Introduction to ROS Documentation

Sommersemester 2024

Autonomous Driving - Team 3

Contents

1 Introduction

In this project the task was to drive a predefined track in an urban environment as fast as possible without leaving the road, crashing into other cars and while obeying traffic lights. Figure 1.1 shows a top down view of the urban environment as well as the predefined track to be driven.

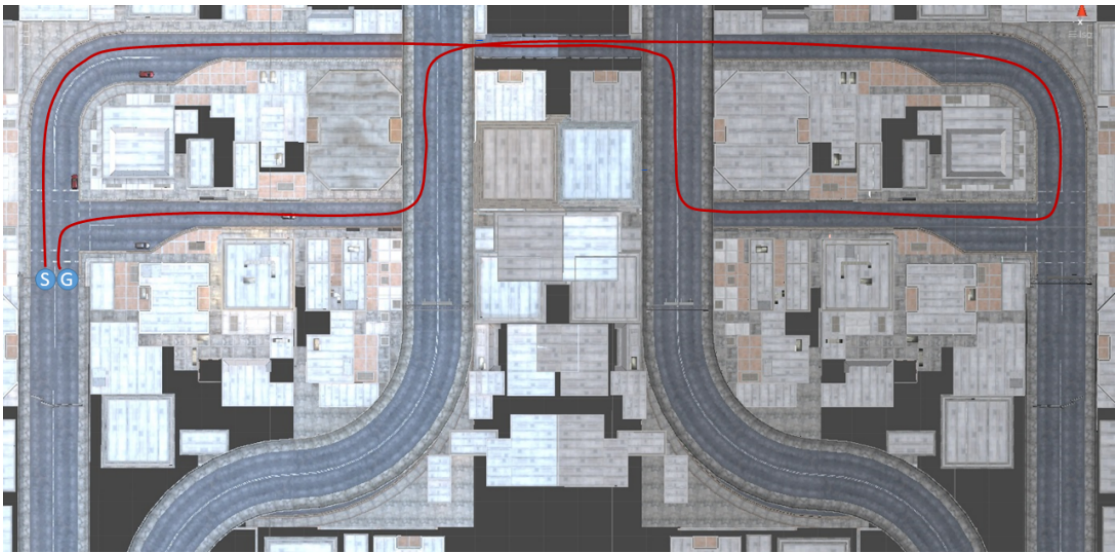


Figure 1.1: Predefined track through the urban environment.

2 General Structure

The environment as well as the car itself were simulated in Unity while a ROS-Simulation-Bridge was used to process data provided by the simulation. The ROS-Simulation-Bridge then in turn provides messages for the car controller to drive the simulated car along the predefined track.

A base version of the car controller was provided beforehand as well as multiple cameras attached to the body of the car, images of which can be seen in figure 2.1.

(a) RGB-Camera. (b) Depth-Camera. (c) Semantic-Camera.

Figure 2.1: Camera images provided by the Unity simulation.

As outlined in the project document the provided data should be combined into the following components:

- A state machine which which manages the street rules e.g. staying on the road and stopping at red traffic lights
- A perception pipeline that converts the provided depth camera information into a virtual representation of the environment and extract relevant information from the semantic and RGB cameras
- A path planner that generates a path through the environment adhering to the pre-defined track
- A trajectory planner that plans a trajectory based on the planned path

Thus, we decided on the following initial structure for the project, shown in figure 2.2.

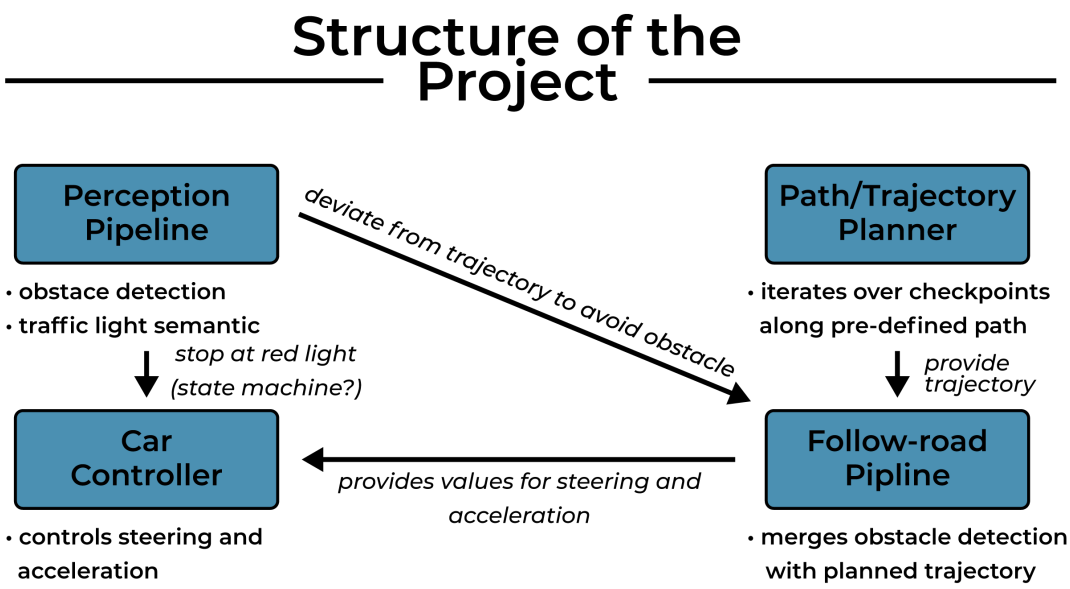


Figure 2.2: Overview of the initial idea of the project structure.

3 Implementation

The actual implementation deviates from the initial idea due to better understanding of the task as well as problems during development and within the team. These problems will be described in sections 4 and 6.

3.1 Perception Pipeline

Figure 3.1 gives an overview over the process of constructing an octomap, a 3 dimensional voxel representation of the environment.

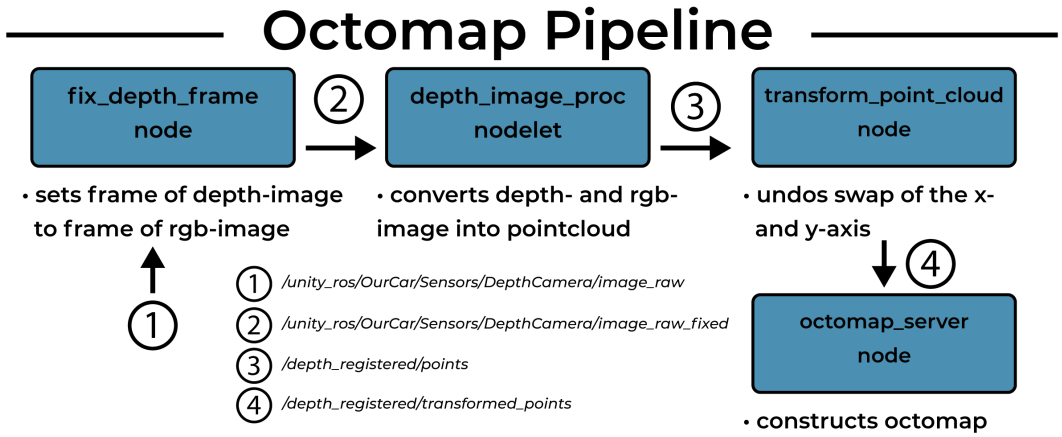
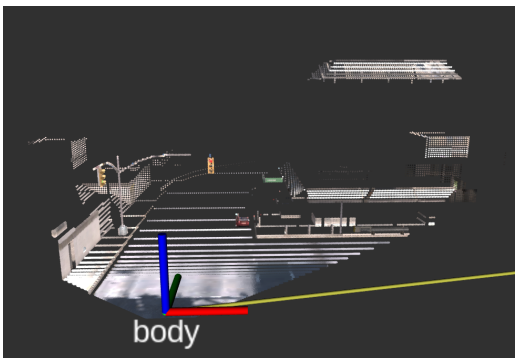


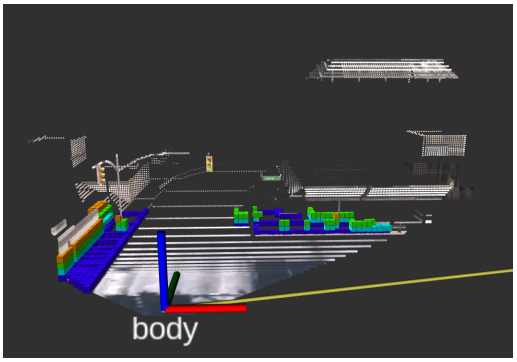
Figure 3.1: Overview of the initial idea of the project structure.

We use the ROS `depth_image_proc`-nodelet to project the RGB camera data into 3D-space, see figure 3.22(a). This step requires both images to be in the same frame which the `fix_depth_frame` ensures. We also noticed that the y- and z-axis were swapped in the `true_state_parser.h` file, and wrote a `transform_point_cloud`-node to revert that to avoid wrong orientation of the resulting pointcloud and thus octomap.

Finally we use the `octomap_server`-package to construct an octomap from the point-cloud. For performance reasons we chose a relatively rough, but sufficient resolution of 0.6 m/voxel. Additionally we only take points between -1.05 and 1 m into account for the pointcloud. This ensures that the road itself is not classified as occupied space but the curb and other relevant obstacles are, while irrelevant points, like a ceiling for example, are excluded. See figure 3.22(b).



(a) Projected pointcloud.



(b) Octogrid contructed from projected pointcloud.

Figure 3.2: Camera images provided by the Unity simulation.

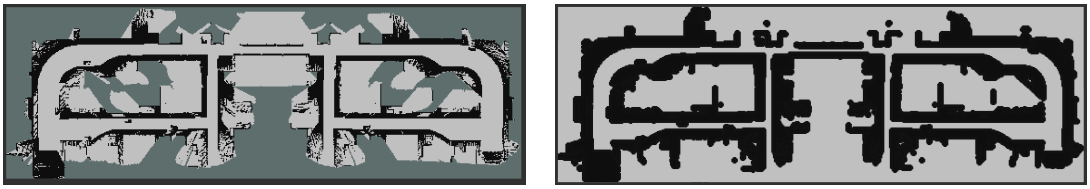
Additionally to the octomap we handle the traffic lights in the perception pipeline with the `traffic_light_detector`-node. Here we also implemented an own message `RedLightDetection`, which is essentialy just a bool. The idea is to segment the part of the rgb image containing the traffic lights using the semantic camera and then check for pixel colors within the segmented image, see figure 3.3. This ensures that only the color of the traffic lights is used for decision-making. The node publishes true or false under the `/red_light`-topic depending on the traffic light color.

(a) RGB-Camera. (b) Semantic-Camera. (c) Segmented-Camera.

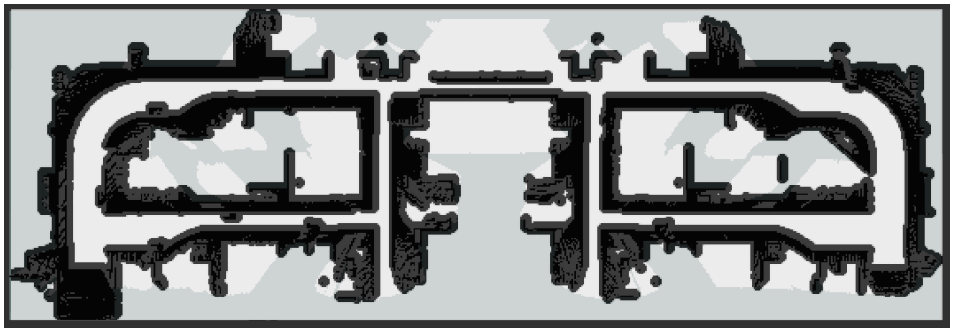
Figure 3.3: The traffic light segmentation.

3.2 Path/Trajectory Planner

For path and trajectory planning we use the `move_base`-package, which takes the down-projected octomap (see figure 3.44(a)) as input and converts it to a global and a local costmap.
The car is roughly 2.4 m by 4.8 m, to increase the safety margins we found an exaggerated footprint of the of 4 m by 8 m to work best, such that no collision is possible with any planned path. Together with an inflation radius of 2 m, the octomap and costmap of the whole environment can be seen in figure 3.4.



(a) Downprojected occupancy map. (b) Costmap.



(c) Both maps overlayed, with the second darkes areas indicating the inflation.

Figure 3.4: Side by side of octomap and inflated costmap.

The trajectory planner of the `move_base`-package then publishes the velocity needed to follow the calculated path under the `/cmd_vel`-topic.

To ensure that the car actually drives the predefined path, six predefined checkpoints were chosen as shown in figure 3.5. These are published one after another by the `checkpoint_handler`-node. After startup, when the costmaps are available the first checkpoint is published as a navigation goal, then the `checkpoint_handler`-node checks the position of the car and publishes the next checkpoint as soon as the car reaches the current checkpoint, within a 5 m radius.

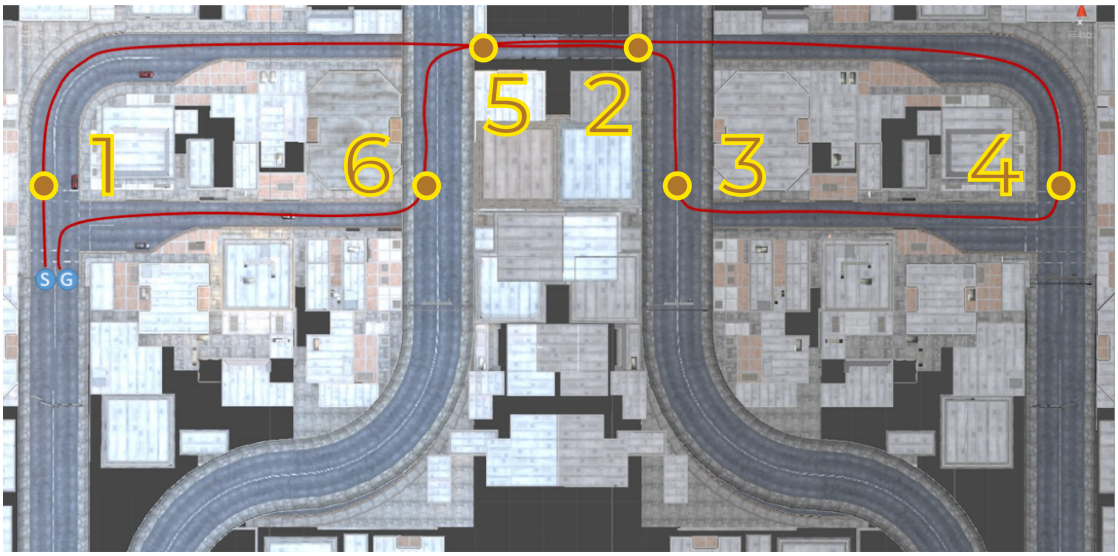


Figure 3.5: The six checkpoints along the predefined path.

Figure 3.6 shows the global path to the first checkpoint planned by the global path planner, which dynamically updates with the movement of the car.

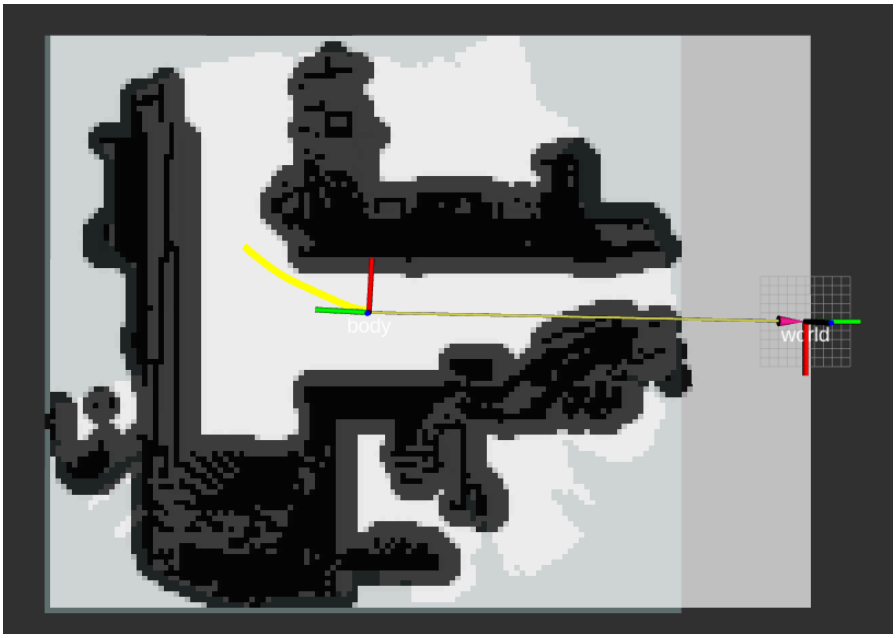


Figure 3.6: Path planned by the global planner (yellow line).

3.3 Car Controller

The `car_controller`-node is extended to subscribe to the `/cmd_vel`-topic and simply publish the velocity under the `/car_commands`-topic.

3.4 State Machine

The initial idea of a suitable state machine is shown in figure 3.7, and was implemented with the ROS `smach`-package. But we soon realized that the lack of complexity of the task doesn't really warrant a state machine and just makes the codebase more convoluted. We decided not to incorporate a state machine into our implementation.

It is simply more efficient to put the traffic light logic into the car controller by subscribing to the `/red_light`-topic and handling it directly there. A separate obstacle pipeline is also not needed, because the obstacles are static and thus correctly incorporated into the octomap, which in turn ensures that the path planner automatically avoids these obstacles.

If there were multiple different tasks like driving, parallel parking or dynamic obstacle avoidance, a state machine would indeed be useful.

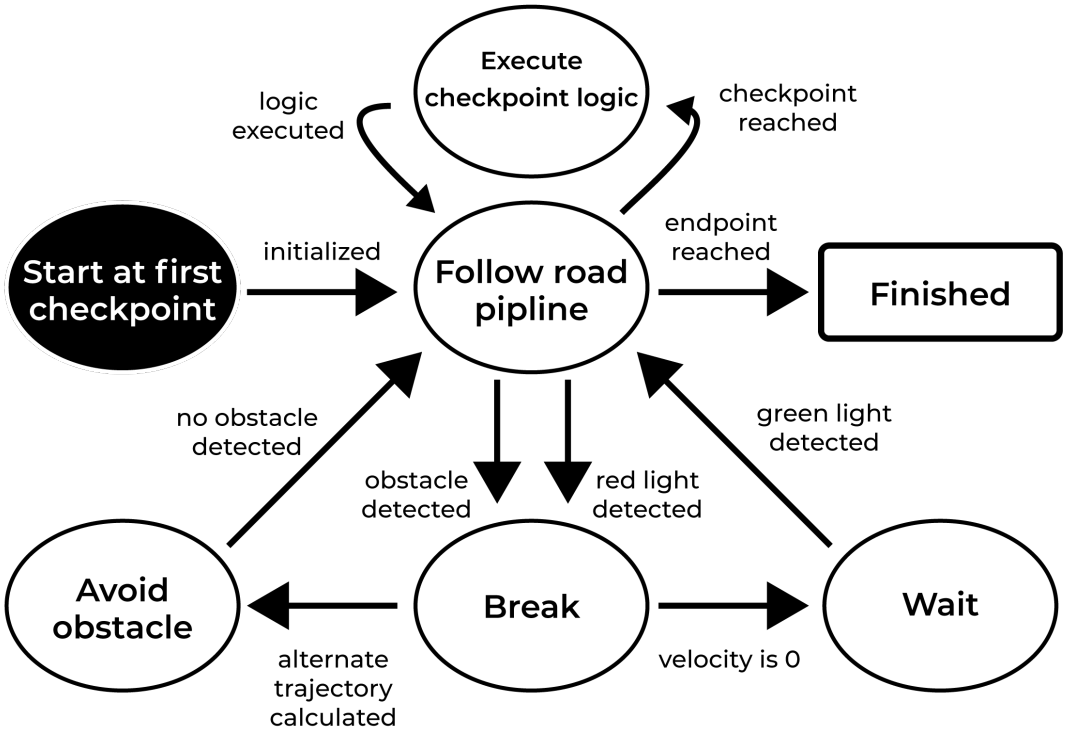


Figure 3.7: Overview of the initial idea of the state machine.

4 Open Problems

1. Drift-compensation: Although the corrupted pose and twist estimates are used, we could not observe the estimated position of the car drifting. Thus there probably is a mistake in the implementation somewhere.

2. Only the global planner correctly plans the path. Although the local planner receives the exact same costmap its goal is always the current position of the car itself resulting in a `/cmd_vel` of zero. We suspect that the local planner somehow doesn't receive the correct navigation goal.

3. Integration of the `/red_light`-topic into the car controller. Since we don't know how the car controller will behave due to the local/trajectory planner not working we have not yet implemented this into the car controller.

4. Checkpoint publishing: The current global costmap is only as big as the inital observed area. Thus no navigation goals outside the global costmap can be published. The fix for this would be to check if the checkpoint to publish lies on the global costmap or not. If not, search the costmap for the nearest unoccupied point to the actual checkpoint and publish this point as an intermediate checkpoint, repeat as often as needed. This functionality is also not yet implemented.

5. Since the very beginning, the unity simulation can randomly crash with the error code -6. Simply restarting a few times fixes this. Rviz can also crash after a few seconds into the simulation, this seems to be a bug in Rviz itself judging by the error messages, also exit codes -6, -11.

5 Results and discussion

The current state of the project is a working perception pipeline, a (semi) working path planner pipeline with some remaining problems in the local planner as well as the checkpoint handling. Whether or not the car controller works as intended has not been tested due to the local planner not working correctly.

6 Who Did What?

After the first team meeting the task distribution was as follows: Baskhar Sah and Ramkrishna Chaudhari were to understand and improve the car controller, Victor Kawai was to implement the perception pipeline, Jinchi Peng was to implement the path planner and Tibor von Gencsy was to implement the state machine. Baskhar Sah and Ramkrishna Chaudhari left the team and did not contribute. Leaving too much work for the remaining members to finish the project in time before having to prepare for exams.

Leaving this 90 % finished implementation as is, is unsatisfying and hard for us, but the right decision given our exam situations. This is how the tasks were distributed in the end:

Victor:
- Perception pipeline implementation
- frame IDs cleaned and attributed the correct ones
- general work/prototyping (commit history should provide more info if needed)

Jinchi:
- Path planner concepts

Tibor:
- Documentation
- path planner concepts and implementation
- general work/prototyping (commit history should provide more info if needed)
- (state machine implementation)

7 Bibliography

[1] Official ROS Tutorials, https://wiki.ros.org/ROS/Tutorials. [Accessed: 17-Jul-2024].

[2] Move Base Package, https://wiki.ros.org/move_base. [Accessed: 17-Jul-2024].

[3] Octomap Server, https://wiki.ros.org/octomap_server. [Accessed: 17-Jul-2024].

[4] ROS Navigation, https://wiki.ros.org/navigation. [Accessed: 17-Jul-2024].

8 ROS Graphs

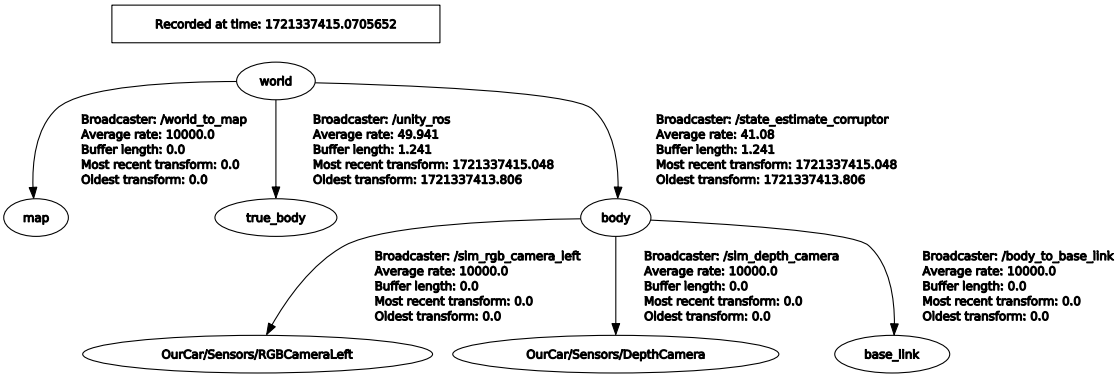


Figure 8.1: Frames.

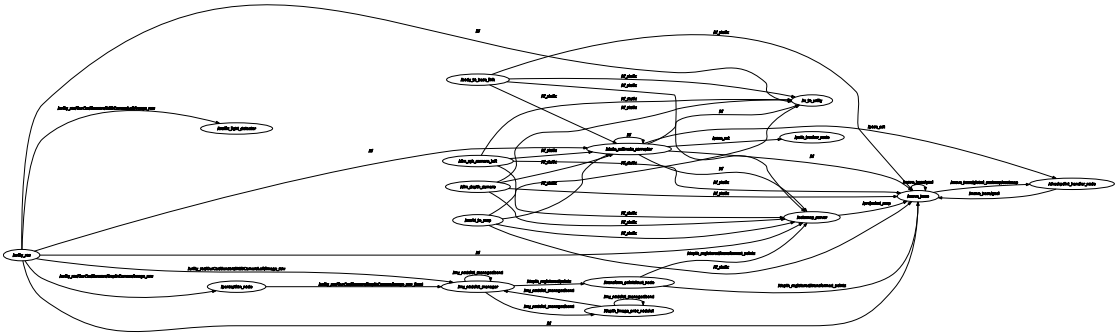


Figure 8.2: Rosgraph (should be zoomable).