CarSensor: Smart Street-side Parking with the Internet of Things

Daniel Lynch
Deparment of Mechanical Engineering
Northwestern University
2145 Sheridan Road
Evanston, IL 60208, USA
daniellynch2021@u.northwestern.edu

Yong Zhao
MPM Program
Northwestern University
2145 Sheridan Road
Evanston, IL 60208, USA
yongzhao2019@u.northwestern.edu

***Abstract*—We propose `CarSensor`, a networked array of sensors that provide parking information to users via a smartphone app. The design is guided by a desire for simplicity and modularity, two important attributes of Internet-of-Things systems and of cyberphysical systems in general. We describe our proof-of-concept prototype, built using popular open-source hardware and software. Based on this prototype, we then identify areas for further development that constitute prerequisites for full-scale implementation.**

I. Introduction

The recent surge in popularity of the Internet of Things (IoT) highlights some fundamental aspects of the design and analysis of cyberphysical systems, such as deterministic behavior, robustness, and extensibility. For the final project of EECS 395/495 Cyberphysical Systems (Winter 2018), we created `CarSensor` [1], a prototype of an IoT-inspired system for smarter street-side parking, which addressed these aforementioned aspects of cyberphysical systems.

A. Motivation

Street parking can be a hassle, especially in urban areas. Knowing which parking spaces are full or empty can save a lot of time and aggravation. One way to realize this is by outfitting a street with an IoT system consisting of networked sensors and a client application all communicating with a central hub.

B. Modular System Design

`CarSensor` is intended to modular, so that it can be expanded to support an arbitrary number of parking spaces. As shown in Figure 1, `CarSensor` consists of three types of components:

- an arbitrary number of *sensor modules* located at parking spaces,
- a *server* that collects data from the sensor modules and reasons about the state of each parking space, and
- a *smartphone app* that provides parking information to the end user.

Each sensor module consists of two sensors, a microcontroller, and a network adapter. Each module is responsible for sensing two adjacent parking spaces, as shown in Figure 2.

[1] `https://github.com/yongllyong123/project-1`

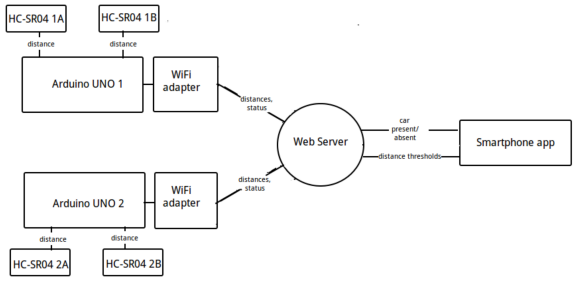

Fig. 1. Block diagram of components used in our `CarSensor` prototype.

The parking space in the middle of the figure is sensed by module 1's sensor B and module 2's sensor A; the parking space on the right would be sensed by module 2's sensor B and module 3's sensor A, and so on.

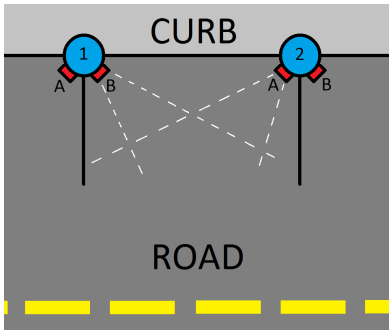

Fig. 2. Physical layout of networked sensors used in `CarSensor`.

Each sensor module must also have a unique network identifier related to its physical location to allow the server to reason about the state of each parking space. In our prototype, we used IP addresses, but the limitations of that approach will be discussed in Section IV.

II. Physical Layer

Our prototype design was motivated by speed and ease of development: we selected components that were easy to get up and running, bearing in mind that we were only developing a prototype and not a production-ready system.

A. Arduino UNO microcontroller

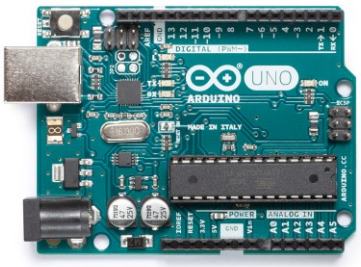

Fig. 3. Arduino UNO (image credit: Arduino)

Shown in Figure 3, the Arduino UNO is an open-source development board built around the ATmega328P microcontroller and has become a popular platform for prototyping simple embedded applications. We selected it for its overall ease of use due to its integrated development environment (IDE) and the large user base.

B. HC-SR04 ultrasonic distance sensor

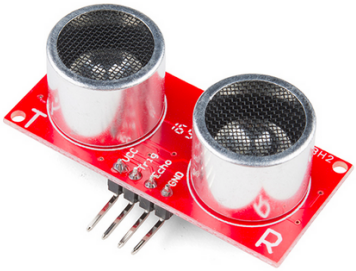

Fig. 4. HC-SR04 ultrasonic distance sensor (image credit: SparkFun Electronics ®)

The HC-SR04 sensor, pictured in Figure 4, has two communication pins: `TRIG` (a digital input) and `ECHO` (a digital output). Its operational principle is similar to sonar: when `TRIG` is toggled high, the sensor emits an ultrasonic pulse; if an object reflects the pulse back to the sensor, it sets `ECHO` high. The microcontroller simply times the delay between setting `TRIG` high and receiving a logical 1 on `ECHO` and multiplies that time by the speed of sound (343 $\frac{m}{s}$ in air at sea level) to estimate how far away the object is.

`CarSensor` is more concerned with detecting the presence of a vehicle in a parking space than with measuring distance, so measurements made with the HC-SR04 are compared to a threshold. If the measured distance is less than the threshold, a `proximity` bit, specific to that sensor, is set to 1 and sent to the server via the WiFi adapter; otherwise, `proximity` is set to 0 and then sent to the server. Since each Arduino uses two HC-SR04 sensors, each Arduino sends two `proximity` bits to the server, delimited by a space. This transmission occurs at a regular frequency (0.5 Hz in our prototype); this will be expanded on in Section IV.

C. ESP8266 WiFi adapter

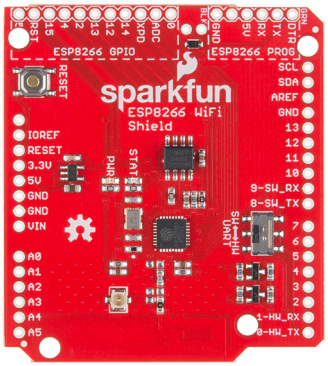

Fig. 5. SparkFun ESP8266 WiFi shield for Arduino UNO (image credit: SparkFun Electronics ®)

After selecting our microcontroller, we needed to select a compatible WiFi adapter. We chose SparkFun's ESP8266 shield, pictured in Figure 5, which is built around the ESP8266 system-on-a-chip (SoC) and provides TCP server and client functionality. It interfaces with the Arduino UNO over "software serial", leaving the Arduino UNO's hardware serial port available for other communication uses (debugging via a terminal emulator, in our case). We made use of SparkFun's Arduino library and starter code[2] to accelerate the ESP8266's learning curve

III. Network Layer

A. Sensor clients

Using the ESP8266 in TCP client mode, each Arduino UNO sends proximity information from both HC-SR04 sensors to the server at 0.5 Hz.

B. Server

We used `Node.js`® [3] to develop a rudimentary server to collect data from each sensor module and handle requests from the smartphone app. Specifically, we used the `net`[4] module to create an asynchronous streaming TCP server with that listens for requests on two ports (one for sensor modules, another for the smartphone app). In our prototype, we ran the server on a Lenovo laptop running Ubuntu 14.04 LTS.

1) TCP sockets: Sensor modules communicate with the server on port 8080, and the smartphone app communicates with the server on port 8000. This decision was motivated by the need for a modular system; moreover, it simplifies the server design because TCP requests from the smartphone app do not need to be filtered out from sensor modules' TCP requests.

[2] `https://github.com/sparkfun/SparkFun_ESP8266_AT_Arduino_Library`

[3] `https://nodejs.org/en/`

[4] `https://nodejs.org/api/net.html#net_net`

2) State machine: The core of `CarSensor` is a simple finite state machine (FSM) running on the server. As shown in Figure 6, the FSM has four states:

- `empty`,
- `pending full`,
- `full`, and
- `pending empty`.

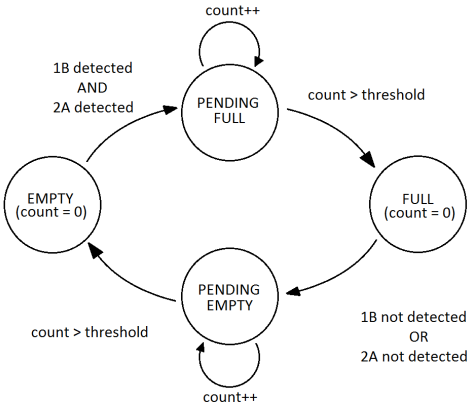

Fig. 6. Finite state machine for vehicle detection

A change in the two `proximity` bits that correspond to a parking space triggers a transition from `empty` or `full` to one of the "pending" states. The FSM then uses a counter to determine when to transition from a "pending" state to `full` or `empty`. If the change persists until `count` reaches a `threshold` value, the state transitions; otherwise, it reverts to the state before the "pending" state. Using a counter and a threshold filters out high-frequency events (e.g., pedestrians walking through parking spaces) and preserves low-frequency information (the presence/absence of a vehicle in the parking space). In this sense, `CarSensor`'s FSM is functionally equivalent to a *switch debouncer*.

Note that `count` increments at the rate at which the sensor module sends new `proximity` data, so `threshold` represents a time duration. In our prototype, the value of `threshold` is 5, corresponding to 10 seconds.

C. Smartphone app client

We elected to use Android Studio[5] to build the end user's interface with our `CarSensor` system. This decision was largely motivated by our familiarity with Android Studio and our unfamiliarity with developing iOS apps.

The app, shown in Figure 7, is a simple TCP client that communicates with the server on port 8000. When the user presses "CONNECT...", the app sends a request to the server and the server replies with its estimate of the state of the parking space(s).

To ease the development process, we provided a field for the user to enter the server's IP address, because it changed depending on whether we were working on campus or at home.

[5]`https://developer.android.com/studio/index.html`

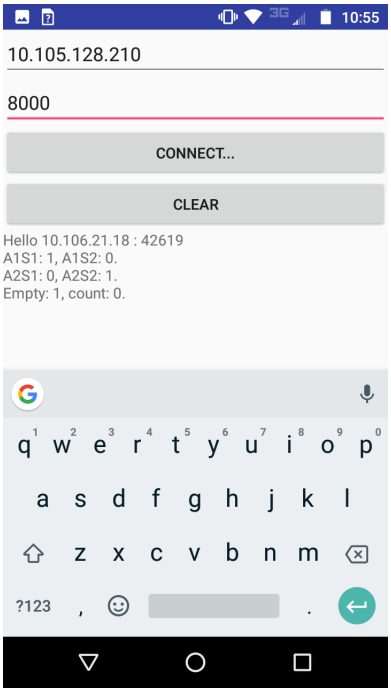

Fig. 7. Screenshot of the `CarSensor` Android app

IV. Future Work

While developing our prototype, we identified many avenues for improvement. Some of these ideas were not implemented due to a lack of time, others due to a lack of technical experience.

A. Ease of use

In the prototype, the user has to click "connect" every time he/she wants to see the newest information about parking spaces. An automatic notification feature would enhance the user experience. More fundamentally, the prototype app requires the user to know where each parking space is, which is unreasonable. A better app would integrate a maps-type feature with the existing notification system.

B. Extensibility

As alluded to in Section I, identifying each sensor module by IP address is a flawed approach for a couple reasons: first, IP address does not correlate with physical location, so the server cannot use IP addresses alone to reason about the state of each parking space; second, routers can only distribute a limited number of IP addresses. One option is to simply program each sensor module with an address upon installation, but this is labor-intensive and not very flexible. Programming each sensor module to transmit its ID along with its data also frees the server from having to know the IDs of all the sensor modules, thus making the system easier to expand.

C. Robustness

Recall that each sensor module transmits once every two seconds. This signal contains data (and could also contain

an ID) but also functions like a heartbeat. The server simply needs to keep track of time since the last transmission from each sensor module. If "too much" time has elapsed since a particular module's last transmission, the server can advise a maintainer to check on that module.

D. Power

We offer two suggestions for improved power efficiency. We considered using solar power for our project but quickly decided against it, due to unnecessary complexity and the short time frame alloted for the project. Solar power is problematic because of fluctuating and unpredictable power output, so a batter and both charging and discharging circuits are also necessary. Nevertheless, solar power could be useful in a full-scale implementation of our system.

Our second recommendation is a low-power mode for the sensor modules. The ESP8266 draws up to 170 mA of current when transmitting. If the sensor module is running on a battery, a low-power mode with reduced transmission frequency (e.g., 0.05 Hz or even 0.005 Hz instead of 0.5 Hz) would extend battery life.

E. Security

We did not consider security risks when designing this prototype. There are two primary attack vectors: attacking the server through the app's port (port 8000) or through the sensor modules' port (port 8080). Neither of these require authentication in our prototype, so they are security risks. A third (albeit more difficult) attack vector is through the sensor module itself: since the ATmega328P can be reprogrammed, it can be hacked. Countermeasures that achieve immunity to hacking without sacrificing reprogrammability warrant further investigation.

V. Conclusion

We presented `CarSensor`, an IoT-inspired system for notifying drivers of empty and full parking spaces. The system is highly modular and consists of a physical layer and a network layer. We built a prototype using widely available open-source development tools, including the Arduino UNO, `Node.js`®, and Android Studio. Our prototype demonstrated `CarSensor`'s viability and also illuminated a number of areas for improvement that are relevant to most cyberphysical systems.

References

1) SparkFun ESP8266 AT Arduino Library. `https://github.com/sparkfun/SparkFun_ESP8266_AT_Arduino_Library`
2) `Node.js` main page. `https://nodejs.org/en/`
3) `Node.js net` API documentation. `https://nodejs.org/api/net.html#net_net`
4) Android Studio main page. `https://developer.android.com/studio/index.html`