CS498 IoT Lab 2
Rick Bischoff (rdb4)

For this assignment, we want you to think about what wireless technologies you would ideally use. If you were designing a real, self-driving car platform, with real (big) cars, that would drive all over the United States, which technology would you use? ***Please include in your report a description and analysis of what protocols you would use and why (brief is fine - a paragraph or so).***

There are a variety of use-cases to consider with a self-driving car. For communication with the driver and passenger devices, Bluetooth is the current obvious choice and its use is already pervasive for this function. For fleet management (tracking cars across the globe), there are a few choices to make here. There is the expense of communicating on the network to consider, how reliable service is, coverage area and so on. For the sake of argument, I'd go with a 3G-type network. The coverage here is very good (across continental United States), the cost is dropping, equipment is ubiquitous, and bandwidth requirements for tracking coordinates are not very high. I would not use this type of network for firmware patches or uploading camera data, however--likely these are high-bandwidth, high-reliability situations. You'd want these to be done in a safe environment, probably while the car is parked in the customer's garage. For this, I would think Wifi would be a better choice.

The last use case to consider is crash-avoidance. The lecturers mentioned there isn't a good solution for this, so we'd likely have to push our own. I'd envision the requirements mid-near range. The range needs to be far enough so that you can pick up a signal before it's too late to do anything about--e.g. A car traveling 90 mph at you, ¼ mile away, you only have 10 seconds to get out the way. It gets worse when you're thinking of two cars traveling 90 miles towards each other, you only have 5 seconds. That's likely enough time, so we'll say ¼-½ mile range. Further you'd not want point to point communications, but rather every communication to be multicast. Example data might be something like "My heading is x,y @ v. I have other vehicles in my sight [x1,y1,v1] [x2,y2,v2]." Possibly other messages like "I have detected vehicle v1 is about to collide with me. I am veering right." The larger point is that you can come up with an agreed protocol to avoid the need for send-response type communication, and just have broadcast/listen mode instead.

Video Demos

- Please see Video
- https://www.youtube.com/watch?v=h1Aw4qXkjko
-

Notes About Architecture

I found the lab instructions for the full architecture to be ambiguous so I took some liberties in how I structured the final front- and back-end interfaces.

Mimicking what I'd perceive a "real" system to be, I didn't think it'd make a ton of sense for the car itself to provide much of the heavy lifting in terms of compute or presentation. For that reason, the car communicates solely with a Linux server via Bluetooth. The car runs a light-weight server that listens for commands via bluetooth and calls out to various python functions to implement.

The Linux server serves as a middleman. The server itself runs python/Flask, and is connected to the Internet as well as the car (via Bluetooth). It runs Flask to listen for web traffic, which it then translates into Bluetooth messages. It handles all communication of the car.

The front-end itself can be distributed to anything with Internet access. It is built on Electron, and communicates with the above Linux server via the HTTP protocol. There are various end-points exposes that correlate with the underlying car mechanisms. For example, a POST request to "/forwards", will result in a "forwards" signal being given to the car via Bluetooth, and then the Car itself listens for that, and translate that into a "Car" class "forward" method, which ultimately results in a "picar_4wd.forward()" function call.

The same architecture applies to the camera. I couldn't find generic code to both send/receive files over Bluetooth, other the OBEX protocol. Not willing to implement that, I wrote a straightforward method that base64-encodes a stream of bytes into 512-byte chunks, and sends them 1-by-1 to the server. It is pretty slow but it gets the job done.

Overview of File Structure

https://github.com/rdbisch/cs498_iot_lab2

The files highlighted yellow are the important ones.

/

/Car.py - This is a Python class I designed to abstract Car functions
/bt_client.py - Provided code for demonstrating Bluetooth. Not used for final solution.
/bt_server.py - This code runs on the car and functions as the Bluetooth listener.
/electron_backend.py - This code runs on the server and functions as a python/Flask server and the Bluetooth sender.
/start.sh - Simple shell script to start execution of electron_backend.py
/wifi_client.py - Provided code to demonstrate wifi connectivity. Not used in final.
/wifi_server.py - Provided code to demonstrate wifi connectivity. Not used in final.
/web/ - Electron files
/web/index.html - Provided file to demonstrate electron UI. Not used in final
/web/index.js - Primary front-end engine. It is a lot of boiler plate for making POST/GET requests to Flask, and parsing the results.
/web/main.html - The front-end web page that provides the interface.
/web/main.js - Provided boilerplate to start the Electron app.
/web/package.json - Provided package requirements for electron
/web/preload.json - Empty file to remove error message.