Documentation for *Autonomous Drones* for team 12

Authors:	Mohamad Ammach
	Vegard Haraldstad
	Thomas Borge Skøien
	Geir Ola Tvinnereim
Last compiled:	2022–08–02
Source code:	`https://gitlab.lrz.de/GeirOlaTvinnereim/autonomous_drones_ss2022`

The work on the tasks was divided in the following way:

Mohamad Ammach	Task 1	50%
	Task 2	0%
	Task 3	25%
Vegard Haraldstad	Task 1	0%
	Task 2	50%
	Task 3	25%
Thomas Borge Skøien	Task 1	0%
	Task 2	50%
	Task 3	25%
Geir Ola Tvinnereim	Task 1	50%
	Task 2	0%
	Task 3	25%

Introduction to ROS (Robot Operating System)

Team 12: Help me, I'm droneing!

For our final project we will utilise two drones in order to generate a single combined 3D voxel-grid representation for an unknown environment (50m x 50m) as fast as possible. In order to solve this mission we will implement Cpp-nodes and use ROS[1] as a middleware. The mission is simulated using Unity[2].

[1] https://www.ros.org
[2] https://unity.com

Contents

1 Overview

For our project we decided to go with a few key packages. *Explore_lite* [1] is responsible for the overall exploration of the drone, finding the frontiers from the map and choosing new goals. This package is closely linked to *move_base* [2] which takes these goals and interprets them in addition to the current view from the drone, of where it is and where it should go. All of this is possible with the help of *depth_image_proc* [3] and *octomap_server* [4] to generate point clouds, a voxel grid and also a 2d map of the environment that the aforementioned packages can use.

2 Project start

The group met for the first time the next work day after the source code was published. At that time there was still uncertainties around the group size, so we assumed a group of four people. By running the initial code we figured out that there was already a controller implemented alongside with an interface for one of the drones. By displaying the RQT-graph, shown in figure 1, we also got an overview of the used publishers and subscribers.

From this starting point we pointed out which functionality remained to be implemented in order to complete the mission. We boiled it down into four modules in addition to implementing the interface to the second drone and understanding the bridges between *unity* and *ros*. We defined the remaining modules and divided task responsibilities among the team members as presented in the initial presentation. Unfortunately already after the first week of implementation one group member decided to leave the group, luckily we got Thomas as a replacement. After having worked on the project for some days we realised that the tasks are more involved. In particular there is a significant correlation between the perception-pipeline and voxel-grid representation, and between the path- and trajectory planning. Therefore we only defined 3 different tasks in this documentation. The corresponding responsibility assignment is shown in table 1.

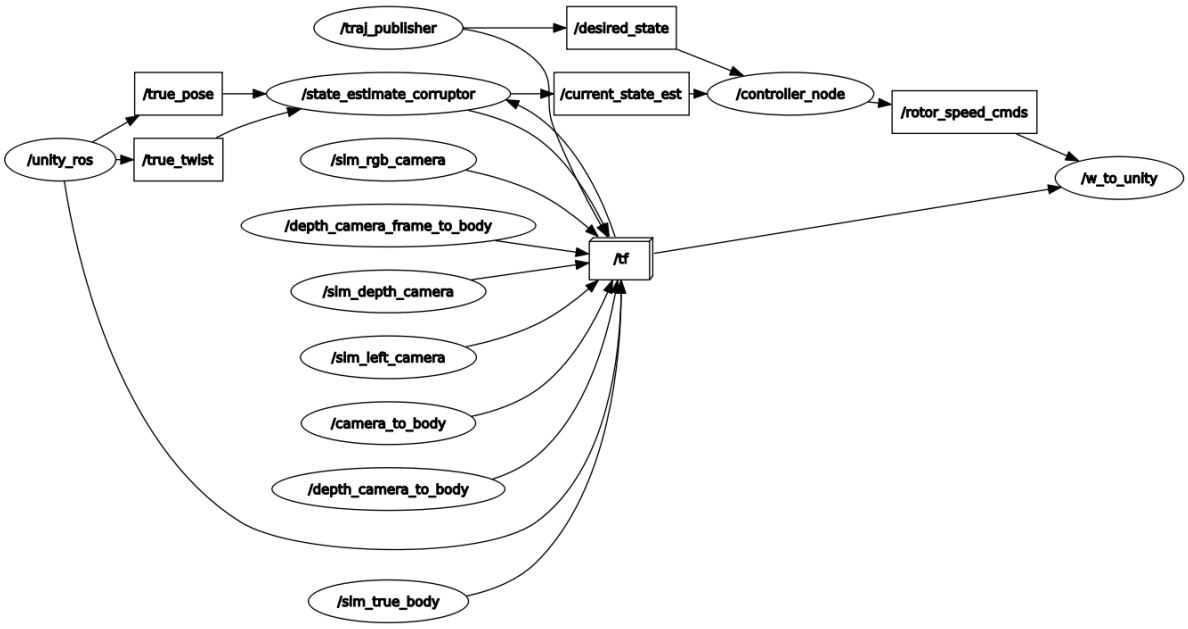

Figure 1: *Original rqt-graph*

Tasks	Responsibility assignment
1. State machine	Geir Ola - Mohamad
2. Perception-pipeline	Thomas - Vegard
3. Path-trajectory planning	*Everyone*

Table 1: *Modularization and corresponding resposibility assignment*

2.1 Interfacing two drones

Since the initial code already provided an interface to one drone, it was tempting to simply copy-paste the code for each module. However this approach turned out to expand the overhead of the code dramatically, disobeying a minimal interface. A larger interface also leads to a harder debugging routine. In order to get good code quality, implying readable and easily maintainable code, we decided on reusing the given functionality by spawning multiple modules with different arguments within the `roslaunch` files. For instance by spawning two controller nodes, they could control each drone individually. Regarding the implementation we would only need to assess the control and logic for **one** individual drone. Simultaneously we could easily expand the code base for the case of exploring an unknown environment with more then two drones. The spawning logic is found in the *simulation/launch* folder. The corresponding rqt-graph for our configuration is shown in figure 4.

Report on task 1, State machine

3 State Machine

The state machine's responsibility is to keep track of the different parts of the mission for both drones. We decided to specify 4 different states (takeoff, hover, explore, landing) which are defined in an `enum class State`. The node subscribes to the *current_state* and *cmd_vel* topics from *state_estimate_corruptor* and *move_base* nodes respectively, and publishes on *desired_state* topic to the *controller*.
The state machine is launched along with the *state_estimate_corruptor* and a *controller*, in the *mission.launch* file, where the *perception.launch* [4] and *planning.launch* [5] files are also included.

3.1 Preliminary configuration

Our first milestone was to interface the controller and the state machine setting way-points[3]. After that the logic of the states was implemented. We took inspiration from the state machine in exercise 3 in order to develop our own solution, where we spawn a loop-function within a C++ class using the `ros::Timer` object. The state machine delivers a desired state for the controller, composed of a position, linear and angular velocities, and an acceleration. Along with the *controller* and the *state_estimate_corruptor* nodes, the state machine is spawned twice in order to maintain good code quality and a reasonable abstraction level as explained in 2.1.

3.2 Takeoff

The goal for this state is to start the mission, commanding the two drones to take off to a specified altitude of 2 m. This is done by feeding the controller the current x and y coordinates of origin for each drone and a z component corresponding to the desired altitude, with zero valued velocities and accelerations. Implementation is found in `StateMachineNode::takeoff()`.

3.3 Hover

After successfully taking off, the drones enter a hover state where they keep their current position. So, the current position is stored and fed continuously into the controller with zero velocities and accelerations in order to maintain the hovering position. Using the controller implementation, the drone can enter the hovering state from any pose. This state continues until the initial mapping is done and the path planner decides where to explore, which is assumed to take about 5 sec. The state machine enters then the *explore* state. Implementation is found in `StateMachineNode::hover()`.

3.4 Explore

In *explore*, desired position and orientation are fed to the controller. Desired position is the current position added to a desired direction which is the commanded x and y velocities rotated by the current yaw angle implemented in `StateMachineNode::twist_to_pos()`. For the desired orientation we just need the yaw, so the current yaw is added to a scaled value of the commanded yaw rate. This logic is found in `StateMachineNode::twist_to_quat()`. The commanded velocities are the ones given by *move_base* through the topic *cmd_vel*. Implementation is found in `StateMachineNode::explore()`.

3.5 Landing

The *landing* functionality is similar to the *takeoff*, the only difference is that we need to enter the *hover* state in order to have a defined point to decend from, and feed in a zero altitude instead of a specified height. The *landing* state is the final state before mission is accomplished. Implementation is found in `StateMachineNode::landing()`.

[3] an intermediate point for the drones to explore

Report on task 2, Perception pipeline

4 Perception pipeline

4.1 Perception

For the perception pipeline we used packages that were already readily available. Depth_image_proc [3] is the package we used to transform the image data into point clouds. This was implemented as two nodelets, one for each drone in the *perception.launch* file.

4.2 Voxel-grid-representation

For the voxel-grid-representation we opted with octomap_server [4] as was suggested in the original project outline. There were, however, quite a few parameters that had to be tuned here. In addition, we had to make a *pointcloud_merger* node to be able to generate one common map for both drones.
In order to visualize the voxel grid, in RViz add *Map* and *MarkerArray* with the topics */map* and */occupied_cells_vis_array* respectively.

4.2.1 Point Cloud Merger

The *pointcloud_merger* is responsible for taking in point clouds from both drones and publish both on the topic *pointclouds*. Octomap_server then subscribes to this topic and uses it to build a map.

Report on task 3, Path- and trajectory planner

5 Path- and trajectory planning

5.1 Exploration and goal generation

Explore_lite [1] package, with its *explore* node, was used to explore the unknown environment and provide goals to the path planning. It searches for frontiers of the known area, and calculates a desired goal for the drones to pursue. This goal is sent to *move_base* as a *move_base action goal* through an action server. Separate action servers were created for the two drones using different namespaces for each drone. The node was also spawned twice for each drone in the *planning.launch* file along with the *move_base* node.

5.2 Planning

As the path and trajectory planner we chose move_base [2]. To make move_base work in our pipeline we had to write a point cloud converter node and a map service node. In addition much of the job was to configure and tune the parameters correctly to make move_base function properly. The move_base node was spawned twice, one for each drone, publishing two *cmd_vel* topics for each of the state machines, where the commanded velocities are transformed into desired position and orientation, as explained before in 3.

5.2.1 Point Cloud Converter

This node converts the *pointcloud2* output of the *depth_image_proc* nodes to *pointcloud1* since *move_base* uses the depreciated *pointcloud1* standard.

5.2.2 Map Service

We also had to create a service as a link between the topic *map* and the *move_base* node as it needed a service to get the latest map of the environment. This is just a simple node that subscribes to the map and provides a simple service to give it on request.

6 Challenges

In this section we discuss weaknesses and remaining functionalities in our code base in order for the drones to complete the mission with a high performance.

6.1 Points of improvement

6.1.1 Path-Trajectory planner

As mentioned above, since the used packages are implemented in 2D, this provides a certain weakness to the algorithm. The reliability of the path and trajectory planner is one point that could be improved. It has to be one of the most "intelligent" parts, especially when it comes to getting out of tricky situations and continue mapping afterwards. The interfacing between move_base, thought the state machine and to the controller could also be improved. The drone is a bit jiggly and a smoother reference should help on that.

Moreover, the packages themselves have their own weaknesses, in respect to time and accuracy. There are other trajectory planning packages with much higher accuracy. But using them comes with the price of more complexity in the implementation and configuration.

6.1.2 Inefficient mapping

Explore_lite gives some degree of collaboration between the drones, but a more collaborative goal generation and exploration would be beneficial.
In addition, the drones gives both false positive and false negatives. This again makes the mapping more inefficient since the planning is not optimal with a map that is not completely correct.

6.1.3 Time of the mission

Having a non-optimal trajectory planner, along with an inefficient mapping, affects the time to complete the mission. Right now the drones move in a slow rate, going over the same point more than once in some cases. This makes the mission time much more higher than wanted. To reduce it, the aforementioned weaknesses should be lifted, providing a more optimal algorithm.

6.2 Unresolved functionality

6.2.1 Tracking in 3D

The path and trajectory planning of the drones while exploring was done in 2D. *Move_base* and *explore_lite* are based on a 2D logic, where the commanded velocities to the drones are given in the x-y plane along with a yaw rate. The altitude is set to a desired constant value, which is a limiting factor.
In this area the project could get to a higher level of functionality if a 3D logic is to be implemented. Such logic would give more accuracy allowing a more accurate localization so that the drones can move more efficiently in their surrounding, having more freedom in the z-axis.

6.2.2 Full mapping of the environment

Since the mission is taking much more time than we would like it to at the moment, not the whole environment is being mapped. The resulting map doesn't resemble the whole environment accurately, and may contain some missed points.

6.2.3 Transitioning to landing

The state machine switches between the different states upon the completion of a certain condition. Switching to landing is based on the mapping being completed. This can be perceived through the frontiers marker array output of *explore*. Nonetheless, due to the incomplete mapping this functionality has not been implemented.

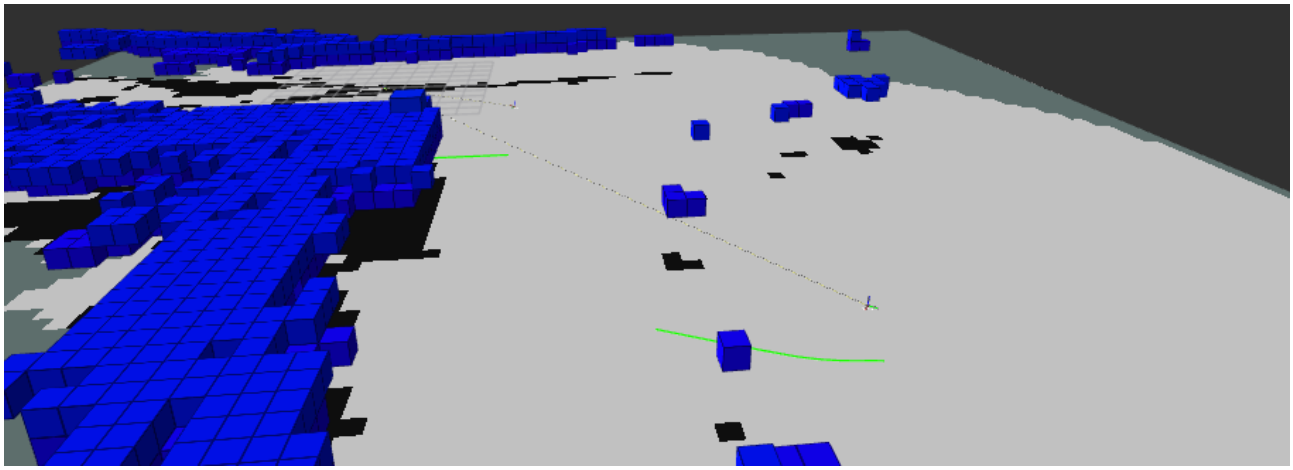

Figure 2: *Trajectory planning and mapping in RViz*

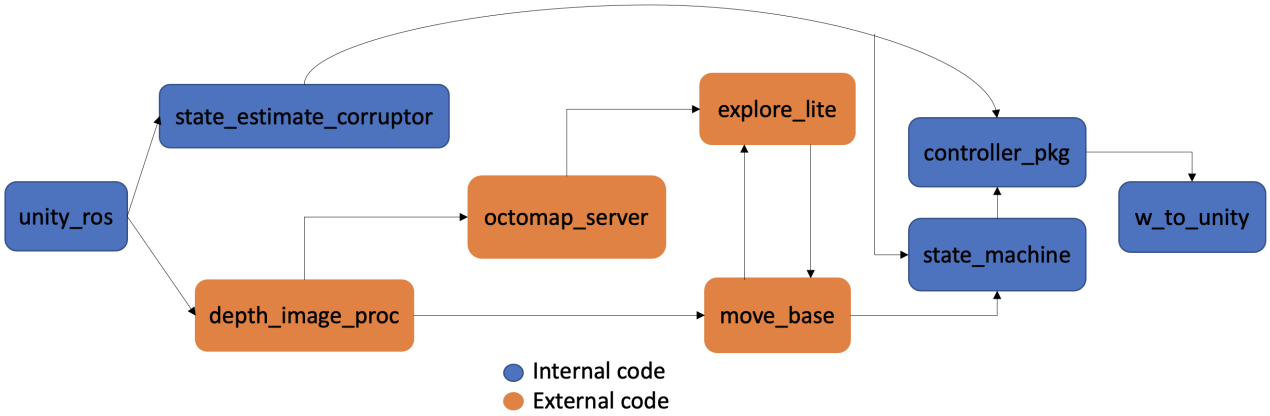

Figure 3: *Modularization of the code*

7 Summary

The project is composed of three main sections. The state machine which determines the status of the drones, giving commands to the controller, and receiving commanded velocities while exploring from the trajectory planner.

Another section is the perception. It transforms the inputs of the cameras into a voxel grid, creating a map for visualization and also for the trajectory planner to use. The merged voxel grid representation is of good quality, but can also be improved in the area of tuning false positives and false negatives.

The trajectory planner uses *move_base* and *explore_lite* to decide the next goal of the drones, based on the map provided from perception. This is the most "dynamical" part of the pipeline and this is also where improvements can be done to improve the reliability. The relation between the different modules is illustrated in the figure 3. This is a simplified overview which also shows the categorization of internal and external code.

In a summary, most of the requirements were met. From taking off, to mapping and representing a voxel grid, to path and trajectory planning. What remains is for the mission to be completed in a reasonable time, and land at the final goal. Other challenges are assessed in section 6. A final graph of the whole setup can be seen in figure 4. Figure 2 also shows a visualization of the mission in RViz.

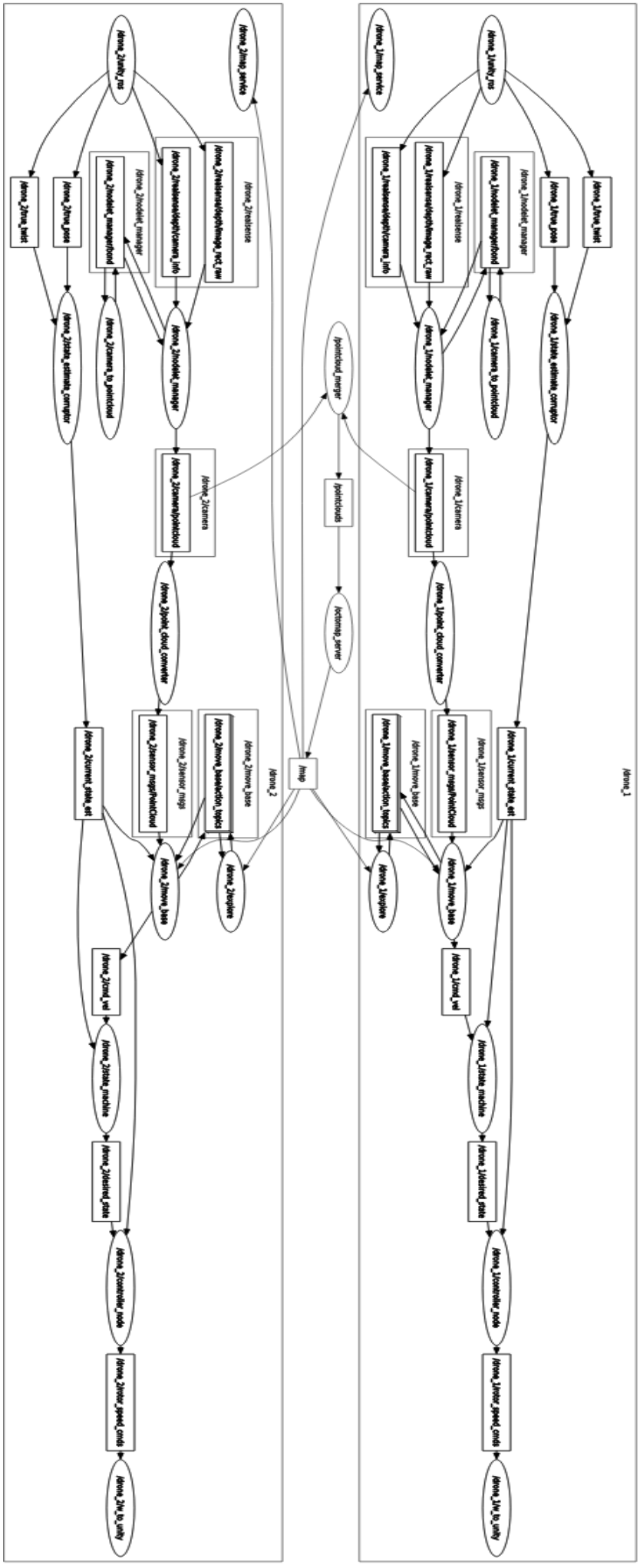

Figure 4: *Final rqt-graph*

References

[1] ROS package: *explore_lite*
`https://wiki.ros.org/explore_lite`

[2] ROS package: *move_base*
`https://wiki.ros.org/move_base`

[3] ROS package: *depth_image_proc*
`https://wiki.ros.org/depth_image_proc`

[4] ROS package: *occupancy_grid*
`https://wiki.ros.org/octomap`