

# CSci 5551 Introduction to Intelligent Robotics Systems

## ROS Programming

College of Science & Engineering
University of Minnesota

# Sense-Think-Act

- Reminder: Sense the world, reason about it, then act upon it.
  - We do this every instant of every day, and it's easy.
- The sense-think-act cycle is the core of computational robotics.
- Not the mechanical aspect of robotics, but the tools, techniques and algorithms which make robots smart.

# Tools

- MatLab/Octave
  - Helps us prototype our algorithms
- ROS
  - Provides libraries to implement algorithms on real robots
- OpenCV
  - Helps us process vision data
- SciKit-Learn and TensorFlow
  - Helps us learn from (a large volume of) data
- And more

# Our job...

- ...is to use such tools to write code, so that
  - Robots can access sensor data
  - Interpret them and visualize them if necessary
  - Interact with people and the environment they are at
  - Affect changes in the environment

# Focus on ROS

- Why ROS ?
  - Benefits
    - Runs on a number of robot platforms
      - Check out robots.ros.org!
    - Easily adaptable to new ones
    - Makes code portable across robots
    - Makes code reusable – stops “reinvention of the wheel”
    - Large user base – encourages code sharing, dissemination of research results, improves verifiability
    - Supported by hardware vendors and the Open Source Robotics Foundation (OSRF)

# Why not X?

- As in, why not use something else other than ROS?
  - Proprietary solutions
  - If open-source, use is not as widespread
  - Harder to share your work, or use someone else's work
  - Not as well-supported or well-documented
  - Not possible to disseminate findings or verify other's results

- Note: This was the world before ROS, and it wasn't great.

# Advice and A Word of Warning

- You should really go through the tutorials on the wiki:
  - http://wiki.ros.org/ROS/Tutorials
  - We will be going through some of that information today.
- The wiki is your friend!
  - The wiki is a great source of information, especially about basics.
  - You'll see references to wiki pages throughout these slides.
- The wiki can also be a bit misleading.
  - Information is often out of date, occasionally just wrong.
  - That's the nature of open-source projects.

# What's ROS, really?

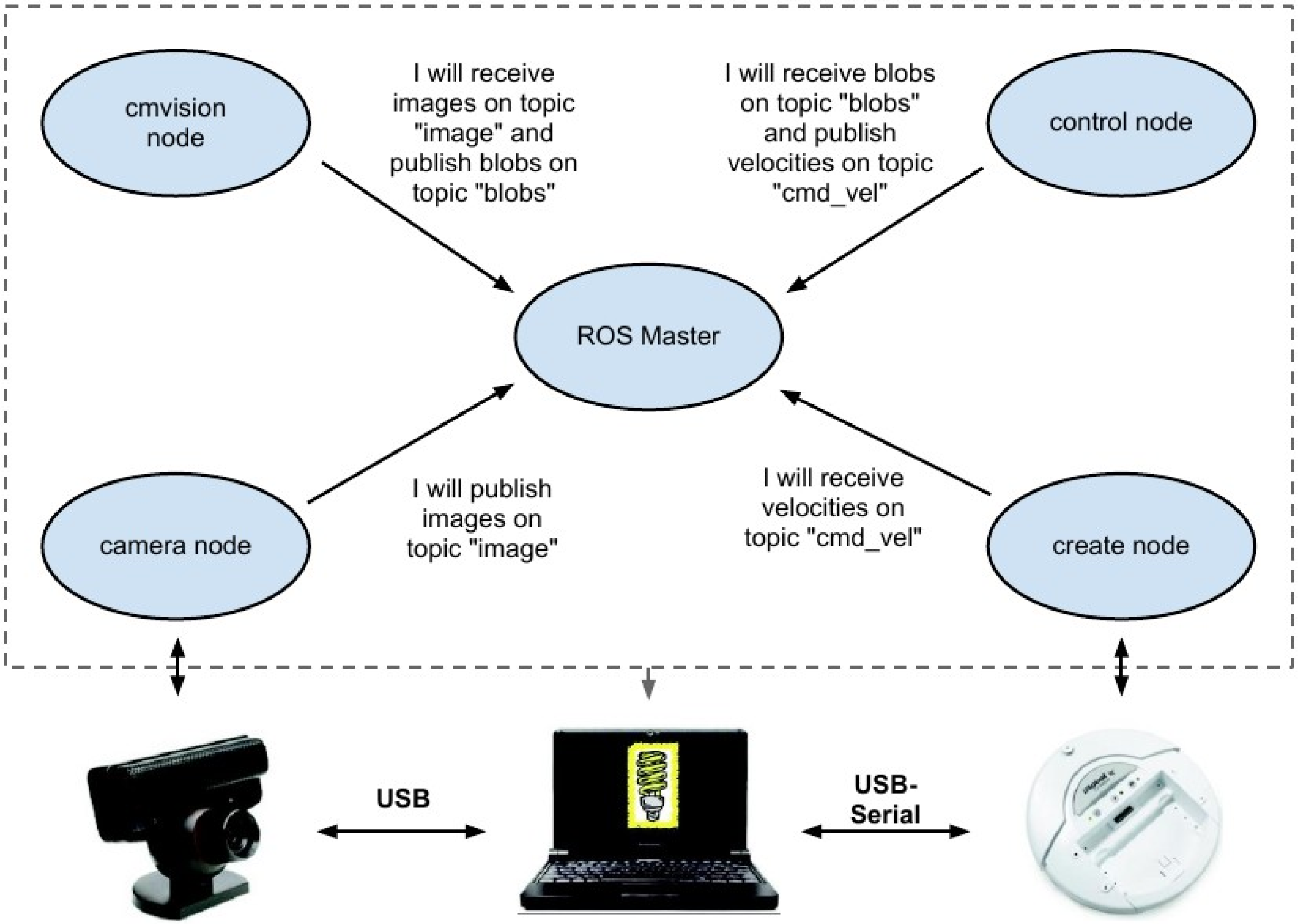
- It's not a “true” OS
  - a peer-to-peer robot middleware package
  - sits between OS and robot hardware
- allows for easier hardware abstraction and code reuse
- all major functionality is broken up into a number of chunks that communicate with each other using messages

# ROS Nodes

- Each chunk is called a **node** and is typically run as a <u>separate process</u>

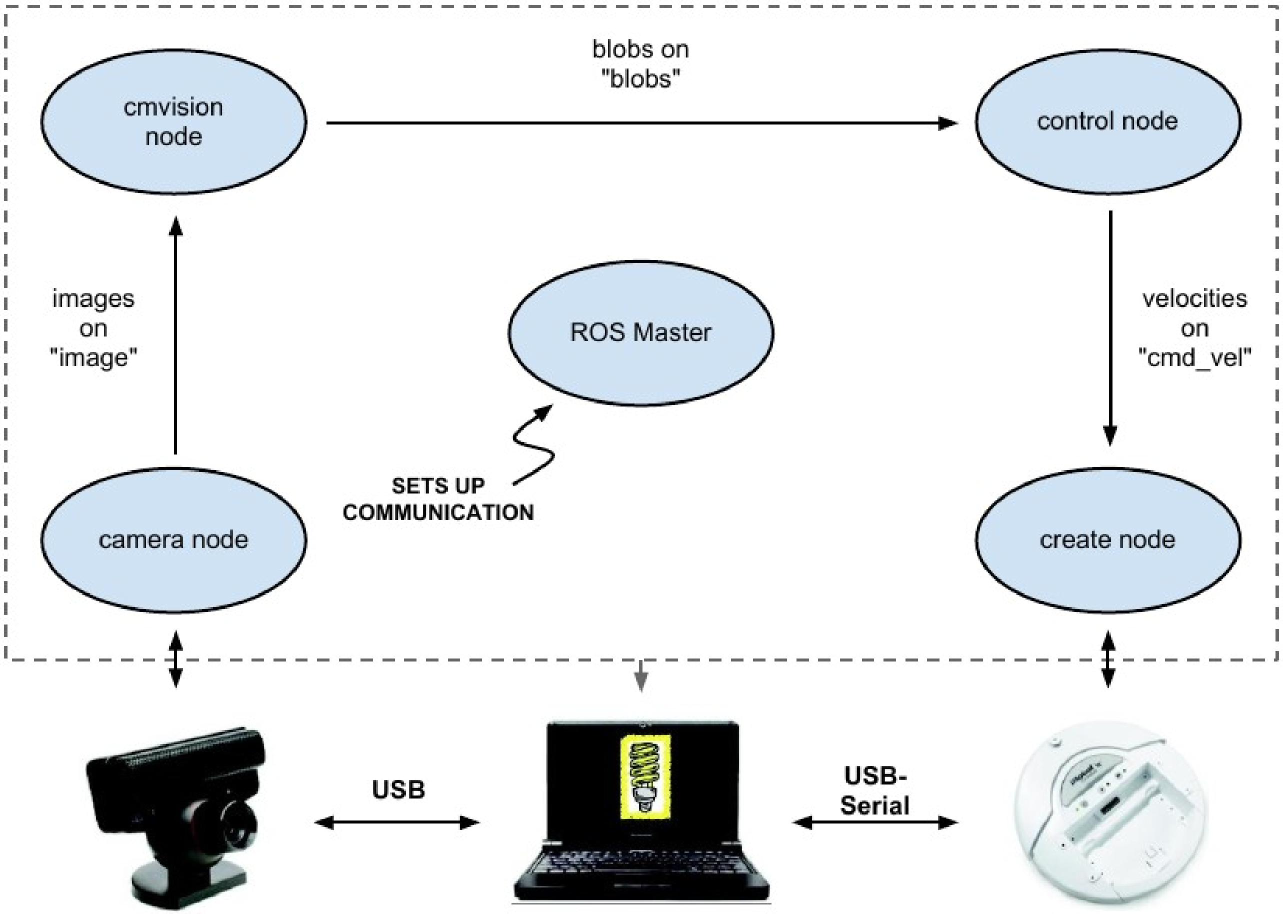
- Matchmaking between nodes is done by the **ROS Master**
  - Communication, however, is done on a peer-to-peer basis by the nodes themselves.

# ROS Node Communications




College of Science & Engineering
University of Minnesota

# ROS Node Communications



# Nodes

- A node is **a process** that performs some computation.
- Usual approach:
  - divide the entire software functionality into different modules
  - each one is run over **a single node or multiple nodes**
- Nodes are combined together into a graph and communicate with one another using streaming topics, RPC services, and the Parameter Server
- Nodes are meant to operate at a fine-grained scale
  - a robot control system will usually comprise many nodes

http://wiki.ros.org/Nodes

# Topics

- Topics are named buses over which nodes exchange messages
- Topics have anonymous publish/subscribe semantics
  - A node does not care which node published the data it receives or which one subscribes to the data it publishes
- There can be multiple publishers and subscribers to a topic
- Each topic is strongly typed by the ROS message it transports
  - Transport is done using TCP or UDP

http://wiki.ros.org/Topics

# Messages

- Nodes communicate with each other by publishing messages to topics
- A message is a simple data structure, comprising typed fields
- Some basic types:
  - std_msgs/Bool
  - std_msgs/Int32
  - std_msgs/String
  - std_msgs/Empty (huh?)
- Messages may also contain a special field called header which gives a timestamp and frame of reference

College of Science & Engineering
University of Minnesota

http://wiki.ros.org/Messages

# ROS Structure

- So, when working with ROS, think about your system as a network of interconnected nodes, each doing one part of the work.

- This structure allows for easy re-use and refactoring, because you can change one node independent of the others.

# ROS Lectures

- Writing a ROS program (C++)
- ROS structure
- ROS build system
- ROS visualization
- ROS command-line tools

# Demo ROS Code

- First we're going to run the code, then look at it in more detail.
- There are a lot of small details about the build system, package structure, etc.
- We'll get to some of that on Wednesday.

# Code Anatomy – talker.cxx

```cpp
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <sstream>

int main(int argc, char **argv) {
    ros::init(argc, argv, "talker");
    ros::NodeHandle n;
    ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
    ros::Rate loop_rate(1);
    int count = 0;
    while (ros::ok()) {
        std_msgs::String msg;
        std::stringstream ss;
        ss << "hello world " << count;
        msg.data = ss.str();
        ROS_INFO("%s", msg.data.c_str());
        chatter_pub.publish(msg);
        ros::spinOnce();
        loop_rate.sleep();
        ++count;
    }
    return 0;
}

```

# Breakdown

- **talker.cxx**

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <sstream>
```

- ros/ros.h is a convenience header that includes most of the pieces necessary to run a ROS System
- std_msgs/String.h is the message type that we will need to pass in this example
  - include a different header if you want to use a different message type
- sstream is responsible for some string manipulations in C++

# Breakdown

```
ros::init(argc, argv, "talker");
ros::NodeHandle n;
```

- **ros::init** is responsible for collecting ROS specific information from arguments passed at the command line
  - It also takes in the name of our node
  - Node names need to be unique in a running system
- The creation of a **ros::NodeHandle** object does a lot of work
  - It initializes the node to allow communication with other ROS nodes and the master in the ROS infrastructure
  - Allows you to interact with the node associated with this process

# Breakdown

```
ros::Publisher chatter_pub =
n.advertise<std_msgs::String>("chatter
", 1000);
ros::Rate loop_rate(1);
```

- **NodeHandle::advertise** is responsible for making the XML/RPC call to the ROS Master advertising std_msgs::String on the topic named "chatter"
- **loop_rate** is used to maintain the frequency of publishing at 1 Hz (i.e., 1 message per second)

# Breakdown

```
int count = 0;
while (ros::ok()) {
```

- **count** is used to keep track of the number of messages transmitted. Its value is attached to the string message that is published
- **ros::ok()** ensures that everything is still alright in the ROS framework. If something is amiss, then it will return false effectively terminating the program.
  - Examples of situations where it will return false:
    - You Ctrl+c the program (SIGINT)
    - You open up another node with the same name.
    - You call ros::shutdown() somewhere in your code

# Breakdown

```
std_msgs::String msg;
std::stringstream ss;
ss << "hello world " << count;
msg.data = ss.str();
```

- These 4 lines do some string manipulation to put the count inside the String message
  - The reason we do it this way is that C++ does not have a good equivalent to the toString() function
- msg.data is a std::string
- Aside: look into **boost::lexical_cast()** in place of the toString() function. lexical_cast() pretty much does the thing above for you (Look up this function if you are interested)

COLLEGE OF Science & Engineering
UNIVERSITY OF MINNESOTA

# Breakdown

```
ROS_INFO("%s", msg.data.c_str());
chatter_pub.publish(msg);
```

- **ROS_INFO** is a macro that publishes a information message in the ROS ecosystem. By default **ROS_INFO** messages are also published to the screen.
  - There are debug tools in ROS that can read these messages
  - You can change what level of messages you want to be have published
- **ros::Publisher::publish()** sends the message to all subscribers

College of Science & Engineering
University of Minnesota

# Breakdown

```
ros::spinOnce();
loop_rate.sleep();
++count;
```

- **ros::spinOnce()** is analogous to the main function of the ROS framework.
  - Whenever you are subscribed to one or many topics, the callbacks for receiving messages on those topics are not called immediately.
  - Instead they are placed in a queue which is processed when you call **ros::spinOnce()**
  - What would happen if we remove the spinOnce() call?
- **ros::Rate::sleep()** helps maintain a particular publishing frequency
- **count** is incremented to keep track of messages

# Code Anatomy – listener.cxx

```
#include "ros/ros.h"
#include "std_msgs/String.h"

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

int main(int argc, char **argv) {
    ros::init(argc, argv, "listener");
    ros::NodeHandle n;
    ros::Subscriber sub =
        n.subscribe<std_msgs::String>("chatter", 1000, chatterCallback);
    ros::spin();
    return 0;
}

```

# Breakdown – the listener

```
int main(int argc, char **argv) {

   ros::init(argc, argv, "listener");
   ros::NodeHandle n;
   ros::Subscriber sub = n.subscribe<std_msgs::String>("chatter", 1000,
chatterCallback);
   ros::spin();
   return 0;
}
```

- **ros::NodeHandle::subscribe** makes an XML/RPC call to the ROS master
  - It subscribes to the topic chatter
  - 1000 is the queue size. In case we are unable to process messages fast enough. This is only useful in case of irregular processing times of messages. (Why?)
  - The third argument is the callback function to call whenever we receive a message
- **ros::spin()** a convenience function that loops around **ros::spinOnce()** while checking ros::ok()

College of Science & Engineering
University of Minnesota

```
#include "ros/ros.h"
#include "std_msgs/String.h"

void chatterCallback(const
   std_msgs::String::ConstPtr msg) {
   ROS_INFO("I heard: [%s]", msg>data.c_str());
}
```

- Same headers as before
- **chatterCallback()** is a function we have defined that gets called whenever we receive a message on the subscribed topic
- It has a well-typed argument.

# Python Variant – talker.py

```
#!/usr/bin/env python
# license removed for brevity
import rospy
from std_msgs.msg import String

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

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

# Python Variant – listener.py

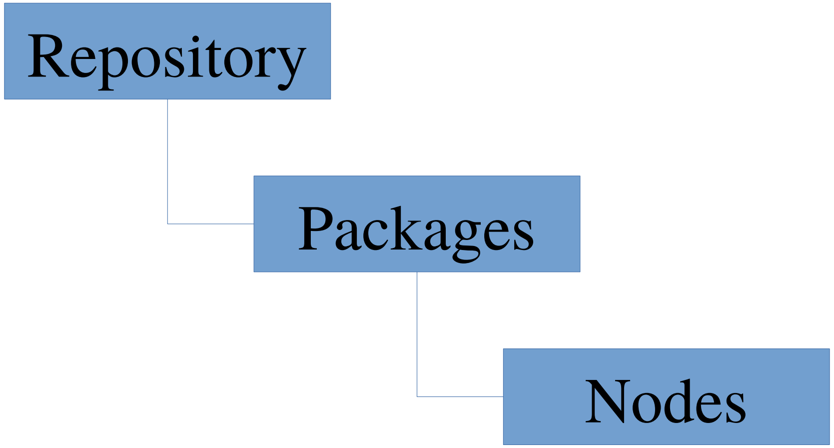
```python
#!/usr/bin/env python
import rospy
from std_msgs.msg import String

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

def listener():

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

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

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

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

# ROS CLI Tools

- rqt_graph
  - Shows publishers, subscribers, messages and connection in a graph form
  - Useful for debugging, visualization etc
- rqt_console
  - Console to view ROS messages, such as INFO, DEBUG, WARN, ERROR etc.
- roswtf(!)
  - Sanity checker for your ROS setup

# ROS Code Structure

- Repository: Contains all the code from a particular development group.
- Packages: Separate modules that provide different services
- Nodes: Executables that exist in each model (You have seen this already)



# Anatomy of a Package

- CMake-based system
  - CMakeLists.txt drives build system
- Package configuration is stored in XML format
  - package.xml is the store/configurator
- Packages reside in catkin workspaces
  - More detail next class

# Catkin

- Catkin is the ROS build system
  - Supersedes the old ROSMake system
    - ROSMake still works but heavily deprecated
- Every ROS 'project' needs to reside in a catkin workspace
  - There can be multiple catkin workspaces
- Must initialize workspace, and use catkin build tools (such as catkin_make) to build and run nodes

# Basic Concepts

- A node is…
  - A process that performs some computation, operating at a fine-grained scale.
- A topic is…
  - A named bus over which nodes can exchange messages, and which can be anonymously published/subscribed to by any number of nodes.
- A message is…
  - The data structure used to pass data through Topics.

# A Quick Review

- Hopefully you went through tutorials 1.1-1.6.

- As we’ve seen, ROS program structure is:
  - A group of nodes, each executing a task in a unique process, communicating with each other using messages sent across strongly typed buses called topics, to which any node can anonymously publish or subscribe, with the peer-to-peer communication being set up by the ROS master.
  - This is what we spent Monday learning.

# Code Anatomy – talker.cxx

```cpp
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <sstream>

int main(int argc, char **argv) {
    ros::init(argc, argv, "talker");
    ros::NodeHandle n;
    ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
    ros::Rate loop_rate(1);
    int count = 0;
    while (ros::ok()) {
        std_msgs::String msg;
        std::stringstream ss;
        ss << "hello world " << count;
        msg.data = ss.str();
        ROS_INFO("%s", msg.data.c_str());
        chatter_pub.publish(msg);
        ros::spinOnce();
        loop_rate.sleep();
        ++count;
    }
    return 0;
}

```

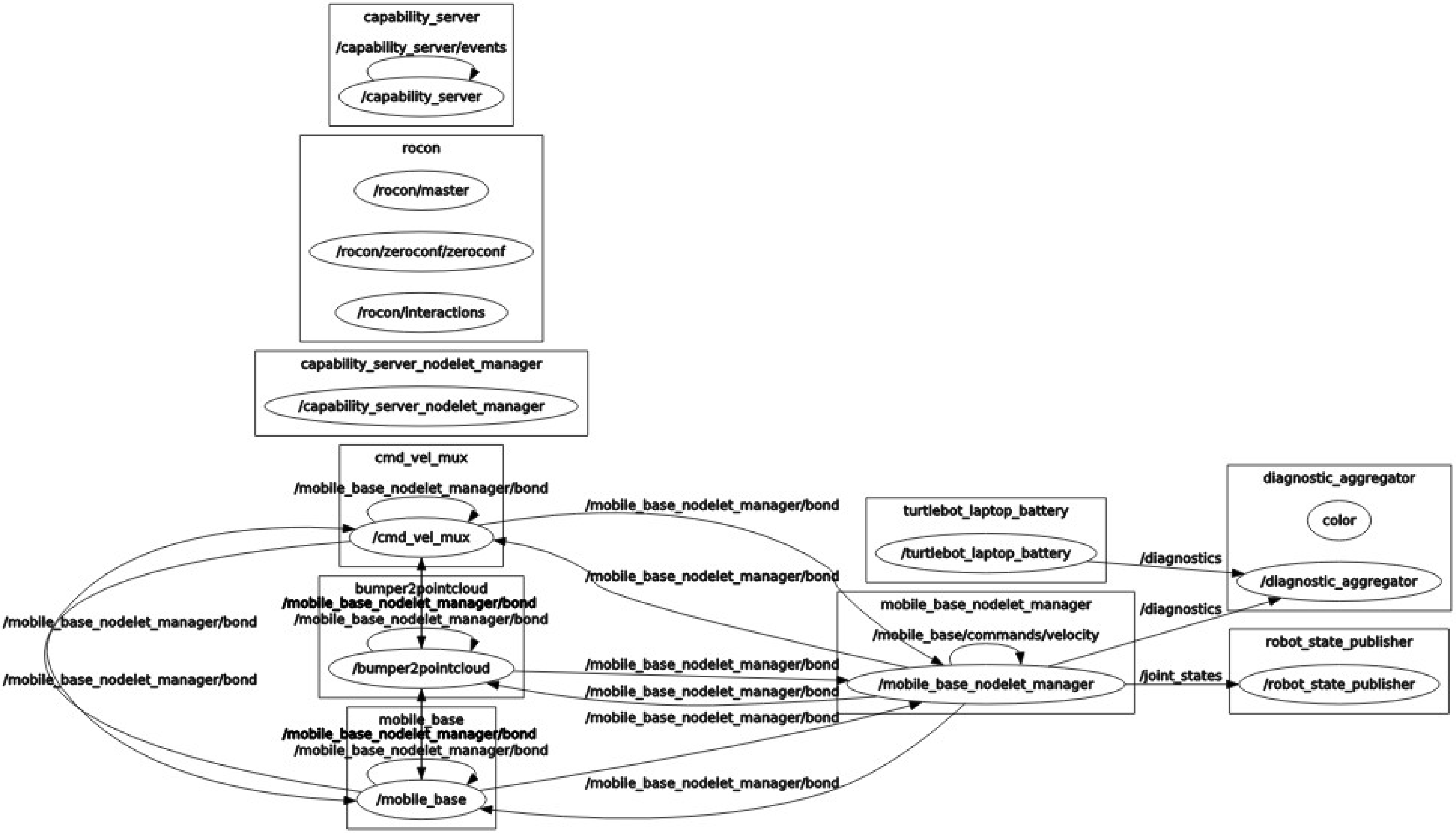
# Code Anatomy – listener.cxx

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

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

int main(int argc, char **argv) {
    ros::init(argc, argv, "listener");
    ros::NodeHandle n;
    ros::Subscriber sub =
        n.subscribe<std_msgs::String>("chatter", 1000, chatterCallback);
    ros::spin();
    return 0;
}
```

# Turtlesim Demo

- You can find this in tutorials 1.5 and 1.6

# A Real ROS Graph

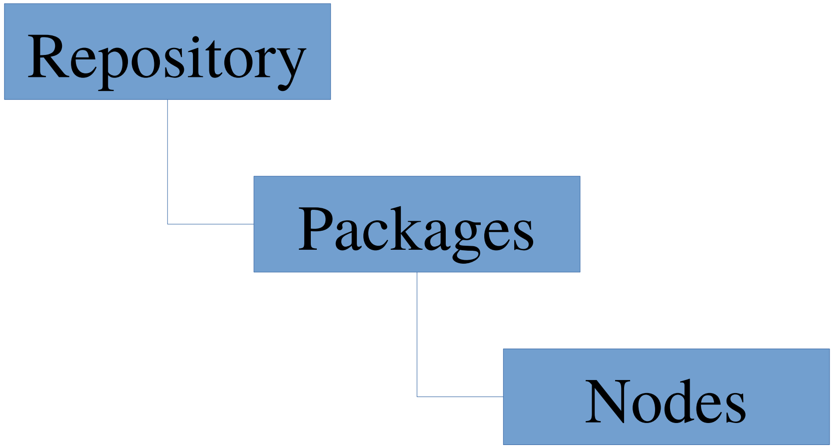


# So far

- Seen:
  - Writing a ROS program (C++)
  - ROS command-line tools
  - ROS structure
- Next
  - ROS build system
    - Including catkin and package manifests.
  - More CLI tools
  - ROS visualization
  - ROS insights

# ROS Code Structure

- Repository: Contains all the code from a particular development group.
- Packages: Separate modules that provide different services
- Nodes: Executables that exist in each model (You have seen this already)



# Anatomy of a Package

- CMake-based system
  - CMakeLists.txt drives build system
- Package configuration is stored in XML format
  - package.xml is the store/configurator
- Packages reside in catkin workspaces

# How do you cook your own package?

- By using `catkin_create_pkg`
  - Remember, a 'package' is not necessarily one node
- Good practice: have a common 'root' of all ROS catkin workspaces
  - Add it to the ROS_WORKSPACE environment variable
    - So a simple `roscd` will take you to that directory
- In your ROS common root, do this:

```
catkin_create_pkg sample_package roscpp std_msgs
```

- catkin will create package.xml and CMakeLists.txt files
  - You need to modify them to do anything useful with this package

College of Science & Engineering
University of Minnesota

# CMakeLists.txt

- You must create at least one library target or one executable target
  - Otherwise your code is useless
- You need to know some basic CMake syntax to make it work
- CMake is a useful tool to have under your belt
  - Used by many software projects
  - KDE, OpenCV, libpng, llvm, FlightGear just to name a few
  - A good tutorial is available at: https://cmake.org/cmake-tutorial

# CMake mini tutorial

- CMake is a 'build process manager'
  - “A maker of makefiles”
  - Not just unix makefiles, but Eclipse projects, VS solutions, Xcode project configurations etc
- The build script generation is controlled by the CMakeLists.txt configuration file
- CMake also creates a configuration cache that can be tailored to provide particular build features
  - e.g., using ccmake

# ROS CMakeLists.txt

- See example

# Package configuration

- See package.xml

# Package Contents

- Obviously packages contain nodes, but they also can contain other things.
- Messages, services, launch files, etc.

# ROS Messages and Services

- Messages == data
- How do you write custom messages?
  - Or generate service (not data) messages?
    - What are those??
- writing a simple service server and client

# ROS Messages

- Till now, we have only used predefined ROS messages
- Also possible to create messages as necessary
- Take a look at the tutorial on creating messages and services on the ROS wiki: http://www.ros.org/wiki/ROS/Tutorials/CreatingMsgAndSrv
  - This is tutorial 1.10

COLLEGE OF Science & Engineering
UNIVERSITY OF MINNESOTA

# Sample msg File

- Very simple, just list the types of the fields of the message along with their names.
- Like so:
  - string first_name
  - string last_name
  - uint8 age
  - uint32 score
- Optionally add a header.

# Messages

- Message files are stored in the msg directory in the package
- The C++ and Python interfaces to these messages are generated automatedly
  - To enable generation, open up CMakeLists.txt and uncomment the line:

    #rosbuild_genmsg()
- Generated python files are placed in src/<packagename>/msg inside the package
- Generated cpp files are placed in msg_gen/cpp/include inside the package
  - This folder is automatically included when you compile your code, as a result the following line works:

    **#include<std_msgs/Float32.h>**

# Services

- When you don’t want to do something in the publish/subscribe model, one option is the request/response option provided by services.

- Services are written in srv files, which are stored in the *srv* directory.

- Again, more info can be found in tutorial 1.10

# Launch Files

- A launch file allows you to specify nodes to start up, so that you don’t have to do it all by hand.
- Very useful, so that you don’t have to run twenty rosrun commands.
- Launch files are stored in the launch directory.

# Parameter Server

- A shared, multi-variate dictionary which nodes use to store and retrieve data at runtime.

- A good place to store static information which isn't going to change much, like configuration parameters.

- Runs within the ROS Master.

# CLI Tools

- roslaunch
  - Used to run launch scripts.
- rosbag
  - Used to record data running through ROS topics.
- roscore
  - Runs the ROS Master, parameter server, and rosout
- roscreate-pkg
  - Creates Cmake files, package manifest, etc.

# Even More CLI Tools

- rosmsg/rossrv
  - Display information about ROS message and service files.
- rospack
  - Displays information about ROS packages.
- rosnode
  - Displays runtime information about ROS nodes.
- rostopic
  - Provides real-time Topic information and enables command-line publishing and subscribing.

# Even More CLI Tools

- rosservice
  - Display info about available services and use them.
- rosparam
  - Allows you to access and store data the ROS Parameter server

# Rosbash (File System CLI Tools)

- A suite of commands providing bash-like functionality within the ROS package system.
  - roscd [package_name] takes you to the directory containing that package.
  - rosls [package_name] lists the files in the package.
  - rosed [package_name] [file_name] edits that file.
    - You can customize which editor is used.
  - roscp [package_name] [file_name] [target_location]
  - rosrun [package_name] [executable]

College of Science & Engineering
University of Minnesota

# CLI GUI Commands

- rqt_graph
  - Displays the current Node graph. (We saw this on Monday)
- rqt_plot
  - Displays plots of messages across a Topic.
- rqt_bag
  - Qt front end to rosbag.

# CLI Tools Summary

- We’ve talked about a lot of CLI tools, but there are still others beyond that.

- Knowing when and how to use each of them, and how to combine their use is part of what makes a good ROS developer.

# rviz

- rviz is the ROS visualization tool.
  - You can use it to display all kinds of information about your robot, its path, and especially its environment.
  - Quick Demo

# Next

- I would recommend that you go over the rest of the tutorials in section 1.
  - Tutorials 1.7 to 1.20
  - Shouldn’t take much more than an hour or two.