# ROS <Overall>



## Introduction

ROS is an open-source, meta-operating system for your robot. It provides the services, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers.

The ROS runtime "graph" is a peer-to-peer network of processes (potentially distributed across machines) that are loosely coupled using the ROS communication infrastructure. ROS implements several different styles of communication, including synchronous RPC-style communication over services, **asynchronous streaming of data over topics**, and storage of data on a Parameter Server.



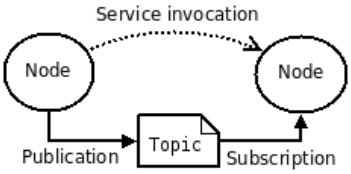
ROS Wiki: http://wiki.ros.org/

## Concepts

ROS has three levels of concepts: the Filesystem level, the Computation Graph level, and the Community level. Here mainly talk about **ROS Computation Graph Level**.

### Computation Graph Level

The Computation Graph is the peer-to-peer network of ROS processes that are processing data together. The basic Computation Graph concepts of ROS are **nodes, Master, Parameter Server, messages, services, topics, and bags**, all of which provide data to the Graph in different ways.




- Nodes: A node is an executable that uses ROS to communicate with other nodes.
- Messages: ROS data type used when subscribing or publishing to a topic.
- Topics: Nodes can *publish* messages to a topic as well as *subscribe* to a topic to receive messages.
- Master: Name service for ROS (i.e. helps nodes find each other)
- rosout: ROS equivalent of stdout/stderr
- roscore: Master + rosout + parameter server (parameter server will be introduced later)

## Nodes:

- Nodes are **processes that perform computation**. ROS is designed to be modular at a fine-grained scale. A robot control system usually comprises many nodes.
- For example,one node controls the wheel motors, one node performs path planning, one Node provides a graphical view of the system, and so on.
- A ROS node is written with the use of a ROS client library, such as roscpp or rospy.

## Messages:

- **Nodes communicate with each other by passing messages**. A message is simply a data structure, comprising typed fields.
- Standard primitive types (integer, floating point, boolean, etc.) are supported, as are arrays of primitive types.

## Topics:

- Messages are routed via a transport system with **publish / subscribe** semantics.
- **A node sends out a message by *publishing* it to a given topic**. The **topic is a name that is used to identify the content of the message**.
- A node that is **interested in a certain kind of data** will ***subscribe* to the appropriate topic**.
- In general, publishers and subscribers are not aware of each others' existence.

## Master:

- Master provides **name registration and lookup** to the rest of the Computation Graph. With the Master, **nodes** are able to **find each other, exchange messages, or invoke services**.
- **Names** have a very important role in ROS: nodes, topics, services, and parameters all have names. Every ROS client library supports command-line remapping of names.
- **Nodes connect to other nodes directly**; the Master only provides lookup information.
- The Parameter Server allows data to be stored by key in a central location. It is part of Master.

## Services:

- The publish / subscribe model is a very flexible communication paradigm, but its many-to-many, one-way transport is not appropriate for request / reply interactions.
- **Request / reply** is done via services, which are defined by **a pair of message structures**: one for the request and one for the reply. A providing **node offers a service under a name** and a **client uses the service by sending the request message and awaiting the reply**.
- **ROS client libraries** generally present this interaction to the programmer.

## Bags:

- Bags are a format for **saving and playing back ROS message data**.
- Bags are an important mechanism for storing data, such as sensor data, that can be difficult to collect but is necessary for developing and testing algorithms.

### Examples

One camera wants to cast screen to other three monitors. Instead of transferring data to each monitors. The camera publish messages into Screen topic and monitors subscribe the Screen topic to aquire screen messages. Both sides do not know their existence.

## Common Commands

- Firstly start ROS

```
$ roscore
```

- Create workspace

```
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/src
$ catkin_init_workspace

# make workspace
$ cd ~/catkin_ws
$ catkin_make

# Add source commands into bash
$ echo "source catkin_ws/devel/setup.bash" >> ~/.bashrc
$ source ~/.bashrc
```

- ros packages

```
# Create packages (packages under workspace\src)
$ cd ~/catkin_ws/src
$ catkin_create_pkg <package_name> [depend1] [depend2] [depend3]
$ cd ~/catkin_ws
$ catkin_make

# Find packages and run it
$ rospack find [package name]

# Check dependencies of packages
$ rospack depends <package_name>
$ rospack depends1 <package_name>

# Run node of the package
$ rosrun [package_name] [node_name]
```

- rosnode

```
#Check rosnode
$ rosnode list
$ rosnode info [node_name]

#Remapping/clean/test node
$ rosrun turtlesim turtlesim_node __name:=my_turtle
$ rosnode cleanup
$ rosnode ping my_turtle
```

- rqt tools (ROS GUI tools)

```
# rqt_graph creates a dynamic graph of system
$ rosrun rqt_graph rqt_graph
$ rosrun rqt_plot rqt_plot
# rqt_console attaches to ROS's logging framework to display output from nodes.
$ rosrun rqt_console rqt_console
# rqt_logger_level allows us to change the verbosity level (DEBUG, WARN, INFO, and ERROR) of nodes as they run.
$ rosrun rqt_logger_level rqt_logger_level
```

- rostopic

```
# check topic list
$ rostopic list

# check sub-commands for rostopic
$ rostopic -h

# check infomation of the topic.
$ rostopic echo [topic]

# publish messages to topic
$ rostopic pub [-1] <topic> <msg_type> [-r 1] -- [args] [args]
```

- rosbag

```
# rosbag record subscribes to topics and writes a bag file with the contents of all messages published on those topics
# all topics
$ rosbag record -a
# specific topics
$ rosbag record rosout tf cmd_vel

# check rosbag function and summary
$ rosbag check -h
$ rosbag info <bagfile_name>

# rosbag compress/decompress
$ rosbag compress *.bag
$ rosbag decompress *.bag

# read and plays rosbag back in time-synchronized
$ rosbag play <bagfile_name>
$ rosbag play <your bagfile name> --topics <topics>

# rosbag filter
$ rosbag filter my.bag only-tf.bag "topic == '/tf'"

# get rosbag data
$ rostopic echo -b 2021-02-05-18-37-47.bag -p /sensor/wheel/front/steerangle > data.csv
```

- ros service

```
# print information about active services
$ rosservice list
# print service type
$ rosservice type [service]
# call the service with the provided args
$ rosservice call [service] [args]
```

- ros launch

```
# roslaunch is a tool for easily launching multiple commands
# Firstly create launch file
$ mkdir launch
```

```
$ touch launch/test.py
$ roslaunch [package_name] [file.launch]
```

- ros Editor

```
# Use rosed to edit file
$ rosed [package_name] [filename]
$ rosed [package_name] <tab><tab>

# Default editor is Vim, may change into nano
$ export EDITOR='nano -w'
```

# Other Commands

## System information

check the Environment Variables in ROS

```
$ export | grep ROS
$ echo $ROS_ROOT
$ echo $ROS_MASTER_URI
$ echo $ROS_HOSTNAME
```

## Clean log file

```
$ rosclean check
$ rosclean purge
```