CS 5412 Final Report: Smart Air Traffic Control

Rohit Khanwani (rk632), Harshwardhan Jain (hj364), Sanjana Kaundinya (ssk228)

May 15th, 2018

1 Overview

We implemented an application for airports and airlines that uses flight schedules and real time data from airborne aircrafts that offers real time collision detection and gives routes based on this service, to prevent aircrafts in the air from crashing into one another. Our architecture is built in such a way that in the future, we can add more microservices for things such as fuel optimization and weather detection. We foresee our clients to be the sensors that communicate with the aircrafts and signaling how to go on with their route.

2 Data

We visualize our system, on a higher level, dealing with two kinds of data:

- **Static Data**: Data that doesn't change much, for example flight numbers, airports, possible routes and long-term flight schedules.
- **Dynamic Data**: Data that flows in through the sensors in real-time, for example aircraft altitude, location and weather updates.

We researched free APIs that could provide us with the data we needed to test our application. Using the Bureau of Transportation Statistics' Transtats portal, we were able to obtain most of the static data that we require. This information included flight information as well as flight schedules. In addition, we used an online flight simulator to obtain information for route data. These two components make up our static data for the project.

For our dynamic data, we wrote scripts to generate dynamic real-life mimicking data. These scripts mimic the potential live data coming in through sensors on a real time basis. This data has components such as latitudes, longitudes, weather, speed, altitude, and other properties to simulate live data being fed into the application.

For storing all this data, we chose Google Cloud's Datastore system, for a variety of reasons, but mainly because it was 1) a non-relational database, allowing for a highly scalable and high performance application as well as 2) its high compatibility with our App Engine based application on Google Cloud; there were many readily available APIs that allowed for easy communication between our application architecture and Datastore.

3 Interface

We structured the interface to our application through a multi-tiered architecture with microservices to help with the features of our application. The specifics and implementation of our architecture will be discussed in the next section. In this section, we will discuss the interactions between the layers and how it interfaces with the rest of the application.

First, the sensors on the aircrafts send requests with data such as latitude, longitude, weather, and speed to our system. This first request interacts with the first tier of our application. The specific method that deals with this request is called `incoming_flight_data()`. From here the requests are filtered through by various other methods in this tier and put into Datastore. Once into Datastore, the second tier accesses this data, and with its various microservices does the necessary computations (such as collision detections and waypoint generation), and stores these values into Datastore, as well as the shared memcache in the system. From here, the first tier reads the value from the memcache (if it's a cache miss, it reads from Datastore), and from there sends back updates to the sensors to give back the appropriate data and information needed by the aircrafts.

Additionally, we define a consistency constraint over our application, such that no flight receives updates that are more than one minute old. To maintain this constraint, we expose an endpoint in tier 2 that allows for updates to be requested for specific flights. The function that corresponds to this is `specific_waypoint_update()` Here is a rough idea of what the query and response look like:

3.1 Input

```
{[flight number], [location], [altitude], [speed], [weather conditions]}
```

3.2 Output

```
{[next optimal waypoint], [next optimal speed], [next optimal altitude]}
```

In addition to these services, we have structured our architecture in such a way that several other features can be added to enhance the overall application, without compromising the performance and efficiency of the application. The idea is to use several microservices working in parallel to each other, and a central master service responsible for asynchronously requesting for, and collating their, responses. Some examples of such services would be a fuel optimization service, and a weather check service.

4 Implementation/Architecture

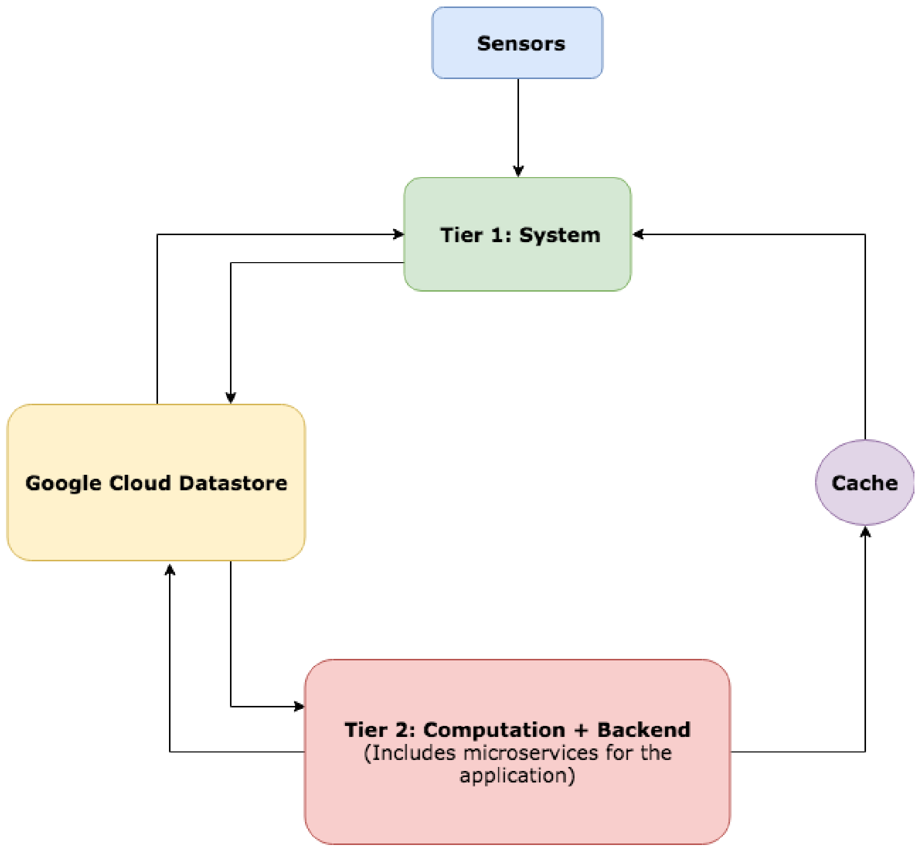
Our choice of cloud services was App Engine from Google Cloud. We chose this because of it automatic scaling and efficient load balancing, as well as its ease to use with the non relational database also provided by Google Cloud called Datastore. We implemented our system in a multi-tiered architecture that has 2 main tiers, as well as a storage database. In addition to these, there also exist the senors (which in our case will be mocked by our scripts for data generation) that are part of our overall infrastructure. The following is a description of each part of our architecture followed by diagram to further illustrate our overall system:

- **Sensors**: The senors on the aircrafts will be sending real time data to our system, with data such as temperature, latitude, longitude, speed, etc. This information is then used to fulfill the requests of the flights in a timely manner, sending back crucial information regarding the next appropriate waypoint and collision detection information back to the aircrafts.

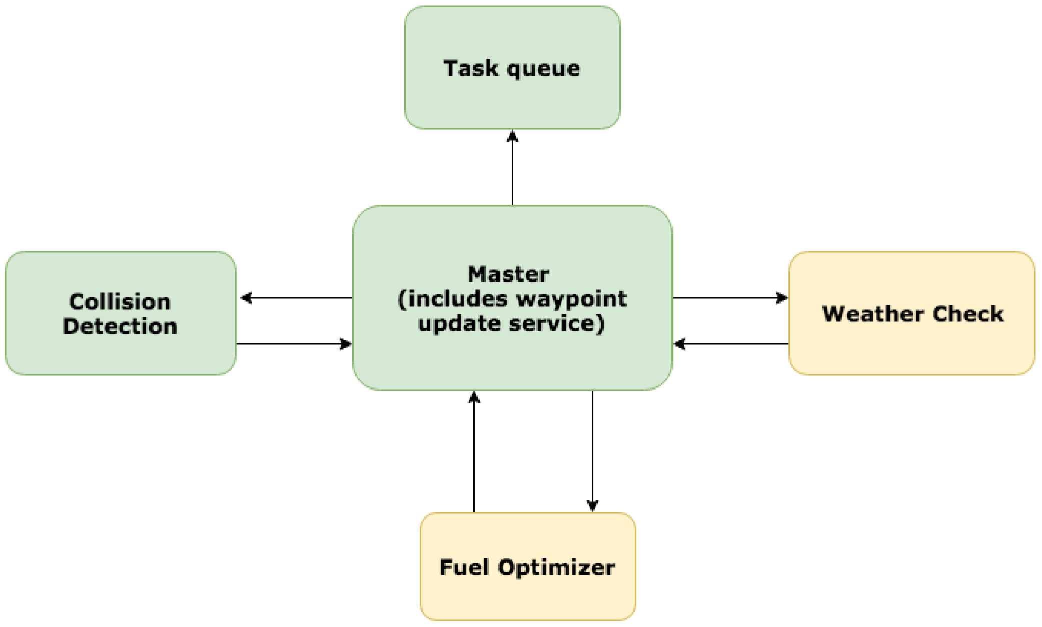
- **Tier 1**: The sensors will send their requests to this tier. This tier will then filter out the unnecessary data and store the necessary information regarding the flights into the Datastore. It also sends back information requested by the flights back to the senors, either by reading from the memcache, or if it's not there, reading from Datastore.

- **Tier 2**: This tier contains the entirety of the backend of our application. Here, data is read from Datastore, appropriate computations are made with the help of microservices (more information regarding this later in this section), and stored back into Datastore and written to the cache (in the Google Cloud Platform, this is the memcache).

- **Google Cloud Datastore**: This is the storage unit for our application. All data (static and dynamic) resides on this layer. Both the first and second tier store their data into this layer, and they both read the data they need from this layer. In the future, for enacting machine learning models, this layer would help to have the data easily accessible to train our models on.


Sensors
Tier 1: System
Google Cloud Datastore
Cache
Tier 2: Computation + Backend
(Includes microservices for the application)

(a) Overview



(b) Tier 2

Figure 1: Application Architecture

- **Cache**: This is part of the shared memcache that we get as a part of using the Google Cloud Platform. In here is where we cache all the data outputted from Tier 2. Tier 2 is the only layer that writes to this component, and Tier 1 is the only layer that reads from this component.

As it is seen in Figure 1(b), we built Tier 2 to handle a variety of different microservices. In the diagram, the boxes shaded in green are those that we have completed and those in yellow are those that are ideas to further build upon this project. The overall task queue exists to make sure that tier 2 is constantly running. The idea is to add to the queue the HTTP request that invokes the main method every time a single execution is done. The master box is the centralized unit that takes in the outputs from all the microservices and outputs the most optimized response for the request received. Currently we have the collision detection microservice fully implemented and functional, but services such as fuel optimization and weather checking can also be incorporated to produce a response that considers all these options to give the best response for incoming flight requests.

Regarding our overall system, we made a couple of decisions to enhance overall performance and scalability. First, we made sure all calls to and from Datastore are asynchronous. This is very because requiring synchronous calls to and from the database makes it very hard to scale the application up, because it takes quite a bit of time to for the RPC calls to return. We were able to properly implement this, thanks to the asynchronous API provided by Datastore for App Engine. Next, we wanted to make sure our requests were appropriately batched, instead of the system doing real time requests, it's more appropriate and efficient to do it in batches, as we avoid the RPC overhead as much as possible. After doing research into this, we found that Google's ndb library for Datastore takes care of batching requests when scaling up and down the application, in something known as auto-batching. We decided to use this library, as the batching features were dynamic and well suited to our application.

Another crucial thing we needed to make sure that happened was that our system always maintained consistency. For this, we put down a consistency constraint such that any flight requesting information received information calculated on updates no more than one minute. Any time this constraint was violated, we made sure that specific flight requested for a specific flight update, with the tier 2 method `specific_waypoint_update()`. This ensured that our system maintained consistency throughout, with a hyperparameter that could be changed as desired. While the Datastore implements automatic replication for some degree of fault tolerance for us, we also implemented request replication for greater fault tolerance, in order to ensure that requests coming in wouldn't be dropped in the case of a request failing. We implemented this with a task queue that was updated in Tier 1. If the request comes back as a failure, it will still remain on the queue; otherwise it will be deleted off the queue. All of these enhancements and tweaks that we made were made to ensure the easy scalability and performance of the application. This helps build a good infrastructure around the system, as well as allows for the easy addition of microservices in the future, to further enhance the features of our project.

5 Evaluation

For testing, we had a two part strategy. The first part dealt with the overall correctness of the system, as well as ensuring that extreme edge cases would be taken care of in this testing. The second part looked more at the infrastructure we had built around the application as a whole: load balancing, stress testing, scalability, performance with several requests, and more. To do the tests we had a two part method to it, first to write tests on our own and test correctness, and then to stress test it with running several instances on different computers to mimic many flights going through the system.

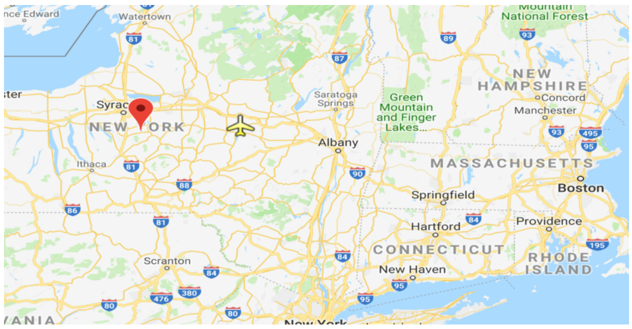
For the first part of testing, we wrote several different test cases, testing for the overall correctness of the system. For this, we generated possible flights and route plans, and tested whether our system would output the correct routes and waypoints, given these inputs. In addition, we wrote edge cases and extreme cases (such as a lot of flights going between the same airports) to check how our code would hold up against this. After a significant amount of testing, we were confident in the correctness of our system and its ability to work as intended.

For the second part of testing, we had to make sure we covered all our bases when checking the overall robustness of the system, when put under several different types of conditions. For this, we ran our program with the live data generator with three different laptops, and was able to run a total of over 700 concurrent flights with requests being serviced at an average latency of 346 ms. To do this testing, we first had one laptop spawn 350 different threads (basically simulating 350 flights requesting information from our system). Then we tried to slowly ramp up the requests and scaling of our system, by adding an additional 200 threads, and then adding another 150 threads. By doing this, we were able to mimic a slow ramping of the system's ability to respond, and the App Engine was able to handle it, with its own algorithms increasing the number of instances running, depending on the number of requests coming in. We were able to service over 2500 requests per minute, and this ran on the App Engine platform with a total of 12 instances (out of which 2 instances were idle, and only there as backup in case the load spiked up). Due to hardware constraints, we weren't able to spawn more threads, but the latency difference between 100 concurrent flights and 700 concurrent flights was around 50 ms, and the difference in the number of instances was 7. This was the case with the cheapest instances available on Google Cloud, and with a 0 flights to 700 flights in a matter of minutes. We are confident that the application would translate well to a much higher load, especially with the higher end instances. Our caching policy also worked out quite well, having over a 90% hit rate in the shared memcache. We expect this rate to be 100% with a dedicated memcache. From this testing, we were able to conclude that our system was able to easily scale up and scale down, and was generally a robust application. For further details regarding our testing, please refer to the screenshots below from the App Engine platform.

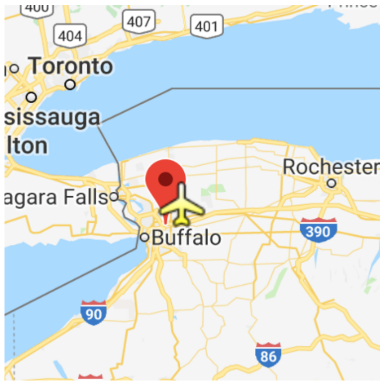
The figure below shows an example of our tests for correctness. The below screenshots show an example flight going from Boston to Buffalo. The red markers are the next waypoint and the flight icon shows where the flight is currently at. Please refer to the following screenshots and code snippets for references to our test for correctness and overall statistics for our system:

NEW HAMPSHIRE
Concord
Manchester
Albany
MASSACHUSETTS
Boston
Springfield
Hartford
Providence
CONNECTICUT
RHODE ISLAND
New Haven

(a) First Waypoint



(b) Second Waypoint



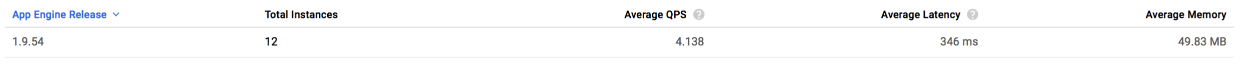
(c) Third Waypoint

Figure 2: Example flight from Boston to Buffalo

The following is a code snippet to show an example of how we ran our tests for correctness:

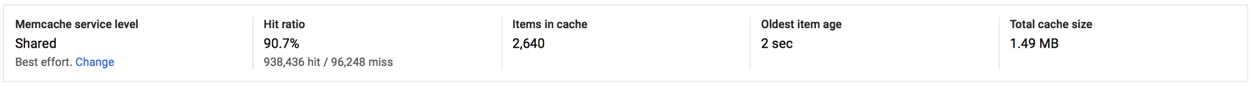
```
URL = "http://localhost:8080/insert_flight_plan"}
route = [[42.36305577000002, -71.00696199],
[42.77981700000001, -73.372475],
[42.74727777999999, -73.80319444],
[42.85334399999999, -75.952197],
[42.871871999999996, -76.443075],
[42.94048674999999, -78.73060625]]
requests.post(URL,
json={'flight_num': "AI973", 'origin': "BOS", 'dest': "BUF",
'dep_date': "2018/06/10", 'arr_date': "2018/06/10", 'dep_time':
"10:30", 'arr_time': "11:30", 'current_route': route, 'carrier': "AI"})
```

Figure 3: Overall Statistics for our System



App Engine Release	Total Instances	Average QPS	Average Latency	Average Memory
1.9.54	12	4.138	346 ms	49.83 MB

Figure 4: Memcache Details

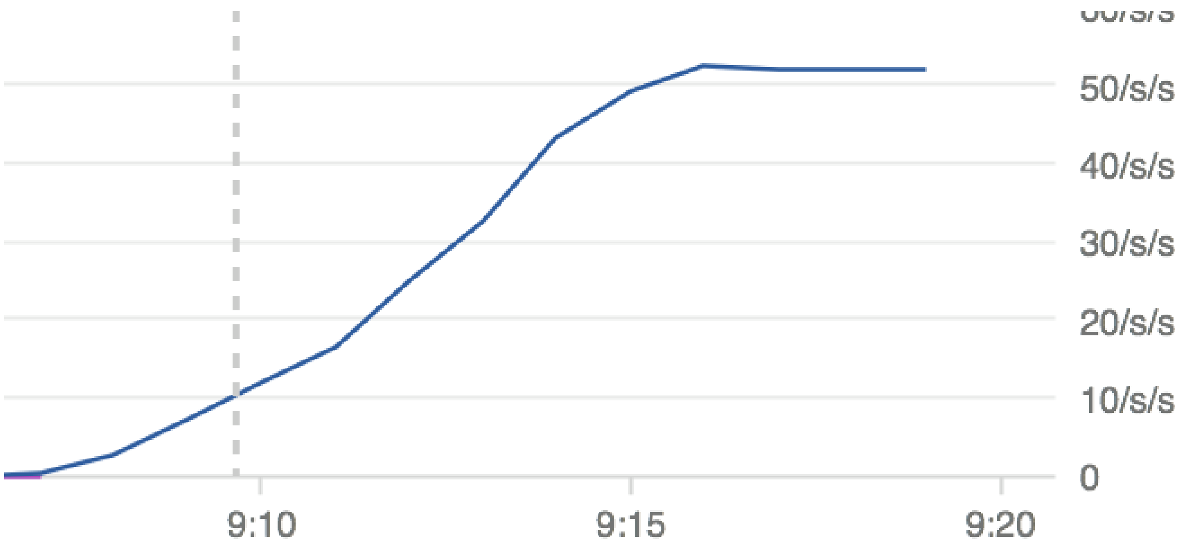


Memcache service level	Hit ratio	Items in cache	Oldest item age	Total cache size
Shared Best effort. Change	90.7% 938,436 hit / 96,248 miss	2,640	2 sec	1.49 MB

Figure 5: Graph of Load Increase

(a) The graph below shows how the load was increased from 0 to over 700 flights in the span of 7 minutes

(b) The horizontal axis is the time lapse, and the vertical axis denotes the rate of increase



6 Milestones and Group Assignments

- March 7th, Sanjana and Harsh: Research appropriate server side languages and start laying foundation of detailed implementation for the tiered system
- March 14th, Rohit and Harsh: Research ways to get all the data needed. If required, look for appropriate ways to construct data points.

- March 18th, Rohit and Sanjana: Look into Google Cloud storage and ML services. Decide on a provider and follow online tutorials by that provider to get a feel of the development environment
- March 21st, Sanjana, Harsh and Rohit: Start planning out architecture and begin the coding process
- March 21st, Harsh and Rohit: Research ML models we can potentially use for our system
- March 28th, Rohit: Start integration of application into App Engine; lay out basic structure of the different tiers and Datastore
- April 4th, Harsh and Rohit: Finalize on an ML model and being implementation
- April 12th, Rohit and Harsh: Integrate ML model into Google Cloud services
- April 15th, Sanjana: Make all calls to and from Datastore completely asynchronous by using API for App Engine
- April 18th, Harsh: Train ML model on data
- April 21st, Rohit and Sanjana: Look into how to batch requests from incoming sensors in Tier 1
- April 30th, Rohit: Make changes to system to do request replication and reach eventual consistency in Tier 1 and 2
- April 30th, Harsh: Write a script to dynamically generate data to test on the application, as well as a live data script
- May 2nd, Harsh: Create testing suite and try to further optimize ML model
- May 2nd, Harsh and Rohit: Start creating a testing suit to rigorously test the Tier 2 and Tier 3 services
- May 5th: Work on other microservices for better feature enhancement of Tier 2
- May 9th, Harsh and Rohit: Rigorously test the system for fault tolerance, race conditions, latency, etc.
- May 9th, Sanjana: Work on report and poster

Our original plan had lots of plans of artificial intelligence involved in the system. However while building our system, we realized it would be better to focus our efforts on building a highly scalable and performance efficient model instead of trying to figure out the right machine learning models to integrate the features we previously had discussed. So instead, we focused on building a system that was highly scalable, and adaptable. We modeled and potential machine learning features as microservices to our system, so that adding these additions would be easy and not a need to rescale the whole application. We still include some of the features we had previously discussed, such as collision detection and optimal route selection. However features such as fuel optimization and weather checking are still in progress, as we wanted to make sure we could build a properly scalable system, before adding in more complicated components.