Second Interim Report
Gazebo simulations of contract-based design for an automated valet parking system

SURF student: Tom Andersson
Mentor: Richard M. Murray
Co-mentor: Josefine Graebener

July 31, 2020

1 Introduction

1.1 Background

The increased complexity in cyber-physical systems calls for a modular system design that guarantees safe, correct and reliable behavior, both in normal operation, in the presence of dynamic scenario changes and in failure scenarios [5]. One way of achieving a modular system is to use contract-based design and split up the system into layers, where each layer have specific tasks. Information is sent between the layers according to the rules set up by the contracts. This system architecture not only simplifies the system by splitting it up into several subsystems but in a failure scenario this design also makes it clear in which layer the failure occurred. Other layers can then react and adapt to the new restrictions the failures entail.

These advantages match well with several problems within the development of autonomous vehicles since they have to be developed for an environment constantly affected by dynamic scenario changes due to the interaction with other cars, pedestrians and so on. A first step towards full autonomy can be to limit the scope and focus on an automated valet parking system, a system where customers drop of their cars and an autonomous robot picks them up one at a time and drives them to a free parking spot. The research group at Caltech has developed python simulations of an automated valet parking system, modularly designed using contracts between the system layers. The simulations demonstrate the ability to control several cars that can be parked and recalled within two minutes in the absence of failure scenarios and in the presence of a simulated undesired stop of a car in the system. The project is stored in this Github repository [3]. Figure 1 shows the layout of the valet parking in the simulations [5].

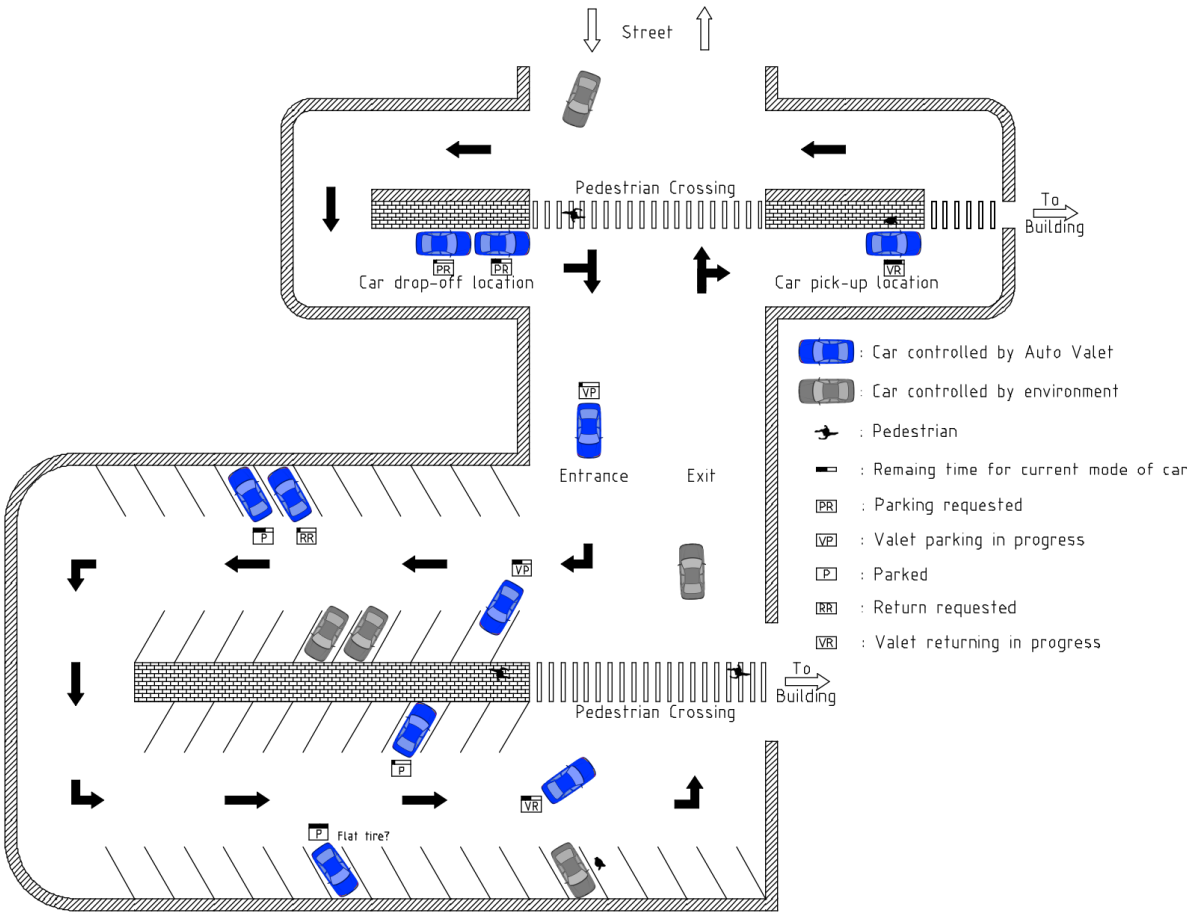

Figure 1: The layout of the valet parking in the python simulations [5].

1.2 Goals

The goals of this project are to prepare the python simulations for a small scale test that would incorporate real life aspects and the possibility to physically separate components in the system by converting the code into ROS-nodes and performing simulations of 6 real robots in the Gazebo simulator. If this is done on time, the system will also be further developed by implementing the functionality to under certain conditions lighten the specifications when a car is unable to achieve its goal. For example, when the route of a car is blocked but there are no other cars nearby the car can be allowed to track the path more loosely, if the car cannot reach its assigned parking spot but there are plenty of other free spots it can be allowed to pick another one or, as a last resort when a goal cannot be achieved, the car can instead focus on not being in the way for any other car in the system until the blocking is cleared.

1.3 Initial objectives

- Become familiar with the work so far done by the research group. (1 week)
- Learn about Gazebo and choose appropriate robots to simulate. Perform simulations by using commands to set speed and steering angle. (2 weeks)
- Build a model of the valet parking environment in Gazebo. (1 week)
- Adapt the simulations of the camera to detect and distinguish walls, lanes, pedestrian crossings, moving pedestrians, parking spots and moving cars in Gazebo. (1 weeks)
- Adapt the other programs into ROS-nodes and perform the simulations on the Gazebo model. (2 weeks)
- If the project is on time at this point, implement the functionality to under certain conditions lighten the specifications when a car is unable to achieve its goal. This shall be done for three different cases, allowing the car to choose another parking spot if there are plenty of available spots, allowing the car to deviate from its specified route if there are no other cars nearby and, as a last resort, commanding the car to keep lanes and parking spots sought by other cars free until the blocking is cleared. (4 weeks)
- Finalize a written report. (1 week)

1.4 Redefined objectives

Once the three first bullet points in section 1.3 were completed it had become apparent that the initial objectives were too optimistic. The time for implementing static obstacles and pedestrians, included in the third bullet point, had been underestimated. Additionally, at this point the project and the knowledge of the project had also evolved since when the initial objectives were written. This called for a redefinition of the objectives.

To solve this issue, it was decided not to adapt the simulations of the camera. The pure python simulations doesn't use this feature so not having this in the gazebo simulations will not be a limitation compared to the pure python simulations. Adapting the other programs into ROS-nodes is something that still made sense at the current state of the project but it had become apparent that that was only a fraction of what had to be done to integrate the Gazebo simulations with the main program so it was chosen to put a larger focus on the integration itself. The objective regarding lightening specifications did not make the most sense to include during this state of the project. Some of the ideas had already been implemented into the system and the time duration of that bullet point had to be lowered. A feature that was considered more relevant to the project was to instead, if time would permit, implement a delayed failure where the system gets a notification that a failure is about to occur within an estimated distance. The system will then have to determine how far it can keep going and where the safest and most efficient way to stop within that distance is. The reasoning behind this is that it is normal that a car doesn't come to a complete stop immediately once a failure occurs but gets some time to, for example, drive to the side of the road, drive into a parking spot or drive to a less busy area depending on the severeness of the failure.

In conclusion, the redefined objectives are the following.

- Become familiar with the work so far done by the research group. (1 week)
- Learn about Gazebo and choose appropriate robots to simulate. Perform simulations by using commands to set velocities and read velocities and position. (2 weeks)
- Build a model of the valet parking environment in Gazebo including a ground plane of the parking lot, walls, the robots, traceable static obstacles and controllable and traceable pedestrians. (3 weeks)
- Implement the functionality to dynamically launch a chosen number of robots, pedestrians and obstacles at chosen initial positions in the Gazebo world. (1 week)
- Integrate the Gazebo simulations with the main AVP simulations. (4 weeks in parallel with the next bullet point)
- If time permits, implement the delayed failure feature described earlier in this section. (4 weeks in parallel with the last bullet point)
- Finalize a written report. (1 week)

The fifth and sixth bullet points will be done simultaneously, making a total time for the project of 12 weeks.

2 Progress

This section describes the progress of the system so far. More detailed information on each script and function can be found in the corresponding docstrings. The source code can be found on the Github repository in the branch called gazebo_ros [3].

2.1 The robots

After the initial process of learning about the current state of the project the choice of robot to simulate was analysed through a comparison between turtlebot2 and turtlebot3 burger. These two robots were chosen because the research group already had access to 2-3 turtlebot2 robots and turtlebot3 burger is its successor. This comparison was made to optimise for a small scale test at a later point in time. The collected data can be found in figure 2. This comparison shows that turtlebot3 burger is a good choice when it comes to price, official compatibility and size, a smaller size reduces the size of the physical area needed during a small scale test. In parallel with this comparison, the process of learning about the Gazebo simulator was ongoing by going through tutorials and running demos with turtlebots in Gazebo.

Current official compatibility

Robot	Latest ROS version	EOL
Turtlebot2	Kinetic	April 2021
Turtlebot3	Melodic	May 2023

Size

Robot	Size (L x W x H)
Turtlebot2	354 x 354 x 420 mm
Turtlebot3 burger	138 x 178 x 192 mm

Cheapest price from official resellers (US/worldwide)

Robot	Including	Price (USD)	Reseller
Turtlebot2	Everything but computer and camera	1049	CLEARPATH ROBOTICS
Turtlebot2	Everything	1900	Dabit Industries
Turtlebot3 burger	Everything	549	Dabit Industries

Cost to equip the lab with 6 robots

Robot	Nbr in lab already	Total price (USD)
Turtlebot2	2-3	5700 - 7600 3147 - 4196 plus laptops and cameras
Turtlebot3 burger	0	3294

Figure 2: A comparison between turtlebot2 and turtlebot3 burger.

With the help from these tutorials, [6], a launch file for the turtlebot3, one_robot.launch, one launch file for launching multiple turtlebots at specific positions, robots.launch, and one main launch file for launching the turtlebots in an empty world, main.launch was constructed. Figure 3 shows 2 turtlebots launched in an empty world.

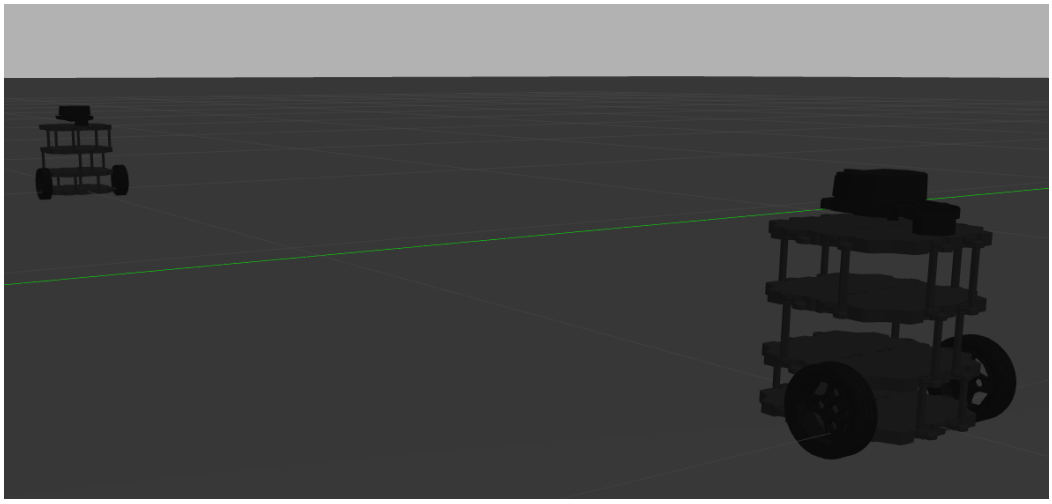

Figure 3: 2 turtlebots launched in an empty world.

The next step was to control the turtlebots through ROS. The turtlebots can be controlled by setting linear and rotational velocity. To keep going, the turtlebots need to continue to receive new data, if the data stops coming the turtlebot will stop. The robots also publishes a ROS topic that describes the current states of the robot, i.e position, orientation, velocity etc. A python program, communication, that publishes robot commands and subscribes to the states of the robots was written. To change the commands that are published and request information of the states of the robots, the class RobotCtrl was written. This class will only receive and send topics when asked to while communication does it at a given frequency. To test this functionality, a test program was written. The described ROS-structure is shown in figure 4. One can choose how many robots to simulate by changing a global variable and change the number of robots in the robots.launch file.

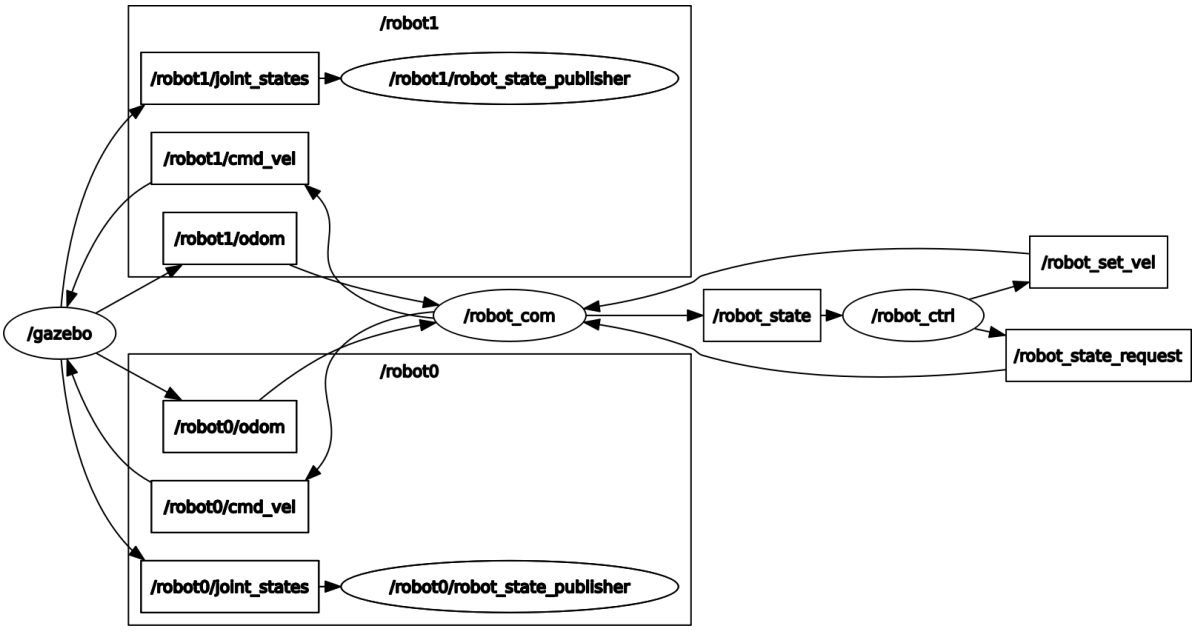

Figure 4: The ROS structure when 2 robots are launched.

The shape of the turtlebots is not similar to the shape of a normal car and the shape of the cars used in the pure python simulations. Therefore, the base model for the turtlebots was altered by adding a hollow box with no bottom, see figure 5. This created a shape more similar to normal cars while keeping the increased weigh to a minimum.

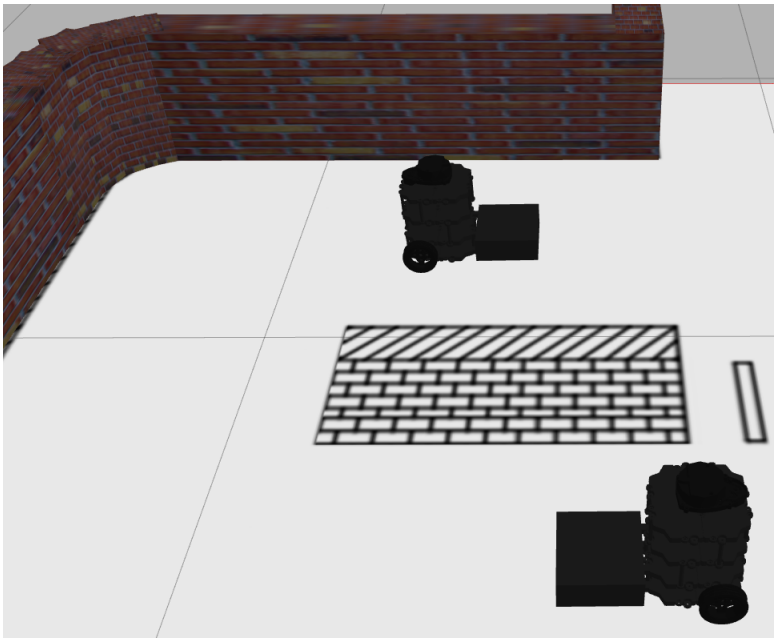

Figure 5: The altered appearance of the turtlebots.

2.2 The world

After the control of the robots was implemented a parking lot environment was built in a Gazebo world. The environment was reduced in size, compared to figure 1, to be better fitted for simulations with only 6 robots and to take up less space during a future physical test. This world was built by adding an image of a reduced parking lot as a ground plane in the world. This image was scaled so that the turtlebot3 burger would fit in the parking spots. Thereafter, walls were added on top of the ground plane image. See figure 6. The next task will be to add pedestrians and static obstacles into the Gazebo world.

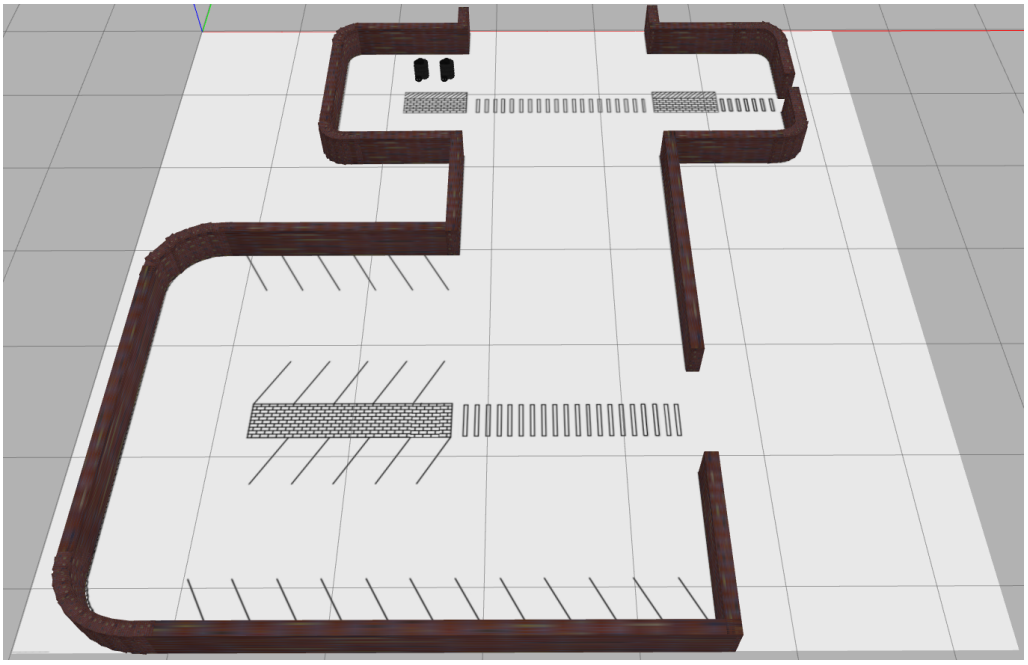

Figure 6: The parking lot environment implemented in a Gazebo world.

2.3 Pedestrians and static obstacles

By taking models from the main Gazebo library and reducing their size to fit the parking lot environment as well as reducing the pedestrians friction in the x- and y-direction to allow for easier control, model descriptions files for static obstacles and pedestrians were created in the format sdf. By following a tutorial, [2], a launch file for launching sdf-files into the Gazebo world was written. To get the state of a model, Gazebo publishes a ROS-topic called model_states at a certain rate. The script communication subscribes to this topic. When requested through another topic, the script will publish the relevant states, i.e x- and y-position, yaw angle and, for pedestrians, the corresponding velocities, for a specified model.

Another tutorial was used to learn how to control models in a Gazebo world [1]. The tutorial describes that models are controlled using plugin programs written in C++. Each model description file is linked to a specific plugin file and the plugins are compiled together with the entire ROS-workspace. As an example, for launching two pedestrians one would need two model description files and two plugin files. This makes it complicated to allow the user to choose any number of pedestrians to launch without having to add more files and recompile the workspace, which is time consuming if it had to be done every single time. A solution to that is to limit the system to work with a maximum of a certain number of pedestrians and have that many model files and plugin files stored. This solution was implemented with five as the maximum number. Another solution could be to look into the more complex description files that are used for robots and write such description files for the pedestrian models or to use some kind of robot with existing description files as pedestrians. Since the system is meant to simplify for a future real world test, the later of those options would make sense to implement but due to time restraints it was chosen to not look into that. Each plugin is setting the linear and rotational velocity of its corresponding pedestrian model when it receives a ROS-topic containing the wanted velocities.

Similar to the class RobotCtrl, the class ModelCtrl was written. It sets velocities for pedestrian by publishing the ROS-topics that the plugins are subscribed to and receives the states of the models by requests to the communication script. Two obstacles and two pedestrians that are launched in the Gazebo world can be seen in figure 7.

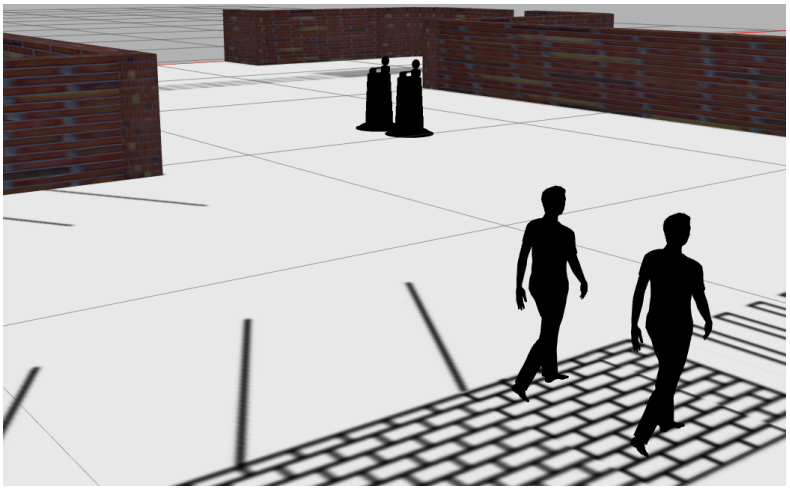

Figure 7: Pedestrians and static obstacles launched in the Gazebo world.

2.4 Dynamic launching

Up to this point, the number of robots, pedestrians and obstacles as well as their initial positions were hard coded into the launch files. Therefore, the user would have perform these kind of changes by changing the launch files manually. A more dynamic solution was now going to be implemented. This was done by changing the structure of the launch file into 4 separate files, one for launching the world, one for launching a robot, one for launching an obstacle and one for launching a pedestrian. Except for the launch file for the world, each launch file is launched together with arguments describing the initial position and rotation as well as an identification number. For launching these files the python script gazebo_launcher was written. It is subscribed to a ROS-topic containing the initial positions of robots, pedestrians and obstacles. Upon receiving this topic the script launches the files. When the script receives a topic without any data in it, everything that has been launched is terminated. This script was written by following yet another tutorial. [4]. To publish the ROS-topic, a class named GazeboCtrl was written. To change the number of robots, pedestrians or obstacles that are being launched, the user still has to change the global variables representing these numbers manually. This could be considered tedious and a solution could be to let the GazeboCtrl class set these values once a world is being launched. However, since several other scripts are using these variables one would also have to change those programs so that the variables aren't used before GazeboCtrl has given them their values. Implementing this is considered to improve the program but due to time limitations it has been chosen to not do that unless there is time left by the end of the project.

2.5 System summary

The ROS-structure, previously seen in figure 4 for a world only containing robots, has with the inclusion of obstacles, pedestrians and dynamic launching evolved into the ROS-structure seen in figure 8. In figure 8, the node named /test_ctrl works as a main script that creates an instance of each of the RobotCtrl, ModelCtrl and GazeboCtrl classes. These could have been split up into different nodes but for the sake of simplicity they were kept in the same node in this example.

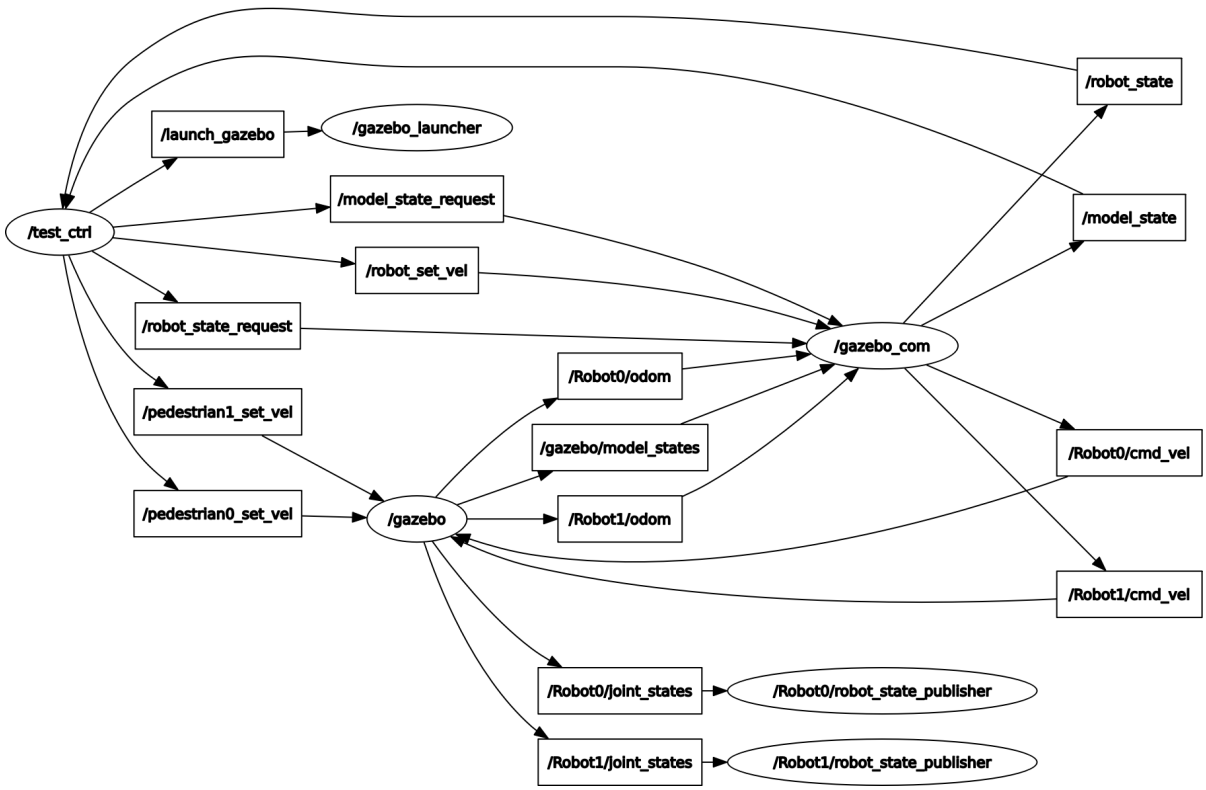

Figure 8: The ROS structure with robots, pedestrians, static obstacles and dynamic launching.

3 Challenges

So far, the major challenges has been to identify how ROS, Gazebo and the turtlebot description are supposed to interact with each other. For example, how to launch turtlebots in a gazebo world, how to set velocity of a robot and how to read the states of a robot. This was solved by taking one problem at a time and search the web for demos and tutorials. Another major challenge was to get the plugins working. It took a relatively long time to figure out that the plugins had to be compiled together with the workspace. Once that had been figured out there were still some problems, just recompiling the workspace was not enough, one had to compile it completely from scratch by deleting the devel and build folder as well as the CMakeLists for the workspace and then compile it. After compiling, one had to manually copy the file setup.sh in the devel folder and paste it into the gazebo folder within devel/share. Why this has to be done manually remains unknown. The time it took to get this working was longer then expected according to section 1.3 and this is one major reason why the objectives were redefined into the ones explained in section 1.4.

An anticipated challenge that is coming up is the implementation of the control system to make the turtlebots follow a given trajectory.

References

[1] Miguel Angel. *Gazebo QA 002 – How to create Gazebo plugins in ROS Part 2*. URL: https://www.theconstructsim.com/gazebo-qa-002-create-gazebo-plugins-ros-part-2/ (visited on 07/28/2020).

[2] Miguel Angel. *Gazebo QA 003 – How to spawn an SDF custom model in Gazebo with ROS*. URL: https://www.theconstructsim.com/gazebo-qa-003-spawn-sdf-custom-model-gazebo-ros/ (visited on 07/28/2020).

[3] Josefine Graebener et al. *Automated Valet Parking*. URL: https://github.com/jgraeb/AutoValetParking (visited on 07/01/2020).

[4] David Lu. *roslaunch/API Usage*. URL: http://wiki.ros.org/roslaunch/API%5C%20Usage (visited on 07/28/2020).

[5] Richard M. Murray and Josefine Graebener. *SURF 2020: Hardware Implementation of Contract-Based Design for Automated Valet Parking System*. URL: http://www.cds.caltech.edu/~murray/wiki/index.php?title=SURF_2020:_Hardware_Implementation_of_Contract-Based_Design_for_Automated_Valet_Parking_System (visited on 07/01/2020).

[6] Arif Rahman. *ROS QA 130 – How to launch multiple robots in Gazebo simulator?* URL: https://www.theconstructsim.com/ros-qa-130-how-to-launch-multiple-robots-in-gazebo-simulator/ (visited on 07/01/2020).