Multi-Agent Systems Using Kobuki Turtlebots: Leader-Follower Premise

Sinchiguano Cesar

***Abstract*— Multi-agent system is a topic that has generated a lot interest in the research community. This interest is because a multi-agent systems present a more robust and cheaper solution to a certain tasks that are better performed using several low-cost robots rather than a single one. Multi-agent system can be achieved through many approaches. The approach used in this paper is the one based on the leader-follower premise.**

I. Introduction

Multi-robot systems present a more robust and cheaper solution to certain tasks that are better performed using several low-cost robots rather than a single one. This gives rise to the formation control problem. The formation problem has been regarded as an important problem in multi-robot systems where the goal is to make a fleet of autonomous mobile robots move toward and maintain a desired geometric shape. According to the survey presented in (Guanghua et al., 2013)[1], and the references therein, formation structure can be divided into three strategies: the leaderfollower strategy, the behavioural and the virtual structure approaches. Several approaches have been proposed in the literature to solve this problem. However, most of the existing literature tackle the theoretical side of the problem mainly the controller design. Nevertheless, some of them have carried on real experiments to prove the effectiveness of their proposed controller.

In this paper, we propose the use of ROS as a new framework so that real experiments for the formation control problem can be conducted effectively. Due to its simplicity and scalability, the leader-follower approach is considered in this paper also. However, the proposed framework can be extended so that other formation strategies or consensus algorithm can be implemented. The rest of this paper will be organized as follow: The introduction about ROS concepts is highlighted in Section II, with the software setup in Section III, followed by a detailed explanation of the software processes and the needed ROS packages are presented in Section IV, finally conclusions drawn from this experiment and some future work directions in Section V

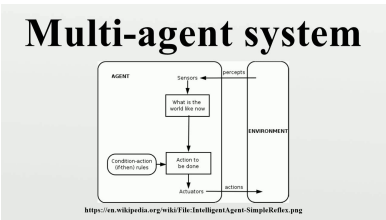

Fig. 1. An example of multi-agent system.

II. Backgroung

1) Robotic Operating System (ROS)

ROS is a flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behaviour across a wide variety of robotic platforms. It is based on the concepts of nodes, topics, messages and services. A node is an executable program that performs computation. Nodes need to communicate with each other to complete the whole task. The communicated data are called messages. ROS provides an easy way for passing messages and establishing communication links between nodes, which are running independently. They pass these messages to each other over a Topic, which is a simple string. However, topics are asynchronous, synchronous communication is provided by services. Services act in a call-response manner where one node requests that another node execute a one-time computation and provide a response. For more details about ROS, the reader can refer to [2].

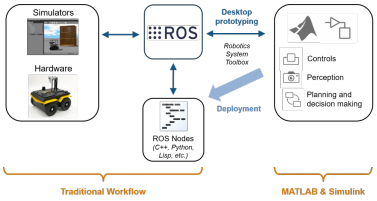

Fig. 2. A ROS Overview [2]

2) Architecture of the System

The objective of this section is to introduce an architecture for coordinating multi-agent systems. The proposed architecture subsumes leader-following approach, where one of the vehicles (Turtlebot robot or agent) is designated as the leader, with the rest of the vehicle designated as followers.The proposed architecture is modular.

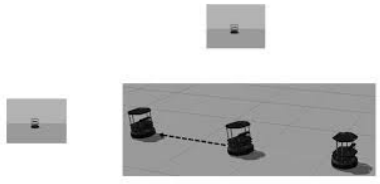

Fig. 3. Leader-Follower Premise.

III. Experimental setup

The test space consist of two Kobuki TurtleBots. It is a low-cost, open source differential drive robot. It consists of a mobile base, a RGB-D sensor and a CPU making it a good entry-level mobile robot platform. The Kobuki was chosen because it is an open source UGV (Unmanned ground vehicle) platform, making it perfect for research and development. The Kobuki SDK (Software development kit) is based on ROS, which is the preferred development platform by the Intelligent and Mobile Robotics Group because of its intuitive publisher/subscriber message passing structure that allows robust and simple communication within multiple facets of a robotic system. The Fig 2 shows two Turtlebot robots which are equipped with Kinect Sensors. The robots run ROS (Robot Operating System), and are supported by various ROS libraries.

Fig. 4. Two turtlebots used in the experimental test bed.

IV. Software Processes

In addition to the basic nodes called turtlebot_bringup needed for running the robots, four additional nodes are indispensable for controlling the robots to achieve formation. The multi_agents node for deploying the Turtlebot robots to work with, the tf_turtle responsible for keeping track of the robot's postures, multi_master node for data transmission and the turtle node that executes the control algorithm. The nodes are running simultaneously and thus they have to communicate to each other through ROS topics or ROS services as depicted in fig.3. Note that on each robot, all the nodes are executed under a specific namespace for the robot. The ROS parameter tf_prefix is exclusive for each robot as well.

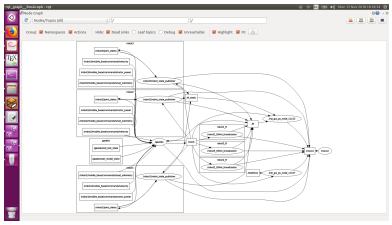

Fig. 5. A rosgraph showing what talks to what.

1) **Simulating Multiple Turtlebots in Gazebo**
This is an example of simulating and controlling multiple Turtlebot robots. It is built as an extension of the Simulating Turtlebot tutorial on [4].
   a) **Quick Start**

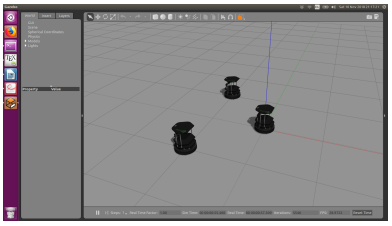

Fig. 6. A general view of the system.

The main.launch file is a working example that instantiates two simulated Turtlebots in Gazebo.
You will first need to install the following packages in your workstation:

i) turtlebot[6].
ii) kobuki[8].
iii) turtlebot_simulator[7].
iv) turtlebot_interactions[9].

Then after you have these packages in your workspace, place the multi_agents[5] package in your src diretory.
As long as all of the system dependencies of your packages are installed, we can now build your new packages.
In a catkin workspace

i) $ catkin_make.

Before continuing source your new setup.*sh file:

i) $ source devel/setup.bash.

To make sure your workspace is properly overlayed by the setup script, make sure ROS_PACKAGE_PATH environment variable includes the directory you are in with the following command:
$ echo $ROS_PACKAGE_PATH

   b) **Description: multi_agents package**
If everything was done with the commands above then it should just work. From there, to learn what it does, I woud recommend going over the multi_agent.launch, one_turle.launch and main.launch files those who can be located inside your catkin_ws with the following command:
$ cd src/multi_agents/launch
which will lead you to the launch files mentioned above. All of them are binding one another. The only one that it should be launched is the one named as main.launch file which does the heavy lifting of setting up each Turtlebot. Here is a high-level outline of what this package does.
main.launch:

i) Starts Gazebo (both the sim engine and the gui)
ii) Start some visualization and debugging tools (rviz,rqt_console, etc.)
iii) multi_agents.launch
   A) Accept namespace and initial pose arguments from one_turtle.launch.
   B) Set the tf_prefix based on the namespace

C) Call a xacro robot_description with namespace and tf_prefix arguments
D) Spawn the Turtlebot model in Gazebo using gazebo_ros
E) Start a robot_state_publiser node

Here are some screen captures and images to illustrate what you should see if things are working properly.
Four Turtlebot robots in Gazebo

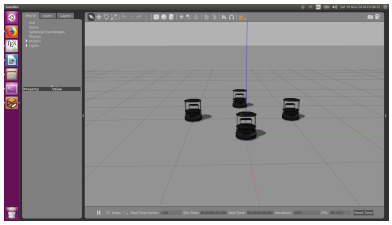

Fig. 7. A general view of the system with four multi-agents.

2) **Transform**
When doing tasks with a robot it is crucial that the robot be aware of where it is itself as well as where the rest of the world is in relation to itself. For the leader-follower premise each robot takes another neighboring robot as a leader to determine its motion so that for the control laws proposed in this paper show that the leaders relative coordinates and velocities are needed in the control laws implemented on the followers. The package called tf_turtle will deal with it. It will start a static_transform_publisher node to establish the tf transfrom from the world to the specific odometry frame e.g robot2_tf/odom.

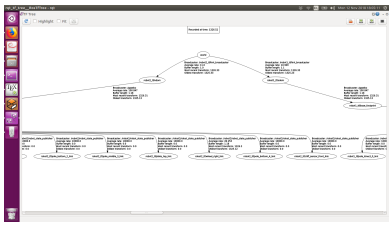

Fig. 8. View from rqt_tf_tree of the tf arrangement.

a) **Quick Start**
The robot_1.launch file is a working example that initialize the transform node for two Turtlebot robots. You will first need to install the following packages in your workstation:

i) geometry[10].

Then after you have these packages in your workspace, place the tf_turtle[11] package in your src diretory.
As long as all of the system dependencies of your packages are installed, we can now build your new package.
In a catkin workspace

i) $ catkin_make.

Before continuing source your new setup.*sh file:

i) $ source devel/setup.bash.

3) **Communication data**
A robust communication between the Turtlebot robots is needed in order to transmit the leaders postures and velocities to its follower. These data are essential for controlling the follower to keep the desired separation and bearing with the leader. The leader-follower premise is implemented using two methods: (1)Single-master system, and (2)multi-master system. The methods aforementioned make use of the distributed computing and communication architecture of the Robot Operating System (ROS). The detail of each implementation and their respective use are presented next in this section.

a) **Single Master System**

In a single master system, ROSCORE runs on one machine which is called the master. Other nodes work in a distributed similar fashion on different machines. The nodes can run anywhere on the network except the driver nodes, which runs on the system that is directly connected to the hardware. All the nodes need to connect to the master. They connect via ROS_MASTER_URI which can be set in .bashrc file of the respective machines as shown below. All the machines in the network have a bidirectional connection with each other. Also, the host IP and the master IP will be same in case of the master machine.

i) **Quick Start**
The following commands are required to be executed to activate the single master system: **For computer one (turtlebot@turtle01)**. export ROS_MASTER_URI=http://10.37.2.152 :11311
export ROS_IP=10.37.2.152
For computer two (turtlebot@turtle02). export ROS_MASTER_URI=http://10.37.2.152 :11311
export ROS_IP=10.37.2.243

For more details about NetworkSetup the reader to should refer to the ROS/NetworkSetup [11]

b) **Multi Master System**
Many of the limitations of a single master system can be overcome by having multiple masters running their own independent roscore. This makes the system robust as the failure of one will not lead to the failure of the complete system. Since the visibility of topics is limited to the scope of each roscore environment, there are no namespace conflict with topics in a multi-master system.
To implement a multi-master System, a package called multimaster_fkie is needed [12] and is already installed from source in each one of the Turtlebot. This allows two important pro-cesses, master_discovery and master_sync to run simultaneously. The function of master_discovery is to send multicast messages to the network so that all

roscore environments become aware of each other. The other process called master_sync enables us to select which topics can shared between different roscore. Without master_sync node, no information can be accessed by other roscores.

i) **Quick Start**
The following commands are required to be executed to activate multi-master mode in each machine.
For computer one (turtlebot@turtle01).
export ROS_MASTER_URI=http://localhost :11311
export ROS_IP=10.37.2.152
rosrun master_discovery_fkie master_discovery
rosrun master_sync_fkie master_sync
For computer two (turtlebot@turtle02).
export ROS_MASTER_URI=http://localhost :11311
export ROS_IP=10.37.2.243
rosrun master_discovery_fkie master_discovery
rosrun master_sync_fkie master_sync

For more details about the configuration of the multi-master System, the reader to should refer to multimaster_fkie[11]

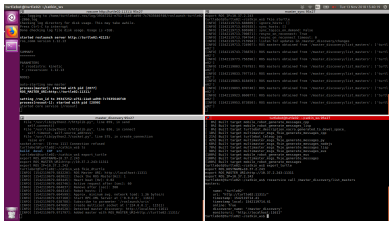

Fig. 9. A view of a successfull setup of the multimaster_fkie.

V. Conclusion

Multi-agent system is a topic that has generated a lot interest in the research community. This interest is because a multi-agent systems present a more robust and cheaper solution to a certain tasks that are better performed using several low-cost robots rather than a single one. Multi-agent system can be achieved through many approaches. The approach used in this paper is the one based on the leader-follower premise.

The details of implementation for both simulation as well as actual experiment is provided which will be useful for students and practicing engineers alike. These details provide an insight into the working of each of the these modes of operation allowing us to identify the strengths and weaknesses of each one of them.

Multi-master system solution is proposed to facilitate the communication amongst the agents based on decentralized architecture by using the FKIE package. The main advantages of the presented solution are modularity and its wide applicability. Modularity allows to increase the number of robots with its own rosmaster capability without modifying it from the core or facing the problems presented with a single rosmaster system. Moreover, the proposed multi-master solution permits to accomplish the assigned mission also in case of communication or vehicle faults. Thanks to the FKIE package the formation of vehicles are both more fault tolerant (through redundancy achieved with a multi-master system) and more efficient (through parallelism) than single vehicles.

As recommendation to successfully accomplish the coordination tasks, particular attention must be paid to the localization problem. Indeed, the most straightforward approach for performing formation control would be to measure the absolute position of each agent and use that information to keep the relative distances at the desired values. This approach, however, can not be used for the Turtlebot vehicle as the odometry sensor is prone to uncertainty in its measurement, and thus a suitable localization procedure is necessary.

References

[1] https://www.researchgate.net/publication/261160544_Study_on_Formation_Control_of_Mu Robot_Systems
[2] https://blogs.mathworks.com/racing-lounge/2017/11/08/matlab-simulink-ros/
[3] http://www.ros.org/about-ros/
[4] http://wiki.ros.org/turtlebot/Tutorials/indigo
[5] https://github.com/Sinchiguano/Multi-agent-system/tree/master/src/multi_agents
[6] http://wiki.ros.org/turtlebot
[7] http://wiki.ros.org/turtlebot_simulator
[8] http://wiki.ros.org/kobuki
[9] http://wiki.ros.org/turtlebot_interactions
[10] https://github.com/ros/geometry
[11] http://wiki.ros.org/ROS/NetworkSetup
[12] multimaster FKIE, http://wiki.ros.org/multimaster_fkie
[13]