Embedded Systems Project Report

(IoT)-based Vehicle Tracking System

Fall 2023
SUT

Master: Dr. Ansari
TA: Mr. Oustad and Mr. Ansari

Arash Yadegari 99105815
Ava Cyrus 99171344

Table Of Content

Project Objective

The primary objective of the project is to develop and implement an advanced Internet of Things (IoT)-based Vehicle Tracking System, which is capable of efficiently locating, analyzing, and transmitting data from the devices integrated into cloud systems. The device must also be able to take pictures of its surroundings.

Additional Point: Define Web Applications for querying and inspecting the location of Raspberry Pi devices.

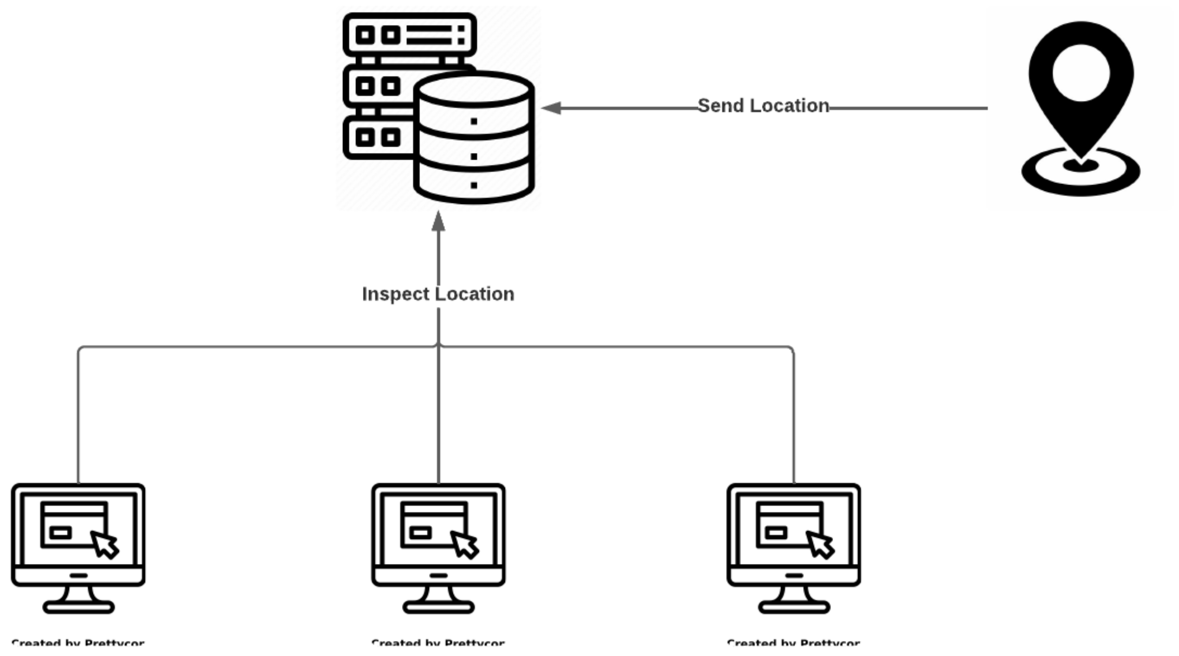


Figure 1: System Overview

Hardware Prequestics

The location of the device needs to be retrieved by a GPS module. The location must be transmitted to a cloud using a WIFI module.

List of our modules:

- Neo 6M as GPS module
- Raspberrry Pi 3 Model B as WIFI module (integrated into it)
- Camera (cell phone able to connect to a wifi)

Software Prequestics

Cloud storage is needed for saving and reading data. Back-end application for querying data. Front-end app for displaying data.

- Virtual Private Server is used as cloud storage
- Express app as back-end service
- Vanilla + http-server as front-end service

GPS Monitoring System

In this section, the infrastructure for capturing data from the GPS module and subsequently transmitting it to the server is outlined.

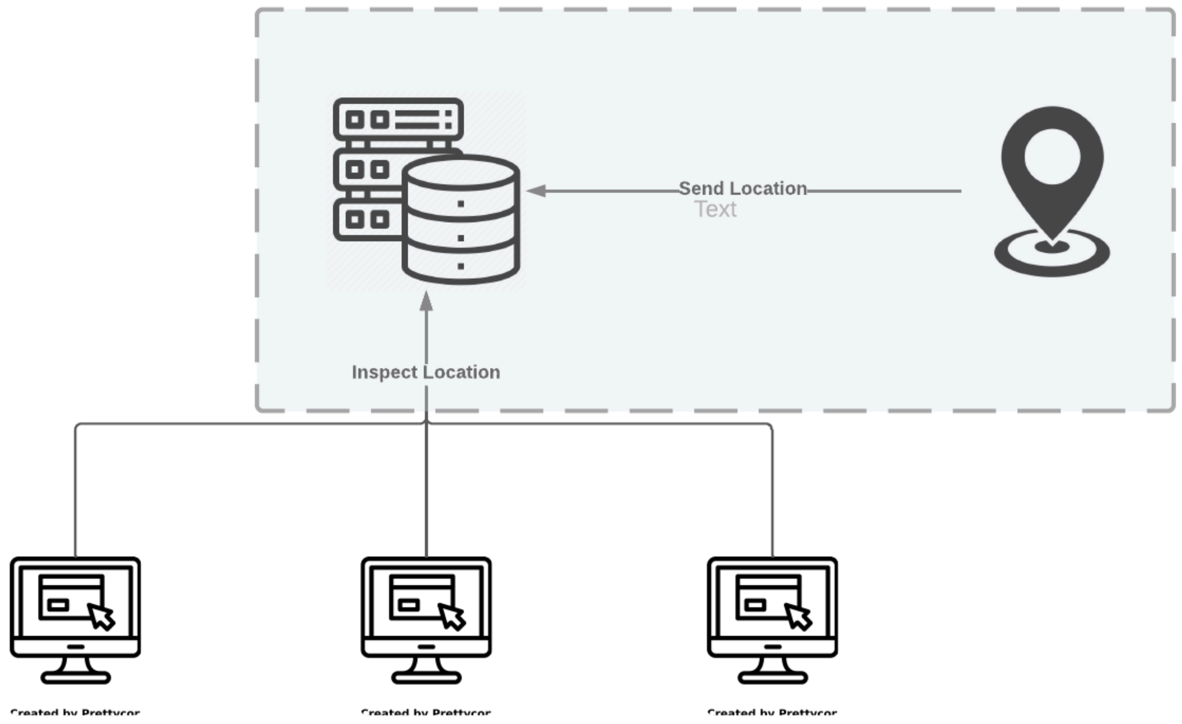


Figure 2: GPS Monitoring

To receive and transmit location, following steps must be taken.

- Use GPS modules to receive NMEA format data
- Parse NMEA format data into geo point (latitude, longitude).
- Send data to the Server
- Receive data from Client

Connecting Neo 6M to Raspberry

The GPS module needs to be connected into the raspberry. The mapping is done as below:

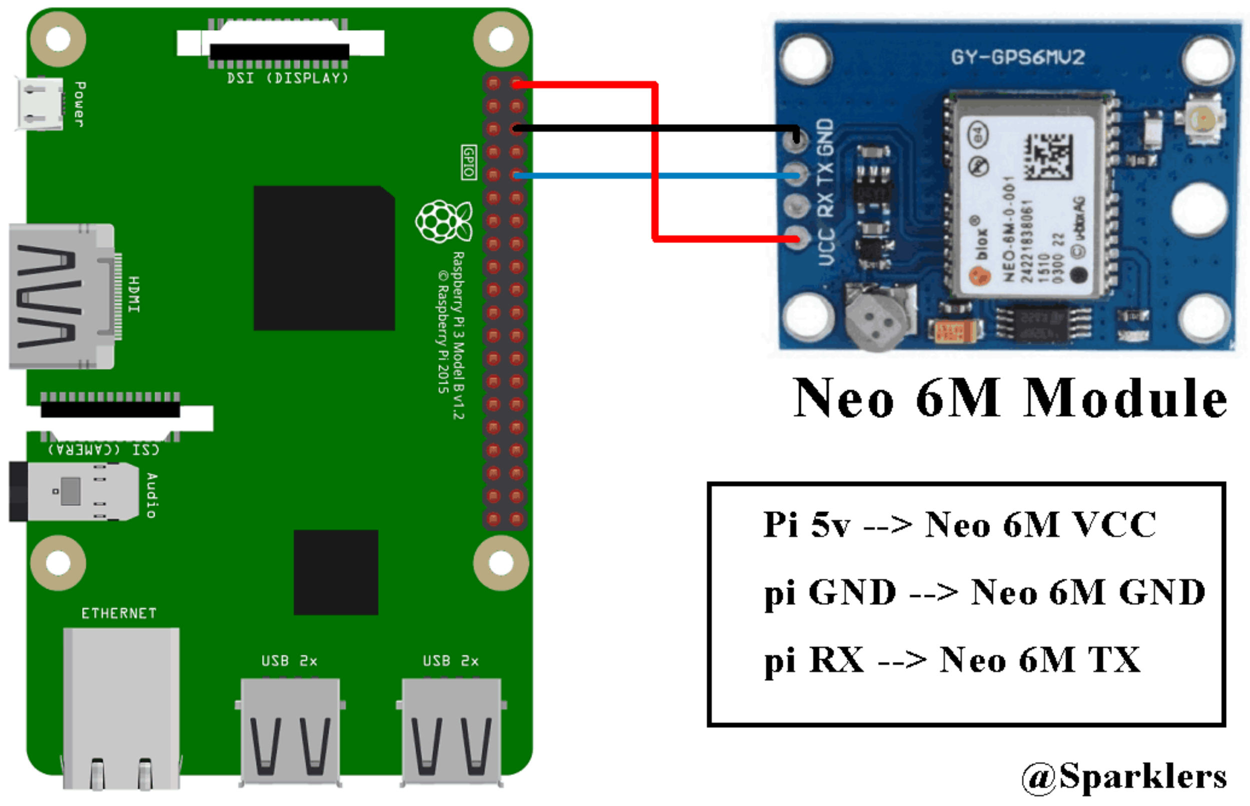


Figure 3: Neo6m Connection

The Transmit Pin of Neo 6M is connected into the Receive Pin of Raspberry (P 5). VCC of GPS can be set to 3.3V (P2) or 5V (P1). However, 3.3V is preferred based on the Neo 6M data sheet.

Baud Rate Configuration

The default baud rate of Raspberry is set to 115200. Neo 6M transmits data at 9600 baud rate. For retrieving correct data, the baud rate of Raspberry needs to be reconfigured. This can be done by setting up /env/config.

Raspberry Pi 3 Issue

On the Raspberry Pi Model 3B the hardware-based serial/UART device /dev/ttyAMA0 has been re-purposed to communicate with the built-in Bluetooth modem and is no longer mapped to the serial RX/TX pins on the GPIO header. Instead, a new serial port "/dev/ttyS0" has been provided which is implemented with a software-based UART (miniUART). This software-based UART ("/dev/ttyS0") does not support PARITY and some have experienced some stability issues using this port at higher speeds.

The bluetooth functionality is not needed therefore the serial URT can be remapped to ("/dev/ttyAMA0"). In this case the transmitted data are more reliable.

Sampling From Neo 6M

By configuring the above settings, the `cat /dev/ttyAMA0` command can be used to retrieve data.

```
Sample:
```

```
$GNRMC,005551.00,A,3543.93696,N,05122.60002,E,0.058,,310124,,,A*60
$GNVTG,,T,,M,0.058,N,0.107,K,A*36
$GNGGA,005551.00,3543.93696,N,05122.60002,E,1,05,1.84,1372.0,M,-17.6,
M,,*56
```

The retrieved information are sent in NMEA format. Lines starting with GNxxx points to Glonass Satellite data and GPxxx points to regular (USA) satellite data.

3 last characters show message class. For example RMC contains latitude, longitude and timestamp.

Parse NMEA format data

There exist different NMEA data parsers. In this project, `pynmeagps` is used. The following code reads data from serial0(ttyAMA0) and parses it. Only RMA type data are used for better precision.

```
import serial
from pynmeagps import NMEAReader, NMEAMessage

stream = serial.Serial(SERIAL, baudrate=9600)
nmr = NMEAReader(stream, validate=0x01)


def get_location():
    global CL
    for (raw_data, parsed_data) in nmr:
        msg: NMEAMessage = parsed_data
        pdt = msg.msgID
        if pdt == 'RMC':
            lat = None
            lon = None
            if msg.lat != "":
                lat = float(msg.lat)
            if msg.lon != "":
                lon = float(msg.lon)
            if msg.lon and msg.lat:
                logger.info("latitude : " + str(lat) + " longitude :
" + str(lon))
                return lat, lon, get_timestamp()
```

Send Data to Server

Data can be transmitted by opening a UDP socket and sending data to the listeners address of the server. This can be done by following code:

```
d_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)


def post_data(lat, lon, timestamp):
    logger.info("POST UPDATE")
    msg = f"UPDATE {timestamp} {lat} {lon}"
    d_socket.sendto(msg.encode(), SERVER_ADDRESS)
```

Too Many Requests Issue

Sending the same data over and over may cause overhead on communication. Two policy are used for addressing this issue:

- Every 15 sec: Check if Current Location is Far Enough From previous location. If yes, then send a new location.

- Every 60 sec: Push location into server. This must be done due to updating the timestamps which are used in querying by back-end system

The implementation is done by `schedule`:

```
schedule.every(15).seconds.do(update_location)
schedule.every().minutes.do(update_location, force=True)
```

```
def update_location(force: bool = False):
    if not force:
        logger.info("Check location...")
    else:
        logger.info("Update location...")

    global CL
    lat, lon, timestamp = get_location()
    if not lat or not lon:
        return

    d = get_dist(lat, lon)
    if d > DIST_THRESHOLD or d < 0:
        post_data(lat, lon, timestamp)
    elif force:
        post_data(lat, lon, timestamp)

    CL = (lat, lon)
```

Receive Data from Client

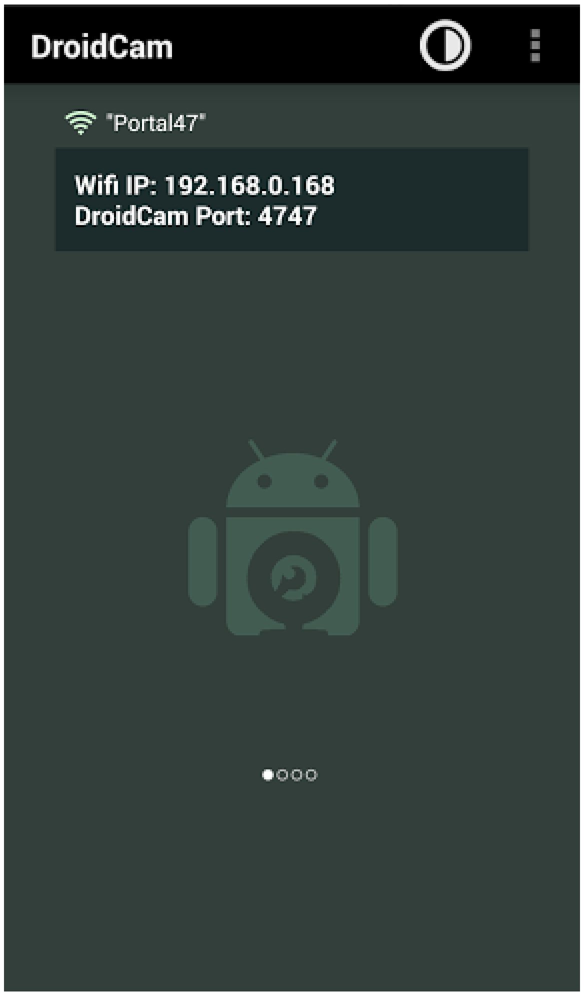
Open and bind a UDP socket. Listen for incoming connections.

```
def run():
    s_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    s_socket.bind(SERVER_ADDRESS)

    while True:
        data, adr = s_socket.recvfrom(4096)
        logger.info("RECEIVE: " + data.decode())
        update_csv(data.decode())
```

Camera Setup

The camera device can be virtualized using v4l2lookback. In this case, the cellphone camera can be used as a camera as long as it is connected to the same network. DroidCam app is used to set up an IP and connection for sending frames.



Wifi IP: 192.168.0.168
DroidCam Port: 4747

Figure 4: Droidcam app view

The following bash script is written for connecting into the camera and receiving the data.

```
#!/bin/bash
sudo modprobe v4l2loopback devices=1 max_buffers=2 exclusive_caps=1
```

```
card_label="VirtualCam #0"
sudo ffmpeg -i http://$1:$2/video/ -f v4l2 -pix_fmt yuv420p
/dev/video0
```

This will create one virtual device (this can be any /dev/videoX, in our case it is 0). Ffmpeg will capture the data at $1:$2 and send it into the created virtual device.

At this point, the frame can be saved by using following script

```
#!/bin/bash
fswebcam -r 1920x1080 --png 9 -d /dev/video0 -S 40 -s Gamma=50% -s
Contrast=15% -s Brightness=70% -s Sharpness=100% --no -banner
./shots/$1.png
```

Back-end Service

The transmitted data must be queried to retrieve desirable results. Express framework is used as the back-end stack of this server.

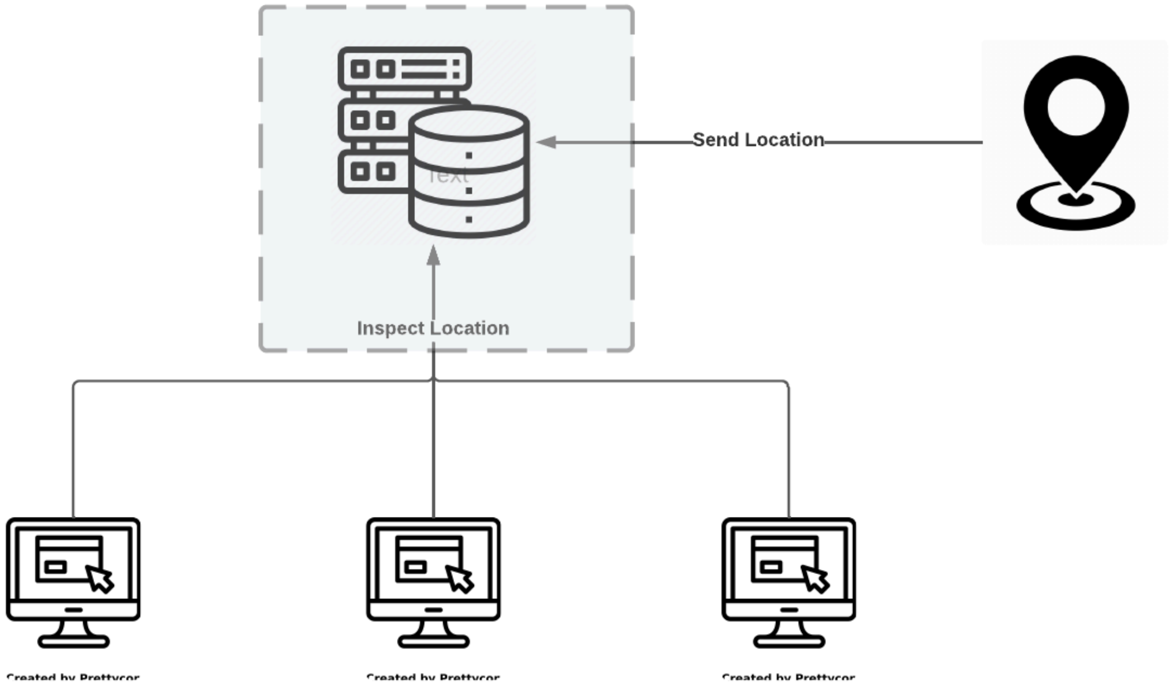


Figure 5: Back-end Service

APIs

Following APIs are implemented:

Get Current/Most Recent Location

Brief

Retrieve the most recent location

Request Format

```
GET {{base_url}}/location
```

Response Format

```
{
  "timestamp": "...",
  "lat": "...",
  "lon": "..."
}
```

Search Date and Time

Brief

Return at most 10 data related to input date and time.

Request Format

```
GET {{local}}/location/search?date=$1&time=$2
```

Response Format

```
[{
  "timestamp": "...",
  "lat": "...",
  "lon": "..."
}, ...]
```

Front-end Service

Show location of GPS on map.

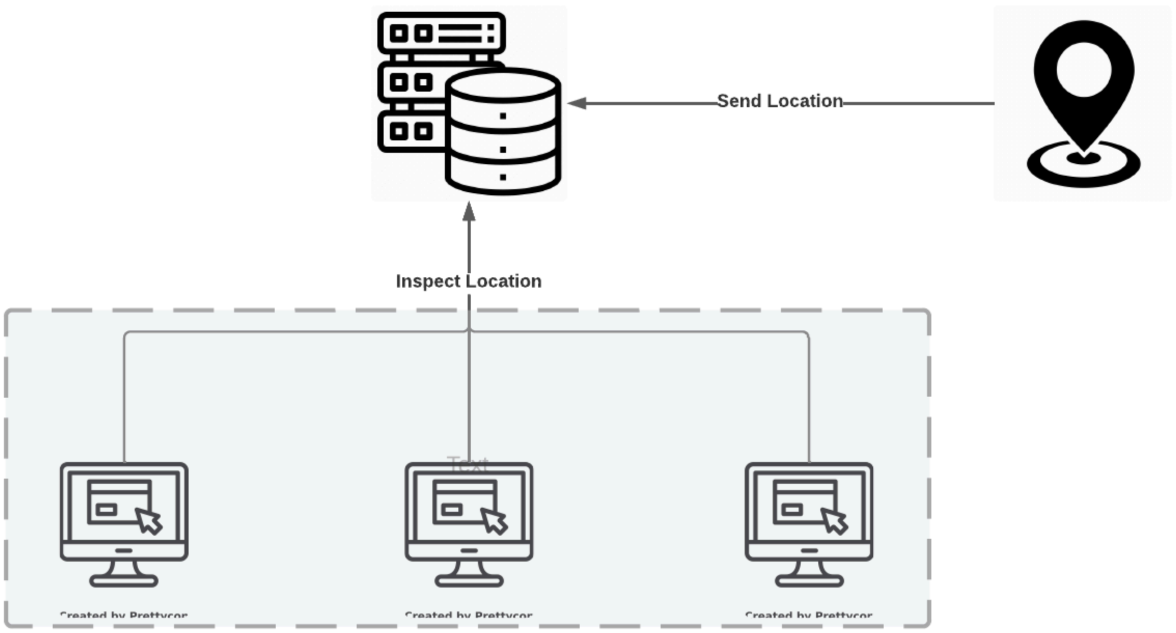


Figure 6: Front-end Service

The following shot shows the query result of /location.

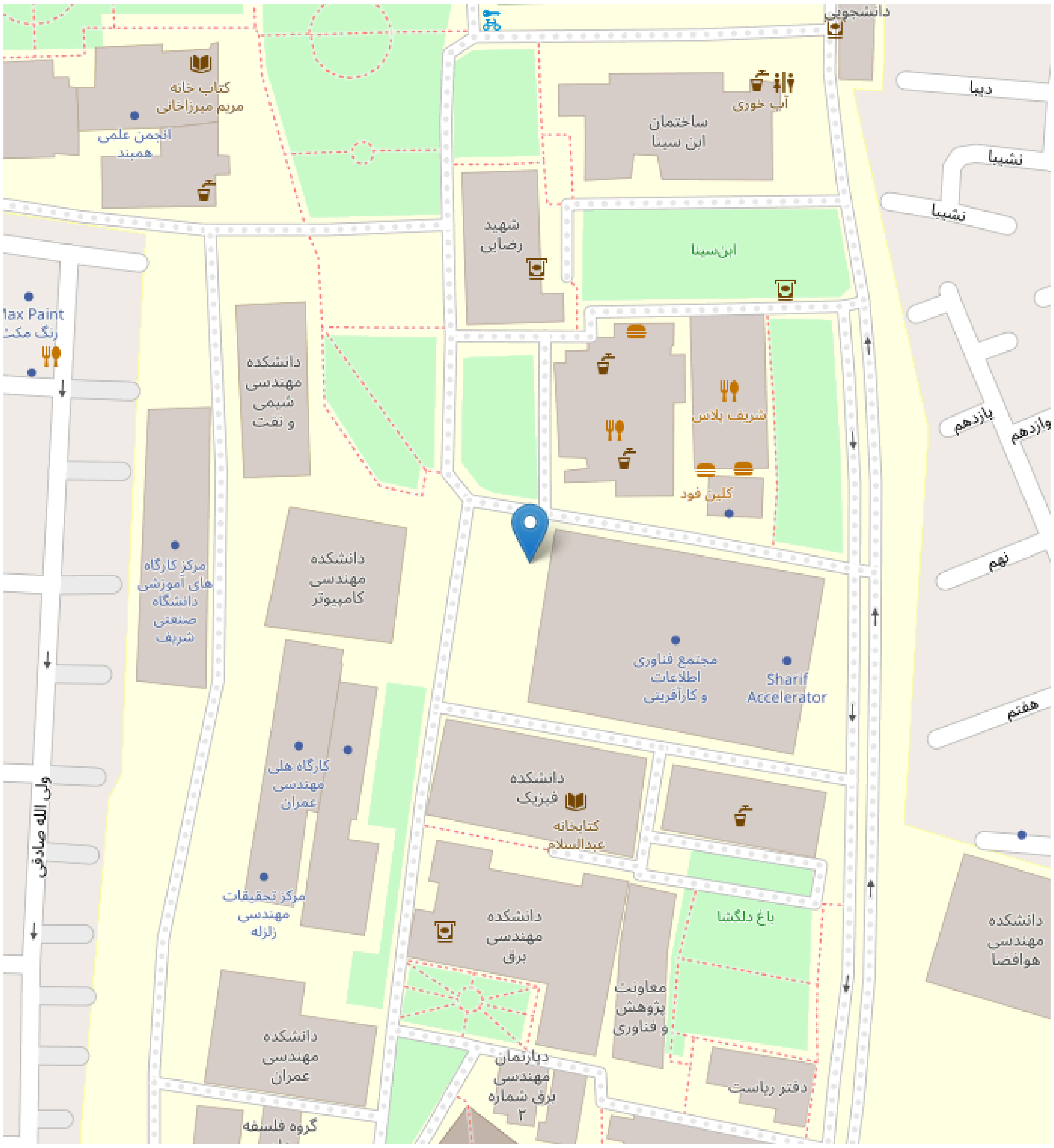


Figure 7: university test log