Web-Ros-CodeSys

Zhicheng Yu
Mahmoud Omar Ouali

1. Introduction

This document is intended to explain the connection between CodeSys-ROS-Web in the SmartWrist. The first chapter reviews the Multithreading server and Codesys communication, the second chapter reviews the connection between Web commands to CodeSys. The last chapter reviews in detail the rosbridge connection between the Web and ROS and the design of the web page.

2. MultiThreading ROS server - CodeSys client

The communication between ROS and Codesys is implemented through the TCP sockets. The implementation of the TCP sockets is chosen based on the research document (ROS and CodeSys communication).

2.1 MutiThreading ROS server

Multiple CodeSys clients needs to be served from ROS side at the same time. To create an illusion of multithreading program threads have being implemented.

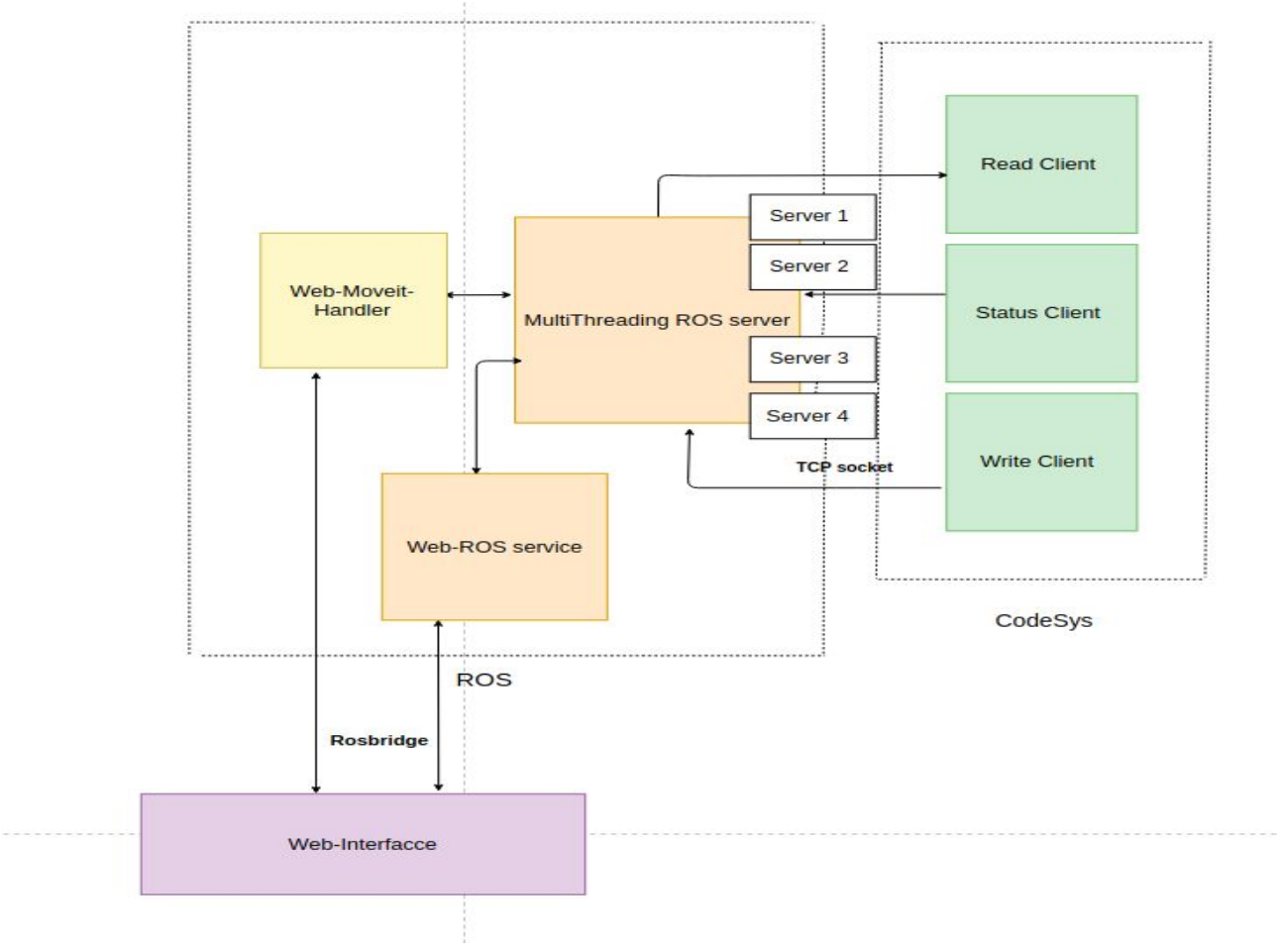

Figure 1 CodeSys-ROS-Web Architecture

Thread is a small sequence of instructions that lives inside the process and can be executed independently by a scheduler[1]. Threads inside the process share the same memory, which makes the inter-communication between the threads fast.

To work with threads in python, the threading module has being used. It provides simple and well documented API to create multiple threads in the program. Creating a thread requires resources from the process, multiple threads in one process means less resources to each thread which can affect the performance of the server. To avoid this problem, the threading pool was limited to four worker threads at a time.

As illustrated in Figure1, there are four servers in the MultiThreading ROS server node, each server represents an independent thread in the thread pool. Server one is listening for new connections coming from the CodeSys

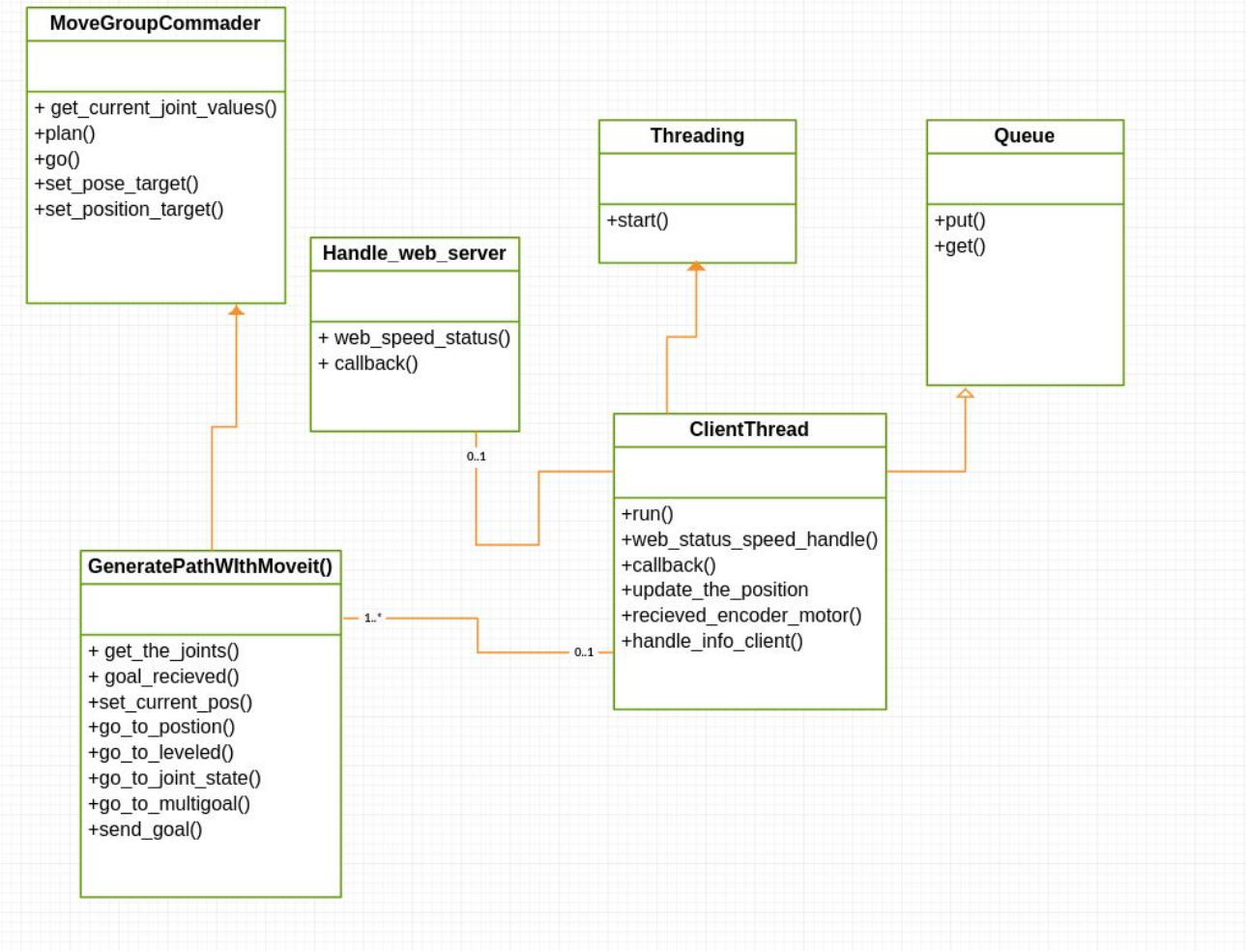

Figure 2 ROS UML diagram

As shown in Figure2, ClientThread class represent the servers, it inherits the threads from the parent class threading and inherits the queues from the Queue module.
start() method in the Threading module starts the threads and puts them in the Threading pool. One thread is given the task to listen to the new tcp socket connections from CodeSys.

When a Codesys Client tries to connect, it's data will be stored in the queue on the server side and one thread will be allocated to serve it. When more than three clients tried to connect to the ROS server, their data will be stored in the queue and wait for one of the threads to be done with the other client.

A protocol was developed for the communication between the clients and the server.

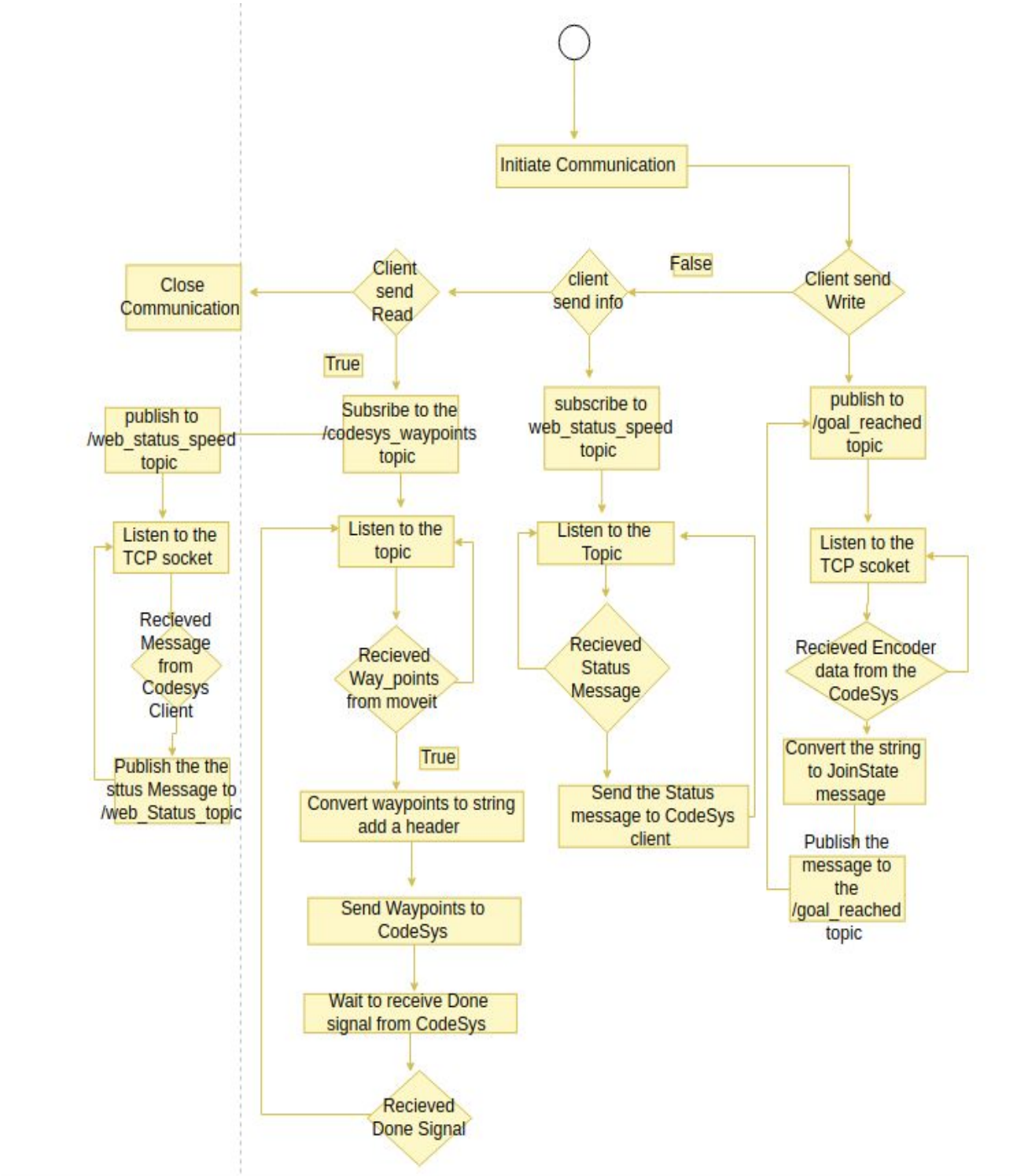

Figure 3 MultiThreaded ROS server flowchart

As shown in figure3, the CodeSys has three TCP clients:

1) **Read client:** To receive waypoints from Moveit, receive status commands (Change speed, Brake, Start, Stop) from the web client
2) **Info client**: Send the status of the SmartWrist(Speed information, Brakes ON/OFF) to the html web client
3) **Write client**: Send the encoder data to the ROS server

For the ROS server to distinguish between the clients, a communication protocol was developed, where each client sends after establishing a TCP communication Read or Write or Info to the ROS server as shown in Figure3.

Done signal is sent from Codesys client to the ROS server, to ensure that SmartWrist has reached the final waypoint. After receiving the Done signal, the ROS server goes to the listening state for new waypoints coming from Moveit.

The encoder data is published to the /goal_reached topic to be visualized in the HTML web client.

3. Web-ROS services

The communication between the web client and ROS is done through rosbridge that will be explained in detail in the next chapter.

The HTML web client commands (update speed, get robot status, start, stop) do not require a continuous data stream, thus services where used instead of topics.

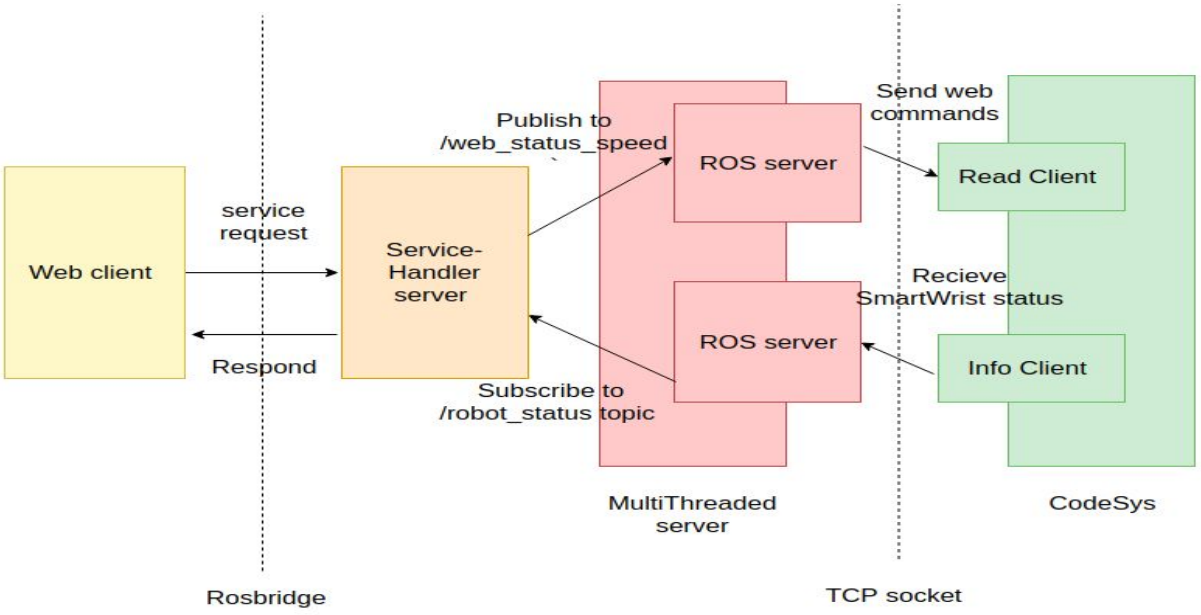

Figure 4 Web-ROS-Codesys

Each service request from the Web client, expects a response (Speed, status, ON, Off) from the SmartWrist to keep the user updated of the robot state.

The service-Handler server handles the web client requests and publish them to /web_status_speed topic in the MultiThreaded server node side. The commands will be sent to the Read client on the codeSys side through TCP socket. After receiving the commands, codeSys updates the web client back of the robot status, to check whether the commands have been successfully executed as shown in Figure4.

4. ROS communication with Webserver

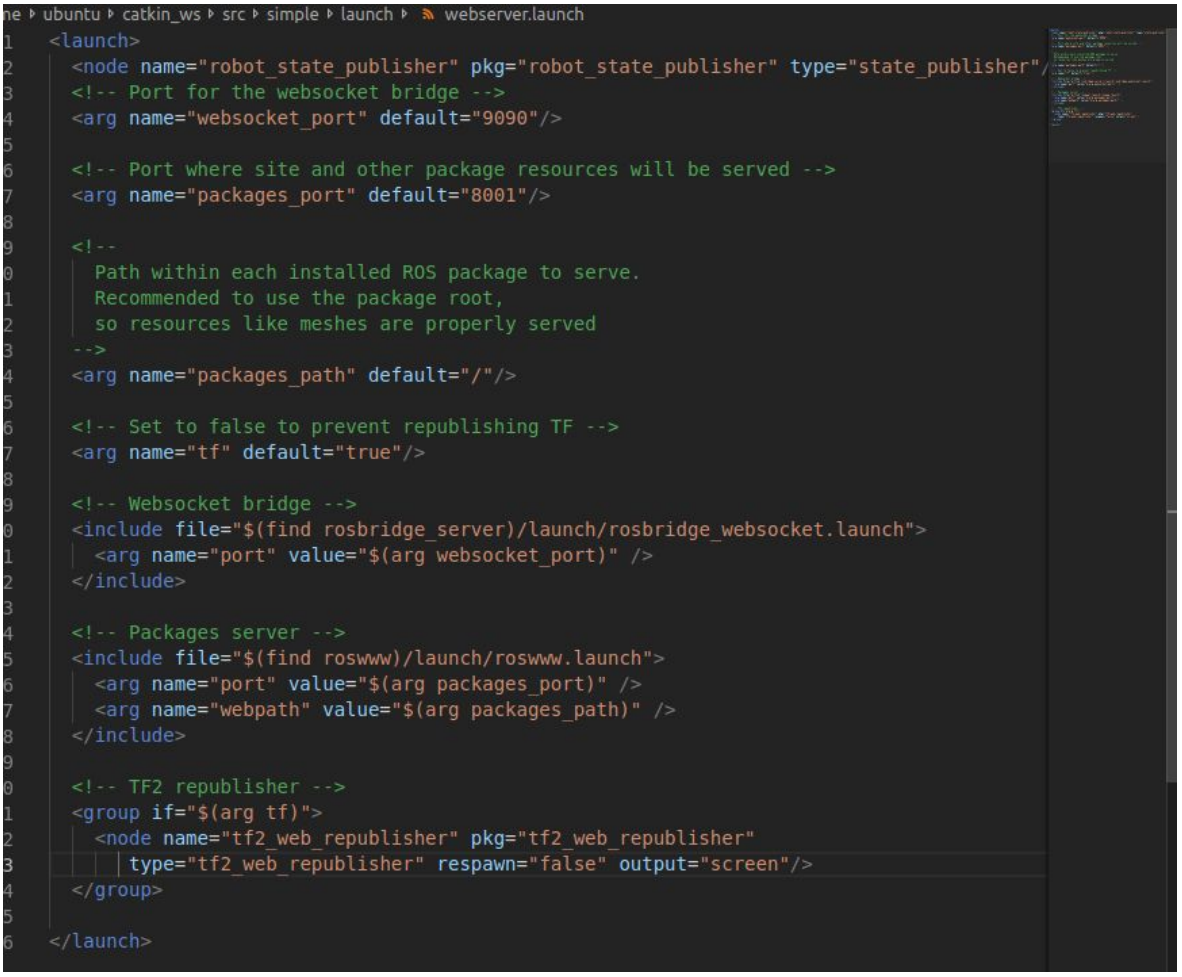

Figure 5 webserver.launch

WebSockets are defined as a two-way communication between the servers and clients, which means both the parties communicate and exchange data at the same time. The key points of WebSockets are true concurrency and optimization of performance, resulting in more responsive and rich web application.

Rosbridge provides a JSON API to ROS functionality for non-ROS programs. There are a variety of front ends that interface with rosbridge, including a WebSocket server of web browsers to interact with.

In the webserver.launch file, the package called robot_state_publisher uses the URDF specified by the parameter robot_description and the joint positions from the topic

joint_states to calculate the forward kinematics of the robot and publish the results via tf. So We use this package to publish the tf transform of our robot arm based on its URDF file.

The tf2_web_republisher is a useful tool for interacting with robots via web browser. The main function of this package is to precompute tf data and send it via rosbridge_server to a ros3djs client. The tf data is essential to visualizing the posture and movement of the robot in a web browser.

3D visualization library for use with the ROS JavaScript libraries:
The ros3djs is the standard JavaScript 3D visualization manager for ROS. It is build on-top of roslibjs and utilizes the power of three.js. Many standard ROS features like interactive markers, URDFs, and maps are included as part of this library.

The Standard ROS JavaScript library:
roslibjs is the core JavaScript library for interacting with ROS from the browser. It uses WebSockets to connect with rosbridge and provides publishing, subscribing, service calls, actionlib, tf, URDF parsing, and other essential ROS functionality.

Figure 6 connect_ros.js

This JavaScript file is used for connecting to ROS by creating a connection to the rosbridge WebSocket server. This JavaScript file also shows those statuses on the web application if the connection is made or the connection has error or the connection is closed.

```
/**
 * Setup all visualization elements when the page is loaded.
 */


function show3DJS() {
  // Connect to ROS.
  var ros = new ROSLIB.Ros({
    url : 'ws://145.93.176.181:9090'
    // url : 'ws://localhost:9090'
  });

  var viewer = new ROS3D.Viewer({
    divID : 'urdf',
    width : 1286,
    height : 1080,
    antialias : true,
    background : '#EEEEEEE',
  });
  // Add a grid.
  viewer.addObject(new ROS3D.Grid({
    cellSize : 0.5,
    color : '#0099cc'
  }));

  // Setup a client to listen to TFs.
  var tfClient = new ROSLIB.TFClient({
    ros : ros,
    angularThres : 0.01,
    transThres : 0.01,
    rate : 10.0
  });

  // Setup the URDF client.
  var urdfClient = new ROS3D.UrdfClient({
    ros : ros,
    tfClient : tfClient,
    path : 'http://145.93.176.181:8001',
    // path : 'http://localhost:8001',
    rootObject : viewer.scene,
    loader : ROS3D.COLLADA_LOADER_2
  });
}
```

Figure 7 show_urdf.js

This JavaScript file creates a ROS node object to communicate with a rosbridge server. Then creates a 3D viewer. We provide the dimensions as well as the HTML div where the viewer will be placed. We add a simple grid to the scene to help visualize the X-Y plane. Then we setup the client to listen to tfs and setup the URDF client.

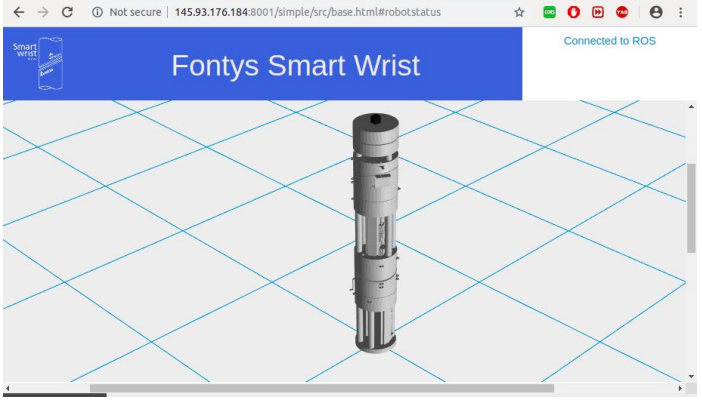

Figure 8 web interface of URDF

From the webserver interface, the rosbridge WebSocket server is connected to ROS and loads the URDF file.

```
<script src="../javascript/navigationbar.js"></script>
<script src="../javascript/jointcontrol.js"></script>
<script src="../javascript/waypointsender.js"></script>
<script src="../javascript/grid.js"></script>
<script src="../javascript/status.js"></script>
```

Figure 9 JavaScript files

Those scripts are including some functions which are used for hardware control by using the web application.

```
<script src="../javascript/jointcontrol.js"></script>
```

Figure 10 jointcontrol.js

The JavaScript file called jointcontrol.js does the joint control.

```
ubuntu ▸ catkin_ws ▸ src ▸ simple ▸ javascript ▸ JS jointcontrol.js ▸
/* This script does the joint control*/

// Create a topic for sending joint angles.
var sendAngles = new ROSLIB.Topic({
  ros : ros,
  name : '/web_goal',
  messageType : 'std_msgs/Float32MultiArray'
})
// Setup a zero message.
var jointGoal = new ROSLIB.Message ({
  data: [0.0, 0.0, 0.0, 0.0, 0.0]
})
```

Figure 11 jointcontrol.js

In this JavaScript file, firstly creating a ROS topic for sending joint angles.

```
firstButton.onclick = function(event) {
  var home = [0, 0, 0, 0, 0]
  setSliders(home);

  function setSliders (joints) {
    rangeslider1.value = joints[0];
    rangeslider2.value = joints[1];
    rangeslider3.value = joints[2];
    rangeslider4.value = joints[3];
    rangeslider5.value = joints[4];
    changeValue1(rangeslider1.value);
    changeValue2(rangeslider2.value);
    changeValue3(rangeslider3.value);
    changeValue4(rangeslider4.value);
    changeValue5(rangeslider5.value);
  }
```

Figure 12 jointcontrol.js

The function for the first button click (home button) will change each joint value to zero. Each range slider can be controlled the joint value individually.

```
secondButton.onclick = function(event) {
  sendTopic();
}
function sendTopic () {
  jointGoal.data[0] = jointValue1;
  jointGoal.data[1] = jointValue2;
  jointGoal.data[2] = jointValue3;
  jointGoal.data[3] = jointValue4;
  jointGoal.data[4] = jointValue5;
  sendAngles.publish(jointGoal);
}
```

Figure 13 jointcontrol.js

The function for the second button click (send button) will send joint angles value to the ROS.

```
thirdButton.onclick = function(event) {
  waitForService('Current');
}
```

Figure 14 jointcontrol.js

The function for the third button click (current button) does a service request to get the current joint values from ROS.

```
fourthButton.onclick = function(event) {
  rangeslider1.value = Math.round(Math.random() * (100)) - 50
  changeValue1(rangeslider1.value)
  rangeslider2.value = Math.round(Math.random() * (100)) - 50
  changeValue2(rangeslider2.value)
  rangeslider3.value = Math.round(Math.random() * (100)) - 50
  changeValue3(rangeslider3.value)
  rangeslider4.value = Math.round(Math.random() * (100)) - 50
  changeValue4(rangeslider4.value)
  rangeslider5.value = Math.round(Math.random() * (100)) - 50
  changeValue5(rangeslider5.value)
}
```

Figure 15 jointcontrol.js

The function for the fourth button click (Random button) will set a random value for each of the range sliders.

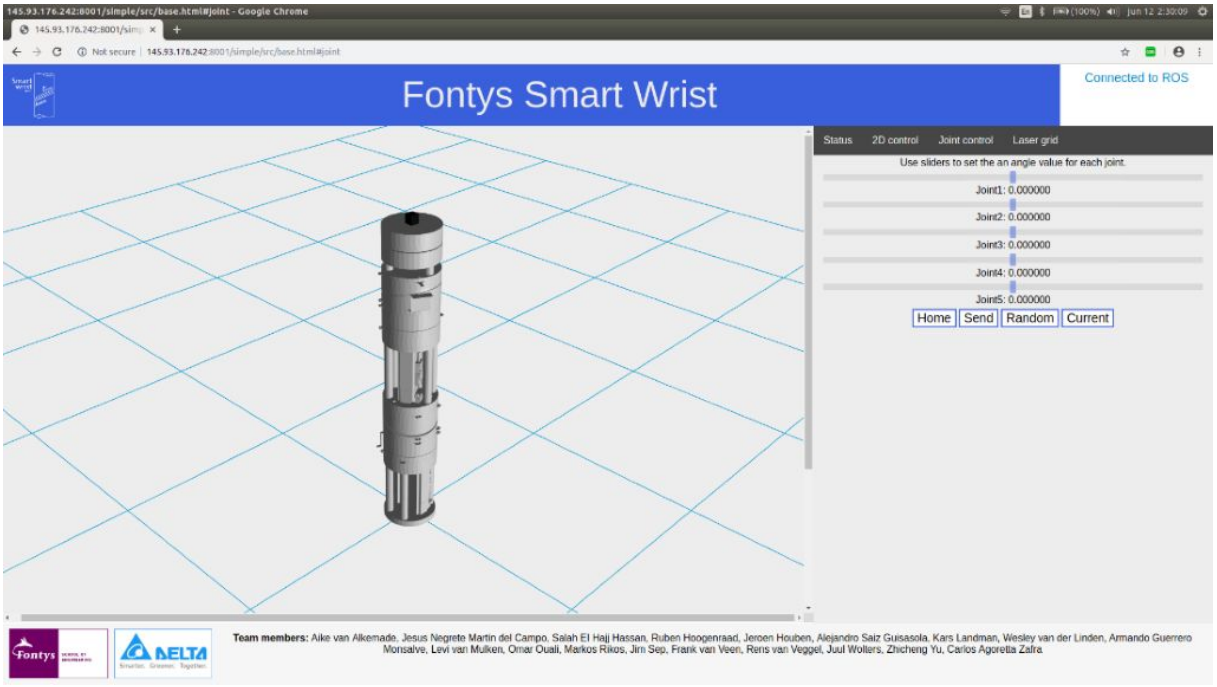

Figure 16 web interface of joint control

The Joint control page shows the Joint control of robot arm. User can use the slide bar to set each joint angle. Then user can click Send button to send the angle value for each joint. User can click the home button to get the home position of robot arm which all the joints will be zero. User can click random button to set the random angle value for each joint. User can click current button to get current angle value for each joint.

```
<!-- three views of robot arm and with send bottom -->
<script src="../javascript/waypointsender.js"></script>
```

Figure 17 waypointsender.js

The JavaScript file called waypointersender.js does 2D control.

```
// create the leveled goal sender ros topic with message type point
var sendPosition = new ROSLIB.Topic({
  ros : ros,
  name : '/web_leveled_goal',
  messageType : 'geometry_msgs/Point'
})

// clarify how message looks like
var xyz = new ROSLIB.Message ({
  x: 0.0,
  y: 0.0,
  z: 1.3229
})
```

Figure 18 waypointsender.js

In this JavaScript file, firstly creating the leveled goal sender ros topic with message type point.

```
//main draw method
function draw() {
  var fontsize = c.width/12;
  //clear canvas
  ctx.clearRect(0, 0, c.width, c.height);
  ctx.beginPath();
  ctx.arc(c.width/2, c.height/2, radius, 0, 2 * Math.PI);
  ctx.stroke();
  ctx2.clearRect(0, 0, c2.width, c2.height);
  sideView();
  circle.draw();
  showCoords();
  ctx.font = fontsize.toString() + "px Arial";
  ctx.fillStyle = "black";
  ctx.fillText("xy view",0,fontsize);
  ctx2.font = fontsize.toString() + "px Arial";
  ctx2.fillStyle = "black";
  ctx2.textAlign = "center";
  ctx2.fillText("xz view",c2.width/2,fontsize*1.5);
  ctx2.fillText("yz view",c2.width/2,fontsize*1.5 + c2.height/2);
};
```

Figure 19 waypointsender.js

The main function draw method makes the three views of robot arm. The xy view is a top view of the robot arm. The xz and yz views are the side view of the robot arm.

```
/* Draw the xz and yz viewing arcs*/
function sideView() {
  ctx2.beginPath();
  ctx2.lineWidth = 1;
  ctx2.strokeStyle = "black";
  var xx = 0, x0 = 0, zz, z0 =c2.height / 2;
  for (var j=0; j<2; j++) {
      for (var i=-1; i<=1.05; i+=0.05) {
          zz = (Math.pow(i,2) * 0.266 - 0.27)*c2.height;
          if (i==-1) {
              ctx2.moveTo(x0+xx, z0+zz);
          }
          else {
              ctx2.lineTo(x0+xx, z0+zz);
          }
          xx+=0.05*c2.width/2;
      }
      ctx2.moveTo(x0, z0);
      ctx2.lineTo(x0+c2.width, z0);
      xx = 0, z0 = c2.height;
  }
  ctx2.stroke();
}
```

Figure 20 waypointsender.js

The function sideview draws two figures for two side views (xz and yz views) of robot arm

```
function Circle(x, y, z, theta, r, fill, stroke) {
  this.startingAngle = 0;
  this.endAngle = 2 * Math.PI;
  this.x = x;
  this.y = y;
  this.z = z;
  this.theta = theta;
  this.r = r;
  this.fill = fill;
  this.stroke = stroke;
  this.draw = function () {
      ctx.beginPath();
      ctx.arc(this.x, this.y, this.r, this.startingAngle, this.endAngle);
      ctx.fillStyle = this.fill;
      ctx.lineWidth = 1;
      ctx.fill();
      ctx.strokeStyle = this.stroke;
      ctx.stroke();
      drawOtherCircles();
  }
}
```

Figure 21 waypointsender.js

The function Circle creates the circle object. If Circle.draw is called it draws the circle and starts the function drawOtherCIrcles.

```
// creates an element of the send button
var secondButton = document.getElementById('buttonSend2D')

// if clicked, publisch the the xyz to the position publisher
secondButton.onclick = function(event) {
    xyz.x = calcX;
    xyz.y = calcY;
    xyz.z = calcZ;
    sendPosition.publish(xyz)
}
// draw the view on page load
  draw();
```

Figure 22 waypointsender.js

The button click function is sending the x, y, z position.

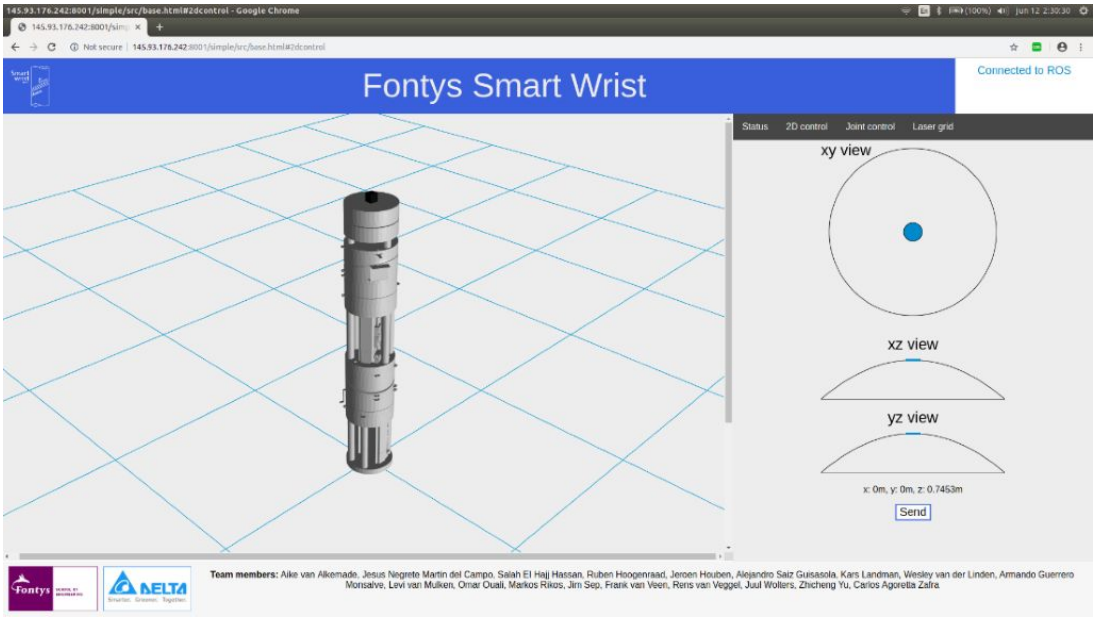

Figure 23 web interface of 2D control

The 2D control page shows the 2D control of robot arm. User can move the position of blue point which is moving the top view (x, y view) of robot arm. User can also check the x, y, z position. Then User can click the send button to send the x, y , z value to ROS.

```
<script src="../javascript/grid.js"></script>
```

Figure 24 grid.js

The JavaScript file called grid.js does add waypoint on the grid.

```
function drawGrid() {
  ctx3.beginPath();
  ctx3.lineWidth = 1;
  ctx3.strokeStyle = "black";
  for (var left = 1; left <= columns+ 1; left++) {
    ctx3.moveTo(left*stepLeft, stepTop);
    ctx3.lineTo(left*stepLeft, c3.height - stepTop);
  };
  for (var down = 1; down <= rows +1 ; down++) {
    ctx3.moveTo(stepLeft, down*stepTop);
    ctx3.lineTo(c3.width - stepLeft, down*stepTop);
  };
  ctx3.stroke();
};
```

Figure 25 grid.js

The function drawGrid draws the grid.

```
function addWaypoint() {
  setMousePosition(event);
  drawCanvas();

  if (mousePos.x > stepLeft && mousePos.x < c3.width - stepLeft
    && mousePos.y > stepTop && mousePos.y < c3.height - stepTop) {
    colourBlock();
  };

};
```

Figure 26 grid.js

The function addwaypoint finds the position of click on the grid. Then check the position on the grid. If true, then start the function colourblock to show it was pressed.

```
function calculateTheta () {
  var x = block.x;
  var y = block.y;
  var xreal = -(5.5-x)*cellwidth;
  var yreal = (5.5-y)*cellwidth;
  var distancefromcenter = Math.sqrt(Math.pow(xreal,2)+Math.pow(yreal,2));
  var theta2 = 2*Math.atan(distancefromcenter/length);
  var theta1;
  if (xreal > 0 && yreal >= 0) {
    theta1 = Math.atan(yreal/xreal) - ((Math.PI-theta2)/2);
  }
  else if (xreal < 0 && yreal >= 0) {
    theta1 = Math.atan(yreal/xreal) - ((Math.PI-theta2)/2) + Math.PI;
  }
  else if (xreal < 0 && yreal < 0) {
    theta1 = Math.atan(yreal/xreal) - ((Math.PI-theta2)/2) + Math.PI;
  }
  else if (xreal > 0 && yreal < 0) {
    theta1 = Math.atan(yreal/xreal) - ((Math.PI-theta2)/2);
  };
  if (theta1 > Math.PI) {
    theta1 = theta1 - 2 * Math.PI;
  };
  sendLaserTopic(theta1,theta2);
}
function sendLaserTopic (theta1,theta2) {
  jointGoal.data[0] = 0
  jointGoal.data[1] = 0
  jointGoal.data[2] = theta1
  jointGoal.data[3] = theta2
  jointGoal.data[4] = 0
  sendAngles.publish(jointGoal)
}
function clearGrid() {
```

Figure 27 grid.js

The function calculateTheta calculates the joint angles for one wrist. The function sendLaserTopic publishes the joint angles.

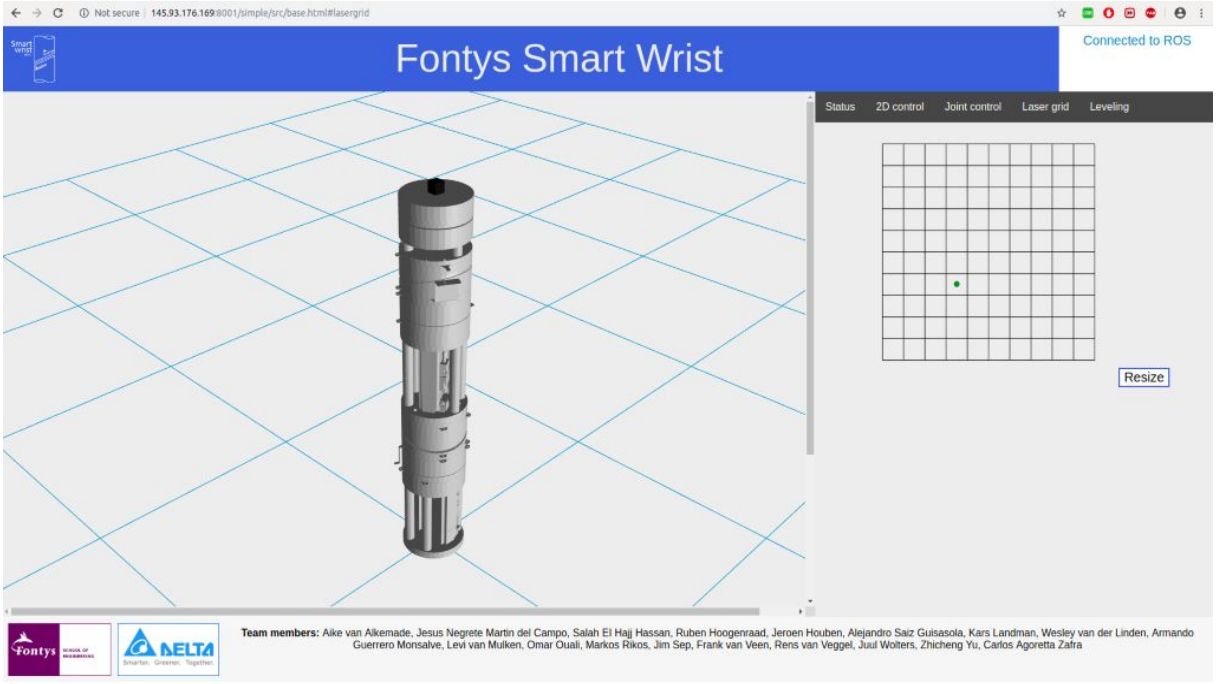

Figure 28 web interface of Laser grid

The laser grid page shows the laser grid area. There is a laser will be mounted on the top of the robot arm. User can select a waypoint on the grid then the arm moves and point to that position on the grid area. The same laser grid will be set up in front of the arm. In that way the user can set a goal for the laser to go to and see the laser beam moves to the same position on the physical grid. On the poster with grid there is actually a finer grid within the grid. This way the user can see how small the orientation error of the smart wrist really is.

```
<script src="../javascript/status.js"></script>
```

Figure 29status.js

The JavaScript file called status.js does show the status of robot arm.

```
var robotStatusClient = new ROSLIB.Service({
  ros : ros,
  name : 'web_status_speed_srv',
  serviceType : 'smart_movement/RequestStatus'
});

// Then we create a Service Request.
var servicerequest = new ROSLIB.ServiceRequest({
  statusreq: "req:[1]"
});
```

Figure 30 status.js

In the JavaScript file, firstly we create a service client with details of the service's name and service type. Then we create a Service request.

```
function waitForService (servicemessage) {
  // Put the String in the Request
  servicerequest.statusreq = servicemessage;
  // Now it's time to call the service, with the result as the responce from the S
  robotStatusClient.callService(servicerequest, function(result) {
    // Look if the speedmode was send from CODESYS and displays it in the status w
    if (result.statusrep.includes("Speed")) {
      var speedpos = result.statusrep.indexOf("Speed");
      var speedsetting = result.statusrep[speedpos+7];
      var speedname;
      if (speedsetting == "3") {
        speedname = "Swift"
      }
      else if (speedsetting == "2") {
        speedname = "Moderate"
      }
      else if (speedsetting == "1") {
        speedname = "Debug"
      }
      else {
        speedname = "Error"
      }
      document.getElementById("speedMode").innerHTML = "Speed mode: " + speedname;
    };
```

Figure 31 status.js

The function check if the speed mode was sent from CodeSys then displays it in the status window.

```
// Status Window.
if (result.statusrep.includes("Motor")) {
  var motorpos = result.statusrep.indexOf("Motor");
  var motor = result.statusrep.substr(motorpos+6,9);
  document.getElementById("motorSetting").innerHTML = "Motors: " + motor;
};
```

Figure 32 status.js

The function check if the status of the motors was sent from CodeSys then displays it in the status window.

```
// Status Window. Acts says "Pushed!" if an emergency button error occurs.
if (result.statusrep.includes("Emergency")) {
  var emergencypos = result.statusrep.indexOf("Emergency");
  var emergency;
  if (result.statusrep[emergencypos+11] == "1"
  || result.statusrep[emergencypos+13] == "1") {
    emergency = "Pushed!";
  }
  else {
    emergency = "Operational";
  }
  document.getElementById("emergency").innerHTML = "Emergency button: " + emergency;
};
```

Figure 33 status.js

The function check if the emergency button status was sent from CodeSys and displays it in the status window. Also shows “Pushed!” if an emergency button error occurs.

```
// window. Also shows that the connection to CODESYS was established.
if (result.statusrep.includes("Start")) {
  var startpos = result.statusrep.indexOf("Start");
  var start
  if (result.statusrep[startpos+7] == "1") {
    start = "Ready for new command";
  }
  else {
    start = "Halted";
  }
  document.getElementById("systemStart").innerHTML = "Movement: " + start;
  document.getElementById("codesysCon").innerHTML = "CODESYS connection: Established"
};
```

Figure 34 status.js

The function check if the Smart Wrist status was sent from CodeSys and displays it in the status window.

```
// window. Also shows that the connection to CODESYS was established.
if (result.statusrep.includes("Start")) {
  var startpos = result.statusrep.indexOf("Start");
  var start
  if (result.statusrep[startpos+7] == "1") {
    start = "Ready for new command";
  }
  else {
    start = "Halted";
  }
  document.getElementById("systemStart").innerHTML = "Movement: " + start;
  document.getElementById("codesysCon").innerHTML = "CODESYS connection: Establish
};
```

Figure 35 status.js

The function check if the Smart Wrist status was sent from CodeSys and displays it in the status window. Also shows that the connection to CodeSys was established.

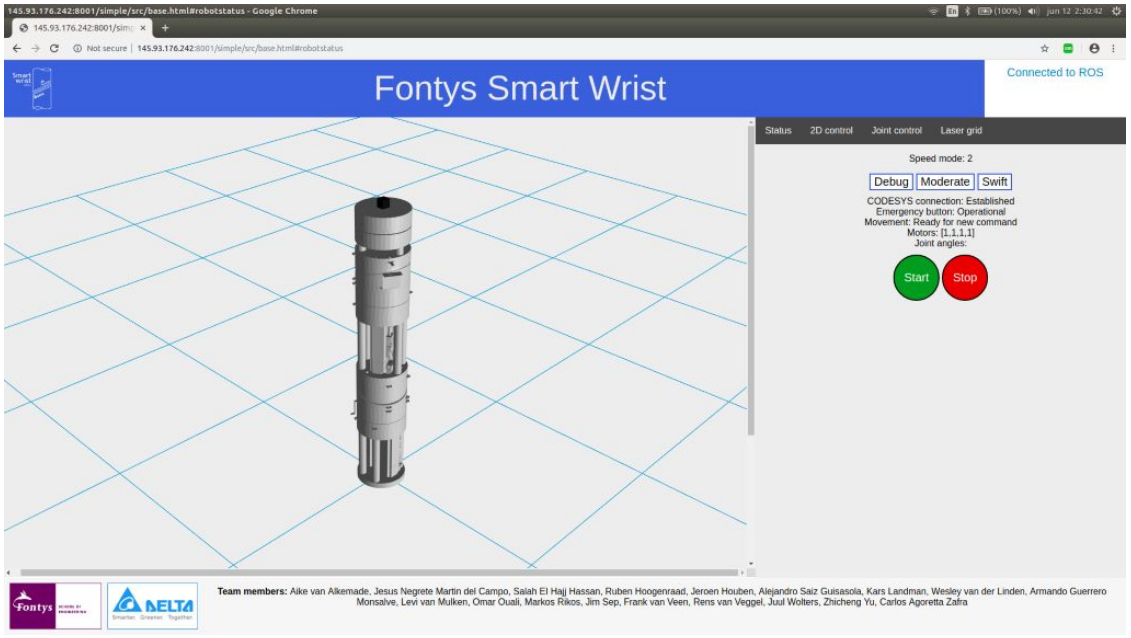

Figure 36 web interface of Status

The Status page shows the status of the robot arm. In the status page user can select three different speed mode to change the movement speed of the robot arm. User can check the connection of ROS and CodeSys is established or not. Then user can check the Emergency button is operational or not. User can check if the movement is ready for receiving new joint goal. User can check each motor is ON or not. User also can click the start button to release the break of the motor and click the stop button to stop the motor.

```
<!-- orientation value and with send button -->
<script src="../javascript/leveling.js"></script>
```

Figure 37 leveling.js

The Javascript file called leveling.js shows the orientation value and with the send button.

```
var sendQuatTopic = new ROSLIB.Topic({
ros : ros,
name : '/web_orientation_goal',
messageType : 'geometry_msgs/Quaternion'
});

// Make a object that holds the quaternion message
var quaternionMessage = new ROSLIB.Message({
x: 0.0,
y: 0.0,
z: 0.0,
w: 0.0
});
```

Figure 38 leveling.js

This Javascript file creates a Ros topic with that name '/web_orientation_goal' and message type Quaternion from geometry_msgs. Then makes an object that holds the quaternion message.

```
var imu2Sensor = new ROSLIB.Topic({
ros : ros,
name : '/tfsensors/imu1',
messageType : 'sensor_msgs/Imu'
});
```

Figure 39 leveling.js

Setup the object that has the name and message type.

```
// Subscribe to to the imu sensor
imu2Sensor.subscribe(function(message) {
    // Store the xyzw in the xyzw list. The z and x values are swapped to correct
    // the mistake in the topic
    xyzw[0] = message.orientation.z;
    xyzw[1] = message.orientation.y;
    xyzw[2] = message.orientation.x;
    xyzw[3] = message.orientation.w;
    // Show the in quaternion values in Strings of 6 decimals
    document.getElementById("x").innerHTML = "x: " + x.toFixed(6).toString();
    document.getElementById("y").innerHTML = "y: " + y.toFixed(6).toString();
    document.getElementById("z").innerHTML = "z: " + z.toFixed(6).toString();
    document.getElementById("w").innerHTML = "w: " + w.toFixed(6).toString();
});
```

Figure 40 leveling.js

The function subscribes to the imu sensor and stores the xyzw value in the xyzw list

```
var sendQuatButton = document.getElementById("sendQuatButton");

// If the send button is pressed publish the inverse of the quater
// MoveIt!
sendQuatButton.onclick = function () {
    quaternionMessage.x = xyzw[0];
    quaternionMessage.y = xyzw[1];
    quaternionMessage.z = xyzw[2];
    quaternionMessage.w = -xyzw[3];
    sendQuatTopic.publish(quaternionMessage);
};
```

Figure 41 leveling.js

The send click button function publishes the inverse of the quaternion as goal to Moveit!

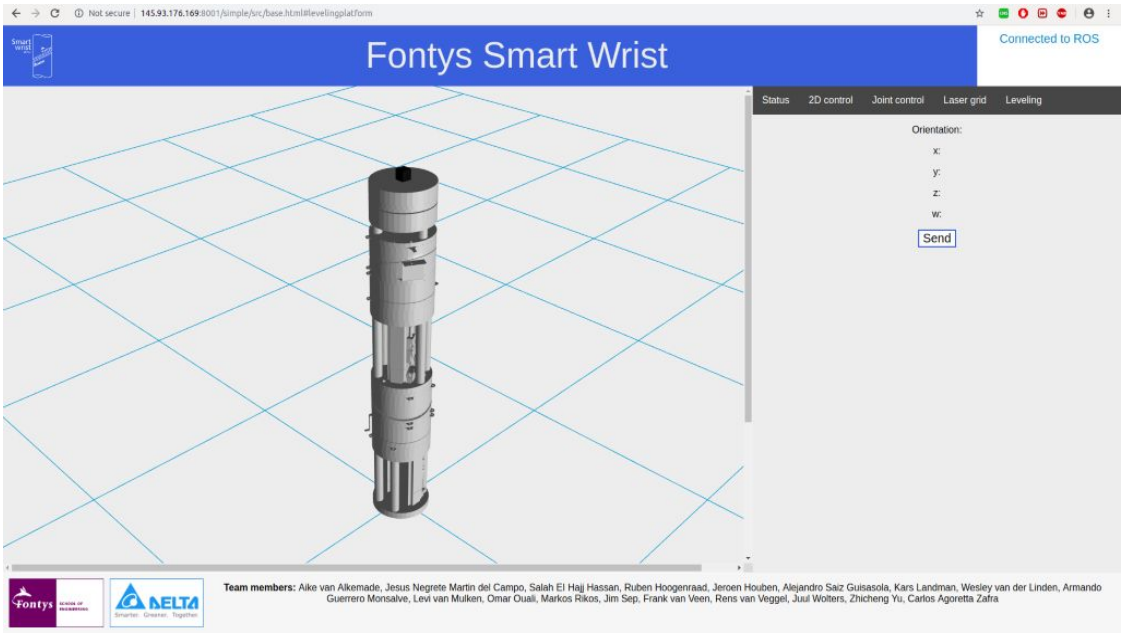

Figure 42 web interface of leveling page

The leveling page shows the leveling position. There is a leveling platform at the bottom of the arm will be rotated to an orientation by the operator, after that the robot arm should measure this orientation and move so that the end platform will be perfectly leveled. There is a sensor called IMU brick 2 with be mounted to the platform. User can see the orientation value (x, y, z and w) from the sensor in the web server. Then user can click send button to make the robot arm moves to the leveled position.

5. References

1. ***Lamport, Leslie (September 1979). "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs" (PDF). IEEE Transactions on Computers. C-28 (9): 690–691. doi:10.1109/tc.1979.1675439.***