ROS Concepts 2

ECE 495/595 Lecture Slides

Winter 2017

Instructor: Micho Radovnikovich

Summary and Quick Links

These slides contain the following concepts:

- ▷ Details about ROS node handles (Slide 3)
- ▷ ROS time stamps (Slide 8)
- ▷ Message headers (Slide 11)
- ▷ Timers (Slide 13)
- ▷ Launch files (Slide 15)

Node Handles and Namespaces

- ▷ NodeHandle objects are used to interact with the ROS core and communicate with the rest of the system.
- ▷ NodeHandle objects can be instantiated in different **namespaces**.
- ▷ Typically, a node will have a **global** and **private** node handle:

```
int main(int argc, char** argv)
{
  ros::init(argc, argv, "nodehandle_example");
  ros::NodeHandle global_handle;
  ros::NodeHandle private_handle("~");
```

Node Handles and Namespaces

- Global node handles are instantiated without arguments and default to the root namespace '/'
- Private node handles are instantiated with the '~' argument to put it in a local namespace that has the same name as the node itself.
- The namespace of a given node handle transforms the name of everything initialized with the node handle.

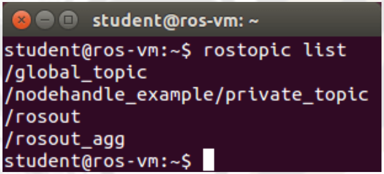
Node Handles and Namespaces

- A good example of how namespaces work can be illustrated with topic names:

```
int main(int argc, char** argv)
{
  ros::init(argc, argv, "nodehandle_example");
  ros::NodeHandle global_handle;
  ros::NodeHandle private_handle("~");

  ros::Publisher pub_global_topic =
    global_handle.advertise<std_msgs::String>("global_topic", 1);

  ros::Publisher pub_private_topic =
    private_handle.advertise<std_msgs::String>("private_topic", 1);

  ros::spin();
}
```

Node Handles and Namespaces

- This example advertises two different topics, but one is in the global namespace and the other is in the private namespace of the node.
- After running this node, running **rostopic list** in the terminal will show:



student@ros-vm: ~

```
student@ros-vm:~$ rostopic list
/global_topic
/nodehandle_example/private_topic
/rosout
/rosout_agg
student@ros-vm:~$ 
```

- Notice how the private topic is inside the **nodehandle_example** namespace.

Node Handles and Namespaces

- There is a big difference between these two lines of code:

```
private_handle.advertise<std_msgs::String>("private_topic", 1);
private_handle.advertise<std_msgs::String>("/private_topic", 1);
```

- Using the / in front of the private topic name overrides to the global namespace.
- In this case, the topic will be advertised in the global namespace, despite using the private node handle to do so.

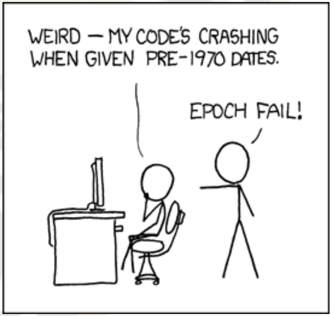
ROS Time Stamps

- ROS defines a **ros::Time** object that is used to time stamp messages. Details about how the ros::Time objects work can be found at `http://wiki.ros.org/roscpp/Overview/Time`
- This object has two properties: **sec** and **nsec**, which are both 32-bit unsigned integers.
- Combined, these two numbers make a decimal representation of the time with resolution of one nanosecond, and are accessible using two methods:
  - **toSec()** returns a floating point number with the combined whole value of **sec** and the fractional value of **nsec**.
  - **toNSec()** returns a 64-bit unsigned integer of nanoseconds.

ROS Time Stamps

- The real-world time is encoded using the standard UNIX *epoch*, which is defined to be the number of seconds since January 1st, 1970 at midnight.
  - Saturday, November 22nd, 2014 at 19:18:10 GMT is encoded as 1,416,683,890.

BONUS: You now are capable of getting this joke:



"The universe started in 1970. Anyone claiming to be over 45 is lying about their age."

http://xkcd.com/376/

ROS Time Stamps

- A **ros::Duration** object is used to represent a period of time in ROS.

- **ros::Time** objects represent an absolute time w.r.t. the Linux epoch, **ros::Duration** objects represent a relative *amount* of time.

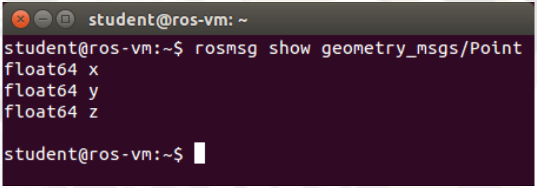
- Arithmetic operators are overloaded to support the following:
  - Duration + Duration = Duration
  - Time + Duration = Time
  - Time − Time = Duration

Message Headers

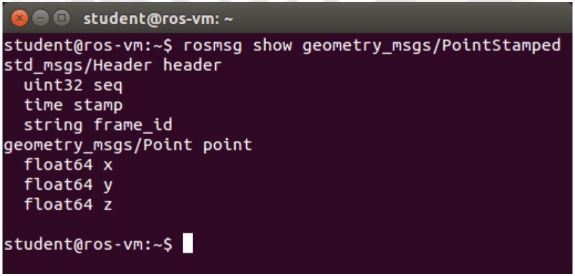
- Many messages published on ROS topics contain a *header* (**std_msgs::Header**)
- A std_msgs::Header is a structure that contains the following information:
  - **seq** — An unsigned 32-bit integer indicating the sequence number of the given message. This value is automatically incremented every time a message is published on the topic.
  - **stamp** — A ROS time stamp that is typically used to specify when the data contained in the message was generated.
  - **frame_id** — A string that indicates which reference frame the message's data is represented in (more on this later).

Message Headers

- Many ROS messages also have *stamped* versions, which means there is a **std_msgs::Header** attached to the original message type.



```
student@ros-vm:~$ rosmsg show geometry_msgs/Point
float64 x
float64 y
float64 z

student@ros-vm:~$ 
```



```
student@ros-vm:~$ rosmsg show geometry_msgs/PointStamped
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
geometry_msgs/Point point
  float64 x
  float64 y
  float64 z

student@ros-vm:~$ 
```

Timers

- Timers are used to execute code at periodic intervals.
- Node handles manage the timer. The user just specifies the desired period.
- Timers are usually instantiated in the main function of a node:

```
ros::Timer timer = nh.createTimer(ros::Duration(0.5),
                                  timerCallback);
```

- In this case, the timer is:
  - Set to trigger at a frequency of 2 Hz (period = 0.5 sec).
  - Set to call the **timerCallback** function when triggered.

Timers

- Timer callbacks are called with a "ros::TimerEvent" structure argument:

```
void timerCallback(const ros::TimerEvent& event)
{
  // Code goes here
}
```

- The TimerEvent contains four ROS time stamp values:
  - **current_real** – The system time as of when the callback function was called.
  - **last_real** – The system time at the *last* time the function was called.
  - **current_expected** – The scheduled time when the callback function was supposed to be called.
  - **last_expected** – When the last time this function was supposed to be called.

Launch Files

- ▷ Launch files are scripts to automate the running of a ROS system.
- ▷ They can be used to spawn any number of nodes, while also configuring parameters and topic names.
- ▷ All the vivid details can be found on the ROS wiki:
  - > Command line usage
    http://wiki.ros.org/roslaunch/Commandline%20Tools
  - > File syntax
    http://wiki.ros.org/roslaunch/XML

Launch Files

- Launch files follow XML syntax:

```
<?xml version="1.0"?>

<launch>
  <node pkg="package_name" type="node_type" name="node_name"
                                       output="screen" />
</launch>
```

- This launch file runs a single node using the **node** tag:
  - The compiled node name is **node_type**.
  - The node is compiled in the package **package_name**.
  - Its run-time name is changed to **node_name**.
  - The optional **output="screen"** allows any console output to be displayed in the terminal.

Launch Files

- While the **name** property must be set, it can be the same name as the node type.

- However, when launching multiple instances of the same node, their run-time names have to be different:

```
<launch>
  <node pkg="package_name" type="node_type" name="inst_1" />
  <node pkg="package_name" type="node_type" name="inst_2" />
</launch>
```

Launch Files

- Some of the common XML tags used in ROS launch files are:
  - **<node>** – Launches a particular node.
  - **<include>** – Used to include other launch files.
  - **<param>** – Sets a particular ROS parameter to a specific value.
  - **<rosparam>** – Used to load a set of parameters specified in a YAML file.
  - **<arg>** – Used to specify variable arguments to the launch file.