CS179I: Senior Design in Networks

Final Report

Aaron Chen
Marco Baez Pulido

Abstract

We are building the basis for a serverless parking structure. This parking structure would be containerized and be able to be completely self-sufficient. This self-sufficient parking structure would eliminate nearly all human costs that would come from a traditional parking structure.

Introduction

Parking structures have, for the most part, stayed the same since their inception. A person works at the parking garage and charges the person as they leave the garage. However, as machine learning has advanced, the opportunity to create a self-sufficient parking garage has risen. Specifically, a serverless parking structure would be optimal to handle each of the necessary tasks independently. Ideally the solution would be to have a docker container for each task required such as client interaction, machine learning processing, payment processing and database access. This would allow for an efficient and reliable process to be able manage all aspects of a parking structure. The end user would then just have to sign up upon the first time entering the structure but after that the structure would automatically charge the customer upon their departure. To begin this process we have designed and implemented a client and server system that enables the client node to send a picture and receive the output from the machine learning algorithm that is run on the server docker container.

Related Works

Our mentor, Shixiong Qi, wrote several scripts to install Kubernetes and Knative onto our CloudLab experiments. His repository can be found here:
https://github.com/ShixiongQi/serverless-IoT-script/tree/nas21-motion_gen
We used this script every time we needed to restart our experiments due to them expiring. Or when they encountered problems with installation (which was fairly often in the beginning of this project).

We also are building off of the work of the CNR Parking Database (http://cnrpark.it/). We are using the pretrained AlexNet models for our project, and the model detects whether a parking spot is occupied or empty. We are using this model in the node we are using as our server. It reports back a result based on the image that we send it.

In its current form, our project resembles several implementations of parking structures that report how many parking spaces are open. A quick google search yields this commercially available product by Hanwha Techwin.
(https://www.hanwha-security.eu/hanwha-techwin-launches-serverless-wisenet-parking-guidance-solution/) While not serverless in terms of networking but serverless in “not needing a server to run,” it still runs machine learning software to recognize whether there are parking spaces open.

Another search also yields this GitHub repo (https://github.com/simeiko/aws-serverless-parking-lot-api) which would have been similar to our original spec, since it has functions for payment and tickets. However, it does not seem to use any form of image recognition, which is different from our implementation.

Design

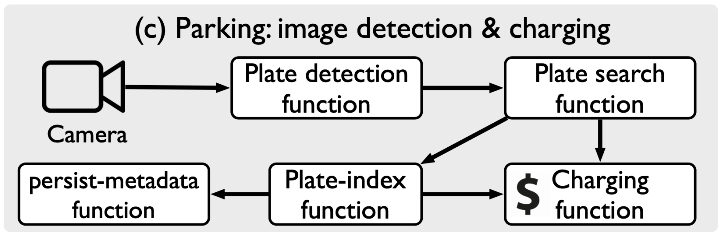
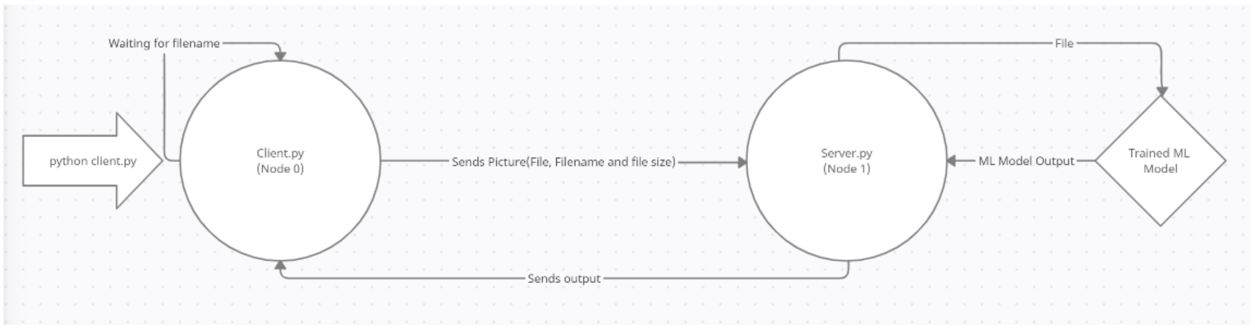


Figure 7: Serverless function chains setup

Above was our original design for our project. We were unable to finish what we originally set out to do since we realized really late into the project that the model that we spent 9 weeks trying to get running was not actually the model we needed to read license plates. Instead, the AlexNet model from the CNR Parking Database is made to return whether or not a parking space is occupied or empty. While this is useful in parking structures, it isn't the functionality we thought it was. With less than one week left in the quarter, all we could do was run some analysis on what we did have working.



Our current design is above. We have a server/client pair that can communicate with each other. The server is only called when a client sends a JPG file to the server.

Our design would perform the best when the latency between the server and the client is very low. The best case scenario would be if the server had a wired connection to the client for the lowest latency. However, this would probably not be how real-world applications would pan out, since the point is to host the trained AlexNet model on a serverless cloud provider, like AWS Lambda. In terms of real-life, an ethernet connection to the internet for each client would be the most optimal.

The worst case scenario would be a weak or spotty WiFi connection between the client and their access points, which is highly likely in WiFi-based concrete or underground parking structures. This would increase latency and possible packet loss between the server and client.

We also believe that running the pretrained model on a GPU instead of on the CPU would run much faster, but we were only able to test it with however the CloudLab nodes are set up. Since we do not know what hardware the CloudLab nodes run on, and we are unable to access the hardware to change it, we are unable to test that assumption. This does reflect how serverless functions are in the consumer space, since the customer is unable to modify what hardware the cloud service provider runs their code on.

We, with the help of Shixiong, set up five milestones for us to accomplish throughout the project. First, we were to learn how to deploy Knative on Cloudlab which would ensure that we got familiar with starting to build serverless functions on Cloudlabs infrastructure. Second, we were to get familiar with the parking workload. Specifically, to examine what we needed to accomplish this plan and how many nodes we needed to do so. Next, we were to get familiar with different image detection algorithms as being able to run these algorithms would be a pivotal part in accomplishing our goal. Fourth, we were to learn how to use Python to build Knative functions that would allow everything to be run within Knative's serverless platform. Finally, After implementing all necessary functions we would deploy in Knative and do some measurements to test the time it would take to send files back and forth but also to see the time to be able to run images through a pre-trained AlexNet model.

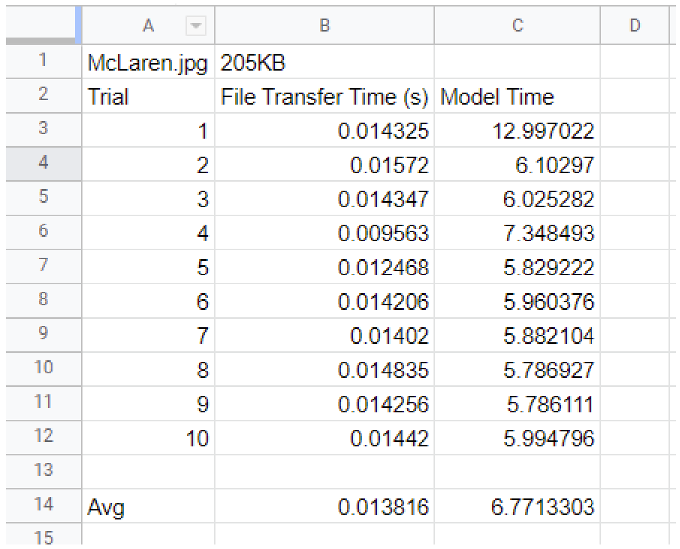
Implementation

Our implementation is a client and server based machine learning model. The client, after running client.py, is prompted to enter a picture name and sends that picture onto the server, the server then receives the file, prepares it, and runs a pre-trained machine learning algorithm on the picture. Once the machine learning algorithm is complete the server will print out the results onto the console and also will send the results to the client for them to view. Our code is pretty simple and is merely a server.py that runs on the server node within a docker container that runs caffe, the necessary framework for machine learning. The server.py stays on indefinitely waiting for a picture. Once a connection has been established with the client the server accepts the connection and begins to receive three things. The file, the filename, and the filesize. As the files are being received the progress bar is updated to accurately estimate the completion time. Once all the files have been received the server.py closes the connection with the client and passes in the filename into the caffe image function which then begins to prepare the image. Once all the preparation is done the pre-trained machine learning algorithms are called on the image. After a bit of a wait the algorithms finish and output the result to the console. At this point the server.py begins another connection to the client.py but this time sends the output to the client. Once the text is successfully sent the server once again closes the connection.

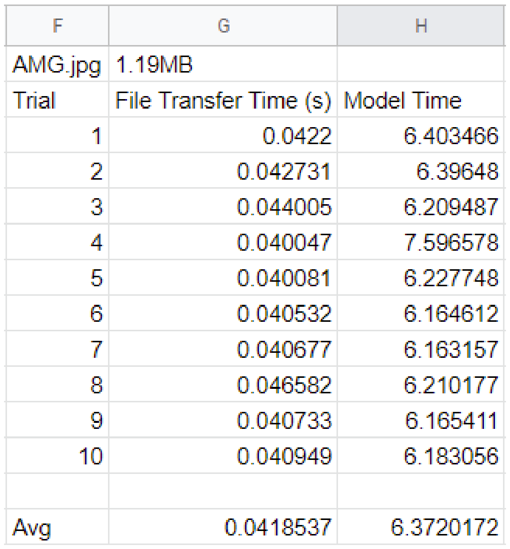
Throughout this project we used a lot of preexisting open source code which we modified to work within our implementation. First we used this Caffe dockerfile that contains all the necessary dependencies to run the caffe framework. We modified this by adding an exposed port and fixing some dependencies. Building this dockerfile into a docker container allowed us to run the pre-trained machine learning algorithms found here. For the client and server python files we used this as a general template but also used examples provided by Aditya and Shixiong to be able to not only receive but also send files back to the client. Shixiong also provided a general template for how to run the pre-trained models within Caffe which we modified to work within our server python file.

Testing and Results

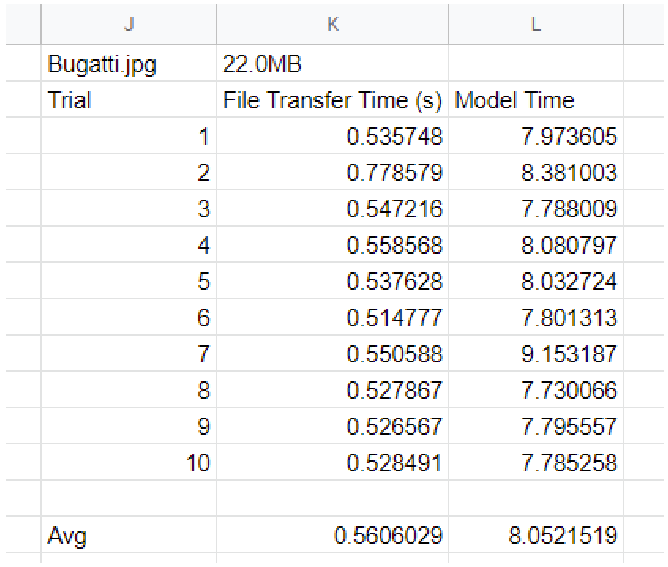
We were not able to completely accomplish what we originally set out to do. However, these are some analyses we did based on what were did manage to accomplish. We ran time trials to see how long the function would take to process the image sent to it by the client.



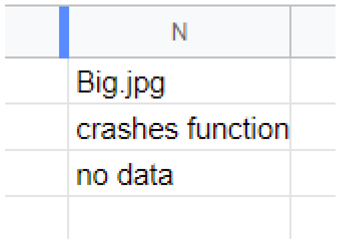
	A	B	C	D
1	McLaren.jpg	205KB		
2	Trial	File Transfer Time (s)	Model Time	
3	1	0.014325	12.997022	
4	2	0.01572	6.10297	
5	3	0.014347	6.025282	
6	4	0.009563	7.348493	
7	5	0.012468	5.829222	
8	6	0.014206	5.960376	
9	7	0.01402	5.882104	
10	8	0.014835	5.786927	
11	9	0.014256	5.786111	
12	10	0.01442	5.994796	
13				
14	Avg	0.013816	6.7713303	
15				



F	G	H
AMG.jpg	1.19MB	
Trial	File Transfer Time (s)	Model Time
1	0.0422	6.403466
2	0.042731	6.39648
3	0.044005	6.209487
4	0.040047	7.596578
5	0.040081	6.227748
6	0.040532	6.164612
7	0.040677	6.163157
8	0.046582	6.210177
9	0.040733	6.165411
10	0.040949	6.183056
Avg	0.0418537	6.3720172

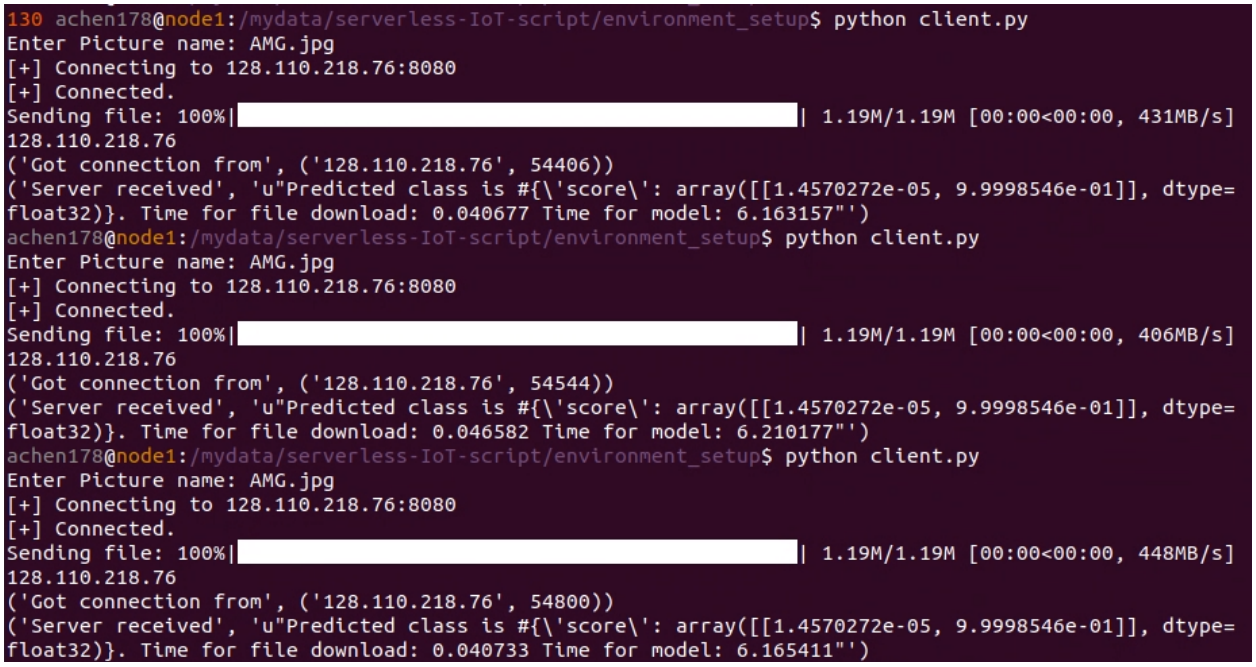


J	K	L
Bugatti.jpg	22.0MB	
Trial	File Transfer Time (s)	Model Time
1	0.535748	7.973605
2	0.778579	8.381003
3	0.547216	7.788009
4	0.558568	8.080797
5	0.537628	8.032724
6	0.514777	7.801313
7	0.550588	9.153187
8	0.527867	7.730066
9	0.526567	7.795557
10	0.528491	7.785258
Avg	0.5606029	8.0521519



N
Big.jpg
crashes function
no data

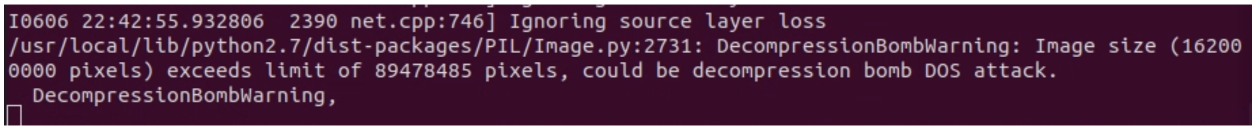
We ran multiple trials where we submitted the same file, with 4 different files. We sent each file from the client to the server 10 times and recorded the time it took to transfer the file as well as how long it took to process the image through the pretrained model.



```
130 achen178@node1:/mydata/serverless-IoT-script/environment_setup$ python client.py
Enter Picture name: AMG.jpg
[+] Connecting to 128.110.218.76:8080
[+] Connected.
Sending file: 100%|██████████████████████████████████████████████| 1.19M/1.19M [00:00<00:00, 431MB/s]
128.110.218.76
('Got connection from', ('128.110.218.76', 54406))
('Server received', 'u"Predicted class is #{\'score\': array([[1.4570272e-05, 9.9998546e-01]], dtype=
float32)}. Time for file download: 0.040677 Time for model: 6.163157"')
achen178@node1:/mydata/serverless-IoT-script/environment_setup$ python client.py
Enter Picture name: AMG.jpg
[+] Connecting to 128.110.218.76:8080
[+] Connected.
Sending file: 100%|██████████████████████████████████████████████| 1.19M/1.19M [00:00<00:00, 406MB/s]
128.110.218.76
('Got connection from', ('128.110.218.76', 54544))
('Server received', 'u"Predicted class is #{\'score\': array([[1.4570272e-05, 9.9998546e-01]], dtype=
float32)}. Time for file download: 0.046582 Time for model: 6.210177"')
achen178@node1:/mydata/serverless-IoT-script/environment_setup$ python client.py
Enter Picture name: AMG.jpg
[+] Connecting to 128.110.218.76:8080
[+] Connected.
Sending file: 100%|██████████████████████████████████████████████| 1.19M/1.19M [00:00<00:00, 448MB/s]
128.110.218.76
('Got connection from', ('128.110.218.76', 54800))
('Server received', 'u"Predicted class is #{\'score\': array([[1.4570272e-05, 9.9998546e-01]], dtype=
float32)}. Time for file download: 0.040733 Time for model: 6.165411"')
```

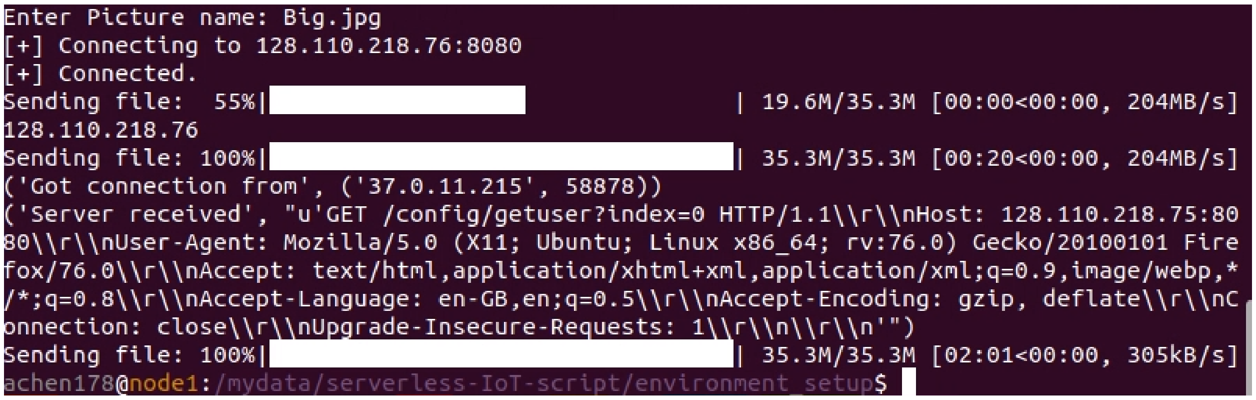
However, in the case of a rather large image from NASA (37MB), we encountered this error.

server.py



```
I0606 22:42:55.932806  2390 net.cpp:746] Ignoring source layer loss
/usr/local/lib/python2.7/dist-packages/PIL/Image.py:2731: DecompressionBombWarning: Image size (16200
0000 pixels) exceeds limit of 89478485 pixels, could be decompression bomb DOS attack.
  DecompressionBombWarning,
```

client.py



```
Enter Picture name: Big.jpg
[+] Connecting to 128.110.218.76:8080
[+] Connected.
Sending file:  55%|█████████████████                | 19.6M/35.3M [00:00<00:00, 204MB/s]
128.110.218.76
Sending file: 100%|██████████████████████████████| 35.3M/35.3M [00:20<00:00, 204MB/s]
('Got connection from', ('37.0.11.215', 58878))
('Server received', "u'GET /config/getuser?index=0 HTTP/1.1\\r\\nHost: 128.110.218.75:80
80\\r\\nUser-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:76.0) Gecko/20100101 Fire
fox/76.0\\r\\nAccept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*
/*;q=0.8\\r\\nAccept-Language: en-GB,en;q=0.5\\r\\nAccept-Encoding: gzip, deflate\\r\\nC
onnection: close\\r\\nUpgrade-Insecure-Requests: 1\\r\\n\\r\\n'")
Sending file: 100%|██████████████████████████████| 35.3M/35.3M [02:01<00:00, 305kB/s]
achen178@node1:/mydata/serverless-IoT-script/environment_setup$
```

There appears to be an upper limit to how large the image can be before the model rejects it outright. It thinks that the image that was sent was a DOS attack.

Lessons Learned

We cannot overestimate how much was learned throughout this process. We can confidently say that we now much better understand how Docker containers work and are very comfortable with creating and modifying Dockerfiles. Although we both have had plenty of experience with Linux, this project forced us to become extremely comfortable within the Linux command line. Finally we spent a ton of time researching to understand sockets and how they work to be able to successfully implement a client and server to send and receive files back and forth efficiently.

We were unable to complete our senior design project to our original spec because we ran into many roadblocks along the way. Our biggest obstacle was that we spent a large amount of time installing software, reinstalling software, and encountering many issues attempting to train the machine learning algorithm which resulted in countless hours of debugging. We worked on this up till week 8. After much deliberation with our Mentors, we were instead advised to use the pre-trained models. Once this was decided the next obstacle was that we both had very little experience with Docker and had to learn how to modify a pre-existing Dockerfile to fit our needs. The last obstacle was researching how to create client and server python files to use sockets to send back and forth files. We wrote our server/client programs in week 9, where we then realized that our model was actually not suited for our original spec. The pretrained model actually only processes if the image sent to it has a parking space that is occupied or not. However, we decided to carry on and present what we had. We did not communicate with Shixiong, Aditya, or Dr. Ramakrishnan that we had encountered this issue until the presentation. It was there that we were told that we could have reached out and shifted the focus of our project to be around what the AlexNet model could do instead of trying and failing to live up to the original spec.

This quarter, we could have definitely benefited from reaching out to our mentors more so they would have more information to guide us with. We worked mostly separately from them, only meeting with Shixiong once a week, and so if we encountered any bottlenecks throughout the week, we would remain stuck on it until we met with him. If we'd reached out as soon as we encountered issues, perhaps we could have accomplished a lot more. We wasted a lot of time with installation issues, which were easily solved once we reached out to Shixiong and Aditya. If we had been more proactive on this project and getting it working, we could have found out that the model was for something else a lot earlier, and been able to accomplish more with the time we were given. We wish that there was more time so we could continue our work on this project.

Appendix

All of our source code including the Dockerfile and client and server python files as well as extremely specific instructions on how to run everything is listed on Marco's Github repo here.