Design and Performance Modelling of Cloud-Based Sensing-as-a-Service Platform

Student Name: Amit

IIIT-D-MTech-CS-MC-12062
July 4, 2014

Indraprastha Institute of Information Technology
New Delhi

Thesis Committee
Vinayak Naik, IIIT Delhi (Chair)
Tridib Mukherjee, Xerox Research, Bangalore (Co-Advisor)
Anirban Mondal, Xerox Research, Bangalore (External Examinar)
Sambuddho Chakravarty, IIIT Delhi (Internal Examinar)

Submitted in partial fulfillment of the requirements
for the Degree of M.Tech. in Computer Science,
with specialization in Mobile Computing

Keywords: Cloud-Based Sensing-as-a-Service, Performance modelling, Benchmarking

Certificate

This is to certify that the thesis titled **"Design and Performance Modelling of Cloud-Based Sensing-as-a-Service Platform"** submitted by **Amit** for the partial fulfillment of the requirements for the degree of *Master of Technology* in *Computer Science & Engineering* is a record of the bonafide work carried out by him under my guidance and supervision in the Mobile Computing group at Indraprastha Institute of Information Technology, Delhi. This work has not been submitted anywhere else for the reward of any other degree.

Professor Vinayak Naik
Indraprastha Institute of Information Technology, New Delhi

Abstract

The area of sensing has witnessed significant development over the past two decades. Body of work on sensors has focused on individual sensors, sensor networks, in-network data processing and distributed query processing. Along with using dedicated sensors, smartphones are being used for sensing, because of available on-board storage and computation capabilities. But gathering all the sensor data and offering *Cloud-based Sensing-as-a-Service(CSS)* has not been explored. In a CSS platform, data from sensors is collected on a cloud-based server, then collected data is processed to provide services to the consumers. In CSS, the whole infrastructure of sensor data collection, followed by data processing to provide services to consumers, is offered as a service on a cloud. *CSS* opens up considerable avenues for research pertaining to new and emerging commercial applications.

Two folded contribution of this work can be described as below:

1. Architecture of Cloud-based Sensing-as-a-Service platform
2. Performance Modeling of designed platform

The CSS platform consists of APIs for sensor data collection, APIs for using platform services, primitive queries, abstract queries and set of modules to process sensor data to provide consumer services. Platform services can be in terms of alert notifications for consumer subscriptions or consumer initiated queries for particular type of event(s). The concept of abstract queries abstracts over underlying platform technical details, therefore allows end consumers to frame queries in user friendly manner. Each module processes one corresponding abstract query, which includes generating equivalent primitive queries and processing database results.

Performance modeling, of a CSS platform, is about modeling usage of platform resources. Modelling is done in terms of incoming event rate and total number of subscriptions for platform services. A benchmark application, for measuring resource utilization, has been developed. The result of this benchmark application is resource usage data for complete duration of benchmarking. This result can further be used to come up with system design policy, about using a particular alert notification mechanism, for a CSS kind of platform.

Acknowledgments

I thank Dr. Vinayak Naik for his support and guidance. He has been very supportive throughout this work and never lost his faith in me. I have got all the necessary help from him every time I approached him.

I thank Dr. Anirban Mondal and Dr. Tridib Mukherjee from Xerox Research Center India, for their support during this work. Their suggestions have been very helpful for me throughout this work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Sensors have many applications which make them an important part of our life. For example, light sensors are used on streets to switch on/off street lights, water sensors are used in water tanks to determine current water level, temperature/heat sensors are used to identify activity in some place. These sensors can communicate over internet to pass this information to remote locations which make them suitable for remote sensing applications. Satellite images are one such example of it. All these sensors are dedicated devices so one need to deploy them to desired locations to get sensor data. Deployment of sensors is costly, time consuming process and sometimes are vulnerable to theft and damage. This scenario has changed significantly with the evolution of smartphones which contain many sensors on-board. Modern day smartphones have sensors like camera, accelerometer, light sensors, proximity sensors and gyroscope which can be used for various sensing applications. Wide market penetration of smartphones is a big advantage of using smartphones as sensors. All one need for sensing is, a smartphone and an application on it which can sense data using built-in sensors. Smartphones have storage and computation capacity, so sensed data can be processed and results can be stored on smartphone itself.

The area of sensing has witnessed significant development. Sensing started with dedicated sensor devices e.g., light sensors, temperature sensors and then smartphone based sensing applications have emerged in market. Work around sensors so far, has been around individual sensors, sensor networks. Several bodies of work have emphasized importance of in-network processing. In in-network data processing, instead of sending all results to the server, the end nodes compute intermediate results, merges them with other results and then these aggregated results are sent to the server. This paradigm saves network bandwidth and reduces server-side computation.

Query processing is another area that has been studied widely. In query processing [9], none of the sensor uploads it data to server. Each node has its own data stored with it. Whenever some query comes in, from some consumer, it is routed to respective nodes for processing. These nodes processes the query and return the corresponding results. So, all query processing happens on demand basis and no data is uploaded on server. This approach saves network bandwidth but to facilitate query routing, information about each node has to be maintained somewhere. This information may include current status of availability of node, computation power and kind of data a node stores. Maintaining this data all the time is clearly an overhead and unavailability of some node at a time might render system useless at some point of time.

Cloud based sensing is a sensing paradigm wherein various sensors upload data to a cloud. The

uploading of data can be a periodic process or immediately as and when the data is sensed. Sensed data can contain fields like geo-location of the sensor, some unique id of the sensor and the sensed value.

Given all these advancements in area of sensing, providing services on top of sensor data is still not a simple task because it requires some sort of mechanism for sensor data collection, then this data has to be processed to come up with consumer services. All these tasks require significant infrastructure and effort, rendering it a costly and time consuming activity. But if these resources are available as a service, then it simplifies the problem to large extent. Imagine a platform, capable of collecting any type of sensor data, and capable of processing this data according to requirements of specific consumer(s) or class of consumer(s). If this whole platform is available as a service, then sensing and providing services on top of sensed data would become more efficient. Since the platform is available as a service, it can be used when it is needed, so the cost of using the platform depends only on its usage. Since the platform can be used with any type of sensors, so no user need to put effort in developing such a platform for different type of sensors. Since, the platform can offer services to multiple classes of consumers, so no consumer need to develop their own system for their personalized services.

Platform services can be used by individual consumers for their daily life e.g., consumers can use route suggestion services offered by the platform, for their daily commutes. Similarly, platform services can also target organizations. These organizations can be government bodies as well as private sector organizations e.g. fleet management agencies, municipal corporations. Platform can provide subscription based services as well as demand based services to both type of consumers.

The platform services, e.g. business report generation, route suggestions, are *near* real-time and not strictly real-time in nature. Hence some delays, because of processing, might come while delivering services to users.

1.2 The Proposed System

This work proposes an architecture for Cloud-based Sensing-as-a-Service platform. The CSS platform provides APIs for receiving noisy data from different types of sensors and then provide services to end consumers on top of this data. A typical CSS platform is shown in figure 1.1. For uploading sensor data on cloud, the platform provides REST based web service which communicates using HTTP protocol. Sensors can upload data in JSON format. So, any sensor which understand HTTP protocol and JSON, can contribute to the platform. Similar, to sensor data upload APIs, the platform provides HTTP based APIs for those who want to subscribe to system services and frame queries to the system. The detailed platform architecture would be discussed in following chapters.

The data uploaded from sensors is called reports. Multiple reports can point to same event, e.g. multiple people might report same traffic jam, so for effective services, a mechanism for identification of events from reports, has to be employed. This work doesn't focus on this event identification mechanism. For all further discussions, we talk about events only and not reports.

The uploaded sensor data is passed to event identification module which processes these reports. Using the spatio-temporal attributes of the data, event identification module identifies main events from these reports. These identified events are stored in database by processing engine.

Consumers can subscribe for system services, so they will receive alert notifications for desired events. The process engine sends these alerts to all registered subscribers. The mechanism to

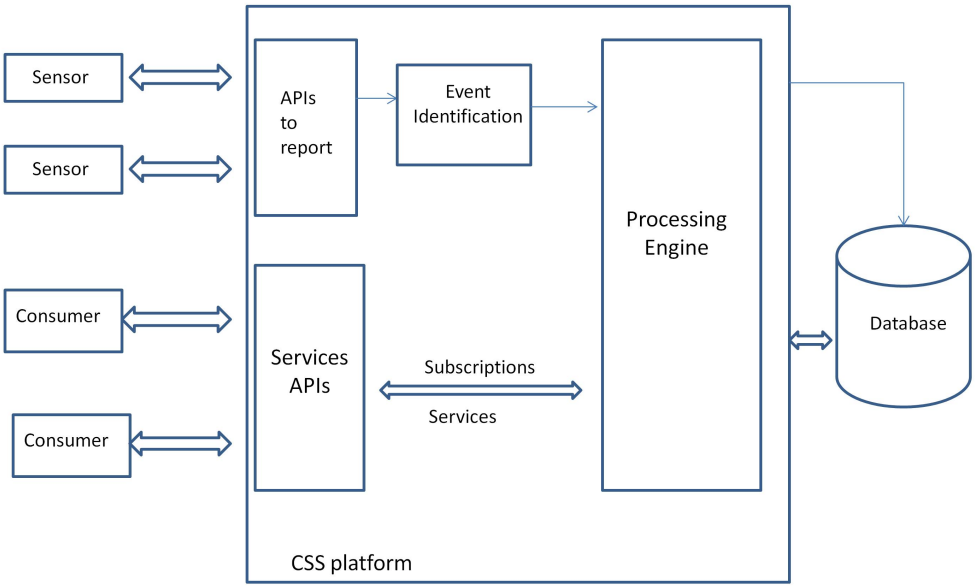

Figure 1.1: CSS Platform

deliver these alert notifications, called notification policy, would be discussed in more detail in further chapters.

Similar to notification services, the platform can also serve queries requesting specific events data e.g., if somebody is interested in all events happened in some particular area during some particular time period, then this type of query can also be framed to the platform.

Second part of this work focuses on performance modeling aspect of the proposed system. Performance modeling of the platform includes mathematically formulating the resource utilization, in terms of total number of people who subscribed for platform services and incoming event rate. Since resource utilization is crucial for efficient working of any system, so this performance modeling is important activity. A bench-marking application, which measures resource utilization for various incoming event rates and various number of subscribers for system services, has been developed. The immediate result of this bench-marking is resource utilization data for the complete duration of bench-marking. These results can be further analyzed and processed to define a decision policy for a system. This policy can be used to decide for what values of the incoming event rate and total subscribers, for system services, which alert notification policy is better in terms of resource utilization.

Such decision policy is very important for designing CSS kind of platforms. The bench-marking application, its usage and results, decision policy are described in detail in following chapters.

1.3 Contributions

This work is focused towards designing a Cloud-based Sensing-as-a-Service(CSS) platform and its performance modeling. The work has made following contributions:

1. Designing overall architecture and a query model architecture for a CSS platform.

   This includes:

   (a) Designing low level or primitive queries

   (b) Designing High level queries and corresponding binaries.

   (c) Designing a Core Engine

2. Performance modeling of platform and design policy for CSS platform.

   This includes:

   (a) Developing a benchmark application for resource utilization measurement

   (b) Benchmark the system using above developed benchmark application and get resource utilization results

   (c) Develop a system design policy for CSS kind of systems, using the bench-marking results

Part of this work, titled as *CityZen: A Cost-Effective City Management System with Incentive-driven Resident Engagement*, [22] has been published in MDM 2014.

Chapter 2

Literature Review

Area of sensing has witnessed significant amount of work, starting from dedicated hardware sensors and moving towards smartphone based sensing, in the few decades.

Sensing has many application in day to day life. Yong Yao et al. [9] proposed *Query processing in sensor networks* which talks about the various applications of sensing, ranging from inventory maintenance over health care to military applications and designing query layer for sensor networks. Their work emphasizes in-network processing/aggregation in order to save network bandwidth and properly utilize the available computation power. The design is motivated by following goals:

1. Declarative queries are especially suitable for sensor networks
2. Since sensor nodes have the ability to perform local computation, part of the computation can be moved from the clients and pushed into the sensor network, aggregating records, or eliminating irrelevant records

Moo Ryong et al. [17] proposed Medusa(A Programming Framework for Crowd-Sensing Applications). Developing a crowd-sensing application from scratch is not always a feasible and quick task, especially for non-technical people whose focus is just the end results. Medusa provides high-level abstractions for specifying the steps required to complete a crowdsensing task. Medusa has a distributed run time system that coordinates the execution of tasks between smartphones and a cluster on the cloud. Anybody using medusa can specify this task in a user friendly manner like as follows:

$$Recruit \rightarrow TakeVideo \rightarrow ExtractSummary \rightarrow UploadSummary \rightarrow Curate \rightarrow UploadVideo$$

Medusa runtime can understand these tasks and with the help of binaries that it has, it can execute each task. Medusa uses Amazon Mechanical Turk [1] for recruiting volunteers. In this manner, doing a crowd sourcing task has become easy.

Medusa is more towards easing development of crowd sourcing applications whereas our work

is more towards designing the Cloud-enabled Sensing-as-a-Service platform and its performance modeling. Although architecture of our proposed platform is such that if required, it can leverage all the services provided by medusa.

Charith P et al. [16] talks about *Sensing as a Service Model for Smart Cities Supported by Internet of Things*. The Internet of Things (IoT) has gained significant attention over the past decade. IoT is all about connecting billions of sensors to the Internet and expects to use them for efficient and effective resource management in Smart Cities.

The idea of offering Sensing as a service promotes the *"pay as you go"* method or in other terms *"pay only for what you use"* as suggested by [16]. This allows the consumers to consume a service from a service provider by paying only for the amount of resources they use. This is an efficient way compared to the traditional methods of consuming resources where consumers need to buy resources in predefined discreet quantities with higher expenses. In context of smart cities, platform as a service model can be very helpful because arranging dedicated resources for each type of problem at correct time is not always possible. For example, waste management is one of the toughest challenge that modern cities have to deal with. Although there are several parties who are interested in waste management (e.g. city council, recycling companies) but deploying required infrastructure is not feasible for each body. Instead of deploying sensors and collecting information independently, the sensing as a service model allows all the interest groups to share the infrastructure and bare the related costs collectively.

Tathagata Das et al. [5] proposed a platform called PRISM(Platform for remote sensing using smartphones). This work talks about potential of opportunistic and participatory sensing using mobile smartphones. PRISM allows application writers to package their applications as executable binaries. PRISM platform then pushes the automatically to appropriate set of phones.

Nericell [12] proposes an interesting idea of monitoring road & traffic conditions using smartphones. The idea is to use accelerometer readings, taken automatically, to detect potholes or bumps etc. So in one sense this data collection is done without human intervention. This is like opportunistic sensing paradigm. Similarly using built in microphone readings to detect honking etc. in traffic. Since smartphones have inbuilt GPS and/or GSM radios the location of places where bumps or honking is detected can be easily taken.

Similar to Nericell, Pothole patrol [8] is another good example of pothole kind of problem detection using mobile phone.

Ushahidi System [24] in Beijing, China is another good example of a crowd-souring application. This system is being used for improving transportation in Beijing. This platform is being used by Chinese municipal authorities to mitigate city traffic and other urban problems.

Fix my street system [23] developed for UK is a good application of participatory sensing. In this system anybody can report any problem he/she notices around, using a smartphone. One can see all reports reported so far and also which of them have been solved. It involves people in developing their own area.

Another work targeting participatory sensing paradigm, focused on pollution monitoring is PEIR [13]. Personal Environmental Impact Report, is a participatory sensing application. It uses location data collected from everyday mobile phones and then calculates personalized estimates of environmental impact and exposure.

Development of Publish-subscribe systems, their performance analysis and comparison with other communication models, have seen significant work. [25] models the high performance publisher subscriber systems. Work in [14] talks about cost model for publish-subscribe systems, followed by a comparison with other communication models namely request/reply and polling mechanisms.

[7], [4], [15], [18] and [2] describes methods and implementations for improving performance of publish-subscribe systems. Various publish-subscribe systems have been analyzed and compared for performance.

Few benchmarks have been developed for modeling performance of publish/subscribe systems. [20] and [19] represents such a benchmark. They developed a benchmark, named jms2009-PS, for modeling a JMS(Java Message Service) based publish/subscribe system.

SPECjms2007 [21] is the first industry-standard benchmark for Message-Oriented-Middleware(MOM). MOM also includes publish/subscribe systems.

Several benchmarks have been developed for Event Processing Systems(EPS). FINCos framework [10], [11] proposed a Java-based set of benchmarking tools for load generation and measurement of performance for EPS systems.

COPS [3] proposes a Content oriented publish/subscribe system. COPS proposed topic based subscription to services, instead of particular source or particular location based services.

Cayuga [6] proposed a stateful publish/subscribe system. The system allows subscribers to express their topic of interest in a user friendly manner. It is based on Non-Deterministic Finite Automata.

Chapter 3

Platform Architecture

This chapter discusses the design of proposed CSS platform, in detail. Figure 1.1 in chapter 1 describes the platform briefly but this chapter mainly focuses on two architectures, as part of CSS platform. These two architectures are:

1. Query Model Architecture
2. Notification System Architecture

A CSS platform supports two types of queries, namely notification subscription queries and event data driven queries. Notification subscription queries are those which are meant for subscription to platform services. For example, if a consumer is interested in all traffic jams happening in particular area during a particular time of the day, then he can subscribe to alert notification service of the platform. The queries meant for such subscriptions are notification subscription queries. So, all queries related to alert notifications, including creating alerts, modifying existing alerts or deleting alerts, are notification subscription queries. Each consumer has to register only once for a service and then, the platform will keep on sending alert notifications for each relevant event.

Event data driven queries are framed by consumers for retrieving event related data, time to time. For example, a query for retrieving all events in some area from time t1 to time t2, is an example of event data driven query. Unlike notification subscription queries, every time a consumer needs event data, he has to frame a new query and get corresponding results.

Query model architecture, as shown in figure 3.1, represents component of CSS platform that serves event data driven queries. Notification system architectures, as shown in figure 3.2 and figure 3.3, represent components of CSS platform which serve notification subscription queries. Both of these architectures are discussed in following sections.

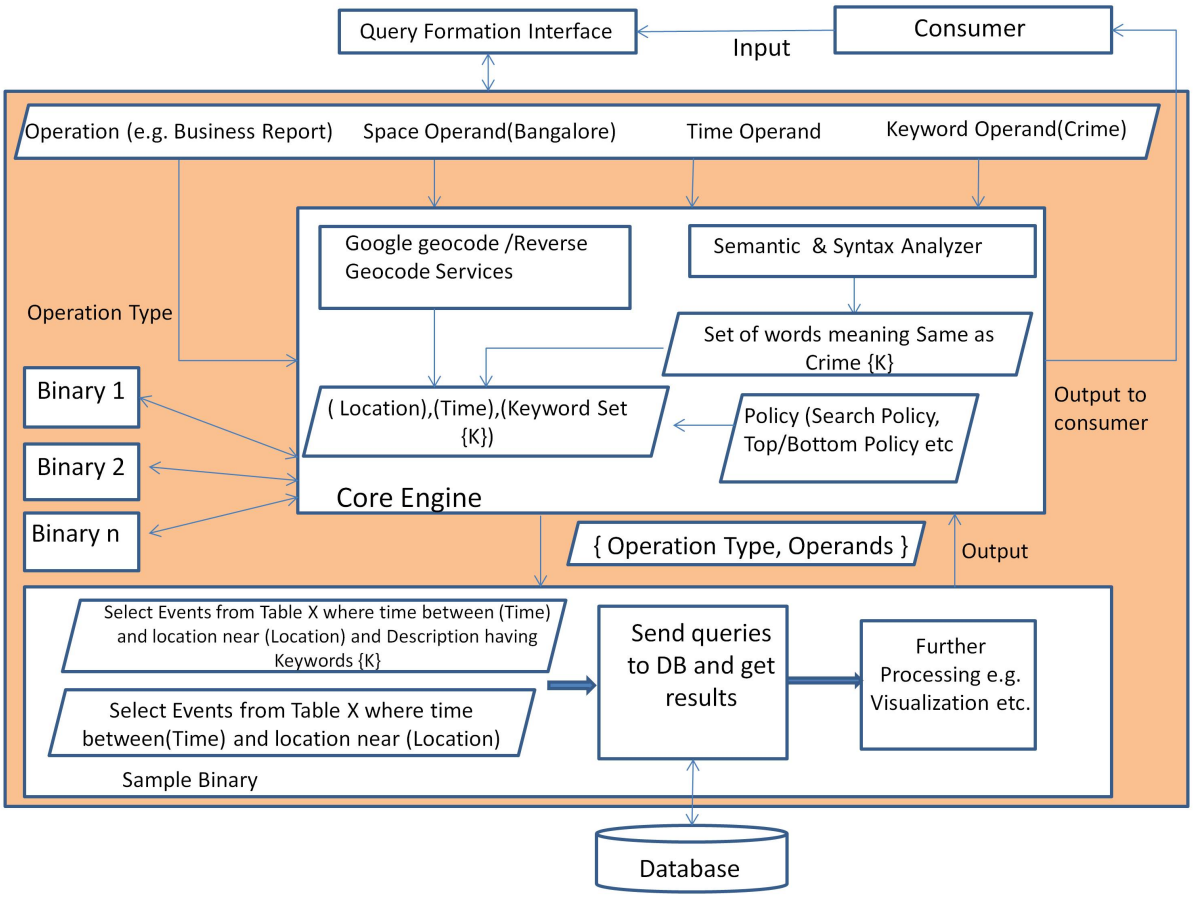

Figure 3.1: Query Model Architecture

3.1 Query Model Architecture

Query model, as shown in figure 3.1, consists of all queries, including primitive and abstract queries, supported by the platform. The model also describes how does the platform generate results for consumer framed abstract queries. Consumers may not be interested in raw data uploaded by the sensors but they might be interested in some consumer targeted analysis, done by the platform, on top of that data. So the query model provides an abstraction over sensor data and allows consumers to frame simple abstracted queries which will be processed by query model to return user friendly results. To provide abstraction over the potentially distributed sensor database, unknown to end consumers, an intermediate layer between end consumers and the system is required and query model works like that abstraction layer. The query model itself is composed of following three components:

1. Set of primitive queries
2. Abstracted queries and their corresponding binaries
3. Query Engine

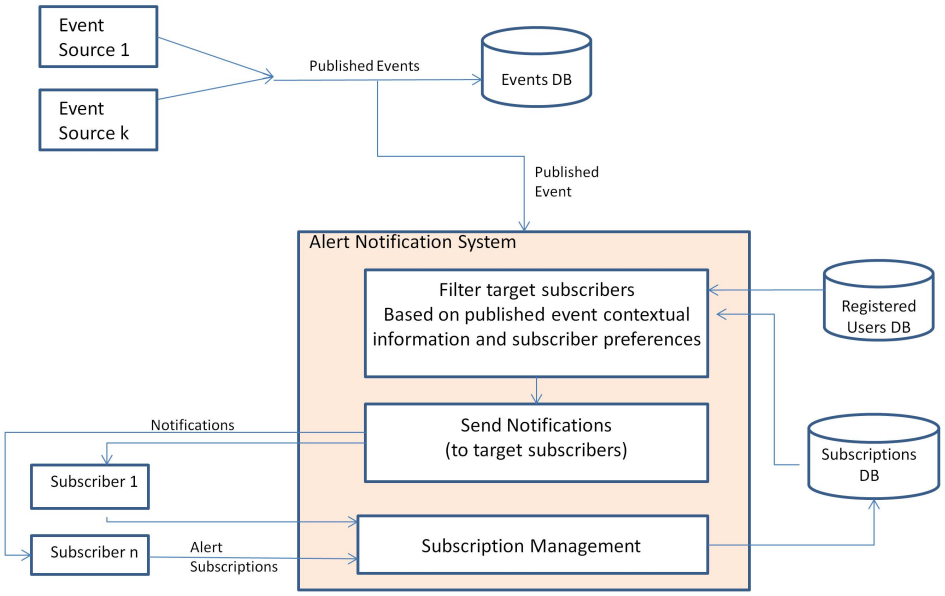

Figure 3.2: Event Triggered alert notification system

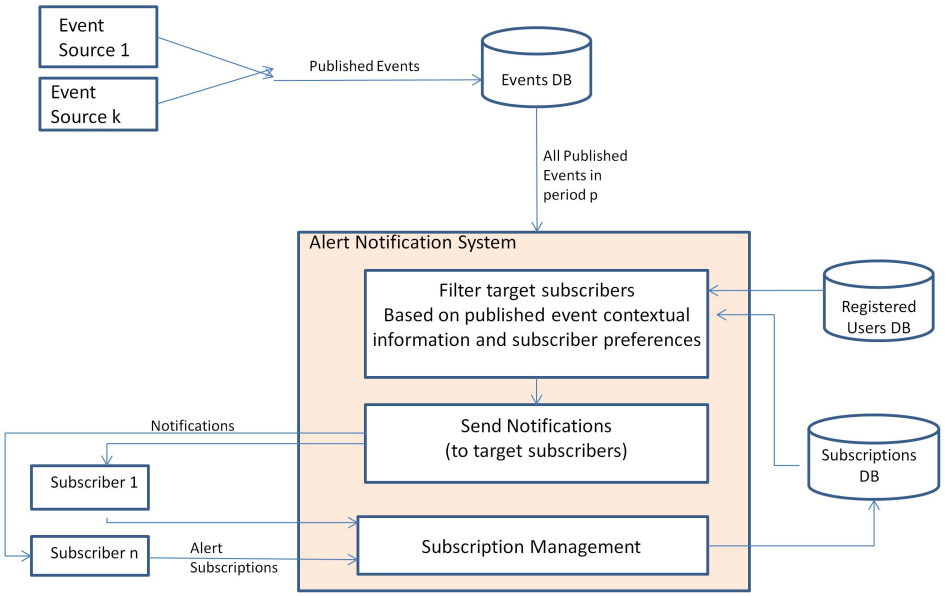

Figure 3.3: Periodic alert notification system

3.1.1 Primitive Queries

Primitive queries for the underlying system are those which operate on single table in database i.e. no table joins are required. Broadly the underlying spatio-temporal data has three dimensions i.e. space, time and keywords. Primitive queries can retrieve a specific dimension out of above three given the other two. Also the system implementing these primitive queries can have some default values for the non-specified parameters like using the current location of the consumer if he/she hasn't specified the location.

Primitive query set should be exhaustive enough so that variety of abstract queries can be served using these queries. Performance of the system relies heavily on the efficiency of the primitive queries, if they are slow whole system will suffer, so while implementing these queries, efficiency should be taken care of. Some of the possible primitive queries are listed below.

General queries

General queries as suggested by the name returns all events satisfying the given conditions. They are not targeting any specific problem. These queries are basic queries which perform very fundamental tasks like returning events, comments or tags for events. Some of the potential general queries are listed below.

- All events in [area] from [datetime1] to [datetime2] having [keywords]
- Latest/first [k] events [area] [datetime1] to [datetime2]
- [k] most/least discussed events [area] [datetime1] to [datetime2]
- [k] most/least tagged events [area] [datetime1] to [datetime2]
- [k] most/least rated events [area] [datetime1] to [datetime2]
- Get all comments/tags for [event]
- All events reported with text/image/video only in [area] from [datetime1] to [datetime2]

Since primitive queries forms the base for abstracted queries, so they should be efficient in execution. These queries should leverage indexing schemes so that they run faster and properly.

Spatial queries

Spatial queries are those which returns set of locations/areas satisfying the given conditions. Depending on implementation of queries, some of the parameters can be set to default values. These queries consists of aggregate queries, top/bottom k type of queries. Some of the potential spatial queries are listed below.

- Area with Max/Min problems(any type) from [time1] to [time2]

- Area with Max/Min problems from [time1] to [time2] having [keywords]
- Most/Least [K] dirty places [area] from [time1] to [time2]
- Most/Least [K] power/water wasting [area] from [time1] to [time2]
- Most/Least [K] noisy places [area] from [time1] to [time2]
- Top/Bottom [k] congested places in [area] from [datetime1] to [datetime2]

Time queries

Many times, we need to know about what is a good time to do a specific thing in a particular area. For example, if somebody is planning to travel, certainly he/she don't want to get stuck in traffic jams. So for that he has to know what is a good time to go or even which day of the week is a good time to go. This type of questions can be answered by time queries. are those which returns date/time satisfying the given conditions. Some example queries are following.

- Most/Least congested day of the week in [area] from [datetime1] to [datetime2]
- Most/Least congested time of day in [area] from [datetime1] to [datetime2]
- Most/Least overload(transit) timings of day in [area] from [datetime1] to [datetime2]
- Most/Least overloaded(transit) day of week in [area] from [datetime1] to [datetime2]

Queries discussed so far are generic to the problem type but one can specify a particular problem type e.g., garbage, road conditions, using keywords in the query. If no keyword is specified, these queries target all type of problems. If some specific type of problems are more prevalent in some areas then system can offer more specific queries for those problems.

Similarly there can be queries for Garbage problems, Power/Energy issues etc.

3.1.2 Abstract Queries

High level/Abstracted are those queries which have been targeted for specific group e.g. some govt. agency or fleet management agency etc. The goal of these queries is to ease the query framing at consumer end. Given a set of primitive queries, a set of abstracted queries can be served by the system. The system internally uses the primitive queries to get required data from database and then combines and analyse the results according to abstracted query. Let's take an example of such a query and see how can one provide its implementation. Consider a govt agency is using the system. The agency is looking forward for city development and is looking for areas across the city which need development. The system can aid the agency by listing all different areas of the city with their respective problems, areas can be rated with respect to intensity of the problem. The system can generate a complete business report in desired format about these areas so that the agency can focus on more critical areas directly.

So an abstracted query could be like following.

Business report of [traffic problems] from [datetime 1] to [datetime 2] in [city name]

This query generates a report for all traffic related problems reported from datetime 1 to datetime 2 in vicinity of city name. Depending on the implementation of corresponding binary, generated report may contain statistical information, graphs/charts giving a visual representation of desired problem type. Similarly abstracted queries can target other agencies e.g. consider a fleet management agency. This agency needs real time information about current traffic updates and road conditions in order to serve its customers in a economic and comfortable way. They won't prefer to go over possibly congested roads, the roads that are badly conditioned, are having lot of potholes or are having lot of garbage etc. All these problems leads to poor performance of vehicles and discomfort of the customers. So for such an agency, the system can support dynamic route suggestions taking into consideration all road related problems described above. The abstracted queries for route suggestion may look like following.

Route suggestion from [location 1] to [location 2]

So, this query when comes into the system, the system looks for all possible routes from location 1 to location 2 and then suggests a route which is better in terms of all factors discussed above like road condition, traffic jam, garbage etc. Once a route is selected by the consumer, then depending on the implementation, the system can use existing navigation technologies to navigate the consumer on his/her preferred route.

Now there are already existing systems which supports this type of queries e.g. Google maps. But most of them rely on traffic conditions only. Now for example a route may have less traffic but the road is having potholes on it or may be a lot of garbage along its sides or some part of the road is under construction. So in these conditions it may not a good idea to follow that route. Since different people have different priorities, like some may want to shorten the travel time and for them it is ok to travel on roads with little bad condition but for others comfort during travel may be on high priority. The implementation of the system should consider all these factors while suggesting routes to consumers. The underlying system has the advantage of having a spatio-temporal data for the problems across places and hence in a better position to suggest better routes for example. If the data available with the system is leveraged to max extent, then system can prove very helpful in overall city management and easing daily life of people using the system.

All notifications that the system sends to its subscribers can be sent in real time so that right information can be used at right time. Also while implementing, system can offer two set of same APIs but with different trade off. For example some people would like to have a trade off between alert subscription cost and delay after which they get the notification. The system can

prioritize its services based on consumer desires.

Now when we have a set of primitive queries and some abstract queries that the system supports, next requirement is a comfortable and easy to use mechanism to express abstracted queries. If consumers are not able to express their requirements in a friendly manner then the system is not so useful to them. End consumers, individuals or agencies, should be able to express their queries easily. This mechanism could be a user friendly GUI for example. The system can offer a portal kind of interface wherein consumers can login and customize the system services according to their needs. Also, an easy to use descriptive language is also a good choice. This approach has been followed by medusa framework [17].

Implementation

Implementation of these queries follows a modular approach. Every abstracted query has a corresponding binary file which understands the abstracted query and how to process it. Each abstract query can be divided into two components, one component being the *Opcode* and other component as *Operands*. *Opcode* specifies what is required by the end consumer and *Opcodes* aids in identifying relevant data from database to serve the query. *Operands* can be many in number whereas *Opcode* is only one. *Operands* can belong to any of the three dimensions space, time or keywords. A typical example of *Opcode* can be seen from any of the above discussed abstract queries. For example, in business report query, traffic problems and duration of events and area name are are all *Operands*.

Connecting to the underlying database for fetching results is also a part of the binaries. This can make the binaries vulnerable to change whenever the underlying database changes. For example if the system initially follows a centralized approach but later on it is distributed over multiple clusters, then all the binaries have to be changed to accommodate this change in database scheme. Changing all binaries is clearly not a good idea. So, instead of communicating directly to the database, binaries communicates to a software layer which in turn talks to the underlying database. This layer understands the underlying database distribution i.e. centralized or distributed. In case of distributed database, this layer should be able to fetch all relevant data from multiple databases if required. So, in case some changes happen in database distribution, only this layer has to be changed and all binaries remain intact. So this layer provides an abstraction over the underlying database. For good performance, this layer should be very efficient otherwise this itself can become the performance bottleneck for the whole system.

Depending on the specific *Opcode* a binary can perform some post processing on the data fetched from the database, like visualizations etc. Binary can also format the output as desired by the end consumer. For example, consider the binary file for generating a *Business Report*. This binary will first get all the relevant events using specified *Operands*. This might involve using a set of primitive queries. The underlying system uses a rule based mapping of *Opcode* to primitive queries. For example, a business report binary will first get relevant events using *Operands* and then analyse these results to come up with required statistics and finally since this report is a

govt. document, so it would be having a specific format, so business report binary will format the final results in that format. This might include plotting graphs or charts etc. and making a pdf like format.

3.1.3 Core Engine

Core Engine is the center of query model. It works like a mediator between end consumers and the spatio-temporal data in database. It manages all of the system binaries which process abstract queries. The detailed architecture of query model is shown in Figure 3.1.

Core engine has its own APIs using which abstracted queries can be passed to the engine. Currently these APIs have been developed using a REST based web service which works on top of HTTP protocol. So any device which can communicate using HTTP protocol and understand JSON(JavaScript Object Notation) objects, can use these core engine APIs. These APIs expects the abstract query in two parts, as discussed above, *Opcode* and a set of *Operands*. Now given an abstract query expression interface, this interface sends the query input from the end consumer to the core engine using core engine APIs. Then it becomes the job of core engine to process that query and send results back to the end consumer. In order to accomplish this, the core engine needs to know which binary to use for a given abstract query. The core engine uses a rule based approach to map an abstract query to the corresponding binary. So the engine has a set of rules mapping each supported abstract query to one of the available binaries. This is a one to one mapping which means for a given abstract query, there exist exactly one binary. This approach makes the system modular and allows system extension in an easy way. Whenever a new binary is added on to the system or an existing binary is removed from the system, only this mapping rule set has to be modified. This approach makes it easier to extend the system.

Now since none of the binary file is communicating to the end consumer, each binary only does its processing according to the specified query and returns the results to the core engine, and the core engine has to communicate the results back to the end consumer. The communication between core engine and the end consumers is totally based on HTTP protocol, so any device which understand HTTP can be a potential consumer.

Now lets look at the query model architecture shown in figure 3.1. The coloured box represents the platform consisting of core engine and binaries. On one end of this platform is query formation interface which communicates with the end consumer and on other end is the underlying database containing all the spatio-temporal data collected so far.

Consider an abstract query for business report generation coming into the system. As discussed before, it consists of two parts *Opcode* and *Operands*. All these inputs goes into the core engine.

The engine first does some pre-processing on the *Operands*. Since location can be a human readable string or geo-coordinates, so the engine may need to use existing geocode/reverse-geocode service. Time dimension is already very well understood, so it doesn't need any kind of pre-processing. Keyword operand may contain multiple keywords and each keyword in turn can

have many synonyms. Since the data is submitted by humans, which can express same event e.g. a broken road, with different words. So in order to work effectively the system have to look for the possible similar words for the given keyword set. This mapping between a word and set of similar words will involve some syntax and semantic analysis. Assuming such a system is there in place, the system generates an equivalent set of words for given keywords set, to look for in database.

Now the system can allow some sort of human judgement into the reported events, for example by allowing tags and comments on reported events. Then the next problem that comes is where does the system looks for the given set of keywords. In order to solve this problem, the idea is to come up with a *Search Policy* module. This module can specify where to look for the given set of keywords. This module can be changed easily thus making the search policy an adaptive policy.

After this pre-processing on *Operands* is done, next job is to invoke corresponding binary. As the figure shows, the system can have a set of binaries with it. So the engine uses its rule based mapping to identify which binary to be invoked. For this example, since the *Opcode* is business report, lets say the corresponding binary name is same. So this binary is invoked and all the operands that the engine received are passed to the binary. Then further on the binary does whatever is required depending on the *Opcode* and *Search Policy*. It generates set of equivalent primitive queries and pass these queries to the software layer on top of database.

After receiving data from database, if required some further processing in terms of visualization etc. is done and the report is generated in required format. Then this report is passed to the core engine which sends it back to the end consumer.

3.2 Notification System Architecture

Architecture of notification system is shown in figure 3.2 and figure 3.3. Each figure represents one notification approach.

In both the figures, event sources are generating events at a specific rate. In the actual system, these events will be generated, from reports, by event identification system. Generated events are stored in events database. For each event, alert notification system sends alert to all relevant subscribers.

The notification system works in following manner. First, the system has events coming in at a certain rate. Events are stored in events database. The system has n number of subscribers who have subscribed to platform services and would like to receive alerts for events happening around them. So for each published event the alert notification system finds out all eligible subscribers by matching the subscriptions in database with the event published. After that alerts are sent to all of these subscribers.

Once an event gets published, system can follow one of the two following approaches to send

alerts to the subscribers.

1. Event Triggered Notifications
2. Periodic Notifications

First mechanism, Event Triggered Notifications, suggests that as and when some event comes in, the event is stored in Events database and also a notification is triggered to be sent to all eligible subscribers. So publishing of event invokes the alert notification system and passes it the published event. In this approach, event alerts are delivered quickly to all subscribers.

Second approach, Periodic Notifications, rather suggests that instead of sending alerts on each event arrival, send alerts periodically with period P. It is a batch processing kind of approach. The published events keep on coming in Events database, the alert notification system reads all events arrived in last P time and send alerts for all of those events one by one to all eligible subscribers. For this approach, deciding a value of period is an important problem because system performance is a function of period. We will describe a decision guideline in the following chapter. Delay in delivering alerts to subscribers is a function of period. Increasing the period will increase this delay and decreasing the period will decrease the delay.

Both these approaches have their own pros and cons. The following sections of this chapter will discuss about performance of both these approaches considering varying number of subscribers and incoming event rate.

3.3 Database Schemas

This section describes the database schema of underlying database.

The database contain several tables like Reports table, Events table, Comments table and many more. Schema of some of these tables are shown below and a description about their attributes is provided.

Reports Table

Schema for Reports table is shown in Table 3.1. Data uploaded from sensors is stored in reports database.

Id is an integer field and is automatically incremented by the underlying database management system, MySQL in this case. Id is being used as a primary key in reports database.

UserId represents unique id of a reporter. This key should be able to identify a reporter uniquely. Value of this attribute can be either of email address, mobile number of reporter, social network id e.g. facebook id or twitter id etc. For this system, email address is being used as UserId.

Field Name	Field Type	Description
Id	Integer	Auto Increment. Primary Key
UserId	Varchar(128)	A unique id to identify the user
date time	datetime	The datetime when report arrived at the server
location	varchar(128)	Geo-Coordinates of the report. Captured automatically
content type	varchar(64)	Determines what type of data report contain. Possible options are image, text etc.
description	longtext	Description about the event. It is optional to provide.
category	varchar(64)	The category in which this report lies e.g. Garbage/Traffic etc.
url	varchar(512)	The url of the any image/video etc. uploaded by the reporter in report.
eventid	int	foreign key in table Events.

Table 3.1: Schema for Reports table

Datetime represents the date and time when a particular report arrived on the server. Location represents geo-coordinates of event location. Geo-coordinates are taken automatically while somebody reports to the server. Geo-coordinates are very important for a report, without geo-location, a report would be barely useful. Content type determines type of content e.g. audio/video/text/image etc. This field can prove very important if distributing a database based on content type. So, if a database distribution scheme is such that all images are stored at one location and all videos at other location, then this field can prove to be very important. The value of this field is automatically taken at client side and reporter don't have to put it explicitly.

Description is simple text about the report. It is completely optional to provide a description to the report. If present, this field can add human judgement into the system. Descriptions can be used in Event Identification process(generating of event from multiple similar reports). Presence of this field increases the human involvement in the system.

Category attribute is a list of some predefined categories e.g. Garbage, Traffic Jam etc. If some report doesn't lie within any of the predefined categories, then category can be specified as miscellaneous.

The system stores all type of media on the file system instead of database. URL field is a pointer to that media on the file system. Eventid is a foreign key in another table *Events* shown in Table 3.2.

Events Table

The idea, of having Events table, is that many reports might be pointing to one kind of event in same area, e.g. many reporters might report about traffic jam in same area. These reports

Field Name	Field Type	Description
Eventid	Integer	Auto Increment. Primary Key
UserId	Varchar(128)	A unique id to identify the user
date time	datetime	The datetime derived from mulitple reports
location	varchar(128)	Geo-Coordinates of the report. Captured automatically
content type	varchar(64)	Determines what type of data event contain. Possible options are image, text etc.
description	longtext	Description about the event. It is optional to provide.
category	varchar(64)	The final category in which this event lies e.g. Garbage/Traffic etc. This value comes out from event identification process.
url	varchar(512)	The url of the any image/video etc. uploaded by the reporter in report.
quality	int	This field determines the overall quality of the event. Quality forms the basis for providing incentives to the reporters. Quality is decided by analysis of content in the reports.

Table 3.2: Schema for Events table

might come at different times, but the context information in all of them is pointing to the same event. So all these reports should be merged together intelligently to make up one event out of them. These events after identification are stored in event table, having schema as shown in Table 3.2. Each report is mapped to some event, by event identification system, if not rejected because of quality concerns. This mapping of report to its corresponding event is maintained with the help of foreign key *Eventid* in reports table. This foreign key points to primary key in Events table. In this way each report can be mapped to its corresponding Event.

User Comments Table

The system allows users to comment on individual reports submitted on platform. The idea behind this feature is to involve more human judgement in the system. If people can add their views towards identified events and posted reports, then the quality of these events and reports can be improved further. For example, consider an example that some user reported a road condition, his report contains picture of the road under problem, but the report doesn't contain any useful description. Now it is likely that other users are also familiar to this location, they can go ahead and add some more description about the report so that it becomes more useful. Similarly, people can vote up a report or event which increases the credibility of the reporter and also improves his incentives. Also people can tag reports and events, these tags can play an important role in categorising a particular report in proper category. Reports and events can be indexed on basis of their tags, which increases the chances of retrieving proper data in less time. Tags for the reports and events are stored in the reports and events tables, there

are no separate tables for tags. But since comments can grow huge by time and it is require to maintain information about who commented at what time, so the comments are stored in Comments table shown in Table 3.3.

Field Name	Field Type	Description
Id	Integer	Primary key, auto increment.
Reportid	Integer	Foreign key in Reports table.
UserId	Varchar(128)	A unique id to identify the user
date time	datetime	The datetime when user commented.
comment	text	The comment posted by the user.

Table 3.3: Schema for User Comments table

Since multiple reports may form one single event, so each report stores a pointer to its corresponding event. Similarly, each comment also stores a pointer to the report to which it belong.

Chapter 4

Performance Modelling

The architecture of underlying CSS platform is such that, on one end of it events are coming into the system at a certain rate, and on the other end, platform is offering services to consumers. These services can be in terms of queries framed by end consumers or alert notifications for the subscribers. This chapter is about modeling the performance of such a platform in terms of incoming event rate and number of subscribers. The following sections describes the architecture of benchmark application, used for modeling, and then further describes performance model and its results in detail. The designed CSS platform is scalable which can be justified by bench-marking results, discussed in following sections.

Several experiments have been performed on the system for measuring its performance, with different number of subscribers and different incoming event rates. The following subsection describes the architecture of the benchmark used for evaluating the system.

Given this platform design, compliant with the design approach discussed in chapter 3, bench-mark application has been developed and executed on the platform. Output of bench-marking process is a performance model which can be used for developing a decision guideline for a CSS kind of platform.

The platform and benchmark application have been developed using *Python* as main scripting language, *Django* as the web platform for hosting python scripts, *Apache* as web server and *MySQL* as the back end system. This system can receive and store events and has a notification system, capable of handling both event triggered and periodic notification approaches. All results shown in further sections are dependent on underlying machine configuration. The machine configuration used for this benchmark is listed below.

Number of CPUs= 32

Sockets = 2

Cores per socket = 8

Threads per core = 2

CPU MHz = 2100

L1 cache = 64k

L2 cache = 256k

L3 cache = 20480k

Physical Memory= 32 GB

4.0.1 Benchmark Application Architecture

The developed platform, as described above, is then tested for performance using a benchmarking tool. The architecture of benchmark application is shown in figure 4.1

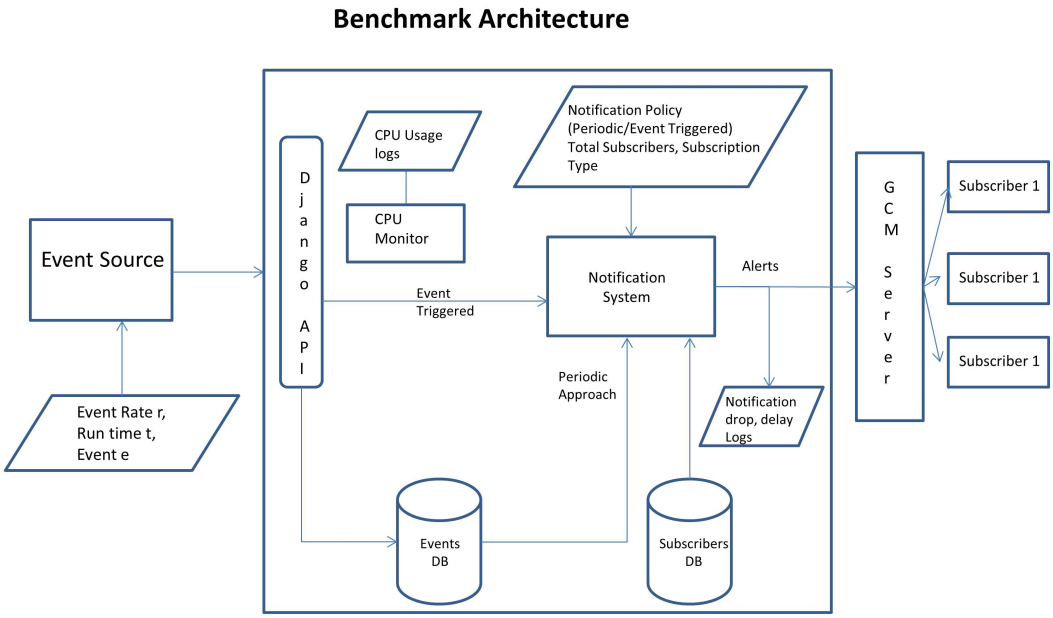

Figure 4.1: Benchmark application architecture

Apache Benchmark(ab) tool has been used as the event source. This benchmarking tool is a standard way of generating HTTP traffic and is used for benchmarking web servers, for number of requests they can handle per second. This tool can be used to post data to a URL using HTTP POST method. For the underlying system, whenever somebody reports, a JSON object is created out of the contextual data and then this JSON object is sent to the server using HTTP POST method. Same thing is done with the help of ab tool. This ab tool is invoked programmatically using shell script. The tool reads a ready to submit JSON object from a file and submits it to the specified URL(Server APIs) using HTTP POST method. For the sake

of event diversity, some attributes of event, like longitude and latitude of event location, are generated randomly.

Benchmark application parameters e.g. type of Event (e) to be generated from event source, are inspired from internal pilot conducted at Xerox Research Center Bangalore, and are not pure hypothesis.

Since this is a benchmark application, so it is configurable. The event source takes as input many things namely event rate, total run time and type of event to be submitted to the server. For controlling event rate, ab has a concurrency parameter, on increasing concurrency parameter, event rate can be increased. Similarly, ab has an option to specify the duration of its run. And lastly for event type, since the data is submitted using HTTP POST method, so ab can read this data from a file, and depending on the type of event required, one can specify a corresponding JSON object in the file.

Event source submits events using a REST based API provided by Django platform. These events are stored in Event database. Now, the notification system can read events and send alert for each event. Depending on the notification approach, event triggered or periodic, notification system will process events as and when they arrive or in periodic manner respectively.

The notification system shown in the architecture can also be configured. The first argument it takes is notification policy which essentially determines whether notification system will be event triggered or periodic and if it is periodic then what should be the period value. The value of this parameter can be changed dynamically. Further section will provide a mechanism to calculate a better value of this parameter. Next argument, to notification system, is about what is the total number of subscribers. Last argument is about what type of subscriptions do those subscribers have. Each Subscription can target specific problems e.g. traffic jam, garbage etc. and specific time duration or a subscription can be a generic subscription which covers all problems types during any duration in all areas.

Now depending on the notification approach, Event Triggered or Periodic, the notification system sends alert to the subscribers. These alerts are being sent through Google GCM server. So the notification server hands over the event to the GCM server.

Platform Performance Metrics

For measuring overall performance of the system, we used following metrics.

1. % CPU usage
2. Average Notification Delay
3. % Notification Drop

% CPU usage is average % CPU used during the bench-marking process. Notification delay is the time difference between arrival of event on platform and departure of corresponding alert. This

delay is measured in seconds. Now, when notification system tries to send alerts to subscribers, some of the alerts are dropped because of congestion at system. The % of such dropped events is called % notification drop.

From CSS platform point of view, measuring these metrics is important because in any system, CPU is always an asset and hence CPU usage is important to monitor. Since the platform offer alert notification services, so delay in notification also becomes an important factor, because all subscribers would like to receive alerts timely. Similarly, because of subscribers, % notification drop becomes crucial factor because whosoever has subscribed to an alert, should get that alert.

Along with these metrics, there can be other factors that can be monitored. Some examples are disk I/O, network bandwidth used, platform memory used. All these factors have been added to the benchmark application, later on, but only the above three metrics have been used for performance modeling.

Benchmarking Process

While the platform is receiving events and providing services to consumers, the benchmark application monitors the CPU usage and keeps on logging % CPU usage in a file. Similarly, notification system also keep on logging notification drop and notification delay in another separate file. These files are later on used for modeling the underlying system. These files are the output of the benchmarking scripts. Further analysis has to be done on these files to come up with CPU usage equations, average notification delay and aggregate % notification drop.

Since notification system can follow one of two approaches: *Event Triggered or Periodic*, so benchmarking have been done for both these approaches. Now lets consider both these approaches one by one and see what is the system performance using the above discussed benchmark application.

4.0.2 Event Triggered Approach

In this approach, each incoming event invokes the alert notification system which sends alert to subscribers. Now given this approach, we varied the number of subscriptions and incoming event rate. The number of subscriptions is varied from *500 to 8000* and the event rate is varied from *35 events/second to 560 events/second*, doubling each time. For a given number of subscriptions and given incoming event rate, the event source(apache bench-marking tool ab) sends events to server for a duration of ten minutes. For this duration of ten minutes, % of CPU usage and notification delay are measured and logged into files. The frequency of % CPU measurement was every ten seconds.

After ten minutes of events generation, average CPU usage is computed to get an idea of how much CPU has been used by this approach for given number of subscriptions and incoming event rate. In the similar manner, 25 readings have been taken by varying number of subscriptions and incoming event rate. Table 4.1 shows the average CPU usage during these experiments.

Event Rate	Total number of subscriptions				
	500	1000	2000	4000	8000
35	4.815384615	5.423809524	6.849180328	9.875409836	15.25737705
70	5.767213115	6.731147541	8.908196721	13.59836066	23.47
140	10.00327869	12.37213115	17.8704918	30.28360656	71.24166667
280	13.28333333	18.05833333	29.19677419	51.5295082	96.65616438
560	13.345	20.06213592	29.27142857	57.42461538	96.85421687

Table 4.1: Average % CPU usage for event triggered notification approach

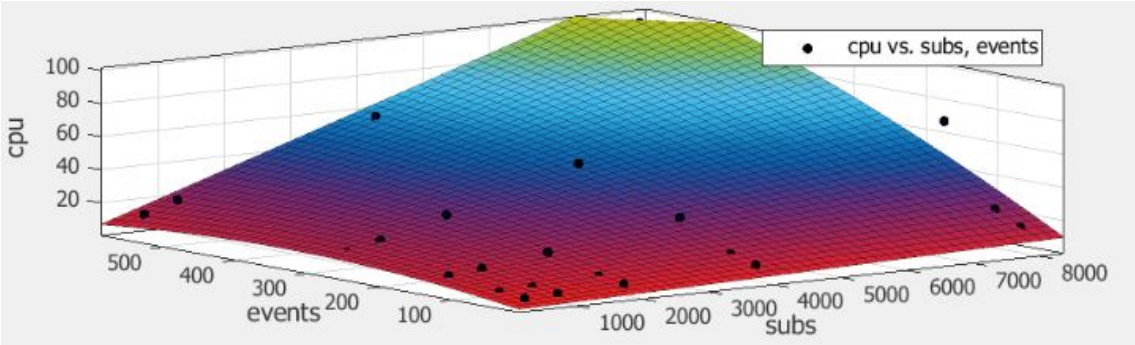

Figure 4.2: Surface fit for event triggered approach

Clearly the average % CPU usage increases with the increase in either of number of subscriptions or incoming event rate. When the number of subscriptions and event rate are low, then CPU usage is quite good but when incoming event rate increases beyond 70 events/second, then having higher number of subscriptions leads to very high CPU usage. In case number of subscriptions increase to 8000, the CPU usage goes even upto 95% which is very high. So, it clearly states that the event triggered notification approach is not a good choice for a system having lot of subscriptions and higher rate of incoming events.

Table 4.1 shows a clear dependence of CPU usage on both the factors, number of subscriptions and incoming event rate. This dependence can be linear or polynomial of some degree. So to further understand the relationship between CPU usage and number of subscriptions and incoming event rate, curve fitting tool has been used, after plotting a three dimensional graph from table 4.1. X-axis of the graph represents total number of subscriptions ranging from 500 to 8000 and Y-axis represents incoming event rate ranging from 35 to 560 events/second. Z-axis represents the % average CPU usage, so it can range from 0 to 100. Values on x-axis and y-axis are doubled each time i.e. first the system measures CPU usage for 35 events/second and 500 subscriptions, then number of subscribers are doubled upto 8000 and CPU usage is measured in similar manner. When this step is complete, incoming event rate is doubled to 70 events/second and number of subscriptions again reset to 500, and then doubling number of subscriptions each time upto 8000. In this manner, CPU usage becomes a matrix of dimension 5x5. The surface graph and closet surface fit are shown in figure 4.2

Then in order to formulate the CPU usage in terms of number of subscriptions and incom-

ing event rate, curve fitting tool has been used and the best possible curves are identified. This process gives us representative equations for the % CPU usage. The equation looks like following.

$$\textbf{Avg. \% CPU Usage} = c1 + c2 * x + c3 * y + c4 * x * y - c5 * y^2$$

Where c1, c2, c3, c4 & c5 are constants having following values.

c1= 1.365

c2= 0.0007655

c3= 0.06318

c4= $2.125e-05$

c5= $9.654e-05$

Value of all of these constants is dependent on the underlying machine configuration. The values may increase or decrease depending on the machine configuration.

Along with measuring CPU usage, % drop in notifications is also a good performance metric to be measured. So every single event being published by analysis engine, alert notification system tries to send alert for it. While doing so some of the notifications get dropped because of congestion at the system. Percentage of such dropped notification is what we call % drop in notifications. Since the notification system is logging notification drop, we plot a similar three dimension graph. Similar to CPU usage, curve fitting techniques have been used to come up with a mathematical equation representing the % notification drop. The following equation represents the % drop for event triggered approach.

$$\textbf{\% Notification drop} = c1 + c2 * x + c3 * y$$

Where c1, c2 & c3 are constants having following values.

c1= -0.1529

c2= $-3.02E-05$

c3= 0.002328

The % drop in notification for event triggered approach is quite low which suggests that this approach can be a good approach for a system wherein sending alerts is a critical activity. But later sections will compare all possibilities and give a decision flow chart for selecting a notification approach.

For the third metric, Notification delay, we found that average notification delay for event triggered approach is 1 second which suggests that as and when events come in to the system, corresponding alerts are quickly delivered to subscribers.

4.0.3 Periodic Notification approach

In the last section, we have described about event triggered approach for alert notification. This section discusses about the other possible approach i.e. periodic approach. Unlike event triggered approach, periodic approach doesn't trigger alert with each incoming event. Incoming events are stored in database and after a period P, all events in last P time are processed for sending alerts. While talking about periodic approach, we need to have a certain value for period. This value can range from few seconds to few minutes depending on the requirements of the system. Basically, choosing this value is a trade-off between the notification delay time(the time it takes a notification to reach GCM server once an event has been published) and average CPU usage. If a system is very sensitive to the notification delay time then they may make some trade-off on CPU usage and hence may even opt to use the event triggered approach, but on the other hand, if some system is willing to allow some delay in notification but can't allow CPU usage to go beyond certain values, for that system may choose to use periodic approach. The latter situation will hold for systems which are very sensitive to energy consumption.

The delay in notification increases with the increase in period, which is quite intuitive because the alert notification system won't run always but runs only every P seconds, so average notification delay would be approximately equal to the period P.

We experimented with periods 20 seconds, 40 seconds and 60 seconds. The following sections will talk about results in each case in detail.

Period 20 seconds

With periodic approach, having a period of 20 seconds, the alert notification system runs every 20 seconds and processes all events published in last 20 seconds. The objective behind periodic approach is to lower the CPU usage without effecting the system's notification services. Lowering the CPU usage will reduce the energy consumed by the system. The reason why periodic approaches may reduce CPU usage is that since the notification system is not running all the time, there would be some time when the CPU usage is very low because the system won't have any event to process during that time. This will reduce the average CPU usage although it might increase the maximum CPU usage, because of busty nature of processing.

We used the same benchmark described in section 4.0.1 with only change being in notification system used. We used notification system shown in figure 3.3. Table 4.2 shows the % CPU usage when period is 20 seconds.

Using surface fitting techniques, we plotted 3-dimensional graph for average CPU usage. Figure 4.3 shows the closest surface fit.

The mathematical equation, drawn from the surface fit, representing the CPU usage for period 20 is given below.

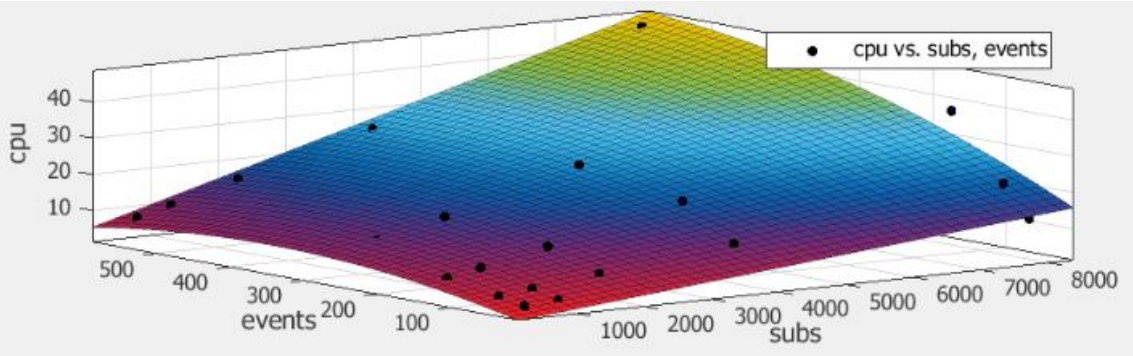

Figure 4.3: Surface fit for periodic notification approach, period 20 seconds

$$\textbf{Avg. \% Cpu usage} = c1 + c2 * x + c3 * y + c4 * x * y - c5 * y^2$$

Where c1, c2, c3, c4 & c5 are constants with following values.

c1=0.5813

c2=0.00171

c3=0.04961

c4=$5.95E-06$

c5=$7.25E-05$

Clearly the CPU usage is much lesser than the event triggered approach especially when number of subscriptions goes high and event rate also goes high. The maximum CPU used is approximately 46% which is even less than half of CPU usage in case of event triggered approach. Also, when incoming event rate reaches 280 events/second, the CPU usage kind of saturate and further increase in incoming event rate doesn't change the CPU usage significantly although changing number of subscribers still influences the CPU usage.

So, clearly for a system having high number of subscriptions and high incoming event rate, periodic approach is way better than the event-triggered approach. But the only problem is that in case of event triggered approach, the average delay in notification is approximately 2 seconds while in periodic approach with period 20, this delay increases upto 20 seconds. So now a trade-off is required between the CPU usage and notification delay. This trade-off is dependent on the system requirements. The following sections will give more insights about this trade off.

Along with measuring CPU usage for all periodic approaches, % drop in notifications is also measured. Similar to CPU usage, curve fitting techniques have been used to come up with a mathematical equation representing the % notification drop. For a period of 20 seconds, this equation comes out to be as following.

Event Rate	Total number of subscriptions				
	500	1000	2000	4000	8000
35	3.542857143	4.057142857	5.264516129	7.696774194	12.4983871
70	4.86875	5.985714286	8.050793651	12.39206349	20.96349206
140	7.357142857	9.181538462	13.09230769	21.52698413	38.43870968
280	8.813333333	11.51222222	15.97578947	26.36947368	46.46060606
560	8.297927461	10.82164948	15.846875	25.67563452	45.91684783

Table 4.2: Average % CPU usage for periodic (period 20 seconds) notification approach

Event Rate	Total number of subscriptions				
	500	1000	2000	4000	8000
35	3.566153846	3.99047619	4.938709677	6.993548387	11.41904762
70	5.015625	5.965625	7.938095238	11.6578125	20.83492063
140	8.14375	9.9171875	13.509375	21.2796875	37.7296875
280	8.16969697	10.318	14.76923077	25.74554455	46.48118812
560	7.985221675	10.46699507	15.3245098	25.44541063	46.0565

Table 4.3: Average % CPU usage for periodic (period 40 seconds) notification approach

$$\textbf{\% notification drop} = c1 + c2 * x + c3 * y$$

Where c1, c2, c3 are constants having following values.

c1=1.899

c2=5.31E-05

c3=0.00109

Average notification delay, for period 20 seconds, is approximately 22 seconds.

Period 40 seconds

The notification alert system with period 40 seconds runs every 40 seconds and processes all events published in last 40 seconds. We experimented with many periods because it is quite possible that different periods may have their different effects on the system performance. Similar to period 20 seconds approach, we used notification system shown in figure 3.3 and same benchmark setup described in section 4.0.1.

The table 4.3 represents the average CPU usage during the bench-marking process.

Using surface fitting techniques, we plotted 3-dimensional graph for average CPU usage. Figure shows 4.4 the closest surface fit.

The mathematical equation representing the CPU usage for period of 40 seconds is shown below.

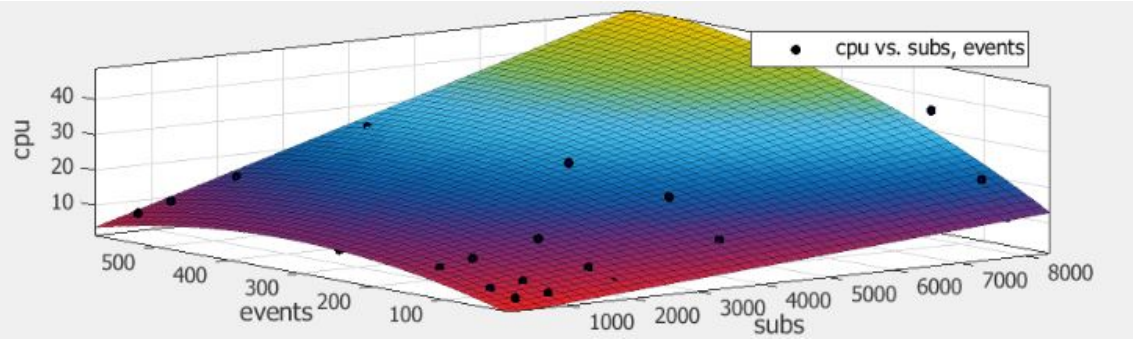

Figure 4.4: Surface fit for periodic notification approach, period 40 seconds

$$\textbf{Avg. \% Cpu usage} = c1 + c2 * x + c3 * y + c4 * x * y - c5 * y^2$$

where c1, c2 , c3, c4 & c5 are constants having following values.

c1 = 0.0973

c2 = 0.001407

c3 = 0.06214

c4 = 7.04E-06

c5 = 9.75E-05

A closer look at both Table 4.2 and Table 4.3 shows that these two scenarios doesn't differ much in performance. The difference in CPU usage in both cases, period 20 seconds and period 40 seconds, is very low. In both the cases, CPU usage tend to saturate after incoming event rate reaches 280 events/second.

Similarly, the equation representing % notification drop for period 40 seconds is given below.

$$\textbf{\% notification drop} = c1 + c2 * x + c3 * y$$

Where c1, c2, c3 are constants having following values.

c1=1.195

c2= 3.42E-05

c3=0.000341

Unlike % CPU usage, notification drop changes significantly. It reduces with the increase in the period. So it is a favourable factor for increasing period whenever possible.

Average notification delay, for period 40 seconds, is approximately 43 seconds.

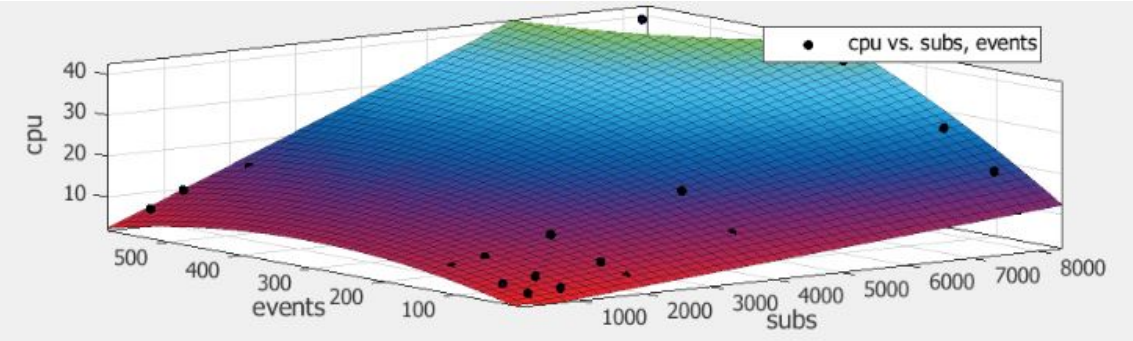

Figure 4.5: Surface fit for periodic notification approach, period 60 seconds

Period 60 seconds

The notification alert system with period 60 seconds runs every 60 seconds and processes all events published in last 60 seconds. With the same benchmark setup used in period 20 and period 40 seconds approach, period 60 seconds reflects difference from other two approaches, period 20 and period 40 seconds. Average CPU usage, in this case, is lower than both former approaches.

Table 4.4 shows average CPU usage for period 60 seconds.

Event Rate	Total number of subscriptions				
	500	1000	2000	4000	8000
35	3.58125	4.022222222	5.006349206	6.8359375	11.10793651
70	4.778125	5.7265625	7.493333333	10.92615385	19.265625
140	6.623611111	8.218571429	11.92794118	19.18656716	27.56097561
280	7.96122449	10.25833333	13.80612245	22.05588235	39.6254902
560	7.193534483	10.96770833	14.85789474	24.9102439	40.6039801

Table 4.4: Average % CPU usage for periodic (period 60 seconds) notification approach

Using surface fitting techniques, we plotted 3-dimensional graph for average CPU usage. Figure 4.5 shows the closest surface fit.

Mathematical equation representing CPU usage pattern with period 60 seconds is shown below.

$$\textbf{Avg \% CPU usage} = c1 + c2 * x + c3 * y + c4 * x * y + c5 * y^2$$

Where c1, c2, c3, c4 & c5 are constants having following values.

c1 = 0.4585

c2 = 0.00129

c3 = 0.04857

c4 = 8.85E-06

c5 = -7.97E-05

Compared to period 20 and period 40 seconds approaches, period 60 second approach have low CPU usage even in case of higher number of subscriptions and higher incoming event rate. Also like period 20 and period 40, period 60 too shows a saturation after the incoming event rate reaches to 280 events/second. Marginal increase in CPU usage decreases when incoming event rate reaches to 280 events/second.

This trend shows that period 60 second can be good choice for a system if it is not very sensitive to alert notification delay because period 60 approach would have an average notification delay of approximately 60 seconds.

Similarly, the equation representing % notification drop for period 40 seconds is given below.

$$\textbf{\% notification drop} = c1 - c2 * x + c3 * y$$

Where c1, c2, c3 are constants having following values.

c1= 0.9376

c2= 1.71E-05

c3= 0.0003419

The notification drop further decreases in case of period 60. So it follows a decreasing trend which is a favourable factor to increase period.

Average notification delay, for period 60 seconds, is approximately 65 seconds.

4.0.4 Dependence of CPU usage and notification drop on period

So far, we have seen dependence of CPU usage on incoming event rate and total number of subscriptions, for a given period. But it is important to know how does this CPU usage changes with the change in period. For computing a mathematical equation for dependence of CPU usage on all three factors i.e. number of subscribers, incoming event rate and period, curve fitting techniques are used. Since we have all the coefficients, from three equations for three periods, for all periods, so plotting these coefficients against period and then fitting a curve would give us the required relationship. The following equation represents this relation.

$$\textbf{Avg. \% CPU usage (x, y, p)} = (c1 + c2 * p) + (c3 + c4 * p) * x + (c5 + c6 * p) * y + (c7 + c8 * p) * x * y - (c9 + c10 * p) * y^2$$

Where c1 to c10 are constants with following values:

c1= 0.5018

c2= -0.00307

c3= 0.001889

c4= -1.05e-05

c5= 0.05448

c6= -2.6e-05

c7= 4.38e-06

c8= 7.25e-08

c9= 7.603e-05

c10= 1.8e-07

Similar to CPU usage, % notification drop also depends on the period. So, similar to calculations of CPU usage, dependence of notification drop on period has been calculated, and the following equation represents that.

$$\textbf{\% notification drop} = (c1 + c2 * p) + (c3 + c4 * p) * x + (c5 + c6 * p) * y$$

Where c1 to c6 are having following values:

c1 = 2.305

c2 = -0.02403

c3 = 9.36e-05

c4 = -1.755e-06

c5 = 0.001339

c6 = -1.87e-05

Value of these constant may vary across different machines, these values are for our experimental configuration discussed in earlier sections. In the above equations, x can have a value from *1 to 8000* and y can have a value from *1 to 560*.

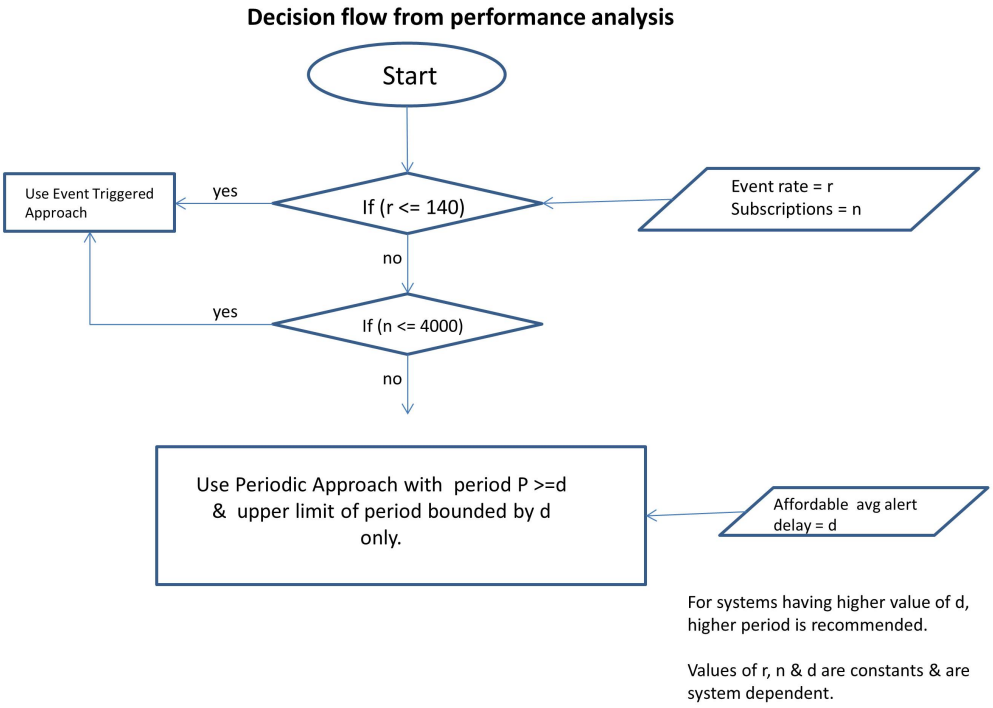

Figure 4.6: Decision Flow Chart

4.0.5 Potential Design Guidelines

Based on the above bench-marking results and mathematical equations, we can think of a recommended notification approach for a similar CSS kind of platform. Depending on the given system parameters like expected number of subscriptions, average incoming event rate & total notification delay that the system can tolerate, one can find out which approach out of two, event triggered or periodic approach, is better to use and at what time it is better to switch from one approach to another. If periodic approach is used, a major decision is to decide what is better value of period to use. This section presents a decision flow chart to decide these things.

For a system, given the average number of subscriptions and total incoming event rate, the design policy tries to select an approach yielding in minimum average CPU usage and minimum & notification drop. Design policy also suggests a way to select period, in case of periodic approach.

Figure 4.6 represents a decision flow chart. The flow chart suggests that if the expected event rate r is less than 140 events per second, then irrespective of number of subscriptions, one can use event triggered approach. Event triggered approach, upto this value of event rate, won't have high CPU usage. % Notification drop and notification delay values are also low. So for event rate less than 140, event triggered approach is better to use.

Further, if the event rate, r, is greater than 140 events per second, but average number of subscriptions is less than or equal to 4000, still event triggered approach can be used. So, for these two scenarios, event triggered approach is a better approach.

But, if the average incoming event rate and average number of subscribers, both are not satisfying any of the above two scenarios, then better approach is to shift from event triggered approach to periodic notification approach. At this point, using event triggered approach would cost heavy CPU usage which will be consuming a lot of system energy, so for those systems which are sensitive to energy consumption, it is probably good point to shift to periodic approach. Now the next question to be answered is what should be the value of the period.

From our bench-marking, we have seen that since the % CPU usage is decreasing with the increase in the period value and also % notification drop value is also decreasing with the increase in period, so both these factors point higher period value. But only factor that puts a cap on the period is average notification delay that the system can tolerate because with the increase in period, average notification delay also increases. In fact delay is almost equal to the period value. So, if a system is sensitive to delays, then this factor can put a cap on the period value.

The values of constants, r, n and d, are dependent on the underlying machine configuration and likely to change if bench-marking is repeated on machine with different configuration. These constants are given for the machine configuration defined in subsection 4.0.2.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

Although research work in area of sensing, including participatory sensing and opportunistic sensing, has been significant, yet none of the work focuses on providing Sensing-as-a-Service platform. This work focuses on designing a Cloud-based Sensing-as-a-Service platform and its performance modeling.

This work proposed architecture of Cloud-based Sensing-as-a-Service platform. In the proposed CSS platform, users can report events happening around them, to a cloud-hosted service. The platform can provide services to its consumers, in terms of subscription based queries and event data driven queries. Architectures for both types of service, subscription based and event data driven, have been discussed separately.

The proposed CSS platform has HTTP based APIs for sensor data collection, HTTP based APIs for using platform services, primitive queries, abstract queries and set of modules to process sensor data to provide consumer services. These services can be in terms of alert notifications for consumer subscriptions or consumer initiated queries for particular type of event(s).

The concept of abstract queries have been used to help end consumers frame queries easily. A set of modules, also called binaries, can be used for sensor data processing. Each module processes one corresponding abstract query, which includes generating equivalent primitive queries, from abstract query, and processing database results. The binaries communicate to core engine, which in turn communicates with the end consumer. Core engine manages all binaries and invokes a corresponding binary for incoming abstract queries.

Platform architecture design is followed by its performance modeling, using the developed bench-mark application. The platform has been benchmarked for modeling CPU usage, % notification drop and notification delay, in terms of incoming event rate and subscription based platform services. The result of all this bench-marking is a system design guideline, represented by a decision flow chart, which helps in identifying a better approach for providing platform services. The design guideline describers, for a given CSS kind of platform, which notification approach,

out of event triggered(used by traditional publisher-subscriber systems) or periodic notification, is better to use. Based on this guideline, system can dynamically switch between these two approaches as and when needed.

5.2 Future Work

Future work would be extension of platform, in terms of platform services and performance modeling. The platform can serve two type of queries, subscription based and event data driven, performance modeling at current state, considers subscription based queries and not event data driven queries. So, one extension of existing system would be to extend the benchmark application to include event data driven queries along with subscription based queries. Then, bench-marking can be repeated for both type of services.

The benchmark application considers subscribers upto 8000. This should be increased further, to improve the benchmarking results and to represent a more realistic scenario. Number of subscriptions can be varied as Gaussian distribution or Poission distribution, which can be determined using ethnographic study of user behaviour.

This work benchmarks the platform for three periods namely 20 seconds, 40 seconds and 60 seconds. Future work will include more number of periods in benchmark application, making it applicable for more systems, capable of handling more incoming events and offering more platform services.

Bibliography

[1] Amazon. Amazon mechanical turk. `https://www.mturk.com/mturk/welcome`.

[2] Caporuscio, M., Carzaniga, A., and Wolf, A. Design and evaluation of a support service for mobile, wireless publish/subscribe applications. *Software Engineering, IEEE Transactions on 29*, 12 (Dec 2003), 1059–1071.

[3] Chen, J., Arumaithurai, M., Jiao, L., Fu, X., and Ramakrishnan, K. Copss: An efficient content oriented publish/subscribe system. In *Architectures for Networking and Communications Systems (ANCS), 2011 Seventh ACM/IEEE Symposium on* (Oct 2011), pp. 99–110.

[4] Costa, P., Migliavacca, M., Picco, G., and Cugola, G. Epidemic algorithms for reliable content-based publish-subscribe: an evaluation. In *Distributed Computing Systems, 2004. Proceedings. 24th International Conference on* (2004), pp. 552–561.

[5] Das, T., Mohan, P., Padmanabhan, V. N., Ramjee, R., and Sharma, A. Prism: Platform for remote sensing using smartphones. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services* (New York, NY, USA, 2010), MobiSys '10, ACM, pp. 63–76.

[6] Demers, A., Gehrke, J., Hong, M., Riedewald, M., and White, W. Towards expressive publish/subscribe systems. In *Proceedings of the 10th International Conference on Advances in Database Technology* (Berlin, Heidelberg, 2006), EDBT'06, Springer-Verlag, pp. 627–644.

[7] Deng, G., Xiong, M., Gokhale, A., and Edwards, G. Evaluating real-time publish/subscribe service integration approaches in qos-enabled component middleware. In *Object and Component-Oriented Real-Time Distributed Computing, 2007. ISORC '07. 10th IEEE International Symposium on* (May 2007), pp. 222–227.

[8] Eriksson, J., Girod, L., Hull, B., Newton, R., Madden, S., and Balakrishnan, H. The pothole patrol: Using a mobile sensor network for road surface monitoring. In *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services* (New York, NY, USA, 2008), MobiSys '08, ACM, pp. 29–39.

[9] Gehrke, J., and Madden, S. Query processing in sensor networks. *Pervasive Computing, IEEE 3*, 1 (Jan 2004), 46–55.

[10] Marcelo. https://code.google.com/p/fincos/.

[11] Mendes, M. R., Bizarro, P., and Marques, P. Fincos: Benchmark tools for event processing systems. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering* (New York, NY, USA, 2013), ICPE '13, ACM, pp. 431–432.

[12] Mohan, P., Padmanabhan, V. N., and Ramjee, R. Nericell: Rich monitoring of road and traffic conditions using mobile smartphones. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems* (New York, NY, USA, 2008), SenSys '08, ACM, pp. 323–336.

[13] Mun, M., Reddy, S., Shilton, K., Yau, N., Burke, J., Estrin, D., Hansen, M., Howard, E., West, R., and Boda, P. Peir: the personal environmental impact report, as a platform for participatory sensing systems research. In *in Proc. ACM/USENIX Int. Conf. Mobile Systems, Applications, and Services (MobiSys) Krakow* (2009).

[14] Oh, S., Kim, J.-H., and Fox, G. Real-time performance analysis for publish/subscribe systems. *Future Gener. Comput. Syst. 26*, 3 (Mar. 2010), 318–323.

[15] Pereira, J., Fabret, F., Llirbat, F., and Shasha, D. Efficient matching for web-based publish/subscribe systems, 2000.

[16] Perera, C., Zaslavsky, A. B., Christen, P., and Georgakopoulos, D. Sensing as a service model for smart cities supported by internet of things. *CoRR abs/1307.8198* (2013).

[17] Ra, M.-R., Liu, B., La Porta, T. F., and Govindan, R. Medusa: A programming framework for crowd-sensing applications. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services* (New York, NY, USA, 2012), MobiSys '12, ACM, pp. 337–350.

[18] Ramasubramanian, V., Peterson, R., and Sirer, E. G. Corona: A high performance publish-subscribe system for the world wide web. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3* (Berkeley, CA, USA, 2006), NSDI'06, USENIX Association, pp. 2–2.

[19] Sachs, K., Appel, S., Kounev, S., and Buchmann, A. Benchmarking publish/subscribe-based messaging systems. In *Proceedings of the 15th International Conference on Database Systems for Advanced Applications* (Berlin, Heidelberg, 2010), DASFAA'10, Springer-Verlag, pp. 203–214.

[20] Sachs, K., Kounev, S., Appel, S., and Buchmann, A. A Performance Test Harness For Publish/Subscribe Middleware. In *SIGMETRICS/Performance 2009 International Conference, Seattle, WA, USA, June 15–19, 2009* (June 2009). (Demo Paper).

[21] SPEC. Specjms2007. http://www.spec.org/jms2007/, 2009, 2009.

[22] Tridib, D., Koustuv, A., and Amit, A. Cityzen: A cost-effective city management system with incentive-driven resident engagement. In *15th International conference* (2014), MDM, 2014, IEEE.

[23] UK. Fix my street, a smart system for fixing problems around your streets in uk. `http://www.xmystreet.com/`.

[24] Ushahidi. Ushahidi system, china. `http://blog.ushahidi.com/2012/06/05/ushahidi-beijing/`.

[25] Zhai, L., Sun, L., and Liu, Y. Modeling and evaluation of high-performance publish-subscribe system. In *Computational Intelligence and Design, 2008. ISCID '08. International Symposium on* (Oct 2008), vol. 1, pp. 457–460.