| Module specification | Explanation | | |
|---|---|---|---|
| Teacher Name | -- | | |
| Training Topic | Robotics application in Virtual Laboratory | | |
| Training Code | UNIOULU_01_ROS | | |
| Module Name | Robotic operating system | | |
| Module duration | 100 min | | |
| Module objective | • Introduction to robotic operating system<br>• Installing ROS<br>• Understanding of ROS Topics, Services and Parameters<br>• Understanding of Simple Publisher and Subscriber<br>• Understanding of Simple service and client | | |
| Mode of provision | Classroom | | |
| Laboratory structure | Time (min) | Objective | Performed by? |
| | 10 | Introduction | Teacher |
| | 15 | Installation | Students |
| | 25 | First node | Teacher/Students |
| | 25 | Topics | Teacher/Student |
| | 20 | Services | Teacher/Student |
| | 5 | Finishing | Teacher |
| | | | |

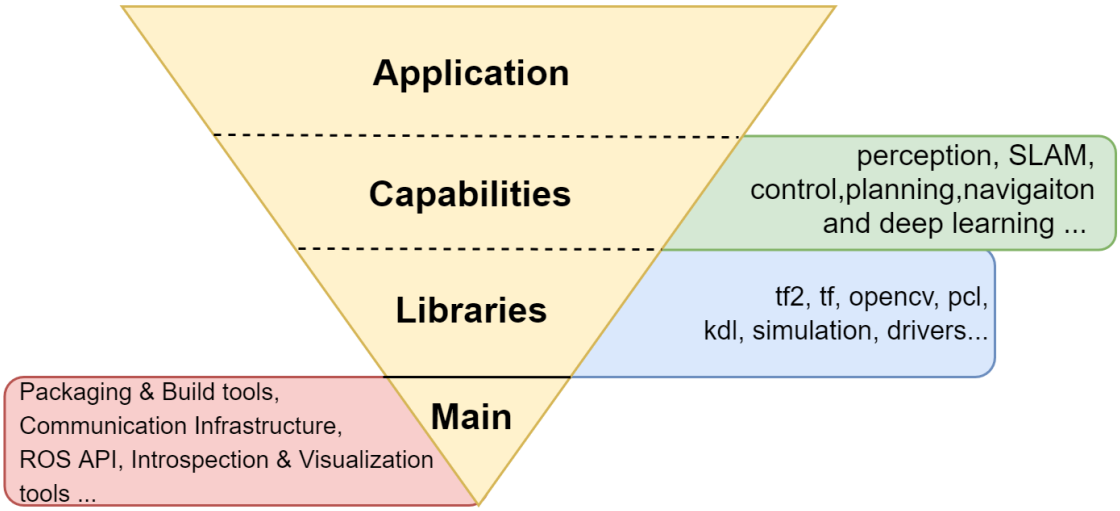



*Figure 1: Levels of development in ROS.*

The main reasons for opting ROS is easy construction of distributed computing systems, extensive tool-sets for debugging, visualization, logging, introspecting and saving the data in a bag format, has capability to provides libraries from low level system integration to control and perception of mobile systems and manipulators, it is supported by large ecosystem and improved constantly and finally, support for high-performance (C++) and high-level scripting (Python, Lisp) languages. Even though ROS has many advantages at conceptual level, one of the main drawbacks in ROS can be seen at design level. At design level ROS is implemented using XML-RPC which is not light-weight and ROS doesn't support real-time systems, this causes latency at data transport level. The *table 2* shows the features in ROS.

| **Features** | |
|---|---|
| High-level Language | *Roscpp , Java* |
| Scripting | *rospy, roslisp* |
| Embedded Systems support | *rosserial* |
| Message Transport | *TCP(TCPROS), UDP(UDPROS)* |
| Package Repositories | *Yes* |
| Publisher/Subscriber | *Yes* |
| Real-time support | *no* |
| Lightweight | *Stand-alone libraries are wrapped around with a thin ROS layer* |

*Table 2: Features in ROS.*

## ROS installation:

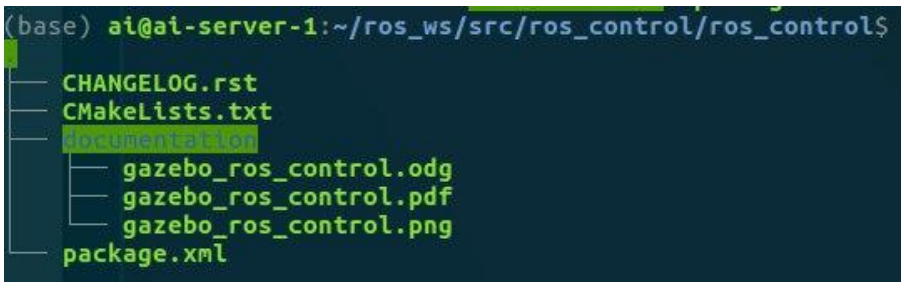
1) Follow the script, open a new terminal and run step by step or the whole script.

```
#!/bin/bash

#Install ROS-melodic


#Setup sources.list

sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -
sc) main" > /etc/apt/sources.list.d/ros-latest.list'


#Setup key

sudo apt-key adv --keyserver 'hkp://keyserver.ubuntu.com:80' --recv-
key C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654


#Installation

sudo apt-get update

sudo apt-get install ros-melodic-desktop-full


sudo rosdep init

rosdep update

sudo apt-get update

sudo apt-get dist-upgrade


#Environment Setup

echo "source /opt/ros/melodic/setup.bash" >> ~/.bashrc

source ~/.bashrc

source /opt/ros/melodic/setup.bash


#Dependencies
sudo apt-get install python-rosinstall python-rosinstall-generator python-
wstool build-essential python-catkin-tools
```

```
sudo apt-get install python-pip

pip install future


#Catkin Workspace setup

mkdir -p ~/ros_ws/src


echo "source ~/ros_ws/devel/setup.bash" | tee -a .bashrc


#Build using catkin tool

cd ~/ros_ws/src

git clone https://github.com/ros-controls/ros_control.git

cd ros_control

git checkout melodic-devel

cd ../..

catkin build
```

Once the installation and build are successfully completed you should have these directories in your catkin workspace (*ros_ws*).


And inside your source directory these files have generated.



(1) *CMakeLists.txt* - these are build instructions for your nodes, you need to edit this file if you want to compile any node, we will do it later.

(2) *package.xml* - this file contains package metadata like author, description, version or required packages. Package can be built without changing it, but you should adjust this file if you want to publish your package to others.

## Write code for your first node

Let's create C++ file for your node, name it tutorial_pkg_node.cpp and place it in src folder under tutorial_pkg.

```
touch ~/ros_ws/src/tutorial_pkg/src/tutorial_pkg_node.cpp
```

```
#include <ros/ros.h>
int main(int argc, char **argv)
{
ros::init(argc, argv, "example_node");
ros::NodeHandle n("~");
ros::Rate loop_rate(50);
while (ros::ok())
{
ros::spinOnce();
loop_rate.sleep();
}
}
```

## Code explanation line by line:

```
#include <ros/ros.h>
```

Add header files for basic ROS libraries.

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

Beginning of node main function.

```
ros::init(argc, argv, "example_node");
```

Initialization of ROS node, this function contacts with ROS master and registers node in the system.

```
ros::NodeHandle n("~");
```

Get the handle for node, this handle is required for interactions with system e.g. subscribing to topic.

```
ros::Rate loop_rate(50);
```

Define rate for repeatable operations.

```
while (ros::ok()) {
```

Check if ROS is working. E.g. if ROS master is stopped or there was sent signal to stop the system, ros::ok() will return false.

```
ros::spinOnce();
```

Process all incoming messages.

```
loop_rate.sleep();
```

Wait until defined time passes.

You can save the C++ file.

## Building your node

Before you build the node, you need to edit CMakeLists.txt from tutorial_pkg directory. Open it in your visual studio text editor.

Find line:

```
# add_compile_options(-std=c++11)
```

and uncomment it (remove # sign). This will allow to use C++11 standard of C++.
You should also find and uncomment line:

```
# add_executable(${PROJECT_NAME}_node src/tutorial_pkg_node.cpp)
```

This will let the compiler know that it should create executable file from defined source. Created executable will be your node. Variable PROJECT_NAME is defined by line project(tutorial_pkg). This results in tutorial_pkg_node as the name of the executable. You can adjust it to your needs.
After that find and uncomment lines:

```
# target_link_libraries(${PROJECT_NAME}_node
#   ${catkin_LIBRARIES}
# )
```

This will cause compiler to link libraries required by your node. Save the changes and close editor.

Final CMakeLists.txt should look like this:

```
cmake_minimum_required(VERSION 2.8.3)
project(tutorial_pkg)
add_compile_options(-std=c++11)
find_package(catkin REQUIRED COMPONENTS
```

```
roscpp

)

catkin_package(

CATKIN_DEPENDS

)

include_directories(${catkin_INCLUDE_DIRS})

add_executable(${PROJECT_NAME}_node src/tutorial_pkg_node.cpp)

target_link_libraries(${PROJECT_NAME}_node

${catkin_LIBRARIES}
```

Open terminal, move to workspace main directory and build your project with command catkin build:

```
cd ~/ros_ws

catkin build
```

## Running your node

Your node is built and ready for running, but before you run it, you need to load some environment variables:

```
source ~/ros_ws/devel/setup.sh
```

These environment variables allow you to run node regardless of directory you are working in. You have to load it every time you open new terminal, or you can add line:

```
. ~/ros_workspace/devel/setup.sh
```

to your .bashrc file.

To run your node, you can use command line or .launch file as with any other node. Remember that package is tutorial_pkg and node is tutorial_pkg_node.

**Task 1**

Run your node with command line or .launch file. Then use rosnode and rqt_graph tools to examine system and check if your node is visible in the system.
To remind, you can start ROS by typing in the name of the node, you can do this with the following command:

```
rosrun package_name node_type [options]
```

For the node you just created it will be:

```
rosrun tutorial_pkg tutorial_pkg_node
```

If you want to use .launch files associated with your custom package you will have to create launch directory:

```
mkdir ~/ros_workspace/src/tutorial_pkg/launch
```

Place your .launch files there. This way you can start them by typing:

```
roslaunch tutorial_pkg your_launch_file.launch
```

Example launch file for tutorial_pkg_node will be as follows:

```
<launch>

    <node pkg="tutorial_pkg" type="tutorial_pkg_node" name="tutorial_pkg_node" output="screen">

    </node>

</launch>
```

Save it as tutorial_pkg_node.launch in ~/ros_workspace/src/tutorial_pkg/launch directory and launch it:

```
roslaunch tutorial_pkg tutorial_pkg_node.launch
```

## Subscribing to topic

You will modify your node to subscribe to topic /camera/rgb/image_raw and calculate average brightness of image.
To process message received from the camera you need a header file with message type definition. You can include it with:

```
#include <sensor_msgs/Image.h>
```

Image message is an object consisting of following fields:

- `std_msgs/Header header` - header with message metedata
- `uint32 height` - image height in pixels
- `uint32 width` - image width in pixels
- `string encoding` - pixel encoding definition
- `uint8 is_bigendian` - is data expressed in bigendian manner
- `uint32 step` - length of data for one row
- `std::vector<uint8_t> data` - actual image data

Then you need a function for processing received message:

```
void imageCallback(const sensor_msgs::ImageConstPtr &image)

{
```

```
    long long sum = 0;
    for (int value : image->data)
    {
        sum += value;
    }
    int avg = sum / image->data.size();
    std::cout << "Brightness: " << avg << std::endl;
}
```

Code explanation line by line:

```
void imageCallback(const sensor_msgs::ImageConstPtr &image)
```

Function definition, argument is pointer to incoming message.

```
long long sum = 0;
```

Variable for storing sum of all pixel values.

```
for( int value : image->data )
```

Iteration through every pixel and colour.

```
sum+=value;
```

Add current pixel value to sum.

```
int avg = sum/image->data.size();
```

Calculate average value.

```
std::cout << "Brightness: " << avg << std::endl;
```

Print brightness value to screen.

Last thing to do is defining topic to subscribe:

```
ros::Subscriber sub = n.subscribe("/camera/rgb/image_raw", 10, imageCallback);
```

Here we use method subscribe of NodeHandle object. Arguments of method are:

- /camera/rgb/image_raw - name of topic to subscribe.
- 10 - message queue size. Messages are processed in order they come in. In the case that node receives, in short time, more messages than this value, excessive messages will be dropped.
- imageCallback - function to process incoming messages.

Your final code should look like this:

```
#include <ros/ros.h>
#include <sensor_msgs/Image.h>
void imageCallback(const sensor_msgs::ImageConstPtr &image)
{
   long long sum = 0;
   for (int value : image->data)
   {
      sum += value;
   }
   int avg = sum / image->data.size();
   std::cout << "Brightness: " << avg << std::endl;
}

int main(int argc, char **argv)
{
   ros::init(argc, argv, "example_node");
   ros::NodeHandle n("~");
   ros::Subscriber sub = n.subscribe("/camera/rgb/image_raw", 10, imageCallback);
   ros::Rate loop_rate(50);
   while (ros::ok())
   {
      ros::spinOnce();
      loop_rate.sleep();
   }
}
```

**Task 2** Build your node and run it along with `astra.launch`.
Use rosnode, rostopic and rqt_graph tools to examine system and check how data is passed between nodes.
When using launch files, we can make use of previously created files, this way we can make configuration easier and more readable.

Instead of configuring tutorial_pkg_node again, we will include file created in previous step:

```
<launch>

    <arg name="use_gazebo" default="false"/>


    <include unless="$(arg use_gazebo)" file="$(find astra_launch)/launch/astra.launch
"/>
    <include if="$(arg use_gazebo)" file="$(find rosbot_description)/launch/rosbot.lau
nch"/>


    <include file="$(find tutorial_pkg)/launch/tutorial_pkg_node.launch"/>

</launch>
```

Save above file as tutorial_2.launch in ~/ros_workspace/src/tutorial_pkg/launch directory and launch it:

```
roslaunch tutorial_pkg tutorial_2.launch
```

or if you are using **Gazebo** simulator:

```
roslaunch tutorial_pkg tutorial_2.launch use_gazebo:=true
```

## Receiving parameters

Your node can receive parameters, they are used to customize behaviour of node e.g. subscribed topic name, device name or transmission speed for serial port.

You will modify a node to receive boolean parameter which defines if node should print image brightness to screen.

To receive the parameter you need a variable to store its value, in this example variable should have a global scope:

```
bool print_b;
```

Then receive parameter value:

```
n.param<bool>("print_brightness", print_b, false);
```

Here we use method `param` of `NodeHandle` object. Arguments of method are:

- `print_brightness` - name of parameter to receive.
- `print_b` - variable to store parameter value.
- `false` - parameter default value.

Last thing is to print brightness dependent on parameter value:

```
if (print_b)
{
   std::cout << "Brightness: " << avg << std::endl;
}
```

Your final code should look like this:

```
#include <ros/ros.h>
#include <sensor_msgs/Image.h>

bool print_b;

void imageCallback(const sensor_msgs::ImageConstPtr &image)
{
   long long sum = 0;
   for (int value : image->data)
   {
      sum += value;
   }
   int avg = sum / image->data.size();
   if (print_b)
   {
      std::cout << "Brightness: " << avg << std::endl;
   }
}
```

```
int main(int argc, char **argv)
{
   ros::init(argc, argv, "example_node");
   ros::NodeHandle n("~");
   ros::Subscriber sub = n.subscribe("/camera/rgb/image_raw", 10, imageCallback);
   n.param<bool>("print_brightness", print_b, false);
   ros::Rate loop_rate(50);
   while (ros::ok())
   {
      ros::spinOnce();
      loop_rate.sleep();
   }
}
```

**Task 3**
Run your node with parameter print_brightness set to true and again set to false. Observe how behaviour of node changes.
To add parameter for node, you will need to add <param> tag inside <node> tag in tutorial_pkg_node.launch file.

## Publishing to topic

You will modify node to publish brightness value to a new topic with message of type std_msgs/UInt8. Message std_msgs/UInt8 is object with only one field data, which contain actual integer data.
Begin with including message header file:

```
#include <std_msgs/UInt8.h>
```

Next define publisher object with global scope:

```
ros::Publisher brightness_pub;
```

Then register in the system to publish to a specific topic:

```
brightness_pub = n.advertise<std_msgs::UInt8>("brightness" , 1);
```

Here we use method advertise of NodeHandle object. Arguments of method are:

- brightness - topic name.
- 1 - message queue size.

You also need to declare type of message which will be published, in this case it is `std_msgs::UInt8`.
Last thing is to put some data into message and send it to topic with some frequency:

```
std_msgs::UInt8 brightness_value;
brightness_value.data=avg;
brightness_pub.publish(brightness_value);
```

In our example it can be done while processing each message incoming from camera topic.

Your final code should look like this:

```
#include <ros/ros.h>
#include <sensor_msgs/Image.h>
#include <std_msgs/UInt8.h>

bool print_b;
ros::Publisher brightness_pub;

void imageCallback(const sensor_msgs::ImageConstPtr &image)
{
   long long sum = 0;
   for (int value : image->data)
   {
      sum += value;
   }
   int avg = sum / image->data.size();
   if (print_b)
   {
      std::cout << "Brightness: " << avg << std::endl;
   }
   std_msgs::UInt8 brightness_value;
   brightness_value.data = avg;
```

```
   brightness_pub.publish(brightness_value);
}


int main(int argc, char **argv)
{
   ros::init(argc, argv, "example_node");
   ros::NodeHandle n("~");
   ros::Subscriber sub = n.subscribe("/camera/rgb/image_raw", 10, imageCallback);
   n.param<bool>("print_brightness", print_b, false);
   brightness_pub = n.advertise<std_msgs::UInt8>("brightness", 1);
   ros::Rate loop_rate(50);
   while (ros::ok())
   {
      ros::spinOnce();
      loop_rate.sleep();
   }
}
```

**Task 4** Compile your node and run it along with astra.launch.
Use rosnode, rostopic and rqt_graph tools to examine the system, then use rostopic echo tool to read brightness of the image from the camera.

## Calling the service

You will modify node to call to a service with message type `std_srvs/Empty`, this type has no field and cannot carry any data, it can be used only for invoking action in another node and getting reply when it's done.
As a service provider we will use `image_saver` node from `image_view` package. `Image_saver` have one service named `save`. every time it is called, one frame from subscribed image topic is saved to hard drive.
Desired node behavior is to count incoming frames and call service once per given number of frames.

Begin with importing required header files:

```
#include <std_srvs/Empty.h>
```

We need one variable for counting passed frames:

```
int frames_passed = 0;
```

In imageCallback function increment counter with every incoming message:

```
frames_passed++;
```

Create a client which will be calling to service:

```
ros::ServiceClient client = n.serviceClient<std_srvs::Empty>("/image_saver/save");
```

Here we use method serviceClient of NodeHandle object. Method has only one argument, it is the name of service. You also need to determine message type for service: std_srvs::Empty.
Instantiate message object:

```
std_srvs::Empty srv;
```

Check if required number of frames passed and reset counter:

```
if (frames_passed > 100)
{
   frames_passed = 0;
```

Call the service:

```
client.call(srv);
```

Your final code should look like this:

```
#include <ros/ros.h>
#include <sensor_msgs/Image.h>
#include <std_msgs/UInt8.h>
#include <std_srvs/Empty.h>

bool print_b;
ros::Publisher brightness_pub;
int frames_passed = 0;

void imageCallback(const sensor_msgs::ImageConstPtr &image)
{
```

```
    long long sum = 0;
    for (int value : image->data)
    {
        sum += value;
    }
    int avg = sum / image->data.size();
    if (print_b)
    {
        std::cout << "Brightness: " << avg << std::endl;
    }
    std_msgs::UInt8 brightness_value;
    brightness_value.data = avg;
    brightness_pub.publish(brightness_value);
    frames_passed++;
}

int main(int argc, char **argv)
{
    ros::init(argc, argv, "example_node");
    ros::NodeHandle n("~");
    ros::Subscriber sub = n.subscribe("/camera/rgb/image_raw", 10, imageCallback);
    n.param<bool>("print_brightness", print_b, false);
    brightness_pub = n.advertise<std_msgs::UInt8>("brightness", 1);
    ros::ServiceClient client = n.serviceClient<std_srvs::Empty>("/image_saver/save");
    std_srvs::Empty srv;
    ros::Rate loop_rate(50);
    while (ros::ok())
    {
        ros::spinOnce();
```

```
      if (frames_passed > 100)
      {
         frames_passed = 0;
         client.call(srv);
      }
      loop_rate.sleep();
   }
}
```

**Task 5** Build your node and run it with astra.launch and image_saver.
Use rosnode, rostopic and rqt_graph tools to examine the system and check how data is passed between nodes. Let the nodes work for a certain time. Observe as new frames are being saved to your workspace directory.
For image_saver node you can create separate launch file:

```
<launch>
    <node pkg="image_view" type="image_saver" name="image_saver">
        <param name="save_all_image" value="false" />
        <param name="filename_format" value="$(env HOME)/ros_workspace/image%04d.%s"/>
        <remap from="/image" to="/camera/rgb/image_raw"/>
    </node>
</launch>
```

Save it as image_saver.launch in ~/ros_workspace/src/tutorial_pkg/launch directory and include it in tutorial_2.launch:

```
<launch>
    <arg name="use_gazebo" default="false"/>
    <include unless="$(arg use_gazebo)" file="$(find astra_launch)/launch/astra.launch
"/>
    <include if="$(arg use_gazebo)" file="$(find rosbot_description)/launch/rosbot.lau
nch"/>
    <include file="$(find tutorial_pkg)/launch/tutorial_pkg_node.launch"/>
    <include file="$(find tutorial_pkg)/launch/image_saver.launch"/>
</launch>
```

To delete image files created by this example run following command in your ros_workspace directory:

```
rm $(find image*)
```

## Providing a service

You will modify node to provide a service, which returns information regarding how many images were saved. This service will have a message type std_srvs/Trigger, it has no field for request and two fields for response: integer to indicate if service was triggered successfully or not and string for short summary of executed action.
Start with including required header files:

```
#include <std_srvs/Trigger.h>
```

Add variable for storing number of saved images:

```
int saved_imgs = 0;
```

**Copy**
Next, you need a function to execute when service is called:

```
bool saved_img(std_srvs::Trigger::Request &req, std_srvs::Trigger::Response &res)

{

   res.success = 1;

   std::string str("Saved images: ");

   std::string num = std::to_string(saved_imgs);

   str.append(num);

   res.message = str;

   return true;

}
```

Arguments for this function are pointers to request and response data. All services are called the same way, even if it does not carry any data, in that case these are pointer of void type.

Prepare string with response description:

```
std::string str("Saved images: ");

std::string num = std::to_string(saved_imgs);

str.append(num);
```

Fill string field with data:

```
res.message= str;
```

Fill integer field with data, this mean service was executed properly:

```
res.success=1;
```

Finish function, response will be sent to requesting node:

```
return true;
```

next thing to do is to increment image counter after saving frame:

```
saved_imgs++;
```

Last thing to do is to register provided service in the system:

```
ros::ServiceServer service = n.advertiseService("saved_images", saved_img);
```

Here we use method advertiseService of NodeHandle object. Arguments of method are:

- saved_images - service name.
- saved_img - method to execute.

Your final code should look like this:

```
#include <ros/ros.h>

#include <sensor_msgs/Image.h>

#include <std_msgs/UInt8.h>

#include <std_srvs/Empty.h>

#include <std_srvs/Trigger.h>


bool print_b;

ros::Publisher brightness_pub;

int frames_passed = 0;

int saved_imgs = 0;


void imageCallback(const sensor_msgs::ImageConstPtr &image)

{

   long long sum = 0;

   for (int value : image->data)

   {

      sum += value;
```

```
    }
    int avg = sum / image->data.size();
    if (print_b)
    {
        std::cout << "Brightness: " << avg << std::endl;
    }
    std_msgs::UInt8 brightness_value;
    brightness_value.data = avg;
    brightness_pub.publish(brightness_value);
    frames_passed++;
}

bool saved_img(std_srvs::Trigger::Request &req, std_srvs::Trigger::Response &res)
{
    res.success = 1;
    std::string str("Saved images: ");
    std::string num = std::to_string(saved_imgs);
    str.append(num);
    res.message = str;
    return true;
}

int main(int argc, char **argv)
{
    ros::init(argc, argv, "example_node");
    ros::NodeHandle n("~");
    ros::Subscriber sub = n.subscribe("/camera/rgb/image_raw", 10, imageCallback);
    n.param<bool>("print_brightness", print_b, false);
    brightness_pub = n.advertise<std_msgs::UInt8>("brightness", 1);
```

```
    ros::ServiceClient client = n.serviceClient<std_srvs::Empty>("/image_saver/save");
    std_srvs::Empty srv;
    ros::ServiceServer service = n.advertiseService("saved_images", saved_img);
    ros::Rate loop_rate(50);
    while (ros::ok())
    {
        ros::spinOnce();
        if (frames_passed > 100)
        {
            frames_passed = 0;
            client.call(srv);
            saved_imgs++;
        }
        loop_rate.sleep();
    }
}
```