CSCI 473/573
Human-Centered Robotics

Special Thanks: The slides of ROS basics are adapted from Roi Yehoshua
ROS Introduction
Main concepts
Basic commands
What is ROS?

• ROS is an open-source **robot operating system**
• A set of software libraries and tools that help you build robot applications that work across a wide variety of robotic platforms
• Originally developed in 2007 at the Stanford Artificial Intelligence Laboratory and development continued at Willow Garage
• Since 2013 managed by OSRF (Open Source Robotics Foundation)
ROS Main Features

ROS has two "sides"

• The operating system side, which provides standard operating system services such as:
  • hardware abstraction
  • low-level device control
  • implementation of commonly used functionality
  • message-passing between processes
  • package management

• A suite of user contributed packages that implement common robot functionality such as SLAM, planning, perception, vision, manipulation, etc.
ROS Installation

• Install Ubuntu in VirtualBox:

• Install ROS Kinetic:
  http://wiki.ros.org/kinetic/Installation/Ubuntu

We will use Ubuntu 16.04 LTS and ROS Kinetic
(64 bit Ubuntu is suggested)

• ROS Wiki:  http://wiki.ros.org/
ROS Core Concepts

- Nodes
- Messages and Topics
- Services
- ROS Master
- Parameters
- Stacks and packages
ROS Nodes

• Single-purposed executable programs
  • e.g. sensor driver(s), actuator driver(s), mapper, planner, UI, etc.
• Individually compiled, executed, and managed
• Nodes are written using a ROS client library
  • roscpp – C++ client library
  • rospy – python client library
• Nodes can publish or subscribe to a Topic
• Nodes can also provide or use a Service
ROS Topics

• A topic is a name for a stream of messages with a defined type
  • e.g., data from a laser range-finder might be sent on a topic called scan, with a message type of LaserScan

• Nodes communicate with each other by publishing messages to topics

• Publish/Subscribe model: 1-to-N broadcasting
ROS Topics

- Kinect Node: Publishes 3D data from Kinect as messages.
- 3D Processing Node: Processes Kinect data and publishes directions.
- Control Node: Subscribes to directions and commands motors.
ROS Messages

• Strictly-typed data structures for inter-node communication
  • For example, geometry_msgs/Twist is used to express velocity commands:
    
    Vector3 linear
    Vector3 angular

• Vector3 is another message type composed of:
  
  float64 x
  float64 y
  float64 z
ROS Services

• Synchronous inter-node transactions
• Service/Client model: 1-to-1 request-response
• Service roles:
  • carry out remote computation / storage
  • trigger functionality / behavior
• Example:
  • map_server/static_map – retrieves the current grid map used by the robot for navigation
ROS Master

• Provides connection information to nodes so that they can transmit messages to each other

  • Every node connects to a master at startup to register details of the message streams they publish, and the streams to which that they to subscribe

  • When a new node appears, the master provides it with the information that it needs to form a direct peer-to-peer connection with other nodes publishing and subscribing to the same message topics
Parameter Server

- Configuration information in ROS is usually saved to the Parameter server.
- A shared, multi-variate dictionary that is accessible via network APIs
- Nodes use this server to store and retrieve parameters at runtime.
- Best used for static data such as configuration parameters
- Runs inside the ROS master
ROS Packages

• Software in ROS is organized in *packages*.
• A package contains one or more nodes and provides a ROS interface
• Most of ROS packages are hosted in GitHub or GitLab
ROS Environment

• ROS relies on the notion of combining spaces using the shell environment
  • This makes developing against different versions of ROS or against different sets of packages easier

• After you install ROS you will have to setup.*sh files in '/opt/ros/<distro>/', and you could source them like so:

  $ source /opt/ros/indigo/setup.bash

• You will need to run this command on every new shell you open to have access to the ros commands, unless you add this line to your bash startup file (~/.bashrc)
A cool video on drones
ROS Basic Commands

- roscore
- rosrun
- rosnodes
- rostopic

"Sit, boy, sit! Sit, I say, Si... Oh, forget it."
roscore

• roscore is the first thing you should run when using ROS

$ roscore

• roscore will start up:
  • a ROS Master
  • a ROS Parameter Server
  • a roslout logging node
viki@c3po:~$ roscore
... logging to /home/viki/.ros/log/c54cfa00-5cfb-11e4-8e38-000c293f9c00/roslaunch
h-c3po-3511.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://c3po:55749/
ros_comm version 1.11.8

SUMMARY
=======

PARAMETERS

* /rosdistro: indigo
* /rosversion: 1.11.8

NODES

auto-starting new master
process[master]: started with pid [3523]
ROS_MASTER_URI=http://c3po:11311/

setting /run_id to c54cfa00-5cfb-11e4-8e38-000c293f9c00
process[rosout-1]: started with pid [3536]
started core service [/rosout]
**rosrun**

- rosr un allows you to run a node
- Usage:

  ```
  $ rosr un <package> <executable>
  ```

- Example:

  ```
  $ rosr un turtlesim turtlesim_node
  ```
• **Demo – Turtlesim:** In separate terminals, run:
  
  • roscore
  • rosrun turtlesim turtlesim_node
  • rosrun turtlesim turtle_teleop_key
**rosnode**

- Displays debugging information about ROS nodes, including publications, subscriptions and connections

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>$rosnode list</code></td>
<td>List active nodes</td>
</tr>
<tr>
<td><code>$rosnode ping</code></td>
<td>Test connectivity to node</td>
</tr>
<tr>
<td><code>$rosnode info</code></td>
<td>Print information about a node</td>
</tr>
<tr>
<td><code>$rosnode kill</code></td>
<td>Kill a running node</td>
</tr>
<tr>
<td><code>$rosnode machine</code></td>
<td>List nodes running on a particular machine</td>
</tr>
</tbody>
</table>
rosnode info

```bash
viki@c3po:~$ rosnode info turtlesim

Node [/turtlesim]
Publication:
* /turtle1/color_sensor [turtlesim/Color]
* /rosout [rosgraph_msgs/Log]
* /turtle1/pose [turtlesim/Pose]

Subscription:
* /turtle1/cmd_vel [geometry_msgs/Twist]

Service:
* /turtle1/teleport_absolute
* /turtlesim/get_loggers
* /turtlesim/set_logger_level
* /reset
* /spawn
* /clear
* /turtle1/set_pen
* /turtle1/teleport_relative
* /kill

contacting node http://c3po:54205/ ...
Ppid: 3825
Connection:
* topic: /rosout
  * to: /rosout
    * direction: outbound
    * transport: TCPROS
* topic: /turtle1/cmd_vel
  * to: /teleop_turtle (http://c3po:47526/)
    * direction: inbound
    * transport: TCPROS

viki@c3po:~$ ```
rostopic

• Gives information about a topic and allows to publish messages on a topic

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>$rostopic list</code></td>
<td>List active topics</td>
</tr>
<tr>
<td><code>$rosnod node echo /topic</code></td>
<td>Prints messages of the topic to the screen</td>
</tr>
<tr>
<td><code>$rostopic info /topic</code></td>
<td>Print information about a topic</td>
</tr>
<tr>
<td><code>$rostopic type /topic</code></td>
<td>Prints the type of messages the topic publishes</td>
</tr>
<tr>
<td><code>$rostopic pub /topic type args</code></td>
<td>Publishes data to a topic</td>
</tr>
</tbody>
</table>
rostopic list

• Displays the list of current topics:

```
roiyeho@ubuntu:~$ rostopic list
/rosout
/rosout_agg
/turtle1/cmd_vel
/turtle1/color_sensor
/turtle1/pose
roiyeho@ubuntu:~$  
```
Publish to ROS Topic

• Use the **rostopic pub** command to publish messages to a topic

• For example, to make the turtle move forward at a 0.2m/s speed, you can publish a cmd_vel message to the topic /turtle1/cmd_vel:

```
$ rostopic pub /turtle1/cmd_vel geometry_msgs/Twist '{linear: {x: 0.2, y: 0, z: 0}, angular: {x: 0, y: 0, z: 0}}'
```

• To specify only the linear x velocity:

```
$ rostopic pub /turtle1/cmd_vel geometry_msgs/Twist '{linear: {x: 0.2}}'
```
Publish to ROS Topic

• Some of the messages like cmd_vel have a predefined timeout
• If you want to publish a message continuously, use the argument -r with the loop rate in Hz
• For example, to make the turtle turn in circles continuously, type:

$ rostopic pub /turtle1/cmd_vel -r 10 geometry_msgs/Twist '{angular: {z: 0.5}}'
Project 1 is assigned!

http://inside.mines.edu/~hzhang/Courses/CSCI473-573/assignment.html
Recall: ROS and Ubuntu Installation

• Option 1: Ubuntu 16.04 LTS + Kinetic (recommended)

• Option 2: Ubuntu 14.04 LTS + Indigo

• It is **NOT** necessary to make your code of Project 1 to work in both options.

• But your code and document (e.g., package.xml) should clearly explain which option you use in your implementation.
catkin build system
ROS packages
Building ROS nodes
catkin Build System

• **catkin** is the ROS build system
  • The set of tools that ROS uses to generate executable programs, libraries and interfaces

• The original ROS build system was **rosbuild**
  • Still used for older packages (before 2015)

• Implemented as custom CMake macros along with some Python code

FYI: CMake is an open-source, cross-platform family of tools designed to build, test and package software.
catkin Workspace

- A set of directories in which a set of related ROS code lives
- You can have multiple ROS workspaces, but you can only work in one of them at any one time
- Contains the following spaces:

<table>
<thead>
<tr>
<th>Space</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source space</td>
<td>Contains the source code of catkin packages. Each folder within the source space contains one or more catkin packages.</td>
</tr>
<tr>
<td>Build Space</td>
<td>is where CMake is invoked to build the catkin packages in the source space. CMake and catkin keep their cache information and other intermediate files here.</td>
</tr>
<tr>
<td>Development (Devel) Space</td>
<td>is where built targets are placed prior to being installed</td>
</tr>
<tr>
<td>Install Space</td>
<td>Once targets are built, they can be installed into the install space by invoking the install target.</td>
</tr>
</tbody>
</table>
workspace_folder/  -- WORKSPACE
   src/        -- SOURCE SPACE
      CMakeLists.txt -- The 'toplevel' CMake file
      package_1/
         CMakeLists.txt
         package.xml
         ...
      package_n/
         CMakeLists.txt
         package.xml
         ...
    build/      -- BUILD SPACE
       CATKIN_IGNORE -- Keeps catkin from walking this directory
    devel/      -- DEVELOPMENT SPACE (set by CATKIN_DEVEL_PREFIX)
       bin/
       etc/
       include/
       lib/
       share/
       .catkin
       env.bash
       setup.bash
       setup.sh
       ...
install/    -- INSTALL SPACE (set by CMAKE_INSTALL_PREFIX)
   bin/
   etc/
   include/
   lib/
   share/
   .catkin
   env.bash
   setup.bash
   setup.sh
   ...

catkin Workspace Layout
Creating a catkin Workspace

• Initially, the workspace will contain only the top-level CMakeLists.txt

```bash
$ mkdir ~/catkin_ws/src
$ cd ~/catkin_ws/src
$ catkin_init_workspace
```

• `catkin_make` command builds the workspace and all the packages within it

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

http://wiki.ros.org/cn/catkin/Tutorials/using_a_workspace
ROS Package

• ROS software is organized into packages, each of which contains some combination of code, data, and documentation

• A ROS package is simply a directory inside a catkin workspace that has a package.xml file in it

• Packages are the most atomic unit of build and the unit of release

• A package contains the source files for one node or more and configuration files

FYI: Extensible Markup Language (XML) is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable.
# Common Files and Directories

<table>
<thead>
<tr>
<th>Directory</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>include/</td>
<td>C++ include headers</td>
</tr>
<tr>
<td>src/</td>
<td>Source files</td>
</tr>
<tr>
<td>msg/</td>
<td>Folder containing Message (msg) types</td>
</tr>
<tr>
<td>srv/</td>
<td>Folder containing Service (srv) types</td>
</tr>
<tr>
<td>launch/</td>
<td>Folder containing launch files</td>
</tr>
<tr>
<td>package.xml</td>
<td>The package manifest</td>
</tr>
<tr>
<td>CMakeLists.txt</td>
<td>CMake build file</td>
</tr>
</tbody>
</table>
The Package Manifest

• **package.xml** defines properties of the package:
  • the package name, version numbers, authors, dependencies on other catkin packages, and more

```xml
<package>
  <name>foo_core</name>
  <version>1.2.4</version>
  <description>
    This package provides foo capability.
  </description>
  <maintainer email="ivana@willowgarage.com">Ivana Bildbotz</maintainer>
  <license>BSD</license>

  <url>http://ros.org/wiki/foo_core</url>
  <author>Ivana Bildbotz</author>

  <buildtool_depend>catkin</buildtool_depend>

  <build_depend>message_generation</build_depend>
  <build_depend>roscpp</build_depend>
  <build_depend>std_msgs</build_depend>

  <run_depend>message_runtime</run_depend>
  <run_depend>roscpp</run_depend>
  <run_depend>rospy</run_depend>
  <run_depend>std_msgs</run_depend>

  <test_depend>python-mock</test_depend>
</package>
```
Creating a ROS Package

• Change to the source directory of the workspace

```
$ cd ~/catkin_ws/src
```

• `catkin_create_pkg` creates a new package with the specified dependencies

```
$ catkin_create_pkg <package_name> [depend1] [depend2] [depend3]
```

• For example, create a `first_pkg` package:

```
$ catkin_create_pkg first_pkg std_msgs rospy roscpp
```

C++ First Node Example

```cpp
/*
 * hello.cpp
 *
 *  Created on: Feb 6, 2017
 *      Author: First Last
 */

#include "ros/ros.h"

int main(int argc, char **argv)
{
    ros::init(argc, argv, "hello");
    ros::NodeHandle nh;
    ros::Rate loop_rate(10);

    int count = 0;
    while (ros::ok()) // Keep spinning loop until user presses Ctrl+C
    {
        ROS_INFO_STREAM("hello world" << count);

        ros::spinOnce(); // Allow ROS to process incoming messages
        loop_rate.sleep(); // Sleep for the rest of the cycle
        count++;
    }

    return 0;
}
```
C++ First Node Example
ROS C++ Client Library

- **roscpp** is a ROS client implementation in C++
- Library doc can be found at: [http://docs.ros.org/api/roscpp/html/](http://docs.ros.org/api/roscpp/html/)
- ROS main header file is “ros/ros.h”

Here is a simple program example using **roscpp**:

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

int main(int argc, char **argv)
{
    ros::init(argc, argv, "hello");
    ros::NodeHandle nh;
    ros::Rate loop_rate(10);
    int count = 0;
    while (ros::ok()) // Keep spinning loop until user presses Ctrl+C
    {
        ROS_INFO_STREAM("hello world" << count);
        ros::spinOnce(); // Allow ROS to process incoming messages
        loop_rate.sleep(); // Sleep for the rest of the cycle
        count++;
    }
    return 0;
}
```

**roscpp** is a ROS client implementation in C++. The main parts of **roscpp** are:

- **ros::init()**: A version of **ros::init()** must be called before using any of the rest of the ROS system.
- **ros::NodeHandle**: Public interface to topics, services, parameters, etc.
- **ros::master**: Contains functions for querying information from the master
- **ros::this_node**: Contains functions for querying information about this process' node
- **ros::service**: Contains functions for querying information about services
- **ros::param**: Contains functions for querying the parameter service without the need for a **ros::NodeHandle**
- **ros::names**: Contains functions for manipulating ROS graph resource names
ROS Init

• A version of ros::init() must be called before using any of the rest of the ROS system

• Typical call in the main() function:

```cpp
ros::init(argc, argv, "Node Name");
```

• Node names must be unique in a running system

/*
* hello.cpp
* 
* Created on: Feb 6, 2017
*    Author: First Last
*/

#include "ros/ros.h"

int main(int argc, char **argv)
{
    ros::init(argc, argv, "hello");
    ros::NodeHandle nh;
    ros::Rate loop_rate(10);

    int count = 0;
    while (ros::ok()) // Keep spinning loop until user presses Ctrl+C
    {
        ROS_INFO_STREAM("hello world" << count);

        ros::spinOnce(); // Allow ROS to process incoming messages
        loop_rate.sleep(); // Sleep for the rest of the cycle
        count++;
    }

    return 0;
}
ros::NodeHandle

• The main access point to communications with the ROS system.
  • Provides public interface to topics, services, parameters, etc.

• Create a handle to this process’ node (after the call to ros::init()) by declaring:

```
ros::NodeHandle 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.
ros::Rate

- A class to help run loops at a desired frequency.
- Specify in the node’s desired rate to run in Hz

```cpp
ros::Rate loop_rate(10);
```

- ros::Rate::sleep() method
  - Sleeps for any leftover time in a cycle.
  - Calculated from the last time sleep, reset, or the constructor was called
### ros::ok()

- **Call** `ros::ok()` **to check** if the node should continue running
- `ros::ok()` **will return** false if:
  - a SIGINT is received (Ctrl-C)
  - the node has 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

```cpp
/*
 * hello.cpp
 *
 * Created on: Feb 6, 2017
 * Author: First Last
 */
#include "ros/ros.h"

int main(int argc, char **argv)
{
    ros::init(argc, argv, "hello");
    ros::NodeHandle nh;
    ros::Rate loop_rate(10);

    int count = 0;
    while (!ros::ok()) // Keep spinning loop until user presses Ctrl+C
    {
        ROS_INFO_STREAM("hello world" << count);
        ros::spinOnce(); // Allow ROS to process incoming messages
        loop_rate.sleep(); // Sleep for the rest of the cycle
        count++;
    }

    return 0;
}
```
ROS Logging

• ROS_INFO prints an informative message
  • ROS_INFO("My INFO message.");

• All messages are printed with their level and the current timestamp
  • [ INFO] [1356440230.837067170]: My INFO message.

• This function allows parameters as in printf:
  • ROS_INFO("My INFO message with argument: %f", val);

• Also, C++ STL streams are supported, e.g.:
  ROS_INFO_STREAM("My message with argument: " << val);

ROS comes with five classic logging levels: DEBUG, INFO, WARN, ERROR, and FATAL
**ROS Logging**

<table>
<thead>
<tr>
<th></th>
<th>Debug</th>
<th>Info</th>
<th>Warn</th>
<th>Error</th>
<th>Fatal</th>
</tr>
</thead>
<tbody>
<tr>
<td>stdout</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>stderr</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>log file</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>/rosout</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

- ROS also automatically logs all messages that use `ROS_INFO` to log files on the filesystem for you so that you can go back and analyze a test later.
- Your node's log file will be in `~/.ros/log` (defined by the ROS_LOG_DIR environment variable).
/*
 * hello.cpp
 *
 *  Created on: Feb 6, 2017
 *      Author: First Last
 */

#include "ros/ros.h"

int main(int argc, char **argv)
{
    ros::init(argc, argv, "hello");

    ros::NodeHandle nh;
    ros::Rate loop_rate(10);

    int count = 0;
    while (ros::ok()) // Keep spinning loop until user presses Ctrl+C
    {
        ROS_INFO_STREAM("hello world" << count);

        ros::spinOnce(); // Allow ROS to process incoming messages
        loop_rate.sleep(); // Sleep for the rest of the cycle
        count++;
    }

    return 0;
}
CMakeLists.txt

```cmake
# Find catkin macros and libraries
find_package(catkin REQUIRED COMPONENTS roscpp rospy std_msgs)

## Declare ROS messages and services
# add_message_files(FILES Message1.msg Message2.msg)
# add_service_files(FILES Service1.srv Service2.srv)

## Generate added messages and services
# generate_messages(DEPENDENCIES std_msgs)

## Declare catkin package
catkin_package()

## Specify additional locations of header files
include_directories(${catkin_INCLUDE_DIRS})

## Declare a cpp executable
add_executable(hello src/hello.cpp)

## Specify libraries to link a library or executable target against
target_link_libraries(hello ${catkin_LIBRARIES})
```

Before building your node, you should modify the generated CMakeLists.txt in the package
Build and Run the Node

• To build the package in the terminal call `catkin_make`.

• Make sure you have sourced your workspace's setup.sh file after calling `catkin_make`:

  ```bash
  $ cd ~/catkin_ws
  $ source ./devel/setup.bash
  ```

• Can add this line to your `.bashrc` startup file.

• Now you can use `roslrun` to run your node:

  ```bash
  $ roslrun first_pkg hello
  ```
Run the Node

```
/*
   * hello.cpp
   * Created on: Feb 6, 2017
   * Author: First Last
   */

#include "ros/ros.h"

int main(int argc, char **argv)
{
    ros::init(argc, argv, "hello");
    ros::NodeHandle nh;
    ros::Rate loop_rate(10);

    int count = 0;
    while (ros::ok()) // Keep spinning loop until user presses Ctrl+C
    {
        ROS_INFO_STREAM("hello world" << count);

        ros::spinOnce(); // Allow ROS to process incoming messages
        loop_rate.sleep(); // Sleep for the rest of the cycle
        count++;
    }

    return 0;
}
```
Final Project is pre-assigned (sort of)

Team-based project
Consider forming your team; consider topics

Research oriented topics are strongly encourages for both CSCI 473 and CSCI 573 students
Publishers and subscribers
Roslaunch
Example: Move Turtle Node
Publish/subscribe

- Topics implement a *publish/subscribe* communication mechanism
- Before nodes start to transmit data over topics, they must first announce, or *advertise*, both the topic name and the types of messages that are going to be sent
- Then they can start to send, or *publish*, the actual data on the topic.
- Nodes that want to receive messages on a topic can *subscribe* to that topic by making a request to roscore.
- After subscribing, all messages on the topic are delivered to the node that made the request.
Publish/subscribe

• In ROS, all messages on the same topic must be of the same data type
• Topic names often describe the messages that are sent over them
• E.g., on PR2 robots, the topic /wide_stereo/right/image_color is used for color images from the rightmost camera of the stereo pair
#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;
}
Topic Publisher

• Manages an advertisement on a specific topic
• 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 (message buffer)**
Topic Publisher

- To publish a message on a topic call `publish()`

- Example:

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

- The message’s type must agree with the type given as a template parameter to the advertise call
#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;
}
Talker and Listener

• We’ll now create a package with two nodes:
  • *talker* publishes messages to topic “chatter”
  • *listener* reads the messages from the topic and prints them out to the screen
Recall on Creating a catkin Workspace

• Initially, the workspace will contain only the top-level CMakeLists.txt

```
$ mkdir ~/catkin_ws/src
$ cd ~/catkin_ws/src
$ catkin_init_workspace
```

• `catkin_make` command builds the workspace and all the packages within it

```
cd ~/catkin_ws
catkin_make
```

http://wiki.ros.org/cn/catkin/Tutorials/using_a_workspace
# Listener.cpp

```cpp
#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;
}
```
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);
```

• 1\textsuperscript{st} parameter is the **topic name**
• 2\textsuperscript{nd} parameter is the **queue size**
• 3\textsuperscript{rd} parameter is the **callback function** to handle the message
ros::spin() vs ros::spinOnce()

ros::spinOnce();

This code asks ROS to execute all of the pending callbacks from all of the node's subscriptions, and then return control back to us.

ros::spin();

This alternative to ros::spinOnce() asks ROS to wait for and execute callbacks until the node shuts down. In other words, ros::spin() is roughly equivalent to this loop:

while(ros::ok()) {
    ros::spinOnce();
}

The question of whether to use ros::spinOnce() or ros::spin() comes down to this: Does your program have any repetitive work to do, other than responding to callbacks? If the answer is "No," then use ros::spin(). If the answer is "Yes," then a reasonable option is to write a loop that does that other work and calls ros::spinOnce() periodically to process callbacks.
Compile and 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})
```
Running the Nodes

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

$ rosrun chat_pkg talker
$ rosrun chat_pkg listener
rqt_graph

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

```
$ rosrun rqt_graph rqt_graph
```
ROS Names

- ROS names must be unique
- If the same node is launched twice, roscore directs the older node to exit
- To change the name of a node on the command line, the special `__name` remapping syntax can be used
- The following two shell commands would launch two instances of talker named talker1 and talker2

```
$ rosrun chat_pkg talker __name:=talker1
$ rosrun chat_pkg talker __name:=talker2
```
**roslaunch**

- A tool for easily launching multiple ROS nodes as well as setting parameters on the Parameter Server

- **roslaunch** operates on launch files which are XML files that specify a collection of nodes to launch along with their parameters
  - By convention these files have a suffix of .launch

- Syntax:

  ```
  $ roslaunch PACKAGE LAUNCH_FILE
  ```

- roslaunch automatically runs roscore for you

XML: Extensible Markup Language
Launch File Example

• Launch file for launching the talker and listener nodes:

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

• Each `<node>` tag includes attributes declaring the ROS graph name of the node, the package in which it can be found, and the type of node, which is the filename of the executable program

• `output="screen"` makes the ROS log messages appear on the launch terminal window
Launch File Example

```
$ roslaunch chat_pkg chat.launch
```

```
PARMETERS
  * /rosdistro: indigo
  * /rosversion: 1.11.8

NODER
  /listener (chat_pkg/listener)
  /talker (chat_pkg/talker)

ROS_MASTER_URI=http://localhost:11311

core service [/rosout] found
process[talker-1]: started with pid [4346]
  [INFO] [1415527311.166839414]: hello world 0
process[listener-2]: started with pid [4357]
  [INFO] [1415527311.266930155]: hello world 1
  [INFO] [1415527311.366885084]: hello world 2
  [INFO] [1415527311.466933045]: hello world 3
  [INFO] [1415527311.567014453]: hello world 4
  [INFO] [1415527311.567771438]: I heard: [hello world 4]
  [INFO] [1415527311.666931023]: hello world 5
  [INFO] [1415527311.667310888]: I heard: [hello world 5]
  [INFO] [1415527311.767668040]: hello world 6
  [INFO] [1415527311.768178187]: I heard: [hello world 6]
```
Moving Turtle Example

• Velocity Commands: To make a robot move in ROS we need to publish **Twist** messages to the topic **cmd_vel**
• This message has a linear component for the (x,y,z) velocities, and an angular component for the angular rate about the (x,y,z) axes

```
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z

geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```
A Move Turtle Node

- For the demo, we will create a new ROS package called my_turtle

```cpp
#include "ros/ros.h"
#include "geometry_msgs/Twist.h"

int main(int argc, char **argv)
{
    const double FORWARD_SPEED_MPS = 0.5;

    // Initialize the node
    ros::init(argc, argv, "move_turtle");
    ros::NodeHandle node;

    // A publisher for the movement data
    ros::Publisher pub = node.advertise<geometry_msgs::Twist>("turtle1/cmd_vel", 10);

    // Drive forward at a given speed. The robot points up the x-axis.
    // The default constructor will set all commands to 0
    geometry_msgs::Twist msg;
    msg.linear.x = FORWARD_SPEED_MPS;

    // Loop at 10Hz, publishing movement commands until we shut down
    ros::Rate rate(10);
    ROS_INFO("Starting to move forward");
    while (ros::ok()) {
        pub.publish(msg);
        rate.sleep();
    }
}
```

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

MoveTurtle.cpp
Launch File

• Add move_turtle.launch to your package:

```xml
<launch>
  <node name="turtlesim_node" pkg="turtlesim" type="turtlesim_node" />
  <node name="move_turtle" pkg="my_turtle" type="move_turtle" output="screen" />
</launch>
```

• Run the launch file:

```
$ roslaunch my_turtle move_turtle.launch
```
Move Turtle Demo

• You should see the turtle in the simulator constantly moving forward until it bumps into the wall
Print Turtle’s Pose

• In order to print the turtle’s pose we need to subscribe to the topic /turtle1/pose

• We can find the message type of the topic and its structure by running the command

```
$ rostopic type /turtle1/pose | rosmsg show
```

• The message turtlesim/Pose is defined in the turtlesim package, thus we need to include the header file “turtlesim/Pose.h” in the example code
#include "ros/ros.h"
#include "geometry_msgs/Twist.h"
#include "turtlesim/Pose.h"

// Topic messages callback
void poseCallback(const turtlesim::PoseConstPtr& msg)
{
    ROS_INFO("x: %.2f, y: %.2f", msg->x, msg->y);
}

int main(int argc, char **argv)
{
    const double FORWARD_SPEED_MPS = 0.5;

    // Initialize the node
    ros::init(argc, argv, "move_turtle");
    ros::NodeHandle node;

    // A publisher for the movement data
    ros::Publisher pub = node.advertise<geometry_msgs::Twist>("turtle1/cmd_vel", 10);

    // A listener for pose
    ros::Subscriber sub = node.subscribe("turtle1/pose", 10, poseCallback);
// Drive forward at a given speed. The robot points up the x-axis.
// The default constructor will set all commands to 0
geometry_msgs::Twist msg;
msg.linear.x = FORWARD_SPEED_MPS;

// Loop at 10Hz, publishing movement commands until we shut down
ros::Rate rate(10);
ROS_INFO("Starting to move forward");
while (ros::ok()) {
    pub.publish(msg);
    ros::spinOnce(); // Allow processing of incoming messages
    rate.sleep();
}

Print Turtle’s Pose

• `roslaunch my_turtle move_turtle.launch`
MoveTurtle.cpp

```cpp
int main(int argc, char **argv)
{
    const double FORWARD_SPEED_MPS = 0.5;
    string robot_name = string(argv[1]);

    // Initialize the node
    ros::init(argc, argv, "move_turtle");
    ros::NodeHandle node;

    // A publisher for the movement data
    ros::Publisher pub = node.advertise<geometry_msgs::Twist>(robot_name + "/cmd_vel", 10);

    // A listener for pose
    ros::Subscriber sub = node.subscribe(robot_name + "/pose", 10, poseCallback);

    geometry_msgs::Twist msg;
    msg.linear.x = FORWARD_SPEED_MPS;

    ros::Rate rate(10);
    ROS_INFO("Starting to move forward");
    while (ros::ok()) {
        pub.publish(msg);
        ros::spinOnce(); // Allow processing of incoming messages
        rate.sleep();
    }
}
```

• Refer to Chapter 3 of the book “A Gentle Introduction to ROS” by JasonM. O’Kane for more detailed discussion
Gazebo simulator
Turtlebot

Will be discussed very briefly…
ROS Stage Simulator (No longer active)

- [http://wiki.ros.org/simulator_stage](http://wiki.ros.org/simulator_stage)
- A 2D simulator that provides a virtual world populated by mobile robots, along with various objects for the robots to sense and manipulate
ROS Stage Simulator (No longer active)

- In perspective view of the robot
- In the Stage, the robots are considered as Players.

Where does this “player-stage” metaphor come from?
Where is this player-stage comes from?

All the world's a stage, And all the men and women merely players; ...

As You Like It, Act II, Scene VII [All the world’s a stage]

William Shakespeare, 1564 - 1616
Gazebo

• A multi-robot simulator
• Like Stage, it is capable of simulating a population of robots, sensors and objects, but does so in 3D
• Includes an accurate simulation of rigid-body physics and generates realistic sensor feedback
• Allows code designed to operate a physical robot to be executed in an artificial environment
• Gazebo is under active development at the OSRF (Open Source Robotics Foundation)
Gazebo

- ROS Indigo comes with Gazebo V2.2
- ROS Kinetic uses the Gazebo-7.x series
- Gazebo home page - http://gazebosim.org/
- Gazebo tutorials - http://gazebosim.org/tutorials
Gazebo Architecture

Gazebo consists of two processes:

- **Server**: Runs the physics loop and generates sensor data
  - *Executable*: gzserver
  - *Libraries*: Physics, Sensors, Rendering, Transport

- **Client**: Provides user interaction and visualization of a simulation.
  - *Executable*: gzclient
  - *Libraries*: Transport, Rendering, GUI
Running Gazebo from ROS

- To launch Gazebo type:

```bash
$ rosrun gazebo_ros gazebo
```
Inserting PR2 Robot
Modifying the World
Saving a World

• Once you are happy with a world it can be saved through the File->Save As menu.
• Enter my_world.sdf as the file name and click OK
Loading a World

• A saved world may be loaded on the command line:

$ gazebo my_world.sdf

• The filename must be in the current working directory, or you must specify the complete path
Example World Files

• Gazebo ships with a number of example worlds
• World files are found within the /worlds directory of your Gazebo resource path
  • A typical path might be /usr/share/gazebo-2.2
• In gazebo_ros package there are built-in launch files that load some of these world files
• For example, to launch willowgarage_world type:

```bash
$ roslaunch gazebo_ros willowgarage_world.launch
```
Willow Garage World
Clock

• You can start, pause and step through the simulation with the clock
• It is located at the bottom of the World View

• **Real Time Factor**: Displays how fast or slow the simulation is running in comparison to real time
  • A factor less than 1.0 indicates simulation is running slower than real time
  • Greater then 1.0 indicates faster than real time
Meet TurtleBot

• [http://wiki.ros.org/Robots/TurtleBot](http://wiki.ros.org/Robots/TurtleBot)
• A minimalist platform for ROS-based mobile robotics education and prototyping
• Has a small differential-drive mobile base
• Atop this base is a stack of laser-cut “shelves” that provide space to hold a netbook computer and depth camera and other devices
• Does not have a laser scanner
  • Despite this, mapping and navigation can work quite well for indoor spaces
Turtlebot Simulation

• To install Turtlebot simulation stack type:
  $ sudo apt-get install ros-indigo-turtlebot-gazebo ros-indigo-turtlebot-apps ros-indigo-turtlebot-rviz-launchers

• To launch a simple world with a Turtlebot, type:
  $ roslaunch turtlebot_gazebo turtlebot_world.launch
Moving Turtlebot with Teleop

• Let’s launch the teleop package so we can move it around the environment, run:

```
$ roslaunch turtlebot_teleop keyboard_teleop.launch
```