Uber Push and Subscribe Database

June 21, 2016

Clifford Boyce
Kyle DiSandro
Richard Komarovskiy
Austin Schussler

Table of Contents

Introduction

Client Description

Uber is a multinational online ridesharing service based out of San Francisco, California. The service operates by users requesting rides through the Uber mobile app, which is then picked up by drivers employed by Uber, who drive their own cars. Uber operates similar to the way a taxi cab operates, but all pricing is handled through Uber, and not through the driver. Since Uber's launch in 2009, they have expanded to over 66 countries and 449 cities across the globe.

The Uber team currently manages a large distributed Cassandra NoSQL database that is used by many teams throughout Uber. The primary method of querying data is based on a "pull" model--any client wishing to see the latest version of the data must request the data from the database. It is often necessary for a client to know when the data has been changed, and this can be accomplished by pulling data frequently, but it places a much higher load on the database. Pulling data less frequently reduces the load on the database system, but significantly increases the chance that the client is looking at data that is no longer valid.

Product Vision

The goal of the field session project is to create a push and subscribe model on top of the existing database system using a distributed Kafka queue. Under this new model, a small amount of metadata will be pushed to the queue, and any clients subscribed to the queue will be alerted that changes have been made to the database, allowing clients to re-query the database only when they are guaranteed that they will get updated data back. This product should be a fully functioning system containing several clients, able to access and change data in the database, and any client also connected to the system being alerted of the change.

Requirements

This project contains two functional parts that need to be completed. The first is to create a working database system similar to the one used at Uber. The second is to implement the Kafka messaging system described in the *Product Vision* section above. This project also contains several non-functional requirements essential to a working product.

Functional Requirements

System Replication

- Cassandra database - This database contains all geospatial data in the system, and is accessed whenever an update is performed, or a request to view data is sent. This system is closely related to the Java Service Machine (described below) which is used to query and update the geospatial data, otherwise not readily compatible with NoSQL databases.
- Apache Thrift API - This system communicates between the client portal and the Java Service Machine, and is used to take the data given/required by the client and translate it into a form that is usable by the Java Service Machine. Thrift has its own language which is very robust in translating between multiple languages.
- Java Service Machine - This machine communicates with the Thrift web server and the Cassandra database. When writing data to the database, this machine uses an algorithm to assign a geospatial grid sequence to each point based off of the latitude and longitude of the point. This machine implements another algorithm which is used to determine what grid ID sequences lie within a rectangle of points. These two algorithms allow the service machine to expedite both pushes and pulls from the database in the most efficient manner possible.
- Client portal (Java, C#, JavaScript) - This will be a standalone standard/web application that allows the user to view and modify geospatial data as they see fit. This is where the implementation of Kafka will be seen, by updating other client applications when one makes and update that is common to both clients.

Kafka Messaging System

- Kafka queue - This is a system that takes the result of pulling data from the Cassandra database, and updates all relevant clients with the updated data. This process is the focus of the project, as it eliminates unnecessary queries to the database by letting clients know when data has been updated, rather than having clients regularly pull from the database even though the data may not have changed.

Non-Functional Requirements

- The ability to handle large quantities of data. This system should be able to handle large amounts of data because an actual implementation of this messaging system will be handling all clients on the system at Uber.
- The ability to handle large number of clients. The Uber team employs many people who are dedicated to handling reports from the public about old information, so any solution system should be able to handle a large number of clients at one time.
- High availability. This system should be able to work for extended periods of time under large loads, for it to be an effective tool that can be used by Uber.
- Exceptional partition tolerance. This system should be able to continue handling and processing data if a network partition begins to cause communication errors.

System Architecture

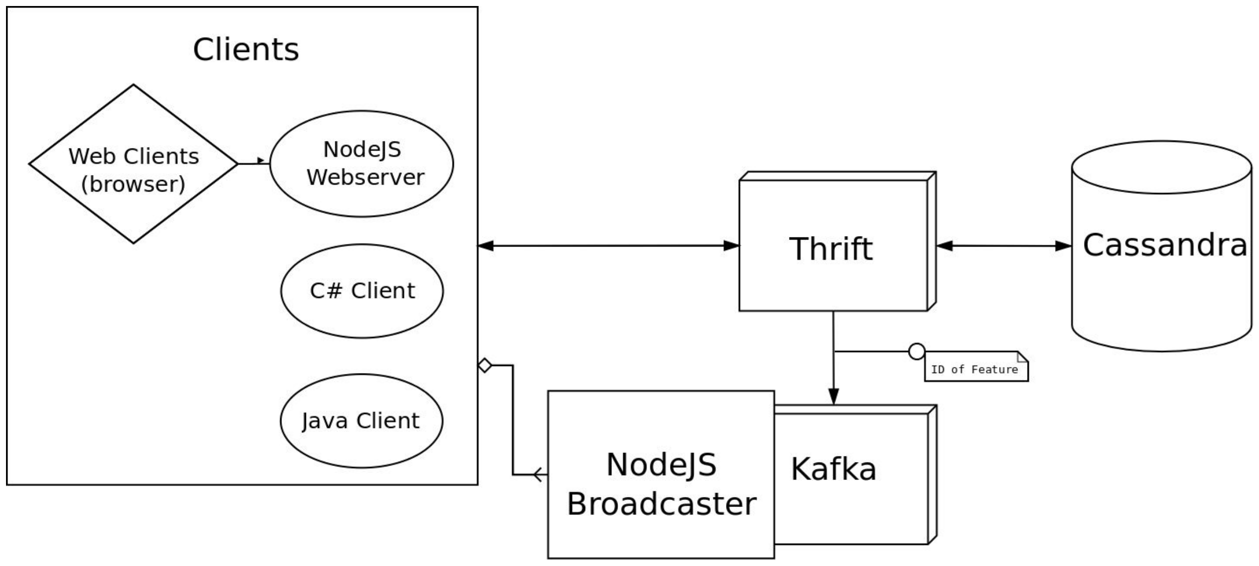


Figure 1: Software Architecture and Relationships

Cassandra

Cassandra is a NoSQL database that is highly scalable, performant, and available. Uber uses Cassandra in their technology stack, so it was required that it be used in this project. Cassandra is a good choice for this project because it handles large amounts of data at a high performance level. It offers linear scalability and protection against single points of failure because of its data redundancy model. Cassandra also works exceptionally well with data models that rely heavily on time reliant writes and reads. Rather than manually setup Cassandra clusters, a tool called CCM, Cassandra Cluster Manager was used. With CCM, deploying a Cassandra cluster took minutes, allowing for more time to develop the software. The Cassandra database communicates with the Thrift server as seen in Figure 1, and simply returns to the server whether or not the data transaction was successful.

Thrift

Thrift is a server that exposes a public facing API to Cassandra. Clients can connect to it and issue create, read, update or delete requests for features. The reason for using Thrift is it has a multitude of different client libraries in different languages. This allows for Java, C#, and JavaScript clients to all simultaneously connect to the Thrift service. Thrift is also part of Uber's architecture, thus it was required for this project. Whenever a request is issued to Thrift and it is successful, a message containing the ID of a feature is produced to Kafka, as seen in the center of Figure 1.

Kafka

Kafka is a high performance messaging queue. It stores the IDs of features that are produced by Thrift. Consumers can connect to it and retrieve the IDs of features that have been modified. The Kafka service is private with a public facing NodeJS server that clients can connect to and have the Kafka messages broadcasted to. This can be seen in Figure 1 where the ID is sent to Kafka from Thrift, and then forwarded to the NodeJS broadcaster.

NodeJS Broadcaster

The purpose of the NodeJS Broadcaster is to act as an intermediary between clients and Kafka. The reason for this is the Kafka client libraries in the various languages used (C# and JavaScript) are very poor in performance or even non-existent. By having a single server to connect to, clients can be easily coded in the native language using sockets. This client relationship can be seen in Figure 1, where it connects the Kafka message to all clients. The broadcaster was chosen to be coded using NodeJS because it is very fast and can handle many clients. Javascript, the language that is used in NodeJS, is easy to get started with and doesn't take a lot of time for development. Also, a developer on the team is fluent with it.

Clients - Web Server

The web server is a single client that connects to Thrift, through the use of Thrift generated code, and issues requests to it. It also connects to the NodeJS broadcaster to receive incoming updates from Kafka. Browser clients connect to the server and issue request to it that are then used to build a map with features on it. The web server connects to clients using Socket.io to allow for a real-time connection. The server pushes any messages from Kafka to the client through this connection and allows the client to determine what to do with the information. The web server was also written using NodeJS and the Express library. NodeJS was chosen because it was already used in the Broadcaster and it allowed for high developer productivity.

Clients - Java, C#, and Others

The Java and C# clients connect to Thrift, using code generated by Thrift. They also connect to the NodeJS Broadcaster using the built in socket APIs. These clients can essentially do anything that they are programmed to do and will receive updates on features from Kafka. Any language that Thrift supports can be used to make a client. Java and C# were two languages that were specified by Uber to be used as clients; however, any supported language can be used as a client with this architecture.

Clients - Google Maps Website Client

For demonstration purposes the Javascript/CSS/HTML client was the most robust client made. Using the Google Maps API and their various source code demonstrations, the website was able to implement almost any type of map configuration that best suited the project's needs. The JavaScript web client connects to NodeJS which communicates with Thrift when transferring or pulling data from the Cassandra server. NodeJS is listening to the Kafka server for updates, and when an update is received it will send out that update with a feature ID to the Javascript browser. The website is designed to check any update it receives against it's currently plotted points. If the point exists then the website will remove that individual point and move on to the redrawing step. If the point does not yet exist, or hasn't been loaded into the client's existing points, then it also moves on to the redrawing step. In the redrawing step a getCorners function is called which checks the client's google map window for that point that was just added. If the point is within the window the function will redraw that individual point without affecting any prior points that the client has loaded or that are loaded on the client's screen. If the point is not within the window the function will not draw that point because it is an unnecessary pull from the database.

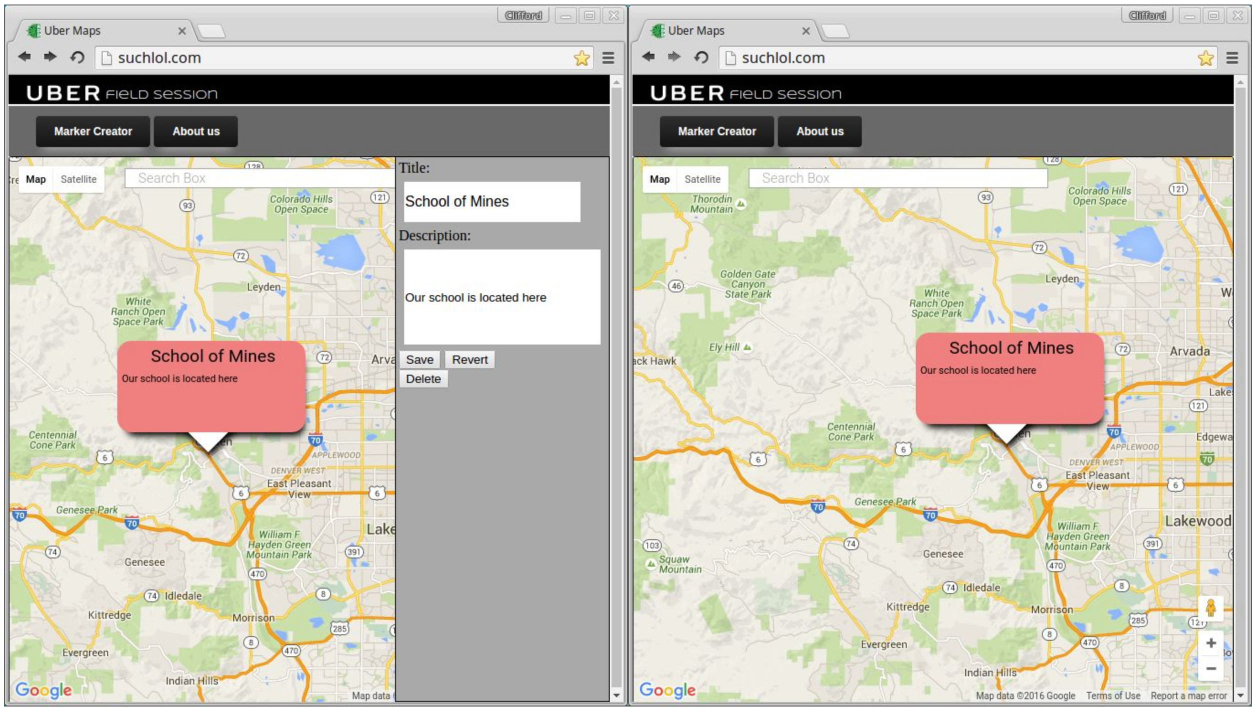


Figure 2: Website Client Visual

Technical Design

Flow of Information

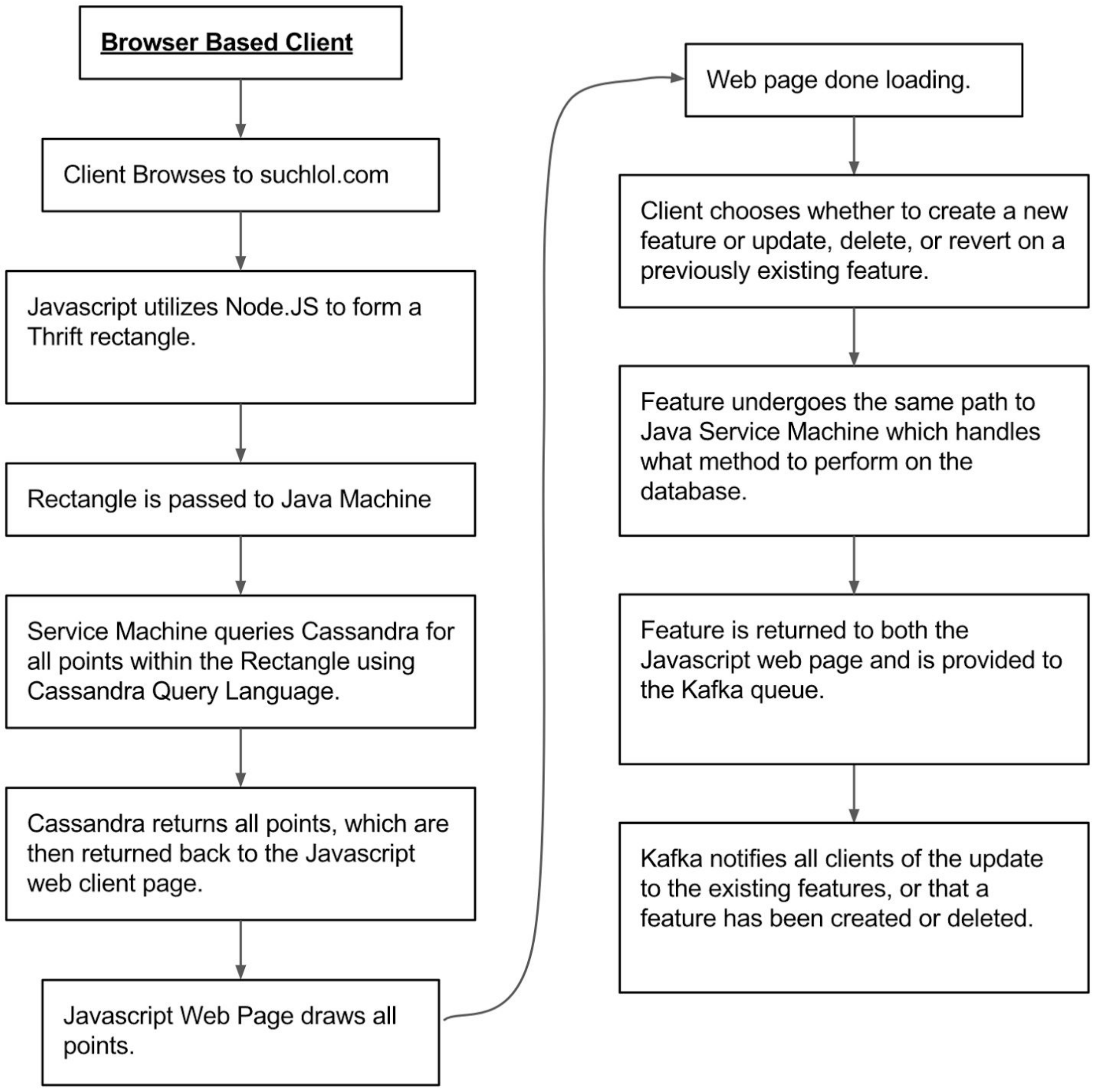

Browser Based Client
Client Browses to suchlol.com
Javascript utilizes Node.JS to form a Thrift rectangle.
Rectangle is passed to Java Machine
Service Machine queries Cassandra for all points within the Rectangle using Cassandra Query Language.
Cassandra returns all points, which are then returned back to the Javascript web client page.
Javascript Web Page draws all points.
Web page done loading.
Client chooses whether to create a new feature or update, delete, or revert on a previously existing feature.
Feature undergoes the same path to Java Service Machine which handles what method to perform on the database.
Feature is returned to both the Javascript web page and is provided to the Kafka queue.
Kafka notifies all clients of the update to the existing features, or that a feature has been created or deleted.

Figure 3: Flow of Information Through the System

Figure 3 illustrates the process of using the browser based JavaScript client. The first step involves the client navigating their web page to suchlol.com, the website used to host this project. When loading the page, the JavaScript client sends the top left and bottom right points of the area that is being viewed to Node.JS which then utilizes the Thrift API to create a rectangle. This rectangle is then passed on to the Java Service Machine which finds all of the quadkeys that have any area within the rectangle that is being queried. The Java Service Machine then prepares a query statement which is then run on the Cassandra database. Cassandra returns all of the points back to the Java Service Machine. The Java Service Machine then creates the Thrift feature and returns this feature back to the web client. The web client then creates all of the features so that they are visible and at this point the web page is done loading. At this point the client can view existing features by clicking on them or they can perform updates, deletes, and reverts on previously existing features. When the Java Service Machine makes the update to Cassandra, the update is also passed to the Kafka queue. Kafka then distributes the message notifying all users that the change has been made.

Geospatial Database Schema

Table 1: Cassandra GeoSpatial Table Schema

Grid(Primary Key)	Feature ID(Clustering Column)	Payload	Point X	Point Y	State
0130232	fsefsd-sdfsfa-3osjdf-sdfw38	{JSON}	13.123	12.321	1

Our schema has a single model that is suited specifically for a geospatial database. Each geospatial feature contains a complete set of data corresponding to the data table model in Table 1. Queries in Cassandra are optimized when utilizing the primary key as the search parameter. Due to this property, we assigned the Grid to be the primary key in our table which will allow the Java Service Machine to retrieve all of the features (points) within a certain QuadKey area in the most efficient way possible. Additionally, we assigned the clustering column to be Feature ID time signature. These two design decisions group points that share QuadKeys together, and then further organize those points based off the time signature.

Design Decisions

Cassandra Database

In order to closely mimic the system that Uber has set up, our team was required to use a Cassandra NoSQL database which is currently implemented in the Uber system model. Rather than go through the time consuming process of setting up a complete database cluster with multiple nodes, our team opted to use Cassandra Cluster Manager. CCM is a tool that enables users to quickly and efficiently set up a cluster on a local computer or server. This implementation decision was made to reduce time spent on areas of the project that were not the focus of the project.

Google Maps API

We decided to implement the Google Maps API because it offered a simple, well-documented, and lightweight map interface. One requirement that our team needed from the API was the ability to place features on the map. In our case, Google Maps allowed point placement and point modification features to easily be implemented on their maps interface. For this section it was quite preferable to use existing code in some areas. The team member placed on this section of the project had limited knowledge of Javascript, HTML, and CSS, so reusing code where possible is likely more reliable and allowed for quicker implementation. It also allows for the programmer to have limited knowledge of exactly how a programming language behaves, yet still achieve the desired result.

JSON

We opted to use the JSON data format for exchanging data because it is extremely lightweight and the value to name structure that the data format implements is extremely easy to read, write, and manipulate.

NodeJS

In order to ease the connection between our client applications and the Kafka queue, we decided to use NodeJS. The Kafka distributed messaging service libraries are poorly supported or non-existent for the languages that we were writing the client applications in. We decided to use NodeJS because it allowed us to skip using these libraries and instead connect our client applications to a web socket. We chose NodeJS because it is extremely fast, can handle high loads, and is written in Javascript, which one of our team members is proficient in.

Results

Overview

The system we created meets the requirements listed above. Our system contains clients in several languages such as JavaScript, Java and C#, which are able to make various operations on the data in the geospatial database. Once an operation is completed, a message is sent to each client connected to the system that a feature was updated. For demonstration purposes, the JavaScript client is a fully functioning web site, which allows a user to create a feature with a title and description. This web site has the Google Maps API included, making it more realistic to a system that Uber could actually use. The Java and C# do not have UI components, but do have processes in place to demonstrate the message system is functioning correctly. The Java and C# clients are mainly for proof of concept that the system can be applied to a variety of clients running different languages (Uber came from Microsoft, who ran a lot of C# so compatibility with legacy systems is important, hence C# client proof of concept).

Lessons Learned

Technical

- Websockets are extremely useful. When creating a system that contains several languages, it can be difficult to have them all communicate. In this project, the issue comes down to having clients in several languages, and them needing to take information passes to them, and then do something with that data. This was accomplished using web sockets, as each language has a socket type class which allows data to be read from a socket on a server.
- Javascript is a difficult language to grasp at first. Our project contains a web server that communicates with the clients, and this server runs JavaScript. JavaScript as a language is different in terms of syntax and usage than the object oriented languages we have learned thus far. If we tried to get too in depth in understanding about the specific syntax, we would just confuse ourselves, so just writing what needed to be wrote turned out best for the JavaScript portions of the project.
- Node.js libraries are handy. This project heavily uses several node.js libraries to complete what was needed. A library such as the dedicated Kafka library of node.js was surprising easy to implement, because of the complexity of JavaScript as mentioned above.

Non-technical

- Root access is needed. Some of the early work on the Cassandra database was done on the Alamode machines, as such there were issues related to operations that could not be installed/performed, making it more difficult to accomplish what was needed. This was solved by using personal machines that gave this access.
- Long work periods accomplish a lot. In the beginning of the project, a lot of work was done individually. We all started working on our own parts, but when it came time to start thinking about integrating them all, it became a lot clearer to everyone how their part was going to fit into the project, and it helped everyone understand the nuances of the project better. These longer work sessions also made the team a lot more productive in terms of raw code output, as each member was focused on a problem, and instead of getting stuck and spending time finding the solution online, as another member could jump in and help, increasing productivity.

Testing Summary

Currently the Javascript web browser client works in Google Chrome, Mozilla Firefox, Safari, and Internet Explorer. The site has many hours of fiddling and testing while constantly being updated. Most of the minor bugs or usability annoyances have been removed and overall the user experience has been improved. With all of the testing there have been many times that the Javascript imploded because of improperly called functions or variables and not once has any part of the back end crashed. With that being said the front end never crashed either but would not do what was being requested. Other testing would include the website's efficiency at pulling points from the database, which was really one of the main reasons this project was brought up. The website is designed to check any update it receives against it's currently plotted points. If the point exists then the website will remove that individual point and move on to the redrawing step. If the point does not yet exist, or hasn't been loaded into the client's existing points, then it also moves on to the redrawing step. In the redrawing step a getCorners function is called which checks the client's google map window for that point that was just added. If the point is within the window the function will redraw that individual point without affecting any prior points that the client has loaded or that are loaded on the client's screen. If the point is not within the window the function will not draw that point because it is an unnecessary pull from the database.

Appendix

Libraries

Github Repository

- https://github.com/cboyce5/UberFieldSession

Cassandra

- http://cassandra.apache.org/

Thrift

- https://thrift.apache.org/

Kafka

- http://kafka.apache.org/

NodeJS

- https://nodejs.org/en/

Client Figures

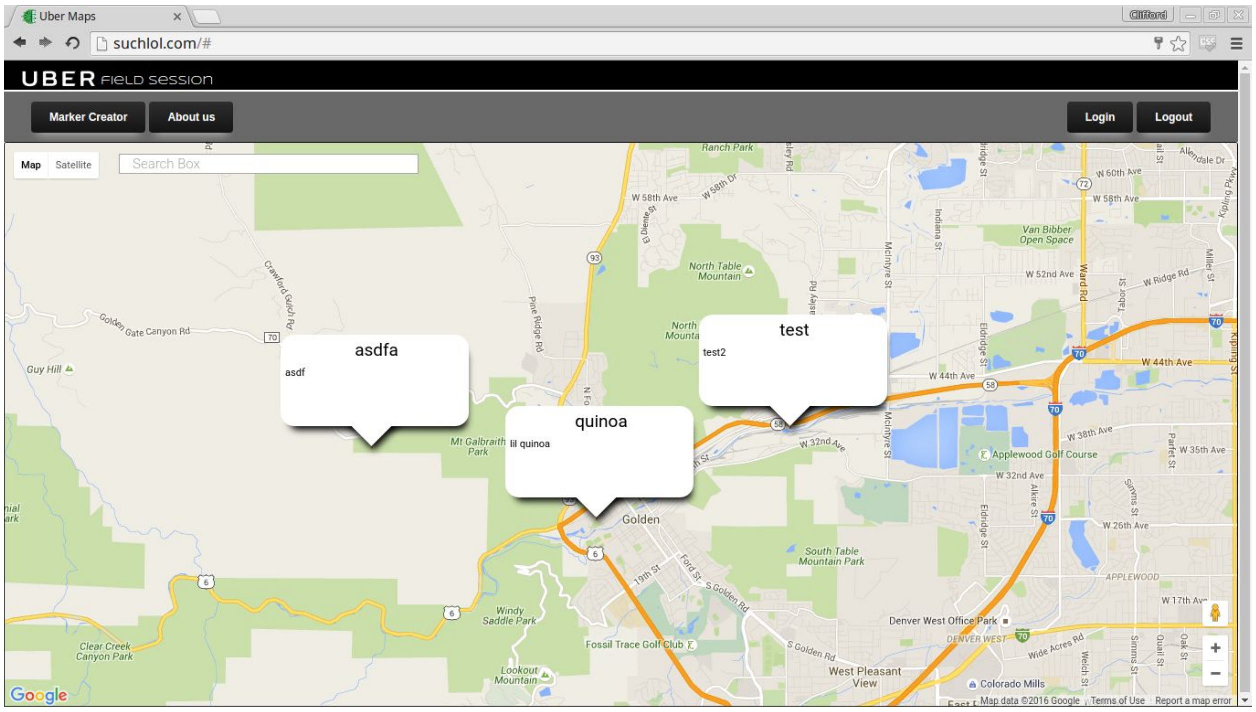
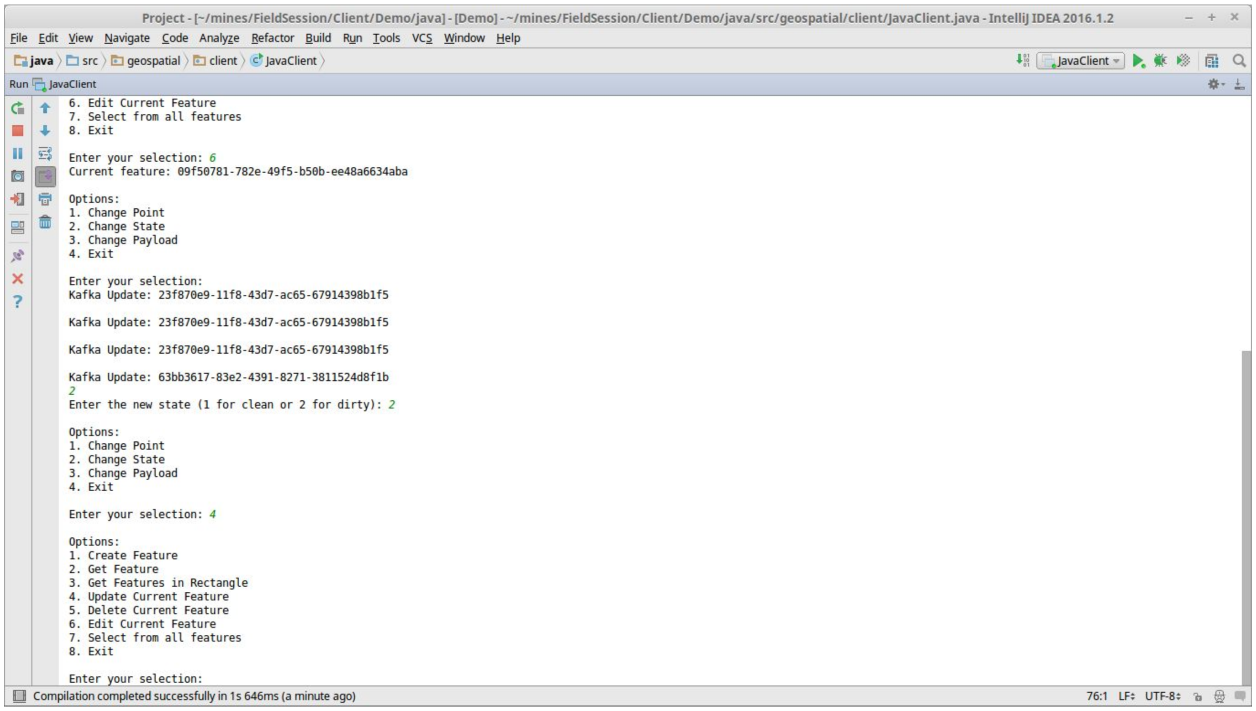


Figure 4: JavaScript Client Demo

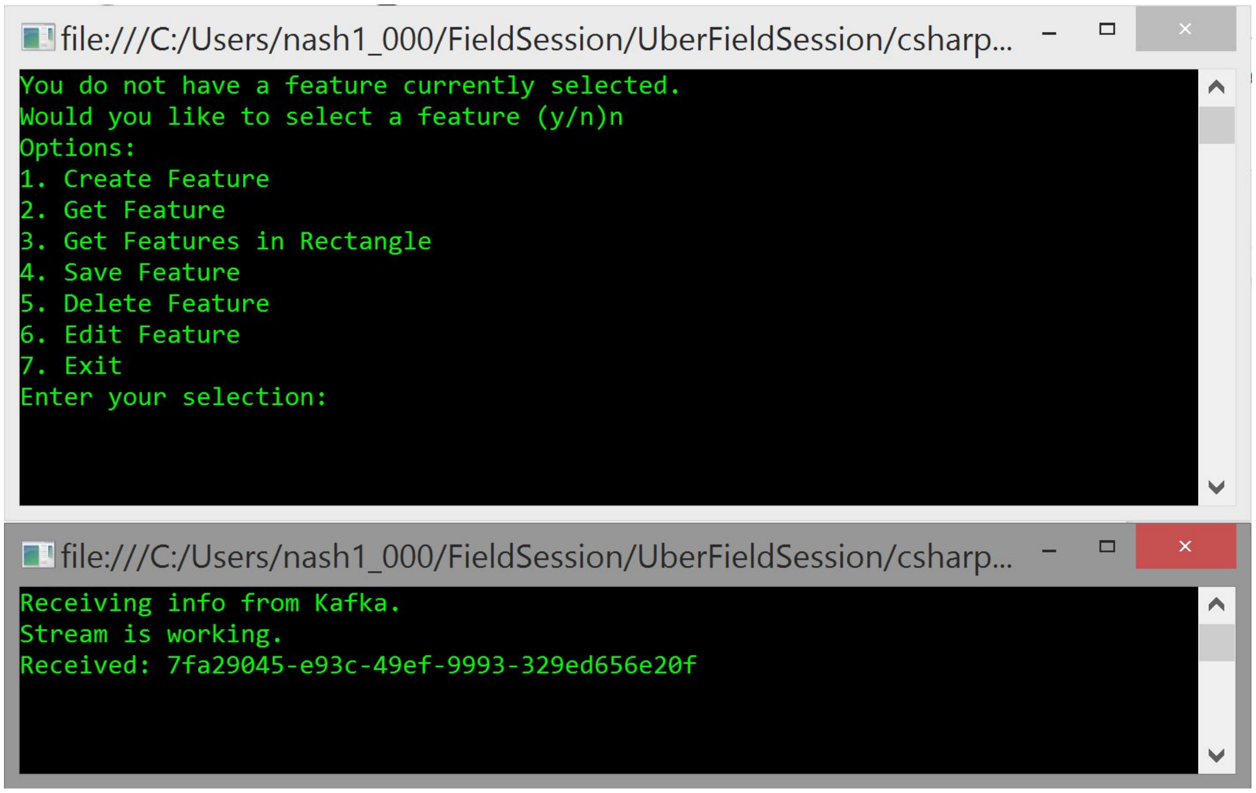


Project - [~/mines/FieldSession/Client/Demo/java] - [Demo] - ~/mines/FieldSession/Client/Demo/java/src/geospatial/client/JavaClient.java - IntelliJ IDEA 2016.1.2
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
java 〉 src 〉 geospatial 〉 client 〉 JavaClient
Run JavaClient

```
6. Edit Current Feature
7. Select from all features
8. Exit

Enter your selection: 6
Current feature: 09f50781-782e-49f5-b50b-ee48a6634aba

Options:
1. Change Point
2. Change State
3. Change Payload
4. Exit

Enter your selection:
Kafka Update: 23f870e9-11f8-43d7-ac65-67914398b1f5

Kafka Update: 23f870e9-11f8-43d7-ac65-67914398b1f5

Kafka Update: 23f870e9-11f8-43d7-ac65-67914398b1f5

Kafka Update: 63bb3617-83e2-4391-8271-3811524d8f1b
2
Enter the new state (1 for clean or 2 for dirty): 2

Options:
1. Change Point
2. Change State
3. Change Payload
4. Exit

Enter your selection: 4

Options:
1. Create Feature
2. Get Feature
3. Get Features in Rectangle
4. Update Current Feature
5. Delete Current Feature
6. Edit Current Feature
7. Select from all features
8. Exit

Enter your selection:
```

Compilation completed successfully in 1s 646ms (a minute ago)
76:1 LF: UTF-8:

Figure 5: Java Client Demo



file:///C:/Users/nash1_000/FieldSession/UberFieldSession/csharp...

```
You do not have a feature currently selected.
Would you like to select a feature (y/n)n
Options:
1. Create Feature
2. Get Feature
3. Get Features in Rectangle
4. Save Feature
5. Delete Feature
6. Edit Feature
7. Exit
Enter your selection:
```

file:///C:/Users/nash1_000/FieldSession/UberFieldSession/csharp...

```
Receiving info from Kafka.
Stream is working.
Received: 7fa29045-e93c-49ef-9993-329ed656e20f
```

Figure 6: C# Client Demo