Sapienza University of Rome

Robot Programming
Robotic Middlewares

Giorgio Grisetti
Cristiano Gennari

https://groups.google.com/forum/#!forum/laboratorio_ai

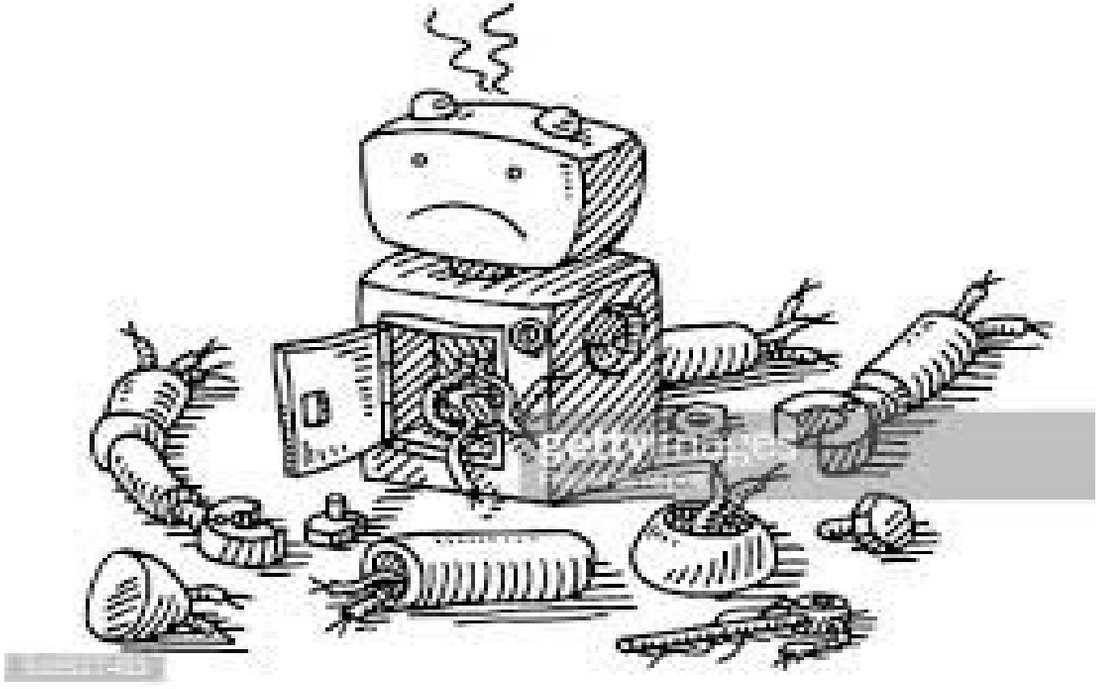
How was it in the past

- One single program was in charge of
  - Sensing
  - Planning
  - Acting

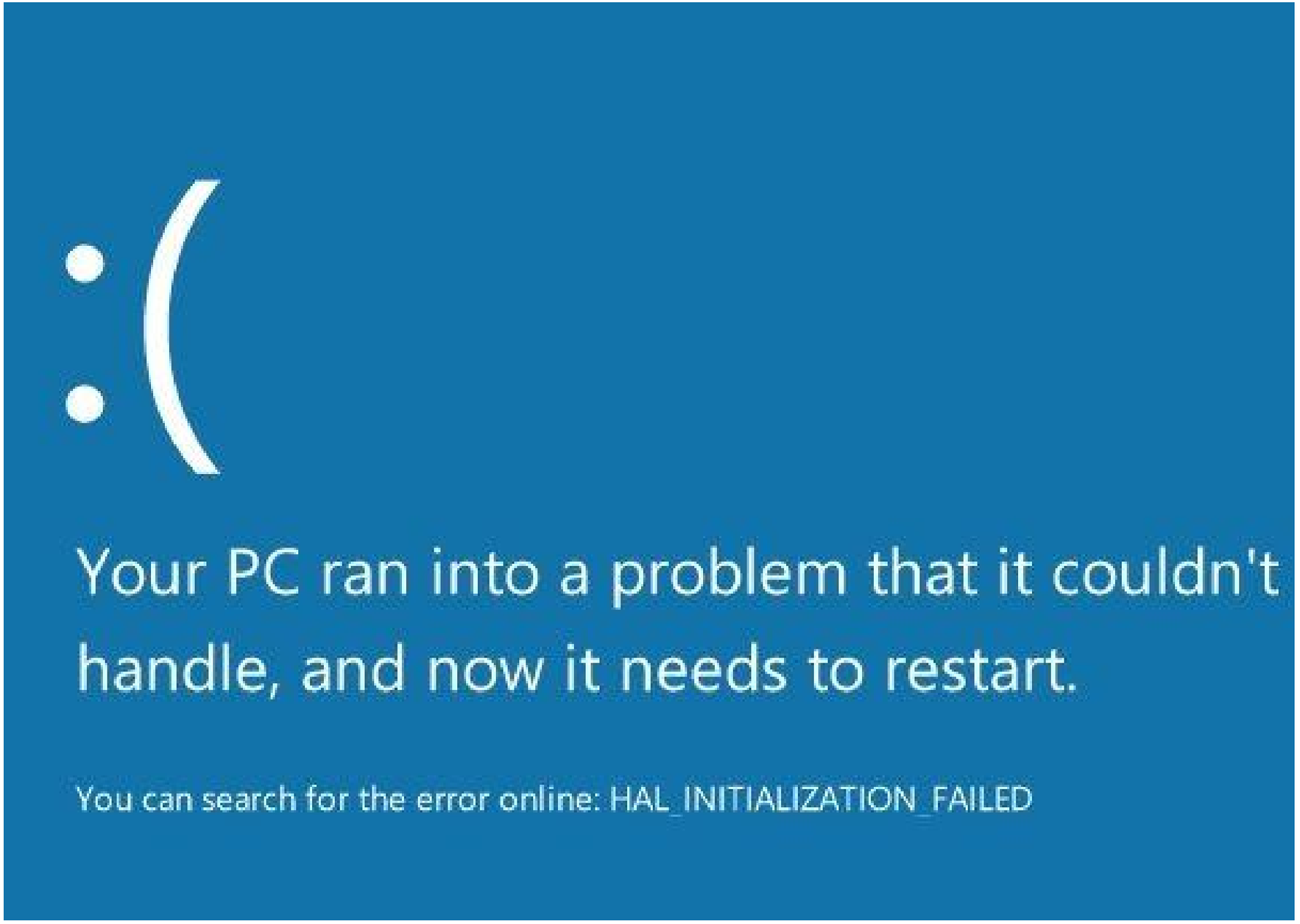
```
int main (int argc, char** argv) {
   doStuff();
}
```

How was it in the past

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



Robots are Dangerous

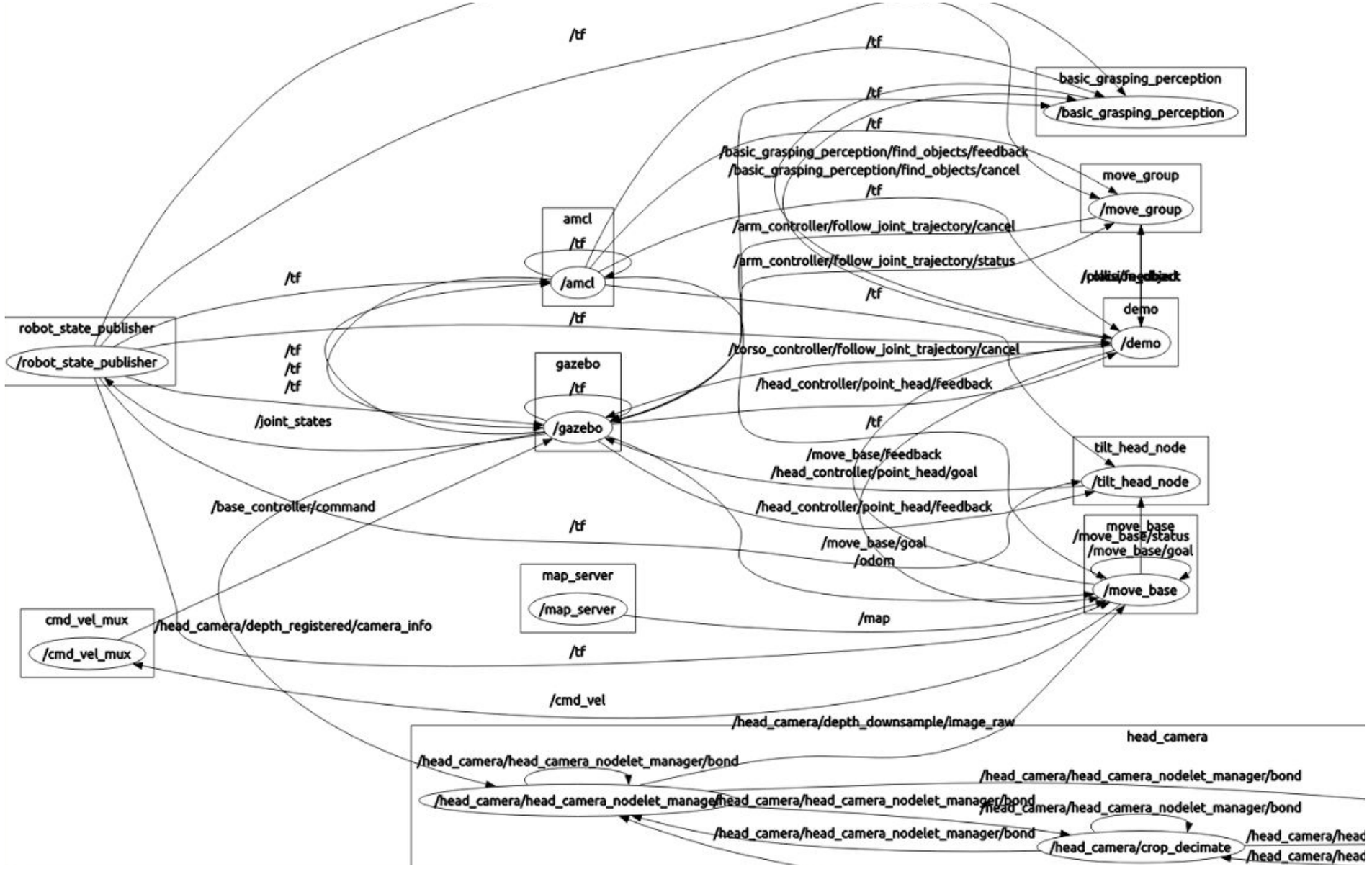


Robots are Dangerous

Actions Taken

- In the good old times, people aware of these aspects 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)

Anatomy of a complex Robot System



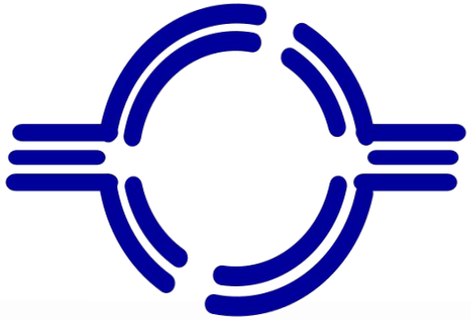
Anatomy of a complex Robot System

- Functionalities encapsulated in processes
- Processes 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
- SPQR-RDK
- OROCOS
- Microsoft Robotic Studio
- Player/Stage
- .....



Microsoft®
Robotics
Developer
Studio

ROS

- Robot Operating System
- Provides tools for
  - Message Definition
  - Process Control
  - File System
  - Build System

- Designed around the PR2 Robot

- Provides basic functionalities like:
  - Device Support
  - Navigation
  - Control of Manipulator
  - Object Recognition

ROS

«Why ROS instead of OROCOS, Player, Robotics Studio, (…)?»

- Code reuse (exec. *nodes*, grouped in *packages*)
- Distributed, modular design (scalable)
- Language independent (C++, Python, Java, …)
- ROS-agnostic libraries (code is ROS indep.)
- Easy testing (ready-to-use)
- Vibrant community & collaborative environment

Integration with libraries

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

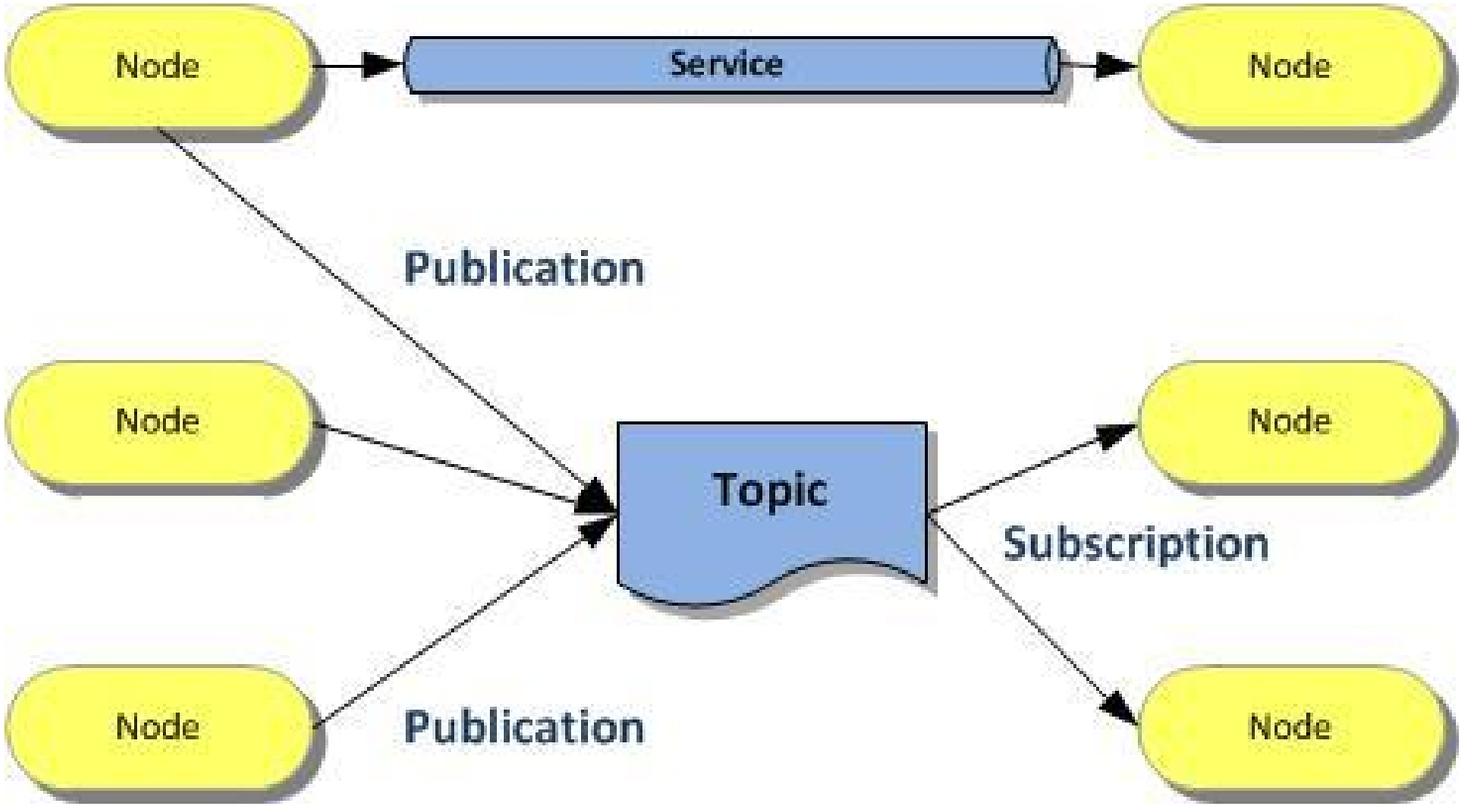
ROS installation

http://wiki.ros.org/indigo/Installation/Ubuntu

ROS definitions

- **Node**: process
- **Message**: Type of a data structure used to communicate between processes
- **Topic**: stream of message instance of the same type used to communicate the evolution of a quantity
  - e.g. A CameraNode will publish a stream of images.
  - Each image is of type ImageMessage (a matrix of pixels).
- **Publishing**: the action taken by a node when it wants to broadcast a message
- **Subscribing**: requesting messages of a certain topic

ROS definitions



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

Viewing topics

- Listing active topics:

```
rostopic list
```

- Seeing all messages published on topic:

```
rostopic echo topic-name
```

- Checking publishing rate:

```
rostopic hz topic-name
```

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

```
rostopic info topic-name
```

- Publishing messages trough terminal line:

```
rostopic pub -r rate-in-hz topic-name message-type message-content
```

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

Services

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

Parameters

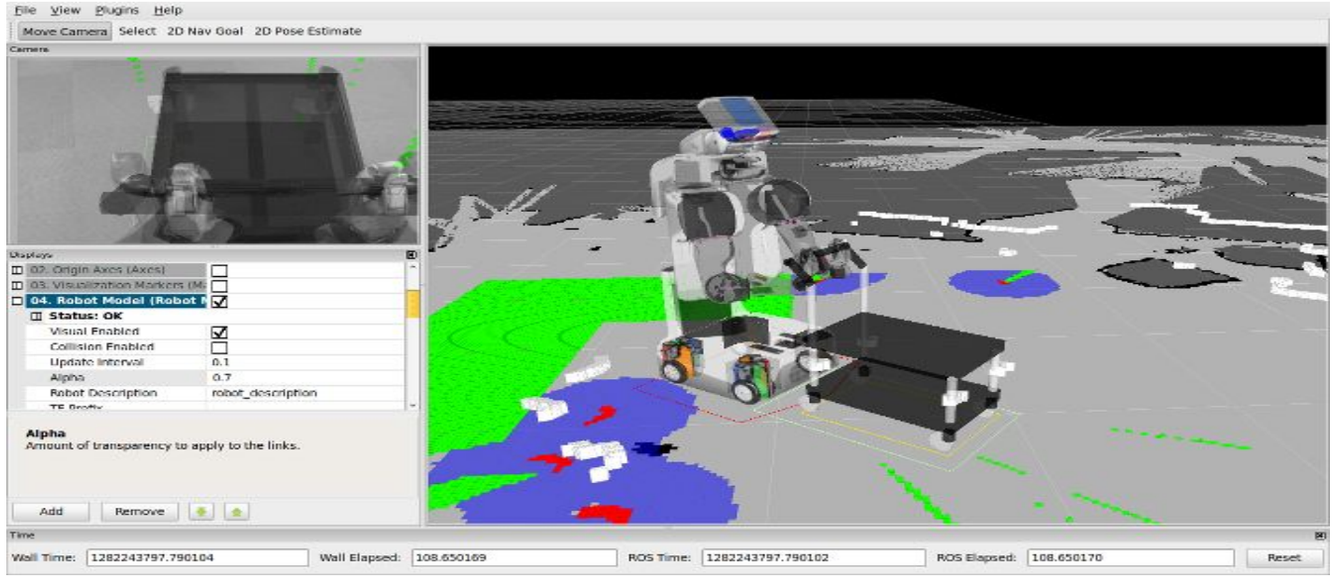
- Matching namespaces
- Tracking or setting values to nodes
- Actively queried by the nodes, they are most suitable for configuration information that will not change (much) over time

```
double max_tv;
private_nh.param("max_tv", max_tv, 2.0);

double max_rv;
private_nh.param("max_rv", max_rv, 2.0);

planner->setMaxVelocity(max_tv, max_rv);
```

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

ROS Tools

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



ROS Master

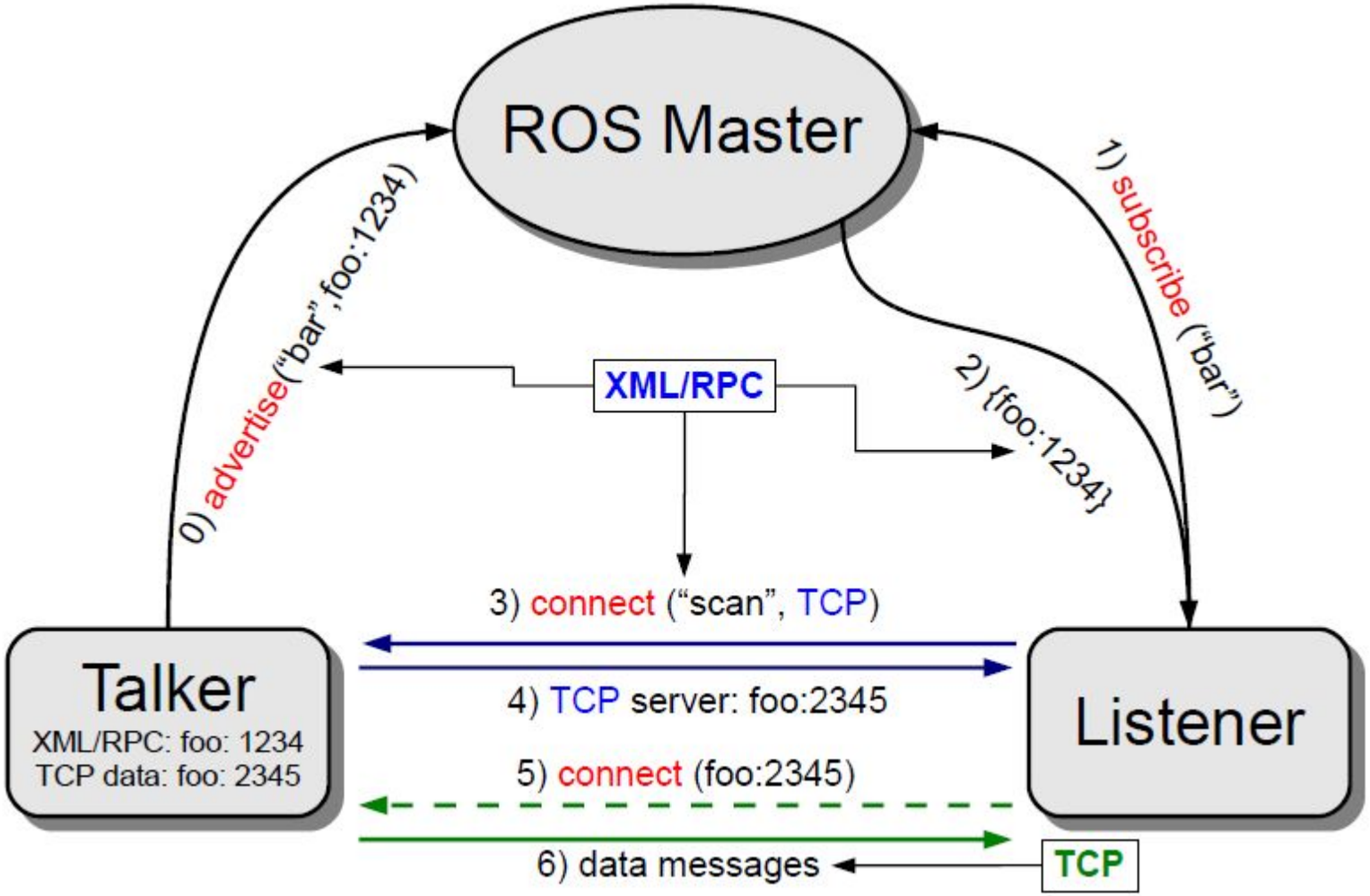
- One instance of a special program should run in the system to support the ROS infrastructure: **`roscore`**
- Start it on a terminal with
  - `$> roscore`
- It provides bookkeeping of which nodes are active, which topics are requested by whom, and other facilities
- Nodes need to communicate with the master only at the beginning to know their peers, and which topics are offered
- After that the communication among nodes is peer-to-peer

ROS core

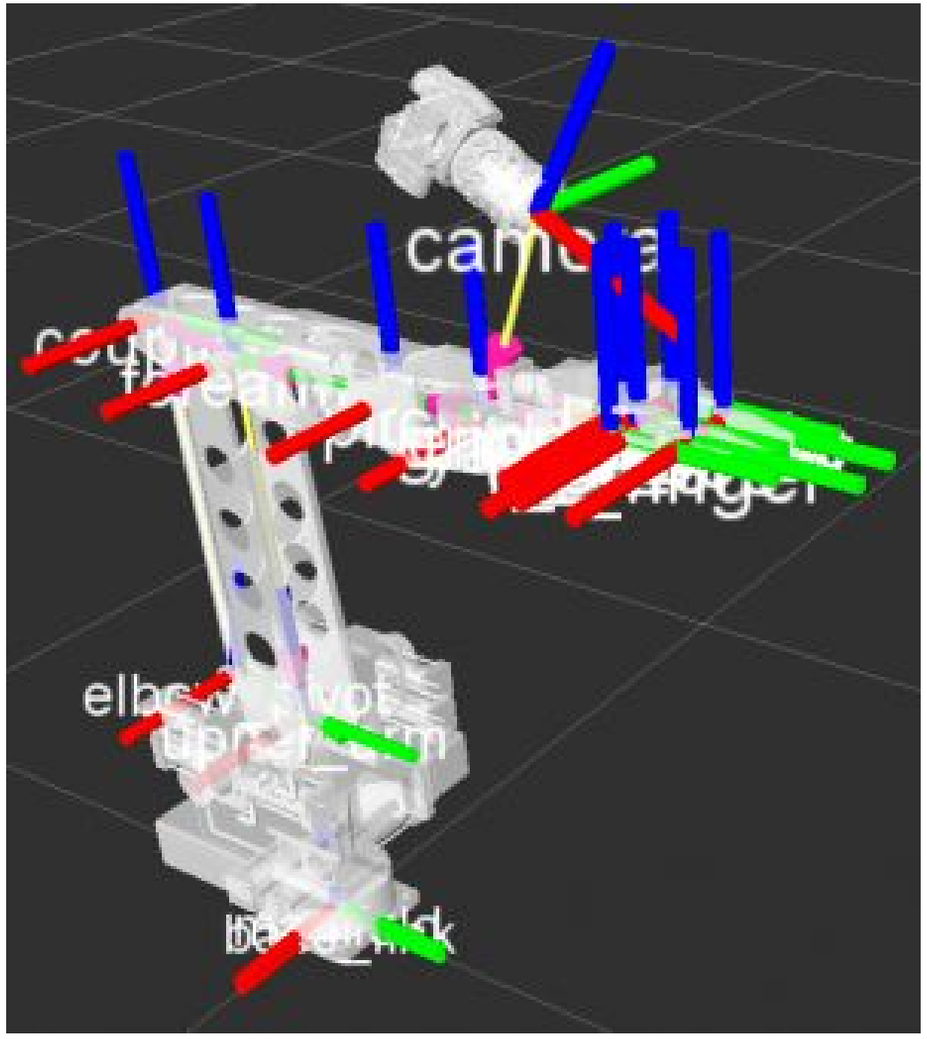
Roscore = rosmaster + parameter server + log aggregator

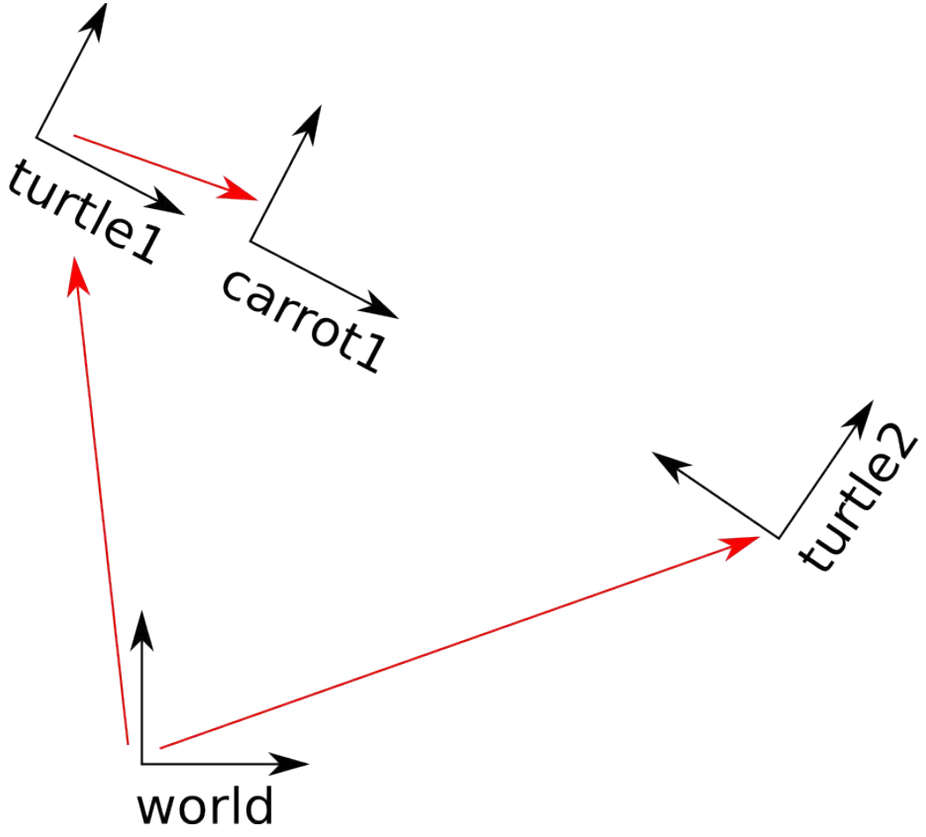
- Rosmaster
  - **directory for publisher / subscribers / services,XMLRPC API,not a central communication node.**
- Parameter server
  - **centralized parameter repository, provides parameter access to all nodes, XMLRPC data type.**
- Log aggregator
  - **subscribes to */out* topic, store output on filesystem.

ROS Master

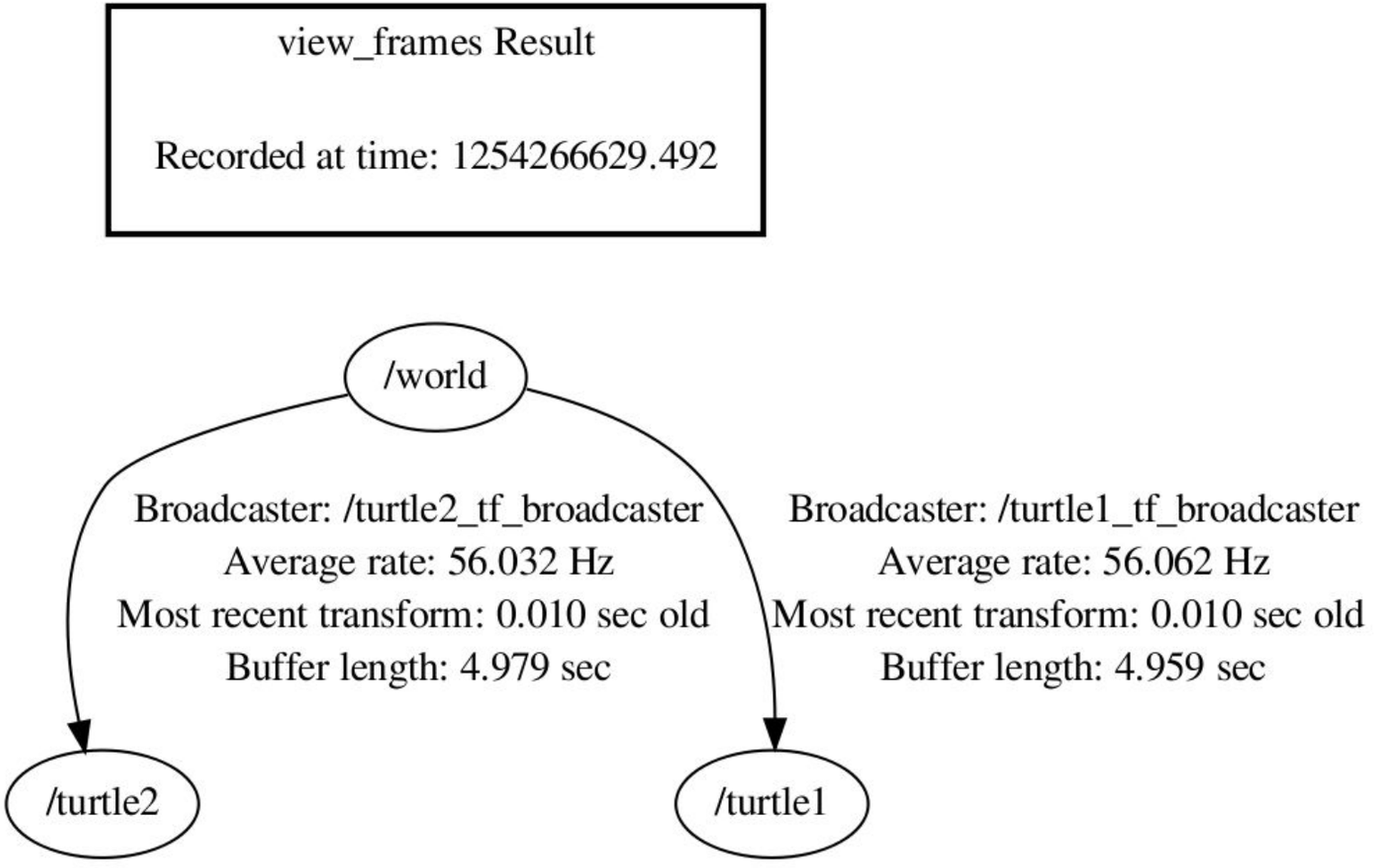

ROS Master
0) advertise("bar",foo:1234)
1) subscribe ("bar")
XML/RPC
2) {foo:1234}
3) connect ("scan", TCP)
4) TCP server: foo:2345
5) connect (foo:2345)
6) data messages
Talker
XML/RPC: foo: 1234
TCP data: foo: 2345
Listener
TCP

ROS tf





ROS tf



http://wiki.ros.org/tf/Tutorials

ROS Filesystem

- Groups of programs in ROS are organized in Packages
- Each package 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> <process-name> <args>
```

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/indigo/setup.bash
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/src
$ catkin_init_workspace
$ cd ~/catkin_ws/
$ catkin_make
```

Open ~/.bashrc and add the following lines:

```
# ROS
source ~/catkin_ws/devel/setup.bash
```

Catkin Make

- catkin_make is a convenience tool for building code in a catkin workspace
- Running the command:

```
~/catkin_ws$ catkin_make
Base path: ~/catkin_ws
Source space: ~/catkin_ws/src
Build space: ~/catkin_ws/build
Devel space: ~/catkin_ws/devel
Install space: ~/catkin_ws/install
####
#### Running command: "make cmake_check_build_system" in "~/catkin_ws/build"
####
####
#### Running command: "make -j8 -l8" in "~/catkin_ws/build"
####
```

Nodes

- Running instance of a ROS program
- Starting a node:

```
rosrun package-name executable-name
```

- Listing running nodes:

```
rosnode list
```

- Inspecting a node:

```
rosnode info node-name
```

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

Anatomy of a ROS Node

```
ros::Publisher pub;

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

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

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

    // tell the world that you will provide a topic named "published_topic"
    pub.advertise<OtherMessageType>("published_topic");

    // tell the world that you will provide a topic named "published_topic"
    Subscriber s =n.subscribe<MessageType*>("my_topic",my_callback);
    ros::spin();
}
```

Namespaces

- A directory which contents are items of different names.
  - nodes
  - topics
  - other namespaces

- Namespaces can be organized in hierarchies of arbitrary depth

Namespaces

- Global

```
/turtle1
/cmd_vel
/turtle1/pose
/run_id
/count_and_log/set_logger_level
```

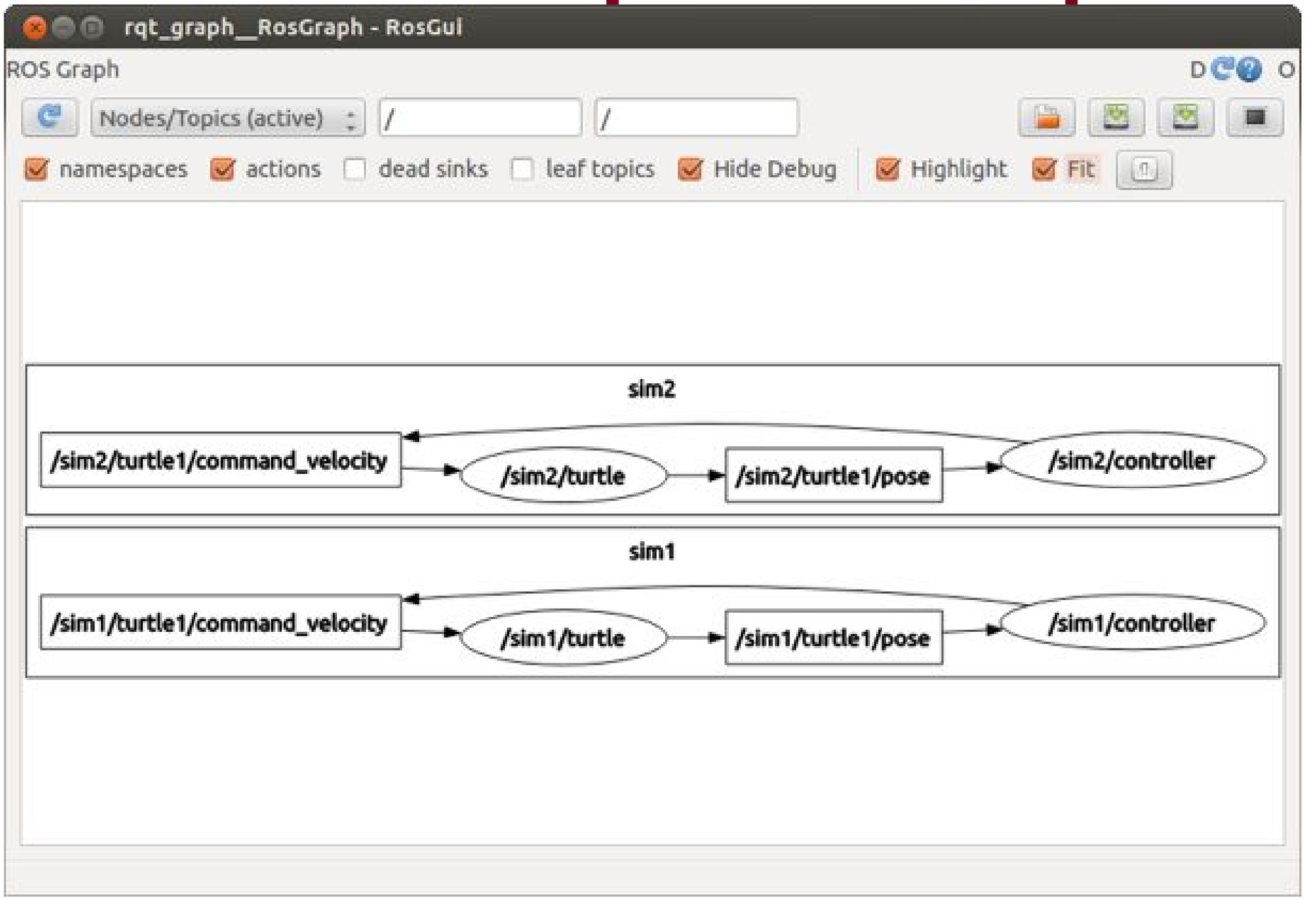
- Relative (with a default namespace)

```
/turtle1    +    cmd_vel    =    /turtle1/cmd_vel
(default)       (relative)      (complete namespace)
```

- Private (with the node's name)

```
/sim1/pub_vel    +    ~max_vel    =    /sim1/pub_vel/max_vel
 (node's name)          (private)         (complete namespace)
```

ROS namespaces example



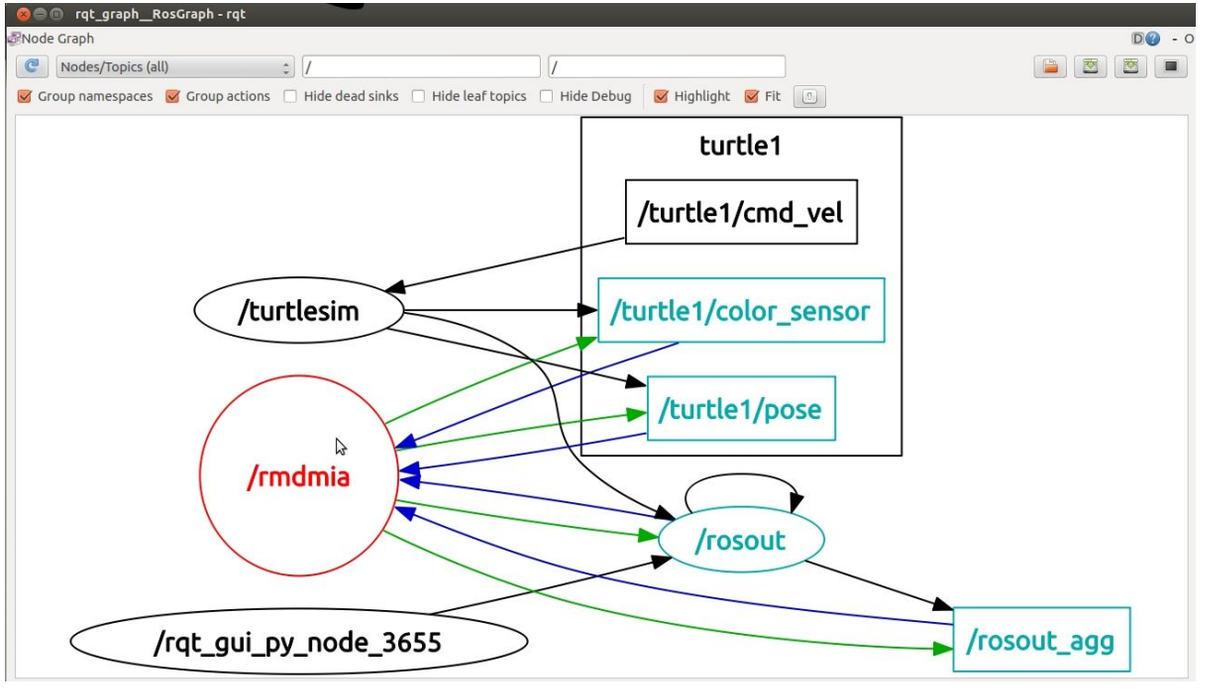
Viewing the graph

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

  `rqt_graph`

- All nodes publish on the topic /rosout (not the node!) subsribed by the node

  `/rosout`



Roslaunch

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

Homework (½)

- In a ros workspace clone and compile the ros_tutorials node from:

  https://github.com/gennari/ros_tutorials

- Write a new node that has a service for creating N different circles in random points on the turtlesim windows.
- You can use the service provided from turtlesim node: *spawnCircle*. It takes as input *x* and *y* and returns a list of created circles in the form *[(id,x,y),...]*.

Homework (2/2)

- Write a new node that deletes the red circles when hit by the turtle
- You can use the services provided from turtlesim node:
  - ***removeCircle*** : it takes as input an *id* and returns a list of remaining circles in the form *[(id,x,y),...]*.
  - ***getCircles*** *:* it takes an empty input and returns a list of circles in the form *[(id,x,y),...]*.
- And the topics:
  - ***/turtle1/color_sensor*** to check if a turtle hits a red circle.
  - ***/turtle1/pose*** to check the position of the turtle.