Mike Leske
R00183658

COMP9069 – Robotics & Autonomous Systems
Assignment 1 – ROS

Table of Contents

1 Introduction

This report aims to provide additional information on the implemented solution. Special focus is put on describing how assignment requirements have been implemented.

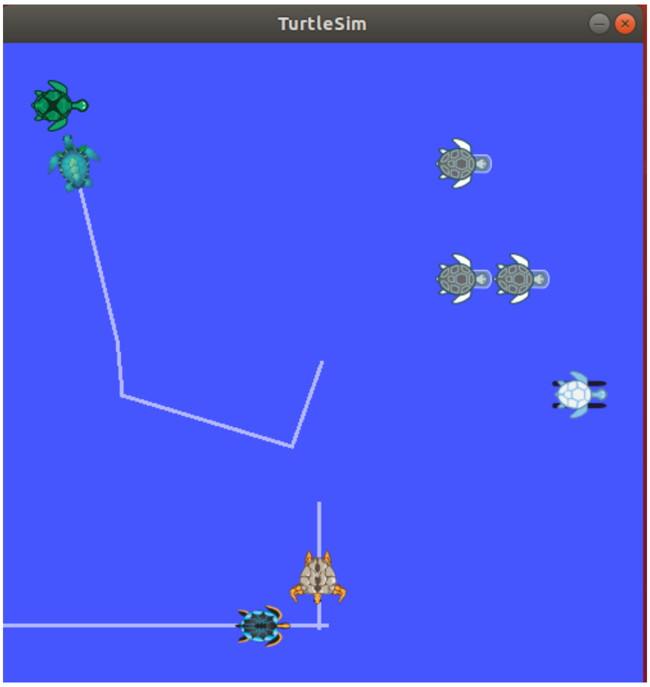

Figure 1 Screenshot from running simulation

2 Component Architecture

Figure 2 displays the system component architecture snapshot taken with `rqt_graph`. It contains topic Publishers and Subscribers, but lacks Service calls.

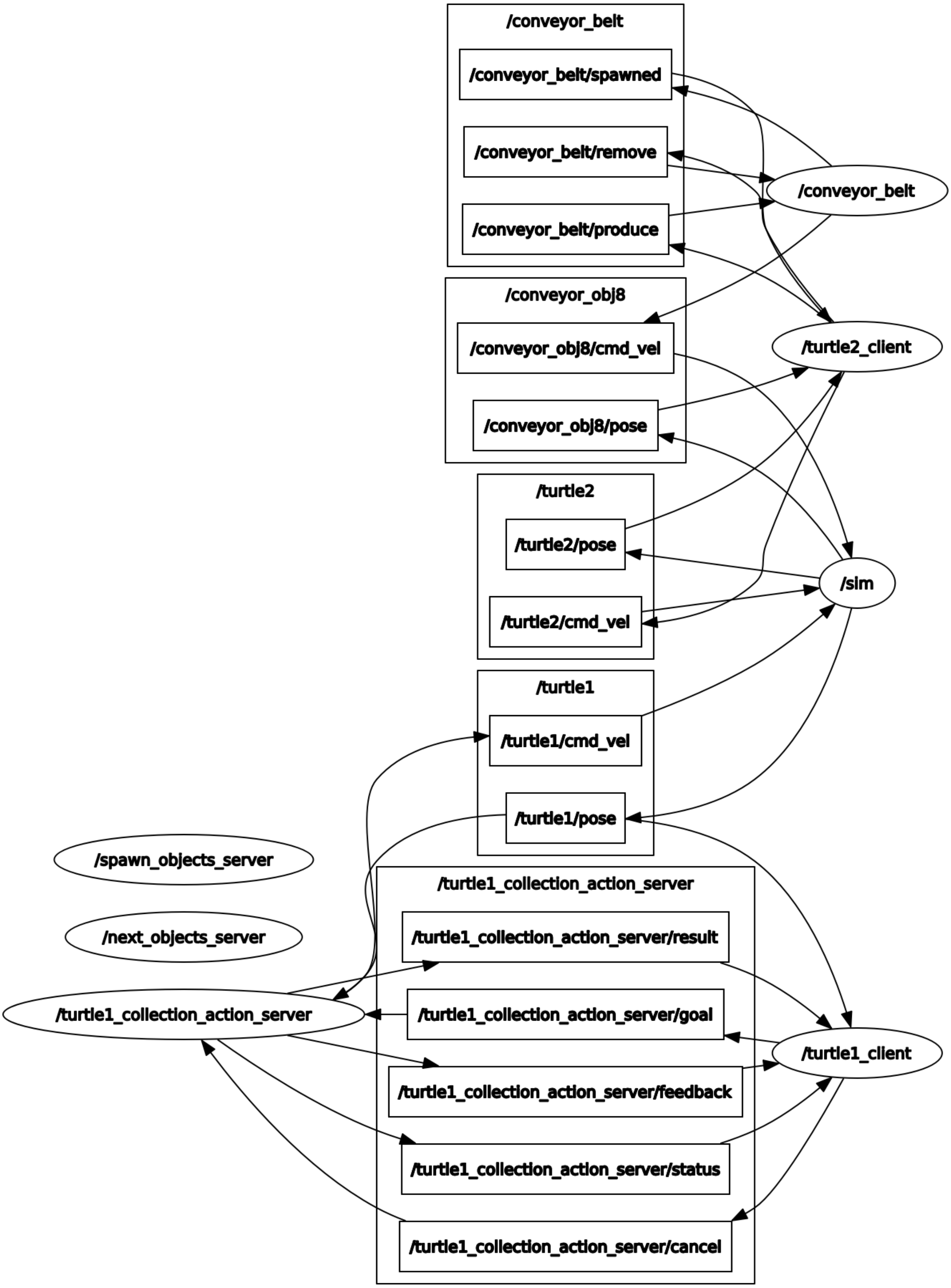

Figure 2 Assignment 1 ROS component architecture

3 Grading Requirements

1. Create a ROS service server that will spawn a given number of turtles at random coordinates throughout the workspace. The number of turtles to be spawned should be passed to the server as the message request. The server response will depend on how you choose to implement the remainder of the assignment. [10%]

- Implemented in `SpawnObjectsClient.py` and `SpawnObjectsServer.py`
- `SpawnObjectsServer.py`
    - Mounts service at `/spawn_objects`
        - Reads object count from service message
    - Calls turtlesim `/spawn service` to create objects in random locations (the conveyor belt area is excluded from possible coordinates)
    - Calls `/spawned_object` service to inform ROS node that selects next object about newly created object
- `SpawnObjectsClient.py`
    - Calls `/spawn_objects` service and provides a random number of objects to be created by `SpawnObjectsServer.py`

2. Create a ROS service server that determines the next object to be collected by turtle1. Alternatively, determine the entire route in advance by solving the travelling salesperson problem. Note: turtle2 and "conveyor belt" objects should not be included in the calculations here. [15%]

- Implemented in `NextObjectServer.py`
    - Mounts service at `/next_object`
        - Receives service message from turtle1 with current coordinates of turtle1
        - Loops over known objects to be collected by turtle1 and determines the closest objects (**greedy implementation**)
        - Returns closest next object
    - Mounts service at `/spawned_object`
        - Receives messages from `SpawnObjectsServer.py` about newly spawned objects
        - Updates local data structure to track objects
- Implements 2 measures to distinguish between turtle1 objects and turtle2 objects
    - Object name prefix set to "object" followed by id, e.g. object1
    - Explicit object notification via `/spawned_object` service

3. Create a ROS action server that moves turtle1 in a straight line to a specified location. The action server should provide the following feedback during execution:
   - An estimate of the remaining journey time.
   - A progress bar (in the form of a string) indicating the percentage of the journey completed, e.g. if the turtle has travelled 60% of the distance to the specified location, the progress bar should be `|======....|`

[25%]

- Implemented in `CollectionActionServer.py`
  - Action message contains:
    - Goal: target (x, y) coordinate
    - Result: success state when target location reached
    - Feedback: remaining_time and progress bar

```
float32 x
float32 y
---
bool success
---
string remaining_time
string progress
```

  - When goal is received, the ROS node calculate the smallest angle to turn turtle1 towards next target location is calculated. Next, turtle1 is instructed to move to target coordinates.
  - Feedback is provided during movement only. The discussion to include turtle turn time was a bit late and I understood "remaining journey time" as turtle moving in space, not theta. Including turn time in Feedback would be possible, but would require some redesign of appropriate functions.
  - Feedback is implemented in function `update_feedback()` and updated twice per second
  - Sample feedback output:

```
rostopic echo /turtle1_collection_action_server/feedback
…
feedback:
  remaining_time: "9.30905139717"
  progress: "|====......|"
```

4. Spawn new "conveyor belt" turtles at random intervals (but spaced at least five seconds apart). These turtles should be spawned at coordinates (0, 1) and should move in a horizontal path at a speed of 1 m/s. [10%]

- Implemented in `ConveyorBelt.py`
  - ConveyorBelt operation at 2Hz and generates a random number between 10 and 16 until the next object is spawned: `next_spawn = random.randint(10, 16)`
  - Hence, a new object is spawned every 5-8 seconds
- Function `spawn()`
  - Calls turtlesim `/spawn` service with proper coordinates and orientation
  - Creates a Publisher on `/<object_name>/cmd_vel` to move objects

- Function `kick()`
  - Iterates over existing object Publishers every step and ensure object movement
- Publishes to `/conveyor_belt/spawned`
  - To make turtle2 aware of new objects
  - Isolate turtle2 objects from turtle 1 objects
- Subscribes to `/conveyor_belt/produce`
  - To listen if turtle2 is instantiated and ready to collect objects
- Subscribes to `/conveyor_belt/remove`
  - To listen which objects turtle2 collected from the belt
  - To call turtlesim `/kill` service to destroy object

5. Ensure that "conveyor belt" turtles are collected by turtle2 when they reach the collection point at coordinates (5.5, 1). [10%]

- Implemented in `ConveyorBelt.py` in function `spawn()`

6. Create a launch file that will start your simulation. When executed, it should:
   - Start the turtlesim simulator. [2%]

- Implemented in `assignment1.launch`
  `<node pkg="turtlesim" name="sim" type="turtlesim_node"/>`

   - Spawn turtle2 (the "fixed-trajectory robot") at coordinates (5.5, 3) and orientation $-\pi/2$. [2%]

- Implemented in `SpawnTurtle2Client.py`
- `<node pkg="assignment1" name="spawn_turtle2_client" type="SpawnTurtle2Client.py" output="screen"/>`

   - Spawn a random number of additional turtles (between 5 and 10), randomly distributed over the workspace. These are the objects to be collected by turtle1 (the "mobile robot"). [2%]

- Implemented in `SpawnObjectsClient.py` and `SpawnObjectsServer.py`
- `<node pkg="assignment1" name="spawn_objects_server" type="SpawnObjectsServer.py" output="screen"/>`
- `<node pkg="assignment1" name="spawn_objects_client" type="SpawnObjectsClient.py" output="screen"/>`

   - Activate turtle1, i.e. set it on a trajectory to collect all required objects and return to its starting point (providing progress reports on its journey between objects as outlined above). [7%]

- Implemented in `CollectionActionServer.py` and `Turtle1Client.py`
- `<node pkg="assignment1" name="turtle1_collection_action_server" type="CollectionActionServer.py" output="screen"/>`
- `<node pkg="assignment1" name="turtle1_client" type="Turtle1Client.py" output="screen"/>`

- Spawn "conveyor belt" turtles at random intervals and set them on a horizontal trajectory ("along the conveyor belt") as outlined above. [2%]

- Implemented in `ConveyorBelt.py`
- `<node pkg="assignment1" name="conveyor_belt" type="ConveyorBelt.py" output="screen"/>`

- Ensure that objects (turtles) are removed from the simulation once collected. Note: you should verify in your code that the object has been successfully collected before removing it from the simulation, i.e. verify that it has been reached by the relevant "robot" (turtle1 or turtle2). [15%]

- Implemented in `CollectionActionServer.py` and `Turtle2Client.py`
- `CollectionActionServer.py`
  - Tracks the Euclidean distance between turtle1 and the next target object.
  - If distance is within a small tolerance it stops turtle1 and sends a goal result to turtle1
  - Only when turtle1 received an action result with `result.success == True`, it continues to request the `/kill` service
- `Turtle2Client.py`
  - Tracks the Euclidean distance between turtle2 and the conveyor belt objects
  - If a conveyor belt object is in a certain distance from turtle2, turtle 2 moves down towards the belt.
  - Turtle 2 measures distance to target object.
  - If distance is within a small tolerance turtle2 Publishes to `/conveyor_belt/remove`, based on which ConveyorBelt will `/kill` the object

- Retrospectively, a different, maybe more intuitive approach, could have been to let the ConveyorBelt node evaluate turtle2-to-object distance and instruct turtle2 movement. Such implementation would have bundled all major decision-making into the ConveyorBelt object.