ROSE: ROS for Engineers

Siddhartha Vibhu Pharswan

Indian Institute of Technology, Kanpur

AI Workshop, 2018

Table of Contents I

Table of Contents II

Dedicated to

Dedicated to the combined effort of

Willow Garage

&

Stanford Artificial Intelligence Laboratory

Motivational Quote

Chanakya

Learn from the mistakes of others, you can't live long enough to make them all yourselves!

An Effort

$ Suggestion $

Softwares can not be taught but can be practiced!

ROS: Robot Operating System

- What is ROS?
- A bit of History!
- Why are we using ROS?
- Will it support my system (Dabba!)?

What is ROS?

1. A large project being continuously developed by so many contributors, researchers and young ones like us :-).
2. It is a framework for writing robot softwares.
3. Collection of tools like RViz (ROS Visualization tool), rqt_ graph, rqt_ plot, rqt_ bag (visualization tools), rosbag (logging data), so many command line tools etc., libraries like rospy, roscpp, roslisp etc. and lots of conventions that simplify the tasks.

A bit of History!

1. Mid 2000's at STAIR (*Stanford Artificial Intelligence Robot*) it was needed to design a big framework for complex robotic tasks.
2. *Willow Garage (Robot Research Laboratory)* has hand-shaked with Stanford researchers and with collaborative work a large **unbeatable** architecture is designed.
3. For more information on ROS read *ROS: an open-source Robot Operating System. Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y Ng. ICRA 2009.*

Why are we using ROS?

1. No need to think about the how to write the drivers which can interact with hardware directly (a bit of hardware abstraction), low level device control.
2. Multilingual: C++ **(preferable)**, Python **(preferable)**, Ruby, Lisp, MATLAB, Java, R, Julia etc.
3. Very easy to communicate with robot hardware without even thinking about it.
4. Many robot hardwares support ROS frameworks to interact with.
5. Flexibility with client libraries to communicate with robot hardware.
6. Concurrently handle so many resources information without any headache.
7. Last but important : **A very active community.**

Will it support my system (Dabba!)?

1. Operating System: **Ubuntu Linux, MacOS (Experimental and community supported.)**
2. ROS installation: fully supported on ROS official website, just google it and follow the instructions.
3. Client Libraries: **roscpp, rospy, roslisp, rosjava** etc.
4. **Windows Users: Use Oracle VM box and install Ubuntu in it to save my time and your as well (:-?).**

My suggestion: Use Python (no headache to worry about the code style, just dive into ROS details.) but it does not mean learning it by one programming you will be **HERO** there are some functionality and packages which some client libraries support but others don't like PCL (Point Cloud Library, **roscpp**).

Analogies

1. Mechanical and Civil Engineers : A fluid flow
   - Reservoirs as nodes, fluid as messsages and pipe(medium) as topics.
2. Electrical Engineers : Kirchhoff's law
   - Circuit end connections as nodes (difference in potentials), current as messages and wire as topics.
3. Computer Science Engineers : ROS
   - **No need for any kind of correlation.** (Smart Enough :-;).

roscore

1. Starts ROS Master, ROS parameter server & rosout logging nodes.
2. Node **rosout:** collect and store log messages on topic (/rosout) from other ROS nodes (client libraries nodes rospy, roscpp) to store them in log files and rebroadcast the collected messages to another topic (/*rosout_agg*).
3. Helps nodes to find each other.
4. Without roscore there could be no graph.
5. *ROS_MASTER_URI*: environment variable contains IP and port of ROS Master. This variable helps to locate the master. In single system it is a localhost itself or name of localhost.
6. ROS parameter server used to store robots descriptions, parameters for algorithms so on.

```
vibhu@dell:~$ roscore
... logging to /home/vibhu/.ros/log/3b6f0ce6-7265-11e8-aad3-08002708c725/roslaunch-dell-2227.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://dell:36953/
ros_comm version 1.11.21


SUMMARY
========

PARAMETERS
 * /rosdistro: indigo
 * /rosversion: 1.11.21

NODES

auto-starting new master
process[master]: started with pid [2373]
ROS_MASTER_URI=http://dell:11311/

setting /run_id to 3b6f0ce6-7265-11e8-aad3-08002708c725
process[rosout-1]: started with pid [2386]
started core service [/rosout]
```

Figure: roscore running in terminal

Catkin

1. It is a ros build system
2. It manages all executables, libraries, and many more. Earlier version used was **rosbuild**.
3. Some CMake macros and Python scripts to provide some functionality on the top of CMake's normal workflow.

Catkin usage: catkin_make

1. Run the command 'mkdir -p ~/catkin_ws/src '
2. cd ~/catkin_ws/src
3. catkin_init_workspace
4. cd ..
5. catkin_make

1. catkin_init_workspace: creates a *CMakeLists.txt* in *src* directory.
2. *build*: libraries and executable programs (if using C++, not an interest as using *Python*).
3. *devel*: number of files and directories (setup files important ones '**devel/setup.bash**').

NOTE: If you have more than one workspace don't forget to run 'source devel/setup.bash ' after opening terminal every time for the workspace you want to use.

1. cd ~/catkin_ws/src
2. Run the command catkin_create_pkg <package_name> <dependencies>
3. **Example:** catkin_create_pkg <ros_tutorials><rospy roscpp std_msgs>

Fact: When uses roscpp as the dependencies include directory is created automatically (for your header files).

NOTE: 'TAB', a key helps to save time in many places for ROS users.

package.xml

```xml
<?xml version="1.0"?>
<package format="2">
  <name>ros_tutorials</name>
  <version>0.0.0</version>
  <description>The ros_tutorials package</description>

  <!-- One maintainer tag required, multiple allowed, one person per tag -->
  <!-- Example:  -->
  <!-- <maintainer email="jane.doe@example.com">Jane Doe</maintainer> -->
  <maintainer email="vibhu@todo.todo">vibhu</maintainer>


  <!-- One license tag required, multiple allowed, one license per tag -->
  <!-- Commonly used license strings: -->
  <!--   BSD, MIT, Boost Software License, GPLv2, GPLv3, LGPLv2.1, LGPLv3 -->
  <license>TODO</license>


  <!-- Url tags are optional, but multiple are allowed, one per tag -->
  <!-- Optional attribute type can be: website, bugtracker, or repository -->
  <!-- Example: -->
  <!-- <url type="website">http://wiki.ros.org/ros_tutorials</url> -->
```

package.xml

```
<!-- Author tags are optional, multiple are allowed, one per tag -->
<!-- Authors do not have to be maintainers, but could be -->
<!-- Example: -->
<!-- <author email="jane.doe@example.com">Jane Doe</author> -->


<!-- The *depend tags are used to specify dependencies -->
<!-- Dependencies can be catkin packages or system dependencies -->
<!-- Examples: -->
<!-- Use depend as a shortcut for packages that are both build and exec dependencies -->
<!--   <depend>roscpp</depend> -->
<!--   Note that this is equivalent to the following: -->
<!--   <build_depend>roscpp</build_depend> -->
<!--   <exec_depend>roscpp</exec_depend> -->
<!-- Use build_depend for packages you need at compile time: -->
<!--   <build_depend>message_generation</build_depend> -->
<!-- Use build_export_depend for packages you need in order to build against this package: -->
<!--   <build_export_depend>message_generation</build_export_depend> -->
<!-- Use buildtool_depend for build tool packages: -->
<!--   <buildtool_depend>catkin</buildtool_depend> -->
<!-- Use exec_depend for packages you need at runtime: -->
```

package.xml

```
<!--   <doc_depend>doxygen</doc_depend> -->
<buildtool_depend>catkin</buildtool_depend>
<build_depend>rospy</build_depend>
<build_depend>sensor_msgs</build_depend>
<build_depend>actionlib_msgs</build_depend>
<build_depend>std_msgs</build_depend>
<build_depend>message_generation</build_depend>
<build_export_depend>rospy</build_export_depend>
<build_export_depend>sensor_msgs</build_export_depend>
<build_export_depend>std_msgs</build_export_depend>
<exec_depend>message_runtime</exec_depend>
<exec_depend>actionlib_msgs</exec_depend>
<exec_depend>rospy</exec_depend>
<exec_depend>sensor_msgs</exec_depend>
<exec_depend>std_msgs</exec_depend>


<!-- The export tag contains other, unspecified, tags -->
<export>
  <!-- Other tools can request additional information be placed here -->

</export>
</package>
```

CMakeLists.txt

```
cmake_minimum_required(VERSION 2.8.3)
project(ros_tutorials)

## Compile as C++11, supported in ROS Kinetic and newer
# add_compile_options(-std=c++11)

## Find catkin macros and libraries
## if COMPONENTS list like find_package(catkin REQUIRED COMPONENTS xyz)
## is used, also find other catkin packages
find_package(catkin REQUIRED COMPONENTS
  rospy
  sensor_msgs
  std_msgs
  actionlib_msgs
  message_generation
)

## System dependencies are found with CMake's conventions
# find_package(Boost REQUIRED COMPONENTS system)


## Uncomment this if the package has a setup.py. This macro ensures
## modules and global scripts declared therein get installed
```

CMakeLists.txt

```
## System dependencies are found with CMake's conventions
# find_package(Boost REQUIRED COMPONENTS system)


## Uncomment this if the package has a setup.py. This macro ensures
## modules and global scripts declared therein get installed
## See http://ros.org/doc/api/catkin/html/user_guide/setup_dot_py.html
# catkin_python_setup()

################################################
## Declare ROS messages, services and actions ##
################################################

## To declare and build messages, services or actions from within this
## package, follow these steps:
## * Let MSG_DEP_SET be the set of packages whose message types you use in
##   your messages/services/actions (e.g. std_msgs, actionlib_msgs, ...).
## * In the file package.xml:
##   * add a build_depend tag for "message_generation"
##   * add a build_depend and a exec_depend tag for each package in MSG_DEP_SET
##   * If MSG_DEP_SET isn't empty the following dependency has been pulled in
##     but can be declared for certainty nonetheless:
```

CMakeLists.txt

```
##   * uncomment the generate_messages entry below
##   * add every package in MSG_DEP_SET to generate_messages(DEPENDENCIES ...)

## Generate messages in the 'msg' folder
 add_message_files(
   FILES
   new_msg.msg
#   Message1.msg
#   Message2.msg
 )
```

CMakeLists.txt

```
## Generate services in the 'srv' folder
add_service_files(
   FILES
   Add.srv
   Square.srv
#   Service1.srv
#   Service2.srv
 )

## Generate actions in the 'action' folder
add_action_files(
   FILES
   timer.action
#   Action1.action
#   Action2.action
 )

## Generate added messages and services with any dependencies listed here
 generate_messages(
   DEPENDENCIES
   actionlib_msgs
#   sensor_msgs
   std_msgs
```

CMakeLists.txt

```
  actionlib_msgs
#   sensor_msgs
  std_msgs
 )

################################################
## Declare ROS dynamic reconfigure parameters ##
################################################

## To declare and build dynamic reconfigure parameters within this
## package, follow these steps:
## * In the file package.xml:
##   * add a build_depend and a exec_depend tag for "dynamic_reconfigure"
## * In this file (CMakeLists.txt):
##   * add "dynamic_reconfigure" to
##     find_package(catkin REQUIRED COMPONENTS ...)
##   * uncomment the "generate_dynamic_reconfigure_options" section below
##     and list every .cfg file to be processed

## Generate dynamic reconfigure parameters in the 'cfg' folder
# generate_dynamic_reconfigure_options(
#   cfg/DynReconf1.cfg
#   cfg/DynReconf2.cfg
```

CMakeLists.txt

```
## LIBRARIES: libraries you create in this project that dependent projects also need
## CATKIN_DEPENDS: catkin_packages dependent projects also need
## DEPENDS: system dependencies of this project that dependent projects also need
catkin_package(
#  INCLUDE_DIRS include
#  LIBRARIES ros_tutorials
  CATKIN_DEPENDS message_runtime rospy sensor_msgs std_msgs
  actionlib_msgs
#  DEPENDS system_lib
)

###########
## Build ##
###########

## Specify additional locations of header files
## Your package locations should be listed before other locations
include_directories(
# include
  ${catkin_INCLUDE_DIRS}
)

## Declare a C++ library
```

Command line tools

1. roscd <package name>: navigation over ros packages and directories
2. rosrun <package name><executable name>, roslaunch <package name><launch file>: running executables and launch files respectively.
3. rqt_graph: visualiizing graph (if roscore is running).
4. rostopic list, rosservice list, rospack list, rosnode list: to list topics, services, packages and nodes respectively.
5. rospack find <package name>: when package is newly created.
6. rossrv show <service path>, rosmsg show <msg path >: to get msg/srv information.
7. rostopic info <topic name>: topic related information.

Names, namespaces and remapping

1. ROS names must be unique. (Otherwise one node of same name gets exit.) (__name:=new_node_name)
2. Namespaces are used to avoid topic collisions. Two cameras on same robot subscribing same topic /image can be conflicting. TO avoid different namespaces are given left and right like /home/vibhu/py.py, /home/vibhu/Desktop/py.py.
3. Remapping: in order to avoid changes in program string are mapped. image:=right/image , image:=left/image.

Two Characters

VIBHS

VIBHS'MOM

ROS Topics

1. Topics: A source of communication between two ROS nodes.
2. Messages: information sent via topic.
3. Publisher and Subscriber: A node that subscribes the data is subscriber while the one that publishes is publisher.
4. Do rostopic -h. Play with those commands.

NOTE: Use "chmod +x <py file name>" to make the file executable.

Publisher node: Vibhs(v1.py)

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import Bool

if __name__ == "__main__":
    rospy.init_node("publisher_node_v1", anonymous=True)
    pub = rospy.Publisher('tick_tock', Bool, queue_size=5)
    rate = rospy.Rate(2)
    init_bool = True
    while not rospy.is_shutdown():
        pub.publish(init_bool)
        rate.sleep()
```

Publisher node (Latched): Vibhs(v1_latched.py)

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import Bool

if __name__ == "__main__":
    rospy.init_node("publisher_node_v1", anonymous=True)
    pub = rospy.Publisher('tick_tock', Bool, queue_size=5, latch=True)
    rate = rospy.Rate(2)
    init_bool = True
    pub.publish(init_bool)
    init_bool = False
    pub.publish(init_bool)
    rate.sleep()
    rospy.spin()
```

Subscriber node: Vibhs'mom(m1.py)

```
#! /usr/bin/env python

import rospy
from std_msgs.msg import Bool

count = 0


def callback(msg):
    global count
    if msg.data == True:
        count +=1
        print " Number :: {}".format(count)
    else:
        print "Don't send me False messages"


if __name__ =="__main__":
    rospy.init_node("m1_subscriber_m1", anonymous=True)
    sub = rospy.Subscriber("/tick_tock", Bool, callback)
    rospy.spin()
```

Messages

1. ROS has so many types of built in messages.
2. std_msgs, geometry_msgs, sensor_msgs etc.
3. Messages from built in messages or from custom messages can create a new message definition.
4. Custom messages are made if needed.
5. Make a directory (msg) in src.
6. Keep <message name>.msg file in msg dir.

****** **Play with rosmsg commands.******

How to make custom messages

```
int64[]   intarray
float64[] floatarray
```

1. First : data type, second: variable name
2. Make changes in CMakeLists.txt and package.xml file.

NOTE: MD5 checksum (mostly error with C++ but can be issue with .pyc files also)

Publisher node : Vibhs(v2.py)

```
#!/usr/bin/env python
import rospy
from random import *
from ros_tutorials.msg import new_msg

if __name__ == "__main__":
    rospy.init_node("publisher_node_v2", anonymous=True)
    pub = rospy.Publisher('tick_tock', new_msg, queue_size=5)
    rate = rospy.Rate(2)

    while not rospy.is_shutdown():
        msg = new_msg()
        msg.intarray = [int(random()*20), int(random()*30), int(random()*40)]
        msg.floatarray = [random(), random(), random()]
        pub.publish(msg)
        rate.sleep()
```

Subscriber Node: Vibhs'mom(m2.py)

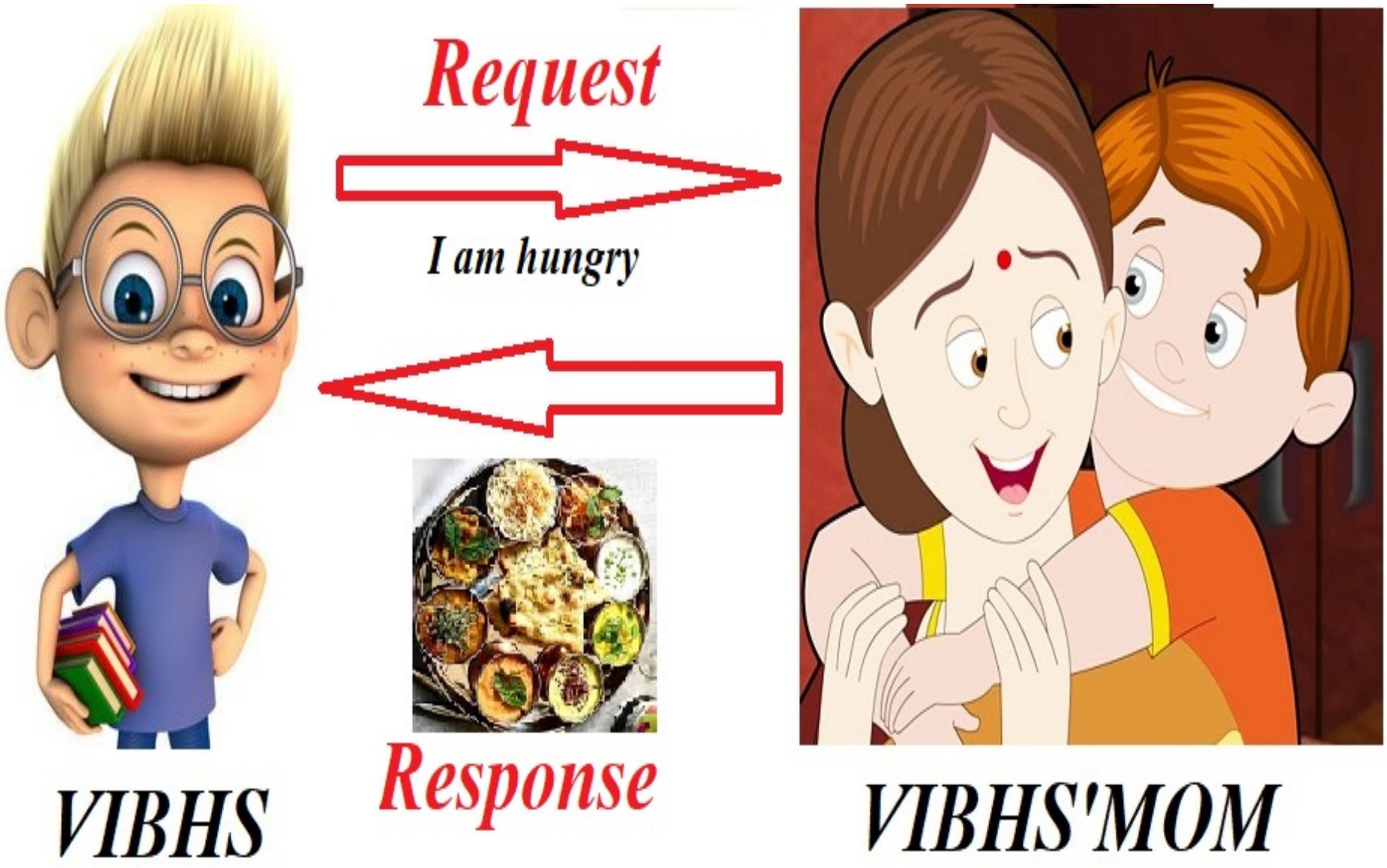
```
#! /usr/bin/env python

import rospy
from ros_tutorials.msg import new_msg


def callback(msg):
    print " Integer array received :: {}".format(msg.intarray)
    print "Float array received :: {}".format(msg.floatarray)


if __name__ =="__main__":
    rospy.init_node("m1_subscriber_m2", anonymous=True)
    sub = rospy.Subscriber("/tick_tock", new_msg, callback)
    rospy.spin()
```

Services

1. Services are synchronous call.
2. To define a service we need an input and output.
3. Until the response is sent system will wait.
4. To get an image from sensor (a good example).
5. Service call from terminal when server is running: "rosservice call <service name >input".

****** **Play with rossrv and rosservice commands** ******

Service Intuition



How to make a service

```
float64[] num
---
float64[] square
```

1. '- - -'seperates input(request) and output(response).
2. first: data types, second: variable name
3. Keep <service name >.srv in srv folder(in src of package).

Service Server Node: Vibhs'mom (ms1.py)

```
#! /usr/bin/env python

import rospy
from ros_tutorials.srv import Square, SquareResponse


def square_calculate(request):
    array = request.num
    print " Request Array received :: {}".format(array)
    sq_list = []
    for i in array:
        sq_list.append(i**2)
    return SquareResponse(sq_list)


if __name__ == "__main__":
    rospy.init_node("simple_server1", anonymous=True)
    service = rospy.Service('calculate_square', Square, square_calculate)
    rospy.spin()
```

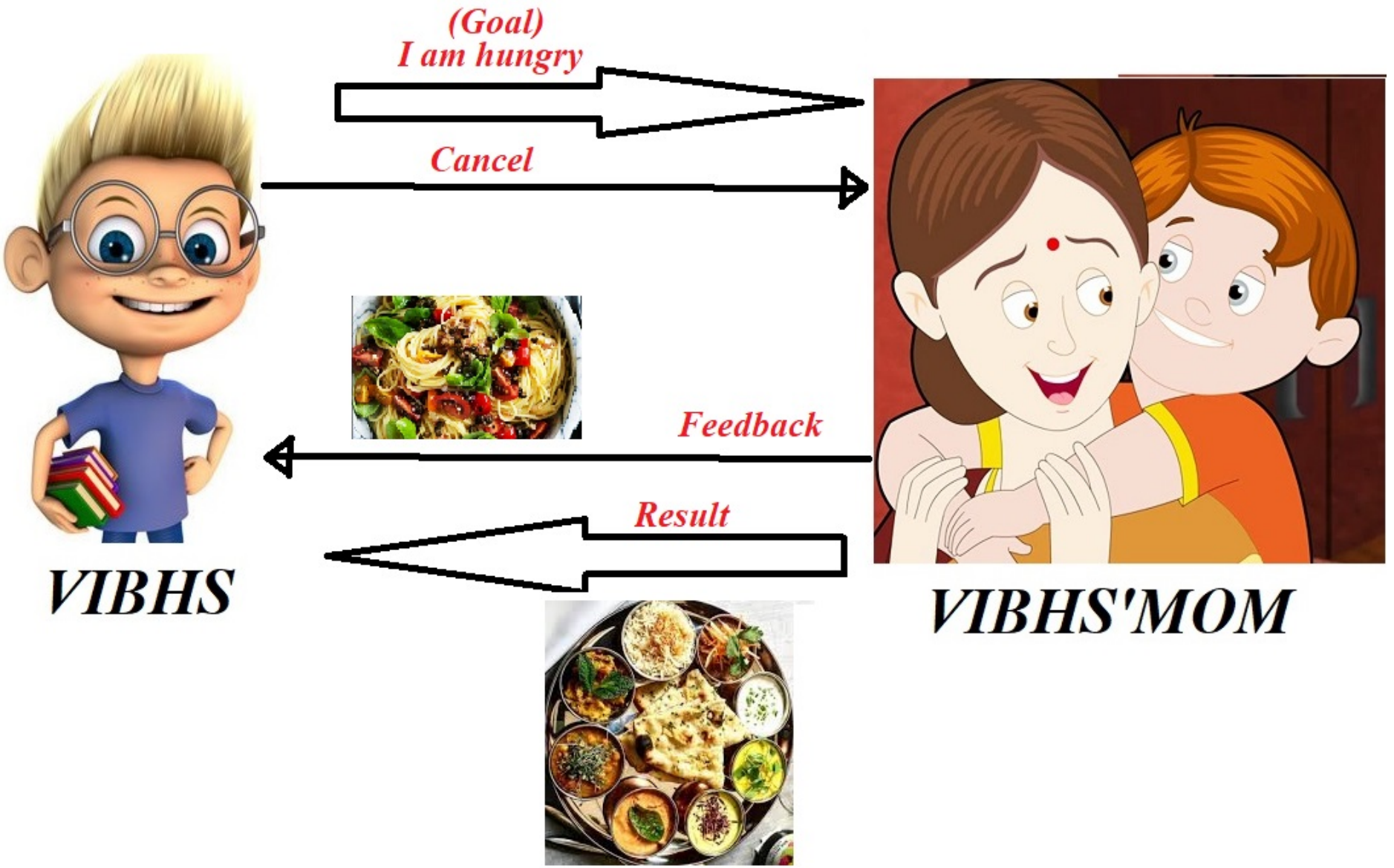
Service Client Node: Vibhs (vc1.py)

```
#! /usr/bin/env python

import rospy
from ros_tutorials.srv import Square
from random import *

if __name__ == "__main__":
    rospy.init_node("simple_client1", anonymous=True)
    rospy.wait_for_service('calculate_square')
    while not rospy.is_shutdown():
        request_array = []
        for i in range(randint(0, 6)):
            request_array.append(randint(0, 11))
        word_counter = rospy.ServiceProxy('calculate_square', Square)
        square_array = word_counter(num=request_array)
        print " Square of array elements :: {}".format(square_array.square)
```

Actions

1. Actions are asynchronous calls.
2. The system won't wait for the final result to come like services.
3. Feedback based mechanisms.
4. Robot has to go some position but do not know about the time elapsed in it then actions are used. (a good example).

Action Intuition



How to make an action

```
duration time_to_wait
---
duration time_elapsed
uint32 updates_sent
---
duration time_elapsed
duration time_remaining
```

1. '- - -'seperates goal, result and feedback.
2. first: data types, second: variable name
3. Keep <action name >.action in action folder(in src of package).

NOTE: Never forget to stop autostarting of action server.

Action Server Node: Vibhs'mom (mas1.py)

```
#!/usr/bin/env python

import rospy
import time
import actionlib
from ros_tutorials.msg import timerAction, timerGoal, timerResult


def do_timer(goal):
    start_time = time.time()
    time.sleep(goal.time_to_wait.to_sec())
    result = timerResult()
    result.time_elapsed = rospy.Duration.from_sec(time.time() - start_time)
    result.updates_sent = 0
    server.set_succeeded(result=result)


rospy.init_node("timer_action_server", anonymous=True)
server = actionlib.SimpleActionServer('timer', timerAction, do_timer, False)
server.start()
rospy.spin()
```

Action Client Node: Vibhs(vac1.py)

```
#!/usr/bin/env python

import rospy
import actionlib
from ros_tutorials.msg import timerAction, timerGoal, timerResult

if __name__ == "__main__":
    rospy.init_node("timer_action_client", anonymous=True)
    client = actionlib.SimpleActionClient('timer', timerAction)
    client.wait_for_server()
    goal = timerGoal()
    goal.time_to_wait = rospy.Duration.from_sec(5.0)
    client.send_goal_and_wait(goal, execute_timeout=rospy.Duration(5))
    print "Time elapsed: %f"%(client.get_result().time_elapsed.to_sec())
```

roslaunch

```
<launch>
<node name="subscriber_node_m1" pkg="ros_tutorials" type="m1.py" output="screen"/>
<node name="publisher_node_v1" pkg="ros_tutorials" type="v1.py" output="screen"/>
</launch>
```

1. launch file opens roscore when launched.
2. " /" don't forget to put this mark before the end of line.
3. launches multiple nodes at the same time.

rosbag

1. rosbag record -O <filename.bag><topic_name>
2. rosbag info <filename.bag >
3. rosbag play -l <filename.bag>(-l is used to loop for infinitely until you Ctrl+C)