**Robotic Operating System**

Index

[**Introduction** 3](#_heading=h.30j0zll)

[**What is ROS** 3](#_heading=h.1fob9te)

[**Definition** 3](#_heading=h.3znysh7)

[**Description** 3](#_heading=h.2et92p0)

[**Computation Graph level** 4](#_heading=h.tyjcwt)

[**ROS Master** 4](#_heading=h.3dy6vkm)

[**Nodes** 4](#_heading=h.1t3h5sf)

[**Executable python node.py** 4](#_heading=h.4d34og8)

[**Executable C++ node.cpp** 5](#_heading=h.2s8eyo1)

[**Communication** 5](#_heading=h.17dp8vu)

[**Topics** 5](#_heading=h.3rdcrjn)

[**Services** 6](#_heading=h.26in1rg)

[**Actions** 6](#_heading=h.lnxbz9)

[**Bags** 7](#_heading=h.1ksv4uv)

[**Launch files** 7](#_heading=h.44sinio)

[**Launch architecture** 8](#_heading=h.2jxsxqh)

[**Structure** 8](#_heading=h.z337ya)

[**Tags** 8](#_heading=h.3j2qqm3)

[**Attributes** 10](#_heading=h.1y810tw)

[**Extra functionalities** 11](#_heading=h.4i7ojhp)

[**Filesystem** 11](#_heading=h.2xcytpi)

[**Packages** 11](#_heading=h.1ci93xb)

[**package.xml** 12](#_heading=h.3whwml4)

[**CMakeLists.txt** 12](#_heading=h.qsh70q)

[**Message data structure** 13](#_heading=h.3as4poj)

[**.msg** 13](#_heading=h.1pxezwc)

[**.srv** 14](#_heading=h.2p2csry)

[**.action** 14](#_heading=h.3o7alnk)

[**Dependencies** 15](#_heading=h.23ckvvd)

[**Python modules** 15](#_heading=h.ihv636)

[**C++ libraries** 16](#_heading=h.32hioqz)

[**High-level** 17](#_heading=h.1hmsyys)

[**Coordinate Frames/Transform (tf/tf2)** 17](#_heading=h.41mghml)

[**Design** 17](#_heading=h.2grqrue)

[**Static transform broadcaster (Launch)** 18](#_heading=h.vx1227)

[**Robot Description Model (urdf)** 19](#_heading=h.3fwokq0)

[**Sources [TEMP]** 19](#_heading=h.1v1yuxt)

[**robot\_state\_publisher** 19](#_heading=h.4f1mdlm)

[**joint\_state\_publisher** 20](#_heading=h.2u6wntf)

[**Filters** 20](#_heading=h.19c6y18)

[**Sources [TEMP]** 20](#_heading=h.3tbugp1)

[**Plugins** 20](#_heading=h.1lhlbphkoyb6)

[**Sources [TEMP]** 20](#_heading=h.28h4qwu)

[**Client Libraries** 21](#_heading=h.nmf14n)

[**rospy** 21](#_heading=h.37m2jsg)

[**Initialization** 21](#_heading=h.1mrcu09)

[**Topics** 21](#_heading=h.46r0co2)

[**Services** 22](#_heading=h.2lwamvv)

[**Parameter server** 22](#_heading=h.111kx3o)

[**Logging** 23](#_heading=h.3l18frh)

[**Time/Duration** 24](#_heading=h.206ipza)

[**Exception** 24](#_heading=h.4k668n3)

[**TF** 25](#_heading=h.n5os6kz2mfva)

[**roscpp** 27](#_heading=h.2zbgiuw)

[**Sources [TEMP]** 27](#_heading=h.1egqt2p)

[**Tools** 27](#_heading=h.3ygebqi)

[**Rviz** 27](#_heading=h.2dlolyb)

[**rqt** 28](#_heading=h.sqyw64)

# **Introduction**

**Sources**

<http://wiki.ros.org/ROS/Introduction>

# **Computation Graph level**

## **ROS Master**

## **Nodes**

### **Executable python node.py**

1. Package structure

/*pkg\_name*

/nodes

*node*.py

CMakeLists.txt

package.xml

1. Make the node executable

$ chmod +x *node*.py

### **Executable C++ node.cpp**

1. Package structure

/*pkg\_name*

/src

node.cpp

/include

node.h

CMakeLists.txt

package.xml

1. Modify CMakeList.txt

include\_directories(

include

${catkin\_INCLUDE\_DIRS}

)

add\_excutable(node src/node.cpp)

## **Communication**

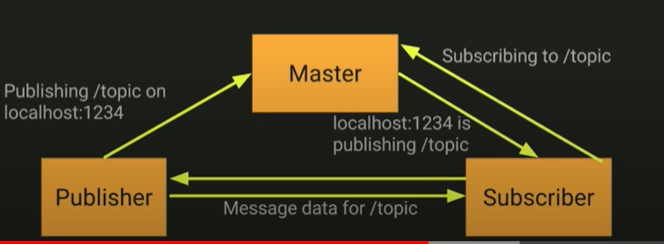
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Messages** | **Structure** | **Description** | **Applications** | **Examples** | **Tools** |
| ***Topics*** | .msg | Continue data streams | One-way continuous data flow | Sensor data, robot state | rostopic |
| ***Services*** | .srv | Blocking call for processing a request | Short triggers | Trigger change, request state, computations | rosservice |
| ***Actions*** | .action | Non-blocking preemptable goal-oriented tasks | Task executions and robot actions | Navigation, grasping, motion |  |
| ***Parameter Server*** | .param | Global constant parameters | Constant settings | Names, settings, calibration data, robot setup | rosparam |
| ***Dynamic reconfigure*** | .cfg | Local, changeable parameters | Tuning parameters | Controller parameter |  |

### **Topics**

Topics are named buses for unidirectional, streaming communication between nodes:

* ***Publisher*** – nodes that send data
* ***Subscriber*** – nodes interested in data

The master initializes the topic communication and tracks all the information.



**Rules**

* Any node can publish a message to any topic
* Any node can subscribe to any topic
* Any node can both publish and subscribe to any topics at the same time
* Multiple nodes can publish and subscribe to the same topic

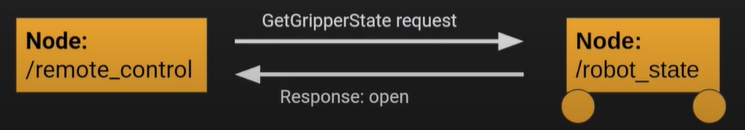
**Protocol**

Topics’ default protocol is TCPROS (TCP/IP-based). Another protocol is UDPROS (UDP-based) only for roscpp, that is a low-latency, lossy transport, so is best suited for tasks like teleoperation.

### **Services**

Services are a pair of messages exchanged between nodes:

* ***Client (Response)*** – node providing a service
* ***Server (Request)*** – node requesting a service which gives back the response



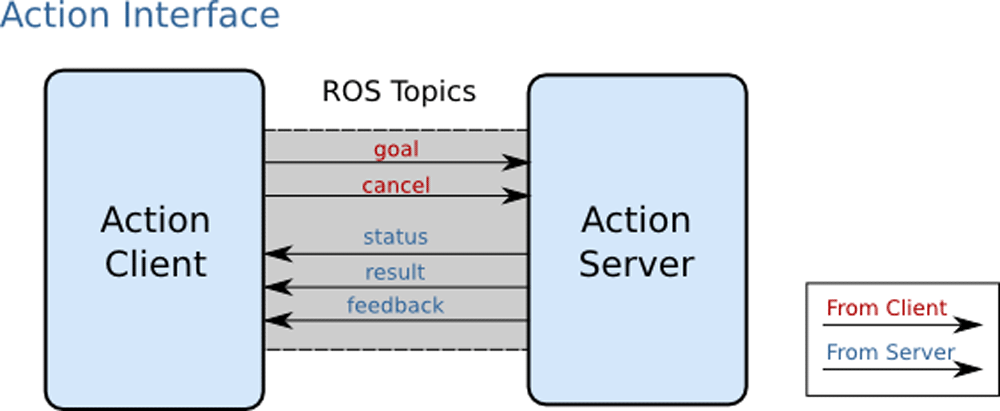
**Rules**

* Any node can implement services (server)
* Any node can call services (client)
* Servers call are synchronous/blocking

### **Actions**

Actions, like services, are messages exchanged between two nodes, providing additional features:

* ***ActionClient*** – node providing a task and the possibility to cancel that task
* ***ActionServer*** – node receiving a task which give back the status, the result and the feedback



**Protocol**

The used protocol is the “*ROS Action Protocol”* which is built on top of ROS messages (High-level).

**Rules**

* Goal - can be messages, services or parameters data structures used to send goals
* Cancel – used to send cancel requests
* Status – used to notify clients on the current state of every goal in the system
* Feedback - provides information about the incremental progress of a goal
* Result - send upon completion of the goal

## **Bags**

It’s a format for saving, storing and playback ROS message data in .bag files.

Bag files can be played back in ROS to the same topic they were recorded from, or even remapped to new topics.

## **Launch files**

### **Launch architecture**

In general, for multi-packages projects, the workflow consists in creating a hierarchical launch files:

* a local launch file starts a single node or couple of related nodes with their configurations
* (optional) a intermediate launch file for a single package
* a top-level launch file, independent from packages, executes all the launch files needed for that application

### **Structure**

<launch>

<!-- arguments definition -->

<arg name="arg\_1" default="1" doc="documentation" />

<arg name="arg\_2" value="0" doc="documentation" />

<!-- parameters definition -->

<param name="param\_1" value="0.0" />

<param name="param\_2" value="value" type="type" />

<param name="param\_3" textfile="$(find pkg\_name)/path/file.txt" />

<param name="param\_4" binfile="$(find pkg\_name)/path/file"/>

<param name="param\_5" command="$(find pkg\_name)/exe '$(find pkg\_name)/file.txt" />

<!-- loading/saving/deleting parameters -->

<rosparam command="load" file="$(find pkg\_name)/path/file.yaml" ns="namespace" />

<rosparam command="dump" file="$(find pkg\_name)/path/file.yaml" ns="namespace" />

<rosparam command="delete" file="$(find pkg\_name)/path/file.yaml" ns="namespace" />

<!-- environment variables -->

<env name="env\_variable" value="value" />

<!-- set a machine -->

<machine name="local\_machine" address="local\_hostname" env-loader="/opt/ros/$ROS\_DISTRO/env.sh" default="true" />

<machine name="remote\_machine" address="remote\_hostname" env-loader="/opt/ros/$ROS\_DISTRO/env.sh" default="false" user="username" password="password" timeout="10.0" />

<!-- launch a simple node -->

<node pkg="pkg\_name" type="node\_ex.py" name="node\_name" />

<node pkg="pkg\_name" type="node\_ex.py" name="node\_name" args="arg" ns="namespace" output="screen" machine="machine\_name" required="true" respawn="false" launch-prefix="option " >

<remap from="original\_name" to="new\_name" />

<env />

<rosparam />

<param />

</node>

<!-- launch a launch file -->

<include file="$(find pkg\_name)/path/launch\_file.launch" ns="namespace" >

<arg name="arg\_x" value="" />

</include>

</launch>

### **Tags**

**<node>**

It launches a ROS node.

* pkg (required) - the name of the pkg containing the node
* type (required) - the name of the executable file
* name (required) - the name of the node, overriding the name generated by the executable (ros\_init)
* respawn (optional) - default false; if true the node restarts when it terminates
* required (optional) - default false; if true, when the node terminates, roslaunch terminates all the nodes within
* output (optional) - screen (on the terminal), log
* ns (optional) - set namespace
* machine (optional) - machine name (tag) on which the node has to run
* launch-prefix (optional): insert a given prefix at the start of the command line that runs the node
* xterm -e gdb --args: run your node in a gdb in a separate xterm window
* gdb -ex run --args: run your node in gdb in the same xterm as your launch
* stterm -g 200x60 -e gdb -ex run --args: run your node in gdb in a new xterm window
* valgrind: run your node in valgrind
* xterm -e: run your node in a separate xterm window
* nice: nice your process to lower its CPU usage
* screen -d -m gdb --args: useful if the node is being run on another machine; you can then ssh to that machine and do screen -D -R to see the gdb session
* xterm -e python -m pdb: run your python node a separate xterm window in pdb for debugging; manually type run to start it
* yappi -b -f pstat -o : run your rospy node in a multi-thread profiler such as yappi
* /path/to/run\_tmux: run your node in a new tmux window; you'll need to create /path/to/run\_tmux with the contents

**<include>**

It launches another launch file.

Required arguments

* file – file path from the package

Optional arguments

* ns – import the file relative to the function namespace
* pass all arg
  + true - set all arguments according to the included file
  + false (default)

**<arg>**

It creates an argument acting as a variable which stores its value within the scope which is defined in. If defined in the top level launch file, they can be used via command-line or passed to another launch/file that “includes” it.

1. The arguments are defined at the beginning of the launch file in two ways:
   1. with a default value that can be overwritten when passed
   2. with a constant value which cannot be overwritten
2. The arguments are passed via:
   1. launch command in CLI
   2. launch file within <include> tag

Required arguments

* name – argument name
* default – default value
* value – value to assign

Optional arguments

* doc – documentation

**<param>**

It defines parameters.

Optional arguments

* value – define parameter value; if omitted it must be specified by a binfile, textfile or command
* type – define parameter type (str, int, double, bool, yaml)
* textfile – read a text file and store as string
* binfile – read stored as base64-encoded XML-RPC binary object
* command – the output of the command will be read and stored as a string

**<rosparam>**

It defines yaml files to load parameters from Parameter Server.

Optional arguments

* command – define a command (load, dump, delete)
* ns – scope the parameters to a specific namespace

**<remap>**

It allows to remap a ROS node from a name to another.

**<machine>**

It defines the machine on which the launch file will run, then it is useful to declare SSH and ROS environment variables settings for remote machines. It is valid for all the nodes launched after.

* name (required) - name to assign to machine, used by a node
* address (required) - hostname/IP address of machine
* env-loader (required) - specify environment file on remote machine
* default (optional) - sets this machine as the default to assign nodes to
* user (optional) -SSH username
* password (optional) -SSH password
* timeout (optional) - seconds before launch on the machine fails

**<env>**

It allows to set the environment variables (PATH, PYTHONPATH, ROS\_PACKAGE\_PATH). It can be defined for all the launch file (globally) or nested in “node”, “include”or “machine” tags (locally).

**<group>**

<http://wiki.ros.org/roslaunch/XML/group>

### **Attributes**

**Replacing attributes**

Roslaunch attributes can make use of substitution args, resolved prior to launching the file.

* Replace the value of a variable from the current environment. If it is not set, the launch fail.

$(env *ENV\_VARIABLE*)

* Replace the value of an environment variable if it is set. If default\_value is provided, it will be used if the environment variable is not set

$(optenv *ENV\_VARIABLE* *default\_value*)

* Specifies a package-relative path in ROS workspace.

$(find *pkg\_name*)

* It assigns the arg value (arg tag) in the same launch file that declares that argument.

$(arg *arg\_name*)

* It allows to evaluate arbitrary complex python expressions.

$(eval *function*)

**If/Unless attributes**

All the tags support two special attributes:

* If a variable is true, then it includes tag and its content

if=variable

* Unless a variable is true, then includes tag and its content

unless=variable

### **Extra functionalities**

**Node time delay**

<arg name="node\_start\_delay" default="1.0" />

<node name="listener" pkg="roscpp\_tutorials" type="listener" launch-prefix="bash -c 'sleep $(arg node\_start\_delay); $0 $@' " />

# **Filesystem**

## **Packages**

The package is the smallest unit that is possible to build in ROS and the main unit for organizing software, which can contain:

* ROS runtime processes (nodes)
  + scripts: contains executables python scripts
  + src: contains C++ cource code
  + include: contains headers and libraries needed
  + launch: contains launch files
* ROS-dependent libraries (dependencies)
  + msg: custom messages structure definition
  + src: custom services structure definition
  + action: custom actions structure definition
  + config: configuration files
* Configuration files
  + package.xml: manifest file
  + CMakeLists.txt: CMake build file
* Anything else that is usefully organized together

**Package structure**

catkin\_ws

/src

**/pkg\_name**

/config

file.config

/include

/pkg\_name

header.hh

/scripts

node.py

/src

node.cpp

/launch

node.launch

/msg

message.msg

/srv

message.srv

/param

message.param

/action

message.action

package.xml

CMakeLists.txt

### **package.xml**

The package manifest provides metadata about a package, including its name, version, description, license information, dependencies, and other meta information like exported packages.

<package format="2">

<name>*pkg\_name*</name>

<version>1.2.4</version>

<description>*pkg\_description*</description>

<maintainer email="*maintainer\_email*">*maintainer\_name*</maintainer>

<license>[sw license]</license>

<url>http://ros.org/wiki/*pkg\_name*</url>

<author>*author\_name*</author>

<buildtool\_depend>catkin</buildtool\_depend>

<depend>roscpp</depend>

<depend>rospy</depend>

<depend>...</depend>

</package>

### **CMakeLists.txt**

The CmakeLists file describes the build instructions for the Cmake.

########## Informations ##########

cmake\_minimum\_required()

project(pkg\_name)

add\_compile\_options(-std=c++11) # for C++ compiling

########## Dependencies ##########

find\_package(catkin REQUIRED COMPONENTS

rospy

roscpp

...

)

########## Catkin specific configuration ##########

catkin\_package(

DEPENDS rospy roscpp ...

CATKIN\_DEPENDS rospy roscpp ...

)

include\_directories(

include

${catkin\_INCLUDE\_DIRS}

)

## **Message data structure**

The message data structure is the specific format used to compose messages:

* .msg – topic data structure

# defining topic message

message\_type message\_name

* .srv – service data structure

# defining request message

request\_type request\_name

---

# defining response message

response\_type response\_name

* .action – action data structure

# goal definition

goal\_type goal\_name

---

# result definition

result\_type result\_name

---

# feedback

feedback\_type feedback\_name

### **.msg**

* + - 1. Create messages dependencies package “msg\_dep”

catkin create pkg msg\_dep -c message\_generation rospy ...

* + - 1. Package architecture

/msg\_dep

/msg

message.msg

CMakeLists.txt

package.xml

* + - 1. Create the msg message “message.msg”
      2. CMakeLists.txt

find\_package(catkin REQUIRED COMPONENTS

rospy

message\_generation

)

# Messages

add\_message\_files(

DIRECTORY msg

FILES message.msg

)

generate\_messages(

DEPENDENCIES ...

)

catkin\_package(

CATKIN\_DEPENDS rospy message\_runtime

)

* + - 1. package.xml

<build\_depend>message\_generation</build\_depend>

<build\_export\_depend>message\_runtime</build\_export\_depend>

<exec\_depend>message\_runtime</exec\_depend>

### **.srv**

* + - 1. Create messages dependencies package “msg\_dep”

catkin create pkg msg\_dep -c message\_generation rospy rospy ...

* + - 1. Package architecture

/msg\_dep

/srv

message.srv

CMakeLists.txt

package.xml

* + - 1. Create the srv message “message.srv”
      2. CMakeLists.txt

find\_package(catkin REQUIRED COMPONENTS

rospy

message\_generation

)

# Services

add\_service\_files(

DIRECTORY srv

FILES message.srv

)

generate\_messages(

DEPENDENCIES ...

)

catkin\_package(

CATKIN\_DEPENDS rospy message\_runtime

)

* + - 1. package.xml

<build\_depend>message\_generation</build\_depend>

<build\_export\_depend>message\_runtime</build\_export\_depend>

<exec\_depend>message\_runtime</exec\_depend>

### **.action**

* + - 1. Create messages dependencies package “msg\_dep”

catkin create pkg msg\_dep -c message\_generation actionlib actionlib\_msgs rospy ...

* + - 1. Package architecture

/msg\_dep

/action

message.action

CMakeLists.txt

package.xml

* + - 1. Crete the action message “message.action”
      2. CMakeLists.txt

find\_package(catkin REQUIRED COMPONENTS

message\_generation

actionlib\_msgs

rospy

)

# Actions

add\_action\_files(

DIRECTORY action

FILES massage.action

)

generate\_messages(

DEPENDENCIES

actionlib\_msgs

...

)

catkin\_package(

DEPENDS message\_runtime actionlib\_msgs

)

* + - 1. package.xml

<build\_depend>message\_generation</build\_depend>

<build\_export\_depend>message\_runtime</build\_export\_depend>

<depend>actionlib</depend>

<depend>actionlib\_msgs</depend>

<exec\_depend>message\_runtime</exec\_depend>

## **Dependencies**

Python modules and C++ libraries are dependencies that can be imported in any other package in the workspace.

### **Python modules**

* + - 1. Creating dependency package

catkin create pkg dep\_pkg -c rospy

* + - 1. Package architecture

/dep\_pkg

/scripts

/dep\_pkg

\_\_init\_\_.py

module.py

CMakeLists.txt

package.xml

setup.py

* + - 1. Create the python module “module.py”
      2. Create the “\_\_init\_\_.py” empty file
      3. Create the “setup.py” file which allows the modules defined in “src” directory, to be visible by other packages in the workspace and insert the proper package name

from distutils.core import setup

from catkin\_pkg.python\_setup import generate\_distutils\_setup

setup\_args = generate\_distutils\_setup(

packages = [‘dep\_pkg’], # package name

package\_dir = {‘’:scripts}, # module folder name

)

setup(\*\*setup\_args)

* + - 1. CMakeLists.txt

find\_package(catkin REQUIRED COMPONENTS

rospy

)

catkin\_python\_setup()

* + - 1. package.xml

<depend>rospy</depend>

### **C++ libraries**

* + - 1. Creating dependency package

catkin create pkg dep\_pkg -c roscpp

* + - 1. Package architecture

/dep\_pkg

/include

/dep\_pkg

lib.h

/src

CMakeLists.txt

package.xml

setup.py

* + - 1. CMakeLists.txt

catkin\_package(

INCLUDE\_DIRS include

LIBRARIES ${PROJECT\_NAME}

CATKIN\_DEPENDS roscpp

)

include\_directories(

${PROJECT\_NAME}

include

${catkin\_INCLUDE\_DIRS}

)

add\_library(

${PROJECT\_NAME}

src/lib.cpp

)

target\_link\_libraries(

${PROJECT\_NAME}

${catkin\_LIBRARIES}

)

install(

TARGETS ${PROJECT\_NAME}

ARCHIVE DESTINATION ${CATKIN\_PACKAGE\_LIB\_DESTINATION}

LIBRARY DESTINATION ${CATKIN\_PACKAGE\_LIB\_DESTINATION}

RUNTIME DESTINATION ${CATKIN\_GLOBAL\_BIN\_DESTINATION}

)

install(

DIRECTORY include/${PROJECT\_NAME}/

DESTINATION ${CATKIN\_PACKAGE\_INCLUDE\_DESTINATION}

)

* + - 1. package.xml

<depend>roscpp</depend>

# **High-level**

## **Coordinate Frames/Transform (tf/tf2)**

**Sources**

<http://wiki.ros.org/tf2>

<https://github.com/ros/geometry2>

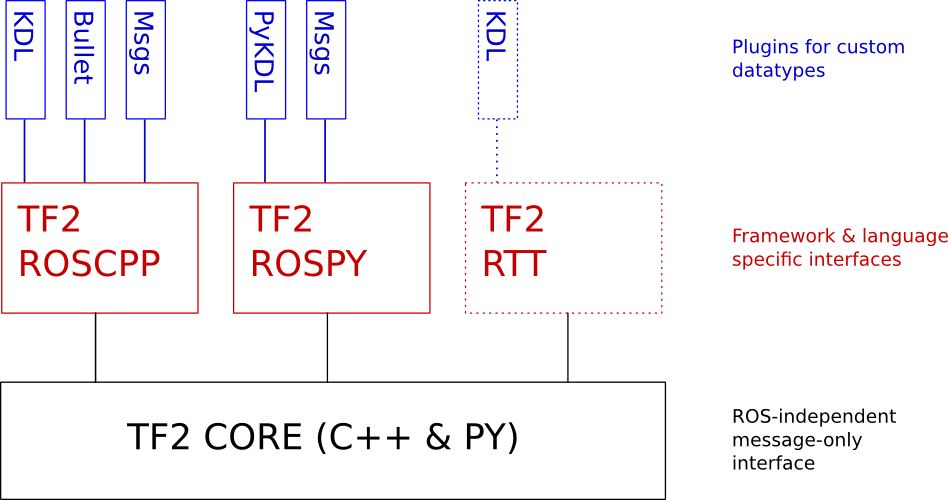
The ***tf2 library*** provides functionalities to:

* keep track of multiple coordinate frame over time
* maintain the relationship between coordinate frames in a tree structure buffered in time
* transform points, vectors, etc between any two coordinate frames at any desired point in time

There are two fundamentals tasks that tf2 is used for:

* ***listening for transforms***: receiving and buffer all coordinates frames that are broadcasted in the system and query for specific transforms between frames
* ***broadcasting transforms***: sending out the relative pose of coordinate frames to the rest of the system

### **Design**



1. A distributed system: Everything is broadcast and reassembled at end consumer points.
2. Only transform data between coordinates frames at the time of use
3. Support queries on data which are timestamped at times other than the current time: Interface class stores all transform data in memory and traverses tree on request
4. Only have to know the name of the coordinate frame to work with data: Use string frame\_ids as unique identifiers
5. The system doesn't need to know about the configuration before hand and can handle reconfiguring on the fly: Use directed tree structure
6. Core is ROS agnostic: Core library is C++ class. A second class provides ROS interface and instantiates the core library
7. Thread Safe Interface: Mutexes around data storage for each frame. Mutexes around frame\_id lookup map
8. Multi-Robot Support: Use a tf\_prefix similar to a namespace for each robot
9. Native Datatype Interfaces: There is a tf2::convert(A, B) templated method that converts from type A to type B using the geometry\_msg types as the common factor

template <class A, class B>

void convert(const A& a, B& b)

{

fromMsg(toMsg(a), b);

}

### **Static transform broadcaster (Launch)**

* Static coordinate transform to tf2 using an x/y/z offset in meters and yaw/pitch/roll in radians

<node pkg="tf2\_ros" type="static\_transform\_publisher" name="link1\_broadcaster" args=" x y z yaw pitch roll parent\_frame\_id child\_frame\_id " />

* Static coordinate transform to tf2 using an x/y/z offset in meters and quaternions

<node pkg="tf2\_ros" type="static\_transform\_publisher" name="link1\_broadcaster" args=" x y z qx qy qz qw parent\_frame\_id child\_frame\_id " />

## **Robot Description Model (urdf) [WIP]**

The ***urdf package*** defines an XML format for representing a robot model and provides a C++ parser.

### **Sources [TEMP]**

catkin\_ws

/src

/robot\_description

CMakeLists.txt

package.xml

/urdf

robot\_description.urdf

Robot model are defined in XML format:

* kinematic description
* visual representation (mesh)
* collision model

URDF generation can be scripted with XACRO

URDF is stored on the parameter server under /robot\_description

<https://www.youtube.com/watch?v=pD1RwMZ1YgM&list=PLK0b4e05LnzbHiGDGTgE_FIWpOCvndtYx&index=8&t=0s>

<https://www.youtube.com/watch?v=KKXF4NKAZy8&list=PLK0b4e05LnzbHiGDGTgE_FIWpOCvndtYx&index=10&t=0s>

### **robot\_state\_publisher**

It uses the URDF in rebot\_description and the joint positions from the joint\_states topic, to calculate the forward kinematic of the robot and publish it via tf.

It’s possible to use this publisher as:

* standalone ROS node:

robot\_state\_publisher

* library

<http://wiki.ros.org/robot_state_publisher>

<http://wiki.ros.org/robot_state_publisher/Tutorials/Using%20the%20robot%20state%20publisher%20on%20your%20own%20robot>

<http://wiki.ros.org/urdf/Tutorials/Using%20urdf%20with%20robot_state_publisher>

### **joint\_state\_publisher**

<http://wiki.ros.org/joint_state_publisher>

<https://answers.ros.org/question/275079/joint-state-publisher-and-robot-state-publisher/>

<https://answers.ros.org/question/186976/how-to-combine-joint-state-publisher-with-urdf/>

<https://www.youtube.com/watch?v=J0ErFG6FIps&list=PLK0b4e05LnzbHiGDGTgE_FIWpOCvndtYx&index=12&t=0s>

## **Filters [WIP]**

The ***filter package*** provides a C++ library for processing data using a sequence of filters.

### **Sources [TEMP]**

<http://wiki.ros.org/filters>

<http://wiki.ros.org/message_filters?distro=melodic>

<http://wiki.ros.org/geometry2?distro=melodic>

<http://wiki.ros.org/urdf>

<http://wiki.ros.org/urdf_tutorial?distro=groovy>

<http://wiki.ros.org/robot?distro=melodic>

## **Plugins [WIP]**

The ***pluginlib package*** provides a library for dynamically loading libraries in C++ code.

### **Sources [TEMP]**

<http://wiki.ros.org/pluginlib>

# **Client Libraries**

## **rospy**

### **Initialization**

Importing rospy

import rospy

Messages generation import

- package\_msgs/msg/message.msg → package\_msgs.msg.message

- package\_msgs/srv/message.srv → package\_msgs.srv.message

import package\_msgs.msgs

from package\_msgs.srv import message

Initializing ROS node

rospy.init\_node(*node\_name*, anonymous=True)

Testing for publishing

while not rospy.is\_shutdown():

...

Testing for subscribing

rospy.spin()

### **Topics**

**Publisher**

Publisher initialization

pub = rospy.Publisher(“topic\_name”, package.msg.message, queue\_size=10)

Note: queue\_size is the maximum size of messages in queue before they are dropped. The bigger the queue, the more number of messages received are stored. This is useful in asynchronous behaviour.

Publishing

pub.publish(message\_name)

Creating a message

* No arguments: it is useful for nested fields

msg = package.msg.message()

msg.field = value

* In-order arguments: it is useful for messages changes notification

msg = package.msg.message(value)

* Keyword arguments: it’s recommended since it is resilient to messages changes

msg = package.msg.message(field=value)

**Subscriber**

Subscriber initialization

rospy.Subscriber(“topic\_name”, package.msg.message, callback)

Callback

def callback(msg\_data):

msg = msg\_data

### **Services**

**Calling services (Client)**

Waiting till a service is available

rospy.wait\_for\_service(“service\_name”, timeout=s)

Service definition

srv = rospy.ServiceProxy(“service\_name”, package.srv.message, persistent=False, headers=None)

Service request and response

* Explicit style

req = package.msg.message()

resp = service\_name(req)

* In-order implicit style

resp = service\_name(value)

* Keyword implicit style

resp = service\_name(arg=value)

**Provide services (Server)**

Server

srv\_name = rospy.Service(“service\_name”, package.srv.message, callback)

Callback

def callback(req)

return package.srv.message

Shutdown

* Explicit shutdown

srv\_name.sutdown(“Service shoutdown”)

* Wait for shutdonw

srv\_name.spin()

### **Parameter server**

**Get parameters**

Get a parameter

rospy.get\_param(“param\_name”)

* “/global\_name”
* “relative\_name”
* “~private\_name”
* “default\_param”, “default\_value”

Get a set of parameters

x, y = param\_name[‘par\_1’], param\_name[‘par\_2’]

**Set parameters**

Using rospy and raw objects

rospy.set\_param(“param\_name”, value)

Using rosparam and yaml strings

rosparam.set\_param(“param\_name”, “value”)

**Existence**

rospy.has\_param(“param\_name”)

**Delete parameters**

rospy.delete\_param(“param\_name”)

**Retrieve list of parameters**

Retrieve list of existing parameters names on the Parameter Server

rospy.get\_param\_names()

**Searching for parameters keys**

rospy.search\_param(param\_name)

### **Logging**

Debug

rospy.logdebug(msg, \*args, \*\*kwargs)

Info

rospy.loginfo(msg, \*args, \*\*kwargs)

Warn

rospy.logwarn(msg, \*args, \*\*kwargs)

Error

rospy.logerr(msg, \*args, \*\*kwargs)

Fatal

rospy.logfatal(msg, \*args, \*\*kwargs)

### **Time/Duration**

**Get current time**

rospy.Time.now()

rospy.get\_rostime()

**Create time instance**

t = rospy.Time(sec, nsec)

**Sleeping/Rate**

rospy.sleep(sec)

rospy.Rate(Hz)

**Timer**

rospy.Timer(period, callback, oneshot=False)

### **Exception**

Base exception class for ROS clients

except ROSException:

Exception for message serialization errors

except ROSSerializationException:

Exception for errors initializing ROS state

except ROSInitException:

Exception for operations that interrupted. This is most commonly used with rospy.sleep() and rospy.Rate

except ROSInterruptException:

Base class for exceptions that occur due to internal rospy errors (i.e. bugs).

except ROSInternalException:

Errors related to communicate with ROS [Services](http://wiki.ros.org/Services)

except ServiceException:

Errors related to parameter server

except KeyError:

### **TF**

**Static transform broadcaster**

1. Import modules

import rospy

from geometry\_msgs.msg import TransformStamped

from tf2\_ros import StaticTransformBroadcaster

1. Transform object

static\_transform = TransformStamped()

1. Broadcaster object

static\_broadcaster = StaticTransformBroadcaster()

static\_broadcaster.sendTransform(static\_transform)

**Transform broadcaster**

1. Import modules

import rospy

from geometry\_msgs.msg import TransformStamped

from tf2\_ros import TransformBroadcaster

1. Subscribing to a topic to get the pose (Pose) from the callback function

rospy.Subscriber(‘/topic’, module, callback)

1. Transform object

transform = TransformStamped()

1. sending the transform

broadcaster = TransformBroadcaster()

broadcaster.sendTransform(t)

**Transform listener**

1. Creating the listener object, which receive tf2 transformations, and buffer them

tfBuffer = Buffer()

listener = TransformListener(tfBuffer)

1. Query the listener for the transformation at a specific time

transform = tfBuffer.lookup\_transform(

target\_frame=turtle\_name,

target\_time=rospy.Time.now(),

source\_frame='carrot1',

source\_time=rospy.Time.now() - rospy.Duration(5.0),

fixed\_frame='world',

timeout=rospy.Duration(1.0)

)

Given the transformation:

1. from this frame
2. (optional) at this time
3. to this frame
4. at this time
5. (optional) specify the frame that does not change in time
6. (optional) timeout

Time debugging

1. Get the latest available transform

listener.lookupTransform("/turtle2",

"/turtle1",

rospy.Time(0),

transform);

1. Wait for the transform at the current time

listener.waitForTransform(

"/turtle2",

"/turtle1",

rospy.Time.now(),

rospy.Duration(1.0));

listener.lookupTransform(

"/turtle2",

"/turtle1",

rospy.Time.now(),

transform);

**Rotations**

import rospy

from tf.transfomations import \*

q\_2 = quaternion\_multiply(q\_rot, q\_1)

**Representation conversions**

* + 1. Euler RPY (radiant) to Quaternions

import rospy

from tf.transformations import transformations

q = transformations.quaternion\_from\_euler(R, P, Y)

* + 1. Quaternions to Euler RPY (radiant)

from tf.transformations import transformations

rot = transformations. euler\_from\_quaternion(q)

**Pose transform**

import rospy

from tf2\_geometry\_msgs import do\_transform\_pose

# Transform pose message

pose\_transformed = do\_transform\_pose(pose\_msg, transform)

## **Roscpp [WIP]**

**TEMP (Topics)**

[http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28c%2B%2B%29](http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber(c%2B%2B))

<https://www.youtube.com/watch?v=2Cmdu6mkxD0&list=PLJNGprAk4DF5PY0kB866fEZfz6zMLJTF8&index=4&t=122s>

**TEMP (Service)**

[http://wiki.ros.org/ROS/Tutorials/WritingServiceClient%28c%2B%2B%29](http://wiki.ros.org/ROS/Tutorials/WritingServiceClient(c%2B%2B))

<https://www.youtube.com/watch?v=XrMFh3xQcmM&list=PLJNGprAk4DF5PY0kB866fEZfz6zMLJTF8&index=7&t=0s>

## **Sources [TEMP]**

<http://wiki.ros.org/Client%20Libraries>

<http://wiki.ros.org/Implementing%20Client%20Libraries>

# **Tools**

## **Rviz [WIP]**

RViz is a 3D visualization tool for ROS.

Resources

<http://wiki.ros.org/visualization>

<http://wiki.ros.org/visualization/Tutorials>

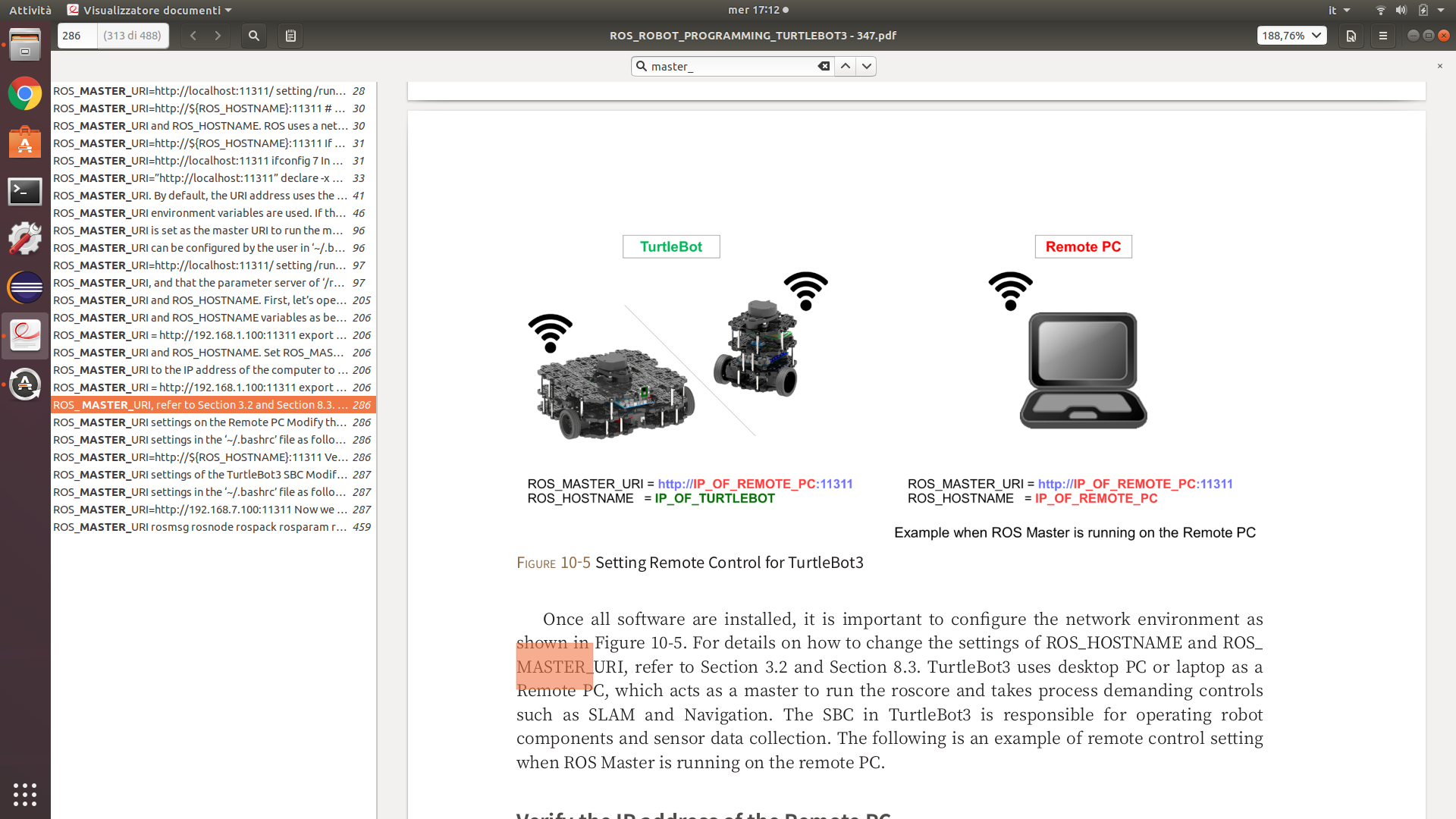
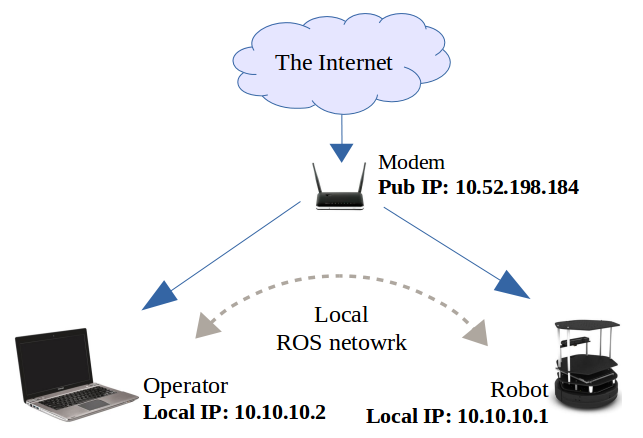
<https://www.youtube.com/watch?v=N1QsVM5imsU&list=PLK0b4e05LnzbHiGDGTgE_FIWpOCvndtYx&index=9&t=0s>

<https://www.youtube.com/watch?v=OiNll0YJ_Us&list=PLK0b4e05LnzbHiGDGTgE_FIWpOCvndtYx&index=11&t=0s>

## **rqt [WIP]**

# **Extra**

## **ROS Networking**



Requirements:

1. A common network
2. ROS on every machine
3. IP address of every machine
4. **IP addresses configuration**

All machines have to be configured to use the same master (ROS\_MASTER\_URI)

1. Check IP

>> ifconfig

1. Check machine hostname

>> hostname

>> hostname -i

1. Export environment variable

>> gedit ~/.bashrc

export ROS\_MASTER\_URI=http://*master\_hostname*:11311

export ROS\_HOSTNAME=*machine\_hostname*

export ROS\_IP=*machine\_ip*

1. (Optional) Set the proper local IP and hostname if hostname cannot be resolved

>> sudo gedit /etc/hosts

>> sudo gedit /etc/hostname

1. **Connectivity check**

Complete bi-directional connectivity between all pairs of machines on all ports

1. Check I: self ping

ssh *master\_hostname*

ping *master\_hostname*

1. Check II: mutual ping

ssh *master\_hostname*

ping *machine\_hostname*

1. Check III: port connection

ssh *master\_hostname*

netcat -l 1234

ssh *machine\_hostname*

netcat *master\_hostname* 1234

1. **Start master**

Start one master (roscore) on a selected machine

ssh *master\_hostname*

roscore

## **Remote ROS Networking [WIP]**

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