https://www.google.ch/maps?source=tldso. Consulted from September 2015 to January 2016.

[33] http://images.slideplayer.com/21/6271527/slides/slide_91.jpg. Consulted from September 2015 to January 2016.

[34] https://upload.wikimedia.org/wikipedia/commons/thumb/c/c1/Yaw_Axis_Corrected.svg/2000px-Yaw_Axis_Corrected.svg.png. Consulted from September 2015 to January 2016.

[35] https://upload.wikimedia.org/wikipedia/commons/c/c0/LIDAR-scanned-SICK-LMS-animation.gif. Consulted from September 2015 to January 2016.

[36] http://gulpjs.com/. Consulted from September 2015 to January 2016.

[37] https://en.wikipedia.org/wiki/Model-view-controller. Consulted from September 2015 to January 2016.

[38] https://promisesaplus.com/. Consulted from September 2015 to January 2016.

[39] https://github.com/leeluolee/stateman. Consulted from September 2015 to January 2016.

[40] https://github.com/young-steveo/bottlejs. Consulted from September 2015 to January 2016.

[41] https://github.com/mozilla/localForage. Consulted from September 2015 to January 2016.

[42] https://github.com/petkaantonov/bluebird/. Consulted from September 2015 to January 2016.

[43] http://expressjs.com/. Consulted from September 2015 to January 2016.

[44] https://github.com/jquery/jquery. Consulted from September 2015 to January 2016.

[45] https://github.com/rapidoid/rapidoid. Consulted from September 2015 to January 2016.

[46] https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers. Consulted from September 2015 to January 2016.

[47] http://addyosmani.com/blog/essential-js-namespacing/. Consulted from September 2015 to January 2016.

[48] https://developer.mozilla.org/fr/docs/Web/API/URL/createObjectURL. Consulted from September 2015 to January 2016.

[49] http://svgjs.com/. Consulted from September 2015 to January 2016.

[50] https://github.com/stampit-org/stampit. Consulted from September 2015 to January 2016.

[51] https://lh3.googleusercontent.com/dDwals-EOH5pDiGzQZB4R0LT8Hy5Yjs_Fwkk5rO3sbqwYyINEo4d4i4-4Wpt7ZYSd38=h900. Consulted from September 2015 to January 2016.

[52] https://lh3.googleusercontent.com/Z0VSekUXl2Mqhs7ID02ivEghtffDIZmRx1jOwIKKKEOt0fYXrhll
h900. Consulted from September 2015 to January 2016.

[53] https://lh3.googleusercontent.com/0BIR8quOsXf2cc9ACNgLzqu3LHshPMh8y7N64sjGwwAA0eRh2nWQU
h900. Consulted from September 2015 to January 2016.

[54] http://www.parrot.com/media/slideshows/slides/2015/07/03/171857165724.jpg.
Consulted from September 2015 to January 2016.

[55] https://developer.android.com/reference/android/webkit/WebView.html#
addJavascriptInterface(java.lang.Object,java.lang.String). Consulted from
September 2015 to January 2016.

[56] http://www.qinetiq.com/services-products/survivability/UGV/robot-controllers/
PublishingImages/lcu.png. Consulted from September 2015 to January 2016.

[57] https://www.sensefly.com/uploads/contentElements/05-plan-emotionx.jpg. Consulted from September 2015 to January 2016.

[58] https://www.sensefly.com/fileadmin/user_upload/sensefly/images/products/
emotion-explained-02.jpg. Consulted from September 2015 to January 2016.

[59] https://ecs7.tokopedia.net/img/product-1/2015/8/28/165560/165560_
18bfaef8-76df-4bb8-a79f-c9ee61d57495.png. Consulted from September 2015
to January 2016.

[60] http://www.irobotweb.com/~/media/Files/Robots/Defense/uPoint/
iRobot-uPoint-SpecSheet.pdf. Consulted from September 2015 to January
2016.

[61] http://doctordrone.com.br/wp-content/uploads/2015/04/
xcontrole-novo-phantom-3.png.pagespeed.ic.Kk_7imQwXE.png. Consulted
from September 2015 to January 2016.

[62] https://3drobotics.com/wp-content/uploads/2015/06/controller_
bottom-1411x1249.jpg. Consulted from September 2015 to January 2016.

[63] http://www.techrepublic.com/article/androids-lack-of-dhcpv6-support-poses-security-and-ipv6-deployment
Consulted from September 2015 to January 2016.