Robotics and Intelligent Systems Lab

Lecture 4

Bilal Wehbe

WiSe 21/22

Recap

Launch Files

```
<launch>

   <arg name="red_color" default="255"/>

   <group ns="turtlesim1">
      <node pkg="turtlesim" name="sim" type="turtlesim_node"/>
   </group>

   <group ns="turtlesim2">
      <node pkg="turtlesim" name="sim" type="turtlesim_node"/>
   </group>

   <node pkg="turtlesim" name="mimic" type="mimic">
      <remap from="input" to="turtlesim1/turtle1"/>
      <remap from="output" to="turtlesim2/turtle1"/>
   </node>

   <param name="/sim/background_r" value="$(arg red_color)" type="int"/>

</launch>
```

Creating messages and services

Student.msg

```
string first_name
string last_name
uint8 age
uint32 score
```

AddTwoInts.srv

```
int64 A
int64 B
---
int64 Sum
```

Simple publisher and Subscriber in rospy

talker.py

```python
#!/usr/bin/env python

import rospy
from std_msgs.msg import String

def talker():
    pub = rospy.Publisher('chatter', String, queue_size=10)
    rospy.init_node('talker', anonymous=True)
    rate = rospy.Rate(10) # 10hz
    while not rospy.is_shutdown():
        hello_str = "hello world %s" % rospy.get_time()
        rospy.loginfo(hello_str)
        pub.publish(hello_str)
        rate.sleep()

if __name__ == '__main__':
    try:
        talker()
    except rospy.ROSInterruptException:
        pass
```

listener.py

```python
#!/usr/bin/env python
import rospy
from std_msgs.msg import String

def callback(data):
    rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.data)

def listener():

    # In ROS, nodes are uniquely named. If two nodes with the same
    # name are launched, the previous one is kicked off. The
    # anonymous=True flag means that rospy will choose a unique
    # name for our 'listener' node so that multiple listeners can
    # run simultaneously.
    rospy.init_node('listener', anonymous=True)

    rospy.Subscriber("chatter", String, callback)

    # spin() simply keeps python from exiting until this node is stopped
    rospy.spin()

if __name__ == '__main__':
    listener()
```

Publisher & Subscriber in rospy

Exercise

- Create a message WaveParams.msg that contains 3 elements:
  - period
  - magnitude
  - phase
- create a node that imports the created type and publishes a topic with the same message type created
- create another node that subscribes to the published topic and generates a sinusoidal signal and prints it on the screen.

WaveForm.msg

```
1 float64 period
2 float64 magnitude
3 float64 phase
```

signal_configurator.py

```
1 #!/usr/bin/env python
2
3 import rospy
4 from beginner_tutorials.msg import WaveForm
5 import numpy as np
6
7 def signal_configurator():
8     pub = rospy.Publisher('signal_config', WaveForm, queue_size=10)
9     rospy.init_node('signal_configurator', anonymous=True)
10    rate = rospy.Rate(10) # 10hz
11    while not rospy.is_shutdown():
12        waveform = WaveForm()
13        waveform.period = 1.0
14        waveform.magnitude = 1.0
15        waveform.phase = np.pi/4
16        pub.publish(waveform)
17        rate.sleep()
18
19 if __name__ == '__main__':
20    try:
21        signal_configurator()
22    except rospy.ROSInterruptException:
23        pass
```

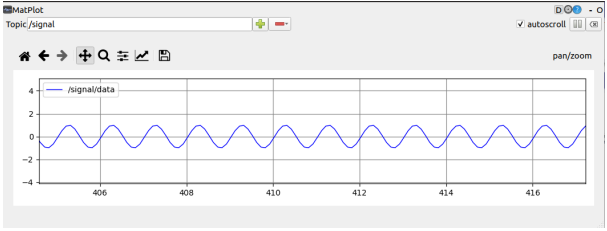
signal_generator.py

```
1 #!/usr/bin/env python
2 import rospy
3 from beginner_tutorials.msg import WaveForm
4 from std_msgs.msg import Float64
5 import numpy as np
6
7 pub = rospy.Publisher('signal', Float64, queue_size=10)
8
9 def callback(data):
10     signal = data.magnitude*np.sin(2.0*np.pi*rospy.get_time()/data.period + data.phase)
11     rospy.loginfo("generated signal %.3f" %signal)
12     pub.publish(signal)
13
14 def signal_generator():
15
16     rospy.init_node('signal_generator', anonymous=True)
17
18     rospy.Subscriber('signal_config', WaveForm, callback)
19
20     # spin() simply keeps python from exiting until this node is stopped
21     rospy.spin()
22
23 if __name__ == '__main__':
24     signal_generator()
```

- running both nodes

```
▶ rosrun beginner_tutorials signal_generator.py
[INFO] [1632650801.662060]: generated signal -0.974
[INFO] [1632650801.761503]: generated signal -0.655
[INFO] [1632650801.861268]: generated signal -0.087
[INFO] [1632650801.961854]: generated signal 0.517
[INFO] [1632650802.061362]: generated signal 0.921
[INFO] [1632650802.161907]: generated signal 0.974
[INFO] [1632650802.261424]: generated signal 0.656
[INFO] [1632650802.361775]: generated signal 0.085
[INFO] [1632650802.461437]: generated signal -0.516
[INFO] [1632650802.561452]: generated signal -0.921
```

- plotting the topic `/signal`



Publisher & Subscriber in roscpp

roscpp

- roscpp is a C++ implementation of ROS.
- It provides a client library that enables C++ programmers to quickly interface with ROS Topics, Services, and Parameters.
- roscpp is the most widely used ROS client library and is designed to be the high-performance library for ROS.

Writing the Publisher Node using rospy I

- let's create a ROS node in the pakcage `beginner_tutorials` which will continuously publish a message

```
$ roscd beginner_tutorials
$ mkdir src
$ touch src/talker.cpp
```

Writing the Publisher Node using roscpp

```
1 #include "ros/ros.h"
2 #include "std_msgs/String.h"
3
4 #include <sstream>
5
6 int main(int argc, char **argv)
7 {
8   ros::init(argc, argv, "talker");
9
10  ros::NodeHandle n;
11
12  ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
13
14  ros::Rate loop_rate(10);
15
16  int count = 0;
17  while (ros::ok())
18  {
19    /**
20     * This is a message object. You stuff it with data, and then publish it.
21     */
22    std_msgs::String msg;
23
24    std::stringstream ss;
25    ss << "hello world " << count;
26    msg.data = ss.str();
27
28    ROS_INFO("%s", msg.data.c_str());
29
30    chatter_pub.publish(msg);
31
32    ros::spinOnce();
33
34    loop_rate.sleep();
35    ++count;
36  }
37
38  return 0;
39 }
```

- This includes ros.h header which includes all the headers necessary to use the most common public pieces of the ROS system.
- This includes the std_msgs/String message, which resides in the std_msgs package. This is a header generated automatically from the String.msg file in that package
- sstream is only for streaming strings

Writing the Publisher Node using roscpp

```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(10);

  int count = 0;
  while (ros::ok())
  {
    /**
     * This is a message object. You stuff it with data, and then publish it.
     */
    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;
}
```

This simply starts the main function

Writing the Publisher Node using roscpp

```
1 #include "ros/ros.h"
2 #include "std_msgs/String.h"
3
4 #include <sstream>
5
6 int main(int argc, char **argv)
7 {
8   ros::init(argc, argv, "talker");
9
10  ros::NodeHandle n;
11
12  ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
13
14  ros::Rate loop_rate(10);
15
16  int count = 0;
17  while (ros::ok())
18  {
19    /**
20     * This is a message object. You stuff it with data, and then publish it.
21     */
22    std_msgs::String msg;
23
24    std::stringstream ss;
25    ss << "hello world " << count;
26    msg.data = ss.str();
27
28    ROS_INFO("%s", msg.data.c_str());
29
30    chatter_pub.publish(msg);
31
32    ros::spinOnce();
33
34    loop_rate.sleep();
35    ++count;
36  }
37
38  return 0;
39 }
```

- Initialize ROS with a node of name ”talker”
- The ros::init() function needs to see argc and argv so that it can perform any ROS arguments and name remapping that were provided at the command line.
- The third argument to init() is the name of the node.
- You must call one of the versions of ros::init() before using any other part of the ROS system.

Writing the Publisher Node using roscpp

```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(10);

  int count = 0;
  while (ros::ok())
  {
    /**
     * This is a message object. You stuff it with data, and then publish it.
     */
    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;
}
```

- Create a handle to this process' node.
- NodeHandle is the main access point to communications with the ROS system.
- The first NodeHandle constructed will fully initialize this node, and the last NodeHandle destructed will close down the node cleaning up any resources the node was using.

Writing the Publisher Node using roscpp

```
1 #include "ros/ros.h"
2 #include "std_msgs/String.h"
3
4 #include <sstream>
5
6 int main(int argc, char **argv)
7 {
8   ros::init(argc, argv, "talker");
9
10  ros::NodeHandle n;
11
12  ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
13
14  ros::Rate loop_rate(10);
15
16  int count = 0;
17  while (ros::ok())
18  {
19    /**
20     * This is a message object. You stuff it with data, and then publish it.
21     */
22    std_msgs::String msg;
23
24    std::stringstream ss;
25    ss << "hello world " << count;
26    msg.data = ss.str();
27
28    ROS_INFO("%s", msg.data.c_str());
29
30    chatter_pub.publish(msg);
31
32    ros::spinOnce();
33
34    loop_rate.sleep();
35    ++count;
36  }
37
38  return 0;
39 }
```

- Tell the master that we are going to be publishing a message of type std_msgs/String on the topic chatter.
- This lets the master tell any nodes listening on chatter that we are going to publish data on that topic
- The second argument is the size of our publishing queue.
- In this case if we are publishing too quickly it will buffer up a maximum of 1000 messages before beginning to throw away old ones.

Writing the Publisher Node using roscpp

```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(10);

  int count = 0;
  while (ros::ok())
  {
    /**
     * This is a message object. You stuff it with data, and then publish it.
     */
    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;
}
```

- NodeHandle::advertise() returns a ros::Publisher object, which serves two purposes
  - it contains a publish() method that lets you publish messages onto the topic it was created with
  - when it goes out of scope, it will automatically unadvertise

Writing the Publisher Node using roscpp

```
1 #include "ros/ros.h"
2 #include "std_msgs/String.h"
3
4 #include <sstream>
5
6 int main(int argc, char **argv)
7 {
8   ros::init(argc, argv, "talker");
9
10  ros::NodeHandle n;
11
12  ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
13
14  ros::Rate loop_rate(10);
15
16  int count = 0;
17  while (ros::ok())
18  {
19    /**
20     * This is a message object. You stuff it with data, and then publish it.
21     */
22    std_msgs::String msg;
23
24    std::stringstream ss;
25    ss << "hello world " << count;
26    msg.data = ss.str();
27
28    ROS_INFO("%s", msg.data.c_str());
29
30    chatter_pub.publish(msg);
31
32    ros::spinOnce();
33
34    loop_rate.sleep();
35    ++count;
36  }
37
38  return 0;
39 }
```

- A ros::Rate object allows you to specify a frequency that you would like to loop at.
- It will keep track of how long it has been since the last call to Rate::sleep(), and sleep for the correct amount of time
- the count integer keeps a count of how many messages we have sent. This is used to create a unique string for each message.

Writing the Publisher Node using roscpp

```
1 #include "ros/ros.h"
2 #include "std_msgs/String.h"
3
4 #include <sstream>
5
6 int main(int argc, char **argv)
7 {
8   ros::init(argc, argv, "talker");
9
10  ros::NodeHandle n;
11
12  ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
13
14  ros::Rate loop_rate(10);
15
16  int count = 0;
17  while (ros::ok())
18  {
19    /**
20     * This is a message object. You stuff it with data, and then publish it.
21     */
22    std_msgs::String msg;
23
24    std::stringstream ss;
25    ss << "hello world " << count;
26    msg.data = ss.str();
27
28    ROS_INFO("%s", msg.data.c_str());
29
30    chatter_pub.publish(msg);
31
32    ros::spinOnce();
33
34    loop_rate.sleep();
35    ++count;
36  }
37
38  return 0;
39 }
```

- By default roscpp will install a SIGINT handler which provides Ctrl-C handling which will cause ros::ok() to return false if that happens.
- ros::ok() will return false if:
  - a SIGINT is received (Ctrl-C)
  - we have been kicked off the network by another node with the same name
  - ros::shutdown() has been called by another part of the application.
  - all ros::NodeHandles have been destroyed
- Once ros::ok() returns false, all ROS calls will fail.

Writing the Publisher Node using roscpp

```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(10);

  int count = 0;
  while (ros::ok())
  {
    /**
     * This is a message object. You stuff it with data, and then publish it.
     */
    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;
}
```

- We broadcast a message on ROS using a message-adapted class, generally generated from a msg file
- We use here the standard String message, which has one member: ”data”.
- We store ss as a string into the data memeber of the string message.

Writing the Publisher Node using roscpp

```
1 #include "ros/ros.h"
2 #include "std_msgs/String.h"
3
4 #include <sstream>
5
6 int main(int argc, char **argv)
7 {
8   ros::init(argc, argv, "talker");
9
10  ros::NodeHandle n;
11
12  ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
13
14  ros::Rate loop_rate(10);
15
16  int count = 0;
17  while (ros::ok())
18  {
19    /**
20     * This is a message object. You stuff it with data, and then publish it.
21     */
22    std_msgs::String msg;
23
24    std::stringstream ss;
25    ss << "hello world " << count;
26    msg.data = ss.str();
27
28    ROS_INFO("%s", msg.data.c_str());
29
30    chatter_pub.publish(msg);
31
32    ros::spinOnce();
33
34    loop_rate.sleep();
35    ++count;
36  }
37
38  return 0;
39 }
```

- ROS_INFO is our replacement for printf/cout
- the chatte_pub broadcasts the generated msg onto the chatter topic.
- Calling ros::spinOnce() here is not necessary for this simple program, because we are not receiving any callbacks. However, if you were to add a subscription into this application, and did not have ros::spinOnce() here, your callbacks would never get called. So, add it for good measure

Writing the Publisher Node using roscpp

```
#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(10);

  int count = 0;
  while (ros::ok())
  {
    /**
     * This is a message object. You stuff it with data, and then publish it.
     */
    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;
}
```

- Now we use the ros::Rate object to sleep for the time remaining to let us hit our 10Hz publish rate.
- ++count increases the message count by 1

Writing the Publisher Node using roscpp

```
1 #include "ros/ros.h"
2 #include "std_msgs/String.h"
3
4 #include <sstream>
5
6 int main(int argc, char **argv)
7 {
8   ros::init(argc, argv, "talker");
9
10  ros::NodeHandle n;
11
12  ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
13
14  ros::Rate loop_rate(10);
15
16  int count = 0;
17  while (ros::ok())
18  {
19    /**
20     * This is a message object. You stuff it with data, and then publish it.
21     */
22    std_msgs::String msg;
23
24    std::stringstream ss;
25    ss << "hello world " << count;
26    msg.data = ss.str();
27
28    ROS_INFO("%s", msg.data.c_str());
29
30    chatter_pub.publish(msg);
31
32    ros::spinOnce();
33
34    loop_rate.sleep();
35    ++count;
36  }
37
38  return 0;
39 }
```

- Here's the condensed version of what's going on
  - Initialize the ROS system
  - Advertise that we are going to be publishing std_msgs/String messages on the chatter topic to the master
  - Loop while publishing messages to chatter 10 times a second

Writing the Subscriber Node using roscpp

```
1 #include "ros/ros.h"
2 #include "std_msgs/String.h"
3
4 void chatterCallback(const std_msgs::String::ConstPtr& msg)
5 {
6   ROS_INFO("I heard: [%s]", msg->data.c_str());
7 }
8
9 int main(int argc, char **argv)
10 {
11   ros::init(argc, argv, "listener");
12
13   ros::NodeHandle n;
14
15   ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);
16
17   ros::spin();
18
19   return 0;
20 }
```

- This is the callback function that will get called when a new message has arrived on the chatter topic.
- The message is passed in a boost shared_ptr, which means you can store it off if you want, without worrying about it getting deleted underneath you, and without copying the underlying data.

Writing the Subscriber Node using roscpp

```
1 #include "ros/ros.h"
2 #include "std_msgs/String.h"
3
4 void chatterCallback(const std_msgs::String::ConstPtr& msg)
5 {
6   ROS_INFO("I heard: [%s]", msg->data.c_str());
7 }
8
9 int main(int argc, char **argv)
10 {
11   ros::init(argc, argv, "listener");
12
13   ros::NodeHandle n;
14
15   ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);
16
17   ros::spin();
18
19   return 0;
20 }
```

- Subscribe to the chatter topic with the master. ROS will call the chatterCallback() function whenever a new message arrives.
- The 2nd argument is the queue size, in case we are not able to process messages fast enough.
- NodeHandle::subscribe() returns a ros::Subscriber object, that you must hold on to until you want to unsubscribe.
- When the Subscriber object is destructed, it will automatically unsubscribe from the chatter topic.

Writing the Subscriber Node using roscpp

```
1 #include "ros/ros.h"
2 #include "std_msgs/String.h"
3
4 void chatterCallback(const std_msgs::String::ConstPtr& msg)
5 {
6   ROS_INFO("I heard: [%s]", msg->data.c_str());
7 }
8
9 int main(int argc, char **argv)
10 {
11   ros::init(argc, argv, "listener");
12
13   ros::NodeHandle n;
14
15   ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);
16
17   ros::spin();
18
19   return 0;
20 }
```

- ros::spin() enters a loop, calling message callbacks as fast as possible.
- Don't worry though, if there's nothing for it to do it won't use much CPU.
- ros::spin() will exit once ros::ok() returns false, which means ros::shutdown() has been called, either by the default Ctrl-C handler, the master telling us to shutdown, or it being called manually.

Writing the Subscriber Node using roscpp

```
1 #include "ros/ros.h"
2 #include "std_msgs/String.h"
3
4 void chatterCallback(const std_msgs::String::ConstPtr& msg)
5 {
6   ROS_INFO("I heard: [%s]", msg->data.c_str());
7 }
8
9 int main(int argc, char **argv)
10 {
11   ros::init(argc, argv, "listener");
12
13   ros::NodeHandle n;
14
15   ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);
16
17   ros::spin();
18
19   return 0;
20 }
```

- Again, here's a condensed version of what's going on
  - Initialize the ROS system
  - Subscribe to the chatter topic
  - Spin, waiting for messages to arrive
  - When a message arrives, the chatterCallback() function is called

Building your Nodes

- To build the nodes simply simply add these few lines to the bottom of your CMakeLists.txt:

```
add_executable(talker src/talker.cpp)
target_link_libraries(talker ${catkin_LIBRARIES})
add_dependencies(talker beginner_tutorials_generate_messages_cpp)

add_executable(listener src/listener.cpp)
target_link_libraries(listener ${catkin_LIBRARIES})
add_dependencies(listener beginner_tutorials_generate_messages_cpp)
```

- This will create two executables, talker and listener, which by default will go into package directory of your devel space, located by default at ~/catkin_ws/devel/lib/<package name>

Building your Nodes

- Note that you have to add dependencies for the executable targets to message generation targets:

```
add_dependencies(talker beginner_tutorials_generate_messages_cpp)
```

- This makes sure message headers of this package are generated before being used.
- If you use messages from other packages inside your catkin workspace, you need to add dependencies to their respective generation targets as well, because catkin builds all projects in parallel.

Building your Nodes

- You can use the variable ${catkin_LIBRARIES} to depend on all necessary targets:

```
target_link_libraries(talker ${catkin_LIBRARIES})
```

- No navigate ot your catkin workspace and build the new node you created

```
$ cd ~/catkin_ws
$ catkin build
```

Running the Nodes

- To run the talker

```
$ rosrun beginner_tutorials talker
```

```
~/ros_workspaces/catkin_ws
▶ rosrun beginner_tutorials talker
[ INFO] [1632662982.666944339]: hello world 0
[ INFO] [1632662982.767074721]: hello world 1
[ INFO] [1632662982.867194856]: hello world 2
[ INFO] [1632662982.967094710]: hello world 3
```

- Now start the listener using

```
$ rosrun beginner_tutorials listener
```

```
~/ros_workspaces/catkin_ws
▶ rosrun beginner_tutorials listener
[ INFO] [1632662982.867819465]: I heard: [hello world 2]
[ INFO] [1632662982.967731520]: I heard: [hello world 3]
[ INFO] [1632662983.067801757]: I heard: [hello world 4]
[ INFO] [1632662983.167350265]: I heard: [hello world 5]
```

Exercise

- redo the previous signl generator exercise using roscpp

```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(10);

  int count = 0;
  while (ros::ok())
  {
    /**
     * This is a message object. You stuff it with data, and then publish it.
     */
    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;
}
```

```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("chatter", 1000, chatterCallback);

  ros::spin();

  return 0;
}
```

Service & Client

Writing a Server Node

```cpp
#include "ros/ros.h"
#include "beginner_tutorials/AddTwoInts.h"

bool add(beginner_tutorials::AddTwoInts::Request  &req,
         beginner_tutorials::AddTwoInts::Response &res)
{
  res.Sum = req.A + req.B;
  ROS_INFO("request: x=%ld, y=%ld", (long int)req.A, (long int)req.B);
  ROS_INFO("sending back response: [%ld]", (long int)res.Sum);
  return true;
}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "add_two_ints_server");
  ros::NodeHandle n;

  ros::ServiceServer service = n.advertiseService("add_two_ints", add);
  ROS_INFO("Ready to add two ints.");
  ros::spin();

  return 0;
}
```

- beginner_tutorials/AddTwoInts.h is the header file generated from the srv file that we created previously.

Writing a Server Node

```
1 #include "ros/ros.h"
2 #include "beginner_tutorials/AddTwoInts.h"
3
4 bool add(beginner_tutorials::AddTwoInts::Request  &req,
5          beginner_tutorials::AddTwoInts::Response &res)
6 {
7   res.Sum = req.A + req.B;
8   ROS_INFO("request: x=%ld, y=%ld", (long int)req.A, (long int)req.B);
9   ROS_INFO("sending back response: [%ld]", (long int)res.Sum);
10  return true;
11 }
12
13 int main(int argc, char **argv)
14 {
15   ros::init(argc, argv, "add_two_ints_server");
16   ros::NodeHandle n;
17
18   ros::ServiceServer service = n.advertiseService("add_two_ints", add);
19   ROS_INFO("Ready to add two ints.");
20   ros::spin();
21
22   return 0;
23 }
```

- This function provides the service for adding two ints, it takes in the request and response type defined in the srv file and returns a boolean.
- Here the two ints are added and stored in the response. Then some information about the request and response are logged.
- Finally the service returns true when it is complete.

Writing a Server Node

```
1 #include "ros/ros.h"
2 #include "beginner_tutorials/AddTwoInts.h"
3
4 bool add(beginner_tutorials::AddTwoInts::Request  &req,
5          beginner_tutorials::AddTwoInts::Response &res)
6 {
7   res.Sum = req.A + req.B;
8   ROS_INFO("request: x=%ld, y=%ld", (long int)req.A, (long int)req.B);
9   ROS_INFO("sending back response: [%ld]", (long int)res.Sum);
10  return true;
11 }
12
13 int main(int argc, char **argv)
14 {
15   ros::init(argc, argv, "add_two_ints_server");
16   ros::NodeHandle n;
17
18   ros::ServiceServer service = n.advertiseService("add_two_ints", add);
19   ROS_INFO("Ready to add two ints.");
20   ros::spin();
21
22   return 0;
23 }
```

- Here the service is created and advertised over ROS.
- Print out that the server is ready.
- spin to enter a loop and wait on requests from a client.

Writing a Client Node

```
1 #include "ros/ros.h"
2 #include "beginner_tutorials/AddTwoInts.h"
3 #include <cstdlib>
4
5 int main(int argc, char **argv)
6 {
7   ros::init(argc, argv, "add_two_ints_client");
8   if (argc != 3)
9   {
10    ROS_INFO("usage: add_two_ints_client X Y");
11    return 1;
12  }
13
14  ros::NodeHandle n;
15  ros::ServiceClient client =
  n.serviceClient<beginner_tutorials::AddTwoInts>("add_two_ints");
16  beginner_tutorials::AddTwoInts srv;
17  srv.request.A = atoll(argv[1]);
18  srv.request.B = atoll(argv[2]);
19  if (client.call(srv))
20  {
21    ROS_INFO("Sum: %ld", (long int)srv.response.Sum);
22  }
23  else
24  {
25    ROS_ERROR("Failed to call service add_two_ints");
26    return 1;
27  }
28
29  return 0;
30 }
```

- Initialize a ros node.
- argc is used here to check is the arguments passed via the command line are of the form ”X Y”

Writing a Client Node

```
1 #include "ros/ros.h"
2 #include "beginner_tutorials/AddTwoInts.h"
3 #include <cstdlib>
4
5 int main(int argc, char **argv)
6 {
7   ros::init(argc, argv, "add_two_ints_client");
8   if (argc != 3)
9   {
10    ROS_INFO("usage: add_two_ints_client X Y");
11    return 1;
12  }
13
14  ros::NodeHandle n;
15  ros::ServiceClient client =
    n.serviceClient<beginner_tutorials::AddTwoInts>("add_two_ints");
16  beginner_tutorials::AddTwoInts srv;
17  srv.request.A = atoll(argv[1]);
18  srv.request.B = atoll(argv[2]);
19  if (client.call(srv))
20  {
21    ROS_INFO("Sum: %ld", (long int)srv.response.Sum);
22  }
23  else
24  {
25    ROS_ERROR("Failed to call service add_two_ints");
26    return 1;
27  }
28
29  return 0;
30 }
```

- This creates a client for the add_two_ints service. The ros::ServiceClient object is used to call the service later on
- The srv type beginner_tutorials::AddTwoPoints is given as a template argument to instantiate the serverClient to call services of that type.

Writing a Client Node

```
1 #include "ros/ros.h"
2 #include "beginner_tutorials/AddTwoInts.h"
3 #include <cstdlib>
4
5 int main(int argc, char **argv)
6 {
7   ros::init(argc, argv, "add_two_ints_client");
8   if (argc != 3)
9   {
10    ROS_INFO("usage: add_two_ints_client X Y");
11    return 1;
12  }
13
14  ros::NodeHandle n;
15  ros::ServiceClient client =
    n.serviceClient<beginner_tutorials::AddTwoInts>("add_two_ints");
16  beginner_tutorials::AddTwoInts srv;
17  srv.request.A = atoll(argv[1]);
18  srv.request.B = atoll(argv[2]);
19  if (client.call(srv))
20  {
21    ROS_INFO("Sum: %ld", (long int)srv.response.Sum);
22  }
23  else
24  {
25    ROS_ERROR("Failed to call service add_two_ints");
26    return 1;
27  }
28
29  return 0;
30 }
```

- Here we instantiate an autogenerated service class, and assign values into its request member.
- A service class contains two members, request and response.
- atoll is used here to cast the parses arguments as a value of type long long int.
- the first argument is stored in the member 'A' and the second in 'B'.

Writing a Client Node

```
1 #include "ros/ros.h"
2 #include "beginner_tutorials/AddTwoInts.h"
3 #include <cstdlib>
4
5 int main(int argc, char **argv)
6 {
7   ros::init(argc, argv, "add_two_ints_client");
8   if (argc != 3)
9   {
10    ROS_INFO("usage: add_two_ints_client X Y");
11    return 1;
12  }
13
14  ros::NodeHandle n;
15  ros::ServiceClient client =
    n.serviceClient<beginner_tutorials::AddTwoInts>("add_two_ints");
16  beginner_tutorials::AddTwoInts srv;
17  srv.request.A = atoll(argv[1]);
18  srv.request.B = atoll(argv[2]);
19  if (client.call(srv))
20  {
21    ROS_INFO("Sum: %ld", (long int)srv.response.Sum);
22  }
23  else
24  {
25    ROS_ERROR("Failed to call service add_two_ints");
26    return 1;
27  }
28
29  return 0;
30 }
```

- Here client.call actually calls the service.
- Since service calls are blocking, it will return only 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.

Building the Nodes

- Similar to the publisher and subscriber nodes, the server and client nodes has to be declared in the CMakeLists.txt

```
add_executable(add_two_ints_server src/add_two_ints_server.cpp)
target_link_libraries(add_two_ints_server ${catkin_LIBRARIES})
add_dependencies(add_two_ints_server beginner_tutorials_generate_messages_cpp)

add_executable(add_two_ints_client src/add_two_ints_client.cpp)
target_link_libraries(add_two_ints_client ${catkin_LIBRARIES})
add_dependencies(add_two_ints_client beginner_tutorials_generate_messages_cpp)
```

- Build the package

Running the Nodes

- Run the server

```
$ rosrun beginner_tutorials add_two_ints_server
```

```
~/ros_workspaces/catkin_ws
▶ rosrun beginner_tutorials add_two_ints_server
[ INFO] [1632669735.477405723]: Ready to add two ints.
[ INFO] [1632669741.809262696]: request: x=1, y=2
[ INFO] [1632669741.809310082]: sending back response: [3]
```

- Now start the client giving in some arguments

```
~/ros_workspaces/catkin_ws
▶ rosrun beginner_tutorials add_two_ints_client 1 2
[ INFO] [1632669741.809538239]: Sum: 3

~/ros_workspaces/catkin_ws
▶ rosrun beginner_tutorials add_two_ints_client 1 2 3
[ INFO] [1632669745.685908926]: usage: add_two_ints_client X Y
```

Writing a Server and Client in rospy

```python
#!/usr/bin/env python

from beginner_tutorials.srv import AddTwoInts,AddTwoIntsResponse
import rospy

def handle_add_two_ints(req):
    print("Returning [%s + %s = %s]"%(req.A, req.B, (req.A + req.B)))
    return AddTwoIntsResponse(req.A + req.B)

def add_two_ints_server():
    rospy.init_node('add_two_ints_server')
    s = rospy.Service('add_two_ints', AddTwoInts, handle_add_two_ints)
    print("Ready to add two ints.")
    rospy.spin()

if __name__ == "__main__":
    add_two_ints_server()
```

Writing a Server and Client in rospy

```python
#!/usr/bin/env python

import sys
import rospy
from beginner_tutorials.srv import *

def add_two_ints_client(x, y):
    rospy.wait_for_service('add_two_ints')
    try:
        add_two_ints = rospy.ServiceProxy('add_two_ints', AddTwoInts)
        req = AddTwoIntsRequest()
        req.A = x
        req.B = y
        resp1 = add_two_ints(req) # (x, y) can be passed directly as well
        return resp1.Sum
    except rospy.ServiceException as e:
        print("Service call failed: %s"%e)

if __name__ == "__main__":
    if len(sys.argv) == 3:
        x = int(sys.argv[1])
        y = int(sys.argv[2])
    else:
        print("usage: add_two_ints_client.py X Y")
        sys.exit(1)
    print("Requesting %s+%s"%(x, y))
    print("%s + %s = %s"%(x, y, add_two_ints_client(x, y)))
```

Using Parameters in rospy and roscpp

Parameters in rospy

- You can use integers, floats, strings and booleans as Parameter values.
- You can also use lists and dictionaries of these types.
- Dictionaries are equivalent to ROS Namespaces. They are an effective way of grouping similar Parameters together so that you can get and set them atomically.

```
/gains/P = 1.0
/gains/I = 2.0
/gains/D = 3.0
```

- In rospy the parameter /gains has the Python dictionary value

```
{'P': 1.0, 'I' = 2.0, 'D' = 3.0}
```

Parameters in rospy

- Let us define in a launch file a set of parameters.

```
<launch>

  <!-- set a /global_example parameter -->
  <param name="global_example" value="global value" />

  <group ns="parent">

    <!-- set /parent/utterance -->
    <param name="utterance" value="Hello World" />

    <param name="to_delete" value="Delete Me" />

    <!-- a group of parameters that we will fetch together -->
    <group ns="gains">
      <param name="P" value="1.0" />
      <param name="I" value="2.0" />
      <param name="D" value="3.0" />
    </group>

  </group>

</launch>
```

Getting parameters in rospy

- Getting a parameter is as simple as calling `rospy.get_param(param_name)`
- to get any global paramter simply:

```
rospy.get_param('/global_param_name')
```

Getting parameters in rospy

- Getting a parameter is as simple as calling `rospy.get_param(param_name)`
- to get any global paramter simply:

```
rospy.get_param('/global_param_name')
```

- To get a parameter from the parent namespace

```
rospy.get_param('param_name')
```

Getting parameters in rospy

- Getting a parameter is as simple as calling `rospy.get_param(param_name)`
- to get any global paramter simply:

```
rospy.get_param('/global_param_name')
```

- To get a parameter from the parent namespace

```
rospy.get_param('param_name')
```

- To get a parameter from our private namespace

```
rospy.get_param('~private_param_name')
```

Setting parameters in rospy

- Similarly, you set a parameter by calling rospy.set_param(param_name, param_value)

```
rospy.set_param('some_numbers', [1., 2., 3., 4.])
rospy.set_param('truth', True)
rospy.set_param('~private_bar', 1+2)
```

Setting parameters in rospy

- Similarly, you set a parameter by calling rospy.set_param(param_name, param_value)

```
rospy.set_param('some_numbers', [1., 2., 3., 4.])
rospy.set_param('truth', True)
rospy.set_param('~private_bar', 1+2)
```

- You can delete parameter by calling rospy.delete_param(param_name):

```
rospy.delete_param('param_name')
```

Setting parameters in rospy

- Similarly, you set a parameter by calling rospy.set_param(param_name, param_value)

```
rospy.set_param('some_numbers', [1., 2., 3., 4.])
rospy.set_param('truth', True)
rospy.set_param('~private_bar', 1+2)
```

- You can delete parameter by calling rospy.delete_param(param_name):

```
rospy.delete_param('param_name')
```

- If you don't know whether or not a parameter exists, you can call rospy.has_param(param_name):

```
if rospy.has_param('to_delete'):
rospy.delete_param('to_delete')
```

Resolving names and searching parameters

- As names in ROS can be remapped and your Node may get pushed into a namespace.
- rospy does most of the work for you by automatically resolving any names you pass into get_param, set_param
- For debugging purposes, you can print out the names of the Parameters that you are accessing

```
rospy.resolve_name(name)
```

Resolving names and searching parameters

- As names in ROS can be remapped and your Node may get pushed into a namespace.
- rospy does most of the work for you by automatically resolving any names you pass into get_param, set_param
- For debugging purposes, you can print out the names of the Parameters that you are accessing

```
rospy.resolve_name(name)
```

- You can search for a Parameter if you don't know what namespace it is set in
- This starts in the Node's private namespace and proceeds upwards to the global namespace.
- To find the resolved Parameter name:

```
full_param_name = rospy.search_param('param_name')
param_value = rospy.get_param(full_param_name)
```

Parameters in roscpp

- There are two way to get parameters using a `NodeHandle`
  - using `getParam()`

```
bool getParam (const std::string& key, parameter_type& output_value) const
```

  - key is a Graph Resource name (/global, /parent/param, ..)
  - output_value is the place to put the retrieved data
  - parameter_type: [bool, int, double, string, XmlRpcValue]

Parameters in roscpp

- There are two way to get parameters using a `NodeHandle`
  - using `getParam()`

```
bool getParam (const std::string& key, parameter_type& output_value) const
```

  - key is a Graph Resource name (/global, /parent/param, ..)
  - output_value is the place to put the retrieved data
  - parameter_type: [bool, int, double, string, XmlRpcValue]
- Usage:

```
std::string s;
if (n.getParam("my_param", s))
{
    ROS_INFO("Got param:  %s", s.c_str());
}
else
{
    ROS_ERROR("Failed to get param 'my_param'");
}
```

Parameters in roscpp

- There are two way to get parameters using a `NodeHandle`
  - using `param()`, similar to `getParam()`, but allows you to specify a default value in the case that the parameter could not be retrieved
  - Usage:

```
int i;
n.param("my_num", i, 42);
```

```
std::string s;
n.param<std::string>("my_param", s, "default_value");
```

Parameters in roscpp

- Setting parameters

```
n.setParam("my_param", "hello there");
```

- Deleting parameters

```
n.deleteParam("my_param");
```

- Checking for existence

```
if (!n.hasParam("my_param"))
{
    ROS_INFO("No param named 'my_param'");
}
```

Parameters in roscpp

- Searching for Parameters

```
std::string param_name;
if (n.searchParam("b", param_name))
{
    // Found parameter, can now query it using param_name
    int i = 0;
    n.getParam(param_name, i);
}
else
{
    ROS_INFO("No param 'b' found in an upward search");
}
```

Exercise

Change the `signal_configurator` node to accept the period, magnitude and phase as parameters