



UNIVERSITÀ  
di **VERONA**

Dipartimento  
di **INFORMATICA**



# Laboratorio ciberfisico (2018/2019): ROS 1

Diego Dall'Alba

Altair robotics lab

Department of computer science – University of Verona, Italy

# Sommario di oggi

- Valutazione Elaborato 2.0:
- Cenni di architettura di ROS
- La configurazione dell'ambiente
- La gestione dei packages
- Il processo di compilazione con catkin
- Assumo che tutti voi abbiate un'installazione di ROS Melodic (su Ubuntu Bionic) Funzionante



# 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 (esempio: librerie di supporto)
- Come orientarsi con le versioni di ROS e relative versioni di Ubuntu
- Processo di Installazione di (Ubuntu) e ROS

Applications

ROS

Operating System  
(Linux Ubuntu)

ROS Melodic Morenia

Released May, 2018

Latest LTS, supported until May, 2023



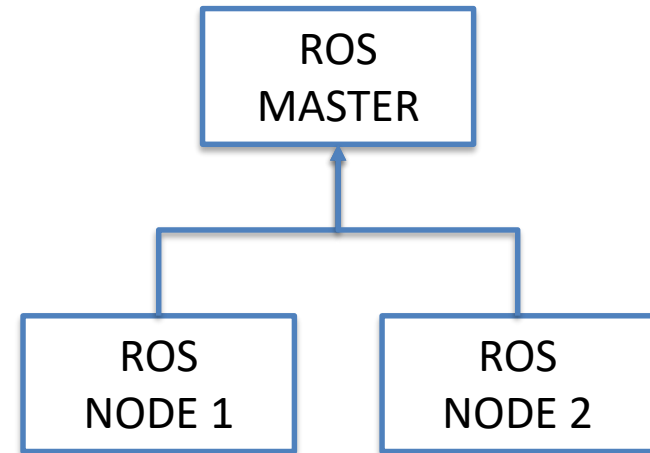
# Architettura di ROS

## 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 emebdedd 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-platorm)?
- 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



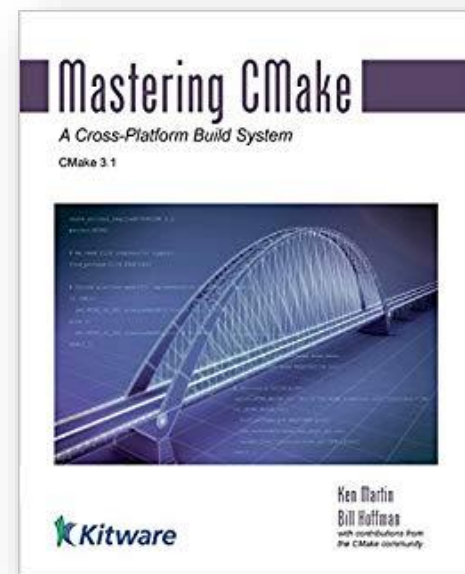
# Cmake

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/>

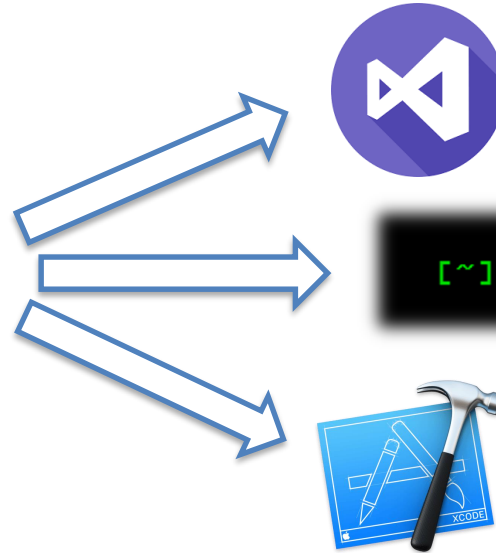


**Oltre 300 pagine!!!**

# Cmake workflow



## CMake



```
[~]$ make
```

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)
```

Vedremo  
utilizzando ROS  
come 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

# Esempio uso Cmake: compilazione OpenIGTlink



I passi principali sono:

- Scaricare sorgenti e estrarli in directory source
- Creare directory build
- Lanciare cmake-gui (per configurare i parametri)
- Make (oppure cmake --build)

Vediamo insieme come fare...

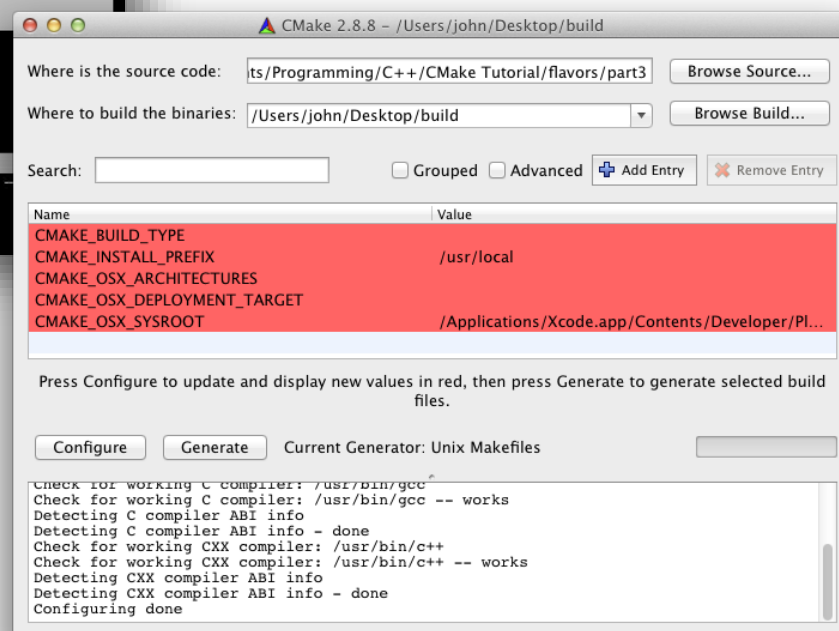
Dobbiamo installare cmake-gui:  
`sudo apt-get install cmake-gui`

# Sessione interattiva sull'uso di cmake

```
Page 1 of 1
BUILD_EXAMPLES      OFF
BUILD_SHARED_LIBS    ON
CMAKE_BACKWARDS_COMPATIBILITY 2.1
CMAKE_BUILD_TYPE
CMAKE_INSTALL_PREFIX /usr/local
UTK_DATA_ROOT        UTK_DATA_ROOT-NOTFOUND
UTK_USE_HYBRID        OFF
UTK_USE_PARALLEL      OFF
UTK_USE_PATENTED      OFF
UTK_USE_RENDERING     ON
UTK_WRAP_JAVA         OFF
UTK_WRAP_PYTHON       OFF
UTK_WRAP_ICL          OFF

BUILD_EXAMPLES: Build UTK examples.
Press [enter] to edit option
Press [c] to configure      Press [g] to generate and exit
Press [h] for help          Press [q] to quit without generating
Press [t] to toggle advanced mode (Currently Off)
```

Dobbiamo installare cmake-gui:  
`sudo apt-get install cmake-gui`



# Catkin build system



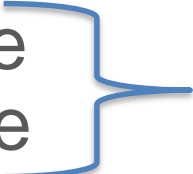
- Catkin è il build system di ROS a partire dalla versione Groovy, ha sostituito rosbUILD
- Catkin (e il predecessore rosbUILD) è basato su Cmake e vari tool di supporto per estenderne le funzionalità e gestire l'eterogeneità/complessità pacchetti ROS.
- La differenza principale consiste nell'utilizzo 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.**



build

# 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!**

# Creare un 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
```



# Fine lezione