

# Cpp Publisher Node for ROS

July 14, 2024



Write a cpp publisher node for ros 1, also explain it in detail in easy to understand



Sure, let's dive into writing a simple C++ publisher node for ROS (Robot Operating System) 1 and then explain it in detail. We'll create a publisher node that publishes a string message at a regular interval.

## Step 1: Setting Up Your ROS Package

First, let's create a ROS package. Open your terminal and navigate to your ROS workspace, usually named `catkin_ws`.

```
bash

cd ~/catkin_ws/src
catkin_create_pkg my_publisher_node std_msgs roscpp
```

This command creates a package named `my_publisher_node` that depends on `std_msgs` (standard messages) and `roscpp` (ROS C++ client library).

## Step 2: Writing the Publisher Node

Next, navigate to the `src` directory of your new package and create a new C++ file, `publisher_node.cpp`.

```
bash

cd my_publisher_node/src
touch publisher_node.cpp
```

Open `publisher_node.cpp` in your favorite text editor and write the following code:

```
cpp

#include "ros/ros.h"
#include "std_msgs/String.h"
#include <sstream>

int main(int argc, char **argv)
{
    // Initialize the ROS system
    ros::init(argc, argv, "my_publisher");

    // Create a node handle
```

```

ros::NodeHandle nh;

// Create a publisher object
ros::Publisher pub = nh.advertise<std_msgs::String>("my_topic", 1000);

// Set the loop rate
ros::Rate loop_rate(10); // 10 Hz

int count = 0;
while (ros::ok())
{
    // Create a new String ROS message
    std_msgs::String msg;

    // Create a string for the data
    std::stringstream ss;
    ss << "Hello ROS: " << count;
    msg.data = ss.str();

    // Publish the message
    ROS_INFO("%s", msg.data.c_str());
    pub.publish(msg);

    // Sleep for the remaining time until we hit our 10 Hz rate
    loop_rate.sleep();
    ++count;
}

return 0;
}

```

### Step 3: Building the Node

Now that we've written our publisher node, we need to build it. Go back to the root of your workspace and build your package.

```
bash
```

```

cd ~/catkin_ws
catkin_make

```

### Step 4: Running the Node

Finally, let's run our publisher node. First, make sure the ROS master is running:

```
bash
```

```
roscore
```

Then, in a new terminal, run your publisher node:

```
bash
```

```
roslaunch my_publisher_node publisher_node
```

### Detailed Explanation

## ROS Initialization

cpp

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

This line initializes the ROS system. It parses any ROS-specific arguments passed at the command line, sets up the node name (`my\_publisher`), and contacts the ROS master.

## Node Handle

cpp

```
ros::NodeHandle nh;
```

The `NodeHandle` is the main access point to communicate with the ROS system. It initializes the node, sets up topics to publish or subscribe to, and manages various resources.

## Creating a Publisher

cpp

```
ros::Publisher pub = nh.advertise<std_msgs::String>("my_topic", 1000);
```

Here, we create a publisher object. This publisher will send messages of type `std\_msgs::String` on the topic `my\_topic`. The `1000` is the size of the message queue. If the queue is full, old messages will be discarded.

## Loop Rate

cpp

```
ros::Rate loop_rate(10); // 10 Hz
```

This sets the frequency at which the loop runs. In this case, it's set to 10 Hz, meaning the loop will run 10 times per second.

## Publishing Messages

cpp

```
std_msgs::String msg;
std::stringstream ss;
ss << "Hello ROS: " << count;
msg.data = ss.str();
```

```
ROS_INFO("%s", msg.data.c_str());
pub.publish(msg);
```

Inside the loop, we create a `std_msgs::String` message, populate it with data, and publish it. `ROS_INFO` is a convenience macro for logging.

## Looping and Sleeping

cpp

```
loop_rate.sleep();
++count;
```

The `loop_rate.sleep()` call makes the loop run at the desired rate (10 Hz). The `++count` increments the count variable for each loop iteration, so each message will be slightly different.

## Checking Your Understanding

To ensure you understand the prerequisites needed to grasp this topic, here are a few questions:

1. Are you familiar with the basics of C++ syntax and programming concepts?
2. Do you have a basic understanding of ROS concepts, such as nodes, topics, and messages?
3. Are you comfortable using the terminal and basic command-line operations in Linux?

Let me know your familiarity with these concepts so I can tailor the explanation to your level!



```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <sstream>

int main(int argc, char **argv)
{
    // Initialize the ROS system
    ros::init(argc, argv, "my_publisher");

    // Create a node handle
    ros::NodeHandle nh;

    // Create a publisher object
    ros::Publisher pub = nh.advertise<std_msgs::String>("my_topic", 1000);

    // Set the loop rate
    ros::Rate loop_rate(10); // 10 Hz

    int count = 0;
    while (ros::ok())
    {
```

```
// Create a new String ROS message
std_msgs::String msg;

// Create a string for the data
std::stringstream ss;
ss << "Hello ROS: " << count;
msg.data = ss.str();

// Publish the message
ROS_INFO("%s", msg.data.c_str());
pub.publish(msg);

// Sleep for the remaining time until we hit our 10 Hz rate
loop_rate.sleep();
++count;
}

return 0;
}
write the same code in python
```



Certainly! Below is the equivalent Python code for the C++ ROS publisher node you provided:

## Step 1: Setting Up Your ROS Package

Before running the Python node, make sure you have a ROS package. If you don't have one, you can create it by navigating to your ROS workspace and creating a package.

```
bash

cd ~/catkin_ws/src
catkin_create_pkg my_publisher_node std_msgs rospy
```

## Step 2: Writing the Publisher Node in Python

Navigate to the `scripts` directory of your package and create a new Python script, `publisher_node.py`.

```
bash

cd my_publisher_node/scripts
touch publisher_node.py
chmod +x publisher_node.py
```

Open `publisher_node.py` in your favorite text editor and write the following code:

```
python
```

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

import rospy
from std_msgs.msg import String

def publisher_node():
    # Initialize the ROS system
    rospy.init_node('my_publisher', anonymous=True)

    # Create a publisher object
    pub = rospy.Publisher('my_topic', String, queue_size=1000)

    # Set the loop rate
    rate = rospy.Rate(10) # 10 Hz

    count = 0
    while not rospy.is_shutdown():
        # Create a new String ROS message
        msg = String()

        # Create a string for the data
        msg.data = "Hello ROS: {}".format(count)

        # Publish the message
        rospy.loginfo(msg.data)
        pub.publish(msg)

        # Sleep for the remaining time until we hit our 10 Hz rate
        rate.sleep()
        count += 1

if __name__ == '__main__':
    try:
        publisher_node()
    except rospy.ROSInterruptException:
        pass
```

### Step 3: Running the Node

Make sure you have sourced your ROS workspace:

```
bash
```

```
source ~/catkin_ws/devel/setup.bash
```

Then, make the script executable (if you haven't already):

```
bash
```

```
chmod +x ~/catkin_ws/src/my_publisher_node/scripts/publisher_node.py
```

Now, you can run the node:

```
bash
```

```
roslaunch my_publisher_node publisher_node.py
```

## Detailed Explanation

### ROS Initialization

```
python
```

```
rospy.init_node('my_publisher', anonymous=True)
```

This line initializes the ROS system in Python. It sets up the node name (`my\_publisher`). The `anonymous=True` flag ensures that if you run multiple instances of the node, they will have unique names.

### Creating a Publisher

```
python
```

```
pub = rospy.Publisher('my_topic', String, queue_size=1000)
```

Here, we create a publisher object. This publisher will send messages of type `std\_msgs.msg.String` on the topic `my\_topic`. The `queue\_size=1000` parameter sets the size of the message queue. If the queue is full, old messages will be discarded.

### Loop Rate

```
python
```

```
rate = rospy.Rate(10) # 10 Hz
```

This sets the frequency at which the loop runs. In this case, it's set to 10 Hz, meaning the loop will run 10 times per second.

### Publishing Messages

```
python
```

```
msg = String()
msg.data = "Hello ROS: {}".format(count)
rospy.loginfo(msg.data)
pub.publish(msg)
```

Inside the loop, we create a `String` message, populate it with data, and publish it. `rospy.loginfo` is used for logging.

## Looping and Sleeping

python

```
rate.sleep()
count += 1
```

The `rate.sleep()` call makes the loop run at the desired rate (10 Hz). The `count += 1` increments the count variable for each loop iteration, so each message will be slightly different.

## Checking Your Understanding

To ensure you understand the prerequisites needed to grasp this topic, here are a few questions:

1. Are you familiar with the basics of Python syntax and programming concepts?
2. Do you have a basic understanding of ROS concepts, such as nodes, topics, and messages?
3. Are you comfortable using the terminal and basic command-line operations in Linux?

Let me know your familiarity with these concepts so I can tailor the explanation to your level!



```
int main(int argc, char **argv)
{
    // Initialize the ROS system
    ros::init(argc, argv, "my_publisher");

    // Create a node handle
    ros::NodeHandle nh;

    // Create a publisher object
    ros::Publisher pub = nh.advertise<std_msgs::String>("my_topic", 1000);

    // Set the loop rate
    ros::Rate loop_rate(10); // 10 Hz

    int count = 0; explain each line in detail, like what is argc, **argv, NodeHandle, nh, Publisher,
    pub, rate and where theyre defined?
```



Let's break down the C++ ROS publisher node code line by line, explaining the purpose and details of each element, including variables like `argc`, `argv`, `NodeHandle`, `nh`, `Publisher`, `pub`, `rate`, and where they're defined.



```
cpp
```

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

- **int main(int argc, char **argv**):** This is the main function of the C++ program, which serves as the entry point of execution.
  - `argc` (argument count) is an integer representing the number of command-line arguments passed to the program.
  - `argv` (argument vector) is an array of character pointers listing all the arguments. The first argument (`argv[0]`) is the program name, and subsequent elements are the arguments provided.

```
cpp
```

```
// Initialize the ROS system
ros::init(argc, argv, "my_publisher");
```

- **ros::init(argc, argv, "my\_publisher"):** This function initializes the ROS system.
  - `ros` is the ROS namespace containing ROS functionalities.
  - `init` is a function that initializes the ROS client library and prepares it for communication with the ROS master.
  - `argc` and `argv` are passed to handle any ROS-specific command-line arguments.
  - `"my_publisher"` is the name of the ROS node. This name must be unique among nodes running in the ROS network.

```
cpp
```

```
// Create a node handle
ros::NodeHandle nh;
```

- **ros::NodeHandle nh:** This line creates a `NodeHandle` object named `nh`.
  - `NodeHandle` is a class in the `ros` namespace that provides communication with the ROS system.
  - It initializes the node, sets up topics to publish or subscribe to, and manages various resources.
  - `nh` (node handle) is the object used to interact with the ROS master to communicate with other nodes.

```
cpp
```

```
// Create a publisher object
ros::Publisher pub = nh.advertise<std_msgs::String>("my_topic", 1000);
```

- **ros::Publisher pub = nh.advertise<std\_msgs::String>("my\_topic", 1000):** This line creates a publisher object named `pub`.
  - `Publisher` is a class in the `ros` namespace used to publish messages to a topic.

- ``nh.advertise<std_msgs::String>("my_topic", 1000)`` sets up the publisher to send messages of type ``std_msgs::String`` to the topic named ``"my_topic"``.
- ``1000`` is the size of the message queue. If the queue is full, old messages will be discarded to make room for new ones.

cpp

```
// Set the loop rate
ros::Rate loop_rate(10); // 10 Hz
```

- **ros::Rate loop\_rate(10):** This line creates a ``Rate`` object named ``loop_rate``.
  - ``Rate`` is a class in the ``ros`` namespace that helps maintain a desired frequency for a loop.
  - ``10`` is the frequency in Hertz (Hz), meaning the loop should run 10 times per second.

cpp

```
int count = 0;
```

- **int count = 0:** This line initializes an integer variable named ``count`` to ``0``. This variable will be used to count the number of iterations of the loop.

cpp

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

- **while (ros::ok()):** This line starts a loop that will continue to run as long as ROS is running properly.
  - ``ros::ok()`` is a function that returns ``true`` if the ROS system is functioning correctly and hasn't been shut down.

cpp

```
// Create a new String ROS message
std_msgs::String msg;
```

- **std\_msgs::String msg:** This line creates a new ``String`` message object named ``msg`` from the ``std_msgs`` package.
  - ``std_msgs`` is a package that contains standard message definitions, including ``String``.

cpp

```
// Create a string for the data
std::stringstream ss;
ss << "Hello ROS: " << count;
msg.data = ss.str();
```

- **std::stringstream ss:** This line creates a `stringstream` object named `ss`.
  - `stringstream` is a part of the C++ Standard Library and allows for easy construction of strings.
- **ss << "Hello ROS: " << count:** This line appends the string `"Hello ROS: "` and the value of `count` to the `stringstream`.
- **msg.data = ss.str():** This line sets the `data` field of the `msg` object to the string representation of the `stringstream`.

cpp

```
// Publish the message
ROS_INFO("%s", msg.data.c_str());
pub.publish(msg);
```

- **ROS\_INFO("%s", msg.data.c\_str()):** This line logs the message to the ROS console.
  - `ROS_INFO` is a macro for logging informational messages in ROS.
  - `msg.data.c_str()` converts the `data` field of the `msg` object to a C-style string for logging.
- **pub.publish(msg):** This line publishes the `msg` object to the topic specified when the publisher was created (`"my_topic"`).

cpp

```
// Sleep for the remaining time until we hit our 10 Hz rate
loop_rate.sleep();
++count;
}
```

- **loop\_rate.sleep():** This line sleeps for the remaining time needed to maintain the loop rate at 10 Hz. It ensures that the loop runs at the desired frequency.
- **++count:** This line increments the `count` variable by 1 for each iteration of the loop.

cpp

```
return 0;
}
```

- **return 0:** This line ends the main function and returns `0` to indicate that the program completed successfully.

## Summary

- `argc` and `argv` handle command-line arguments passed to the ROS node.
- `ros::init` initializes the ROS system.
- `ros::NodeHandle nh` creates a handle to the ROS node for communication.
- `ros::Publisher pub` creates a publisher object to send messages to a topic.
- `ros::Rate loop_rate(10)` sets the frequency of the loop.
- The loop runs while `ros::ok()` is true, creating, populating, and publishing messages at the specified rate.

Do you have any specific questions about any part of this explanation, or is there a particular aspect you'd like more detail on?



WRITE A subscriber node in cpp and also explain it similarly



Sure! Let's write a C++ ROS subscriber node and then break it down line by line, explaining each part in detail.

## Step 1: Setting Up Your ROS Package

Assuming you already have a ROS package created, we'll add a new file for the subscriber node. If not, create a package as shown in the previous example.

## Step 2: Writing the Subscriber Node

Navigate to the `src` directory of your package and create a new C++ file, `subscriber\_node.cpp`.

bash

```
cd ~/catkin_ws/src/my_publisher_node/src
touch subscriber_node.cpp
```

Open `subscriber\_node.cpp` in your favorite text editor and write the following code:

cpp

```
#include "ros/ros.h"
#include "std_msgs/String.h"

// Callback function that gets called when a new message is received
void chatterCallback(const std_msgs::String::ConstPtr& msg)
{
    ROS_INFO("I heard: [%s]", msg->data.c_str());
}

int main(int argc, char **argv)
{
    // Initialize the ROS system
    ros::init(argc, argv, "my_subscriber");

    // Create a node handle
    ros::NodeHandle nh;

    // Create a subscriber object
    ros::Subscriber sub = nh.subscribe("my_topic", 1000, chatterCallback);

    // Enter a loop, pumping callbacks
    ros::spin();

    return 0;
}
```

## Step 3: Building the Node

Now that we've written our subscriber node, we need to build it. Go back to the root of your workspace and build your package.

```
bash

cd ~/catkin_ws
catkin_make
```

## Step 4: Running the Node

Make sure the ROS master is running:

```
bash

roscore
```

Then, in a new terminal, run your subscriber node:

```
bash

roslaunch my_publisher_node subscriber_node
```

## Detailed Explanation

### ROS Initialization

```
cpp

int main(int argc, char **argv)
{
    // Initialize the ROS system
    ros::init(argc, argv, "my_subscriber");
```

- **`int main(int argc, char argv)`**: This is the main function of the C++ program, serving as the entry point.
  - `argc` and `argv` handle command-line arguments.
- **`ros::init(argc, argv, "my_subscriber")`**: Initializes the ROS system.
  - `argc` and `argv` are passed to handle any ROS-specific command-line arguments.
  - `"my_subscriber"` is the name of the ROS node. This name must be unique among nodes running in the ROS network.

### Node Handle

cpp

```
// Create a node handle
ros::NodeHandle nh;
```

- **ros::NodeHandle nh:** Creates a `NodeHandle` object named `nh`.
  - `NodeHandle` is the main access point to communicate with the ROS system, initializing the node, setting up topics, and managing resources.

## Subscriber Callback Function

cpp

```
// Callback function that gets called when a new message is received
void chatterCallback(const std_msgs::String::ConstPtr& msg)
{
    ROS_INFO("I heard: [%s]", msg->data.c_str());
}
```

- **void chatterCallback(const std\_msgs::String::ConstPtr& msg):** Defines a callback function that gets executed each time a message is received.
  - `std_msgs::String::ConstPtr` is a pointer to a constant `String` message from the `std_msgs` package.
  - `ROS_INFO("I heard: [%s]", msg->data.c_str())` logs the received message to the ROS console.

## Creating a Subscriber

cpp

```
// Create a subscriber object
ros::Subscriber sub = nh.subscribe("my_topic", 1000, chatterCallback);
```

- **ros::Subscriber sub = nh.subscribe("my\_topic", 1000, chatterCallback):** Creates a subscriber object named `sub`.
  - `nh.subscribe("my_topic", 1000, chatterCallback)` sets up the subscriber to receive messages of type `std_msgs::String` from the topic `"my_topic"`.
  - `1000` is the size of the message queue. If the queue is full, old messages will be discarded.
  - `chatterCallback` is the function to call when a new message is received.

## Entering the ROS Event Loop

cpp

```
// Enter a loop, pumping callbacks
ros::spin();
```

- **ros::spin():** Enters a loop that pumps callbacks. This means it will keep the node running, processing incoming messages and calling the appropriate callback functions.

cpp

```
return 0;
}
```

- **return 0:** Ends the main function and returns `0` to indicate that the program completed successfully.

## Summary

- **Initialization:** `ros::init` initializes the ROS system.
- **Node Handle:** `ros::NodeHandle nh` creates a handle to communicate with the ROS master.
- **Callback Function:** `chatterCallback` processes incoming messages.
- **Subscriber:** `nh.subscribe` sets up the subscriber to receive messages from a topic and specifies the callback function.
- **Event Loop:** `ros::spin` keeps the node running and processing incoming messages.

## Checking Your Understanding

To ensure you understand the prerequisites needed to grasp this topic, here are a few questions:

1. Are you familiar with the basics of C++ syntax and programming concepts?
2. Do you have a basic understanding of ROS concepts, such as nodes, topics, messages, and callback functions?
3. Are you comfortable using the terminal and basic command-line operations in Linux?

Let me know your familiarity with these concepts so I can tailor the explanation to your level!



i was suggested to use catkin build instead of catkin make, what is the difference between catkin build and catkin\_make? how to use catkin build



Great question! Both `catkin_make` and `catkin build` are tools used to build ROS workspaces, but they have some differences. Let's go over these differences and how to use `catkin build`.

## Differences Between `catkin_make` and `catkin build`

### 1. Tooling:

- `catkin_make`: This is the traditional build tool for catkin workspaces. It generates a single `CMake` build directory and compiles all packages in the workspace together.
- `catkin build`: This is part of the `catkin_tools` package, which provides more advanced build management features compared to `catkin_make`. It allows for better parallel builds, more control over the build process, and improved output formatting.

## 2. Workspace Management:

- ``catkin_make``: Uses a single build and devel directory for all packages in the workspace.
- ``catkin build``: Uses separate build and devel directories for each package, allowing more granular control and better dependency management.

## 3. Incremental Builds:

- ``catkin_make``: Less efficient for incremental builds since it treats the whole workspace as a single project.
- ``catkin build``: More efficient for incremental builds because it can rebuild individual packages that have changed.

## 4. Flexibility and Control:

- ``catkin_make``: Simpler to use but with fewer options for customization.
- ``catkin build``: Provides more flexibility and options, such as selectively building specific packages or configuring build profiles.

## How to Use ``catkin build``

### Step 1: Install ``catkin_tools``

First, if you haven't already, you need to install ``catkin_tools``. You can do this using ``apt`` or ``pip``.

```
bash

sudo apt-get update
sudo apt-get install python3-catkin-tools
```

Alternatively, you can use ``pip``:

```
bash

pip install -U catkin_tools
```

### Step 2: Initialize and Build Your Workspace

Navigate to the root of your catkin workspace and initialize it using ``catkin build``.

```
bash

cd ~/catkin_ws
catkin init
```

Now, you can build your workspace:

```
bash

catkin build
```

### Step 3: Building Specific Packages



If you want to build a specific package (e.g., `my_publisher_node``), you can specify it:

```
bash

catkin build my_publisher_node
```

## Step 4: Cleaning the Workspace

To clean the entire workspace:

```
bash

catkin clean
```

To clean a specific package:

```
bash

catkin clean my_publisher_node
```

## Example Workflow

Here's a typical workflow using `catkin build``:

### 1. Initialize the workspace (only needed the first time):

```
bash

cd ~/catkin_ws
catkin init
```

### 2. Build the entire workspace:

```
bash

catkin build
```

### 3. Build a specific package:

```
bash

catkin build my_publisher_node
```

### 4. Run your nodes (after sourcing the setup file):

```
bash
```

```
source ~/catkin_ws/devel/setup.bash
roslaunch my_publisher_node publisher_node
roslaunch my_publisher_node subscriber_node
```

## Summary

- `catkin_make` is simpler but less flexible and efficient for larger workspaces.
- `catkin build` (from `catkin_tools`) provides better control, efficiency, and output formatting.
- **Usage:** Initialize your workspace with `catkin init`, then build using `catkin build`. You can build specific packages, clean the workspace, and manage your builds more effectively.

Using `catkin build` can significantly improve your development experience, especially as your ROS projects grow in complexity. Let me know if you need any further details or assistance!



i have a basler camera, i have downloaded the basler pylon suite for ubuntu and interfaced the videostream using ipconfig, how do use the basler camera in ros?



To use a Basler camera with ROS, you'll need to integrate the Basler Pylon SDK with a ROS package that can publish the camera data as ROS messages. Here's a step-by-step guide on how to achieve this.

## Step 1: Install Basler Pylon SDK

Since you've already downloaded and installed the Basler Pylon SDK, ensure it is correctly installed. Verify the installation by running a simple test application provided by Pylon.

## Step 2: Create a ROS Package

Create a new ROS package to integrate the Basler camera.

```
bash
```

```
cd ~/catkin_ws/src
catkin_create_pkg basler_camera_node roscpp std_msgs sensor_msgs image_transport cv_bridge
```

This command creates a package named `basler_camera_node` with dependencies on `roscpp`, `std_msgs`, `sensor_msgs`, `image_transport`, and `cv_bridge`.

## Step 3: Write the Camera Node

Navigate to the `src` directory of your package and create a new C++ file, `basler_camera_node.cpp`.

```
bash
```

```
cd basler_camera_node/src
touch basler_camera_node.cpp
```

Open `basler\_camera\_node.cpp` in your favorite text editor and write the following code. This code will initialize the camera, capture frames, and publish them as ROS messages.

```
cpp
```

```
#include <ros/ros.h>
#include <image_transport/image_transport.h>
#include <cv_bridge/cv_bridge.h>
#include <sensor_msgs/image_encodings.h>
#include <pylon/PylonIncludes.h>
#include <opencv2/opencv.hpp>

using namespace Pylon;

int main(int argc, char **argv)
{
    // Initialize the ROS system
    ros::init(argc, argv, "basler_camera_node");
    ros::NodeHandle nh;
    image_transport::ImageTransport it(nh);
    image_transport::Publisher pub = it.advertise("camera/image", 1);

    // Initialize Pylon
    Pylon::PylonInitialize();

    try
    {
        // Create an instant camera object with the camera device found first
        CInstantCamera camera(CTlFactory::GetInstance().CreateFirstDevice());

        // Start the camera
        camera.StartGrabbing();

        // Create a pylon image format converter and set parameters
        CImageFormatConverter formatConverter;
        formatConverter.OutputPixelFormat = PixelType_BGR8packed;

        // Create a PylonImage that will be used to convert pylon images to OpenCV images
        CPylonImage pylonImage;
        cv::Mat cvImage;

        // Grabbing loop
        while (ros::ok() && camera.IsGrabbing())
        {
            // Wait for an image and then retrieve it
            CGrabResultPtr ptrGrabResult;
            camera.RetrieveResult(5000, ptrGrabResult, TimeoutHandling_ThrowException);

            // Image grabbed successfully?
            if (ptrGrabResult->GrabSucceeded())
            {
                // Convert the grabbed buffer to a pylon image
                formatConverter.Convert(ylonImage, ptrGrabResult);

                // Convert the pylon image to an OpenCV Mat
                cvImage = cv::Mat(ptrGrabResult->GetHeight(), ptrGrabResult->GetWidth(),
CV_8UC3, (uint8_t *)pylonImage.GetBuffer());

                // Convert OpenCV Mat to ROS sensor_msgs::Image
                sensor_msgs::ImagePtr msg = cv_bridge::CvImage(std_msgs::Header(), "bgr8",
cvImage).toImageMsg();

                // Publish the image
```

```

        pub.publish(msg);

        // Handle ROS callbacks
        ros::spinOnce();
    }
}
}
catch (const GenericException &e)
{
    // Error handling
    ROS_ERROR("An exception occurred: %s", e.GetDescription());
}

// Clean up
Pylon::PylonTerminate();
return 0;
}

```

## Step 4: Build the ROS Package

Navigate back to the root of your workspace and build the package using `catkin build`.

```
bash
```

```

cd ~/catkin_ws
catkin build basler_camera_node

```

## Step 5: Run the Camera Node

Make sure the ROS master is running:

```
bash
```

```
roscore
```

Then, in a new terminal, run your camera node:

```
bash
```

```
roslaunch basler_camera_node basler_camera_node
```

## Detailed Explanation

### 1. ROS Initialization:

```
cpp
```

```

ros::init(argc, argv, "basler_camera_node");
ros::NodeHandle nh;
image_transport::ImageTransport it(nh);
image_transport::Publisher pub = it.advertise("camera/image", 1);

```

- Initializes the ROS system and sets up a node handle, image transport, and publisher to publish images to the `camera/image` topic.

## 2. Pylon Initialization:

cpp

```
Pylon::PylonInitialize();
```

- Initializes the Pylon SDK.

## 3. Camera Setup:

cpp

```
CInstantCamera camera(CTlFactory::GetInstance().CreateFirstDevice());
camera.StartGrabbing();
```

- Creates and starts an instant camera object to grab images from the first available camera device.

## 4. Image Conversion:

cpp

```
CImageFormatConverter formatConverter;
formatConverter.OutputPixelFormat = PixelType_BGR8packed;
```

- Sets up the Pylon image format converter to convert images to BGR8 format, compatible with OpenCV.

## 5. Grabbing and Publishing Images:

cpp

```
while (ros::ok() && camera.IsGrabbing())
{
    CGrabResultPtr ptrGrabResult;
    camera.RetrieveResult(5000, ptrGrabResult, TimeoutHandling_ThrowException);

    if (ptrGrabResult->GrabSucceeded())
    {
        formatConverter.Convert(pylonImage, ptrGrabResult);
        cvImage = cv::Mat(ptrGrabResult->GetHeight(), ptrGrabResult->GetWidth(),
CV_8UC3, (uint8_t *)pylonImage.GetBuffer());
        sensor_msgs::ImagePtr msg = cv_bridge::CvImage(std_msgs::Header(), "bgr8",
cvImage).toImageMsg();
        pub.publish(msg);
        ros::spinOnce();
    }
}
```

- Grabs images from the camera, converts them to OpenCV format, converts them to ROS messages, and publishes them.

## 6. Exception Handling:

cpp

```

catch (const GenericException &e)
{
    ROS_ERROR("An exception occurred: %s", e.GetDescription());
}

```

- Catches and logs any exceptions that occur during the camera operation.

## 7. Cleanup:

cpp

```

Pylon::PylonTerminate();

```

- Cleans up the Pylon SDK.

## Summary

- Install the Basler Pylon SDK.
- Create a ROS package and write a node to interface with the Basler camera.
- Use the Pylon SDK to grab images and convert them to ROS-compatible messages.
- Build and run the ROS node.

This setup allows you to use your Basler camera within the ROS framework, enabling you to integrate it into your ROS-based applications. Let me know if you need any further assistance!



is it must that i code the cpp file? i thought there is a pylon cam ros package and it might already include these codes ?



You're right! There is indeed an existing ROS package called ``pylon_camera`` that provides an interface for Basler cameras using the Pylon SDK. This package already includes the necessary code to interface with Basler cameras, so you don't have to write the code from scratch.

## Step-by-Step Guide to Using the ``pylon_camera`` ROS Package

### 1. Install the Basler Pylon SDK

Ensure that the Basler Pylon SDK is installed on your system. You mentioned you have already done this, so we'll skip the detailed steps.

### 2. Install ``pylon_camera`` Package

You can install the ``pylon_camera`` package from source. First, clone the repository into your catkin workspace:

```
bash
```

```
cd ~/catkin_ws/src
git clone https://github.com/magazino/pylon_camera.git
```

### 3. Build Your Workspace

Navigate to the root of your workspace and build it using `catkin build`:

```
bash
```

```
cd ~/catkin_ws
catkin build
```

If you haven't initialized your workspace with `catkin build` yet, initialize it first:

```
bash
```

```
catkin init
catkin build
```

### 4. Source Your Workspace

After building your workspace, source the setup file:

```
bash
```

```
source ~/catkin_ws/devel/setup.bash
```

### 5. Run the Camera Node

Launch the `pylon_camera` node using a ROS launch file. Create a launch file in your package, e.g., `start_camera.launch`.

```
xml
```

```
<launch>
  <node name="pylon_camera" pkg="pylon_camera" type="pylon_camera_node"
    output="screen">
    <!-- Parameters can be set here -->
    <!-- <param name="some_param" value="some_value"/> -->
  </node>
</launch>
```

Save this file and then run it using `roslaunch`:

```
bash
```

```
roslaunch your_package_name start_camera.launch
```

## Configuration and Usage

The `pylon_camera` package provides various parameters that can be set via the launch file or dynamically via the ROS parameter server. Common parameters include camera settings such as exposure time, gain, frame rate, etc.

Here is an example of a launch file with some parameters set:

```
xml

<launch>
  <node name="pylon_camera" pkg="pylon_camera" type="pylon_camera_node" output="screen">
    <!-- Set the frame rate -->
    <param name="frame_rate" type="double" value="30.0"/>
    <!-- Set the camera exposure time in microseconds -->
    <param name="exposure_time" type="int" value="10000"/>
    <!-- Set the camera gain -->
    <param name="gain" type="int" value="10"/>
  </node>
</launch>
```

## Viewing the Camera Output

Once the `pylon_camera` node is running, you can view the camera output using `rqt_image_view`:

```
bash

roslaunch pylon_camera pylon_camera.launch
```

Select the topic `/pylon_camera_node/image_raw` (or similar, depending on the topic naming).

## Summary

- The `pylon_camera` ROS package allows you to interface with Basler cameras without writing custom code.
- Clone the `pylon_camera` repository into your catkin workspace.
- Build your workspace with `catkin build`.
- Use a launch file to start the `pylon_camera` node and set camera parameters.
- View the camera output using `rqt_image_view`.

This approach leverages existing tools and packages, streamlining the process of integrating Basler cameras into your ROS environment. Let me know if you need any more details or help with specific steps!



is this pylon ros available for ros1 ? i have pylon 6.2





Yes, the ``pylon_camera`` ROS package is available for ROS1 and is compatible with the Pylon 6.2 SDK. Here's how you can set it up and use it with ROS1.

## Step-by-Step Guide to Using ``pylon_camera`` with ROS1

### 1. Install Basler Pylon SDK

Ensure that the Basler Pylon SDK 6.2 is installed on your system. You mentioned you already have this installed.

### 2. Clone the ``pylon_camera`` Repository

Clone the ``pylon_camera`` repository into your ROS workspace:

```
bash

cd ~/catkin_ws/src
git clone https://github.com/magazino/pylon_camera.git
```

### 3. Build Your Workspace

Navigate to the root of your workspace and build it using ``catkin build``. If you haven't installed ``catkin_tools``, install it first:

```
bash

sudo apt-get update
sudo apt-get install python3-catkin-tools
```

Initialize your workspace (if not done already) and build:

```
bash

cd ~/catkin_ws
catkin init
catkin build
```

### 4. Source Your Workspace

After building your workspace, source the setup file:

```
bash

source ~/catkin_ws/devel/setup.bash
```

### 5. Run the Camera Node

Launch the ``pylon_camera`` node using a ROS launch file. Create a launch file in your package, e.g., ``start_camera.launch``:

```
bash

cd ~/catkin_ws/src/pylon_camera/launch
```

```
touch start_camera.launch
```

Edit `start_camera.launch` with the following content:

```
xml
```

```
<launch>
  <node name="pylon_camera" pkg="pylon_camera" type="pylon_camera_node"
    output="screen">
    <!-- Parameters can be set here -->
    <!-- <param name="some_param" value="some_value"/> -->
  </node>
</launch>
```

Save this file and then run it using `roslaunch`:

```
bash
```

```
roslaunch pylon_camera start_camera.launch
```

## Configuration and Usage

The `pylon_camera` package provides various parameters that can be set via the launch file or dynamically via the ROS parameter server. Common parameters include camera settings such as exposure time, gain, frame rate, etc.

Here is an example of a launch file with some parameters set:

```
xml
```

```
<launch>
  <node name="pylon_camera" pkg="pylon_camera" type="pylon_camera_node" output="screen">
    <!-- Set the frame rate -->
    <param name="frame_rate" type="double" value="30.0"/>
    <!-- Set the camera exposure time in microseconds -->
    <param name="exposure_time" type="int" value="10000"/>
    <!-- Set the camera gain -->
    <param name="gain" type="int" value="10"/>
  </node>
</launch>
```

## Viewing the Camera Output

Once the `pylon_camera` node is running, you can view the camera output using `rqt_image_view`:

```
bash
```

```
roslaunch rqt_image_view rqt_image_view
```

Select the topic `/pylon_camera_node/image_raw` (or similar, depending on the topic naming).

## Summary

- The `pylon_camera` ROS package is available for ROS1 and compatible with Pylon SDK 6.2.
- Clone the `pylon_camera` repository into your catkin workspace.
- Build your workspace with `catkin build`.
- Use a launch file to start the `pylon_camera` node and set camera parameters.
- View the camera output using `rqt_image_view`.

This setup allows you to efficiently integrate your Basler camera with ROS1. Let me know if you need any more details or help with specific steps!



is launch file necessary or can you just `roslaunch pylon_camera pylon_camera_node`?



A launch file is not strictly necessary to run the `pylon_camera_node`; you can indeed use `roslaunch` to start the node directly. However, using a launch file can simplify the process of setting parameters and launching multiple nodes at once.

If you prefer to use `roslaunch`, here's how you can do it, along with how to set parameters either via the command line or a YAML configuration file.

## Using `roslaunch` to Start the Node

### 1. Run the Node Directly

You can run the `pylon_camera_node` directly with `roslaunch`:

```
bash

roslaunch pylon_camera pylon_camera_node
```

### 2. Setting Parameters

To set parameters without a launch file, you can use `roslaunch` to set them on the command line or through a YAML file.

## Setting Parameters via Command Line

You can set parameters using `roslaunch` before running the node:

```
bash

roslaunch set /pylon_camera_node/frame_rate 30.0
roslaunch set /pylon_camera_node/exposure_time 10000
roslaunch set /pylon_camera_node/gain 10
roslaunch pylon_camera pylon_camera_node
```

This approach sets the parameters for the node before it starts.

## Setting Parameters via YAML File

Alternatively, you can create a YAML file with your parameters and load it before running the node. Create a YAML file named `camera\_params.yaml`:

```
yaml

frame_rate: 30.0
exposure_time: 10000
gain: 10
```

Then load the parameters and run the node:

```
bash

rosparam load camera_params.yaml /pylon_camera_node
roslaunch pylon_camera pylon_camera_node
```

## Example Workflow

Here's a step-by-step example of setting parameters via the command line and running the node:

### 1. Set Parameters

```
bash

rosparam set /pylon_camera_node/frame_rate 30.0
rosparam set /pylon_camera_node/exposure_time 10000
rosparam set /pylon_camera_node/gain 10
```

### 2. Run the Node

```
bash

roslaunch pylon_camera pylon_camera_node
```

## Viewing the Camera Output

Once the node is running, you can view the camera output using `rqt\_image\_view`:

```
bash

roslaunch rqt_image_view rqt_image_view
```

Select the topic `/pylon\_camera\_node/image\_raw` (or similar, depending on the topic naming).

## Summary

- **Launch File:** Optional but helpful for setting parameters and launching multiple nodes.
- **Command Line:** Use `rosparam set` to configure parameters before running the node.
- **YAML File:** Use a YAML file to manage parameters and load them with `rosparam load`.

Using `roslaunch` is a valid and straightforward way to start the `pylon_camera_node`, and setting parameters via `rosparam` or a YAML file provides flexibility for configuration. Let me know if you need any further assistance or details!



when interfacing the basler cam with the ubuntu pylon suite i had to set the ip of the ethernet and also the ip in the pylon ipconfig, but there is nothing like this, is it not necessary in ros? or what?



When interfacing a Basler camera with ROS, you indeed need to ensure that your camera is correctly configured and accessible over the network. This includes setting the correct IP addresses for both your camera and the network interface on your computer. These settings are necessary for the camera to communicate with your ROS nodes.

## Configuring IP Addresses

### 1. Set the IP Address of the Ethernet Interface on Your Computer:

Ensure that the IP address of your computer's Ethernet interface is set correctly to be in the same subnet as the Basler camera.

```
bash
```

```
sudo ifconfig eth0 192.168.0.1 netmask 255.255.255.0
```

Replace `eth0` with your Ethernet interface name and `192.168.0.1` with an IP address in the same subnet as the Basler camera.

### 2. Set the IP Address of the Basler Camera:

Use the Pylon IP Configurator tool to set the IP address of your Basler camera. This ensures that the camera is on the same subnet as your computer's Ethernet interface.

- Open Pylon Viewer or IP Configurator.
- Select your camera.
- Set the IP address (e.g., `192.168.0.2`) and subnet mask (`255.255.255.0`).

## Running the `pylon_camera` Node

Once the IP addresses are set and the camera is accessible over the network, you can proceed with running the `pylon_camera` node in ROS.

### 1. Install Dependencies:

Ensure you have the necessary ROS packages and Pylon SDK installed.

```
bash
```

```
sudo apt-get update
sudo apt-get install ros-<distro>-image-common ros-<distro>-image-transport ros-
<distro>-cv-bridge ros-<distro>-camera-info-manager
```

Replace ``<distro>`` with your ROS distribution (e.g., ``melodic``, ``noetic``).

## 2. Clone the `pylon\_camera` Repository:

```
bash
```

```
cd ~/catkin_ws/src
git clone https://github.com/magazino/pylon_camera.git
```

## 3. Build Your Workspace:

```
bash
```

```
cd ~/catkin_ws
catkin build
```

## 4. Source Your Workspace:

```
bash
```

```
source ~/catkin_ws/devel/setup.bash
```

## 5. Run the `pylon\_camera` Node:

You can either use ``roslaunch`` with a launch file or ``roslaunch`` directly. Here's how to do both.

### Using ``roslaunch`` Directly

If you prefer not to use a launch file, you can run the node directly with ``roslaunch``:

```
bash
```

```
roslaunch pylon_camera pylon_camera_node
```

### Using a Launch File

Create a launch file to start the `pylon\_camera` node and set parameters if needed. Here's an example:

```
xml
```

```
<launch>
  <node name="pylon_camera" pkg="pylon_camera" type="pylon_camera_node" output="screen">
    <!-- Set the frame rate -->
```

```
<param name="frame_rate" type="double" value="30.0"/>
<!-- Set the camera exposure time in microseconds -->
<param name="exposure_time" type="int" value="10000"/>
<!-- Set the camera gain -->
<param name="gain" type="int" value="10"/>
</node>
</launch>
```

Save this as `start_camera.launch` and run it using:

```
bash
```

```
roslaunch pylon_camera start_camera.launch
```

## Viewing the Camera Output

Use `rqt_image_view` to view the camera output:

```
bash
```

```
roslaunch rqt_image_view rqt_image_view
```

Select the topic `/pylon_camera_node/image_raw` (or similar, depending on the topic naming).

## Summary

- **Network Configuration:** Ensure your computer's Ethernet interface and the Basler camera are on the same subnet.
- **Pylon IP Configuration:** Use Pylon Viewer or IP Configurator to set the camera's IP address.
- **ROS Node:** Use `roslaunch` or `roslaunch` to start the `pylon_camera` node.
- **View Output:** Use `rqt_image_view` to view the camera feed.

These steps ensure your Basler camera is correctly interfaced and accessible in ROS, allowing you to stream and process camera data within your ROS environment. Let me know if you need further assistance or details!



how to calibrate a the camera using ros



Camera calibration is a crucial step in ensuring accurate computer vision tasks, such as 3D reconstruction, stereo vision, or even simple tasks like undistorting images. In ROS, the `camera_calibration` package provides tools to calibrate cameras. Here's a step-by-step guide on how to calibrate your camera using ROS.

## Prerequisites

1. **ROS Installed:** Ensure you have ROS installed and set up.
2. **camera\_calibration Package:** Ensure the `camera_calibration` package is installed.

```
bash
```

```
sudo apt-get install ros-<distro>-camera-calibration
```

Replace `<distro>` with your ROS distribution, e.g., `melodic`, `noetic`.

3. **Calibration Target:** You will need a checkerboard pattern or a circle grid pattern. The default checkerboard size is 8x6 squares. You can print a calibration pattern from [here](#).

## Step-by-Step Calibration

### 1. Start the Camera Node:

Ensure your camera node is running and publishing images. If you're using the `pylon_camera` node, run:

```
bash
```

```
roslaunch pylon_camera pylon_camera_node
```

### 2. Start the Calibration Tool:

Run the `camera_calibration` tool for monocular calibration:

```
bash
```

```
roslaunch camera_calibration cameracalibrator.py --size 8x6 --square 0.025  
image:=/camera/image_raw camera:=/camera
```

- `--size 8x6`: Specifies the number of internal corners in the checkerboard pattern. Adjust this to match your printed pattern.
- `--square 0.025`: Specifies the size of each square in meters (25mm in this case). Adjust this to match your printed pattern.
- `image:=/camera/image_raw`: The topic where the camera images are published.
- `camera:=/camera`: The base name of the camera.

### 3. Collect Calibration Data:

The calibration tool will open a window showing the camera feed. Move the checkerboard pattern in front of the camera to different positions and angles. The tool will automatically detect the checkerboard and collect samples. Ensure you cover a variety of orientations and distances. The tool will indicate the number of samples collected and the quality of those samples. Aim for at least 20 good samples.

### 4. Perform Calibration:

Once enough good samples are collected, press the "CALIBRATE" button in the calibration window. The tool will compute the camera's intrinsic parameters and distortion coefficients.

### 5. Save the Calibration Data:

After calibration, you can save the results by pressing the "SAVE" button. The calibration data will be saved to a file and also published on ROS topics.



## 6. Verify the Calibration:

To verify the calibration, look at the undistorted images. The calibration tool should automatically switch to showing the undistorted images. You can also run:

```
bash

roslaunch image_view image_view image:=/camera/image_raw
```

Compare the raw and undistorted images to ensure the calibration is correct.

## Using the Calibration Data

The calibration data is usually saved in a YAML file. You can use this data to undistort images in your ROS applications.

Here's an example of how to load and use the calibration data in a ROS node:

```
cpp

#include <ros/ros.h>
#include <image_transport/image_transport.h>
#include <cv_bridge/cv_bridge.h>
#include <sensor_msgs/image_encodings.h>
#include <opencv2/opencv.hpp>
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>

cv::Mat camera_matrix, dist_coeffs;

void imageCallback(const sensor_msgs::ImageConstPtr& msg)
{
    cv_bridge::CvImagePtr cv_ptr;
    try
    {
        cv_ptr = cv_bridge::toCvCopy(msg, sensor_msgs::image_encodings::BGR8);
    }
    catch (cv_bridge::Exception& e)
    {
        ROS_ERROR("cv_bridge exception: %s", e.what());
        return;
    }

    // Undistort the image
    cv::Mat undistorted_image;
    cv::undistort(cv_ptr->image, undistorted_image, camera_matrix, dist_coeffs);

    // Show the undistorted image
    cv::imshow("Undistorted Image", undistorted_image);
    cv::waitKey(30);
}

int main(int argc, char** argv)
{
    ros::init(argc, argv, "camera_calibration_example");
    ros::NodeHandle nh;
    image_transport::ImageTransport it(nh);
    image_transport::Subscriber sub = it.subscribe("/camera/image_raw", 1, imageCallback);

    // Load calibration data
    cv::FileStorage fs("/path/to/calibration.yaml", cv::FileStorage::READ);
    if (!fs.isOpened())
    {
        ROS_ERROR("Failed to open calibration file");
        return -1;
    }
}
```

```

}
fs["camera_matrix"] >> camera_matrix;
fs["distortion_coefficients"] >> dist_coeffs;
fs.release();

ros::spin();
return 0;
}

```

Replace `/path/to/calibration.yaml` with the path to your calibration file.

## Summary

1. **Start the Camera Node:** Ensure your camera node is running.
2. **Run the Calibration Tool:** Use `cameracalibrator.py` to collect calibration data.
3. **Collect Samples:** Move the checkerboard pattern to different positions.
4. **Calibrate:** Press the "CALIBRATE" button to compute camera parameters.
5. **Save Data:** Save the calibration results.
6. **Verify:** Check the undistorted images to verify calibration.
7. **Use Calibration Data:** Load and use the calibration data in your ROS nodes.

This guide provides a comprehensive overview of calibrating a camera in ROS, from collecting data to using the calibration results in your applications. Let me know if you need any further assistance!

[| Terms of use](#) [| Privacy policy](#) |