Primes Tutorial

This tutorial is an introduction to ROS and to version control systems that we can do as a group!

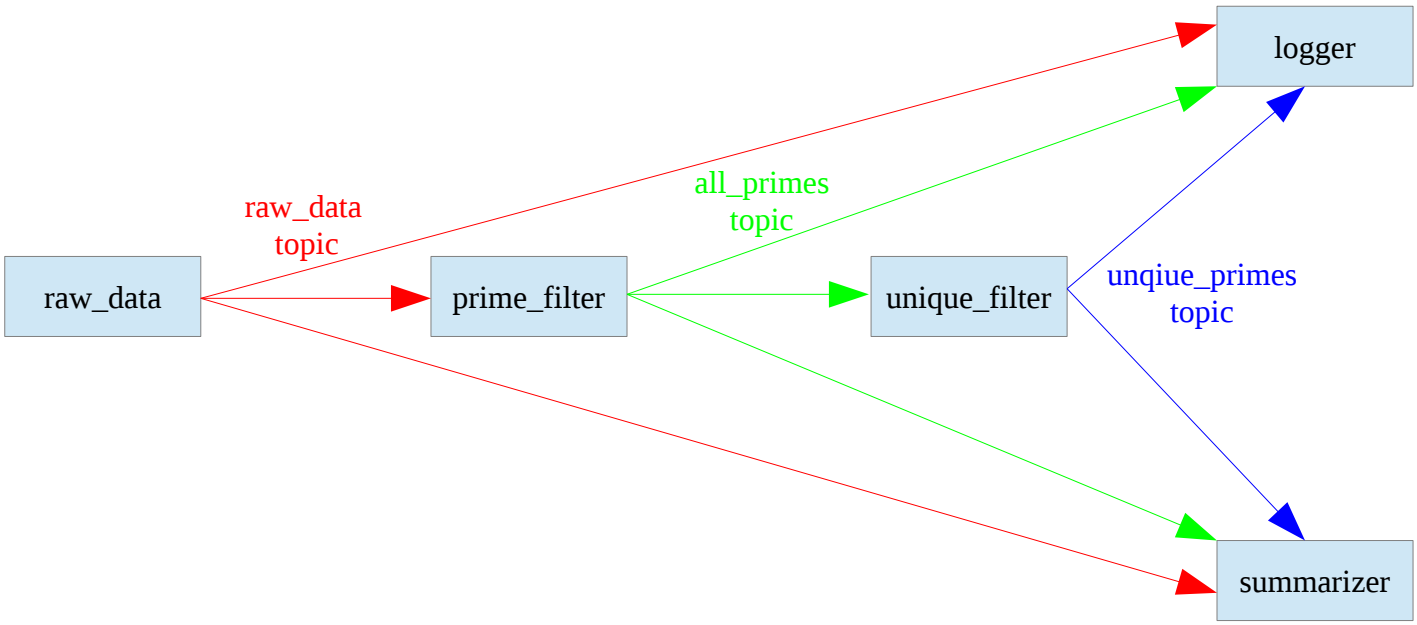
What is ROS?
ROS (Robot Operating System) is not actually an operating system. Instead, it's a framework that is designed to make robot programming easier. It's a bunch of tools and libraries that are useful for robot programming, combined with a particular convention for how to structure your programs: instead of writing one program to do everything, write a collection of small programs, called “ROS nodes”, which each accomplish one task. ROS provides a powerful message-passing system for the nodes to communicate with one another. We'll talk later about why this is a good architecture.

The Task
We'll be creating a simple ROS system that takes raw data from a “sensor”, and keeps a record of all of the unique primes seen in the data stream. This task will be split into five sub-tasks:

- raw_data module: pretend to be a sensor by generating a stream of random data
- prime_filter module: filter the data from the sensor, and keep only prime numbers
- unique_filter module: filter the list of primes, and keep only one occurrence of each number (so we get a list of *unique* primes)
- logger module: take the raw_data, prime_filter, and unique_filter streams, and write them to three separate files
- summarizer module: every second, print a status message to the screen showing how many raw data points, primes, and unique primes we've processed so far

Each sub-task will be assigned to a different team, and will be implemented in a ROS node.

System Overview



The raw_data node publishes random data on the raw_data topic. The prime_filter node subscribes the raw_data topic, and publishes all of the prime numbers it sees on the all_primes topic. The unique_filter node subscribes to the all_primes topic, and publishes all of the unique numbers it sees on the unique_primes topic. The logger node is subscribed to all three topics, so it can write the

three lists into files. The summarizer node is subscribed to all three topics, so it can count the elements in each list and periodically print the counts to the screen.

Subscribing and Publishing
The nodes in a ROS system exchange messages to communicate. There are many different types of messages, for exchanging different types of data. You can even define your own message types, that package together many different types of data. For this tutorial, our nodes only need to exchange integers, so we'll use the Int32 message (for 32-bit integers).

To use the Int32 messages, you'll need to add the following include at the top of your code:
```
#include "std_msgs/Int32.h"
```

To declare a new message and put data inside of it, do this:
```
std_msgs::Int32 msg; // This declares a new message called 'msg'
msg.data = 27; // This puts the number 27 inside the message
```

Messages in ROS are exchanged via *topics*. A node with a message will *publish* it to a particular topic. Any other nodes interested in those messages can *subscribe* to that topic, and they'll receive the messages. Every time you publish a message on a topic, it is delivered to *all* nodes that are subscribed to that topic.

To declare and use a publisher, do the following:
```
ros::Publisher pub; // This declares a publisher named 'pub'
```
NOTE: for this tutorial, if you need to access your publisher from somewhere outside of main, you may want to declare it globally.
```
pub = n.advertise<std_msgs::Int32>("topic_name", 1000); // This initializes the publisher.
```
- 'n' here is the name of your ros::NodeHandle (see skeleton code).
- 1000 is the size of the message queue – so if we're publishing faster than ROS can handle, it'll store 1000 messages before it starts dropping messages.

```
pub.publish(msg); // This is how you publish the message that you declared above
```

To declare and use a subscriber, do the following:
```
ros::Subscriber sub = n.subscribe("topic_name", 1000, callbackFunction);
```
This declares and initializes a subscriber names 'sub'.
- 1000 is the size of the message queue – so if we're receiving messages faster than we process them, ROS will store 1000 messages before it starts dropping messages.
- callbackFunction is the name of the function you want ROS to call every time a new message arrives.

You can create a callback function like this:
```
void callbackFunction(const std_msgs::Int32::ConstPtr& msg)
{
        // Get the number out of the message
        int num = msg->data;
        // Do something with num
}
```

Note that ROS will only process incoming messages and call your callback functions when you tell it to.

If you want ROS to just keep doing this indefinitely, use:
```
ros::spin();
```

Be careful though! When you do this, you are completely relinquishing control to ROS, and you won't get it back until the ROS node has been asked to shut down. In other words, execution will not continue past the ros::spin line until shut down.

If you want ROS to process any messages which are currently queued, but then give control back to you, use:
ros::spinOnce();

Sub-tasks

1. raw_data
Your job is to produce a stream of random numbers. It's up to you how you do this, but maybe have a look at the rand() function. Also, don't forget to include any headers that are necessary for this! Some requirements:

- Numbers should be between 0 and 999, inclusive
- You should publish your numbers on the raw_data topic
- You should publish these numbers at a rate of around 200 Hz (200 messages per second)
- You should wait for a little while when your program starts up, before publishing and numbers, to allow the other nodes a chance to get up and running. This ensures that none of the data you produce is dropped. Around 5 seconds should be okay.

2. prime_filter
Your job is to take the list of raw data, and keep only the prime numbers. It's up to you to decide how to tell whether a number is prime or not, but you don't need to do anything fancy for this tutorial – keep it simple, stupid! (KISS, a principle of software engineering). Some requirements:

- You should subscribe to the raw_data topic to get your input
- Whenever you receive a prime number on raw_data, you should re-publish it to the all_primes topic
- You can assume that all numbers you get from raw_data are non-negative, and less than 1000
- We recommend you invoke all your processing (prime test, re-publish) from inside the raw_data subscriber callback, but this isn't a strict requirement

3. unique_filter
Your job is to take the list of primes, and keep only unique entries. In other words, you should only publish each prime once. So if you see these messages: 3, 5, 7, 3, 3, 5, 5, 3, your output should be just 3, 5, 7. It's up to you how to do this. Some requirements:

- You should subscribe to the all_primes topic to get your input
- Whenever you receive a number on all_primes that you haven't seen yet, you should re-publish to the unique_primes topic
- You can assume that all numbers you get from all_primes are non-negative, and less than 1000
- We recommend you do your uniqueness test and publishing from inside the all_primes subscriber callback, but this isn't a strict requirement

4. logger
Your job is to process the lists produced by the first three modules, and to log them to three separate files. It's up to you how to do this, but maybe take a look at std::ofstream for file output. Also, don't forget to include any headers that are necessary for this! Some requirements:

- You should subscribe to the raw_data, all_primes, and unique_primes topics to get your input

- There are names inside your skeleton code for the log files for each of these three sets of numbers. Every time you get a new number on any of these three topics, you should write it to the corresponding log file
- You can assume that all numbers you get are non-negative, and less than 1000
- We recommend that you write these numbers to file from inside the subscriber callbacks, but this isn't a strict requirement

5. summarizer
Your job is to periodically print a status message to the screen, reporting the total number of data points we've processed on each of the raw_data, all_primes, and unique_primes topics. It's up to you how to do this, but maybe take a look at std::cout for output to the screen. Also, don't forget to include any headers that are necessary for this! Some requirements:

- You should subscribe to the raw_data, all_primes, and unique_primes topics to get your input
- You should print a status message about once every second
- This status must report the number of messages you've received on each of the three topics you are subscribed to, in whatever format you see fit
- We recommend that you use the ros::spinOnce() pattern, since you need to do periodic processing (i.e. printing the message) in addition to processing messages, but this isn't a strict requirement. Once you've got the base system working, if you're feeling ambitious, you can take a look at the ros::Timer class

Building ROS nodes
You can compile your changes by going in to the root directory of your workspace (~/art-meta/ on the VMs), and calling 'catkin_make'. You can also build your individual ROS node separately by running 'catkin_make <your_node_name>', e.g. 'catkin_make logger'. Make sure to check for compilation errors! If you see no errors, you are good to go.

Running ROS nodes
There are two ways to launch ROS nodes: manually, and via a launch file.

A. Manual Launch

- In a terminal, type 'roscore' to start the core ROS utilities. Leave that running. You can kill it later by typing Ctrl+C.
- In a new terminal or tab, type 'rosrun primes_tutorial <your node name>' to start your node. Leave that running. You can kill it later by typing Ctrl+C.

B. Launch Files

- In the root directory for this package (~/art-meta/src/primes_tutorial), type 'roslaunch system.launch'. This will start roscore for you, and will also launch all of the nodes in the package, not just the one you are working one.
- You can take a look at the system.launch file if you like to see what it's doing.
- Note that the skeleton implementations for the ros nodes do nothing, so if you are just trying to test your node, there is no harm in starting up the other nodes too.

Testing ROS nodes
Before you share your code, you need to test it to make sure it works! How can you do that, when your code depends on other nodes to work properly? You can pass your node fake input and make sure you get the behaviour you expect!

- Compile and launch the project, as described above.
- You can spoof a message by typing the following into a new terminal/tab: 'rostopic pub

--once *topic_name* std_msgs/Int32 *message*'. For example, to publish the number 10 to the raw_data topic, you can do: 'rostopic pub --once raw_data std_msgs/Int32 10'.

- If you're so inclined, check out http://wiki.ros.org/rostopic#rostopic_pub to learn about how to read a bunch of messages from a file and publish them one by one.
- You can “listen in” on the messages that your node is producing on a specific topic by typing (in a new terminal/tab) 'rostopic echo *topic_name*'. For example, to listen to messages on the unique_primes topic, you'd type 'rostopic echo unique_primes'. You'll see output like this:
  data: 5
  ---
  data: 7
  ---
  data: 13
  ---
  data: 2

  (Every time a new message arrives, you get a new 'data' line). You can stop listening by pressing Ctrl+C.

Version Control
We'll talk about how to use the version control system to share your changes with the rest of the team.

Good luck!