ФИСТ
УлГТУ

Robotics & Data Mining Summer School

Lesson 06. Robotics Operating System

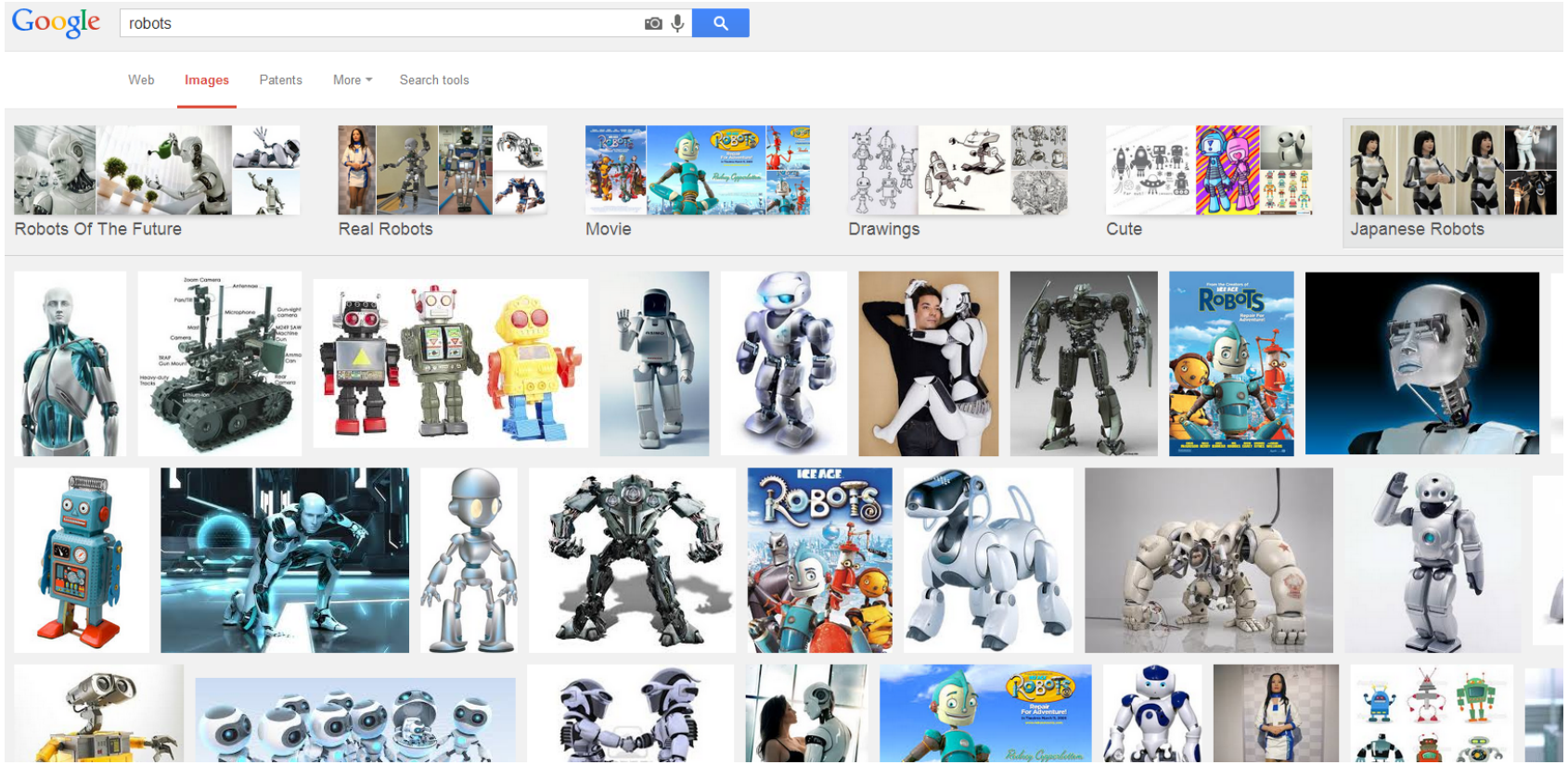
Kirill Svyatov, Alexander Miheev

Ulyanovsk State Technical University,

Faculty of Information Systems and Technologies

The Problem

- Lack of standards for robotics



What is ROS?

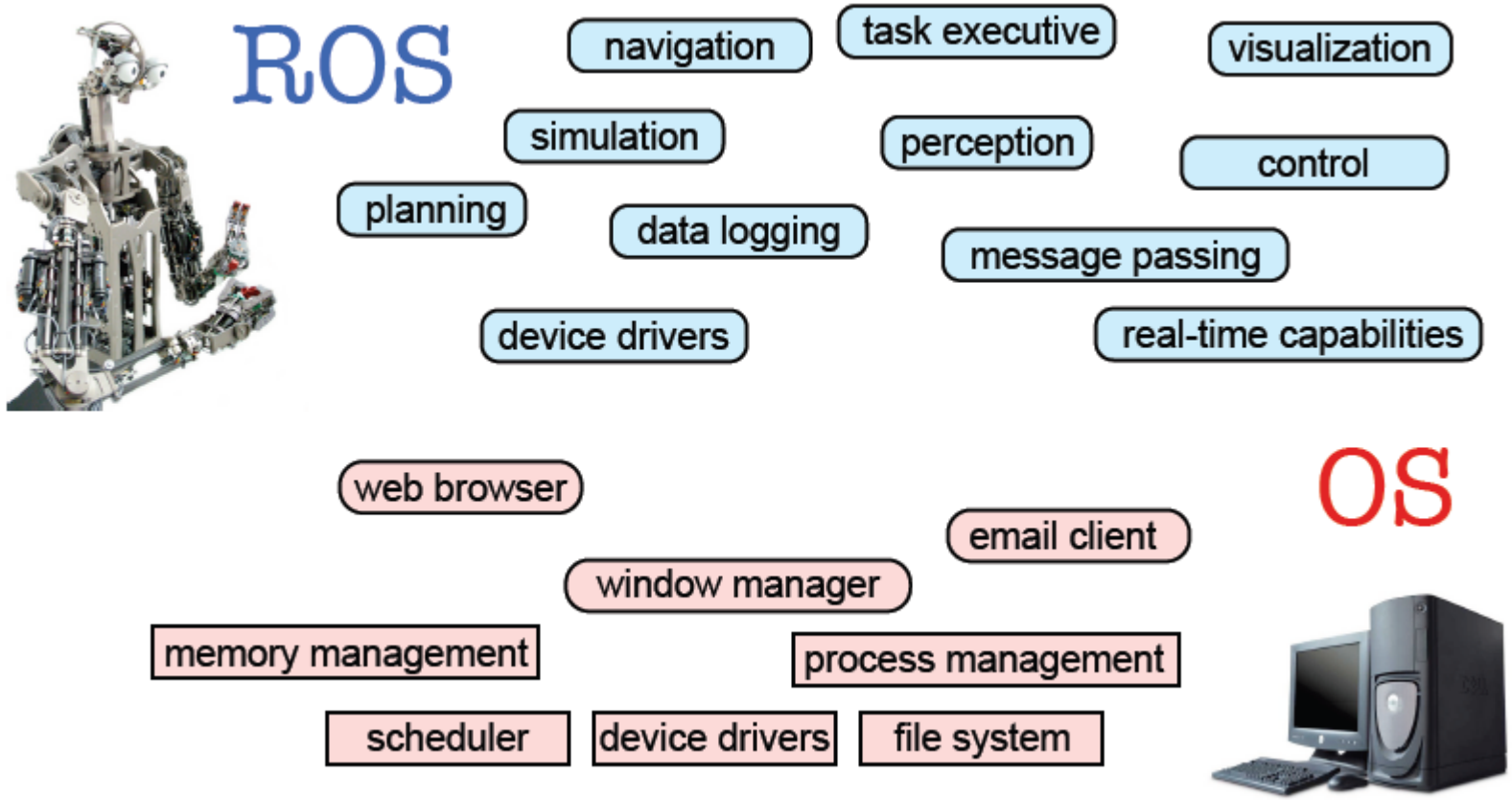
- ROS is an open-source **robot operating system**
- A set of software libraries and tools that help you build robot applications that work across a wide variety of robotic platforms
- Originally developed in 2007 at the Stanford Artificial Intelligence Laboratory and development continued at Willow 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 Main Features



Taken from Sachin Chitta and Radu Rusu (Willow Garage)

ROS Philosophy

- **Peer to Peer**
  - ROS systems consist of numerous small computer programs which connect to each other and continuously exchange *messages*
- **Tools-based**
  - There are many small, generic programs that perform tasks such as visualization, logging, plotting data streams, etc.
- **Multi-Lingual**
  - ROS software modules can be written in any language for which a *client library has been written.* Currently client libraries exist for C++, Python, LISP, Java, JavaScript, MATLAB, Ruby, and more.
- **Thin**
  - The ROS conventions encourage contributors to create stand-alone libraries and then *wrap those libraries so they send and receive messages to/from other ROS modules.*
- **Free and open source**

ROS Wiki

- http://wiki.ros.org/
- Installation: http://wiki.ros.org/ROS/Installation
- Tutorials: http://wiki.ros.org/ROS/Tutorials
- ROS Tutorial Videos
  - http://www.youtube.com/playlist?list=PLDC89965A56E6A8D6
- ROS Cheat Sheet
  - http://www.tedusar.eu/files/summerschool2013/ROScheatsheet.pdf

Robots using ROS

http://wiki.ros.org/Robots

Fraunhofer IPA Care-O-bot

Videre Erratic

TurtleBot

Aldebaran Nao

Lego NXT

Shadow Hand

Willow Garage PR2

iRobot Roomba

Robotnik Guardian

Merlin miabotPro

AscTec Quadrotor

CoroWare Corobot

Clearpath Robotics Husky

Clearpath Robotics Kingfisher

Festo Didactic Robotino

ROS Core Concepts

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

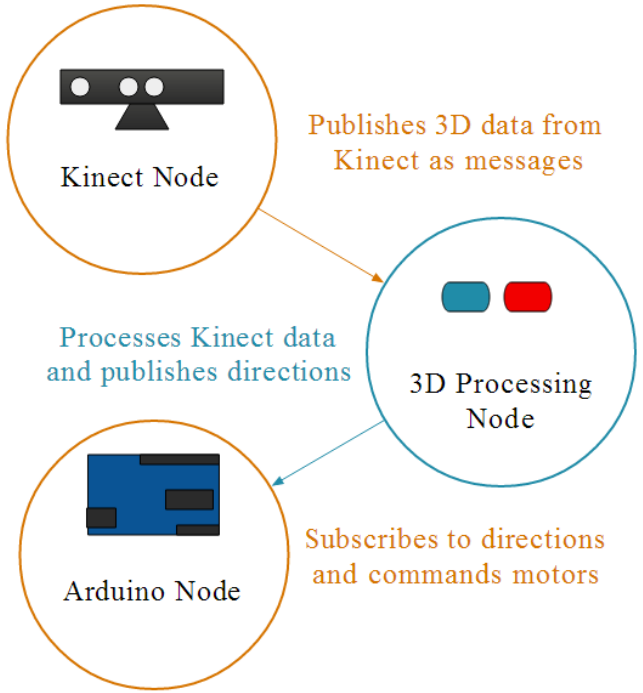
ROS Nodes

- 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

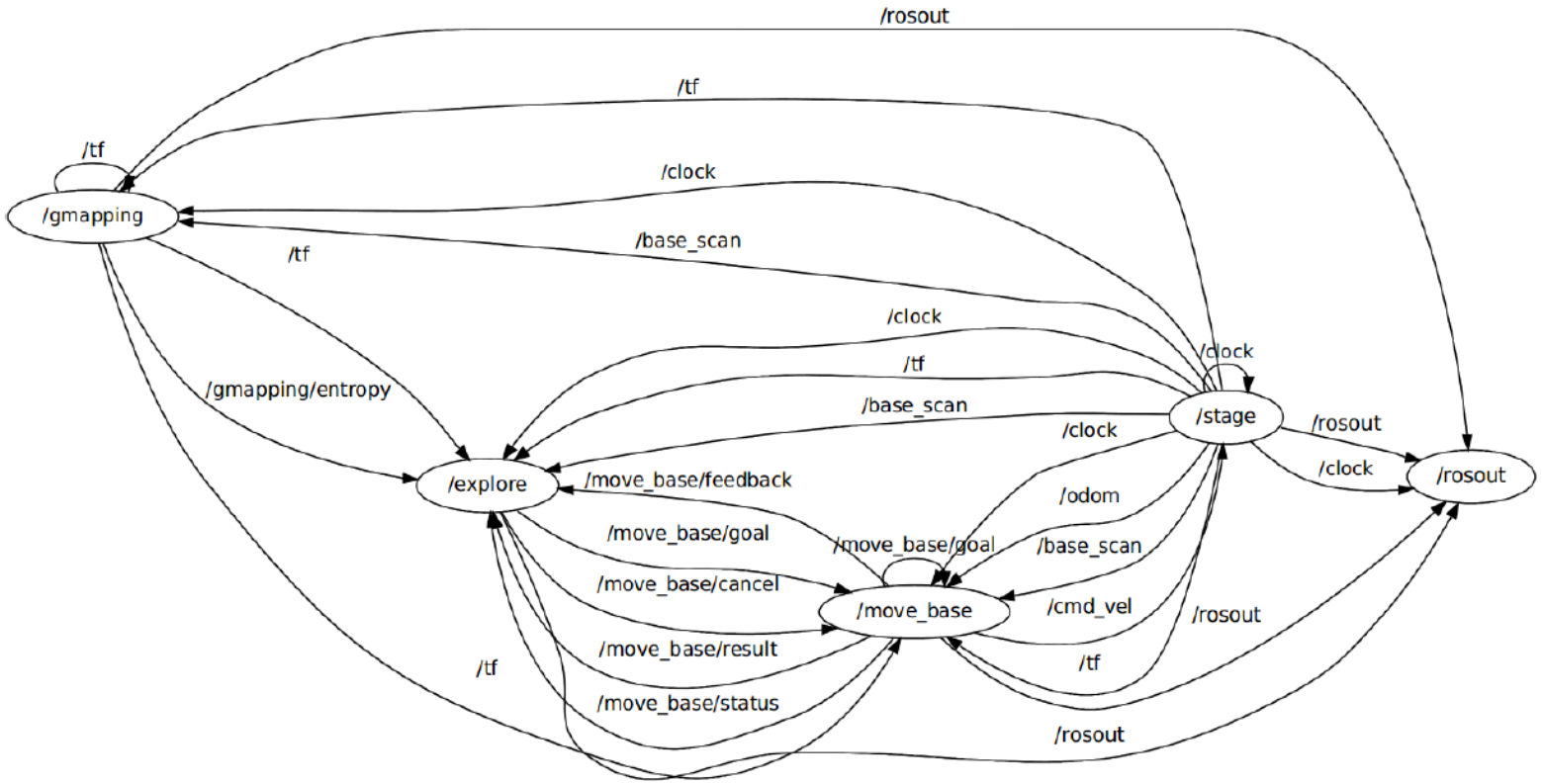
ROS Topics

- 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

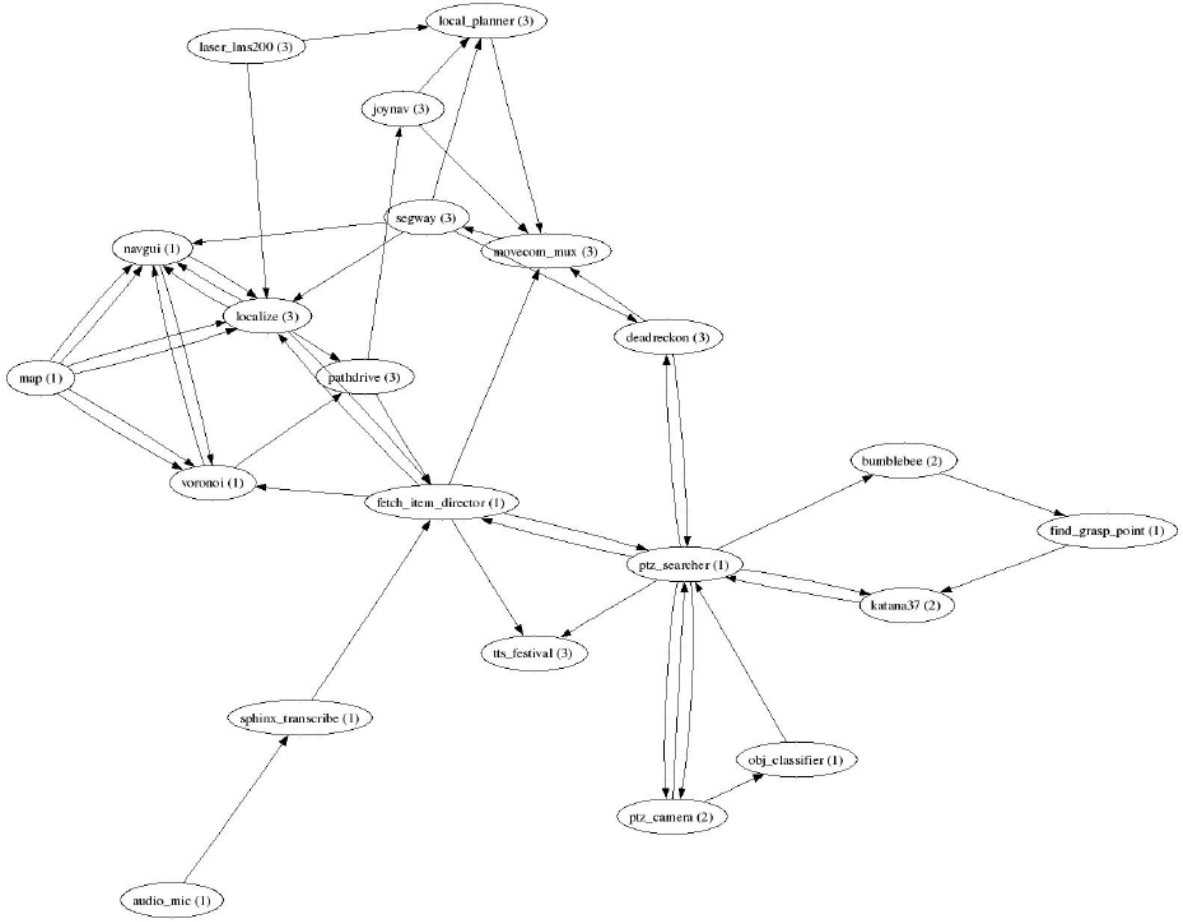
ROS Topics


Kinect Node
Publishes 3D data from Kinect as messages
3D Processing Node
Processes Kinect data and publishes directions
Arduino Node
Subscribes to directions and commands motors

The ROS Graph



Fetch an Item Graph



Taken from Programming Robots with ROS (Quigley et al.)

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

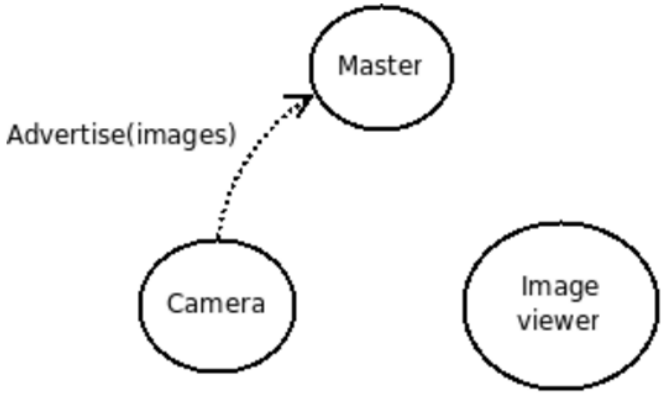
- Synchronous inter-node transactions / RPC
- Service/Client model: 1-to-1 request-response
- Service roles:
  - carry out remote computation
  - trigger functionality / behavior
- Example:
  - map_server/static_map – retrieves the current grid map used by the robot for navigation

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

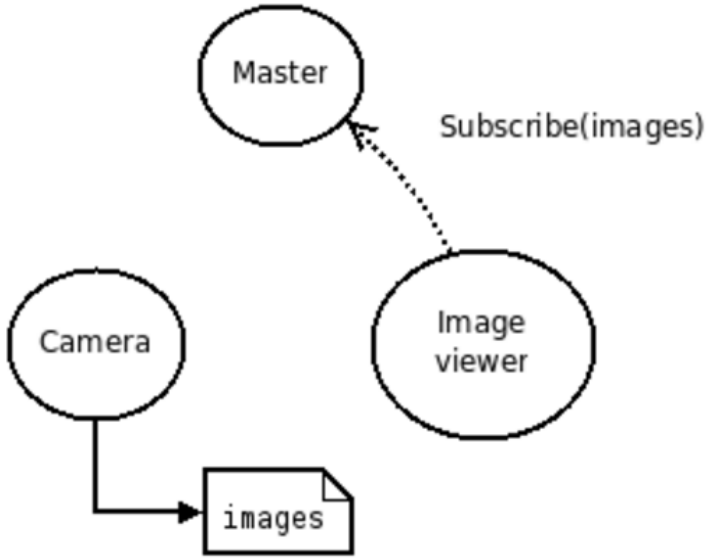
ROS Master

- Let’s say we have two nodes: a Camera node and an Image_viewer node
- Typically the camera node would start first notifying the master that it wants to publish images on the topic "images":



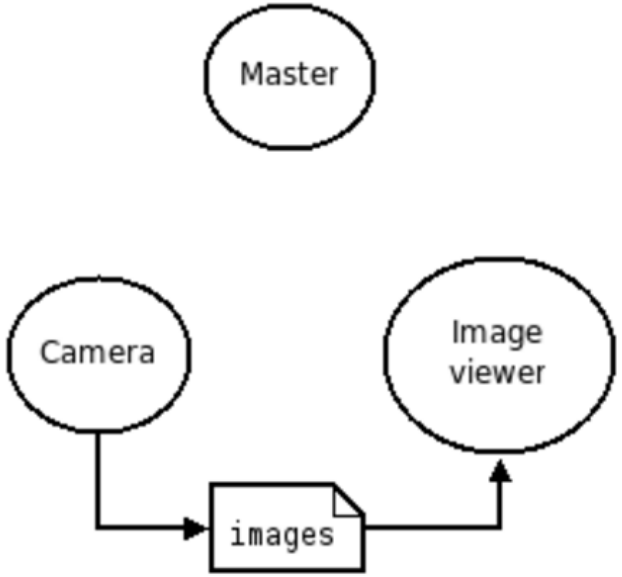
ROS Master

- Now, Image_viewer wants to subscribe to the topic "images" to see if there's maybe some images there:



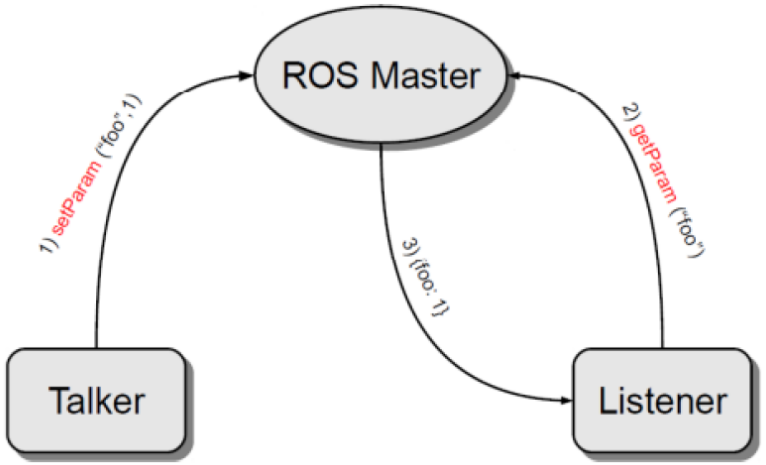
ROS Master

- Now that the topic "images" has both a publisher and a subscriber, the master node notifies Camera and Image_viewer about each others existence, so that they can start transferring images to one another:



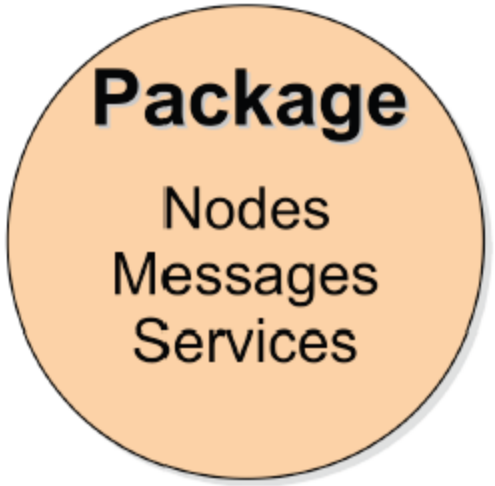
Parameter Server

- A shared, multi-variate dictionary that is accessible via network APIs
- Best used for static, non-binary data such as configuration parameters
- Runs inside the ROS master

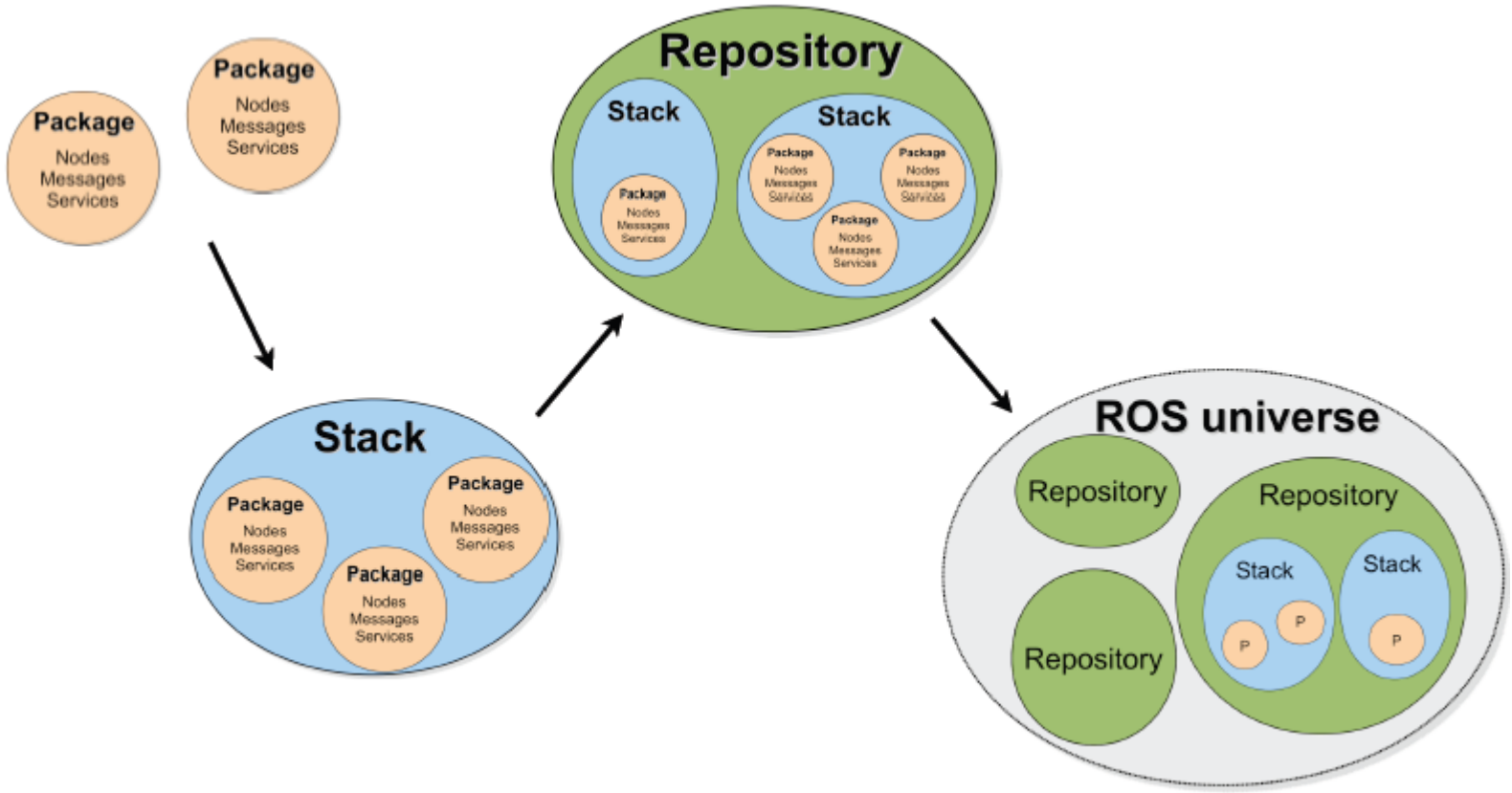


ROS Packages

- 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



ROS Package System



Taken from Sachin Chitta and Radu Rusu (Willow Garage)

ROS Distribution Releases

Distro	Release date	Poster	*Tuturtle*, turtle in tutorial	EOL date
ROS Kinetic Kame (**Recommended**)	May 23rd, 2016			May, 2021
ROS Jade Turtle	May 23rd, 2015			May, 2017
ROS Indigo Igloo	July 22nd, 2014			April, 2019 (Trusty EOL)
ROS Hydro Medusa	September 4th, 2013			May, 2015

ROS Supported Platforms

- ROS is currently supported only on Ubuntu
  - other variants such as Windows and Mac OS X are considered experimental (will be supported on ROS 2.0)
- ROS distribution supported is limited to <=3 latest Ubuntu versions
- ROS Jade supports the following Ubuntu versions:
  - Vivid (15.04)
  - Utopic (14.04)
  - Trusty (14.04 LTS)
- ROS Indigo supports the following Ubuntu versions:
  - Trusty (14.04 LTS)
  - Saucy (13.10)

ROS Installation

- If you already have Ubuntu installed, follow the instructions at:
  - http://wiki.ros.org/indigo/Installation/Ubuntu
  - You can also download a VM with ROS Indigo Pre-installed from here:
  - http://nootrix.com/downloads/#RosVM
- Two VMs are available: one with Ubuntu 32Bits and the other with Ubuntu 64Bits (.ova files)
- You can import this file into VirtualBox or VMWare

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 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)
  - If you used the pre-installed VM it’s already done for you

ROS Basic Commands

- roscore
- rosrun
- rosnode
- rostopic

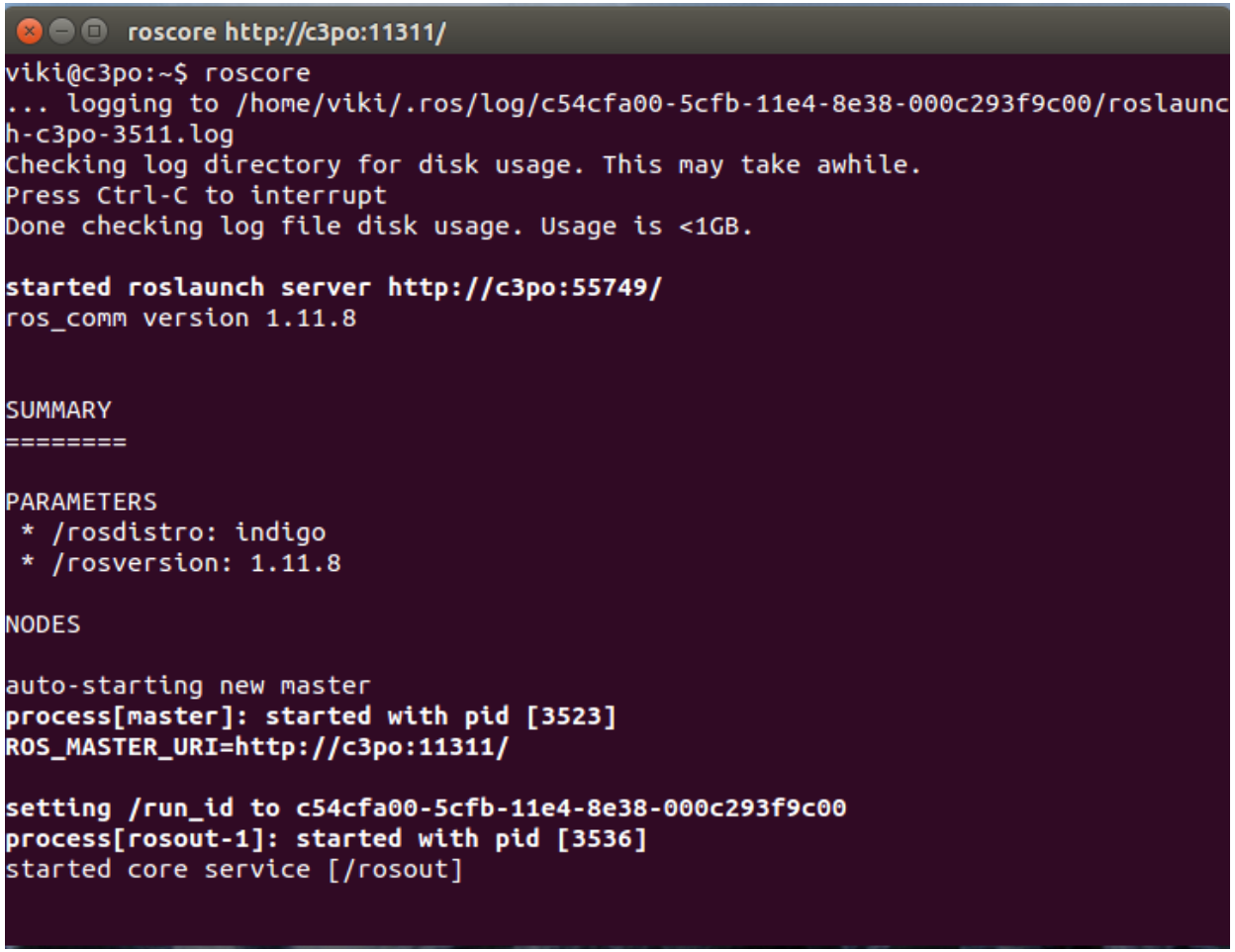
roscore

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

```
$ roscore
```

- roscore will start up:
  - a ROS Master
  - a ROS Parameter Server
  - a rosout logging node

roscore


roscore http://c3po:11311/
viki@c3po:~$ roscore
... logging to /home/viki/.ros/log/c54cfa00-5cfb-11e4-8e38-000c293f9c00/roslaunch-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

- rosrun allows you to run a node
- Usage:

```
$ rosrun <package> <executable>
```

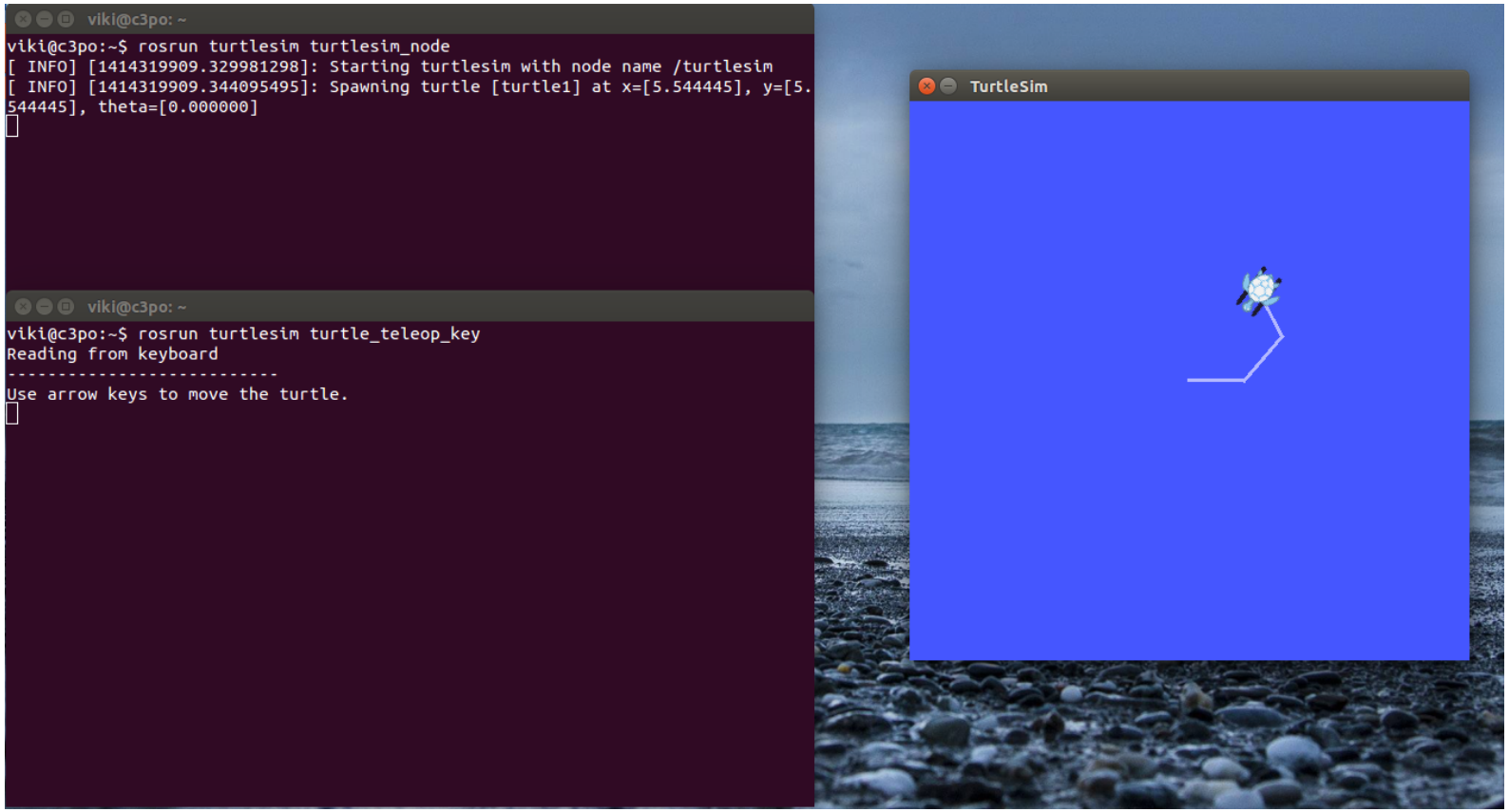
- Example:

```
$ rosrun turtlesim turtlesim_node
```

Demo - Turtlesim

- In separate terminal windows run:
  - roscore
  - rosrun turtlesim turtlesim_node
  - rosrun turtlesim turtle_teleop_key

Demo - Turtlesim

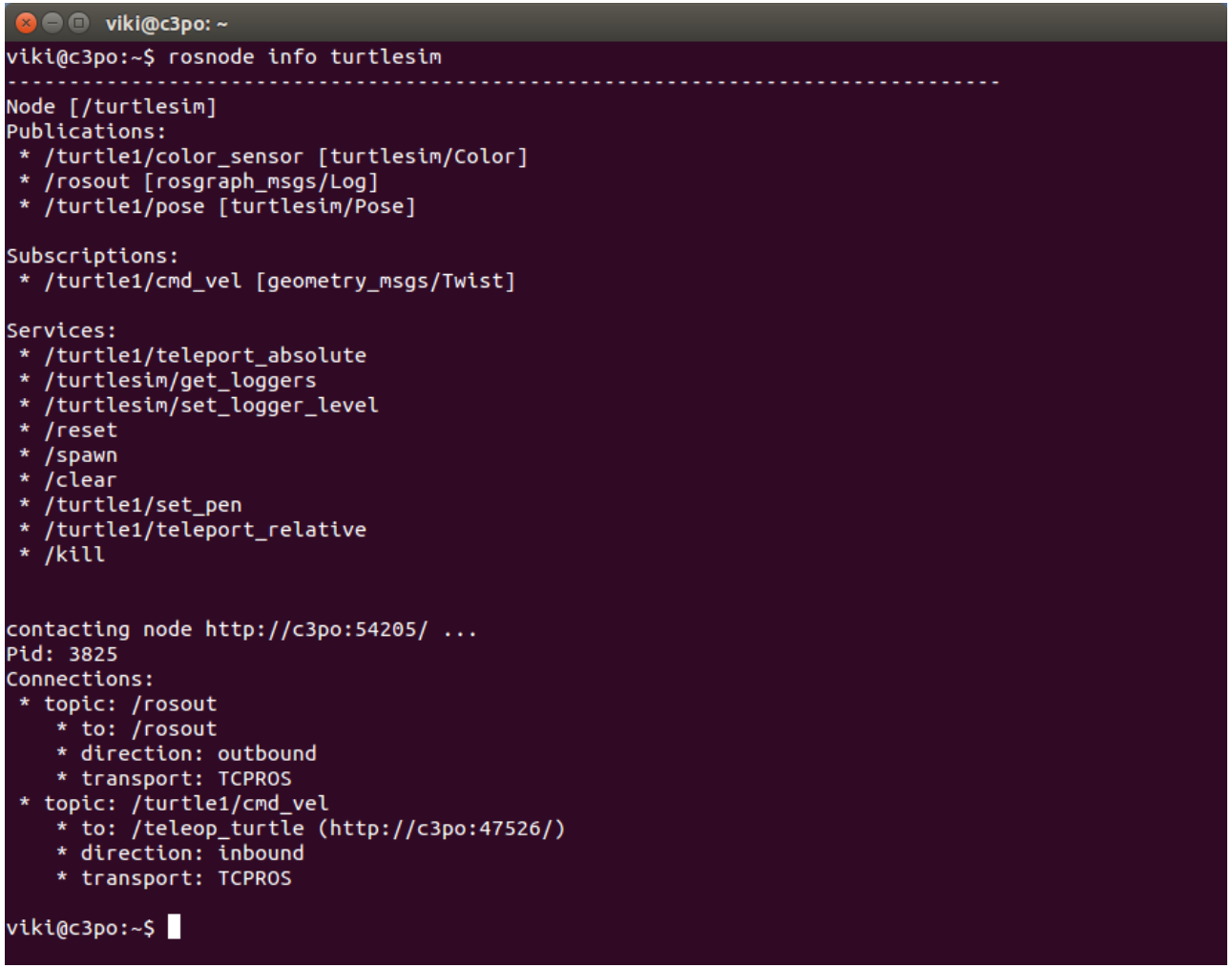


rosnode

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

Command	
$rosnode list	List active nodes
$rosnode ping	Test connectivity to node
$rosnode info	Print information about a node
$rosnode kill	Kill a running node
$rosnode machine	List nodes running on a particular machine

rosnode info



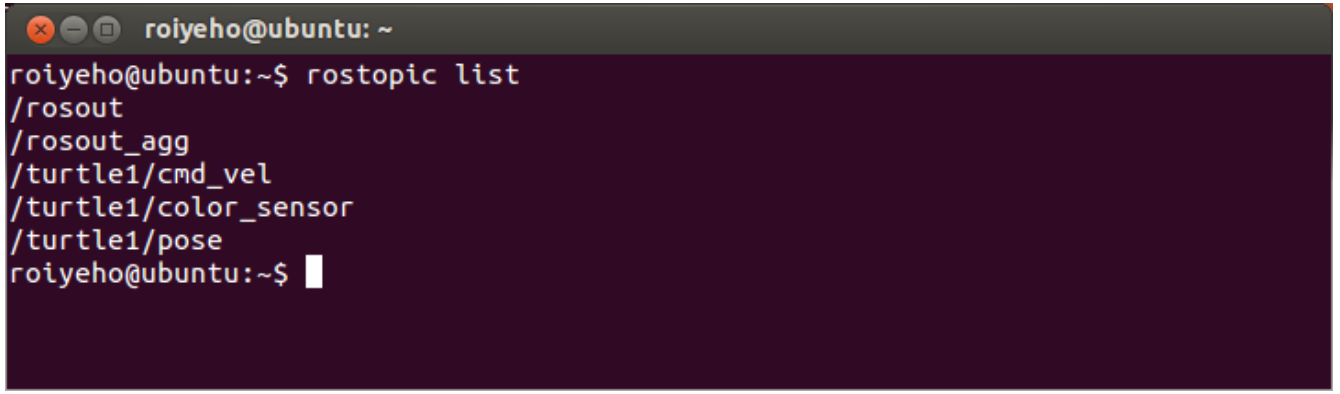
rostopic

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

Command	
$rostopic list	List active topics
$rosnode echo /topic	Prints messages of the topic to the screen
$rostopic info /topic	Print information about a topic
$rostopic type /topic	Prints the type of messages the topic publishes
$rostopic pub /topic type args	Publishes data to a topic

rostopic list

- Displays the list of current topics:



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:

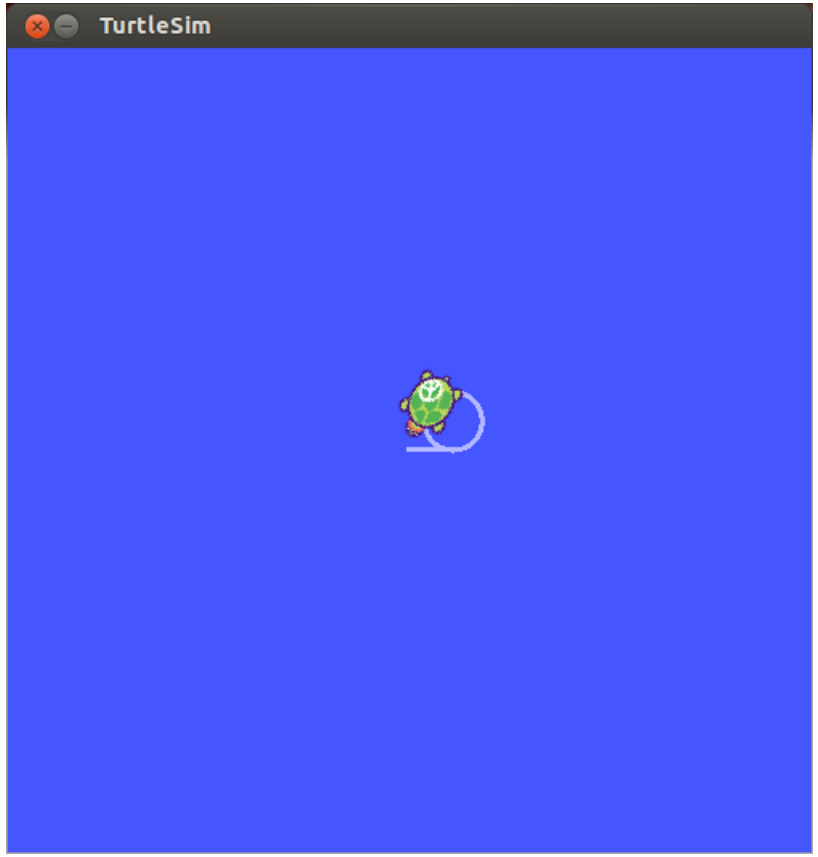
```
$ rostopic pub /turtle1/cmd_vel geometry_msgs/Twist '{linear: {x: 0.2}}'
```

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}}'
```

Publish to ROS Topic



Ex. 1

- Run the turtlesim node
- Send a command to turtlesim to move backwards continuously at 5Hz rate

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
- Implemented as custom CMake macros along with some Python code

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:

Source space	Contains the source code of catkin packages. Each folder within the source space contains one or more catkin packages.
Build Space	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.
Development (Devel) Space	is where built targets are placed prior to being installed
Install Space	Once targets are built, they can be installed into the install space by invoking the install target.

catkin Workspace Layout

```
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
    ...
```

ROS Development Setup

- Create a new catkin workspace
- Create a new ROS package
- Download and configure Eclipse
- Create Eclipse project file for your package
- Import package into Eclipse
- Write the code
- Update the make file
- Build the package

Creating a catkin Workspace

- Creating a Workspace Tutorial

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

- Initially, the workspace will contain only the top-level CMakeLists.txt
- Note that in the ready-made ROS Indigo VM you already have a catkin_ws workspace with the beginner_tutorials package

Building catkin Workspace

- **catkin_make** command builds the workspace and all the packages within it

```
cd ~/catkin_ws
catkin_make
```

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

Common Files and Directories

Directory	Explanation
include/	C++ include headers
src/	Source files
msg/	Folder containing Message (msg) types
srv/	Folder containing Service (srv) types
launch/	Folder containing launch files
package.xml	The package manifest
CMakeLists.txt	CMake build file

The Package Manifest

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

The Package Manifest

- Example

```
<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

- Creating a ROS Package Tutorial
- 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
```

Integrate Eclipse with ROS

- Make an entry in the Unity Dash for easier access

  $sudo gedit /usr/share/applications/eclipse.desktop

```
[Desktop Entry]
Name=Eclipse
Type=Application
Exec=bash -i -c "/opt/eclipse/eclipse"
Terminal=false
Icon=/opt/eclipse/icon.xpm
Comment=Integrated Development Environment
NoDisplay=false
Categories=Development;IDE
Name[en]=eclipse.desktop
```

  - The bash –i - c command will cause your IDE's launcher icon to load your ROS-sourced shell environment before launching eclipse

Make Eclipse Project Files

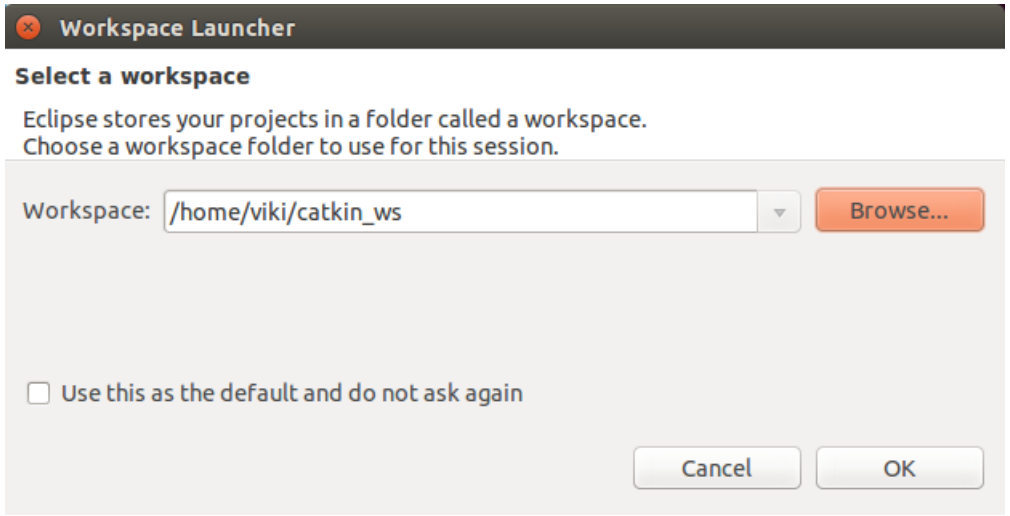
- Go to workspace directory and run catkin_make with options to generate eclipse project files:

```
$ cd ~/catkin_ws
$ catkin_make --force-cmake -G"Eclipse CDT4 - Unix Makefiles"
```

- The project files will be generated in ~/catkin_ws/build

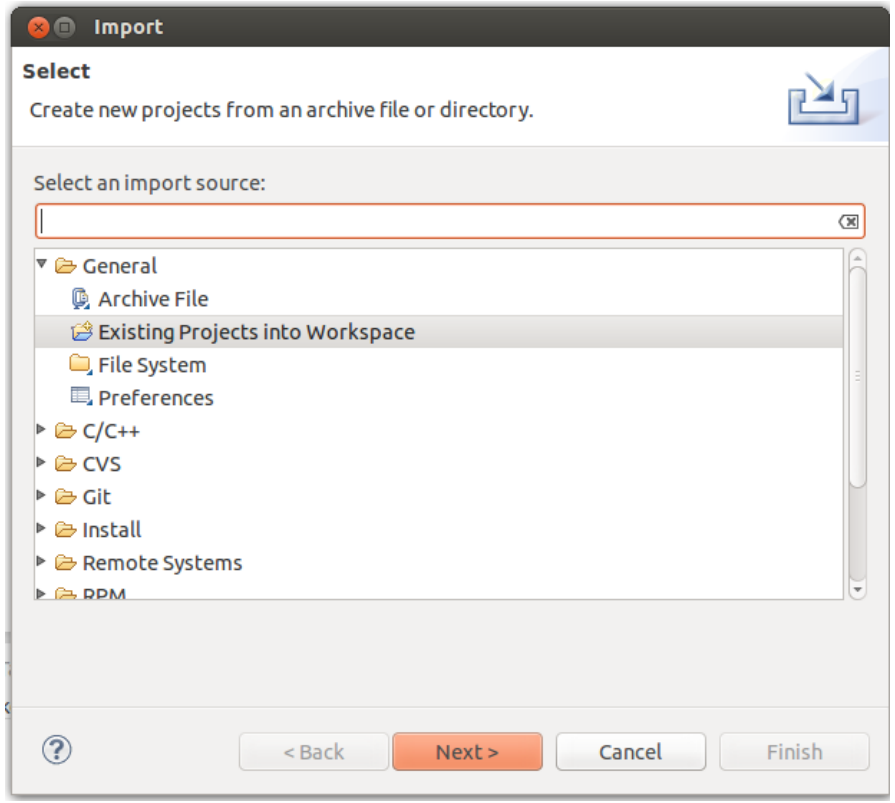
Import the Project into Eclipse

- Now start Eclipse
- Choose catkin_ws folder as the workspace folder



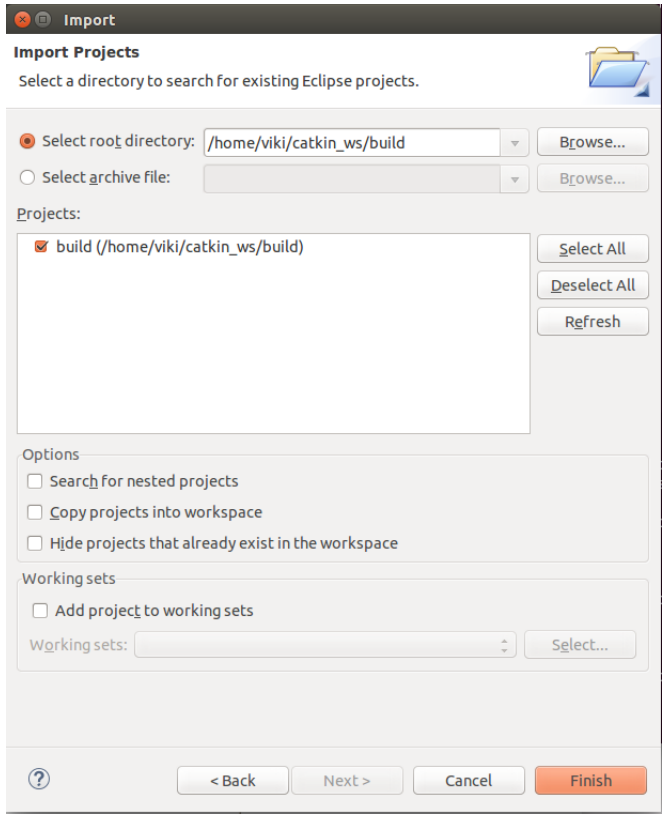
Import the Project into Eclipse

- Choose File --> Import --> General --> Existing Projects into Workspace



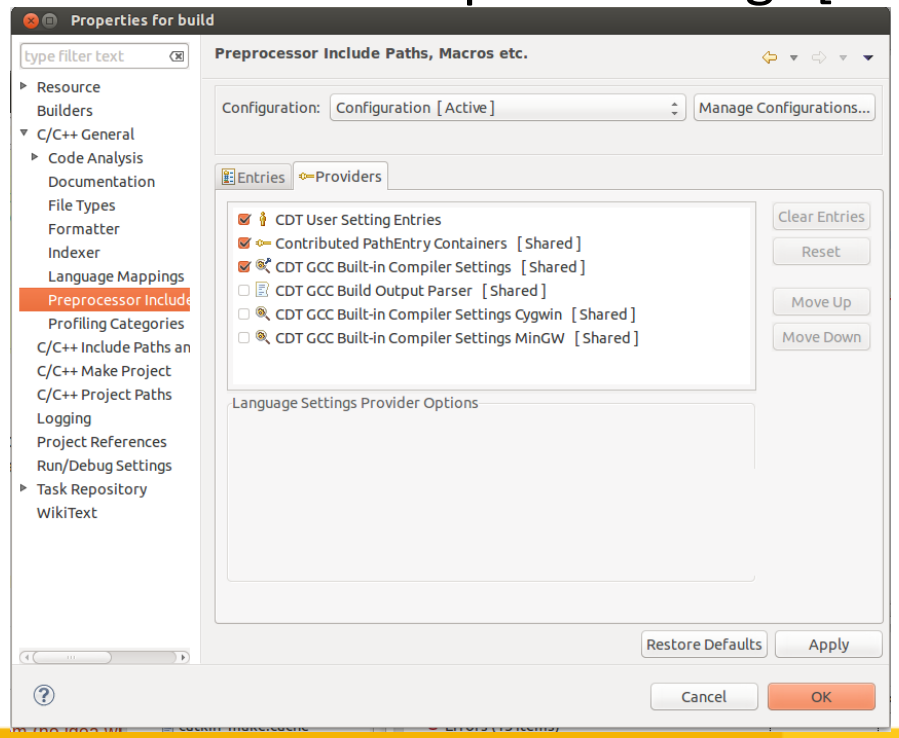
Import the Project into Eclipse

- Now import the project from the ~/catkin_ws/build folder



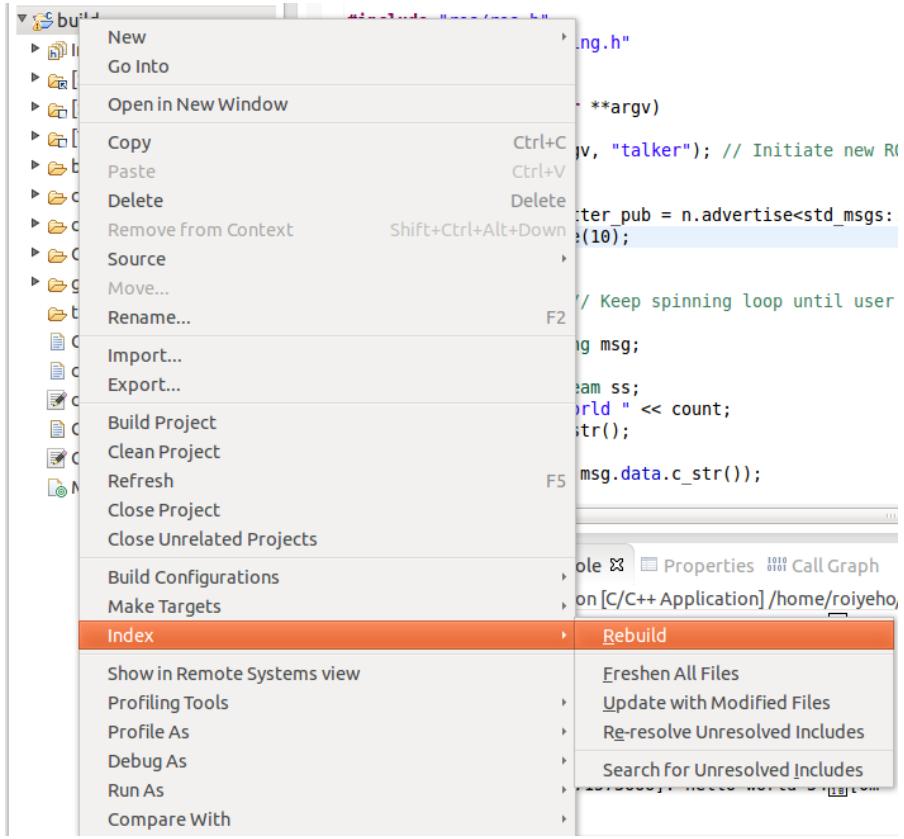
Fix Preprocessor Include Paths

- By default, the intellisense in Eclipse won't recognize the system header files (like <string>). To fix that:
  - Go to Project Properties --> C/C++ General --> Preprocessor Include Paths, Macros, etc. --> Providers tab
  - Check CDT GCC Built-in Compiler Settings [Shared]



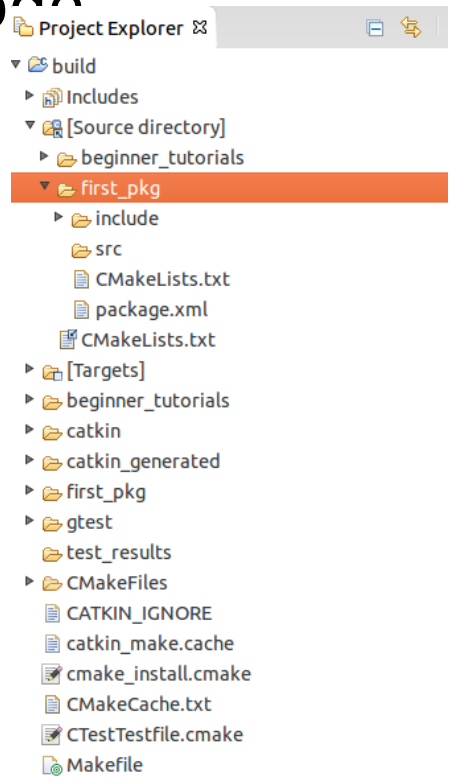
Fix Preprocessor Include Paths

- After that rebuild the C/C++ index by Right click on project -> Index -> Rebuild



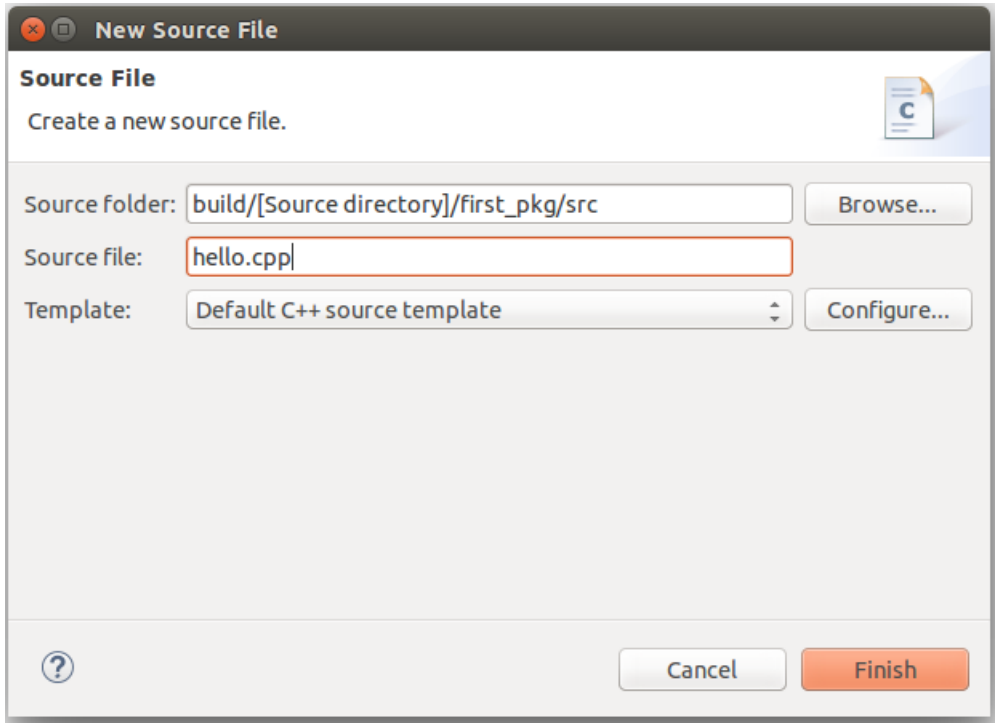
Project Structure

- Eclipse provides a link "Source directory" within the project so that you can edit the source code



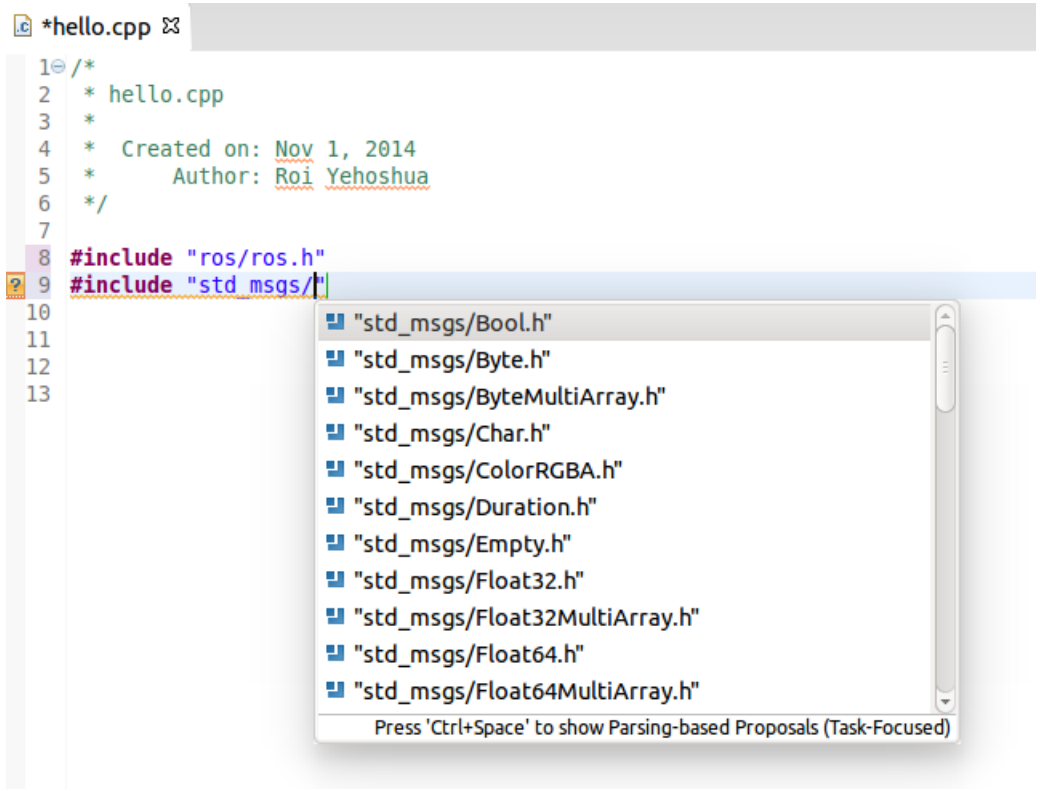
Add New Source File

- Right click on **src** and select **New** –> **Source File**, and create a file named hello.cpp



Code Completion

- Use Eclipse standard shortcuts to get code completion (i.e., Ctrl+Space)



C++ First Node Example

```
/*
 * hello.cpp
 *
 *  Created on: Nov 1, 2014
 *      Author: Roi Yehoshua
 */

#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 C++ Client Library

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

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:

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

- Node names must be unique in a running system

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;
```

  - The first NodeHandle constructed will fully initialize the current node
  - The last NodeHandle destructed will close down the node

ros::Rate

- A class to help run loops at a desired frequency.
- Specify in the c'tor the destired rate to run in Hz

```
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)
  - 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

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);
- ROS comes with five classic logging levels: DEBUG, INFO, WARN, ERROR, and FATAL
- Also, C++ STL streams are supported, e.g.:

```
ROS_INFO_STREAM("My message with argument: " << val);
```

ROS Logging

- 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)

	Debug	Info	Warn	Error	Fatal
stdout	X	X			
stderr			X	X	X
log file	X	X	X	X	X
/rosout	X	X	X	X	X

Building Your Node

- Before building your node, you should modify the generated CMakeLists.txt in the package
- The following slide shows the changes that you need to make in order to create the executable for the node

CMakeLists.txt

```
cmake_minimum_required(VERSION 2.8.3)
project(first_pkg)

## 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})
```

Building Your Nodes

- To compile the project in Eclipse press Ctrl-B
- To build the package in the terminal call catkin_make

Running the Node From Terminal

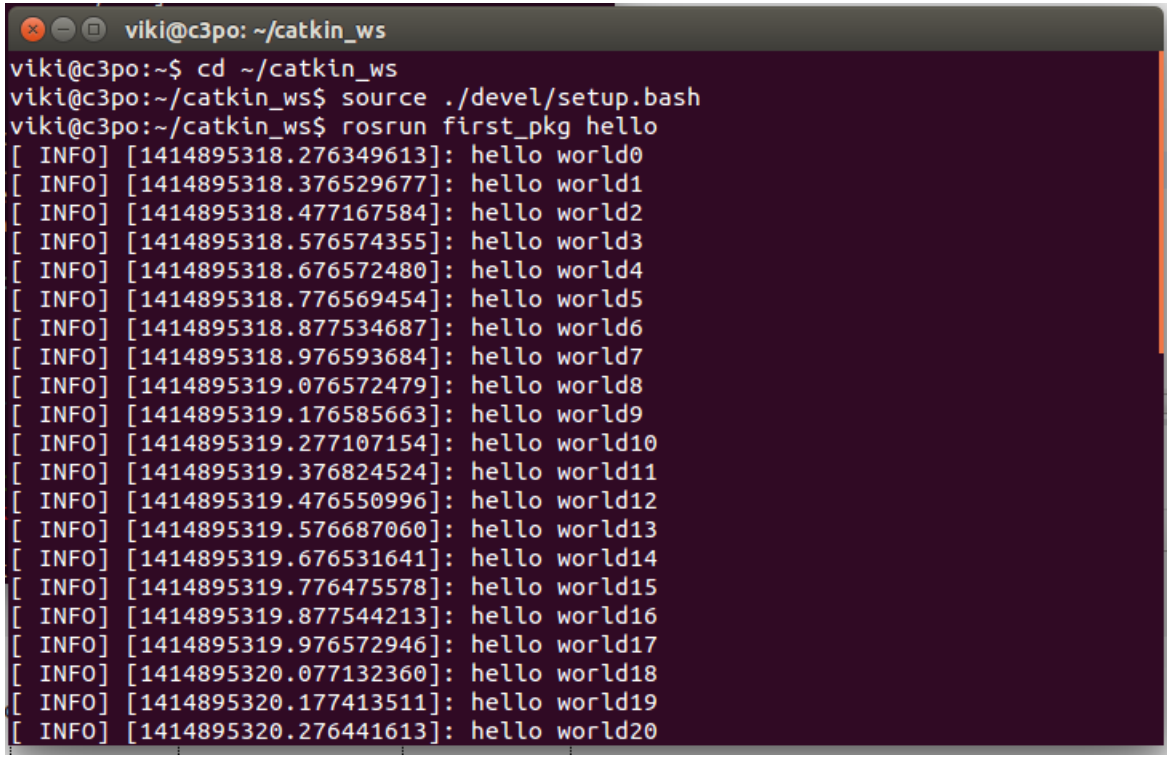
- Make sure you have sourced your workspace's setup.sh file after calling catkin_make:

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

  - Can add this line to your .bashrc startup file
- Now you can use rosrun to run your node:

```
$ rosrun first_pkg hello
```

Running the Node From Terminal

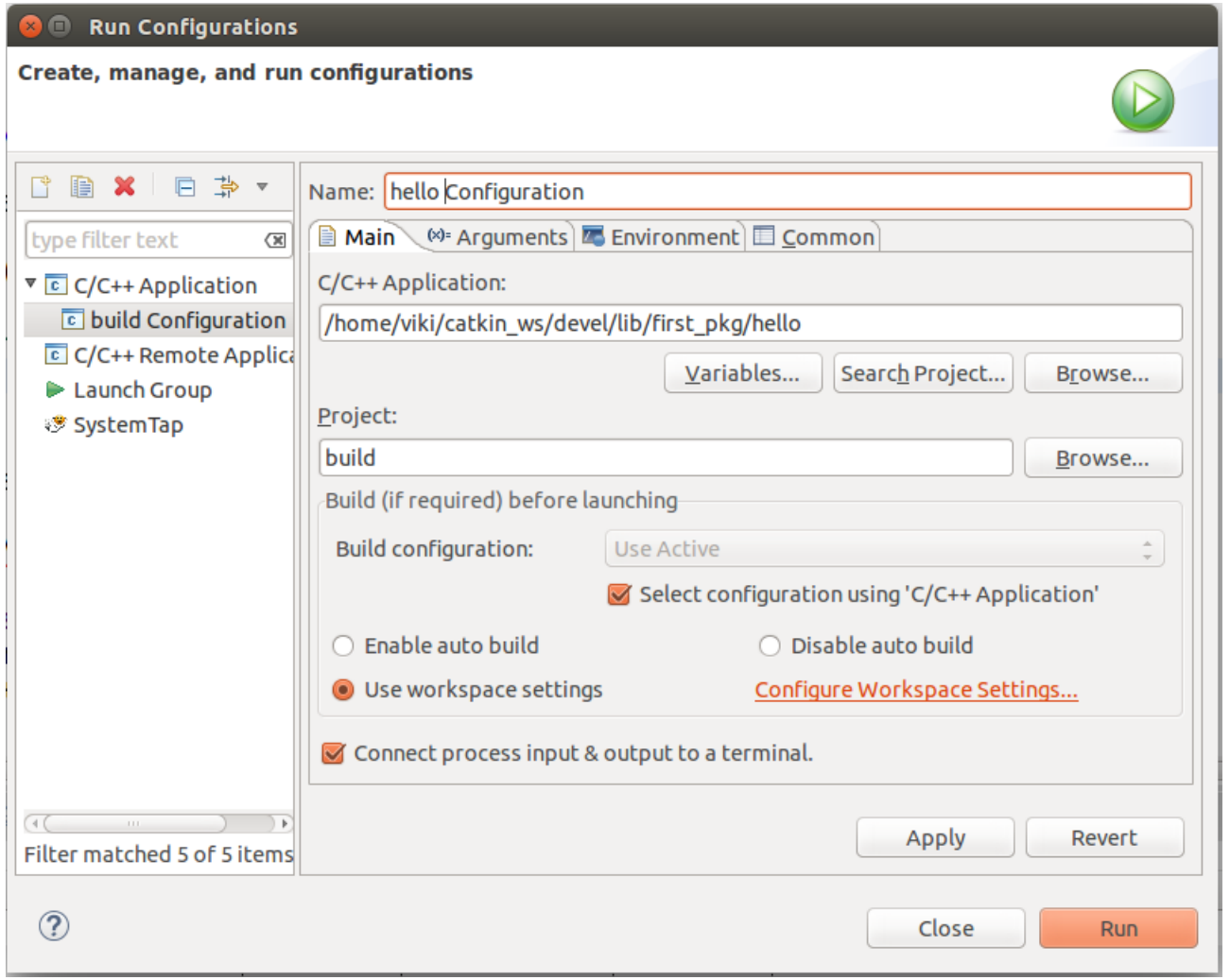


Running the Node Inside Eclipse

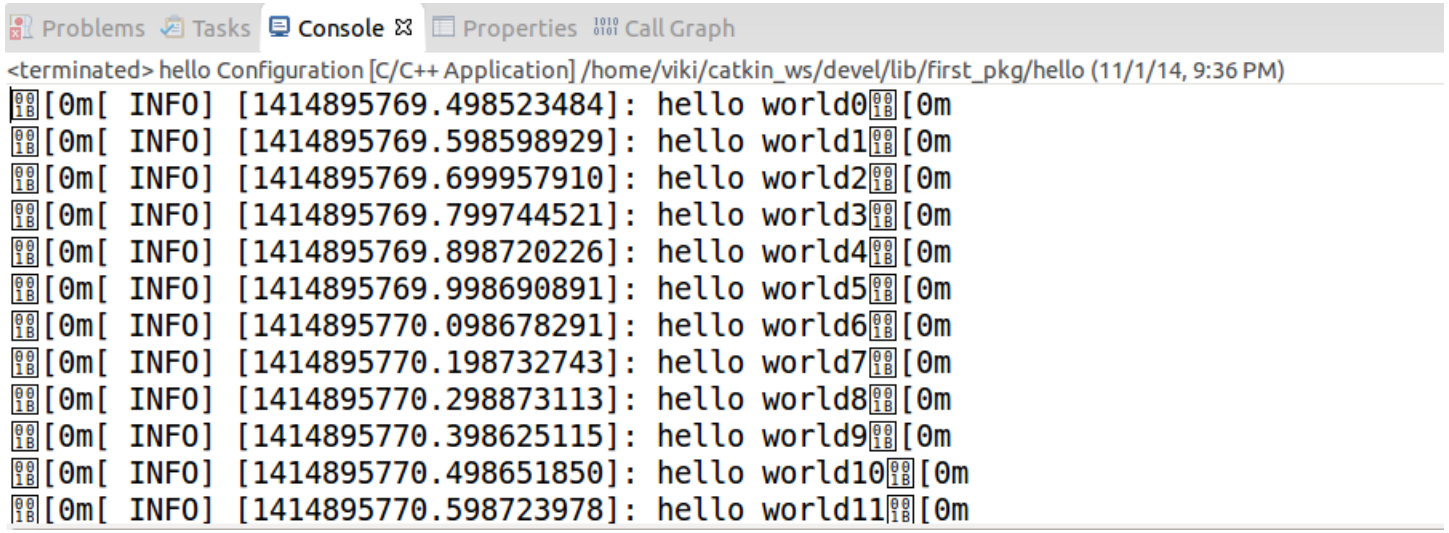
- Create a new launch configuration, by clicking on Run --> Run configurations... --> C/C++ Application (double click or click on New).
- Select the correct binary on the main tab (use the Browse… button)

  ~/catkin_ws/devel/lib/first_pkg/hello
- Make sure roscore is running in a terminal
- Click Run

Running the Node Inside Eclipse



Running the Node Inside Eclipse



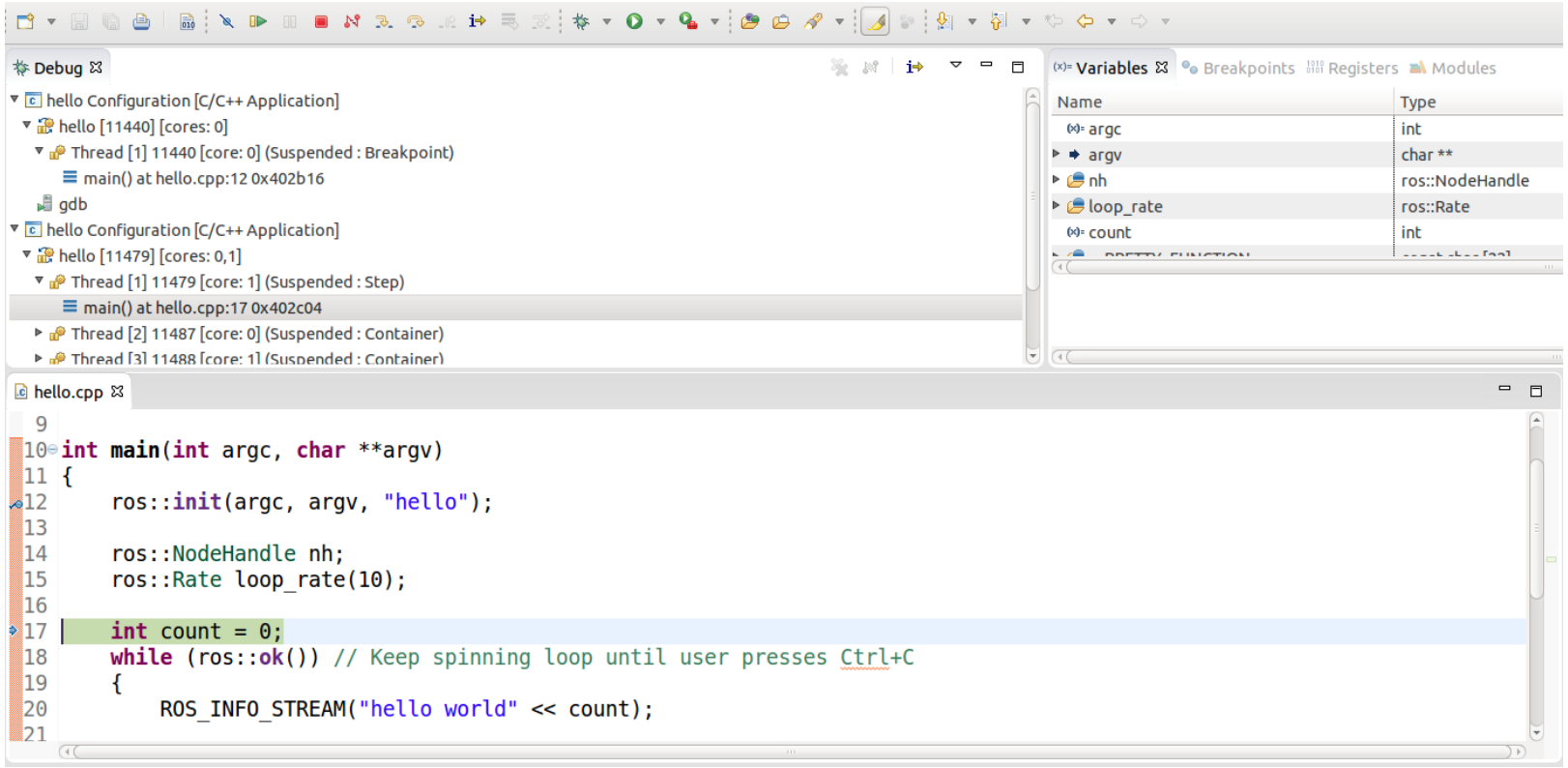
Debugging

- To enable debugging, you should first execute the following command in catkin_ws/build:

```
$ cmake ../src -DCMAKE_BUILD_TYPE=Debug
```

- Restart Eclipse
- Then you will be able to use the standard debugging tools in Eclipse

Debugging


Debug
hello Configuration [C/C++ Application]
hello [11440] [cores: 0]
Thread [1] 11440 [core: 0] (Suspended : Breakpoint)
main() at hello.cpp:12 0x402b16
gdb
hello Configuration [C/C++ Application]
hello [11479] [cores: 0,1]
Thread [1] 11479 [core: 1] (Suspended : Step)
main() at hello.cpp:17 0x402c04
Thread [2] 11487 [core: 0] (Suspended : Container)
Variables
Breakpoints
Registers
Modules
Name
Type
argc
int
argv
char **
nh
ros::NodeHandle
loop_rate
ros::Rate
count
int
hello.cpp
9
10 int main(int argc, char **argv)
11 {
12 ros::init(argc, argv, "hello");
13
14 ros::NodeHandle nh;
15 ros::Rate loop_rate(10);
16
17 int count = 0;
18 while (ros::ok()) // Keep spinning loop until user presses Ctrl+C
19 {
20 ROS_INFO_STREAM("hello world" << count);
21

Ex. 2

- Create a new ROS package called "timer_package"
- Create a node in this package called "timer_node"
- The node should print to the console the current time every 0.5 second

ROS Communication Types

Type	Best used for
Topic	One-way communication, especially if there might be multiple nodes listening (e.g., streams of sensor data)
Service	Simple request/response interactions, such as asking a question about a node's current state
Action	Most request/response interactions, especially when servicing the request is not instantaneous (e.g., navigating to a goal location)

ROS Topics

- Topics implement a *publish/subscribe* communication mechanism
  - one of the more common ways to exchange data in a distributed system.
- 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.

ROS Topics

- 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
- For example, on the PR2 robot, the topic /wide_stereo/right/image_color is used for color images from the rightmost camera of the wide-angle stereo pair

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:

```
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

Topic Publisher

- To publish a message on a topic call **publish()**
- Example:

```
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

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
- First create the package chat_pkg

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

- Open the package source directory in Eclipse and add a C++ source file named Talker.cpp
- Copy the following code into it

Talker.cpp

```cpp
#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);
```

  - 1st parameter is the topic name
  - 2nd parameter is the queue size
  - 3rd parameter is the function to handle the message

Listener.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;
}
```

ros::spin() vs ros::spinOnce()

- **ros::spin()** gives control over to ROS, which allows it to call all user callbacks
  - will not return until the node has been shutdown, either through a call to ros::shutdown() or a Ctrl-C.

```
#include <ros/callback_queue.h>
ros::NodeHandle n;
while (ros::ok())
{
  ros::getGlobalCallbackQueue()->callAvailable(ros::WallDuration(0.1));
}
```

- **ros::spinOnce()** calls the callbacks waiting to be called at that point in time

```
#include <ros/callback_queue.h>

ros::getGlobalCallbackQueue()->callAvailable(ros::WallDuration(0));
```

ros::spin() vs ros::spinOnce()

- A common pattern is to call spinOnce() periodically:

```
ros::Rate r(10); // 10 hz
while (should_continue)
{
  ... do some work, publish some messages, etc. ...
  ros::spinOnce();
  r.sleep();
}
```

Using Class Methods as Callbacks

- Suppose you have a simple class, Listener:

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

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

```
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_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:

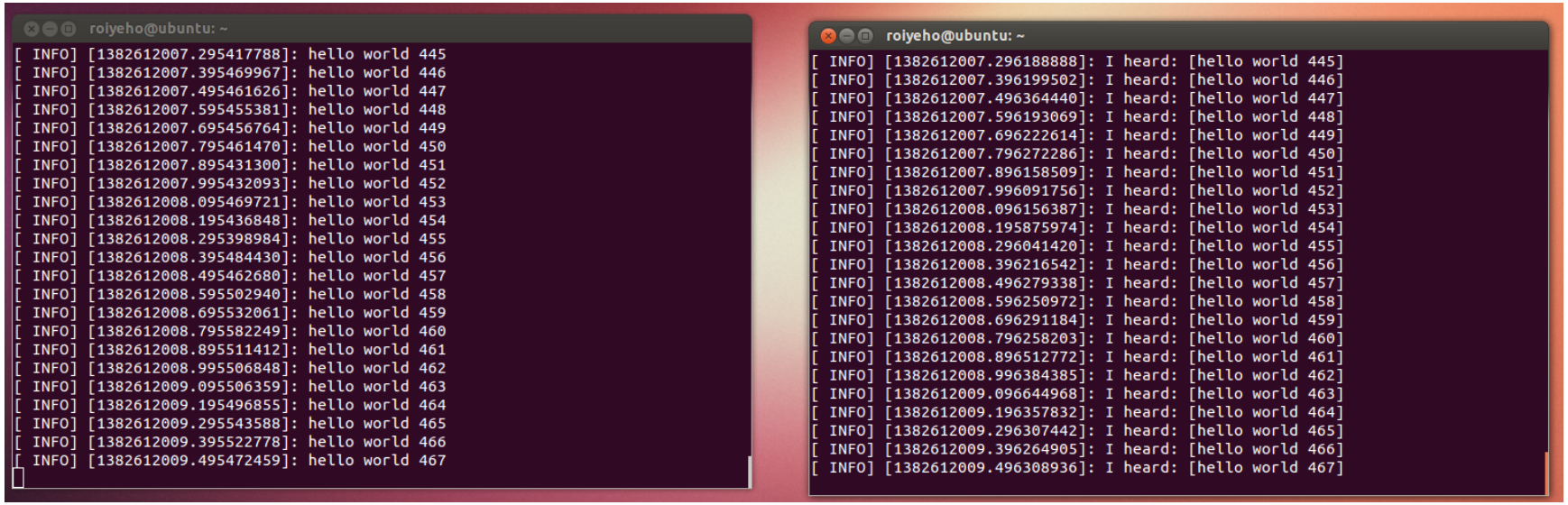
```
cd ~/catkin_ws
catkin_make
```

- This will create two executables, talker and listener, at ~/catkin_ws/devel/lib/chat_pkg

Running the Nodes From Terminal

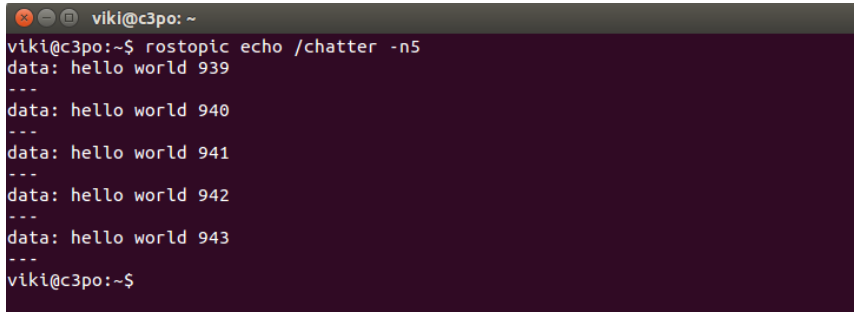
- Run roscore
- Run the nodes in two different terminals:

```
$ rosrun chat_pkg talker
$ rosrun chat_pkg listener
```



Running the Nodes From Terminal

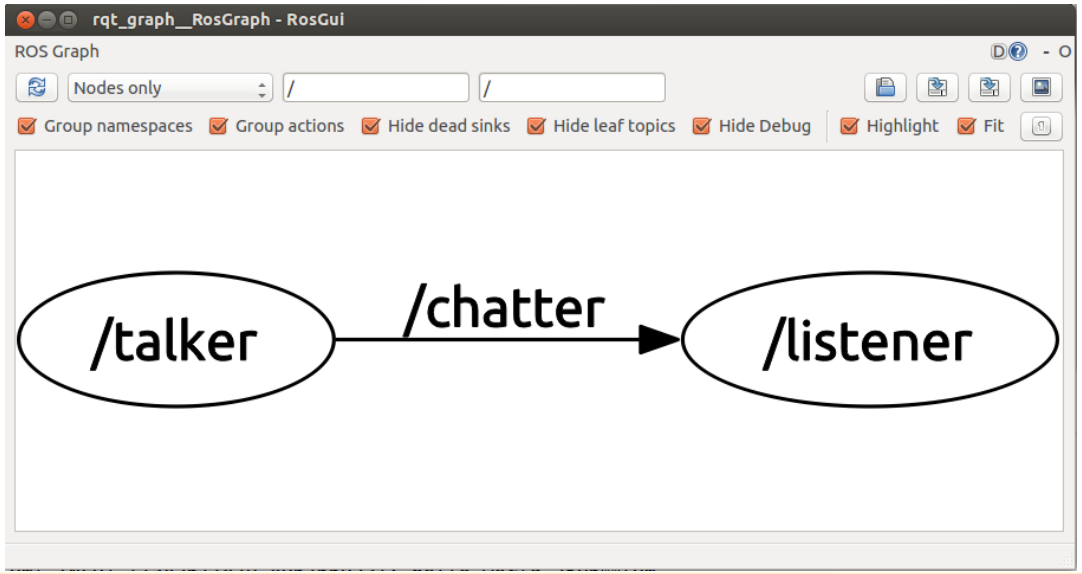
- You can use rosnode and rostopic to debug and see what the nodes are doing
- Examples:
  - $rosnode info /talker
  - $rosnode info /listener
  - $rostopic list
  - $rostopic info /chatter
  - $rostopic echo /chatter



rqt_graph

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

```
$ rosrun rqt_graph rqt_graph
```

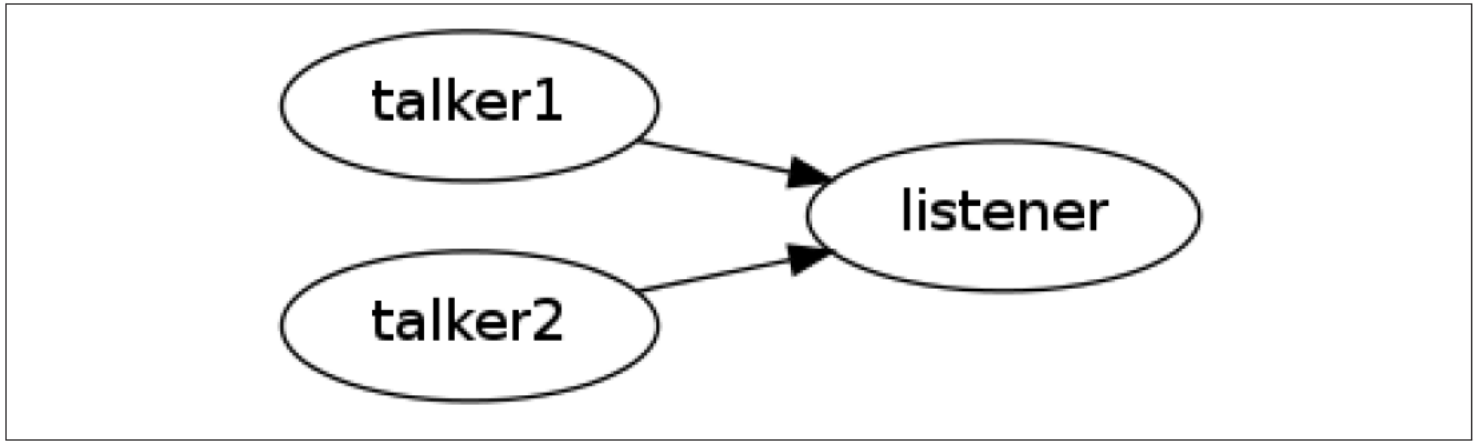


ROS Names

- 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
```

ROS Names



Instantiating two talker programs and routing them to the same receiver

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

Launch File Example

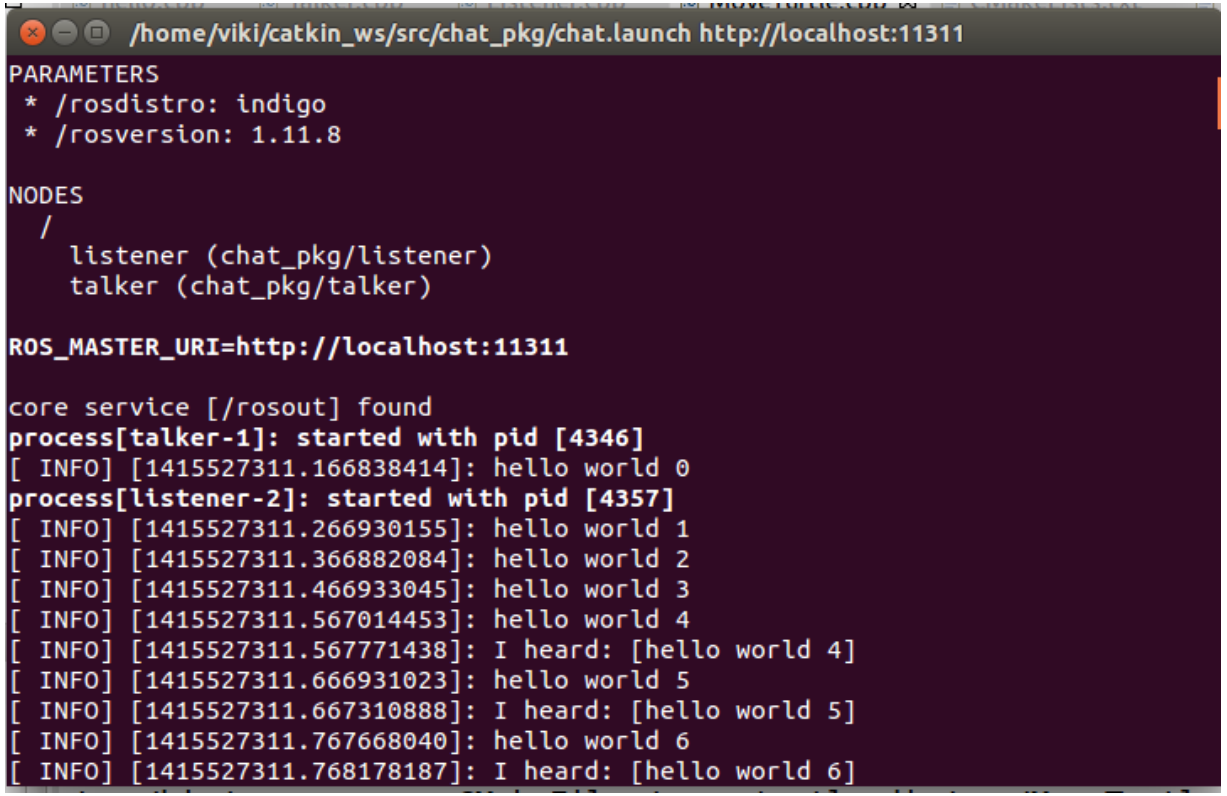
- Launch file for launching the talker and listener nodes:

```
<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
```



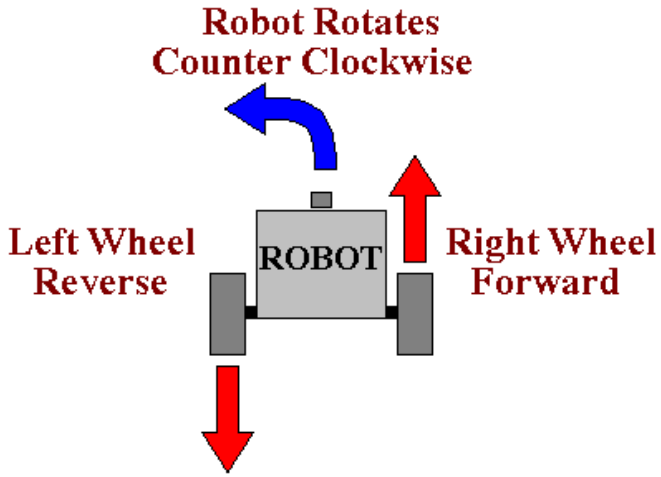
Velocity Commands

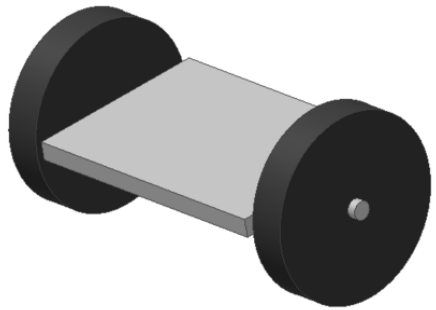
- To make a robot move in ROS we need to publish **Twist** messages to the topic **cmd_vel**
- This message has a linear component for the (x,y,z) velocities, and an angular component for the angular rate about the (x,y,z) axes

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

Differential Drive Robots

- A differential wheeled robot consists of two independently actuated wheels, located on its two sides
- The robot moves forward when both wheels turn forward, and spins in place when one wheel drives forward and one drives backward.





Differential Drive Robots

- A differential drive robot can only move forward/backward along its longitudinal axis and rotate only around its vertical axis
  - The robot cannot move sideways or vertically
- Thus we only need to set the linear **x** component and the angular **z** component in the Twist message
- An omni-directional robot would also use the linear y component while a flying or underwater robot would use all six components

A Move Turtle Node

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

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

- In Eclipse add a new source file to the package called Move_Turtle.cpp
- Add the following code

MoveTurtle.cpp

```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();
    }
}
```

Launch File

- Add move_turtle.launch to your package:

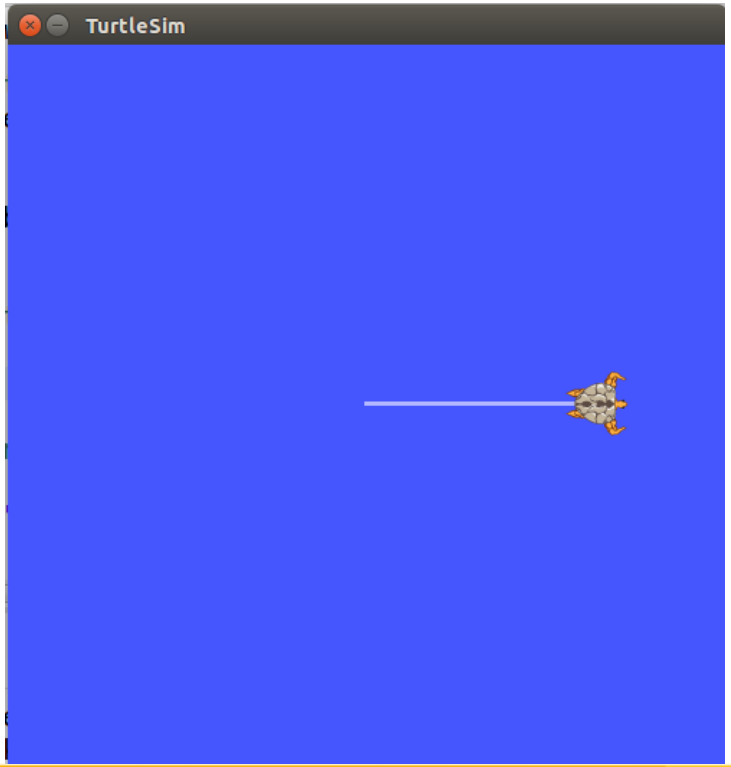
```
<launch>
  <node name="turtlesim_node" pkg="turtlesim" type="turtlesim_node" />
  <node name="move_turtle" pkg="my_turtle" type="move_turtle"
output="screen" />
</launch>
```

- Run the launch file:

```
$ roslaunch my_turtle move_turtle.launch
```

Move Turtle Demo

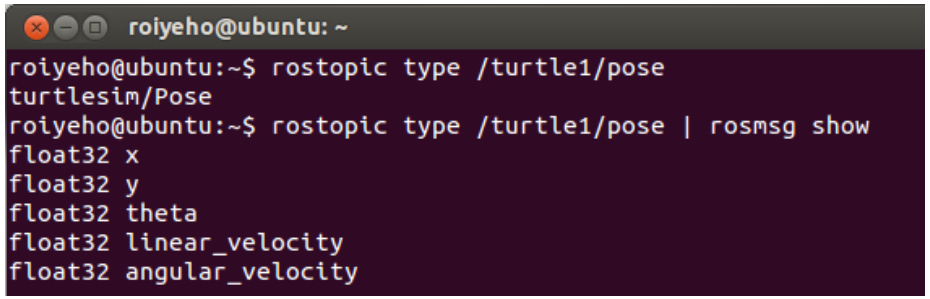
- You should see the turtle in the simulator constantly moving forward until it bumps into the wall



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 our code

MoveTurtle.cpp (1)

```cpp
#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);
```

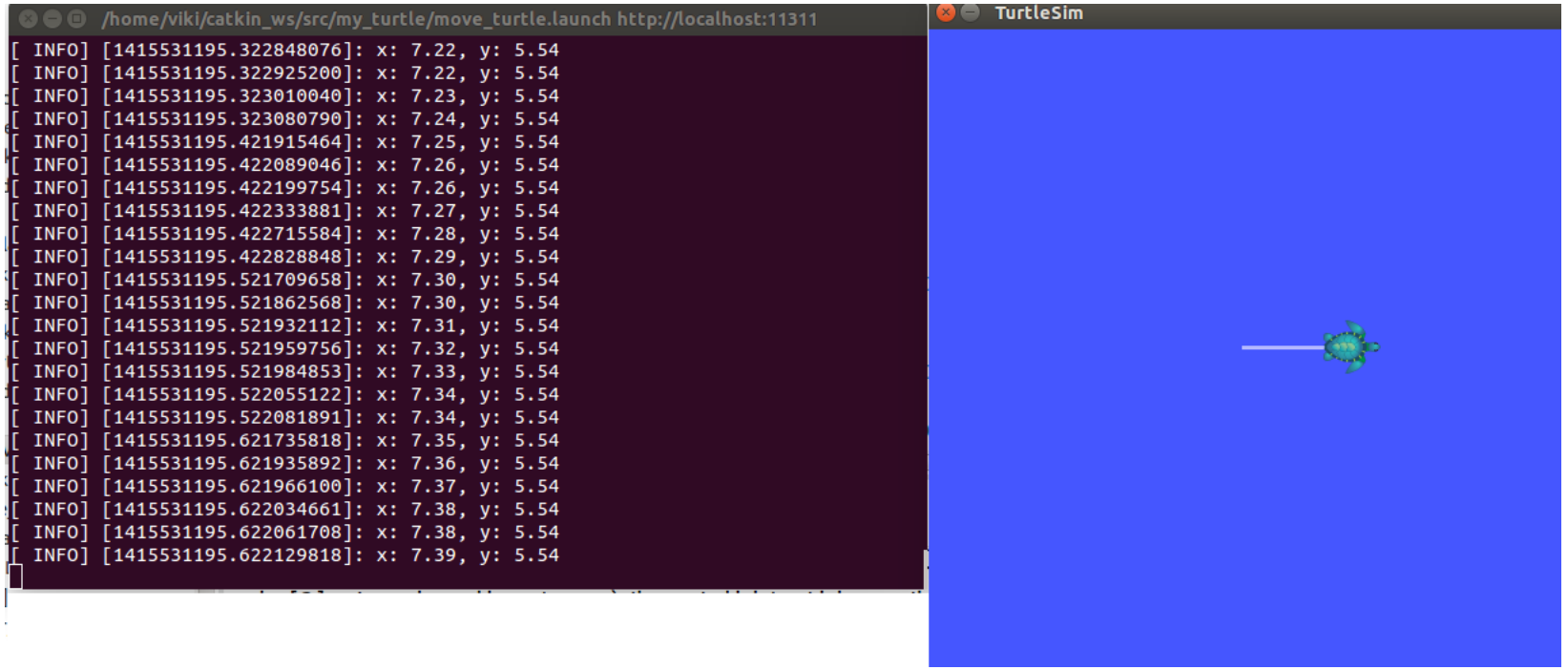
MoveTurtle.cpp (2)

```
    // 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



Passing Arguments To Nodes

- In the launch file you can use the **args** attribute to pass command-line arguments to node
- In our case, we will pass the name of the turtle as an argument to move_turtle

```
<launch>
  <node name="turtlesim_node" pkg="turtlesim" type="turtlesim_node" />
  <node name="move_turtle" pkg="my_turtle" type="move_turtle"
args="turtle1" output="screen"/>
</launch>
```

MoveTurtle.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();
    }
}
```

Creating Custom Messages

- ROS offers a rich set of built-in message types
- The std_msgs pa

Built-in types:

Primitive Type	Serialization	C++	Python
bool (1)	unsigned 8-bit int	`uint8_t` (2)	`bool`
int8	signed 8-bit int	`int8_t`	`int`
uint8	unsigned 8-bit int	`uint8_t`	`int` (3)
int16	signed 16-bit int	`int16_t`	`int`
uint16	unsigned 16-bit int	`uint16_t`	`int`
int32	signed 32-bit int	`int32_t`	`int`
uint32	unsigned 32-bit int	`uint32_t`	`int`
int64	signed 64-bit int	`int64_t`	`long`
uint64	unsigned 64-bit int	`uint64_t`	`long`
float32	32-bit IEEE float	`float`	`float`
float64	64-bit IEEE float	`double`	`float`
string	ascii string (4)	`std::string`	`string`
time	secs/nsecs signed 32-bit ints	ros::Time	rospy.Time
duration	secs/nsecs signed 32-bit ints	ros::Duration	rospy.Duration

Creating Custom Messages

- These primitive types are used to build all of the messages used in ROS
- For example, (most) laser range-finder sensors publish sensor_msgs/LaserScan messages

```
# Single scan from a planar laser range-finder

Header header
# stamp: The acquisition time of the first ray in the scan.
# frame_id: The laser is assumed to spin around the positive Z axis
# (counterclockwise, if Z is up) with the zero angle forward along the x axis

float32 angle_min # start angle of the scan [rad]
float32 angle_max # end angle of the scan [rad]
float32 angle_increment # angular distance between measurements [rad]

float32 time_increment # time between measurements [seconds] - if your scanner
# is moving, this will be used in interpolating position of 3d points
float32 scan_time # time between scans [seconds]

float32 range_min # minimum range value [m]
float32 range_max # maximum range value [m]

float32[] ranges # range data [m] (Note: values < range_min or > range_max should be
discarded)
float32[] intensities # intensity data [device-specific units]. If your
# device does not provide intensities, please leave the array empty.
```

Creating Custom Messages

- Using standardized message types for laser scans and location estimates enables nodes can be written that provide navigation and mapping (among many other things) for a wide variety of robots
- However, there are times when the built-in message types are not enough, and we have to define our own messages

msg Files

- ROS messages are defined by special message-definition files in the *msg* directory of a package.
- These files are then compiled into language-specific implementations that can be used in your code
- Each line in the file specifies a type and a field name

Message Field Types

Field types can be:

- a built-in type, such as "float32 pan" or "string name"
- names of Message descriptions defined on their own, such as "geometry_msgs/PoseStamped"
- fixed- or variable-length arrays (lists) of the above, such as "float32[] ranges" or "Point32[10] points“
- the special Header type, which maps to std_msgs/Header

Creating Custom Messages

- As an example we will create a new ROS message type that each robot will publish when it is ready to perform some task
- Message type will be called **RobotStatus**
- The structure of the message will be:

```
Header header
int32 robot_id
bool is_ready
```

- The header contains a timestamp and coordinate frame information that are commonly used in ROS

Message Header

```
#Standard metadata for higher-level flow data types
#sequence ID: consecutively increasing ID
uint32 seq
#Two-integer timestamp that is expressed as:
# * stamp.secs: seconds (stamp_secs) since epoch
# * stamp.nsecs: nanoseconds since stamp_secs
# time-handling sugar is provided by the client library
time stamp
#Frame this data is associated with
string frame_id
```

- stamp specifies the publishing time
- frame_id specifies the point of reference for data contained in that message

Creating a Message Type

- First create a new package called custom_messages

```
$ cd ~/catkin_ws/src
$ catkin_create_pkg custom_messages std_msgs rospy roscpp
```

- Now create a subdirectory msg and the file RobotStatus.msg within it

```
$ cd ~/catkin_ws/src/custom_messages
$ mkdir msg
$ gedit msg/RobotStatus.msg
```

Creating a Message Type

- Add the following lines to RobotStatus.msg:

```
Header header
int32 robot_id
bool is_ready
```

- Now we need to make sure that the msg files are turned into source code for C++, Python, and other languages

Creating a Message Type

- Open package.xml, and add the following two lines to it

```
<build_depend>roscpp</build_depend>
<build_depend>rospy</build_depend>
<build_depend>std_msgs</build_depend>
<build_depend>message_generation</build_depend>
<run_depend>roscpp</run_depend>
<run_depend>rospy</run_depend>
<run_depend>std_msgs</run_depend>
<run_depend>message_runtime</run_depend>
```

- Note that at build time, we need "message_generation", while at runtime, we need "message_runtime"

Creating a Message Type

- In CMakeLists.txt add the message_generation dependency to the find package call so that you can generate messages:

```
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
  message_generation
)
```

- Also make sure you export the message runtime dependency:

```
catkin_package(
#  INCLUDE_DIRS include
#  LIBRARIES multi_sync
    CATKIN_DEPENDS roscpp rospy std_msgs message_runtime
#  DEPENDS system_lib
)
```

Creating a Message Type

- Find the following block of code:

```
## Generate messages in the 'msg' folder
# add_message_files(
#   FILES
#   Message1.msg
#   Message2.msg
# )
```

- Uncomment it by removing the # symbols and then replace the stand in Message*.msg files with your .msg file, such that it looks like this:

```
add_message_files(
  FILES
  RobotStatus.msg
)
```

Creating a Message Type

- Now we must ensure the generate_messages() function is called
- Uncomment these lines:

```
# generate_messages(
#   DEPENDENCIES
#   std_msgs
# )
```

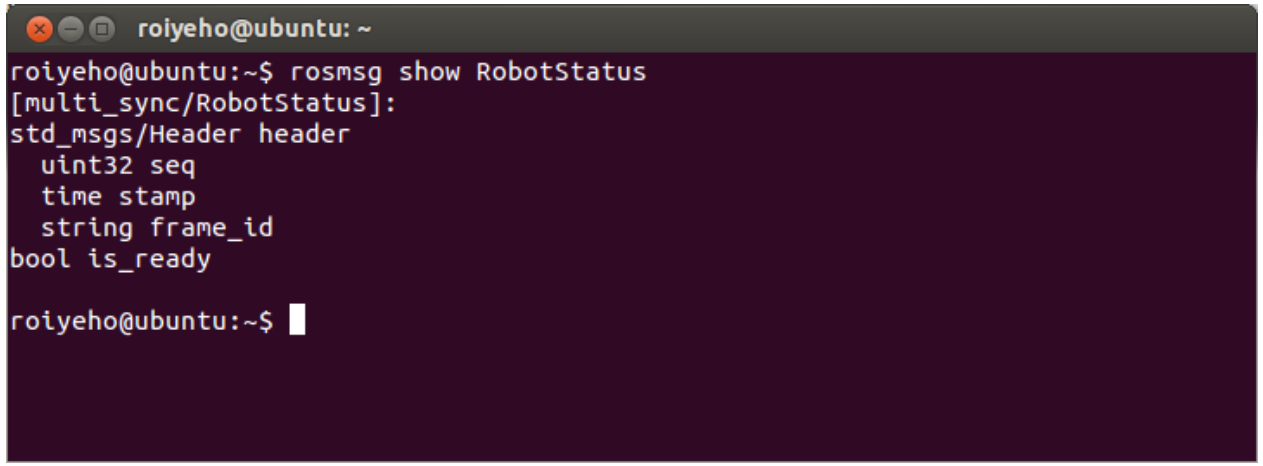
- So it looks like:

```
generate_messages(
  DEPENDENCIES
  std_msgs
)
```

Using rosmsg

- That's all you need to do to create a msg
- Let's make sure that ROS can see it using the rosmsg show command:

```
$ rosmsg show [message type]
```

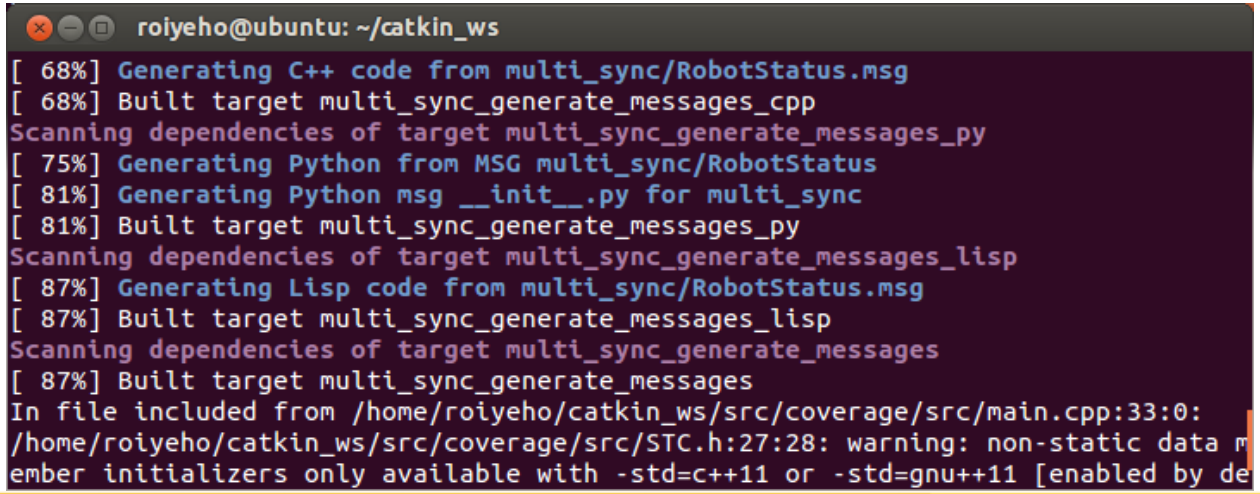


Building the Message Files

- Now we need to make our package:

```
$ cd ~/catkin_ws
$ catkin_make
```

- During the build, source files will be created from the .msg file:



Building the Message Files

- Any .msg file in the msg directory will generate code for use in all supported languages.
- The C++ message header file will be generated in ~/catkin_ws/devel/include/custom_messages/

RobotStatus.h

```
#include <std_msgs/Header.h>
namespace multi_sync
{
template <class ContainerAllocator>
struct RobotStatus_
{
  typedef RobotStatus_<ContainerAllocator> Type;
  RobotStatus_()
    : header()
    , robot_id(0)
    , is_ready(false)  {
    }
   typedef  ::std_msgs::Header_<ContainerAllocator>  _header_type;
  _header_type header;

   typedef uint32_t _robot_id_type;
  _robot_id_type robot_id;

   typedef uint8_t _is_ready_type;
  _is_ready_type is_ready;

  typedef boost::shared_ptr< ::multi_sync::RobotStatus_<ContainerAllocator> > Ptr;
  typedef boost::shared_ptr< ::multi_sync::RobotStatus_<ContainerAllocator> const> ConstPtr;
  boost::shared_ptr<std::map<std::string, std::string> > __connection_header;

}; // struct RobotStatus_
typedef ::multi_sync::RobotStatus_<std::allocator<void> > RobotStatus;
```

Importing Messages

- You can now import the RobotStatus header file by adding the following line:

```
#include "custom_messages/RobotStatus.h"
```

- Note that messages are put into a namespace that matches the name of the package

Ex. 3

- Write a program that moves the turtle 1m forward from its current position, rotates it 45 degrees and then causes it to stop
- Print the turtle's initial and final locations