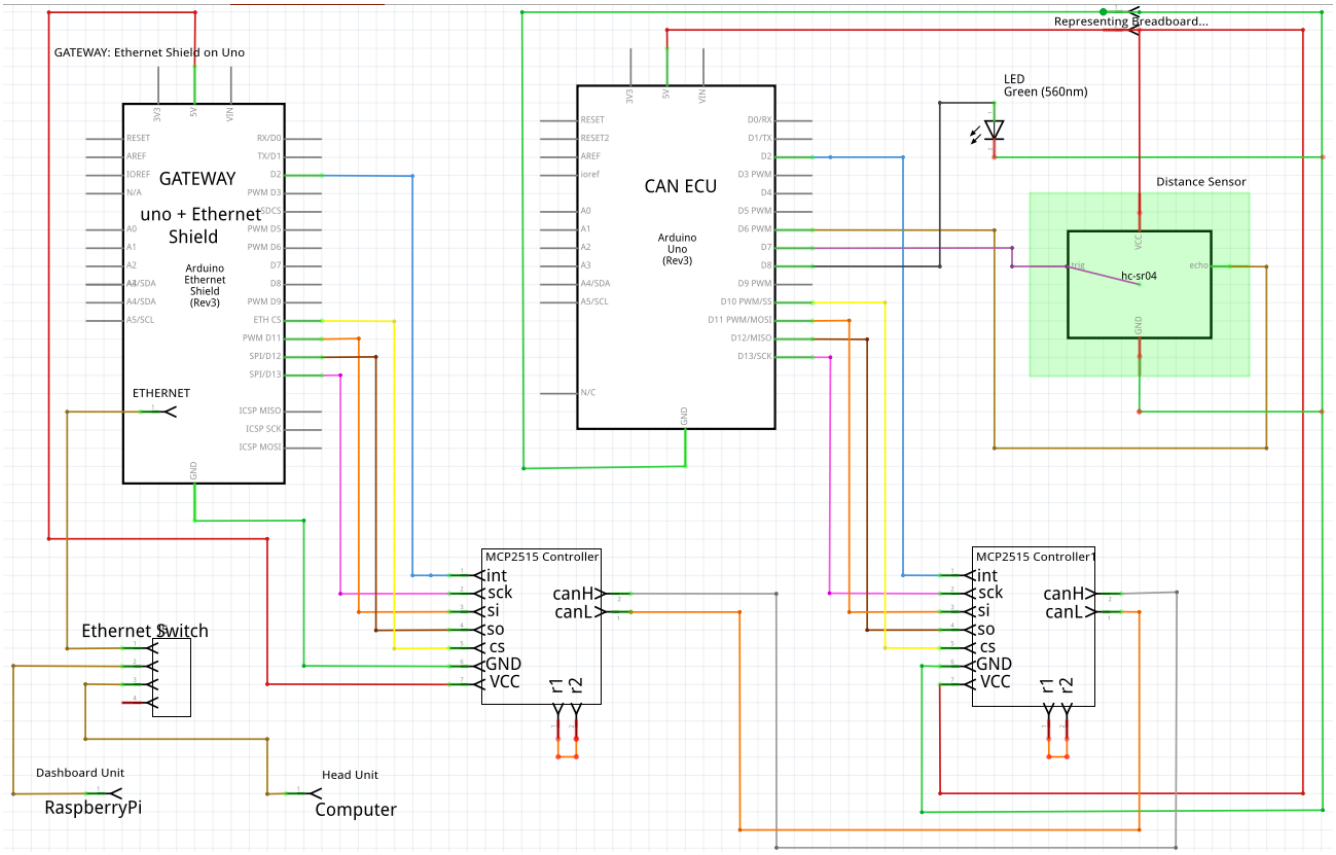
Steven Rossi
netid: sbrossi2
team: Butterscotchers
Internet of Things Lab 1: IoT Devices

Part 1. Design Topology



(Please also find a zoomed-in version submitted with this report)

Part 2. Discussion of Design Considerations.

Advanced Driver-Assistance System (Raspberry Pi)

It was easy and fun to setup the raspberry pi with a 5mega-pixel camera, and begin processing images with openCV. Classification was made easy with tensorflow's object detection api, which was easy to download and easy to use with examples.

At this stage, I experimented with a few alternative design considerations. First, I found openCV to be a great package, but somewhat heavy-weight in terms of installation and configuration; and I hoped that I could process images with picamera, numpy and tensor flow alone. This proved to be true! However, openCV was the most effective way to draw labels and coordinate boxes onto images, and then to display those images. OpenCV had to stay!

Next, I wanted to explore other options for tensor-flow models, specifically the latest quantized TF Lite models which had been shown to work with raspberry pi 3b+. I excitedly drove ahead and got them working relatively easily! They were extremely fast and easy to execute! But, in my excitement, I had pushed forward too quickly-- none of the released quantized models could do more than 1 object-detection per image, and none provided the coordinates for the detection! That is a requirement of our lab!

This left me with two options – revert back to non-quantized models; or to attempt to use tensorflow to convert an existing model to a quantized model. The latter is an involved and challenging task – but one that I would be excited to try in the future. I chose the former, and reverted back to a regular tensor flow object detection model from their api package.

I found many of their models to demonstrate impressive accuracy in detections. And most could successfully detect several objects. I ultimately chose the ssdlite_mobilenet as a standout model in terms of speed and accuracy.

Part 2: Microcontroller

In programming the Arduino Uno's, I made several choices in implementation. I used the mcp_can library to setup my CAN bus with the arduino units. I ultimately landed on sending 8-byte packets for the distance measurements. This provided the needed flexibility and ample speed.

The distance sensor was easy to setup, and to program with the Arduiono using basic, publicly available code for those devices. My calculation of the distance is discussed in detail in the "Questions & Discussion" section.

I found the distance calculation to be one of the more challenging aspects of this lab, due to the general inaccuracy of the HC-SR04 ultrasonic sensor. But, what is loses in accuracy it makes up for in speed! So, I chose to use a loop-variable to average over a number of successful distance readings. After experimentation, I ended up using an average over 100 readings – more than I would have expected – but this resulted in the best accuracy / speed trade off for my deployment. Ultimately, this is a variable that can be easily modified for difference uses and needs.

Additionally, it was easy to program the pin setting to ensure the brake-light operated as expected, given instructions from my CAN network.

Part 3: CAN to IP Gateway.

For my Ethernet gateway, I connected my Ethernet shield to an Arduino Uno as described in the device setup.

I chose to use a UDP connection from my Gateway to my ADAS unit (raspberry pi). The UDP connection is more "lightweight" allowing for faster communications. Speed is the key driver for my decision to use UDP. It also ensures that I am constantly communicating the latest information as quickly as possible (without a need to re-send lost packets). TCP's ability to re-send lost packets as needed is not viewed as a strength here, since those late packets would already be stale measurements. Hence the preference to use UDP for it's speed, and to ensure new measurements are prioritized over

older measurements. This speed is key to ensuring real-time functioning of my object detection system.

Part 4:

My head unit is represented by my computer, and is connected via Ethernet to ADAS (raspberry pi). For this connection, I chose to use a TCP connection as the images that I would be sending are relatively large in size. TCP allowed me to reliably communicate a high image quality. This allowed me to accurately study the performance of the system.

For my head unit, I chose to use python to receive and render the image – the same language that my ADAS used to produce the image. I found that multi-threading in the head-unit python script helped to improve performance of the overall system. So, I used multi-threading to simultaneously receive & render the image. (My home computer has fairly week networking components and a cheap GPU, so isn't ideal for this exercise – but multi-threading really helped!)

From the start of the project, to the end, I was able to increase the frame rate from ~.2 FPS to well-over .5 FPS. That increase in performance was largely attributable to multi-threading in the ADAS as well as on the Head Unit. Looking to further increase speed we must note that image processing and rendering is a key bottle-neck for our system. Therefore GPU acceleration would be a key area for development if speed is desired.

Measurements of end-to-end performance in obstacle detection:

Scenarios:
a) person-from-side. Myself enter from side, stand 5ft from device for 5 secs. (+ Detection)
b) halloween-from-side. A halloween decoration flies in from same view as in (a). (- Detection)
c) child-from-front. My son enters view from hallway ~ 15ft away. (+ Detection)
d) halloween-from-front. My son enters room with dinosaur costumer. (? Detection…)

Please see my extra video submission: "halloween extra" for one fun trial on my (d) scenario!

Performance Measurements:

Scenario	Trials	+ Detections	- Detections	Correct Distance	Correct Object	Frames per Second
(a)	10	9	1	90%	100%	0.65
(b)	10	0	10	70%	100%	0.6
(c)	5	0	5	100%	100%	0.58
(d)	5	4	1	70%	? *	0.56

* It is unclear whether the ADAS should detect a dinosaur or a kid in this scenario.

General notes were that the distance reading had some difficulty when objects were at 150cm, which was the distance I chose to implement the breaklight.

The other big challenge I found was that the distance sensor needed to be properly aimed at the object we wanted measurements for. Hence the low accuracy rates on this dimension of the project. If things weren't aligned, the sensor would read past objects. It would be ideal to be able to point the sensor towards objects of interest; or to have many sensors all able to detect independent directions.

Additional Questions:

Why are quantized models better for resource constrained devices?
"Quantized" models are deep learning models that are represented using lower-than-normal "bit-widths". For example, many Quantized learning models use INT8 data-types to store tensors (rather than, for example, floating point, 32 bit types). In that given example (INT8 type vs FLOAT32), the model is 4x smaller in memory! This is particularly important for resource-constrained devices, which (for a variety of reasons) are limited in their available memory and memory bandwidth. In those cases, quantized models are often a great choice because they represent a compact version of state-of-the-art models. Without quantization, many of these models would simply be too large to load and execute on resource constrained devices.

Quantization can also improve execution speed, because of the lower memory bandwidth required by the smaller representations.

Quantization can be coupled with custom hardware that can offer further benefits by greatly decreasing execution time. This hardware can be built for high performance operations that maximize the low bandwidth, quantized representations.

It is common for quantized models to be ¼ the size of comparable models, and, execution speed can be up to 2-4x faster!

There is no free lunch! Most available packages don't allow for Quantized models to be trained – only forward passes can be executed; and only pre-existing models can be converted to quantized models. There is an obvious loss of accuracy, which can be significant, and should be evaluated. In some cases, post-quantization re-training may be required (if possible) due to the loss of accuracy from the "quantization" from a higher bit representation to the lower bit representation.

The Best Practices I found:
Ultimately, the best practices that I used to achieve strong system performance are:

- UDP communication from CAN (through gateway) to ADAS
- Use of measurement aggregation (averaging) over very short periods on distance sensor.
- Multi-Threading of all processes on ADAS (image processing, object detection, receiving from CAN, sending to CAN and sending to Head Unit)
- Multi-Threading receiving & image rendering on the Head Unit
- tensorflow ssdlite_mobilenet for use for inference. While not quantize, this did provide the necessary multi-object detections and object labeling.

Would hardware acceleration help? Have the packages mentioned above leveraged it? If not, how could you properly leverage hardware acceleration?

In this implementation, neither image processing nor our object detection uses hardware acceleration.

Image processing is done in python by OpenCV. This efficient and powerful software does not utilize the GPU, or any other customized hardware in it's native deployment. This is similar to tensor flow, which does not, by default leverage acceleration. It is very reasonable to believe that this would be a prime area for speed improvement – the model inference and image processing both require a large number of similar operations, and so are prime candidates for custom hardware acceleration.

There is currently an OpenCV working project to leverage hardware acceleration (through the use of the GPU), which claims speed increases of 5x – 100x on select functions! This would be a good first step for further research.

Object Detection in my deployment is done with Tensor flow, without the use of hardware acceleration. But, tensor flow can be configured to use hardware acceleration, through the use of a GPU or with a "TPU" (tensor-processing-unit). I believe using either of these methods could increase the execution speed of our inference model; but both are relatively costly compared to the overall cost of our project. That would make then "runners-up" to exploring openCV's use of raspberry pi's GPU and openCV.

Would Multi Threading Help the performance of your program?
At first, I didn't think multi-threading would significantly aid the performance of my program! I was wrong! My initial thoughts were that image processing and object detection needed to be performed sequentially, and that process was the clear bottle-neck. The other routines (distance detection & communications with breaklight & head-unit) are so fast – so I didn't think multi-threading would add much.

I was wrong- multi-threading provided benefits on several levels. First, there were clear improvements to execution speed, against my initial expectations. Multi-threading almost doubled the FPS my program could execute: from ~.35 FPS to ~.7FPS! Processing all the communications on the raspberry pi was very fast, but still benefited from multi-threading. And, while the image processing & object detection needed to be done sequentially, they can still be run in parallel, and indeed benefit from being run concurrently.

The other improvement to multi-threading was that the overall system was more responsive to changes – that's because the deployment concurrently updated all data-inputs and communicated and acted on only the latest data.

How would you choose the trade-off between frame rate and detection accuracy?

This trade off should be taken in the context of the finished project. For a self driving vehicle on public streets, we might want a very high accuracy, but for a military vehicle in battle we might wish for higher speed.

I aspired to deliver both – to achieve nearly 1 FPS rate and 100% accuracy. And I was able to do that.

The object detection models I explored via tensor flow's object detection API were all impressively accurate! From early in the project, I felt I could use any of them and achieve my desired accuracy (at least 95% +). So, with several options that exceeded my desired accuracy, most subsequent design decisions were based on other considerations, especially speed. Also important were execution stability (low overall memory requirements), the number of objects that could be detected in a single image, and a requirement to output the box-coordinates of detected objects.

Ultimately, ssdlite_mobilenet provided all the “nice to have’s” (multiple objects, low memory requirements) and the must have’s (speed, accuracy and coordinate labeling). So it was the preferred model in my implementation.

Distance Calculation

I found the distance calculation to be one of the more challenging aspects of this lab, due to the general inaccuracy of the HC-SR04 ultrasonic sensor. But, what is loses in accuracy it makes up for in speed! So, I chose to use a loop-variable to average over a number of successful distance readings. After experimentation, I ended up using an average over 100 readings – more than I would have expected – but this resulted in the best accuracy / speed trade off for my deployment. Ultimately, this is a variable that can be easily modified for difference uses and needs.

With my average sensor reading, which I call duration, I have a fairly accurate reading on the time it took for sound to travel from my speaker, to some object, and bounce and return to the receiver. This duration needs to be converted to distance by multiplying it by the speed of sound through my environment. Based on my location, I determined that to be 34,326 cm / sec; I call this speed_of_sound_in_cm

From here, I have an estimate of the trip distance from my speaker, to some object, back to my receiver. The speaker and receiver are 2.5 cm apart, and I wish to calculate the distance from the mid-point of the speaker & receiver (my sensor) to the object. I triangulate using the Pythagorean theorem, and calculated the distance as: ((duration * speed_of_sound_in_cm)^2 – (1.25^2))^ (½)

Will you use UDP or TCP to communicate between Gateway and ADAS?

(copied from above)
I chose to use a UDP connection from my Gateway to my ADAS unit (raspberry pi). The UDP connection is more “lightweight” allowing for faster communications. Speed is the key driver for my decision to use UDP. It also ensures that I am constantly communicating the latest information as quickly as possible (without a need to re-send lost packets). TCP’s protocol to re-send lost packets to ensure in-order-delivery is not viewed as a strength here, since re-sending late packets would only waste time to distribute stale measurements. Hence we use UDP for it’s speed, and to ensure new measurements are prioritized over older measurements.

Can you realize full-duplex communication between the CAN ECU and ADAS?

Yes, we achieve full-duplex communication between the CAN ECU and ADAS because we have parallel lines of communication from the ECU to ADAS and from the ADAS to ECU. Using the Serial Peripheral Interface for communicating through the CAN network enables this simultaneous, bi-directional communication. For example, when the ADAS concludes that it needs to send and update to the ECU, it is free to send that message as soon as it can, with a free line of communication through the gateway into the CAN network. Similarly, as the ECU receives sensor readings and processes them into distance readings, it sends those readings freely as messages along the CAN bus to the ADAS. With simultaneous, bi-directional communication, we have achieved full-duplex

Head Unit – Can you make the time interval smaller? Also, you should mention how sending data over the network influences the Pi's performance.

From the start of the project, to the end, I was able to increase the frame rate from ~.2 FPS to well-over .5 FPS. That increase in performance was largely attributable to multi-threading in the ADAS as well as on the Head Unit. Additional aspects, such as the use of UDP and the selection of the ssdlite_mobilenet object detection model also added performance gains.

Sending data over the network, especially images which can result in large data loads, it quite costly for the raspberry pi. It does weigh down resources. And, while my hope computer has quite a powerful core, it's network components are certainly sub-standard, and this was a drain on system speed. But I do not believe these networking constraints were the limiting factor for the overall system. Based on my readings of system performance, image processing and rendering is the primary bottle-neck with the system as it stand today. Therefore GPU acceleration would be a key area for development if speed is desired.

Bottom line: Sending data over the network does weight on the Pi's performance, but is not the limiting factor for speed for this system.

How should we interconnect the 3 Ethernet Devices (pi ADAS, CAN Gateway and Head Unit)?

To connect these devices via Ethernet, I coordinated all of the sending and receiving ip addresses. After properly configuring those addresses, I hard-coded them into the procedures of each device. For my Gateway and Raspberry pi ADAS, I chose to use the subnet assignments 192.168.86.247 / 248 for convenience. My head unit has been around a while, and already has an assigned ip, which I coded into the procedure of the ADAS. This simple, straightforward implementation works very effectively.

Great lab!

All components work as expects!

My Github repository is available at:

https://github.com/sross444/fa20-cs498it-lab1/

netid: Sbrossi2

All contribution are my own.