ROS Getting Started

Installazione di ROS

Di seguito è riportata una breve guida per installare ROS Melodic Morenia su una macchina con sistema operativo Linux, distribuzione Ubuntu 18.04.4 LTS Bionic Beaver. Per maggiori dettagli è possibile consultare la guida web ufficiale.

Setup sources.list

Per prima cosa si aggiunge il repository ufficiale dei package ROS alla lista dei repository del sistema operativo, così da renderli disponibili al software di gestione dei pacchetti `apt`.

```
$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -
sc) main" > /etc/apt/sources.list.d/ros-latest.list'
```

Set up keys

Aggiungere la chiave pubblica per consentire il download dei pacchetti dal repository di ROS.

```
$ sudo apt-key adv --keyserver 'hkp://keyserver.ubuntu.com:80' --recv-key
C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654
```

Installazione

Si consiglia di effettuare preliminarmente un aggiornamento della lista dei pacchetti alla versione più recente, dopodicché si può procedere all'installazione più completa del framework ROS (`ros-melodic-desktop-full`) la quale include le `robot-generic libraries`, software per effettuare simulazioni 2D/3D, e una serie di strumenti e librerie utili nello sviluppo di applicazioni robotiche.

```
$ sudo apt update
$ sudo apt install ros-melodic-desktop-full
```

L'installazione può richiedere un pò di tempo (in genere dai 15 ai 20 minuti) a causa del download di tutte le librerie necessarie.

Inizializzazione rosdep

Prima di procedere all'utilizzo di ROS è bene inizializzare il sistema di gestione delle dipendenze fra moduli software necessario a compilare i codici sorgenti ed eseguire alcune componenti fondamentali di ROS.

```
$ sudo apt install python-rosdep
$ sudo rosdep init
$ rosdep update
```

Dipendenze per la compilazione dei pacchetti

È consigliato installare una serie di pacchetti contenenti strumenti utili frequentemente adoperati durante lo sviluppo di applicazioni robotiche con ROS, per la creazione e la gestione del workspace e dei packages.

```
$ sudo apt install python-rosinstall python-rosinstall-generator python-
wstool build-essential
```

Setup dell'ambiente di lavoro

Affinché sia possibile richiamare i comandi ROS da terminale è necessario configurare l'ambiente di lavoro andando ad aggiungere al file `~/.bashrc` la riga `source /opt/ros/melodic/setup.bash`, che permette, all'avvio di ogni terminale, di caricare la lista dei comandi e dei package principali installati. Tale operazione può essere effettuata digitando da terminale

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

oppure modificando attraverso editor di testo il file nascosto `.bashrc` presente nella cartella `/home/<user>` (`CTRL+h` per mostrare file nascosti dal gestore di cartelle grafico), aggiungendo in coda la riga equivalente:

```
# Set ROS Melodic environment
source /opt/ros/melodic/setup.bash
```

Prima di andare avanti, riavviare il terminale affinché il caricamento avvenga con successo, o digitare

```
$ source ~/.bashrc
```

Creazione del workspace

È bene creare un workspace per ogni progetto ROS di una certa consistenza. La cartella del workspace conterrà i diversi packages utilizzati nell'applicazione, i file sorgente e i relativi compilati.

```
$ mkdir -p ros_ws/src
$ cd ros_ws/
```

Impostare Python 3 come linguaggio di default (è necessario effettuare tale operazione solo la prima volta) per il sistema di compilazione di packages ROS **catkin** ed effettuare la prima compilazione attraverso il comando `catkin_make`.

```
$ catkin_make -DPYTHON_EXECUTABLE=/usr/bin/python3
```

Una volta che il processo di compilazione ha terminato è possibile notare come nella cartella del workspace siano stati creati il file `CMakeList.txt` e le cartelle `build` e `devel`. Il sistema di compilazione ROS usa un sistema basato su *CMake* (*Cross Platform Make*), e le opzioni di compilazione sono descritte all'interno dei file `CMakeList.txt`. I file compilati vengono salvati all'interno della cartella `build`, mentre i file eseguibili associati vengono salvati nella cartella `devel`.
La visibilità dei file contenuti nel workspace del progetto può essere ottenuta digitando il seguente comando all'avio di ogni terminale

```
$ source ~/ros_ws/devel/setup.bash
```

o modificando in maniera definitiva il file `~/.bashrc` aggiungendo alla fine la linea

```
# Set ros_ws workspace environment
source ~/ros_ws/devel/setup.bash
```

Creazione di un package

I progetti possono essere organizzati all'interno di differenti package per perseguire la *modularità del software* e migliorarne la navigabilità e la riusabilità del codice. I package raggruppano i moduli software per pertinenza delle funzionalità esportate, e devono essere creati all'interno della cartella`src` del workspace

```
$ cd ~/ros_ws/src
$ catkin_create_pkg test_pkg
```

Il comando `catkin_create_pkg` crea due file: il *manifesto* `package.xml`, che contiene informazioni sul package, quali nome, autore, licensa e dipendenze da altri package; e il `CMakeList.txt` contenente le informazioni di configurazione per la fase di *build*.

> ⚠️ Attenzione poiché non è permessa la creazione di package innestati in ROS, per raggruppare più pacchetti sotto un unico package logico è necessario utilizzare un **metapackage**. Questi richiedono un `CMakeList.txt` e un `package.xml` configurati nel seguente modo:
>
> ```
> <?xml version="1.0"?>
> <package format="2">
>   <name>METAPACKAGE_NAME</name>
>   <version>0.0.1</version>
>   <description>METAPACKAGE_DESCRIPTION</description>
>   <buildtool_depend>catkin</buildtool_depend>
>   <export>
>     <metapackage />
>   </export>
> </package>
> ```
>
> ```
> cmake_minimum_required(VERSION 2.8.3)
> project(PACKAGE_NAME)
> find_package(catkin REQUIRED)
> catkin_metapackage()
> ```

In aggiunta sono create anche due cartelle all'interno del package: `src`, per i codici sorgente e `include`, per gli eventuali file *header*. Un package può contenere una serie di cartelle per raggruppare i file in funzione del proprio scopo, in genere si segue la seguente convenzione dei nomi:

- `/action`: *ROS actions* personalizzate
- `/include`: header C++
- `/launch`: *launch* files
- `/msg`: *ROS messages* personalizzati
- `/scripts` o `/nodes`: script python
- `/src`: codice sorgente C++
- `/srv`: *ROS services* personalizzati

In fase di creazione di un package da linea di comando è possibile specificare eventuali dipendenze in termini di package ausiliari da cui questo dipende. Per fare ciò basta giustapporre al nome del package gli eventuali package dipendenti, come potrebbero essere `std_msgs` (che contiene gli *standard messages* di ROS) o `rospy` (la *ROS client library* per Python). Il comando per la creazione di un package con le relative dipendenze diventa:

```
$ catkin_create_pkg test_pkg std_msgs rospy
```

che consente di aggiungere automaticamente le dipendenze ai file package.xml e CMakeList.txt

```
<?xml version="1.0"?>
<package format="2">
  <name>test_pkg</name>
  <version>0.0.0</version>
  <description>test_pkg package for examples code</description>
  <maintainer email="user_name@todo.todo">user_name</maintainer>
  <license>TODO</license>

  <buildtool_depend>catkin</buildtool_depend>

  <build_depend>rospy</build_depend>
  <build_depend>std_msgs</build_depend>

  <build_export_depend>rospy</build_export_depend>
  <build_export_depend>std_msgs</build_export_depend>

  <exec_depend>rospy</exec_depend>
  <exec_depend>std_msgs</exec_depend>

  <export> </export>
</package>
```

```
cmake_minimum_required(VERSION 2.8.3)
project(test_pkg)
find_package(catkin REQUIRED COMPONENTS
  rospy
  std_msgs
)
include_directories(
# include
  ${catkin_INCLUDE_DIRS}
)
```

Nodi e Comunicazione

ROS si basa su un grafo di entità indipendenti che comunicano tra loro attraverso un'astrazione detta *messaggi*. Un **nodo** è la più piccola unità di processamento che può esistere in ROS (si può pensare ad essa come a un file eseguibile nell'ottica della visione di ROS come meta-OS). La documentazione ufficiale di ROS raccomanda di creare un nodo ROS per ogni singolo task specifico che si vuole realizzare, in modo da incentivare la riusabilità e la modularità del software.La comunicazione fra nodi può avvenire attraverso tre metodi principali:

- Topic
- Service
- Action

Topics

Il modo più semplice per realizzare lo scambio di informazioni fra nodi attraverso i messaggi è dato dall'utilizzo del meccanismo dei **topic**. Tale meccanismo astratto, in accordo ad una logica come quella descritta dal *software design pattern Observer*, prevede la presenza di un nodo **publisher** che scrive i messaggi (che possono essere variabili come interi, stringhe, ecc o dati strutturati più complessi) su questa "bacheca virtuale", e un insieme di nodi **subscriber** che leggono i messaggi pubblicati su di essa.
È possibile implementare in maniera semplice un primo esempio di nodi che realizzano una comunicazione base attraverso il meccanismo dei topic.

Nodo Publisher

Lo scopo è quello di creare un nodo che pubblica il valore di un contatore con una cadenza di circa un secondo su un determinato topic. Di seguito il codice di uno script Python che esegue questo task:

```
#!/usr/bin/env python3
import rospy
from std_msgs.msg import Int32

rospy.init_node('counter')
pub = rospy.Publisher('count_topic', Int32)
rate = rospy.Rate(1)
count = 0
while not rospy.is_shutdown():
    pub.publish(count)
    count += 1
    rate.sleep()
```

Le prime due linee

```
import rospy
from std_msgs.msg import Int32
```

provvedono a importare la *ROS client library* per Python e la *ROS standard message library* che verrà utilizzata per incapsulare e inviare sul topic valori interi a 32-bit.

```
rospy.init_node('counter')
```

Il modulo `rospy` mette a disposizione la funzione `init_node` che permette di instanziare un **nodo ROS**, nell'esempio identificato dal nome `'counter'`.

```
pub = rospy.Publisher('count_topic', Int32)
```

All'interno di `rospy` sono presenti anche diverse classi che possono essere richiamate per mezzo dei propri costruttori, ne è un esempio la classe `Publisher` che consente l'instanziazione di un oggetto specifico che avrà il compito di inviare messaggi di tipo `Int32` sul topic `'count_topic'`.

```
rate = rospy.Rate(1)
count = 0
```

Attraverso `Rate` è possibile creare un oggetto che consente di immagazzinare il valore di frequenza (in Hertz) con il quale si vorrà pubblicare il valore contenuto nella variabile `conut`, inizialmente posta a 0.
Il ciclo

```
while not rospy.is_shutdown():
    pub.publish(count)
    count += 1
    rate.sleep()
```

è eseguito fintanto che il nodo non viene stoppato (in genere è molto simile all'implementazione di un `while True`. Il ciclo di vita del nodo è triviale: pubblica il valore del contatore sul topic per mezzo dell'oggetto `Publisher`,aggiorna il valore della variabile `count`, infine si pone in `sleep` per un tempo tale da garantire che il corpo del ciclo `while` venga eseguito approssimativamente con la frequenza impostata nell'oggeto `Rate`.
Per avviare il nodo salvare lo script nel file `counter_node.py` nella cartella `scripts` del package `test_pkg`, dopodiché aggiungere i permessi di esecuzione allo script attraverso i comandi

```
$ chmod u+x counter_node.py
```

Da terminale, eseguire dapprima

```
$ roscore
```

successivamente, in un nuovo terminale, eseguire lo script precedente attraverso il comando `rosrun`:

```
$ rosrun test_pkg counter_node.py
```

In un nuovo terminale è possibile verificare i nodi ROS in esecuzione attraverso il comando

```
$ rosnode list
/counter
/rosout
```

o, in aggiunta, la lista dei topic esistenti (almeno un *publisher* o un *subscriber* presente)

```
$ rostopic list
/count_topic
/rosout
/rosout_agg
```

Per visualizzare il messaggi pubblicati dal nodo `counter`, digitare

```
$ rostopic echo /count_topic
data: 261
---
data: 262
---
data: 263
---
data: 264
---
data: 265
---
```

Nodo Subscriber

Supponiamo di modificare il _ nodo publisher_ precedente in modo tale che questo invii dei valori pseudo-casuali come se fosse un sensore (`writer_node.py`). Un *nodo subsciber* potrebbe essere interessato, ad esempio, a leggere tali valori dal *topic* e immagazzinarli all'interno di un buffer.
Quando un nodo si sottoscrive a un topic è necessario che questo specifichi una funzione ***callback***, la quale sarà invocata ogni qual volta un publisher scriverà un messaggio su quel topic. Di seguito è riportato uno script che instanzia un nodo ROS e si sottoscrive a un topic:

```
#!/usr/bin/env python3
import rospy
from std_msgs.msg import Int32

global buffer, buffer_size
buffer_size = 5
buffer = []

def read_callback(msg):
    global buffer, buffer_size
    if len(buffer) == buffer_size:
        buffer.pop(0)
    buffer.append(msg.data)
    print(buffer)

rospy.init_node('reader')
```

```
sub = rospy.Subscriber('data_topic', Int32, read_callback)
rospy.spin()
```

Poiché la variabile `buffer` sarà utilizzata per immagazzinare i dati all'interno della *funzione callback* è necessario che essa sia definita `global`, così che i valori permangano al suo interno e possano essere utiizzati anche all'esterno della funzione.

```
global buffer, buffer_size
buffer_size = 5
buffer = []
```

Nell'esempio il buffer è inizializzato vuoto con una capienza massima di 5 elementi. La funzione `read_callback` sarà invocata ogni volta che il nodo publisher invierà un messaggio `msg` sul topic:

```
def read_callback(msg):
    global buffer, buffer_size
    if len(buffer) == buffer_size:
        buffer.pop(0)
    buffer.append(msg.data)
    print(buffer)
```

Per richiamare le variabili globali all'interno della funzione è necessario specificare a inizio funzione che con tali nomi si farà riferimento appunto a loro; dopodicché la funzione provvede a inserire il valore letto dal topic, il quale è incapsulato all'interno del messaggio `msg` passato come parametro della funzione. Con `msg.data` si accede al campo del messaggio contenente il valore vero e proprio, e si provvede al suo inserimento all'interno del buffer, rimuovendo, qualora questo risulti pieno, il valore più vecchio tra quelli presenti (logica FIFO).

```
rospy.init_node('reader')
sub = rospy.Subscriber('data_topic', Int32, read_callback)
```

Dopo l'inizializzazione del nodo, è possibile effettuare la sottoscrizione al topic `data_topic` e agganciare ad esso la funzione `read_calback`.

```
rospy.spin()
```

Il metodo `spin` messo a disposizione dalla *ROS client library* `rospy` è equivalente alle istruzioni

```
while not rospy.is_shutdown():
    rate.sleep()
```

e viene utilizzato quando un nodo non ha altre azioni da compiere all'infuori delle operazioni effettuate all'interno delle *callbacks* associate ai topic ai quali è sottoscritto.
Per eseguire i nodi, assicurarsi come sempre che questi siano dotati dei permessi di esecuzione.

```
chmod u+x writer_node.py reader_node.py
```

Eseguire in un terminale il `writer`:

```
$ rosrun test_pkg writer_node.py
```

e in un altro il `reader`:

```
$ rosrun test_pkg reader_node.py
[7]
[7, 5]
[7, 5, 4]
[7, 5, 4, 5]
[7, 5, 4, 5, 5]
[5, 4, 5, 5, 5]
[4, 5, 5, 5, 3]
[5, 5, 5, 3, 3]
[5, 5, 3, 3, 6]
[5, 3, 3, 6, 5]
[3, 3, 6, 5, 5]
```

> ❗ TOPIC QUEUE
> Tutti i messaggi pubblicati su un topic vengono posti all'interno di una coda e processati, secondo l'ordine di arrivo dai nodi subscribers. Nel caso in cui un publisher invii i messaggi con una frequenza maggiore di quanto i subscriber riescano a processare, è possibile scartare i messaggi più vecchi andando a limitare la coda di arrivo. Quando si crea un publisher è possibile settare la `queue_size` a 1 per far sì che venga conservato solo l'ultimo messaggio pubblicato.
>
> ```
> pub = rospy.Publisher('TopicName', CustomMessage, queue_size=1)
> ```

Messaggi Personalizzati

ROS mette a disposizione una vasta gamma di tipi di messaggi già pronti, racchiusi nel package `std_msgs` in cui sono definiti i messaggi associati ai tipi primitivi.

ROS type	C++ type	Python type
bool	uint8_t	bool
int8	int8_t	int
uint8	uint8_t	int
int16	int16_t	int
uint16	uint16_t	int
int32	int32_t	int
uint32	uint32_t	int
int64	int64_t	int
uint64	uint64_t	int
float32	float	float
float64	double	float
string	std::string	string
time	ros::Time	rospy.Time

Vettori di questi tipi possono essere interpretati da Python sia come tuple che come liste. Oltre a `std_msg` vi sono altri package (come `geometry_msgs` o `sensor_msgs`) che definiscono dei messaggi di utilzzo corrente estendendo i messaggi standard andando a definire messaggi strutturati come composizione di tipi primitivi.
È possibile inoltre creare messaggi personalizzati andando a definirne la struttura all'interno di un apposito file di configurazione all'interno della directory `msg` del package. I file `*.msg` sono semplici file di testo all'interno dei quali è specificata la struttura generale di un messaggio di quel tipo:

```
# CustomMessage.msg
string name
float32 value
```

Dopo aver definito la struttura base del messaggio è necessario andare a modificare i file `CMakeList.txt` e `package.xml` presenti nel package per consentire a ROS di generare delle strutture *language-specific* per consentirne l'uso da parte dei nodi. Nel file `package.xml` devono essere aggiunte le seguenti linee

```
<build_depend>message_generation</build_depend>

<exec_depend>message_runtime</exec_depend>
```

mentre nel file `CMakeList.txt` devono essere decommentate e modificate le seguenti linee:

```
find_package(catkin REQUIRED COMPONENTS
  rospy
  std_msgs
  message_generation # Add message_generation here, after the other
packages
)
#...
# Generate messages in the 'msg' folder
add_message_files(
  FILES
  CustomMessage.msg
)
#...
# Generate added messages and services with any dependencies listed here
generate_messages(
  DEPENDENCIES
  std_msgs
)
#...
catkin_package(
  CATKIN_DEPENDS message_runtime
```

Una volta effettuate tali modifiche è possibile eseguire la compilazione del package attraverso il comando `catkin_make` lanciato dalla cartella del workspace (`~/ros_ws/`). Non resta che implementare un semplice nodo che faccia uso del messaggio personalizzato appena definito per appurare che l'intero processo sia andato a buon fine.

```
#!/usr/bin/env python3
import rospy
import random
from test_pkg.msg import CustomMessage

rospy.init_node('pub_node')
pub = rospy.Publisher('custom_topic', CustomMessage)
rate = rospy.Rate(0.5)
count = 0
while not rospy.is_shutdown():
    msg = CustomMessage()
    msg.name = 'CustomMessage_'+str(count)
    msg.value = random.random()
    pub.publish(msg)
    count += 1
    rate.sleep()
```

Per testare il tutto eseguire `roscore` in un terminale, aggiungere i permessi di esecuzione ed eseguire lo script `custom_node.py` in un nuovo terminale; a questo punto possono essere mostrati a schermo i messaggi pubblicati dal nodo usando:

```
$ rostopic echo /custom_topic
name: "CustomMessage_0"
value: 0.376212626696
---
name: "CustomMessage_1"
value: 0.0756872668862
---
name: "CustomMessage_2"
value: 0.0291156787425
---
```

Servizi

Un'altra strada per far comunicare diversi nodi ROS è data dai **servizi**, i quali possono essere associati a *chiamate sincrone a procedure remote*, permettendo di fatto a un nodo di invocare una funzione definita in un altro nodo.
È necesario definiri gli input e gli output del servizio in maniera simile a quanto fatto per un messaggio personalizzato.
Il meccanismo dei *servizi* prevede la presenza di un **nodo server** che fornisce il servizio e almeno un **nodo client** che richiama il servizio per fruirne.
Inputs e outputs di un servizio sono definiti in un file `*.srv` locato nella cartella `srv/` del package.

```
# WordCount.srv
string words # input
---
uint32 count # output
```

Come per i messaggi personalizzai, è necessario andare a modificare i file `package.xml` e `CMakeList.txt` prima di di compilare il package. Nel file `package.xml` aggiungere quanto segue:

```
<build_depend>message_generation</build_depend>

<exec_depend>message_runtime</exec_depend>
```

nel `CMakeList.txt` decommentare e modificare le linee:

```
find_package(catkin REQUIRED COMPONENTS
  rospy
  std_msgs
  message_generation # Add message_generation here, after the other
packages
)
#...
# Generate services in the 'srv' folder
```

```
add_service_files(
  FILES
  WordCount.srv
)
#...
# Generate added messages and services with any dependencies listed here
generate_messages(
  DEPENDENCIES
  std_msgs
)
#...
catkin_package(
  CATKIN_DEPENDS message_runtime
```

Dopo aver apportato tali modifiche eseguire `catkin_make`, il quale genererà tre classi: `WordCount`,`WordCountRequest`, e `WordCountResponse`. Tali classi saranno utilizzate durante le interazioni dei nodi con il servizio.

Service Server

Il **nodo server**, colui che fornisce il servizio, implementa una *callback* in accordo con la *service request*, ed esporta il servizio all'esterno.

```
#!/usr/bin/env python3
import rospy
from test_pkg.srv import WordCount,WordCountResponse

def count_callback(request):
    return WordCountResponse(len(request.words.split()))

rospy.init_node('service_server_node')
rospy.Service('word_count', WordCount, count_callback)
rospy.spin()
```

Il server deve includere il messaggio del servizio e la classe associata alla risposta

```
import rospy
from test_pkg.srv import WordCount,WordCountResponse
```

Dopo di che definisce la funzione *callback* che sarà invocata ogni volta che il servizio verrà richiamato

```
def count_callback(request):
    return WordCountResponse(len(request.words.split()))
```

In questo esempio la funzione associata al servizio restituisce il numero di parole, separate da spazi bianchi, contenute in una frase.

```
rospy.init_node('service_server_node')
```

inizializza il nodo server, mentre con

```
rospy.Service('word_count', WordCount, count_callback)
```

si crea il servizio denominato `word_count`, il quale riceve un messaggio di tipo `WordCount` e a cui è agganciata la funzione `count_callback`.

```
rospy.spin()
```

mantiene in vita il nodo e in attesa che qualcuno richiami il servizio.
Per eseguire il nodo server (salvato nella cartella `scripts/`) aggiungere i permessi di esecuzione e usare `rosrun`. Eseguendo su un terminale

```
$ rosservice list
```

è possibile verificare che il servizio sia attivo. È possibile inoltre testare immediatamente il servizio esportato digitando

```
$ rosservice call word_count 'one two three'
count: 3
```

o, qualora fossero presenti più parametri di input alla funzione, specificando

```
$ rosservice call word_count "words"='one two three'
count: 3
```

Service Client

Un client è un nodo che usa un *local proxy* per richiamare la funzione remota. Di seguito è riportato un esempio di implementazione.

```
#!/usr/bin/env python3
import rospy
from test_pkg.srv import WordCount,WordCountRequest

rospy.init_node('service_client_node')
rospy.wait_for_service('word_count')
word_counter_proxy = rospy.ServiceProxy('word_count', WordCount)
req = WordCountRequest()
req.words = 'one two three'
res = word_counter_proxy(req)
print('The number of words are '+str(res.count))
```

Il nodo client deve importare il messaggio del servizio e la classe associata alla richiesta da effettuare

```
import rospy
from test_pkg.srv import WordCount,WordCountRequest
```

Dopo,

```
rospy.init_node('service_client_node')
```

inizializza il nodo client e verifica che il server sia attivo

```
rospy.wait_for_service('word_count')
```

rimanendo eventualmente in attesa finché non vi sia in esecuzione un nodo server che implementa il servizio `word_count`.

```
word_counter_proxy = rospy.ServiceProxy('word_count', WordCount)
```

Il client provvede a creare una *proxy function* che consenta di richiamare la funzione associata al servizio `word_count`, in esecuzione su un altro nodo.

```
req = WordCountRequest()
req.words = 'one two three'
```

Crea un oggetto di tipo `WordCountRequest`, il quale contiene i campi specificati nel messaggio `WordCount` e setta la variabile `words` con il valore di input da passare al servizio.

```
res = word_counter_proxy(req)
```

Richiama il servizio attraverso la funzione callback e rimane in attesa che venga restituito il valore di ritorno dal server.

```
print('The number of words are '+str(res.count))
```

Infine viene stampato a schermo il numero si parole accedendo alla risposta utilizzando il nome del campo definito nel file di definizione del servizio.

Azioni

Un altro - più sofisticato - metodo di comunicazione tra nodi è dato dal meccanismo delle **azioni**, il quale sfrutta alla base sempre il meccanismo di comunicazione via topic per instaurare una *comunicazione asincrona bidirezionale*. Similmente ai *servizi*, le azioni sono utilizzate per demandare delle operazioni a un nodo server, in particolare nel caso in cui la richiesta che viene effettuata corrisponde all'esecuzione di un task che dura nel tempo e per cui si vuole avere la possibilità di ricevere delle risposte intermedie (*feedback*) e la possibilità di interrompere in qualsiasi momento l'esecuzione del task.
Anche le azioni sono basate dunque su un'architettura di tipo *client-server*, il primo passo per creare una nuova azione e definire i messaggi associati a **goal**, **result**, e **feedback** all'interno di un file di definizione`*.action`, nella cartella `action/` del package.
A titolo d'esempio è riportata l'implementazione di una semplice azione per modellere il comportamento di un timer, il quale aspetta e conta sino al valore impostato ritornando un feedback ogni secondo.

```
# Timer.action
float32 duration_time # goal
---
float32 total_time_elapsed # result
---
float32 partial_time_elapsed # feedback
```

Prima di compilare attraverso il comando `catkin_make` è necessario modificare il file `package.xml` aggiungendo le linee

```
<build_depend>actionlib_msgs</build_depend>

<exec_depend>actionlib_msgs</exec_depend>
```

e, all'interno del `CMakeList.txt`, decommentare e modificare le seguenti linee:

```
find_package(catkin REQUIRED COMPONENTS
  rospy
  std_msgs
  message_generation
  actionlib_msgs # Add message_generation here, after the other packages
)
#...
# Generate services in the 'srv' folder
add_action_files(
  DIRECTORY action
  FILES
  Timer.action
)
#...
# Generate added messages and services with any dependencies listed here
generate_messages(
  DEPENDENCIES
  actionlib_msgs
  std_msgs
)
#...
catkin_package(
  CATKIN_DEPENDS message_runtime actionlib_msgs
```

Dopo il processo di compilazione verranno creati 7 messaggi in una sottocartella di `devel/` (presente nel workspace), per consentire il funzionamento del meccanismo delle azioni: *TimerAction.msg*, *TimerActionFeedback.msg*, *TimerActionGoal.msg*, *TimerActionResult.msg*, *TimerFeedback.msg*, *TimerGoal.msg* and *TimerResult.msg*.

Action Server

Lo **action server** riceve il *goal* dal client e realizza il task necessario a perseguirlo all'interno di una *callback*.

```
#!/usr/bin/env python3
import rospy
import time
import actionlib
from test_pkg.msg import TimerAction, TimerGoal, TimerResult, TimerFeedback

def timer_callback(goal):
    start_time = time.time()
    result = TimerResult()
    curr_time = time.time()
    while (curr_time - start_time) < goal.duration_time:
        if server.is_preempt_requested():
            # Cancel goal request received
            result.total_time_elapsed = curr_time - start_time
            server.set_preempted(result,'Timer PREEMPTED')
            return
```

```
            feedback = TimerFeedback()
            feedback.partial_time_elapsed = curr_time - start_time
            server.publish_feedback(feedback)
            time.sleep(1)
            curr_time = time.time()

        result.total_time_elapsed = curr_time - start_time
        server.set_succeeded(result,'Timer SUCCEFULLY COMPLETED')

    rospy.init_node('timer_action_server_node')
    server = actionlib.SimpleActionServer('timer', TimerAction, timer_callback,
    False)
    server.start()
    rospy.spin()
```

Per prima cosa è necessario importare la libreria `actionlib` e i messaggi associati all'*azione*, e quelli associati a *goal*, *result* e *feedback*.

```
import rospy
import time
import actionlib
from test_pkg.msg import TimerAction, TimerGoal, TimerResult, TimerFeedback
```

Il server implementa all'interno di una funzione *callback* tutte le azioni necesarie al fine di perseguire il goal ricevuto.

```
def timer_callback(goal):
    start_time = time.time()
    result = TimerResult()
    curr_time = time.time()
    while (curr_time - start_time) < goal.duration_time:
        if server.is_preempt_requested():
            # Cancel goal request received
            result.total_time_elapsed = curr_time - start_time
            server.set_preempted(result,'Timer PREEMPTED')
            return
        feedback = TimerFeedback()
        feedback.partial_time_elapsed = curr_time - start_time
        server.publish_feedback(feedback)
        time.sleep(1)
        curr_time = time.time()

    result.total_time_elapsed = curr_time - start_time
    server.set_succeeded(result,'Timer SUCCEFULLY COMPLETED')
```

Il task implementato dal server inizia col registrare il tempo corrente non appena riceve un nuovo goal e creare un messaggio vuoto di tipo `TimerResult` il quale conterrà il risultato finale al termine dello svolgimento dell'azione.

Il ciclo interno itera con cadenza di un secondo fintanto che la differenza tra lo `start_time` e il `curr_time` è minore della durata indicata dalla variabile `duration_time` settata nel *goal*. Prima di ogni ciclo il server effettua un controllo se una richiesta di **preemption** è stata attivata (cancellazione del task), in caso affermativo il `result` del task conterrà il tempo trascordo sino al momento della richiesta di interruzione e lo *stato* dell'azione viene settato *preempted*. In caso non siano pervenute richieste di cancellazione dell'azione, un messaggio `TimerFeedback` contenente il tempo trascorso viene inviato al client.
Alla fine del ciclo, quando il goal viene raggiunto, `result` conterrà il valore del tempo realmente trascorso dall'attivazione dell'azione e lo stato dell'azione sarà settato a *succeeded*.
Dopo l'inizializzazione del nodo

```
rospy.init_node('timer_action_server_node')
server = actionlib.SimpleActionServer('timer', TimerAction, timer_callback,
False)
server.start()
rospy.spin()
```

viene creato il servere come istanza dell'oggetto `SimpleActionServer`. Il primo argomento del costruttore è il nome del server, il quale determina il *namespace* all'interno del quale i topics da lui usati verranno pubblicati. Il secondo argomento è il tipo di azione che il server implementa, mentre il terzo è la callback associata al *goal* ricevuto. L'ultimo argomento è una flag per abilitare o meno l'autostart del server. Al termine di tali operazioni il nodo server si pone in attesa - attraverso la funzione `spin` - che un client invii un goal.

Action Client

Per richiamare un'azione un **client** deve inviare un messaggio contenente il *goal* al server mentre rimane in attesa di eventuali feedback sullo stato di avanzamento dell'azione.

```
#!/usr/bin/env python3
import rospy
import time
import actionlib
from test_pkg.msg import TimerAction, TimerGoal, TimerResult, TimerFeedback

def feedback_callback(feedback):
    print('Time elapsed: '+str(round(feedback.partial_time_elapsed,2)))

rospy.init_node('timer_action_client_node')
client = actionlib.SimpleActionClient('timer', TimerAction)
client.wait_for_server()
goal = TimerGoal()
goal.duration_time = 10.0
client.send_goal(goal, feedback_cb=feedback_callback)
client.wait_for_result()
print('Time elapsed: '+str(client.get_result()))
```

Lo script del client deve importare i messaggi associati all'azione e la libreria `actionlib`

```
import rospy
import time
import actionlib
from test_pkg.msg import TimerAction, TimerGoal, TimerResult, TimerFeedback
```

dopodicché implementa una funzione *callback* che sarà agganciata all'arrivo dei messaggi di feedback ricevuti dal server.

```
def feedback_callback(feedback):
    print('Time elapsed: '+str(round(feedback.partial_time_elapsed,2)))
```

Come di prassi si inizializza il nodo

```
rospy.init_node('timer_action_client_node')
```

e si crea lo *action client* che dovrà avere lo stesso *namespace* del server

```
client = actionlib.SimpleActionClient('timer', TimerAction)
```

Dopo aver creato l'oggetto client si aspetta che un server che implementi l'azione associata veng attivato

```
client.wait_for_server()
```

dopodicché

```
goal = TimerGoal()
goal.duration_time = 10.0
client.send_goal(goal, feedback_cb=feedback_callback)
```

crea un messaggio di tipo `TimerGoal` che contiene il numero di secondi da attendere (nell'esempio 10 secondi) e lo invia al server, specificando - attraverso la parola chiave `feedback_cb` - il nome della callback che deve esssere agganciata ai messaggi di feedback.

```
client.wait_for_result()
```

Una volta inviato il messaggio di goal al server, il client attende sino al ragiungimento del termine dell'azione, la quale può terminare con sucesso o meno, risultato che può essere verificato utilizzando il metodo `get_result` sull'oggetto `client`. Aggiungendo le seguenti linee di codice alla `feedback_callback` del `timer_action_client_node`

```
if feedback.partial_time_elapsed >= 5:
    client.cancel_all_goals()
```

una *cancel request* verrà inviata al server e il nodo `timer_action_server_node` riceverà istantaneamente una *preempt request* a cui conseguirà l'interruzione del task.
Per testare l'azione, eseguire `roscore`, dunque i nodi server e client dopo aver modificato come al solito i permessi di esecuzione.

Parameters Server

Il **parameters server** in ROS fa riferimento a uno spazio di memoria condiviso tra i nodi in cui sono presenti una serie di parametri che possono essere utilizzati all'interno dei diversi nodi. È possibile pensare ad esso come a un file di configurazione condiviso. I nomi dei parametri e i loro valori vengono settati all'interno di un file di confgurazione `*.yaml` e possono essere letti o scritti dai nodi quando necessario, meccanismo particolarmente utile quando i valori dei parametri devono poter variare in real-time.
Facendo riferimento a uno dei primi esempi visti, in cui si faceva uso dei due nodi *reader* e *writer*, è possibile settare alcuni dei parametri utilizzati all'interno del file di configurazione `*.yaml`

```
reader: {'buffer_len':3}
writer: {'freq':1, 'min':3, 'max':7}
```

e modificare i nodi nei punti in cui si faceva uso dei parametri utilizzando la funzione `get_param` per prelevare i valori dal *parameters server*.

```
# reader_node.py
#!/usr/bin/env python3
import rospy
from std_msgs.msg import Int32

global buffer, buffer_size
buffer_size = rospy.get_param('/reader/buffer_len', 5)
buffer = []

def read_callback(msg):
    global buffer, buffer_size
    if len(buffer) == buffer_size:
        buffer.pop(0)
    buffer.append(msg.data)
    print(buffer)

rospy.init_node('reader')
```

```
sub = rospy.Subscriber('data_topic', Int32, read_callback)
rospy.spin()
```

```
# writer_node.py
#!/usr/bin/env python3
import rospy
import random
from std_msgs.msg import Int32

rospy.init_node('writer')
pub = rospy.Publisher('data_topic', Int32)
freq = rospy.get_param('/writer/freq', 1)
min_value = rospy.get_param('/writer/min', 1)
max_value = rospy.get_param('/writer/max', 10)
rate = rospy.Rate(freq)
while not rospy.is_shutdown():
    pub.publish(random.randint(min_value, max_value))
    rate.sleep()
```

Nella funzione `get_param` della libreria `rospy` il primo argomento è il path del nome con cui si ci riferisce al parametro , il secondo - opzionale - è il valore di default che verrà associato alla variabile qualora il parametro non sia presente nel *parameters server*.
Per includere il file `*.yaml` contenente i parametri quando vengono eseguiti i nodi si faccia riferimento alla sezione successiva trattante i *launch files*.

Launch Files

ROS fornisce uno strumento per consentire di mandare in esecuzione contemporaneamente più nodi senza far ricorso ogni volta a nuovi terminali e al comando `rosrun`. Tale meccanismo, utile per progetti più articolati e di dimensioni crescenti, prevede l'utilizzo di file `*.launch` (che sono dei file *XML-based* e prevedono l'so di *tag* specifici per esprimere le diverse opzioni) per specificare quali nodi devono essere eseguiti.
Facendo riferimento sempre all'esempio coinvolgente i nodi *reader* a *writer*, questi possono esssere eseguiti attraverso il seguente *launch-file* (convenzionalmente situato nella cartella `launch` del package).

```
<launch>
    <node pkg="test_pkg" type="reader_node.py" name="reader"
output="screen"/>
    <node pkg="test_pkg" type="writer_node.py" name="writer" />
</launch>
```

Un tag `<node>` è usato per ogni nodo che deve essere eseguito, specificando il package `pkg` di appartenenza, il nome dello script `type` e il nome del nodo `name`; il parametro `output` settato al valore `"screen"` viene utilizzato per mostrare a schermo i valori stampati dal `reader_node`.

Utilizzando il comando da terminale

```
$ roslaunch test_pkg multinodes.launch
```

i due nodi vengono messi in esecuzione, il che può essere verificato attraverso

```
$ rosnode list
/reader
/rosout
/writer
```

e

```
$ rostopic list
/data_topic
/rosout
/rosout_agg
```

I file `*.launch` permettono anche di rinominare i parametri del nodo che si vuole eseguire, o il nome dei topic, attraverso il tag `<remap>`, il quale permette di specificareil vecchio nome e il nuovo nome:

```
<launch>
    <node pkg="test_pkg" type="reader_node.py" name="reader"
output="screen">
        <remap from="data_topic" to="data"/>
    </node>
    <node pkg="test_pkg" type="writer_node.py" name="writer">
        <remap from="data_topic" to="data"/>
    </node>
</launch>
```

La rinominazione può essere verificata digitando:

```
$ rostopic list
/data
/rosout
/rosout_agg
```

Nei file `*.launch` è inoltre possibile includere i file di configurazione `*.yaml` in cui sono conservati i parametri, utilizzando il tag `<rosparam>`. Il file precedente può essere modificato inglobbando l'inclusione del file `param.yaml` conteunto nella cartella `param` del package:

```
<launch>
    <rosparam command="load" file="$(find test_pkg)/param/param.yaml" />

    <node pkg="test_pkg" type="reader_node.py" name="reader"
output="screen">
        <remap from="data_topic" to="data"/>
    </node>
    <node pkg="test_pkg" type="writer_node.py" name="writer">
        <remap from="data_topic" to="data"/>
    </node>
</launch>
```