EPD1: Introduction to ROS

Luis Merino

Introduction to ROS

- Introduction to ROS
- Basic ROS commands
- Developing in ROS

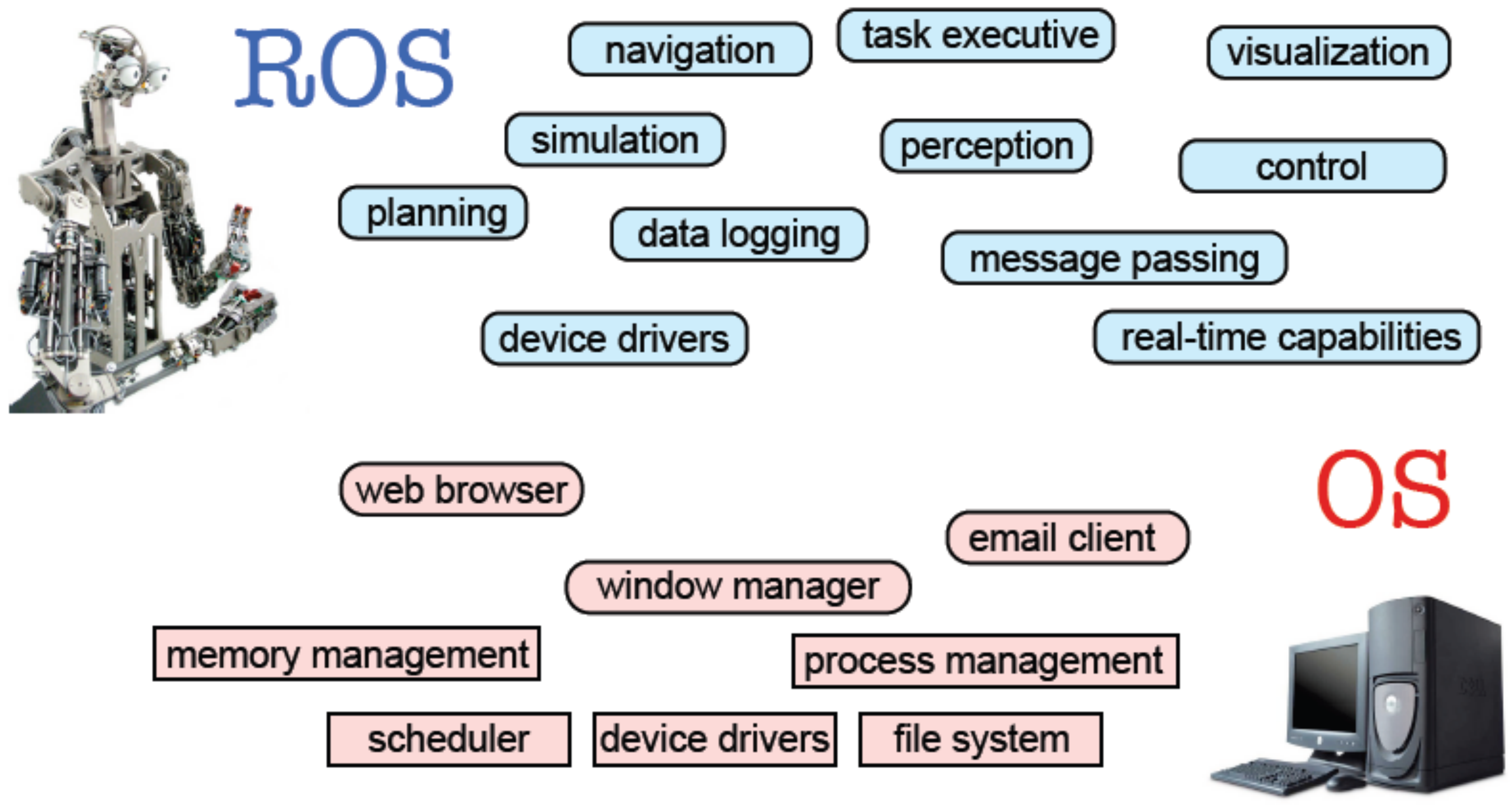
Introduction to ROS

(some slides adapted from Roi Yehoshua, Bar-Ilan University)

What is ROS?

- ROS is an open-source **robot “operating system”**
- The primary goal of ROS is to support code *reuse in robotics research and development*
- ROS was originally developed in 2007 at the Stanford Artificial Intelligence Laboratory
- Development continued primarily at Willow Garage, a robotics research institute/incubator
- Since 2013 it is managed by OSRF (Open Source Robotics Foundation)

ROS Main Features



Taken from Sachin Chitta and Radu Rusu (Willow Garage)

ROS Main Features

- Hardware and network abstraction
- Low-level device control
- Message-passing between processes
- Implementation of commonly-used functionality
- Package management

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 Philosophies

- Modularity & Peer-to-peer
- Language Independent
- Thin
- Free & Open-Source

Modularity & Peer-To-Peer

- ROS is basically a distributed system
- ROS consists of a number of processes
  - potentially on a number of different hosts,
  - connected at runtime in a peer-to-peer topology
- No central server

Language Independent

- Client interfaces:
  - Stable: roscpp, rospy, roslisp
  - Experimental: rosjava, roscs
  - Contributed: rosserial, roshask, ipc-bridge (MATLAB), etc...
- Common message-passing layer
  - Interface Definition Language (IDL)

Thin

- Library-style development
  - all development occurs in standalone libraries with minimal dependencies on ROS
- ROS re-uses code from numerous other open-source projects, such as the navigation system simulators and vision algorithms from OpenCV

Free & Open-Source

- Source code is publicly available
- Contributed tools are under a variety of open-source (& closed-source) licenses
- Promotes code-reuse and community-building

ROS Core Concepts

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

ROS Nodes

- Single-purposed executable programs
  - e.g. sensor driver(s), actuator driver(s), mapper, planner, UI, etc.
- Modular design
  - Individually compiled, executed, and managed
- Nodes are written with the use of a ROS client library
  - roscpp – C++ client library
  - rospy – python client library

ROS Client Libraries

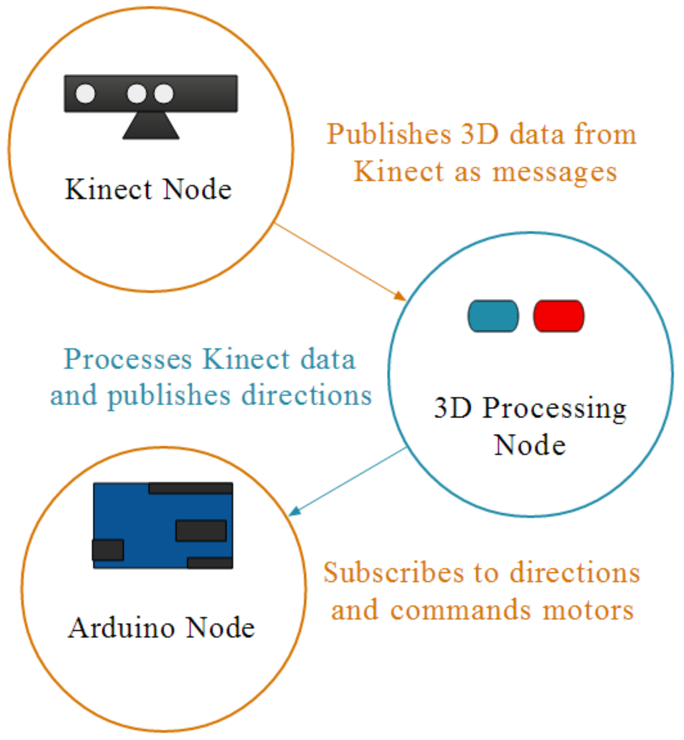
- A collection of code that eases the job of the ROS programmer.
- Libraries that let you write ROS nodes, publish and subscribe to topics, write and call services, and use the Parameter Server.
- Main clients:
  - roscpp = C++ client library
  - rospy = python client library

ROS Master

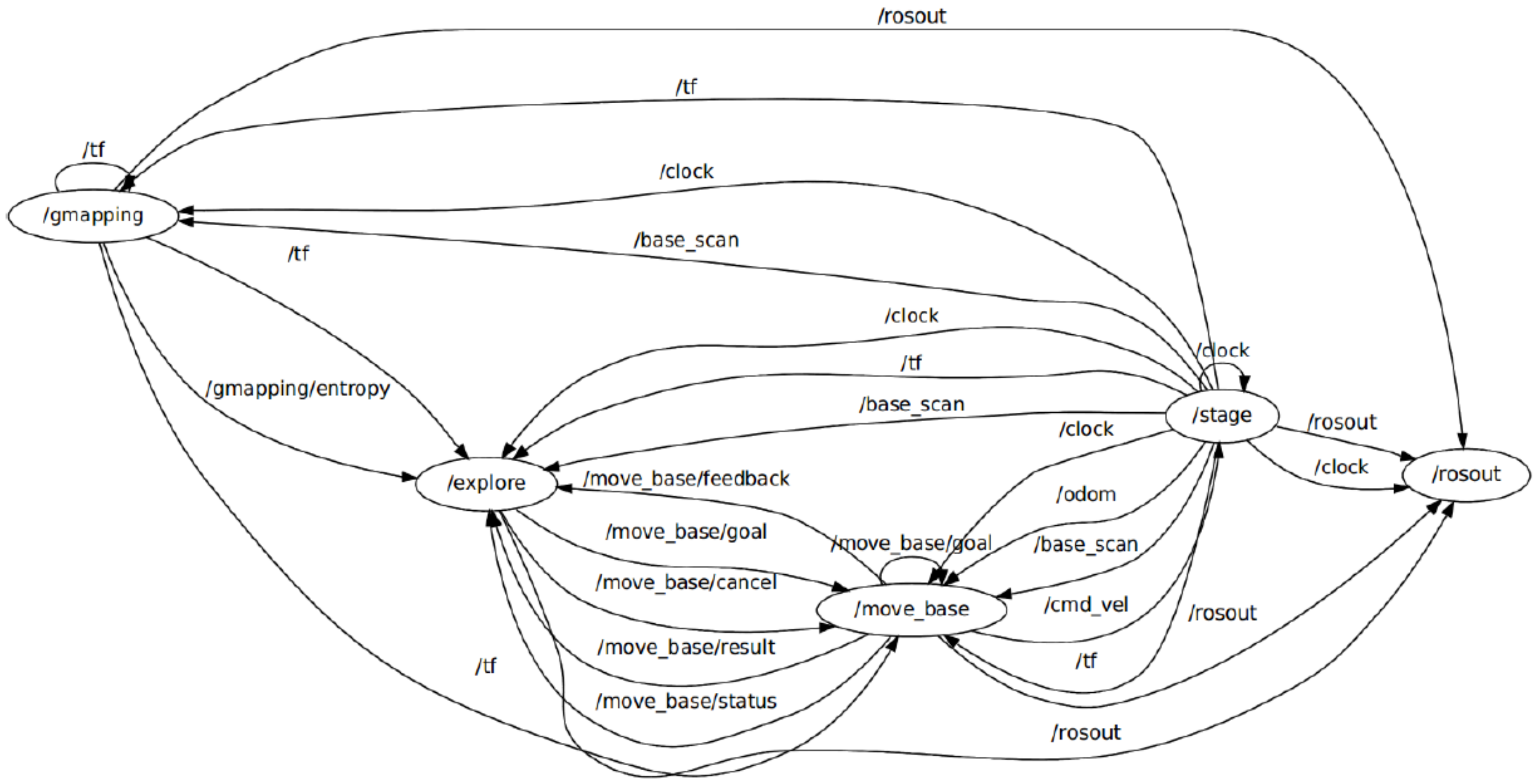
- The role of the master is to enable ROS nodes to locate one another
- Naming & registration services for nodes, topics, services, etc
- Run using the roscore command

ROS Topics

- Nodes communicate with each other by publishing messages to topics
- Publish/Subscribe model: 1-to-N broadcasting


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

More Complex Example



This can be shown by executing the command **rxgraph**

ROS Messages

- Strictly-typed data structures for inter-node communication
- Messages can include:
  - Primitive types (integer, floating point, boolean, etc.)
  - Arrays of primitives
  - Arbitrarily nested structures and arrays (much like C structs)
- For example, geometry_msgs/Twist.msg

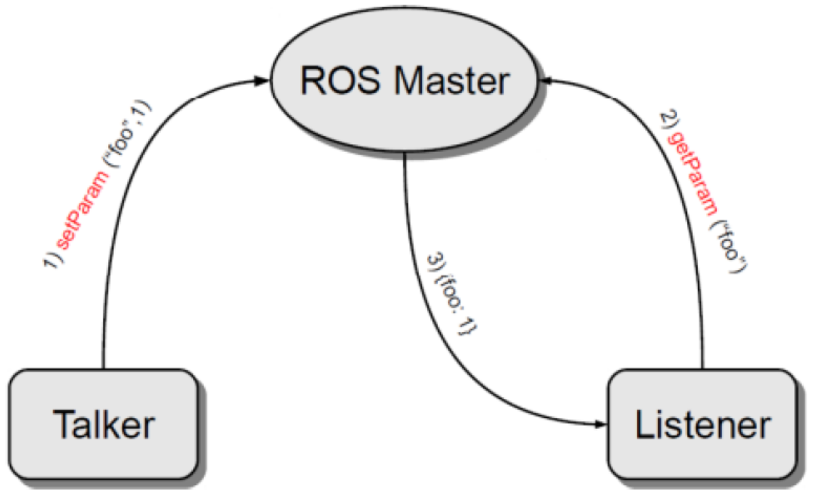
```
Vector3 linear
Vector3 angular
```

ROS Services

- Synchronous inter-node transactions / RPC
- Service/Client model: 1-to-1 request-response
- Service roles:
  - carry out remote computation
  - trigger functionality / behavior
- For example, the explore package provides a service called explore_map which allows an external user to ask for the current map

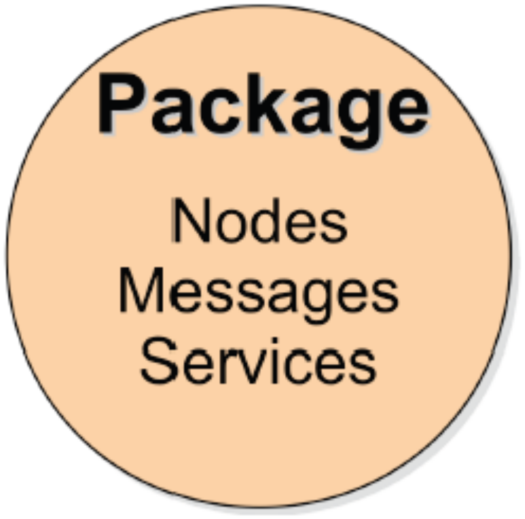
Parameter Server

- 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



ROS Package Repositories

- Collection of packages and stacks
- Many repositories (>50): Stanford, CMU, Leuven, USC, …
- Most of them hosted in GitHub
- http://wiki.ros.org/RecommendedRepositoryUsage/CommonGitHubOrganizations

Basic ROS Commands

(slides adapted from Roi Yehoshua, Bar-Ilan University)

Developing in ROS

- Download the file:
- Unzip it at the folder catkin_ws/src
- In the folder catkin_ws, execute the command:

```
catkin_make
```

- Execute the following command:

```
roslaunch robotcontrol turtlebot_in_stdr.launch
```

- This will launch a simulation of a Turtlebot robot

Developing in ROS

- Execute the following command:

```
roslaunch turtlebot_gazebo turtlebot_world.launch
```

- This will launch a simulation of a Turtlebot robot

ROS Basic Commands

- roscore
- roscd
- rosrun
- rosnode
- rostopic

Basic ROS Commands

- **roscore** – a collection of nodes and programs that are pre-requisites of a ROS-based system
- If your ROS system uses communications, it should be run before
- roscore is defined as:
  - master
  - parameter server
  - rosout
- Usage:
  - $roscore

Navigating through ROS packages

- **roscd**: roscd is part of the rosbash suite. It allows you to change directory (cd) directly to a package or a stack.
- Before using it, the correct environment variables should be set
  - Source correct the .bash file
- Usage:
  - $ roscd [locationname[/subdir]]

Executing a node within a package

- **rosrun** – allows you to run an executable in an arbitrary package without having to cd (or roscd) there first
- Usage:
  - $rosrun package executable
- Example
  - Run turtlesim
    - $rosrun turtlesim turtlesim_node

Basic ROS Commands

- **rosnode** – Displays debugging information about ROS nodes, including publications, subscriptions and connections
- Commands:

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

Basic ROS Commands

- Open a different terminal and run the following command:

```
rosnode list
```

- This shows the list of the nodes currently running

```
rosnode info /amcl
```

- This shows information about the node amcl
- A general tool for that is rqt

```
rqt
```

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

Basic ROS Commands

- Open a different terminal and run the following command:

```
rostopic echo /odom
```

- This shows the topic in which the estimated motion of the robot (propioceptive sensors) is published

Visualization

`rviz`

Developing in ROS

catkin Build System

- catkin is the official build system of ROS
- The original ROS build system was rosbuild
  - Still used for older packages
- Catkin is implemented as custom **CMake** macros along with some Python code
- Supports development on large sets of related packages in a consistent and conventional way

ROS Development Setup

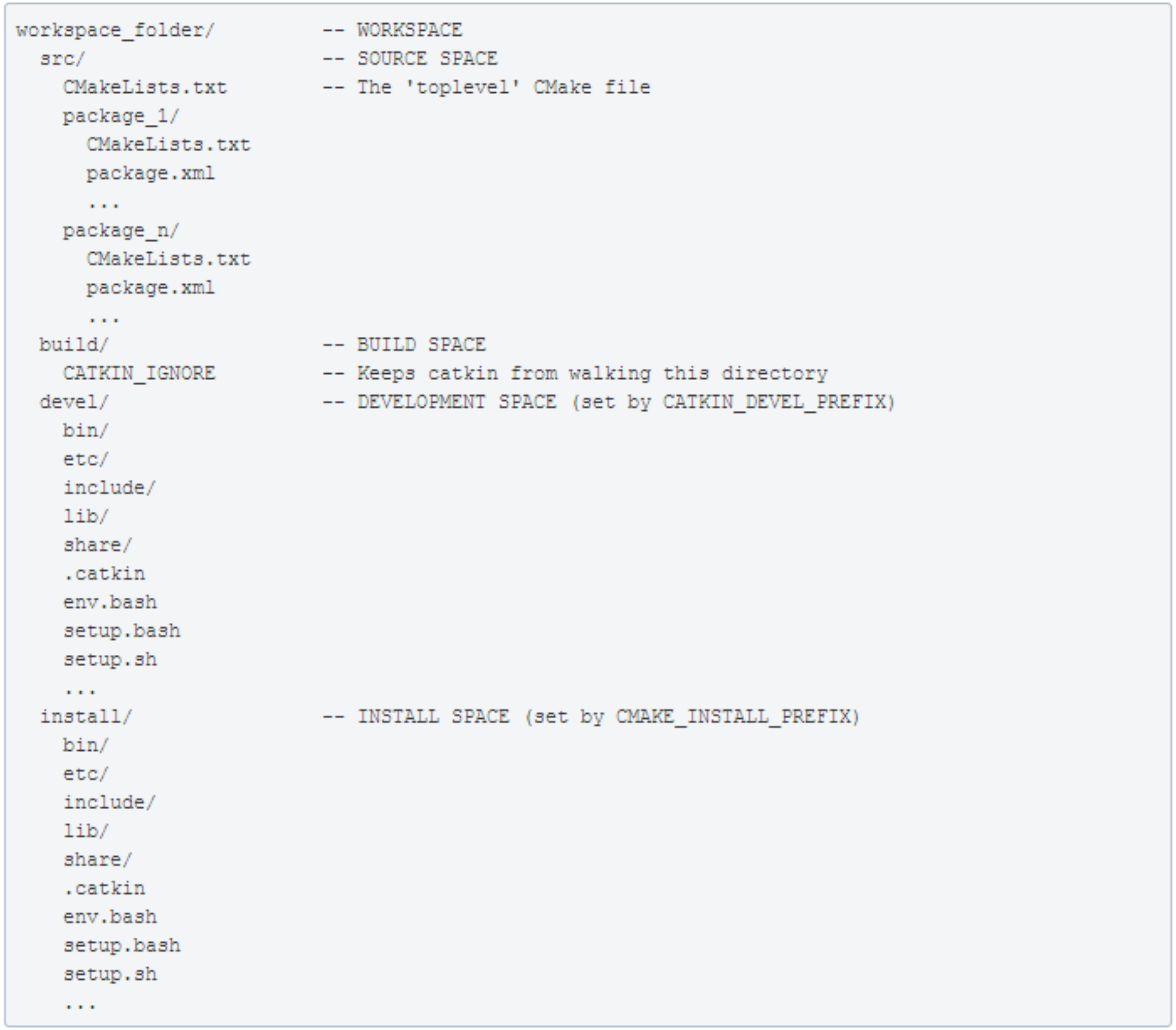
- Create a new catkin workspace
- Create a new ROS package
- Write the code
- Update the make file
- Build the package

catkin Workspace

- A workspace in which one or more catkin packages can be built
- Contains up to four different spaces:

Space	
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 Package

- 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/	C++ source files
scripts/	Python scripts
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

- XML file that defines properties about the package such as:
  - the package name
  - version numbers
  - authors
  - dependencies on other ROS packages

The Package Manifest

- Example for a package manifest:

```
<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>
```

CMakeLists.txt

- ROS uses CMake to build ROS packages
- The CMakeLists.txt file is the equivalent to a Makefile
- This file is the way we indicate how to build our package’s executables
- If you're unfamiliar with CMakeLists.txt, that's ok, because most ROS packages follow a very simple pattern that is described in the following slides

A basic ROS node in Python

```
if __name__ == '__main__':
    try:
           # initiliaze
        rospy.init_node('robotcontrol', anonymous=False)
          # tell user how to stop TurtleBot
          rospy.loginfo("To stop TurtleBot CTRL + C")

        robot=Turtlebot()
           # What function to call when you ctrl + c
        rospy.on_shutdown(robot.shutdown)

          #TurtleBot will stop if we don't keep telling it to move.  How often should we
tell it to move? 10 HZ
        r = rospy.Rate(10);

          # Twist is a datatype for velocity
        move_cmd = Twist()
          # let's go forward at 0.2 m/s
        move_cmd.linear.x = 0.2
          # let's turn at 0 radians/s
          move_cmd.angular.z = 0

          # as long as you haven't ctrl + c keeping doing...
        while not rospy.is_shutdown():
              # publish the velocity
            self.cmd_vel.publish(move_cmd)
              # wait for 0.1 seconds (10 HZ) and publish again
            r.sleep()
    except:
        rospy.loginfo("robotcontrol node terminated.")
```

A basic ROS node in Python

```
rospy.init_node('robotcontrol', anonymous=False)
```

- Initialize ROS. This allows ROS to do name remapping through the command line -- not important for now.
- This is also where we specify the name of our node. Node names must be unique in a running system (with anonymous=True a random name will be created).
- The name used here must be a base name, ie. it cannot have a / in it.

A basic ROS node in Python

```
rospy.on_shutdown(robot.shutdown)
```

- Callback that will be called when a signal terminates the node
  - Typically, CRTL+C (SIGINT)

A basic ROS node in Python

```
r = rospy.Rate(10);
```

- A Rate object allows you to specify a frequency that you would like to loop at. It will keep track of how long it has been since the last call to the sleep() method of the object, and sleep for the correct amount of time.
- In this case we tell it we want to run at 10hz.

```
r.sleep();
```

- Now we use the Rate object to sleep for the time **remaining** to let us hit our 10 Hz rate.

A basic ROS node in Python

```
while not rospy.is_shutdown():
```

- By default rospy will install a SIGINT handler which provides Ctrl-C handling which will cause rospy.is_shutdown() to return true if that happens.
- rospy.is_shutdown() will return true if:
  - a SIGINT is received (Ctrl-C)
  - we have been kicked off the network by another node with the same name
  - rospy.shutdown() has been called by another part of the application.

A basic ROS node in Python

```
rospy.loginfo("To stop TurtleBot CTRL + C")
```

- Output information to the console
- It is logged by ROS

Publishing and subscribing to data

- Your node typically needs to communicate with other nodes
- By publishing information
- By subscribing to information

Publishing data

```
def __init__(self):

        # Create a publisher which can "talk" to TurtleBot and tell
it to move
         # Tip: You may need to change cmd_vel_mux/input/navi to
/cmd_vel if you're not using TurtleBot2

        self.cmd_vel = rospy.Publisher('cmd_vel_mux/input/navi',
Twist, queue_size=10)

        self.listener = tf.TransformListener()
```

Publishing data

```
rospy.Publisher('cmd_vel_mux/input/navi', Twist, queue_size=10)
```

- Tell the master that we are going to be publishing a message of type **`Twist`** on the topic **`cmd_vel_mux/input/navi`**.
  - This lets the master tell any nodes listening on **`cmd_vel_mux/input/navi`** that we are going to publish data on that topic.
- The third argument is the size of our publishing queue.
  - In this case if we are publishing too quickly it will buffer up a maximum of 10 messages before beginning to throw away old ones.
- `rospy.Publisher` returns an object, which serves two purposes:
  - 1) it contains a publish() method that lets you publish messages onto the topic it was created with,
  - and 2) when it goes out of scope, it will automatically unadvertise.

Publishing data

```
# Twist is a datatype for velocity
move_cmd = Twist()

# let's go forward at 0.2 m/s
move_cmd.linear.x = 0.2
# let's turn at 0 radians/s
move_cmd.angular.z = 0

self.cmd_vel.publish(move_cmd)
```

- Now we actually broadcast the message to anyone who is connected.

Subscription on topics

```
rospy.Subscriber("/scan", LaserScan, self.callback)
```

- Subscribe to the `/scan` topic with the master.
- ROS will call the `callback()` function whenever a new message arrives.
- The 2nd argument is the data type.
- It can be also specified a queue. If the queue is full of messages, we will start throwing away old messages as new ones arrive.

Callback

```
def callback(self,data):
        self.laser = data
        rospy.loginfo("Laser received " + str(len(data.ranges)))
```

- This is the callback function that will get called when a new message has arrived on the subscribed topic.
- You should know which kind of data is on the topic
  - In this case, a ROS LaserScan
- Many types for the data are defined in sensor_msgs