



UNIVERSITÀ
di **VERONA**

Dipartimento
di **INFORMATICA**



Robotica 2019/2020: Introduzione (rapida) a ROS

Diego Dall'Alba

Altair robotics lab

Department of computer science – University of Verona, Italy

Sommario di oggi

- Cenni di architettura di ROS
- La configurazione dell'ambiente
- Build system: Il processo di compilazione con catkin
- Approfondimenti sulla struttura e la gestione 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 messaggi.
- Client Libraries e implementazione nodi



Nelle puntate precedenti...

- **Distribuzione ROS:** Uno specifico insieme di pacchetti «versionato» (con specifiche versione dei pacchetti), legato a una specifica distribuzione Linux per la gestione delle componenti di sistema
- Come orientarsi con le versioni di ROS e relative versioni di Ubuntu
- **Assumo che tutti voi abbiate un'installazione di ROS Melodic (su Ubuntu Bionic) Funzionante**

Applications

ROS

Operating System
(Linux Ubuntu)

ROS Melodic Morenia

Released May, 2018

Latest LTS, supported until May, 2023



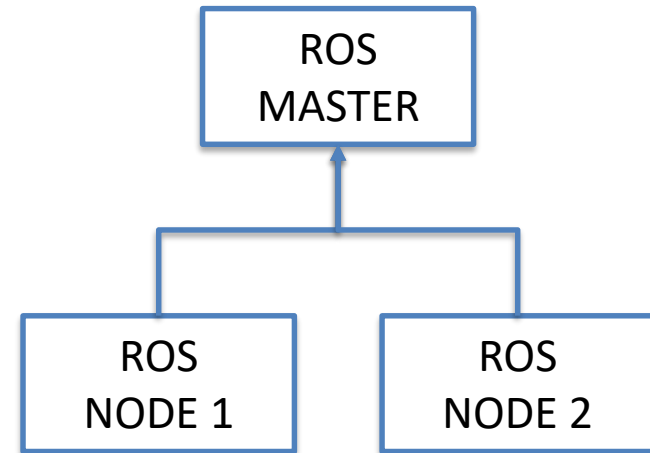
Architettura di ROS

ROS Master:

- Si occupa di gestire la comunicazione tra nodi (processi)
- Ogni nodo, quando viene avviato, si registra con il master
- I nodi possono essere in esecuzione su diversi computer e possono quindi comunicare attraverso la rete.

ROS Nodes:

- Un programma eseguibile con una funzione specifica
- Compilato, eseguito e gestito in modo indipendente
- I nodi sono organizzati (raggruppati) in packages



Configurazione dell'ambiente

Inizializziamo le variabili di ambiente di ROS (solo per la shell corrente):

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

Comando fondamentale per poter lanciare il ROS master:

```
roscore
```

In realtà il comando roscore non lancia solo il master ma anche dei servizi fondamentali per gli altri nodi

```
es Terminal ven 14:55
roscore http://victors:11311/
File Edit View Search Terminal Help
ai-ray@victors:~$ roscore
... logging to /home/ai-ray/.ros/log/4699893e-522a-11e9-ad61-
unch-victors-2205.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://victors:41423/
ros_comm version 1.14.3

SUMMARY
=====

PARAMETERS
* /rostdistro: melodic
* /rosversion: 1.14.3

NODES

auto-starting new master
process[master]: started with pid [2216]
ROS_MASTER_URI=http://victors:11311/

setting /run_id to 4699893e-522a-11e9-ad61-0800271b6865
process[rosout-1]: started with pid [2227]
started core service [/rosout]
```

Configurazione dell'ambiente

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

Lo scopo di questo comando è configurare le variabili di ambiente in modo da rendere accessibili i comandi di ROS e da permettere il corretto funzionamento delle diverse componenti

```
ai-ray@victors: ~  
File Edit View Search Terminal Help  
ai-ray@victors:~$ source /opt/ros/melodic/setup.bash  
ai-ray@victors:~$ printenv | grep -e ros -e ROS  
LD_LIBRARY_PATH=/opt/ros/melodic/lib  
ROS_ETC_DIR=/opt/ros/melodic/etc/ros  
CMAKE_PREFIX_PATH=/opt/ros/melodic  
ROS_ROOT=/opt/ros/melodic/share/ros  
ROS_MASTER_URI=http://localhost:11311  
ROS_VERSION=1  
ROS_PYTHON_VERSION=2  
PYTHONPATH=/opt/ros/melodic/lib/python2.7/dist-packages  
ROS_PACKAGE_PATH=/opt/ros/melodic/share  
ROSLISP_PACKAGE_DIRECTORIES=  
PATH=/opt/ros/melodic/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/bin:/usr/games:/usr/local/games:/snap/bin  
PKG_CONFIG_PATH=/opt/ros/melodic/lib/pkgconfig  
ROS_DISTRO=melodic  
ai-ray@victors:~$
```

<http://wiki.ros.org/ROS/Tutorials/InstallingandConfiguringROSEnvironment>

Variabili di ambiente in ROS

Le variabili di ambiente forniscono funzioni utili per il corretto funzionamento di ROS, le principali:

1. **Trovare i pacchetti**
2. **Influenzare l'esecuzione di un nodo**
3. **Modificare le impostazioni del “Build system”**

Molte di queste variabili (ma non tutte) possono essere sovrascritte e impostate da un nodo,

In **ROSSO** verranno segnalate in seguito le variabili fondamentali da impostare.

Per maggiori dettagli: <http://wiki.ros.org/ROS/EnvironmentVariables>

Trovare i pacchetti

ROS_ROOT e ROS_PACKAGE_PATH permettono a ROS di trovare i pacchetti nel filesystem del Sistema operativo Linux.

```
export ROS_ROOT=/home/user/ros/ros  
export PATH=$ROS_ROOT/bin:$PATH
```

Attenzione! DEVE essere settata anche la variabile **PYTHONPATH** per permettere a interprete python di trovare le librerie ROS.

```
export PYTHONPATH=$PYTHONPATH:$ROS_ROOT/core/roslib/src
```

Questa variabile va settata anche nel caso non si utilizzi python per scrivere codice, in quanto questo linguaggio è usato durante il processo di compilazione ed esecuzione dei nodi.

Influenzare l'esecuzione di un nodo

- **ROS_MASTER_URI** è importante per dire ai nodi “dove si trova” il MASTER.

```
export ROS_MASTER_URI=http://victors:11311/
```

- ROS_LOG_DIR permette di impostare la directory dove vengono salvati i file di log.
- ROS_IP e ROS_HOSTNAME permettono di modificare le impostazioni di rete di un nodo (in configurazioni con interfacce di rete multiple, non molto comuni)
- ROS_NAMESPACE permette di cambiare il namespace di un nodo (permette di risolvere situazioni di nomi di nodi duplicati)

Modificare le impostazioni del “Build system”:

Le seguenti variabili modificano le impostazioni per trovare librerie, su come avviene il processo di compilazione e su quali component escludere dalla compilazione:

- ROS_BINDEPS_PATH,
- ROS_BOOST_ROOT
- ROS_PARALLEL_JOBS
- ROS_LANG_DISABLE

Possono tornare utili per compilare packages non scritti da noi, oppure quando compiliamo codice su piattaforme emebedded con vincoli di memoria, supporto per specifiche librerie ecc...

Un passo indietro: “Build system”

Un «build system» (build automation software) è un insieme di strumenti software che mirano ad automatizzare il processo di compilazione del codice sorgente e la relativi test e distribuzione degli eseguibili.

Le principali funzioni svolte sono:

- compilazione del codice sorgente in codice binario (gestione dipendenze e impostazioni compilatore)
- pacchettizzazione del codice binario (gestione dipendenze a runtime)
- esecuzione di test (esempio unit o functional test automatico)
- deployment di sistemi di produzione (installazione)
- creazione di documentazione e/o note di rilascio (generazione automatica a partire dai commenti nel codice)

Perchè utilizzare un “Build system” ?

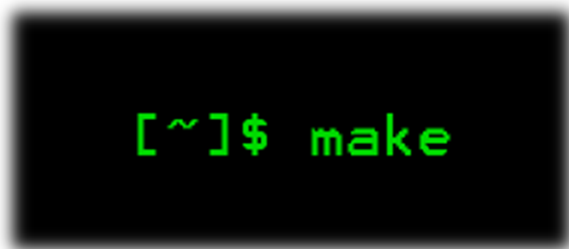
- Avete mai provato a scrivere un programma multi-piattaforma?
- Avete mai compilato da sorgenti una libreria o meglio ancora un framework?
- Vi siete mai trovati a dover distribuire un eseguibile di un vostro programma (anche non cross-platform)?
- Avete mai dovuto testare seriamente per trovare bug o verificare l'aderenza alle specifiche di un software da voi sviluppato?
- Avete mai scritto della documentazione per un programma complesso o una libreria scritta da voi?

Queste sono solo alcune delle motivazioni per utilizzare un build system

Se dovete scrivere codice su una singola piattaforma che non deve essere distribuito ad altri probabilmente non vi serve un build system (tipico caso degli elaborati per corsi universitari)

Esempi di “Build system”

- Esistono numerosi «build system», alcuni dei quali nascosti all'interno di IDE comunemente utilizzati.
- Non tutti svolgono tutte le funzioni elencate in precedenza, appoggiandosi in caso a strumenti esterni (per la generazione della documentazione o il testing)
- Esistono degli standard de-facto per ciascun OS principale:



LINUX



OSX



WIN

Altri “Build system”

Esistono anche altri build system ad esempio:

- Integrati con IDE cross platform: come ad esempio Eclipse o Codeblocks, che utilizzano dietro le quinte strumenti da line di comando (come make).
- Che fanno parte di framework cross-platform, come ad esempio BOOST (boost.build) o Qt (qmake)
- Che sono stati sviluppati per specifici linguaggi o progetti, ad esempio Apache Ant



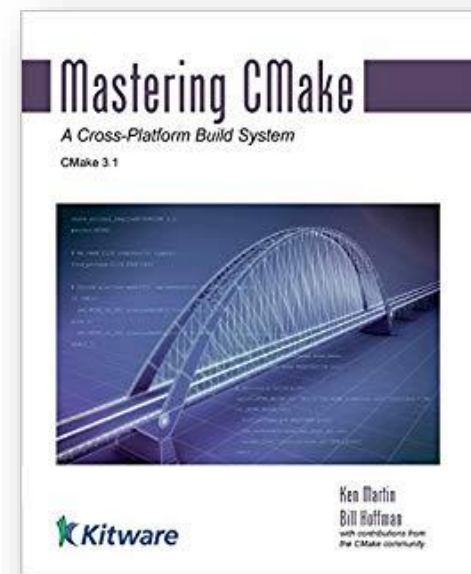
Cmake

Rispetto agli esempi precedenti Cmake può essere considerato come un «meta build tool»:

a partire da un file di configurazione «CMakeLists.txt» configura l'ambiente di compilazione «nativa» di ciascuna piattaforma.

Cmake supporta molte piattaforme, «build tool nativi» e funzionalità avanzate, vedi documentazione ufficiale su:

<https://cmake.org/>

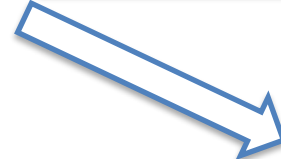
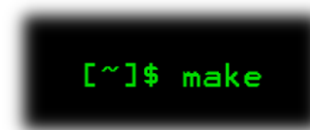


Oltre 300 pagine!!!

Cmake workflow



CMake



File config.

CMakeLists.txt

Il tipico build workflow con Cmake su piattaforma Linux è il seguente:

- Invoco cmake per configurare correttamente i parametri di build
- Invoco make per effettuare la compilazione vera e propria
- Invoco make install per effettuare installazione (facoltativo)

Su Windows Cmake di solito genera un progetto VisualStudio che poi verrà compilato all'interno dell'IDE.

Struttura CMakeLists.txt

```
cmake_minimum_required(VERSION 2.8)
project(app_project)
add_executable(myapp main.c)
install(TARGETS myapp DESTINATION bin)
```

```
cmake_minimum_required(VERSION 2.8)
project(libtest_project)
add_library(test STATIC test.c)
install(TARGETS test DESTINATION lib)
install(FILES test.h DESTINATION include)
```

```
cmake_minimum_required(VERSION 2.8)
project(myapp)
add_subdirectory(libtest_project)
add_executable(myapp main.c)
target_link_libraries(myapp test)
install(TARGETS myapp DESTINATION bin)
```

Il concetto di build target è legato a qualsiasi risultato della compilazione:

- Eseguibili
- Librerie statiche
- Librerie dinamiche

Il file CMakeLists.txt ha una sua struttura e supporta un linguaggio di scripting specifico (dipende dalla versione)

L'unico file che deve essere modificato dall'utente, mai alterare un Makefile generato con cmake

I veri CMakeLists.txt

```
ExternalProject_Add(project_luajit
  URL http://luajit.org/download/LuaJIT-2.0.1.tar.gz
  PREFIX ${CMAKE_CURRENT_BINARY_DIR}/luajit-2.0.1
  CONFIGURE_COMMAND ""
  BUILD_COMMAND make
  INSTALL_COMMAND make install
  PREFIX=${CMAKE_CURRENT_BINARY_DIR}/luajit-2.0.1
)
ExternalProject_Get_Property(project_luajit install_dir)
add_library(luajit STATIC IMPORTED)
set_property(TARGET luajit PROPERTY IMPORTED_LOCATION
  ${install_dir}/lib/libluajit-5.1.a)
add_dependencies(luajit project_luajit)
add_executable(myapp main.c)
include_directories(${install_dir}/include/luajit-2.0)
target_link_libraries(myapp luajit)
```

utilizzando ROS
sarà necessario
modificare i file
CMakeLists.txt
preparati dal build
system di ROS
(catkin)

Molti problemi
frequenti con ROS
sono legati ad un
uso errato di
Cmake e in
generale del build
system

Catkin build system




- Catkin è il build system di ROS a partire dalla versione Groovy, ha sostituito rosbUILD
- Catkin (e il predecessore rosbUILD) è basato su Cmake e vari tool di supporto per estenderne le funzionalità e gestire l'eterogeneità/complessità pacchetti ROS.
- La differenza principale consiste nell'utilizzo di python al posto di shell-scripting per gli script di supporto, questo dovrebbe migliorare il processo e renderlo cross-platform.
- Il nome è legato al fiore della pianta Willow (un riferimento a Willow Garage)

**CMake**

Catkin workspace (1)

- I packages catkin per il momento possono essere visti come un sinonimo dei packages di ROS (= insieme di nodi ros)
- I packages catkin possono essere compilati come progetti indipendenti, allo stesso modo di quanto si fa con Cmake
- Può tornare molto utile lavorando in ROS, compilare più packages indipendenti in un singolo processo
- Per supportare questa funzionalità sono stati introdotti i catkin workspaces
- Ricordiamoci che ROS è utilizzato da una comunità eterogenea, molti degli utilizzatori non hanno un forte background informatico (Engineering 😊)

Catkin workspace (2)

- Forniscono una struttura standard con cui organizzare un insieme di packages catkin
 - Permettono la compilazione utilizzando Cmake, gestendo dipendenze complesse
 - Un workspace catkin può contenere 4 (5) spazi:
 - Source Space
 - Build Space
 - Devel Space
 - Install Space
 - (Log Space)
- Result Space
- 

Si tratta sostanzialmente di un albero di directory

Source space

- Il source space contiene il codice sorgente dei catkin packages.
- In questo spazio si deve trovare il codice sorgente dei pacchetti catkin che vogliamo compilare
- Può contenere il codice sorgente di più packages in una singola directory



src

Build space

- Il build space è dove CMake viene invocato per compilare i catkin packages che si trovano nel source space.
- CMake and catkin mantengono in questo spazio le informazioni di cache e altri file intermedi.
- Il build space non deve per forza essere contenuto nel workspace corrente e neppure deve essere in una directory separate rispetto al source space
- **Anche se non è obbligatorio, è caldamente consigliato mantenere build e source space separati e all'interno del workspace corrente.**



Development space (Devel space)

- Il development space (o più comunemente devel space) è dove gli eseguibili sono messi prima di essere installati
- Gli eseguibili (e i relative file di supporto) sono organizzati all'interno del devel space nello stesso modo di come saranno installati.
- Questo spazio è molto utile per testare durante lo sviluppo, senza bisogno di invocare il passaggio di installazione.
- Una volta compilati gli eseguibili, essi possono essere installati nel install space (di solito con make install).
- The install space non deve (e solitamente non lo è) essere contenuto all'interno del workspace corrente.



Catkin workspace (3)

Work Here



src

Don't Touch



build

Don't Touch



devel

- Prima di poter creare il nostro primo workspace dobbiamo installare i tool catkin:

```
sudo apt-get install python-catkin-tools
```

- Quando usate ros sostituite eventuali catkin_AZIONE con l'equivalente comando catkin SPAZIO AZIONE',
- Esempio: catkin_make → catkin build
- **Non mescolare mai le due tipologie di comandi!**

Esempio creazione nuovo workspace

```
source /opt/ros/melodic/setup.bash
mkdir -p /tmp/quickstart_ws/src          # Make a new workspace
cd /tmp/quickstart_ws                    # Navigate to the workspace root
catkin init                              # Initialize it
cd /tmp/quickstart_ws/src                # Navigate to the source space
catkin create pkg pkg_a                  # Populate the source space
catkin create pkg pkg_b
catkin create pkg pkg_c --catkin-deps pkg_a
catkin create pkg pkg_d --catkin-deps pkg_a pkg_b
catkin list                              # List the packages in the workspace
catkin build                             # Build all packages in the workspace
source /tmp/quickstart_ws/devel/setup.bash
```

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 devono 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
- **rosls** 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>

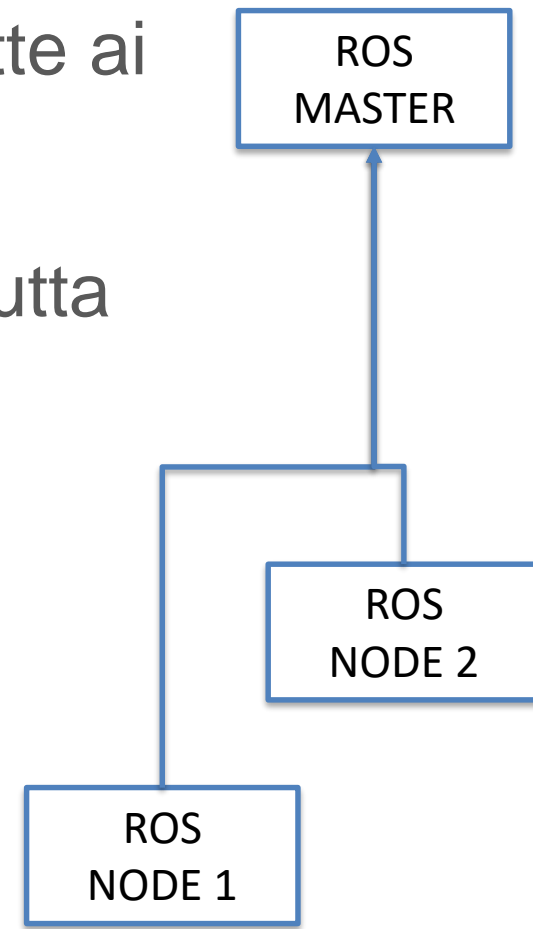
ROS Master, Nodi e dintorni

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:

```
rosgnode list
```

Ci permette di avere la lista dei nodi attualmente in esecuzione

```
rosgnode info [nome_nodo]
```

Ci permette di avere informazioni più dettagliate su uno specifico nodo

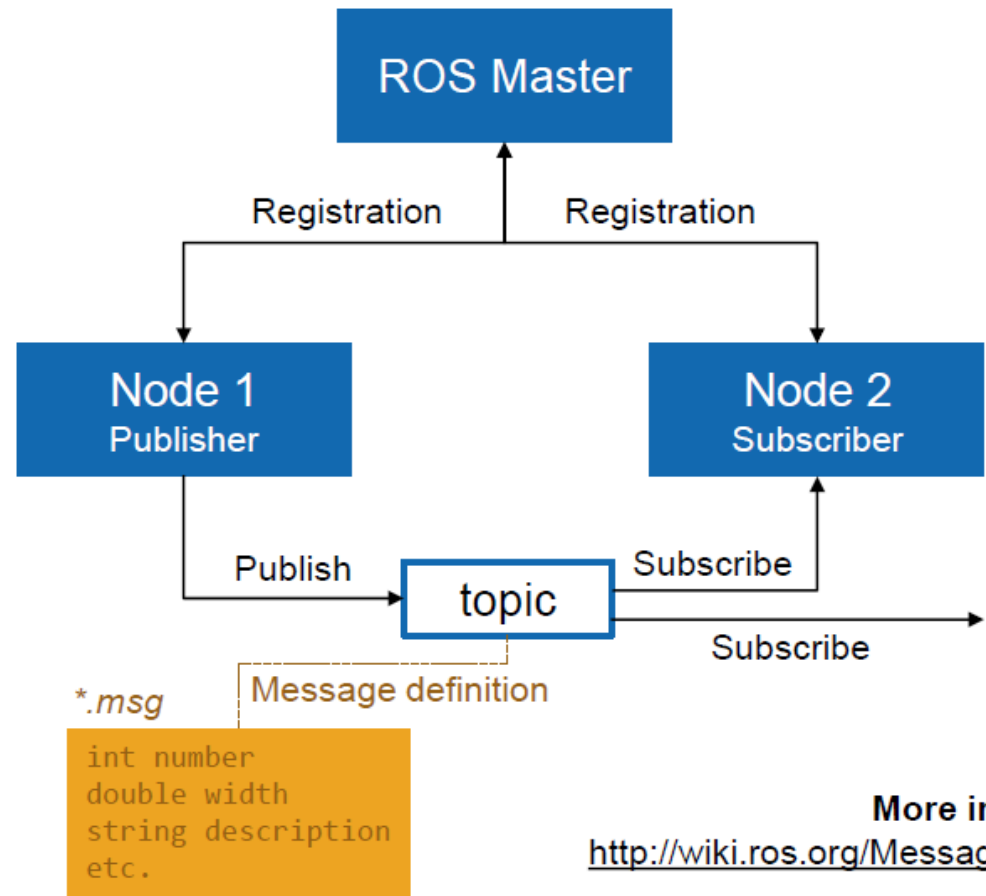
```
rosgrun [nome_package] [nome_nodo]
```

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 sottoscrivere 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:

```
rosmmsg show [tipo_dato]
```

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

```
rostopic type [topic_name] | rosmmsg 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
- rosnod e rostopic (cercando ci caratterizzare meglio rosout)

Proviamo poi il comando rosrn:

```
rosrn turtlesim turtlesim_node
```

Istruzioni esercizi (2)

```
roslaunch turtlesim turtlesim_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...

Client Libraries (1)

Le ROS client libraries permettono a nodi scritti in diversi linguaggi di programmazione di comunicare e sfruttare le funzionalità messe a disposizione dal middleware:

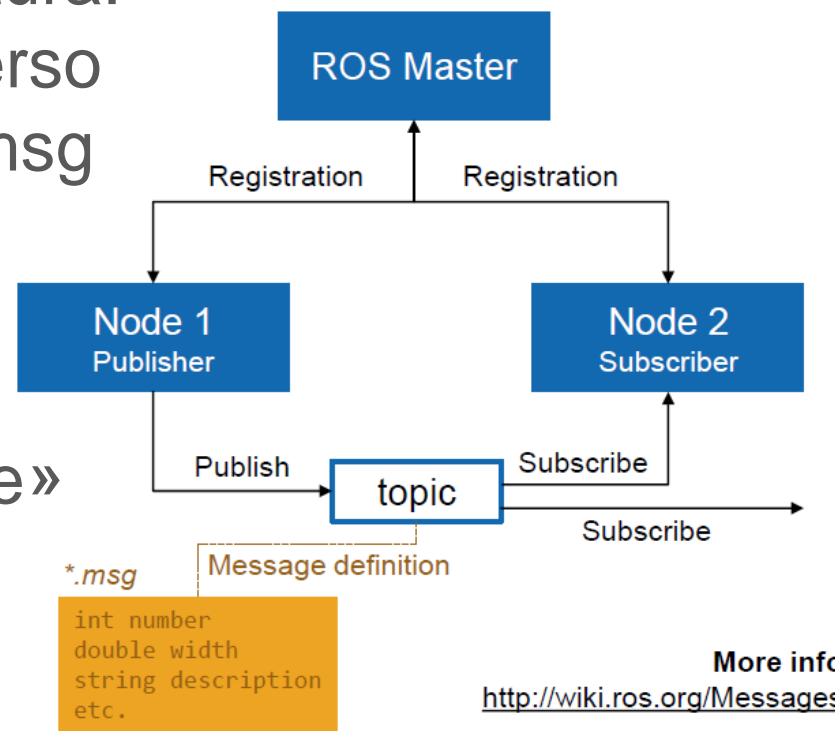
- **rospy** = python client library
<https://wiki.ros.org/rospy>
- **roscpp** = c++ client library
<http://wiki.ros.org/roscpp>



Esercitazione

Implementeremo questa architettura:

- 2 nodi che comunicano attraverso un topic per lo scambio di un msg
- Un nodo «scriverà» le info pubblicandole sul topic
- L'altro nodo si sottoscriverà (allo stesso topic) per «leggerle»
- Vedremo come i diversi passi rappresentati si «mappano» in funzioni e comandi forniti dalle client libraries



Client Libraries (2)

- Siete liberi di utilizzare il linguaggio che preferite per lo svolgimento delle esercitazioni.
- Nella maggior parte dei casi sarà sufficiente l'utilizzo di python
- Le client libraries sono molto potenti, vedremo solo una minima parte delle funzionalità
- Roscpp tutorial:
http://wiki.ros.org/roscpp_tutorials/Tutorials/WritingPublisherSubscriber
- Rospy tutorial:
<http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29>



Pre-requisiti e Passi preliminari

- Installazione funzionante (nativa o su VM) di ROS Melodic su Ubuntu Bionic (18.04)
- «Ambiente ROS» configurato correttamente
- Un editor di testo per il codice
- Consiglio Visual Studio Code visto che fornisce una buona integrazione con ROS, vedi ad esempio:

<https://medium.com/@tahsincankose/a-decent-integration-of-vscode-to-ros-4c1d951c982a>

<https://erdalpekel.de/?p=157>

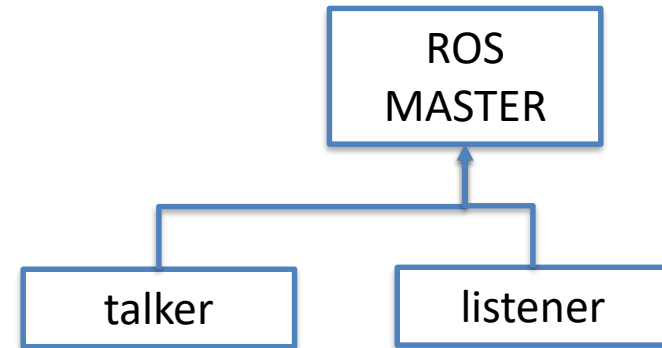
Passi principali

- Scaricate il codice sorgente di talker e listener:

https://raw.githubusercontent.com/ros/ros_tutorials/kinetic-devel/roscpp_tutorials/talker/talker.cpp

https://raw.githubusercontent.com/ros/ros_tutorials/kinetic-devel/roscpp_tutorials/listener/listener.cpp

- Modificare il file CMakeLists.txt
- Provare a compilare il codice (con catkin)
 - in caso di errori provare a risolverli e ritentare la compilazione
- Provare a lanciare i nodi (roslaunch) e verificarne il funzionamento con i comandi visti (rostopic, roscore, ecc...)



Elaborato R1 – Parte A (Base)

- Avere una installazione funzionante di ROS Melodic (anche su macchina virtuale)
- Riuscire a utilizzare i comandi principali di ROS (vi potrà essere richiesto di utilizzare un comando specifici durante la consegna dell'elaborato)
- Riuscire a far funzionare il listener-talker appena descritto.



```
ven 14:35
roscore http://victors:11311/

File Edit View Search Terminal Help
al-ray@victors:~$ roscore
... logging to /home/al-ray/.ros/log/4699893e-522a-11e9-ad61-
unch-victors-2285.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://victors:41423/
ros_comm version 1.14.3

SUMMARY
=====
PARAMETERS
 * /roscpp: melodic
 * /rosversion: 1.14.3

NODES
auto-starting new master
process[master]: started with pid [2216]
ROS_MASTER_URI=http://victors:11311/

setting /run_id to 4699893e-522a-11e9-ad61-0800271b6865
process[roscout-1]: started with pid [2227]
started core service [/roscout]
```

Elaborato R1 – Parte B (ROS)

Scopo: Creare un nodo ROS che faccia muovere la tartaruga in turtlesim lungo una circonferenza.

Passi principali:

1. Creare un nuovo package ROS nel vostro workspace catkin, dovrà avere le giuste dipendenze
2. Creare un nuovo nodo, a partire da listener o talker visti a lezione
3. Modificare il codice in modo da pubblicare o sottoscrivere correttamente al topic per controllare il turtlesim
4. Compilare il codice e verificare il funzionamento.

