Computers **2015**, *4*, 176-214; doi:10.3390/computers4030176

OPEN ACCESS

computers

ISSN 2073-431X

www.mdpi.com/journal/computers

Project Report

A Cloud Based Mobile Dispatching System with Built-in Social CRM Component: Design and Implementation

Cosmina Ivan * and Razvan Popa

Department of Computer Science, Faculty of Automation and Computer Science, Technical University of Cluj Napoca, Baritiu 26, 400027 Cluj, Romania; E-Mail: Ryan.Popa@yahoo.com

* Author to whom correspondence should be addressed; E-Mail: cosmina.ivan@cs.utcluj.ro; Tel.: +40-744-223-652.

Received: 20 April 2015 / Accepted: 25 June 2015 / Published: 2 July 2015

Abstract: *Mobile dispatching applications* have become popular for at least two major reasons. The first reason is a more mobile-centric usage pattern, where users relate to apps for fulfilling different needs that they have. In this respect, a vehicle dispatching application for mobile phones is perceived as a modern way of booking a vehicle. The second reason has to do with the advantages that this method has over traditional dispatching systems, such as being able to see the vehicle approaching on a map, being able to rate a driver and the most importantly spurring customer retention. The taxi dispatching business, one of the classes of dispatching businesses, tends to be a medium to lower class fidelity service, where users mostly consider the closest taxi as opposed to quality, which is regarded as being at a relatively consistent level. We propose a new approach for the taxi ordering application , a mobile dispatching system, which allows for a more engaged user base and offers fidelity rewards that are used to enhance the customer retention level based on a built in social customer relationship management (CRM) component. With this approach, we argue that in a business world which is shifting from a consumer-centric marketing to a human-centric model, this apps will allows taxi businesses to better interact with their clients in a more direct and responsible manner. Also this distributed system helps taxi drivers, which can receive orders directly from their clients and will be able to benefit from offering quality services as they can get higher ratings.

Keywords: mobile computing; cloud; dispatching system; social CRM

1. Introduction

The growth of *mobile computing* has outpaced any previous consumer electronic in terms of growth speed. This has enabled traditional expensive equipment to be displaced by easy to use Software as a Service product. We have seen the smartphone becoming the interface for anything that can be remotely controlled and the center of the entire online presence including email, social networks and professional activities [1]. This growth of software designed for mobile devices is set to continue over the course of the following decades. The mobile revolution brought changes in consumer behavior and the way people interact. New mobile devices come equipped with sensors such as GPS, accelerometer, gyroscope, altimeter, camera and thermometer. It also generates huge amount of metadata that can be captured and analyzed for discovering useful hidden patterns. Not to mention that they make use of a large bandwidth Internet connection that over 4G LTE (Long Term Evolution) can reach speeds of tens of megabits per second.

The rise of the *cloud* has changed from being a novelty in five years ago, to being the default deployment choice for most software products. While scalability was one of the top concerns five years ago, it has been now elegantly addressed by virtualization. Also, the price of cloud hosting has decreased with increased competition in the field, from both large enterprises such as Amazon, Microsoft or Google but also new organizations that found a niche with specific programming languages or types of software such as Digital Ocean or SoftLayer [2]. Competition between the cloud providers has become extremely sharp across all layers of the cloud, especially PAAS (Platform as a Service) and SAAS (Software as a Service), but we also observed that all IAAS (Infrastructure as a Service) vendors have included tools that compete with platform offerings, this is due to commoditization of the IAAS layer, which makes all providers want to raise on the ladder of technology offerings and extend their services into higher margin areas.

Given the ubiquity of smartphone devices, we are seeing an increased integration between different systems in the sense of reducing the time between activities required to be performed by people who are part of the same process. In this respect, CRM applications have integrated real time filtering of social networking data, in order to identify possible risks and opportunities as early as possible. These features have become a common component of any customer-facing B2C system. In this respect is not enough anymore to just have a working system, but to have a complete ecosystem that can track users from acquisition towards using the product and being able to reactivate them. The smartphone install base will continue to expand and this creates opportunities for mobile based systems to displace traditional more expensive systems. One of these systems is *mobile dispatching.* Traditional dispatching systems have been expensive as they required both hardware and software purchases. The hardware system is composed of a mobile data terminal with either an Internet connection or a radio connection, now these hardware terminals can be completely replaced with a smartphone or a tablet, which offers the same functionality. Dispatching applications were the first candidates for displacing a higher cost existing technology. By using cheap consumer level mobile devices it is possible to offer complex dispatching systems on a SAAS basis. With the rising adoption of smartphones by the consumer base, a large variety of applications meant at simplifying users daily life and enhancing productivity are appearing. Vehicle booking applications have first become popular in western countries, as this region offered a higher income level that determined a larger smartphone penetration.

Modern dispatching systems can now be built purely using software, and using mobile devices as the in-vehicle hardware. Furthermore the dispatching system can be hosted in a *public cloud*, which offers built in scalability and reliability. In this case, when hardware cannot be seen as a differentiator between systems and considering the rapid review cycle that an application such as a *taxi booking engine* usually receives, having an *integrated customer analysis solution* is extremely important. The dispatching system can be offered as white label product and sold on a monthly subscription. While the margins on selling software are generally high (80%–90%), the competition can also be very tough. Also when addressing current customers that have implemented traditional dispatching systems that use proprietary hardware equipment, the sale proposition will be weak unless the current offering includes *strong marketing or CRM (Customer relationship management) capabilities* that can be correlated to revenue growth. One important feedback mechanism which can be used in such a system is social networks. Considering for example a taxi ordering application, drawing new clients to using the service is strongly connected to the influence their network of friends have on promoting this service.

This paper is organized as follows: Section 2 encompasses the research we have done on existing dispatching systems with the aim of understanding the current enterprise and technological landscape and be able to contribute with new improvements. Section 3 describes the architectural design decisions and implementation considerations for the system, based on the integration of two subsystems: the *Dispatching application* and the *Social CRM component.* Section 4 describes the testing scenarios that we have envisaged and considered for an initial evaluation and Section 5 make conclusions and presents future developments.

2. Bibliographic Research

2.1. Similar Systems and Applications

Traditional dispatching applications have incurred high costs attributed to purchases of physical equipment and revolving licensing fees. The first attempts to use mobile devices for logistic activities took place in the last decade, with GPRS based positioning. However the potential complexity of the applications built on this infrastructure was extremely low and they did not succeed on taking enough market share. Also these systems required a human operator that would manually insert or confirm the orders into the system. As more automation was necessarily, some logistic companies started using voice recognition and SMS as a more efficient way of placing an order into the system. In the last years, the number of proprietary dispatching apps has skyrocketed, with Uber being the most visible international player in the space. However the market remains segmented, with very few companies offering complete dispatching solutions.

We present here a set of similar services that we have analyzed from the literature based on their functionalities, and then in Table 1 creating a comparison matrix. We have inspired form these systems in determining the specifications of the system we are building.

Table 1. Similar services comparison matrix.

Criteria/Service	***T Dispatch***	***SAP-SCM***	***DDS***	***Hailo***	***Taxi Magic***	***Uber***	***MTData***	***OUR SYSTEM***
Web presence	https://tdispatch.com	http://Go.sap.com	http://digital-dispatch.co.uk	https://www.hailoapp.com	http://gocurb.com	https://www.uber.com	http://www.mtdata.com.au	www.taxiarmy.com
iOS support	X	-	X	X	X	X	X	X
Android support	X	-	X	X	X	X	X	X
Cross platform support	X	-	-	-	-	-	-	X
Client Usage Analytics	X	X	-	-	X	X	X	X
Show nearby vehicles	-	-	-	X	X	X	-	X
Calculate shortest route	-	X	-	-	X	X	-	X
API available	-	X	X	X	-	-	-	X
Customizable design	X	X	X	-	-	X	-	X
Management panel	X	X	X	-	-	-	-	X
Geofencing	-	X	-	X	-	-	X	X
Shared vehicle ownership	-	-	-	-	-	-	-	X
Payment processing	X	X	X	X	X	X	X	X
Social Media integration	-	-	-	X	X	-	-	X
User satisfaction dashboard	-	-	-	-	-	-	-	X
Customer lifecycle management	-	-	-	-	-	-	-	X

- *T-Dispatch* is the most similar system to our project, offering emerging taxi companies the ability to offer their users a modern app based dispatching solution. They have both driver and user apps.
- *SAP SCM* is a product line offered by the largest software enterprise in Europe. SAP enhanced their logistic module with the ability to book, track and dispatch vehicles using mobile devices and management boards. Their entrance into this space was generated by demands from their costumers that required lower cost alternatives for managing their logistics. SAP does not offer the dispatching system as a separate product, but as an extension to their mature logistics system.
- *DDS* is a UK based company that has been focused on dispatch technology for over two decades. They have clients around the world and until mobile devices became available, they were one of the leading global suppliers of Mobile Dispatch Terminals (MDT). They entered into the mobile

dispatching market in 2013 with a lower cost dispatching system. The key differentiator for our product versus DDS is the powerful management dashboard. DDS is not focused on customers, preferring to push forward their dispatching technology and not focusing on user retention and other metrics that are important for smaller companies.

- *Hailo* is an app that has attracted attention in UK, Ireland and North America, offering a simpler way for users to book a taxi using their mobile phone. They use the existing taxis and operated at a level equal to taxi dispatch companies. Their model is similar to other common apps as GetTaxi and MyTaxi.
- *Taxi Magic* was one of the first services that offered the ability to order a vehicle without calling, a system in which ordering a taxi is fairly simple: after launching Taxi Magic, your iPhone will use GPS or triangulation to determine your general location, and will present a list of nearby cab services (some listings are only phone numbers, while others fully support Taxi Magic's ordering system). After choosing a supported cab company, the application will ask for your exact street address and will then take you to a status screen that will alert you once your ride is dispatched (wait times can vary depending on the time of day and location). The status screen also allows you to view how far away the cab is and the driver's name. From this system we make use of ideas on localization.
- *Uber* .Luxury transportation in sleek black car is based on the business model of offering private drivers as taxi. It is available in largest cities around the world. The app offers functionalities of ordering a taxi by getting the users GPS positions, it allows tracking the driver as he approaches and when the ride is over, it allows for payment using credit card. While TaxiMagic operates by simply displacing the telephone as the ordering channel with an order sent from the smartphone, other services have appeared which work differently. This service also operate by having both a driver app and a client app. When the client orders a taxi, the order is submitted to the closest driver, instead of going through an operator. By having a mobile app, the driver can be tracked by the client so the client can see when his taxi arrives.
- *MTData* is another traditional dispatch system provider that has entered the mobile dispatch market. They are still pushing their traditional dispatch technology including physical MDT (Mobile Data Terminal), but they also adopted SAAS products that can be offered independently on in conjuncture with their traditional systems.

From the Table 1 we can conclude that our system plan to include functionalities from analyzed solutions on the market, emphasizing on the specific CRM component which offer a unique characteristic to the proposed approach.

2.2. Social CRM Component

As the dispatching marketplace is becoming overcrowded with both providers that have tailored their existing systems to offer mobile dispatching and new providers, we needed a way to differentiate our product. We identified that most enterprise clients interested in accessing the technology that we offer have two primarily goals:

- Better understanding of their clients, retention levels, marketing analytics, customer activation levels.

- New marketing channel and consumer feedback mechanism that can be used to improve the product and the customer experience.

We are differentiating ourselves from the other alternatives by being a more lightweight system, that is easy to set up and which provides the business with actionable insights on their customers. The strong integration between the *dispatching component* and the *marketing-purposed customer analytics dashboard* provides a unique selling point. Most companies already have a mechanism of handling each of these two use cases, but the problem is that the systems are decoupled. This makes automation of some processes impossible. Our system is able to aggregate data on the users, create actionable metrics and automate specific tasks that offer maximum benefit to the business.

One such scenario is gathering data across all major social networks for a specific filter. For example we can filter for any mentions of the company name. After all the data is gathered, it is normalized and aggregated into our data stores in real time. At this point we also analyze the data, for example we analyze what kind of sentiment is associated to the data, and scale the sentiment on a −20 to +20 scale. If the sentiment is highly negative, we can automate the system to reply to the users post, offering him a coupon or apologizing for any inconvenience he faced using the system. If the sentiment is highly positive, we might encourage the user to invite his friends by offering discount rates or free coupons.

Statistics show that is four times harder to regain a lost customer than to get a new one. However, if the company is able to respond quickly to a negative experience suffered by the client, it can avoid losing the customer. With customer satisfaction increasing in importance every day across all enterprises, this is to become a major incentive for adoption of our system. For example if a client booked a vehicle but experienced any kind of frustration, there is a large chance this user will tweet about it or post it on Facebook. Our system will capture this negative comment within less than 10 s and offer any customer support people the means to identify what the issue was in the specific case and convince the client to retry the service. Our analytics dashboard can also be used for monitoring the impact of specific marketing campaigns. In order for this data to provide actionable insights, it is important to have a minimum delay between the moment any comment is published on the Internet and the moment an action is taken, being it an automatic action (such as a *thank you note*) or a customer support call meant at retaining a user that has gone through a negative experience.

Most of this data is gathered across social networks, but we are not limited to them. Twitter is one place where customers express their frustration or excitement within seconds, thus we needed access to the Twitter dataset. Facebook is another social network that can offer important information on the user, but as we mentioned we are not limited to using social networks. The service also allowed the enterprise to access any information matching the specific filters anywhere the Internet, such as Blog posts or comments written on news websites. In order to have access to all this data sources, we are using DataSift [3], which together with Gnip [4] are the only two companies in the world that have access to all the Twitter data in real time, and also their systems span across Twitter, offering access to over 20 data sources including Facebook, Blogs, Q&A websites and many other.

3. System Design

After analyzing the currently available systems and the functionalities they offer, we started analyzing the infrastructure that should back our system. According to [1] we opted for a Cloud-based service to offer the infrastructure for data synchronization between the apps and the operator panel. We also make use of the best practices of developing mobile applications by migrating data and processing power inside the Cloud. We have studied how to realize the integration between the service and the Cloud and the benefits of this approach and we analyzed the means of implementing web services for fulfilling this goal.

We developed an API for our service which would be consumed inside the mobile application. We decided on using REST web services as they preferred when working with mobile applications. We needed the application to be available on as many mobile phone operating systems as possible, so we continued by analyzing cross platform mobile development solutions. After comparing Titanium development with other development frameworks we decided on using this platform because it offered a series of advantages such as compiling natively and being able to be extended in order to support more advanced functionality available for iOS or Android.

One important requirement was that the cross platform development supports *GPS*. According to [5] we make use of how to implement *GPS* into our application and how to determine *GPS* coordinates with various precisions. We can still determine the approximate user position when *GPS* is not available, such as when the user is located inside a building, which is a very common use case for our system. The way to identify the position of the user in such condition is by *GPS* triangulation. Even without a *GPS* receiver, the cell phone can provide information about your location. A computer can determine the location based on measurements of your signal, such as: its angle of approach to the cell towers, how long it takes the signal to travel to multiple towers and the strength of your signal when it reaches the towers.

The specific system that we are developing allows clients (users) to book vehicles directly from their mobile device without having to call through an operator. Having an app for this use case opens the doors for a wide variety of marketing strategies such as offering promotions, coupons, paying by credit card, offering subscriptions. This allows users to have an alternative to the traditional method of booking a vehicle by phone calling, based on the new and powerful trend of mobile cloud applications.

3.1. Functional and Nonfunctional System Characteristics

Software is becoming critical in driving the information-based economy. Time-to-market, robustness, and quality are important factors for measuring the success of systems in competitive economies and environments. There is also a need to evolve and adapt rapidly in an environment of continuous changes of business requirements. Therefore, there are stronger needs for standards and formalizations of the processes by which we specify and capture these requirements. Requirements capture is one of the initial phases to undertake in the system development process.

The functional requirements (FR) have been decomposed into low-level functional specifications that are depicted in Table 2. These specifications have been assigned priorities which are consistent with the iterative approach of development that we are following.

Table 2. Functional Requirements.

FR1	**Booking a vehicle by clients**	High
FR1.1	Ordering based on GPS location	High
FR1.2	Add precise location	High
FR1.3	Remember used locations for suggestions	High
FR1.4	Display map with user location	High
FR1.5	Allow dragging the position marker on the map	Medium
FR1.6	Get GPS location based on wireless	High
FR1.7	Handle cases when GPS location cannot be obtained by allowing only exact address input	High
FR2	**Vehicle Dispatching**	High
FR2.1	View active orders	High
FR2.2	Respond to active order	High
FR2.2.1	Respond with car id and time until it arrives	High
FR2.2.2	Mark initial car position on a map	Medium
FR2.3	Blink and sound alert on new order	High
FR2.4	Call phone for order not processed within 1 min	Low
FR3	**Order confirmation**	High
FR3.1	Display vehicle id and time until it arrives	High
FR3.2	Allow order canceling	High
FR3.3	Allow vehicle check-in	Medium
FR3.4	Allow push notification if app is closed	High
FR4	**Authentication**	High
FR4.1	Allow Facebook authentication	High
FR4.2	Allow authentication with username, password and phone number	High
FR4.3	Verify phone number	Low
FR5	**App promotion trough social networks**	High
FR5.1	Allow Facebook sharing of app use	High
FR5.2	Prompt for app review after 1 successful ride	High
FR6	**Feedback**	High
FR6.1	Allow feedback from button in tab bar	High
FR6.2	Allow vehicle check-out	Low
FR7	**User post-ride functionalities**	High
FR7.1	Allow distance displayed on check out	Low
FR7.2	Prompt user for driver rating on check out	Low
FR8	**Management Dashboard and Social CRM**	High
FR8.1	Filter data in real time across multiple social networks	High
FR8.2	Get an overall sentiment analysis of the service provided for specific time intervals	High
FR8.3	Data augmentation in real time and Metadata generation	High
FR8.4	Export data to a data store	High

Non-functional requirements (NFR) are properties and qualities the software system must possess while providing its intended functional requirements or services. These types of requirements have to be considered while developing the functional counterparts. They greatly affect the design and implementation choices a developer may make. They also affect the acceptability of the developed software by its intended users. In the following, we briefly describe the three categories of non-functional requirements that may be imposed on a software system.

System-related NFRs: These types of requirements impose some criteria related to the internal qualities of the system under development and the hardware/software context in which this system will operate.

- Operational requirements: The Dispatching Application will be running on iOS and Android phones, and will be using Cloud services for data synchronization and storage.
- Manageability requirements: The manager will be able to administer customer complaints and to scan social networks for potential issues regarding the service. He should also be able to use the information in the system databases regarding clients previous booking in order to direct advertising campaigns and monitor the overall client satisfaction with the service.
- Performance requirements: For our system these requirements are considering the response time of data synchronization.
- The response time to an order is dependent of the other user inserting the response manually (the operator or the driver being a human not a system that can respond automatically). Data synchronization must be assured to take place bellow 2 s.
- Maintainability requirements: The system must be designed in such a way that it allows a modifiable, component based system that allows future modifications and functionality extensions without breaking the present working version. This requirement is especially valid for our mobile application, as different mobile application versions installed on smartphone must communicate effectively with the Cloud-based API. In order to achieve maintainability we relied on API versioning for this specific requirement.
- Portability requirements: The system must work on a variety of mobile OS, of which iOS and Android are mandatory. A cross platform solution must be implemented to allow maximum code reuse with little adaptation.
- Security requirements: The application must use the .NET security framework for assuring authentication and authorization security constraints, and must also allow for SSL communication.

Process and Project-related NFRs: These types of NFRs impose criteria to be followed while developing the project.

- Conformance to standards requirements: The standard developing pattern for Titanium applications is MVC, which should be followed in order to assure maximum code reusability and maintainability.
- Development process requirements: The project needs to be developed in an iterative method, allowing for partial system components development and testing.
- Testing requirements: The API for the Cloud infrastructure needs to be tested using a unit testing approach, while the mobile vehicle booking application need to be tested using use-case testing.
- Installation and deployment requirements: The mobile applications need to be deployed to the AppStore and GooglePlay accordingly.

Human-related NFRs: These types of requirements deal with constraints related to the stakeholders and the social and societal context in which the system is deployed.

- Usability requirements: The application needs to be easy to understand and use by new users. It must also feature a demo the first time the application is launched.
- Look and Feel requirements: The mobile application needs to adhere to the usual UI components and not define custom components that might reduce the learnability of the system.
- Legal requirements: The application should not allow for malicious uses of personal data required by the application such as mobile phone numbers. This data must be stored on the Cloud and should be accessible only for specific users, if needed.

Device-related NFRs

- GPS: The device must feature GPS capable services for approximate user position identification.
- Internet: The device must allow for Internet connection for sending order, receiving responses and for the overall interconnection with the Cloud services.
- Operating system: The device must work with any iOS or Android version.
- Screen resolution: The application must work properly on any screen resolution. The application must be compatible with both the Retina display and the lower resolution display screens.

Social Network related NFR

- Completeness: The system must be able to scan for customer feedback across multiple social networks.
- Metadata: The data should be augmented in order to provide location context to the social network posts.
- Usability: The dashboard should be easy to operate and with meaningful.

3.2. Technological Perspective

3.2.1. Cross Platform Mobile Applications

According to Gartner, at the beginning of 2014, Android had 78.4% market share, while iOS had 15.6% of new worldwide sales. This means that their combined market share is more than 90% of all new smartphone sales. However, even if the two players dominate the market, developing for two frameworks is time consuming as very little work can be reused, the programming language and development environment for *Android* and *iOS* being very different. Also if another *OS* must be targeted in the future, this would require rewriting the application. In order to increase productivity a cross platform approach can be used. There is currently a wide range of cross platform solutions available including *Appcelerator Titanium*, *RhoMobile* and *PhoneGap*. *Appcelerator Titanium* [6] is an open source project and has integrated most common functionalities of *iOS* and *Android*. Developing with Titanium compared to other Cross Platform solutions is that it allows for native development. When using *Titanium* you are programming in *JavaScript* and the framework translates the code into native *Java* or *Objectual-C*. The development environment for Titanium is using its own *IDE* based on *Eclipse*, *Titanium Studio*, and it uses the JavaScript development language. It can also be extended using module in order to offer complete native functionality, which is not supported by the Titanium platform, and the recommended way of development is using a *MVC* pattern.

3.2.2. Windows Communication Foundation (WCF)

Windows Communication Foundation [7] is a framework for building service-oriented applications. Using *WCF*, applications can send data or asynchronous messages from one service endpoint to another. An endpoint can be a client of a service that requests data from a service endpoint. The messages can be as simple as a single character or word sent as XML, or as complex as a stream of binary data. In WCF all the communication details are handled by channel, it is a stack of channel components that all messages pass through during runtime processing. When are calling WCF service through a proxy class on the client side, it will send message (request soap message mainly includes some parameter values of method) and first the message will go through protocol Channels which mainly supports for security, transactions and reliable messaging, and second the message will go through encoder which convert messages into an array of bytes for transport, finally, the encoder message will go through the bottom transport channel which are responsible for transporting raw message bytes. WCF provides a number of transport protocols, including HTTP, TCP, MSMQ, peer-to-peer, and named pipes. In summary, *WCF* is designed to offer a manageable approach to creating Web services and Web service clients.

3.2.3. Microsoft Entity Framework

The Microsoft ADO.NET Entity Framework [8] is an Object/Relational Mapping (ORM) framework that enables developers to work with relational data as domain-specific objects, eliminating the need for most of the data access plumbing code that developers usually need to write. Using the Entity Framework, developers issue queries using LINQ, then retrieve and manipulate data as strongly typed objects. The Entity Framework's ORM implementation provides services like change tracking, identity resolution, lazy loading, and query translation so that developers can focus on their application-specific business logic rather than the data access fundamentals.

3.2.4. SignalR

Traditionally web applications used a polling technique for getting updates. For the client application this is not feasible as the polling interval is unknown. For example the driver could wait for hours without receiving an order, so he would poll thousands of times to receive one favorable result. Also the polling interval would have to be below 2 s (to offer a response time as low as possible), which would mean a high demand on the server and low scalability.

SignalR [9] is an Asynchronous library for .NET to help build real-time, multi-user interactive applications and is based on long polling techniques. Long polling is a variation of the traditional polling technique and allows emulation of an information push from a server to a client. With long polling, the client requests information from the server in a similar way to a normal poll. However, if the server does not have any information available for the client, instead of sending an empty response, the server holds the request and waits for some information to be available. Once the information becomes available (or after a suitable timeout), a complete response is sent to the client. The client will immediately re-request information from the server, so that the server will always have an available waiting request that it can use to deliver data in response to an event. Long polling is not a push code, but can be used under circumstances where a real push is not possible. Using long polling in the client mobile application gets notified to update its data, and thus to synchronize with the server, exactly as the new data is available. Using a long polling approach there is a very low demand on the server which translates into a high scalability, as new users and drivers are added to the distributed dispatching system.

In the .NET framework SignalR represents an implementation of long polling. However SignalR does not rely only on the long polling technique. SignalR has a concept of transports, each transport decides how data is sent/received and how it connects and disconnects. Transports build into SignalR are: WebSockets, Server Sent Events, Forever Frame, Long polling. SignalR tries to choose the "best" connection supported by server and client.

3.2.5. Push Notifications

Mobile apps are allowed to perform only very specific activities in the background, so battery life is conserved. However, it is required that there is a way to alert the user of interesting things that are happening even if the user is not currently inside the app. Mobile OS have provided a solution to this. Instead of the app continuously checking for events or doing work in the background, a server-side component for doing this can be written. When an event of interest occurs, the server-side component can send the app a push notification, which is intended for signaling the app that there are event pending on the server. Push Notifications are not intended for transmitting data to the client app, but are used as a signaling mechanism when the app is not running. Push Notifications makes no guarantee about delivery or the order of messages. Apps need to register with the Notification Service that is APNS (Apple Push Notifications Service) [10] for Apple and C2DN for Android [11]. This assigns a specific ID called device token that the app can send to the server. If the server needs to perform a push notification, it uses this device token to specify the app it want to send the notification to. Device tokens change so it is advisable to perform device token updates on the mobile preferably every time the app is opened. The Notification service offers an API that the server can poll periodically to determine what device tokens are still active. Because communication between the server and the notification Service is not trivial, 3rd party services are available such as Urban Airship [12].

3.2.6. Social Network Data Filtering

DataSift [3] is a platform that enables cost efficient and simple retrieval of data cross platform with built in augmentation for data. The main usage of this data at the moment is for marketing purposes, for example a marketer might be interested in how a specific new product is received by the market, the sentiment towards various functional aspects or some specific idea. DataSift bridges all this different data sources and delivers insights, and as this data is created and analyzed in real time, it is possible to respond with very low latency to the current public sentiment. Because some specific search queries might be too complex for search engines, this is the point where search technology can be improved by using crowd-sourced data across a multitude of systems. This data can be combined in order to respond to more complex queries. DataSift can be configured to act as the query engine for answers to question that come from various data sources.

3.2.7. Data Sources

Twitter was the first Data Source supported by DataSift and still is the most important. However once the infrastructure was created, DataSift has added more data sources that can be analyzed in real time.

- Facebook—license cost FREE—With more than 1.1 Billion active users, Facebook public data provides a unique view into what the world is watching, sharing and interested in.This data source provides access to the stream of public Facebook status updates, which include text-updates, links, photos and videos. Location names (e.g. restaurants) are added if the status update includes a location or check-in. This provides valuable insights to help market researchers. Author information including full name, profile photo and Facebook user-id is removed in accordance with Facebook's Platform Policies on Data Collection and Usage.
- Twitter—license cost $0.10 per 1000 interaction—The Twitter source delivers the text of Tweets plus more than 40 other pieces of data associated with Tweets. With more than a quarter of a billion Tweets every day, Twitter is an immensely significant source of data, delivering both breadth and depth of coverage. With this source you can receive data such as the author's name and screen name, their location, their preferred language, the number of people they follow on Twitter and how many people follow them. Other examples include filtering for Retweets, or for Tweets sent in reply to a particular author, or for particular keywords in users' bios.
- Wordpress—license cost $0.50 per 1000 interactions—WordPress powers 20 percent of blogs and sites on the web and has more than 1 million posts per day from blogs on wordpress.com, as well as content from sites hosted elsewhere that utilize WordPress as a content management platform if they enable the Jetpack plug-in. Comments and likes are also available from this data source.
- Blogs—license cost $4.00 per 1000 interactions—The Blog data source combines material from a wide variety of sites, ranging from well-known hosts such as Blogger with very large numbers of active users to small, single-user sites that run as blogs or incorporate a blog. You want to count mentions of a competitor's brand across as many social media sites as possible. Activate the Blog data source and use the interaction content target in your filter to capture posts from every available source. By spreading your "net" in this way you can add extra volume to your input data, to make your statistical analysis more significant.

3.3. System Use Cases

There are four types of actors that the system supports, each having specific capabilities. We will analyze each actor together with the associated use cases.

The client actor is the user that has the booking application installed on his mobile phone. He uses the application to order taxis by sending his location to the server that consists of the *Cloud*. This is the main functionality of the system, which we will be presenting later on in a more detailed fashion. We note that this use case is composed of two elements: getting the *GPS* position of the user, which is used by the system to identify the address of the user, and it also offers the possibility of fix tuning this location to specify it more precisely, as this location will be sent to the operator or the driver. The User use cases are presented in Table 3.

Table 3. Client Use Case specifications.

Category	Use case
Book vehicle	Access map view of nearby vehicles
Book vehicle	Specify exact location
Book vehicle	Get time and cost estimates
Book vehicle	Specify advance booking time
Book vehicle	See vehicle details
Track Vehicle	Track vehicle on map
Track Vehicle	Get notifications
Pay for the ride	In app payment
Rate vehicle	Rate vehicle
Share on Social Network	Share on social networks

The vehicle tracking functionality allows the user to see the taxi approaching on a map. In addition to this time estimation will be displayed together with a distance approximation. In case the response to an order comes from the operator, this functionality will not be available, as the taxi will not have GPS capabilities that allow tracking. The Share on Social networks use case allows the user to share the current ride with his friends on social media. This is an element that enhances the marketing potential for the app and a key component in attracting new users. There are two types of services which users can use with the app: Facebook and Twitter. One of the main benefits of using this app is the possibility of accumulating points, as a rewards program envisioned to spur app usage and attract new users. Each time the user will be making orders using the app, he will accumulate points which can be converted into free apps.

The Driver is responsible for responding to incoming orders. It does this by accepting or declining an order using a simple two buttons interface. The orders are assigned to drivers based on the proximity of the client to the driver. In addition to these functionalities, the Driver can also call the client in case he cannot see him at the specified pick up location or if he needs additional information. The driver cannot see the clients phone number, but only a call button. The rate client functionality allows the driver to rate the client based on his behavior. The rating is based on considerations such as: was the client present at the meeting point, did he leave a tip, was he loudly *etc*. The rating is based on a 5 star system subjective to the driver. Future orders placed by the client will show the overall rating that this client has.

The operator is responsible for responding to orders that were not answered by drivers directly. This allows companies to use the traditional mechanism for finding a taxi and send a response to the client. Using this mechanism, the client always gets a response to his order, even if no driver is available in his proximity that is active.

Responding to a client involves two things. First, an identifier for the taxi must be specified. This consists of the car number or some other way for the client to identify the taxi. Also the operator must specify the approximate location where the taxi is at the moment, by selecting it from a map. This creates a more visual context for the client and creates a sense of consistency between orders answered by the driver and the operator. Similar to the driver, the operator can call the client to ask him more details. Distinctly from the driver, the operator can see the users number. As we earlier specified, when a response to an order is set, the operator selects an address from the map which will display the initial taxi position to the client. It is possible for the operator to cancel an order from the client, in which case can write a cancelation message explaining the decision. The operator use cases are illustrated in Figure 1.

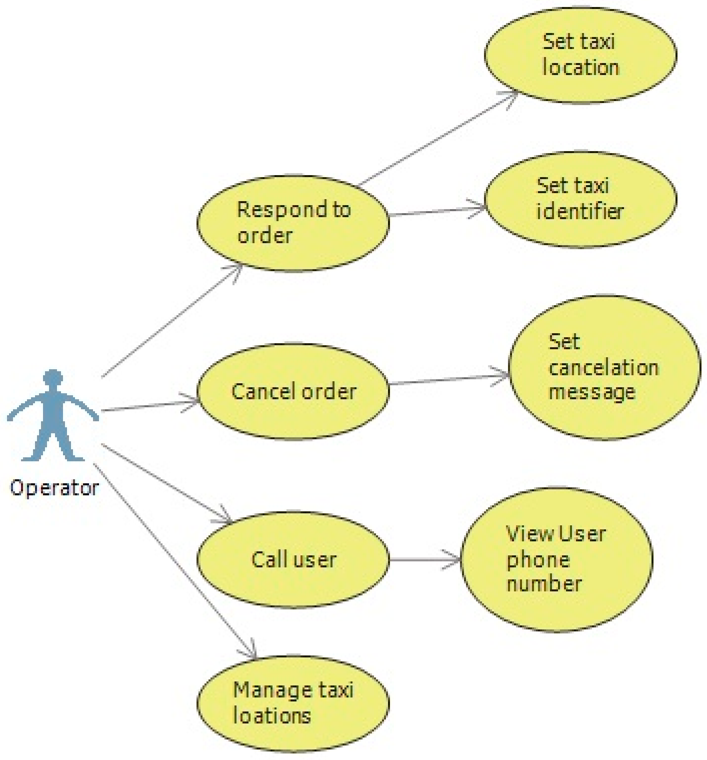

Figure 1. Operator use case.

The Manager actor is responsible with registering taxi companies into the system, setting the areas in which they operate, adding drivers and creating operator accounts. The manager is a role comparable with the administrator of the system. It has the power to change accounts, activate or deactivate them.

The Social CRM can provide valuable information regarding the user satisfaction level with the service and the potential direct improvement points. The person responsible for the Social CRM management will have access to a dashboard offering metrics such as overall customer satisfaction levels for specific time intervals, highly negative or positive comments in real time and automation and filtering tools. Also the Social CRM component will augment user data with statistics obtained using DataSift. The list of Social CRM use case is presented in Table 4.

Table 4. Social Customer Relationship Management (CRM) Use Case specifications.

Category	Use Case
User data augmentation	See user satisfaction metric
User data augmentation	Segment users by activity levels
User data augmentation	Identify promotion targets
Data filtering	Identify comments in real time
Data filtering	React to comments in real time
Customer satisfaction	Create promotion
Customer satisfaction	Assign to-do items

Social CRM—React to Comment Use Case Detailed

When a highly positive or negative comment regarding the service is published on the web either on a social media website such as Twitter or Facebook, or through a blog comment or even an article, DataSift will extract that interaction and will deliver it to our application in real time as presented in Figure 2. The complete process through which DataSift is capable of extracting this interactions and augmenting them in real time will be explained in the following section.

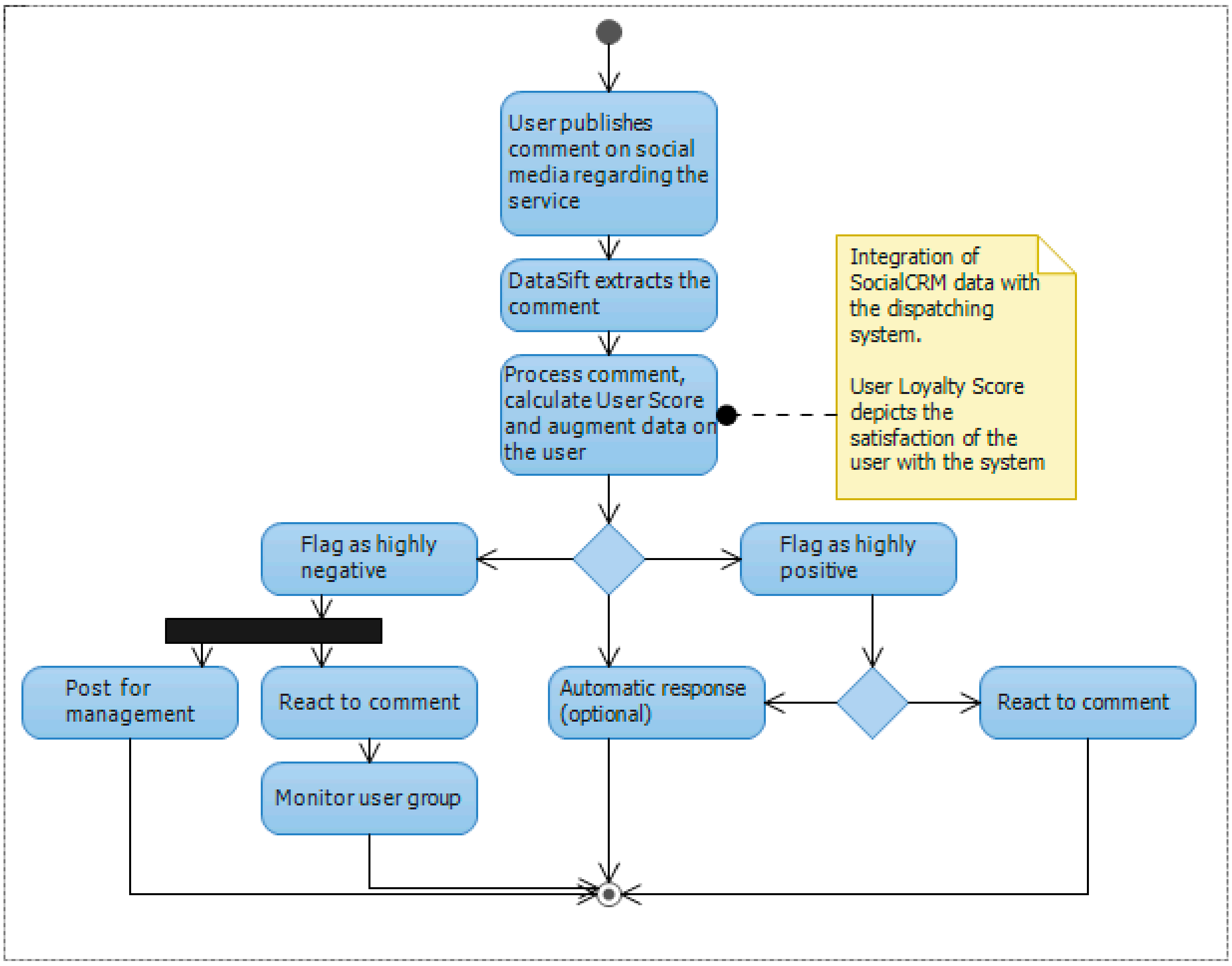
User publishes comment on social media regarding the service
DataSift extracts the comment
Process comment, calculate User Score and augment data on the user
Integration of SocialCRM data with the dispatching system.
User Loyalty Score depicts the satisfaction of the user with the system
Flag as highly negative
Flag as highly positive
Post for management
React to comment
Automatic response (optional)
React to comment
Monitor user group

Figure 2. Social CRM react to comment use case.

Once the interaction enters our system, it is already augmented with sentiment analysis, which means it can easily be catalogued by our application. The comments that have a sentiment rating between 15 and 20 (20 being the maximum) are tagged as highly positive. Comments having a rating between −20 and −15 (−20 being the minimum) are tagged as highly negative. Both of this categories are extremely important from a marketing perspective.

If a comment is highly positive, the marketer is interested in building on the positive experience of the user and will offer him bonus point or extra credit for inviting new people to the platform.

3.4. Conceptual Design

The proposed system is a complex solution which can be decomposed into multiple components. From a high level architectural point of view, the core of the product is represented by the *Cloud Engine*, which is stored on Windows Azure. The Cloud Engine uses a *relational SQL database* for persistence and interacts with third party services. The mobile applications for clients and drivers access the service through the *custom built API*. The *management panel* is also a separate component that is built on the Cloud service. The *social media analysis tool* cuts across both the cloud service and the operator management panel, providing insights and automating social media data augmentation. A high level representation of the system can be observed in Figure 3.

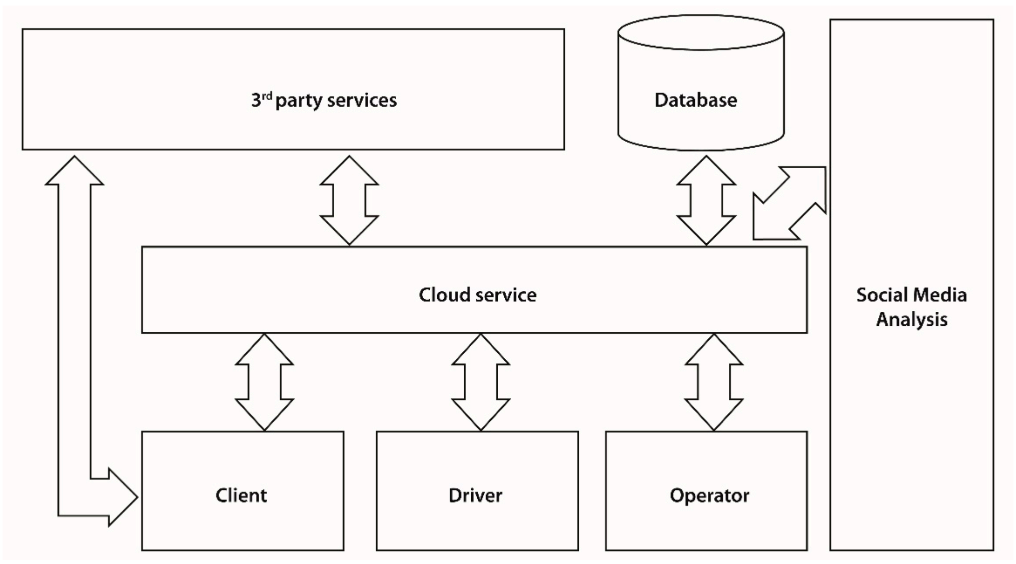

Figure 3. The block diagram of the system.

Client and Driver apps and the Operator web interface are the consumers of the *API* exposed by the Cloud service. The Cloud Service acts as the intermediary between all communication that takes place between these components and handles the data synchronization. The Cloud service relies on its functionality on using a database for persisting information such as orders, responses, users *etc*. The Cloud service also interacts with external services for performing specific tasks. Examples of this include sending *push notifications* or *reverse geocoding*. The client mobile applications can also access external services. For example the Client app accesses Facebook and analytics services.

The Cloud service acts as *a hub* and intermediating all communication between the clients. While for a client booking a vehicle seems as a P2P operation, it actually constitutes a client-server operation. As illustrated, numerous operators, divers and Clients can connect to the Cloud, new ones can be added, or some can be removed at any time. For example a taxi driver that terminates his work schedule for the day, can disconnect from the Cloud service and will not receive any new orders until he checks in again with the service.

All data is stored into a proprietary SQL database, and specific tasks that have been delegated to external services are managed by the Cloud service. When a client books a vehicle, the Cloud service checks through the fleet of vehicles that are available on that area at that moment and selects one for being dispatched to the client. In case no vehicle is received, the Cloud service sends the order to a specific vehicle company in order to be processed. Not all mobile applications are required to have a server component. However applications that require a higher degree of complexity need to create their own model and an API for exposing the services. *Cloud* computing is known to be a promising solution for mobile computing due to reasons including mobility, communication, and portability, reliability, security. In the following, we describe how the Cloud can be used to overcome obstacles in mobile computing, thereby pointing out advantages of *Mobile Cloud Computing (MCC)* [15] MCC refers to an infrastructure where both the data storage and the data processing happen outside of the mobile device. Mobile Cloud applications move the computing power and data storage away from mobile phones and into the Cloud, bringing applications and mobile computing to not just smartphone users but also a much broader range of mobile subscribers.

Figure 4 depicts a high level overview of the Social media analysis tool as implemented by DataSift. Facebook, Twitter, Blogs and other online data sources are monitored in real time and processed by the DataSift Engine.

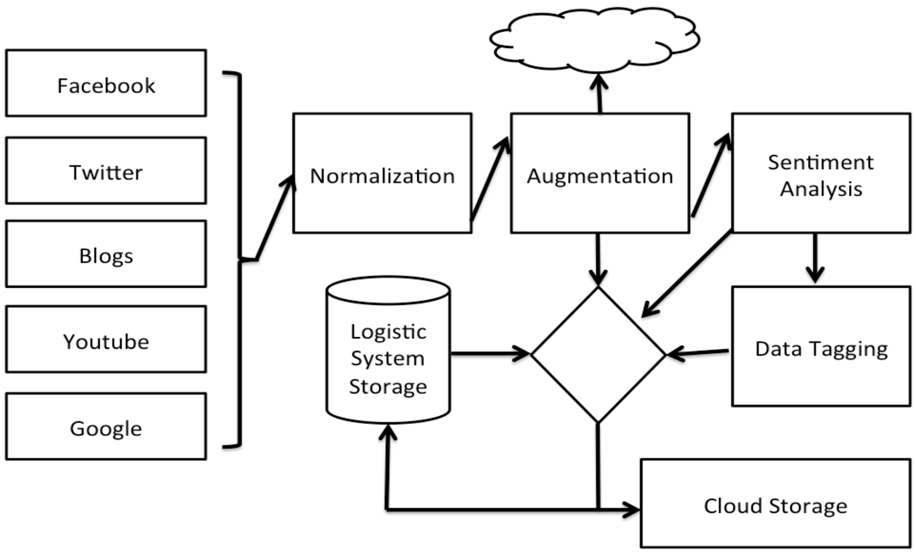

Figure 4. Social CRM components built on DataSift.

The process of augmentation of the user data in our system and the insights provided to the customer relationship responsible goes through a pipeline of steps. The normalization step converts all interactions into a similar format independent of their source, whether it is Facebook, Twitter or a blog post. Augmentation provides additional data on the interaction such as location context and fetching of the content of a link. The sentiment analysis tool provides a mean of grading the sentiment associated with a given post on a scale from −20 to +20. The sentiment analysis is provided by a Lexalytics algorithm that is offered as a black box solution. Data integration with the dispatching system is handled in the Logistic System Storage, where the data is stored into one common database.

We have analyzed the theoretical approaches and the frameworks required for implementing the proposed application. In the following chapter we are detailing how we implemented the vehicle booking application.

3.5. System Implementation

In this section we will present the system architecture and some implementation deign concerns for all major components. As we have stated in the previous sections, our system consists in a dispatching system which is vertically integrated with a Social CRM system. The Social CRM system is responsible for aggregating information about our users from online data sources and are used for improving the customer relationships or better targeting the users using advertising.

In Figure 5 we have represented the two systems in more detail. The Social CRM system is built on top of the DataSift platform. The dispatching system is responsible for controlling the DataSift platform using the API it exposes. Through the API we can use filters for sifting only the interactions we need, and then integrating the two systems using a common SQL Azure database.

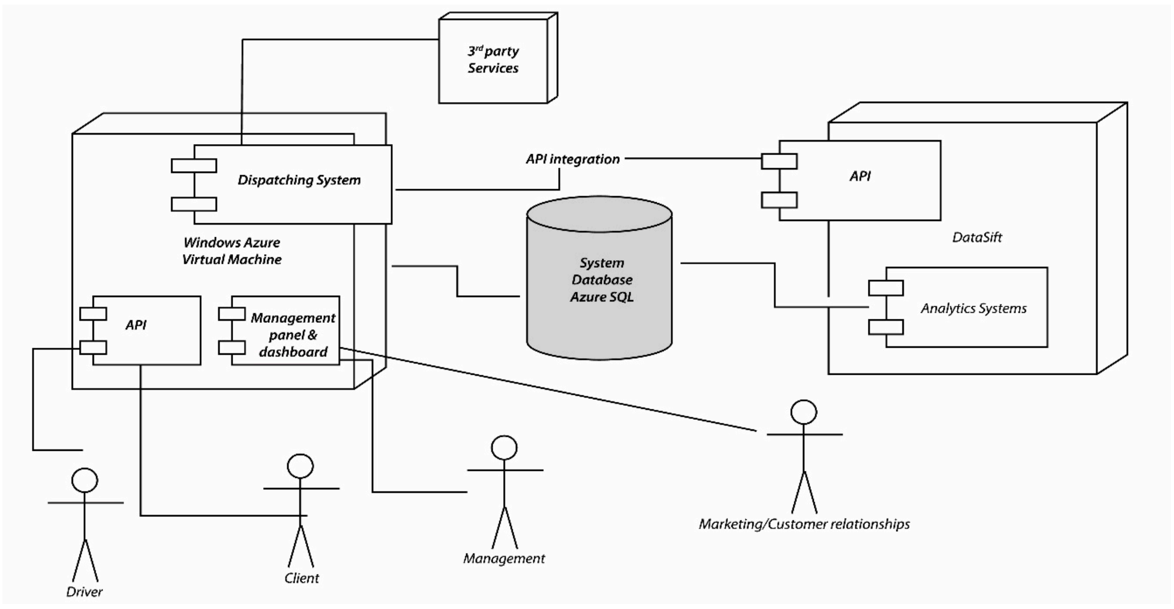
3rd party Services
API integration
Dispatching System
Windows Azure Virtual Machine
API
Management panel & dashboard
System Database Azure SQL
API
DataSift
Analytics Systems
Marketing/Customer relationships
Management
Driver
Client

Figure 5. System integration.

Next we will analyze both of the two systems and the integration between them. First we will address the dispatching system, and in the second half of this chapter we will analyze the DataSift platform in detail.

3.5.1. General Mobile Dispatching System Architecture and Implementation

The Dispatching System can operate as a standalone application. There can be identified two types of components. The first type includes the proprietary components. This need to be developed by the developers, are based on specific frameworks and include: the components constituting the Cloud service (designating the service built by the developers), the mobile application, the browser client and the management interface. The second type of components are represented by the 3rd party services (distinctly from proprietary services, 3rd party services or external services are built by other organizations and are made available to the public for use) are exposed trough certain API.

In this example we are using five types of external services: The Notification Service—used for sending Push Notifications (a way of alerting a mobile device), social services that offer integration with Facebook and Twitter, Analytics services for tracking usage patterns, SMS gateway service for sending the user an SMS with the activation code (required by the business rules), and a geocoding and reverse geocoding service for converting GPS position into street address. All of these components will be explained together with their implementation technologies and internal working

A request is serialized as *JSON*, signed using the user key and sent to the server through the server *API*. The server determines the nearest driver to the user, by interrogating the database based on the business logic. Once the nearest driver is determined, the *Signaling Framework* (used to alert the mobile application) is used to determine the selected driver's app to download or update the order sent by the client. After the driver confirms the order, the server uses the same approach to inform the user that his order was responded. All requests sent to the proprietary server components need to use SSL for proving security to the system. The dispatching application exposes *JSON REST* services using *WCF* Framework. Working with *JSON* in the context of *WCF* is easy as the framework automatically serializes *JSON* request and responses and performs data binding between *JavaScript* and *CLR* objects. Once the data is received on the server, the server performs some security check on the request. This usually implies authentication and authorization of the request. As there is no password required for the Taxi Ordering application, signing the data using a key, which is known only by the client and the server, and is unique for each client mobile application, does this.

Figure 6 depicts the integration between the client mobile application and the server. On The client we have the *MVC* pattern. The view sends an event to the controller, which calls the send order operation on the *Model*. The model component responsible for sending the request (implemented in Titanium using an *HTTP Socket*), serializes the data and sends them to the specified service endpoint exposed by the API. Once the request arrives on the server, the reverse process occurs as the JS data objects are translated into *CLR* objects by the *WCF* framework. The security component is responsible for checking the signature of the user in the case of Order Taxi applications. In other applications a username and password approach can be used for obtaining the same result. It is essential to observe that once the request passes the security verification, it is recorded and the registration key for the order is returned to the client. Further on the client uses this registration key for referring to his order. At this point there is no response available for the client yet.

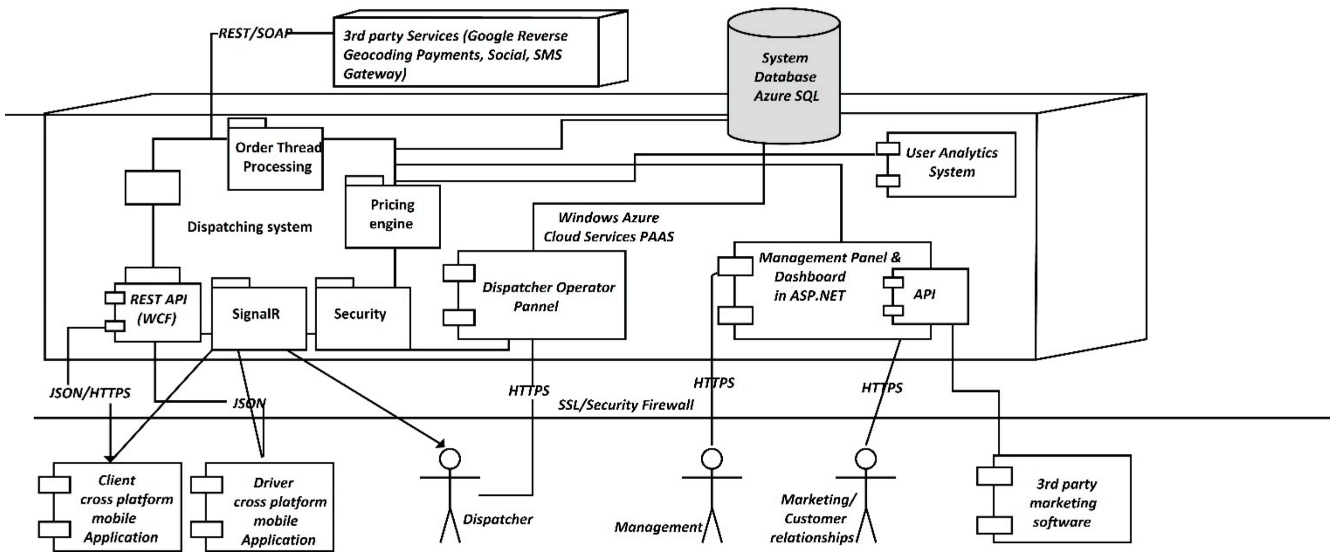
REST/SOAP
3rd party Services (Google Reverse Geocoding Payments, Social, SMS Gateway)
System Database Azure SQL
Order Thread Processing
User Analytics System
Pricing engine
Dispatching system
Windows Azure Cloud Services PAAS
Management Panel & Dashboard in ASP.NET
API
REST API (WCF)
SignalR
Security
Dispatcher Operator Pannel
JSON/HTTPS
JSON
HTTPS
SSL/Security Firewall
HTTPS
HTTPS
Client cross platform mobile Application
Driver cross platform mobile Application
Dispatcher
Management
Marketing/ Customer relationships
3rd party marketing software

Figure 6. General Dispatching System Architecture.

The order placed by the user in the system is assigned to the nearest driver. When the driver confirms the order, a response is sent to the client mobile application. This response can come after a variable amount of time which can be anywhere between a few seconds and 3 min. In consequence *SignalR* is used to inform the client when the response is ready. SignalR is not used to pass the response directly to the user, but rather the user applications download the response once it receives the update signal.

An important characteristic of the service is that it should offer scalability. All the components depicted can be hosted in the *Cloud*. This allows an elastic hosting environment, which can be used easily. For Order Taxi application, we are using *Windows Azure*. Using *Windows Azure* we can easily deploy the server component as it was built completely on the *.NET* framework.

Location determination. Titanium offers the possibility of interacting with device capabilities such as using the GPS and allows for specific parameter to be set on the GPS such as the precision of positioning, as a balance between time and required accuracy. When the app starts the coordinates are the ones cached on the device, in other words the ones detected at the last time you used the geolocation on your device. Detecting the current position takes time so if you have an app based on geolocation you might want to take this in account to improve the user experience and avoid to get false results. Titanium.Geolocation [6] provides the necessary methods to manage geolocation.

Implementation of the Model. The business logic of the system is implemented in the model. The service model is composed of a set of classes representing the implementation of the domain model. Figure 7 represents the most important objects that constitute the domain model. Because Entity Framework was used for mapping CLR objects to database records, these entities have a corresponding implementation in the SQL server database. We are outlining the most important entities presented in our system:

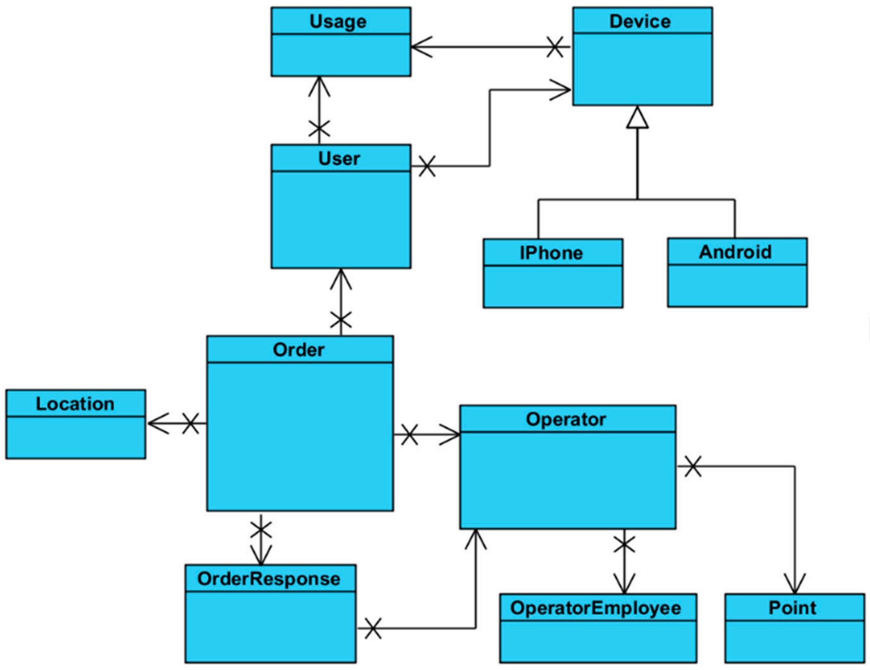

Figure 7. The entities of the system model.

Order: represents an order placed by the client into the system. The Order has a set of attributes representing:

State of the order: *IsResolved*, *IsCanceled*, *IsPendingResponse*. This attributes are used for identifying the current status of the order.

Modification time: Each modification of the state of the order also sets a time stamp on the object, which is used by the system. For example when an order is created, it is displayed to the operator together with a label showing how much time has passed since the order was submitted.

Data associated with an order: Represent information such as the rating of the taxi driver or the feedback of the client following the ride.

Associated data such as the pick-up location or the owner is represented as references to the specific entities.

User: represents the client who is placing orders into the system. The user has the following attributes associated with him:

Name represents the name chosen by the person submitting orders.

Registration date the date when this user registered by submitting his phone number and name

Phone number is used for contacting the person in case he does not show up at the pick-up location

Phone is validated is used to determine whether the specific user was validated by the validation code sent through the *SMS*.

The user must be marked as active in order to be able to place orders into the system. A user state can be changed to inactive in case abuse has been observed for this particular user.

The confirmation code is the activation code sent to the user through *SMS* that is used to determine if the phone number is valid.

Location: Represents the position where an order is set. It contains the *GPS* coordinates for the location together with an optional name for it. There is also a property used for counting how often that location is used. The more frequent locations will be displayed first in the location suggestion list.

Operator: This entity represents an operator, which might me understood as a taxi company or a collection of taxi companies unified as a group. This entity presents the following properties:

Activation date and time, when this operator has been registered

A perimeter that represents where the operator is active, represented by a collection of points.

Stations represent predefined locations on the map where taxis are expected to be present.

OperatorEmployee: This class represents the operator that responds to orders.

The name is the real name of the operator.

IsActive property is set to true for those *Operators* which are active, meaning having permission to access the system, permission set the manager.

OrderResponse: created by the *OperatorEmployee* and represents the response the user receives for his order. It contains the necessary information required by the client such as the car number, the name of the driver, and the expected waiting time for the user. It also has an *IsRead* property which is used for determining whether the response has been read by the user application. If the Response is not read in a certain amount of time, this means that that the client application is closed and as a consequence the response needs to be delivered using push notifications.

Point: This class is used for defining perimeters in which an operator activates. Ay order placed within this perimeter might come to the operator. The number represents the count for the point, so that given a collection of points and their orders, the system can check whether a specific point is inside that perimeter or outside of it.

The entities are mapped by Entity Framework into the CLR classes which are further extended in functionality by means of partial classes, by adding extra functionality.

Service Database. The service relies on a SQL rational database. The service database is generated using *Entity Framework*, and we designed the *Entity Framework* entities directly mapped to the tables, and providing the same properties as columns in the rational database.

UserSet table represents the client who is placing orders into the system. The most important columns of this table include the name which represents the name chosen by the person submitting orders, the registration date – the date when this user registered by submitting his phone number and name. The phone number is used for contacting the person in case he does not show up at the pick-up location, and is represented as an nVarChar.

We used this representation especially because phone number formats are very different for distinct regions. *IsPhoneValidated* column is used to determine whether the specific user was validated by the validation code sent through SMS. The user must be marked as active in order to be able to place orders into the system, this column being represented as a non-nullable Boolean type. A user state can be changed to inactive in case abuse has been observed for this particular user. The confirmation code is the activation code sent to the user through SMS that is used to determine if the phone number is valid. This column is also represented as an nVarChar.

The table *OrderSet* has the foreign key represented by *User_id* that specifies the User which placed the order inside the system. *OrderSet* and *OrderResponseSet* have a one to one relationship between them. We decided to separate these two tables, even though there is a synonymy relationship between them as there can be order, which do not have a response, such as for example an order that was canceled.

We can observe that there was no case for splitting entities in our model. In this mapping scenario, properties from a single entity in the conceptual model are mapped to columns in two or more underlying tables. In this scenario, the tables must share a common primary key. In our mapping scenario each entity in the conceptual model is mapped to a single table in the storage model. This is the default mapping generated by Entity Data Model tools. Each operator can define multiple stations which represent positions on the map which an associated name (records in the *LocationsSet* Table) that are by the operator to specify the initial taxi position easily, by clicking the marker on the map associated to that specific location. The mapping of a list of stations (defined as actually being of type location) to the *OperatorSet* is done by *EntityFramework* by introducing a new table called *OperatorLocation*, as can be observed in Figure 8.

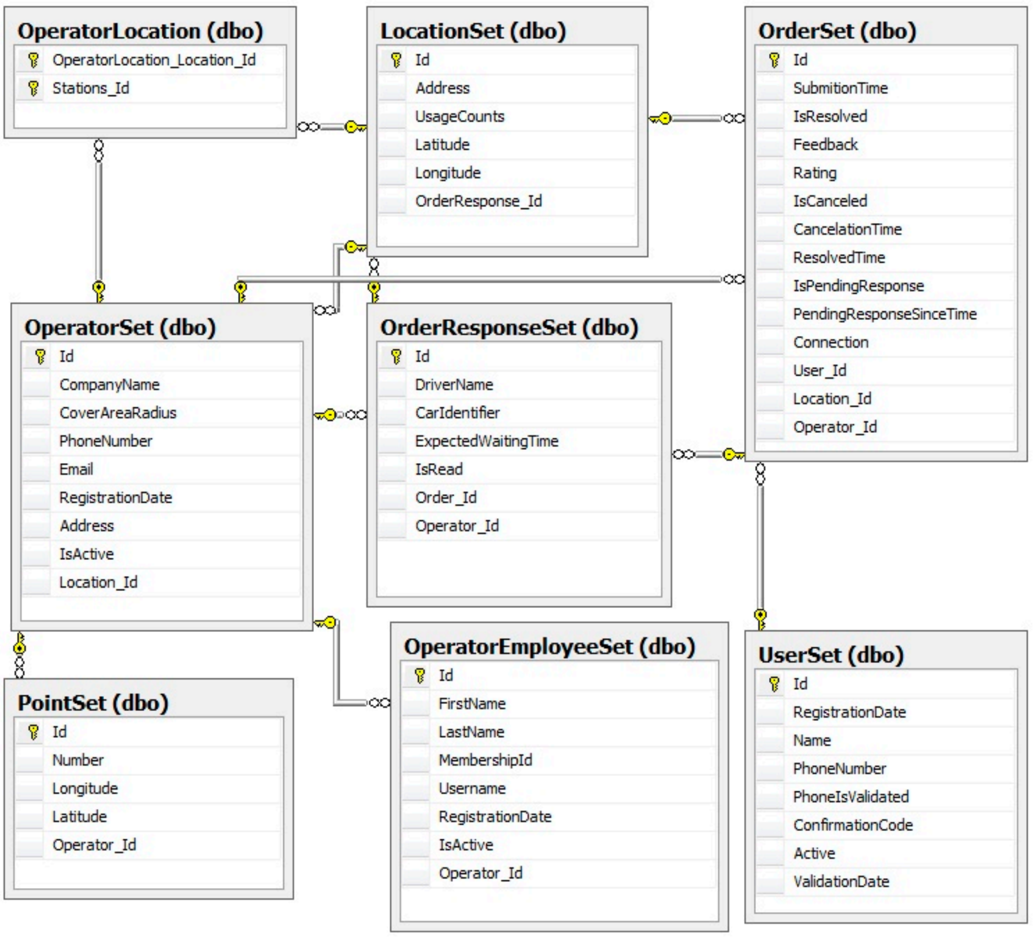

Figure 8. The MS-SQL database.

We are using this database model representing the conceptual model of the system inside *SQL Server 2010*. We have decided on implementing our database using Entity Framework generate Database from Model approach, as it offered a productive environment where changes in the model could be easily translated into updates of the database. We also notice that there is a balance between the numbers of read and write operations as every record inserted into the database needs to be read usually just one time, processed and then archived. For example an Order is created, added to the system, it is answered by the operator, and the response is sent back to the client and after this there response is no longer used actively and constitutes solely a history used for creating suggestions such as, for example, suggesting pick up locations.

System API specification. The interaction between the client mobile and the server is performed through the means of a *REST* API and we implemented the *REST* API trough WCF. For example, IApi is the interface which is implemented by the API class. IApi exposes multiple methods including: UserOrderNew, UserOrderCancel, UserOrderResponse and UserNew.

UserOrderNew—this method is exposed as a resource using the verb POST and is available by invoking the service at the location /Api/user/order/new. The data passed to the method are encoded as JSON and consist of the User name and phone number. Once this data are received on the server the system sends the user an SMS message at the specified phone number. At this moment the user status is set as invalidated. In order to validate the phone number the user needs to send the confirmation code sent to him through SMS by using the next method described, ValidatePhone.

UserOrderCancel—this method is used to validate the users phone number by sending using the POST verb the code sent to him by SMS. This rest resource is available at /Api/user/order/cancel. The data needs to be submitted in the JSON format, as with all API calls to our service. When the code reaches the server, it is checked whether the code matches the one in the system. If the code is correct, the user is allowed to submit the order.

UserOrderResponse—this method allows for creating a new order inside the system. This is performed by sending the information required the order to the address /Api/user/order/response using the POST verb. Once an order is received on the server, it is recorded inside the database; it gets displayed on a specific operator panel. Once the operator responds to the order, an OrderResponse entry is created and an update signal is sent to the client. Among the attributes in a create order request the most important are:

- The location where the order was sent for. This consists of the GPS position
- The user who submitted the request
- The date and time the order was created
- The device information

GetOrderResponse—allows the client to read the response to his order. It is accessible using the GET verb at the address /Api/order/response. The request contains the identity of the order. The response is encoded as JSON and contains the information for the client such as:

- The car number which was assigned for the order
- The information regarding the driver
- The estimated arrival time of the driver

UserOrderCancel—allows an order to be canceled by the user. The cancelation is performed by the client, and is accessible at /Api/user/order/cancel. The request verb is POST, as the submitted information contains the identity of the order that should be canceled. We note that we decided on not implementing the DELETE verb combined with the call at /Api/order as some Api best practices recommend as in our case there is no associated delete operation on the server. When an order is canceled what happens is that the IsCanceled property is set for the order, and an update notice is sent to the subscribers, such as the operator.

Security and Membership Framework. The API is secured using ASP.NET Membership Framework that handles the authentication and authorization logic of the application. The benefit of the Membership Framework is that a robust and throughout tested framework offers much better security than self-implemented framework. As a consequence, the Security framework was not extended and was allowed to operate as it is. By not extending or close integrating any services of the Membership framework with our system, we decoupled the two subsystems and allowed for further changes.

Also, all API calls are authenticated. The authentication is done using the user id, password and the session id. Each session is started when a user logs in. This prevents users to run the application on multiple devices in parallel and improves the overall security of the system. For the driver application authentication is done using tokens that are valid for a specific time interval. When a token expires, it needs to be renewed. Using tokens we are able to reduce the resource utilization on the server. If a call fails to authenticate, an error code is retuned. All communication is forced to use SSL, which eliminates or reduces the chance of Man in the middle attacks.

SignalR. It is not feasible to implement data synchronization using a polling interval, as the polling interval is both unknown and will generate large amount of data and CPU usage. In the .NET framework SignalR represents an implementation of long polling. However SignalR does not rely only on the long polling technique. SignalR has a concept of transports, each transport decides how data is sent/received and how it connects and disconnects. SignalR uses two concepts on the server implementation: *Hubs* and *persistent connections*. The Client app uses *Hubs*. Extending the *Hub* class represents the implementation of a hub on the server. Each method defined inside this class is mapped to a JavaScript implementation at compile time. The JavaScript document is saved on the project root folder and the client must reference it. It is possible to have broadcast events, which alert all the clients or only one in particular. It is also possible to group client and inform a whole group at a time. When the mobile application registers using SignalR, on the server a session ID called client ID is given to it. We are storing this client ID for each order. When a response is available and needs to be sent from the server to the client, we are sending an update request to the mobile app, using the client ID stored earlier.

There are three components of the system that need to be updated from the server. This are the client app, the driver app and the operator panel. The update scenario for the driver app is similar to the one for the client app. We note that the client app and the driver app are components that need to be independently updated, so having groups in this scenario is unnecessarily. However for the operator panel we need multiple operators to be able to respond to an incoming order. In this scenario it is more feasible to group the operators in groups, a group representing one operating company. When a new order is registered into the system the first operator who sees the new order will be able to respond to it.

3.5.2. Management Panel for human aided dispatch

Some orders need to be inserted by the dispatcher, or might come from an external API that integrates with other SCM or ERP systems. Management Panel represents a web interface intended for responding to orders through a human actor acting as a dispatcher. The webpage is developed using *ASP.NET* and is secured using *.NET Membership Framework Forms Authentication*. A logged in user will be displayed a panel consisting of the current orders. Color codes are used for different order statuses. Brown is used for canceled orders, Green for orders that were responded successfully. No color is used for orders which have not been responded yet. The webpage uses AJAX techniques for receiving new orders and sending them without requiring complete page reloads. When a new order is inserted into the system, *SignalR* is used to notify the browser webpage to reload its orders panel. Partial page reloads are designed using the *ASP.NET UpdatePanel* and the *AJAX* functionality is built on JQuery. Using the Operator panel website the operator is able to respond to client orders. Whenever a new order that is destined for that operator is inserted into the system, SignalR that the orders grid needs to be updated notifies the webpage. Using an update panel control, as we specified earlier, does the update.

We implemented the operator panel in such a way to allow multiple operators to work on the same set of orders without worrying of responding to the same order. Whenever an operator views the content of an order, the order is marked as pending. When this happens a message is sent to the user notifying him that the operator viewed his order. Also another operator is announced that his colleague is already processing the specific order. If he tries to process the same order he will get an alert.

Feature usage metrics using Analytics. Analytics provide the means of measuring user experience inside the app and becomes a key component of the mobile ecosystem. Analytics provide the means of observing how users are interacting with the app and this provides the information required for further updates. Traditionally two analytics were used: *daily active users* and *user session time*. However it has been observed that user engagement falls sharply few days after the app first installation. As a consequence another key metric has become the *user retention level* after one day, one week and one month. Unless the app uses analytics it cannot understand its users base accordingly.

Analytics provide information regarding usage statistics such as new users, active users (which combines new and returning users into one stat), and sessions (how many times your app is used over a time period). You can grow this stat several ways such as acquiring new users, getting existing users to use more often or retaining users longer), session length (Session Length is a key engagement metric. Although different kinds of apps have different average session lengths, you would like this stat to improve over time with each additional update), frequency of use (this stat reveals the intensity with which your app is used), retention level, and app versions. It is also possible to track custom events such as the user clicking a menu button or completing a game level. We are tracking the address textbox editing feature, the map marker move, the amount the map marker is moved, the amount of time spent editing the address textbox, the amount of time until the order is placed.

Social media integration. Using Social Networks such as Facebook and Twitter we provide a mean of authentication using *OAuth v2*. For example if the app requires the users email address it can prompt the user to authenticate in the app using his *Facebook* account. If the user agrees to allow *Facebook* to share the email address with the app, the app is authorized to retrieve the users email from the Facebook service using the *Facebook* API. *OAuth* is an open standard for authorization, it allows users to share their private resources stored on one site with another site without having to hand out their credentials, typically supplying username and password tokens instead.

The other common use of *social services* integration is for marketing purposes. Many apps allow users to post messages to *Facebook* or to post on the users behalf. The posted messages should be based on users intention to share them, but they also often serve as a marketing strategy inside the users social network. *Order Taxi* application uses *social services* for marketing purposes by sharing the users ride information with the users consent. In order to use social services, as with most third party *APIs* an *API* key must be obtained by registering on the providers website. For *Facebook* this is available in the developers section of their website. Titanium offers modules for integrating *Facebook* and *Twitter* services and encapsulating the communication with the REST *API*. In our Taxi Ordering Application, we are integrating Facebook in order to allow people to post messages on their wall to inform their friend they are using Order Taxi application. They can also invite friends to join the application and earn points.

Management and configuration .There are three types of components that need to be managed: The proprietary server, the mobile client and the third party services. Managing the third party services is usually done by a management panel on the providers' website. A management panel for changes that have a high occurrence rate manages the proprietary server. For example new drivers are added manually to the system from this management panel. The hardest component to be managed is the mobile client application as there is no direct control over it. If major changes to the system are required for implementing new functionalities and this requires a change in the API it is important that API versioning is used to secure reliability for old clients that do not install the last updates. Backward compatibility of the system must always be respected. If the changes made to the system do not require any change in the API, but rather a change in the address, this can be resolved if the client was designed to adapt to such changes. Otherwise an update of the client app is required.

3.5.3. Social CRM System Using DataSift

DataSift [3] supports a variety of datasources. However the idea and first implementation addressed just one datasource, which is the most active of them, all that being the Twiter Firehose. Twitter does allow anyone to access its streaming API. However only a subset of all tweets are available through the streaming API, which are less than 1%. Twitter does however have two affiliated companies which get all the tweets through the Twitter Firehose, those being DataSift and Gnip (which is now a subsidiary of Twitter).

In this section we will describe how the DataSift architecture is implemented and build for a massively scalable requirements. DataSift implemented both their software and hardware solution because the large amount of data required to process offered both technical and economic challenges. DataSift runs on Hadoop and has one of the largest Hadoop clusters in the world. By creating their own hardware cluster they were able to take advantage of more low level performance optimizations than if they had used a public cloud provider such as Amazon, Google or Microsoft. Also considering the massive explosion in Big Data over the past few years, their infrastructure is expanding massively which would have been much more costly if they decided to use a public clod solution.

The DataSift architecture can be broken up in three large areas as follows that being *data extraction, filtering and delivering*.

(1) Data Extraction and Preprocessing

We call data preprocessing the act of extracting or receiving data from the defined datasources. Given the multitude of data sources one important aspect of this step is normalizing the data. The Redis Queue is used as a buffer during high peak usage. This allows the DataSift architecture to remain reliable at all times. Data is extracted out and normalized. Twitter data is highly dimensional, it has 30 plus attributes and you get access to them all. These attributes include geolocation, name, profile data, the tweet itself, timestamp, number of followers, number of retweets, verified user status, client type, *etc*.

The Interaction Generation thus acts similar to a JOIN operation over multiple streams of proprietary data structures and formats. As one can see the data sources are received using both Push and Pull HTTP requests and the formats are ranging from JSON and XML to Binary Streams. The result of the Interaction Generation block is a normalized object. This object is further augmented with data extracted from third party sources or by analyzing the data using proprietary or licensed algorithms such as Sentiment analysis or Language Detection. Entity extraction and natural language processing is applied to the tweet. Language is determined, for example, and that result is made available in the meta-data. Each link is resolved and the contents fetched so filters can be applied to content and the links themselves. Data is fully elaborated before the filtering process occurs.

Bringing the datasets into their system they needed to form close partnerships. Their sentiment analysis engine was licensed, made fast, made clusterable, and made suitable to handle 500 million hits a day with low latency. Each service must have < 100 ms response time. After 500 ms it is thrown away. In order to respect this they needed to cache the augmentation data sources locally. Also one in 10 tweets is a link, so the system creates a request for fetching the content of the referenced page in order to allow filters against it. So if a brand is mentioned it can be resolved against the actual content to figure out what it all means. This can provide quick insights into new blog posts or product reviews mentioned on social media.

Metadata Generation.

- Tagging allow adding custom metadata to interactions. For example, a specific car model can be tagged as luxury or economy. This information can be very useful later on into post processing. Adding tags based on conditional rules gives you the power to code business rules directly into your CSDL code. Normalized data is augmented in order to provide more context into the application. By enrichment the interactions will gather more metadata that can be queried using CSDL. The supported Enrichment engines are Demographics, Sentiment, Gender, Entities, Topics and Trends.
- Sentiment scoring allows a computer to consistently rate the positive or negative assertions that are associated with a document or entity. The scoring of sentiment (sometimes referred to as tone) from a document is a problem that was originally raised in the context of marketing and business intelligence, where being able to measure the public's reaction to a new marketing campaign (or, indeed, a corporate scandal) can have a measurable financial impact on your business.
- The Salience Sentiment data source delivers a measure of the positive or negative sentiment in a post. Within DataSift, sentiment is typically scored from −20 to 20 although values outside this range do sometimes occur. A score of zero is neutral, scores below zero indicate negative sentiment, and scores above zero indicate positive sentiment. For example, a score of −1 is mildly negative while a score of 15 is strongly positive.
- The Salience Topics data source lists the topics it finds in a post. Salience Topics are derived from an analysis of all the content in Wikipedia. As such, they allow for robust detection of content that is "about" a particular topic. A major advantage of this approach is that the topics analysis engine can easily be extended. The engine independently analyzes the content and the title.
- The Salience Entities data source lists the entities it finds in a post. Entities are typically proper nouns (people, companies, products, places) but can also include special patterns like URLs, phone numbers, addresses, currency amounts, and quotes. It can also recognize entities that we have never seen before. Each entity has sentiment associated with it so, if there is a Tweet that says "I really like Brand X, but don't like Brand Y", the entity discovery feature will correctly associate the sentiment with the individual entities. Sentiment analysis is also able to understand emoticons and nuances in the language. Only English is available however.
- Classifiers take a large sample of interactions from the platform, manually classify the interactions and use machine learning to learn key signals, which dictate which category interactions should belong to. The result is a set of scoring rules that form the classifier. The resulting classifier can be run against live or historic data ongoing. For example a classifier can be used to asses a user's probability of preaching a specific product or to route specific interactions to specific customer support or marketing teams inside a company.

(2) Filtering the Interactions

The cost of licensing the Twitter Firehose is extremely high and would become an inefficient investment for the majority of companies. Datasift created a declarative language called CSDL is order to filter out the Interaction (such as Tweets, Facebook posts, blog entries *etc*.) you do not want and keep the Interactions you do want. As we have explained in the previous sections the cost of running a queue includes both a licensing fee and a processing fee. The licensing fee is based on the number of interactions that match the defined query.

CDSL language CDSL allows for defining filtering conditions that specify what information the stream will include and augmenting the objects in the stream with data from third-party sites. A filtering condition has three elements: target, operator and argument. The target specifies the operand such as the content of a Twitter post. The operator defines the comparison Datasift should make. For example, it might search for Tweets from authors who have at least 100 followers. More complex filters can be built by combining simple filters using the logical operators: AND, OR, and NOT. Once a result stream has been defined through a filter, it can be reused as part of the definition of another stream by using the stream keyword to combine it with further filtering conditions. The argument can be any value, as long as the type of the argument matches the type of the target.

All interactions matching a filter form a stream. Each stream is assigned an identifier such as "4e8e6772337d0b993391ee6417171b79. Stream identifiers are included in rules to further qualify which tweets should be in a stream. Rules build on filters and other rules: rule "4e8e6772337d0b993391ee6417171b79" and language.tag = "en" This allows filters to grow on one another. DataSift also pushes for public streams which are more cost effective to process because of resource sharing the computation of a specific rule.

In order to create scalable filters DataSift uses a compiler to schedule filters efficiently across a cluster. Rules are made from a combination of filters and can have 1000 s or millions of terms. Millions of rules can be evaluated simultaneously, scheduled across hundreds of machines. Rules can be reused and hierarchically order. Filters include regular expressions, you can filter out tweets based on text in the profile, for example. There are many different targets and operators. The core filter engine is in C++ and is called the Pickle Matrix and is depicted in the Figure 9 below. The filtering engine is implemented using a custom built compiler and virtual machine based on Distributed Complex Event Processing with Query Rewriting.

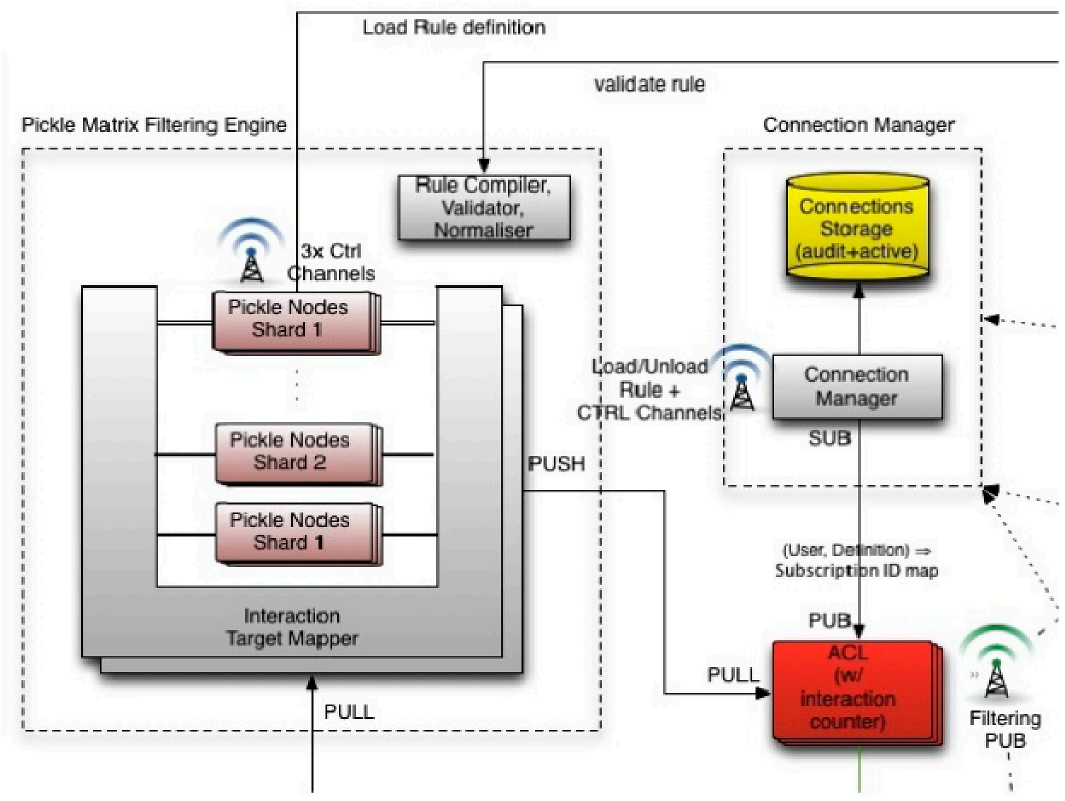
Load Rule definition
validate rule
Pickle Matrix Filtering Engine
Connection Manager
Rule Compiler, Validator, Normaliser
3x Ctrl Channels
Pickle Nodes Shard 1
Pickle Nodes Shard 2
Pickle Nodes Shard 1
Interaction Target Mapper
Connections Storage (audit+active)
Load/Unload Rule + CTRL Channels
Connection Manager
SUB
PUSH
(User, Definition) ⇒ Subscription ID map
PUB
PULL
ACL (w/ interaction counter)
Filtering PUB
PULL

Figure 9. Real time filtering of normalized posts based on precompiled rules [3].

A compiler takes filters and targets a Hadoop cluster with a Manager and Node servers. The Manager's job is to determine that if a rule is loaded anywhere else and if can be put somewhere that is highly optimized, like close to someone else that is running a similar stream. Node's job is filter tweets by rules, get a list of matches, and push the matches down the pipeline.

Custom algorithms are built so rules are shared as much as possible. No filter is run in isolation. The filters form a shared space. If there is a filter that filters only for "computer" references then that filter is run on each tweet once and the result is shared between all rules using the same filter. Instead of continually running duplicate filters they are only run once. Rules can be a hierarchical and the compiler is merges them together so that they share rule evaluations. If a node already has a rule then it will attract filters that use that rule. That rule is run just once for all the filters on the node.

By combining workloads of multiple customers together they can exploit any commonality in the filters. DataSift is paid for every operation even though they only run it once. If a regular expression is shared, for example, it is only run once. It cannot always work this way, perhaps a node is loaded so cannot take another client, so it must run on another machine.

The entire configuration dynamically changeable at runtime. The scheduling algorithm is constantly looking at monitoring data. If latency crosses a threshold then the system will be reconfigured. Filters are processed immediately in memory. Each server, for example, can run 10,000 streams. Nodes have a rule cache so they can be shared.

Compiler supports short circuiting to optimize filters. If a regular expression is bigger than an entire machine, for example, then they will automatically load balance across nodes.

The filtering engine is sharded. Each shard receives the full firehose and every "interaction" (*i.e.* a "tweet" or "FB status" or "blog post") is processed by all shards, the results of which are collated before being sent downstream. Each of the N nodes in a single shard (which are replicas of each other) receives 1/N of the hose on a round robin basis. So, supposing there are two shards, with three nodes each, 50% of the filters would be found on each shard. Each node in a shard would receive 1/3 of the hose (and would have 50% of the filters on it). And it would be a replica of the other nodes in that shard. Thus, if a very heavy filter is loaded, that will be balanced by adding more nodes to the shard the heavy rule is loaded into, thus reducing the amount of data that a single node has to process.

A big push is for customers to create reusable public rules to encourage the reusability of data streams. People can use any of the public streams in their own rules. For example, someone could make a word filter that creates a stream. You can build off that stream by using it. Everyone shares that same stream so if a 1000 users are using the dirty word filter the filter is not getting evaluated 1000 times. The filter is getting evaluated once for everyone, which is highly efficient. The platform will thus offer even better performance for a higher number of workloads and become highly scalable with resource usage relative to processing requirement being reduced.

(3) Loading and Delivering the Results

Once the filtering process identifies an interaction that matches the specified query, the interaction is delivered to the specific stream. It is important to mention that DataSift buffers the data it receives for up to one hour. During this time it is possible to request specific interactions from DataSift. The data is stored in Hadoop HBase. However most implementations will require specific custom processing on the results created by DataSift. For example a client might be interested in extracting information such as addresses or mentions of specific users or places and store them later in their own database.

When creating a DataSift Stream running task it is possible to specify a specific output location. Tweets that match the filter are accessible to external applications via a REST API, HTTP Streaming, or Web sockets with a client library exists for: PHP, Ruby, C#, Java, Node.js. The results are JSON objects containing all the data in a normalized, fully augmented format. Augmentations include gender, geolocation, point-of-interest, country, author, the tweet, follower count, Klout score, *etc.* in the stream. There up to 80 fields for each message. Tweets are not guaranteed to arrive in any order and you are not guaranteed to see all the tweets ever sent. This is because Hadoop MapReduce jobs are used to extract the data out of HBase, which involves sharding and thus change the order. However Hadoop does offer solutions for persisting the order of the analyzed data.

Recording interactions on DataSift

DataSift records the result of the Filtering Task in HBase tables. When data is streamed out of DataSift, Hadoop MapReduce jobs will extract the data in the specific desired format. Most of the time this format will be JSON. DataSift took the decision of storing the normalized data in HBase table instead of pure JSON because it allows further querying if necessarily or exporting it in other formats such as CSV or Excel documents as depicted in the Figure 10. This storage component is fundamental to DataSift for two reasons. First it is used as an Achiever, where all the Interactions that came through the system are stored in a normalized form and can be later processed using the Historic Task Services. The recorder is used for buffering the data until it is transferred to the client in the specified way (live stream, push to a specific data source, download using HTTP *etc*.) The recording table uses entries similar to <interaction ID><recording ID> in order to map the interactions that should be exported to a specific stream.

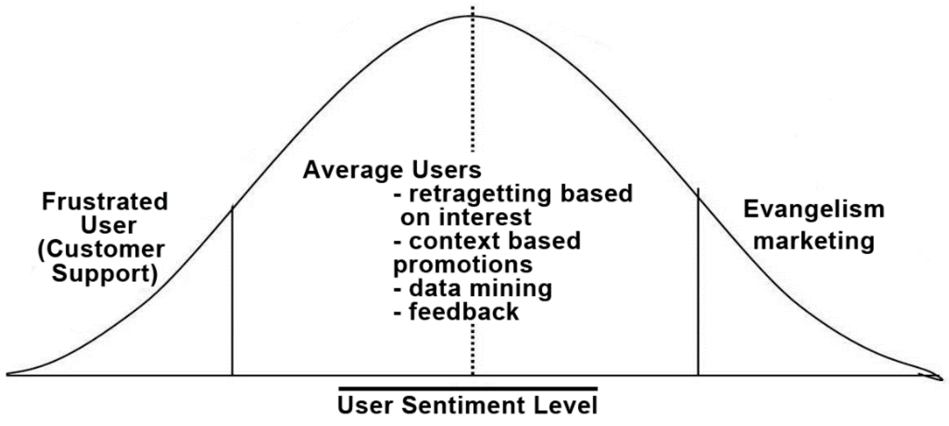
Frustrated User (Customer Support)
Average Users
- retragetting based on interest
- context based promotions
- data mining
- feedback
Evangelism marketing
User Sentiment Level

Figure 10. Bell Curve of our User Base. Social CRM has the purpose of moving the curve to the right and increasing the number of users.

The content of each interaction is only stored once, and will be archive by DataSift. When DataSift introduced Historic Queues, it bought the data from Twitter for the previous two years. This data amounted to 300 TB, which was delivered over the Internet and normalized according to the way all interaction objects are processed, including augmentations.

Social CRM integration with the Dispatching System

The dispatching system is offered as a SAAS product to enterprise clients. The Social CRM system builds upon it and has the purpose of aggregating as much information about the users as possible. The most important data that we extract using DataSift and utilize in our system is presented below.

User sentiment score—describes the sentiment the user has towards the application. This metric is closely monitored as it groups the users into three major categories.

The first category is composed of users that are neither highly enthusiastic about using the system nor highly negative. They are the regular users and compose 80%–90% of the user base. In general no human interaction with these users is performed. They are advertised and retargeted using their profile and interests, as we will describe bellow. The second category is composed of users that are negative on the system. They have probably experienced either a problem with the system or in the interaction with the driver for example. Whatever the cause, the problem should be addressed so that it will not affect other users or have a spillover effect. The most effective way of dealing with highly negative users is an apologetic customer support phone call during which the user is offered free credit for retrying the system and with the promise that his problem will be addressed as soon as possible.

This method has been used effectively by Uber for monitoring their drivers. The last category is composed of users who are extremely positive on using the system, which we will be calling evangelizers. This category should be enticed to promote the system by offering them a more privileged role that allows better service and social recognition. This method of gamification at the high levels of the customer spectrum has been proven by numerous companies including Foursquare, Path, Quora and StackExchange.

Interaction History—this provides anyone interested in a specific user the ability to see the exact posts that constitute the sentiment index, or to identify what the pain points of a highly negative user are. All the interactions provided by DataSift are stored in our database and augment the user profile.

Promotions History—all the promotions and advertising campaigns that have ever been targeted to a specific user are recorded. This is for two reasons. One of them is to see the conversion and activation factors of specific campaigns. The other is to aid us in future campaigns by selecting only those users that have or have not been targeted previously within a specific program.

More information on the users—by aggregating more information about each user such as age, locations, interests, network of friends *etc*., we are able to understand the user base better and it helps with categorizing them better. The purpose of the entire Social CRM component is to improve the service and increase the success rate of our marketing campaigns by better targeting which will reduce the total cost. We can thus create online marketing campaigns that can be extremely targeted and still benefit from low CPC rates.

In this section we have detailed the implementation of the proposed system and explained the functionality of the complete architecture by breaking it up into the composing parts. We also analyzed how we managed to integrate the two systems in an efficient and scalable manner by using Windows Azure Cloud Computing as the underlying architecture for our system. We decomposed the system into two large subsystems: the dispatching system and the Social CRM system.

We explained how a cross platform approach can be used to developing mobile application and we have proved the native capacities of the system by using the GPS API. We also presented how the mobile application can be integrated with the Cloud and how the Cloud can act as a data synchronization mechanism between the client mobile application, the driver application and the operator panel. Regarding the Social CRM system, we explained how it can be built to extract and augment the data in our database and bring more value and customer engagement.

4. Testing and Validation

4.1. Mobile based Client Application Testing

Testing of the mobile applications has been performed, covering usability testing and integration testing. Usability testing of software applications developed for mobile devices faces a variety of challenges due to their unique features, such as limited bandwidth, unreliability of wireless networks, as well as the changing context. For assuring the system correctness Use Case Driven Testing of the functionalities of the system must be addressed in various environmental contexts such as no Internet access, no GPS signal *etc*. The client application was designed with these considerations in mind and the testing approach was by using Use Case Driven testing in various conditions. The following issues have been tested:

- **Mobile application context.** It can be defined as "any information that characterizes a situation related to the interaction between users, applications, and the surrounding environment [2]." For our client application we have tested how the app was performing in different times and places, especially how GPS was acting inside buildings and how well wireless networks aided the positioning based on the accuracy of position determination. It is very difficult to select a methodology that can include all possibilities of mobile context in a single usability test. If the user position cannot be obtained and the GPS was turned off, we asked the user to turn it on or input the address manually.

- **Connectivity and Mobile Internet Coverage.** Logistic Systems are characterized by very high availability constraints. Lack of connectivity or slow Internet speed is common with mobile applications. We tested whether the application can identify this situation and alert it to the user, notifying him that the app cannot work properly and he has to either turn on internet (if this is not turn on already), or just report that the app cannot work in the absence of an internet connection. According to Ericsson, 85% of the world population will have access to 3G coverage by 2017. In the European Union, over 95% of the metropolitan areas are covered by 3G. According to the same report from Ericsson, by 2017 50% of the world population will have access to 4G which will enable even richer interactions in terms of data synchronization and reach media content, especially voice and video streaming.
- **UI/UX testing for different screen sizes.** The mobile screen resolution is extremely fragmented. We also realized that many mobile devices that will be used as dispatching terminals will range from small 3 inch diagonal low resolution devices, up to tablets reaching over 10 inch in diagonal. These concerns affect mobile applications. We tested the applications on multiple devices and also used UI elements that can adapt proportionally to screen sizes. We have observed that for some small screens the application was not displayed properly because we have implemented some minimum size attributes in order to display the text completely. In order to resolve the issue we added test cases for small screen displays, and if the screen width was bellow a specific value, we used a smaller font and minimum width, which solved the problem. In order to test the aspect of the application rendered on different devices, virtualization tools such as Samsung Remote Test Lab. The Remote Test Lab is a free service from Samsung Developers. With it, you can access a remote physical device through a network, install an application and test it remotely.
- **CPU and Memory performance considerations.** Mobile devices have a high diversity of performance specifications. As we intend to support old and new devices as well, we need to test how the application performs on different hardware environments. These requirements must be tested especially if the work is not mitigated to the Cloud. For our system this was less of a concern as we perform the processing and storage on the Cloud. The order history and user data are all stored in the Cloud, and is consumed using the service API. Because we do not have access to all hardware devices, we used pCloud platform [18], which allows virtualization of different devices using screen streaming. Using their service we were able to deploy the application to a large variety of Android handsets and test the behavior of the system on both old and new devices.

4.2. API Testing and Validation

The API was implemented using the REST protocol and the JSON data passing format. All communication between the Cloud System and other systems is done using the API (excluding the integration with the Social CRM System which is also done using the database). In the first instance the validation of the service was performed by adding a new project to the solution and creating *Unit Test* classes. Inside this class specific methods for testing functionalities have been defined. Sending JSON requests required a series of step including creating the JSON request data from the CLR object, sending the request using a selected verb, and handling the HTTP specific errors, we defined a method called SendRequest.

Using this method we specified the resource we wanted to test, the verb and the data.

The data object was defined as dynamic to allow for easier operation with encoding and decoding JSON data. We designed the system to return specific error codes for specific errors such as duplicate telephone number. When designing the Unit Test we have tested if this error codes are returned appropriately. For example when creating a new user we tested different combinations of telephone and user name.

As we added new functionality to the system we expended the test scenarios and we rerun the test periodically to ensure that the system remains valid after performing modifications.

4.3. Load Testing for Scalability and Synchronization

For Load Testing, we were interested in how the system would reach to a high increase in usage. As we built the System on Windows Azure, we were certain that the scalability aspect will be mitigated effectively as long as we did not have any bottlenecks. Load testing also allowed for an efficient way of discovering errors in the system that occurred in scenarios such as concurrent access to shared resources and overriding of data that was not protected using locks. For Load Testing we first used a number of 10000 concurrent threads that were fired from a virtual machine in the cloud. We used as virtual machine in the Windows Azure Cloud as the communication latency would be minimized and it would help us discovering synchronization issues.

Once we handled all synchronization issues that have arisen through our testing, we needed a better way of testing the scalability beyond what we could have obtained using simple threads. For this we relied on LoadStorm, which is a cloud based tool that can create multiple client virtual machines firing requests at the test server requests can be customized with specific automate trigger based on the received responses. Up to 250 requests/second Windows Azure used just one medium size virtual machine. As the stress on the server reached over 90% CPU usage, automatic scaling was performed, which increased the number of virtual machines and decreased the response time to around 0.5 s. The time required for a virtual machine to launch was around 3–4 min.

4.4. Social Media Analysis

For the Social CRM component, the data is received in real time. However delays are to be expected. According to DataSift SLA (Service level agreement), the maximum delay between a Tweet being posted by a user and the time it is processed by DataSift should be less than 4 s. In this respect we analyze what is the average delay we will encounter from a tweet being posted on Twitter and the time it is analyzed by our Social CRM component.

In order to calculate the time delay we fired a number of 300 threads that each posted a message on Twitter. Then we used a stopwatch to obtain the time interval between when the message was posted and the moment the interaction was delivered from DataSift into our system. It is important to understand that the delays between the time the user publishes an interaction on other dataset such as blogs or discussion groups and the time DataSift processes this interaction can be much larger and span up to one hour.

4.5. Coverage Ratio of Matching Interactions

While analyzing the processing lag described in the previous paragraph we observed that for the 300 interactions we submitted on twitter using Twitters public API, DataSift received only 248. This means that 52 interactions were not identified by DataSift. Further investigation revealed that the reason why this was happening was because we had a sudden burst in activity that lasted only a few seconds. In this test case we tried to identify if the DataSift platform provides a high enough coverage ratio. For this we fired a number of 1000 Twitter posts over a range of 10 different accounts. Each interaction was submitted at a time interval ranging randomly between 1 and 10 s from the last one. A number of 10 threads were used for posting the threads. In this scenario the number of interactions captured by DataSift have been of 972, which translates into a 97.2% coverage ratio.

In conclusion we have tested the various components of the system and we concluding that the system is valid and works according to the initial functional and non-functional requirements.

5. Conclusions and further developments

We have detailed the implemented the system according to the specifications. We started by analyzing the requirements for our system by comparing similar applications in order to identify the most important features. We than stated the functionalities which the system was designed to offer and we created the use cases for our actors. We explained the two large subsystems that constitute our project: the dispatching system and the Social CRM system. Before implementation we created the conceptual model and listed the suitable technologies necessarily for meeting our requirements. In this respect we used DataSift as the underlying platform for building the Social CRM engine and Titanium as the best solution for developing cross platform mobile applications We also analyzed what the main threats to REST security are and we have enumerated some actions that we have taken in order to lower the chances of an attacker breaking the system or to help identify and mitigate an attack. We identified what are the main differences in terms of security between running the applications on premise *vs*. Cloud, as this has been a particular controversial topic. We than analyzed what methods can be used to secure REST services.

The Social CRM component of our application was designed in order to augment the information on our users and enhance the performance of our marketing campaigns. For this we used DataSift as the underling platform for extraction of interactions from Social Networks and other online public data sources. We than detailed the implementation of DataSift and how it extracts, normalizes and delivers the data into our system. We also examined some technical challenges that DataSift has encountered and explained the solutions that were applied such as defining time intervals for processing historic content. The Datashift remains a good option which is able to sustain a growing level of scalability for our system.

In conclusion we have tested the various components of the system and we are affirming that the system is valid and works according to the initial specified functional and non-functional requirements. The system is for now up and running [19], but as for any real system there are a lot of further developments and improvements which are determined by production usage.

As further developments, we would like to explore some other aspects for improving the system as elaborating on the dispatch strategy [20], integrating a solution for cruising mile reduction as presented in [21], or even a component for finding potential passengers as proposed in [22]. We also plan in the next release to complement our services with other services such as for example Google traffic API [23] or Skyhook [24] in order to create some new functionalities. Concerning the data, we started the project with relational database, but also we take into account the alternative of transferring the data persistence layer to a noSQL solution (such as for example Redis or Couchdatabase) as a more scalable solution. As a potential problem from a real world environment perspective exploitation of the system, capturing massive data from social networks could represent a possible problem, but designing appropriate filters for data, must be able to solve such a prospective bottleneck.

Author Contributions

This paper was a result of a collaboration between the two authors. Razvan Popa came with the idea and implemented the system software. Cosmina Ivan refined the initial idea, supervised the project and contributed to the writing and editing of the manuscript. Both authors have read and approved the final manuscript.

Conflicts of Interest

The authors declare no conflict of interest.

References

1. Ericsson Mobility Report, 2014. Available online: http://www.ericsson.com/res/docs/2014 /ericsson-mobility-report-june-2014.pdf (accessed on 25 September 2014).
2. Sharma, A.; Soni, P. Mobile Cloud Computing (MCC) Open Research Issues, IJIET, 2013, Available online: http://ijiet.com/wp-content/uploads/2013/02/5.pdf (accessed on 29 September 2014).
3. DataSift Guide. Available online: http://dev.datasift.com/docs (accessed on 29 June 2015).
4. Gnip, Social Media in Financial Markets, 2013. Available online: http://gnip.com/docs/social-media-and-markets-the-coming-of-age.pdf (accessed on 15 November 2014).
5. Wilson, T. "How GPS Phones Work", 2011. Available online: http://www.howstuffworks.com /gps-phone.htm (accessed on 15 January 2015).
6. Titanium. Available online: http://www.appcelerator.com/product/ (accessed on 29 June 2015).
7. Microsoft WCF Feature Details. Available online: http://msdn.microsoft.com/en-us/library/ ms733103.aspx (accessed on 29 June 2015).
8. Microsoft ADO.NET Entity Framework At-a-Glance. Available online: http://msdn.microsoft.com /en-us/data/aa937709 (accessed on 5 December 2014).
9. ASP.NET SignalR. Available online: http://signalr.net/ (accessed on 29 June 2015).
10. Apple Push Notification Service. Available online: https://developer.apple.com/notifications/ (accessed on 15 December 2014).
11. Android Cloud to Device Messaging. Available online: https://developers.google.com/android/ c2dm/ (accessed on 15 January 2015).
12. Urban Airship. Available online: http://urbanairship.com/ (accessed on 29 June 2015).
13. Alonso. G.F.; Casati, F.H.; Kuno, H. *Web Services: Concepts, Architectures, Applications*; Springer Berlin, Germany, 2004.

14. Dinh, H.T.; Lee, C.; Niyato, D.; Wang, P. A survey of mobile cloud computing: Architecture, applications, and approaches. *Wirel. Commun. Mobile Comput.* **2013**, *13*, 1587–1611.
15. Tyree, J.; Akerman, A. Architecture decisions: Demystifying architecture. *IEEE Softw.* **2005**, *22*, 19–27.
16. Schultz-Møller, N.P.; Migliavacca, M. Distributed Complex Event Processing with Query Rewriting, Imperial College London. In Proceedings of the Third ACM International Conference on Distributed Event-Based Systems, 6–9 July 2009, Nashville, TN, USA; doi:10.1145/1619258.1619264.
17. Bass, L.; Clements, P.; Kazman, R. *Software Architecture in Practice*; Wesley: Boston, MA, USA, 2003.
18. Pcloud. Available online: www.pcloud.com (accessed on 29 June 2015).
19. Taxy Army. Available online: www.taxiarmy.com (accessed on 29 June 2015).
20. Hou, Y.; Li, X.; Zhao, Y.; Jia, X.; Sadek, A.W.; Hulme, K.; Qiao, C. Towards efficient vacant taxis cruising guidance. In Proceedings of the 2013 IEEE Global Communications Conference (GLOBECOM), Atlanta, GA, USA, 9–13 December 2013; pp. 54–59.
21. Zhang, D.; He, T. pCruise: An On-demand Cruising Miles Reduction for Taxicab Networks. In Proceedings of the 2012 IEEE 33rd Real-Time Systems Symposium, San Jan, PR, USA, 4–7 December 2012; pp. 85–94.
22. Yuan, N.J.; Zheng, Y.; Zhang, L.; Xie, X. T-finder: A recommender system for finding passengers and vacant taxis. *IEEE Trans. Knowl. Data Eng.* **2013**, *25*, 2390–2403.
23. Google Traffic API. Available online: https://developers.google.com/maps/documentation/javascript/examples/layer-traffic (accessed on 26 June 2015).
24. Skyhook Technology. Available online: http://www.skyhookwireless.com/ (accessed on 26 June 2015).