ROS BASICS

NODES

- A ROS system is made up of many different programs, running simultaneously, which communicate with each other by passing messages
- It resembles a **graph structure**
- ROS programs are called **nodes** and each program is just one piece of a much larger system.

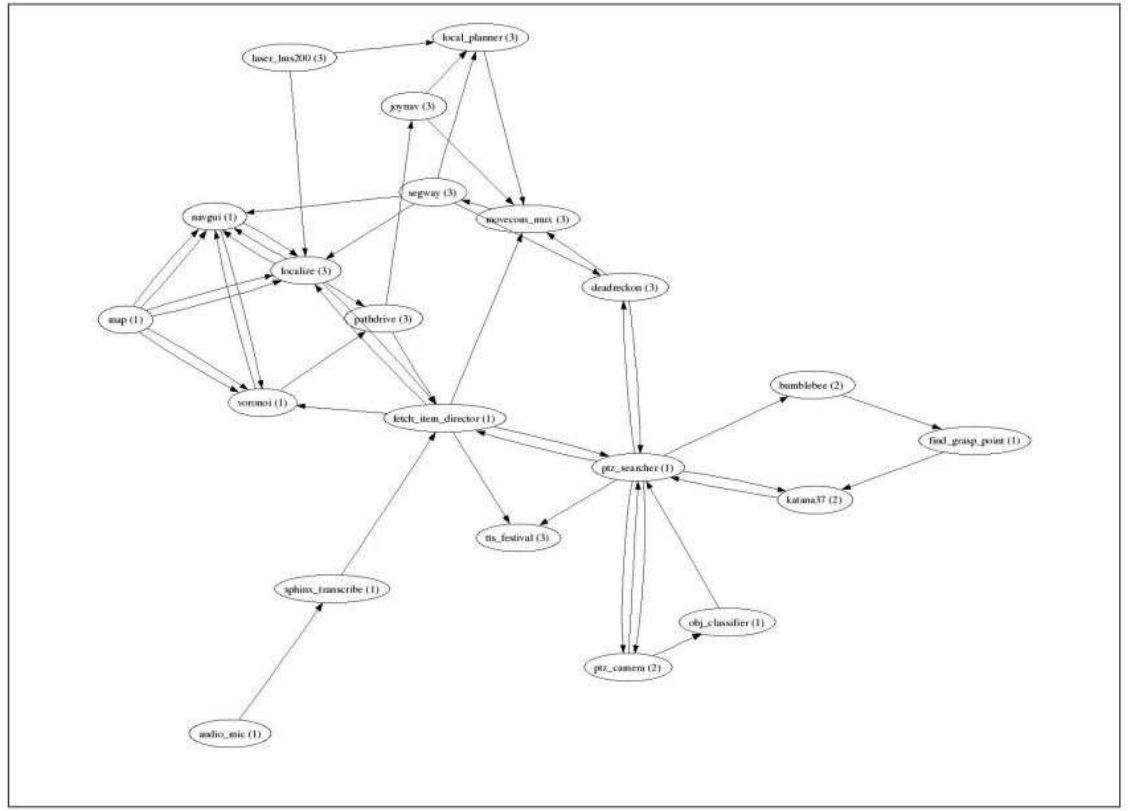


Figure 2-1. An hypothetical ROS graph for a fetch-an-item robot. Nodes in the graph represent individual programs; edges represent message streams.

- These nodes by themselves are typically not very useful

- Things only get interesting when nodes communicate with each other, exchanging information and data. The most common way to do that is through **topics**

- Important node commands
  - *rosrun <pkg_name> <node_name>*
  - *rosnode list*
  - *rosnode info <node_name>*

ROSCORE

- **Roscore** provides connection information to nodes so that they can transmit messages to each other
- Nodes register details of their messages to roscore and the information is stored in roscore
- When a new node appears, roscore provides it with the information that it needs to form a direct peer-to-peer connection with other nodes with which it wants to swap messages.
- Every ROS system needs a running roscore without which nodes cannot find other nodes to get messages from or send messages to

ROS BUILD SYSTEM

- A build system is responsible for generating 'targets' from raw source code that can be used by an end user. These targets may be in the form of libraries, executable programs, generated scripts, exported interfaces (e.g. C++ header files) or anything else that is not static code
- In ROS terminology, source code is organized into 'packages' where each package typically consists of one or more targets when built.
- Popular build systems that are used widely in software development are GNU Make, GNU Autotools, CMake, and Apache Ant (used mainly for Java)
- **Catkin** is the official build system of ROS
- Catkin combines CMake macros and Python scripts to provide some functionality on top of CMake's normal workflow

ROS WORKSPACE

- A catkin workspace is a folder where you modify, build, and install catkin packages

- A catkin workspace can contain up to four different spaces which each serve a different role in the software development process

- *mkdir -p ~/catkin_ws/src*
  cd ~/catkin_ws/
  catkin_make
  source devel/setup.bash

```
workspace_folder/          -- WORKSPACE
  src/                     -- SOURCE SPACE
    CMakeLists.txt         -- The 'toplevel' CMake file
    package_1/
      CMakeLists.txt
      package.xml
      ...
    package_n/
      CATKIN_IGNORE        -- Optional empty file to exclude package_n from being processed
      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

- Software in ROS is organized in **packages**
- Packages are the most **atomic unit** of build and the unit of release. This means that a package is the smallest individual thing you can build in ROS and it is the way software is bundled for release
- A package might contain ROS nodes, a ROS-independent library, a dataset, configuration files, a third-party piece of software, or anything else that logically constitutes a useful module
- The goal of these packages it to provide this useful functionality in an easy-to-consume manner so that software can be easily reused

- For a package to be considered a catkin package it must meet a few requirements:
  - The package must contain a catkin compliant package**.xml file**.
    - That package.xml file provides meta information about the package.
  - The package must contain a **CMakeLists.txt** which uses catkin.
    - If it is a catkin metapackage it must have the relevant boilerplate CMakeLists.txt file.
  - Each package must have its own folder
    - This means no nested packages nor multiple packages sharing the same directory.

```
workspace_folder/        -- WORKSPACE
  src/                   -- SOURCE SPACE
    CMakeLists.txt       -- 'Toplevel' CMake file, provided by catkin
    package_1/
      CMakeLists.txt     -- CMakeLists.txt file for package_1
      package.xml        -- Package manifest for package_1
    ...
    package_n/
      CMakeLists.txt     -- CMakeLists.txt file for package_n
      package.xml        -- Package manifest for package_n
```

PACKAGE CREATION

- First change to the source space directory of the catkin workspace you created

  cd ~/catkin_ws/src

- Now use the catkin_create_pkg script to create a new package

  *catkin_create_pkg <**package name**> std_msgs rospy roscpp*

- This will create a folder which contains a package.xml and a CMakeLists.txt, which have been partially filled out with the information you gave catkin_create_pkg

- Now you need to build the packages in the catkin workspace

  cd ~/catkin_ws
  catkin_make
  . ~/catkin_ws/devel/setup.bash

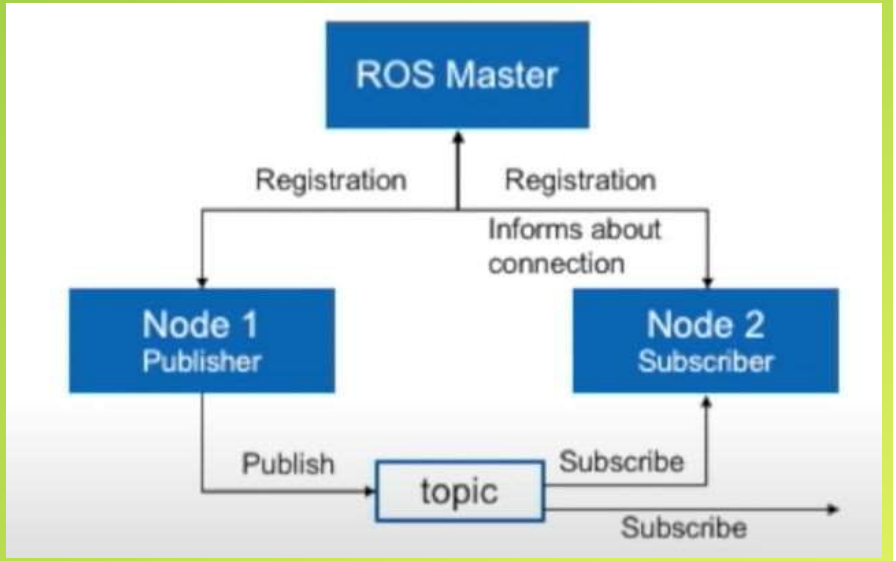
TOPIC

- A **topic** is a channel that acts as a pipe, where other ROS nodes can either publish or read information

- For example, the data from a laser rangefinders might be send on a topic called scan, with a message type of LaserScan, while the data from a camera might be sent over a topic called image, with a message type of Image

- Important Topic commands

  rostopic list
  rostopic info /topic
  rostopic echo /topic
  rostopic pub /topic

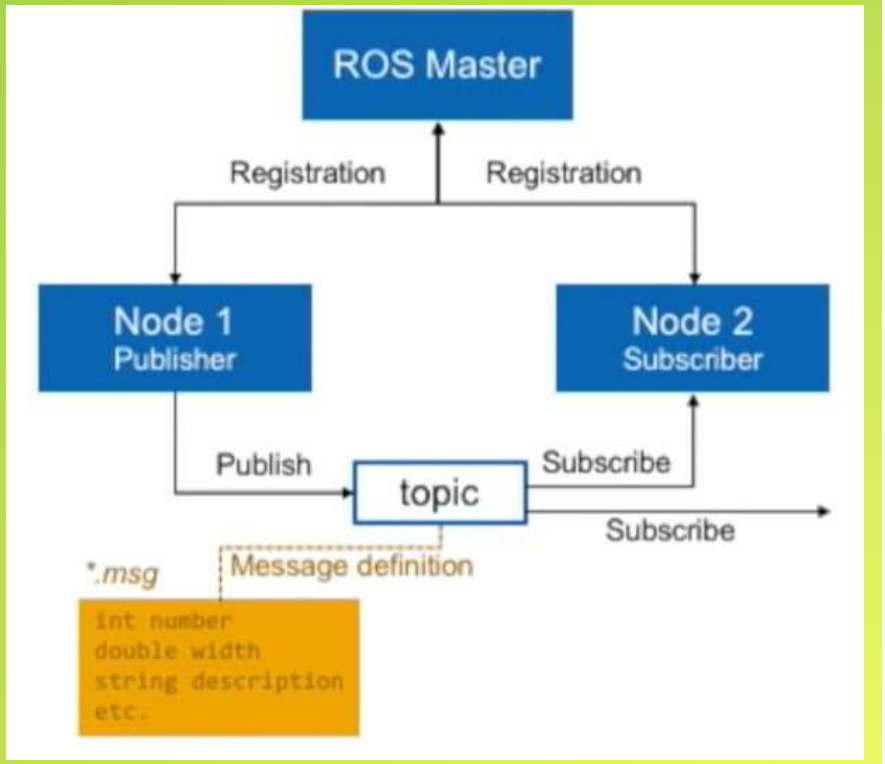


MESSAGES

- Data structure defining the type of Topic
- Comprised of nested structure of integers, floats, booleans, strings etc. and array of objects
- Messages are defined in *.msg files
- Important Message commands

  rostopic info /topic

  rosmsg show /message



PUBLISHER

PUBLISHER

- Before they can start to transmit data over topics, nodes must first **announce, or advertize**, both the topic name and the type 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

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

WRITING A PUBLISHER - GENERAL GUIDELINES

- Defining the environment
- Importing libraries
- Creating a publisher instance
  - Topic definition
  - Message definition
- Initialize node
- Define Rate
- Publish Data
- Define Main function

PUBLISHER - HELLO WORLD

```
#!/usr/bin/env python
# import libraries
import rospy
import time
from std_msgs.msg import String
# initialize publisher
pub = rospy.Publisher('chatter', String, queue_size=10)
# initialize node
rospy.init_node('talker',anonymous=True)
# define rate of publishing
rate = rospy.Rate(10) #10Hz
# run an infinite loop
while not rospy.is_shutdown():
    hello_str = 'Hello world'
    rospy.loginfo(hello_str)
    # publish the message
    pub.publish(hello_str)
    rate.sleep()
```

PUBLISHER - CODE EXPLAINED

 # ! /usr/bin/env python

➔ The first line makes sure your script is executed as a Python script

import rospy
from std_msgs.msg import String

➔ You need to import rospy if you are writing a ROS Node

➔ The std_msgs.msg import is to use the std_msgs/String message type for publishing

pub = rospy.Publisher('chatter', String, queue_size=10)

➔ It declares that your node is publishing to the chatter topic using the message type String

➔ The queue_size argument limits the amount of queued messages

rospy.init_node('talker', anonymous=True)

➔ It tells rospy the name of your node -- until rospy has this information, it cannot start communicating with the ROS Master

➔ anonymous = True ensures that your node has a unique name by adding random numbers to the end of NAME

rate = rospy.Rate(10) # 10hz

➔ This line creates a Rate object . With the help of its method sleep(), it offers a convenient way for looping at the desired rate.

➔ With its argument of 10, we should expect to go through the loop 10 times per second

```
while not rospy.is_shutdown():
        hello_str = "hello world %s" % rospy.get_time()
         rospy.loginfo(hello_str)
         pub.publish(hello_str)
         rate.sleep()
```

- ➔ This loop is a fairly standard rospy construct: checking the rospy.is_shutdown() flag and then doing work.
- ➔ You may also run ***rospy.sleep()***
- ➔ This loop also calls rospy.loginfo(str), which performs triple-duty: the messages get printed to screen, it gets written to the Node's log file, and it gets written to rosout. rosout is a handy tool for debugging

CHECKING EVERYTHING

rostopic/list

rostopic/info

rostopic/type

TURTLESIM SAMPLE PUBLISHER

```
#! /usr/bin/env python
import rospy
import time
from geometry_msgs.msg import Twist

pub = rospy.Publisher('/turtle1/cmd_vel',Twist,queue_size=10)
rospy.init_node('turtlesimdemo',anonymous = True)
move = Twist()

i = 0
while (i<3):
rospy.loginfo('moving straight')
move.linear.x = 2
pub.publish(move)
i+=1
rospy.sleep(1)

rospy.loginfo('stop')
move.linear.x=0
pub.publish(move)
```

Thank You