Object detection

And its need for a new system architecture

Contents

[Contents 2](#_Toc137487176)

[Context 3](#_Toc137487177)

[Problem 3](#_Toc137487178)

[Goal 3](#_Toc137487179)

[System Architecture 4](#_Toc137487180)

[Solution 4](#_Toc137487181)

[How does the object detection get its data? 5](#_Toc137487182)

[Method of data transmission 6](#_Toc137487183)

[How to connect an additional device to ROS network 7](#_Toc137487184)

[What is the current system overview? 9](#_Toc137487185)

[Object detection 12](#_Toc137487186)

[Integration on a jetson 13](#_Toc137487187)

[Validation 14](#_Toc137487188)

### Context

To draw points of interests (POI) on the map, the JACKAL makes use of the installed camera to detect objects of interest. It does this using a recognition model using python. The operator can see these points of interest on the map and inform others around him.

### Problem

While the code for object detection is already present, the integration with the exploration package is very poor. This integration essentially contains a launch file which launches everything at once. The JACKAL currently does not have enough processing power to accommodate for both these high intensity programs at once. This causes the JACKAL to lag tremendously. Indirectly this causes both the exploration and the object detection processes to not perform, or perform very poorly.

### Goal

Create a different system architecture where computational loads are divided or made more efficient to not impede upon the quality of already well performing processes. After this has been established, make sure the object detection can be ran parallel to the exploration process.

# System Architecture

## Solution

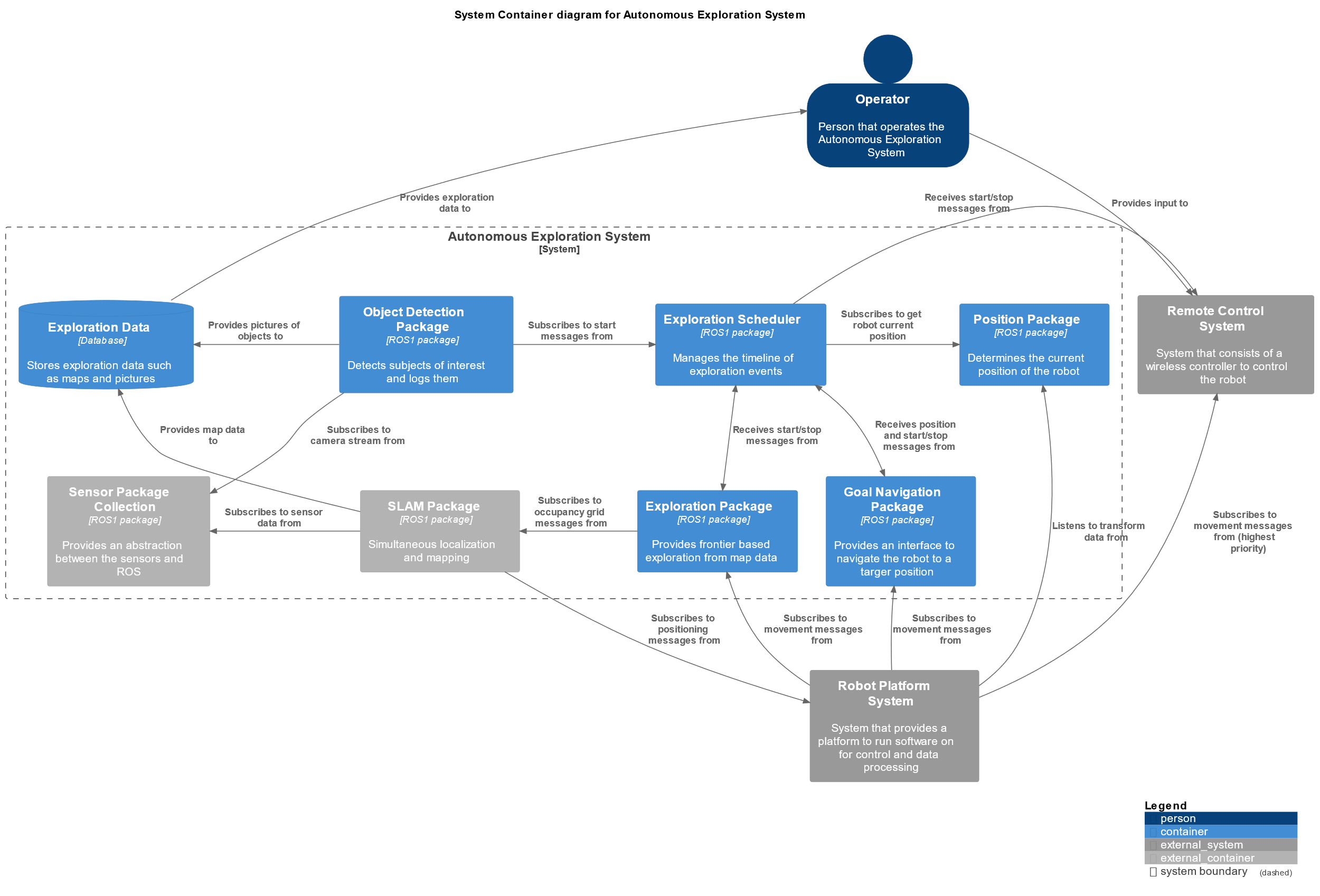
The solution to this problem hides in either making known processes more efficient (multithreading, filtering out lengthy statements etc); or making the existent hardware more powerful to accommodate for more computational power/divide the workload. The software currently in place is not only complicated to improve, but already proves to be very efficient in calculating and providing the data we use. Sometimes too much is too much, which proves to be the case here.

This leaves us with improving the hardware or dividing its workload onto it. The JACKAL still has one free RAM slot to install additional RAM. However, this change is so insignificant that I don’t expect there to be a massive jump in performance. Especially since image detection is heavily CPU dependent. Even heavier in the absence of a dedicated GPU.

The final option is then to divide the workload, we can do this by introducing a second computational device to the architecture. How this new architecture was achieved is found below.

## How does the object detection get its data?

In order to determine HOW to best split up computational load, we have to first look at how the object detection package is going to work. Where does it get its data from?



From the software overview diagrams we can conclude that the object detection package gets its data from subscribing to the camera stream. This means that the object detection package only needs to have access to this specific ROS topic to process its data! We can also see the Object detection package provides data to the exploration database. These are the pictures it takes from objects of interest.

## Method of data transmission

Now that we know how the package gets its data, we know how we could possibly connect it.   
  
ROS works with nodes. Some publish data, some listen to data. In our case, the package listens to another node for its data. This means that we have two options:

1. Export the data to the object detection node and export the output back into the ROS environment
2. Connect the object detection node inline with the ROS master

Obviously #1 is much more complicated than #2.

The developers of ROS also thought that multiple systems should be able to run inline with each other, on the same ROS network. That is why they introduced a slight hierarchy in their ROS machines. There is, usually, one master; and as many slaves as desired. We are going to make an external device a slave to our master and listen and publish to topics on the same ROS network as the master.

## How to connect an additional device to ROS network

Since we want to introduce a new device to the robot, it would make the most sense to physically link this device to the robot. Ethernet is the fastest and most reliable option to do this with.   
  
The JACKAL has two ethernet ports on it, which are configured quite weirdly. There is a software bridge in place. This means that all ethernet ports are synchronized to have the same Ip, much like there is only one port.

This is very tricky to work with, since the network manager of both the external device and the JACKAL are constantly reverting themselves back to the default values. Luckily, the bridge appoints one static port; an ethernet port which is always the same Ip. If we know this Ip, we can create a script which connects to this Ip and synchronizes the two environments.

The official JACKAL documentation tells us that we can connect with the static Ip using this command:

ssh administrator@192.168.131.1

We can deduct from this that *192.168.131.1* is the static Ip. We are going to use this to setup our script to set a few variables.

export ROS\_IP=$host\_ip

export ROS\_MASTER\_URI=$current\_uri

Namely the *ROS\_IP* and *ROS\_MASTER\_URI*, these two variables determine the place of a computer in the ROS network.

The *host\_ip* refers to the computer this script is ran on, you could say your local ip.   
The *current\_uri* switches dynamically based on if a connection is possible with the given Ip for the JACKAL.   
If a connection is possible: The ROS master will be set to the JACKAL.  
If a connection is NOT possible: The ROS master will be set to the local Ip.

The whole script is very short:

host\_ip="192.168.131.50"

host\_uri="http://localhost:11311"

echo $host\_ip

echo $host\_uri

# get Jackal IP and save URI

jackal\_ip="192.168.131.1"

jackal\_uri="http://cpr-j100:11311"

echo $jackal\_ip

echo $jackal\_uri

# ping Jackal and save exist status

ping -c1 $jackal\_ip 2>/dev/null 1>/dev/null

connected="$?"

# set ROS master to Jackal if connected to same network

if [[ $connected -ne 0 ]]

then

    echo "Jackal not connected to the same network as host, host is set as ROS master"

    current\_uri=$host\_uri

else

    echo "Jackal is set as ROS master"

    current\_uri=$jackal\_uri

fi

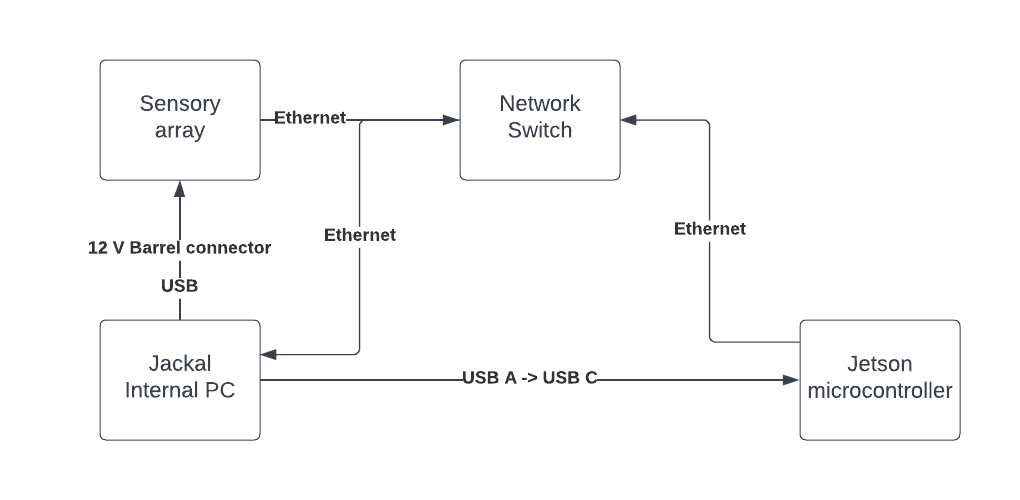
export ROS\_IP=$host\_ip

export ROS\_MASTER\_URI=$current\_uri

By using the static port, we can guarantee that the Ip always remains the same as given in the script.  
Currently the host Ip is hardcoded, an improvement for the future could be to make this dynamic; and adjust itself to its host machine. However in practice this wouldn’t be needed very often, since connections aren’t going to be changed much.

## What is the current system overview?

Since we added a new piece of hardware, we should acknowledge that the system overview has changed. Provided below are some models to gain insight into how the system is currently structured.

Hardware setup

The C1 model of interacting with the system has changed, it is updated to look like this:

A picture containing text, screenshot, font, diagram

Description automatically generated

C2 model of interacting with the system

A diagram of a system

Description automatically generated with low confidence

C3 model of interacting with the system

A picture containing text, diagram, screenshot, plan

Description automatically generated

# Object detection

As we learned before, the object detection package just listens in to the topics created by the sensory. You can start this as follows.

On your external device, link the two ROS environments and set the JACKAL as ROS master. You can do this by using this command:  
source wired-jackal.sh

You can verify you did this correctly, by using the *rostopic list* command and check for topics which have do with the lidar, or camera for example.

After that, we need to launch the appropriate packages for object detection.

In the order as shown, run:

rosrun object\_detection\_and\_marking object\_detector.py

rosrun object\_detection\_and\_marking object\_detection\_and\_marking\_node

rosrun object\_detection\_and\_marking object\_marker\_node

The object detection nodes are now live and will be looking for objects which fit their training. In this version of the detector node these objects will be stairs and doors.

When an object is found, a marker will be placed on the map. Sadly, there currently is a major bug with this marker placement, which is why we have disabled it for now.  
Aside from a marker, a picture is taken and saved. This picture shows the room and the object which is detected. This object is denoted with a box around it, with a prefix as to what the program thinks it is.

## Integration on a jetson

To extend the proof of this object detection package. The software was moved from a testing laptop to a jetson microcontroller.

This microcontroller gets its power from an USB cable plugged directly into the jetson. And publishes data via an ethernet cable.

The code was instantly deployable on the jetson, which further proves the modularity of the product made.

To mimic real life, the jetson launches and connects all nodes and services to accommodate for the object detection functionality.

Currently the pictures are stored on the jetson itself, though the scope of this product is fairly limited to the connectivity of making a new system part of an already existing system. Therefore, the processing of these images is a job left to do for another time.

## Validation

To validate that our system is working correctly, we tested the robot in front of a door; to see if it would detect it. These are some pictures the robot took, indicating that it saw a door.



From this picture we can clearly conclude the robot sees doors and marks them as interesting objects.

Furthermore, the program publishes object data to the corresponding topics:

/object\_detection\_and\_marking/box\_information

* Publisher: object\_detector.py
* Subscriber: object\_detection\_and\_marking\_node
* Message type: object\_detection\_and\_marking/BoxInfo
* Description of topic: Provides information of the boundary box of detected objects

/object\_detection\_and\_marking/marker\_information

* Publisher: object\_detection\_and\_marking\_node
* Subscriber: object\_marker\_node
* Message type: object\_detection\_and\_marking/MarkerInfo
* Description of topic: Provides information of the marker that needs to be visualized

/visualization\_marker\_array

* Publisher: object\_marker\_node
* Subscriber: RVIZ (when visualizing the topic information)
* Message type: visualization\_msgs/MarkerArray
* Description of topic: Provides the array of markers for visualization