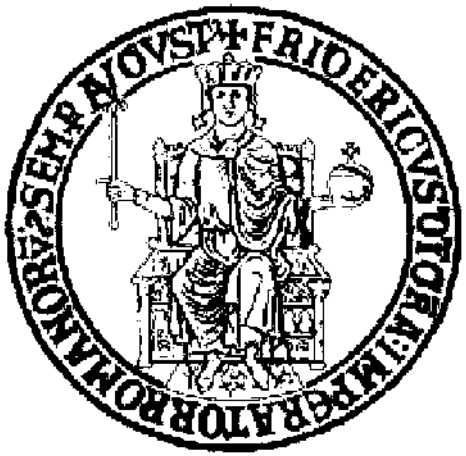


ROS.org

Robot **O**perating **S**ystem

Lezione 1

A cura di:
Jonathan Cacace

Outline

Lezione 1:

- Introduzione;
- Architettura ROS;
- Protocolli di comunicazione;
- Concetti base;
- Primi passi.

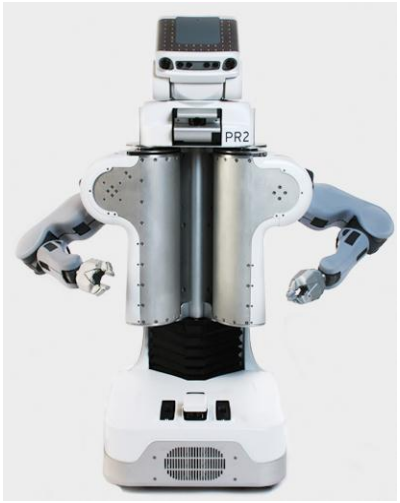
Lezione 2:

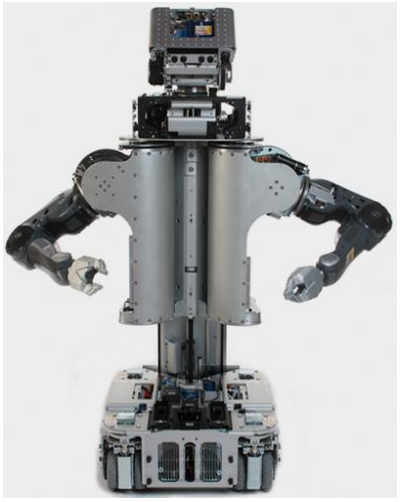
- ROS Stage;

Introduzione

ROS: *meta* sistema operativo per la programmazione dei robot .

“ROS: è un framework di programmazione che mette a disposizione strumenti e librerie per aiutare gli sviluppatori software nello sviluppo di applicazioni robotiche.”
(www.ros.org)





Inizialmente sviluppato per la piattaforma robotica **PR2**

Introduzione

ROS: *meta* sistema operativo per la programmazione dei robot .

HW Abstraction

Modular

Multi-Language

Rich Communication infrastructure

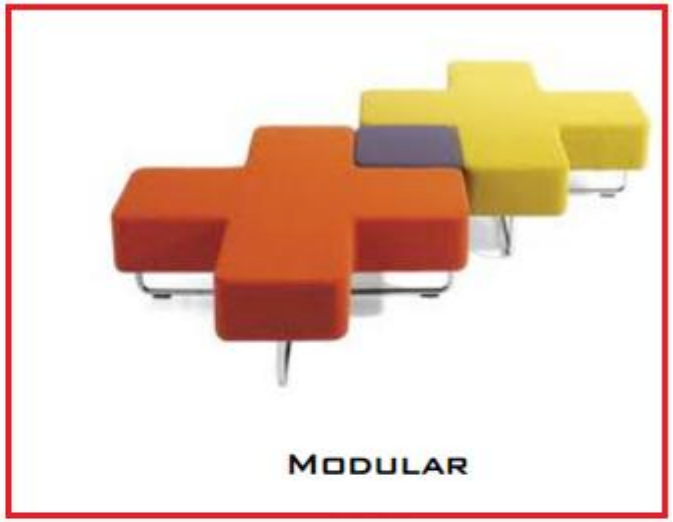
Introduzione

ROS: *meta* sistema operativo per la programmazione dei robot .

Introduzione

ROS: *meta* sistema operativo per la programmazione dei robot .

HW ABSTRACTION



MODULAR

MULTI-LANGUAGE

RICH COMMUNICATION INFRASTRUCTURE

Introduzione

ROS: *meta* sistema operativo per la programmazione dei robot .

HW ABSTRACTION

MULTI-LANGUAGE

MODULAR

RICH COMMUNICATION INFRASTRUCTURE

Introduzione

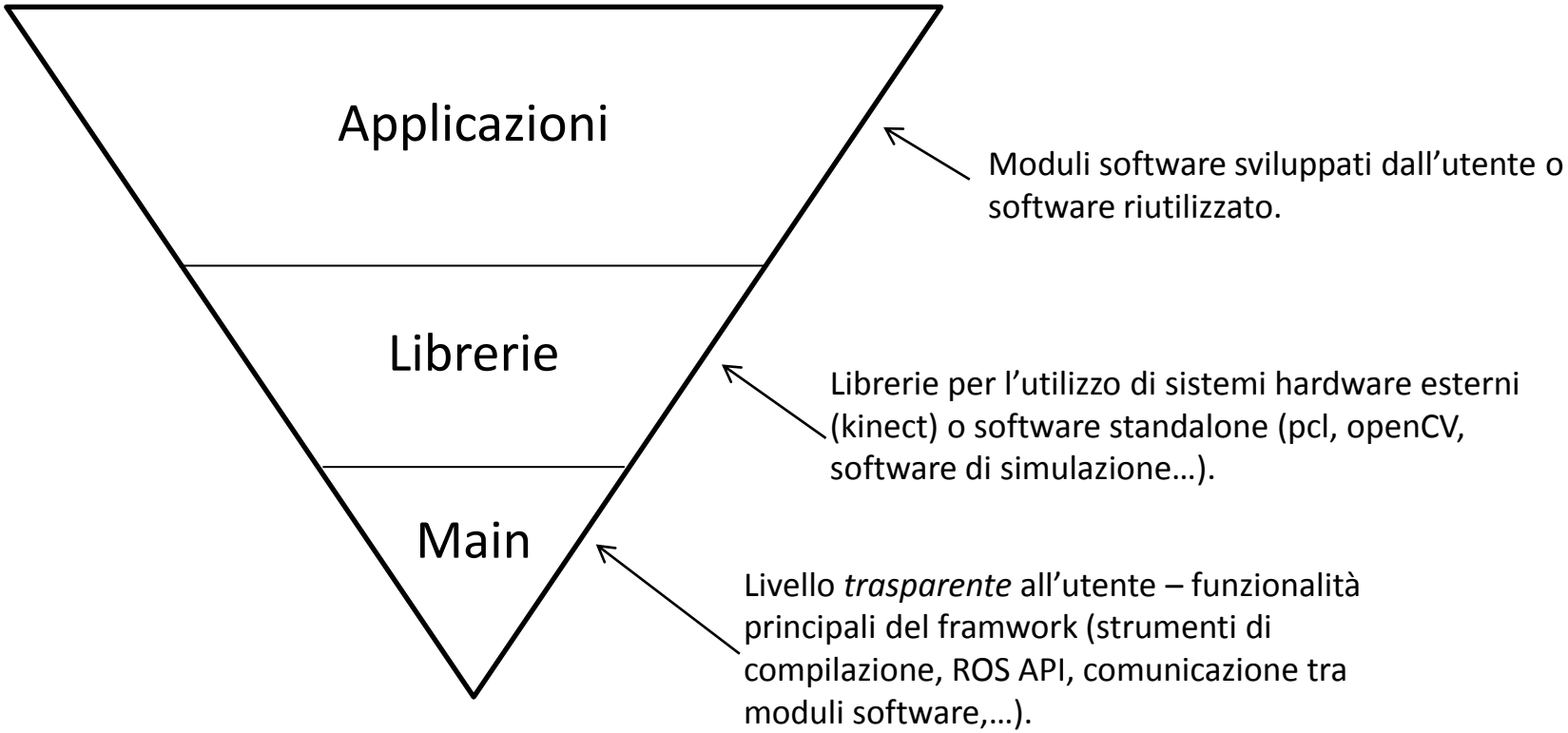
ROS: *meta* sistema operativo per la programmazione dei robot .

- Insieme di pacchetti software;
- Architettura distribuita in cui è possibile gestire in maniera asincrona un insieme di moduli software;
- Open-source rilasciato sotto licenza BSD;
- Permette lo sviluppo di moduli software in linguaggi: *C++* e *Python.*

***Comunità molto vasta* = supporto per il problem solving e sviluppo frequente di nuovi pacchetti**

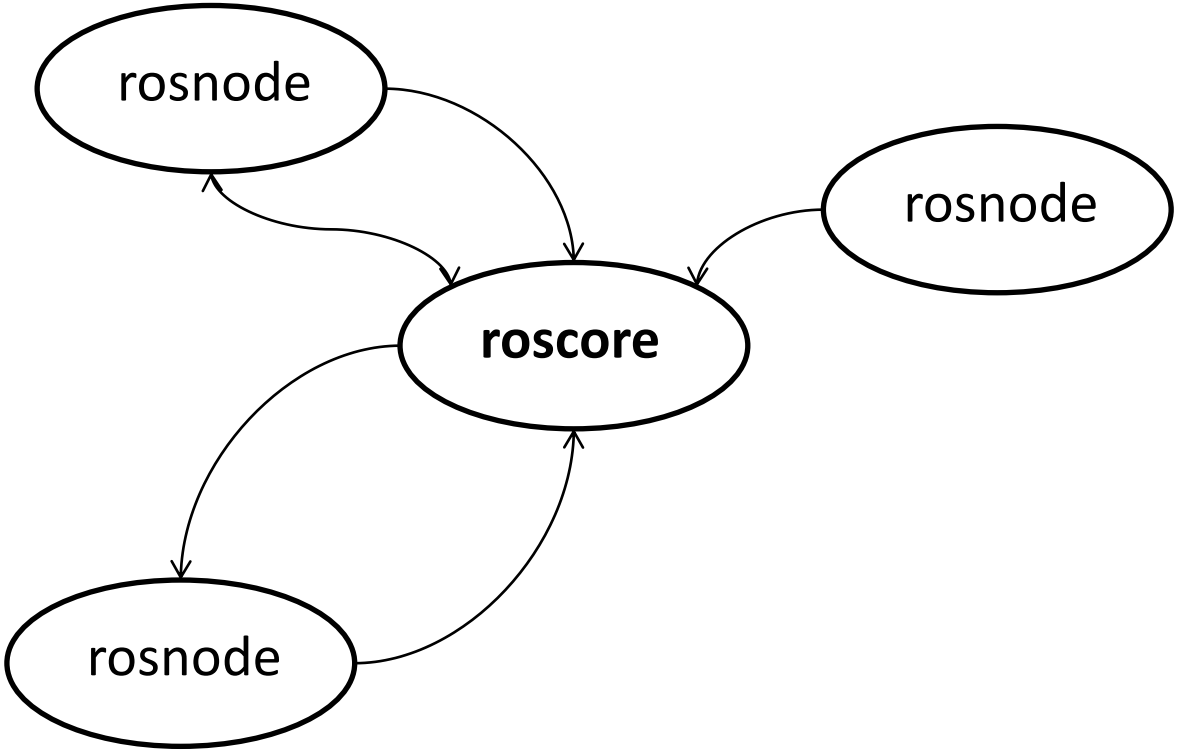
ROS

ROS è disponibile per diverse distribuzioni *Linux* (consigliato Ubuntu), disponibile anche per *OsX e Windows* (Sperimentali).



ROS

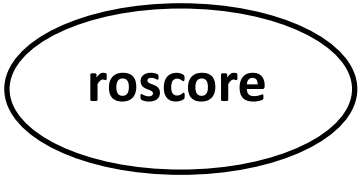
Architettura:



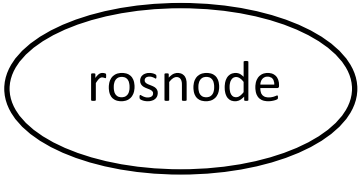
roscore: master coordination node.

ROS

Architettura:



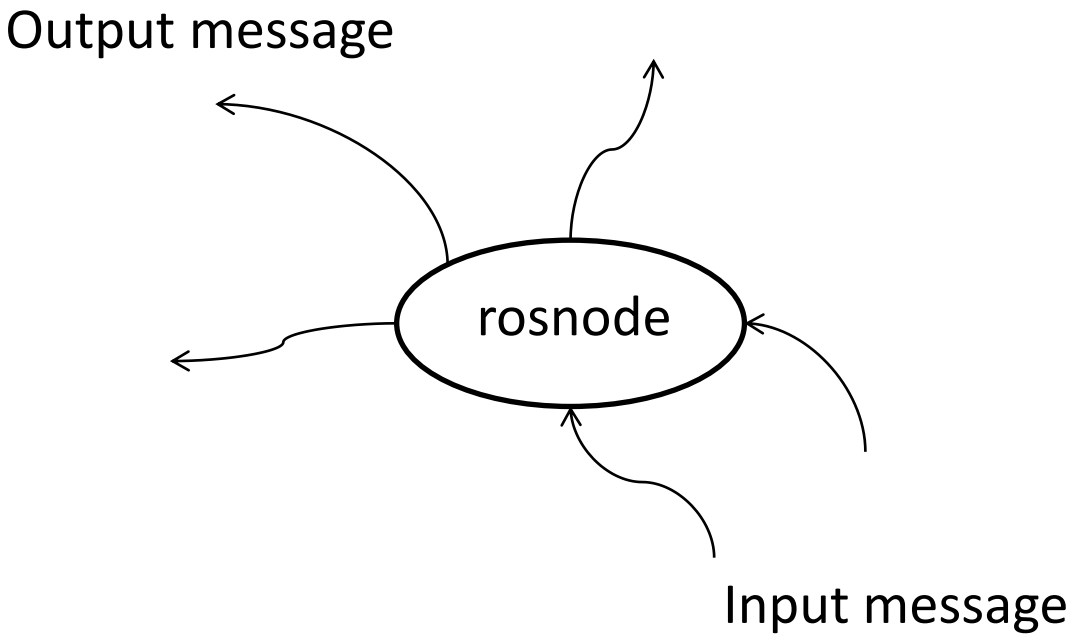
Orchestra i nodi ad esso connessi gestendone l'esecuzione parallela e la comunicazione tra questi.



Codice sviluppato dall'utente, o pacchetto software fornito da ROS.

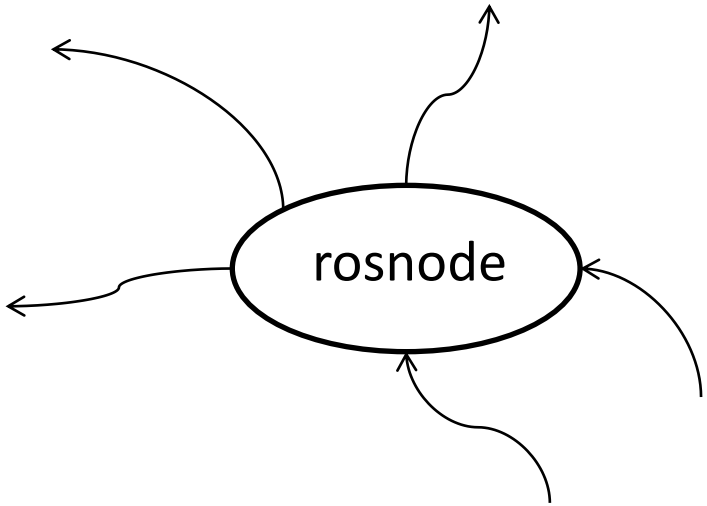
roscore: master coordination node.

Filosofia



roscore: master coordination node.

Filosofia



Modularità:

Un nodo identifica un particolare modulo del sistema, la possibilità di gestire l'interoperabilità tra i vari nodi attraverso dei semplici messaggi permette la creazione di un codice con **basso accoppiamento.**

roscore: master coordination node.

Librerie e pacchetti software

ROS mette a disposizione un vasto numero di pacchetti software già disponibili in versioni stand alone. Utilizzare le versioni disponibili per ROS aiuta lo sviluppatore e contribuisce alla creazione di un codice portabile.

Es. openNI (librerie per l'utilizzo della kinect in sistemi Linux)

Soluzione 1:

1. Installazione autonoma delle librerie openNI;
2. Interfacciamento della propria applicazione con le librerie openNI;
3. Inclusione del codice per la gestione (avvio, image capture,…) della kinect;
4. Utilizzo dei dati ottenuti dalla kinect.

Librerie e pacchetti software

ROS mette a disposizione un vasto numero di pacchetti software già disponibili in versioni stand alone. Utilizzare le versioni disponibili per ROS aiuta lo sviluppatore e contribuisce alla creazione di un codice portabile.

Es. openNI (librerie per l'utilizzo della kinect in sistemi Linux)

Soluzione 2: **ROS**

1. Download del pacchetto openNI offerto da ros (www.ros.org);
2. Avvio del modulo (rosnode) ROS per la gestione della kinect;
3. Utilizzo dei dati ottenuti dalla kinect forniti dal modulo ROS.

Librerie e pacchetti software

ROS mette a disposizione un vasto numero di pacchetti software già disponibili in versioni stand alone. Utilizzare le versioni disponibili per ROS aiuta lo sviluppatore e contribuisce alla creazione di un codice portabile.

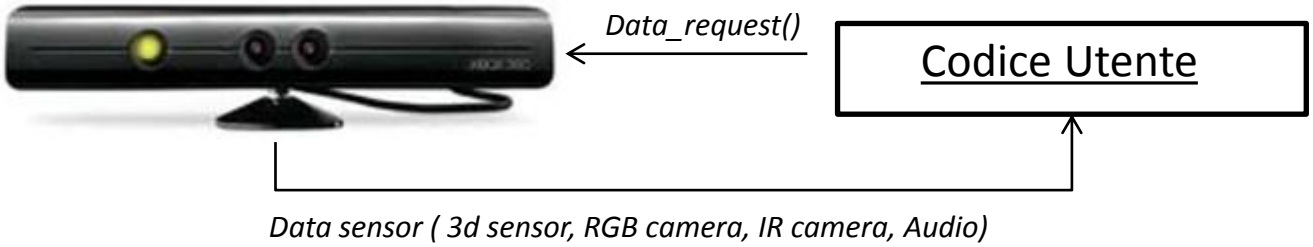
Es. openNI (librerie per l'utilizzo della kinect in sistemi Linux)

Soluzione 2: **ROS**

A differenza della Soluzione 1, il codice sviluppato dall'utente potrà essere eseguito facilmente compilato ed eseguito su tutte le macchine che dispongono di una propria versione di ROS, senza necessità installare librerie necessarie al suo funzionamento .

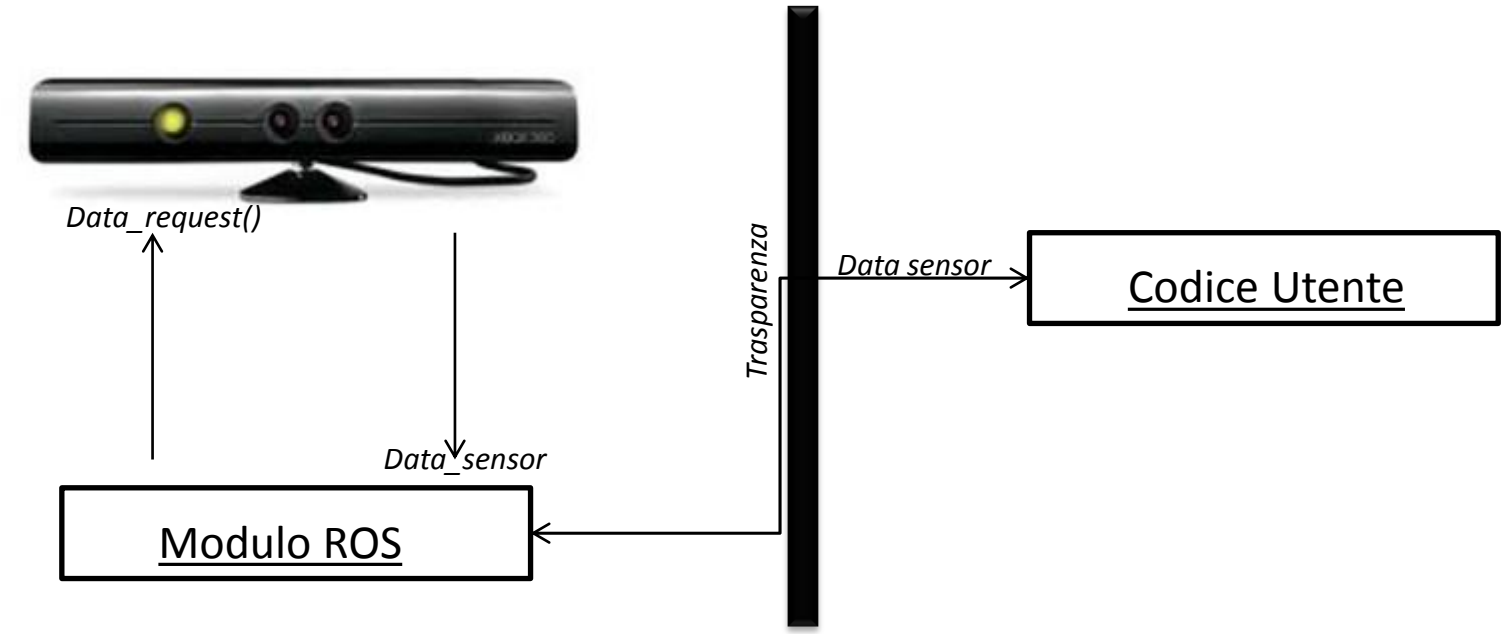
Librerie e pacchetti software

ROS mette a disposizione un vasto numero di pacchetti software già disponibili in versioni stand alone. Utilizzare le versioni disponibili per ROS aiuta lo sviluppatore e contribuisce alla creazione di un codice portabile.



Librerie e pacchetti software

ROS mette a disposizione un vasto numero di pacchetti software già disponibili in versioni stand alone. Utilizzare le versioni disponibili per ROS aiuta lo sviluppatore e contribuisce alla creazione di un codice portabile.



Librerie e pacchetti software

Popular package:

- **Simulazione:** Simulatori di ambienti 2D/3D (Stage / Gazebo).
- **Navigazione:** Odometria, flusso sensoriale, pose estimation…
- **SLAM:** Simultanea localizzazione e mapping.
- **Percezione:** *point cloud processing* (PCL), *openCV*…

Permettono l'astrazione di dispositivi hardware a basso livello:

- Joystick;
- GPS;
- Controller;
- Laser e sonar;

Protocolli di comunicazione

La comunicazione rappresenta uno degli elementi più significativi del sistema ROS. La possibilità di permettere un facile scambio di messaggi tra i nodi rende possibile la creazione di un programma **multi-threading** tralasciando le problematiche relative alla politica di sincronizzazione e comunicazione tra i vari processi.

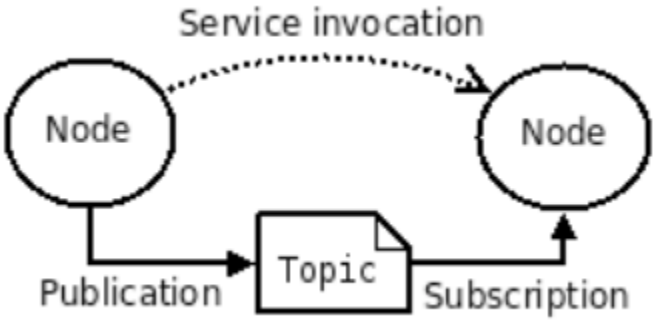
Comunicazione:

1. Publish/subscribe: modalità di comunicazione asincrona in broadcasting;
2. Service: modalità di comunicazione sincrona, secondo la semantica *request / reply*.

Protocolli di comunicazione

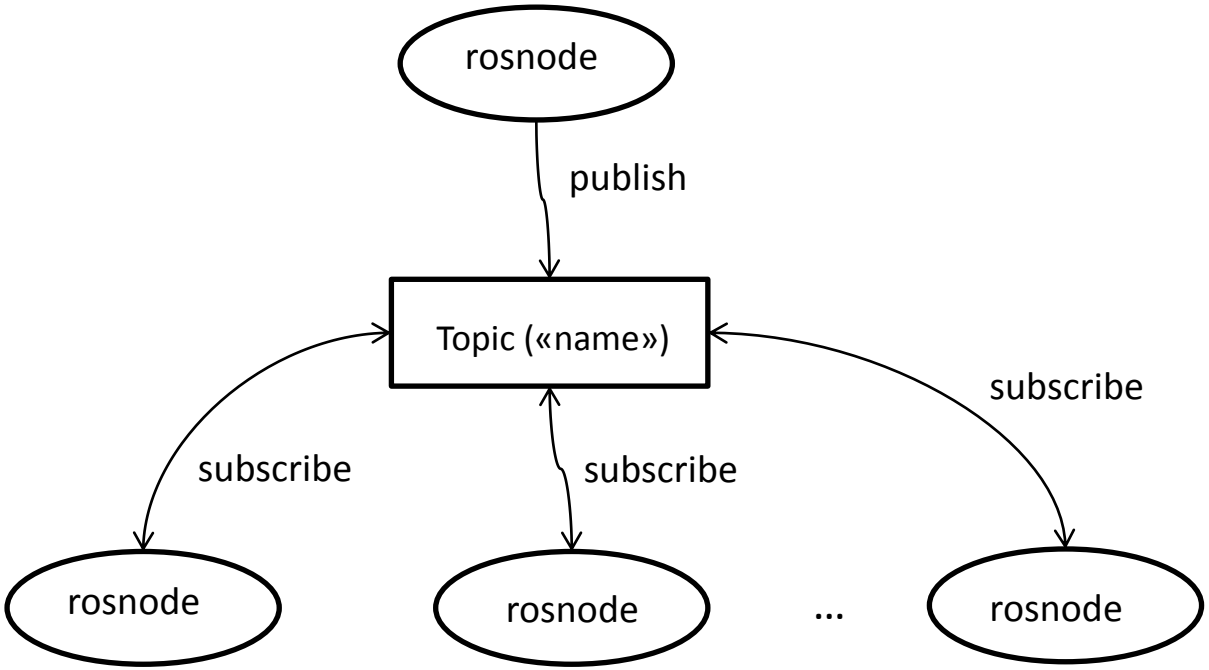
Publish / subscribe: Scrittura di un messaggio su di un topic messo a disposizione dal **roscore**. Tutti i nodi che desiderano ricevere il messaggio possono richiederlo al **roscore**.

Service: Un nodo invia una richiesta a tutti i nodi in grado di soddisfarla. Da questi nodi riceverà una risposta.



Protocolli di comunicazione

Pubslih / subscribe: Scrittura di un messaggio su di un topic messo a disposizione dal **roscore**. Tutti i nodi che desiderano ricevere il messaggio possono richiederlo al **roscore**.



Perché ROS???

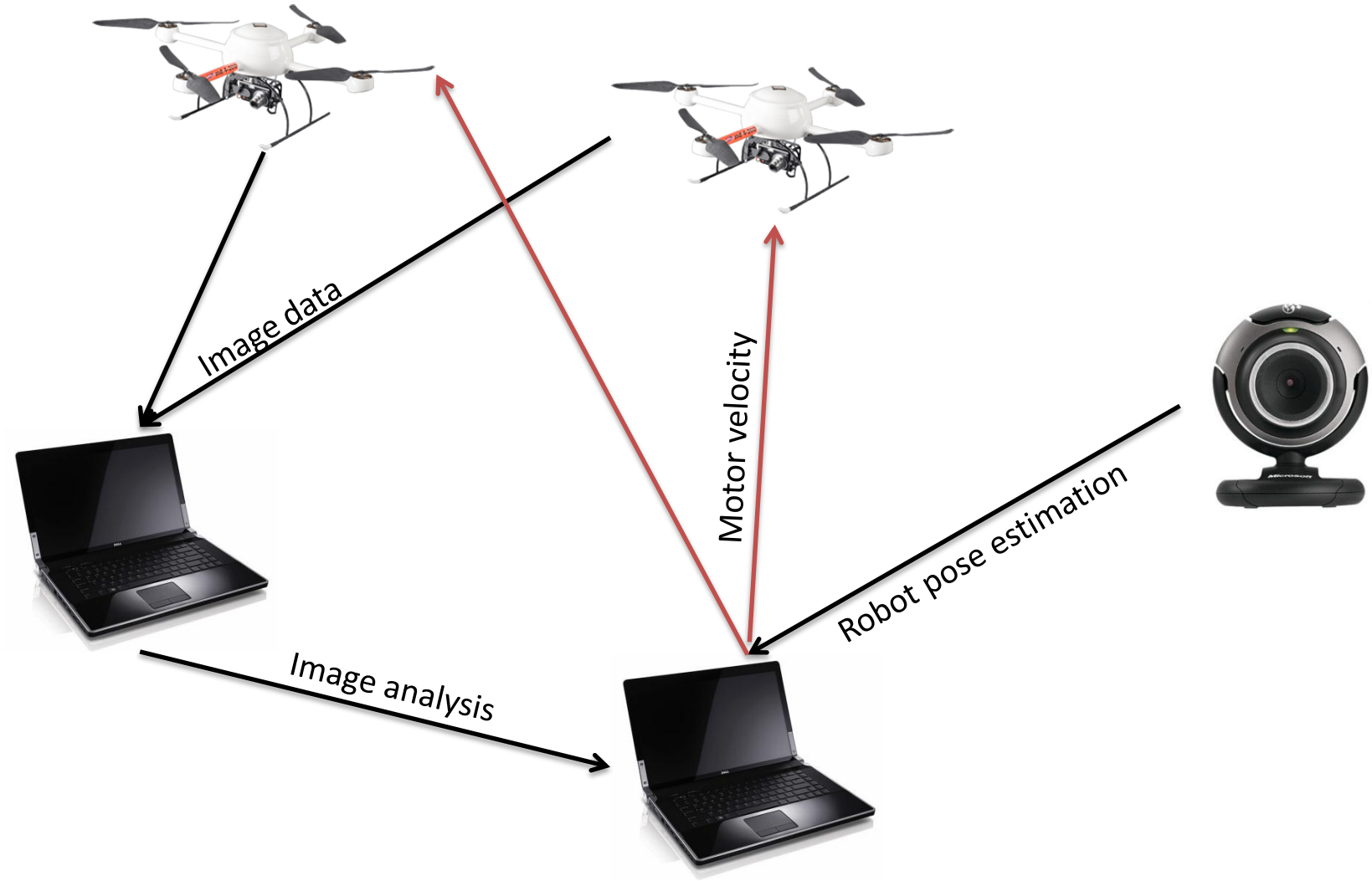
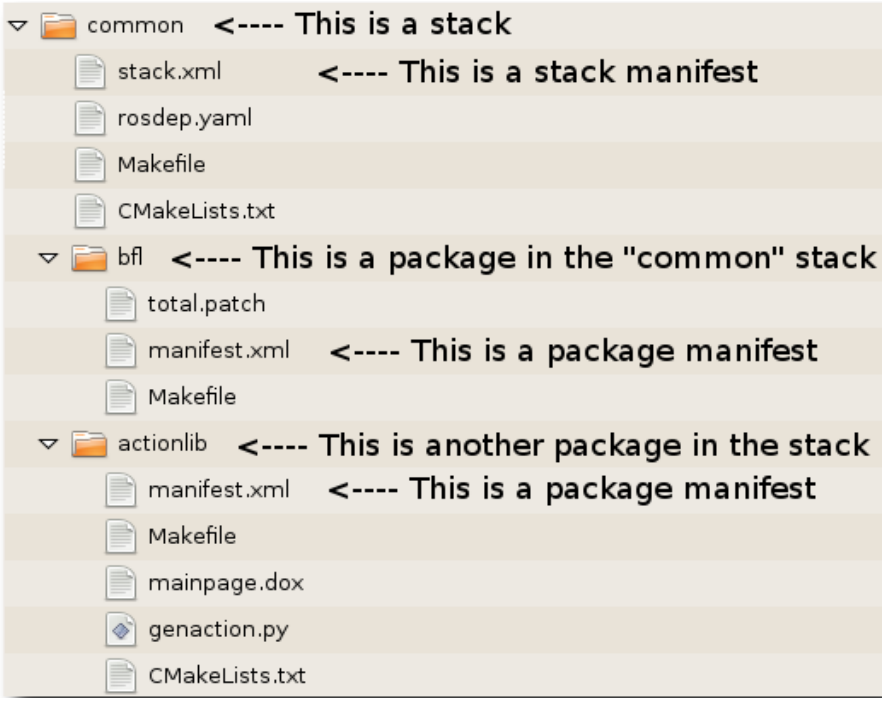


Image data
Motor velocity
Robot pose estimation
Image analysis

Concetti base

File system:

- *Package*: contengono librerie, eseguibili, codice sorgente…
- *Stack*: contiene diversi package.



Concetti base

File system:

- *Package*: contengono librerie, eseguibili, codice sorgente…
- *Stack*: contiene diversi package.

Comandi:

- *rospack:* permette di ricevere informazioni sui package;
- *rosstack:* permette di ricevere informazioni sugli stack;
- *roscd:* apre la directory contenente i package;
- roscreate-pkg [package_name]: crea un package;

Concetti base

Nodi ROS:

In ROS, un nodo non è altro che un eseguibile (generato da un insieme di codici sorgente) che fa parte di un package.

Un nodo ROS utilizza la librerie dedicate per comunicare con gli altri nodi, e con il core. Può leggere o pubblicare su un topic e fornire o richiedere servizi.

NB: anche il roscore, è da considerare come un nodo.

Comandi:

- *roscore:* avvia il nodo che rappresenta il core di ROS;
- *rosnode:* fornisce informazioni riguardo i nodi;
- *rosrun:* manda in esecuzione un nodo;

Concetti base

Nodi ROS *(Attributi di un nodo):*

*ros::*NodeHandle *nodehandle:* è un **oggetto** che permette di gestire tutte le attività legate al nodo in cui è dichiarato. Attraverso il nodehandle è possibile, ad esempio, registrarsi ai topic per leggere o pubblicare, accedere al parameter server.

Callback: è un **metodo** definito dall'utente che viene eseguito parallelamente al nodo in cui è dichiarato attraverso la generazione di un nuovo thread. I callback solitamente sono sempre in attesa e si attivano quando si attiva il nodo e non terminano fino alla fine dell'esecuzione.
Solitamente i callback vengono utilizzati per gestire la comunicazione attraverso i topic o i service.

Concetti base

Altri comandi utili:

rostopic: permette di visualizzare informazioni sui topic attivi.

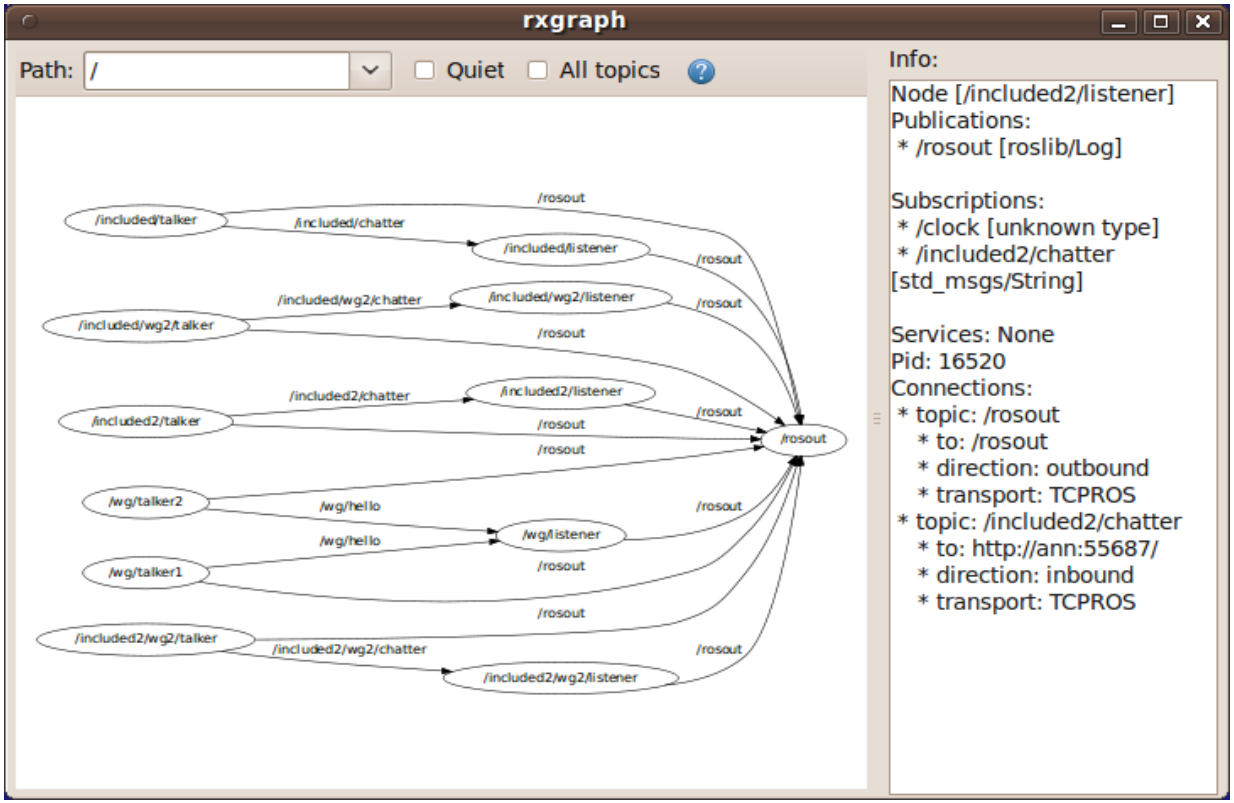
rostopic list: ritorna la lista dei topic attivi;
rostopic info [nome_topic]: ritorna il tipo di messaggio pubblicato sul topic;
rostopic echo [nome_topic]: mostra i messaggi inviati al topic.

rxgrpah: fornisce una rappresentazione grafica dei nodi attivi assieme ai messaggi che questi si scambiano.

Concetti base

Altri comandi utili:

rxgrpah: fornisce una rappresentazione grafica dei nodi attivi assieme ai messaggi che questi si scambiano.



Primi passi: creare un nodo (1/4)

Un nodo ROS fa sempre parte di un pakage.

Per creare un nuovo package, spostarsi all'interno della directory desiderata ed utilizzare il seguente comando:

$ roscreate-pkg [nome pacchetto]
Es. $ rocreate-pkg tutoral

Dopo aver creato un package è possibile utilizzare i comandi:

roscd [nome pacchetto]: per spostarsi all'interno della directory del package.

Primi passi: creare un nodo (2/4)

Main.h:

```
#include "ros/ros.h"

#ifndef MAIN_H_
#define MAIN_H_

class Main{
public:
        Main();
        void run();
};

#endif /* MAIN_H_ */
```

ros.h: librerie per l'utilizzo delle funzioni di ros.

Primi passi: creare un nodo (3/4)

Main.cpp:

```
#include "tutorial/main.h"

Main::Main() {
}
void Main::run() {
        //ros::spin(): mantiene il nodo in esecuzione,
        //in modo da renderlo disponibile al core in qualsiasi momento
        //senza ros::spin() il nodo termina la sua esecuzione,
        //a meno di un ciclo infinito.
        ros::spin();

}
int main(int argc, char** argv) {
        //ROS_INFO: printf con timestamped
        ROS_INFO("Main node Start");
        //ros::init presentazione del nodo al core -
        //dopo ros::init il core riconoscerà il nodo "main"
        //come nodo attivo
        ros::init(argc, argv, "main");
        Main main;
        main.run();
        return 0;
}
```

ros::init presentazione del nodo al roscore.

Primi passi: creare un nodo (3/4)

CMakeList:

```
cmake_minimum_required(VERSION 2.4.6)
include($ENV{ROS_ROOT}/core/rosbuild/rosbuild.cmake)

# Set the build type.  Options are:
#  Coverage       : w/ debug symbols, w/o optimization, w/ code-coverage
#  Debug          : w/ debug symbols, w/o optimization
#  Release        : w/o debug symbols, w/ optimization
#  RelWithDebInfo : w/ debug symbols, w/ optimization
#  MinSizeRel     : w/o debug symbols, w/ optimization, stripped binaries
#set(ROS_BUILD_TYPE RelWithDebInfo)

rosbuild_init()

#set the default path for built executables to the "bin" directory
set(EXECUTABLE_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/bin)
#set the default path for built libraries to the "lib" directory
set(LIBRARY_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/lib)

#uncomment if you have defined messages
#rosbuild_genmsg()
#uncomment if you have defined services
#rosbuild_gensrv()

#common commands for building c++ executables and libraries
rosbuild_add_executable(main src/main.cpp)
```

Primi passi: creare un nodo (3/4)

CMakeList:

```
cmake_minimum_required(VERSION 2.4.6)
include($ENV{ROS_ROOT}/core/rosbuild/rosbuild.cmake)

# Set the build type.  Options are:
#  Coverage       : w/ debug symbols, w/o optimization, w/ code-coverage
#  Debug          : w/ debug symbols, w/o optimization
#  Release        : w/o debug symbols, w/ optimization
#  RelWithDebInfo : w/ debug symbols, w/ optimization
#  MinSizeRel     : w/o debug symbols, w/ optimization, stripped binaries
#set(ROS_BUILD_TYPE RelWithDebInfo)

rosbuild_init()

#set the default path for built executables to the "bin" directory
set(EXECUTABLE_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/bin)
#set the default path for built libraries to the "lib" directory
set(LIBRARY_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/lib)

#uncomment if you have defined messages
#rosbuild_genmsg()
#uncomment if you have defined services
#rosbuild_gensrv()

#common commands for building c++ executables and libraries
rosbuild_add_executable(main src/main.cpp)
```

Per compilare il pacchetto creato, spostarsi all'interno della direcotry del pacchetto (è possibile utilizzare roscd) ed esgeuire il comando:

$ rosmake

Primi passi: creare un nodo (4/4)

rosnode:

Dopo aver compilato il pacchetto (senza errori!!) è possibile mandare il nodo in esecuzione:

Prima di tutto è necessario avviare il core:

$ roscore

In un altro terminale invece è necessario avviare il nodo:

$ rosrun [nome package] [nome nodo]

Es.

$ rosrun tutorial main

```
pacifica@pacifica-Dell-System-XPS-L502X:~/fuerte_workspace/tutorial$ rosnode list
/main
/rosout
```

Primi passi: [topic] creare un publisher

Creazione di un nodo ros che pubblica continuamente su un Topic con nome “publisher” dei dati di tipo Std_msgs::String.

Primi passi: [topic] creare un publisher

Pubslisher.h:

NodeHande: oggetto per l'accesso agli attributi e ai metodi per la gestione del nodo.

```
#include "ros/ros.h"

#ifndef PSUB_H_
#define PSUB_H_

class Publisher{
public:
        Publisher();
        void run();

private:
        //Handle del nodo
        ros::NodeHandle nodehandle;
        //Oggetto per la gestione del topic in scrittura
        ros::Publisher pubslisherHandle;
};

#endif /* PSUB_H_ */
```

Primi passi: [topic] creare un publisher

Pubslisher.h:

Publisher: oggetto per l'accesso ai metodi di pubblicazione.

```
#include "ros/ros.h"

#ifndef PSUB_H_
#define PSUB_H_

class Publisher{
public:
        Publisher();
        void run();

private:
        //Handle del nodo
        ros::NodeHandle nodehandle;
        //Oggetto per la gestione del topic in scrittura
        ros::Publisher pubslisherHandle;
};

#endif /* PSUB_H_ */
```

Primi passi: [topic] creare un publisher

Pubslisher.cpp:

```
#include "tutorial/publisher.h"
//Tipo di messaggio da inviare
#include "std_msgs/String.h"

//Costrusttore
Publisher::Publisher() {
        //Associazione del nodo al topic in pubblicazione
        // pubslisherHandle di tipo ros::Publisher
        // il metodo advertise, della classe NodeHandle, accetta in input
        // il nome del tipo della pubblicazione
        // il buffer da dedicare alla scrittura
        // in questo caso il nodo invia un messaggio di tipo std_msgs::String
        pubslisherHandle = nodehandle.advertise<std_msgs::String>("publisher", 1000);
}
void Publisher::run() {
        int count = 0;
        //Finche il core è attivo
        while (ros::ok()) {
                //Creazione del messaggio da pubblicare
                std_msgs::String msg; std::stringstream ss;
                ss << "hello world " << count; msg.data = ss.str();
                //pubblicazione del messaggio sul topic - metodo .publish
                pubslisherHandle.publish(msg);
                //ros::spinOnce(): essendo in un ciclo
                // si utilizza spinOnce che richiama periodicamente ros::spin().
                ros::spinOnce();
                usleep(1);
                ++count;
        }
}
```

advertise<Message Type>: Creazione di un topic su cui possono essere scritti messaggi di tipo «Message Type»

Primi passi: [topic] creare un publisher

Pubslisher.cpp:

```
#include "tutorial/publisher.h"
//Tipo di messaggio da inviare
#include "std_msgs/String.h"

//Costrusttore
Publisher::Publisher() {
        //Associazione del nodo al topic in pubblicazione
        // pubslisherHandle di tipo ros::Publisher
        // il metodo advertise, della classe NodeHandle, accetta in input
        // il nome del tipo della pubblicazione
        // il buffer da dedicare alla scrittura
        // in questo caso il nodo invia un messaggio di tipo std_msgs::String
        pubslisherHandle = nodehandle.advertise<std_msgs::String>("publisher", 1000);
}
void Publisher::run() {
        int count = 0;
        //Finche il core è attivo
        while (ros::ok()) {
                //Creazione del messaggio da pubblicare
                std_msgs::String msg; std::stringstream ss;
                ss << "hello world " << count; msg.data = ss.str();
                //pubblicazione del messaggio sul topic - metodo .publish
                pubslisherHandle.publish(msg);
                //ros::spinOnce(): essendo in un ciclo
                // si utilizza spinOnce che richiama periodicamente ros::spin().
                ros::spinOnce();
                usleep(1);
                ++count;
        }
}
```

publish: Metodo per pubblicare su topic.

Primi passi: [topic] creare un publisher

Dopo aver creato e mandato in esecuzione (come visto in precedenza) un nodo pubblica su un topic, è possibile monitorare lo stato di quel topic attraverso il comando rostopic:

$ rostopic list: Lista dei topic creati sul rosnode

$ rostopic info "Nome Topic": Informazioni sul topic (tipi di dati pubblicabili);

$ rostopic echo "Nome Topic": Monitoring dell'attività di publishing sul topic .

Primi passi: [topic] creare un publisher

Rostopic list:

```
pacifica@pacifica-Dell-System-XPS-L502X:~/fuerte_workspace/tutorial$ rostopic list
/publisher
/rosout
/rosout_agg
```

rostopic echo /publisher:

```
data: hello world 224270
---
data: hello world 224271
---
data: hello world 224272
---
data: hello world 224273
---
data: hello world 224274
---
data: hello world 224275
---
data: hello world 224276
---
data: hello world 224277
---
```

Primi passi: [topic] creare un subscriber

Subscriber.h:

sub: oggetto per l'accesso ai metodi per la gestione del nodo in sottoscrizione.

```
#include "ros/ros.h"
#include "std_msgs/String.h"

#ifndef SUBSCRIBER_H_
#define SUBSCRIBER_H_

class Subscriber{
public:
        Subscriber();
        void run();
        void topicCallback(const std_msgs::String::ConstPtr& msg);
private:
        ros::NodeHandle nodeHandle;
        //Oggetto per la gestione del topic in lettura
        ros::Subscriber sub;
};

#endif /* SUBSCRIBER_H_ */
```

Primi passi: [topic] creare un subscriber

Subscriber.h:

Dichiarazione di un callback

```
#include "ros/ros.h"
#include "std_msgs/String.h"

#ifndef SUBSCRIBER_H_
#define SUBSCRIBER_H_

class Subscriber{
public:
        Subscriber();
        void run();
        void topicCallback(const std_msgs::String::ConstPtr& msg);
private:
        ros::NodeHandle nodeHandle;
        //Oggetto per la gestione del topic in lettura
        ros::Subscriber sub;
};

#endif /* SUBSCRIBER_H_ */
```

Primi passi: [topic] creare un subscriber

Subscriber.cpp:

subscribe: Metodo per la sottoscrizione ad un topic

```
#include "tutorial/subscriber.h"

//Costrusttore
Subscriber::Subscriber() {
        //Associazione del nodo al topic in lettura
        // il metodo subscribe della classe NodeHandle
        // accetta in input il nome del topic da cui leggere
        // il buffer
        // ed il callback: il metodo che gestisce gli input sul topic
        sub = nodeHandle.subscribe("publisher", 1000, &Subscriber::topicCallback, this);
}
//Callback del topic "pusblisher"
//Viene richiamato ogni volta che sul tipic "publsher viene scritto qualcosa"
void Subscriber::topicCallback(const std_msgs::String::ConstPtr& msg) {
        ROS_INFO("I heard: [%s]", msg->data.c_str());
        sleep(1);
}
void Subscriber::run() {
        ros::spin();
}
int main(int argc, char** argv) {
        //ROS_INFO: printf con timestamped
        ROS_INFO("subscriber node Start");
        //ros::init presentazione del nodo al core -
        //dopo ros::init il core riconoscerà il nodo "main"
        //come nodo attivo
        ros::init(argc, argv, "subscriber");
        Subscriber subscriber;
        subscriber.run();
        return 0;
}
```

Primi passi: [topic] creare un subscriber

Subscriber.cpp:

```
#include "tutorial/subscriber.h"

//Costrusttore
Subscriber::Subscriber() {
        //Associazione del nodo al topic in lettura
        // il metodo subscribe della classe NodeHandle
        // accetta in input il nome del topic da cui leggere
        // il buffer
        // ed il callback: il metodo che gestisce gli input sul topic
        sub = nodeHandle.subscribe("publisher", 1000, &Subscriber::topicCallback, this);
}
//Callback del topic "pusblisher"
//Viene richiamato ogni volta che sul tipic "publsher viene scritto qualcosa"
void Subscriber::topicCallback(const std_msgs::String::ConstPtr& msg) {
        ROS_INFO("I heard: [%s]", msg->data.c_str());
        sleep(1);
}
void Subscriber::run() {
        ros::spin();
}
int main(int argc, char** argv) {
        //ROS_INFO: printf con timestamped
        ROS_INFO("subscriber node Start");
        //ros::init presentazione del nodo al core -
        //dopo ros::init il core riconoscerà il nodo "main"
        //come nodo attivo
        ros::init(argc, argv, "subscriber");
        Subscriber subscriber;
        subscriber.run();
        return 0;
}
```

Primi passi: [topic] creare un subscriber

Subscriber.cpp:

```
#include "tutorial/subscriber.h"

//Costrusttore
Subscriber::Subscriber() {
        //Associazione del nodo al topic in lettura
        // il metodo subscribe della classe NodeHandle
        // accetta in input il nome del topic da cui leggere
        // il buffer
        // ed il callback: il metodo che gestisce gli input sul topic
        sub = nodeHandle.subscribe("publisher", 1000, &Subscriber::topicCallback, this);
}
//Callback del topic "pusblisher"
//Viene richiamato ogni volta che sul tipic "publsher viene scritto qualcosa"
void Subscriber::topicCallback(const std_msgs::String::ConstPtr& msg) {
        ROS_INFO("I heard: [%s]", msg->data.c_str());
        sleep(1);
}
void Subscriber::run() {
        ros::spin();
}
int main(int argc, char** argv) {
        //ROS_INFO: printf con timestamped
        ROS_INFO("subscriber node Start");
        //ros::init presentazione del nodo al core -
        //dopo ros::init il core riconoscerà il nodo "main"
        //come nodo attivo
        ros::init(argc, argv, "subscriber");
        Subscriber subscriber;
        subscriber.run();
        return 0;
}
```

spin: Permette ai callback di restare in vita anche dopo la terminazione del «codice»

Primi passi: [topic] creare un subscriber

output:

```
pacifica@pacifica-Dell-System-XPS-L502X:~/fuerte_workspace/tutorial$ ./bin/subscriber
[ INFO] [1352633878.367201469]: subscriber node Start
[ INFO] [1352633878.592303051]: I heard: [hello world 128602]
[ INFO] [1352633879.592830481]: I heard: [hello world 137798]
[ INFO] [1352633880.593134859]: I heard: [hello world 148358]
[ INFO] [1352633881.593462446]: I heard: [hello world 159022]
```

Primi passi: Service

Server.h: (Somma di due interi)

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include "tutorial/AddTwoInts.h"
#ifndef ADDER_H_
#define ADDER_H_

class Add{
public:
        Add();
        void run();
        bool add(tutorial::AddTwoInts::Request &req, tutorial::AddTwoInts::Response &res);
private:
        //Handle del nodo
        ros::NodeHandle nodehandle;
        ros::ServiceServer service;
};

#endif /* ADDER_H_ */
```

service: Metodo per la gestione delle chiamate a servizio.

Primi passi: Service

Server.h: (Somma di due interi)

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include "tutorial/AddTwoInts.h"
#ifndef ADDER_H_
#define ADDER_H_

class Add{
public:
        Add();
        void run();
        bool add(tutorial::AddTwoInts::Request &req, tutorial::AddTwoInts::Response &res);
private:
        //Handle del nodo
        ros::NodeHandle nodehandle;
        ros::ServiceServer service;
};

#endif /* ADDER_H_ */
```

Dichiarazione metodo per la gestione delle chiamate service in **ingresso.**

Primi passi: Service

Server.h: (Somma di due interi)

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include "tutorial/AddTwoInts.h"
#ifndef ADDER_H_
#define ADDER_H_

class Add{
public:
        Add();
        void run();
        bool add(tutorial::AddTwoInts::Request &req, tutorial::AddTwoInts::Response &res);
private:
        //Handle del nodo
        ros::NodeHandle nodehandle;
        ros::ServiceServer service;
};

#endif /* ADDER_H_ */
```

Include della definizione del messaggio da inviare al service.

AddTwoInts.h è la definizione del messaggio utilizzato nel service. Il path di questo tipo di messaggi è package/srv/AddTwoInts.srv

Primi passi: Service

AddTwoInts.srv**:** (Somma di due interi)

```
int64 a
int64 b
---
int64 sum
```

AddTwoInts.h è la definizione del messaggio utilizzato nel service. Il path di questo tipo di messaggi è package/srv/AddTwoInts.srv

Primi passi: Service

Server.cpp: (Somma di due interi)

advertiseService: Presentazione al core del servizio offerto!!

```
//Costrusttore
Add::Add() {
        //il metodo advertiseService accetta in input
        // il nome del servizio da offrire
        // il metodo che risponde al servizio
        service = nodehandle.advertiseService("add_two_ints", &Add::add, this);
}

//Callback del servizio offerto dal nodo
//Il messaggio di input e di outpu è di tipo:
//tutorial::AddTwoInts
//E' definito in:
// $Package_directory/srv/
bool Add::add(tutorial::AddTwoInts::Request  &req, tutorial::AddTwoInts::Response &res ) {
        //Somma 2 interi.
        //Request è l'argomento di input del servizio (inviato dal nodo chiamante)
        //Response è il messaggio di ritorno di servizio
        res.sum = req.a + req.b;
        ROS_INFO("request: x=%ld, y=%ld", (long int)req.a, (long int)req.b);
        ROS_INFO("sending back response: [%ld]", (long int)res.sum);
        return true;
}
void Add::run() {
        ros::spin();
}
```

Primi passi: Service

Server.cpp: (Somma di due interi)

add: Metodo per la gestione del servizio

```
//Costrusttore
Add::Add() {
        //il metodo advertiseService accetta in input
        // il nome del servizio da offrire
        // il metodo che risponde al servizio
        service = nodehandle.advertiseService("add_two_ints", &Add::add, this);
}

//Callback del servizio offerto dal nodo
//Il messaggio di input e di outpu è di tipo:
//tutorial::AddTwoInts
//E' definito in:
// $Package_directory/srv/
bool Add::add(tutorial::AddTwoInts::Request  &req, tutorial::AddTwoInts::Response &res ) {
        //Somma 2 interi.
        //Request è l'argomento di input del servizio (inviato dal nodo chiamante)
        //Response è il messaggio di ritorno di servizio
        res.sum = req.a + req.b;
        ROS_INFO("request: x=%ld, y=%ld", (long int)req.a, (long int)req.b);
        ROS_INFO("sending back response: [%ld]", (long int)res.sum);
        return true;
}
void Add::run() {
        ros::spin();
}
```

Primi passi: Service

Server.cpp: (Somma di due interi)

```
//Costrusttore
Add::Add() {
        //il metodo advertiseService accetta in input
        // il nome del servizio da offrire
        // il metodo che risponde al servizio
        service = nodehandle.advertiseService("add_two_ints", &Add::add, this);
}

//Callback del servizio offerto dal nodo
//Il messaggio di input e di outpu è di tipo:
//tutorial::AddTwoInts
//E' definito in:
// $Package_directory/srv/
bool Add::add(tutorial::AddTwoInts::Request  &req, tutorial::AddTwoInts::Response &res ) {
        //Somma 2 interi.
        //Request è l'argomento di input del servizio (inviato dal nodo chiamante)
        //Response è il messaggio di ritorno di servizio
        res.sum = req.a + req.b;
        ROS_INFO("request: x=%ld, y=%ld", (long int)req.a, (long int)req.b);
        ROS_INFO("sending back response: [%ld]", (long int)res.sum);
        return true;
}
void Add::run() {
        ros::spin();
}
```

Primi passi: Service

Client.h: (Somma di due interi)

client: Oggetto per la gestione delle richieste Service.

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include "tutorial/AddTwoInts.h"
#ifndef ADDERC_H_
#define ADDERC_H_

class AddC{
public:
        AddC();
        void run(int,int);
private:
        //Handle del nodo
        ros::NodeHandle nodehandle;
        //Client
        ros::ServiceClient client;
};

#endif /* ADDERC_H_ */
```

Primi passi: Service

Client.cpp: (Somma di due interi)

```
#include "tutorial/add2Client.h"
AddC::AddC() {
        //il metodo advertiseClient accetta in input
        // il nome del servizio da utilizzare
        // in questo caso il tipo di messaggio
        // che viene trasmesso per il servizio è di tipo
        // tutorial::AddTwoInts
        client = nodehandle.serviceClient<tutorial::AddTwoInts>("add_two_ints");
}
void AddC::run(int a, int b) {
                                                        al serivzio
        srv.request.a = a;
        srv.request.b = b;
        //Chiamata al servizio
        if (client.call(srv))  {
                ROS_INFO("Somma: %ld", (long int)srv.response.sum);
        }
        else {
                ROS_ERROR("Failed to call service add_two_ints");
        }
}
int main(int argc, char** argv) {
        //ROS_INFO: printf con timestamped
        ROS_INFO("adder client node Start");
        //ros::init presentazione del nodo al core -
        //dopo ros::init il core riconoscerà il nodo "main"
        //come nodo attivo
        ros::init(argc, argv, "add_client");
        AddC addc;
        int a = atoll(argv[1]);
        int b = atoll(argv[2]);
        addc.run(a,b);
        return 0;
}
```

serviceClient(«Message to Send»)

Primi passi: Service

Client.cpp: (Somma di due interi)

```
#include "tutorial/add2Client.h"
AddC::AddC() {
        //il metodo advertise
        // il nome del servizi
        // in questo caso il
        // che viene trasmesso per il servizio è di tipo
        // tutorial::AddTwoInts
        client = nodehandle.serviceClient<tutorial::AddTwoInts>("add_two_ints");
}
void AddC::run(int a, int b) {
        //Creazione del messaggio da inviare al serivzio
        // "request"
        tutorial::AddTwoInts srv;
        srv.request.a = a;
        srv.request.b = b;
        //Chiamata al servizio
        if (client.call(srv))  {
                ROS_INFO("Somma: %ld", (long int)srv.response.sum);
        }
        else {
                ROS_ERROR("Failed to call service add_two_ints");
        }
}
int main(int argc, char** argv) {
        //ROS_INFO: printf con timestamped
        ROS_INFO("adder client node Start");
        //ros::init presentazione del nodo al core -
        //dopo ros::init il core riconoscerà il nodo "main"
        //come nodo attivo
        ros::init(argc, argv, "add_client");
        AddC addc;
        int a = atoll(argv[1]);
        int b = atoll(argv[2]);
        addc.run(a,b);
        return 0;
}
```

Creazione del messaggio da inviare al gestore del servizio richiesto

Primi passi: Service

Client.cpp: (Somma di due interi)

.call: chiamata del servizio

```
#include "tutorial/add2Client.h"
AddC::AddC() {
        //il metodo advertiseClient accetta in input
        // il nome del servizio da utilizzare
        // in questo caso il tipo di messaggio
        // che viene trasmesso per il servizio è di tipo
        // tutorial::AddTwoInts
        client = nodehandle.serviceClient<tutorial::AddTwoInts>("add_two_ints");
}
void AddC::run(int a, int b) {
        //Creazione del messaggio da inviare al serivzio
        // "request"
        tutorial::AddTwoInts srv;
        srv.request.a = a;
        srv.request.b = b;
        //Chiamata al servizio
        if (client.call(srv))  {
                ROS_INFO("Somma: %ld", (long int)srv.response.sum);
        }
        else {
                ROS_ERROR("Failed to call service add_two_ints");
        }
}
int main(int argc, char** argv) {
        //ROS_INFO: printf con timestamped
        ROS_INFO("adder client node Start");
        //ros::init presentazione del nodo al core -
        //dopo ros::init il core riconoscerà il nodo "main"
        //come nodo attivo
        ros::init(argc, argv, "add_client");
        AddC addc;
        int a = atoll(argv[1]);
        int b = atoll(argv[2]);
        addc.run(a,b);
        return 0;
}
```

Primi passi: Service

Client:

```
pacifica@pacifica-Dell-System-XPS-L502X:~/fuerte_workspace/tutorial$ rosrun tutorial adderClient 5 8
[ INFO] [1352747286.133851741]: adder client node Start
[ INFO] [1352747286.145111018]: Somma: 13
pacifica@pacifica-Dell-System-XPS-L502X:~/fuerte_workspace/tutorial$
```

Server:

```
pacifica@pacifica-Dell-System-XPS-L502X:~/fuerte_workspace/tutorial$ rosrun tutorial adder
[ INFO] [1352747278.438954586]: adder node Start
[ INFO] [1352747286.144617671]: request: x=5, y=8
[ INFO] [1352747286.144719863]: sending back response: [13]
```

Wiki

http://www.ros.org/wiki/

http://www.ros.org/wiki/ROS/Installation

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