mXpress: Real Time Vehicle Routing System Experience Report

University of Regina

ENSE 477

Date: April 10th 2020

Ranil Fernando SID: 200261463

Scott Thomas SID: 200361594

Jonathon Florek SID: 200280706

Supervisor: Mohamed El-Darieby

Contents

Abstract

mXpress is a big data framework and real time vehicle routing system that utilizes a variety of cutting-edge technologies in order to provide an adaptable platform that can be used to allow users and customers alike the ability to quickly and accurately create routes for vehicles. This report documents the experience, challenges, and solutions discovered when developing such a platform. This includes a section outlining technology selection and the infrastructure deployment while describing testing and integration challenges faced during project development.

Introduction

The design and implementation of mXpress posed a significant integration challenge. A great deal of time spent during development was determining how technologies worked, what prerequisite requirements were needing to be implemented, and how each technology could be networked and integrated together. This report will outline the challenges, solutions, and future considerations throughout the development of mXpress.

Infrastructure

Design considerations for the backend infrastructure were driven by comparisons of relative performance and cost. Apache Spark requires large amounts of system resources because it utilizes in-memory processing of the input datasets. Many of the server deployment considerations revolved around these memory constraints. The large number of virtual machines required to deploy the mXpress backend also mandated large amounts of processing cores to handle the multi-threaded workload of many of the backend services. Due to these factors, the mXpress backend consists of multiple physical servers totalling 186 cores/372 threads, 1128 TB of RAM, and 40TB of storage. The backend includes enterprise servers from Dell, Oracle, and Cisco. These were purchased second hand outside of the warranty support period after decommissioning from various datacenters. The server infrastructure that comprises the backend of the project has an approximate cost of $2000. The average power consumption of the servers and networking equipment used in mXpress is around 3A on 240V service, averaging a 24/7 load of 720w, or 518.4 kWh/month under load. In Regina, this costs around $71/month of residential electricity. With sandbox/test equipment powered off, the draw is approximately 400w or $39.50/month. In contrast, a similarly performant cloud deployment of mXpress would cost between $2000 and $15000 monthly depending on the service level required.

The infrastructure of the mXpress system is designed to be deployed to any supported hardware. In the current state, every virtual machine and service is deployable with Docker on any host. mXpress also uses Kubernetes to manage the deployment of these containers across several machines. There are small deficiencies in setting up networking between the machines using the data from the City of Toronto. Future development will expand the capabilities of the containerized infrastructure to allow one touch deployment of all the backend services. These improvements will allow simple deployment to a Kubernetes Service/Engine hosted on a cloud service provider like GCP, AWS, or Azure.

Technologies

mXpress uses several open source technologies in the backend architecture to parse input data, perform compute tasks and provide a routing service. Selecting appropriate technologies took quite a bit more time than anticipated at the outset of the project. The initial architecting phase took over 3 months to finalize and changes still arose during development. Many of the selected technologies are on the cutting edge of what is currently available and thus still in active development. Critical technologies such as Apache Spark were especially challenging to develop for due to complexity and a significant lack of documentation. These challenges were eventually overcome through a combination of trial and error as well as piecing together what little documentation could be found online. Outlined below is a brief description of the selected technologies.

Apache Kafka is a distributed stream processing platform. Due to the lack of direct access to live sensor data mXpress uses Kafka to stream legacy sensor data into Apache Spark in order to emulate live data.

Using Apache Kafka allows all networking and external sensor communications to be handled by a single point of entry, abstracting these concerns from Spark.

Apache Spark is the heart of mXpress. Spark is a distributed cluster computing framework that facilitates mXpress' batch and streaming pipeline. Spark allows for extremely large data sets to be processed in memory and across multiple different servers. This allows for processing time to be significantly reduced and makes mXpress scalable across hundreds if not thousands of nodes in a larger application.

HDFS or Hadoop Distributed File System is the underlying file system that Spark utilizes for data storage. It allows for a single set of data to be stored and accessed across multiple nodes.

MongoDB is an intermediary database acting as a buffer between a Spark data stream and the OSRM shell that requires speed updates periodically.

OSRM or Open Source Routing Machine is the routing engine. OSRM takes the road speed data provided by Spark and uses it to calculate the most efficient route possible when queried by leaflet.

Leaflet is a JavaScript library for interactive maps. Leaflet allows for online maps to be quickly and easily built using JavaScript. The leaflet-based interface allows the user to interact with the map and a plugin called leaflet routing machine facilitates routing requests. When a request is made leaflet routing machine queries OSRM for a route and then displays it on the map.

Architecture

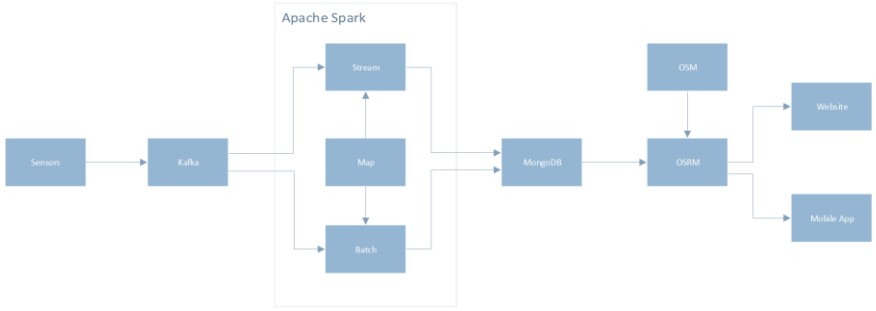

Figure 1: Data Flow

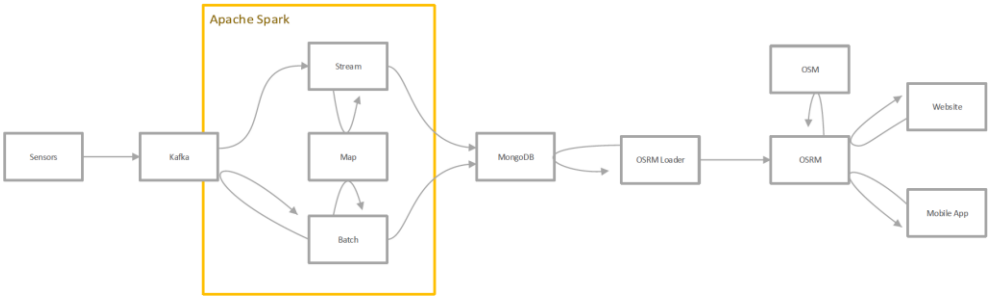

Figure 2: Components

Figure 2 describes how the data flow described in figure 1 in more detail. It clarifies which data flows are push-oriented and which are request/response; it also explains in greater detail how data is fed into OSRM (Open Source Routing Machine) using the OSRM Loader module.

A single Kafka topic for sensor telemetry is the source of truth for sensor data in this system. This Kafka topic contains tuples of UTC timestamps, sensor IDs, and speed readings.

Acquisition of data was initially provided by the University of Toronto, this data included road speed telemetry for thousands of sensors on various highways in Toronto. However, the location information for each sensor consisted only of its latitude and longitude, and so street segments had to be manually assigned to each sensor. To remap these road segments, Routetagger was developed; this program would provide a digital interface to manually assign road segments. Unfortunately, this task even with the aid of Routetagger proved to be infeasible within the time constraints, thus an alternative data set was required. Fortunately, trial sensor data from HERE Maps was provided by our supervisor Mohammed and the University of Toronto; this data included speeds for HERE road segments which could be mapped automatically to OSM segments. The automated assignment of road segments did create a small number of inaccuracies but was deemed acceptable.

The 'Map' file stored on HDFS maps sensors to OSM speed segments. The Map file is computed by querying OSRM for the route from the start to the end of a HERE road segment. A potential future upgrade would see the Map file move it to another MongoDB database to allow new sensors to be added without touching HDFS files. This would require an investigation into how updating documents in MongoDB affects derived spark RDD's.

Stream Processing

The stream job, named 'speedstream,' listens for input from the Kafka topic, splits each sensor reading into readings for each OSM segment covered by the sensor, and applies the update to a MongoDB database of live road speeds. Sensors are mapped to OSM segments through the use of a CSV file in HDFS, which maps each sensor name to a list of OSM nodes representing the street segment covered by the sensor.

The desired update algorithm has the following properties:

1. Historical speeds are taken into consideration when reporting the current speed, as sensors may produce erroneous values
2. Multiple sensors may produce different speed values for the same road segment at the same time; both sensors' telemetry should be considered.

To satisfy these properties, an exponential weighted average based algorithm was constructed. A live speed record for a segment has a value, weight, and timestamp; the present speed in kilometers per hour is the value divided by the weight. A 'half-life' parameter configures the speed of the exponential weight. To apply a new speed record, the following formulas are applied:

$$timestamp = new.timestamp$$
$$weight = old.weight \times 0.5^{\frac{new.timestamp - old.timestamp}{halflife}} + new.weight$$
$$value = new.value \times 0.5^{\frac{new.timestamp - old.timestamp}{halflife}} + new.value$$

In the event an old record does not exist, the new record simply inserted. As this algorithm was created arbitrarily to meet the aforementioned requirements and has not been heavily evaluated or compared to any possible alternatives, future work should include considering alternative algorithms and re-evaluating whether the desired properties listed actually add value to the project.

There are three options for applying a segment speed to the segment document in the MongoDB database:

1. Read the existing document, apply the update, and write the document back to the database with optimistic concurrency. This guarantees the update will be applied risks repeating the operation after optimistic concurrency checks fail. This risk is increased if sensors tend to send data to the system in waves.
2. Read the existing document, apply the update, and write the document back to the database without concurrency checking. This option risks sensor updates being lost, especially if two sensors for a single segment are sent at close to the same time.
3. Construct a MongoDB update query that applies the update in a single request. This guarantees that the update will be applied without adversely affecting performance due to optimistic concurrency checks. However, this means the program must construct a JSON expression to send to MongoDB which is less readable code than simply manipulating an object with well-defined operations in memory. Additionally, this update document must handle the edge case of a missing pre-existing record in the same transaction.

Of these alternatives, option 3 is the most suitable for this project as this project is intended to function as a distributed system with many sensors, so performance is paramount. Integration tests verifying that a datum has the desired effect on a document in a database are written to verify this aspect of the project's functionality.

Batch Processing

The batch job divides all historical data into buckets representing the time of day and applies an exponential moving average to reduce the dataset into a single value for each bucket and sensor. This assumes that traffic at a time of day in previous days predicts traffic at the same time of day at the present day.

In the future, the ideal batch job implementation would apply Spark MLlib to gain intelligent insights from a much larger dataset of historical road speed telemetry in order to predict future speeds. This would likely require the entire data model of the batch database to be revised. This is intent of batch layer; the present implementation is largely a proof-of-concept and a placeholder for this more advanced future implementation.

MongoDB

Initially, Cassandra was selected for use as the database, however due to difficulties installing Cassandra on the Spark cluster MongoDB was used instead.

OSRM Usage and Integration

OSRM supports two pre-processing pipelines: Contraction Hierarchies (CH) and Multi-Level Dijkstra (MLD); this project uses MLD. CH is optimized for query performance and long-distance queries but is less suitable when there are live updates to the data, such as for regular traffic data. The OSRM team is focusing development on the MLD pipeline and recommends MLD unless there is a special use-case that requires CH. As live updates are they key of this project, MLD is the optimal pre-processing pipeline for this project. OSRM requires an OpenStreetMap (OSM) map file; this project uses the complete map of Canada as provided by Geofabrik. (Running-OSRM, 2020)

OSRM accepts live updates to road speeds through traffic updates provided in a single CSV file. To apply updates, the provided program 'osrm-customize' is run with the CSV. While this file-oriented update process is not the most ideal (an option to accept the data from the standard input would allow the file or an equivalent output stream to be piped into the program), it is the only method provided by OSRM. (Traffic, 2020)

The OSRM module 'osrm-routed' provides OSRM's HTTP API to the project's website and mobile app.

OSRM Loader is sometimes called the 'microbatch' as it is a batch job but it runs frequently and produces smaller result than the batch job. OSRM Loader reads the last output of the batch process and the live data from the MongoDB database and combines these two with a simple weighted average. It then computes a map of OSM road segments to live speeds. As UTC time repeats leap seconds, the time of day can be computed from the UTC timestamp modulo 86400; no complex time calculations are required.

While the live speeds database contains most recent sensor data, it is the project's intention to use the batch data to gain insights into what the road speed for a given segment will be. Ideally, OSRM would accept predictions for future traffic speeds and adopt those into its routing algorithm as the routing engine gets further from the start point; however, as OSRM does not support this at present future speeds must be incorporated into a combined present value if they are to be used at all.

The current implementation of the OSRM loader is a shell script on the OSRM server that:

1. remotely through SSH runs the OSRM update batch job on the Spark master machine,
2. downloads the resulting file over SCP to the OSRM server,
3. Runs osrm-customize to apply the resulting file,
4. Spawns a new osrm-routed instance, and
5. Sends a graceful shutdown signal to the old osrm-routed instance, allowing it to finish pending requests.

Leaflet Interface

Leaflet is a library for JavaScript that was selected to develop the web interface for mXpress. Using the basic functions of Leaflet was relatively simple and did not require much code; due to its popularity there was a significant number of tutorials online that could be followed. Most basemaps are provided through MapBox and ESRI. These basemaps are free to use for development, however in a commercial setting this would require a paid license. Other functionality such as routing and geocoding require plugins. In order to facilitate routing with the OSRM server Leaflet Routing Machine was selected. This plugin creates a simple interface to setup routing functionality. Geocoding was likewise facilitated through the Leaflet Control Geocoder plugin; Bing Maps was selected as the Geocoding provider however a better option is highly desired due to the unreliability of its data. Due to time constraints several interface features were cut and could be implemented in the future. This includes a search bar, a vector speed layer to display average traffic flow, and biking, walking, and bus navigation.

Docker

Docker supports containerizing each service on the backend of mXpress. These containers can be deployed on any supported hardware and eliminates much of the setup, networking, and configuration process involved in starting these processes. Some of the containers used in the mXpress backend

deployment are adapted from first party containerizations and docker images provided by the respective technologies, while several were built entirely for the mXpress project.

Resulting from the multitude of technologies used in the backend of the mXpress system, a large number of containers are required to be deployed. These can be more easily managed, and deployed systematically using Kubernetes. Currently, mXpress has been deployed on a locally running Kubernetes cluster, but without further testing and validation this will remain on the sandbox environment away from production. Future development of this project will complete Kubernetes integration allowing for simple deployment to AWS, GCP, Azure, or other providers.

Limitations and Optimization Opportunities

The existing implementation's main limitation is that it is composed of shell scripts piping files through standard Linux programs and so incurs the overhead of using these programs. An alternative is to design a distributed networked program that communicates with its nodes directly:

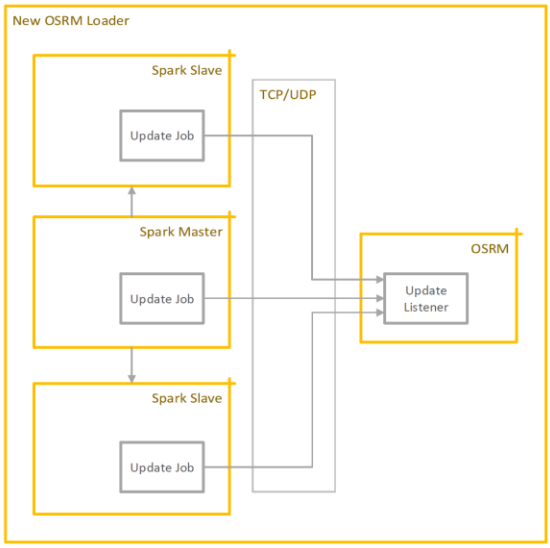

Figure 3: OSRM-Loader

In this new implementation described by Figure 3, the OSRM update job is run periodically. As this job is finished, the resulting dataset is sent entry by entry from each of the spark nodes to the OSRM server over TCP. The OSRM server will have an update listener listening for these TCP packets and would append the data in these packets to the update speeds CSV file; once all nodes sent a signal indicating they are done sending data, the listener program would then trigger the OSRM reload with the completed CSV.

As the speeds for a given road segment is temporary and will be overwritten by the next update job, there may not be a penalty for missed updates and so network performance could be improved by using UDP instead of TCP for the individual segment speed data points. However, without additional programming missing the 'finished' signal would prevent the data from being flushed to OSRM and so the 'finished' signal must be sent over TCP to guarantee delivery.

Additionally, If OSRM provided a means to pipe update speeds into osrm-customize without using the file system, this listener could use that instead of the present file loading as an additional optimization.

Testing

Our problem is an integration challenge, not a programming challenge. Due to the team's lack of experience in testing complex systems, very few automated tests were developed. The tests that were developed include:

1. Verifying our sensors are correct: fake congestion data was inserted into the system to simulate a traffic jam on Don Valley Parkway. A route on the frontend was then requested along this segment; the reported travel time along this segment was extremely long and routes to and from nearby points avoided this expressway, verifying that our sensor data is integrated into OSRM correctly.
2. Verifying spark jobs: running the program and manually checking that the database contains the expected results.
3. Integration tests performed on database loading component of streaming module; verify a single unit of telemetry has desired effect on db.
4. Basic forms of unit testing as components were developed
5. Load testing on the system as a whole

Development

Multiple challenges were encountered during development. A large amount of coding was done on a text editor on the local Spark VM. This required the programmer to be remoted into the local network through a VPN and then remoted to the server through VMware vSphere which in itself was quite buggy. A continuous deployment (CD) architecture where the code is automatically pushed to GitHub, tested, compiled using SBT (Scala Build Tool) and automatically implemented would make this process much simpler. This would allow a developer to write code for the program without having to have local access to the machines and have their changes automatically pushed live to the sandbox or production environments. Server administration, technology setup, and virtual machine deployment was also a challenge. One of the drawbacks of using older server equipment is the lack of support for newer versions of the vSphere software suite. This meant that our team was stuck using older and unsupported versions of the software, many of which contains bugs when using modern browsers and hardware. One of the most frustrating bugs that was one that required the complete restart of the user interface in order to make any changes to a virtual machine. Another major downside of working directly with server equipment was the time it takes to simply start a server. Some physical servers take upwards of 20 minutes for a reboot due to memory testing. In the future, a better solution may involve utilizing a service such as AWS, Azure, or Google Cloud. The move to one click deployments of the backend infrastructure using containerization provides a variety of improvements to minimize the limitations of self-hosted hardware. However, the move to Kubernetes in the production environment will also require some redevelopment of the deployment pipeline.

Conclusion

The development of mXpress was certainly a challenge. The obstacles faced by using cutting-edge technologies, lacking documentation, lacking the appropriate hardware, and being novice coders were difficult to overcome. However, the lessons learned and the experience gathered when facing these obstacles were extremely valuable.

References

Running-OSRM: Running OSRM. (n.d.). Retrieved April 9, 2020, from https://github.com/Project-OSRM/osrm-backend/wiki/Running-OSRM

Traffic. (n.d.). Retrieved April 9, 2020, from https://github.com/Project-OSRM/osrm-backend/wiki/Traffic