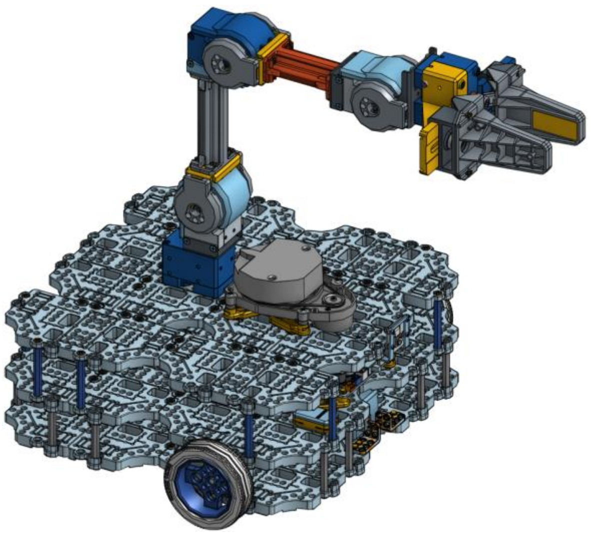


BASE WALL CLEANER

Technical report

GROUP

Project group 2

Josh Bindels
Xiangkuan Peng
Darryl Auguste
Sven Hermsen
Stan Vergeldt
Josh Bindels 3233642

Table of Contents

Introduction

The Base Wall Cleaner Robot is a robot that autonomously navigates along the walls of a room while cleaning the wall base boards. The robot runs on a ROS platform and uses a web application to interface with users. This technical document will cover both the ROS application as the web application in terms of requirements and implementation details. As well as discussing possibilities for future work.

The project team consists of:

- Xiangkuan Peng
- Darryl Auguste
- Josh Bindels
- Sven Hermsen
- Stan Vergeldt

Architecture

Application architecture

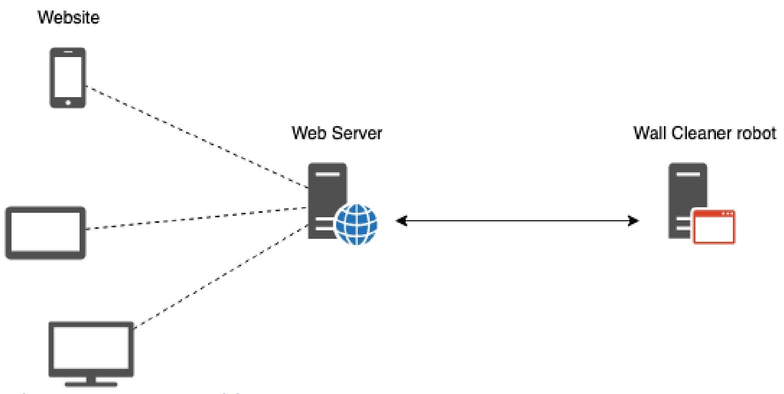


Figure 1 System architecture

The system is designed to run on two machines:

1. The web server
2. The robot

The web server runs a Python Flask Web Application. The website allows a user to control the robot by telling it to go or stop, as well as receiving information from the robot such as status messages and photographs.
The robot is responsible for running the ROS application. This allows it to move through a room autonomously. To allow for communication between the robot and the web server the robot also runs an instance of RosBridge this enables the use of Topics over the socket connection.

Code base architecture

The code base is available on git at https://github.com/fontysrobotics/ARMinor-2020-Base-Wall-Cleaner-Robot .

The code has been split into 3 separate directories:
1. base_wall_ws
2. scripts
3. web_application

The base_wall_ws directory contains all code related to the ROS Application including the turtlebot3 source. The source for the ROS application can be found at base_wall_ws/src/turtlebot3_simulations/turtlebot3_gazebo/src. The entire ROS application was written in Python.

The scripts directory contains a number of bash scripts. These include scripts to setup or install requirements, as well as scripts to run all applications (Web and ROS). A full explanation of these scripts can be found in the manual.

The web_application directory contains all code related to the web application. The web application consists of Python, HTML, JavaScript and CSS.

Web Application

Requirements

The web application is built on Python 2.7 and uses the Flask micro web framework. It uses the pip package-management system, all required packages can be found in requirements.txt.

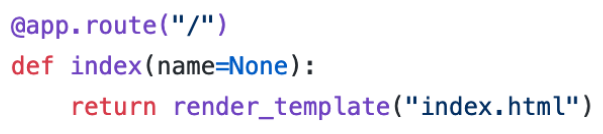
Implementation

The web application can be split into three parts:

1. python flask application.
2. ROSLIB JavaScript application.
3. The frontend HTML.

1. Flask application

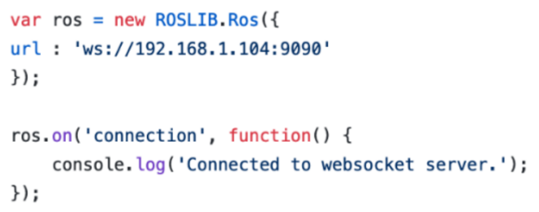
The flask application serves the index.html file which is the entry point of the application. Only one route is available "/" this means that if the user browses to http://<ip>:<port> they will automatically be send to index.html.



```
@app.route("/")
def index(name=None):
    return render_template("index.html")
```

2. JavaScript

The JavaScript files loaded in the index.html file contain the functions to communicate with the ROS instance.
Main.js connects to the RosBridge web socket.



```
var ros = new ROSLIB.Ros({
url : 'ws://192.168.1.104:9090'
});

ros.on('connection', function() {
    console.log('Connected to websocket server.');
});
```

The URL IP address and port number should be changed according to the ROS machine's IP and port.

The website makes use of three ROS Topics:

1. **Commands:** Used for sending commands like Start and Stop to the robot.
2. **Photographer**: Used for receiving images from the robot's camera.
3. **Status:** Used for receiving status messages from the robot.

Topics that send a message follow the format as can be seen in Figure XXX. Where a ROSLIB Topic object is created and a callback method is assigned by using the subscribe method.

```
var status_listener = new ROSLIB.Topic(
{
        ros: ros,
        name: "status",
        messageType: "std_msgs/String"
});

status_listener.subscribe(function(message)
{
        document.getElementById("status_msg").innerText = message.data;
});
```

Topics that receive messages follow the format as can be seen in Figure XXX. data over the Topic.

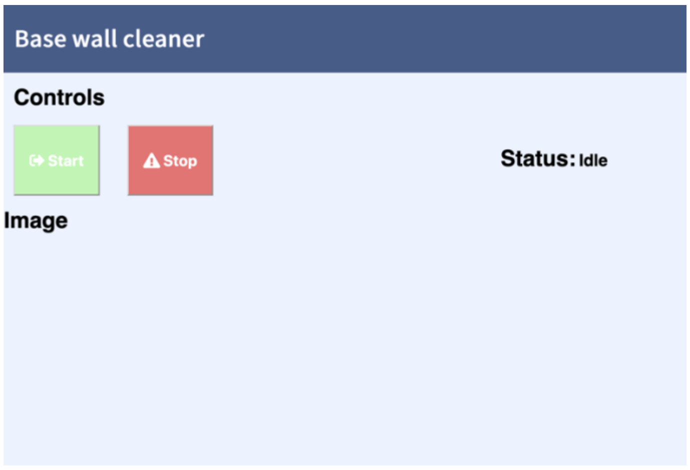
```
var cmdChatter = new ROSLIB.Topic({
    ros : ros,
    name : '/commands',
    messageType : 'std_msgs/String'
});

function SendMessageStart()
{
    cmdChatter.publish(chatter_messages[0]);
}
```

3. HTML

A view of the website can be seen in Figure XXX. The website can be split in to three sections:

1. Controls: Where a user can send commands to the robot.
2. Status: Where the current status of the robot is displayed.
3. Image: Where an image is displayed when the robot has gotten stuck on something.



ROS Application

Requirements

The ROS Application is a robot simulation program built on python 2.7 that uses the Turtlebot3 as the simulated robot (included in git) and gazebo for the simulation world.

Implementation

The ROS application can be divided into 3 parts:

1. Photographer
2. WebController
3. Cleaning robot

1. Photographer

This class is responsible for capturing image using the camera on the Turtlebot3. It is a publisher of a user defined topic “photographer” and can send CompressedImage to the topic.

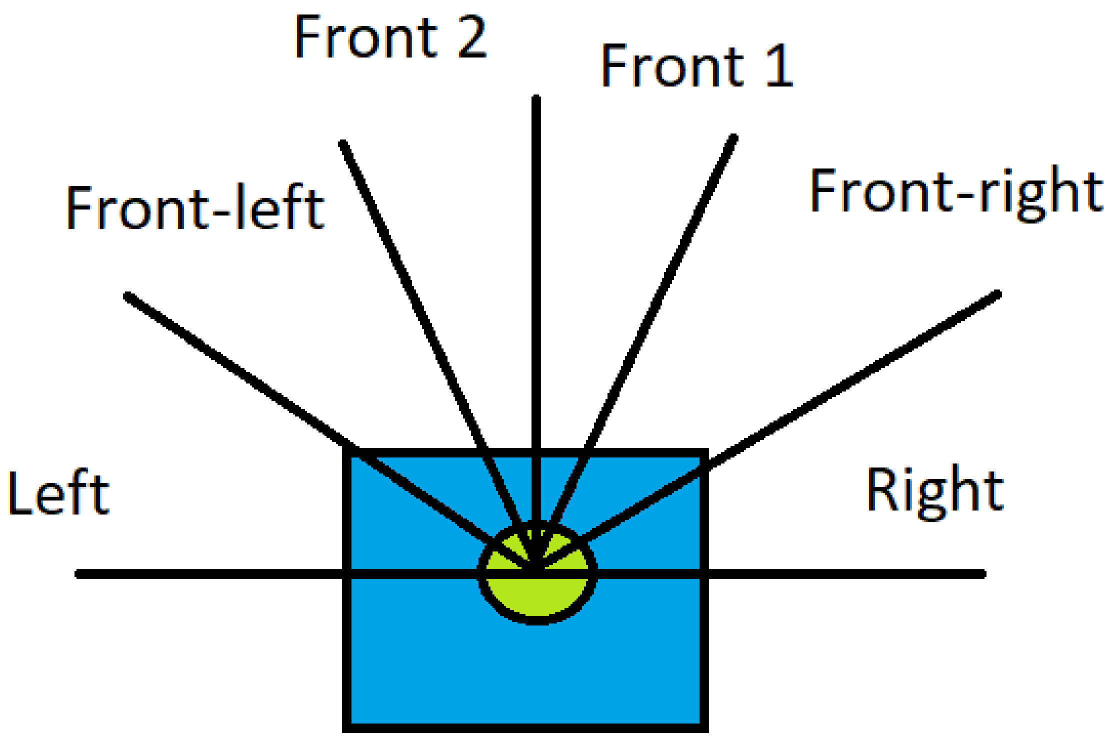
2. WebController

WebController class handles the communication between the Web UI and the Cleaning robot. It was used to receive string type commands from the Web UI to the robot and transmitting various status message back to the Web UI.

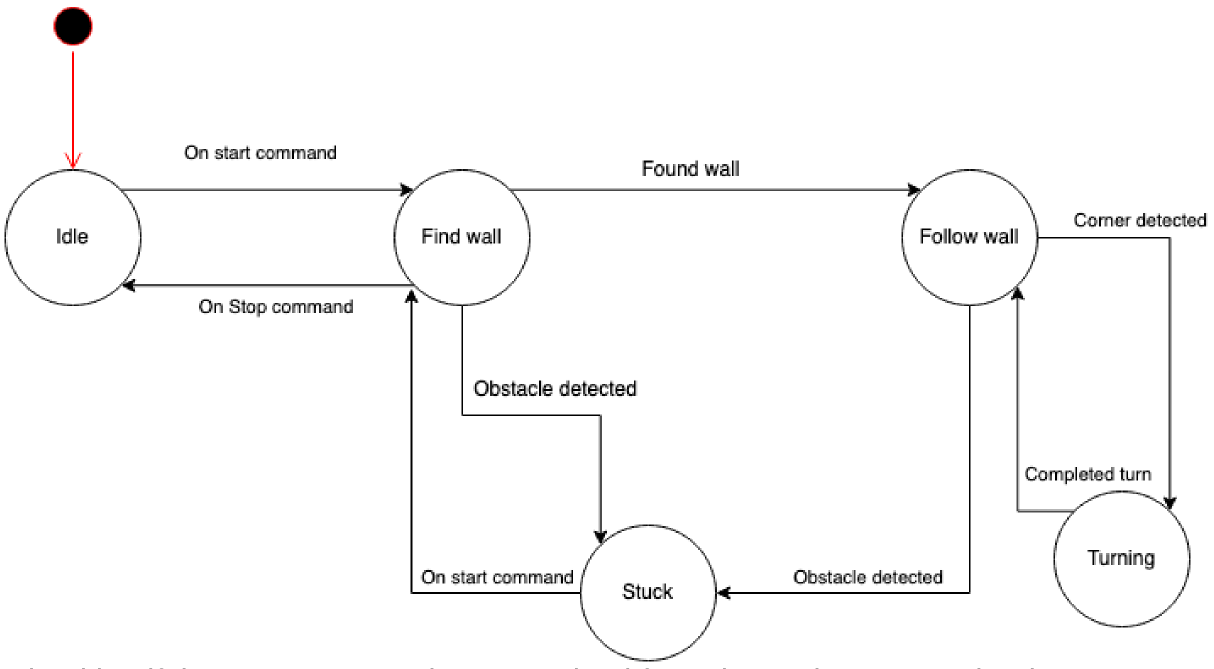
3. Cleaning robot

A class was created for the cleaning robot, it operates like a state machine and has a Photographer object for capturing image and WebController for communication with the web UI. The cleaning robot is configured to follow wall on the right side by default and can be configured to follow wall on left side.

The cleaning robot subscribes to the “/cmd_vel” topic to control the robot’s linear and angular speed and “/scan” topic to receive the lidar sensors value.180 degrees out of the Lidar sensor’s 360 degrees detection zone was used. Then these 180 degrees was divided into 6 sections: right (36 degrees), front-right (36 degrees), front1 (18 degrees), front2 (18 degrees), front-left (36 degrees), left (36 degrees). Each zone, a distance value was stored which represents the distance from the closest obstacle in that zone to the lidar sensor.



The cleaning robot has a state machine to perform different functions in different states.



Checking if the stop command was received from the Web UI was prior than any state execution. When the stop command was received, the state of the robot will be changed to idle.

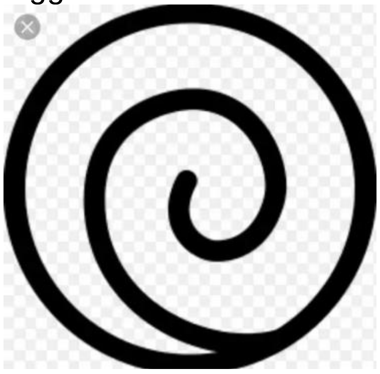
Idle state

After initialization, the cleaning robot will stay in Idle state until it received a “start” commands using its WebController object and then change to Find wall state

Find wall
In this state, it will firstly determine whether change state to “Stuck” or not depending on the stuck detection function, the stuck detection was achieved by checking the minimum value of among all 6-detection zone and see if the value is smaller than the setpoint value (currently 0.20 meter) and will report the robot is stuck if the minimum distance is smaller than the setpoint after a configurable time period.

After that, it will check if the wall is found and change to Follow wall state if it is. To determine whether the wall is found or not, it will use minimum value in the detection zones depending on which side the robot was configured to follow.

If above conditions are not met, then the cleaning robot will start searching walls in a vortex-searching pattern. This searching movement can be configured to cover bigger area.



Follow wall
Just like Find wall state, this state also checks if the robot is stuck to determine whether continue the state or change to Stuck state.

After that, if the robot is not stuck it will perform a check on whether the robot reached a corner and will simply turn left or right depending on the which side of the wall the robot was configured to follow. The corner check is achieved by checking if the front and left or right sides of the robot has obstacle detected and the distance are smaller than the setpoint.

Lastly, when above conditions were not met, the robot will simply performing wall following movement which was using a PID controller to calculate angular and linear speed using the minimum value from the detection zone depending on the configuration of the wall follow side.

Stuck
When the robot is in the stuck state, it will stay in this state until the start command was received again from the Web UI.

Turning
Simply turn left or right depending on the which side of the wall the robot was configured to follow