CS 378: Autonomous Intelligent Robotics (FRI)

Dr. Todd Hester

Are there any questions?

Logistics

- Readings Monday
  - Pick your own paper from the wiki

- Post for teammates on Piazza
  - Project topics, skills

- Talks Tomorrow
  - Dr. Mohan Sridharan
  - Towards Autonomy in Human-Robot Collaboration
    - 11 am, ACES 2.402
  - Integrating Answer Set Programming and Probabilistic Planning on Robots
    - 3 pm, ACES 2.402

Assignment 1

- Laptop Issues
  - There will be issues
  - Start early
  - Strongly encouraged to use lab machines
- Debug and Troubleshooting
  - In the lab
  - Post on Piazza
  - Copy and paste from terminal
- Cutting-Edge & Complex Code
  - There will be problems
  - START EARLY
  - Get help IN PERSON - Come to office hours
- Now due tomorrow 4pm!

Assignment 1

- Gazebo and Rviz
- Any interesting behaviors driving the robot around?
- Any issues with navigation?
- Try blocking the robot's path?
- Any issues navigating with the Kinect?

Today

- ROS Tutorial
  - Setting up two simple nodes to send messages to each other
- Kalman Filters

Example 1 - Publisher and Listener

- The first example is directly from ROS Tutorials
  - http://www.ros.org/wiki/ROS/Tutorials
- I *highly recommend* going through these tutorials on your own time
- We'll take a look at C++ tutorial today (Tutorial 11)
- If you are interested in using ROS in Python go through the Python tutorial (Tutorial 12). The tutorials are fairly similar

talker.cpp (intro_to_ros)

```cpp
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <sstream>

int main(int argc, char **argv) {
  ros::init(argc, argv, "talker");
  ros::NodeHandle n;
  ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
  ros::Rate loop_rate(1);
  int count = 0;

  while (ros::ok()) {
    std_msgs::String msg;
    std::stringstream ss;
    ss << "hello world " << count;
    msg.data = ss.str();
    ROS_INFO("%s", msg.data.c_str());
    chatter_pub.publish(msg);
    ros::spinOnce();
    loop_rate.sleep();
    ++count;
    }
  return 0;
}
```

listener.cpp (intro_to_ros)

```
#include "ros/ros.h"
#include "std_msgs/String.h"

void chatterCallback(const std_msgs::String::ConstPtr msg) {
  ROS_INFO("I heard: [%s]", msg->data.c_str());
}

int main(int argc, char **argv) {
  ros::init(argc, argv, "listener");
  ros::NodeHandle n;
  ros::Subscriber sub =
    n.subscribe<std_msgs::String>("chatter", 1000, chatterCallback);
  ros::spin();
  return 0;
}
```

talker.cpp

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <sstream>
```

- *ros/ros.h* is a convenience header that includes most of the pieces necessary to run a ROS System
- *std_msgs/String.h* is the message type that we will need to pass in this example
  - You will have to include a different header if you want to use a different message type
- *sstream* is responsible for some string manipulations in C++

talker.cpp

```
ros::init(argc, argv, "talker");
ros::NodeHandle n;
```

- ros::init is responsible for collecting ROS specific information from arguments passed at the command line
  - It also takes in the name of our node
  - Remember that node names need to be unique in a running system
- The creation of a ros::NodeHandle object does a lot of work
  - It initializes the node to allow communication with other ROS nodes and the master in the ROS infrastructure
  - Allows you to interact with the node associated with this process

talker.cpp

```
ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
ros::Rate loop_rate(1);
```

- NodeHandle::advertise is responsible for making the XML/RPC call to the ROS Master advertising std_msgs::String on the topic named "chatter"
- loop_rate is used to maintain the frequency of publishing at 1 Hz (i.e., 1 message per second)

talker.cpp

```
int count = 0;
while (ros::ok()) {
```

- *count* is used to keep track of the number of messages transmitted. Its value is attached to the string message that is published
- *ros::ok()* ensures that everything is still alright in the ROS framework. If something is amiss, then it will return *false* effectively terminating the program. Examples of situations where it will return false:
  - You *Ctrl+c* the program (SIGINT)
  - You open up another node with the same name.
  - You call *ros::shutdown()* somewhere in your code

talker.cpp

```
std_msgs::String msg;
std::stringstream ss;
ss << "hello world " << count;
msg.data = ss.str();
```

- These 4 lines do some string manipulation to put the count inside the *String* message
- *msg.data* is a *std::string*

talker.cpp

```
ROS_INFO("%s", msg.data.c_str());
chatter_pub.publish(msg);
```

- *ROS_INFO* is a macro that publishes an information message in the ROS ecosystem. By default *ROS_INFO* messages are also published to the screen.
  - There are debug tools in ROS that can read these messages
  - You can change what level of messages you want to be have published

- *ros::Publisher::publish()* sends the message to all subscribers

talker.cpp

```
ros::spinOnce();
loop_rate.sleep();
++count;
```

- *ros::spinOnce()* is analogous to the *main* function of the ROS framework.
  - Whenever you are subscribed to one or many topics, the *callbacks* for receiving messages on those topics are not called immediately.
  - Instead they are placed in a queue which is processed when you call *ros::spinOnce()*
  - What would happen if we remove the *spinOnce()* call?
- *ros::Rate::sleep()* helps maintain a particular publishing frequency
- *count* is incremented to keep track of messages

listener.cpp - *in reverse!*

```
int main(int argc, char **argv) {
  ros::init(argc, argv, "listener");
  ros::NodeHandle n;
  ros::Subscriber sub =
    n.subscribe<std_msgs::String>("chatter", 1000, chatterCallback);
  ros::spin();
  return 0;
}
```

- *ros::NodeHandle::subscribe* makes an XML/RPC call to the ROS master
  - It subscribes to the topic *chatter*
  - 1000 is the *queue size*. In case we are unable to process messages fast enough. This is only useful in case of irregular processing times of messages. Why?
  - The third argument is the *callback* function to call whenever we receive a message
- *ros::spin()* a convenience function that loops around *ros::spinOnce()* while checking *ros::ok()*

listener.cpp

```
#include "ros/ros.h"
#include "std_msgs/String.h"

void chatterCallback(const std_msgs::String::ConstPtr msg) {
  ROS_INFO("I heard: [%s]", msg->data.c_str());
}
```

- Same headers as before
- *chatterCallback()* is a function we have defined that gets called whenever we receive a message on the subscribed topic
- It has a *well typed* argument.

Running the code

- Build the example package
  - *rosmake intro_to_ros*
- In separate terminal windows, run the following programs:
  - *roscore*
  - *rosrun intro_to_ros talker*
  - *rosrun intro_to_ros listener*
- *To view messages:*
  - *rostopic list*
  - *rostopic echo chatter*

Example 2 - Adding a Messenger node

- A number of times in ROS you will have a bunch of nodes processing data in sequence. For instance a *blob detection node* provides the location of blobs *for every* camera image it receives
- To demonstrate this, we'll change our previous example in the following ways:
  - Introduce a *messenger* node that listens for messages on the topic *chatter* and forwards them on the topic *chatter2.* (I couldn't think of a cute name for this topic)
  - At the command line remap the listener to subscribe to *chatter2* instead of *chatter*

messenger.cpp (intro_to_ros)

```
#include "ros/ros.h"
#include "std_msgs/String.h"

ros::Publisher chatter_pub;
std_msgs::String my_msg;

void chatterCallback( const std_msgs::String::ConstPtr msg) {
  ROS_INFO("I heard: [%s]", msg->data.c_str());
  my_msg.data = msg->data + ". Dont kill the messenger!";
  chatter_pub.publish(my_msg);
}

int main(int argc, char **argv) {
  ros::init(argc, argv, "messenger");
  ros::NodeHandle n;
  ros::Subscriber sub =
    n.subscribe<std_msgs::String>("chatter", 1000, chatterCallback);
  chatter_pub = n.advertise<std_msgs::String>("chatter2", 1000);
  ros::spin();
  return 0;
}
```

Running the code

- You will have to execute the following steps to get this example working
- In separate terminal windows, run the following programs:
  - *roscore*
  - *rosrun intro_to_ros talker*
  - *rosrun intro_to_ros listener chatter:=chatter2*
  - *rosrun intro_to_ros messenger*

Review

- ROS is a peer-to-peer *robot middleware* package
- We use ROS because it allows for easier *hardware abstraction* and *code reuse*
- In ROS, all major functionality is broken up into a number of chunks that communicate with each other using messages
- Each chunk is called a *node* and is typically run as a separate process
- Matchmaking or bookkeeping between nodes is done by the ROS Master

Assignments Due Next Week

- HW1 - Due tomorrow 4pm
- Reading Due Monday night
  - Pick any paper you want!
- Add a new paper to the wiki (by class time Tuesday)
- Post Teammate Search
  - Project Topics, Skills
  - Thursday