Operation Of and Communication Between Turtlebots Using ROS

Daniel Heideman

1 Summary

This report covers the test code written to get multiple turtlebot robots moving and communicating with each other through Robot Operating System (ROS). Issues that were challenging or not well documented on the ROS wiki are described here. The tests allow multiple turtlebots to run with the same master, demonstrate remapping and publishing to more than one topic, allowing multiple computers to communicate through ROS, and moving based on subscribed messages.

2 Big Picture

The ultimate goal of the use of turtlebots and ROS in Dr. Cortes's and Dr. Martinez's labs is to test multi-robot control systems with real robots. The current goal is to get multiple turtlebots to work together using SLAM algorithms to map out a room and create one master map from which the turtlebots can all navigate. A SLAM localization algorithm already exists for ROS, so our work will focus on combining the data into a master copy.

A TurtleBot 2, with Microsoft Kinect 3d sensor and Kobuki base

3 My Contributions

My work on the turtlebots is involved in the communication between nodes and robots within ROS and with moving the turtlebot along with Randy Lewis. I have been looking specifically into motion, while Randy has been looking at image data from the Kinect sensor. Both will form the base of the lab's ROS system, upon which we will be able to write more complex nodes with test algorithms.

To communicate between individual programs, or nodes, ROS provides a framework for reading and writing data to topics, to which each node can publish or subscribe. I have been working on getting subscriber and publisher nodes working in ROS, specifically how to read and use data from subscribers. When we can act based on subscribed data, we will be able to communicate between nodes and robots and read sensor data. Before we can get any high-level control algorithms running, we need to know where we are and how to communicate.

I have also been working with getting the turtlebots initialized and moving. The turtlebots are not set up to work with more than one at a time out of the box, so some tweaking has been required to get multiple turtlebots driving together. With a few minor tweaks to the initialization, we have gotten this to work.

4 Methodology

This section contains a list and high-level descriptions of test programs completed during Winter Quarter 2014.

4.1 Turtlebot named launch files

To run multiple robots with the same master, the turtlebot_bringup launch file nodes need to be renamed, as ROS does not allow multiple nodes running at the same time to have the same names. This is done by applying a namespace with the <group> tag. For titianbot, all nodes and topics are preceded by /titianbot, such as “/mobile_base/commands/velocity” becoming “/titianbot/mobile_base/commands/velocity”.

```
<launch>
  <group ns="titianbot">

    <include file="$(find turtlebot_bringup)/launch/minimal.launch"/>
    <include file="$(find turtlebot_bringup)/launch/3dsensor.launch"/>

  </group>
</launch>
```

titianbot.launch

4.2 Two robots spinning from the same node

You cannot remap the same topic twice, as remapping the first time means that the original topic is never published to. To get multiple robots to perform the same motions, the command needs to be sent on multiple topics.

In the example below, two instances of the same node are run from the launch file, so the command can be remapped to both turtlebots. These nodes require different names as ROS does not allow two nodes with the same name to run concurrently, as noted before with the turtlebot launch files.

```
<launch>
  <node pkg="turtlebot_test_daniel" name="boticelli_circle" type="circle">
    <remap from="/mobile_base/commands/velocity"
to="/boticellibot/mobile_base/commands/velocity"/>
  </node>

  <node pkg="turtlebot_test_daniel" name="giottobot_circle" type="circle">
    <remap from="/mobile_base/commands/velocity"
to="giottobot/mobile_base/commands/velocity"/>
  </node>
</launch>
```

circle_in_place.launch

4.3 Getting nodes on different computers to talk

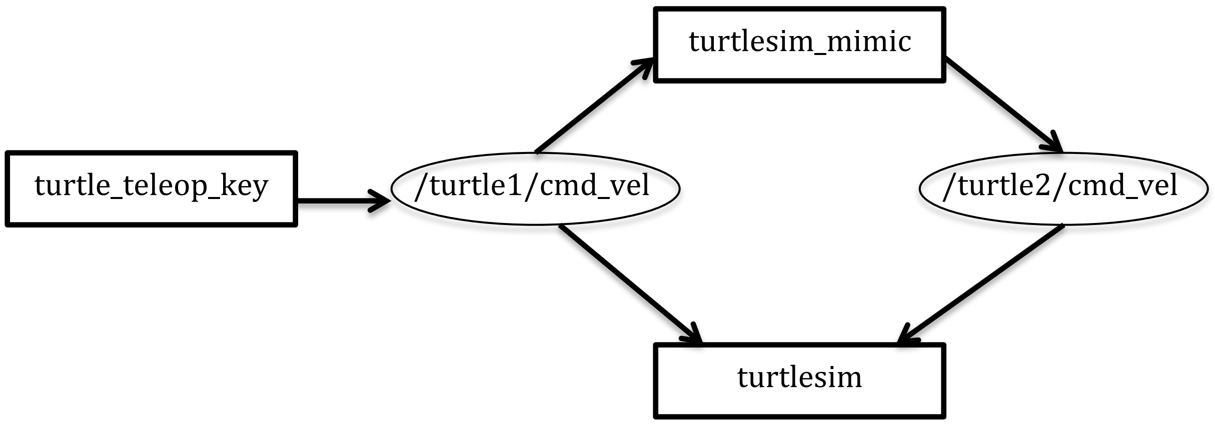
For two nodes to be able to talk in ROS, they must have the same master. However, they also need to have hostnames. Hostnames are names, not addresses, so "http://titianbot.dynamic.ucsd.edu" is not a valid name, but "titianbot" is. The exact hostname does not appear to matter, so long as it is a unique, valid name and the computers have the same master.

<u>Mencia</u>
ROS_MASTER_URI="http://mencia.ucsd.edu
ROS_HOSTNAME="mencia"

<u>Titianbot</u>
ROS_MASTER_URI="http://mencia.ucsd.edu
ROS_HOSTNAME="titianbot"

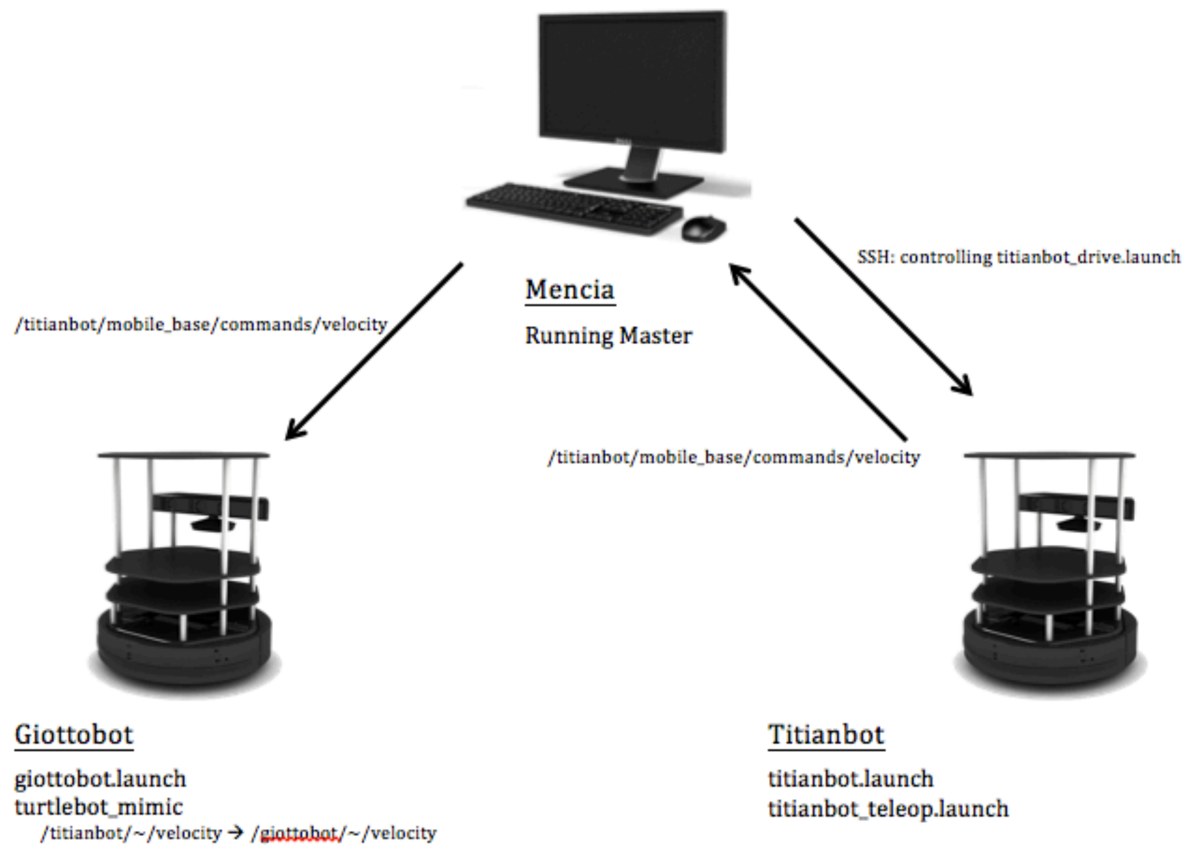
<u>4.4 Subscriber node-based mimic</u>

Although a topic cannot be remapped more than once in a launch file, this does not stop a node from subscribing to the topic and republishing the data on a different topic. To demonstrate this node-based topic copying and test methods of turtlebot drive, we designed the subscriber node-based mimic test based on mimic.cpp from ros.org and Evan's spawner.cpp.



Because subscribing is more complicated than publishing in ROS, the mimic test was first made to work with turtlesim, as turtlesim does not have the added complexity and possible points of failure of moving a robot the turtlebots have. Here, the node subscribes to turtle1's velocity topic and re-publishes the message contents to turtle2's velocity topic. Turtle1 is driven manually with the turtle_teleop_key node. Turtle2 then moves in roughly the same way that turtle1 does, with variations stemming from message lags and starting position and orientation. The publisher-subscriber worked sufficiently well, although problems encountered with this setup are discussed in-depth in the pitfalls section.

The turtlebot node works very similarly, with an initial command sent through keyboard_teleop and a node, turtlebot_mimic, copying data meant for one turtlebot and delivering it to the other. Added are the movement of physical robots and the operation of multiple nodes across three computers.


Mencia
Running Master
SSH: controlling titianbot_drive.launch
/titianbot/mobile_base/commands/velocity
/titianbot/mobile_base/commands/velocity
Giottobot
giottobot.launch
turtlebot_mimic
/titianbot/~/velocity → /giottobot/~/velocity
Titianbot
titianbot.launch
titianbot_teleop.launch

To simulate one robot mimicking another, giottobot runs the turtlebot_mimic node locally, rather than on mencia, the master. Likewise, titianbot is driven with a teleop node locally, controlled on mencia over ssh. All communications go through the master on mencia, but giottobot's mimicking is self-contained and independent from titianbot. For example, if titianbot were set up to drive, giottobot could be set up to mimic it. But if giottobot is not present, nothing related to the mimicking ever happens, and titianbot moves alone. Conversely, if bellinibot were also given the turtlebot_mimic node and ran it locally, it could be added to the system with both giottobot and bellinibot mimicking titianbot independently of each other. This setup is thus also a test of a simple modular system.

5 Tips

I have found that while trying out a new part of ROS, it is best to try a test with parts you know are reliable and build from there. ROS is not very intuitive, so it is usually a good idea to test the basics first. For instance, a subscriber node might be tested by subscribing to something that you know is being published to consistently, and it might only write the data to the terminal. After this test, you at least know the subscriber works and that the problem does not lie elsewhere in a more complicated program.

This also applies to the turtlebots. The turtlebots are temperamental and will often not work for unknown reasons. Usually the problem can be resolved by waiting or by restarting the computers or mobile base. But if you are testing a subscriber on a turtlebot and it does not work, it is likely the problem lies in the turtlebot having trouble initializing, which has nothing to do with your code. This is why I used turtlesim to test the subscriber node-based mimic. Turtlesim does work much in the way the turtlebots do, but it is far more reliable and simpler.

6 Pitfalls

In my Turtlesim tests with the subscriber node-based mimic, I found that listening to topics that are not constantly being published can present some problems with the way we have been working with subscribers. In our present way, we declare a public variable in a class, set the variable in the subscriber callback function, and read it in our main() loop.

When subscribed to a topic that is published constantly and frequently, every time the program loops through and the ROS::spinOnce() command is called, the data we're subscribed to is read and the variable we hold is updated, and we have this new value to act on. Even if the program loops faster than the data is published, so long as the data is published fast enough, at least a couple times per second, the amount of time the program is acting on old data is so short that we cannot notice it visually in the performance of the turtlesim turtles.

In the case of the turtles in turtlesim, the turtle_teleop_key node only publishes data when an arrow key is pressed and the turtles perform the latest command for about a second until they stop. For example, if we tell turtle1 to drive forward by hitting the up arrow once, it will drive forward for a second and then stop, but the only message sent is a brief "forward" command, which the turtle follows for a second before stopping on its own. When we read the data in our subscriber node, the last command sent is recorded into a public variable and then executed in the next loop of our main() while loop. As this variable is not overwritten in my test nodes, the last message is published continuously until a new message has been received. So whereas turtle1 will go forward for one second and then stop, turtle2, which is mimicking turtle1, will drive forward until another command is issued to turtle1. While there are ways to figure out when a message should stop being published, this means that simple publishing/subscribing pairs may not always be reliable and checks for acting on old data should be used.

This same problem is encountered when messages are interrupted by bad Wi-Fi connections. For instance, when titianbot was being driven with a teleop node running onboard and giottobot was running a mimic node, whenever titianbot or giottobot lost connection with the master on mencia, the subscriber node on giottobot would stop receiving new commands and would instead continue executing the last command it received. And because the connection was bad, it would be impossible to immediately remotely stop either robot. Most of the time this would occur when the turtlebots were stationary, but this ended up being a problem with the robots crashing into each other when they lost connection while moving.

7 Conclusions & Future Directions

These tests have proven that we can operate multiple turtlebots and have them communicate between themselves and a central master and have provided a base for future, more complicated nodes with more sophisticated intercommunication. As these nodes and launch files are all just tests, designed to figure out how to make the robots drive and communicate, they are not designed to be used to implement and test algorithms. However, we hope to use these to learn how to build such programs that can be useful for such testing later.

With these programs, we can move the turtlebots and have them share information and send commands, but we cannot yet really sense where the turtlebots are or where anything else is. We do not even know if the turtlebots are doing the same things. To run any of the end-goal programs on these turtlebots, we will need to first get them to interact with their surroundings through the sensors. This will involve heavy use of the Kinect and the odometer, as these are the basis of the SLAM algorithms we will ultimately be testing.

The subscriber nodes provide the programming base required for reading sensor data, as these are sent as messages from the Kinect and sensors on the Kobuki base. A test I would like to try out next would be to subscribe to /odom data to track how far the turtlebot has moved and stop the robot after moving a set distance. The next step from there would be more complicated motions to figure out how best to drive based on the /odom data, eventually reading in data from other sensors as well. The end goal of such a series of tests might be to drive in a square 1m on a side, stopping whenever it hits something with its bump sensor.

I also plan to revise the turtlebot_mimic node to account for a lack of incoming data. What most nodes do when they lose connection is stop. I believe that when titianbot lost contact while being teleoperated directly, it would stop moving almost immediately, whereas giottobot, running my custom mimic node, would just keep going. These revisions would introduce some protocol where the node only broadcasts old messages for a certain period of time before stopping, so that in the event that the node stops receiving messages, the turtlebot does not damage itself.

There is also a serious problem with the direct remote operation of the turtlebot drive and other time-critical nodes over Wi-Fi where the network we currently use does not offer reliable enough connection to send so much time-critical data. The streams are frequently interrupted, causing the robots to either stop in their tracks or continue on blindly. There does not appear to be much of a lag time for communications within one computer, although I have not tested this on a non-master computer.

A better approach would probably be to send the turtlebots data about where to go and where objects and other robots are in relation themselves and have them direct their drive systems from there. This way we can send data that updates an internal map that does not necessarily need to arrive constantly and on time, rather than having the turtlebots be fully controlled remotely and have to send constant drive data to each robot. If there is a short interruption in communication, the turtlebot will just continue to run off its old map rather than completely screwing up and stopping or crashing before new data is received. This will be the subject of a later test.

Our final goal involves the implementation of mostly-autonomous turtlebots using the master only for communication, not for receiving commands. With this setup, the turtlebots would be controlled by onboard algorithms, but when they wanted to know their position they might ask the master, or consult the other turtlebots. Whatever sensors or calculations the other turtlebots have access to might also be available to our individual turtlebot, just as onboard sensors work in ROS with their states being constantly published and accessible to the local computer. It also could be used as a virtual sensor, such as simulated GPS provided by an observing computer. The turtlebots would be making decisions based on the data they receive much like the “better approach” above, but without the central control.

8 Referenced Code

8.1 mimic.cpp

http://docs.ros.org/groovy/api/turtlesim/html/mimic_8cpp_source.html

```
#include <ros/ros.h>
#include <turtlesim/Pose.h>
#include <turtlesim/Velocity.h>

class Mimic
{
public:
  Mimic();

private:
  void poseCallback(const turtlesim::PoseConstPtr& pose);

  ros::Publisher vel_pub_;
  ros::Subscriber pose_sub_;
};

Mimic::Mimic()
{
  ros::NodeHandle input_nh("input");
  ros::NodeHandle output_nh("output");
  vel_pub_ = output_nh.advertise<turtlesim::Velocity>("command_velocity", 1);
  pose_sub_ = input_nh.subscribe<turtlesim::Pose>("pose", 1, &Mimic::poseCallback,
this);
}

void Mimic::poseCallback(const turtlesim::PoseConstPtr& pose)
{
  turtlesim::Velocity vel;
  vel.angular = pose->angular_velocity;
  vel.linear = pose->linear_velocity;
  vel_pub_.publish(vel);
}

int main(int argc, char** argv)
{
  ros::init(argc, argv, "turtle_mimic");
  Mimic mimic;

  ros::spin();
}
```

8.2 spawner.cpp

Evan Gravelle

```
#include <ros/ros.h>
#include <turtlesim/Spawn.h>
#include <turtlesim/Pose.h>
#include <turtlesim/Velocity.h>
#include <stdlib.h>
#include <math.h>


class SpawnerClass {
public: //Things that can be seen outside of class
  SpawnerClass();
  turtlesim::Pose pose1, pose2;
  private: //Things only seen by the class
  ros::Subscriber sub1, sub2;
  void poseCallback1(const turtlesim::PoseConstPtr& msg);
  void poseCallback2(const turtlesim::PoseConstPtr& msg);
};


SpawnerClass::SpawnerClass() {
  ros::NodeHandle n1, n2;
  //define sub as a subscriber to pose
  sub1 = n1.subscribe("/turtle1/pose", 1000, &SpawnerClass::poseCallback1, this);
  sub2 = n2.subscribe("/turtle2/pose", 1000, &SpawnerClass::poseCallback2, this);
  //ROS_INFO("The turtle is currently at x=%.3f y=%.3f", pose1.x, pose1.y);

}


void SpawnerClass::poseCallback1(const turtlesim::PoseConstPtr& msg) {
  pose1 = *msg;
  ROS_INFO("The turtle is currently at x=%.3f y=%.3f", pose1.x, pose1.y);
}

void SpawnerClass::poseCallback2(const turtlesim::PoseConstPtr& msg) {
  pose2 = *msg;
  ROS_INFO("The turtle is currently at x=%.3f y=%.3f", pose2.x, pose2.y);
}


int main(int argc, char **argv) {
```

```
ros::init(argc, argv, "spawner");
ros::NodeHandle n1, n2;
ros::Publisher pub1, pub2;

//define srv as calling the existing service Spawn
ros::ServiceClient spawnClient = n2.serviceClient<turtlesim::Spawn>("/spawn");
turtlesim::Spawn srv;

//seeds the random variable generator
srand(time(0));

//defines the request variables
srv.request.x = 10 * float(rand())/float(RAND_MAX);
srv.request.y = 10 * float(rand())/float(RAND_MAX);
srv.request.theta = 2 * 3.14159 * float(rand())/float(RAND_MAX);
srv.request.name = "turtle2";

//waits half a second
ros::Duration(0.5).sleep();

bool success = spawnClient.call(srv);
if(success) {
  ROS_INFO("Spawned a turtle named %s at (%f,%f).",
           srv.response.name.c_str(), srv.request.x, srv.request.y);
} else {
  ROS_INFO("Spawn failed.");
}

//Calls the function SpawnerClass to subscribe to both
SpawnerClass hi;

//Creates pubs, sets spin rate
//define pub as a publisher outputting command_velocity
pub1 = n1.advertise<turtlesim::Velocity>("/turtle1/command_velocity", 1000);
pub2 = n2.advertise<turtlesim::Velocity>("/turtle2/command_velocity", 1000);

ros::Rate rate(60);

turtlesim::Velocity vel1, vel2;
double dx, dy;


while (ros::ok()) {
  ROS_INFO("IN WHILE LOOP The turtle is currently at x=%.3f y=%.3f", hi.pose1.x,
hi.pose1.y);

    dx = hi.pose2.x - hi.pose1.x;
    dy = hi.pose2.y - hi.pose1.y;


    ROS_INFO("p1 is %f %f %f", hi.pose1.x, hi.pose1.y, hi.pose1.theta);
    ROS_INFO("p2 is %f %f %f", hi.pose2.x, hi.pose2.y, hi.pose2.theta);
    float theta1 = atan2(dy,dx); //between -pi and pi
    float theta2 = theta1 + 3.14159; //between 0 and 2*pi
    float delta = sqrt(pow(dx,2) + pow(dy,2));
    vel1.linear = fmin(2,delta/5);
    vel2.linear = vel1.linear;


    //depends on the robot's current theta!
    if (dx < 0 && dy < 0) {
      vel1.angular = theta1 + 2*3.14159 - hi.pose1.theta;
      vel2.angular = theta2 + 2*3.14159 - hi.pose2.theta;
    }
    else if (dx < 0 ) {
      vel1.angular = theta1 - hi.pose1.theta;
      vel2.angular = theta2 - hi.pose2.theta;
    }

    pub1.publish(vel1);
    pub2.publish(vel2);
    ros::spinOnce();
    rate.sleep();
    }
}
```