FrostServer

Lewis Collum

Contents

1 Front-End: FPV Live Stream & System Statistics

Everything starts with the server on the Raspberry Pi. On the server we have a website that allows us to monitor the vehicle from a client device. At the top, we have an FPV livestream (see README.org.d/figure 1). Here we can examine the internals of our image processing system. The livestream is achieve by long-polling JPEG images from our main (Python) application to our website.

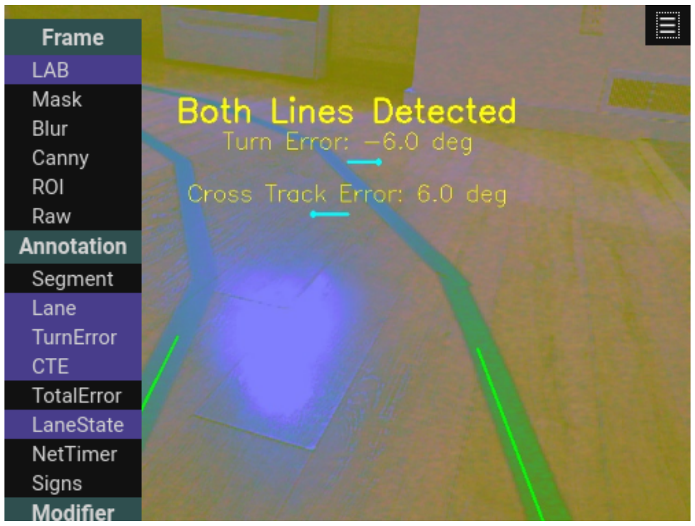

Fig. 1. Snapshot from the vehicle's website illustrating the FPV livestream. We can choose the type of frame we want to see (e.g. the LAB color-space frame) and the type of annotations we want to see (e.g. lane state and lane lines).

We are also logging various system statistics, like load, temperature, and memory. Load is measured in number of jobs in the run queue averaged over a minute. Thanks to our custom designed power supply, we can also view power usage, battery voltage and current draw (see README.org.d/figure 2).

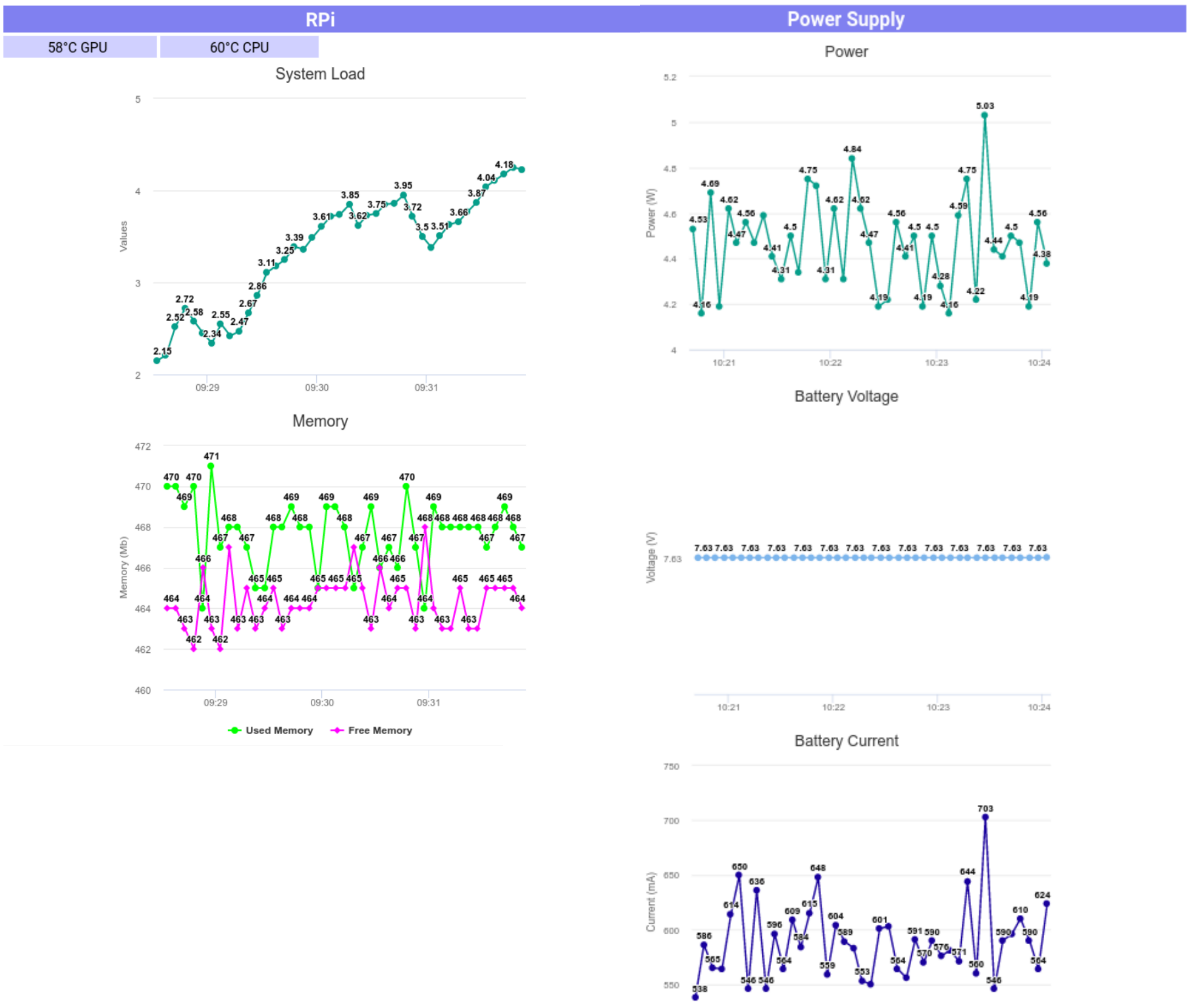

Fig. 2. Snapshots from the vehicle's website illustrating Raspberry Pi system load, memory, and temperature, as well as power supply power, voltage, and current over time.

2 Server Overview: Flask, GUnicorn, and Nginx

We use Flask, GUnicorn, and Nginx. Let's work our way down, from the website, to our application code (in python).

Nginx acts as our web server, fulfilling requests from clients for static content from our website (e.g. HTML pages, files and images). Nginx forwards dynamic requests (e.g. all of the requests that we want Python to handle) to GUnicorn, our application server.

GUnicorn ensures that Nginx and our Flask Python application can talk to each other. Ultimately, GUnicorn routes requests (passed through Nginx) to their corresponding function in our Flask application.

Flask is a web microframework in the form of a Python library. It is used in our Python application to provide functions that can receive requests, and return a response (e.g. sensor data). See README.org.d/figure 3.

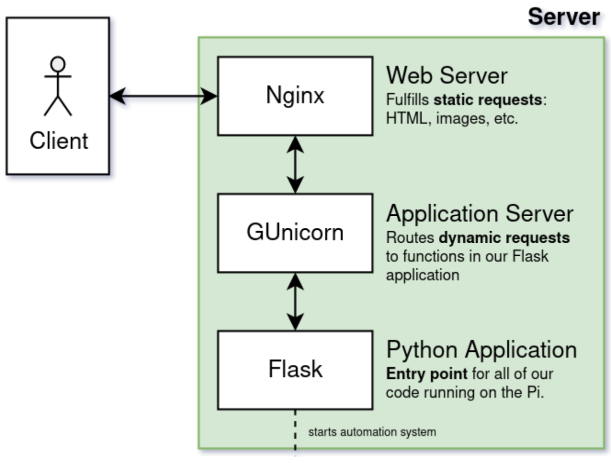
Server
Client
Nginx
Web Server
Fulfills **static requests**: HTML, images, etc.
GUnicorn
Application Server
Routes **dynamic requests** to functions in our Flask application
Flask
Python Application
Entry point for all of our code running on the Pi.
starts automation system

Fig. 3. The client can see the server's website by typing in the server IP into a browser. All static requests, for the website, like html pages, styles and images, are handled directly by Nginx. Dynamic requests, for fetching system statistics for example, are routed to GUnicorn and then to our Flask application.

2.1 Example: Real-Time Sensor Reading

We have real-time plots on our front-end that request sensor data. Eventually, we expect to receive this sensor data from a URL of our choice (e.g. "/sensor_reading"). These requests are sent to Nginx, which are then passed to GUnicorn (since they are dynamic requests). GUnicorn sends the requests to a function in our python application that is registered for the formerly mentioned URL. The function is registered with the given URL using Flask. This function has code to get sensor data from the underlying linux system (typically, by reading a sensor file or running a shell command). Finally, the function returns a response which contains the sensor data.

3 Installing Nginx, Flask, and GUnicorn

Since we are using Arch Linux ARM on our Pi, we used Arch Linux's package manager `pacman` for Nginx and Gunicorn. We used `pip` to install Flask.

```
sudo pacman -S nginx
sudo pacman -S gunicorn
sudo pip install flask
```

4 Starting the Server the First time

For ease, this repository was cloned on our Raspberry Pi to the user directory (e.g. "/home/alarm"). In the `frostServer/config` directory, there is a setup script[1] which does the following:

1) links the frostServer.service to the system directory;
2) links the frostServer.nginx config file to nginx's enabled-sites directory;
3) enables both the nginx and the frostServer services;
4) starts both services.

Now, on a computer connected to the same WiFi as our Pi, we can type the IP of the Pi into the browser search bar (e.g. "192.168.0.106") and the Frost website appears. Since we enabled the server service, the server will start at boot without any user intervention.

1. The setup script should be run as `sudo`.

5 Automation System

The server is also responsible for starting the automation system. The automation system begins in code with a Frame Subject (see README.org.d/figure 4), which is a simple object that captures images, or as we will call them, frames, continuously from the camera. We have two models which take a copy of each frame: the Lane Model, and the Sign Detection Model. Since these models are both computationally expensive, we set them up to process the frames from the Frame Subject in their own thread. This ensures that other parts of our application are not slowed down by these models. It is also important to note that these models are fully independent from each other. That is, one model does not affect the result of the other model.

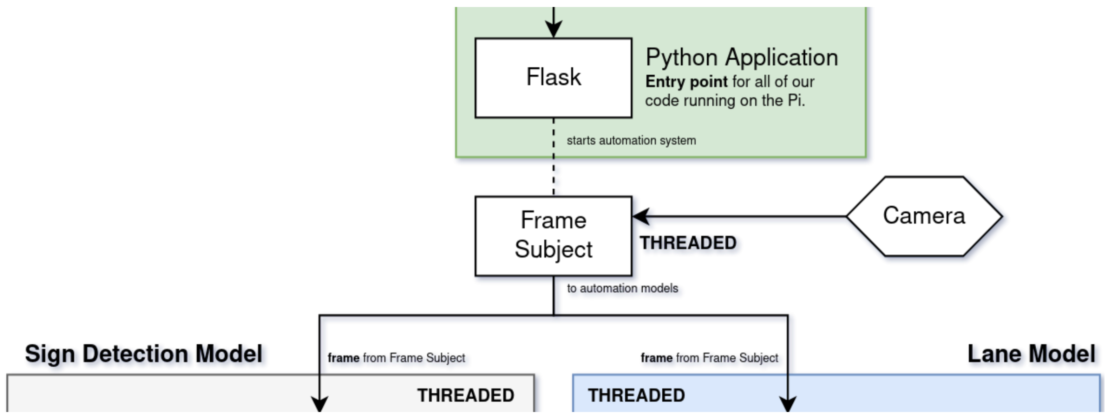

Fig. 4. Illustrates the Frame Subject being started (as a thread) from the server (Flask) application code, from which, the lane and sign detection models can pull camera frames.