City Crime Reporting Network:
Project Report

Joe Jones (s2990652), Charles Randolph (s2897318), Barnabas Busa (s2922673)
Group 4

April 2, 2018

Contents

1 Project Description

Case Study

Some cities make use of networks of listening devices to help authorities locate and respond to crime. An example of this is Shotspotter, which is installed in Los Angeles County, California. Shotspotter uses microphones mounted on utility poles and on the roofs of buildings to listen for gunshots. When a gunshot is detected, the system attempts to pinpoint the signal origin and report the event to the authorities.

Objective

Our project will provide a framework that is more generalized than Shotspotter, allowing the system to pick up and attempt to triangulate other signals such as screams or alarms (it is of course adequate for gunshots as well). It additionally allows users of the system to track events of interest through both a simple web interface as well as a client application, and performs several clustering and filtering tasks to present users with useful information.

2 Architecture Description

Overview

The system architecture is composed of the following components, each of which will be covered in more depth in subsequent sections.

1. **Sensor**: A sensor is a fundamental device to our system and is responsible for generating signals and other useful information for our higher level components to process. Sensors are deployed in large numbers and in strategic locations in order to maximize the effectiveness of our system.

2. **Server**: The server is our main data-processing entity. It receives communications from the sensors directly and performs both filtering and sorting in order to produce meaningful information for end users. The server dispatches important events to our message exchange, and provides a basic web page for monitoring sensor activity.

3. **Message Exchange**: In order to consolidate events and lighten the load on sensor servers, the Message Exchange is a message broker that passes messages from the server to all subscribed clients. It allows client to pull information from it concerning events, and provides a secondary location for events to be held if the server goes offline.

4. **Client**: The client is a desktop application that presents users with a basic way to view ongoing activity in the sensor network. It is merely a visualization tool for users of the system.

2.1 Sensor

A sensor is typically a microphone, camera, or other device specialized in monitoring some phenomena of interest. In our case study, this particular interest was sound. However, because we'd like to keep our system general and not sound-specific, we don't mandate that it must monitor anything in particular. A sensor, therefore, is a device that transmits a datatype (defined in `data.go`) containing the following fields:

1. **Id**: A unique identifier for the sensor. In our case, this is an integer.

2. **When**: A time-stamp. When anything is sent to the server, it must time-stamped with the time at which it detected the signal. We store this as a `long` (64-bit integer).

3. **Location**: A sensor's location is critical to it's usefulness in data-analysis. Therefore, a location (described as a `float` pair) is bundled in any message sent to the server.

4. **Signal**: A signal is rather difficult to describe given a wide variety of potential applications. We describe the signal as an array of 4-bit floating points (`double`).

Sensors deploy messages (which we denote as "Grams") to the sever through **sockets** in a plain text **JSON** format using **TCP**. Whenever a sensor is triggered, a **socket** connection is opened to the server. The data (described above) is then sent over, after which the server closes the socket and the sensor resumes its duties. The choice of the **TCP** protocol is done to ensure reliable delivery of the data. Because we don't expect sensors to be constantly firing (if tuned to detect gunshots for instance), this protocol choice makes sense. Finally, since we don't have access to a large array of sensors, we simulate these in our project through a custom built Java app. The editor allows virtual sensors to be placed over a geographical area and triggered through the use of simulated waves. The app opens multiple connections to the server to simulate sensors being triggered.

2.2 Server

The server is by far the most complicated component of our system, and is deployed with the sensors in a classical **star topology** (all connections lead to the sever). While we considered P2P early on as a potential way to circumvent the need of a central server, we felt that the requirement for sensors to be low-powered, embedded devices, combined with the need for significant processing power and aggregation, necessitated the **star topology** choice. The server itself performs the following **primary** tasks.

1. **Filtering** of unimportant sensor data.
2. **Clustering** of geographically and chronologically related sensor data.
3. **Generation** of important/significant events.

It also performs the **secondary** tasks of:

1. Dispatching events to the message broker.
2. Hosting a web-page for remote monitoring.

Filtering

The server `main` routine is tasked with waiting for incoming connections. When a socket is opened, it launches a `RequestHandler` routine to handle the connection. The handler attempts reads in data from the sensor, close the socket once done, and deserialize the data. It then examines whether the data is interesting enough to be worth considering. This is done (in our case) by looking for whether any values in the data Gram's signal component is over a defined threshold. If it is, the gram is sent on for further processing. It is otherwise discarded.

Clustering

All data grams which were not filtered get piped to a dedicated routine within the handler. This routine, named `QueueHandler`, maintains a queue of clusters. A cluster is essentially a **set** of related grams. It is defined in `data.go` and contains the following fields:

1. **Id**: A unique 64-bit integer identifier (timestamp in nanoseconds).
2. **Updated**: A 64-bit integer (timestamp in milliseconds) containing the time of the last event added to it.
3. **Members**: An array of **Gram** data types.

The `QueueHander` routine first begins by removing all expired clusters. Clusters expire if no new grams are added to them within a user-defined set of time. This is designed to organically prune the queue of outdated clusters and keep it fresh. The `QueueHandler` then attempts to find a matching cluster for the newly arrived gram. A gram "fits" or "matches" a cluster if it is geographically nearby to any of its other members. This is determined in our case by *euclidean distance* and compared against a user-defined radius.

If a gram matches a cluster, it is added to the cluster and the cluster's last updated time is renewed. Otherwise, the routine creates a new cluster and placed the gram within it as the sole member.

Generation

Events are generated within the `QueueHandler` routine. An event of significance is determined using a user-defined threshold of sensors. When a cluster reaches a critical mass of sensors (say, perhaps, five), then it becomes an event. This means that the cluster is sent from the `QueueHandler` routine for dispatch to the message exchange. It is also added to a list of all significant events so far to be presented over the web-interface.

Secondary Tasks

The server **sends messages** to a **RabbitMQ** message broker for later collection by clients. It also provides a means for users to view it's online status through the means of a basic **RESTful** web server. This web-server simple serves users sending **GET** requests with a basic page concerning the status of the server and recent events.

Language and Structure

The server is written in Go, and makes good use of Go's well known **goroutines** to handle the job of opening request handlers, managing the queue, and running the web-server concurrently. When first starting up, the server initializes a set of channels for inter-routine communication, and then launches each task with their appropriate channels. As an after-effect of using Go, the server communicates almost entirely through the use of inter-process channels (similar to UNIX pipes). This avoids the trouble of mutexes and shared memory issues, although it does require a change in programming style.

2.3 Message Exchange

The message exchange is an intermediary between our server and all clients wishing to receive event information. It's purpose is to both provide a means of robustness for generated data and to relieve the server from the task of sending events to clients. For our exchange, we chose to use **RabbitMQ**. It's Go library was installed in the server, and accompanying Java JAR archives in our client application. The exchange itself is a **fanout** exchange. This type of exchange receives messages and publishes them to listening clients. This allows our client applications to tune into the exchange to receive real-time sensor activity from the server.

2.4 Client

Client

The client is a Java application which connects to our **RabbitMQ** exchange. It simply listens for published messages using the bundled RabbitMQ Java library methods, and then presents events to the user in a graphical interface with some visualizations. The messages received by the client are JSON encoded `Cluster` objects generated by the server. They contain (as detailed in more depth earlier) a list of triggered sensors among other things. The client displays to the user a 2D graphical interface and the location of the triggered sensors on top of it. The client also computes a approximation of the signal origin.

Signal Approximation

The client will simply receive events from a sensor server and present it to the end user. These events can be used to help coordinate authorities in responding to potential crime. The client has little other involvement in the system other than being the end of the line for event information. The client calculates the approximate location of the signal by taking an average of the locations of the different sensors in the cluster. We apply a weight to each of the sensor locations by looking at the relative times each sensor receives a signal, with the sensors that received the signal earlier having a stronger weight. As the speed of sound is constant (ignoring obstacles such as buildings), this provides a good estimate of the original location. The approximate location of the signal is marked with a different icon in order to distinguish it to the user.

3 Code Structure

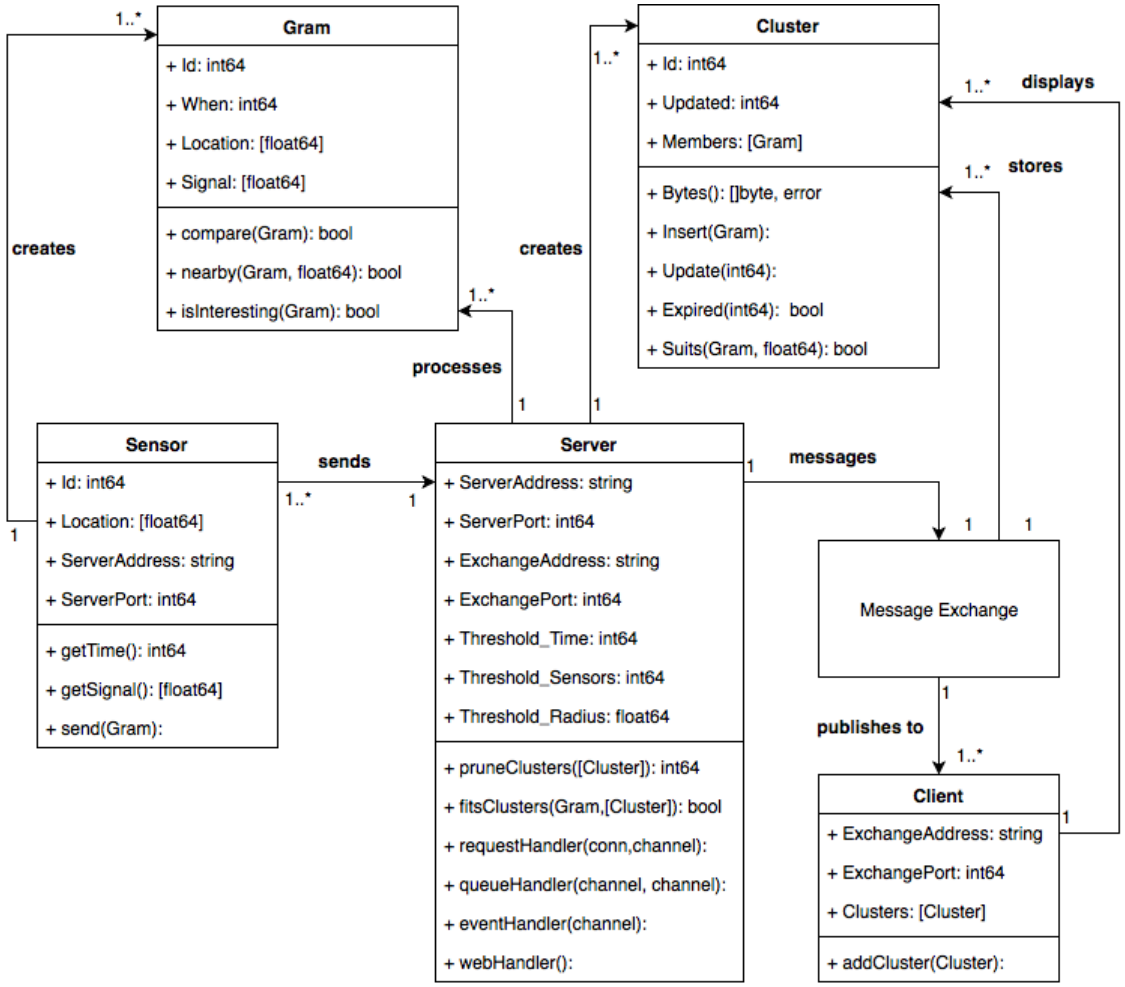
This project uses two main programming languages, those being Go and Java. For this project, the Go language's organization style has been used. The main project repository consists of various packages used by the Go build system. The packages are detailed below:

- **data**: This folder contains the `data.go` package. This package provides definitions for the `Gram` and `Cluster` types used in the server, as well as various methods on these types.
- **github.com**: This folder contains the Rabbit MQ libraries used with the server.
- **java**: These contains all Java applications involved with the project. Both the client Java app and custom editor app are located in this subdirectory.
- **queue**: This folder contains the `queue.go` package used with the server.
- **server**: This folder contains the `server.go` program.
- **web**: This folder contains the `web` package used by the server to present web-pages. It also contains all supporting methods.

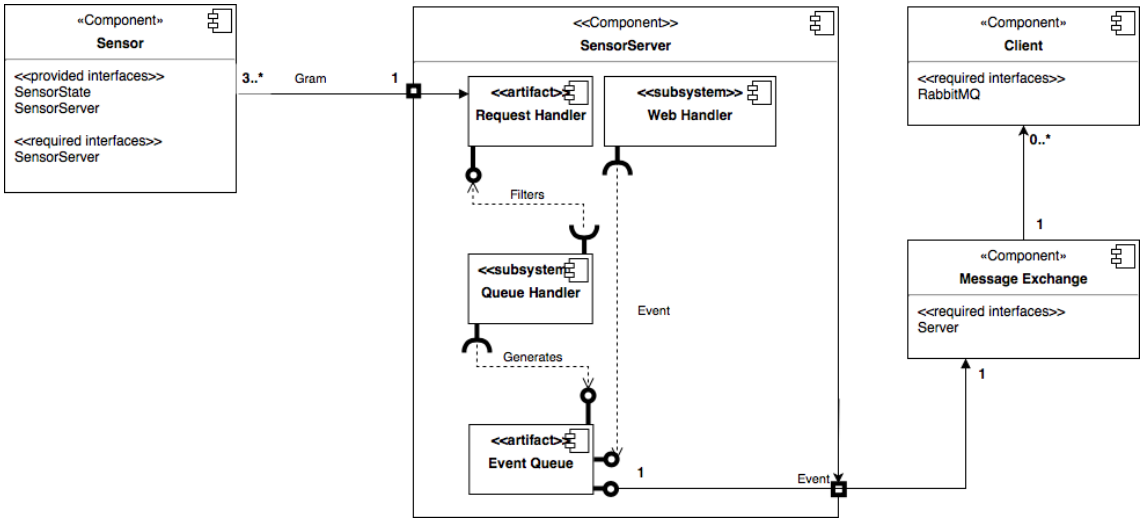
The project repository is available at: https://github.com/Micrified/Spot

4 Appendix

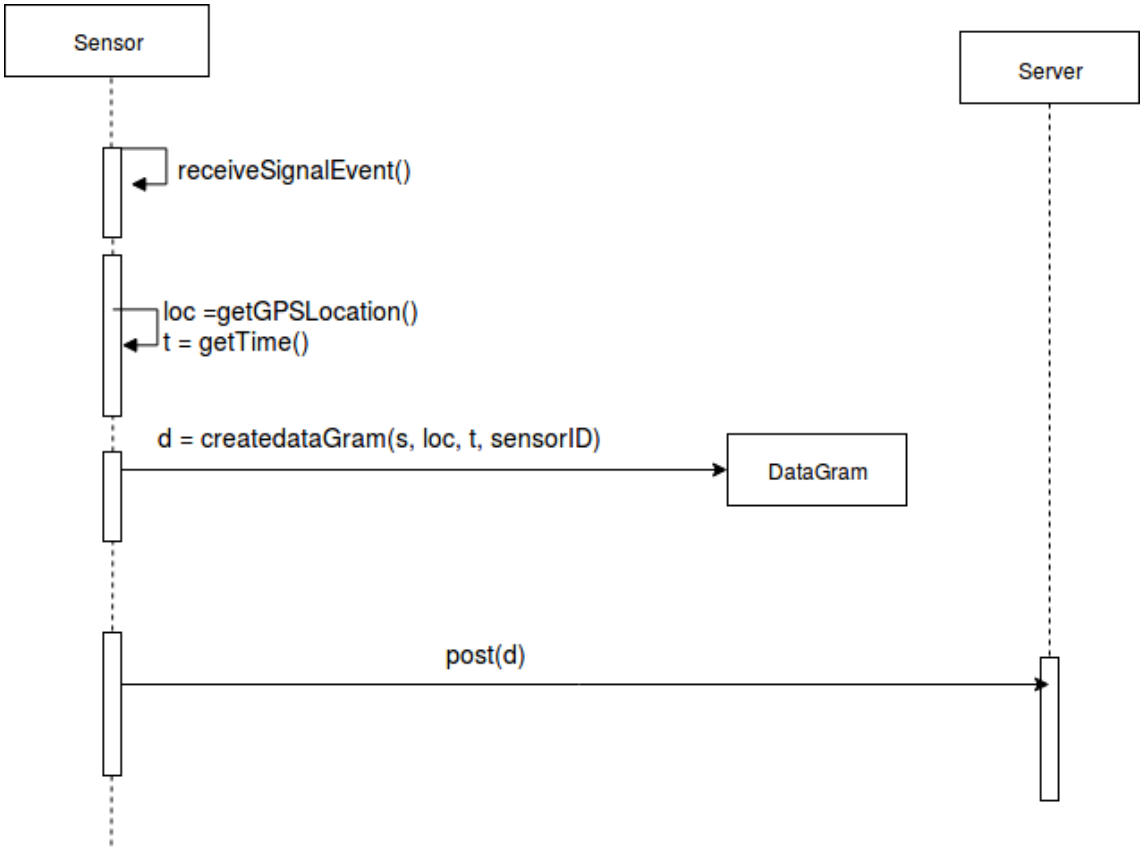
4.1 Class Diagram



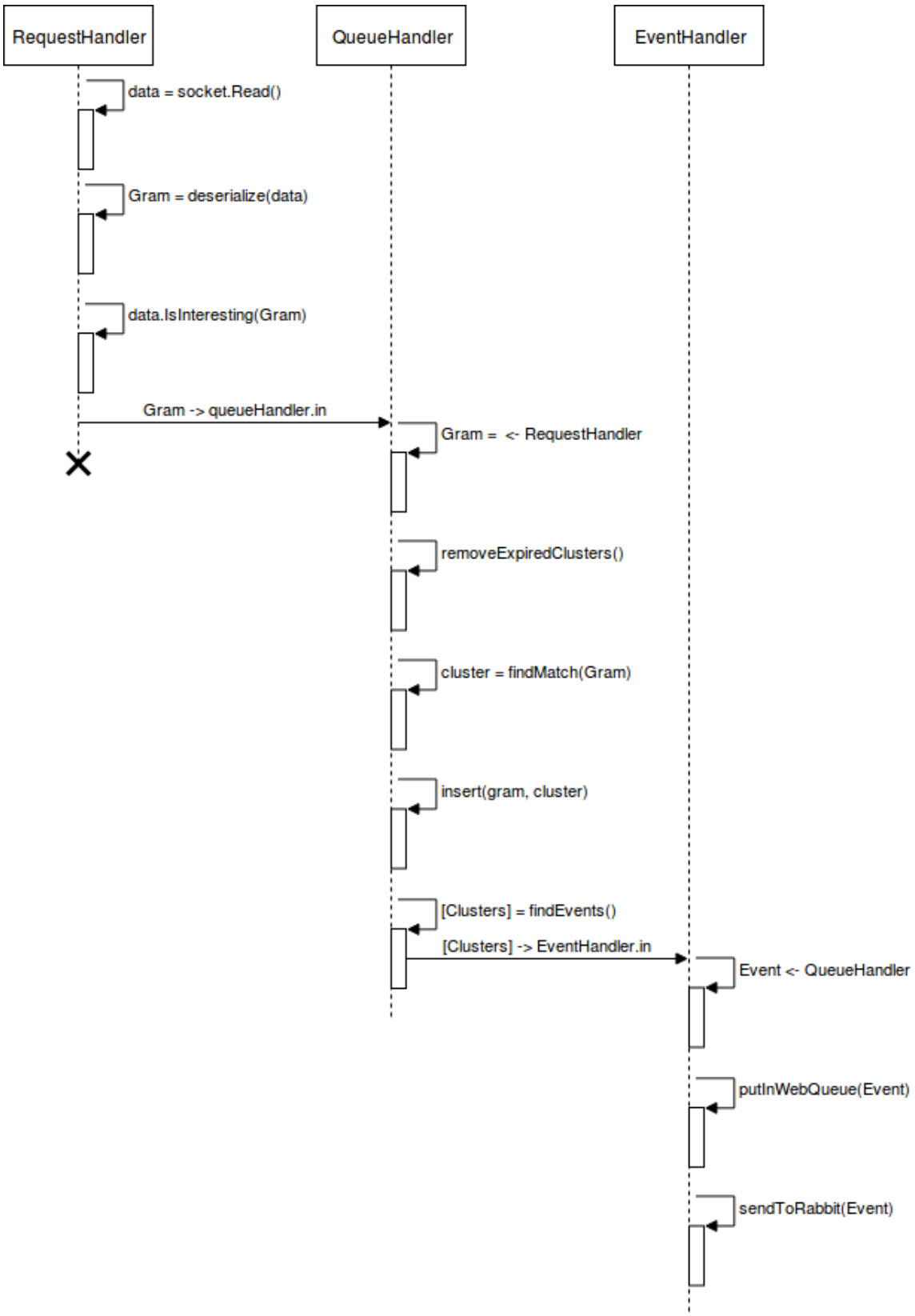
4.2 Component Diagram



4.3 Sensor Sequence Diagram

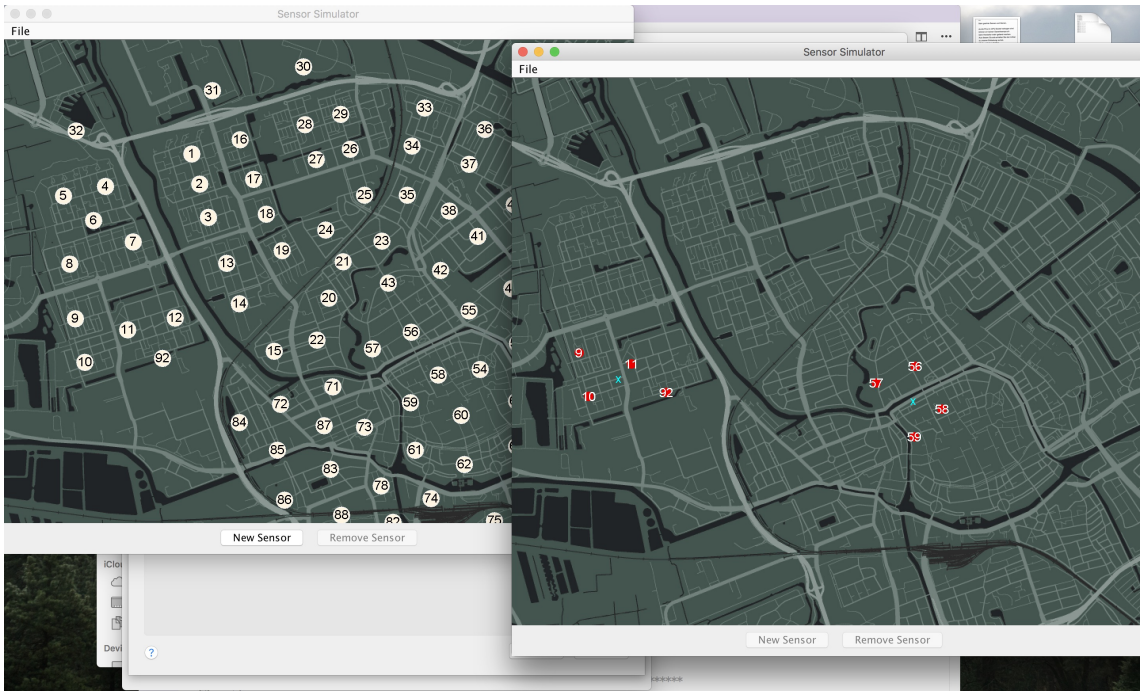


4.4 Server Sequence Diagram

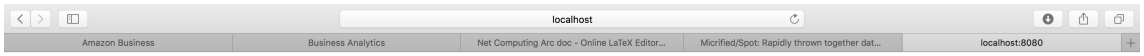


5 Screenshots

5.1 Application in use



5.2 Web service



Incidents

Time	No.Sensors	Appoximate Location
2018-04-02 22:51:22 +0200 CEST	6	(556,500)
2018-04-02 22:51:38 +0200 CEST	5	(543,499)
2018-04-02 22:51:45 +0200 CEST	5	(564,305)
2018-04-02 22:52:39 +0200 CEST	5	(450,348)
2018-04-02 22:52:58 +0200 CEST	6	(442,329)
2018-04-02 22:53:20 +0200 CEST	3	(142,380)