SAPIENZA
UNIVERSITÀ DI ROMA

Robot Programming
Robotic Middlewares

Giorgio Grisetti

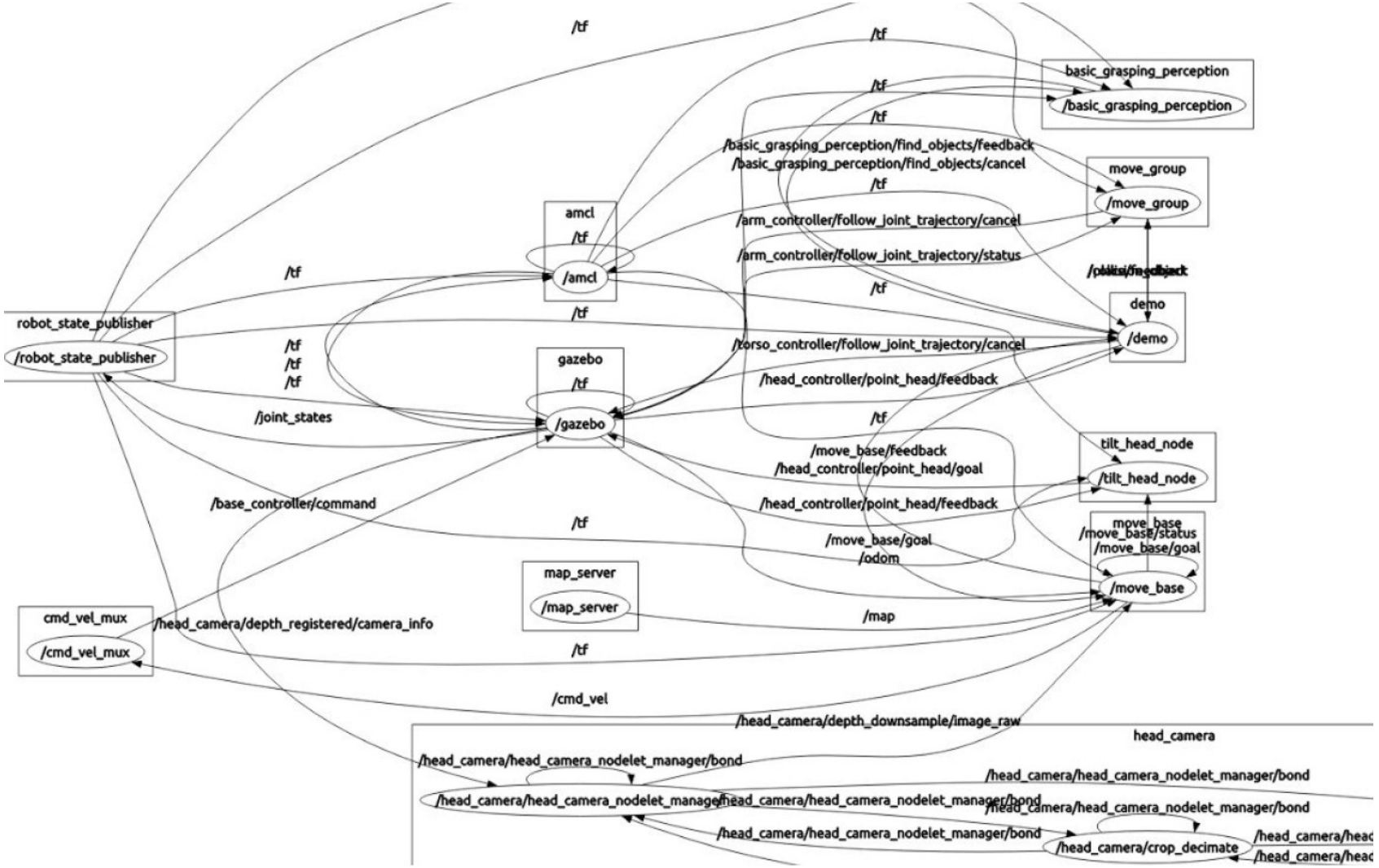
At the Beginning...

One single program was in charge of

- SENSING
- PLANNING
- ACTING

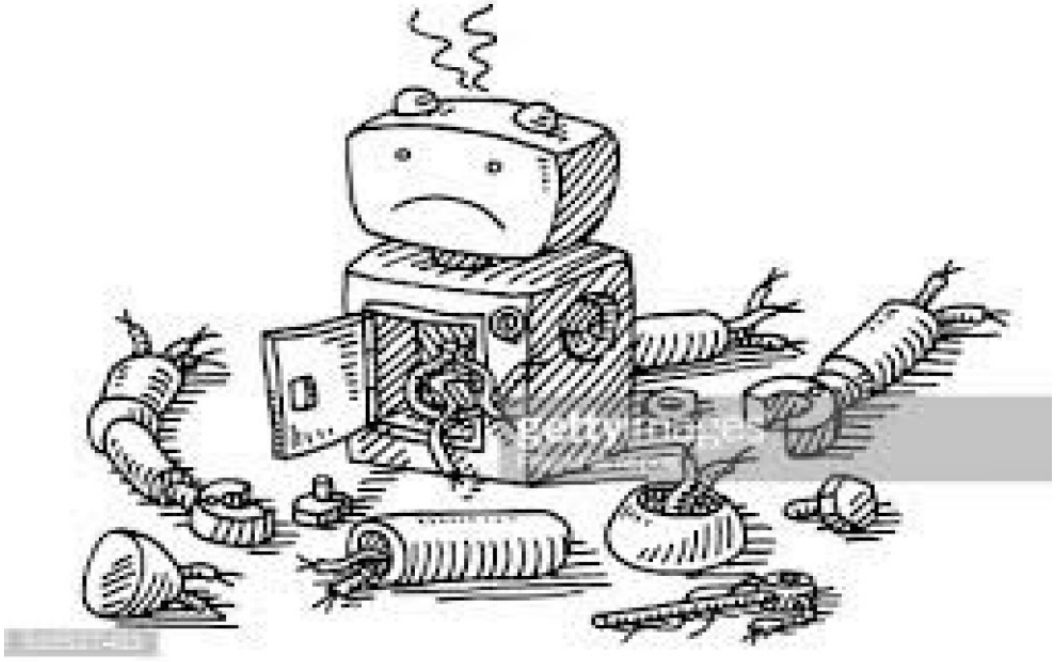
```
int main(int argc, char** argv){
    doStuff();
}
```

Example of a Typical Robotic System



Considerations about the Monolithe

- Robots are very complicated
- A single crash in a function might compromise the behavior of the entire system



Robots might be Dangerous

Ideal Robotic System

Functionalities encapsulated in processes, which communicate through messages

Benefits:

- If a process crashes, it can be restarted
- A functionality can be exchanged by replacing a process that provides it
- Decoupling of modules through IPC

Some Solutions

In the good old times, people aware of these aspect started using:

- Processes to isolate functionalities of the system
  - Camera Reader
  - Led blinker
  - ...
- Processes communicate through some IPC mechanism
  - Messages (less efficient, safer)
  - Shared Memory (more efficient, less safe)

Ideal Robotic System

Functionalities encapsulated in processes, which communicate through messages

Benefits:

- If a process crashes, it can be restarted
- A functionality can be exchanged by replacing a process that provides it
- Decoupling of modules through IPC

Robotic Middlewares in the Past

- Carmen
- OpenRDK
- OROCOS
- Microsoft Robotic Studio
- Player/Stage
- ……

ROS: Robot Operating System

Provides tools for:

- Message Definition
- Process Control
- File System
- Build System

Designed around the PR2 Robot

Standard packages built on ROS provide basic functionalities like:

- Device Support
- Navigation
- Control of Manipulator
- Object Recognition

ROS: Robot Operating System

Provides tools for:

- Message Definition
- Process Control
- File System
- Build System

Designed around the PR2 Robot

ROS

Standard packets built on ROS provide basic functionalities like:

- Device Support
- Navigation
- Control of Manipulator
- Object Recognition

Why ROS? (instead of others)

- **A critical mass of good people designed it**
- Code reuse (exec. nodes, grouped in packages)
- Distributed, modular design (scalable)
- Language independent (C++, Python, MATLAB,…)
- ROS-agnostic libraries (code is ROS independent)
- Easy testing (ready-to-use)
- Well maintained & collaborative environment

Integration with libraries

ROS provides seamless integration of famous libraries and popular open-source projects

pointcloudlibrary

ROS installation

noetic

ROS.org

About | Support | Discussion Forum | Service Status | Q&A answers.ros.org

Search: Submit

Documentation Browse Software News Download

kinetic/ Installation/ Ubuntu

Ubuntu install of ROS Kinetic

We are building Debian packages for several Ubuntu platforms, listed below. These packages are more efficient than source-based builds and are our preferred installation method for Ubuntu. Note that there are also packages available from Ubuntu upstream. Please see UpstreamPackages to understand the difference.

Ubuntu packages are built for the following distros and architectures.

Distro	amd64	i386	armhf
Wily	X	X	
Xenial	X	X	X

If you need to install from source (**not recommended**), please see source (download-and-compile) installation instructions.

Wiki

Distributions
ROS/Installation
ROS/Tutorials
RecentChanges
Ubuntu

Page

Immutable Page
Info
Attachments
More Actions:

User

ROS main concepts

- **Node**: process
- **Message**: Type of data structure used to communicate between processes
- **Topic**: stream of message instances of the same type used to communicate the evolution of a quantity
- **Service**: implements node-to-node RPC

Nodes

- Running instance of a ROS program
- Designed to be modular at a fine-grained scale
- A node can publish or subscribe to topics and provides or uses services
- Nodes are written by using the following libraries
  - roscpp (C++)
  - rospy (python)

http://wiki.ros.org/Nodes

Messages

Nodes communicate with each other by passing **messages**

- A message is a data structure of typed fields.
- Standard primitive types and arrays are supported
- Message can be nested and include arrays (like C structs)

Example:
Person.msg

```
string first_name
string last_name
string gender
uint8  age
```

http://wiki.ros.org/Messages

Topics

Messages are routed via a publish/subscribe transport mechanism based on topics

- A topic is identified by a string eg: "front_camera", or "odom"
- Topics can only transport ROS messages of a single type
- A node interested in a specific kind of data can subscribe to the corresponding topic
- Information production and consumption are decoupled

http://wiki.ros.org/Topics

Services

- Realize request/reply communication
- Defined as structure composed by a pair of messages
- A providing node or provider offers a service
- A client interested in a service sends a request and waits for a reply

Example:
Sum.srv

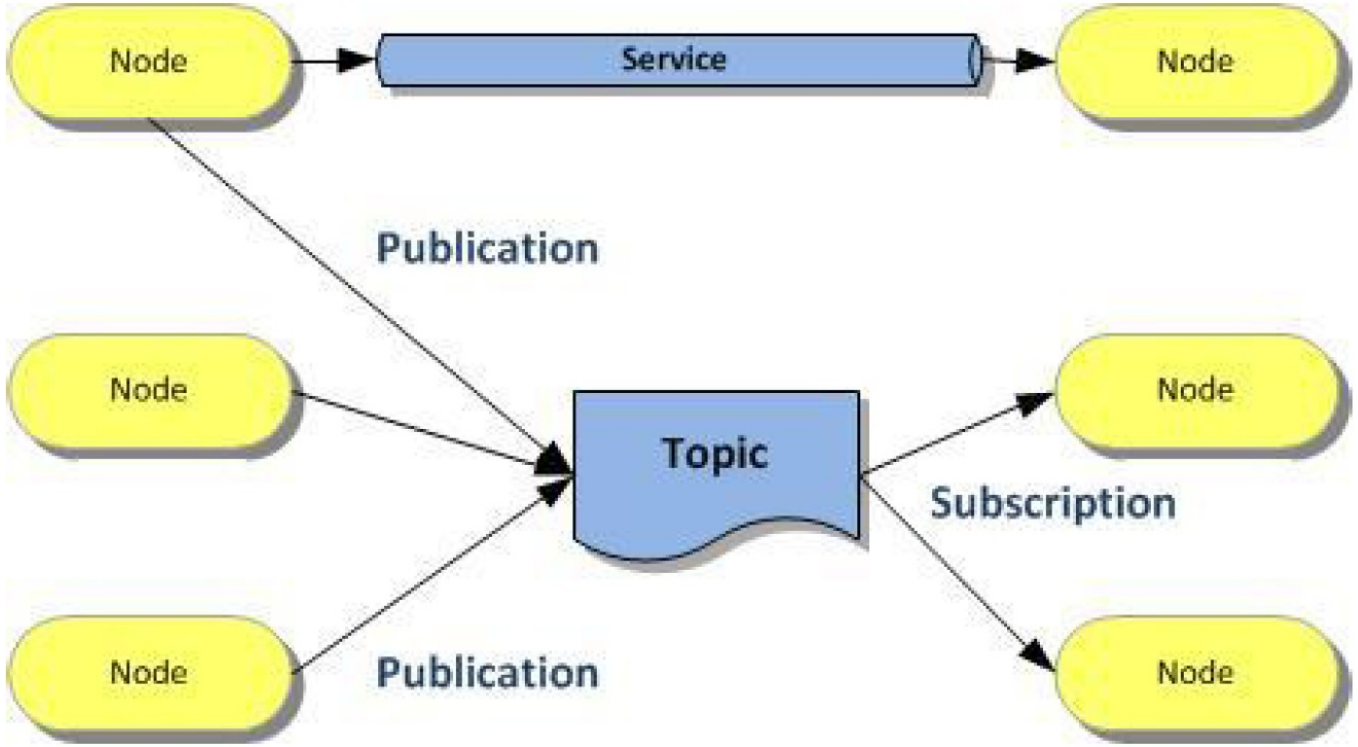
```
int64 a
int64 b
---
int64 sum
```

http://wiki.ros.org/Services

ROS communication scheme

Publishing: the action taken by a node when it wants to broadcast a message

Subscribing: requesting messages of a certain topic

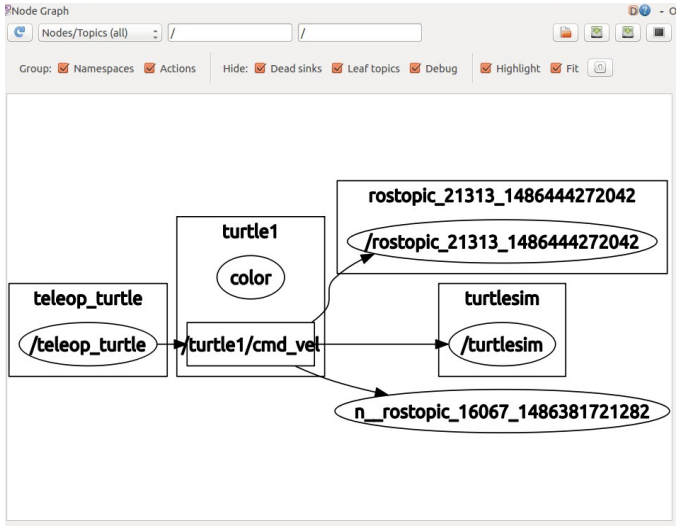


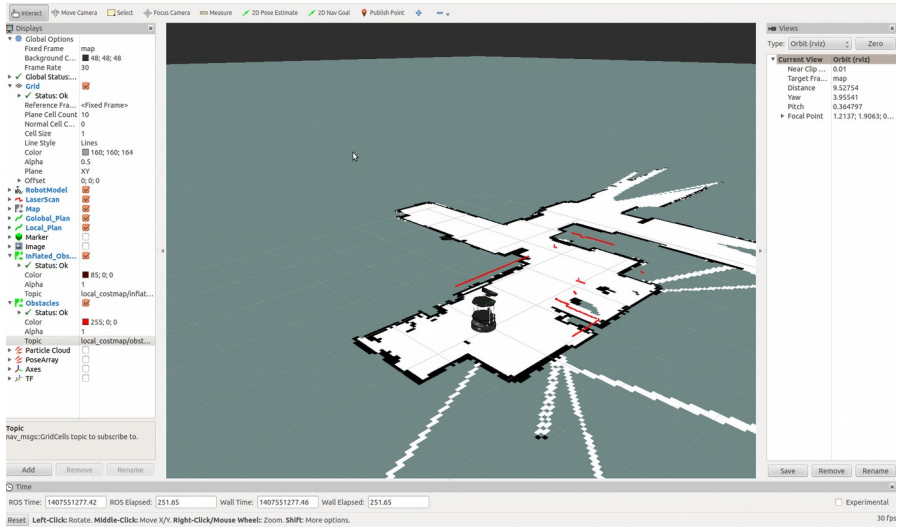
http://wiki.ros.org/ROS/Concepts

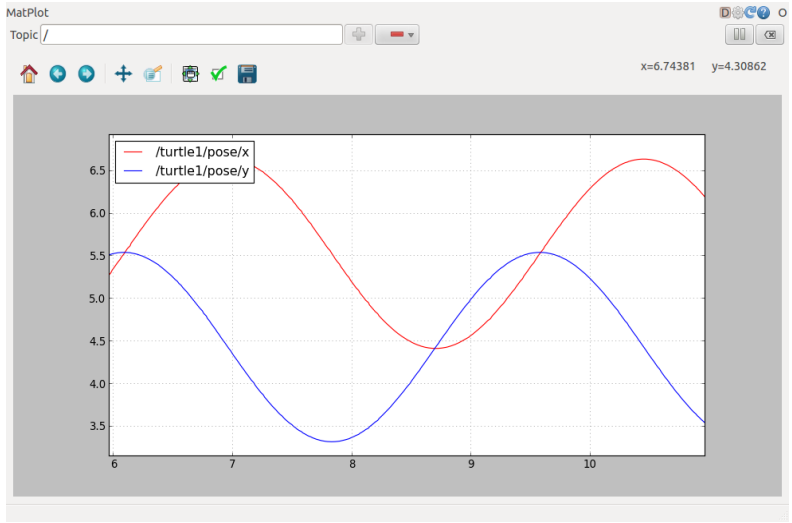
ROS Tools

- Command-line tools
- rqt_suite (e.g. rqt_plot, rqt_graph)
- Rviz

```
turtlebot@turtlebot-X200CA:~$ roscore
... logging to /home/turtlebot/.ros/log/6ef6185c-9127-11e4-83da-0c84dc11754b/ros
launch-turtlebot-X200CA-9168.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://192.168.0.8:45853/
ros_comm version 1.11.9


SUMMARY
========

PARAMETERS
 * /rosdistro: indigo
 * /rosversion: 1.11.9

NODES

auto-starting new master
process[master]: started with pid [9180]
ROS_MASTER_URI=http://192.168.0.8:11311/

setting /run_id to 6ef6185c-9127-11e4-83da-0c84dc11754b
process[rosout-1]: started with pid [9193]
started core service [/rosout]
```







http://wiki.ros.org/Tools

ROS core

Our instance of a special program should run in the system to support the ROS infrastructure:

Start it in terminal with:

```
$> roscore
```

It provides bookkeeping for

- nodes
- topics
- parameters

Once the connection is extablished, two nodes communicate directly (no master required)

http://wiki.ros.org/roscore

ROS core

roscore = rosmaster + parameter server + log aggregator

- **rosmaster**:
  - Directory for publisher/subscribers/services
  - Not a central communication node
- **Parameter server**:
  - Centralized parameter repository
  - Provides parameter access to all nodes
- **Log aggregator**:
  - Subscribes to */out* topic
  - Store output on filesystem

http://wiki.ros.org/ROS/Tutorials/UnderstandingServicesParams

Parameter Server

- The Parameter Server is shared, multi-variate dictionary that is accessible via its own APIs.
- Nodes use this server to store and retrieve parameters at runtime.
- It is intended to be used for static, non-binary data such as configuration parameters.

http://wiki.ros.org/ROS/Tutorials/UnderstandingServicesParams

Using Nodes

- Starting a node:

```
rosrun package_name executable_name args
```

(a node is a linux executable, if you know the path, you can start it witohout rosrun)

- Listing running nodes:

```
rosnode list
```

- Inspecting a node:

```
rosnode info node_name
```

- Killing a node:

```
rosnode kill node_name
```

http://wiki.ros.org/ROS/Tutorials/UnderstandingNodes

Using Topics

- Listing active topics:

  `rostopic list`

- Seeing all messages published on topics:

  `rostopic echo` *`topic_name`*

- Checking publishing rate:

  `rostopic hz` *`topic_name`*

- Inspecting a topic (message type, subscribers, etc…):

  `rostopic info` *`topic_name`*

- Publishing messages through terminal line:

  `rostopic pub -r` *`rate_hz`* *`topic_name`* *`message_type`* *`message_content`*

http://wiki.ros.org/ROS/Tutorials/UnderstandingTopics

Using Messages and Services

- Check **message** fileds:

  `rosmsg show` *`message-type`*

- Display a list of all **messages**:

  `rosmsg list`

- Show **service** description:

  `rossrv show` *`service-name`*

- Display a list of all **services**:

  `rossrv list`

http://wiki.ros.org/rosmsg

Using Parameters

- Set a parameter:

```
rosparam set parameter_name value
```

- Get a parameter:

```
rosparam get parameter_name
```

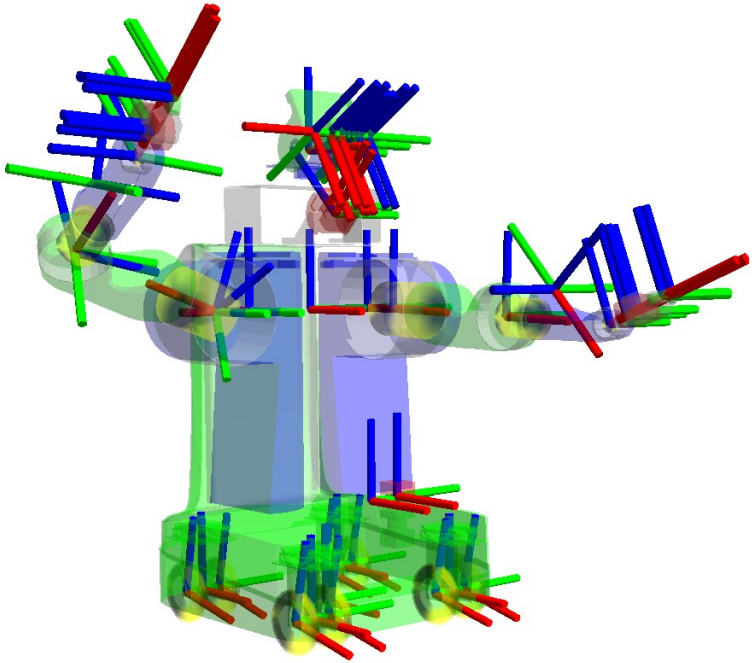
- Display all parameters:

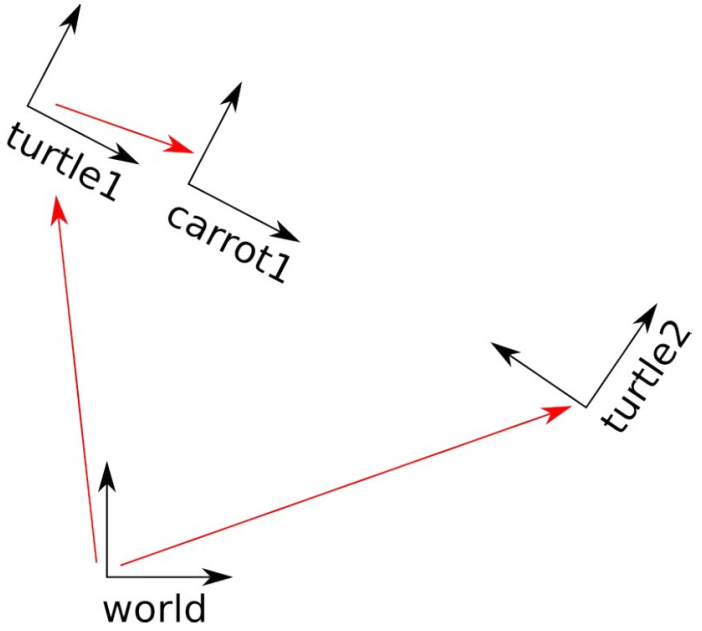
```
rosparam list
```

http://wiki.ros.org/rosparam

ROS tf

- Keep track of multiple coordinate frames over time
- Maintains the relationship between coordinate frames in time
- The user can transform points, vectors, etc between any any desired point in time





http://wiki.ros.org/tf

ROS Filesystem

- Groups of programs in ROS are organized in **packages**
- Each packages is a folder (which may contain also sub folders)
- One can jump to the directory of a package with:

  `roscd` *`package_name`*

- One can run a process of a package by issuing the command

  `rosrun` *`package_name exec_name`*

http://wiki.ros.org/ROS/Concepts#ROS_Filesystem_Level

Catkin

- Official build system of ROS
- Combines CMake macros and Python scripts to provide some functionality on top of CMake's normal workflow
- A build system is responsible for generating *targets* from raw source code that can be used by an end user
- *Targets* may be in the form of libraries, executable programs

http://wiki.ros.org/ROS/Concepts#ROS_Filesystem_Level

Catkin Workspace

```
workspace_folder/          -- WORKSPACE
  src/                       -- SOURCE SPACE
    CMakeLists.txt           -- The 'toplevel' Cmake file
    package_1/
      CMakeLists.txt
      package.xml
      ...
    package_n/
      CMakeLists.txt
      package.xml
      ...
  devel/                     -- DEVELOPMENT SPACE
  build/                     -- BUILD SPACE
```

Catkin Workspace configuration

```
$ source /opt/ros/noetic/setup.bash [setup ros environment]
$ mkdir -p ~/workspaces/[ws_name]/src
$ cd ~/workspaces/[ws_name]/src
$ catkin_init_workspace [initialize the workspace]
$ cd ~/workspaces/[ws_name]/
$ catkin build     [compiles all the package in src folder]
```

Open `~/.bashrc` and add the following lines:

```
#ROS
source ~/workspaces/[ws_name]/devel/setup.bash
```

Or

```
$ cd ~/workspaces/[ws_name]/
$ source devel/setup.bash
```

Anatomy of a ROS Node

```
ros::Publisher pub;
void my_callback(MsgType* m) { // function called whenever a message is received
    OtherMessageType m2;
    … // do something with m and valorize m2
    pub.publish(m2);
}
```

Anatomy of a ROS Node

```
int main(int argc, char** argv){
    ros::init(argc, argv, ''my_node_name''); // initializes the ros ecosystem

    ros::NodeHandle n; // object to access the namespace facilities

    pub.advertise<OtherMessageType>("my_topic"); // tell the core that you will publish
    // messages on a topic named "my_topic"

    Subscriber s =
    n.subscribe<MessageType*>("sub_topic",my_callback);  // subscribe to the topic "sub_topic" and
    // attach "my_callback". It will be called
    // whenever a subscribed message arrives

    ros::spin();       // spin over the callbacks of the node
    // and runs them if needed
}
```

ROS Namespaces

Provide a hierarchical naming structure used for items such as:

- Nodes
- Parameters
- Topics
- Services
- Other namespaces

Namespaces can be organized in hierarchies of arbitrary depth

Useful to encapsulate data under a single name

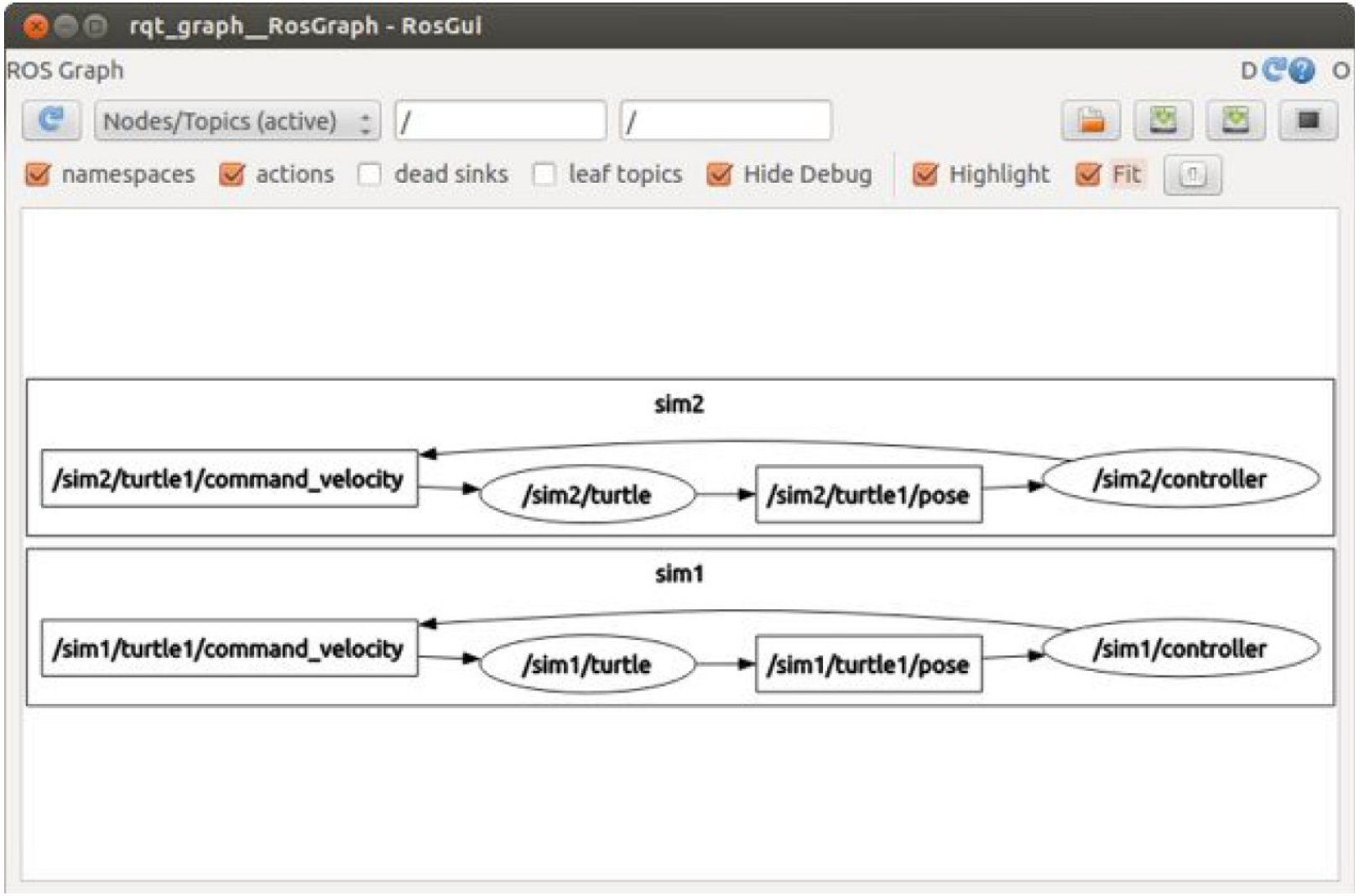
http://wiki.ros.org/Names

ROS Namespaces

- Global: `/global/name`
  - `/odom`
  - `/turtle_1`
  - `/turtle_1/pose`
- Relative: `relative/name`
  - `pose` `[in turtle_1 node -> global = /turtle_1/pose]`
  - `odom` `[outside namespace -> global = /odom]`
- Private: `~private/name`
  - `~foo/bar` `[in turtle_1 node -> global = /turtle_1/foo/bar]`
    this is accessible only from turtle_1

http://wiki.ros.org/Names

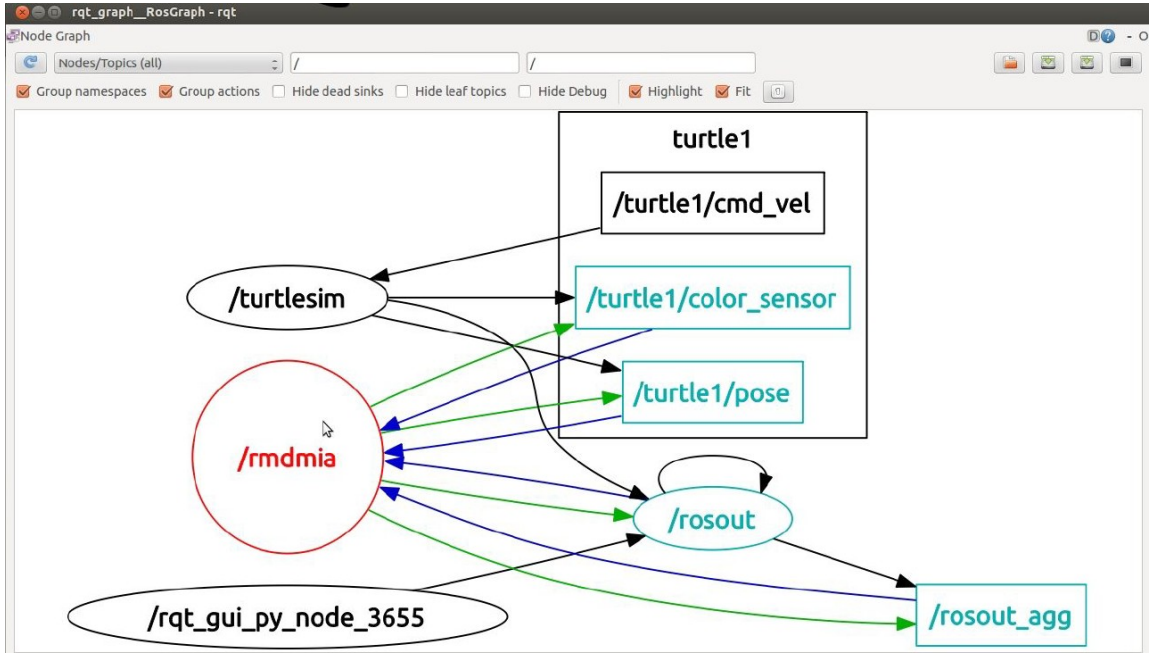
ROS Namespaces



Viewing the graph

- Graphically intuitive, easy to visualize the publish/subscribe relationships between nodes:

```
$ rqt_graph
```



Roslaunch

Mechanism for starting the master and many nodes all at once, using a file called **launch file**

```
<launch>

  <group ns="turtlesim1">
    <node pkg="turtlesim" name="sim" type="turtlesim_node"/>
  </group>

  <group ns="turtlesim2">
    <node pkg="turtlesim" name="sim" type="turtlesim_node"/>
  </group>

  <node pkg="turtlesim" name="mimic" type="mimic">
    <remap from="input" to="turtlesim1/turtle1"/>
    <remap from="output" to="turtlesim2/turtle1"/>
  </node>

</launch>
```

`roslaunch` *`package_name launch_file_name`*

Exercises

1. Modify the program in class to run incremental ICP between consecutive scans, and print a transform (Delta_lidar).

2. Modify the program above output a message of type nav_msgs/Odometry containing the integrated pose of the robot

$$p_t = p_{t-1} * \text{Delta\_lidar}_t$$