Escuela AI HUB de verano
Sobre IA y Educación

ROBÓTICA COLABORATIVA Y SISTEMAS MULTIROBOTS

José Mª Bengochea & Ángela Ribeiro - CAR (CSIC-UPM)
(2 de Julio de 2024)

CSIC Consejo Superior de Investigaciones Científicas | CSIC HUB Inteligencia Artificial
Fundación "la Caixa"
IFIC Institut de Física Corpuscular
Universitat de València

What is ROS?

- ROS is an open-source **robot operating system**

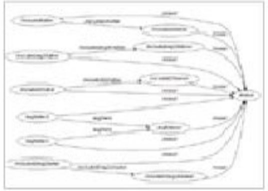
- A set of software libraries and tools that help you build robot applications that work across a wide variety of robotic platforms

- Originally developed in 2007 at the Stanford Artificial Intelligence Laboratory and development continued at Willow Garage

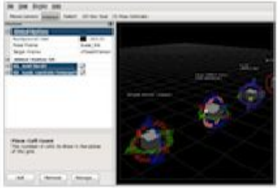
- Since 2013 managed by OSRF (Open Source Robotics Foundation)

- De facto standard for robot programming

ROS = Robot Operating System

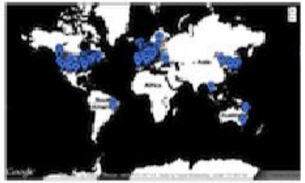


+



+

+



ros.org

Plumbing

- Process management
- Inter-process communication
- Device drivers

Tools

- Simulation
- Visualization
- Graphical user interface
- Data logging

Capabilities

- Control
- Planning
- Perception
- Mapping
- Manipulation

Ecosystem

- Package organization
- Software distribution
- Documentation
- Tutorials

ROS Master

- Manages the communication between nodes
- Every node registers at startup with the master
- Start a master with:

```
> roscore
```

ROS Nodes

- Single-purpose, executable program
- Individually compiled, executed, and managed
- Organized in packages
- Run a node with:

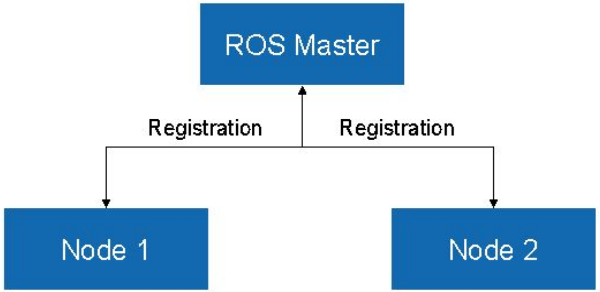
```
> rosrun package_name node_name
```

- See active nodes with:

```
> rosnode list
```

- Retrieve information about a node with:

```
> rosnode info node_name
```



ROS Topics

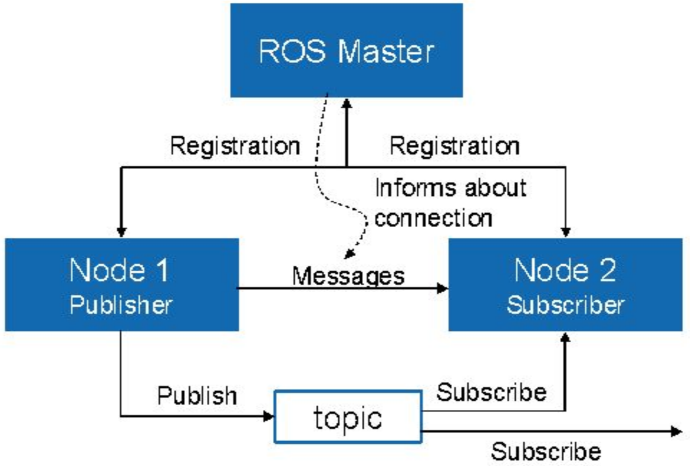
- Nodes communicate over topics
  - Nodes can publish or subscribe to a topic
  - Typically, 1 publisher and n subscribers
- Topic is a name for a stream of message
- List active topics with:

```
> rostopic list
```

- Subscribe and print the contents of a topic with:

```
> rostopic echo /topic
```

```
> rostopic info /topic
```



ROS Messages

- Data structure defining the type of a topic

- Compromised of a nested structure of integers, floats, booleans, strings and arrays of objects

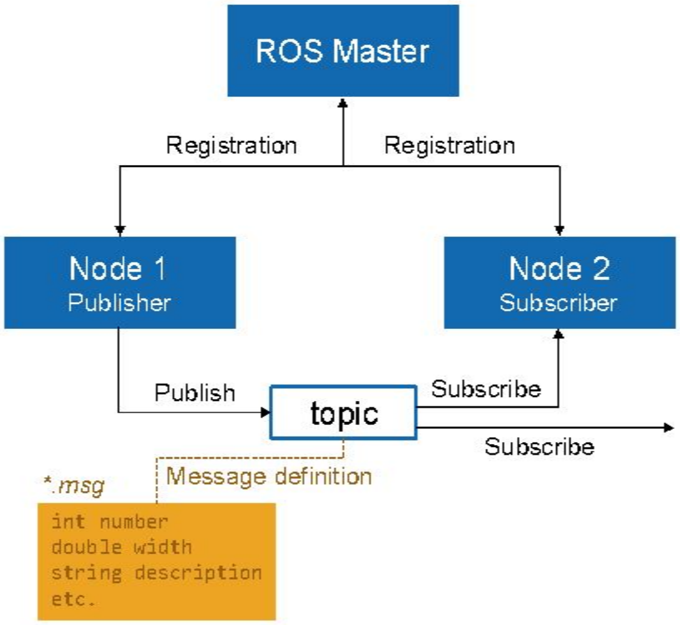
- Defined in *.msg files

- See the type of a topic:

```
> rostopic type /topic
```

- Publish a message to a topic:

```
> rostopic pub /topic type args
```



ROS Launch

- *Launch* is a tool for launching multiple nodes (as well as setting parameters)

- Written in XML as *.launch files

- If not yet running, launch automatically starts a roscore

- `> roslaunch file_name.launch` rt a launch f with:

  `> roslaunch package_name file_name.launch`

- Start a launch file from a package with:

Example console output for
`roslaunch roscpp_tutorials talker_listener.launch`

```
student@ubuntu:~/catkin_ws$ roslaunch roscpp_tutorials talker_listener.launch
... logging to /home/student/.ros/log/794321aa-e950-11e6-95db-000c297bd368/rosl
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://ubuntu:37592/

SUMMARY
========

PARAMETERS
 * /rosdistro: indigo
 * /rosversion: 1.11.20

NODES
  /
    listener (roscpp_tutorials/listener)
    talker (roscpp_tutorials/talker)

auto-starting new master
process[master]: started with pid [5772]
ROS_MASTER_URI=http://localhost:11311

setting /run_id to 794321aa-e950-11e6-95db-000c297bd368
process[rosout-1]: started with pid [5785]
started core service [/rosout]
process[listener-2]: started with pid [5788]
process[talker-3]: started with pid [5795]
[ INFO] [1486044252.537801350]: hello world 0
[ INFO] [1486044252.638886504]: hello world 1
[ INFO] [1486044252.738279674]: hello world 2
[ INFO] [1486044252.838357245]: hello world 3
```

ROS Launch

- File structure:

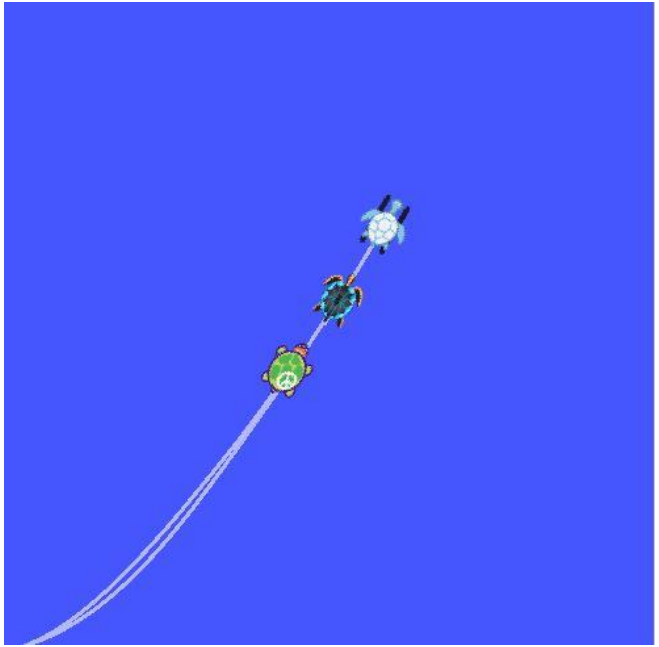
talker_listener.launch

```
<launch>
    <node name="listener" pkg="roscpp_tutorials" type="listener" output="screen"/>
    <node name="talker" pkg="roscpp_tutorials" type="talker" output="screen"/>
</launch>
```

! Notice the syntax difference for self-closing tags: <tag></tag> and <tag/>

- **launch**: Root element of the launch file
- **node**: Each *<node>* tag specifies a node to be launched
- **name**: Name of the node (free to choose)
- **pkg**: Package containing the node
- **type**: Type of the node, there must be a corresponding executable with the same name
- **output**: Specifies where to output log messages (screen: console, log: log file)

Objetivo de la práctica: Convoy



Primeros pasos (I)

- Descargar e instalar VMware Workstation Player (free): https://www.vmware.com/products/workstation-player/workstation-player-evaluation.html

- Descargar máquina virtual para la práctica con Ubuntu y ROS Melodic instalado: https://saco.csic.es/index.php/s/3F6bsNPHsGBnQQH

- Extraer la máquina virtual del .zip y abrirla con el VMware (en caso de que pregunte si la máquina virtual ha sido movida o copiada, seleccionar “I copied it”).

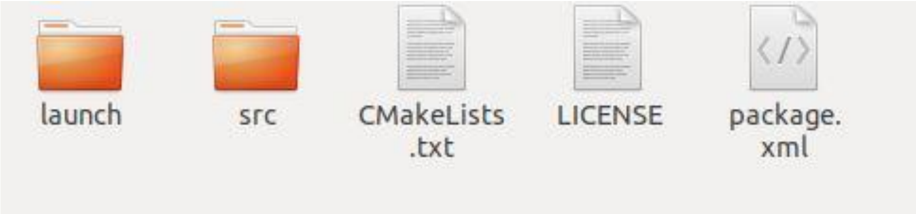
- Configurar teclado español: Region&Language: Input Sources + Spanish. Elegir es en barra de tareas

- Abrir terminal y escribir: echo "source ~/catkin_ws/devel/setup.bash" >> ~/.bashrc

Primeros pasos (II)

- Descargar paquete con código a completar:

```
user@ubuntu:~/catkin_ws/src$ git clone https://github.com/jmbengochea/convoy
```



Launch

```
user@ubuntu:~/catkin_ws$ roslaunch convoy convoy.launch
```

```
<launch>

    <node pkg="turtlesim" name="turtlesim" type="turtlesim_node" >
        <remap from="/turtle1" to="/robot1"/>
        </node>

        <node pkg="rosservice" type="rosservice" name="turtle2" args="call --wait /spawn 0.0 0.0 0.0 robot2" />
        <node pkg="rosservice" type="rosservice" name="turtle3" args="call --wait /spawn 0.0 0.0 0.0 robot3" />

    <node pkg="convoy" name="convoy_node" type="convoy_node" output="screen" >
        <param name="leader_pose" value="/robot1/pose" type="string" />
        <param name="follower_pose" value="/robot2/pose" type="string" />
        <param name="follower_speeds" value="/robot2/cmd_vel" type="string"/>
        </node>

    <node pkg="convoy" name="convoy2_node" type="convoy_node" output="screen" >
        <param name="leader_pose" value="/robot2/pose" type="string" />
        <param name="follower_pose" value="/robot3/pose" type="string" />
        <param name="follower_speeds" value="/robot3/cmd_vel" type="string"/>
        </node>

</launch>
```

ROS Launch

- Editar:

```
user@ubuntu:~/catkin_ws/src/convoy/src$ gedit convoy_node.cpp
```

- Compilar y construir:

```
user@ubuntu:~/catkin_ws$ catkin_make
```

- Controlar con teclado la primera tortuga del convoy:

```
user@ubuntu:~/catkin_ws$ rosrun turtlesim turtle_teleop_key /turtle1/cmd_vel:=/robot1/cmd_vel
```

Nodo (I)

```
class Convoy {
public:
    Convoy();

private:
    Pose Robot_Leader_Pose;
    Pose Robot_Follower_Pose;
    Subscriber Robot_Leader_Sub;
    Subscriber Robot_Follower_Sub;
    Publisher Robot_Follower_Command;
    NodeHandle Listener;
    NodeHandle CommanderNode;

    void Robot_Leader_PoseUpdate(const turtlesim::Pose::ConstPtr& msg);
    void Robot_Follower_PoseUpdate(const turtlesim::Pose::ConstPtr& msg);

    void trackLeader();
    float euclidean_distance();
    float linear_vel();
    float angle_vel();
    float steering_angle();

};
```

Nodo (II)

```
Convoy::Convoy(){

    string leader_pose, follower_pose, follower_speeds;
    ros::param::get("~leader_pose", leader_pose);
    ros::param::get("~follower_pose", follower_pose);
    ros::param::get("~follower_speeds", follower_speeds);

    Robot_Leader_Sub = Listener.subscribe(leader_pose, 10, &Convoy::Robot_Leader_PoseUpdate, this);
    Robot_Follower_Sub = //Suscribir a topic que publica la pose del robot follower
    Robot_Follower_Command = CommanderNode.advertise<geometry_msgs::Twist>(follower_speeds, 10);
}

void Convoy::Robot_Leader_PoseUpdate(const turtlesim::Pose::ConstPtr& msg)
{

    //De igual forma que en la callback Robot_Follower_PoseUpdate, actualizar la pose de Robot_Leader_Pose (en esta callback, obviamente, no se llama a trackLeader()

}

void Convoy::Robot_Follower_PoseUpdate(const turtlesim::Pose::ConstPtr& msg)
{
    //Actualizar la pose del robot follower con los valores contenidos en el mensaje msg recibido en el topic

    Robot_Follower_Pose.x =
    Robot_Follower_Pose.y =
    Robot_Follower_Pose.linear_velocity =
    Robot_Follower_Pose.angular_velocity =
    Robot_Follower_Pose.theta =

    trackLeader();
}
```

Nodo (III)

```
void Convoy::trackLeader(){

    Twist msg;

    if (euclidean_distance() >= distance_tolerance){
        msg.angular.z = angle_vel();

        if (abs(angle_vel()) > 1) msg.linear.x = 0;
        else msg.linear.x = linear_vel();
    }

    Robot_Follower_Command.publish(msg);
}

int main(int argc, char **argv)
{
    ros::init(argc, argv, "convoy");
    Convoy turtle;

    ros::Rate loop_rate(10);

    while (ros::ok())
    {
       ros::spinOnce();
       loop_rate.sleep();
    }

    return 0;
}
```

Nodo (controladores)

- Distancia euclídea: $\sqrt{(x_L - x_F)^2 + (y_L - y_F)^2}$

- Controlador velocidad lineal (proporcional): $K_{lv} \cdot distancia\_euclídea$

- Ángulo de giro: $tan^{-1} \frac{(y_L - y_F)}{(x_L - x_F)}$ (usar atan2)

- Controlador velocidad angular (proporcional): $K_{av} \cdot (ángulo\_giro - \theta_F)$