Robotic Operating System

Index

Introduction

What is ROS

Sources

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

Definition

ROS is an open-source, meta-operating system for robotic systems, providing:

- hardware abstraction
- low-level device control
- implementation of commonly used functionality
- message-passing between processes
- package management
- tools and libraries for obtaining, building, writing, and running code across multiple computers

The purpose of ROS is:

- Support code reuse in robotics research and development
- Thin: code written for ROS can be used with other robot software frameworks
- ROS-agnostic libraries: the preferred development model is to write ROS-agnostic libraries with clean functional interfaces
- Language independence: implementation in different programming language (most used C++ and Phyton)
- Easy testing: built-in unit/integration test framework called *rostest*
- Scaling: ROS is appropriate for large runtime systems and for large development processes.

Description

ROS is a distributed framework of processes called ***nodes*** which enable executables to be individually designed and loosely coupled at runtime. Communication between nodes is implemented by means of ***topics***, ***services***, ***actions*** and ***parameters***, for each of which is defined a data structure. Messages can be recorded through ***bag*** functionality which allows to reproduce offline them. The ***launch*** functionality allows to run multiple nodes simultaneously and settings their parameters. This represents the ***computational graph level***.

Nodes and in general anything that is useful to organize together, are organized into ***packages***, which are the smallest unit that is possible to build in ROS, fulfilling a defined function and defined by their own configurational files. This is the base of the ***filesystem level***. Packages can be grouped into ***stacks***, which can be built as packages in a nested framework. Furthermore, within packages can be defined ***message structures***.

ROS features are accessible via code through ***client libraries***, allowing to implement nodes in different coding languages. Actually, the most supported are C++ (roscpp) and Python (rospy), alongside LISP (roslisp) and Matlab/Simulink.

The ***community level*** is the essence of ROS. Packages and stacks can be shared across the ROS community through a federated system of online ***repositories*** (https://www.ros.org/browse/list.php).

Furthermore, a ***higher-level architecture*** is necessary to build larger systems on top of ROS.

Computation Graph level

The Computation Graph is the peer-to-peer network of ROS processes made up by concepts implemented in the ROSComm repository.

ROS Master

The ROS Master provides name registration and lookup to the rest of the Computation Graph.

It acts as a *nameservice*, storing topics and services registration information for nodes. Nodes communicate with the Master to report their registration information.

As these nodes communicate with the Master, they can receive information about other registered nodes and make connections as appropriate.

Nodes connect to other nodes directly; the Master only provides lookup information.

Nodes communication is established over the TCPROS protocol, which uses standard TCP/IP sockets.

There is always only one master in the network, nodes must know network address of which (ROS_MASTER_URI).

Nodes

Nodes are basic processes that perform computation, written through ROS client library (roscpp, rospy). Each node has its own name (graph resource name) that identify it and a node type (package resource name).

They are combined into a graph and communicate with each other using:

- topics
- services
- actions

Furthermore, they have variables which can be managed using:

- parameter server
- dynamic reconfigures

Executable python node.py

1. Package structure

```
/pkg_name
    /nodes
        node.py
    CMakeLists.txt
    package.xml
```

2. 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
```

2. Modify CMakeList.txt

```
include_directories(
  include
  ${catkin_INCLUDE_DIRS}
)

add_excutable(node src/node.cpp)
```

Communication

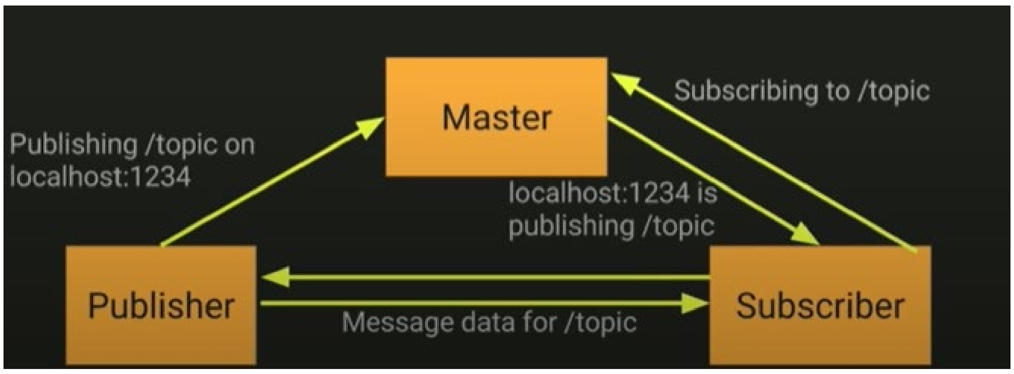
Messages	Structure	Descriptio n	Applications	Examples	Tools
Topics	.msg	Continue data streams	One-way continuous data flow	Sensor data, robot state	rostopic
Services	.srv	Blocking call for processing a request	Short triggers	Trigger change, request state, computations	rosservice
Actions	.action	Non-blocking 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.


Master
Subscribing to /topic
Publishing /topic on localhost:1234
localhost:1234 is publishing /topic
Publisher
Subscriber
Message data for /topic

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

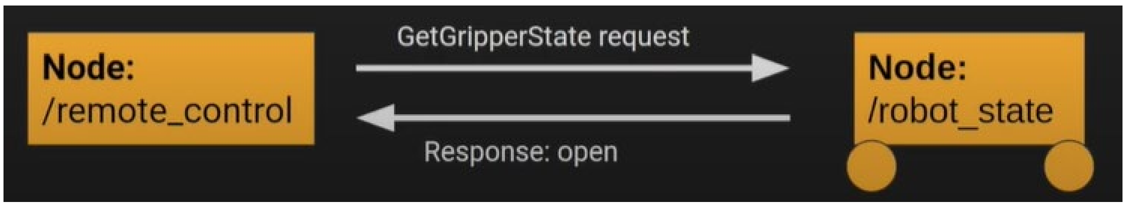
Protocol

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



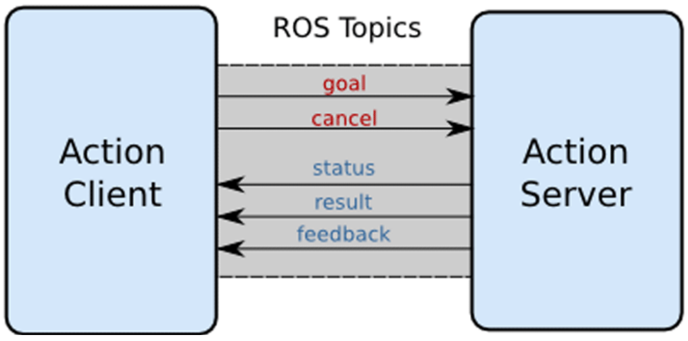
Rules

- 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:
   a. with a default value that can be overwritten when passed
   b. with a constant value which cannot be overwritten

2. The arguments are passed via:
a. launch command in CLI
b. 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 ...
```

2. Package architecture

```
/msg_dep
    /msg
        message.msg
    CMakeLists.txt
    package.xml
```

3. Create the msg message “message.msg”

4. 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
)
```

5. 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 ...
```

2. Package architecture

```
/msg_dep
    /srv
        message.srv
    CMakeLists.txt
    package.xml
```

3. Create the srv message “message.srv”

4. 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
)
```

5. 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 ...
```

2. Package architecture

```
/msg_dep
    /action
        message.action
    CMakeLists.txt
    package.xml
```

3. Crete the action message “message.action”

4. 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
)
```

5. 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
```

2. Package architecture

```
/dep_pkg
    /scripts
        /dep_pkg
            __init__.py
            module.py
    CMakeLists.txt
    package.xml
    setup.py
```

3. Create the python module “module.py”
4. Create the “__init__.py” empty file

5. 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)
```

6. CMakeLists.txt

```
find_package(catkin REQUIRED COMPONENTS
       rospy
)

catkin_python_setup()
```

7. package.xml

```
<depend>rospy</depend>
```

C++ libraries

1. Creating dependency package

```
catkin create pkg dep_pkg -c roscpp
```

2. Package architecture

```
/dep_pkg
    /include
        /dep_pkg
            lib.h
    /src
    CMakeLists.txt
    package.xml
    setup.py
```

3. 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}
)
```

4. package.xml

```
<depend>roscpp</depend>
```

High-level

Coordinate Frames/Transform (tf/tf2)

Sources

http://wiki.ros.org/tf2

https://github.com/ros/geometry2

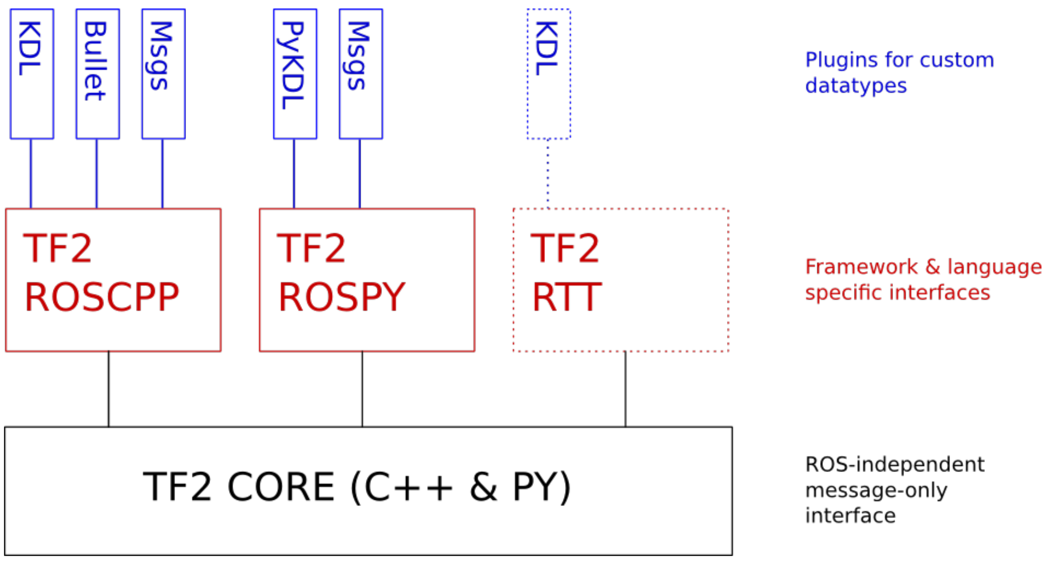
The ***tf2 library*** provides functionalities to:

- 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


KDL
Bullet
Msgs
PyKDL
Msgs
KDL
Plugins for custom datatypes
TF2 ROSCPP
TF2 ROSPY
TF2 RTT
Framework & language specific interfaces
TF2 CORE (C++ & PY)
ROS-independent message-only interface

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

```
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
```

2. Transform object

```
static_transform = TransformStamped()
```

3. 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
```

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

```
rospy.Subscriber('/topic', module, callback)
```

3. Transform object

```
transform = TransformStamped()
```

4. 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)
```

2. 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:

a. from this frame
b. (optional) at this time
c. to this frame
d. at this time
e. (optional) specify the frame that does not change in time
f. (optional) timeout

Time debugging

a. Get the latest available transform

```
listener.lookupTransform("/turtle2",
                         "/turtle1",
                         rospy.Time(0),
                         transform);
```

b. 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)
```

2. 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

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

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

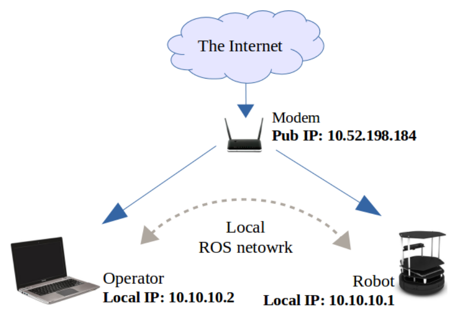
https://www.youtube.com/watch?v=N1QsVM5imsU&list=PLK0b4e05LnzbHiGDGTgE_FIWpOCvndtYx&index=9&t=0s

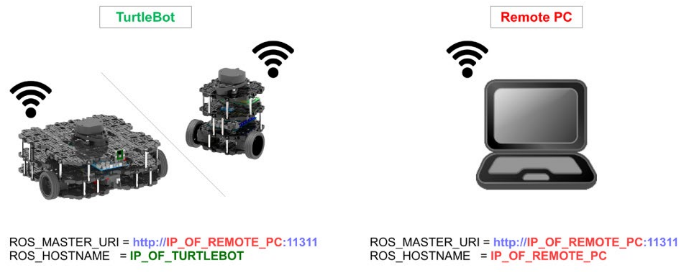
https://www.youtube.com/watch?v=OiNll0YJ_Us&list=PLK0b4e05LnzbHiGDGTgE_FIWpOCvndtYx&index=11&t=0s

rqt [WIP]

Extra

ROS Networking





```
ROS_MASTER_URI = http://IP_OF_REMOTE_PC:11311
ROS_HOSTNAME   = IP_OF_TURTLEBOT
```

```
ROS_MASTER_URI = http://IP_OF_REMOTE_PC:11311
ROS_HOSTNAME   = IP_OF_REMOTE_PC
```

Requirements:

1. A common network
2. ROS on every machine
3. IP address of every machine

1. IP addresses configuration

All machines have to be configured to use the same master (ROS_MASTER_URI)

a. Check IP

```
>> ifconfig
```

b. Check machine hostname

```
>> hostname
>> hostname -i
```

c. Export environment variable

```
>> gedit ~/.bashrc
```

```
export ROS_MASTER_URI=http://master_hostname:11311
export ROS_HOSTNAME=machine_hostname
export ROS_IP=machine_ip
```

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

```
>> sudo gedit /etc/hosts
>> sudo gedit /etc/hostname
```

2. Connectivity check

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

a. Check I: self ping

```
ssh master_hostname
ping master_hostname
```

b. Check II: mutual ping

```
ssh master_hostname
ping machine_hostname
```

c. Check III: port connection

```
ssh master_hostname
netcat -l 1234
```

```
ssh machine_hostname
netcat master_hostname 1234
```

3. 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