ROS
Pub-Sub, Parameters, Services, Roslaunch etc
Agenda

• Publishing messages to topics
• Subscribing to topics
• Differential drive robots
• Sending velocity commands
• roslaunch
ROS Parameters

- Parameters are like global data
- Accessed through the Parameter Server
- Typically handled by roscore
Setting Parameters

• Command line

```bash
rosrun my_pkg load_robot_ip:="192.168.1.21" rosparam set "/debug" true
```

• Programs

```python
nh.setParam("name", "left");
```
Namespaces

• Folder Hierarchy allows Separation:
• Separate nodes can co-exist, in different “namespaces”
  • relative vs. absolute name references
• Accessed through rose::NodeHandle object
  • also sets default Namespace for access
    • Global (root) Namespace

```cpp
ros::NodeHandle global();
global.getParam("test");
```

• Fixed Namespace:

```cpp
ros::NodeHandle fixed("/myApp");
global.getParam("test");
```
Parameters: C++ API

- NodeHandle object methods
  - `nh.hasParam(key)`
    - Returns true if parameter exists
  - `nh.getParam(key, &value)`
    - Gets value, returns T/F if exists.
  - `nh.param(key, &value, default)`
    - Get value (or default, if doesn’t exist)
  - `nh.setParam(key, value)`
    - Sets value
  - `nh.deleteParam(key)`
    - Deletes parameter
ros::Publisher

- Manages an advertisement on a specific topic
- A Publisher is created by calling NodeHandle::advertise()
  - Registers this topic in the master node
- Example for creating a publisher:

```cpp
ros::Publisher chatter_pub = node.advertise<std_msgs::String>("chatter", 1000);
```

- First parameter is the topic name
- Second parameter is the queue size
- Once all the publishers for a given topic go out of scope the topic will be unadvertised
ros::Publisher

• Messages are published on a topic through a call to publish()
• Example:

```cpp
std_msgs::String msg;
chatter_pub.publish(msg);
```

• The type of the message object must agree with the type given as a template parameter to the advertise<>() call
Talker and Listener

• We now create a new package with two nodes:
  • talker publishes messages to topic “chatter”
  • listener reads the messages from the topic and prints them out to the screen
• First create the package

$ cd ~/catkin_ws/src
catkin_create_pkg chat_pkg std_msgs rospy roscpp

• Open the package source directory in QtCreator and add a C++ source file named Talker.cpp
• Copy the following code into it
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <sstream>

int main(int argc, char **argv)
{
    ros::init(argc, argv, "talker"); // Initiate new ROS node named "talker"

    ros::NodeHandle node;
    ros::Publisher chatter_pub = node.advertise<std_msgs::String>("chatter", 1000);
    ros::Rate loop_rate(10);

    int count = 0;
    while (ros::ok()) // Keep spinning loop until user presses Ctrl+C
    {
        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(); // Need to call this function often to allow ROS to process incoming messages

        loop_rate.sleep(); // Sleep for the rest of the cycle, to enforce the loop rate
        count++;
    }
    return 0;
}
Subscribing to a Topic

• To start listening to a topic, call the method subscribe() of the node handle
  • This returns a Subscriber object that you must hold on to until you want to unsubscribe
• Example for creating a subscriber:
  ```
  ros::Subscriber sub = node.subscribe("chatter", 1000, messageCallback);
  ```
  • First parameter is the topic name
  • Second parameter is the queue size
  • Third parameter is the function to handle the message
#include "ros/ros.h"
#include "std_msgs/String.h"

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

int main(int argc, char **argv)
{
    // Initiate a new ROS node named "listener"
    ros::init(argc, argv, "listener");
    ros::NodeHandle node;

    // Subscribe to a given topic
    ros::Subscriber sub = node.subscribe("chatter", 1000, chatterCallback);

    // Enter a loop, pumping callbacks
    ros::spin();

    return 0;
}
**ros::spin()**

- The `ros::spin()` creates a loop where the node starts to read the topic, and when a message arrives `messageCallback` is called.
- `ros::spin()` will exit once `ros::ok()` returns false.
  - For example, when the user presses Ctrl+C or when `ros::shutdown()` is called.
Using Class Methods as Callbacks

• Suppose you have a simple class, Listener:

```cpp
class Listener
{
    public: void callback(const std_msgs::String::ConstPtr& msg);
};
```

• Then the NodeHandle::subscribe() call using the class method looks like this:

```cpp
Listener listener;
ros::Subscriber sub = node.subscribe("chatter", 1000, &Listener::callback, &listener);
```
Compile the Nodes

• Add the following to the package’s CMakeLists file

```cmake
#cmake_minimum_required(VERSION 2.8.3)
project(chat_pkg)
...

## Declare a cpp executable
add_executable(talker src/Talker.cpp)
add_executable(listener src/Listener.cpp)

## Specify libraries to link a library or executable target against
target_link_libraries(talker '${catkin_LIBRARIES}')
target_link_libraries(listener '${catkin_LIBRARIES}')
```
Building the Nodes

• Now build the package and compile all the nodes using the catkin_make tool:

```bash
cd ~/catkin_ws
catkin_make
```

• This will create two executables, talker and listener, at ~/catkin_ws/devel/lib/chat_pkg
Running the Nodes From Terminal

• Run roscore
• Run the nodes in two different terminals:

```bash
$ rosrun chat_pkg talker
$ rosrun chat_pkg listener
```
Running the Nodes From Terminal

- You can use `rosnod`e` and `rostopic` to debug and see what the nodes are doing
- Examples:
  - `$ rosnod`e info /talker`
  - `$ rosnod`e info /listener`
  - `$ rostopic` list`
  - `$ rostopic` info /chatter`
  - `$ rostopic` echo /chatter`
rqt_graph

- rqt_graph creates a dynamic graph of what's going on in the system
- Use the following command to run it:

```bash
$ rosrun rqt_graph rqt_graph
```
ROS Services

• The next step is to learn how to read the map in your ROS nodes
• For that purpose we will use a ROS service called `static_map` from the package `map_server`
• Services use the request/reply paradigm instead of the publish/subscribe model
Service Definitions

• ROS Services are defined by srv files, which contains a request message and a response message.
  • These are identical to the messages used with ROS Topics
• roscpp converts these srv files into C++ source code and creates 3 classes
• The names of these classes come directly from the srv filename:
  my_package/srv/Foo.srv ➔
  • my_package::Foo – service definition
  • my_package::Foo::Request – request message
  • my_package::Foo::Response – response message
namespace my_package
{
  struct Foo
  {
    class Request
    {
      ...
    };

    class Response
    {
      ...
    };

    Request request;
    Response response;
  };
}
Calling Services

• Since service calls are blocking, it will return once the call is done
  • If the service call succeeded, call() will return true and the value in srv.response will be valid.
  • If the call did not succeed, call() will return false and the value in srv.response will be invalid.

ros::NodeHandle nh;
ros::ServiceClient client =
h.nh.serviceClient<my_package::Foo>("my_service_name");
my_package::Foo foo;
foo.request.<var> = <value>;
...
if (client.call(foo)) {
    ...
}
• **roslaunch** is a tool for easily launching multiple ROS nodes as well as setting parameters on the Parameter Server

• It takes in one or more XML configuration files (with the `.launch` extension) that specify the parameters to set and nodes to launch

• If you use **roslaunch**, you do not have to run **roscore** manually
Launch File Example

• Launch file for launching both the talker and listener nodes (chat.launch):

```xml
<launch>
  <node name="talker" pkg="chat_pkg" type="talker" output="screen"/>
  <node name="listener" pkg="chat_pkg" type="listener" output="screen"/>
</launch>
```

• output="screen" makes the ROS log messages appear on the launch terminal window

• To run a launch file use:

```bash
$ roslaunch chat_pkg chat.launch
```
Launch File Example