Creating GUI

Thomas Schenk

Contents

[Context 3](#_Toc136292974)

[Problem 4](#_Toc136292975)

[Goal 5](#_Toc136292976)

[Requirements 6](#_Toc136292977)

[Platform choice 7](#_Toc136292978)

[Note about current architecture 7](#_Toc136292979)

[How to export ROS information outside of a ROS environment 8](#_Toc136292980)

[Technical implementation 8](#_Toc136292981)

[Handling incoming WebSocket connection(s) 9](#_Toc136292982)

[Technical implementation 9](#_Toc136292983)

[Streamlining the prerequisites for startup 10](#_Toc136292984)

[System overview 11](#_Toc136292985)

[Implementing requirements 12](#_Toc136292986)

[Live Camera Feed 12](#_Toc136292987)

[Live Mapping Feed 12](#_Toc136292988)

[Attributes/Markers/Comments must be visible on Live Map 13](#_Toc136292989)

[Physical Controls 14](#_Toc136292990)

[Validation 18](#_Toc136292991)

## Context

The Jackal is a product that will be used in high intensity situations, or very low intensity situations. Either way, it needs to have an intuitive and clear user interface. The operators must be able to react quickly on changing scenarios, without having to have technical knowledge of the product. Without such simplicity in its controls, its functionalities will be forever lost in the complexity of separate terminals and different visualization programs.

## Problem

The Jackal currently has no 1 place where the end user can both see and control the data on the Jackal. To be effectively rolled out as a product; it needs this. The operator in the field cannot be expected to carry a laptop and juggle around CLI (Command Line Interface, otherwise known as Terminal) windows or other programs currently needed for executing certain behavioural actions on the Jackal.

In short, the end user has too much responsibility when it comes to managing and performing the necessities for controlling and monitoring the robot.

## Goal

Something needs to be created to eliminate the need for multiple instances of programs, in whichever way shape or form, and which serves the end user with one concise GUI. The form of this product is very free, yet it must have one specific attribute: it must be able to do everything on one page/in one instance.

We need to take the responsibility of managing the system away from the user and expect our system to manage itself.

## Requirements

To limit the development of an overly complicated GUI (the polar opposite of the goal) and to be able to validate that this GUI is in fact a good solution to the problem; we can set some requirements as to what the GUI needs to contain.

As mentioned before, the platform on where this GUI is placed; can be everything imaginable. These requirements are therefore structured to provide guidance to the functionality of the GUI, rather than its technical implementations.

The functionality of the robot (and therefore the GUI) is dependent on the state the robot is in. The offline mode, or online mode. The online mode allows live data transmission to the gui, the offline mode doesn’t allow data transmission via wireless protocols at all. The requirements are therefore split up into these two categories.

Online

1. Live Camera Feed
2. Live Mapping Feed
3. Attributes/Markers/Comments must be visible on Live Map
4. Physical Controls
   1. Buttons to influence the exploration status
   2. Joystick to manually move the robot

Offline

1. Whenever a connection is permitted by the system again, the created map must be shown

### Requirements in real-life scenarios

To give more context to these requirements, we will give a few examples in which the significance of these requirements come to light.

1. Manual exploration of a building

The operator has his eye on a building, he would really like to explore it. Yet, the building is inaccessible for the operator. He cannot guarantee that he will always have a direct line of sight on the robot. He would like to see what the robot sees, and be able to control it without colliding!

Requirements from this userstory:

* Live Mapping

To see the environment live, in a top down view. Quickly revealing objects around the robot.

* Live Camera

To see what the robot sees, in a FPS view. The operator almost feels like he is in the robot driving it himself; making it easier to navigate.

1. Control of the robot on the fly

When the operator is in the field (police officer), he would like to control the robot and see the map in 1 look at a screen. A quick look and adjustment; and back to running towards the bad guys.

Requirements from this userstory:

* Live Mapping

To quickly see the live map to determine what and where the robot is exploring.

* Live controls

To provide a joystick where the operator can quickly reposition the robot if needed.

* Control buttons

To stop, pause or start certain processes on the robot with a simple buttonpress.

## Platform choice

Like mentioned before, the platform to build this product on was free to choose. There were no limitations.

Since ROS already relies heavily on IP when communicating both internally and to different devices; we decided to make a GUI in a web based form. Currently this is built with HTML, Javascript and a Bootstrap overlay to make visualizing easier.

Furthermore, an online webpage ensures that data can be served to a lot of people at the same time. Very convenient for scenarios where control needs to be transferred to another person.

Lastly, web based applications are often easy to port to standalone applications. Since application development is a lot trickier and time intensive; web pages are usually made in a matter of days. Since the logic will remain the same on either an application or a web page; development wise the choice to opt for web now and application in the future, was the most efficient for us.

### Note about current architecture

Currently, the robot is its own webserver. This means that the html file and its scripts are all still on the robot. There is a good reason for this, we don’t have a webserver (yet). In the future this is recommended.

Please note that the webserver should be configured accordingly to the VPN of the robot.

## How to export ROS information outside of a ROS environment

ROS information is easy to pass to other ROS nodes; or even other ROS machines. This can be explained easily by the fact that ROS standardizes their information in certain message types. It even supports custom message types for even greater development.

It is no surprise that without a supporting middleman, non-ROS environments don’t understand ROS message types. We need a software module/ROS node which exports this data in a readable format.

Rosbridge is a ROS node which supports this functionality. It is a node which launches on the robots side and provides a websocket. This websocket allows for a direct connection to the robot. This makes it possible to send ROS type commands from a non-ROS machine. Things like services, topics, calls and even subscriptions are possible.

This websocket is only available locally, or in a local network. The prerequisite is that the client should be able to ping the IP of the jackal. In our case, we have a VPN running. This makes it possible to ping the jackal, and indirectly the websocket, from multiple computers.

### Technical implementation

To implement rosbridge, it can be downloaded as a ROS package. Personally, I prefer doing this via their git repo; and making it myself.

In your working catkin directory, clone:

git clone <https://github.com/RobotWebTools/rosbridge_suite.git>

Important to note, pick/checkout on the ROS1 branch; they also support ROS2.

After this, make the catkin folder like you are used to.

Rosbridge is now ready to use how it is intended. It supports launch files.

Roslaunch Rosbridge\_suite Rosbridge\_websocket.launch

We adjusted ours to specify a certain IP address; so we can call the service by a standard Ip. The adjusted launch file is in the added files.

## Handling incoming WebSocket connection(s)

To finalize the conversion strategy started by Rosbridge, we need something on the (web)server side to handle the websocket connection.

The same development team which is responsible for Rosbridge, also made this; roslibjs. It is a Javascript library which can directly interact and extract the WebSocket connection.

### Technical implementation

To connect and use a ROS object serverside, we execute code from roslibjs on an HTML page.

var ros = new ROSLIB.Ros({

url : 'ws://100.121.193.21:9090'

});

With this piece of code, we create a ‘ros’ object which we can link our various getters to.

## Streamlining the prerequisites for startup

All the above-mentioned points are still separate pieces of a bigger puzzle to host the jackal information to the internet. To put these all together, we made a launch file which contains everything currently needed to achieve just that.

You can call this launch file like so:

Roslaunch autonomous\_exploration webservice.launch

## System overview

A picture containing text, screenshot, diagram, font

Description automatically generated

## Implementing requirements

All code and infrastructure setup regarding connectivity is now set up correctly and is ready to accommodate the coming development phase. In our case, this phase consists of building the requirements we previously stated.

*Since it is so trivial, building a Bootstrap HTML page will not be explained*

### Live Camera Feed

When we researched implementing a live camera feed on our dashboard, we quickly realized that it seemed to differentiate from other modules such as a live map.

ROS handles camera imagery via a topic, no different than anything else. However due to its data requiring a vast mathematical approach to make visible; there are different ROS packages available to do this for you.

One of these packages hosts this converted data on a pre-set address, effectively making the camera topic more like an online stream.

We can use the URL of this stream inside of an <img> div element to display a live feed of the camera.

<div id="img"> <img src="http://100.121.193.21:8080/stream?topic=/camera/color/image\_raw" height="375px"></img> </div>

### Live Mapping Feed

The mapping feed makes use of the roslibjs connection we previously made. It also utilizes another javascript library in the ‘ROS family’ to display ROS data graphically, ros3djs. There also is its counterpart, ros2djs; the names speak for themselves. One implements 3d visualization whilst the other only implements 2d visualizations.

Intuitively, 2d seems the go to. However since markers and models rely on 3d implementation; a 3d viewer is needed.

To implement this, the following lines of code are used to set it up.

let viewer = new ROS3D.Viewer({

divID : 'map',

width : 1200,

height : 750,

antialias : false,

intensity : 0.5,

cameraPose : {x : -1, y : 0, z : 20},

displayPanAndZoomFrame : true,

lineTypePanAndZoomFrame : true

});

let tfClient = new ROSLIB.TFClient({

ros : ros,

angularThres : 0.01,

transThres : 0.01,

rate : 10.0,

fixedFrame : '/map'

});

let mapClient = new ROS3D.OccupancyGridClient({

ros : ros,

rootObject : viewer.scene,

continuous : false,

opacity : 1

});

These lines of code all make objects which use each other.

* Viewer

The viewer is a black box. It only carries the essential data for camera panning and zooming; and provides an interface for data to be displayed in. The viewer itself does not show any data unless given to it by another object.

* tfClient

The tfClient take telemetry and pose data from the robot to determine a position inside the map. It does this by converting all the frames to the current map, so that it can be approached somewhat correctly.

* mapClient

Like stated previously, the viewer only displays data given to it by another object. The mapClient is that object. As you can see, it sets the ‘rootObject’ as viewer. mapClient supplies the viewer with its map data to visualize it.

To give all these setup objects and viewers a place on your website, we have to refer to the viewer ‘id’ attribute in a div in our body.

<div id="map"></div>

### Attributes/Markers/Comments must be visible on Live Map

To display POI’s on the map, we use markers. These markers obviously need to be visible on the map also. Without them; the operator is kind off lost in the virtual world without indication of special points or objects.

We need to add this information to the previously created viewer, like explained there: the viewer displays information given to it. If we also give this marker information to it; it should display that too.

Keyword: should. Currently we are running into a bug where the library throws an unexpected error.

The github issue post can be found here: <https://github.com/RobotWebTools/ros3djs/issues/620>

### Physical Controls

We differentiate the physical control of the robot in two subsections;

* Influencing the exploration status

The exploration package accepts a few parameters which can directly influence its behaviour. Amongs which are starting, stopping and pausing. We want to be able to access and call these from the dashboard.

* Moving the robot

In some scenarios, driving the robot is necessary. While there is a separate controller, it would be nice to be able to do it from the GUI as well. Mostly for scenarios where the operator is moving. Then they can operate the robot fully on one device instead of awkwardly holding a screen and a controller.

#### Buttons to influence the exploration status

The exploration commands on the robot itself are send using an internal system; for outsiders however, a rosservice has been made. A rosservice is an endpoint where you can send commands to, the rosservice’s underlying logic then decides what to do with that command. You can input a lot of different types of variables into these services.

In our case, this service only needs a string. The current implementation only allows for three strings.   
Start, stop and pause. You can declare these into the code like so:

//ExploreCommands

exploreCommandService = new ROSLIB.Service({

ros : ros,

name : '/explore/command',

serviceType : 'explore\_lite/ExploreCommand'

});

startRequest = new ROSLIB.ServiceRequest({

command : 'start'

});

stopRequest = new ROSLIB.ServiceRequest({

command : 'stop'

});

pauseRequest = new ROSLIB.ServiceRequest({

command : 'pause'

});

We can call these programmatically or via user interaction. In our case, we tied them to a set of buttons.

We made a function which takes a specific string to execute the corresponding command, this function is then used on a buttonclick event.

function sendExploreCommand(command) {

if (command == 'start')

{

exploreCommandService.callService(startRequest, function(result) {console.log('succes');});

}

else if (command == 'stop')

{

exploreCommandService.callService(stopRequest, function(result) {console.log('succes');});

}

else if (command == 'pause')

{

exploreCommandService.callService(pauseRequest, function(result) {console.log('succes');});

}

<button type="button" class="btn btn-primary" id="ajaxSubmit" onclick="sendExploreCommand('start');" > Start </button>

<button type="button" class="btn btn-primary" id="ajaxSubmit" onclick="sendExploreCommand('stop');" > Stop </button>

<button type="button" class="btn btn-primary" id="ajaxSubmit" onclick="sendExploreCommand('pause');" > Pause </button>

In the future, these buttons could easily be extended upon. Just make sure that the backend supports and handles these rosservice calls correctly; and you can basically copy paste the current solution.

#### Joystick to manually move the robot

To move the robot physically, we manipulate a topic which is used for controller data. If the physical controller is connected it uses that topic to push movement data to. We emulate the directional messages of a joystick, by implementing a virtual one. We do this using the nipple.js library; which does a lot of the math and visualizing for us.

The code for that looks like this:

createJoystick = function () {

console.log("hoi2");

var options = {

zone: document.getElementById('zone\_joystick'),

threshold: 0.1,

position: { left: 15 + '%' },

mode: 'static',

size: 300,

color: '#000000',

};

manager = nipplejs.create(options);

linear\_speed = 0;

angular\_speed = 0;

manager.on('start', function (event, nipple) {

timer = setInterval(function () {

move(linear\_speed, angular\_speed);

}, 25);

});

manager.on('move', function (event, nipple) {

max\_linear = 1.0; // m/s

max\_angular = 2.0; // rad/s

max\_distance = 75.0; // pixels;

linear\_speed = Math.sin(nipple.angle.radian) \* max\_linear \* nipple.distance/max\_distance;

angular\_speed = -Math.cos(nipple.angle.radian) \* max\_angular \* nipple.distance/max\_distance;

});

manager.on('end', function () {

if (timer) {

clearInterval(timer);

}

self.move(0, 0);

});

}

Without any endpoint to send the data to however, this joystick won’t do much. This is where we, again, declare some roslibjs objects.

cmd\_vel\_listener = new ROSLIB.Topic({

ros : ros,

name : "/bluetooth\_teleop/cmd\_vel",

messageType : 'geometry\_msgs/Twist'

});

move = function (linear, angular) {

console.log("hoi1");

var twist = new ROSLIB.Message({

linear: {

x: linear,

y: 0,

z: 1

},

angular: {

x: 0,

y: 0,

z: angular

}

});

cmd\_vel\_listener.publish(twist);

These objects make sure that the data ends up in the correct topic. Whilst also declaring and setting how the messages are structured when they are going to said topic.

Lastly, to give the joystick a place on the website, we have to declare it in a div element.

<div id="zone\_joystick"></div>

## Validation

To validate that certain requirements are in place, videos and media are shown below to prove that the functionality of the dashboard is true to its description.

*Some media is still missing due to inaccessibility to the robot; this is discussed with the teacher*

A computer screen shot of a box

Description automatically generated with low confidence

As seen above, this is a screenshot during the development process where we can see both the live map, the live camera feed and the buttons to control some processes.