UNIVERSITÀ di VERONA | Dipartimento di INFORMATICA

LTAIR robotics lab

# Laboratorio ciberfisico (2018/2019): ROS 2

Diego Dall'Alba

Altair robotics lab
Department of computer science – University of Verona, Italy

# Sommario di oggi

- Approfondimenti sulla struttura di un package ROS
- Comandi bash di ROS
- Concetto di Nodo e Topic Ros
- Architettura di comunicazione
- Tipi di messaggi
- Comandi ROS per gestire nodi, topic e massaggi.

- Esempi pratici ☺

# Nelle puntate precedenti...

- Intallazione di ROS Melodic su Linux Ubuntu Bionic
- Approfondimento **configurazione «ambiente» ROS**: variabili di ambiente, cosa si nasconde dietro questo comando:

```
source /opt/ros/melodic/setup.bash
```

- Introduzione ai build system con particolare attenzione a **CMake**
- Descrizione del build system di ROS: **CATKIN**
- Concetto di ROS package and Catkin workspace



# Configurazione preliminare

- Se non lo avete già fatto automatizzate la configurazione dell'ambiente ROS (aggiungete il source … al vostro .bashrc):

```
echo "source /opt/ros/melodic/setup.bash" >> ~/.bashrc
source ~/.bashrc
```

- Verificate di aver effettuato correttamente la configurazione aprendo un nuovo terminale e verificando se potete accedere direttamente ai comandi ROS (roscore ad esempio)

# Struttura di package catkin

Un package catkin deve contenere:

- Un file «manifesto» package.xml compatibile con catkin, questo file fornisce meta-informazioni relative al package e vincoli/dipendenze da gestire in fase di build
- Un file di configurazione CMakeLists.txt sempre basato su catkin.
- Ogni package deve avere la sua directory specifica
  → Questo significa che non posso avere più packages che condividono la stessa cartella e neppure avere sotto-packages (sottocartelle all'interno di un packages)

```
my_package/
        CMakeLists.txt
        package.xml
```

Vedremo che invece più nodi (i.e. eseguibili) potranno essere contenuti in un package

# Struttura tipica source space in un catkin workspace

```
workspace_folder/          --  CATKIN WORKSPACE
  └─src/                       -- SOURCE SPACE
      ├─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
```

# Struttura di package.xml (1)

Un file package.xml deve soddisfare la struttura base avente come root-tag del documento xml <package>

```
<package>

</package>
```

Ci sono poi un set di tag obbligatori che deveono essere contenuti nel root-tag:

- **<name>** il nome del package
- **<version>** il numero di versione (deve essere nel formato 3 interi separati da punti, ad esempio 1.2.3)
- **<description>** una descrizione del package
- **<maintainer>** Il nome di almeno una persona responsabile dello sviluppo e "mantenimento" del package
- **<license>** La licenza software (e.g. GPL, BSD, ASL) sotto la quale il software viene rilasciato.

# Struttura minima di package.xml

```
<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>

 <buildtool_depend>catkin</buildtool_depend>
</package>
```

# Struttura di package.xml (2)

I packages possono avere 4 tipi di dipendenze che non sono specificate nei tag precedenti

<buildtool_depend> **Build Tool Dependencies**
<build_depend> **Build Dependencies**
<run_depend> **Run Dependencies**
<test_depend>**Test Dependencies**

Per maggiori dettagli si veda
http://wiki.ros.org/catkin/conceptual_overview#Dependency_Management

# Struttura di package.xml (3)

**Build Tool Dependencies** specificano i tool di build necessari per compilare il package. Di solito l'unico tool richiesto è catkin.

Se stiamo cross-compilando per un'altra architettura può essere necessario specificare tool aggiuntivi (specifici dell'architettura target)

**Build Dependencies** specifica quali package sono necessari per compilare il package corrente. Questo significa che abbiamo bisogno di usare un file presente in un altro package, ad esempio: includere file header o librerie da altri packages.

Se stiamo cross-compilando, queste dipendenze saranno relative all'architettura target

# Struttura di package.xml (4)

**Run Dependencies** permette di specificare dipendenze necessari per eseguire il package. E' il tipico caso in cui dipendiamo da una Libreria dinamica di Sistema o messa a disposizione da qualche altro package

**Test Dependencies** dipendenze extra da soddisfare per il testing. Non devono duplicare dipendenze di build e run.

# Struttura realistica di package.xml

```
<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>
```

**Vedi slide precedente**

```
 <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>
```

Vedremo più avanti come modificare questo file ed eventualmente la configurazione di CMake

# ROS bash commands

Ros ci mette a disposizione alcuni convenienti comandi shell per gestire funzioni di uso comune:

- **rospack** ci permette di avere informazioni relative ai packages installati (permette di controllare dipendenze tra packages)
- **roscd** ci permette di spostarci tra diversi packages (senza conoscere il path in cui essi si trovano)
- **rosls** ci permette di elencare i file di un package
- **rosrun** ci permette di lanciare un nodo (= eseguibile) contenuto in un package

Permettono di astrarre il filesystem del sistema operativo in cui sono installati i packages

TUTTI I COMANDI SUPPORTANO LA TAB-COMPLETION

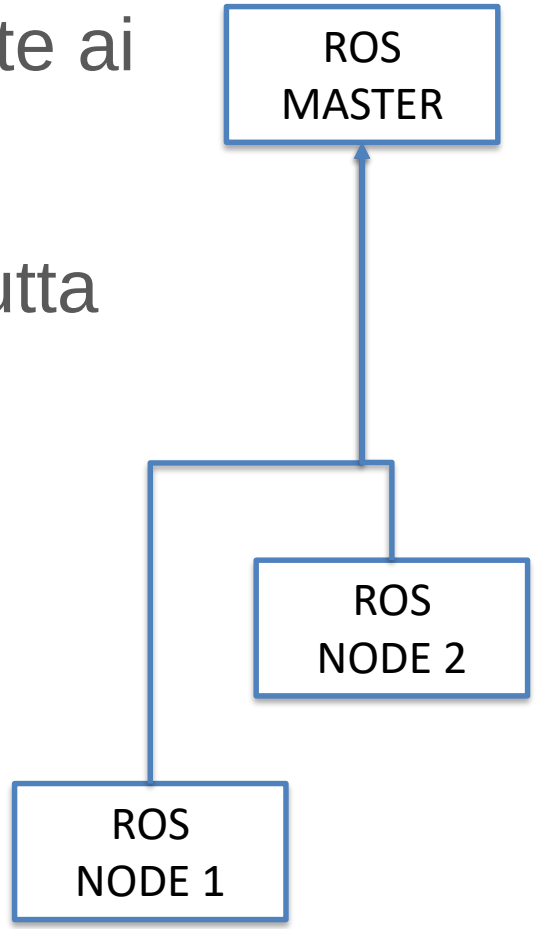
per maggiori dettagli vedi http://wiki.ros.org/rosbash

# ROS Master, Nodi e dintorni

**Master:** Name service di ROS (permette ai nodi di “trovarsi” tra loro)

**Nodi:** Un nodo è un eseguibile che sfrutta ROS per comunicare con altri nodi.
**rosout:** L’equivalente in ROS di stdout/stderr (utile per logging)

**roscore:** Master + rosout + parameter server (introdurremmo più avanti il parameter server)



# Comandi ROS: rosnode e rosrun

Il comando rosnode ci permette di avere informazione su noi in esecuzione, ad esempio:

```
rosnode list
```

Ci permette di avere la lista dei nodi attualmente in esecuzione

```
rosnode info [nome_nodo]
```

Ci permette di avere informazioni più dettagliate su uno specifico nodo

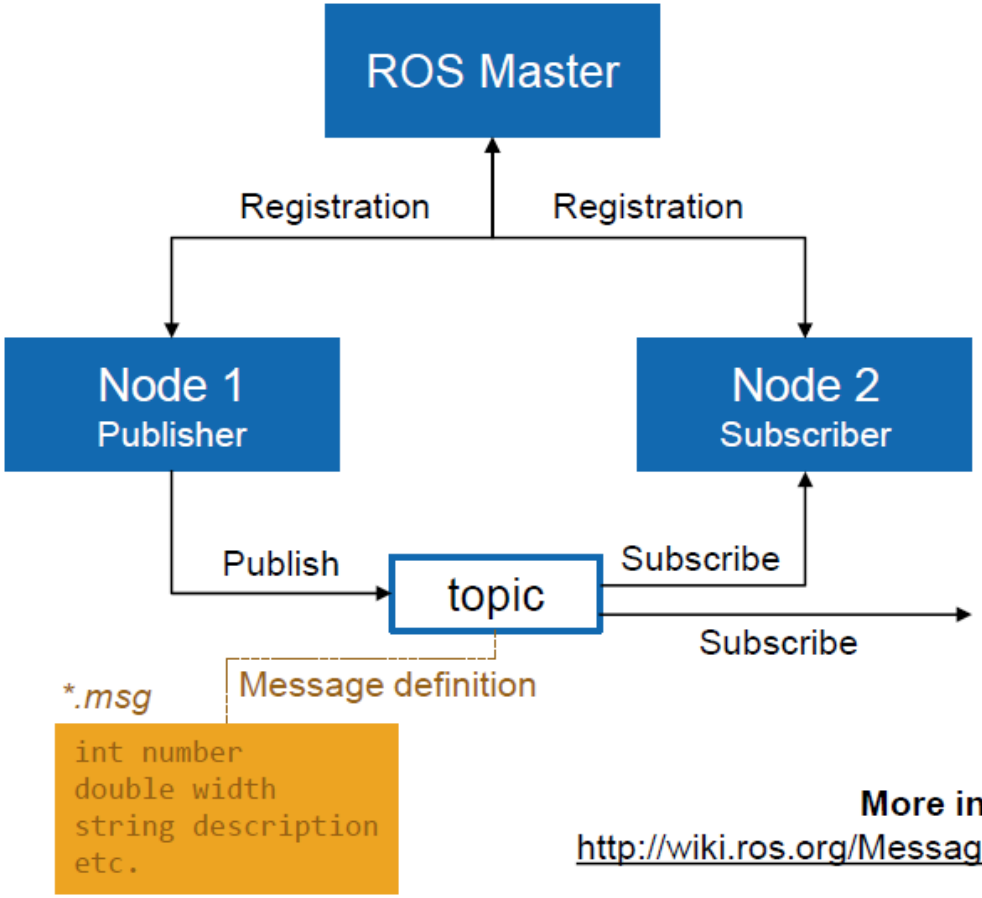
```
rosrun [nome_package] [nome_nodo]
```

Ci permette di eseguire un nodo contenuto in un package

# ROS Topic e Message

**Topic:** I nodi ROS possono pubblicare Messaggi in un Topic e possono anche sottoscriversi a un Topic per ricevere Messaggi.

**Messaggi:** sono tipi di dati ROS (di alto livello) che definiscono il tipo/formato dei dati scambiati attraverso Topic



# Esempio di ROS Message

*geometry_msgs/Point.msg*

```
float64 x
float64 y
float64 z
```

*sensor_msgs/Image.msg*

```
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
uint32 height
uint32 width
string encoding
uint8 is_bigendian
uint32 step
uint8[] data
```

*geometry_msgs/PoseStamped.msg*

```
std_msgs/Header header
 uint32 seq
 time stamp
 string frame_id
geometry_msgs/Pose pose
  geometry_msgs/Point position
    float64 x
    float64 y
    float64 z
  geometry_msgs/Quaternion orientation
    float64 x
    float64 y
    float64 z
    float64 w
```

# Comandi ROS: rostopic

Il comando rostopic ci permette di avere informazione sui topic pubblicati dai nodi in esecuzione:

```
rostopic -h
```

Ci permette di avere la lista dei comandi supportati

```
rostopic list -v
```

Ci permette di conoscere il topic che sono al momento sottoscritti e pubblicati (-v per avere modalità verbose)

```
rostopic echo [topic_name]
```

Ci permette di vedere i dati pubblicati su un topic specifico, in realtà ci siamo implicitamente sottoscritti al topic

# Comandi ROS: informazioni sul topic

Il comando rostopic ci permette di avere informazioni su uno specifico topic con il comando:

```
rostopic type [topic_name]
```

Possiamo poi avere maggiori informazioni sul tipo di dato specifico utilizzando:

```
rosmsg show [tipo_dato]
```

Possiaomo concatenare i due comandi precedenti per avere direttamente informazioni sui tipi di dati pubblicati:

```
rostopic type [topic_name] | rosmgs show
```

# Comandi ROS: rostopic pub

Possiamo fare in modo di pubblicare su un specifico topic con il seguente comando:

```
rostopic pub [topic_name] [tipo_messaggio] [argomenti]
```

Questo comando permette di pubblicare solo su un topic già disponibile (e visibile facendo rostopic list).

Vedremo come creare un nuovo nodo con relativi topic nelle prossime lezioni…

**Vediamo di applicare quanto visto in un esempio interattivo**

# Istruzioni esercizi

```
sudo apt-get install ros-melodic-ros-tutorials
```

Ci permette di installare dei packages con degli esempi che ci torneranno utili per fare delle prove.
Facciamo poi partire il ROS core e facciamo un po' di pratica con i comandi bash introdotti:

- rospack, roscd, rosls
- rosnode e rostopic (cercando ci caratterizzare meglio rosout)

Proviamo poi il comando rosrun:

```
rosrun turtlesim turtlesim_node
```

# Istruzioni esercizi (2)

```
rosrun turtlesim turtle_teleop_key
```

Ci permette di far partire un nodo che ci permette di muovere la turtle nel simulatore precedente.
Cerchiamo di capire l'architettura con cui comunicano i due nodi (chi pubblica cosa e chi si sottoscrive a cosa). Cerchiamo di capire come viene mossa la tartaruga
Proviamo poi il comando:

```
rostopic pub -1 /turtle1/cmd_vel geometry_msgs/Twist
-- '[2.0, 0.0, 0.0]' '[0.0, 0.0, 1.8]'
```

Proviamo a cambiare i parametri per capire cosa alterano…

# Fine lezione

Approfondimenti:

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

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