CS 378 - Autonomous Vehicles in Traffic I
Week 4a - Introduction to ROS Programming

Logistics

- Meeting with CIA recruiter today from 3:15-4:00 PM
  - RLM 4.102 (Ground floor)
- iRobot Create demo today at 4:00 PM in ENS32.
- Programming assignment 1 due Wednesday
  - While writing out your test main function, print out the temperature you are trying to maintain
  - Print out the current temperature in the `update()` function once you receive it from `tick()`
  - Call `update()` a sufficient number of times to show that the temperature does not differ by more than 3 from maintained temperature.
  - I will upload an executable demonstrating my output by tonight

Today

- We'll go over a couple of C++ examples of nodes communicating within the ROS framework
- We will recap the concepts of ROS nodes, topics and messages.
- We'll see how to setup ROS and check out a copy of our code-base.

Review - ROS Overview

- 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 between nodes is done by the ROS Master
**Review - How ROS works**

**ROS Nodes**

- A node is a process that performs some computation.
- Typically we try to divide the entire software functionality into different modules - each one is run over a single or multiple nodes.
- Nodes are combined together into a graph and communicate with one another using streaming topics, RPC services, and the Parameter Server.
- These nodes are meant to operate at a fine-grained scale; a robot control system will usually comprise many nodes.

[http://www.ros.org/wiki/Nodes](http://www.ros.org/wiki/Nodes)

**ROS Topics**

- Topics are named buses over which nodes exchange messages.
- Topics have anonymous publish/subscribe semantics - A node does not care which node published the data it receives or which one subscribes to the data it publishes.
- There can be multiple publishers and subscribers to a topic:
  - It is easy to understand multiple subscribers.
  - Can't think of a reason for multiple publishers.
- Each topic is strongly typed by the ROS message it transports.
- Transport is done using TCP or UDP.

[http://www.ros.org/wiki/Topics](http://www.ros.org/wiki/Topics)
**ROS Messages**

- Nodes communicate with each other by publishing *messages* to topics.
- A message is a simple data structure, comprising typed fields. You can take a look at some basic types [here](http://www.ros.org/wiki/ Messages).
  - `std_msgs/Bool`
  - `std_msgs/Int32`
  - `std_msgs/String`
  - `std_msgs/Empty` (huh?)
- In week 8 we will look into creating our own messages.
- Messages may also contain a special field called header which gives a *timestamp* and *frame of reference*.

[http://www.ros.org/wiki/ Messages]

**Getting our code-base**

- You will need to get a copy of our code-base from our google code repository.
- We have started collecting documentation on the wiki:
- You need to decide whether you wish to work on a departmental machine or your own machine.
  - You'll need Ubuntu 10.04 LTS, 11.04 or 11.10 to install ROS Electric.
- Follow the relevant tutorials to get our code-base working.

**Example 1 - Publisher and Chatter**

- The first example is directly from ROS Tutorials:
- I *highly recommend* going through these tutorials on your own time.
- We'll take a look at C++ tutorial today (Tutorial 12).
- If you are interested in using ROS in Python go through the Python tutorial (Tutorial 11). The tutorials are fairly similar.
- All of today's tutorials are in the ROS package *intro_to_ros*. Once you have our code-base up, you can get to the package using the following command:
  - `roscd intro_to_ros`

**talker.cpp (intro_to_ros)**

```c++
#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();
    ROB_INFO("\%s", msg.data.c_str());
    chatter_pub.publish(msg);
    ros::spinOnce();
  loop_rate.sleep();
  ++count;
  }
  return 0;
}
```
**listener.cpp (intro_to_ros)**

```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());
}

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**

```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++

```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
  - We'll see an example of such an argument in the next example
- 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**

```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)
```
```cpp
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

    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
        • The reason we do it this way is that C++ does not have a good equivalent to the toString() function
    msg.data is a std::string
    • Aside: I typically use boost::lexical_cast() in place of the toString() function. lexical_cast() pretty much does the thing above for you (Look up this function if you are interested)
    
    ROS_INFO("%s", msg.data.c_str());
    chatter_pub.publish(msg);
    • ROS_INFO is a macro that publishes a 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
    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!

```cpp
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()
```

Running the code

- You will have to execute the following steps to get this
  example working
- After you download our 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

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)

```cpp
#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 + ". Don't 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`
  - `rostopic pub /chatter std_msgs::String ["Hello ROS!"]`
  - `rostopic pub /chatter std_msgs::String ["Hello ROS!"]`
  - `rosrun intro_to_ros listener chatter:=chatter2`
  - `rosrun intro_to_ros messenger`

Review

- We briefly went through ROS nodes, topics and messages.
- We also went through an example in C++ for setting up
  nodes within the ROS Framework.
- You should be able to try this code out. Let us know if you
  run into trouble setting up ROS or our code-base.