Brigham Young University
AUVSI Capstone Team (Team 45)

Imaging Server Description

ID	Rev.	Date	Description	Author	Checked By
IM-005	1.0	12-11-2018	Initial release	Tyler Miller & Connor Olsen	Derek Knowles
IM-005	1.1	02-14-2019	Update diagrams. Clarification	Tyler Miller	Connor Olsen

1 Introduction

Visions imaging server will be the central interface between the raw data received from the plane the various ODLC (Object Detection Localization Classification) clients. Below is a brief overview of how it is setup and where its various components reside.

2 Motivation

This major code change was primarily motivated by using last years imaging software and observing its shortcomings. Last years software required overly complex and difficult configuration and setup in order to produce actionable manual classifications. Manual classification required 3-4 people, when it could be easily managed by 1 or 2. Finally, the system was fully dependent on ROS and was interconnected in such a way that it would be next to impossible to change a single module without the entire package being affected.

Given these shortcomings we set out to create an imaging package that emphasized the following:

1. Support up to N manual and autonomous imaging clients out of the box with no special configuration required.

2. Maximize modularity and transferability by separating the platform from ROS as much as possible.

3. Provide extensive documentation to ease future development and enable new developers to learn quickly.

4. Build and release the code base in such a way that it can be run using a single command on any platform.

3 Server

Once we fully understood last years imaging system we were able to lay out its weaknesses and create the above objectives to best address them. One of our main concerns was how the current codebase spread its information across multiple machines: one machine would hold the raw images from the plane's camera, another would hold cropped images, and still another would hold classification information. With these objectives and shortcomings in mind, a basic server-client architecture was defined as shown in Figure 1.

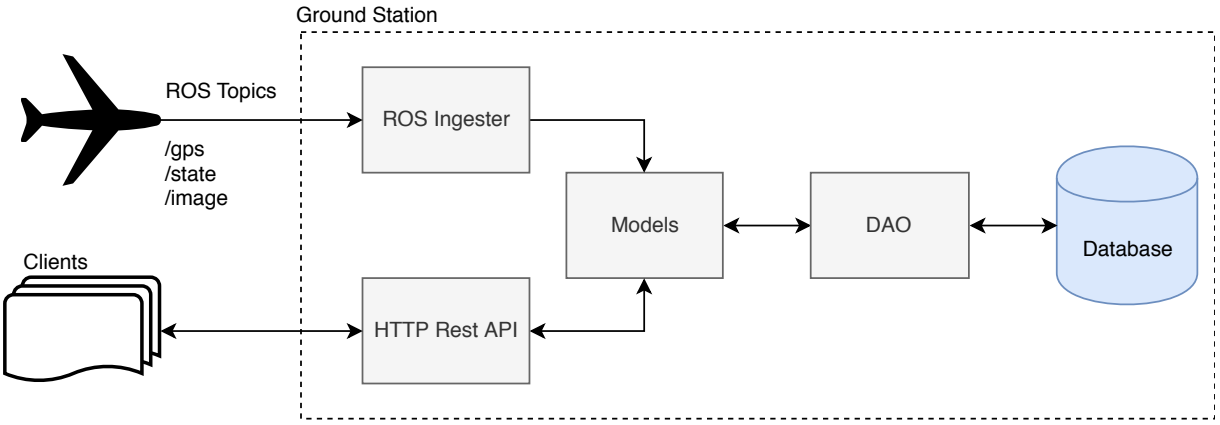

Figure 1: Server Architecture

Notice that the ROS Ingester is the only code block dependent on ROS. In reality, the ingester is a simple 200 line python script which subscribes to the relavent ROS data streams (aka: rostopics), and pushes their information into the database.

Model classes represent a single row in a table and essentially function as a translation layer between the database and the top level ingester/Rest API.

DAO's or Database Access Objects, interact directly with the database by encapsulating SQL queries into methods that can be easily accessed and called by external modules. These methods return model classes or require them as parameters.

For the database we chose Postgresql. Postgres is well known for its powerful feature set, as well as easy out-of-the-box concurrency safety. The later was particularly important in our application since there will often be multiple connections simultaneously querying most of the database.

Rest APIs are a ubiquitous interface and fit Vision's problem space well. By publishing a Rest server and building our package around its ideology, support for up to N clients comes naturally.

The basic data flow of all these components is shown in Figure 2.

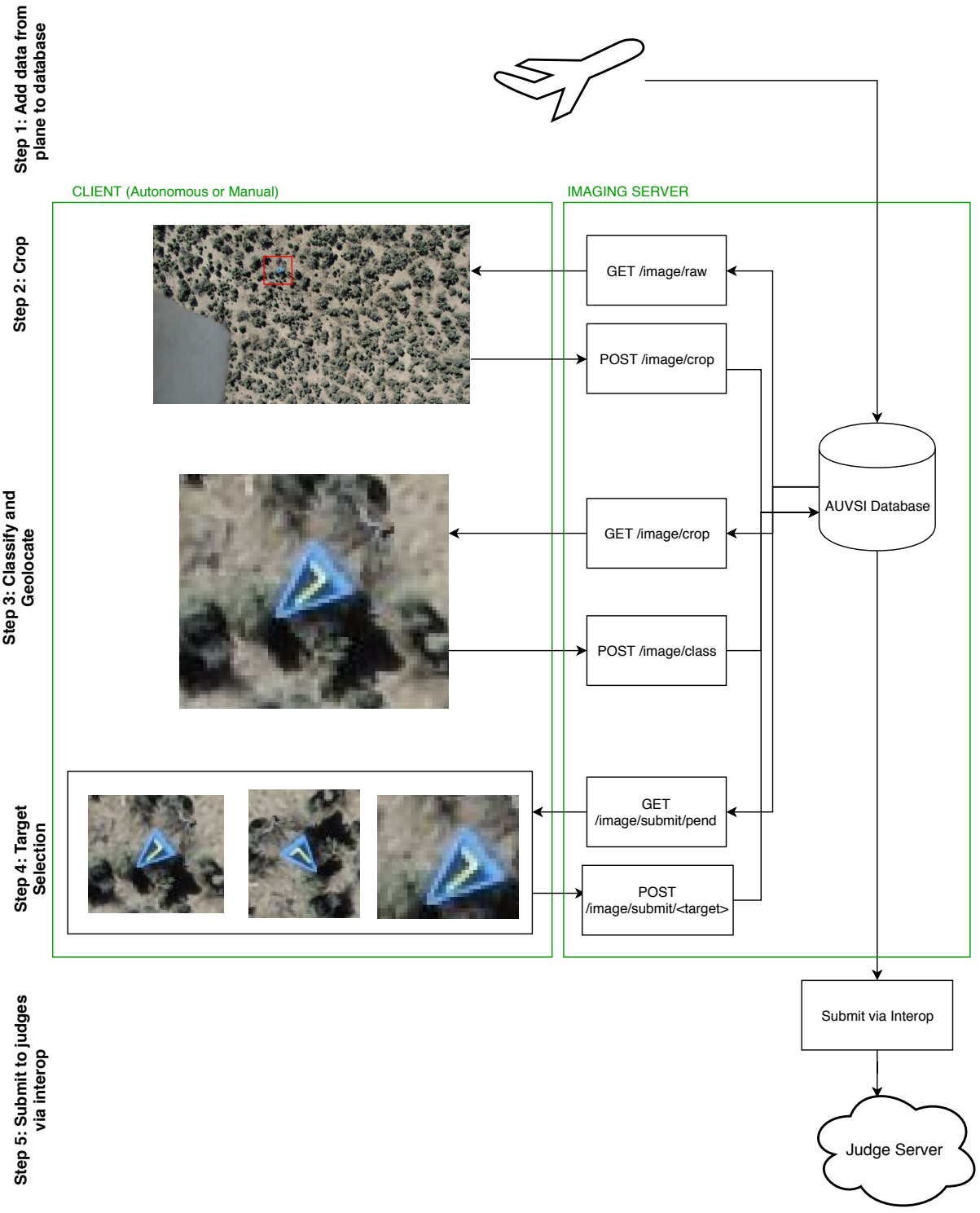

Figure 2: AUVSI Imaging Data Flow Through Autonomous/Manual Classification

Data flows back and forth between client and server, with the server holding a definitive-final copy of a target image, as well as a history of its state during intermediate steps.

Step 1 Takes all the data from the plane and pushes it into the database on the server. This is the "raw" data collected by the plane, which the server will store for the duration of the flight.

Step 2 An autonomous or manual client requests the next raw image from the server. The server automatically keeps track of which images a client has or has not seen and will only send new photos. The client then attempts to locate any targets, and will POST a cropped image to the server if it does.

Step 3 An autonomous or manual client requests an unclassified cropped image. They can then proceed to classify it, identifying attributes such as shape, letter, and color. All of these attributes are required by the AUVSI judges for maximum points. At this point the cropped image is also automatically run through a geolocation script (detailed more in IM-008) which attempts to pinpoint the targets latitude and longitude.

Step 4 Often the plane will capture multiple images of the target, and thus multiple cropped images and classifications for that target will be generated by the previous steps. The server automatically bins similar classifications as a single 'target'. In this step, users can choose which image and classification details will finally be submitted to the judges (since we want to only submit a target once).

Step 5 The server gets any targets declared ready for submission and sends them to the interop subsystem. Interop is in charge of interactions between the judge server and all subsystems and will handle submitting the final targets.

4 Conclusion

Properly implemented, the server infrastructure allows simple one click installation for anyone looking to run imaging. Coupled with the vision team's emphasis on comprehensive documentation and official code releases, the imaging codebase will be as transferable as possible for future AUVSI team members.

Besides this high degree of transferability, this design is also modular. Different portions of the codebase could be re-implemented or changed with little to no effect on other code modules. The strengths of the new imaging model respond to some of the largest issues in the old imaging codebase and create a reliable, easy to use, modular solution to AUVSI's imaging problem