Acknowledgement

Huge thanks and appreciation to all people who had a hand in the actualization of FenceX. First and foremost, my supervisor Salman Toor who trusted in me and my idea. Without his continued feedback and support FenceX wouldn't make it to existence. Also, the reviewer of this thesis, Sadi Alawadi, whose feedback was necessary to keep the work relevant and scientific. Special appreciation to Snic cloud who provided us with computational resources needed for the evaluation of FenceX under project 2020/20-3. Also, Cabonline and my manager there at the time, Emil Ekdahl, for providing us with real taxi trip data (anonymized) which we have used intensively in the evaluation phase. Last but not least my friend and ex-college, Juan Fandino, whose work was inspirational for FenceX.

Open source

All of the work that has been done during this thesis including reports and implementations is available in a Github repository (github.com/bmd007/statefull-geofencing-faas). The content of this repository is available to be used freely. Please feel free to contribute as well.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Geofencing[4] plays an important role in our day-to-day life nowadays. The most famous use case of geofencing is to take a taxi[5] from an online taxi provider like Uber. A traveler selects a pickup point and the system should send the offer (order) to the available taxies. The system can not send the offer to all the taxies available around the world. It might take forever until one of them accepts the offer, which results in a very poor user experience. So what if the taxi provider comes up with an area on the map in which cars can reach the pickup point in less than 5 minutes. Or an area on the map, a circle, with a radius of 1 KM around the pickup point? Let us call this hypothetical area **fence**. On the other side of the story, we have taxies with a GPS sensor attached to them reporting their location every now and again to the system. Maybe every 20 seconds. In this example, to geofence is to find cars whose latest reported location is within that fence. In this type of geofencing, the fences are created on the fly whenever there is a request for a taxi. So the fencing operation happens in an on-demand manner.

Another example of geofencing use cases is pet care[6]. Consider a cat living with a family in a safe and calm neighborhood. The cat can play around the neighborhood with no need for a caretaker. Within that area, the cat can find his way back home easily. The cat owners do not want the cat to go out or stay out of that safe zone. So they attach a GPS sensor to the cat and make it report the cat's location every minute to a geofencing system. They also draw a fence (shape) on a map (using UI of the geofencing system) representing their neighborhood, the safe zone. Then ask the system to notify them by sending a notification to their phone whenever the cat goes outside of that fence. In this example, the fence is not as dynamic compared to the previous example. The cat owners draw the fence once and the system saves it. The fences change if the cat gets older or they move to another area for example. What triggers a fencing operation in this example is each of the location reports which GPS modules sends to the system. The GPS module attached to the cat. So every minute one geofencing operation happens to check if the cat is still within the safe area or not. This type of geofencing is of realtime nature.

The examples above help to clarify how this thesis has categorized geofencing operations. On-demand and realtime. Each of them has different functional and non-functional requirements. The style of geofencing is not the only factor affecting the requirements. In some use cases, like food delivery, the rate of location reports can be as low as one per minute for each mover without any problem. In other cases, like taxi business or police dispatch, the rate of location reports can be as high as one per 30 seconds for each vehicle. These numbers give a better impression about non-functional requirements of geofencing systems when considered together with the number of moving objects interacting with the system. Just for the sake of example consider a taxi company with 6000 active cars, which is not far from reality. 6000 cars sending 1 location update every 30 seconds. It means 12000 location updates to be processed every minute. So a geofencing system is required to be able to handle different rates of input load.

For on-demand geofencing, the system has to keep track of the latest reported location of movers while in realtime style, a database of fences is required. In on-demand geofencing, a geospatial index is queried which results in incomparably higher latency. On the other hand, realtime style only requires a number/id index and has lower latency consequently.

To implement a geofencing system, a database engine that supports geospatial index with a simple MVC[7] web application on top should suffice. One package of code is responsible for receiving location reports, processing them, then saving them in a database table, handling (HTTP) requests coming from UI, and even rendering the web UI. This application can be deployed very easily on a virtual machine.

The described system, however, will not satisfy the non-functional requirements that we have mentioned before. Also, it is not easy to change any part of it. After each change, the whole software should be compiled, one big artifact should be created, all tests should be executed (if any exists) and a time-consuming deployment should happen. If anything happens during run time, the whole service goes down. In case the incoming load increases enough so that the currently available resources get saturated, the only way to handle the situation is to add more resources to the machine running the application. These are some of the common problems with monolith[8] applications.

Our contribution is design, implementation, and evaluation of a geofencing system supporting both styles of geofencing (on-demand and realtime) called FenceX[9]. It relies on stream processing[10] and microservices[11] in order to achieve high throughput, high scalability, and high availability. We have achieved these characteristics by using microservices as the major architecture of the system. This means our system consists of small de-coupled components trying to share as little as possible. These microservices communicate using Kafka[12] which is an event streaming[13] platform. Such style of communication enabled our work by becoming a foundation for stream processing. Combining Kafka and Kafka Streams[14], a stream processing framework, has resulted in a stream processing pipeline that its operations are packaged and deployed as microservices.

Having a stream processing based system that has stateful operations give us the ability to look at the system from function as a service[15] point of view. The easiest approach at the moment to write a code and deploy it in the cloud is to write it as a function. then give the code to one of the FaaS (function as a service) providers like AWS Lambda [16]. They will do the all required work to expose that function, as an HTTP API for example, to the internet. However, such FaaS providers do not support stateful functions. So only stateless operations like compression of an image are well suited to enjoy the convenience of FaaS. Our work, apart from being a geofencing system, is an implementation for the idea of FaaS which supports stateful operations as well. We have modeled execution of a function in a FaaS environment with the execution of an operation in a stream processing pipeline. Since our pipeline supports stateful operations, we can have stateful functions as well.

Chapter 2

Related works

2.1 Large Scale Indexing of Geofences

In [17], an on-demand style geofencing system is introduced that supports large scales of data and load. This system has a concept called worker which refers to instances of an application that answer queries. As opposed to clients which send queries to the worker nodes. Dynamic caching and workload distribution over multiple instances of workers are the bases of achieved scalability.

Each worker instance is responsible for indexing a region of the world, which allows for lower query and index latency. Since each worker instance is responsible for a different part of the world, queries targeting border areas need special treatment. As a solution, each worker instance also is responsible for areas around its dedicated region.

Notably, software in [17] uses a custom geospatial index, custom design and implementation. Using a custom index most probably means that the whole data is stored and processed in memory which allows for higher throughput. We will not, however, describe the indexing strategy further since it is outside the context of this thesis.

2.2 Using Complex Event Processing for implementing a geofencing service

A realtime style geofencing system has been introduced in [18]. It relies on a CEP (complex event processing) for fence entrance/exit detection. This is comparable to functionalities of the push leg of FenceX where realtime fence point intersections happen. System in [18] is using a classic 3 tiers monolith architecture. Unit of concurrency in [18] is not clear. So it is not clear what are the bounds for the scalability of the introduced system. Throughput in this system is defined as the number of location updates processed per second. Evaluations have shown that it has achieved a throughput of around 20k/s.

2.3 Cloudburst: stateful functions-as-a-service

In [3], to achieve stateful FaaS, co-location of cache and functions has been taken into action. Each function accesses the cache as storage of state. Functions are executed by function executors which are deployed in virtual machines. For each virtual machine, one cache instance exists. To keep the caches consistent, they are backed by an auto-scaling key: value store called Anna. And Anna uses lattice composition to fight inconsistency. Anna also has an ACID2 conforming function called merge which is used by [3] to gain high scalability.

2.4 On the FaaS Track: Building Stateful Distributed Applications with Serverless Architectures

The work described in [19] resulted in a FaaS platform; which supports stateful operations in low latency and flexible manner. They achieved stateful FaaS using a global in-memory strongly consistent data layer for state mutation. This data layer can be accessed by all the functions (parallel workers/cloud threads). For workload distribution, they have partitioned the data layer. For durability, they allow replication of data in the data layer. However, by default, no data is replicated (replication factor is 1). Fault tolerance comes from data replication and the possibility of making functions execute again when facing failure. This platform is best suited for computations that require direct state mutations (rather than immutable state).

2.5 Transactional Causal Consistency for Serverless Computing

The word done in [20] can be summarized as a solution for reducing the I/O latency in a stateful FaaS platform. They achieved the stateful aspect of the FaaS by relying on colocated caches to the executing functions. The solution provided by [20] also aims for application-wide consistency. The main challenge in their work was to make Multisite Transactional Causal Consistency (MTCC) possible. MTCC can be explained as having causal consistency within a transaction even if it runs on machines located in different physical sites. To make MTCC possible, they have implemented MTCC protocols using a system called HydroCache.

2.6 Comparison with the state of the art

The work that has been done during this thesis is distinguishable from other reviewed works in terms of the achieved level of availability, scalability, and peak throughput. Using a modern system architecture called microservices and modeling the system as a stream processing pipeline implemented on top of Kafka with help of Kafka Streams make our work different from others.

Bringing stateful FaaS idea into the realm of geofencing is also uniquely done in this thesis. Geofencing operations involved in FenceX can be implemented as custom libraries and be plugged into the system. At the highest level of abstraction, users (developers) can implement those functions using a simple web GUI and deploy the result with a few clicks. This is not a feature of any of the geofencing related reviewed works.

In terms of the level of consistency, FenceX is eventually consistent and as a result, can deliver higher throughput compared to the reviewed works, which are strongly consistent.

The number and range of tests and experiments carried out to evaluate FenceX during this thesis is outstanding. Extensive experiments for checking throughput, availability, and scalability. These experiments have been done with a setup close to production with production level tooling in addition to production level data replication. As a consequence, the result is more reliable.

Chapter 3

Geofencing

Geofencing essentially is checking whether a geospatial point is inside a geospatial shape or not. This can be done in two styles: push and pull. In pull style, a fence (geospatial shape) is queried against a database of coordinates (mover-id: geospatial point). The result is all coordinates residing in the fence. This type of geofencing relies on geospatial indexing. So updating and querying the database of coordinates is CPU intensive.

On the other hand in push style, fences are pretty much predefined. Whenever the coordinates of a mover changes (a location update report arrives), an intersection check between a predefined fence and the newly reported coordinate will happen. Although no geospatial indexing is necessarily involved, the intersection checking operation is still CPU intensive due to the required mathematical calculations.

We can compare these two styles as following:

- In pull style geofencing a **database of locations** exists.
- In push style geofencing, a **database of fences** exists.
- Pull style geofencing has an **on demand** nature.
- Push style geofencing is of **real time** nature.
- Pull style geofencing use case examples: Taxi/police/ambulance dispatch, disaster management.
- Push style geofencing use case examples: Elderly/Kid/Pet care, (gambling) addiction management.

Please note that using pull/push terminology is an idea brought up in this thesis in order to facilitate conversations about different approaches to geofencing.

In order to represent geospatial data several standards (formats) are available, namely WKT[21], WKB[22], GeoJSON[23] and GML[24]. These representation standards use general-purpose data transfer formats like JSON as their base. Table 3.1 shows the data format that is based on the mentioned geospatial standards.

Geofencing format	Parent data format
GeoJSON	JSON
GML	XML
WKB	Binary
WKT	Plain text

Table 3.1: Base standard of geospatial data representation formats

We can compare the geofencing representations mentioned in the table 3.1 fundamentally in the same way in which we compare their corresponding base standards.

In this thesis, we used WKT to represent all the involved geospatial data. We have chosen WKT since our geospatial database (index) of choice, H2[25], supports it well. Also, we couldn't find any information about H2 supporting other geospatial representations out of the box.

3.1 WKT

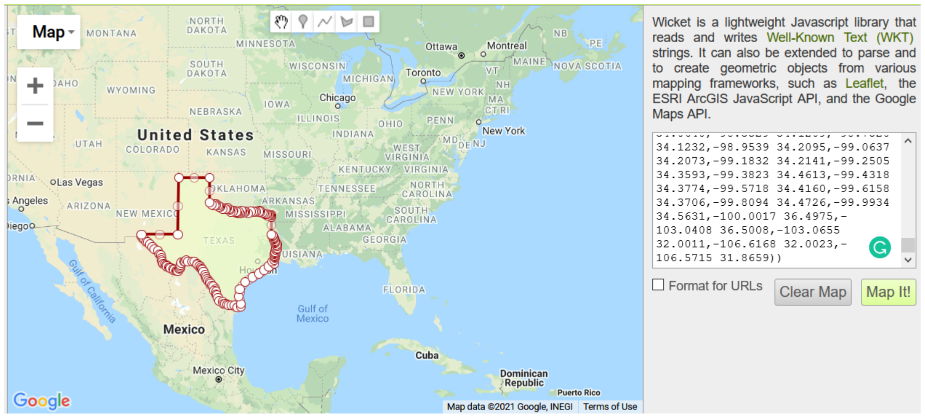
WKT (well known text) is a text mark up language which is a format for geometric information. Items below are examples of WKT strings representing raw geometry data.

- POINT(6 10): a point with x=6 and y=10
- LINESTRING(3 4,10 50,20 25): a line consisting of 3 points.
- POLYGON((1 1,5 1,5 5,1 5,1 1),(2 2, 3 2, 3 3, 2 3,2 2)): two squres one convering the other
- MULTIPOINT((3.5 5.6),(4.8 10.5)): just two points
- MULTILINESTRING((3 4,10 50,20 25),(-5 -8,-10 -8,-15 -4)): two lines
- MULTIPOLYGON(((1 1,5 1,5 5,1 5,1 1),(2 2, 3 2, 3 3, 2 3,2 2)),((3 3,6 2,6 4,3 3)))
- GEOMETRYCOLLECTION(POINT(4 6),LINESTRING(4 6,7 10))
- POINT ZM (1 1 5 60)
- POINT M (1 1 80)
- POINT EMPTY

- MULTIPOLYGON EMPTY

Instead of relative values for X,Y, and Z, geographic coordinates (latitude, longitude) can be used in order to represent geospatial data. We mainly used POINT() to model locations and MULTIPOLYGON() for fences.

Figure 3.1: Texas WKT illustration



Wicket is a lightweight Javascript library that reads and writes Well-Known Text (WKT) strings. It can also be extended to parse and to create geometric objects from various mapping frameworks, such as Leaflet, the ESRI ArcGIS JavaScript API, and the Google Maps API.

```
34.1232,-98.9539 34.2095,-99.0637
34.2073,-99.1832 34.2141,-99.2505
34.3593,-99.3823 34.4613,-99.4318
34.3774,-99.5718 34.4160,-99.6158
34.3706,-99.8094 34.4726,-99.9934
34.5631,-100.0017 36.4975,-
103.0408 36.5008,-103.0655
32.0011,-106.6168 32.0023,-
106.5715 31.8659))
```

Figure 3.1 illustrates usage of an online tool [26] to show a geospatial WKT on a map. The WKT refers to borders of state Texas in the US which is taken from [27].

3.2 Challenges related to geofencing

Both styles of geofencing are CPU intensive. Also, geofencing is usually implemented using databases interacting with disk and accessed over a network. So a combination of natural computational latency and I/O latency limits the overall throughput of geofencing systems. This gets worst in high load high scale use cases. This means the higher the load, the higher the effect of the aforementioned latencies on the overall performance of the system. As a remedy, we can improve the computational aspect of the problem by improving the indexing algorithms. However, usually, the I/O latency is much more effective than the operational burden. So the other approach to improve the throughput is using co-located in-memory databases which do not interact with disk and are accessed through a local network (localhost). In this thesis, we focused on the latter and tried to minimize the I/O latency.

We decided to use co-located in-memory databases to minimize I/O. Everything will be fine until our system faces an amount of load which can not be handled properly even after scaling up. At this stage, we scale the system out and deploy more than one instance of the program. This will also help with the resiliency of the system; if one of the instances of the program goes down or stops responding for some reason, the system can still keep serving with the remaining available instances.

Until now, we achieved high throughput, scalability, and resiliency by minimizing I/O latency and scaling the application out. This means we ended up with a distributed system containing some components (instances of our application) that need to have the same image of the world, same data more precisely. The location updates or queries sent to the system by clients can end up in any of the available instances due to load balancing. Otherwise, there should be a way to shard the data between instances and tell clients how to select the correct instance to which send a location update or query. We went for replication over sharding with the hope of keeping clients and data management simple. So we need to somehow make those instances have the same image of the world (same data) otherwise, their results will be inconsistent.

So far we have exchanged I/O for the holly inconsistency issue of distributed systems. So the challenge is to keep the throughput, scalability, and resiliency of the geofencing system high while avoiding inconsistency among its components.

3.2.1 Functional requirements

Functional requirements refer to the expected behavior of a software system, its features. We tried to describe functional requirements in the style of user stories[28].

Movers want their location reports to be received by FenceX. This report contains the coordinate (latitude, longitude) at which the mover was located when producing the report. A timestamp might be included in the report as well.

Movers want their latest reported location to be available for being queried by their ID (Movers want their latest reported location be memorized by the system).

Users want to be able to send a geospatial shape (fence) to the system and in turn get movers (id: location) who are in the fence (according to their latest location report) = on-demand fencing

Users want to be able to create/update/remove/query a fence (geospatial shape) for each mover (by id).

Movers expect each of the location reports to be checked against a potential fence. Which means for each location report for each mover, potentially a fence-point intersection should be calculated. The result of each intersection is if the reported coordinate is located within the predefined fence. In case no fence is defined for a specific mover, no fence-point intersection should be calculated.

3.2.2 Non-functional requirements

Non-functional requirements are characteristics, attributes or properties of a software system. They are usually in the style of a constrain or a limitation.

As a result of the required system being a general geofencing provider, the required precision of the system might vary based on the users and use cases. So FenceX should be able to ingest very high rates of location reports. Looking at potential use cases, in some, the report rate could be like one report per mover every minute and a maximum of 1K active movers at the same time. Other cases might involve 1 location report per mover every 5 seconds with 1M active movers. So FenceX should have a very high capacity to handle load and stress.

Since based on the use case, the required scalability of a system may differ, the users of the system should be able to tune the system according to their needs. Or at least the fact that scalability requirements are not fixed over time should be factored in the design,

Availability of FenceX should rely on modules so that if something goes wrong in one part of it, the other parts keep working. If FenceX stops receiving location reports, for instance, it should be still possible to query the saved locations by mover id and/or fence. In some cases, it is acceptable to miss some location updates for some movers while in others, missing location reports are not tolerable due to high precision requirements.

The time that takes for FenceX to answer a query by a fence or calculate a fence-point intersection should not be a a factor of the incoming load. In other words, the latency of those two operations should be reasonably low and should not increase as the load on the system increases. Such low latency will play an important role in the overall throughput of FenceX.

The consistency level required for geofencing applications is not strong so eventual consistency will suffice. Once a location report has been received, it might take a fraction of time until the view of FenceX about corresponding mover changes. Before that, the latest previously captured report and the finalized view are valid and used. The same goes for predefined fences. So no need for distributed transactions or implementation of Saga as there is not much of intertwined interactions that can go wrong.

Chapter 4

FaaS, Function-as-a-Service

Apart from being a geofencing system, FenceX a can also be considered as an implementation for the idea of function as a service[15]. Stateful function as a service to be exact. In a sense, geofencing is just one special usage of our system. So other types of computation can also be implemented in the same way without changing the architecture in any significant manner. In the following sections, we will explain what is FaaS and how FenceX is a FaaS provider.

4.1 Function-as-a-Service

Consider a start-up company is shaping around a brilliant brand-new audio compression algorithm. Two computer science engineers came up with an algorithm and now want to make money out of it. They do not want to publish any software or library but an online service, a few APIs (HTTP for example) only. They are confident enough in their work to know that once their service is up and running, a lot of companies want to use their service right away. Which means a huge number of requests to handle. So however they design and implement a system around their algorithm, it should be highly scalable and efficient.

Based on what we described about resilient highly scalable systems with high throughput, they should select microservices as architecture for their software behind their APIs. They need to sit and design different microservices, hire developers to help them with implementation and maintenance, hire an operations team for maintaining container orchestration tools (among others), and so many other activities before even they can start coding. Let alone deploying a feature into production

However, they are thinking about a short time to market since they have noticed an opportunity soon to pop up in the market for their service to be used vastly. Also, since they are a small startup, they are low on budget. Most probably they can only hire one part-time front-end developer to create a website for introduction purposes.

So far they can not achieve a highly scalable system. What if they could just focus on the implementation of their algorithm? Just like they are implementing a function in the middle of nowhere. A function that is written in C, or a python function for that matter. They are implementing an audio compression algorithm. After all, it's just some calculation and computation over an input then producing an output. So what if they could just implement that function, give it to a cloud service with the promise of no more work required for exposing the API to the internet. The compression algorithm will be exposed to the internet as an HTTP API. It can also be configured to be triggered upon receiving a Kafka event, gRPC call, AMQP message, and much more. The start-up people won't need to worry about anything else. The cloud provider will take the code, build it somehow, and bring enough instances of the resulted artifact up and running according to the incoming load. So dynamic scalability out of the box.

Payment will be based on the number of processed requests or the running time of functions.

Any change to the code will be just changing a function.

What we just described is the sell speech for the function as a service idea. Which is provided nowadays by most cloud providers. AWS Lambda is the most famous one.

4.2 Stateless FaaS

In the FaaS model, apart from separating business logic and execution platform, storage is also separated from execution platform. That is because storage not being easy to scale in a distributed fashion. As a consequence, having very small computation units working decoupled comes with the price of not being able to access storage. Or not being able to access storage classically. I/O is usually an expensive operation regardless of the execution model. When it comes to FaaS, I/O becomes much harder to achieve due to the following reasons:

Network: Since there is no storage and file system attached to each function, all I/O should happen over the network which brings about high latency which reduces throughput as a result.

Run time limitation: Each function, once called, has a limited time to live. If computation does not finish during the time limit, there will be no more chance to continue it. The high latency of I/O over the network does not match this limitation.

Decoupled context: Since all the running instances of the function are decoupled, there is no way for them to share something, a connection to the database for example. So each time a function requires a database or storage connection, a brand-new connection should be created. Which results in unnecessary latency, too many connections, and loss of execution time budget.

As a result, the FaaS model is better to be used for stateless operations. Creating a thumbnail for an image, compressing an image, or compressing a piece of audio. All the required computation can be carried out without the need for storage or a database.

So what about stateful operations? Can we enjoy the high-level abstraction of the FaaS model with stateful operations?

4.3 Stream processing and Stateful FaaS

For developers (users of the FaaS platform), the function they implement has only some input parameters and one output. The input parameters can be driven from the triggering event/request/message. This is not different from implementing operations in a stream processing pipeline. Each stream processing operation, regardless of being stateful or stateless, is very similar to a function which has an input (triggering event) and an output (resulting event if any). For stateful operations, another input parameter exists which is the latest captured state for the key of the event. Please remember that in stream processing systems, events and states are key:value tuples.

So we can implement a FaaS platform by modeling it as a stream processing pipeline. What we need is a source operation that has some adaptors. The adaptors turn an incoming request, HTTP for example, into a Kafka event so that operations down the pipeline can process them. Of course, this implementation is asynchronous and eventually consistent. Paying that cost brings about the benefit of stateful operations. Since we can just enjoy stateful operations in a stream processing pipeline in a highly scalable manner, a FaaS platform implemented by stream processing pipeline allows for the definition of stateful functions as well.

The only thing that users of the system (developers) should do is to implement a body of stream processing operations.

4.4 FenceX and FaaS

FenceX, as described previously, is a stream processing pipeline among many other things. It has both stateful and stateless operations. However, FenceX is designed in a way that operations are defined by architecture but their implementation should be provided as libraries. Each operation on execution uses an object of a Java Class called Function. The mentioned library provides instances of those Functions. So such a library can be provided by users of FenceX (developers) in the same way that FaaS users provide an implementation for their functions. The signature of functions however is pre-defined due to architectural decisions. And the available operations to implement are also limited to what is needed for the architecture. So FenceX is an implementation for the idea of Stateful FaaS. Since FenceX needs the compiled version (library) of Functions, they can be implemented in any JVM language as long as they get compiled into a Java class (Java byte code). So for example, an instance of the class Function (from Java) can be created by a Kotlin[29]. Being language agnostic is another promise of the FaaS idea which FenceX delivers to some extent. Table 4.1 compares FenceX with the solution provided by [3].

Property/System	FenceX	[3]
Consistency	Eventual	Strong
Statefulness strategy	Stream processing	Co-located caches
ACID2 conforming	yes	yes
Data replication	yes	yes
Function execution retry	yes	yes
State as KEY:VALUE data	yes	yes
Custom functions	practically yes	yes
Underlying system	Kafka/Spring/KafkaStreams	Anna

Table 4.1: Comparison of [3] with FenceX

Chapter 5

Theoretical concepts and related technologies

In this section, we cover the architectural and scientific ideas and technologies which played a major role in this thesis. We cover them only to the extent that the context of this thesis allows.

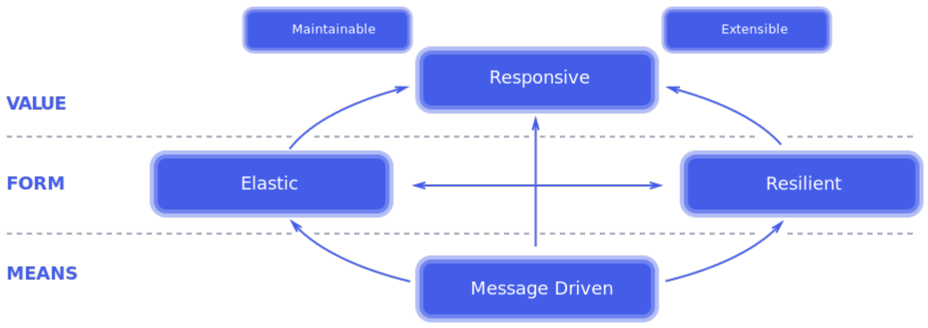
5.1 Reactive systems

A distributed system in simple terms is software having more than one deployable component. These components are sometimes equal and/or sometimes different. However, from point of view of an outside observer, there is only one working system. From within, there are many processes, actors, services or etc working together to serve an overall purpose.

Among distributed systems architectural styles, reactive is of special interest in this thesis. A reactive system, as defined in reactive manifesto [1] has the following properties:

- **Responsive:** a responsive system provides fast response in a consistent manner.
- **Resilient:** A resilient system keeps responding to the possible degree in face of failure. **replication and isolation** are cornerstones of resiliency.
- **Elastic:** An elastic system keeps responding under varying workloads. Avoiding bottlenecks and cost-effective resource allocation plays an important role in achieving elasticity.
- **Message Driven:** Non-blocking asynchronous message passing brings about isolation, location transparency, and back pressure for distributed system components.

Figure 5.1: Reactive systems' value chain [1]



As figure 5.1 suggests, designing and implementing a distributed system in reactive style has the following benefits: easier to maintain, easier to change, and higher responsiveness.

So in summary, a reactive system consists of isolated components communicating using asynchronous non-blocking message passing.

An ideal reactive system conforms to ACID2[30]. ACID2 is the counterpart of ACID in transactional databases. So instead of atomicity, consistency, isolation, durability, ACID2 is:

- **Associative:** Messages can be grouped together, using a key for example.
- **Commutative:** The order in which messages arrive at components does not affect the desired outcome.
- **Idempotent:** Components of the system should receive a message more than once, they act as if the message arrived only once.
- **Distributed:** The system has more than one component.

A reactive system, especially when conforming to ACID2, has the potential to scale out very easily to a very high extent.

When it comes to replication of data over components in a reactive system, in case of consistency matters, order of messages become important. The order in which messages arrive has a logical time aspect which is important for achieving occasional consistency. It is very challenging for a distributed system to be strongly consistent. So if it is acceptable to relax the requirements to occasional/eventual consistency, a reactive system in which the order of messages is reserved works well.

The style with which messages are produced has a direct effect on system design and implementation. A message can be a query (what is X), an event (X has happened), a command (do X), and an update (X is 232 now). The preferred message semantic in reactive systems is **event**s. An event represents something that has happened in the past, an immutable truth. Such events can be published into a queue reserving their order. System components can subscribe to the queue and update their own image of the world (state) according to the events that have happened.

In order to achieve such a system design, events should be considered first in the design rather than the domain. Verbs first rather than nouns, what happened vs what exists, immutable fact vs cumulative knowledge.

5.2 Microservices

Once upon a time the whole software, an information system, for example, was one process deployed manually on a big physically available machine. This software was maintained by a big team in one codebase. To change any part of it, was a headache to make sure even the smallest change will not break the whole software. Everything was tightly coupled after all. It was all about code re-use. If developers managed to change it properly, building the code took hours and forget about how long it took to run the tests (if there were any of course). After manually deploying the executable artifact into a physically available server in the corner of the office, an equal sign forgotten in a loop condition brought the whole service down 11 pm same day. The manager panics and wakes developers up with the hope of a hotfix.

Hotfix applied, everyone back to sleep, customers are happy. In fact so happy that they invite their friends to use the service. More clients, more money, great. Too good to be true though. 14 pm next day, the number of users using the service simultaneously rockets to the outside range of available resources. CPU usage is 100 percent, no free memory left on the machine and only one out of 10 users is actually getting service instead of just looking at the loading bar.

Developers run around the city to find RAMs with bigger capacity as a temporal solution. IT people start to think about assembling a more powerful server with all brand-new more powerful hardware.

Those days are gone due to the adoption of cloud technologies combined with microservices.

Microservices is an architectural style in which software is divided into some small components called services. Those (micro)services communicate with each other and serve the overall purpose of the software. Each service is maintained by a special team, has its own code base, has its own database, exposes its own APIs, is tested independently, is packaged separately, and is deployed as a separate process.

The less these services share, the better. Decoupling services as much as possible is an important aspect of microservices. Because for example, the less services are dependent on each other, the easier they can be changed.

So now that software is developed into microservices, each microservice can be changed independently by the team responsible for maintaining it. Also, deployed separately after a short build and test process. A bug in one of the services can not bring the whole system down as well. When the load on one of the microservices increases, more instances of it can be easily deployed (scale out).

And hopefully, instead of deploying executable artifact into a physically available machine in the corner of the office, microservices are packaged as docker[31] images and deployed into a container orchestration tool provided as a managed cloud service.

Of course, turning a monolith into a distributed system brings all the challenges of distributed systems with it. Having data spread around different services makes strong consistency very hard to achieve. Load balancing between available instances of each service is something new to deal with. Monitoring of and communication between services requires special attention. And so on. In order to solve the mentioned challenges and much more, best practices, design patterns, architectural patterns and tools become handy. System designers and developers select a subset of these helps based on the functional and non-functional requirements of their system, size of their organization, and other relevant factors. API gateway, Service registry, Distributed tracing, CQRS, Saga are some popular examples. [32] [33] [11]

5.3 Stream processing

Stream processing refers to a special style of reactive systems. It is also equivalent to data flow programming in which a graph of operators exists. These operators receive events as a result of the work of other operators, process them, and produce other events. These events represent immutable facts, something that has happened. The order at which events arrive at the next operator matters; so usually in stream processing systems, a publish-subscribe pattern implemented using some sort of queue, journal, commit log, or topic. [2] [34]

The foundation of a stream processing pipeline is its data flow graph.

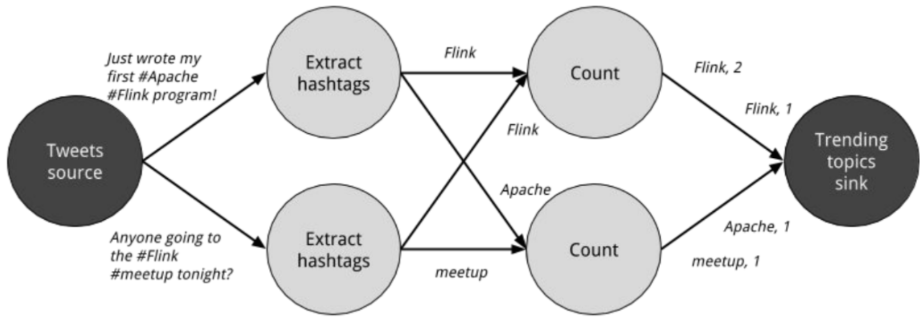
5.3.1 Data flow graph

A data flow graph is a useful tool to express and study how data is flown in a stream processing system. There are two type of data flow graphs: **logical** and **physical**. In a logical data flow, graph nodes represent **operators** like sum, count, filter, and aggregate. It does not give any information about the deployment. On the other hand, a physical data flow graph illustrates how exactly the pipeline looks when deployed. Like how many instances of each **task** should be deployed. A task is the equivalent of an operator in a physical data flow graph. We implement an operator and deploy it as some tasks. Similar to implementing a microservice and deploying some instances of it.

Graph 5.2 shows an example of a physical data flow graph with 6 tasks (3 operators). This pipeline counts occurrences of each hashtag in a stream of tweets which includes the following operators:

- A source that brings tweets into the stream processing pipeline
- A processor that extracts hashtags

Figure 5.2: Example of physical data flow graph. Taken from [2]



- A stateful processor that counts occurrences of each hashtag
- A sink that represents the end of this pipeline which receives the realtime count of each hashtag. It might save results in a file or publish them to another system.

5.3.2 Parallelism

There are two types of parallelism in stream processing systems:

Task parallelism Tasks from different operators process the same data. Consider a stream of tweets for example. One operator extract hashtags, another one extracts mentions, and maybe another counts number of words in each tweet.

Data parallelism Tasks from the same operator process different data. The data (event) in the topics of a pipeline is in key:value format. Data with the same key ends up in the same task unless the corresponding task stops working (re-balancing). Data parallelism is achievable by technics like partitioning of the topic which is implemented by tools like Kafka. Data parallelism is basically equivalent to the concept of scaling out in microservices terminology (scaling horizontally).

5.3.3 Latency

In a stream processing system, latency is defined as the time takes for ONE event to be processed. In an ideal stream processing system, the latency reflects the actual processing work on the events. Which means events do not wait to be processed. In other words, events get processed as soon as they arrive. **95th-percentile latency value of 10ms‘** is an example of expressing latency in a stream processing system. It means 95 percent of events are processed within 10 ms. Obviously, the lower the latency, the better.

5.3.4 Throughput

In a stream processing system, Throughput depends on both latency and input rate. So low throughput does not necessarily mean high latency. A better metric is **peak throughput** which shows the limit on the performance of the system at maximum load. When a system reaches peak throughput, increasing input load will actually reduce throughput and can even lead to data loss (due to buffer overflow).

5.3.5 Operations

There are two types of operations: **Stateless** operations like filter and map are not concerned with the past. They are easier to scale and more resilient. While **stateful** operations keep a snapshot of past (state). The state may mutate after processing each event. Operations like sum, count, average, and aggregate are stateful. Such operations are harder to scale. The hardship comes from the fact that apart from scaling out the tasks, the state part assigned to each task need to become co-located for the task. It brings challenges like the best partitioning policy and correct key selection. For example, when partitioning geospatial data partitioning policy may differ based on the use case.

5.4 Kafka

Kafka[12] is an event streaming tool. It is a distributed implementation of a journal (commit log). The main components in the Kafka ecosystem are a cluster of servers (brokers) and clients (event producers and consumers). The broker is written in Scala while client libraries are available in a variety of programming languages.

Kafka is one of the famous go-to options when an architect goes for the publish-subscribe pattern. It has a concept called **topic** which is a durable journal (a queue). The subscribers to the topic can be grouped for load distribution purposes. Each topic can be partitioned so that each group member subscribes to a partition of a topic. The publisher is responsible for selecting the partition to which an event should be published.

Events in each partition can be replicated over a cluster of Kafka brokers for availability purposes.

only once and **at least once** are the available quality of services that Kafka provides.

When using Kafka, a set of APIs are available to use: producer, consumer, admin, connect, and **Kafka Streams**. There are basically ways for interacting with the broker and their name exposes their functionality.

5.4.1 Kafka Streams

Kafka Streams[35] is a JVM language compatible library which allows for publishing data into and reading data from a Kafka cluster in form of stream processing operations. A microservices can just have it as a library, connects to Kafka, defines an operation like **filer** or **aggregate** and becomes a node in a stream processing pipeline, a node in a data flow graph. The definition of such operations starts with creating a KStream on top of a topic. Each event in the topic will be an event in the stream.

As opposed to other Kafka APIs like producer that is stateless, Kafka Streams supports both stateless and stateful operations. For stateful operations, Kafka Streams ask for defining a KTable. KTable is a key:value table which is essentially backed by a Kafka topic as a source of truth. However, for ease of accessibility, the data in the topic gets loaded into a database which is RocksDB by default. In the case of FenceX, in pull leg, we have used H2 in an in-memory fashion.

Kafka Streams concept of KTable is not much different from the concept of a Table in a traditional database. So it is not unexpected for it to allow for join operations. A join operation can be defined between two KStreams, two KTables, and between a KTable and a KStream. In FenceX we have joined a KTable with a KStream. This means each time an event comes to the stream, a function will be called with two inputs: the event and the key:value record in the KTable with the same key as the event.

Kafka Streams also supports concepts like event time which allows for defining windowing operations. A use case could be counting the number of clicks on a link during each 5 minutes window when each click results in an event.

Chapter 6

FenceX, framework and architecture

FenceX is a stream processing system designed and implemented into microservices. It can also be identified as a reactive distributed system that is applying event-driven architecture for the communication of its modules. These four big software architectural names are pretty much the same thing in the context of FenceX with their point of view being different. To avoid repeating the same vocabulary over and over in this document, we use the terminology specific to each of these realms interchangeably more precisely when applicable. The listing below illustrates words that refer to similar things but originate from different terminologies.

- **Microservice:** microservice, service, operation, event processor, processor, subscriber, publisher, module and subsystem.
- **Instance:** instance, task.

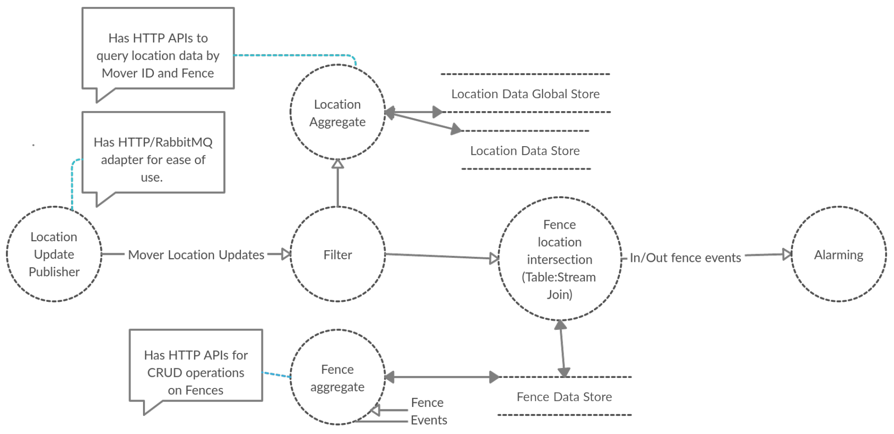
6.1 System overview

As you can see in the figure 6.1, FenceX has 6 operators. Some of them are stateful and backed by a state store and some of them are stateless. The exact details are as below in relation to state:

- **Stateful:** location-aggregate, fence-aggregate, location-fence-intersection
- **Stateless:** location-update-publisher (source), filter

location-update-publisher plays the role of source in the FenceX stream processing pipeline. It has HTTP and RabbitMQ[36] adaptors which are used by movers. Movers send messages or HTTP requests in order to report their

Figure 6.1: Logical data flow graph of FenceX


Has HTTP APIs to query location data by Mover ID and Fence
Location Aggregate
Location Data Global Store
Location Data Store
Has HTTP/RabbitMQ adapter for ease of use.
Location Update Publisher
Mover Location Updates
Filter
Fence location intersection (Table:Stream Join)
In/Out fence events
Alarming
Has HTTP APIs for CRUD operations on Fences
Fence aggregate
Fence Events
Fence Data Store

latest location. Location-update-publisher turns each report into an event and publishes it into a Kafka topic called **location-updates**.

filter , as the name suggests, is responsible for avoiding not desired location updates to find their way further down the stream. Checks can be against null values or invalid latitude/longitude.

location-aggregate is responsible for keeping a view of the latest location of each mover by subscribing to the stream of location updates coming out of the filter. It has a local database of mover-locations that applies a geospatial index to the data. As a result, the query by fence becomes possible. So location-aggregate is responsible for answering queries both by mover id and fence (geospatial shape).

fence-aggregate is responsible to keep track of predefined fences (for real time intersection scenarios). It exposes HTTP APIs for CRUD operations. The fences are kept in a temporal KTable (Kafka streams notion of table) and backed by a durable Kafka topic. In other words, each operation (CRUD) on each fence is initially a Kafka event which is later processed by fence-aggregate into a fence. The producer of those events is also fence-aggregate. In more technical terms, fence-aggregate is using event-sourcing architecture internally.

fence-location-intersection (join) has a view of predefined fences (managed by fence-aggregate) in the shape of a key:value table. Key is the id of the mover for which the fence (value) is defined. fence-location-intersection is a join between that table and stream of location updates coming out of the filter. Those location updates also have a key:value form. Key is the mover id and value is the reported location. And finally, the join operation is to check if the fence in the table contains the location in the update. The result of this intersection will be events like mover X is (not) in the predefined fence.

alarming is responsible for responding to the events coming out of fence-location-intersection. The implementation of this operator is not within the boundaries of this thesis. So we won't describe it in detail.

6.1.1 Push and Pull legs

As described previously, FenceX provides two styles of geofencing, on-demand and real time.

Sending HTTP queries with a geospatial fence (query by fence) to FenceX is of on-demand nature. This means whenever a client demands a geofencing operation, they send an HTTP query. In this style, fences are very dynamic. The fence will be checked against the whole database of mover locations. **location-aggregate** is responsible for this style of fencing. Since sending a query to a system is like pulling data out of it, the parts of FenceX making this possible are considered pull legs. So **location-aggregate** is pull leg of FenceX.

Calculating fence-point intersection for every location report, on the other hand, is of real time nature. For each new location report, the system pushes an intersection calculation result. **fence-location-intersection and fence-aggregate** carry the burden of real time fencing and they are the push leg of FenceX. In this style, the defined fences are more static and do not change frequently over time.

From now on, we will use the notions of push and pull leg frequently especially in the evaluation phase, Each of these legs have their own characteristics and require individual attention.

6.2 Physical data flow graph

Figure 6.2: Physical data flow graph of FenceX

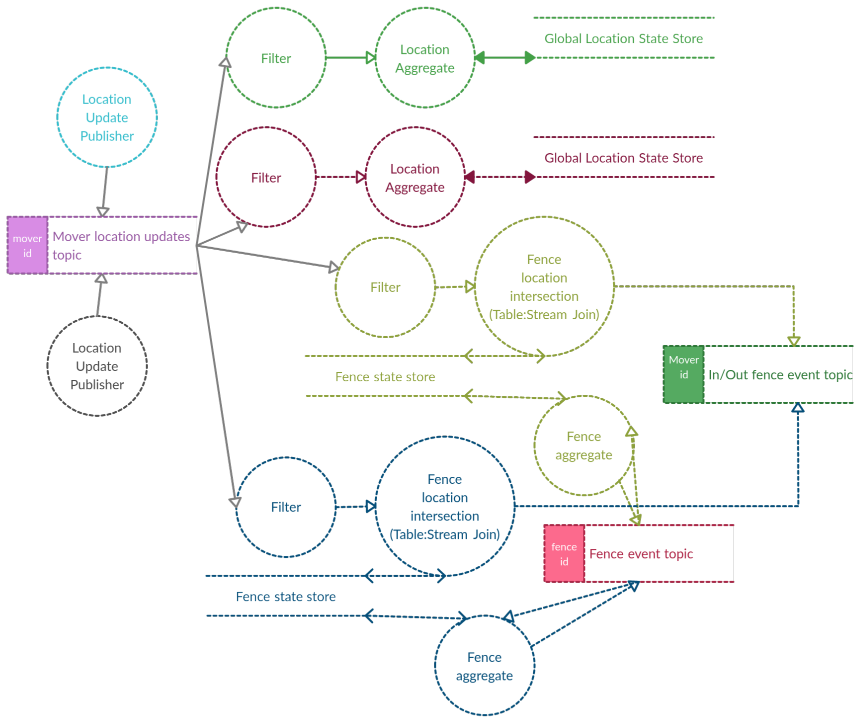

Filter
Location Aggregate
Global Location State Store
Location Update Publisher
Filter
Location Aggregate
Global Location State Store
mover id
Mover location updates topic
Filter
Fence location intersection (Table:Stream Join)
Location Update Publisher
Fence state store
Mover id
In/Out fence event topic
Fence aggregate
Filter
Fence location intersection (Table:Stream Join)
fence id
Fence event topic
Fence state store
Fence aggregate

Figure 6.2 is a physical data flow graph, illustrating how possibly FenceX components can be deployed as instances of microservices containing stream processing pipeline tasks. Each group of tasks that has a similar color is one unit of deployment, an instance of a microservice.

The combination of tasks below will be deployed as one microservice:

- location-update-publisher alone makes a microservice called **location-update-publisher**.
- filter and location-aggregate together make a microservice called **location-aggregate**. This the pull leg of FenceX.
- filter, fence-aggregate and fence-location-intersection together make a microservice called **realtime-fencing**. This the push leg of FenceX.

You can see that each microservice exists two times in 6.2 differed by color. It means FenceX deploys more than one instance of each microservice for scalability and resiliency purposes. Such characteristics will be discussed in detail later.

6.3 Inter microservices communication

At the moment there is no need for HTTP communication between FenceX microservices. However, they can find each other's IP by querying Consul[37] on the fly.

The communication style of microservices in FenceX is asynchronous and event-driven. This means there are Kafka topics available for publishing events and subscribing to them. These topics are partitioned so each instance of the microservice will subscribe to a sub-set of partitions. This is a load-sharing strategy helping with scalability. Figure 6.2 illustrates involved (Kafka) topics in the pipeline in addition to the state stores(database) attached to/used by each task. Fundamentally each of those state stores is backed by a topic (changelog).

6.4 Microservices internal design and implentation

FenceX microservices can be described in details as following:

6.4.1 location-update-publisher

Location-update-publisher is a simple Kafka publisher. A source in the stream processing pipeline. It can be exposed to the outside world using both RabbitMQ and HTTP adapters. The adapters receive location reports. Reports will be mapped to the event schema that other microservices understand and finally the event will be published into a topic called **location-updates**. This is a stateless service, so it can be scaled out very easily. Since the functionality of this service is simple, each instance does not need to be rich in resources.

6.4.2 location-aggregate

Location-aggregate is a Kafka Streams application. Its internal stream starts with subscribing to **location-updates**, continues with a filter, and finally aggregated into a Global KTable which is backed by an in-memory H2 database. This means once a location report received by location-aggregate, it will be first checked for validity and then saved into an in-memory co-located database. The save is an update operation most of the time. Updating the latest captured location for a mover. H2 provides geospatial indexing. Upon each update, the index will be updated as well, so that the database will become ready for queries (query by fence). By database though, we mean a single big table here.

Co-located database

Usually, in information systems, databases have their own cluster and deployed on machines other than the one on which applications are deployed. This is done in order to abstract away the availability and scalability challenges of data(base) from developers. The disadvantage is that accessing such a database can only happen over a network that is an I/O operation with relatively high latency. Since we aimed for high throughput (low latency) geofencing in FenceX, we decided to use a databases process deployed tightly together with location-aggregate. The life cycle of this database is managed by location-aggregate. In fact, it's a library within the application. In summary, we used an embedded database. The direct result is that the network call for accessing this database is only within the range of localhost, so the latency is minimum.

In memory database

Usually, database systems are configured to use the disk as storage which allows them to grow very big. The downside with this approach is the I/O intensiveness of interacting with a disk which means latency. Although very high throughput SSD hard drives exist, not only their price tag is high but also they are still not as fast as memory. Since our goal is minimizing I/O latency, the H2 database not only is configured to use memory as storage but also using it in an asynchronous non-blocking manner (nioMemFS). This means H2 will act as if a file system exists on the memory and will access it using Java NIO [38] technology.

Inconsistency

Using database systems maintaining their own dedicated cluster means no need for being worry about the possible inconsistency problems. The nodes in the cluster need to be consistent after all, even if only they act as cold idle replicas. Co-located in-memory databases, on the other hand, usually do not have any consideration for replication and/or consistency. Since we want to scale out the pull leg of FenceX, facing inconsistency issues is inevitable. So we need a way to make each instance of location-aggregate (all of the H2 instances) have the same data, the same image of the world. H2 provides an out-of-the-box solution for keeping its instances consistent, even when configured to be embedded and in memory. However, after some trial and error, we did not find it reliable. So it is up to us to keep the instances of location-aggregate consistent. Kafka Streams Global KTables are state stores that have the same data regardless of instance. They are equivalent to Kafka subscribers each having a different group name; which means they get events from all partitions. Normal KTables only keep a subset of world image which corresponds to the partitions they listen to.

So if we put the H2 database underneath a Global KTable, Kafka Streams out of the box keeps the H2 instances consistent. **Eventually consistent** However! Eventually consistent means the consistency of data among instances of location-aggregate is not of ACID nature. Once a location update is published, the time taking for each instance to capture the location update and change its state accordingly varies from an instance to another. This time is a factor of the current load on each instance.

Each Kafka Streams state store is backed by a topic called changelog. This topic is durable and is saved on disk by Kafka. Each Kafka Streams application attached to such a store, subscribe to that topic once it's up and running and populate its KTables. That topic contains all changes that have happened to the state of the application, the data in the KTable. So in case, something happens to one of the instances (or all of them for that matter), after a restart or bringing a new fresh instance up, the instance recover/creates it own images of the world just by subscribing to change log topic. The subscription can be from the last seen offset or from offset zero, depending on the state of the underlying store. Sometimes the stores are backed by a database using a file system as storage.

So finally we resolved the rather complex issue of inconsistency in FenceX. Also, we covered some aspects of the work which makes FenceX able to deliver high throughput.

6.4.3 realtime-fencing

Realtime-fencing is a Kafka Streams application responsible for keeping track of fences, receiving location updates and joining them. The join operation is a fence-point intersection. It has two internal streams. One for processing location updates, and the other is for aggregating fences.

Event-sourcing

One of the internal streams starts with subscribing to **fence_event_log** topic. The publisher of this topic is realtime-fencing itself. It exposes two HTTP APIs for CRUD operations on fences (only create is implemented for now). For instance, when create fence API is called, apart from possible initial validations, the only thing that happens is that a (Kafka) event will be published to the **fence_event_log** topic. **fenceCreatedEvent** in this case. No fence is created yet or saved anywhere in the system. However, that's how an event-sourcing based application works. The stream that starts with subscribing to **fence_event_log** topic, receives those events and aggregates them into a Kafka

Streams KTable. A key:value table in which key is the ID of the mover for whom the fence is defined; value is the WKT string describing the geospatial shape of the fence. Any operation on the fences should first be published as an event and later captured by realtime-fencing up and running instances. The event will be processed, and a suitable change will happen on the data in the related state store. This approach (event-sourcing) is one of the special styles of event-driven architectures which feels very natural when implemented using Kafka Streams. Using this architecture leads to eventual consistency which is usually the case with all event-based systems. The benefit is the huge potential for scalability. Also, the source of truth in such systems is the event_log topic rather than the database (state store). Whatever happens to the whole application, the state can be easily recovered by bringing a new instance of the application up and making it subscribe to event_log topic from offset zero. Or in case of a bug in business logic that resulted in a wrong state, a correct state can be rebuilt after fixing the bug. These benefits come from the fact that the only durable data in this architecture are immutable facts, the events that have happened. It's worthy to mention that the event_log topic can be used as an activity log for monitoring and debugging purposes.

In realtime-fencing the KTable keeping the fences is backed by a changelog style topic similar to what we described about location-aggregate. The difference here however is that the KTable keeping track of fences is not global. So each instance of realtime-fencing has a subset of defined fences corresponding to the partitions of Kafka topic listening to. Since load balancers do not know what instances are responsible for what fences at any given point in time, in order to query this KTable, a select all query, for example, each instance that receives the query, might gather data from other instances. Such data gathering should be party implemented by us using some related features of Kafka Streams. In case queries more complex than select one or select all were required, this approach would not work. Mainly because the KTable is only a simple key:value table.

So far we described one of the internal streams of realtime-fencing, the one related to aggregating fences.

Join stream

The other internal stream of realtime-fencing starts with subscribing to **location-updates**, continues with a filter and finally a join operation. A join between this stream of location updates and the table of fences. The stream is a key:value stream. Key is the mover ID for whom the value, location report, is published. A join between this stream and the fence KTable means whenever a new location update is received for a mover, the fence defined for that mover (if exists) will be fetched. Now a function can be called with these inputs: ID of mover, the defined fence, and the newly received location update. In our case implementation of the function is calculating if the new location is within the defined fence, A geospatial point-shape intersection. Which we call point-fence intersection. The output of function should be a Key:Value pair with the mover ID being the key and the value is the result of intersection. So after the join, we end up with a stream of moverId:intersection-status updates. At the moment we have not continued the stream but ideally, the status should be published into another Kafka topic so that other operations down the stream processing pipeline can use it, like alarming.

6.4.4 Facts about FenceX microservices

The points listed below are technical information which is true for all of the FenceX microservices.

- They are implemented using Java 15.
- They are implemented with extensive help from Spring family frameworks and libraries like Spring boot, Spring data, Spring webflux, and Spring cloud.
- They are packaged as layered docker images, so the deployment artifacts of FenceX are docker images.
- They expose some HTTP APIs implemented using Spring webflux which is on top of the project reactor. As a result, they are reactive applications meaning they process HTTP requests in an asynchronous and non -blocking manner.
- They expose an HTTP API for fetching metrics about their resource usage, JVM status, and geofencing operations.

Chapter 7

FenceX characteristics

In this chapter, we describe how different non-functional requirements of the system has been achieved based on the design and architectural decisions.

7.1 Availability

Availability has many aspects in software development specifically when it comes to distributed systems. The main practice to achieve high availability is to scale out meaning deploying more than one instance of the application, application sub-components for that matter. In the case of microservices, more than one instance of each microservice should be deployed. In the same way, deploying more than one task of each operation makes sense in the stream processing realm. The benefits of scaling out in terms of availability are as following:

- **Operational availability:** If a subset of deployed services stop responding for any reason, the rest of up and running instances can take over the work.
- **Data availability:** Data can be replicated over different instances of services so if something goes wrong with one of the instances, the data will not get compromised. The process of data can even be continued without any special change being noticed in the overall behavior of system.
- **Availability under varying load:** Will be explained as scalability in the next section.

When it comes to FenceX, it's easily possible to deploy as many instances of microservice as required by non-functional boundaries of corresponding users and use cases. We used a full-fledged production level container orchestration tool called Nomad[39] for managing the deployment of FenceX microservices.

7.1.1 Back pressure

Back pressure[1], in simple terms, is the ability of the responder to control the flow of incoming requests. If microservice A is sending an ongoing load of requests to microservice B, if back pressure is enabled for their communication solution, at each point in time, B can take as many requests as it can process and respond at that moment. As a result, B won't get overwhelmed and communication will keep flowing smoothly.

HTTP does not support back pressure, so in the case of A and B, A can send a big load of requests to B and essentially bring it down. One of the aspects of a reactive system is back pressure. A component in a reactive system should be able to somehow communicate that it's under pressure instead of catastrophically collapse.

In FenceX, back pressure comes out of the box by using Kafka topics (publish-subscribe pattern) for communicating between microservices. Each instance of microservices (task) takes one event at a time from the kafka topic, process it and then it takes the next event. Events in Kafka topics are also durable so there is no rush for the event processors. As a result, event processing in FenceX will work as a smooth flow and microservices can control their incoming load.

In the case of Kafka, due to support for the acknowledgment of being done with processing an event, even if something happens during processing of an event, the event processor will get that event back from the topic once it's ready again.

7.2 Scalability

When is a software system is scalable in simple terms means that the system can handle different sizes and frequencies of input load successfully. There are different elements in a software system that can grow:

- Data size in databases
- API call (incoming HTTP requests for example)
- Input rate in stream processing systems (Rate at which events arrive)
- Number of parallel interdependent operations
- Number of parallel independent operations
- Number of active threads in thread-based systems
- Queue size (number of events in Kafka topic for example)
- Stack size
- Number of users using the system at the same time
- Number of system subcomponents (number of microservices or number of stream processing operations)

- Number of people can be working on the development of the system at the same time (organizational scalability)

A scalable software system keeps responding/processing successfully with reasonable throughput and latency even if any (or all) of the factors mentioned above grow hugely in size or in rate. The growth can happen step by step over a window of time or like a sudden sock in a moment.

7.2.1 Data scalability

In FenceX, data scalability is limited by the number of rows that an H2 in-memory table can keep. Which is 2^{64}. This is big enough to not be a limit even for the global KTable in location-aggregate. The other stateful operation is fence-aggregation which is highly scalable in terms of data due to the fact that the corresponding data store, is partitioned over available instances of realtime-fencing. So if defined fences grow in size, horizontally scaling realtime-fencing will just deal with it.

The maximum number of open transactions each H2 instance can handle is 65535. It is big enough as well to not limit the scalability of FenceX especially when combined with scale out strategy.

7.2.2 Incoming HTTP requests

All HTTP APIs in FenceX are implemented using reactive stack (instead of thread-based servlet stack) using Spring webflux library. Reactive in this context means asynchronous non-blocking I/O which allows each instance to process much more HTTP requests providing having the same resources. In simple terms, no CPU cycles will be wasted for/while WAITING for I/O results.

Also, deploying more than one instance of each microservice allows for load distribution. A load balancing solution (client-side load balancing in case of FenceX) distributes the HTTP calls over up and running instances of the target microservice. So more requests can be responded at the same time. This is called scaling out which is opposite to scale up. When scaling up, we increase the resources available to each component, to each instance. Scaling up (scaling vertically) is expensive and does not provide other types of scalability, organizational scalability for example.

7.2.3 Stream processing systems input rate

In event-driven systems specifically, stream processing ones, the input is incoming events and messages waiting to be processed. Location reports in case of FenceX. For FenceX to be able to handle high rates of input, the events are published into Kafka topics. Kafka topics are partitioned. Each task (microservice instance) once is up, healthy, and ready for processing, subscribes to a subset of those partitions. This means, the event processing load is distributed among available tasks. This is called **data parallelism** in stream processing.

When the same tasks process different data. Applying this, which is basically scaling out, allows for dealing with high rates of input. Just by deploying more instances of microservices, we end up having more capacity for processing incoming events. It is by the way very important to have more partitions than the number of deployed instances. Otherwise, we will end up with idle tasks.

7.2.4 Design and organizational complexity

Over time as the system evolves and provides more functionality, keeping all the codes in one service brings various non-technical problems. In short, we will end up with one big service which is very hard to maintain, deploy, debug, build, reason about and most importantly change. In such a service, everything is coupled together due to high reliance on code reuse. The alternative is to break down this monolith into different microservices each of them responsible for doing mainly one thing. They will have their own database, build process, deployment process, and monitoring dashboards. These microservices ideally share nothing and communicate ONLY using messages (reactive). Such decoupled microservices can be introduced over time to the whole system whenever new requirements require so. This way development and design of new features is independent of others; so the best fitting internal architecture, language, database, and code style can be selected for that specific service.

Such independence also allows multiple people, multiple teams for that matter, to work in parallel on the system. There is a practical limit on how many people can work at the same time on the same code base. When system code is distributed over different codebases, a lot of people can work at the same time on the system and deliver value.

7.3 Throughput

For FenceX throughput of two types of operations matters the most. Throughput of HTTP requests for querying the database of locations against a fence (pull leg throughput). And, the throughput of fence-point intersections (push leg throughput). Throughput is usually closely related to scalability since high throughput is achievable only under high input load.

7.3.1 Pull leg throughput

Is the number of queries (by fence) that FenceX can respond to successfully in a unit of time (seconds). This throughput is a factor of the number of queries one can send to FenceX which is the input rate in the pull leg context. Another effective factor is the latency of each query. This latency depends on the amount of I/O involved, geospatial index algorithm, network access latency, CPU speed, and the number of parallel queries. In FenecX we minimized the network latency and I/O latency by using an embedded co-located in-memory database. Also using reactive programming for implementing HTTP APIs, removes the latency that waiting for blocking I/O operations imposes.

Also, using microservices architecture and scaling out the pull leg instances leads to more capacity for responding to simultaneous queries.

We calculate the pull leg throughput by increasing a counter every time FenceX responds to a query successfully. This counter is a metric gathered every now and again by a time-series database called Prometheus[40].

7.3.2 Push leg throughput

Is the number of fence-point intersections that FenceX can calculate in a unit of time (seconds). This throughput depends on the input rate (rate of incoming location updates) and how many instances of push leg are up and running. By scaling the realtime-fencing microservice out, we allow for parallel intersections to happen (data parallelism). Which means FenceX can calculate more intersections per unit of time. The trigger for each intersection operation is a location update which is received as a Kafka event coming through Kafka topic partitions. So theoretically number of partitions in the Kafka topic is also a factor.

We calculate the push leg throughput by increasing a counter every time FenceX calculates a fence-point intersection. This counter is a metric gathered every now and again by a time-series database called Prometheus.

Chapter 8

System deployment

The deployment configuration for FenceX microservices in production depends on the exact needs of the business. It is also always limited to the hardware resources available, like any other software. During the evaluation of FenceX for this thesis, we had access to 4 virtual machines with the following resources available to them.

Server	CPU CORES	RAM(GB)	Main responsibility
server1	8	16	Master keeper
server2	4	8	Nomad worker node
server3	4	8	Nomad worker node
server4	4	16	Bench-marking

Table 8.1: Virtual machines used in evaluation of FenceX and their overall responsibility

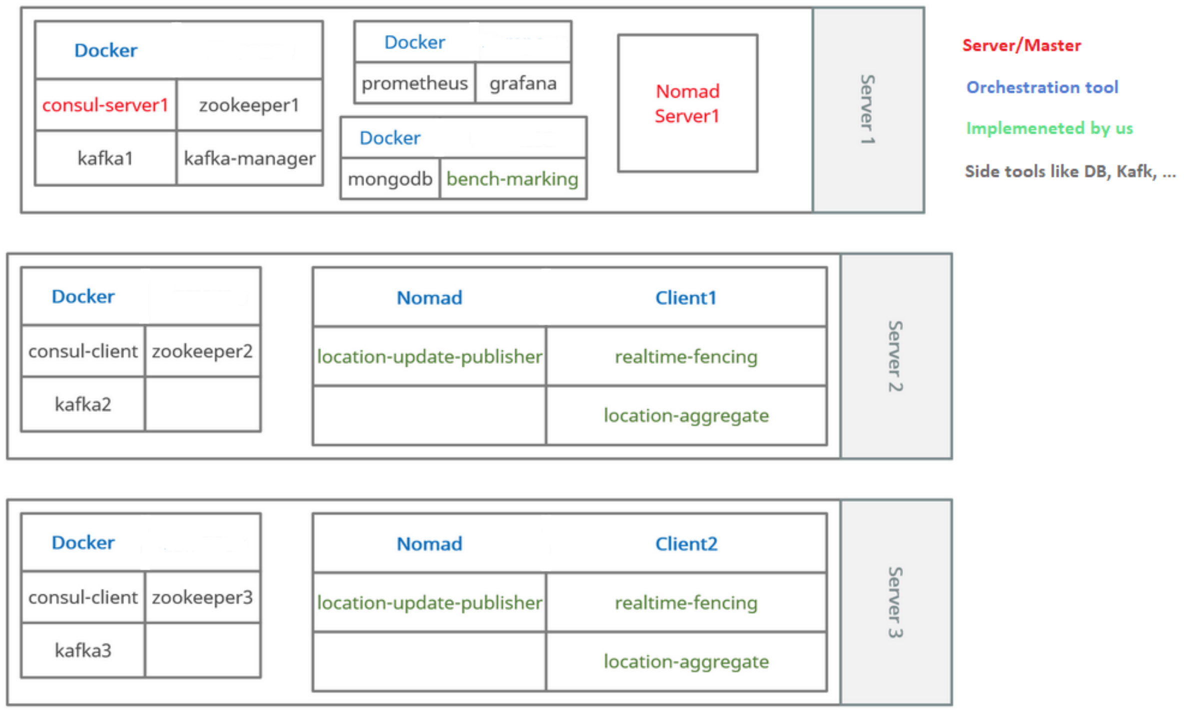
Figure 8.1 illustrates how FenceX deployment looks from the infrastructure point of view in a evaluation scenario. The only difference compared to other possible evaluation scenarios is the number of deployed instances of FenceX microservices, realtime-fencing for example.

Figure 8.1 does not include server4 since that server only runs an instance of the bench-marking application which is not part of the FenceX solution package. Nevertheless, we have deployed another instance of bench-marking into server1 as well; since getting high throughput out of bench-marking applications is very expensive in terms of resources.

Tools like Kafka and Zookeeper are deployed as docker containers to keep the setup phase of evaluation easier. In production, they are usually deployed directly on separate hosts. However, for FenceX evaluation, we have used docker-compose on top of docker to manage the life cycle of docker containers in an easy and convenient way.

It is also important to note that FenceX had 4 major Kafka topics. *fence_event_log, location-aggregate-mover-in-memory-state-store-changelog, mover-position-updates,*

Figure 8.1: FenceX infrastructure setup for evaluation



realtime-fencing-fence-state-store-changelog. All of these topics were configured to have 3 replicas and 12 partitions. These numbers are selected corresponding to the overall available resources. The process of selecting them correctly is out of the context of this thesis. The important factor however is to have a high enough replication factor to achieve availability (3 at least) and high enough number of partitions to avoid idle subscribers.

In terms of bandwidth, between virtual machines hosting FenceX infrastructure during evaluation, in both directions we observed numbers more than 3Gbps. However, since the process of producing load was computationally expensive, we did not even get close to saturating the available bandwidth.

After preparing the infrastructure, Nomad calculated available resources to the applications that will be deployed into worker nodes as 16670 MHz of computation power and 16 GB of memory.

Chapter 9

Experiments and validation

In this chapter, we explain the process and results of evaluating FenceX. This process consists of many experiments with different setups and goals. Each experiment had a preparation phase, execution, observation, and finally discussion.

9.1 Bench-marking application

Before diving into the test cases, it is important to highlight how bench-marking application works and what functionalities it provides. It is a microservice similar to other services in FenceX. It registers itself into Consul and has HTTP APIs. It is based on spring-webflux[41] reactive stack in order to handle I/O better. The role of bench-marking is to imitate the behavior of real-world mover location report publishers, smartwatches, or car GPS modules for example.

bench-marking has a dataset of real taxi trips. Each trip has an (anonymized) ID, a set of location reports (points), a geospatial circle (fence) drawn around one of those points. The points are in form of coordinates (latitude, longitude) and the fences are WKT strings. Each trip has an average of 17 points and in total around 500 trips are in the database.

bench-marking can send HTTP requests to realtime-fencing and define fences for movers.

bench-marking can send HTTP requests to location-update-publisher and report coordinates for each trip. The result in FenceX will be a stream of location updates published into the pipeline.

bench-marking can send HTTP requests to location-aggregate and query its location database against the fence defined for each trip.

bench-marking can do operations mentioned above for all the trips repeatedly very fast in order to push a stressful load of inputs into the FenceX pipeline.

It can also keep the size of the load low while keeping it ongoing (non-stop) over time.

It appeared that producing stressful loads of HTTP request or ongoing flows of them is a very expensive operation in terms of memory and CPU usage. Mostly during our evaluations, the bottleneck to throughput was our input rate.

9.2 System throughput

In this section, we go over the experiments we have conducted in order to not only stress-test FenceX but also calculate the peak throughput based on available resources. Throughput is a direct factor of input rate both for on-demand pull style operations and realtime push style ones. For on-demand style, the input rate is the number of HTTP requests (query by fence) per second that the system receives. For realtime operations, the input rate is the number of location reports that arrive at the system per second. For each setup, exists a threshold for input rate above which increasing input rate won't result in throughput going higher.

In order to calculate the throughput in each experiment, we use a combination of tools, Prometheus and Grafana[42]. Prometheus is a time series database that pulls different types of metrics out of each of FenceX microservices on a frequent basis. Grafana is a visualization tool that reads metric data from Prometheus and illustrates it into a variety of graphs. The metrics can be general like CPU usage or the number of JVM threads. They can also be custom to each application like the number of fence-point intersection calculations. Using this setup, we created realtime graphs that show the pull and push throughput of FenceX in addition to other information like push and pull input rate. Thanks to that, we do not need to calculate the throughput after each experiment. We can just have a look at the corresponding graphs during the execution of test scenarios in order to see the realtime value of throughput and input rate.

During evaluation for both legs of FenceX, we expect the highest throughput that we will observe to be higher than corresponding throughput achieved by [18] and [17]. We will provide a more detailed comparison at the end of this section.

9.2.1 Push leg

Push throughput evaluation scenario:

1- Define fences for movers using all the trips in the bench-marking application database.

2- Send a shocking stream of location reports into the FenceX.

3- Check the throughput.

4- Repeat the experiment after changing the deployment configuration and input rate pattern. (Maybe add more instances or add more CPU) until finding peak throughput and/or a possible upper bound for throughput.

Experiment 1 and 2: Push leg throughput

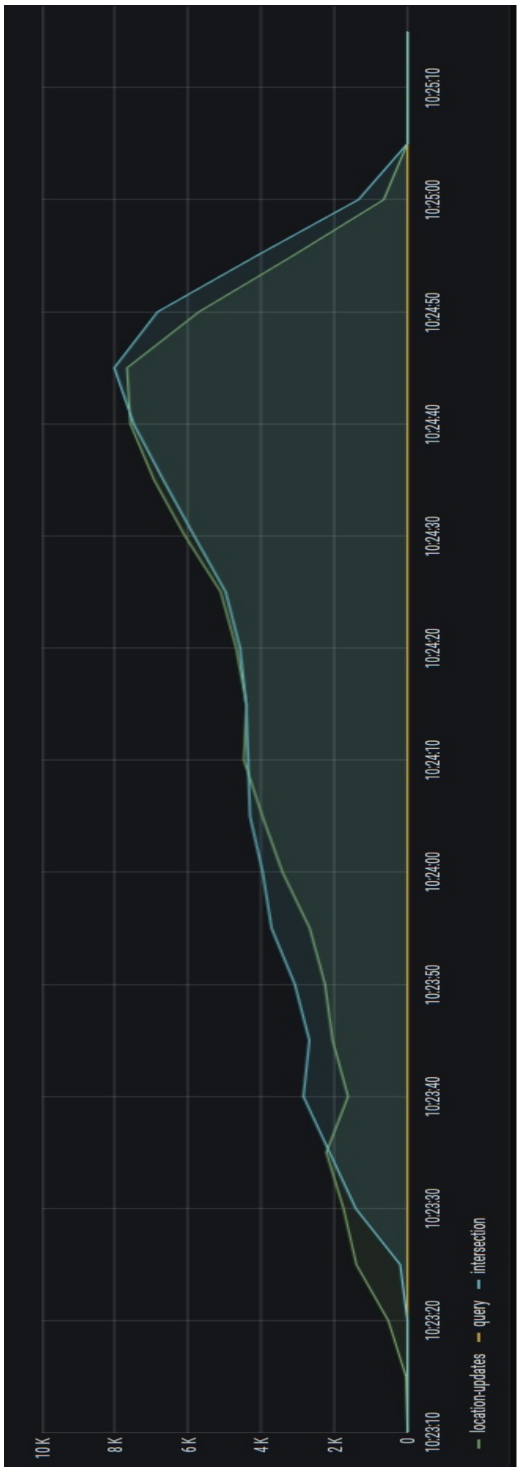
Application	number of instances	CPU(MHz)	RAM(MB)
location-update-publisher	4	500	500
location-aggregate	-	-	-
realtime-fencing	4	700	800

Table 9.1: Experiment 1 deployment view

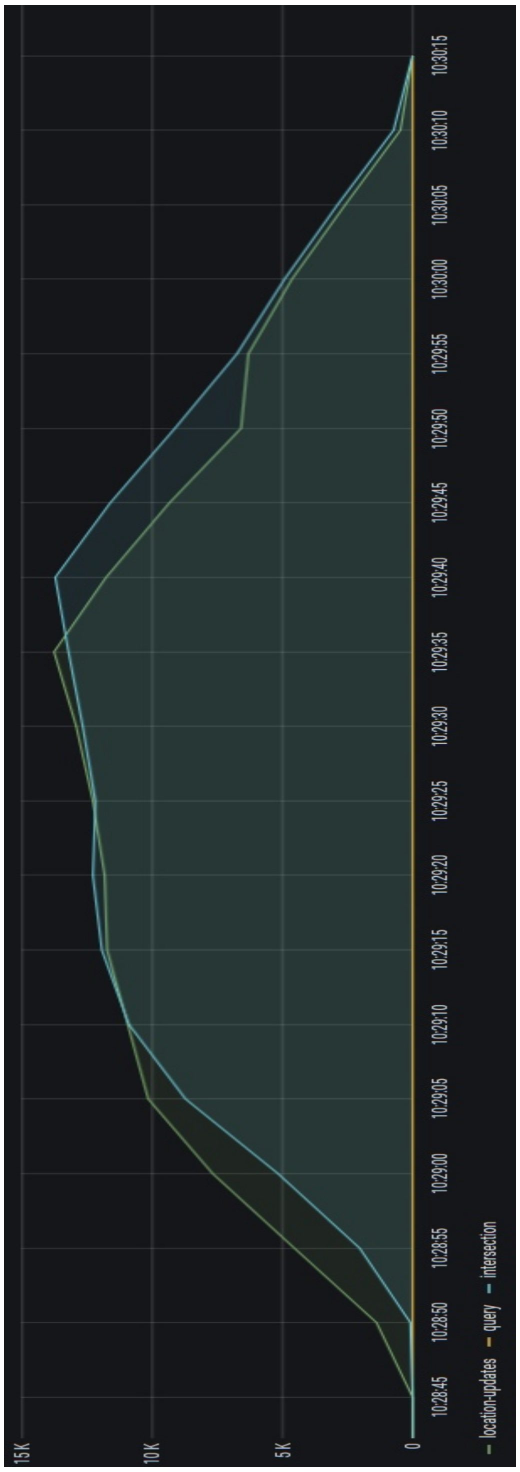
Application	number of instances	CPU(MHz)	RAM(MB)
location-update-publisher	5	400	700
location-aggregate	-	-	-
realtime-fencing	5	400	700

Table 9.2: Experiment 2 deployment view

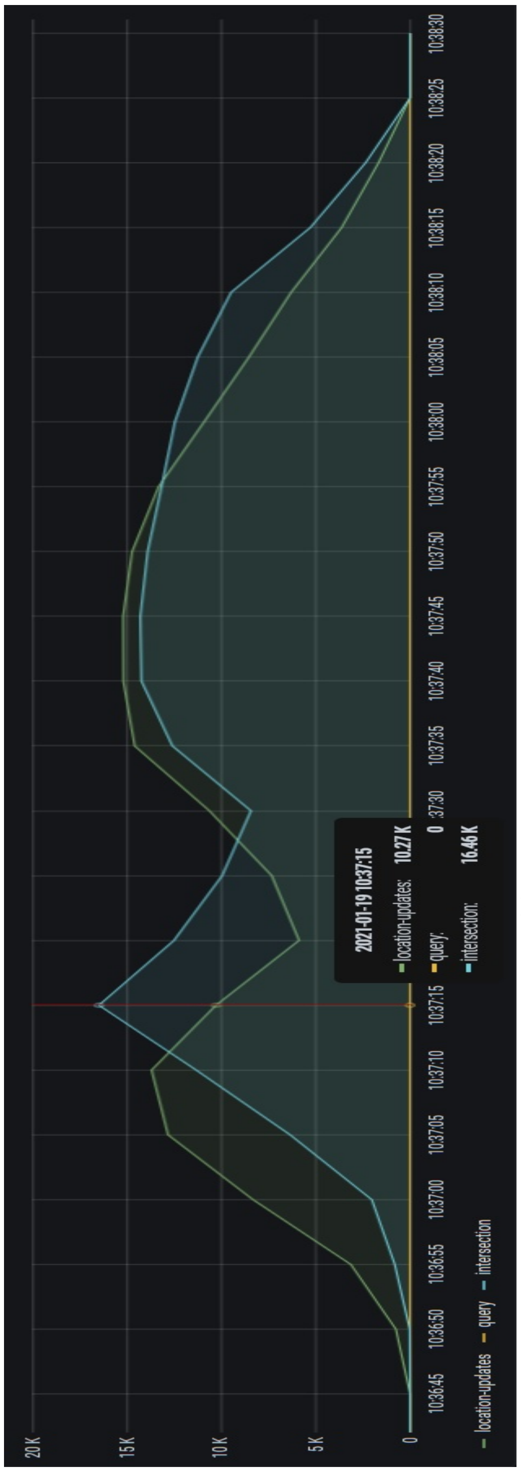
Tables 9.1 and 9.2 show, for experiments 1 and 2, how many instances of each FenceX microservices we have deployed in addition to the amount of resources we have assigned to each. The row in those tables with no values corresponds to a microservice that does not relate to experiments 1 and 2. After successful deployment, we followed the previously mentioned scenario. The following figures illustrate how the input rate and throughput have changed during each repetition of the experiment 1 and also during experiment 2. It is important to note that the pattern at which input rate has changed (which differs among repetitions and experiments) is not of direct relevance to the system throughput. So the pattern of changes in input rate and their differences can be ignored. What matters in these experiments is that if throughout **follows** the input rate pattern. Also, the highest achieved throughput is the important value we look at during these experiments.



(a) FenceX push throughput ex1-1

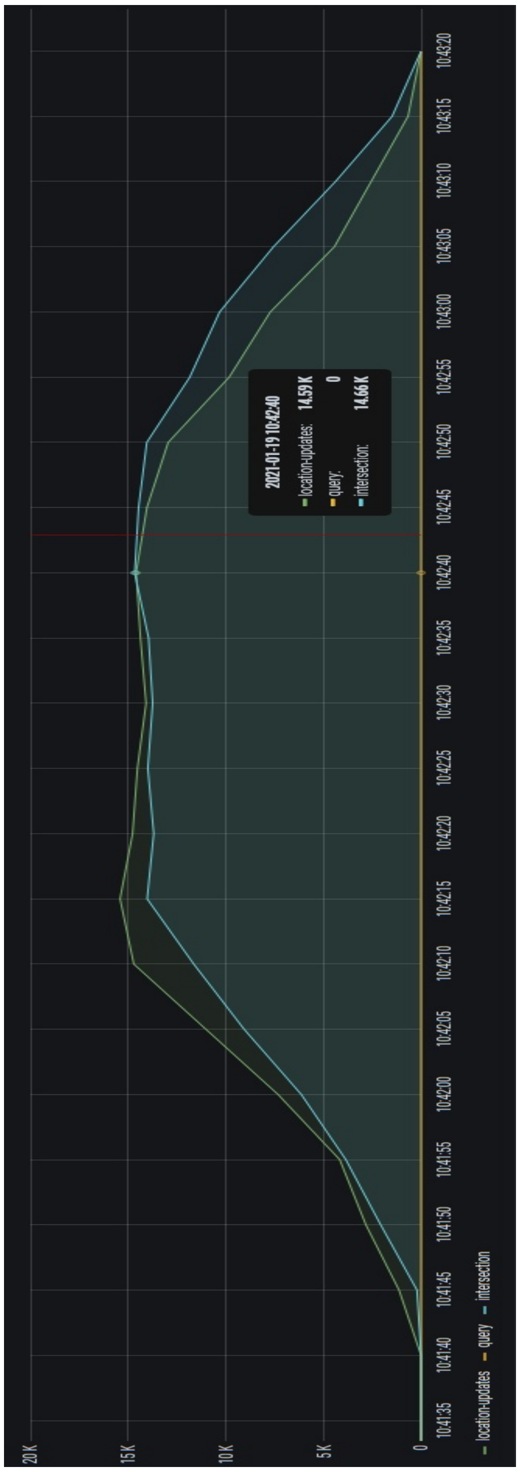


(b) FenceX push throughput ex1-2



2021-01-19 10:37:15
location-updates: 10.27 K
query: 0
intersection: 16.46 K

(a) FenceX push throughput ex1-3



2021-01-19 10:42:40
location-updates: 14.59 K
query: 0
intersection: 14.66 K

(b) FenceX push throughput ex1-4

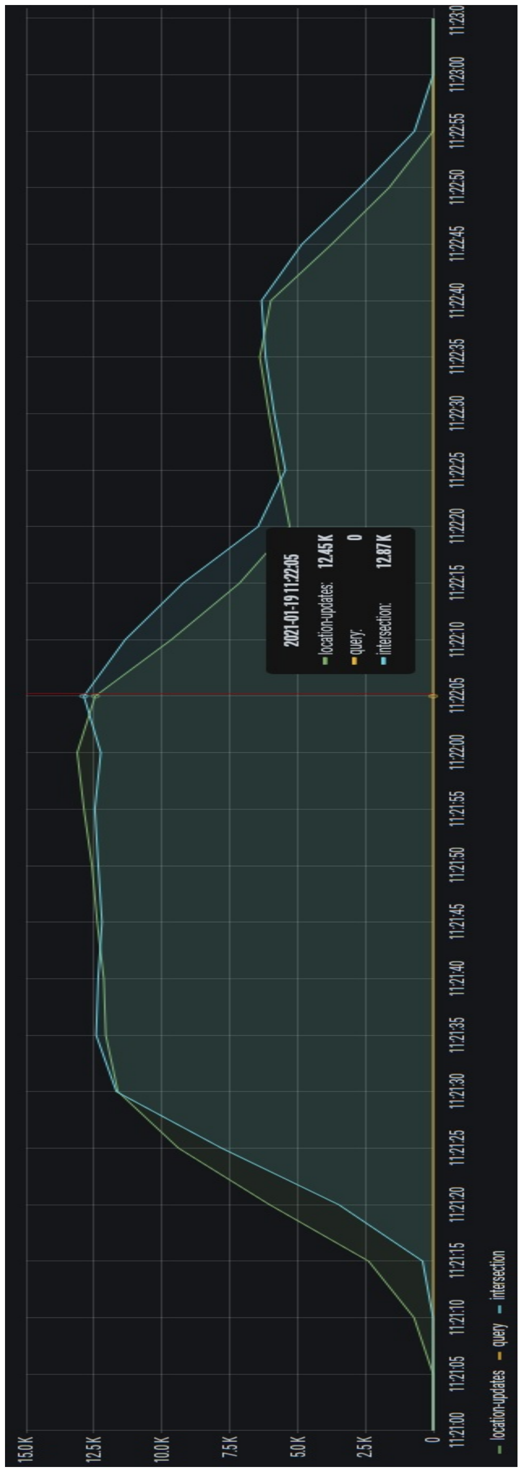


Figure 9.3: FenceX push throughput experiment 2

Discussion of push throughput experiments

Figures 9.1a, 9.1b, 9.2a, 9.2b and 9.3 illustrate input rate(location-updates per second) with green line and push throughput (number of fence-point intersection calculations per second) with the blue line. In experiment 1 we have tried different patterns of input rate and in experiment 2 we have deployed one more instance of the relevant microservice. These experiments show that when we have applied different input rates (ranging from 8k/s to 15k/s), the throughput has followed the change patterns of input rate. Also, throughput was mainly always **as high as** input rate. The highest input rate we managed to produce during these experiments was around 15K location updates per second and FenceX just managed to handle it very well with the given deployment setup.

So far the bottleneck is input rate which is limited by our physical resources available to bench-marking application. We can clearly see in the graphs that regardless of setup, push throughput (intersections/second) follows pretty much the exact pattern of changes in input rate (location updates/second). So there is no point in continuing push throughput experiments with currently available hardware power. The highest throughput observed for push leg so far, therefore, is 15k/s.

The work done in [18] is comparable to the push leg of FenceX. It has achieved a throughput of 20K/s which is higher than the highest throughput we so far managed to observe in FenceX evaluations of push leg. However, since [18] has no considerations for scalability and availability, the lower throughput of FenceX does not mean [18] out performs FenceX. It is also important to mention that so far 15k/s is the highest input rate we could produce. So we are not yet sure if 15k/s is the upper bound on the throughput of FenceX.

9.2.2 Pull leg

Pull throughput evaluation scenario:

1- Send many location updates to FenceX using all the trips in the bench-marking application database.

2- Send a shocking load of queries (query by fence) to FenceX.

3- Check the throughput.

4- Repeat the experiment after changing the deployment configuration. (Maybe add more instances or add more memory) until finding peak throughput and/or a possible upper bound on throughput.

Experiment 3 and 4: Pull leg throughput

Tables 9.3 and 9.4 show, for experiments 3 and 4, how many instances of each FenceX microservices we have deployed in addition to the amount of resources we have assigned to each. The row in those tables with no values corresponds to

Application	number of instances	CPU(MHz)	RAM(MB)
location-update-publisher	4	500	500
location-aggregate	2	2500	3200
realtime-fencing	-	-	-

Table 9.3: Experiment 3 deployment view

Application	number of instances	CPU(MHz)	RAM(MB)
location-update-publisher	4	500	500
location-aggregate	2	2700	2700
realtime-fencing	-	-	-

Table 9.4: Experiment 4 deployment view

a microservice that does not relate to experiments 3 and 4. The following figures illustrate how input rate and throughput have changed during experiments 3 and 4.

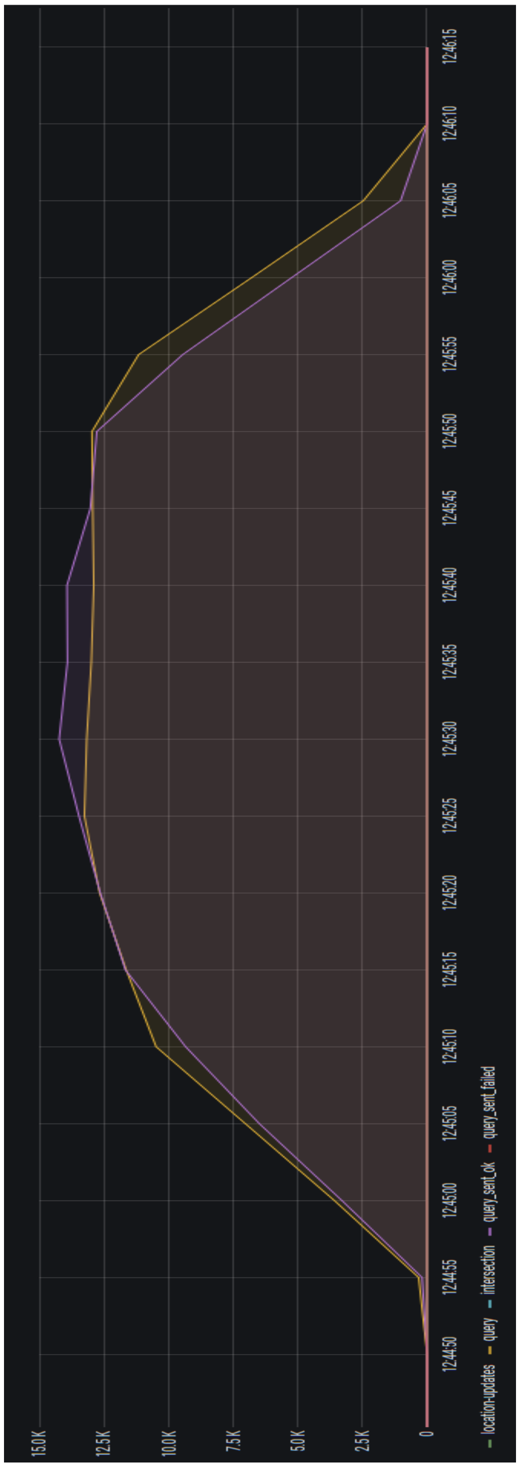
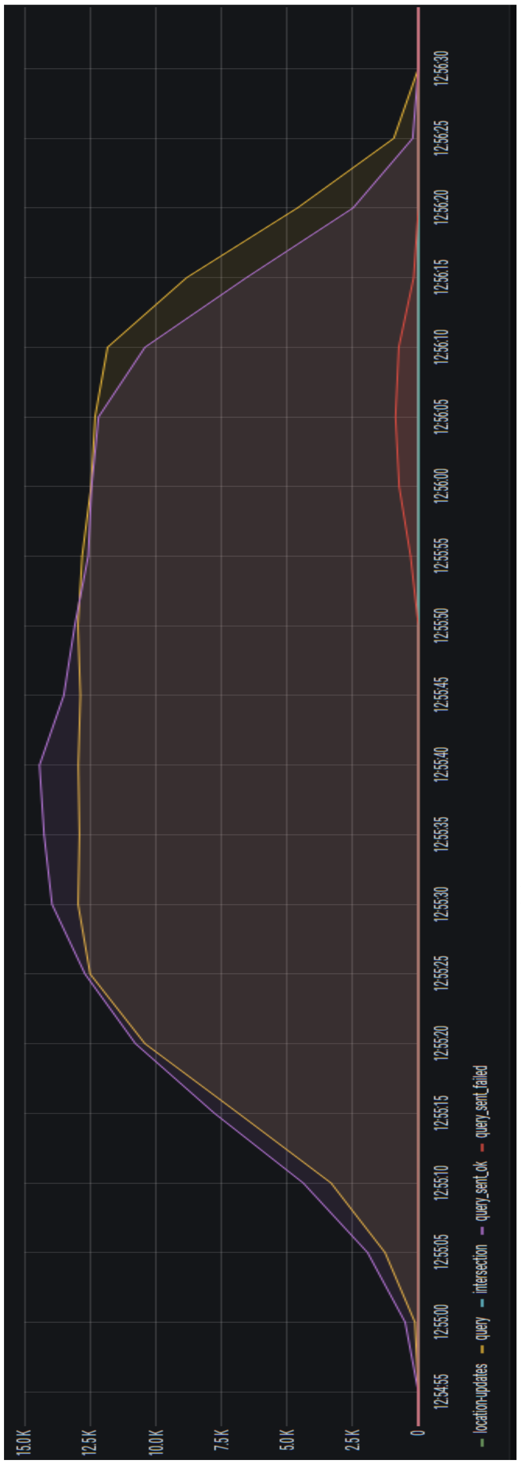
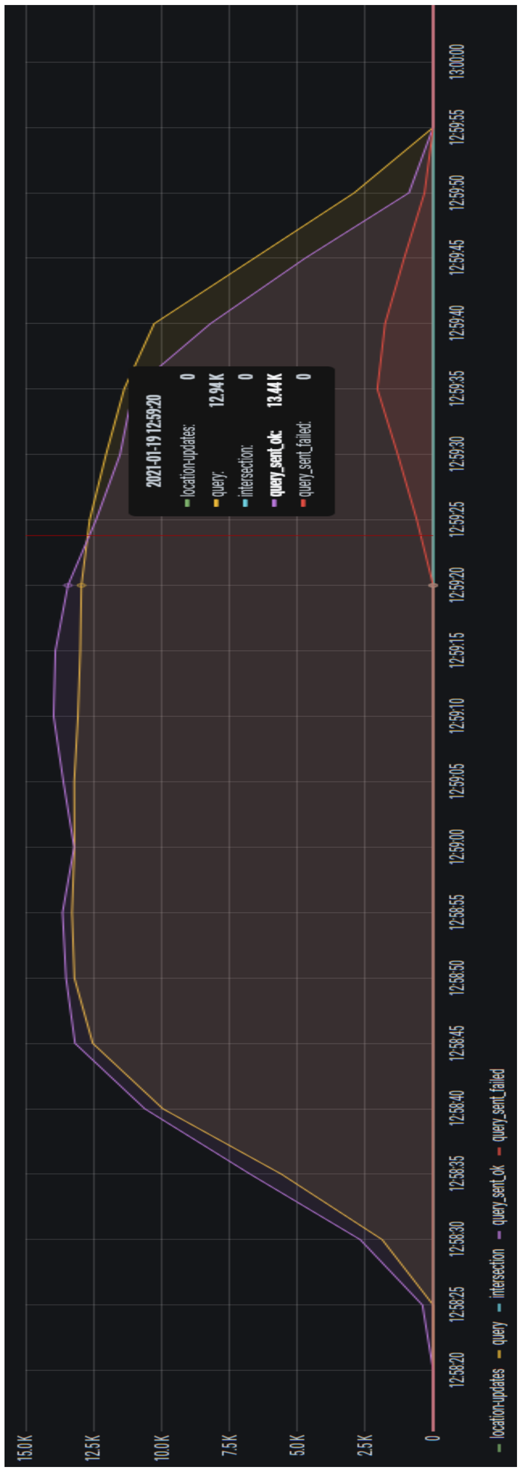


Figure 9.4: FenceX pull throughput experiment 3



(a) FenceX pull throughput ex4-1



2021-01-19 12:59:20
- location-updates: 0
- query: 12.94 K
- intersection: 0
- **query_sent_ok: 13.44 K**
- query_sent_failed: 0

(b) FenceX pull throughput ex4-2

Discussion of pull throughput experiments

Figures 9.4, 9.5a and 9.5b illustrates input rate (queries sent successfully per second) with purple line and pull throughput (number of queries answered without error) with the yellow line. In experiments 3 and 4, the input rate was around 12k/s at the highest which is handled well by 2 rather rich instances of location-aggregate. Comparing to experiment 3 with 4, we have changed the available resources to pull leg plus changing the input rate pattern. The change in the input pattern is subtle, however.

Experiments 3 and 4 show that we have failed to increase the input rate as we progressed. Although the CPU usage on location-aggregate instances peaked to 100 percent during these experiments, again the bottleneck was input rate which is limited by our physical available resources. We can clearly see in the graphs that regardless of setup, pull throughput (queries/sec) **follows** pattern of changes in input rate (queries sent/sec). So there is no point in continuing pull throughput experiments with currently available hardware.

The software introduced in [17] is comparable to pull leg of FenceX. Both of them use load sharing (scale out) and in-memory data processing to achieve high throughput and scalability.

FenceX, however, does not apply partitioning data among worker instances in the pull leg, which makes it much easier to scale out. Also, the load balancing strategy when forwarding client queries to worker instances can be as simple as round-robin in FenceX. As a result, no throughput would be lost during load balancing.

Throughput in [17] is defined as the number of location updates processed per second. Such processing involves updating an index plus a query to find a crossing geofence. When they are using the benefit of localhost communication without any data replication involved (no availability), a throughput of 250k/s has been achieved on average. However, once they introduce data replication and inter-server communication, the throughput drops to sub 10k/s range at best. Although [17] has done wonderful when no availability was in action (no replication), we do not consider it a win over FenceX. FenceX has always been tested with full replication in action. So FenceX is the winner by achieving throughput in the range of 15k/s compared to sub 10k/s.

9.3 Availability

In this section, we cover the experiments we have conducted in order to test the availability of FenceX. We evaluate availability by observing the throughput of the system under non-optimal conditions. In order to test FenceX for availability, roughly speaking, we start an ongoing load or stream of input. Once throughput becomes stable, we restart one of the instances of involved microservices. The overall expectation is that during the restart until the service becomes up and running again, the throughput goes down and afterward goes up again to the previous value. The main factor affecting the time takes until the service is up and running again is re-balancing. Kafka topics have consumer groups. Each member of a group listens to a subset of partitions of that topic. This essentially results in data parallelism. A topic can have multiple groups listening to it which essentially is the foundation of task parallelism. Re-balancing happens when one of the group members becomes unhealthy and stops subscribing to its assigned partitions. At this moment, other healthy group members should take over the dangling partitions. The process of reassigning the dangling partitions to healthy subscribers is called re-balancing. The time a possibly successful re-balancing takes depends on the number of partitions and available healthy subscribers. So the time takes until throughput goes back to the previous value is also not fixed.

Please note that after a restart, there will be two re-balancings. One once the service under restart goes down and another once it comes back up and running. If the remaining services after first re-balancing, had enough resources available, the throughput won't change that much. Especially for lower input loads.

9.3.1 Push leg

Push leg availability evaluation scenario:

1- Define fences for movers using all the trips in the bench-marking application database.

2- Start an ongoing stream of location reports toward FenceX.

3- Wait until throughput becomes stable.

4- Restart one of the instances of realtime-fencing.

5- Assert that throughput drops.

6- Wait until successful re-balancing.

7- Assert that throughput is back to the previous value.

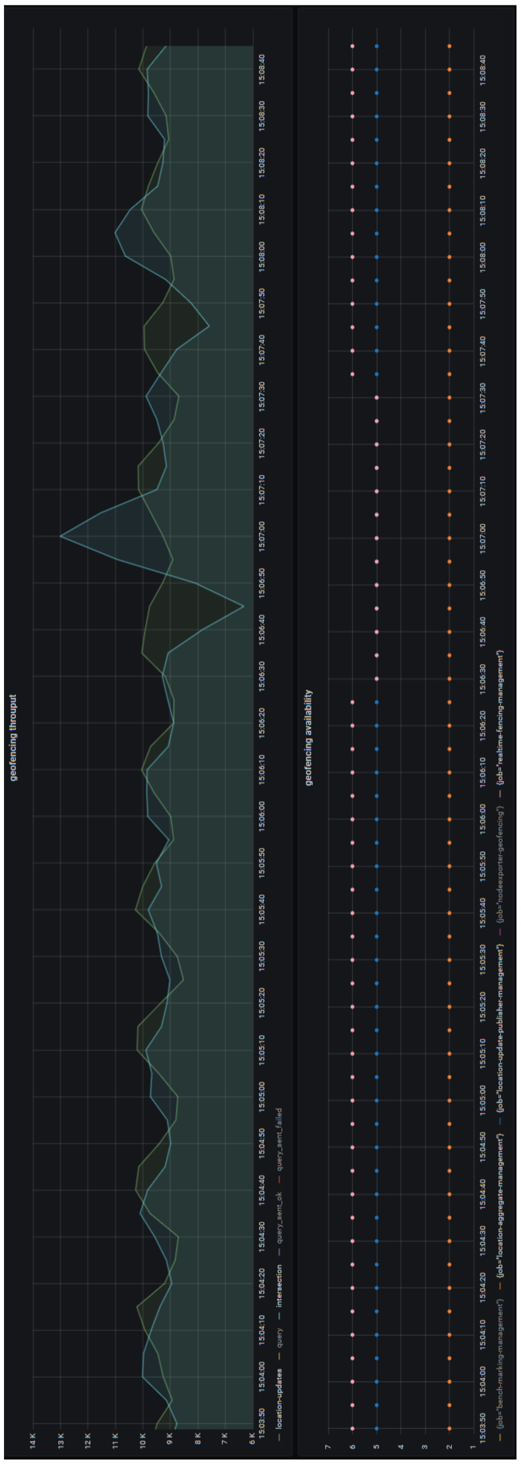
8- Repeat the experiment with the difference of restarting two instances instead of one.

Experiment 5: Availability of push leg

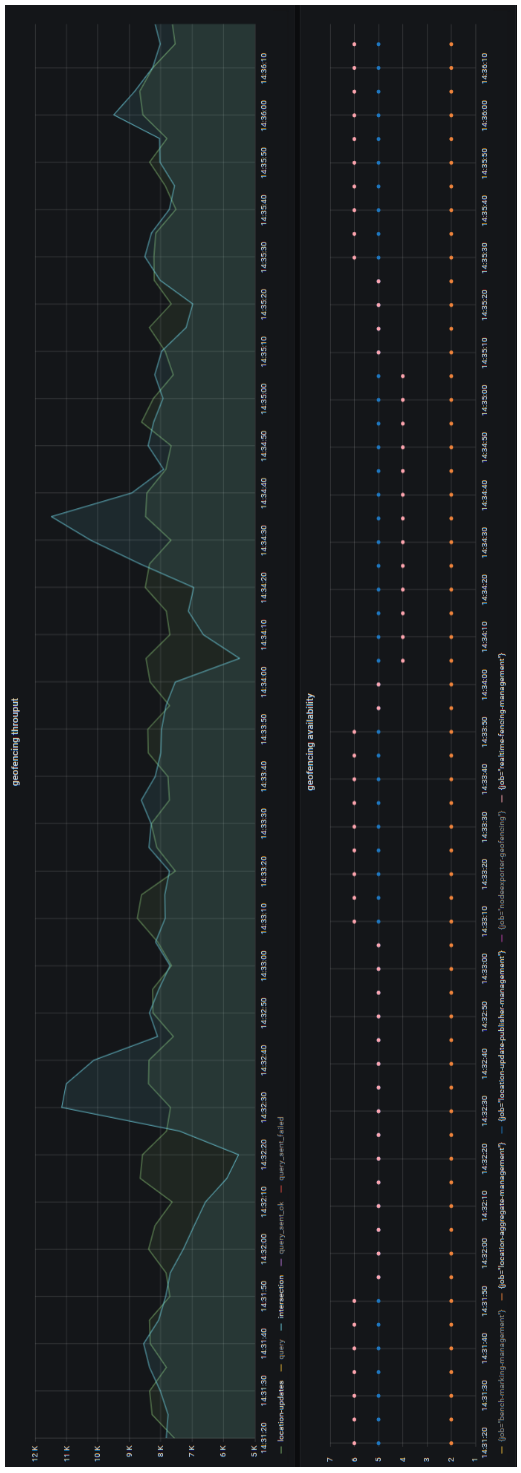
Application	number of instances	CPU(MHz)	RAM(MB)
location-update-publisher	5	400	700
location-aggregate	-	-	-
realtime-fencing	6	400	800

Table 9.5: Experiment 5 deployment view

Table 9.5 shows, for experiment 5, how many instances of each FenceX microservices we have deployed in addition to the number of resources we have assigned to each. The row in table 9.5 with no values corresponds to a microservice that does not relate to the experiment 5. The following figures illustrate how the input rate and throughput have changed during repetitions of experiment 5. Each repetition was with a different input rate pattern. They also show how (and when the) number of deployed instances of related microservices has changed during these repetitions.



(a) FenceX push leg availability ex5-1



(b) FenceX push leg availability ex5-2

Discussion of push leg availability experiments

In figures 9.6a and 9.6b, blue line represents the push throughput. The green line illustrates the ongoing input stream of location updates and the pink point shows the number of up and running instances of realtime-fencing.

As you can see in figures 9.6a and 9.6b, there are points in the timeline at which the number of realtime-fencing instances go down (by 1 and/or 2) and later come back to 6 again. During the time which takes for all 6 instances to be up and running again, throughput slightly drops and eventually recovers. Since we are using Kafka topics as durable storage of location updates, the location updates which didn't get a chance to be processed, get it after re-balancing. As a result, we had even a higher throughput than the input rate temporarily after re-balancing finishes.

Push leg of FenceX can handle some of its instances going down and coming back up. Even if an unhealthy instance does not recover, the only effect will be on the overall throughput of the system under high enough input loads. Also, thanks to the durability and offset system of Kafka topics the probability of an event not getting processed eventually is very low.

9.3.2 Pull leg

Pull leg availability evaluation scenario:

1- Send location updates for movers using all the trips in the bench-marking application database.

2- Start an ongoing load of queries (query by fence) to FenceX.

3- Wait until throughput becomes stable.

4- Restart one of the instances of location-aggregate.

5- Assert that throughput drops.

6- Wait until successful re-balancing.

7- Assert that throughput is back to the previous value.

8- Repeat the experiment with the difference of restarting two instances instead of one.

Experiment 7: Availability of pull leg

Application	number of instances	CPU(MHz)	RAM(MB)
location-update-publisher	1	400	700
location-aggregate	4	2700	2700
realtime-fencing	-	-	-

Table 9.6: Experiment 7 deployment view

Table 9.6 shows, for experiment 7, how many instances of each FenceX microservices we have deployed in addition to the amount of resources we have assigned to each. The row in table 9.6 with no values corresponds to a microservice that does not relate to the experiment 7. The following figure illustrates how to input rate and throughput have changed during experiment 7. It also shows how (and when the) number of deployed instances of related microservices has changed during this experiment.

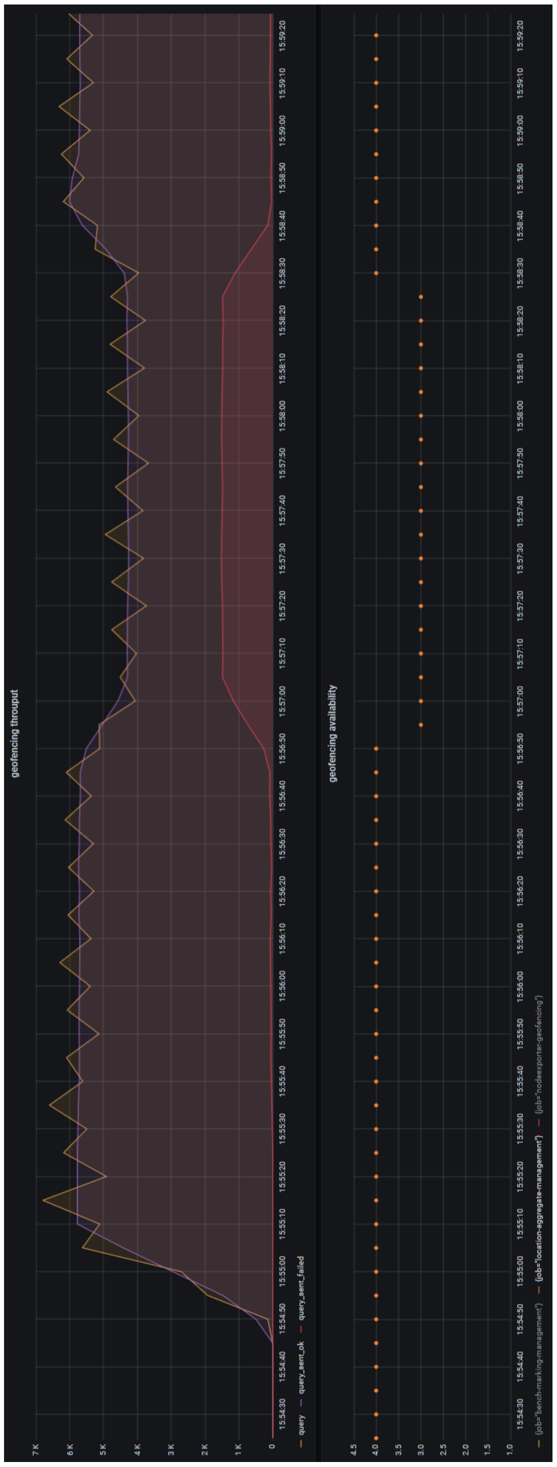


Figure 9.7: FenceX pull leg availability experiment 7

Discussion of pull leg availability experiments

In figure 9.7 the purple line is the input rate (number of queries received by FenceX per second) and the yellow line shows the rate of queries answered successfully. The red line shows how many queries FenceX failed to respond to. A failure to respond in this load range means that the service supposed to answer the query was not up and running at that moment. It is clear that once one of the instances of pull leg goes down (because of restart), the throughput drops and the error rate goes up accordingly. Once restart and re-balancing finishes successfully, the throughput increases back to its previous value and error rate drops back to zero.

Pull leg of FenceX can handle some of its instances going down and coming back up. Even if an unhealthy instance does not recover, the only effect will be on the overall throughput of system temporarily.

9.4 Strong scalability

If software has strong scalability properties means providing the same load, the more you scale it out, the better it performs. For a given circumstance, there might be a point after which scaling out more won't affect the performance anymore. The load can be data set size, table size, or incoming HTTP query rate. Performance is also case and context-dependent. It can be query latency or throughput. In the stream processing systems the load is the rate of input, rate of incoming events, the rate at which the source produces input into the pipeline. In FenceX, for pull leg, similar to previous experiments, the load is the rate of incoming HTTP requests for queries by fence. And, for push leg is the rate at which location reports arrive. Performance in our experiments is throughput of push leg and pull leg. We expect FenceX to show strong scalability characteristics in both of its legs.

9.4.1 Push leg

Push leg strong scalability evaluation scenario:

1- Define fences for movers using all the trips in the bench-marking application database.

2- Start an ongoing stream of location reports with fixed-rate toward FenceX.

3- Deploy only one instance of realtime-fencing (should not be rich in resources as we want it to be overwhelmed).

4- Check the throughput. Hopefully, it is much lower than the input rate.

5- Deploy one more instance of realtime-fencing with exactly the same resources as the previous one.

6- Assert an increase in throughput.

7- Keep adding instances and asserting an increase in throughput until throughput stops increasing.

During these experiments, each time we deploy one more instance of realtime-fencing, a re-balancing happens which avoids some incoming location updates from being processed. Those location updates get buffered in the topic and after re-balancing will get their chance to be processed. A direct consequence of such buffering is that at some points in the experiments, the throughput we observe is higher than the input rate. This is because the input stream is an ongoing and fixed-rate, so the process of buffered location updates only adds to throughput in that specific moment.

Experiments 8 to 11: Push leg strong scalability

Application	number of instances	CPU(MHz)	RAM(MB)
location-update-publisher	4	200	700
location-aggregate	-	-	-
realtime-fencing	1	30	500

Table 9.7: Experiment 8 deployment view

Application	number of instances	CPU(MHz)	RAM(MB)
location-update-publisher	4	200	700
location-aggregate	-	-	-
realtime-fencing	2	30	500

Table 9.8: Experiment 9 deployment view

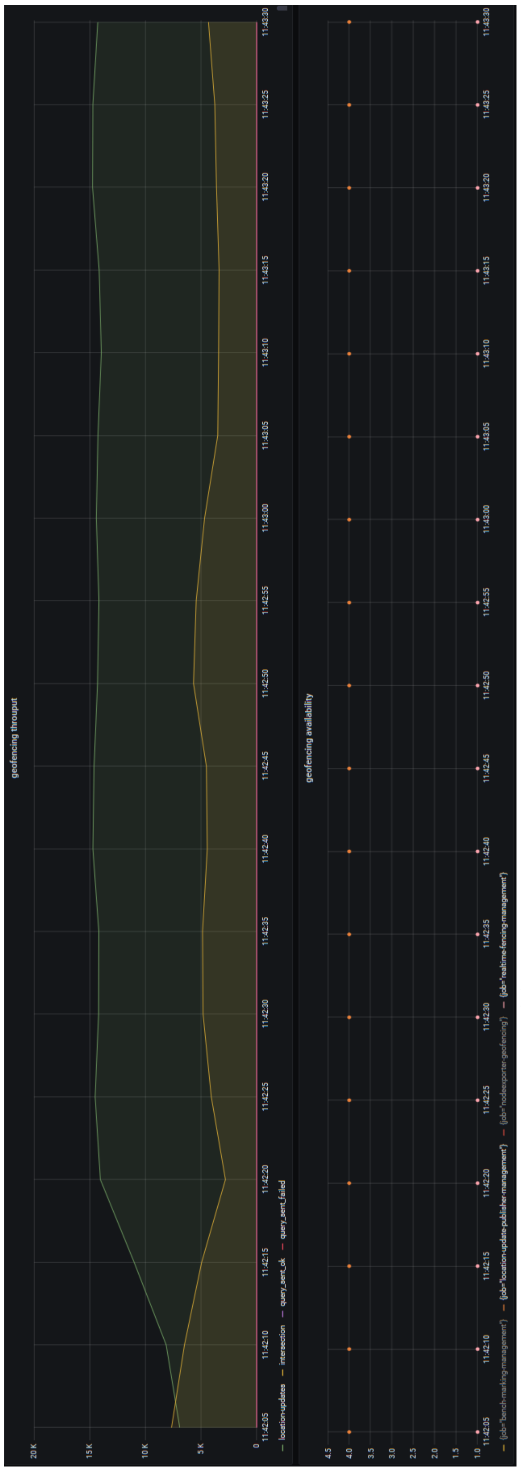
Application	number of instances	CPU(MHz)	RAM(MB)
location-update-publisher	4	200	700
location-aggregate	-	-	-
realtime-fencing	3	30	500

Table 9.9: Experiment 10 deployment view

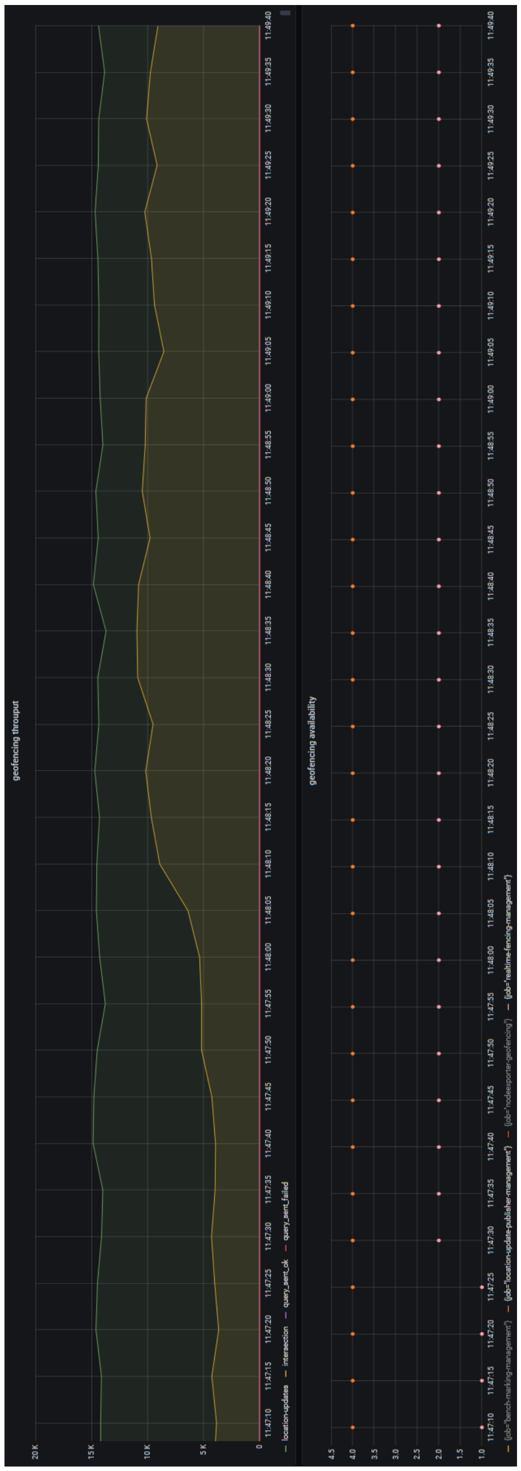
Application	number of instances	CPU(MHz)	RAM(MB)
location-update-publisher	4	200	700
location-aggregate	-	-	-
realtime-fencing	4	30	500

Table 9.10: Experiment 11 deployment view

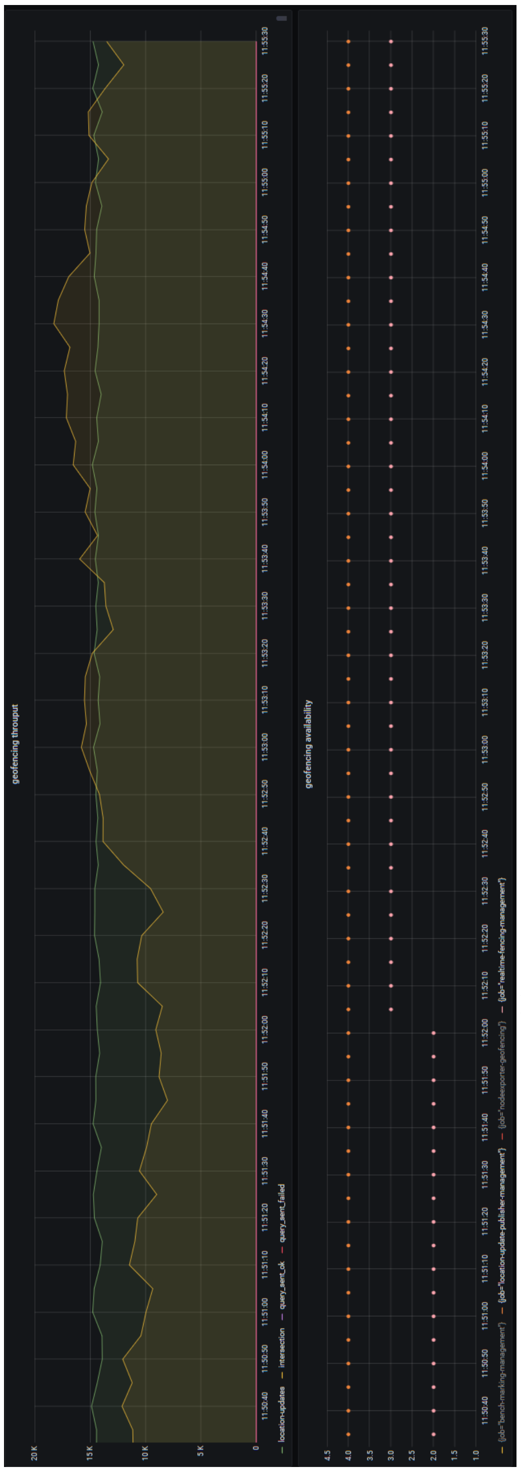
Tables 9.7, 9.8, 9.9 and 9.10 show, for experiments 8 to 11, how many instances of each FenceX microservices we have deployed in addition to the amount of resources we have assigned to each. The row in these tables with no values corresponds to a microservice which does not relate to experiments 8 to 11. The following figures illustrate how to input rate and throughput have changed during those experiments. It also shows the number of deployed instances of related microservices during this experiment.



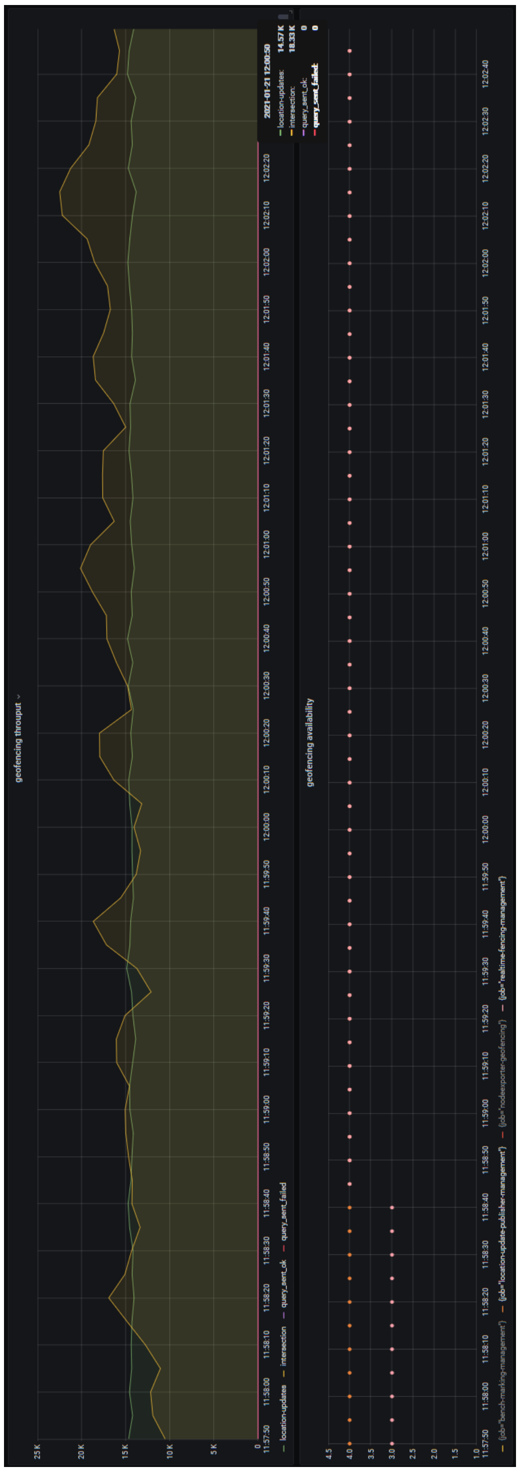
(a) FenceX push leg strong scalability experiment 8



(b) FenceX push leg strong scalability experiment 9



(a) FenceX push leg strong scalability experiment 10



(b) FenceX push leg strong scalability experiment 11

Discussion of push leg strong scalability experiments

In figures 9.8a, 9.8b, 9.9a and 9.9b the green line is essentially the input rate (number of location updates per second), yellow line represents throughput (number of fence point intersections) and the pink dots shows number of deployed instances of realtime-fencing.

During experiments 8 to 11, we have sent an ongoing stream of location updates to FenceX with a fixed rate of 15K location updates per second. It is the highest input rate that we could produce in a stable manner.

Figure 9.8a (experiment 8) illustrates that one instance of realtime-fencing with the rather low given resources can only handle one 3rd of the input rate. Which is 5k intersections per second out of 15k location updates per second.

Looking at figure 9.8b (experiment 9) we can easily observe that once the second instance is up and running, after re-balancing, throughput goes proportionally up, from 5k/s to 10k/s. At this point, we can guess that without changing anything, just adding one more instance of realtime-fencing should be enough to cover the whole 15k/s input rate.

Figure 9.9a (experiment 10) as we expected, shows that deploying three instances of realtime-fencing with the given resources, was enough to process 15K location updates per second. Also in figure 9.9a we can see the effect of buffered location updates on throughput. Around the end parts of the graph, throughput is higher than the input rate.

In order to be just sure that adding more instances, won't affect the throughput in any good or bad way, we add another instance as the final experiment (experiment 11). The clear observation of figure 9.9b (experiment 11) is that adding one more instance did not have any special effect on the throughput; since throughput can not exceed the input rate regardless of processing power.

So as a result of experiments 8 to 11, we can claim that the push leg of FenceX has strong scalability characteristics at least in the range of input rate we have managed to produce.

9.4.2 Pull leg

Pull leg strong scalability evaluation scenario:

1- Send location reports for movers using all the trips in the bench-marking application database.

2- Start an ongoing load of queries (query by fence) with a fixed rate.

3- Deploy only one instance of location-aggregate (should not be too rich in resources, we want it to be overwhelmed).

4- Check the throughput. Hopefully, it is much lower than the input rate.

5- Deploy one more instance of location-aggregate with exactly the same resources as the previous one.

6- Assert an increase in throughput.

7- Keep adding instances and asserting an increase in throughput until throughput stops increasing.

During these experiments, each time we deploy one more instance of location-aggregate, a re-balancing happens which avoids some queries from being answered. Those queries will be considered as failed.

Experiment 17 and 18: Pull leg strong scalability

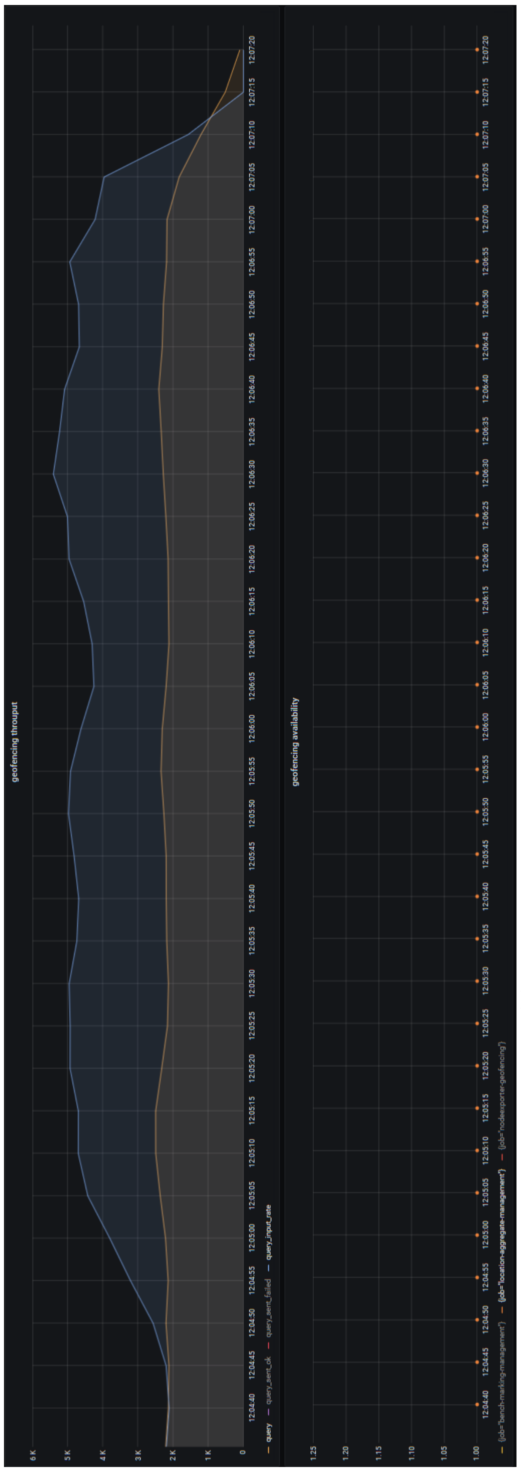
Application	number of instances	CPU(MHz)	RAM(MB)
location-update-publisher	1	200	700
location-aggregate	1	100	1500
realtime-fencing	-	-	-

Table 9.11: Experiment 17 deployment view

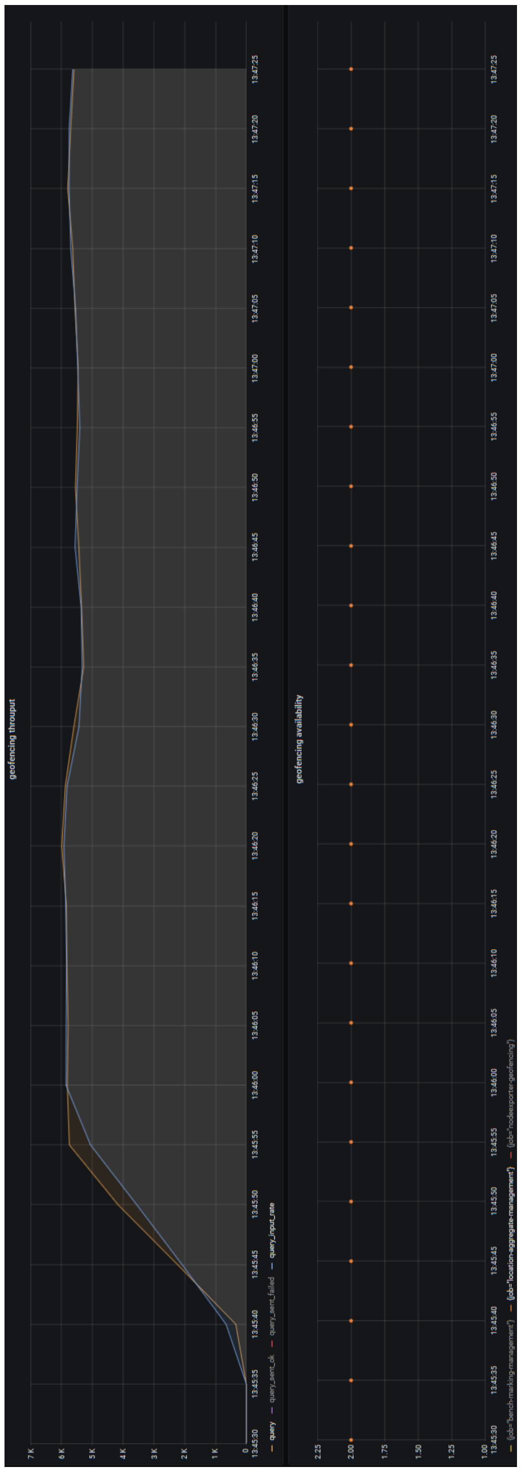
Application	number of instances	CPU(MHz)	RAM(MB)
location-update-publisher	0	-	-
location-aggregate	2	100	1500
realtime-fencing	-	-	-

Table 9.12: Experiment 18 deployment view

Tables 9.11 and 9.12 show, for experiments 17 and 18, how many instances of each FenceX microservices we have deployed in addition to the amount of resources we have assigned to each. The row in these tables with no values corresponds to a microservice which does not relate to experiments 17 and 18. The following figures illustrate how the input rate and throughput have changed during experiments 17 and 18. It also shows how many instances of related microservices were up and running during those experiments.



(a) FenceX pull leg strong scalability experiment 17



(b) FenceX pull leg strong scalability experiment 18

Discussion of pull leg strong scalability experiments

In figures 9.10a and 9.10b the blue line represents input rate, number of incoming HTTP requests. The orange line illustrates throughput which is number of queries responded successfully. Orange dots show how many instances of location-aggregate are up and running. As figures show, FenceX has managed to handle 2k queries per second with one not so rich instance of location-aggregate. Adding one more instance resulted in throughput becoming equal to the input rate (6K queries per second).

One of the differences between push and pull leg is that the minimum resources required for pull leg to function smoothly is much higher than push leg. This is due to the fact that the pull leg has a global in-memory data store. Also, the amount of data that exists in the pull leg is substantially higher. There are usually way more location updates for each mover than fences. In fact, there will be at most one fence in the push leg for each mover compared to many location updates in the pull leg. So pull leg requires more memory comparatively.

As an overall consequence, when we deploy even one instance of location-aggregate, it is already rich enough to be able to answer many HTTP requests (query by fence). Which is about half of the input rate we can produce at maximum in a stable ongoing manner. So when we deploy two instances of location-aggregate with the mentioned resources, two available instances just handle our full capacity of input load production. So there is no point in continuing strong scalability experiments for pull leg within this range of input.

All in all, pull leg of FenceX has strong scalability characteristics at least in the range of input rate we have managed to produce.

9.5 Weak scalability

If software has weak scalability properties, it means providing a load that is increasing step by step over time, the more you scale the software out, the better it performs. For a given circumstance, there might be a point after which more scale out or more load won't affect the performance anymore. The load can be data set size, table size, or incoming HTTP query rate. Performance is also case and context-dependent. It can be query latency or throughput. In the stream processing systems, the load is the rate of input, rate of incoming events, the rate at which source publishes input into the pipeline. In FenceX, for pull leg, similar to previous experiments, the load is the rate of incoming HTTP requests for queries by fence. And, for push leg is the rate at which location reports arrive. Performance in the case of our experiments is throughput of push leg and pull leg. We expect both legs of FenceX to have weak scalability characteristics.

9.5.1 Push leg

Push leg weak scalability evaluation scenario:

1- Define fences for movers using all the trips in the bench-marking application database.

2- Start an ongoing stream of location reports with a fixed low rate toward FenceX.

3- Deploy only one instance of realtime-fencing (should not be too rich in resources as we want it to be overwhelmed).

4- Check the throughput. Hopefully, it is equal to the input rate.

5- Deploy one more instance of realtime-fencing with exactly the same resources as the previous one. Also, increase the input rate (like double).

6- Assert that throughput covers the input rate.

7- Keep adding instances, increasing input rate, and asserting increase in throughput until throughput stops increasing.

During these experiments, each time we deploy one more instance of realtime-fencing, a re-balancing happens which avoids some incoming location updates from being processed. Those location updates get buffered in the topic and after re-balancing will get their chance to be processed. A direct consequence of such buffering is that at some points in the experiments, the throughput we observe might be higher than the input rate. This is because the input stream is an ongoing and fixed rate.

Experiments 19 to 23: Push leg weak scalability

Application	number of instances	CPU(MHz)	RAM(MB)
location-update-publisher	4	400	700
location-aggregate	-	-	-
realtime-fencing	1	60	500

Table 9.13: Experiment 19 deployment view

Application	number of instances	CPU(MHz)	RAM(MB)
location-update-publisher	4	400	700
location-aggregate	-	-	-
realtime-fencing	2	60	500

Table 9.14: Experiment 20 deployment view

Application	number of instances	CPU(MHz)	RAM(MB)
location-update-publisher	4	400	700
location-aggregate	-	-	-
realtime-fencing	4	60	500

Table 9.15: Experiment 21 deployment view

Application	number of instances	CPU(MHz)	RAM(MB)
location-update-publisher	4	400	700
location-aggregate	-	-	-
realtime-fencing	5	60	500

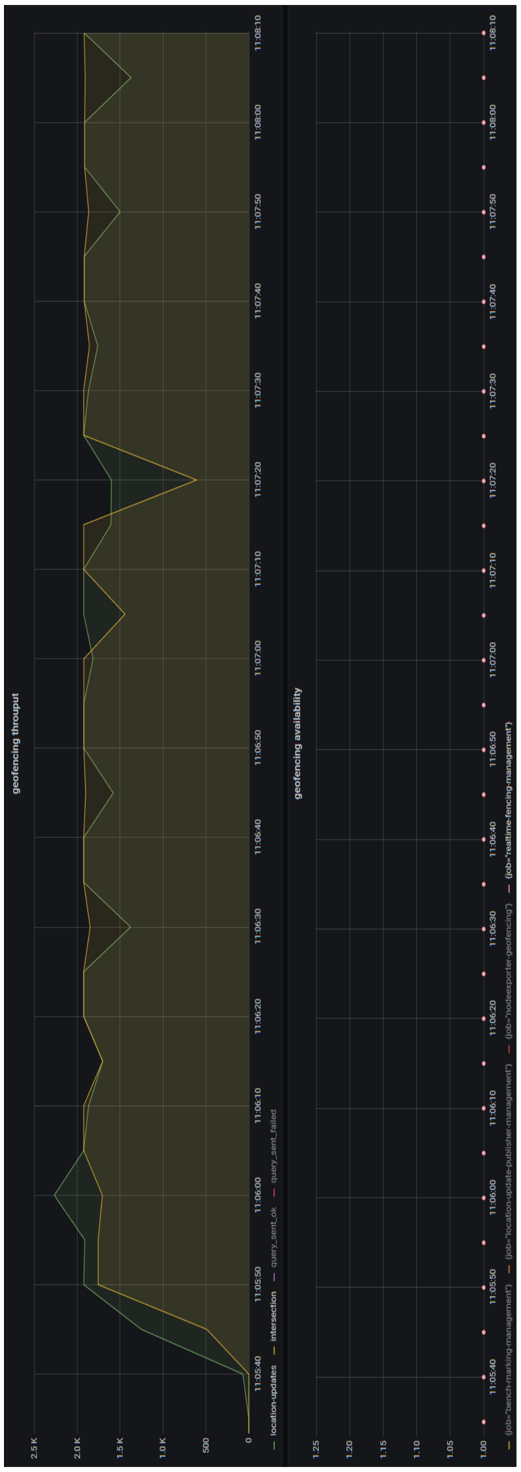
Table 9.16: Experiment 22 deployment view

Application	number of instances	CPU(MHz)	RAM(MB)
location-update-publisher	4	400	700
location-aggregate	-	-	-
realtime-fencing	10	60	500

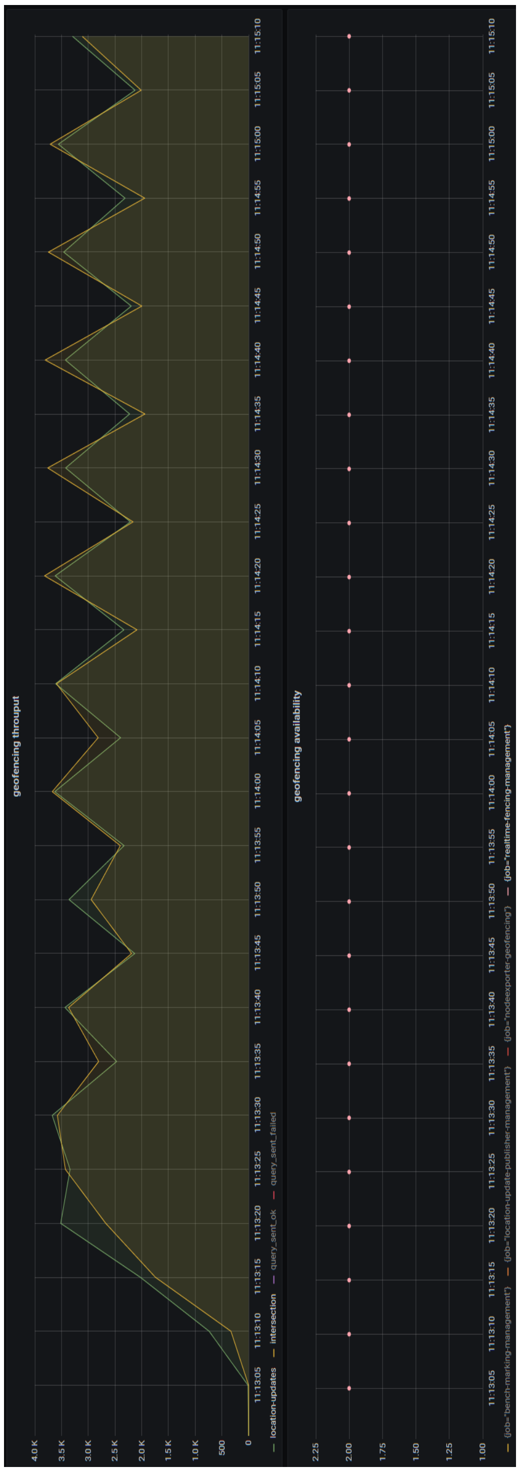
Table 9.17: Experiment 23 deployment view

Tables 9.13, 9.14, 9.15, 9.16 and 9.17 show, for experiment 19 to 23, how many instances of each FenceX microservices we have deployed in addition to the amount of resources we have assigned to each. The row in these tables with no values corresponds to a microservice which does not relate to experiments 19 to 23.

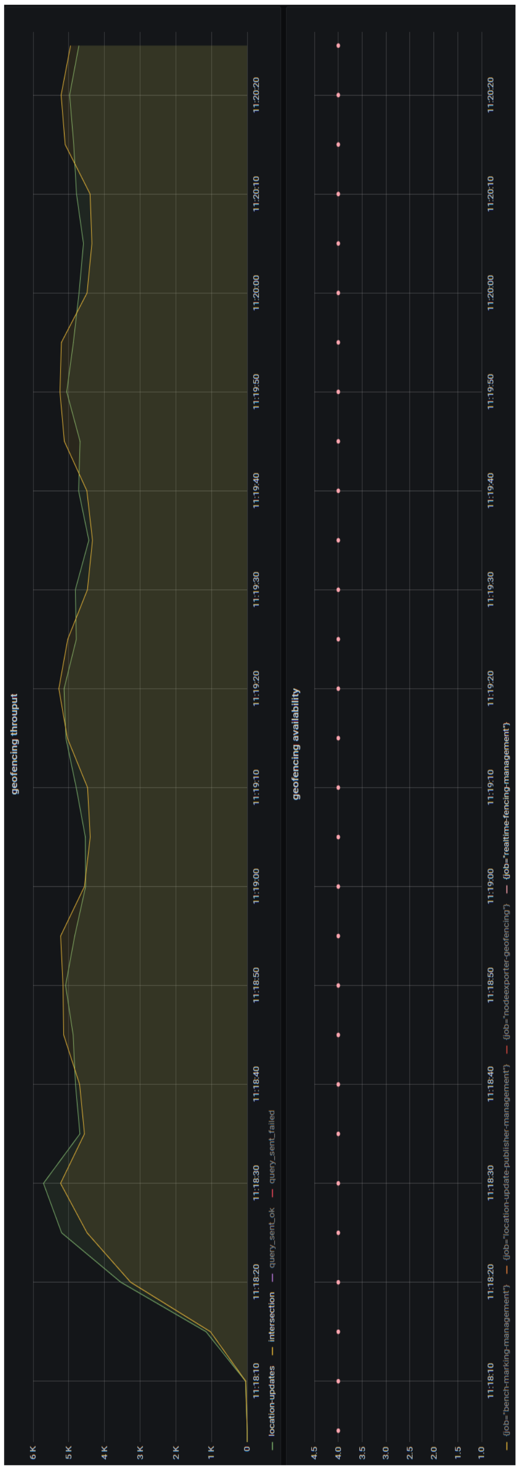
The following figures illustrate how the input rate and throughput have changed during experiments 19 to 23. It also shows how many instances of related microservices were up and running during these experiments.



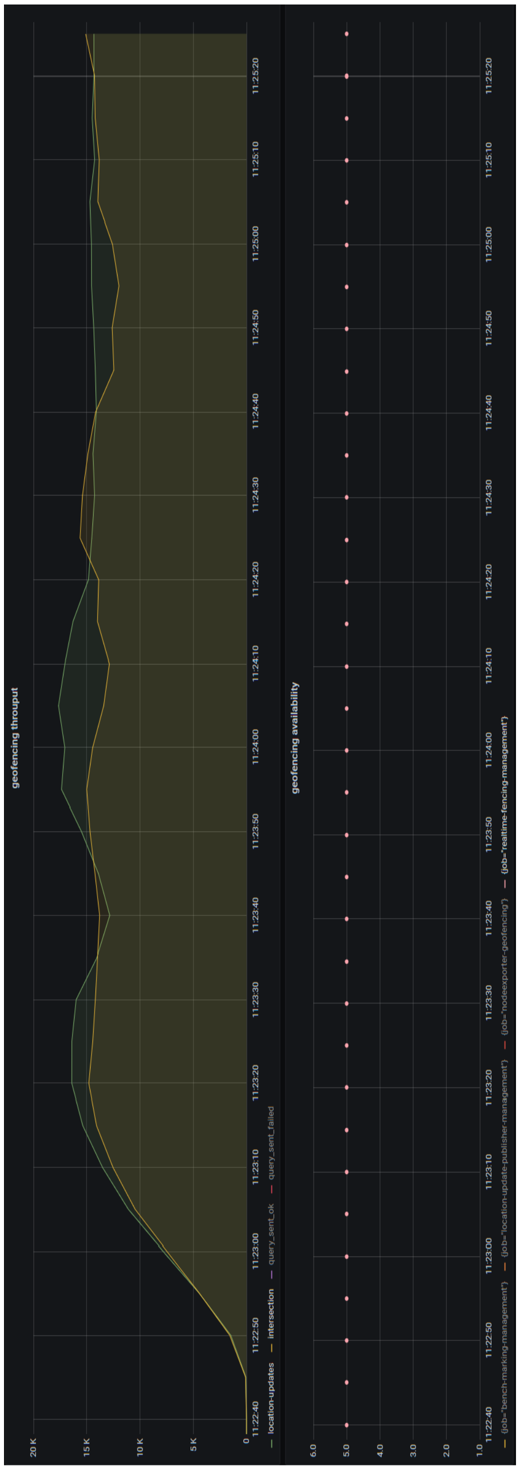
(a) FenceX push leg weak scalability experiment 19



(b) FenceX push leg weak scalability experiment 20



(a) FenceX push leg weak scalability experiment 21



(b) FenceX push leg weak scalability experiment 22

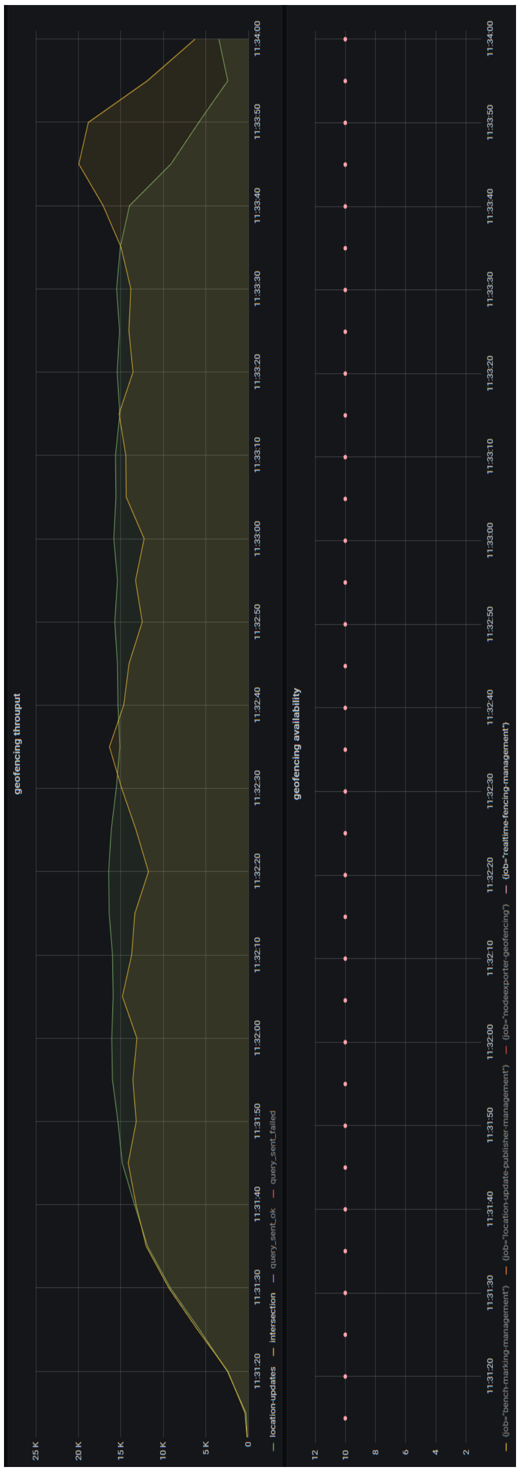


Figure 9.13: FenceX push leg weak scalability experiment 23

Discussion of push leg weak scalability experiments

In figures 9.11a, 9.11b, 9.12a, 9.12b and 9.13 the green line is input rate (number of location updates per second), yellow line represents throughput (number of fence point intersections) and the pink dots shows number of deployed instances of realtime-fencing. As we progressed through experiments 19 to 23, we have increased the input rate from around 2k/s to 15k/s. At the same time in each experiment we have increased the number of deployed instances of realtime-fencing; Which is the microservice working as the push leg of FenceX.

Figure 9.11a (experiment 19) shows that 2 instances of realtime-fencing managed to handle around 2K location updates per second.

Figure 9.11b (experiment 20) shows that 4 instances of realtime-fencing managed to handle around 3K location updates per second.

Figure 9.12a (experiment 21) shows that 5 instances of realtime-fencing managed to handle around 5K location updates per second.

10 instances of realtime-fencing, as depicted in figure 9.12b managed to handle around 15K location updates per second.

In experiment 23, depicted in figure 9.13 our intention was to increase the input rate to above 15k/s which we did not manage due to limitations on the available resources to the bench-marking application. So we will not continue weak scalability experiments for push log further.

We observed that as we increased input rate and deployed more instances of push leg of FenceX, push leg throughput went higher. So push leg of FenceX managed to scale well in face of increasing load and it has weak scalability characteristics.

9.5.2 Pull leg

Pull leg weak scalability evaluation scenario:

1- Send location updates for movers using all the trips in the bench-marking application database.

2- Start an ongoing load of queries (query by fence) with a fixed low rate.

3- Deploy only one instance of location-aggregate (should not be too rich in resources, we want it to be overwhelmed).

4- Check the throughput. Hopefully, it is equal to the input rate.

5- Deploy one more instance of location-aggregate with exactly the same resources as the previous one. Also, increase the input rate (like double).

6- Assert that throughput covers the input rate.

7- Keep adding instances, increasing input rate, and asserting increase in throughput until throughput stops increasing.

Relying on the observations from previous experiments, two instances of location-aggregate (with minimum resources for working smoothly) can easily cover our maximum capacity for producing input load. So there is no point in carrying out weak scalability experiments for the pull leg.

9.6 Stress tolerance

In order to test the tolerance of stream processing systems against a sudden huge load or trying to find out how they can scale under high rates of input load, firstly we need to produce high rates of input. It can be challenging when the input data is complex to produce and/or available resources to the input producers is limited. Stream processing systems rely on some sort of buffering which means events should be stored somewhere so that processors can handle them when they are ready. That's how they get some sort of back pressure in reactive systems terminology. In the case of FenceX, that buffer is the Kafka topics which can grow very large without any problem.

So instead of trying to produce a high rate of input in realtime, we can buffer a large number of events in a Kafka topic (waiting to be picked up and processed) while processors are down. When enough events got accumulated, we stop buffering events and deploy event processors. Now processors can read the events with the least possible I/O involved. Applying this approach, we can assess how well an event processor can be scaled when there is almost no factor involved other than input rate and processing power.

This style of testing is only applicable to push leg since the concept of buffering work differently in terms of HTTP request. Basically, HTTP based communication does not support back pressure.

9.6.1 Push leg

Push leg stress tolerance evaluation scenario:

1- Play with different input rates and deployment configurations to get a feeling of how to tune the system for best performance

2- Undeploy all realtime-fencing instances.

3- Start an ongoing stream of location reports with fixed-rate toward FenceX.

4- Wait 5 minutes or so.

5- Now that there is plenty of location updates buffered in the Kafka topic, stop the stream of location updates.

6- Deploy realtime-fencing instances according to the previously discovered deployment configuration.

7- Check the throughput and assert that it's much higher than the maximum observed throughput in previous experiments.

Experiment 24: Push leg stress tolerance

Application	number of instances	CPU(MHz)	RAM(MB)
location-update-publisher	4 - 0	400	700
location-aggregate	-	-	-
realtime-fencing	0 - 12	60	500

Table 9.18: Experiment 24 deployment view

Table 9.18 shows, for experiment 24, how many instances of each FenceX microservices we have initially (and eventually) deployed in addition to the amount of resources we have assigned to each microservice instance. The "number of instances" row in this table has basically two values. The first value (from left) refers to the number of deployed instances of each microservices during input production (buffering phase) and the other value refers to the same number during the processing phase. The row in table 9.18 with no values corresponds to a microservice which does not relate to experiment 24. The following figure illustrates how to input rate and throughput have changed during experiment 24. It also shows how (and when the) number of deployed instances of related microservices has changed during this experiment.

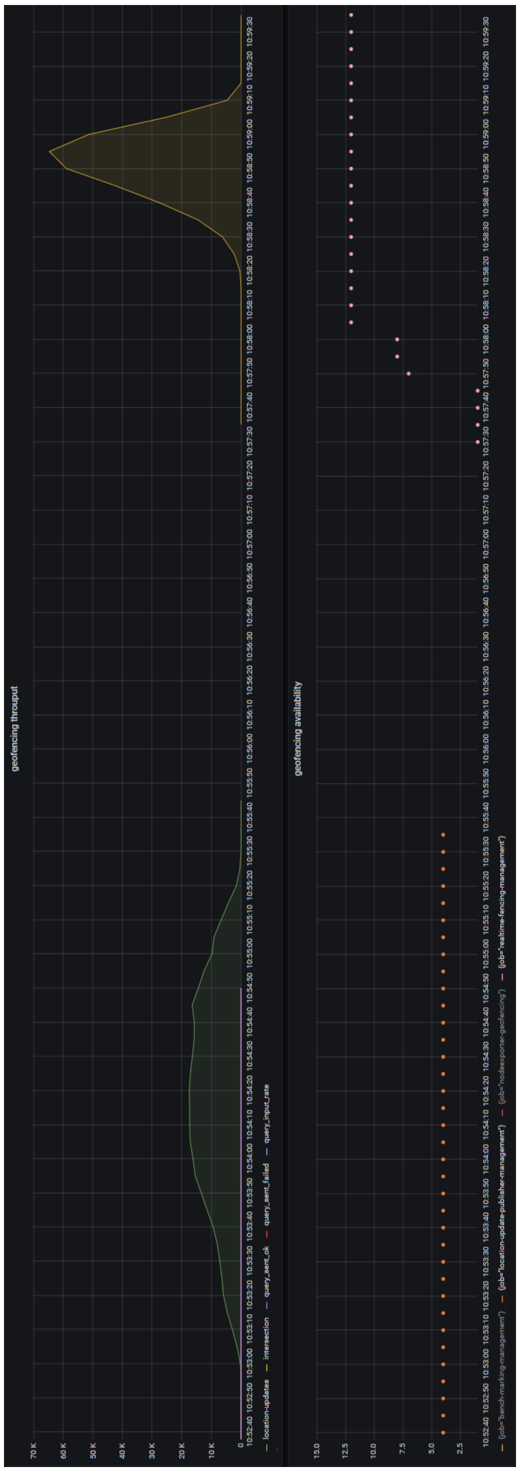


Figure 9.14: FenceX push leg stress tolerance experiment 24

Discussion of push leg stress tolerance experiment

In figure 9.14 the green line is input rate (number of location updates per second), the yellow line represents throughput (number of fence point intersections) and the pink dots shows the number of deployed instances of realtime-fencing. Orange dots illustrate the number of up and running instances of location-update-publisher. We can clearly observe that after re-balancing among 10 newly deployed instances of realtime-fencing is finished, throughput has peaked very fast to about 65K fence-point intersections per second.

We can rely on buffering properties of stream processing systems for testing the throughput of operators. Especially when producing input and process it in realtime involves too much I/O latency. All in all, pull leg of FenceX has shown high tolerance in face of stressful load and the ability to deliver throughput of 65k/s.

The throughput FenceX has achieved here (65k/s) is much higher compared to the throughput achieved by [18] (20k/s). So now we can claim with confidence that FenceX solution has outperformed [18].

Chapter 10

Conclusion and future directions

During this thesis, we have combined microservices and stream processing concepts in order to design, implement and evaluate a highly scalable resilient geofencing system called FenceX. FenceX can handle high input rates and deliver high throughput with low latency. FenceX supports two types of geofencing, on-demand and realtime. The on-demand part is achieved by having a fully replicated co-located in-memory geo-aware database. The realtime part is brought about using a distributed join over a partitioned table of the fences and an ongoing stream of location reports.

The result was so scalable that in most evaluation scenarios, we ran out of resources for producing more load before we can hit the upper bound of throughput (with a given deployment config). At best we have observed that FenceX can easily handle about 65K fence-point intersections and 15K queries (query by fence) per second. FenceX also has proven to have strong and weak scalability characteristics.

When it comes to resiliency thanks to replication, scaling out (instead of up), re-balancing features of Kafka consumers and highly decoupled components, FenceX has demonstrated being reliable under non-ideal conditions.

Another way to look at FenceX is to consider it as an implementation for the idea of FaaS (function as a service). Stateful FaaS to be more precise. FenceX is basically a stream processing pipeline that has both stateful and stateless operations. Although the operations in the pipeline are limited to the design of the pipeline at each point at the time, the implementation of each pipeline can be customized without the need to change the core program. In fact, different implementations can be plugged in as libraries. Which is similar to implementing a custom function and give to a FaaS provider. The major difference between our system and other implementations of FaaS is that we allow for stateful functions. It was possible since we have stateful operations in the FenceX stream processing pipeline.

During our evaluations, we have used only one type of data collected from real taxi trips. Evaluating FenceX using data collected from people walking in an area can be also very insightful.

Apart from the data, the implementation of our bench-marking application might have the potential to be improved so that more load can be produced with less resources consumed.

And in terms of features, pattern detection can be the next step for FenceX. Patterns that can be detected while observing events in windows of time. For example, when a mover enters a no-go area and gets back out within seconds, an alarm might not be needed to be generated. Currently, FenceX just detects enter/exit events on a one-by-one basis.

FenceX is only the core/kernel part of a FaaS platform. In order to make it a fully functioning cloud service, some other tools and layers should be built around and upon it. For example, a web UI in which developers implement their functions (simple cases) and the system compiles the code itself and generate the library.

The next step is to use that library to build microservices involved in FenceX. Then deploying them. This whole process can be automated.

Bibliography

[1] J. Bonér, D. Farley, R. Kuhn, and M. Thompson. The reactive manifesto. [Online]. Available: https://www.reactivemanifesto.org/

[2] F. Hueske and V. Kalavri, *Stream Processing with Apache Flink.* 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly, 2017.

[3] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov, "Cloudburst: Stateful functions-as-a-service," *Proc. VLDB Endow.*, vol. 13, no. 12, p. 2438–2452, Jul. 2020. [Online]. Available: https://doi-org.ezproxy.its.uu.se/10.14778/3407790.3407836

[4] smartcitysweden.com. Geofencing. [Online]. Available: https://smartcitysweden.com/focus-areas/mobility/geofencing/

[5] N. Lahoti. What is geofencing and why is it important in taxi app development? [Online]. Available: https://mytaxipulse.com/blog/geofencing-in-taxi-app-development.html

[6] pettrackerreviews.com. What is geofencing? [Online]. Available: https://www.pettrackerreviews.com/what-is-geofencing/

[7] N. Lahoti. Mvc: Model, view, controller. [Online]. Available: https://www.codecademy.com/articles/mvc

[8] C. Richardson. Pattern: Monolithic architecture. [Online]. Available: https://microservices.io/patterns/monolithic.html

[9] M. Amini. Fencex github repository. [Online]. Available: github.com/bmd007/statefull-geofencing-faas

[10] M. Kleppmann, *Making Sense of Stream Processing.* 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly, 2015.

[11] A. Bucchiarone and N. Dragoni, *Microservices Science and Engineering.* Gewerbestrasse 11, 6330 Cham, Switzerland: Springer, 2020.

[12] Apache. Apache kafka. [Online]. Available: https://kafka.apache.org

[13] B. Stopford, *Designing Event-Driven Systems.* 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly, 2018.

[14] apache.ord. Kafka streams. [Online]. Available: https://kafka.apache.org/documentation/streams/

[15] Redhat.com. What is function-as-a-service (faas)? [Online]. Available: https://www.redhat.com/en/topics/cloud-native-apps/what-is-faas

[16] amazon. Run code without thinking about servers or clusters. only pay for what you use. [Online]. Available: https://aws.amazon.com/lambda

[17] F. Cirillo, T. Jacobs, M. Martin, and P. Szczytowski, "Large scale indexing of geofences," in *2014 Fifth International Conference on Computing for Geospatial Research and Application*, Aug 2014, p. 1–8.

[18] B. Târnaucă, D. Puiu, S. Nechifor, and V. Comnac, "Using complex event processing for implementing a geofencing service," in *2013 IEEE 11th International Symposium on Intelligent Systems and Informatics (SISY)*, Sep 2013, p. 391–396.

[19] D. Barcelona-Pons, M. Sánchez-Artigas, G. París, P. Sutra, and P. García-López, "On the faas track: Building stateful distributed applications with serverless architectures," in *Proceedings of the 20th International Middleware Conference*, ser. Middleware '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 41–54. [Online]. Available: https://doi.org/10.1145/3361525.3361535

[20] C. Wu, V. Sreekanti, and J. M. Hellerstein, "Transactional causal consistency for serverless computing," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 83–97. [Online]. Available: https://doi.org/10.1145/3318464.3389710

[21] wiki.gis.com. Well-known text. [Online]. Available: http://wiki.gis.com/wiki/index.php/Well-known_text

[22] mariadb. Well-known binary (wkb) format. [Online]. Available: https://mariadb.com/kb/en/well-known-binary-wkb-format/

[23] geojson.org. Geojson. [Online]. Available: https://geojson.org

[24] OGC. Geography markup language. [Online]. Available: https://www.ogc.org/standards/gml

[25] h2database.com. H2 database engine. [Online]. Available: http://h2database.com

[26] A. Endsley. Wicket. [Online]. Available: https://arthur-e.github.io/Wicket/sandbox-gmaps3.html

[27] J. Carroll. Wkt polygons of all us states' borders. [Online]. Available: https://gist.github.com/JoshuaCarroll/49630cbeeb254a49986e939a26672e9c

[28] mountaingoatsoftware.com. user stories. [Online]. Available: https://www.mountaingoatsoftware.com/agile/user-stories

[29] JetBrains. A modern programming language that makes developers happier. [Online]. Available: https://kotlinlang.org/

[30] J. Bonér, *Reactive Microsystems.* 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly, 2017.

[31] D. Vohra, *Pro Docker.* Apress, 2017.

[32] I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen, *Microservice Architecture.* 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly, 2016.

[33] M. Richards, *Microservices Antipatterns and Pitfalls.* 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly, 2016.

[34] D. Wampler, *Fast Data Architectures for Streaming Applications.* 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly, 2016.

[35] F. Troßbach and M. J. Sax. Crossing the streams – joins in apache kafka. [Online]. Available: https://www.confluent.io/blog/crossing-streams-joins-apache-kafka/

[36] rabbitmq. Rabbitmq is the most widely deployed open source message broker. [Online]. Available: https://www.rabbitmq.com

[37] HashiCorp. Service networking across any cloud. [Online]. Available: https://www.consul.io

[38] Oracle. java.nio. [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/nio/package-summary.html

[39] HashiCorp. Workload orchestration made easy. [Online]. Available: https://www.nomadproject.io

[40] prometheus.io. From metrics to insight. [Online]. Available: https://prometheus.io

[41] Spring.io. Web on reactive stack. [Online]. Available: https://docs.spring.io/spring-framework/docs/current/reference/html/web-reactive.html

[42] https://grafana.com. Your observability wherever you need it. [Online]. Available: https://grafana.com

[43] J. Boner, *Reactive Microservices Architecture.* 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly, 2016.

[44] B. Wilder, *Cloud Architecture Patterns.* 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly, 2012.

[45] J. Kreps, *I hearts Logs.* 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly, 2015.

[46] M. Al-Ameen and J. Spillner, "A systematic and open exploration of faas research," 2019. [Online]. Available: https://digitalcollection.zhaw.ch/handle/11475/17123

[47] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. E. Gonzalez, J. M. Hellerstein, and A. Tumanov, "Cloudburst," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 2438–2452, Aug. 2020. [Online]. Available: https://doi.org/10.14778/3407790.3407836

[48] E. van Eyk, A. Iosup, S. Seif, and M. Thömmes, "The SPEC cloud group's research vision on FaaS and serverless architectures," in *Proceedings of the 2nd International Workshop on Serverless Computing - WoSC '17.* ACM Press, 2017. [Online]. Available: https://doi.org/10.1145/3154847.3154848

[49] E. van Eyk, J. Scheuner, S. Eismann, C. L. Abad, and A. Iosup, "Beyond microbenchmarks: The SPEC-RG vision for a comprehensive serverless benchmark," in *Companion of the ACM/SPEC International Conference on Performance Engineering.* ACM, Apr. 2020. [Online]. Available: https://doi.org/10.1145/3375555.3384381

[50] G. McGrath and P. R. Brenner, "Serverless computing: Design, implementation, and performance," in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW).* Los Alamitos, CA, USA: IEEE Computer Society, jun 2017, pp. 405–410. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICDCSW.2017.36