ROS

ROBOT OPERATING SYSTEM

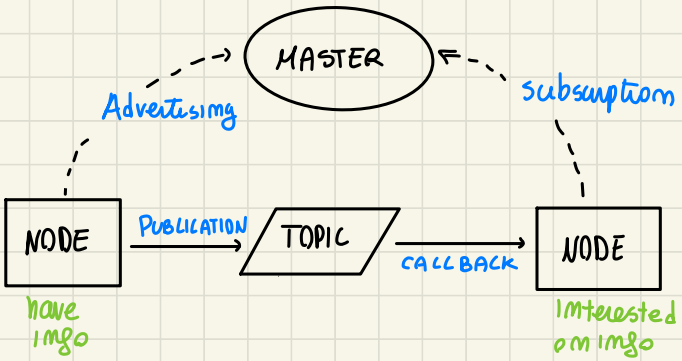
ROS · "ROBOT OPERATING SYSTEM" → build Robot system!

INTRO

↳ it is a middleware (environment) which connect nodes together
↓
thanks to publish / subscriber protocol

↳ { piece of software taking care of simple task }

a Node with an Information to share, it share it by a topic...
↕
a Node interested on that info subscribe to that topic and read info



ROS FEATURES
- Message passing (a)
- Package manage (b)
- Hardware abstraction (c)

PROBLEM STATEMENT...
why ROS is necessary?

Robot necessary sw capability
- Perception: look @ surrandaving
- Actuation: for motion
- Brain: where decision are made

{ 3 system capacity }
simplification
↓
3 scripts necessary

↳ 3 modules, which has to communicate somehow!

Inform brain about obstacle..
decide how to move → ask actuation to moove

perception.py → (visual info) → brain.py → (commands) → actuation.py
actuation.py → (motor states) → brain.py

{ flow of information }

{3 modules → communicating somehow}

how to communicate ?

(1) "POLLING" := interrogazione ciclica

keep update to check IF this is necessary to make an action

shared memory location command given by a shared location between modules

NOT the most efficient solution in a hardware/software co-design system

(2) "INTERRUPTS": event driven system

↳ notify the other modules when the event happen

MORE EFFICENT! event base communication channel

this is what ROS does ⇒ it takes care of the communication between modules

(a) provide the API, where you tell ROS the type of communication you whant between modules

you have only to build properly the modules and tell how the connection should look like → communication channels build by ROS

"MIDDLE WARE" := responsable for message passing and handling communication between modules on a distributed system

ROS := responsable for handling inter-process/inter-module inter-package communications on distributed system

(so ROS is a middleware)

{MESSAGE PASSING !}

- since ROS handle the communication channels
  ↳ we develop the individual modules/ packages

ROS has tools to support creation/maintenance of packages → "catkin"

↓

(b) { standard package management tools in ROS } ⇓

You work on individual packages thanks to these tools!
work on Actuation leaving brain / perception untached

{ PACKAGE MANAGEMENT } → good modularity of the code!

- building a package with lots of embedded drivers and code to talk to hardware...
  ↳ you can build functionality for a specific hardware, than wrap it with all ROS API

(c) ↓

use ROS API to abstract hardware functionality

{ ROS package when using a specific hardware with given functionality } ↓

I can use existing package (many open source package)

↓
- Localization
- Navigation

+ custom packages

hardware functionality ↔ code abstracted

{ HARDWARE ABSTRACTION }

↓

ROS NOT only about ROBOTICS → good software co-design hardware middleware

↓

{ mimic the operating system features }

ROS is NOT an "Operating System" BUT a framework on top of OS
↳ any Robot use a computer (Linux machine) ↳ development environment

to create ROS enabled programs ← ROS installed on Linux system (as set of packages)

↓

programming Robot based on that framework!
(programm that can be shared between ROBOTS)
→ previously each ROBOT used new programm, we didn't have Reusability which is guaranteed by ROS

⇓

already usable features existing in ROS, for example navigation algorithm SLAM, K.F for localization, path planning ... already done in ROS
Built in library for Robotic usage

ROS necessary to build ROBOT behaviours (not on electronics / mechanics)

{(Example) ROBOT Rover used by NASA is implemented in ROS2!
↳ ROS becoming more used in lot of applications}

SO in synthesis: ROS define components, interface, tools necessary to build advanced robot

Robots are made of → actuators, sensors, control systems

↳ ROS helps to build this and interconnect it by messages and topics

↓

- messages can be recorded as ROS bag files (for easy testing)
- messages can also be sent to visualization / teleoperation tool working with a digital twins

- ROS support also lots of hardware interfaces (camera, lidar, motor control)

- its modular architecture allows to build sw independent from vendor locker or licence fees

↓

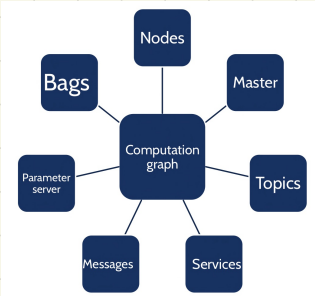
ROS work with any component with a sw interface (ROS 2 supports also microcontroller)

ROS := "set of Software framework for robot software development"

{ framework := already written sw code, used to help you to create your own application }

ROS STRUCTURE, COMPONENTS (FAST OVERVIEW)

CORE ELEMENTS:

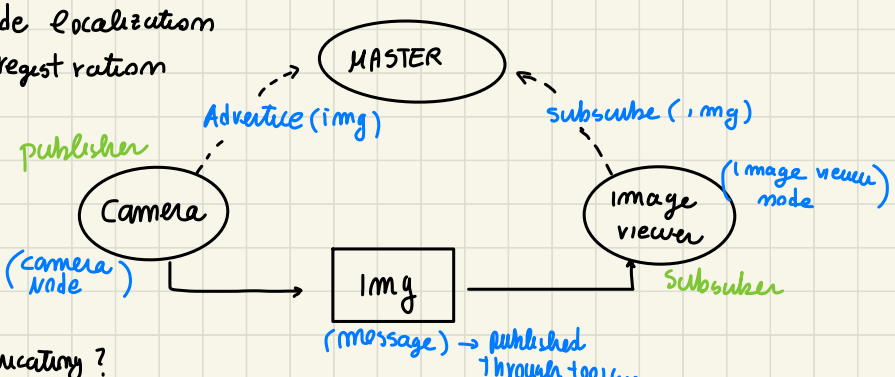


NODES executable units of ROS, basic "programmes" which do some tasks. (create a GUI, send data, use camera..)

- process performing computation
- a Robot system is composed by various node, communicating by the ROS Net

MASTER "centralized" logic of ROS, core element which allow for comunication between nodes, it handle information and communication.

- enables for node localization
- naming and registration



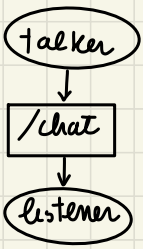
⇓

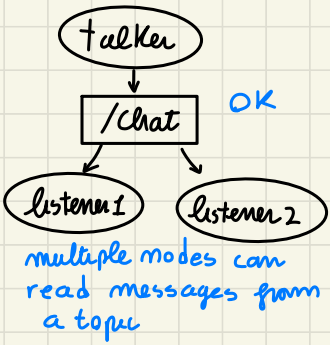
What if more nodes are communicating? how to differentiate it? (topics) ⇒

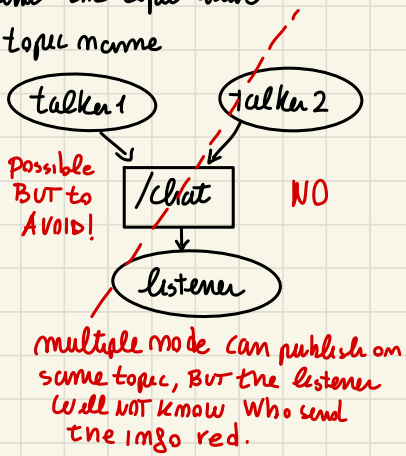
TOPICS

named channels for communication, to differentiate the Ros Data on the Net

- messages are published by node on the topic, and the topic have a specific supported message type and its topic name







MESSAGES

the nodes publish information on topics using messages.
and message type define the topic type!

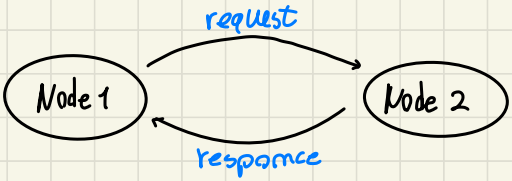
there are a lot of standard messages (std_msgs/...) ready to use
- BUT if necessary you can define your own **custom message**
(NOT a good practice)

SERVICES

more advanced kind of messages.. used to interrogate a node for a specific information (usefull tool when a node request just a simple imgo to another one)

↓

client/server paradigm



example/AddTwoInt.srv

```
int64 A
int64 B
---
int64 Sum
```

request / response

used to ask to a node for a specific information without publishing data from that node so allow for fast computation only when needed

service are called to retrieve specific information

PARAMETER SERVER simple access "multivariable dictionary", store usefull parameter you use in your ROS node, avoiding unnecessary recompile of the code for simple parameter resetting

↑

this is connected to the MASTER, it is a "ROSCORE" functionality

↳ parameters are variables setted on ROS launch file, so read when it start to avoid recompile of code...
otherwise they can also be setted by command line
(with some issues on dynamic reconfiguration while node running)

BAGS

(this is a type of ROS file ".bag" extension) = way to record properly Data inside ROS

↓

Way to record what happens inside the ROS Net, .bag files stores all usefull info for future testing like exchanged message on the net, timestamp, topics...

ROSCORE

collection of nodes and programmes, pre-requisite of ROS-based system ⟩ it start the needed component of a ROS system

- ROS Master
- ROS parameter server
- ROS out loggin node

it must run to allow communication between nodes

[on Terminal]

>> roscore { FIRST command when you start to run the ROS nodes }

(notice that it will occupy one terminal with all ROS usefull informations, you should open another terminal to program/test)

• OTHER USEFUL TOOLS...

LAUNCH FILE

usefull for big projects, allows you to run multiple nodes simultaneusly with one command. → it:
- start roscore
- start all the specified nodes togheither
- set all the specified parameter

(on launch file you can also set parameter)

it is written as an "Xml" file

TIMER

similar to subscribers, set up a callback (function) which will be called at timer data rate

↳ so needed when you wanna publish / compute Data using a specific timer frequency

TF (transformation Frame)

ROS tool to handle Frame transformation

(compute relative position of different components of your environment)

→ connect properly each reference frame (wored, joint ecc...)

very usefull for NAVIGATION

↳ described by a TF_tree

rviz

which is a standard 3D visualization tool

↓ a very usefull Graphical ROS tools you can use to visualize the frames on a 3D space is >> rviz

rqt_graph

Usefull ROS tool that allows you to see what is happening inside the ROS Network graphically

(let you check for node / topics / messages for an easy debug)

Lab1, Basics

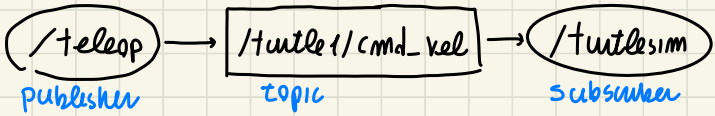
- First, to start running anything, we should run `>> roscore`
  Then it is possible to analyze ROS Net situation:
  - `>> rosnode list` : display all the active node
  - `>> rosnode info /node_name`: show pub/sub topics; where the node is running (usefull if more devices running) ecc.. usefull INFO of node + others...
  - `>> rostopic list` : show all the active topics
  - `>> rostopic echo /topic_name` to echo on terminal all info passing through that topic

NODE: to run a node on the net

`>> rosrun package_name node_name`

example running nodes:
- turtlesim_node → turtle GUI
- teleop_key → motion control

we get an rqt:



`>> rostopic hz topic_name` to check for the frequency of update publish data

you can also publish data on topic by command line

`>> rostopic pub topic_name data_type -r Hz rate -- 'DATA'`

```
simone@robotics:~$ rostopic pub /turtle1/cmd_vel ge
ometry_msgs/Twist -r 1 -- '[2.0, 0.0, 0.0]' '[0.0,
0.0, 1.8]'
```

↑ to force publishing the message @ a certain Hz rate --

otherwise -1 to publish only once

`>> rosrun plotJuggler plotJuggler` → usefull tool for Data visualization and code debugging (very usefull when working with odometry)

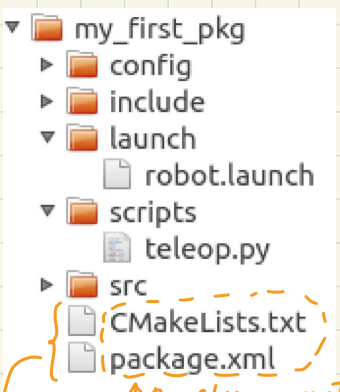
↓

So, Node are the atomic main ROS elements, each one is an independent process, written in C++/Python

(pub-sub)

Lab 2: code organization, Workspace, first package

ROS atomic unit of build are **packages** including: nodes, library, dataset, configuration files, all usefull...



↑ building + meta information

and packages are installed in the **workspace** (system folder) required by ROS build system (catkin)

```
workspace/
├─ build/   } managed by catkin automatically
├─ devel/   }
└─ src/     ↘ src/ folder contains the packages
   ├─ package/
   ⋮
```

WORKSALE FOLDERS

catkin build system of ROS take care of: locating/managing and compile easely your code → catkin is the ROS standard to build packages..

based on CMake (C++ build system)

- **package.xml**: has metadata about packages and dependencies, requirement about packages checked by compiler
- **CMakeLists.txt** to prepare/execute build process (CMake concepts) some changes to make on it (proj name, dependencies ecc..) depends if you have custom messages

- **/src** (SOURCE SPACE) Contains source code of ROS packages
- **/build** (BUILD SPACE) is where CMake build catkin packages
- **/devel** (DEVEL SPACE) where build target are placed before

Create workspace on workspace directory /robotics

» mkdir src

» catkin_make (catkin organization workspace)

↑ to run each time we add something new on /src

PACKAGE CREATION to create a new pckg:

» catkin_create_pkg [pkg_name] [dependency 1] [··] [dependency N]

example: ·· pub_sub_test std_msgs rospy roscpp

std_msgs: required to use standard message type

rospy, roscpp: required to use C++, Python on code

⇓

it create: on

```
src/
    pub_sub_test/
        CMakeLists.txt
        package.xml
        Include/
        src/   here you write the C++ node code
```

pub_sub package → a publisher node "talker" publish on "chatter" topic a string standard msg "hello world" with a certain fquency, and a node "listener" subscribe to that topic


```
robotics/src/pub_sub/
    launch/
    CMakeLists.txt
    package.xml
    src/
        pub.cpp  } nodes
        sub.cpp  }
```

Example

pub.cpp

```
1 #include "ros/ros.h"
2 #include "std_msgs/String.h"
3
4 #include <sstream>
5
6
7 int main(int argc, char **argv){
8
9         ros::init(argc, argv, "talker", ros::init_options::AnonymousName);
10        ros::NodeHandle n;
11
12        ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
13
14        ros::Rate loop_rate(10);
15
16        int count = 0;
17
18
19        while (ros::ok()){
20
21                std_msgs::String msg;
22
23                std::stringstream ss;
24                ss << "hello world " << count;
25                msg.data = ss.str();
26
27                ROS_INFO("%s", msg.data.c_str());
28
29                chatter_pub.publish(msg);
30
31                ros::spinOnce();
32
33                loop_rate.sleep();
34                ++count;
35        }
36
37
38        return 0;
39 }
```

- libraries include..
- allows to start same node multiple time, using different identifier
- identifier: node name
- node init. allocation, unique identifier
- executable handler
- buffer size
- publisher definition
- define msg type to publish
- topic name where you publish
- set a freq at which publish data
- manually ROS loop, ROS::ok() check if it is all OK (no interrupt)
- string ROS message
- c++ string message
- data field of our msg set to our string
- write INFO on terminal for debugging
- publish the msg on topic
- spin() of the node once per loop
- wait for the setted loop rate

→ each 10Hz frequency it publish "hello world [count]" on "chatter" topic

Sub.cpp

```
1 #include "ros/ros.h"
2 #include "std_msgs/String.h"
3
4 (*)↘
5 void chatterCallback(const std_msgs::String::ConstPtr& msg){
6   ROS_INFO("I heard: [%s]", msg->data.c_str());
7 }
8
9 int main(int argc, char **argv){
10
11        ros::init(argc, argv, "listener");
12
13        ros::NodeHandle n;
14        ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);
15
16        ros::spin();
17
18  return 0;
19 }
20
```

- libraries include
- write on terminal the msg (Debug)
- initialize "listener" node
- define the subscriber
- topic where to sub
- Buffer size
- function called by the node each time it receive a msg (*)
- loop automatically the node..

⇓

once written these 2 nodes, to properly compile the package we should add info on

CMakeLists.txt

```
14 add_executable(pub src/pub.cpp)
15 target_link_libraries(pub ${catkin_LIBRARIES})
16
17
18 add_executable(sub src/sub.cpp)
19 target_link_libraries(sub ${catkin_LIBRARIES})
20
```

- "pub" node to build from pub.cpp
- link the node executable to needed libraries

In general

any node registered to ROS master by unique identifier (unique name)

```
ros::init(argc, argv, "my_node_name");
ros::init(argc, argv, "my_node_name", ros::init_options::AnonymousName);
```

↑ IF you want to run the same node simultaneously more times... to set unique identifier

↓ node executable may have multiple handler

```
ros::NodeHandle nh;
```

the node have a loop waiting

1) automatically

```
ros::spin();
```

← check if { message waiting / completed timer / parameter reconfigured

2) manually

```
ros::Rate r(10); //10 hz
while (ros::ok()) {
  /* some execution */
  ros::spinOnce();
  r.sleep();
}
```

← specify loop rate

ros::ok() handle interrupts and stop if necessary

the node can have the role of publisher or subscriber:

if you write publisher node so to publish a msg on a ROS topic

```
ros::Publisher pub = nh.advertise<std_msgs::String>("topic_name", 5);
std_msgs::String str;
str.data = "hello world";
pub.publish(str);
```

when declaring it, connect to a topic

↑ declare the buffer size

declare also the msg type

← to publish msg on topic

while for a subscriber node so to read a msg from a ROS topic

```
ros::Subscriber sub = nh.subscribe("topic_name", 10, callback);

void [class::]callback(const pack_name::msg_type::ConstPtr& msg){
  ...
}
```

connect sub to a topic

↑ define called function when receive a msg

Lab3: Launch File, custom msg, services, parameters, Timers

- **Launch File** to run multiple node simultaneously (+ start Ros core + set specified parameters)

↓

to create launch file: you should create "launch/" folder inside package folder

↓

pub_sub/ launch/ → and inside it: launchFile_name.launch

.launch extension file written in xml language

multi_turtle.launch (group more nodes)

```
1  launch   root tags
2
3    <group ns="turtlesim1">   namespace group tags, to allow more node with same name
4      <node pkg="turtlesim" name="sim" type="turtlesim_node"/>
5      <node pkg="turtlesim" name="teleop" type="turtle_teleop_key"/>   start the node
6    </group>
7
8    <group ns="turtlesim2">
9      <node pkg="turtlesim" name="sim" type="turtlesim_node"/>
10   </group>
11                                   this node "mimic" convert data from 1st to 2nd turtle...
12   <node pkg="turtlesim" name="mimic" type="mimic">
13     <remap from="input" to="turtlesim1/turtle1"/>   } to change topic names
14     <remap from="output" to="turtlesim2/turtle1"/>
15   </node>
16
17 </launch>
```

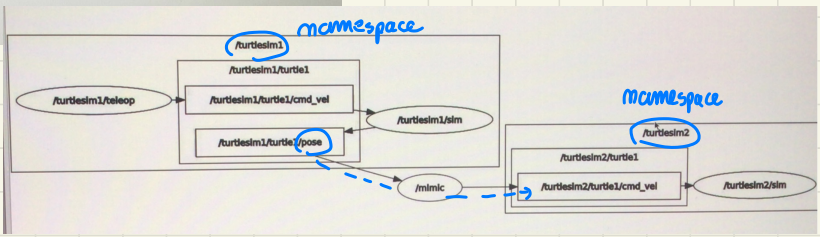
running this launch file as

>> roslaunch pub_sub multi_turtle.launch

or directly

>> multi_turtle.launch

↓



In general between <launch> </launch> tags we can:

we simultaneously start two turtlesim {turtlesim1, turtlesim2} and connect them together to have same pose

- start single nodes: `<node pkg="package" type="file_name" name="node_name"/>`
- create namespace (group of nodes): `< group ns = "namespace_name" > .. < /group >` (between these tag)
- change topics name: `<remap from="original" to="new"/>`

Custom messages → when std msgs are not enough and you need special data structure

(ISSUE: custom msg are based on your definition, if used by someone else it is necessary a specific build)

↓ must be defined on a specific folder: (EXAMPLE)

/package_name/msg/ ⟶ Num.msg

↓

"custom_name.msg" ← field called "num" containing an integer number

↳ and inside it you define its structure!

int64 num

⇓

you need to tell ROS how to properly compile code to include that custom msg: package_name/CMakeLists.txt ← edit CMake to build that msg together with pckg

↓

```
1 cmake_minimum_required(VERSION 2.8.3)
2 project(custom_messages)
3
4 ## Find catkin and any catkin packages
5 find_package(catkin REQUIRED COMPONENTS roscpp std_msgs message_generation)
6
7 add_message_files(
8   FILES
9   Num.msg
10
11 )
12
13 generate_messages(
14    DEPENDENCIES
15   std_msgs
16 )
17
18
19 ## Declare a catkin package
20 catkin_package( CATKIN_DEPENDS message_runtime)
21
22
23
24
25 ## Build talker and listener
26 include_directories(include ${catkin_INCLUDE_DIRS})
27
28
29 add_executable(pub_custom src/pub.cpp)
30 add_dependencies(pub_custom custom_messages_generate_messages_cpp)
31 target_link_libraries(pub_custom ${catkin_LIBRARIES})
32
33 add_executable(sub_custom src/sub.cpp)
34 target_link_libraries(sub_custom ${catkin_LIBRARIES})
35 add_dependencies(sub_custom custom_messages_generate_messages_cpp)
36
```

necessary message_generation to properly add custom msg's dependency

specify files contained on custom msg → adding the PATH to it

tell to compiler the dependences of our custom msg

export msg_runtime dependency!

specify to the compiler to check for that dependencies

↓

so if using our custom msg on a code of a node, we should include it as:

`# include "package_name/custom_name.h"`

ALSO: we must specify that it must be converted into c++ code, on /package_name/package.xml ↴ we uncomment these lines!

```
<build_depend>message_generation</build_depend>
<exec_depend>message_runtime</exec_depend>
```

Example, on pub/sub package:

pub.cpp

```cpp
#include "ros/ros.h"
#include "std_msgs/String.h"
#include "custom_messages/Num.h"

#include <sstream>


int main(int argc, char **argv){

        ros::init(argc, argv, "talker");
        ros::NodeHandle n;

        ros::Publisher chatter_pub = n.advertise<custom_messages::Num>("chatter", 1000);

        ros::Rate loop_rate(10);

        int count = 0;


        while (ros::ok()){

                static int i=0;
                i=(i+1)%1000;
                custom_messages::Num msg;
                msg.num =i;
                chatter_pub.publish (msg);

        }


        return 0;
}
```

modify publisher object using as datatype our custom one

message generation, by accessing num field

sub.cpp

```cpp
#include "ros/ros.h"
#include "std_msgs/String.h"
#include "custom_messages/Num.h"

void chatterCallback(const custom_messages::Num::ConstPtr& msg){
  ROS_INFO("I heard: [%ld]", msg->num);
}

int main(int argc, char **argv){

        ros::init(argc, argv, "listener");

        ros::NodeHandle n;
        ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);

        ros::spin();

  return 0;
}
```

specify msg subscription type → type of msg received by Callback

• Services

creation similar to custom msg ↘ first we need a "srv/" folder inside pckg folder

/package/srv

inside it we define our service.srv file

⇑

{ used to ask a node for a specific info without publishing from the node
↳ FAST computation }

(Example)

AddTwoInts.srv

```
int64 a      } INPUTS
int64 b      }
---          ← division
int64 sum    } OUTPUTS
```

it has similar ISSUES of custom messages for the usage but it is more usefull!

↓ you define IN/OUT of the service on srv file, than properly specify how to compile/use it on CMakeLists.txt

```
1  cmake_minimum_required(VERSION 2.8.3)
2  project(service)
3
4  ## Find catkin and any catkin packages
5  find_package(catkin REQUIRED COMPONENTS roscpp std_msgs message_generation )
6
7  add_service_files(
8    FILES
9    AddTwoInts.srv
10 )
11
12 generate_messages(
13   DEPENDENCIES
14   std_msgs
15 )
16
17
18
19
20 ## Declare a catkin package
21 catkin_package(CATKIN_DEPENDS message_runtime)
22
23 ## Build talker and listener
24 include_directories(include ${catkin_INCLUDE_DIRS})
25
26 add_executable(add_two_int src/add_two_int.cpp)
27 target_link_libraries(add_two_int ${catkin_LIBRARIES})
28 add_dependencies(add_two_int ${catkin_EXPORTED_TARGETS})
29
30
31 add_executable(client src/client.cpp)
32 target_link_libraries(client ${catkin_LIBRARIES})
33 add_dependencies(client ${catkin_EXPORTED_TARGETS})
34
```

needed to use services

specify the service used, its PATH to access it

dependencies of my service

properly generate header files

then we have to implement the node which handle the service computation

⇓

the "SERVER" of that service

add_two_int.cpp

```
1  #include "ros/ros.h"
2  #include "service/AddTwoInts.h"
3
4  bool add(service::AddTwoInts::Request &req,
5           service::AddTwoInts::Response &res)
6  {
7    res.sum = req.a + req.b;
8    ROS_INFO("request: x=%ld, y=%ld", (long int)req.a, (long int)req.b);
9    ROS_INFO("sending back response: [%ld]", (long int)res.sum);
10   return true;
11 }
12
13 int main(int argc, char **argv)
14 {
15   ros::init(argc, argv, "add_two_ints_server");
16   ros::NodeHandle n;
17
18   ros::ServiceServer service = n.advertiseService("add_two_ints", add);
19   ROS_INFO("Ready to add two ints.");
20   ros::spin();
21
22   return 0;
23 }
```

type of the service

include that service

request field := input field of service (pointer)

response field := output of service (pointer)

compute summ

print out on terminal to debug

usual node initialization

callback funct.

service server object which has add as callback function

service name

↪ wait for callback (ROS loop)

↳ this service server can be called directly from cmd line or by a node

to call the service from a node, client.cpp node..

```cpp
#include "ros/ros.h"
#include "service/AddTwoInts.h"


int main(int argc, char **argv)
{
  ros::init(argc, argv, "add_two_ints_client"); ← usual node initialization
  if (argc != 3)
  {
    ROS_INFO("usage: add_two_ints_client X Y");   check for properly passing info!
    return 1;
  }

  ros::NodeHandle n;
  ros::ServiceClient client = n.serviceClient<service::AddTwoInts>("add_two_ints"); ← create the client object
  service::AddTwoInts srv;  ← create the service object
  srv.request.a = atoll(argv[1]);
  srv.request.b = atoll(argv[2]);   } input given by cmd line as INPUT of server
  if (client.call(srv))  try to call service!
  {
    ROS_INFO("Sum: %ld", (long int)srv.response.sum);   } IF good input given, compute the output
  }
  else
  {
    ROS_ERROR("Failed to call service add_two_ints");   } IF error occurs on input
    return 1;
  }

  return 0;
}
```

client object using service type

↑ this node is on /service/client.cpp service pkg..

and we call it as >> rosrun service client INPUT 1 INPUT 2

We can also call the service from cmd line >> rosservice call /add_two_ints IN1 IN2

- we should also edit package.xml to add new dependencies

```
<build_depend>message_generation</build_depend>
<exec_depend>message_runtime</exec_depend>
```

- **Timers** similar to subscribers, we define a callback function which will be called at a specific timer data rate (useful when you want to use a custom rate to do something)

(Example) pub.cpp

```
1 #include "ros/ros.h"
2 #include "std_msgs/String.h"
3 #include <time.h>
4
5
6
7
8 void timerCallback(const ros::TimerEvent& ev){
9
10        ROS_INFO_STREAM("Callback called at time: " << ros::Time::now());
11
12 }
13
14 int main(int argc, char **argv){
15
16        ros::init(argc, argv, "timed_talker");
17        ros::NodeHandle n;
18
19        ros::Timer timer = n.createTimer(ros::Duration(0.1), timerCallback);
20
21        ros::spin();
22
23        return 0;
24 }
```

← not necessary to use timer, here only for debug

timer trigger

time-stamp printed for each callback call

accessing actual time

what to do each 0.1 s (10 Hz) specified by us

create the timer object

specify duration

notice: ros::Time::now() usually referred to system time, sometimes I'm interested on different timestamp → Robot time of simulation

{ When using Timers you don't need to modify CMakeLists.txt or package.xml }

- **parameters** to simplify the changing of some variable inside the code we use parameters, so we don't need to recompile all each time a change happen

  ↳ multiple way to SET param { launch file / cmd line / rqt_configure

  and different way to USE it

Example

param_first.cpp

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

#include <sstream>


int main(int argc, char **argv){

        ros::init(argc, argv, "param_first");
        ros::NodeHandle n;

        ros::Publisher chatter_pub = n.advertise<std_msgs::String>("parameter", 1000);
        std::string name;
        n.getParam("/name", name);

        ros::Rate loop_rate(10);



        while (ros::ok()){

                std_msgs::String msg;


                msg.data = name;

                ROS_INFO("%s", msg.data.c_str());

                chatter_pub.publish(msg);

                ros::spinOnce();

                loop_rate.sleep();

        }

        return 0;
}
```

publisher object, on "parameter" topic will publish the msg

getParam to take the parameter, this is done before the while loop → setted up once, fixed when running the node getted before while

we can have a getParam inside the while

↓

this is a Bad practice, process consuming!

← { good code only if param setted only one time! }

to set the "name" parameter { from cmd line / from launch file

- by launch file

param_set.launch

```
<launch>
  <param name="name" value="second"/>
  <node pkg="parameter_test" name="param_first" type="param_first" output="screen" />

</launch>
```

specify the value of param, info received by node

← running the node and redirect ros info to the terminal

- by cmd line

>> rosparam set name "value" to initialize the value

>> rosparam get name to take name value

IF we try to change dynamically the parameter while node running, Nothing happen!

We need a proper way for dynamical parameter changes → dynamic reconfigure!

Lab 4: Using classes (PUB/SUB same node), TF, bag files

• Using classes: PUB/SUB im same node

how to properly write your code where you have more nodes acting simultaneously (using c++ classes)

↓

until now a node was pub OR sub → we can do both on same node creating a class containing pub/sub

(EXAMPLE) subscribe to two different unsynchronized topics and republish it

test_pub.cpp

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <stdlib.h>

#include <sstream>


int main(int argc, char *argv[])
{

        ros::init(argc, argv, "publisher");

        ros::NodeHandle n;

        ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
        ros::Publisher chatter_pub2 = n.advertise<std_msgs::String>("chatter2", 1000);

        ros::Rate loop_rate(10);

        int count = 0;
        while (ros::ok())
        {

                std_msgs::String msg;

              std::stringstream ss;
                ss << "hello world " << count;
                msg.data = ss.str();

                ROS_INFO("%s", msg.data.c_str());

                if (rand() % 10 <6){
                        chatter_pub.publish(msg);
                }
                if (rand() % 10 <2){
                        chatter_pub2.publish (msg);
                }


                ros::spinOnce();

                loop_rate.sleep();
                ++count;
        }

        return 0;
}
```

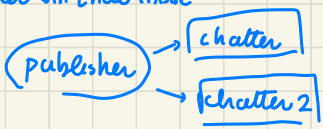
initialize publisher node

create the two publisher on that node

set loop Hz as 10Hz

↑diffent topics where publishing

publish on data on random instant



this node just publish on two topics

test_sub.cpp

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

class pub_sub
{

std_msgs::String messagio;
std_msgs::String messagio2;

private:
ros::NodeHandle n;

ros::Subscriber sub;
ros::Subscriber sub2;
ros::Publisher pub;
ros::Timer timer1;

public:
        pub_sub(){
        sub = n.subscribe("/chatter", 1, &pub_sub::callback, this);
        sub2 = n.subscribe("/chatter2", 1, &pub_sub::callback2, this);
        pub = n.advertise<std_msgs::String>("/rechatter", 1);
        timer1 = n.createTimer(ros::Duration(1), &pub_sub::callback1, this);


}
void callback(const std_msgs::String::ConstPtr& msg){
messagio=*msg;

}

void callback2(const std_msgs::String::ConstPtr& msg){
messagio2=*msg;

}
```

all ROS code inside this class!

2 subscribers
1 Publisher

timer to synchronize the data

on the constructor:

subscribing to topics of test_pub

publish on new topic

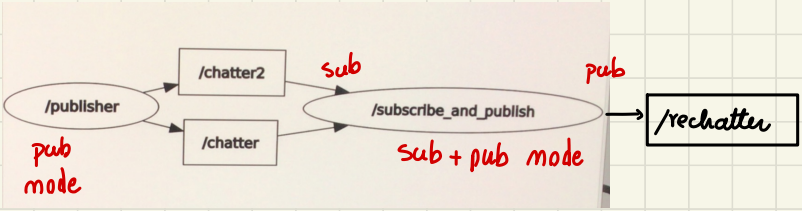
trigger timer event

save received message

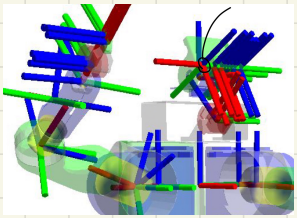
```cpp
void callback1(const ros::TimerEvent&)
{
        pub.publish(messagio);
        pub.publish(messagio2);
        ROS_INFO("Callback 1 triggered");
}




};

int main(int argc, char **argv)
{
        ros::init(argc, argv, "subscribe_and_publish");
        pub_sub my_pub_sub;
        ros::spin();
        return 0;
}
```

publish msg when timer trigger

all run on the main simply calling the constructor after init the node



- **TF** powerfull ROS tool to handle frame transformation ↴
  (so to compute relative position of different components of your environment)



usually ROBOT structure is composed of multiple joints, each one as static / dynamic transformation with another joint
→ lot of computations / transformation needed to compute rel pos. between joints
↓ ROS provide TF tools for this!
(with a tree representation)

in { Automomous Velicles }

- fixed reg F sensor
- base reg frame
- world reg frame
- map reg frame

→ frames described in a tree, each one with a transformation between itself and father / child
↓
and to convert from a frame to another we need MATH ☹ (roto-traslation combinations..)
↓
IF full tranformation tree is available : all becomes simpler
↓
ROS offers tools to analyze the transform tree

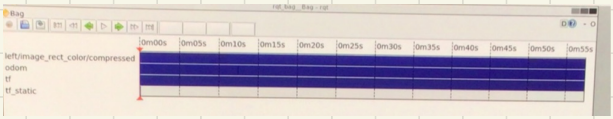
`>>rosrun rqt_tf_tree rqt_tf_tree` show current time tf tree
↑
we can test this tool using .bag file recordings

HOW TO USE .bag FILES ".bag" is a standard way to record data inside ROS → record all that is happening inside ROS on that time (topics, timestamps, messages..)

`>>rosbag play bag_file.bag` to play the file once on terminal
⇓
you can use additional options to play it in loop, set the speed rate, use the proper time reference

`>>rosbag play [options] bag_file.bag`
↓
"-l" : play in loop
"-r [num]" : to play num times faster
"--clock" : to get ROS timestamp of recording instead of system time

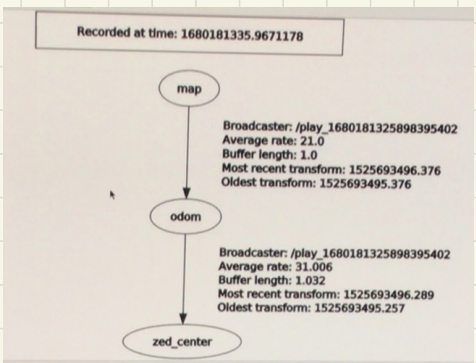
`>> rqt_bag` to proper visualize the bag file with a GUI



in this way you see when the msg are published respect timestamp + topic of ROS net

`>> rosbag info bag_name` to look bag file info

then, to see the TF tree graphically: `>> rosrun rqt_tf_tree rqt_tf_tree`
respect the playing bag file.



..... more dyn components

you can have { STATIC / DINAMIC } TF

(some transformation are static while others dynamic)

- STATIC TF are published only once at the beginning → only one tf_static msg initially
- DYNAMIC TF get updated during time (when restart recording, by refresh the TF tree you get Dynamic TF)
  ↳ tf_dynamic published continuously

HOW TO USE AND PUBLISH THIS TF ?

pub.cpp : node which receive data from a turtlesim node and publish its pose back as TF msg

```cpp
#include "ros/ros.h"
#include "turtlesim/Pose.h"
#include <tf/transform_broadcaster.h>

class tf_sub_pub{
public:
        tf_sub_pub(){
        sub = n.subscribe("/turtle1/pose", 1000, &tf_sub_pub::callback, this);
}


void callback(const turtlesim::Pose::ConstPtr& msg){
  tf::Transform transform;
  transform.setOrigin( tf::Vector3(msg->x, msg->y, 0.0) );
  tf::Quaternion q;
  q.setRPY(0, 0, msg->theta);
  transform.setRotation(q);
  br.sendTransform(tf::StampedTransform(transform, ros::Time::now(), "world", "turtle"));
}

private:
  ros::NodeHandle n;
  tf::TransformBroadcaster br;
  ros::Subscriber sub;
};


int main(int argc, char **argv){
  ros::init(argc, argv, "subscribe_and_publish");
  tf_sub_pub my_tf_sub_bub;
  ros::spin();
  return 0;
}
```

sub take info from pose topic

take position message and convert it into standard ROS transformation as position + orientation

create transform

z = 0 planar

orient. as quaternion

roll pitch yaw

orientation

3D space, set origin of transform., set x,y,z

publish by broadcaster

object to publish a TF

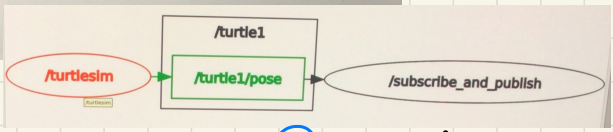
timestamp

route ref. frame

new frame → publish transform by route "world" to "turtle"

- subscribe to turtlesim/pose
- convert pose to a transform
- publish transform referred to world frame

+ CMakeLists.txt modification

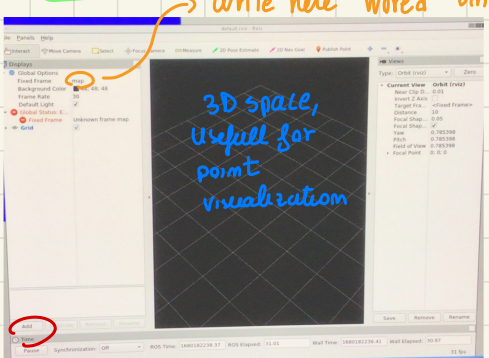


convert position to TF to visualize it

We can use graphical ROS tools to visualise TF in a 3D space **rviz** standard ROS visualization tool

>> rviz

write here "world" on Fixed Frame, which is the one we are using (we are publishing TF between world and turtle)



↓

so you can see the turtle References mooving on world frame (received position broadcast on net)

↳ dynamic TF: require broadcast publish

static TF: you just use static_tf_ publisher, you don't need broadcast

↳ Add/By display type /TF → to visualize TF on your 3D space

STATIC TF TF published using launch file or by a ROS node command

↓

write static TF on launch file

↓ we can add other frame, as static TF reference on the legs (static because fixed with the body)

turtle.launch

~/robotics/src/tf/launch

```
<launch>
<node pkg="tf_examples" type = "tf_pub" name = "tf_pub"/>
<node pkg="turtlesim" type = "turtlesim_node" name = "turtlesim_node"/>
<node pkg="turtlesim" type = "turtle_teleop_key" name = "turtle_teleop_key"/>
<node pkg="tf" type="static_transform_publisher" name="back_right" args="0.3 -0.3 0 0 0 0 1 turtle FRleg 100" />
<node pkg="tf" type="static_transform_publisher" name="front_right" args="0.3 0.3 0 0 0 0 1 turtle FLleg 100" />
<node pkg="tf" type="static_transform_publisher" name="front_left" args="-0.3 0.3 0 0 0 0 1 turtle BLleg 100" />
<node pkg="tf" type="static_transform_publisher" name="back_left" args="-0.3 -0.3 0 0 0 0 1 turtle BRleg 100" />

</launch>

```

} start the node to use

} add static Trans for the 4 legs

↑ to publish the static tf, you need to specify for each frame

args = " x y z a b c d route_frame child_frame publish_freq "

pos — quaternions orient.

(publish TF on ROS launch file)

and on rviz you have more visualization option to properly see the frames...