

MIDDLEWARE PLATFORM FOR MOBILE CROWD SENSING APPLICATIONS
USING HTML5 APIS AND WEB TECHNOLOGIES

by

IOANNIS VAKINTIS

A THESIS

Submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE



DEPARTMENT OF INFORMATICS ENGINEERING

SCHOOL OF APPLIED TECHNOLOGY

TECHNOLOGICAL EDUCATIONAL INSTITUTE OF CRETE

2015

Approved by:

Spyros Panagiotakis
Assistant Professor

Copyright ©

VAKINTIS IOANNIS

2015

All rights reserved. No part of the material protected by this copyright notice may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage and retrieval system, without written permission from the author.

Abstract

Today, smart devices are flooding the internet with data that are everywhere and in any form. In addition, Web technologies, such as HTML5, have made the personalized interaction of a digital artifact with the web easier than ever. One area of ubiquitous computing is the interaction of smart devices with the physical world. The data obtained from a device that can sense the physical world can generate an endless amount of personal applications that make life easier. In this thesis, we design a web platform which is interfaced with the real world through the sensors of various mobile devices in order to group and graphically present the retrieved data following statistical processing. The platform consists of two application specific components: the first, the client part, runs in the user device to collect sensor data and transmit them; the second, the server part, runs in the cloud and is responsible for analyzing and visualizing the data from all devices in a human friendly format, e.g. a map. The application is multi-sensor as it can collect data from almost all sensors of mobile devices. Besides the use of the platform as a participatory and opportunistic sensing application, our endmost aim is to be used with other Internet of Things equipment for the introduction to the third generation of Web characterized as ubiquitous web.

Table of Contents

Copyright ©	ii
Abstract	iv
Table of Contents	vi
List of Figures	viii
List of Tables	x
Acknowledgements	xii
CHAPTER 1: INTRODUCTION	1
1.1 UBIQUITOUS SENSING	1
1.2 CONTRIBUTIONS	4
1.3 STRUCTURE	6
CHAPTER 2: SENSING ARCHITECTURES	7
2.1 SMARTPHONE-BASED SENSING	7
2.2 INTRODUCTION TO MOBILE SENSORS	7
2.3 SURVEY ON SENSING ARCHITECTURES	12
CHAPTER 3: WEB TECHNOLOGIES	20
3.1 HTML5, THE CONNECTOR BETWEEN WEB AND MOBILE	20
3.2 HTML5 OVERVIEW	20
3.3 GOOGLE SERVICES	45
3.4 METEOR PLATFORM	49
3.5 JSON	53
3.6 BSON	54
3.7 GEOJSON	54
3.8 EXT JS FRAMEWORK	55
3.9 X3D & X3DOM	55
CHAPTER 4: PLATFORM ARCHITECTURE	57
4.1 CROWDSENSING PLATFORM INTRODUCTION	57
4.2 COMPONENTS	58
4.2.1 CLIENT COMPONENT	58
4.2.2 SERVER COMPONENT	67

4.2.3 3 RD PARTY COMPONENT	75
4.3 SERVER CONFIGURATION	88
CHAPTER 5: PRIVACY	91
5.1 PRIVACY IN CROWDSENSING.....	91
CHAPTER 6: MOTIVATION	94
6.1 USER INCENTIVES	94
6.1.1 GAMIFICATION	94
CHAPTER 7: EVALUATION	103
7.1 OVERVIEW OF PERFORMANCE & BENCHMARK TESTS	103
7.2 EVALUATION TEST OF PLATFORM TASKS	103
7.3 DATABASE – CENTRIC APPROACH	110
7.4 TEST BED OF DATABASES	114
CHAPTER 8: CONCLUSION & FUTURE WORK	127
REFERENCES	130

List of Figures

Figure 2-1: Classification of Crowdsensing systems.....	13
Figure 2-2: A screenshot of the interactive map used by IBM Almaden Research Center to display the contributions of the users of Creek Watch	18
Figure 3-1: The DeviceOrientation event properties	26
Figure 3-2: Timing attributes of the Resource Timing API.....	32
Figure 3-3:Life-cycle of a WebSocket session	39
Figure 3-4: Audio context.....	43
Figure 3-5: Geocoding services translate an address into geographic coordinates and display a marker in a map.	46
Figure 3-6: Meteor architecture	50
Figure 4-1: Middleware platform architecture.....	58
Figure 4-2: Noise data capture.....	59
Figure 4-3: Sensor switcher	60
Figure 4-4: Permission dialog.....	60
Figure 4-5: Client application screenshot	66
Figure 4-6: Main panel.....	67
Figure 4-7: Geocoding statistics from Google geocoding API.....	68
Figure 4-8: Averages geocoding statistics from Google geocoding API	69
Figure 4-9:Time aggregation sequence job.....	73
Figure 4-10: API documentation	78
Figure 4-12: Reactive real-time map	80
Figure 4-13: Historical map	83
Figure 4-14: Spatial - temporal analytics charts	84
Figure 4-15: Country noise averages with 2D visualization.....	86
Figure 4-16: Country noise averages with 3D visualization.....	88
Figure 5-1: Generic structure of task flow in MCS	91
Figure 6-1: Farm Ville	95
Figure 6-2: The state of flow is achieved when a player is placed between anxiety and boredom over a period of time	97

Figure 6-3: Yahoo! Answers experience point system	100
Figure 6-4: Noise pollution puzzle	101
Figure 6-5: Player leaderboard.....	102
Figure 7-1: Marker test results	105
Figure 7-2: Dynamic map results.....	106
Figure 7-3: Collection API results	107
Figure 7-4: Country charts results.....	108
Figure 7-5: Country averages results	108
Figure 7-6: Locality charts.....	109
Figure 7-7: Historical map results.....	109
Figure 7-8: MongoDB object_id.....	113
Figure 7-9: Benchmark architecture	116
Figure 7-10: Administrator page.....	118
Figure 7-11: Insertion test.....	120
Figure 7-12: Reading test.....	122
Figure 7-13: Reading with sorting	123
Figure 7-14: Searching test	124
Figure 7-15: Removing test	125
Figure 7-16: Aggregation test	126

List of Tables

Table 1.1: High-end smartphones with embedded sensors.....	2
Table 1.2: Typology of “Crowdsensing”	3
Table 2.1: Sensors categories.....	7
Table 2.2: Smartphone sensors	8
Table 3.1: Sensors and Hardware APIS.....	22
Table 3.2: Geolocation API methods.....	24
Table 3.3: Position object properties	24
Table 3.4: Orientation API methods	25
Table 3.5: Device Orientation Event Properties	26
Table 3.6: Device Motion Event Properties.....	27
Table 3.7: Battery Status API Properties	27
Table 3.8: Proximity Sensor API methods	28
Table 3.9: LightLevelEvent values	28
Table 3.10: Network Information API properties.....	31
Table 3.11: User Timing API properties.....	33
Table 3.12: VisibilityState values	35
Table 3.13: Web Workers capabilities and limitations.....	36
Table 3.14: Opcode values for WebSockets.....	40
Table 3.15: Event handlers for WebSockets.....	42
Table 3.16: Audio tag limitations	42
Table 3.17: Types of Web Audio Nodes	44
Table 3.18: Google Geocoding API output: Status codes	47
Table 3.19: Google Geocoding API output: Result codes.....	47
Table 3.20: Returning types [] and address_components [].	48
Table 3.21: Meteor platform methods	53
Table 3.22: JSON basic types	53
Table 4.1: Network Information API.....	62
Table 4.2: Description of API keys.....	76
Table 4.3: Properties of heatmap	80

Table 7.1: SQL vs NoSQL.....	111
Table 7.2: Sensor data structure.....	117

Acknowledgements

Foremost, I would like to express my sincere gratitude to Prof. Spyros Panagiotakis for supervising the thesis work. I especially wish to thank my coworkers in the MCLab, for their help at all stages of the thesis. I am also thankful to my life-mate Eirini Kostaki for her encouragement, and for helping me stay sane through the whole difficult period. Most importantly, nothing would have been possible without the love and patience of my family.

CHAPTER 1: INTRODUCTION

1.1 UBIQUITOUS SENSING

Mobile sensing has changed many forms over the years. In the decade of 80s and 90s, mobile computing point to develop a variety of sensor equipment to monitor phenomena of interest such as atmosphere or potholes on road. In the next years, wireless networking technologies overcome lot of obstacles and sensors via embedded communication modules start to connected not only each other's but also with backend servers [69]. Nowadays, ubiquitous or pervasive sensing [19, 79] enabled by web and mobile technologies related in a wide range of activities in our society. Transportation and Civil Infrastructure Monitoring [44] [45], Environmental Monitoring [31-36] [43], Health and Fitness [41], urban sensing [107] and traffic monitoring, social networks [42] are some areas which benefit from ubiquitous sensing. Smart devices have played a major role to this trend. Smartphones (Samsung Galaxy S5, iPhone 6), tablets (HTC Nexus 9, Samsung Galaxy Tab S), music player (iPods), sensor embedded gaming systems (Wii, Kinect), and in-vehicle sensing devices (GPS) are flooding the market and feed sensor data to the Internet. They have equipped with various sensors (e.g., accelerometer, ambient light, camera, microphone, gyroscope and proximity) and so they transform a near-ubiquitous smart device into a global mobile sensing device [20], [21]. Table 1.1 shows a few modern high-end smartphones with their embedded sensors.

In the upcoming years more sensors will be embedded in the smartphones. The new version of Samsung Galaxy has concluded two more sensors, heart rate sensor and finger scanner [37]. It is the first time in history of smartphones that a smartphone conclude a heart rate sensor and give the capability to the users to monitor their physical information. Mobile users can measure their heart rate before and after a workout to check out their health and workout status. The second sensor that Samsung galaxy S5 features is finger scanner which improves the usability and the security of the smartphone. Some features of Finger Scanner are biometric screen locking, individual file locking with "Private mode" and secure mobile payments.

Table 1.1: High-end smartphones with embedded sensors

Devices\Sensors	Camera	Microphone	Accelerometer	Digital Compass	Light	Proximity	Gyro	Barometer	Temperature/ humidity
Samsung Galaxy S3	✓	✓	✓	✓	✓	✓	✓	✓	
Samsung Galaxy S4	✓	✓	✓	✓	✓	✓	✓	✓	✓
Motorola Nexus 6	✓	✓	✓	✓		✓	✓	✓	
Iphone 6	✓	✓	✓	✓		✓	✓	✓	
Nokia Lumia 720	✓	✓	✓	✓		✓			

Observing the above table can be easily understood that the sensing capabilities of smartphones can measure individual or community phenomena. The category of individual phenomena includes several actions of a specific device owner, which usually are divided into 3 categories a) movement patterns such as walking and running, b) modes of transportation such as biking, driving or taking a bus and c) activities such as listening to music and making coffee. Most of the time, the user can have access to his personal data which are presented graphically as statistic. On the other hand community phenomena are related to the actions of a set of people and are not limited to a specific user. Community phenomena can be characterized real-time traffic patterns, air [46], water or noise pollution and pothole patrol. The way, in which users involved in the process of collections of sensor data, divide community phenomena to participatory and opportunistic sensing category. In chapter 2 we will present a deep analysis for the both categories.

In recent years, a new sensing architecture has spurred the attention of the scientific community in contrast with the others [6]. Mobile Crowdsensing or else MCS, is a new business model that allow to a huge number of mobile users to exchange information not only between them but also for a set of actions that may have effect to the community. In general, the term “Crowdsensing” refers to the collection and sharing of sensor data with the scope to measure a community phenomenon. It is a very attractive solution for companies and organizations to collect significant data without spend enormous amount of money. A very important advantage of Crowdsensing is that unlike an infrastructure-based sensing solution, crowdsensing can potentially be cheaper as it does not require the deployment of expensive fixed infrastructure. A

Crowdsensing application can be incorporate into one of the five categories that listed in the Table 1.2. Those categories are formed with two main criteria in mind which are the involvement of the mobile user in the procedure and the type of the measured phenomenon.

Table 1.2: Typology of “Crowdsensing”

Criterion	<i>Involvement of the user in the Crowdsensing process</i>	<i>Type of measured phenomenon</i>
Types of Crowdsensing	Participatory crowdsensing	Environmental crowdsensing
	Opportunistic crowdsensing	Infrastructure crowdsensing
		Social crowdsensing

Starting with the first criterion, the involvement of the user in the Crowdsensing process, Crowdsensing applications are divided into participatory and opportunistic category. In participatory crowdsensing, the users send sensor data to the server, doing an active effect. On the other hand, in opportunistic crowdsensing applications the sensor data are sending automatic, with little or no involvement of the user. Moving to the second criterion, the type of measured phenomenon, we have 3 categories: 1) Environmental 2) Infrastructure and 3) Social. Environmental crowdsensing used to measure natural phenomena such as noise pollution, level of water and air pollution. Infrastructure crowdsensing used to measure public infrastructure such as road conditions or traffic congestion. Finally, social crowdsensing used to measure social behavior of individuals such as the shops visited by a citizen or the holiday travel destinations. In Chapter 2 we will present a more detailed analysis about Crowdsensing applications which will include state of the art and privacy aspects. Also, it will present it, an extend survey on existing Mobile Crowdsensing applications.

The creation of such sensing architectures has blossomed because the mobile and web technologies offer unlimited possibilities. Modern mobile operating systems fully exploit the features of sensing devices offering multiple capabilities to the mobile and web developers. The developers can build applications for handheld devices with three deferent ways: 1) native, 2) web and 3) hybrid. Native applications build separately for each operating system and they are pre-installed on the mobile phones during manufacturing or can be download from distributed

application stores such as Google play or App store. Web applications delivered using a server-side or client-side processing to provide an “application-like” experience within a Web browser. Last category is hybrid application which is the marriage of web technology and native execution. Hybrid app is built with web technologies and mobile web implementations and it is run inside a native container on a mobile device. Android and iOS are the two most well-known mobile operating systems of the world having the 96, 1% of the Worldwide market share in the 3rd quarter of 2014 [73].

Beyond mobile technologies, the web is gaining momentum in the use of sensing devices. The way that we interact with the web is changing throughout the years. In the upcoming years web will enter to the 3rd phase it will called ubiquitous web or the intelligent web. Now, webpages there are not just pages that have colors, text and logos but are similar to desktop applications and they are turning to web applications [74]. The radical improvement of content usability, help the web applications to present their content more dynamically. Also, a web application can retrieve data from multiple sources and in real time. In many cases, the traditional HTTP communication between a web server and a browser is replaced with a single TCP connection, which it called WebSockets [75, 76, and 77]. The advantage of WebSocket protocol is that is providing a full-duplex bi-directional communication and can be used in both client and server applications. Besides the robust communications protocols such as Websocket, ‘the intelligent Web’ will have much more technology trends to extend. Semantic Web technologies, machine learning and reasoning, autonomous agents and distributed databases will drive the Web to be more open, connected and intelligent ecosystem.

1.2 CONTRIBUTIONS

In this thesis, we design a web-based cross-platform based on HTML5 APIs which is interfaced with the real world through the sensors of various mobile devices [3], [4] in order to group and graphically present the retrieved data following statistical processing. This is the first Crowdsensing platform which uses HTML5 APIS for the collection of the sensor data. The platform consists of two application-specific components: the first, the client part, runs in the user device to collect sensor data and transmit them; the second, the server part, runs in the cloud and is responsible for analyzing and visualizing the data from all devices in a human friendly

format, e.g. a map [10]. The application is multi-sensor as it can collect data from almost all sensors of mobile devices and is totally based on HTML5 features. Besides the use of the platform as a participatory and opportunistic sensing application [3], our endmost aim is to be used with other Internet of Things equipment for the introduction to the third generation of Web characterized as ubiquitous web [151].

In more detail, the client, which is implemented in the form of a web page, acts as the source of data and is located in the front end of our web platform. The end user via the client grants access to the sensors through the respective HTML5 APIs. User only needs to activate the client application to start the automatic procedure of sending the data to the cloud. Sensor APIs obtain the raw information and forward it to the next stage for analyzing. The data analysis is divided into two parts, local analysis [8] (at the client) and aggregate analysis (at the server). When local analysis finishes, useful data are sent to the server and stored in a cloud database. At this point, the data will pass to the second stage of analysis. This stage provides a map visualization and statistical analysis of data collected by all users. Statistical data are presented in the form of various charts. Both interactive map and statistics' charts can be accessed by anyone via a webpage.

The specific scenario that we have implemented concerns the measurement of noise and light. The user gives access to three sensors of his mobile device, namely the microphone (Get user media API [12]), the light sensor (Light sensor API) and Location (Geolocation API). Microphone records the ambient sound [11] and through an algorithm converts values to decibel. These values are periodically sent to the server through a WebSocket connection [13], [14] while the user is active. The server collects the decibel values along with the location of users to export the statistics. Light sensor act with the same procedure as microphone with the only difference that light data doesn't need further analysis. Light sensor API expose data as a lux values rather than Get user media which expose raw data information.

We use mobile and desktop devices as noise and light sensors and track users' location from geolocation API. The user device does not need to be equipped with GPS because geolocation API support several ways to find the location such as Wi-Fi, IP address, GSM and UMTS cell IDs. In chapter 2 we will provide a deeper analysis about the ways of getting the location from geolocation API.

Next, when we get the raw sound data using get user media API, we use an algorithm to transform the obtained data to a measurement unit as decibels. The process is operated at client side or else in the front-end of the server platform. In the front-end of the device the user can see their geo-location parallel with their personal data in a real-time interactive google map. The visitor page contains many capabilities such as statistical representation, google maps visualization, a collection API and dynamic map visualization.

To conclude with, we introduce a crowdsourcing technique mostly in environmental sensing which group and present sensor data based on the geographical position of the users. The web platform is location-aware providing in real-time the noise and light data of live participants.

1.3 STRUCTURE

The rest of the thesis is organized as follows; in chapter 2 we introduce to mobile sensors and survey various sensing architectures. Chapter 3 introduces to the web technologies that have been used for the implementation of our thesis. Chapter 4 presents the architecture of our web platform. Chapter 5 and 6 acquaints various issues of privacy and also incentives for the users to participate. Chapter 7 includes evaluation and performance tests. Finally Chapter 8 concludes the thesis and discusses about future work.

CHAPTER 2: SENSING ARCHITECTURES

2.1 SMARTPHONE-BASED SENSING

With the penetration of smart mobile devices into our daily activities, we rapidly pace from traditional sensor networks to crowdsensing solutions known as “people sensing” [20, 69]. The term “people sensing” was first introduced in 2005 and referred to a moving sensor network of people which carry sensing devices to monitor a phenomenon of interest. In the early years of “people sensing” it was difficult to program a smartphone because they had limited programming and embedded sensor capabilities. In the upcoming years, the two popular smartphone platforms, iOS and Android made their appearance to the market broadening the horizons and opening the doors to the public for rapid and easy programming. In parallel with the continuous development of these two platforms, smartphones embedded more and more sensors transforming them from simple communication devices to personal intelligent assistants [22]. There is a big variety of build-in sensors in smartphones with the ability to measure orientation, motion, light and other environmental conditions that provide high precision and accurate data to the end user.

2.2 INTRODUCTION TO MOBILE SENSORS

Today’s smartphones are moving sensor nodes offering audio and video recording, GPS navigation and Wifi connectivity. The embedded sensors are divided into several categories which are determined by the type of sensor (hardware - software) and the kind of measurement offered (motion – orientation - environmental). Those types of sensors are supported by most mobile operation systems such as Android, iOS, Windows and Blackberry. They provide raw information with high accuracy offering developers tremendous potential at a low cost. Below is a table with a short description of the types of sensors [89].

Table 2.1: Sensors categories

Sensors categories	
	<u>Hardware-based:</u> <i>These are physical components embedded</i>

Types of sensors	<i>into a smartphone or tablet device.</i>
	<u>Software-based:</u> <i>They are not physical devices, although they mimic hardware-based sensors. They obtain data from one or more of the hardware-based sensors. Other titles: Virtual sensors or synthetic sensors.</i>
Kind of measurements	<u>Motion:</u> <i>They measure acceleration and rotational forces along three axes. They include accelerometers, gravity sensors, gyroscopes, and rotational vector sensors.</i>
	<u>Orientation:</u> <i>These sensors measure the physical position of a device. They include orientation sensors and magnetometers.</i>
	<u>Environmental:</u> <i>They measure various environmental parameters, such as illumination, humidity and pressure. This includes barometers, photometers, and thermometers.</i>
	<u>Multimedia:</u> <i>These sensors retrieve multimedia content such as video and audio streams and capture images.</i>

Below is a table of embedded physical sensors in a typical modern smartphone. The table presents the sensors along with the type, kind of measurement and a detail description.

Table 2.2: Smartphone sensors

Sensor	Type	Kind of measurement	Description	Common Uses
Accelerometer	Hardware	Motion	They are the simple MEMS (Microelectromechanical System) devices which are used to measure the acceleration force in m/s ² that is applied to a device on all three physical axes (x, y,	Motion detection (shake, tilt, etc.).

			and z), including the force of gravity. A 3-axis accelerometer senses the orientation of the phone and rotates the screen, images and web browsers accordingly, allowing the user to easily switch between portrait and landscape mode.	
Temperature	Hardware	Environmental	Measures the ambient room temperature in degrees Celsius (°C).	Monitoring air temperatures.
Gravity	Software or Hardware	Motion	Measures the force of gravity in m/s ² that is applied to a device on all three physical axes (x, y, z).	Motion detection (shake, tilt, etc.).
Gyroscope	Hardware	Orientation	Measures a device's rate of rotation in rad/s around each of the three physical axes (x, y, and z).	Rotation detection (spin, turn, etc.).
Light	Hardware	Environmental	Measures the ambient light level (illumination) in lux.	Controlling screen brightness to save battery power.
Linear Acceleration	Software or Hardware	Motion	Measures the acceleration force in m/s ² that is applied to a device on all three physical axes (x, y, and z), excluding the force of gravity.	Monitoring acceleration along a single axis.
Magnetic	Hardware	Orientation	Measures the ambient geomagnetic field for all three physical axes (x, y, z) in μ T. It is relative to the Earth's magnetic north pole.	Creating a compass.
Orientation	Software	Orientation	Measures degrees of rotation that a device makes around	Determining device position.

			all three physical axes (x, y, z).	
Pressure	Hardware	Environmental	Measures the ambient air pressure in hPa or mbar.	Monitoring air pressure changes.
Proximity	Hardware	Orientation	Measures the proximity/position of an object in cm relative to the screen of a device. This sensor determines the position of the phone w.r.t the object.	Phone position during a call.
Humidity	Hardware	Environmental	Measures the relative ambient humidity in percent (%).	Monitoring dewpoint, absolute, and relative humidity.
Rotation Vector	Software or Hardware	Orientation	Measures the orientation of a device by providing the three elements of the device's rotation vector.	Motion detection and rotation detection.
Camera	Hardware	Multimedia	Capture pictures and videos.	Taking photos and recording video.
Microphone	Hardware	Multimedia	Capturing and encoding a variety of common audio formats.	Recording audio

2.2.1 SOCIAL SENSORS

Apart from the categories of physical sensors there is a separate type of sensors which emerges from the rise of Social networks (Facebook, Twitter, Instagram, Pinterest, LinkedIn, etc.) as well as location-based social networking services (Foursquare). As a matter of fact, Social networks act like social sensing devices exposing huge amounts of information such as individual interests, preferences and activities [90]. In [92] the Social Sensing is defined as a procedure that the sensors presented in mobile devices are exploited to infer data about people activities. In a more general definition, social sensors are a continuous source of information derived from social networks by exposing situations of users or social environments. Some

examples of Social sensors include Facebook status updates or posts, Twitter posts and Pinterest pins.

2.2.2 PERVASIVE SENSORS VS SOCIAL SENSORS

Comparing the pervasive sensors along with the social sensors we can derive many conclusions for the usage and gathering of sensor data.

- Pervasive sensors are physical components of a mobile device such as smartphones or tablets. On the other hand social sensors are software tools which originate from modern social networks.
- Social sensors can “sense” information that exists in the mind of the user. For example a user’s favorite song. This information can be obtained from a social post.
- Social sensors can be used as pervasive sensors with an abstract way. For example, understanding the location of the user from an image post.
- Social sensors can predict future situations. For example by reading a shared calendar.

2.2.3 SEMANTICS ON SENSORS

The definition of the word “semantic” is the study of meanings. In mobile sensing “semantics” are any meaningful information that we can be extracted from raw sensor data [70]. For example GPS sensors provide the coordinates of a location. This is a useless information by itself. But if we can combine this kind of information with another source of sensor data we can derive meaningful information, aiding for a better understanding of the phenomena in the mobile sensing era. The stored GPS logs can be used to determine the significant places associated to a user’s daily routines (e.g., home, office, shopping areas). Also, they can be used as travel patterns derivatives providing the means by which we understand how a user moves from one place to another (e.g. by driving or by using public transportation). This information can be

useful for developing an application to inform the user about delays. Similarly, accelerometer logs can be used to measure a user's average walking speed, compute step counts and estimate the calories burnt per day. The main purpose is to provide custom integrated services to the end customer.

2.2.4 ACCESS INTO DEVICE SENSORS

The access to device sensors is accomplished with three different ways: native programming, JavaScript and hybrid. The native access differs as every operating system has its own specific language (e.g. Android has Java, iOS has Objective C). Native applications provide full-access to a device's capabilities because they are running directly in the device. The second approach includes access through JavaScript and this is used for Web applications. JavaScript APIs will be further analyzed in the chapter's sequel. Finally, the hybrid frameworks provide a different approach which gives the capability to the developer to write code with Web technologies, such as HTML5, and create almost native applications. Examples of such frameworks are PhoneGap [48], Intel XDK [49], Enyos [50] and Mosync [51].

2.3 SURVEY ON SENSING ARCHITECTURES

The information that is sensed from the users' devices can be transmitted to a back-end server for further analysis. The combination of information from multiple mobile or desktop devices can reveal significant trends of an environment like predicting air and noise quality. Also, they can help to improve city management issues like the traffic sector, civil complaints or neighborhood problems. All these developments are under the category of Community sensing, People sensing, Participatory sensing, Opportunistic sensing, Crowdsensing, Crowdsourcing and Social sensing. These buzz words all describe the space of sensing architectures from various application perspectives. The purpose of these buzz words is to build platforms or applications that gather sensor data from volunteers belonging to the huge number of people with mobile devices, actively or passively. Nevertheless, a sensing platform can be characterized with more than one of the above names because it may contain characteristics from several sensing architectures.

Below we will survey several of the sensing architectures giving a small description about the scope and the relative applications.

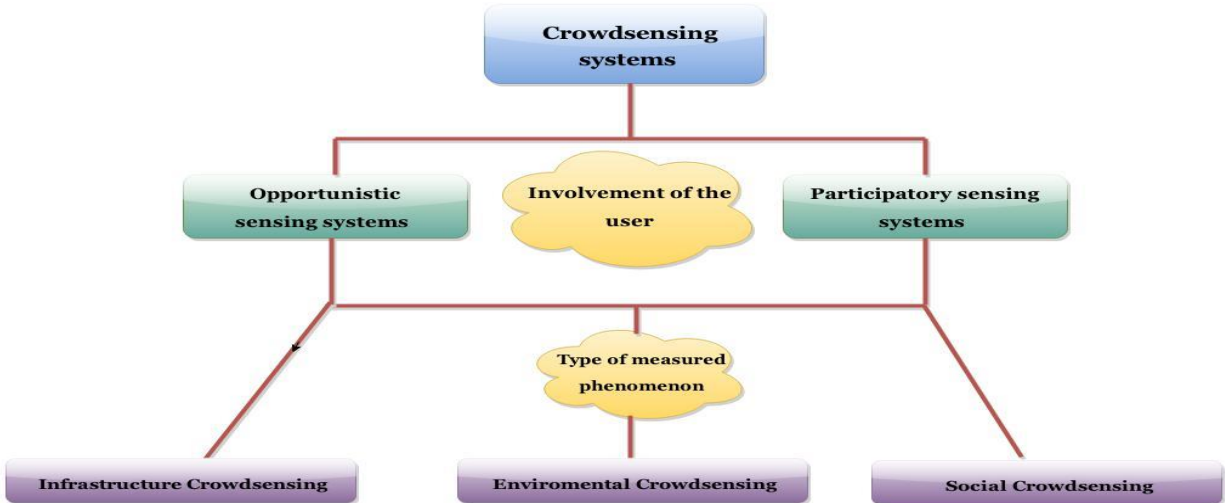


Figure 2-1: Classification of Crowdsensing systems

2.3.1 COMMUNITY AND PERSONAL SENSING

Sensing applications are classified into two categories, community and personal, based on the type of phenomena being measured [6]. Community sensing monitors large-scale phenomena with a large number of participants. Applications for community sensing are for example intelligent transportation systems which measure traffic congestion, air or noise pollution. These applications need to receive daily updates from the sensor data of participants to perform spatio-temporal analysis and determine various phenomenon occurrences. Community sensing is also divided into, popularly called, participatory sensing and opportunistic sensing. Both architectures will be further analyzed in the upcoming sub-chapters. On the other hand, there are personal sensing applications which refer to individuals. An example of a personal sensing application could be the monitoring of movement patterns such as walking or the transportation mode to determine the carbon footprint.

2.3.2 PARTICIPATORY SENSING

Participatory sensing [5, 47, 93, 94] refers to the vision of distributed data collection and analysis at the personal, urban, and global scale, in which an agreement is made with individuals to fulfill the requested sensing activities and are, thus, actively involved in the sensing actions (e.g. by taking a picture).

2.3.2.1 ROAD AND TRAFFIC MONITORING

A problem that concerns a lot of people is the monitoring of road and traffic conditions in a city. In [96] the authors present an application for monitoring road and traffic conditions using smartphones. They use the built-in sensors of smartphones such as the accelerometer, the microphone, and GPS to detect potholes, bumps, braking, and honking. Another similar paper is the Pothole patrol [97]. It uses some sensor-equipped vehicles (e.g. taxis) to gather data from the streets of Boston and then train the detector, a decision-making process, so it reaches to final conclusions. Four years later and 12245.77 kilometers away in the streets of Mumbai Wolverine, the authors of [98] tried also to identify road conditions using the accelerometer, magnetometer and GPS from their smartphones. Compared to the two previous papers they promise more accurate results. Bikestatic [99] is an application for improving the daily life of a cyclist. It documents routes using an Android application and the built-in sensors and then records the roughness and noisiness of a road. Then, the user can share the information and see visualizations of the data.

2.3.2.2 HEALTH

The monitoring of patients and the progress of their health is a very important issue for their relatives. Ambulation [100] is a tool for detecting the movement of patients who suffer from chronic diseases such as Multiple Sclerosis, Parkinson's, and Muscular Dystrophy. It runs as a service on the background of Android and Nokia N95 mobile phones and sends the collected data to a web server for visualization. AndWellness [101] is a web platform for health that uses

mobile devices as real-time sensor data collectors. The evaluation of the platform was made by people who are breast cancer survivors and young mothers. Other than monitoring the health of sick people there are applications for healthy people as well who wish to improve their lives by maintaining a healthier lifestyle. BeWell [102] tracks the everyday behavior of people by absorbing sensor data from a variety of multiple embedded smartphone sensors. It promotes wellbeing by monitoring daily activities such as sleep, physical activity and social interactions.

2.3.2.3 ENVIRONMENTAL MONITORING

In recent years several researches have been carried out in the field of environmental monitoring. Ikarus [103] is a flight recording device that collects sensor data from cross-country flights measuring thermal atmospheric conditions. The application provides thermal maps using GPS and barometer pressure values taken from mobile devices. The total results have been taken from more than 30,000 flights. In the same context, GasMobile [104] provides information about air quality and pollution using an Android client application and displays the data in a high-resolution air pollution map. To collect the data it uses an external sensor which is capable of measuring carbon dioxide values. Peir system [43] is a participatory sensing application for environmental monitoring which uses GPS location data to measure the exposure of the user carbon impact . In the server side, it uses HMM-based activity classification to determine the transportation mode, weather, carbon impact and other context data. Also, it provides new map-matching and GSM-augmented activity classification techniques and a selective hiding mechanism that generates believable proxy traces for times a user does not want their real location revealed. In the end, Peir provides a two-month usage statistics snapshot of a six-month trial.

2.3.3 OPPORTUNISTIC SENSING

Opportunistic sensing [7, 95] is referred to the collection of user data with minimum or no involvement of the participants (e.g. recording sound). In most cases opportunistic applications run in the background of the operating system, so there isn't any interaction with the users of the sensing devices.

2.3.3.1 OPPORTUNISTIC SENSING BACKGROUND

CarTel [104] is an opportunistic sensing project which maps traffic patterns in the streets of Boston and Seattle from small computers installed in vehicles. The computers contain GPS to measure location, speed and direction. Later, CafNet, uploads the sensor data to the relational database. CarTel has a web-based user interface for access to the database and allows for modification of data-gathering rules, such as time granularity. Some of the capabilities of CarTel are traffic monitoring, automotive notification and road-surface diagnostics. Both producers and consumers of the data are ordinary people and not scientists.

MetroSense [105] envisions a future full of sensor data. *Sensing will be people-centric.* The applications of MetroSense include tasking of personal mobile devices and sharing between adjacent hubs in an opportunistic way. The main concern is the use of sensor devices as a platform for opportunistic applications such as BikeNet [106]. BikeNet represents the first working mobile networked. Bicycles are equipped with a large number of sensors which communicate with each other and with adjacent bikes. In more detail, BikeNet contains a system architecture for collecting personal data from cyclists and environmental data along the route. When the sensors come into the range of a sensor access point the data is uploaded to the backend server. It collects various data about the cyclist such as heart rate and galvanic skin response. It also collects data about the cyclist's performance such as wheel speed, pedaling cadence and frame tilt and about the cyclist's surroundings such as sound level, carbon dioxide level, and cars. BikeNet provides many capabilities such as cycling performance metrics (current speed, distance traveled), user experience and environmental mapping, data collection and local presentation.

2.3.4 CROWDSENSING ~ CROWDSOURCING

Crowdsensing is a new way of sensing the real world which encourages people to participate and generate sensor data from their mobile devices. Sensor data is aggregated and fused in the cloud for further analysis and customer service delivery [72]. As mobile devices are referred mobile phones, wearable devices and smart vehicles. The embedded mobile sensors can acquire local knowledge e.g., location, noise level, traffic conditions, and in the future more specialized information such as pollution. A typical functionality of MCS application is first to collect raw sensing data from mobile devices and then to process it to a mechanism for local analytics [6]. Second, privacy preservation, the data is sent to the backend and aggregate analytics will further process it for different applications.

Crowdsensing is low-cost compared to a platform with static sensors and its range is far larger than the typical WSN systems. Moving users create an enormous range which can expand in the most improbable places. The main research challenges for Crowdsensing applications are privacy and incentive mechanisms. The nature of the data that is transferred between the applications is very sensitive and the functionality of privacy is considered very important for the smooth functionality of the application. Another important issue for Crowdsensing is the motivation of the user. The purpose is to keep the user for a long time inside the application and obtain large amount of sensor data. Crowdsensing applications are divided into three categories (i) Environmental, (ii) Infrastructure, and (iii) Social.

2.3.4.1 CROWDSENSING BACKGROUND

Medusa [71] is a platform for crowd-sensing applications with the purpose of simplifying the burden of managing crowd-sensing tasks for non-expert users. It uses MedScript, a high-level programming language and a runtime system called Medusa which is located both in the cloud and in the smartphone. Several scenarios have been created for the Medusa framework such as road-bump monitoring, citizen journalist, collaborative learning and auditioning. The criteria to demonstrate the expressivity of MedScript are the different sensors and different facilities that they use.

Creek Watch [108] is a participatory environmental crowdsensing application for monitoring water levels and the quality of the area around the water by the IBM Almaden Research Center. The user can submit a variety of information that concerns water such as the amount of water, the rate of flow and a picture of the waterway. The obtaining data is aggregated by IBM Almaden Center and then shared with institutes that are responsible for managing water resources. Fig.1 displays a screenshot of the interactive map used by IBM Almaden Research Center to display the contributions of the users.

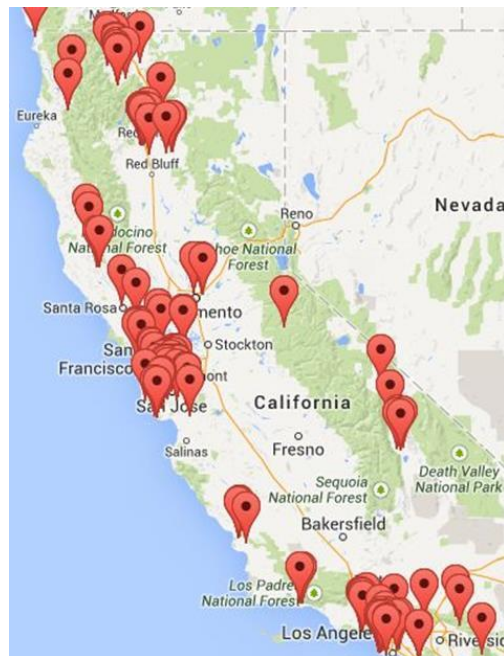


Figure 2-2: A screenshot of the interactive map used by IBM Almaden Research Center to display the contributions of the users of Creek Watch

DietSense [109] displays the eating habits of individuals in a social way, giving the capability to the users to compare their data. Individuals send manually their eating habits. Before that they need to answer to a survey of questions such as the place of the food, a picture of it and motivation for the chosen food. The improvements of eating habits are the incentive to use DietSense.

2.3.5 SOCIAL SENSING

As we mentioned above social sensing is a category in which data comes exclusively from social networks. The popular social sites such as Facebook and Twitter are some of the inexhaustible sources of social data [91]. One such application is the Dartmouth CenceMe [42] which collects data such as events in people lives, called sensing presence, and selectively shares this presence using online social networks. R. Ji et al. [110] report a work on mining famous city landmarks from blogs for personalized tourist suggestions. Their main contribution is a graph modeling framework to discover city landmarks by mining blog photo correlations with community supervision. Q. Zhao et al. [111] propose detecting and framing events from the real world by exploiting the tags supplied by users in Flickr photos. The temporal and locational distributions of tag usage are analyzed, tags related to aperiodic events and those of periodic events are distinguished. Tags are finally clustered and, for each cluster, a representing picture and tag is extracted.

CHAPTER 3: WEB TECHNOLOGIES

3.1 HTML5, THE CONNECTOR BETWEEN WEB AND MOBILE

The rapid evolution of mobile phones in recent years has brought great impact on people's daily lives. From simple calling devices they have become powerful platforms with features such as access to the internet, barcodes scanners, embedded sensors (e.g., GPS, accelerometer, gyroscope, light, video, microphone, etc.). Also, they can communicate with external sensors through network protocols such as Bluetooth. Apart from the evolution of mobiles, Web, also, currently lives its evolutionary phase with HTML5. Thus, several Web applications have been mobilized and many Web sites are now responsive. HTML5 is the definite “software glue” which fills the gap between mobiles and Web and becomes the key to any future development. All these mobile features can be used combinatorial to the advanced features of HTML5 for enabling valuable distributed participatory and opportunistic sensing applications. As we review in the second chapter the crowdsensing applications can be extended to many fields of the daily life, namely: transportation and civil infrastructure monitoring, environmental monitoring, health and fitness, urban sensing and traffic monitoring.

In this chapter, we will introduce web and mobile technologies, which are capable to deal with data of IoT [1], [2]. At first we discuss about HTML5 APIs which are deal with sensor and hardware integration and then later we will analyze every new element that HTML present in the fifth revision. Later in the chapter we comment about others web technologies which help us to construct the architecture of our platform. Also, we will discuss about concepts, protocols, libraries and APIs such as Meteor, Nodejs, Mongoddb, Google geocoding APIv3, Web Audio API, Web sockets, Geolocation API, Sencha ExtJS 4 (Visualization charts) and much more.

3.2 HTML5 OVERVIEW

HTML5 APIs that enable pervasive and adaptive multimedia over the web

HTML5 [144-147] is a programming language used for describing the layout and presenting the contents of Web pages. HTML5 is cooperation between the Web Hypertext

Application Technology Working Group (WHATWG) and the World Wide Web Consortium (W3C). In the early years, HTML had limited potentialities and was designed just for describing static web content. But with the course of time, the language has been dramatically evolved, offering real challenges to developers. Its latest version, HTML5 has completely changed the status in the IT firmament with the breaking through technologies it introduces. For example video files do not require any more external plugins like flash in order to be played back in a browser, as HTML5 with its <video> tag embeds such functionalities, as play, stop/pause, move back/forward, directly into the body of the language. Essentially, HTML5 is not anymore a simple language for describing web pages, but a combination of HTML, CSS and many Javascript APIs, which makes it a powerful platform with rich capabilities. In the following, the attention will be paid on these Javascript APIs, available with the 5th edition of HTML that can enable the provision of pervasive, ubiquitous and adaptive web applications to end users. Because with these APIs web sites and web applications are offered an insight to the personal context and ambient environment of the end users, static or mobile, enabling capabilities for personalized, customized and anticipatory service provisioning. The fifth version of HTML includes a wide variety of new characteristics such as graphical and media support without using external plugins, quick response time and consistency in web applications, device independence. Below we highlight the new capabilities of the HTML5 platform:

1. **Support for media without plugins:** Adding and handling graphical content on the web with new elements such as <video>, <audio>, <canvas> and integration of scalable vector graphics (SVG) content.
2. **Media and Real-Time Communications:** WebRTC, streaming media.
3. **Interoperability:** One of the main objectives of HTML5 has been to ensure interoperability and consistent functionality across browsers.
4. **Better handling of client-side data:** With support for different storage in HTML5 viz. sessionStorage and localStorage, it is possible to store structured data temporarily thereby warding off cookies.
5. **Improved Semantics:** The main aim of HTML5 is to introduce a markup that is easily readable by humans and understood by computers and devices.
6. **Security and Privacy:** identity, crypto, multi-factor authentication, privacy protection

7. **Device Interaction:** access to hardware and sensors such as geolocation, orientation, bluetooth, NFC, vibration, etc.
8. **Application Lifecycle:** HTML5 can be used to write web applications that even work offline and become available after the user refresh the page. Also, support push, geofencing and sync.

3.2.1 HTML5 APIs

Sensors and Hardware Integration

Modern mobile devices of smartphone and tablet style embed a rich variety of sensors. In order applications to have access to the data from sensors, normally a middleware tool needs to mediate to facilitate the communication. Special-purpose Application Programming Interfaces (APIs) expose sensor data to the mobile web developers. Table 3.1 summarizes some critical sensors and hardware APIs from W3C as they are included in [136].

Table 3.1: Sensors and Hardware APIS

	Feature	Working Group	Maturity	Current Implementation
1	Geolocation	Geolocation	W3C Recommendations	Widely deployed
2	Motion sensors		Last Call Working Drafts	Well deployed
3	Battery status	Device APIs	Candidate Recommendations	Very limited
4	Proximity sensors		Candidate Recommendations	Very limited
5	Ambient light sensor		Candidate Recommendations	Very limited
6	Networking information		Discontinued	Very limited

	API			
7	Camera & Microphone streams	Device APIs and Web Real-Time Communications	Working Drafts	Limited but growing

3.2.1.1 GEOLOCATION API

Geolocation API [53] allows the client-side device to provide geographic positioning information to javascript web applications. Geolocation API offers to mobile users the possibility to share their location with anyone they trust (individuals or web sites). The Geolocation API returns the geographical coordinates of the user device in a geodetic datum, that is in the form of latitude and longitude. In order them to be understandable or valuable for the end user, later this information must be translated to something like a city or street name or the name of a favorite area (e.g. my mother's place, my office, my gym), since the user understands better the civil datum. Online services such as Google and Bing maps, can undertake such transformations. Apart from latitude and longitude, the geolocation API can also return additional information such as the user's altitude, heading and speed and the altitude accuracy.

There isn't a standard positioning technology with which the Geolocation API finds the user location; it rather uses any available method offered by the device. Hence, there is no guarantee about the accuracy of the returned data. Several positioning technologies can be used and combined to this end including:

- **Global Positioning System (GPS):** a very promising way to return the location when outdoors but with very bad results when indoors. GPS takes the signal from many GPS satellites (and the serving cellular network in the form of A-GPS) to calculate the final location. Its disadvantages include the draining of devices from power and the need for enough time to yield results especially at start up.
- **Wi-Fi:** the location is found by triangulating the location estimations from several Wi-Fi hot spots. The accuracy depends on the density of the access points in the surroundings of the user.
- **GSM and UMTS cell IDs:** similar to the WiFi method the results are estimated by triangulating the location measurements from the serving cellular network's towers near the user. The accuracy depends on the density of the base stations in the surroundings.
- **IP address:** It is considered an unreliable means to return the location of the user due to the fact that it is greatly based on the ISP provider, which could be far away from the physical address.

Taking into account that user location is included in the personal and sensitive information and with respect to the user privacy; the Geolocation API requires the confirmation of the user before sharing his or her location with any application or individual.

How it works

The Geolocation API uses the navigator.geolocation property which returns a Geolocation object. The Geolocation object contains the location of the user represented in latitude and longitude coordinates. In particular, the Geolocation object has 3 methods, the `getCurrentPosition`, the `watchPosition` and the `clearWatch`, as Table 3.2 depicts.

Table 3.2: Geolocation API methods

Method	Description
<code>getCurrentPosition</code>	Gets the user location once
<code>watchPosition</code>	Keeps polling for user position and returns an associated ID.
<code>clearWatch</code>	Stops polling for user position

The `getCurrentPosition` gets the user location only once. Its successful callback returns a *Position object* as argument, and its fail callback returns an error. The Position object consists of several properties, the most critical of which are the `coords.latitude` and `coords.longitude`. Table 3.3 enlists the properties of a Position object.

Table 3.3: Position object properties

Property	Unit
<code>coords.latitude</code>	degrees
<code>coords.longitude</code>	degrees
<code>coords.altitude</code>	meters
<code>coords.accuracy</code>	meters

coords.altitudeAccuracy	meters
coords.heading	degrees clockwise
coords.speed	meters/second
timestamp	like the Date object

The **watchPosition** method gets user location as the user moves. This function is iteratively called as the user location changes so the web application is always provided with updated location information. It returns the same arguments as the `getCurrentPosition`. In addition, it returns an ID number to ensure the uniqueness of the user. The third method of Geolocation object, **clearWatch**, uses this ID number to stop polling the user location.

Browser Support

Currently the Geolocation API is supported by the Mozilla Firefox, Chrome, Opera, Internet Explorer, Safari, and the native Android and Blackberry browsers.

3.2.1.2 DEVICE ORIENTATION API

The most modern mobile devices are equipped with plenty of motion sensors. Motion sensors include accelerometers, gyroscopes and compasses. The HTML5 Device Orientation API [61] provides developers with access to underlying motion sensors and to associated data from the orientation and movement of the device. The API specifies the events listed in Table 3.4.

Table 3.4: Orientation API methods

Event	Description
DeviceOrientation	It fires whenever a significant change in orientation occurs
CompassNeedsCalibration	It fires when the user agent determines that a compass used to obtain user orientation needs calibration

DeviceMotion	It fires regularly with information about the motion of the device
--------------	--

The *DeviceOrientation* event exposes all the orientation changes. It returns the properties, alpha, beta and gamma that are explained in Table 3.5.

Table 3.5: Device Orientation Event Properties

Property	Description
Alpha	Denotes the direction the device is facing according to the compass
Beta	Denotes the angle in degrees the device is tilted front-to-back
Gamma	Denotes the angle in degrees the device is tilted left-to-right

Figure 3-1 visualizes the values of the *DeviceOrientation* event.

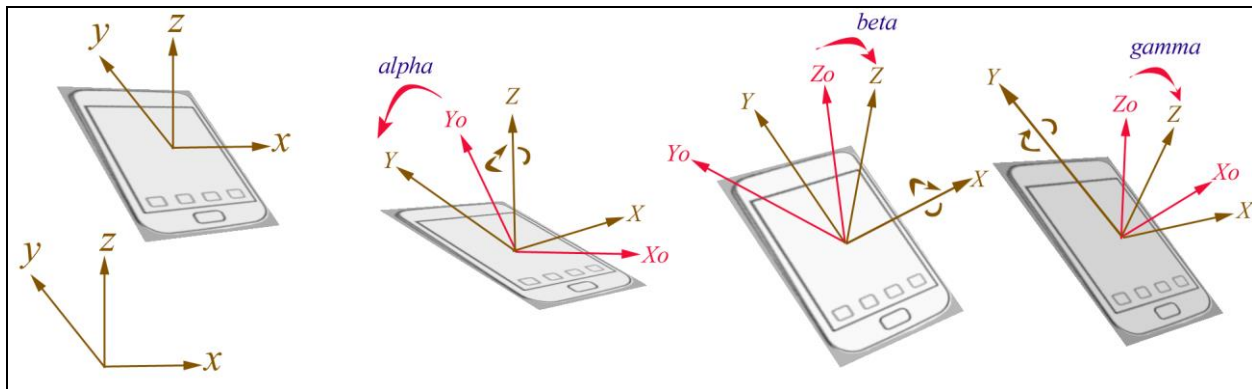


Figure 3-1: The DeviceOrientation event properties

The *DeviceMotion* event exposes the acceleration and rotation rates of the device. It takes 4 properties, acceleration, accelerationIncludingGravity, rotationRate and interval. Table 3.6 details on them.

Table 3.6: Device Motion Event Properties

Property	Description
Acceleration	provides acceleration data, in m/s ² for each of the x, y, and z axes
AcceleratonIncludingGravity	provides same data as above, but with effects due to the Earth's gravity included
RotationRate	provides the rate of rotation in deg/s around each of the axes
Interval	time in milliseconds between samples

Browser Support

Currently the Device Orientation API is supported by the Mozilla Firefox, Chrome, Opera, Internet Explorer, Safari, and the native Android and Blackberry browsers.

3.2.1.3 BATTERY STATUS API

Normally, mobile devices have a life time of 9 to 10 hours of active usage before battery drains all its energy. The Battery Status API [55] can make the web applications smarter and energy-friendly. It uses the navigator.battery property to create a BatteryObject. Table 3.7 depicts the basic properties of a BatteryObject.

Table 3.7: Battery Status API Properties

Properties	Description
navigator.battery.level	Obtains the charging level of the battery. Returns a value between 0 and 1.
navigator.battery.charging	Informs if the device is currently charging or not. Returns true or false.

navigator.battery.chargingTime	The remaining time in seconds until charging level reaches 100%.
navigator.battery.dischargingTime	The time in seconds before the battery is completely discharged and the device shuts down.

Browser Support

Currently the Battery Status API is only supported by Mozilla Firefox.

3.2.1.4 PROXIMITY SENSOR API

The Proximity Sensor API detects the distance between the mobile device and the user or an object. The API [56] has the two methods of Table 3.8 to work with:

Table 3.8: Proximity Sensor API methods

Method	Description
Device proximity	measures the distance in centimeters
User proximity	informs the user if an object is near or not

The **deviceProximityEvent** property measures the distance in centimeters but the minimum and maximum distance a proximity sensor supports varies. Usually the value ranges from 0 to 10cm. The **userProximityEvent** property returns a Boolean value (true or false) which inform the user if an object is near or far.

Browser Support

Currently the Proximity Sensor API is only supported by Mozilla Firefox.

3.2.1.5 AMBIENT LIGHT SENSOR API

The Ambient Light Sensor API [57] senses the environment of the device to provide web applications with the measured luminosity in lux units. The values range from 0 to 10000 lux. Obviously

an embedded Light Sensor is required. When a light change is detected, the *DeviceLightEvent* provides applications with the updated value of luminosity. A second interface is the *LightLevelEvent*, which provides less accurate characterization of the ambient light. In particular it categorizes the light level into 3 categories. The first category is the “Dim” environments with light values below 50 lux, the second is the “Normal” environments with values ranging from 50 to 10000 lux and the third is the “Bright” environments with light values greater than 10000 lux. Table 9 includes this categorization.

Table 3.9: LightLevelEvent values

Light Level	Ambient Characterization
<50 lux	Dim Environment
50 ~ 10000 lux	Normal
> 10000 lux	Bright

Browser Support

Currently the Ambient Light Sensor API is only supported by Mozilla Firefox.

3.2.1.6 MEDIA CAPTURE AND STREAMS API

The Media Capture and Streams API (or GetUserMedia API) [62] [138] offers to web applications access to multimedia streams, such as video and audio, from local devices (webcam or microphone) through a browser. It then capitalizes on the HTML5 <video> and <audio> elements to play them back. In terms of user privacy, the Media Capture and Streams API behave similar to the Geolocation API. Whenever an application attempts to access the local media devices the browser asks the user for his permission. The revolutionary with this API is that access to the local media devices takes place without any need for plugins installation. Below is an example of how access to camera and microphone can be achieved.

```

If (navigator.getUserMedia)
{
  navigator.getUserMedia({audio: true, video: true}, successCallback, errorCallback);
}

```

The method `navigator.getUserMedia()` takes three arguments: *constraints*, *successCallback* and *errorCallback*. In *constraints* the type of media that will be accessed (video, audio or both) are defined. In *successCallback* the success scenario is defined. Hence, when video and audio are loaded; the captured media streams are put in a `<video>` element identifying the object of the *LocalMediaStream* via a BLOB URL [157].

```
function successCallback (MediaStream) {  
    video.src = window.URL.createObjectURL(MediaStream);  
}
```

In *errorCallback* three cases for failure can occur, in general: i) Permission denied by the user, ii) No media tracks are found, iii) The browser does not support the specific constrain. Hence an alternative solution should be offered to the user:

```
function errorCallback (e) {  
    video.src = 'fallbackvideo.webm';  
}
```

Browser Support

Currently the Media Capture and Streams API is supported by the Mozilla Firefox, Chrome, Opera and Blackberry browsers.

3.2.1.7 PERFORMANCE CHARACTERISTICS

Performance is a critical part in web applications development. To this end, several tools for web applications performance optimization can be found. Especially for mobile web applications, their limitations in terms of battery, networking, memory and CPU need to be taken into consideration during development and provision. HTML5 provides tools capable to measure various aspects of mobile resources. These include the Network Information API, the Resource Timing API, the High Resolution Time API and the User Timing API.

3.2.1.8 NETWORK INFORMATION API

The Network Information API [58] measures the available bandwidth and offers to the developers the ability to adapt web media elements, as images, videos, audios and fonts, accordingly for a better user experience with multimedia content. The navigator.connection method provides an object with the two properties, bandwidth and metered, of Table 3.10.

Table 3.10: Network Information API properties

Properties	Description
Bandwidth	It estimates the current bandwidth. Zero means an offline user.
Metered	A connection is characterized as “metered” when the user's connection is subject to a limitation from the Internet Service Provider. Hence the web applications are requested to be careful with the bandwidth usage. It returns a Boolean value.

Despite its obvious value, the development of Network Information API has been discontinued currently. In the following, the Resource timing API will be presented, which is in a stable status of development and can provide developers with similar information.

Browser Support

When discontinued the Network Information API was only supported by the native Android and Blackberry browsers.

3.2.1.9 RESOURCE TIMING API

A factor that affects user experience in web application is latency. Network latency refers to the delay data packets experience as they are transferred from the sender to the receiver. The Resource Timing API [66] measures the time needed for various resources of a web page to be loaded in a browser. During the development phase of a mobile web application, such debugging information can let the developer think and incorporate into his application design various intelligent methodologies for adapting

appropriately the application and offering greater user experience to poor execution environments. Some of the networking information that can be retrieved by this API is the time for redirect, cache, access to DNS, opening a TCP session, transmitting a request and receiving a response. Figure 2 illustrates the measured times from the Resource Timing API.

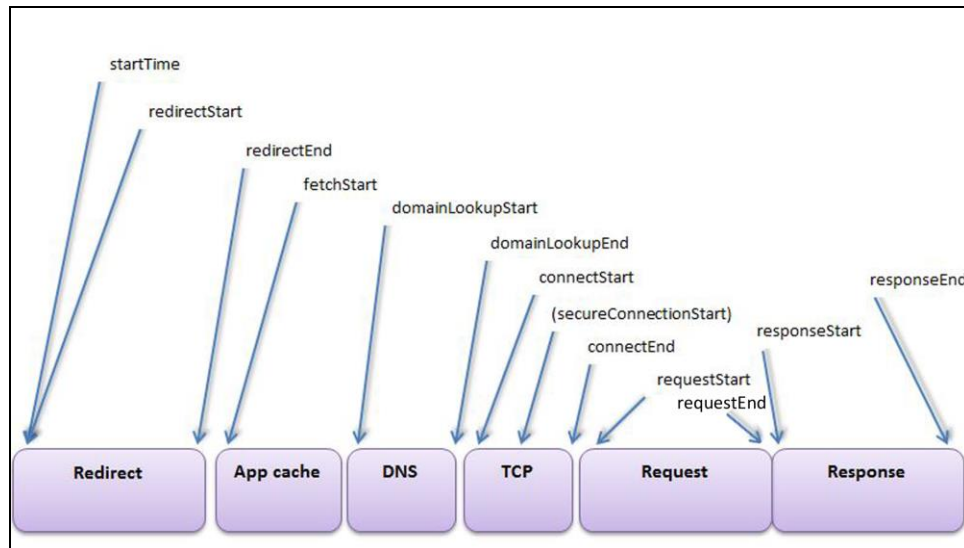


Figure 3-2: Timing attributes of the Resource Timing API

Apart from time-sensitive networking information, the Resource Timing API can also measure the delays from other critical components for most web applications such as the performance of various third part assets including Javascript libraries, social widgets and CSS frameworks.

Browser Support

Currently the Resource Timing API is supported by the Chrome and Internet Explorer browsers.

3.2.1.10 THE HIGH RESOLUTION TIME AND THE USER TIMING API

The High Resolution Time API [60] measures the internal time performance of a web application in terms of function calls, interface callbacks, variable assignments, etc. As it is described in W3C it is “*a JavaScript interface that provides the current time in sub-millisecond resolution and such that it is not subject to system clock skews or adjustments.*” The API uses the *Performance interface* which exposes only the method *now()*. *Now()* returns a *DOMHighResTimeStamp* object which represents the current time in milliseconds. *Performance.now()* is similar to the Javascript function *Date.now()* with the difference that the former is far more accurate with a precision to a thousandth of a millisecond.

Similar to the High Resolution Time API, the User Timing API [61] also measures the internal performance of web applications. However, the former API has the drawback that if someone needs to measure the performance in different files of an application he has to insert global variables. The User Timing API facilitates such situations by offering to web developers access to high precision timestamps. In particular it implements the *PerformanceEntry* interface, which includes the *PerformanceMark* and *PerformanceMeasure* interfaces. The *mark()* method stores a timestamp which is available across all files of the application. In addition, with *mark()* someone can differentiate the time it starts measuring by the time the “*mark*” timestamp was posed in the application. Then, with the *measure()* method the time between marks can be measured. Table 11 includes the properties of the User Timing API and elaborates further on them.

Table 3.11: User Timing API properties

Property	Description
Name	A name correlated to the Mark or the Measure
startTime	For Mark() it is a TimeStamp For Measure() it is the TimeStamp of the start Mark of the Measure.
Duration	For Mark() it is zero. For Measure() it is the time elapsed between marks.
mark(name)	Stores a TimeStamp with the associated name
clearMarks([name])	Deletes the stored Marks.
measure(name[, mark1[, mark2]])	Stores the time elapsed between two Marks with the provided name.
clearMeasures([name])	Deletes the stored Measures.

Browser Support

Currently the High Resolution Time and the User Timing APIs are supported by the Chrome, Internet Explorer and the native Android browsers.

3.2.1.11 VIBRATION API

Vibration is synonym to Mobile devices. Hereafter, this capability is extended to the Web applications. The Vibration API [63] uses the *navigator.vibrate* method to enable vibrations. *Navigator.vibrate()* takes as argument a number in milliseconds denoting the duration of the vibration, e.g. *navigator.vibrate(500)*; Alternatively, an array of delays defining a pattern of vibration can be given. For example, with *navigator.vibrate([200, 400, 600])* the pattern will cause the device to vibrate for 200 ms, be still for 400 ms, and then vibrate again for 600 ms.

Browser Support

Currently the Vibration API is supported by the Mozilla Firefox, Chrome, and the native Android and Blackberry browsers.

3.2.1.12 FULL SCREEN API

The Fullscreen API [64] allows web developers to tag elements in web applications (or documents) for viewing in full-screen mode. Instead of using the F11 keyboard button this can be enabled by simply pressing above the web element. Exiting from the full-screen mode is achieved by clicking again on the web element. It is mostly used with images and videos.

Browser Support

Currently the Full Screen API is supported by the Mozilla Firefox, Chrome, Internet Explorer and the native Blackberry browsers.

3.2.1.13 PAGE VISIBILITY API

The Page Visibility API [65] provides the web developer with the capability to offer better user experience by applying the visibility or not state into their web application. An application will have different behavior when it is visible and different when it is hidden. It can be used to adapt the usage of resources to the need of the Web application, for instance by reducing network activity when a page is

minimized. Furthermore, it can have a great impact on the mobile devices because they can save energy from battery.

The API has two properties the *Hidden* and the *VisibilityState*. The *Hidden* is a Boolean property, which with True indicates a hidden document and with False a visible one. The *VisibilityState* has four options: "hidden", "visible", "prerender", and "unloaded". Table 3.12 elaborates on them. Finally, the API contains the *visibilityChange* event which fires whenever the visibility state of a Document changes.

Table 3.12: VisibilityState values

Values	Description
Hidden	The document is totally hidden.
Visible	The document is visible.
Prerender	Optional. The document contained by the top level browser tab is loaded off-screen and is not visible.
Unloaded	Optional. The User Agent is to unload the document contained by the top level browser tab.

Browser Support

Currently the Page Visibility API is supported by the Mozilla Firefox, Chrome, Opera, Internet Explorer, Safari, and the native Android and Blackberry browsers.

3.2.1.14 WEB WORKERS

Javascript is a single-threaded language, which means that javascript-based web applications can handle only one script at a time. This can do web applications unresponsive offering a poor user experience. HTML5 attempts to solve this problem by introducing the Web workers API [66]. Web Workers are scripts running in the background without influencing the main UI, offering, thus, a concurrent execution.

Web Workers are executed in different files, which are called from the main script with a *postMessage()* method. The *postMessage()* method accepts either a string or a JSON object as argument.

The Web Workers handles the Messages from the main page with the *onmessage* handler. The data between the main thread and the Worker are copied and not shared minimizing, thus, the time required for a large file to be transferred. The Web Workers can transfer different type of objects such as Files, BLOBs, ArrayBuffers, and JSON objects [15]. However, Web Workers have currently some limitations. Table 3.13 enlists the objects a Worker can and cannot have access.

Table 3.13: Web Workers capabilities and limitations

DO have access	DO NOT have access
The navigator object	The DOM (it's not thread-safe)
The location object (read-only)	The “window” object
XMLHttpRequest	The “document” object
setTimeout()/clearTimeout() setInterval()/clearInterval()	The “parent” object
The Application Cache	The local name space. Web Workers will not work if a web page is being served directly from the local filesystem (using file://)
Importing external scripts using the <i>importScripts()</i> method	All Worker scripts must be served from the same domain and protocol as the script that creates the worker.
Spawning other web workers	

3.2.1.15 WEB STORAGE

HTML5 introduces new methodologies for the storage of data by Web applications. In the past, data were stored exclusively in the web server but with the Web storage capability [67] this has changed.

The data can now be stored in the mobile device and be later synchronized with the server. This provides, at first, offline usage of data and, at second, improves the application performance. There are two main web storage types: Local Storage and Session Storage. The Local Storage stores data for ever without to be lost. On the other hand, the Session Storage stores data only for the duration of a session.

The values on the local Storage are stored as key-value pairs, hence whenever the application wants to access the values it must use the respective key. Only strings can be stored via the Storage API. In most browsers, storing of different data types results in automatic conversion of them to a string format. Conversion into JSON, however, allows for effective storage of JavaScript objects. Local Storage has an upper limit at the storage space it can access, which varies from browser to browser (5 Megabytes for Google Chrome, Mozilla Firefox and Opera), but it is definitely far better than the 4 Kilobytes of cookies.

3.2.1.16 WEB INTENTS

Web Intents [68] is a framework for service discovery and inter-application communication enabling information sharing between web applications. Most users use specific web applications such as facebook, twitter, google+, viber, skype, dropbox, each with its own API. With Intents, web developers do not need to deal with each of them because web Intents unify access to them incorporating them to the logic of a web application. Web Intents enable to several different applications to work together. Via Intents several actions can be performed, such as sharing, editing, viewing, picking, subscribing or saving of a document. Also, web Intents are mobile friendly and can be used in a variety of mobile operating systems such as Android and iOS.

Web Intents function similar to web services. Essentially, it is transfer of the well-known publish-subscribe model to the web. The life cycle of a Web Intent consists of 5 stages: registration, invocation, selection, delivery and response [68]. At first, a User Agent is being informed by a web application that it is able to handle Intents for specific actions. Whenever a client page sends to the agent such Intents for handling, the User Agent selects this application as appropriate for the Intent. Then the User Agent delivers the Intent to the application, which responds by passing data to the client page.

A Web Intent object can contain several parameters. The most important parameters are *action* and *type* that cannot be empty. The *action* parameter indicates the action type of the Intent, for example edit. The *type* parameter indicates the type of the data payload. *Data* parameter is optional and it refers to any data including transferables. The next example demonstrates an application submitting Intent for an appropriate service to share a list of images designated via urls.

```
var intent = new Intent({"action":"http://webintents.org/share",
    "type":" text/uri-list ",
    "data":getPublicURIForImage(...)});
```

3.2.1.17 WEBSOCKETS

The WebSocket protocol [134, 135], provides a bidirectional communication channel using a single TCP connection. It has been designed for implementation in both browsers and web-servers and its API [159] has been standardized by the W3C. WebSocket connections are established over the regular TCP port 80, which ensures that the system can run behind firewalls. The life-cycle of a WebSocket session is depicted in Figure 3. At first the client, a browser that supports the WebSocket protocol, requests a server to establish a WebSocket connection. The positive response from the server denotes the start of such a connection. The connection remains open for the whole session, until any endpoint requests its release with the specified procedure. As a WebSocket remains active; WebSocket frames can be transferred from server to client and vice versa with no preceding request. In an indicative implementation the WebSockets server may also host the service logic of a web-application, which might be responsible for maintaining a listing of the clients with active WebSockets and session management. Although logical separated, the web server, the service logic and the WebSockets server could run on the same physical entity.

As it is depicted in Figure 3.3 the HTTP is initially used to establish the WebSocket connection between a compatible browser and a server. When the bidirectional connection is established, the application may transfer data to the server using this dedicated socket. The added value appears in the case that a server needs to push data to the client, which can take place asynchronously. Prior to websockets, a client-sided mechanism to request any updated data from the server should be used, a technique that consumes unnecessarily the network resources.

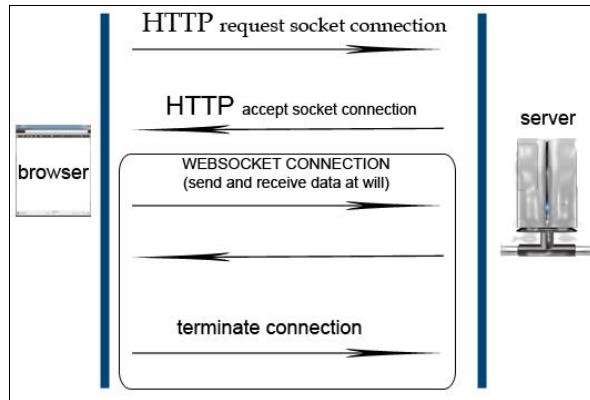


Figure 3-3 4:Life-cycle of a WebSocket session

As it is described in [140], a WebSocket session consists of two parts, the handshake and the data transfer. The handshake is based on HTTP signaling, extended with websocket headers. The HTTP GET message, as it is depicted below, holds information about the websocket server (*host*) and the originating web application (*origin*), as well as the *upgrade* header that indicates a request for switching to a websocket connection. The *sec-websocket* values include information related to security, subprotocols and versioning respectively.

```

GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13

```

The HTTP response from the server acknowledges the transfer of the connection to a websocket and includes the required headers for security verification.

```

HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: chat

```

Once the handshake procedure has been completed successfully, all data between a client and a server are transferred via the WebSocket in the form of WebSocket frames. According to the specification each such frame is defined with three mandatory values; the *opcode*, which defines the type of the payload, the *payload length*, and the *payload data*. The *opcode* has a length of 4 bits, while the size for *payload length* and *data* may vary depending on the data value. The WebSocket protocol supports frame fragmentations to allow streaming of unknown size messages, and multiplexing. For the former, the server chooses a reasonable size buffer, while multiplexing allows several WebSockets within the same origin and host to share a single logical channel for various utilizations. The payload data can be carried both as Text (UTF-8 text) or Binary data. Table 3.14 elaborates on the various opcode values.

Table 3.14: Opcode values for WebSockets

Opcode	Meaning
0	Continuation Frame
1	Text Frame
2	Binary Frame
8	Connection Close
9	Ping
10	Pong

A continuation Frame indicates a frame that is part of a fragmented frame. The Text and Binary Frames indicate whether the payload holds text or binary data respectively. The Connection Close value starts the signaling for closing the WebSocket, and the Ping / Pong Frames are used for the keep-alive process.

```
<!DOCTYPE HTML>
<html>
<head>
<script type="text/javascript">

var socket = new WebSocket('ws://game.example.com:12010/updates');

socket.onopen = function () {
  setInterval(function() {
```

```

    if (socket.bufferedAmount == 0)
        socket.send(getUpdateData());
    }, 50);
};

socket.onmessage = function (evt)
{
    var received_msg = evt.data;
    console.log("received message: "+received_msg);
};

socket.onclose = function()
{
    console.log("Connection is closed...");
};

</script>
</head>
</html>

```

Frameworks are already distributed to provide interoperability and easy use of the WebSocket API. The abstract above illustrates the required javascript code for opening a WebSocket. Once a JavaScript object named *socket* has been created, giving the type “ws://” URL of the server, there are four event handlers to be used as listed in Table 3.15. The *open* and *close* events occur whenever a connection is established or closed, respectively. The *error* event occurs when there is any kind of error in the communication. The *message* handler is triggered whenever data are received via the WebSocket. All parties implementing the WebSocket interface should support these handlers. The *send(data)* method is used for transmitting data via the socket. Data can be of type String, BLOB or ArrayBuffer. Any invocation of this method increases the *bufferedAmount* attribute by the length of the *Data* in bytes.

Table 3.15: Event handlers for WebSockets

Event	Event Handler
Open	Socket.onopen
Message	Socket.onmessage
Error	Socket.onerror
Close	Socket.onclose

3.2.1.18 WEB AUDIO API

HISTORY OF WEB AUDIO

The first contact between Web and Audio was from <bgsound> tag [30], where web pages play background music when a user opens them. Only Internet explorer had this feature [<https://developer.mozilla.org/en-US/docs/Web/HTML/Element/bgsound>]. The successor of <bgsound> was flash, which offer cross-browser functionality. Despite that, Flash had an important drawback, the requirement of plug-in. So, all moderns' browsers move on to another tag, <audio> tag [26]. Audio tag embedded sound content in web pages and can support many audio sources. It had some limitations and for that reason it can't support demanding applications like games. Table 3.16 enlists some of the most important limitations.

Table 3.16: Audio tag limitations

No accurate timing controls
Very low limit for the number of sounds played at once
No way to reliably pre-buffer a sound
No ability to apply real-time effects
No way to analyze sounds

WEB AUDIO API OVERVIEW

Interactive applications, games, advanced music synthesis applications and visualizations need a strong API without the limitations of <audio> tag. This API is the Web Audio API [28,29] which is a high-level versatile JavaScript API for controlling, processing and synthesizing audio. It provides multiple functionalities such as adding multiple audio sources, adding effect [128] and visualizes audio. The Web Audio API is an HTML5 API which has direct access to the audio hardware and has built around the concept of an audio context. An Audio context is a routed graph which contains directed audio nodes from a source (audio file or microphone) to the destination (speakers). Figure 3.4 Shows a simple Audio context where the source and destination node are connected without any distribution between them.

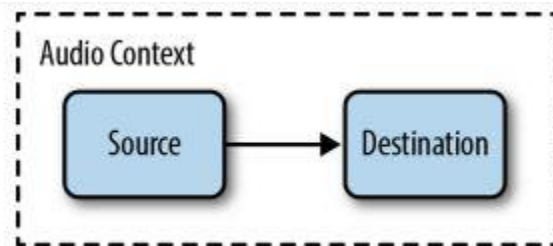


Figure 3-4: Audio context

More complex Audio contexts contain multiple sources and many intermediate nodes for synthesis and analysis before reach to the final destination. Below is a code example on how to create an intermediate node between the source and destination.

```
// Create the source.
Var source = context.createBufferSource();

//Create the gain node.
Var gain = context.createGain();

// Connect source to filter, filter to destination
Source.connect(gain);
Gain.connect(context.destination);
```

In table 3.17 presents source [26], destination and gain nodes that are available for the Web Audio API.

Table 3.17: Types of Web Audio Nodes

Source nodes	Sound sources such as audio buffers, live audio inputs, <audio>tags, oscillators, and JS processors
Modification nodes	Filters, convolvers, panners, JS processors, etc.
Analysis nodes	Analyzers and JS processors
Destination nodes	Audio outputs and offline processing buffers

Browser Support

Currently the web audio API is supported by the Mozilla Firefox, Chrome, Opera, Safari, and for the mobile version is supported for Ios Safari and Chrome for android.

3.2.1.19 FILE API

File API [129] provide a standard way for web applications to interact with local files. The file API includes 3 interfaces for accessing files from a local filesystem.

- 1) File - an individual file
- 2) FileList – an array-like sequence of File Objects
- 3) Blob – allow to slice a file into byte ranges

The conjunction with the above interfaces can be used to asynchronously read a file through familiar Javascript event handling. File API has many functionalities with XMLHttpRequest. Some of them are monitoring of reading, catch errors, completion of upload progress.

- The selection of the files can be with input form or by drag and drop.
- The reading of the files happens with result attribute, after the load finish. FileReader include four options to read a file, asynchronously:

- 1) FileReader.readAsBinaryString (Blob|File) - The result property will contain the file/blob's data as a binary string. Every byte is represented by an integer in the range [0..255].

- 2) `FileReader.readAsText(Blob|File, opt_encoding)` - The result property will contain the file/blob's data as a text string. By default the string is decoded as 'UTF-8'. Use the optional encoding parameter can specify a different format.
- 3) `FileReader.readAsDataURL(Blob|File)` - The result property will contain the file/blob's data encoded as a data URL.
- 4) `FileReader.readAsArrayBuffer(Blob|File)` - The result property will contain the file/blob's data as an ArrayBuffer object.

3.2.1.20 APPLICATIONS SCENARIOS

The combination of the above APIs can provide very interesting pervasive applications, which keep undiminished the interest of the user. The rapid growth of smartphones, with all these advanced sensing and networking capabilities, creates the appropriate infrastructure for HTML5 to thrive.

An interesting such scenario would be a web application that uses the light sensor of a smartphone to detect the luminance in the usage environment and then to dynamically adapt the background of a web page to enhance readability. A variant of this application would be another one that changes the graphics of a webpage based on instantaneous network information. Another intriguing scenario would be a web application using WebSockets to send notifications to its subscribers and upon their receipt a phone to vibrate. In the same context, crowd sensing application is now easy to be implemented. Sensors APIs mine periodically the data from the phone and WebSockets send the valuable information to the central server to analyze and visualize the data creating useful maps.

3.3 GOOGLE SERVICES

3.3.1 GOOGLE MAPS AND GOOGLE MAP API V3

Google Maps [147] is a web mapping service and technology for desktop and mobile devices that provided by Google. The capabilities of the specific platform are satellite imagery, street maps and street view perspectives. Google update the database on a regular basis with images for street maps but it is not in real time. Google Maps can easily be integrated into a

third-party website via the Google Maps API [148]. The API provides the developer with many tools like conclude a marker in a map, add multiple maps or add virtual radius. Also, Google Map API can intergrade with others Google services such as Google Geocoding API.

3.3.2 GOOGLE GEOCODING API v3

Geocoding or forward geocoding is the procedure of translating addresses (e.g. Delaporta, Heraklion 71409, Greece) into geographic coordinates (latitude 35.3191579 and longitude 25.1483078) [23], [24]. The opposite procedure of translating geographic coordinates into an address (in a human readable-way) is called reverse geocoding [27]. Figure 3.5 shows google geocoding API in action. Google geocoding API v3 is included in google maps web services and implement both geocoding and reverse geocoding process. The user can have access to Google Geocoding API via an HTTP request. An API key is necessary to request the service.

Although Google geocoding API is a free web service; it is subject to two limitations from a single IP address. The first limitation is referred to the maximum number of requests per day (2500 geocode requests per 24 hours). The second limitation is referred to the maximum number of requests per second (5 geocode requests per second).

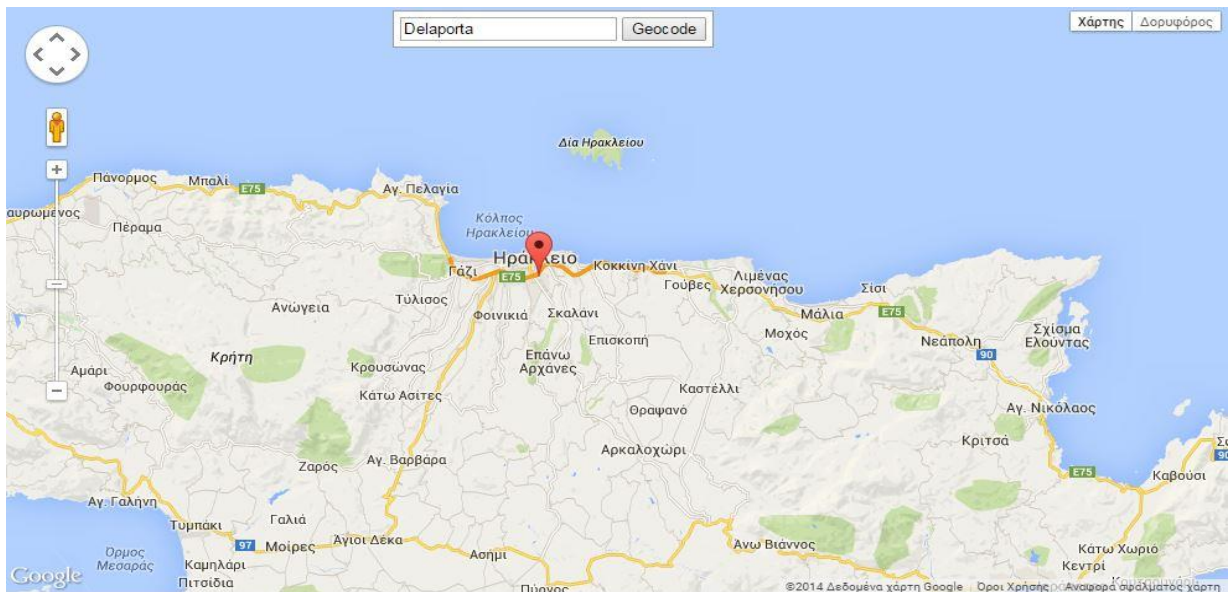


Figure 3-5: Geocoding services translate an address into geographic coordinates and display a marker in a map.

The output of a geocoding API request [25] is either a JavaScript Object Notation (JSON) object or an XML file. Both outputs have two root elements, one for the status, which contains request's metadata for information about the result of the request (successful or not), and one for the result which contains address' and geometry's information. Tables 3.18 and 3.19 elaborate further on the results of a geocoding API response.

Table 3.18: Google Geocoding API output: Status codes

OK	This code shows a successful response.
ZERO_RESULTS	This code shows a successful response but it doesn't return any results, probably by a non-existing address.
OVER_QUERY_LIMIT	This code shows that you have overcome your limits.
REQUEST_DENIED	When you get this code your request is denied. The reason is not specified but the most common issue is that the sensor parameter is missing.
INVALID_REQUEST	This code shows that there was an error with the request.

Table 3.19: Google Geocoding API output: Result codes

types	An array that contain the type of the location. The type of the location is country or locality.
formatted_address	A string that contain the address in a human-readable way. A paradeigm could be Delaporta, Heraklion 71409, Greece. It contains few kind of address information such as name of street, name of city, zip code and country name.
address_components	An array that contain the parts of the location which are listed in formatted_address.
Geometry	An object which contain several information such as position of the location, viewport, bounds and location_type

The results codes indicate two important arrays for the geocoding process: `types[]` and `address_components[]`. The first array (`types`) includes street address, country or political entity, while the second (`address_component`) include the type of each part of the address. Table 3.20 shows the returning types of the above two arrays.

Table 3.20: Returning types [] and address_components [].

street_address	Shows a street address
route	Shows a named route
Intersection	Shows an intersection of two roads
political	Shows a political entity.
country	Shows the national political entity (Highest order type)
administrative_area_level_1	Shows a first-order civil entity below the country level.
administrative_area_level_2	Shows a second-order civil entity below the country level.
administrative_area_level_3	Shows a third-order civil entity below the country level.
administrative_area_level_4	Shows a fourth-order civil entity below the country level.
administrative_area_level_5	Shows a fifth-order civil entity below the country level.
colloquial_area	Shows a commonly-used alternative name for the entity.
locality	Shows an incorporated city
sublocality	Shows a first-order civil entity below a locality.
sublocality_level_1 to sublocality_level_5.	Each sublocality level is a civil entity.
neighborhood	Shows a named neighborhood
Premise	Shows a named location, usually a building or collection of buildings with a common name

Subpremise	Shows a first-order entity below a named location, usually a singular building within a collection of buildings with a common name
postal_code	Shows a postal code
natural_feature	Shows a prominent natural feature
Airport	Shows an airport
park	Shows a named park
point_of_interest	Shows a named point of interest. Typically, these "POI"s are prominent local entities that don't easily fit in another category, such as "Empire State Building" or "Statue of Liberty."

3.3.3 GEO-FENCE

A geo-fence [141] is a virtual boundary around a real-world geographical area which defines a point of interest. Geo-fence could be generated dynamically by giving the capability to the end user to select a point of interest or static by predefined a set of boundaries, like a field boundaries. With the dynamically capability the user can mark the desired region and see the recovered information. It is usually used in location-aware applications sending notifications to the users about a zone violation. For example a child location service, can notify the parents about the status and the location of their child. In some companies, geofencing is used by the Human Resource department to monitor employees working in special locations especially those doing field works. Using a geofencing tool, an employee is allowed to log his attendance using a GPS-enabled device when within a designated perimeter. Google maps and android in mobile technologies contain the functionality of geofence [142].

3.4 METEOR PLATFORM

Our platform has been set up on meteor web platform [163, <http://meteor.academy/>]. Meteor is a real-time Javascript web application framework which is written on top of Node.js

and concludes various packages like MongoDB and jQuery. Meteor consider as the future of the web because it combine a full stack isomorphic system using the same language (Javascript) in both frontend and backend [144]. Also the same APIs can be used for mobile applications along with Cordova. This means that you write your code once and run everywhere. It customizes them to communicate seamlessly with one another via Distributed Data Protocol and a built-in publish-subscribe pattern [145]. The Distributed Data Protocol is a websocket-based protocol, allowing to the user to deliver live updates as data changes. Meteor is designed to work most with one database, MongoDB. MongoDB is a JSON-style, document-based NoSQL database built for flexibility and scalability. The client side act with the same way as the server side, having access to the database from Mongoose. Meteors customized these packages into smart packages and offer to the developer great capabilities such as: automatically real-time, database access from the client (mongoose), latency compensation, doesn't need to write Ajax and there is not any DOM manipulation. Meteor allows us to easily create apps without having to worry about the backend plumbing needed to set this all up. The web application runs both on the client (browser's JavaScript engine) and on the server (node.js). The result of all this is a platform that manages to be very powerful and very simple by abstracting away many of the usual hassles and difficulties of web app development. Figure 3.6 show Meteor environment and separate the components between server and client. In the next sub-chapter we will analyze every component separately.

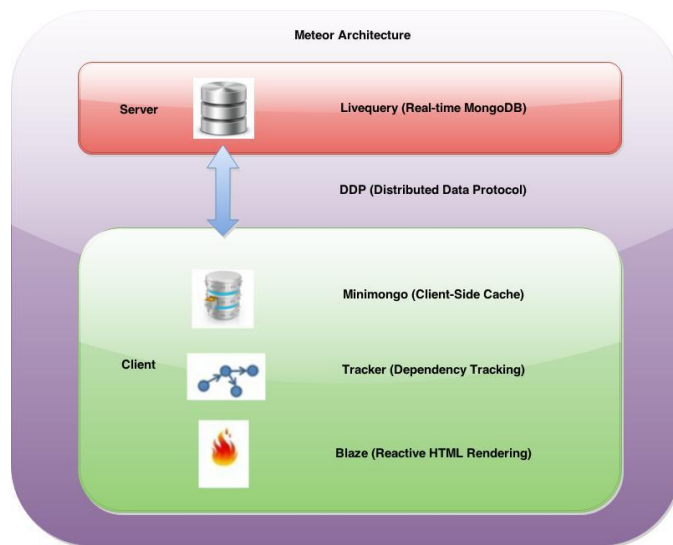


Figure 3-6: Meteor architecture

3.4.1 NODEJS

Meteor is written on top of Node.js. The server side code is running on a Node.js server. Node.js server is a cross-platform (OS X, Microsoft windows, linux and FreeBSD) runtime environment which is used for server-side and networking applications [52]. It is an open source environment and the Javascript is the language for writing node.js application. The architecture is event-driven providing a non-blocking I/O API that optimizes scalability and throughput of applications. Node.js running up to Google V8 Javascript engine containing a large scale of modules written in Javascript. It contains a built-in library which allows applications to act as a Web server. Companies such SAP, LinkedIn, Microsoft and Yahoo are some popular clients who use node.js as a server-side platform. A combination of Node.js, a document DB such as MongoDB or CouchDB and JSON offers a unified JavaScript development stack.

3.4.2 DDP

DDP (Distributed Data Protocol) is running over a websocket and it used for a bidirectional communication between the client and the server. DDP can fetch structured data from a server and received live updates when that data changes. Actually DDP can query the database from client side (database is common for client and server), sending the results to the client and push the changes to client whenever there is a change. The actual implementation of DDP is JSON messages over a websocket connection. DDP protocol is responsible for real time publish – subscribe communication between the server and the client and for remote procedure calls (methods). Currently the ddp package is the same for client and server but the future plan is to be separated.

3.4.3 BLAZE REACTIVE VIRTUAL DOM ENGINE

Blaze is a library which creates live-update user interfaces (<https://www.meteor.com/blaze>). Similar frameworks with Blaze are Angular, Backbone, Ember and Knockout. It has a very simple way to write into a HTML file. In Blaze the developer just writes HTML templates with Spacebars and the Blaze itself handle the updates. The system which is behind the scene and it is responsible for the Blaze simplicity is called Tracker

(<https://github.com/meteor/meteor/wiki/Tracker-Manual>), (<https://www.meteor.com/tracker>). It is a lightweight system which is the middleware for the reactive programming. Reactive programming means that Blaze can automatically infer the data dependencies of arbitrary JavaScript code, allowing Blaze to automatically set up callbacks to detect changes to the template's data sources, recompute any values affected by the change, and patch the DOM in place with the update. Blaze use Spacebars as template processor which has been inspired from Handlebars (<http://handlebarsjs.com/> <http://code.tutsplus.com/tutorials/an-introduction-to-handlebars--net-27761>). Spacebars generate dynamically your HTML page, saving you time from performing manual updates.

3.4.4 PRINCIPLES AND METHODS OF METEOR

Below we will describe in a few words which are the main principles of Meteor. Also, we will refer in the main methods of the platform in table 3.21.

- 1) Full Stack Reactivity. In Meteor, realtime is the default. All layers, from database to template, update themselves automatically when necessary.
- 2) Data on the Wire. Meteor doesn't send HTML over the network. The server sends data and lets the client render it.
- 3) One Language. Meteor lets you write both the client and the server parts of your application in JavaScript.
- 4) Database Everywhere. You can use the same methods to access your database from the client or the server.
- 5) Latency Compensation. On the client, Meteor prefetches data and simulates models to make it look like server method calls return instantly.
- 6) Embrace the Ecosystem. Meteor is open source and integrates with existing open source tools and frameworks.
- 7) Simplicity Equals Productivity. The best way to make something seem simple is to have it actually *be* simple. Meteor's main functionality has clean, classically beautiful APIs.

Table 3.21: Meteor platform methods

Templates	Create views that update automatically when data changes.
Session	Store temporary data for the user interface.
Collections	Store persistent data.
Accounts	Let users log in with passwords, Facebook, Google, GitHub, etc.
Methods	Call server functions from the client.
Publish /Subscribe	Sync part of your data to the client.
Environment	Control when and where your code runs.

3.5 JSON

JSON or else JavaScript Object Notation [146] is a way to store information in an organized, easy-to-access manner. The outcome of JSON is a human-readable file with a structured manner. It is used to transmit data objects between a server and a web application, as an alternative to XML. JSON has many similarities with XML, like both are in plain text, are in a human readable format, use hierarchical dome and can be fetched by an HttpRequest. On the other hand, JSON superior from XML to: it doesn't use end tag, is shorter, is quicker to read and write and can use arrays. Also, the biggest difference is that XML has to be parsed with an XML parser but JSON can be parsed via a standard JavaScript function.

A common use of JSON is to read data from a web server, and display the data in a web page. JSON is derived from JavaScript language but it is considered as an independent language. There is a big list of programming languages that are parsing and generating JSON data except of JavaScript. In table 3.22 we will present the basic types of JSON object.

Table 3.22: JSON basic types

JSON basic types	Description
------------------	-------------

Number	A signed decimal number that may contain a fractional part and may use exponential E notation.
String	A sequence of zero or more Unicode characters.
Boolean	Either of the values true or false.
Array	An ordered list of zero or more values, each of which may be of any type. Arrays use square bracket notation with elements being comma-separated.
Object	An unordered collection of name/value pairs where the names (also called keys) are strings.
Null	An empty value, using the word null.

3.6 BSON

BSON, is [149] based on JSON objects and the “B” is referred to Binary data. It is a data interchange format that is used mainly as data storage. MongoDB database use BSON for data storage and network transfer. The main characteristics of BSON object is that it is lightweight (keeping spatial overhead to a minimum), traversable (design to travel easily) and efficient (very quick encoding and decoding) [150].

3.7 GEOJSON

GeoJSON [153] is an open standard format for encoding a variety of geographic data structures and is based to JavaScript Object Notation. It include points (therefore addresses and locations), line strings (therefore streets, highways and boundaries), polygons (countries, provinces, tracts of land), and multi-part collections of these types. The GeoJSON format differs from other GIS standards [152] in that it was written and is maintained not by a formal standards organization, but by an Internet working group of developers. Below is an example of GeoJSON. Also, there are online tools to create, share and validate GeoJSON data [154].

```
{
  "type": "Feature",
```

```
“geometry”: {  
  “type”: “Point”,  
  “coordinates”: [125.6, 10.1]  
},  
“properties”: {  
  “name”: “Heraklion”  
}
```

3.8 EXT JS FRAMEWORK

Ext JS [155, 157] is a Javascript application framework suitable for interactive web applications. Ajax, DHTML and DOM scripting are some of the techniques that Ext JS use to present graphics in web pages. ExtJS Charts are used to present data visually, usually showing the relationship between different parts of the data. Ext JS excels for master / detail form-heavy applications and no other HTML application framework is going to come close to Ext JS from a feature perspective. Below we highlight some elements of the Ext JS framework which make it distinguish from others frameworks:

- 1) Data management: It contain a clever way to break up the data access responsibilities into Model, Proxy, Store, Reader and Writer components. The typed properties translate JSON to model objects.
- 2) UI Framework: It contains a fleet of components for the user interfaces such as Toolbars, Panels, Buttons, Icons, Cards, Carousel, Tabs.
- 3) Graphing: Charts and drawing with SVG.
- 4) Offline capabilities: It uses SessionStorage and LocalStorage that can be used to make a web application work in an offline state.

3.9 X3D & X3DOM

The World Wide Web Consortium (W3C) and the Web Hypertext Application Technology Working Group (WHATWG) cooperated to include into the emerged HTML5

specification an updated way for the presentation of 3D content in web browsers. HTML5 [159] [160] through its canvas element enabled the presentation of X3D graphics from web pages without requirement for any plugin. That is, all web browsers compatible with WebGL can render interactive 3D graphics. Definitely, the advent of HTML5 associated with other web technologies, such as X3Dom [161], can convert web browsers to 3D friendly multi-platforms with lots of capabilities. X3Dom is a proposed syntax model that translates X3D files to WebGL graphics. X3Dom is considered to be the state of the art technology for the visualization of X3D graphics on the canvas of a web page. With X3Dom's implementation as a JavaScript library, 3D graphics can be readily presented to browsers, without plugins, using only WebGL and JavaScript. More specifically, X3Dom acts as a broker between DOM (front end), which embodies X3D objects, and X3D runtime (backend). The X3Dom library is responsible for the conversion of X3Dom descriptions into the appropriate format for the X3D backend that renders the 3D objects. Hence, X3Dom library and X3D runtime can update the 3D scene whenever any change occurs. The X3Dom library is compatible with a variety of operating systems, devices and web browsers [12].

CHAPTER 4: PLATFORM ARCHITECTURE

4.1 CROWDSENSING PLATFORM INTRODUCTION

In this chapter, we will discuss about the architecture of our crowdsensing platform. Based to the bibliography it is the first platform that use HTML5 APIs to deliver real-time sensor data to the users. Our platform is a modern, real time web application system for gathering sensor data (e.g. noise intensity, luminous intensity and connection type information) and display them in real-time. Apart from displaying the client data, it analyzes them in the server side and offer them back to the community. Visitor will be able to see the sensor data from a separate page. The collected sensor data will be shown in a fully-interactive world map and in nice informative, responsive charts. Also, it offers the data to the community via web services API. The sensor data could then be used for further purposes such as for making surveys, scientific researching or doing experiments. We will start by naming its components and then will explain every component separately. We will also cover the interconnection between components of the architecture. Figure 4.1 shows the architecture of the platform, which is based on the multi-tier paradigm [6]. In software engineering, multi-tier or n-tier architecture is a client-server architecture in which, the presentation, the application processing and the data management are logically separated processes. The most usual "multi-tier architecture" is the three-tier architecture. The tiers can be called layers and it is not need to be in physically different machines.

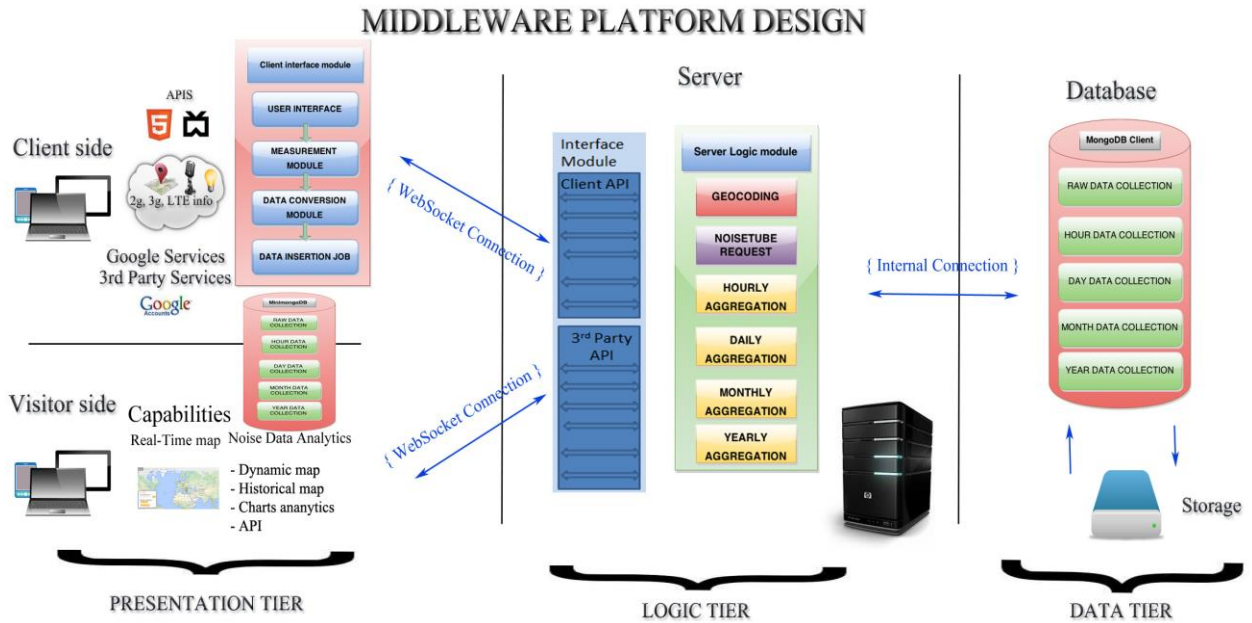


Figure 4-1: Middleware platform architecture

4.2 COMPONENTS

Our platform consists of 3 major components: The client component, which is responsible for gathering the sensor data, the server component, which contains the service logic of the platform, and the 3rd party component, which offers a variety of services to end users. Finally, there is the database component which is responsible for storing the sensor data.

4.2.1 CLIENT COMPONENT

The client component is responsible for the implementation of the HTML5 APIs for the communication with the device sensors, the storing of sensor data to the database and their transfer to the server. The implementation of the client component is based on the Meteor framework, which acts as an application server between the other components. In particular, it undertakes to transfer the information quickly and safely to the server via a distributed data protocol. Probably, the most critical job for Meteor is the synchronization of the data in both client and server side. The client component is the main source of sensor data in our architecture. The other source is provided by the noisetube API and is implemented at server side. The main

source of data is noise data provided by the microphone device of clients by using the Web Audio API and exploiting `getUserMedia()` with a gain node analyzer. Figure 4.2 shows the procedure of capturing the noise data. Also, the application can collect luminosity data by the Ambient light sensor API. The user can gather simultaneously data from both sensors, or switch between noise and light sensor from a button switcher. Figure 4.3 shows the client switcher between the two sensors. The switcher checkbox enables or disables each of the APIs. Both sensor data are combined with location information by Geolocation API. Except from the sensor data, the client side can collect connection type information, such as under 2G, 3G or wifi connection, via the Network information API. Ambient light sensor API and Network information API are in experimental or draft stage currently and browser support is very limited. Both APIs are supported only by mobile Firefox for Android and iOS.

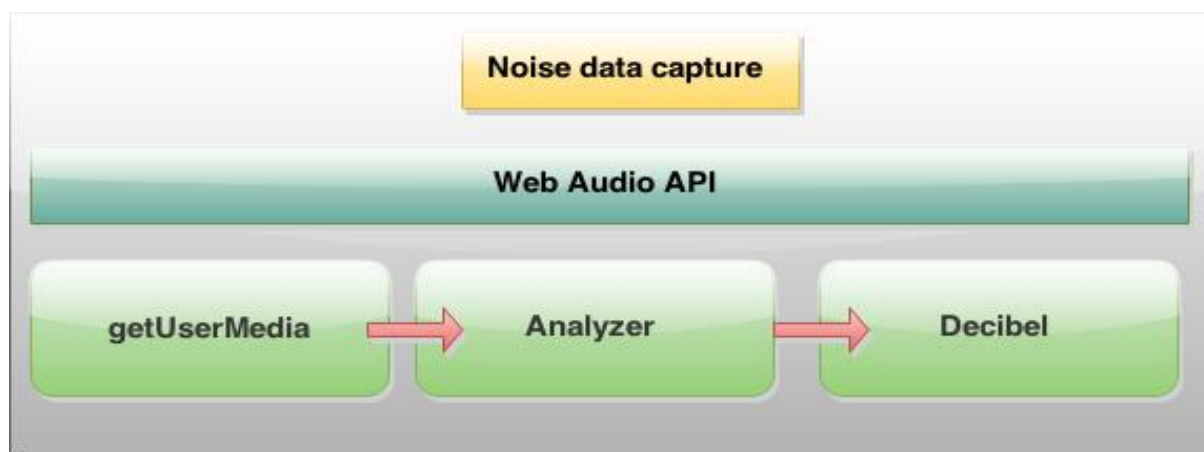


Figure 4-2: Noise data capture

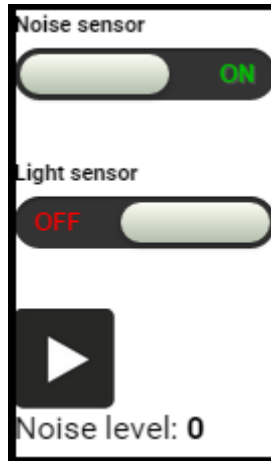


Figure 4-3: Sensor switcher

Data measurement and data conversion module

Noise data are the main source of the sensor data and the most of our services are referred to them. When the user presses the main button at the client interface, automatically activates the media capture API or else `getUserMedia()`. Instantly the user gains access to the microphone of the device. Before gaining access to the microphone, browser will through an info bar to call `getUserMedia()`, which gives users the option to accept or deny access to their microphone. Figure 3.8 shows the permission dialog from Chrome.

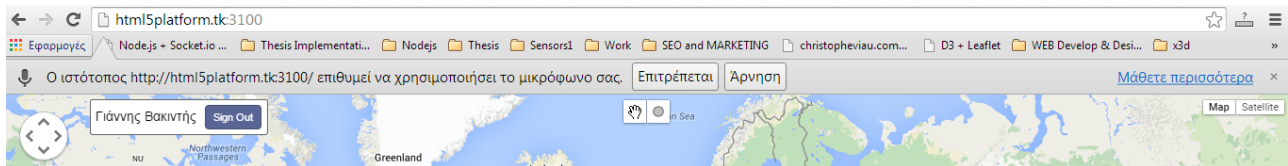


Figure 4-4: Permission dialog

If the user presses the allow button then the procedure of gathering the noise data is started. In the meanwhile we connect the audio stream from `getUserMedia()` as an audio source in an Audio context of Web Audio API. The audio buffer will be connected to the decibel analyzer to transform the PCM samples into decibel values. Below is provided the connection to the Web Audio API and the analyzer code. We have named this procedure as data conversion module because it converts the noise sample from the client device into decibels.

```
//Connect Media stream as a source of Web Audio API
navigator.getUserMedia({audio:true}, function (stream) {
```



```

    src = ac.createMediaStreamSource(stream);
    src.connect(analyser);
}
//Create a decibel analyser
analyser.getTimeDomainData(timeData);
while (i<fftsize) {
    float = (timeData[i++] / 0x80) - 1;
    total += ( float * float );
}
rms = Math.sqrt (total / fftSize);
db = 20 * (Math.log(rms) / Math.log(10) );
db = Math.max(-48, Math.min(db,0));
percentage = 100 + (db * 2.083);
noise = (noise * 99 + percentage) / 100;
Session.set('noiseLevel',Math.round(noise));

```

The second source of sensor data is the ambient light sensor API. Ambient light sensor detects the light level in "lux" units. In Client application, we are using the method `addEventListener` for adding the event “devicelight”. The devicelight event will be fired when fresh data is available from a light sensor. Then it gets the data and save them in the collection.

Geolocation API is coming next. When the user enables the data gathering button simultaneously enables the geolocation API. Geolocation API will through an InfoBar just like the `getUserMedia()`. Then the user will have the option to accept or deny the access to his location information. Upon “accept”, the client starts to retrieve the coordinates of the user.

Also, client application can detect the connection type of the user (e.g. wifi, cellular, bluetooth) using the Network information API. The Network information API is an experimental technology because its specification has not been stabilized yet. It consists of the `NetworkInformation()` method and a single property to the Navigator interface:

Navigator.connection. The connection object contains the property *type*, which returns the user agent’s connection type. The following table shows the values that property type can have.

Table 4.1: Network Information API

Network Information API Values
Bluetooth
Cellular
Ethernet
None
Wifi
Other
Unknown

Data insertion module

We execute a function every 1000 millisecond to send the data in the server/database. We named this procedure as data insertion job. Data insertion job call the server method “*pushSensorData*” and send the data from client to the database. The data are stored in the database such as JSON documents. Server method calculates the current hour and inserts it in the sensor document. Next panel show the structure of the document. It contains coordinates, noise\light level values (embedded subdocument), user ID and timestamp (hour). Every minute the server job updates the data collection with country and locality values by coordinates.

```

Data model (sensors collection):
{
  "_id": mongo ID,
  "country": geolocation data,
  "locality": geolocation data,
  "place": geolocation data,
  "hour": unix timestamp in hours,
  "lat": latitude,
  "lng": longitude,
  "user": id of the user ("guest if data not from client"),
  "sensors":
    {
      "noise": noise data,
    }
}

```

```
    "light": luminosity data
  }
}
```

Distance calculator

We have embedded in the client a real-time noise measure map, for visualizing his sensor data, and a distance calculator. With the later, the user can select from a variety of choices (e.g., whole world, 10000km, 2500km, 500km, 100km, 25km, 5km, 1km) and find live users in proximity to him. Using the Geolocation API [53] we can calculate the distance between the specific live user and other live users within the specified radius and return to the user the results on his live map. The formula takes the coordinates of the starting point of the user and compares them periodically towards his current position. To get the starting coordinates we call `getCurrentPosition()` and then the API asynchronously follows the user's current location and saves it for later use. This call executes only once when the user grants the application with access to geolocation API and then follows the moves of the user. Finally, the formula [38, 39] calculates the distance between the current user and the other live users. Below is a sample of the code.

```
var R = 6371; //km
var dLat = (lat2-lat1).toRad();
var dLon = (lon2-lon1).toRad();
var a = Math.sin(dLat/2) * Math.sin(dLat/2) +
        Math.cos(lat1.toRad()) * Math.cos(lat2.toRad()) *
        Math.sin(dLon/2) * Math.sin(dLon/2);
var c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1-a));
var d = R * c;
return d;
```

We use the haversine formula to calculate the distance between users [40]. Haversine formula calculates the great-circle distance between two points – that is, the shortest distance over the earth's surface – giving an 'as-the-crow-flies' distance between the points (ignoring any hills they fly over, of course!).

$$a = \sin^2(\Delta\phi/2) + \cos \phi_1 \cdot \cos \phi_2 \cdot \sin^2(\Delta\lambda/2)$$

Haversine formula: $c = 2 \cdot \text{atan2}(\sqrt{a}, \sqrt{1-a})$
 $d = R \cdot c$

ϕ is latitude, λ is longitude, R is earth's radius (mean radius = 6,371km);
 where note that angles need to be in radians to pass to trig functions!

```
var R = 6371; // km
var  $\phi$ 1 = lat1.toRadians();
var  $\phi$ 2 = lat2.toRadians();
var  $\Delta\phi$  = (lat2-lat1).toRadians();
var  $\Delta\lambda$  = (lon2-lon1).toRadians();

var a = Math.sin( $\Delta\phi$ /2) * Math.sin( $\Delta\phi$ /2) +
        Math.cos( $\phi$ 1) * Math.cos( $\phi$ 2) *
        Math.sin( $\Delta\lambda$ /2) * Math.sin( $\Delta\lambda$ /2);
var c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1-a));
```

JavaScript: `var d = R * c;`

Geofence functionality

As we described in chapter 3, geo-fence is a virtual boundary around a real-world geographical area which defines a point of interest. It can be generated dynamically from the end user or statically by predefining a set of boundaries. The user can “draw” a point of interest and see the corresponding information from the markers. The selection of the point is made in two steps: 1) at first there is a selection from the database of the items which are located inside the area of the circle. 2) Then it checks if each item is in the distance range from the center of the circle. Later there is a real time reverse geocoding procedure that translates the coordinates of the center into the corresponding location. The geofence calculate all the markers that are located inside the rectangle area and shows the average noise value of them in a dynamic panel.

Real-time live users

The user interface of the client application is an interactive Google Map which is essentially a real-time application. It shows all the live users that use the client application. Figure 4.5 shows a screenshot from the client application. This service displays points with noise and geolocation information over the google map. After the page is rendered, we initialize the google map. Also we initialize the panel overlays for info panels and controls.

After that, we initialize the geofence controls on the map. Geofence enables the figure drawing on the map and retrieving information about the part of the map selected by these

polygons. When the map is loaded we fire the update function which updates and displays the data on the map. The update function detects the coordinates of the part of the map which is now displayed for the user. By these coordinates and the current filter settings, it retrieves the data from server by the server method *getLiveUsers*. This method fetches the current users from the database collection of LiveUsers. The data, aggregated by locality, are displayed as map points over the Google Map. Each point is colored according to its noise level.

```
pinColor=colorByNoise(res[k].sensors.noise);
```

Also we put the data from database into the point variable.

```
markers[key] = new google.maps.Marker({  
  position: new google.maps.LatLng(res[k].lat, res[k].lng),  
  map: map,  
  icon: pinImage,  
  shadow: pinShadow,  
  data: res[k] //data from the server  
});
```

Then, on each point on the map we attach the onclick event listener, which provides the info about point into the info window.

```
google.maps.event.addListener(markers[key],"click",function(){ });
```

The update function fires every 1000ms to make the map a real-time one. Upon an update, we only redraw the new points and delete the expired ones, so we avoid redrawing the whole point array.

```
var key = res[k].lat + ',' + res[k].lng + ',' + pinColor;  
//by coordinates and color  
if (!markers[key]) { //adding new points }  
for (var k in markers) {  
  if (!found[k]) {  
    markers[k].setMap(null);  
    delete markers[k];  
  }  
}
```

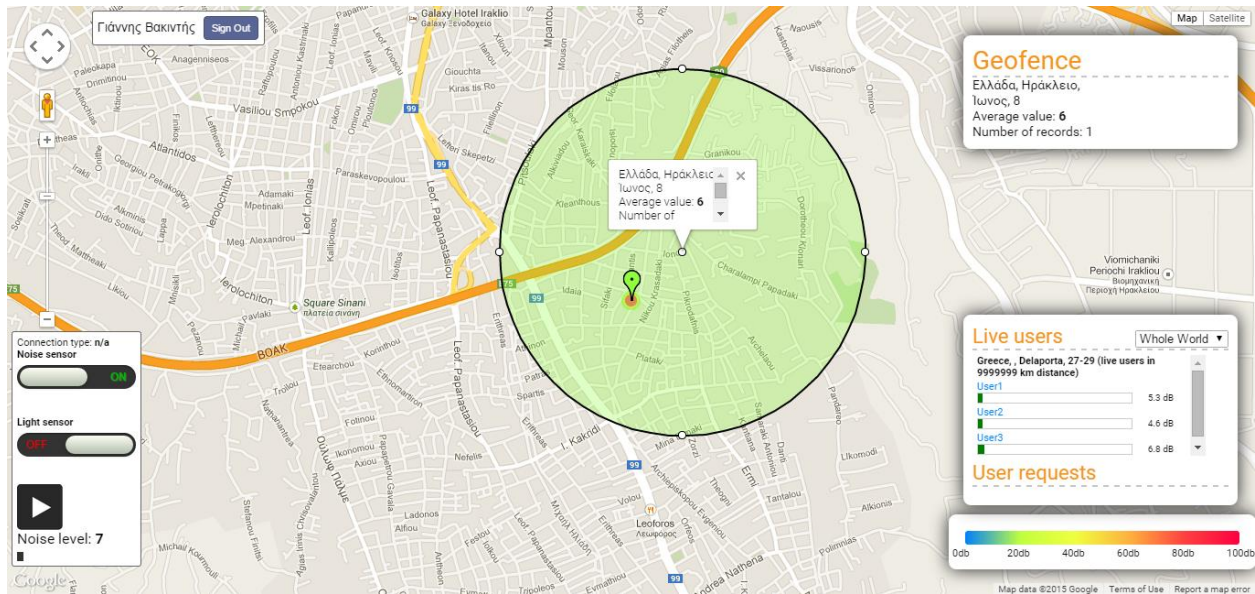


Figure 4-5: Client application screenshot

When the user grants his permission for noise and location retrieval, then we start to gather raw sensor data and a local data analytics function transforms the data to useful information. Later, we send them in the database. Simultaneously, we fire another function which is responsible to save the data to the history collection for client page visualization purposes. The LiveUser function takes the id from the login button and shows the name of the user with his current noise value and geolocation information in the main panel. To ensure the anonymity of the user we replace the real name of the user with a fake name, for example “User 1”. Figure 4.6 shows the main panel of the client component application. Also, from the main panel, the user can see visitor requests from the visitor page. Visitors’ requests are requests made from the visitor page and are based in location information. Visitor requests are stored in a specific collection from where we can fetch and display them in the main panel.

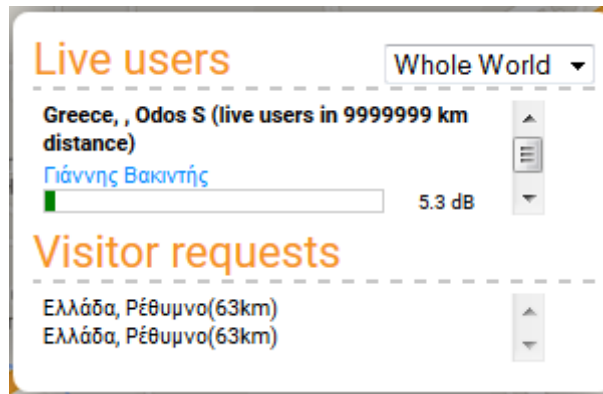


Figure 4-6: Main panel

4.2.2 SERVER COMPONENT

The server component is responsible to store the semantic information from the client side and to distribute it to various collections for visualization purposes. The main server jobs are: *Reverse geocoding, time aggregation and NoiseTube data request*. With reverse geocoding the server updates user data by translating coordinates to country, locality and place information using Google services. Time aggregation job is to manipulate and insert sensor data into collections in order to achieve a spatiotemporal visualization in the third party's page. Finally, there is the NoiseTube data request job with which the server fetches data from the NoiseTube API to create more crowd-full visualization charts.

Reverse Geocoding job

The reverse geocoding job is one of the main jobs in the server logic. In the beginning of the process it selects one record from the Sensor collection which has no geolocation data. This is any record with geographical coordinates but without their translation to an address from the real world (i.e. it has no country field). If there are no records without data we just restart this function in 1000ms. If there is data without country and locality information, we are executing the server method "geocode" which is part of geocoding package [168]. The geocoding package is a ready package which is integrated into the server side of the meteor application. The Geocode method searches by country name to find null fields. Then pass the coordinates in the

geocode method as arguments and wait the response from the procedure. After retrieving the geocode data we update the record in the Sensor collection with the following data:

```
{
  country: res.country,
  locality: res.locality,
  place: res.place //where res – result of geocoding from google reverse geocoding service
}
```

When the record is updated we start the function again in 1ms. Apart from the server geocoding job there is another similar job which is executed at client side and we name it as “*on demand*” geocoding job. When the user wishes to search by location name or information in geofence functionality we fire this job. Figures 4.7 and 4.8 show statistics and traffic reports with the overall request usage of Google APIs. In the google console interface we can see the total requests and the requests per day. Also, it shows the daily quota per day at a rate view.

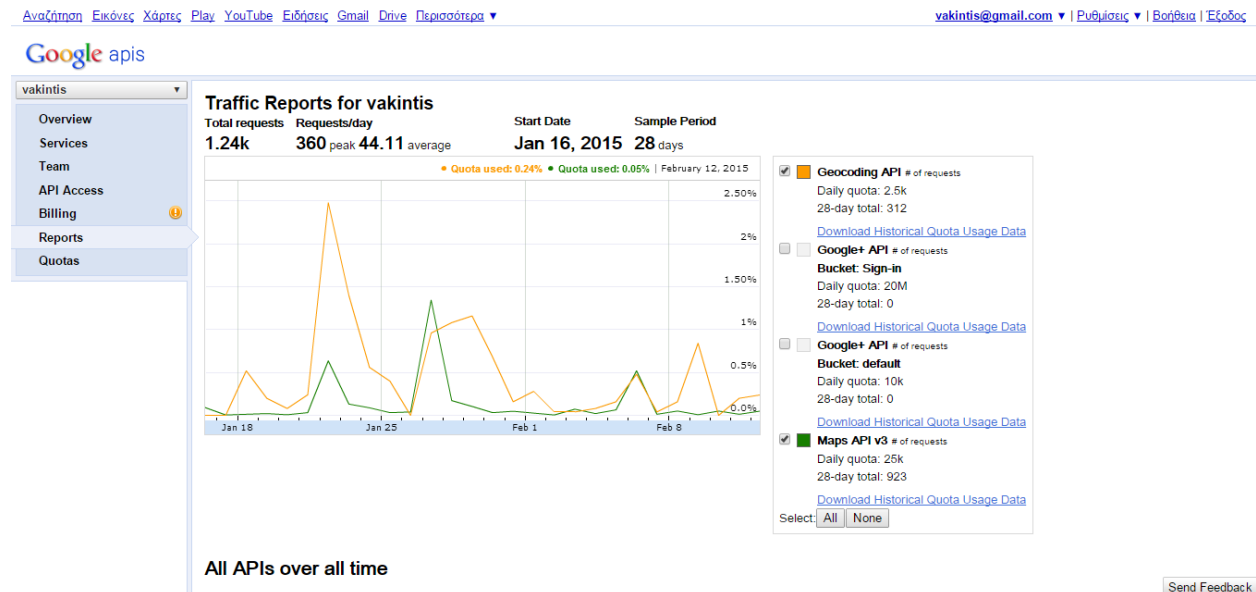


Figure 4-7: Geocoding statistics from Google geocoding API

All APIs over all time

Demographics	Top Countries		
Country / Territory >	Requests	% Requests	Country
Usage	13.21k	51.32%	US
	6.2k	24.08%	RU
Methods	3.14k	12.20%	VN
Users	2.37k	9.20%	GR
Errors	626	2.43%	NL
Referers	91	0.35%	IN
	56	0.22%	MA
	16	0.06%	PH
	10	0.04%	IE
	9	0.03%	KE

Figure 4-8: Averages geocoding statistics from Google geocoding API

NoiseTube request job

The need for a big volume of data, especially for noise data, for visualization and testing of our platform, leads us to search for another source of data. A project with large amount of noise data has been built from the France laboratory of Sony Computer [169] in corporation with VUB BrusSense group <http://www.brussense.be/> and is called NoiseTube [170]. NoiseTube is a research project which started in 2008 and its main scope is to measure noise pollution levels in many cities. It contains a NoiseTube mobile app which turns the mobile devices into noise sensors enabling citizens to measure their daily exposure to sound. Every user can create a collective map of noise pollution by sharing his geolocalized measurements. The noise is measured in dB(A) and the mobile application can be used by iOS, Android and Java ME-based mobile devices.

The NoiseTube research project contains a data collective API and an API which gives access to its database. When the user creates the account then he is given an API key to authorize his identity and allow him to send and retrieve data from the database. The data collective API

lets you create “a track” that when its processing is finished it is published and shared through the website. The second API is the data commons API that lets you retrieve raw collected data from the server. The user can make a query giving his API key along with some other criteria such as location, dbmax and dbmin.

The NoiseTube data request job is actually a parser which requests NoiseTube data from the Noise Tube server and stores them in the sensor collection. This service requests the noise data from the NoiseTube server, converts them to have the same structure as records in our data base and puts them in the sensor collection. We make an http request with the following structure:

```
var a = queue.pop();
ax = a[0], ay = a[1], bx = a[2], by = a[3];
var box = ax+', '+ay+', '+bx+', '+by;
var query = date+', '+box;

var url = 'http://www.noisetube.net/api/search.json?'+
'key=de0d36c700cdfb0412a9cc7a429c788baecaa822&'+
'max=100&since='+date+'&box='+box;

Meteor.http.get(url, function (err, res) {
  if (err) {
    console.log(err);
  } else {
    noisetube.insert({query: query, response: res.data});
    response = res.data;
    g(response, 90000);
  }
});
```

When we initialize the system, it gets the current date and makes request to NoiseTube server to get the last 1 hour data. After that we check if data exists and put it in a queue. The last

part is to convert the data from NoiseTube format to the sensor collection format and put it into database. The format of NoiseTube data are the following:

```
{ "lat":45.75514437370546,"lng":4.8425658840205115,"made_at":"2015-03-03T18:37:20Z","loudness":"61.0","user":null }
```

We change the format of the above document to the following format:

```
Data model (sensors collection):
{
  "_id": mongo ID,
  "country": geolocation data,
  "locality": geolocation data,
  "place": geolocation data,
  "hour": unix timestamp in hours,
  "lat": latitude,
  "lng": longitude,
  "user": id of the user ("guest if data not from client"),
  "sensors":
    {
      "noise": noise data,
      "light": luminocity data
    }
}
```

Id: MongoDB place a unique id for every document when is going to be stored in a collection.

Country-locality-place: Those 3 fields remain empty in this stage. Later when we store them in the sensor collection, geocode method is responsible to transform the lat and lng field into a geographical name.

Lat-lng: Those 2 fields remain like we take it without any intervention.

User: The field user takes the value "guest" because it comes from an external source.

Sensors: The field Sensors is an embedded field with contains two others field: Noise and Light. The field noise will take the value of the corresponding field "loudness", from NoiseTube. The field light will stay null as we have not any data to fill it.

When the entire record has become as it is required, we call the *pushSensorData* function to store the object in the Sensor collection. The function *pushSensorData* fills the last field “hour” with the current time in Unix timestamp format.

History record jobs

History record jobs contain the following jobs: Hourly job, daily job, monthly job and yearly job for both country and locality data. IoT and crowdsensing applications gather a huge amount of sensor information. Multiple clients are feeding the managing system with a big volume of data which in many cases will be difficult to analyze and manage. Some of the capabilities of our platform are the historical map, the access data API and the 2d-3d visualization averages of countries. By real-time we mean that the data are manipulated by the server at regular intervals so charts illustrate the last samples with statistical ways.

Instead of storing the sensor data to one collection and making a heavy query, we create a more effective job. Time jobs are used to aggregate data according to the needs of the charts. We group records with same localities and timestamps and create new collections with averages of the initial data. This kind of job helps us to save time in the visitor page when it needs to visualize the data. We have made 4 server jobs to aggregate raw sensor data based on time. The idea is to take raw sensor data and aggregate them to create country and locality hourly data. Respectively, we will take the hourly sensor data and aggregate them with the same way to create day’s sensor data. The same procedure follow for monthly and yearly data. The server contains 4 jobs for country data and 4 for locality data. There are not any differences between country time and locality time jobs.

Clean and record generator jobs

At frequent intervals we fire a clean job to keep the database in good condition and not overcome the quota offered by the host provider. We calculate the current hour and remove the last two hour data from the sensor collection. The calculation of current time is made with use of the date function. We transform conventional timestamp into unix timestamp in seconds and then we divide by 1000/3600 to gain a 6 digit number with hour timestamp. This way we can keep the

database compact. Each object in database takes about 250 bytes of disk space, so 1GB will contain about 400 localities with hourly data for one year.

```
var curr_hour = Math.floor((new Date() - 0)/1000/3600);
```

Here it is worth mentioning that unix timestamp is a way of storing instants in time, defined as the number of seconds that have elapsed since midnight of Thursday, 1 January 1970, UTC. It has a form of a ten digit number that can represent multiple time zones at once.

Another server job is the record generator job. Due to the small amount of data from client applications and NoiseTube API, we have created a signal generator. The duty of this job is to generate incidentally noise sensor data at frequently intervals. When such a sensor document is ready, it pushes it into the sensor collection with client application and NoiseTube data for the reverse geocoding job. First, we have the user field, which takes the name “gen” due to the generator. Lat and long fields take specific coordinates from an array with a big amount of countries. Hour field takes the current hour in hourly format. Finally, it is the sensor field for noise data which takes a random value produced by a random generator with upper limit the 60 db and lower limit the 10 db.

Time aggregations job

One of the most important jobs for the server is to create new aggregated sensor data from raw data in order to provide quicker services in the 3rd party component. There are 4 jobs for this part: hourly data job, daily data job, monthly data job and yearly data job. Figure 4.9 shows a diagram with the sequence of jobs. Practically there is not any connection between the jobs since they are independent to each other.



Figure 4-9: Time aggregation sequence job

Hourly job

The first job is to convert the data from sensor collection into hourly sensor data. The hourly data job calculates the current hour and gets the sensor data of the last hour based on the unix timestamp. When it has taken all the records with the specific hour then it groups them by country field and by user field. The result is an array of countries, each of which contains an array of users and each user has its average value of noise data. Next, we do a sum aggregation to the array so every country is associated with its sum and number of records. We create an array of values indexed by country in order to have a faster access to the collection. The final step is to insert the aggregated data into the CountryHour collection. For each country we do:

- 1) If there's no data in the CountryHour collection for that country and hour, we insert the average data for that country into the CountryHour collection.
- 2) If data exists, we are updating the record averaging the existing data with the current average data for the country.

The Hourly job updates the hourly statistics per minute. Below there is a table which compares the document of sensor collection with the result of the hourly job. We follow the same procedure for locality data but instead of aggregating with the country field, we aggregate with the locality field.

Sensor collection document	Country hour collection document
<pre>{ "_id": mongo ID, "country": geolocation data, "locality": geolocation data, "place": geolocation data, "hour": unix timestamp in hours, "lat": latitude, "lng": longitude, "user": id of the user ("guest if data not from client"), "sensors": { "noise": noise data, "light": luminocity data } }</pre>	<pre>{ "_id": mongo ID, "country": geolocation data, "hour": unix timestamp in hours, "sensors": { "noise": noise data, } }</pre>

}	
}	

Daily – Monthly – Yearly job

The CountryHour collection is the source of data for the daily job. The latter, instead of taking input from the raw sensor collection, takes as such the result of the hourly job. At first, we calculate the current day in timestamps and then we go through the last two days documents. For every day we find the corresponding hourly data records and then we follow the same procedure as with the Hourly job. That is, we take all the hourly records within the specific period and then group the records by country field and by user field.

The result is an array grouped by countries. Each country contains an array with hours. Next, we do a sum aggregation to the array so every country is associated with a sum value and its number of records. We also create an array of values indexed by country in order to have a fasted access to the collection. The final step is to insert the aggregated data into the CountryDay collection. For each country we do:

- 1) If there's no data in the CountryDay collection for that country and hour, we insert the average data for that country into the CountryDay collection.
- 2) If data exists, we are updating the record averaging the existing data with the current average data for the country.

The Daily job updates the daily statistics per minute. We follow the same procedure with Monthly and Yearly jobs. We don't use anywhere the yearly data for visualization purposes. The only use is to get the country list in the chart page due to the fact that it is the most lightweight collection with the fewer documents.

4.2.3 3RD PARTY COMPONENT

The 3rd party component, similar to the client component, is built upon the Meteor framework. Meteor is a full stack real-time framework that uses MongoDB as its main database and DDP (websocket) as the communication channel between its components. We use Meteor because its nature is to be real-time by default. Also, MongoDB is a next-generation document-

oriented database, which is storing data in a JSON-like format, making the integration of data in certain types of applications easier and faster. MongoDB provides scalability and flexibility to the developer. It is perfect for IoT applications, which need very large databases, and also in real-time analytics that need lightweight data.

The 3rd party component provides a way to visualize the data collected by the client devices of our framework in real-time. It provides two ways for visualization: 1) Google maps and 2) analytics' charts. By real-time we mean that the data are manipulated by the server at regular intervals so the generated charts visualize the last samples with statistical ways. Also, analytics' charts are divided in two categories: 1) time charts aggregated by country and locality information and 2) averages aggregated by country data. Finally, averages' graphs are divided in two categories: 1) 2d graphics provided by the Ext JS framework and 2) 3d graphics provided by the X3dom framework. Also, the 3rd party component provides some others capabilities to the end user which are: dynamic maps, historical maps and an API to retrieve information from the database.

Access Data API

This API allows users to have access to raw sensor data from our server by specifying some parameters [18] and use them at will. The can send a query from their browser directly to the database. Below is an example of such a query:

```
http://html5platform.tk:3300/api?geo=0.805974,-
100.2278493,88.4755191,172.1061351&type=noise&max=10&maxlevel=50&minlevel=48
```

Right after the domain name and port of this service, the end user needs to specify some parameters that elaborate his query. Table 4.2 includes the key parameters and their description.

Table 4.2: Description of API keys

Key	Description
Max	The maximum number of returned items (<=500)
Type	Type of the sensor data. Can be noise/light/both.
Geo	Coordinates. Format: minLat,minLng,maxLat,maxLng

Maxlevel	The maximum noise level
Minlevel	The minimum noise level

The document that is returned to the user contains 5 fields: Hour, Sensors, Country, Locality and Place. We notice that we return to the requesting user only the database fields with value to him, excluding fields such as “mongoDB id” or “id” of the user.

```
var fields={
  '_id':0,
  'place':1,
  'locality': 1,
  'country': 1,
  'hour':1
};

if(query.type!="noise")
  fields['sensors.light']=1;
if(query.type!="light")
  fields['sensors.noise']=1;
```

In order to proceed with such a data request, we need to retrieve data from our database. So, we send a parameterized find request. We define the fields that will be returned with the natural operator in a descent order. Natural order refers to the logical ordering of documents internally within the database.

```
data=Sensor.find(select, { fields:fields, sort:{$natural:-1 }, limit:parseInt(max)}).fetch();
```

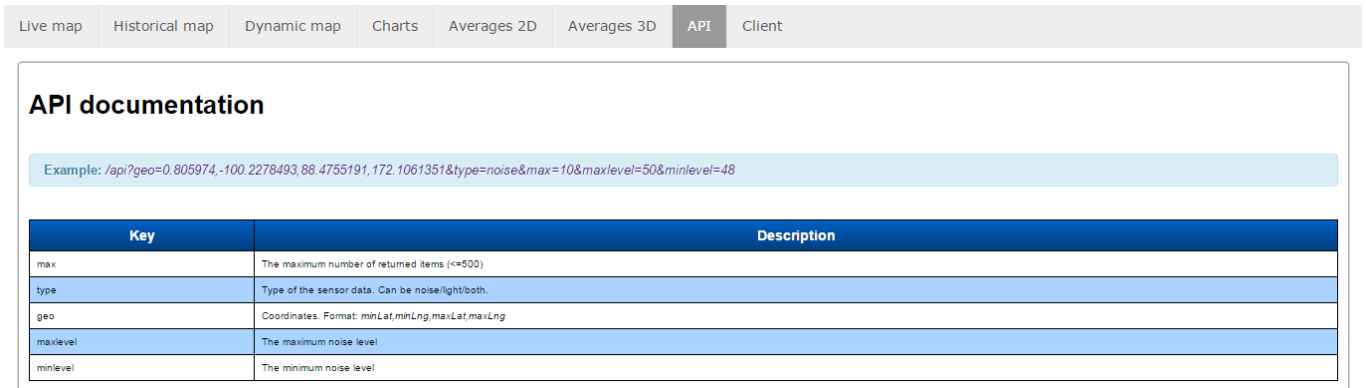
The last step is to flush all the data to the user in a JSON format.

```
this.response.writeHead(200, {'Content-Type': 'text/plain'});
this.response.end(JSON.stringify(data));
```

Below is a box which contains a return object from the access data API.

```
{"hour":396065,"sensors":{"noise":50},"country":"United Kingdom","locality":"Scotland, City of Edinburgh","place":"Waterloo Pl,
```

Figure 4.10 shows the Access Data API documentation from the User Interface of our visitor page.



Key	Description
max	The maximum number of returned items (<=500)
type	Type of the sensor data. Can be noise/light/both.
geo	Coordinates. Format: minLat,minLng,maxLat,maxLng
maxlevel	The maximum noise level
minlevel	The minimum noise level

Figure 4-10: API documentation

Dynamic map and data uploading module

This service offers the capability to the visitor user to upload custom sensor data to our server, which are then displayed as a heatmap. Figure 4.11 shows a sample of such noise data that are appeared as heat points in a google map. Also, the dynamic map service can store such data in the sensor collection of our data base, so it can be also used for other purposes.

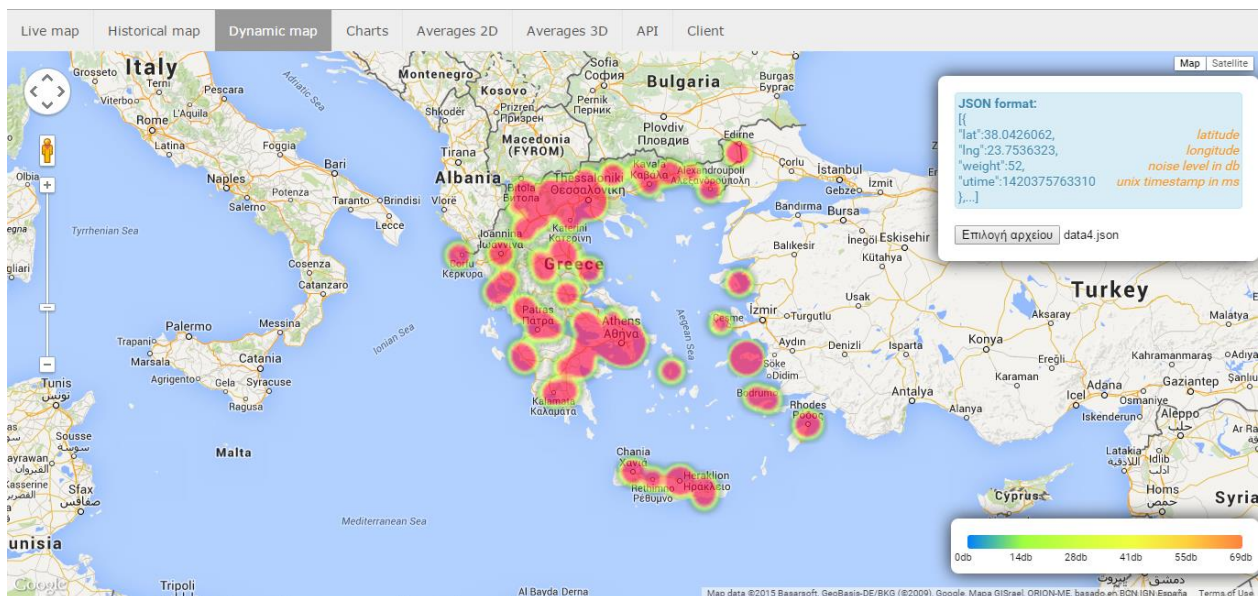


Figure 4-11: Heatmap screenshot

After the page is rendered, we initialize the google map. Also we initialize the panel overlays for legend panel and controls.

```
map = new google.maps.Map($('#map-canvas')[0], mapOptions);
map.controls[google.maps.ControlPosition.RIGHT_TOP].push(
    document.getElementById('uploadContainer'));
map.controls[google.maps.ControlPosition.RIGHT_BOTTOM].push(
    document.getElementById('legendContainer2'));
```

The next step is to draw the legend. The map legend displays the gradient of colors for heatmap in 0...100 range.

```
for(i=0;i<=5;i++) {
    grd.addColorStop(0.2*i, '#' + colorByNoise((maxLevel/5)*i));
    var dbLevel=Math.round(i*(maxLevel/5));
    $('#mylegend').append("<div style='height:16px;position:relative;top:-
"+(i*16)+"px;left:"+(i*(canvas.width/5)-14)+"px'>" +dbLevel+"db</div>");
}
```

In *upload* section we have the simple upload form by using the HTML5 “File API”. Every time the user selects a file, it uploads it to the browser app, which converts it into binary format.

```
'change #json_file': function (event, template) {
    var files = event.target.files;
    for (var i = 0, ln = files.length; i < ln; i++) {
        var reader = new FileReader();
        reader.onload = function(e) {
            update(e.target.result);//update function
        }
        reader.readAsBinaryString(files[i]);
    }
}
```

Then we fire the *Update function*, which is responsible for two jobs. First, it goes through the data array and puts all the data into the heatmap to visualize them. The Update function takes the Json file and parses it to use the noise values as weights in the heatmap. We use the “lat” and “lng” fields to feed the location property of the heatmap and, also, the “weight” field to feed the corresponding weight property. Table 4.3 shows the properties of such a heatmap.

Table 4.3: Properties of heatmap

Properties	Type	Description
Location	LatLng	The location of the data point.
Weight	number	The weighting value of the data point.

The second job is to put the received data into the sensor collection of our database by calling the server method *pushSensorData*. Due to the fact that the user document includes time values in seconds, we need to transform these values into hour timestamps for compatibility with the other database records.

Real-time map

The main functionality of the visitors’ web page is to illustrate the sensor data derived from the live users of our framework into a reactive google map. The data for the real-time map are derived from 3 sources of data: Client components, NoiseTube API and Signal generator. This service displays the last gathered values over the google map as points with noise and geolocation information. Figure 4.12 shows a screenshot from the real-time map of the visitors’ page.

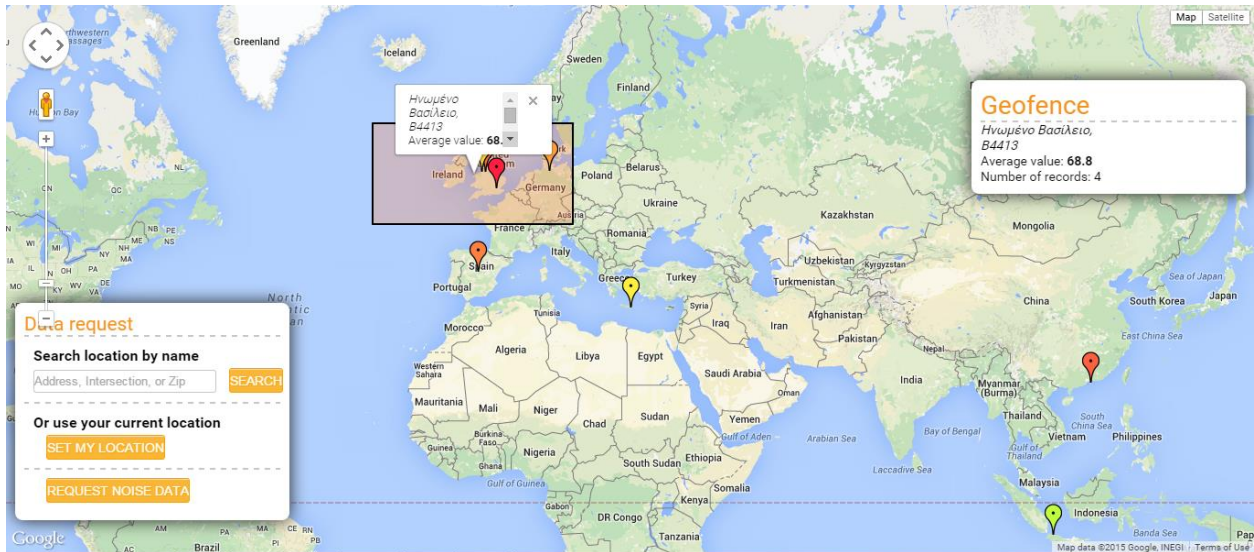


Figure 4-12: Reactive real-time map

When the page is rendered, we initialize the google map. Along with the google map, we initialize the panel overlays for info panels and controls. The next task is the initialization of the geofence controls on the map. Geofence enables polygons drawing on the map and the retrieving of information about the part of the map selected by these polygons.

```
map = new google.maps.Map($('#map-canvas')[0], mapOptions);
map.controls[google.maps.ControlPosition.RIGHT_TOP].push(
  document.getElementById('legendContainer'));
map.controls[google.maps.ControlPosition.LEFT_BOTTOM].push(
  document.getElementById('legendContainer2'));
```

When the map is loaded we fire the update function, which updates and displays the data on the map. The Update function detects the coordinates of the part of the map which is now displayed for the user. By these coordinates and current filter settings it retrieves the specific data by the server method *getLiveUsers*. This data are aggregated by locality and get displayed as map points. Each point has its color according to its noise level.

```
pinColor=colorByNoise(res[k].sensors.noise);
```

Also we put the data from the database into the point variable.

```
markers[key] = new google.maps.Marker({
  position: new google.maps.LatLng(res[k].lat, res[k].lng),
  map: map,
  icon: pinImage,
  shadow: pinShadow,
  data: res[k] //data from the server
});
```

Then, on each point on the map we attach an “onclick” event listener, which provides info about clicked points into an info window.

```
google.maps.event.addListener(markers[key], "click", function(){});
```

The update function fires every 1000ms to make the map real-time. On every update, we only draw new points and delete the expired ones, so we don't redraw the whole point array.

```
var key = res[k].lat + ',' + res[k].lng + ',' + pinColor;
//by coordinates and color
if (!markers[key]) { //adding new points }

for (var k in markers) {

    if (!found[k]) {
        markers[k].setMap(null);
        delete markers[k];
    }
}
}
```

Ticket functionality

Real-time map has a unique functionality that allows visitors to communicate with live users and send them noise request tickets. In particular, visitors have the capability to request noise data for a specific location. The visitor can either use his own location by geolocation API, or specify it by text search. If the user uses the geolocation API, then the geocoding service transforms his coordinates into the corresponding locality or place. When the user uses the text search, the geocoding process, again, checks the location text. Then, the request ticket goes to the request panel of the client page running on the live users' devices. Only live users within the specified location receive such requests. We build a document with 3 fields: lat, lng (for the coordinates) and place and we insert it into the UserRequest collection. Later a confirmation alert box is displayed in the browser. If the data are not valid the alert box displays a message "unknown data". In the client page, clients can see the list of the tickets valid for them. Also, once a new one is added in the database, clients receive a notification message with a sound alert.

Historical map

This service displays a heatmap over the google map with historical noise value data. Figure 4.13 shows a screenshot from the historical map.



Figure 4-13: Historical map

When the page is rendered, we set the default session variables and initialize the google map. After that, we initialize the panel overlays for info panels and controls. When the map is loaded we fire the update function which updates and displays the data for the map overlay. Also we setup the heatmap controls which provide the capability for data filtering.

```
google.maps.event.addListenerOnce(map, 'idle', function(){
  update();
  setControls(map,update);
});
```

The update function detects the coordinates of the map which is now displayed for the user. By these coordinates and current filter settings it retrieves the data by the server method *getHistoryData*. Then the data, aggregated by locality and converted into a Google Heatmap Layer data format, goes into the heatmap and get displayed as an overlay.

```
var pointArray = new google.maps.MVCArray(items2);

heatmap[heatmap.length] = new google.maps.visualization.HeatmapLayer({
  data: pointArray, //data from server
  gradient:[
    'rgba(0, 0, 0, 0)',
```



```

'rgba(128, 255, 64, 1)',
'rgba(240, 255, 64, 1)',
'rgba(255, 128, 64, 1)',
'rgba(255, 112, 64, 1)',
'rgba(255, 0, 64, 1)'
],
maxIntensity: 100,
radius:0.75,
dissipating:false
});
heatmap[heatmap.length-1].setMap(map);

```

Every time the user activates the map filter, we fire the update function using the settings set by the user. After specified the zoom level on the map, we display in sight view the points from the database with info about geolocation and noise level as we presented in the live map section.

Charts by time

This service displays spatial - temporal analytics charts. The charts display daily, monthly and yearly data for all the countries that participate in the project. Figure 4.14 shows a screenshot of the charts page.

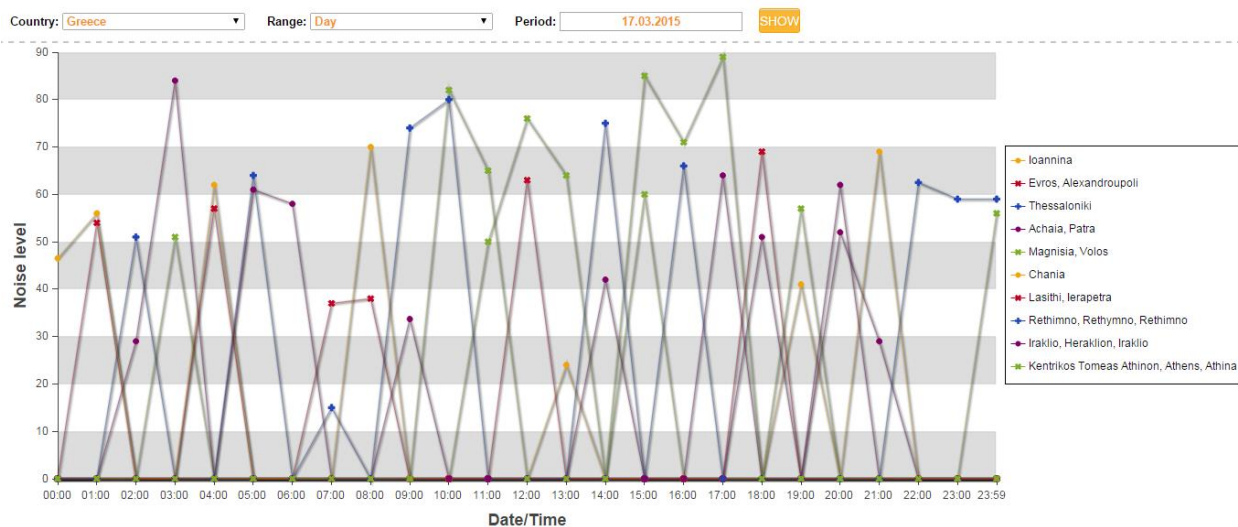


Figure 4-14: Spatial - temporal analytics charts

The logic behind this service is just to convert data from the collection records to chart data format. The charts have 3 types of displayed information: hourly chart for an exact date, daily chart for a selected month of the year, monthly chart for a selected year. Each type can have 2 states: country average values and locality values for the selected country.

When the template is rendered we fire the chart initialization function. First we initialize the ext.js data storage.

```
store1 = new Ext.data.JsonStore({  
  
  fields: ['name', 'data1'], //fields that we parse from given data  
  
  data: generateData($('#calend').val()) //function that returns data  
                                     from server by selected date  
  
});
```

The user interface has options to select country, range type and period. Those options will retrieve data from the *generateData* function. The *generateData* function will call the following server methods: *getCountryDayData*, *getCountryMonthData*, *getCountryYearData*. Each of these methods has option to select data from the database only for the localities of the selected country. Also, every method retrieves data from different collection and has different date type according to its scope.

We take as example the function *getCountryDayData*. It takes the date from the user, and makes the query to database “`{ $gte:curr_hour-1,$lte:curr_hour+24 }`”. The query means that it will take the data for all hours from 00:00 of selected day to 00:00 of the next day. Then it will return an array of hourly data to the client.

When it gets the hourly data then it calls the Ext.js constructor and specifies the data storage.

```
Ext.chart.theme.White = Ext.extend(Ext.chart.theme.Base, {  
  constructor: function() {}  
  
});  
  
store: store1, // specify the data storage
```

```
renderTo: Ext.get('charts'), // element of where we place the chart in HTML
```

Later in the data generation function we specify Y and X axes. The name field in the given data reflects to the column size.

```
xField: 'name',  
yField: res.countries[field],
```

Next comes the rendering function which adds to the column title the noise level.

```
renderer: function (storeItem, item) {  
  this.update(item.series.title + ': ' + storeItem.get(item.series.yField) + '&nbsp;dB');  
}
```

After any change we redraw the chart with the new parameters.

```
chart.store.model.setFields(fields2);  
chart.store.loadData(res.data);  
chart.redraw();  
chart.refresh();
```

Averages 2D ~ Ext JS by country

This service displays average data for each country on a column chart. Each column is painted by color according to its noise level. Figure 4.15 shows a screenshot from the averages page.

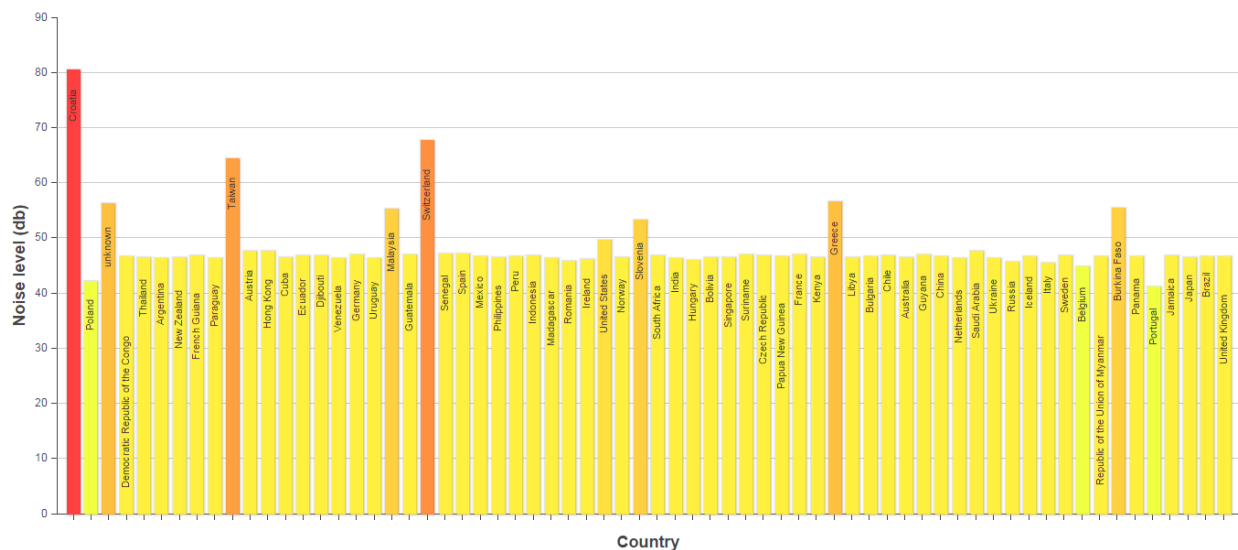


Figure 4-15: Country noise averages with 2D visualization

This service uses the Ext.js visualization framework. First we generate ext.js data storage.

```
var store1 = new Ext.data.JsonStore({
  fields: ['name', 'data1'], //fields that we parse from given data
  data: generateData() //function that returns data from server
});
```

Next, the *generateData* function calls the server method *getCountryAvgData* which returns aggregated data from the database. The *getCountryAvgData* groups documents per country and calculates the average noise per country from the countryHour collection. It returns an array with the name of all the countries and the corresponding average values of noise.

When we obtain the country array with the average values then we set the chart settings in the chart constructor and specify the data storage “store:store1”.

```
Ext.chart.theme.White = Ext.extend(Ext.chart.theme.Base, {
  constructor: function() {}
});
```

Next, we specify the element to put the chart in HTML. Axes x - y take as parameter the elements of country name and average noise in order to create the column size. Finally, we fire the rendering function to colorize the column by the noise level.

```
return Ext.apply(attr, {
  fill: '#'+colorByNoise(record.data.data1)
});
```

Averages 3D ~ X3Dom by country

This service displays the same average data like the Ext.js framework but in 3D format using the X3dom framework. Figure 4.16 shows a screenshot from the country noise averages by x3dom. We create a 3d bar plot with a d3.js approach [158] using x3dom for 3d visualization. After loading the page we fire the *getCountryAvgData* server method. When we get the response we run the *getChart3D* function which provides the drawing of the X3dom scene for the given data. It takes two parameters: the dom element, in which we append the x3dom scene, and the

response from the *getCountryAvgData* function. Then we draw the X and Y axis using the maximum noise value and the number of countries. Later we draw the column chart using the 3D boxes as columns with text on it and with Y-length by noise value. After drawing DOM, we request the X3dom js file which makes the 3D scene by given markup. The X3dom file contains two functions to create the x3d scene. The first is the *initializeAxis*, which creates axis lines in the scene, and the second is the *drawaxis*, which creates the columns on axis and the legend upon them. The colors of the 3d bars change according to the average values for the countries. We change the *diffuseColor* attribute of the 3d bars with the *colorByNoise* function.

Due to the fact that X3Dom is just a plaintext, we use the D3js library to create it dynamically from a javascript function. D3 is a javascript library that creates documents and visualizations which are entirely driven by the data behind them. We are using D3js along with X3dom in order to create hardware-accelerated 3d visualizations directly in the browser.

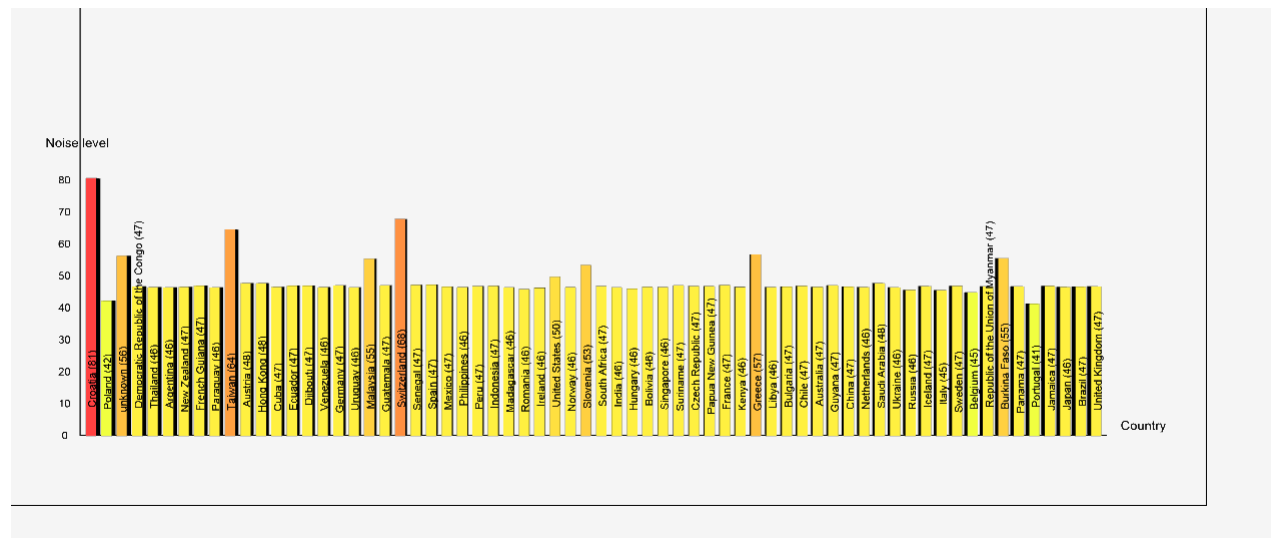


Figure 4-16: Country noise averages with 3D visualization

4.3 SERVER CONFIGURATION

The application is deployed in the digitalocean cloud servers. Everyone can have access to the client web page from the URL: <http://html5platform.tk:3100/>. The visitors' page is available from the URL: <http://html5platform.tk:3300/>. Finally, there is a separate administrator

page from which we run the database benchmarks. The URL for the latter is: <http://html5platform.tk:3400/> and it is closed for the public due to the database access it enables.

Set – up of the server

To set up the server a variety of packages and tools is needed. The first thing that is needed to be installed is a Node.js server, the environment for the server-side.

```
curl -sL https://deb.nodesource.com/setup | sudo bash -apt-get install -y nodejs
```

The next component is the database server. The default database of Meteor is MongoDB.

```
apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10  
echo 'deb http://downloads-distrow.mongodb.org/repo/ubuntu-upstart dist 10gen' | sudo tee  
/etc/apt/sources.list.d/mongodb.list  
apt-get update  
apt-get install -y mongodb-org
```

Then we install the Meteor framework which acts as an application server between client and visitor applications.

```
curl https://install.meteor.com/ | sh
```

We copy the local files of the project into the server folder and after that we install MeteorUP. MeteorUp is a command line tool that allows us to deploy the Meteor app into our server. Along with the MeteorUP we need to setup MUP bundle in the folder.

```
npm install -g mup  
  
setup MUP bundle in your folder  
  mkdir /my-folder-name  
  cd /my-folder-name  
  mup init  
  
configure bundle //by uploading the specific file  
  nano mup.json
```

Finally, we can start the server with the following command.

start the server

```
apt-get install sshpass
```

```
mup setup
```

```
mup deploy
```

starting & stoping the app

```
stop/start noise/noiseclient
```

Bitwise SSH Client

Instead of using the default ssh client of the digitalocean server, we prefer the use of the Bitwise SSH Client. The Bitwise SSH Client provides integrated access to the SSH server's console, either via VT-100 or xterm protocols supported by most SSH servers on any platform. SSH File Transfer Protocol is a network protocol for secure file transfer over secure shell.

CHAPTER 5: PRIVACY

5.1 PRIVACY IN CROWDSENSING

Mobile Crowdsensing applications collect detailed information from sensors and their owners during task management procedures. Most of the time, this kind of information is considered as sensitive and it is endangered if intercepted by a third party malicious program. In this chapter we outline several task management approaches and assess the security issues [78]. Also, we discuss how privacy techniques are utilized in existing sensing applications to address these threats. We will focus on opportunistic people-centric sensing security challenges and we will outline general solutions to this end because of the nature of our HTML5 crowdsensing platform.

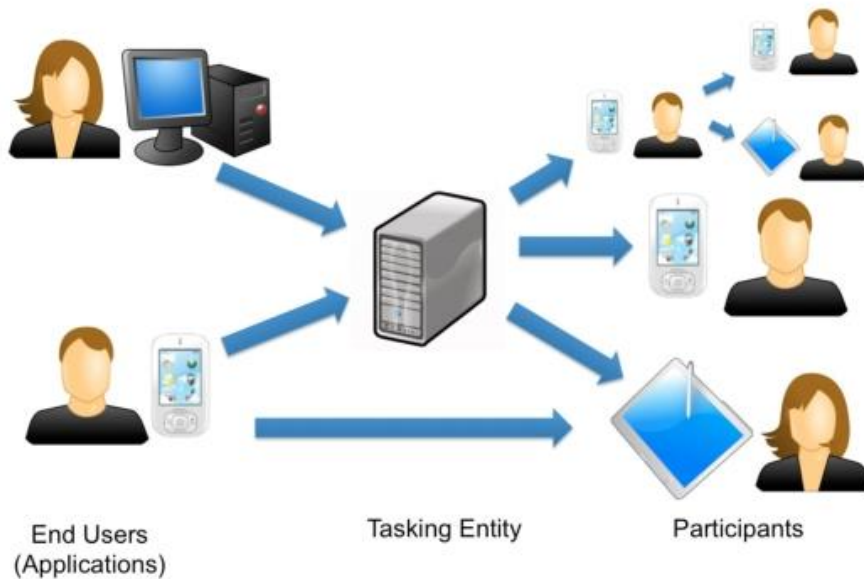


Figure 5-1: Generic structure of task flow in MCS

The issues of security and privacy have changed in comparison with the older static systems. In the past the focus was mainly on security solutions for resource-constrained devices [112,113], secure routing techniques for static sensor networks [114,115,116] and secure data collection and aggregation in static and fixed tree topologies [117], [118] or for providing anonymity in location-based applications [119]. The transfer to more dynamic platforms due to

the domination of mobile devices and anonymous tasking systems has raised other issues that need to be solved. The developers of crowdsensing platforms have to succeed in protecting the data of users and simultaneously the platform must perform all of its tasks. There are three main approaches to solve security issues and protect the users' privacy in crowdsensing platforms. These approaches are anonymization, encryption, and data perturbation.

5.1.1 ANONYMIZATION

Anonymization is a technique which removes the user identity from collected sensor data during the task distribution. In some cases the removal of identifying information cannot ensure the anonymity of the user. Applications which contain the agent location will face such problems. It is very usual for an individual to visit the same place many times. This will result in the identification of the user/individual. For example, we want to measure the noise pollution in a fixed area. The specific task will be credited to a specific person. If this person does not want to have his position revealed, he can answer to the task anonymously. Sometimes, especially in crowdsensing platforms the task itself is more important than who is making the task. Below we will review two anonymization techniques.

- 1) Pseudonyms: This is a technique that hides the real identity of the user and replaces it with a pseudonym [121].
- 2) Connection Anonymization: In this technique we are using IP addresses masking to prevent network-based tracing attacks [120] [122].

5.1.2 ENCRYPTION

Encryption ensures that only an authorized party will have access to the submitted sensor data of the users. Unauthorized third parties will not be able to obtain information even if they ask about it. On the other hand, large volumes of data consume significant computer resources. Recently, a new technique is used for encryption of location-based services [123] and it is called PIR-based method. PIR-based method guarantees cryptographic privacy by allowing data retrieval from a database without revealing any information to the database server about the retrieved item. In crowdsensing applications there are several issues with this approach because

it will suffer from overlapping task selection and bias since sharing entities would not learn which tasks are retrieved.

5.1.3 DATA PERTURBATION

In the data perturbation technique, we add noise to sensor data when the data is submitted by individuals. This will result to non-recognition of the data by a third person. The micro-aggregation is a form of data perturbation in which we replace a selected field with an aggregate or a more general value for example a ZIP code can be replaced by the name of a state. This example was a typical case of micro-aggregation. Micro-aggregation can be operationally defined in terms of two steps, namely partition and aggregation. Partition refers to partitioning a data set into several parts (groups, clusters). The aggregation refers to the replacement of each record in a part with the average record. Such a data perturbation technique is Spatio-Temporal Cloaking: Some applications do not require the exact location, so we can use a perturbed or cloaked location. This technique hides the location of the user into a cloaked region using dummy locations in order to succeed in location privacy [123].

5.1.4 SECURITY IN PARTICIPATORY SENSING

Participatory crowdsensing differs from opportunistic one because it requires from the user to do a specific action such as write a comment about a restaurant or take a photo from a landmark. The security challenges in participatory crowdsensing are different because it intensely penetrates into the human environment [94].

- Privacy: The information that concerns the identity of the user can be leaked easier than in opportunistic sensing. For example the background of a taken picture may contain the home location of the user, or a text comment may contain grammatical errors that can reveal the identity of the user.
- Integrity: In participatory sensing the user can choose what kind of data will be distributed. In this issue, it is in the user's ability if the data will remain intact.
- Availability: In this issue, the user has the ability not to respond to a task request about data or respond with false or useless data.

CHAPTER 6: MOTIVATION

6.1 USER INCENTIVES

Questions about human motivation have been discussed and analyzed in many fields such as philosophy and economics. It is a crucial issue to find the perfect incentives for the participants to join to an application and share their personal data. So what is the perfect incentive for a participant? The promise of financial or monetary gain is an important incentive method for most participants in markets and traditional organizations. Interest and entertainment are important motivators in many situations, even when there is no prospect of monetary gain. Finally, social or ethical reasons, such as socializing with other people or recognition can be good motivational boosts for a participant.

6.1.1 GAMIFICATION

The motivation of using a Crowdsensing application is a crucial issue for the success of the application and can determine the quantity and quality of sensor data. Developers always search for an interesting and motivating point to convince the users to use the application. There is a recent trending technique which is called “Gamification” and it is very popular for motivating user behavior. Gamification [82] is the notion of using various types of game techniques in order to drive desired behaviors. Turning an application into a game and of course by defining some basic principles can inspire a user to visit again and again. With gamification you can incentivize any action you value and can engage an audience either enthusiastic or passive to participate.

Many colossal companies such as Nike, Foursquare, Zynga and Starbucks [85] use Gamification techniques to track and keep loyal customers. For example Nike has created an online “FIT” community and gives the capability to users to share their daily running exercise. After 4 years its market share has increased by 14% and went from 500,000 game members in 2007 to 11 million members in 2013. It has been applied in several fields such as user engagement, physical exercise, teaching and data quality. Also, game mechanics are integral to social networking sites and are being applied to the newest generation of social tools like the Urgent evoke [86] or Stackoverflow [87]. Urgent evoke is a social network game which grants

young people to create solutions for the most urgent social problems of the world. On the other hand, Stackoverflow is a technical question and answer site with a crowdsourced moderation and reputation system. Stackoverflow grants those people who give the best questions and the best answers with points, badges and tags.

6.1.2 FOUNDATIONS OF GAMIFICATION

Game mechanics is not a goose with golden eggs and cannot solve fundamental business problems such as poor infrastructure or a bad customer service. Certainly, if an application includes the basic principles of Gamification then it contains a solid base to build and gain from its benefits. Gamification techniques contain the following basic rules:

Fun is the most important factor of gamification techniques. It is not so significant to base on an extraordinary idea to build a successful and fun game. The last five years, some of the most popular games have used very simple ideas as the main concept of the game case. One example is Farm Ville which involves various aspects of farm management in its gameplay. The player needs to do the basic actions of a real farmer such as planting and harvesting crops and trees in a daily basis to see his farm grow.



Figure 6-1: Farm Ville

Loyalty is a key factor to grow your reputation [83]. In the 19th century in America local merchants used the 10:1 model to persuade the people which arrived in town to buy their products. An example of this model is when someone wants to buy 10 potatoes then the next potato will be with no charge. This is a very effective model and even today huge percentages (about 95%) of loyalty programs use it. A game such as FarmVille will not use such a loyalty program for offering real-world prizes but it will use its status to track loyalty customers. Zynka, the company behind Farm Ville uses word of mouth as a marketing technique to grow its reputation. The tools used to accomplish this are with social networking sites such as facebook and twitter. The players of Farm Ville in order to maximize their reputation and to win social rewards post about the game or invite other people to join it. This method expresses a loyalty to the game without the player even realizing it. Loyalty is no more than a discussion between two people in the back-yard of a house but if it is public then millions of people can view it and participate.

“SAPS” [83] is an acronym which refers to a reward system or a list of rewards for engaging players. The sequence of letters symbolizes the importance of each reward method.

1) S for Status. Status is a relative position of a player in relation to other players. It uses badges and leaderboards to present the actual position of the player. Badges usually are status items and are presented to the other players virtually or physically.

2) A for Access. Access to information, objects or items that other players don't have. For example dinner with the CEO or giving priority to a VIP seat.

3) P for Power. Give the power to a player to control other players in the game. This motivation gives extra responsibility to the player to spend more time on the game.

4) S for Stuff. Although, stuff is last in the list it can be a very strong incentive. The expectation of giving great or free items can motivate the side of the player. The redemption stage is an important factor for stuff. When the item is given away then the engagement is reduced until the next one will come.

6.1.3 THE IDEA OF FLOW

The success of a game is relative with the idea of flow. In psychology, flow is a mental state of an operation in which a person or a player in our case is fully focused and enjoined in the process of activity or game. The psychology professor Mihaly Csikszentmihalyi who is noted for his studies of happiness and creativity is the creator of flow. The idea of flow is explained as the zone between anxiety and boredom. According to Csikszentmihalyi, flow is completely focused motivation. Transporting the idea of flow in computer games, it is easily understood that keeping the player in the flow state is the key to success. So, the game designer has to design the right environment to keep the player focused and simultaneously in an enjoyment stage to guide him into the prized state of flow. Namely to successfully achieve the point between anxiety and boredom. Flow is directly related with a huge spectrum of psychological phenomena, thus the guiding to get a player to master the game is the reinforcement.

Reinforcement has several forms. Fixed-interval reinforcement is a classic form that it is widely used to industrial era jobs. For example workers get paid every two weeks. In the interval between the two weeks workers will do only the job to get paid and nothing more. So, the fixed-interval reinforcement gives limited level of engagement. On the other hand, there is variable schedule reinforcement which doesn't have a fixed type of rewards either in time or size. Gambling and slot games use this type of reinforcement.

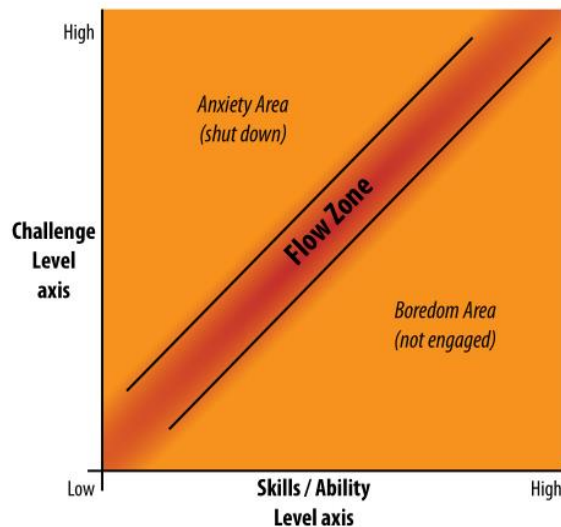


Figure 6-2: The state of flow is achieved when a player is placed between anxiety and boredom over a period of time

6.1.4 TYPES OF PLAYER

The role of the player can be different according to the scope of the game. Richard Bartle has separated players in four types [83,88]: Explorers, Achievers, Killers and Socializers.

Explorers: The explorers will travel around the game maps and will try to find items or unlock hidden stages. A characteristic paradigm was Super Mario in which the player needs to play each stage many times to find every hidden level.

Achievers: The achievers want to win and achieve in an integral part of a competitive game. They are a special type of player because it is difficult to develop a system with all the players' winners. Usually, achievers when losing at the game lose their interest and give it up.

Socializers: The socializers play the games with an ultimate purpose to have social interactions. Mostly, games that are for socializers are classic games like dominoes, poker and mahjong. Of course, they want to win but if not it is not the end of the world.

Killers: The killers look like achievers but with the difference that they want to win and someone else to lose. They are also known as "griefers" and they constitute the smallest population of the players.

6.1.5 GAMIFICATION CASE STUDIES

In the previous subchapters we talked in general about gamification, foundations of gamification, and player's motivation. In this subchapter we will refer to some successful cases with global acceptance and success which use game mechanic techniques and went from theory to practice.

NIKE PLUS: MAKING FITNESS FUN

Nike has built a running game to motivate users for a healthier lifestyle [84]. It uses sophisticated game mechanics to encourage users to join and extend their personal fitness program. Nike has several wins from the specific game. It boosts the brand loyalty and naturally sells more sporting equipment. The game is social and tries to attract a large number of runners which can buy Nike products.

The application is a simple pedometer which track the time and the distance of every run. The routes are recorded and then put on a leaderboard. In the beginning the leaderboard is personal and contains the individual routes of the player. The new user plays against him and tries to make better times. As the player starts exploring the application new games are presented and then he can compete with other social runners in a public leaderboard. The application is connected with Facebook and when a runner starts a new run they can post a notice to their Facebook friends to ask them to join the run. The fun part of the application is that when someone likes the post then the player listens to a crowd cheering as in-run feedback. This is an excellent engagement for the player to continue the run. Also, the application contains secret encouragement feedback from famous runners such as Lance Armstrong and Tracy Morgan adding a variable reinforcement touch. Finally, the routes of every player are presented into beautiful “heat” maps where you can see fast and slow running.

YAHOO! GAMIFIES QUESTIONS

Yahoo uses another approach of gamification and usage pattern to create a community of players who will share their knowledge from a system of asking and answering questions. Yahoo vision was launched in 2005 and it would gain huge success. Google had presented a similar project with the name Google Answers with a far less social pay-for-answers model but it was closed down after a while. The scope of Yahoo was to create a system which will drive the behavior of the player to answering questions, voting, and having his answer voted as a Best Answer. The system contains several types of rewards for those who give the best answers. Unique powers, present the user as featured on the main page and showing avatars with positioned scores are some of the Yahoo rewards. This kind of motivation was so strong that it created a very dedicated community that was driven to ask and answer endless questions.

Going deeper into Yahoo! Answers, it is understood that the reward is not based only in the best answer, but in high-volume participation. Winning the Best Answer game was difficult and needed a lot of time so high-volume participation was also awarded with lots of points. Figure 6.3 shows the answer-question of Yahoo! Answers.

Action	Point value
Begin participating on Yahoo! Answers	One time: 100 points
Choose a best answer for your question	5 points
Put the answers to your question to a vote	5 points
Answer a question	2 points
Log into Yahoo! Answers	Once daily: 1 point
Vote for a best answer	1 point
Rate a best answer	1 point
Have your answer selected as the best answer	10 points
Receive a "thumbs-up" rating on a best answer that you wrote (up to 50 thumbs-up are counted)	1 per "thumbs-up"

Figure 6-3: Yahoo! Answers experience point system

6.1.6 GAMIFICATION IN CROWDSENSING

Gamification has several applications in the field of Crowdsensing. The Crowdsensing developers are using gamification techniques in web and mobile applications as a means to engage the users to use the applications. There are several ways to succeed gathering sensor data using gamification. One way is during the use of the games to create a mechanism which will gather data in background without any interruption from the user. Another way is to create a gamifying process and through it to collect the sensor data.

The author of [81] had created two gamification applications to overpass the boredom of the user when they are using a passive application or doing repetitive tasks. He presents an approach for gathering noise pollution data by using mobile applications. The first application is the NoiseBattle in which the player takes the role of the Achiever. The main scope is to conquer areas and winning points by sending noises to the enemies. Noise Battle has in great status the competition factor in order to make the achievements more pleasurable. The second application is the NoiseQuest where the player takes the role of the explorer. The scope of the application is to walk around the town and take measurements. It is more important to take measurements from different places than the total score achieved from the observations. Competition isn't so serious as in NoiseBattle.

Another paradigm of gamification in Crowdsensing is present in [80]. In [80] they propose a model architecture to solve the problem of coverage of unpopular regions from an area. They

designed a first person shooter sensing game “Alien vs. Mobile User” which provides strong incentives to the players to cover all the regions of a specific area. The main scope is to collect WiFi data (BSSID, SSID, Frequency and Signal strength) to create a campus WiFi coverage map. In more detail the game collects WiFi signal data to provide indoor localization by engaging the player to find indoor aliens. When aliens are getting close to the player then an alert informs them of their presence. The player must start to shoot the alien with the game buttons. If the alien is hit two times then it will escape to another location. The player needs to find it and hit them a third time. The game motivation is doable: 1) Provide an exciting real-world experience and 2) Players can learn useful information about the WiFi coverage map.

6.1.7 NOISE POLLUTION PUZZLES

In our thesis we use gamification techniques to motivate the user to extend the time of using the Crowdsensing application. We embedded a HTML5 game puzzle in the user profile of the client application. The client has a list with 10 different puzzles to solve. Puzzles have a scalable level of difficulty from 1 to 10. The player needs to solve the first puzzle to move on to the second and it continues accordingly. Figure 6.4 shows the first puzzle that needs to be solved to pass the initial stage.

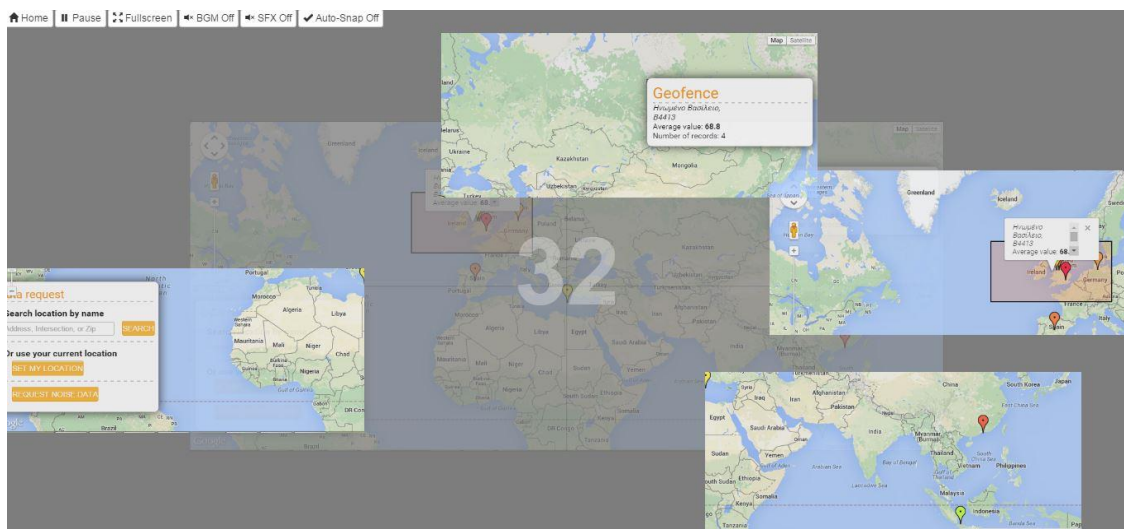


Figure 6-4: Noise pollution puzzle

When the player solves a puzzle he gets the appropriate points. The points are added in the player’s profile which keeps the total player’s score. The user can compare his score with the

this of other users in the leaderboard. The leaderboard shows the 10 top players with the best score. We are using a HTML5 Game framework from [167]. It is loaded in the client component as an iframe. In addition, we have included some .js scripts for calculating scores and storing the highest scores and stages in the users' collection.

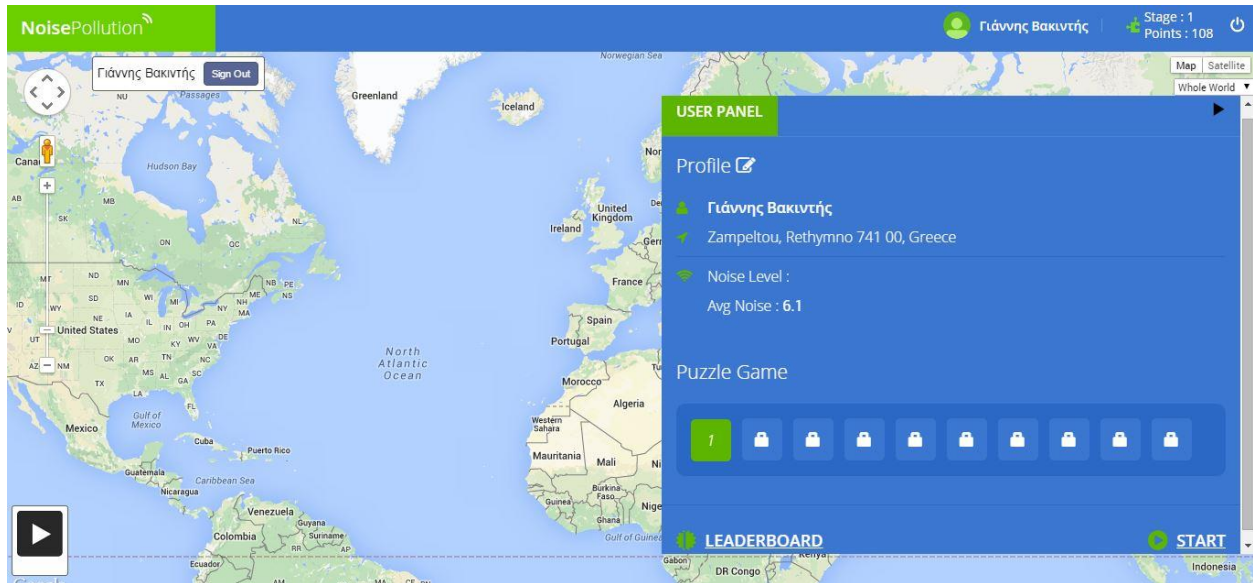


Figure 6-5: Player leaderboard

CHAPTER 7: EVALUATION

7.1 OVERVIEW OF PERFORMANCE & BENCHMARK TESTS

The evaluation and performance tests are divided in two categories. In the first set of measurements we evaluate the performance of our platform under various wireless access technologies (e.g. Wifi, 2G, 3G) in terms of latency. In particular, we measure the latency involved in the performance of several tasks such as: 1) the appearance of the marker from a client-login to the visitors' page, 2) the delay between the uploading of data to the server and their visualization on a dynamic map, 3) elapsed time to visualize data in the historical map, charts and averages, 4) the time to retrieve data from the collection API.

In the second part of evaluation tests, we create a test bed to compare the current database of meteor (Mongo DB) with 2 others databases, MySQL and Redis. In more specific, we benchmarking the 3 databases to evaluate their differences into fundamental operations such as read and write. In this context, we keep the database size stable and are conducting 6 performance tests: Data insertion test, Data reading test, Data reading with sorting test, Data searching test, Data removing test and Data aggregation test.

7.2 EVALUATION TEST OF PLATFORM TASKS

The purpose of making the performance tests is to identify how fast our platform services are delivered to the visitors' page under various conditions (i.e. different wireless network technologies, or transferred data sizes). In order to have reliable results we repeat each test 15 times and write down the average latency value. The majority of our tests concern the tier between visitors' page and server. Only our marker-display test concerns the wholeness of our architecture i.e. the tree tiers: client, server and visitor.

The tool that helps us to complete the elapsed time tests is the “*new Date()*” function of Javascript. The “*new Date()*” returns a data object in the following form: Tue Mar 10 2015 00:03:44 GMT+0200 (Χειμερινή ώρα GTB). When it is specified as current time a new date() minus zero, it responds with the Unix timestamp format of the time. The format of Unix Timestamp has the following structure 1421091697521. It is a big number in seconds which counts the time from the 1st January of 1970. Hence, specifying a second new date() function at

the end of each test and abstracting the first timestamp from the second, we can calculate the duration of each experiment.

In order to emulate different wireless network technologies at the tier between client and server or between visitors and server, we used a proxy server between the client and the server. We emulate 2G, 3G and wifi network via the proxy server. The proxy server that we used is the WinGate, version 8.2.5 [173]. The proxy server acts as an intermediary between the endpoint device, in our case the computer, and another server from which the client is requesting the service. It provides us with options to adjust the link bandwidth in our network to the one we wish. In the bandwidth control panel we can set up the restrictions for our network.

Marker test

In the marker test we measure the time that our system needs to display a marker in the visitors' live map following the login of a client in our system. To be more precise we measure the time from the moment that the client grants the client application with the right to read his sensor data, pushing the OK button, until a marker associated with his presence is displayed in the live map of our system (both at client and visitors' page). Below is the process schema that we follow:

The client user presses the button → the Client application sends the noise level data to the database (every 1000ms) → the Geocoding job updates the data collection → the Data aggregation job inserts the data into the live user collection → the Client application updates the live map (every 1000ms).

Code used:

On Client application

```
insert "new Date()-0" as attribute "user" into collection "sensor"
```

On Visitor application

```
if(parseInt(res[k].user)>142108320){  
    var datenow=new Date()-0;  
    console.log("created: "+res[k].user+";  
    displayed:"+datenow+"; difference:"+(datenow-  
    parseInt(res[k].user))+ "ms");  
}
```

where “res[k]” is the data for each point

The final result is the difference between the two timestamps. The first timestamp is created when the data is sent to the database and the second when the marker is displayed in the google map.

"created: 1421091697521; displayed:1421091698956; difference:1435ms"

Figure 7.1 shows the results of the marker test. The test conducted assuming three different wireless technologies between client and server: Wi-Fi (4Mbps), 2G (250kbps) and 3G (750kbps). As it is depicted in figure 7.1 the difference between the 3 access technologies is small. Wi-Fi is the fastest one (1700ms), 3G is second (2200ms) and 2G follows with 2400ms. The small difference between these measurements denotes that most of the elapsed time is consumed in data processing than in transmission or propagation.

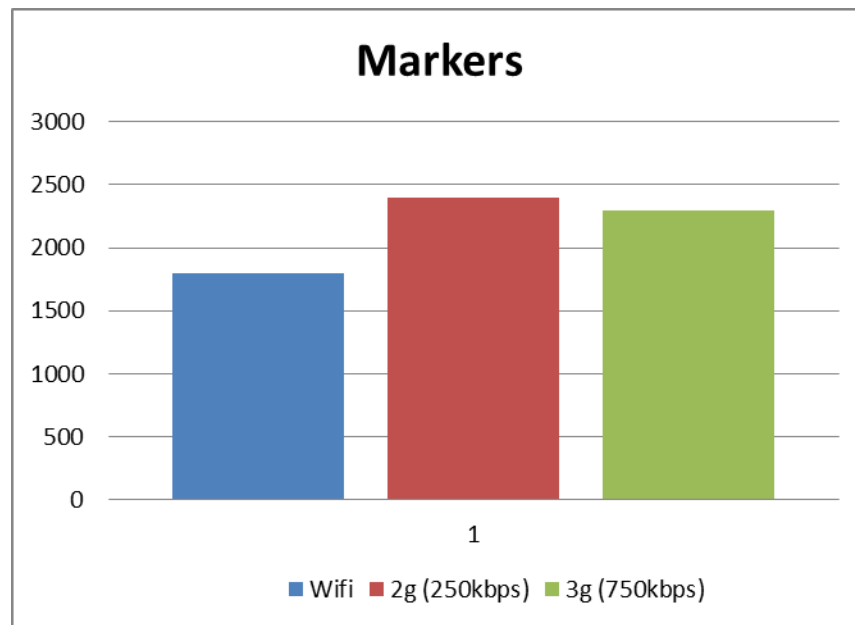


Figure 7-1: Marker test results

Dynamic map test

In the dynamic map test we measure the time elapses between the moment the user presses the button to upload a bundle of sensor data to the server until the time they are displayed in our dynamic map (visitors' page). To this end, we make as samples 4 different JSON files of different size. The structure of the JSON file has similar structure as our sensor data. It contains 4 fields, two with coordinates, one with time (in Unix timestamp format) and one with noise value. The tested samples are of 12, 24, 36, 48 and 96 KB. Figure 7.2 shows the results of the dynamic map test. As we see in the Figure the system takes more time to display the data when the size of the document is bigger. Below is a sample of the code used to take this performance test. This code is also used to the other tests that will be presented hereafter.

Code used:

```
On Visitor application  
on click of the file upload button  
        datenow=new Date()-0;  
after displaying the heatmap  
        console.log((new Date()-0)-datenow);
```

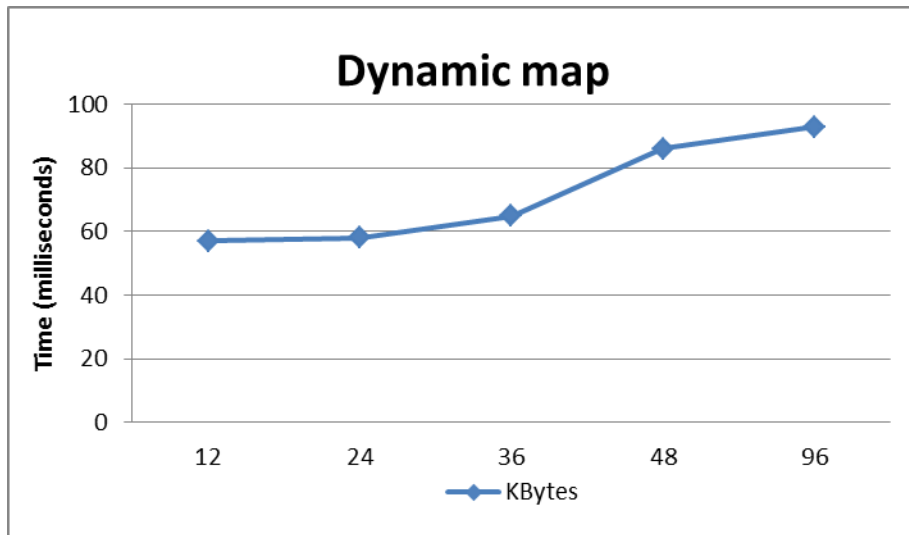


Figure 7-2: Dynamic map results

Access Data API test

In the Access Data API test we measure the time that our system needs to return the requesting JSON objects from the database. We perform 3 queries via the API with different parameters. The first query returns 10 objects, the second 100 and the third 500 objects. Figure 7.2 shows the results of the API test. As we see in the chart, the response time of our system for 10,100 and 500 objects is 15ms, 30ms and 80 ms, respectively.

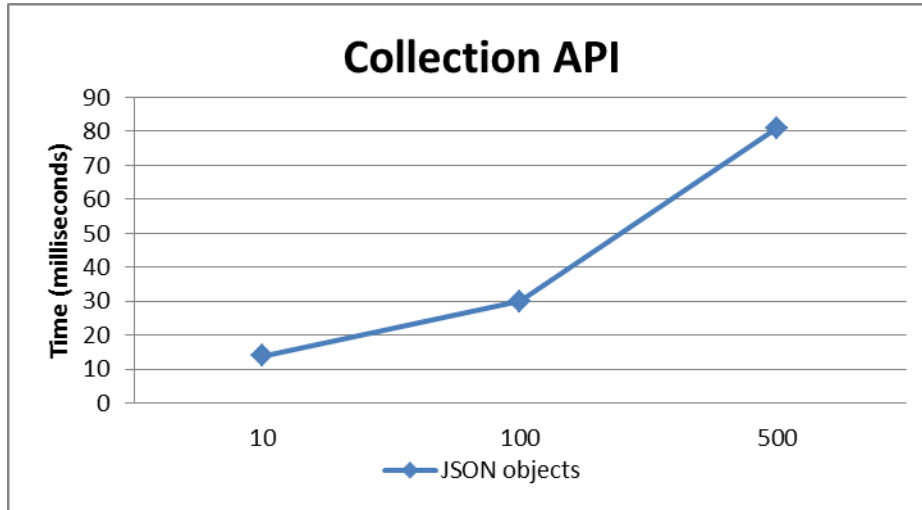


Figure 7-3: Collection API results

Country, locality, averages charts and historical map test

In the country charts test we measure the time that take our graphs to be displayed in the visitors' page. The specific test was conducted on the 31th of January and the total number of countries that were displayed that day in the visitors' page was twelve. Figure 7.4 shows the results of the country charts test. We repeated the test for three different wireless access technologies: Wi-Fi (4 Mbps), 2G-GPRS (250 kbps) and 3G (750 kbps). As it is depicted in Figure 7.4, a Wi-Fi connection takes less time to display the charts than the other two technologies. The Wi-Fi connection takes 2745 ms to conclude, with 2G and 3G to take almost the same times, 2842 ms and 2840 ms respectively.

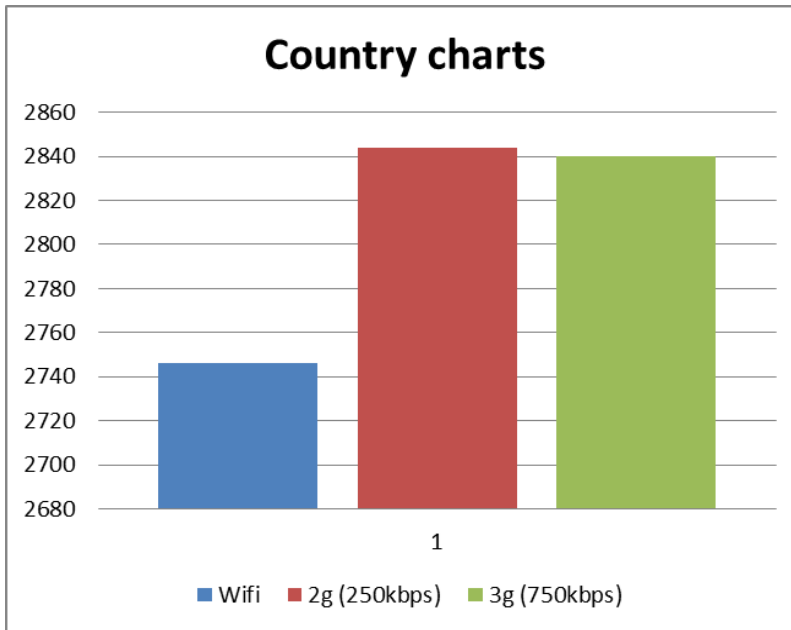


Figure 7-4: Country charts results

Similar tests with country charts were made with the country averages charts. The specific test was also conducted on the 31th of January and the total number of countries that were displayed in the visitors' page was sixty-four. Figure 7.5 shows the results of the country averages charts test. As it is depicted, the Wi-Fi connection takes the less time to display the charts with small difference from the 3G connection. The slowest connection is 2G.

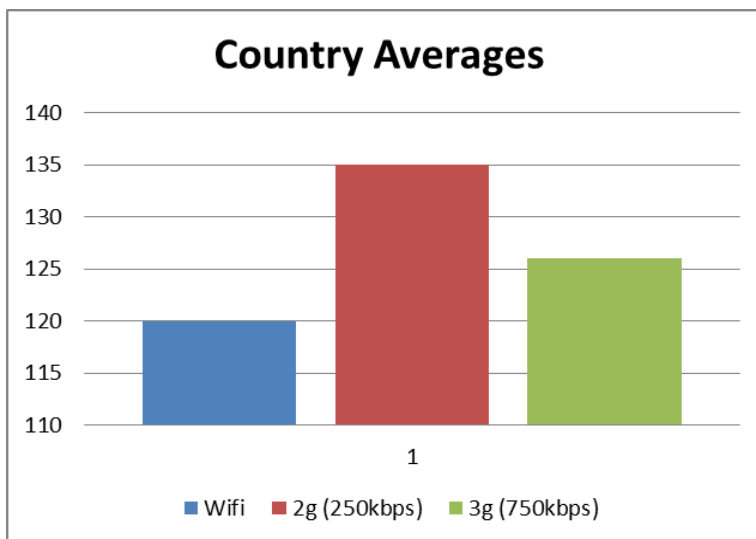


Figure 7-5: Country averages results

Locality charts test was conducted on the 31th of January and the total number of localities that were displayed in the visitors' page was ten. We take as an example localities of Greece. Figure 7.6 shows the results of the locality charts test. As it is depicted, the Wi-Fi connection takes the less time to display the charts. The other two connections 2G and 3G had similar times for the specific charts.

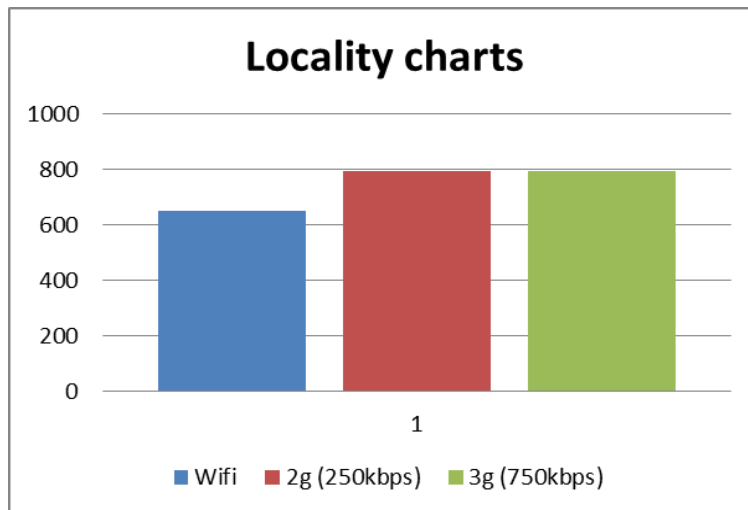


Figure 7-6: Locality charts

The last test conducted with the historical map. Figure 7.7 shows the results of the historical map test. As it is depicted, the Wi-Fi connection takes less time to display the historical map than the other two wireless connections 3G and 2G. The difference between Wi-Fi and 3G is less than the one between Wi-Fi and 2G.

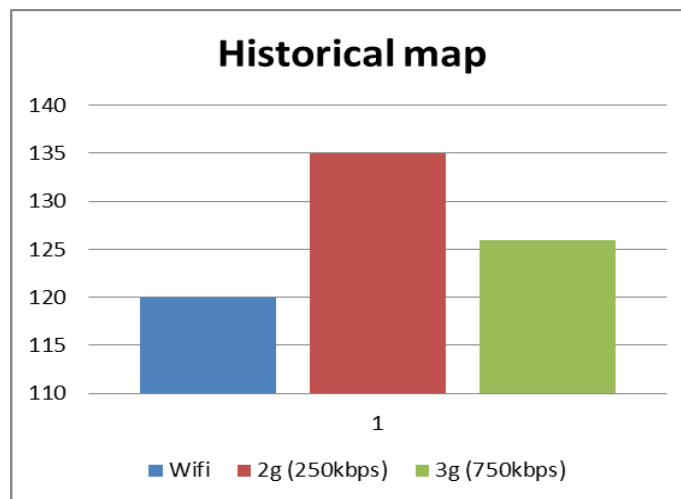


Figure 7-7: Historical map results

7.3 DATABASE – CENTRIC APPROACH

Crowdsensing applications can be regularly established with one of the following 3 models: Remote procedure calls (RPC), publish/Subscribe and database-centric approach. Our platform is based on the third approach, the database-centric. Database-centric is a software architecture where a database plays critical role to all the procedures that take place inside the application. It is the most preferred solution when the application has to do with big data. Generally, a database can provide fault tolerant and reliable transactions. In this benchmark we compare two kind of databases: SQL (MySQL) and NoSQL (MongoDB, Redis).

7.3.1 NoSQL vs SQL

SQL (Structured Query Language) was developed in the 1970s by IBM and since then has become the standard query language for Relational DataBase Management Systems (RDBMS). Databases that belong to SQL category are MySQL, Oracle and SQLServer. They have slightly different syntaxes but there is not required any significant change when switching from one such system to another. A RDBMS is organized into relations between entities, each of which is represented by a *table* consisting of *rows* and *columns*. The header of the table consists of the list of the columns and the body of the table consists of the rows. RDMS are based in the key concept, which is used to order data or map data to relations. The most important key of a table is the primary key which uniquely identifies the rows of the table. A SQL uses the CRUD tasks to access database entries. The initials CRUD is referred to: Creating, Reading, Updating and Deleting data. SQL databases [132] are used for low-volume and low-velocity data such as customer data and billing.

- Pros: Follow ACID (Atomicity, Consistency, Isolation, Durability) rules, Data integrity, data reliability
- Cons: Scaling problem with growing data volume and workload demands.

NoSQL (not only SQL) is another type of DBMS that can be used on the cloud. NoSQL, refers to an well-known group of non-relational database management systems; where databases

are not built primarily on tables, and generally do not use SQL for data manipulation [132]. The NoSQL database is the generation of DBMS which have as scope to eliminate the weak points of relational databases. There are many types of NoSQL databases, such as documents, graph, key-value pairs and column family. All types have as common, that are non-relational. NoSQL deals with data that are more flexible or need a simpler structure. They don't have limitations on data structure, allowing nested documents or multi-dimensional arrays. Also, are meant to be schema-free and suitable to store data that is simple, schema-less or object-oriented. Primary Uses of NoSQL Database are [130]:

- (1) Large-scale data processing
- (2) Basic machine-to-machine information look-up & retrieval
- (3) Exploratory analytics on semi-structured data
- (4) Large volume data storage.

Table 8.1 summarizes the main differences between general SQL databases and NoSQL databases.

Table 7.1: SQL vs NoSQL

SQL	NoSQL
Relational model	Non-relational data (schema-less, unstructured,simpler)
Tables	Key-value, document, graph, column family stores
ACID	BASE
Consistency	Availability, Performance
Single server	Cluster of servers (Horizontal scalability)
SQL query	Simpler and different API

7.3.2 *NOSQL CATEGORIES*

NoSQL databases can be classified into four major categories: Key-Value stores, document stores, column Family stores and graph databases. In this section we will analyze Key-Value stores and document stores because we use databases that belong to these categories.

Key-Value stores

Key-value stores are the simplest type of NoSQL databases. They store data in pairs of key and value. The value is a block of data that have any type and any structure. They don't need any schema to be defined and let the user to define the semantics for the values and how to parse the data. They are easy to build and scale and they have good performance. The basic API to have access to the data are: 1) put (key, value), 2) get (key) and remove (key). Redis belongs to the type of key-value store databases.

Document stores

A document store database uses a database as a collection of documents. It is one step higher than key/value stores. Every document consists of various named fields and one of them is the unique documentID. Document databases are schema free. The data can be of any structure and different among documents. The data types allowed for use vary from strings, numbers, and dates to more complex ones such as trees, dictionaries, or nested documents. The output format can be JSON, BSON or XML. This is a characteristic that makes document stores databases very popular to developers because the server can support not only simple key-value lookup but also queries on the document contents. MongoDB belongs to the type of document store databases.

7.3.3 *OVERVIEW OF TESTED DATABASES*

In this section we describe the characteristics of the 3 databases that we use in our benchmark: MySQL, MongoDB and Redis. We analyze the theoretical approach of those databases and we also underline the main functionalities that each supports.

MySQL

MySQL belong to the category of SQL databases and it is the most popular in business industry. It is owned by Oracle. Many famous applications use SQL such as Facebook, Lindedln, Google and Twitter. MySQL is a relational database and organizes its data into tables, rows and columns. For accessing data it uses SQL statements. The basic statements are: INSERT, SELECT, UPDATE and DELETE. There are also others functionalities such as join, group by and views.

Buffering and caching

MySQL supports a variety of storage engines with different characteristics to manipulate data. The default storage engine is the InnoDB after the version 5.5. It is ACID compliant and supports various kinds of transactions such as commit, roll back and crash-recovery. For caching data in memory it uses a pool buffer. Pool buffer is a linked list of pages, keeping heavily accessed data at the head of the list by using a variation of the least recently used (LRU) algorithm.

MongoDB

MongoDB belongs to the category of document store - NoSQL databases. It has been developed by 10gen and written in C++. It is the most famous No-SQL database and well-known applications such as foursquare use it. Its main characteristic is scalability and speed, so it is suitable to work with large amount of data. The structure of its data has flexible schema. The database itself contains multiple collections and every collection contains multiple documents. Data is stored in BSON format, which is a binary-encoded format of JSON. BSON objects are lightweight, traversable and efficient, so they are very fast in encode and decode operations. Every object contains a unique id that the system automatically adds it to the object if the user doesn't assign it. In figure 7.8 we can see the structure of the Object_id.



Figure 7-8: MongoDB object_id

In the field of querying, MongoDB has a very large set of available queries and user doesn't need to write MapReduce functions. Mongo shell is the MongoDB client that communicates with the database from a command line. The commands to query the database are insert, find, update, and remove. MapReduce operations and a simple aggregation framework are responsive to the aggregations tasks.

Redis

Redis belongs to the category of in-memory key-value store - NoSQL databases. It is considered as a very fast database due to the in-memory storage. It offers high performance and more flexibility than a usual key-value database. A database in Redis is characterized by dictionaries that are pairs of keys and values. Redis offers a lot of choices for data structure. Data can be stored in: String, list of strings, set of strings, sorted set of strings and a hash. Every data structure has its own set of commands.

Redis store data in RAM in order to achieve high performance. Sometimes, this is a drawback because RAM can be used by others applications or services. Redis can also store data in disks. It uses three methods for data persistence: append-only file, snapshots and a combination of both. Snapshots save a dataset periodically when a number of keys is changed. On the other hand, append-only file logs all the write operations.

7.4 TEST BED OF DATABASES

7.4.1 EXPERIMENTAL METHODOLOGY AND SETUP

In this section, we will describe the methodology used to evaluate the performance of 3 databases: MySQL, MongoDB and Redis, when deployed in our platform. For each experiment that was conducted, we will describe the benchmark functions and commands, the procedure and the final results.

7.4.2 EXPERIMENTAL OVERVIEW

The main goal of these tests was to compare the performance of MongoDB, the default database of Meteor, with two other databases Mysql and Redis. The databases were implemented in the cloud server of our platform and the benchmark is considered to be a database benchmark

over sensor data. The benchmark architecture also includes 3 different database clients, one for each database implementation. The benchmark performs basic read, write, search and remove operations and some more advanced, such as search with sorting and aggregation. Each database is evaluated and tested separately, meaning that only one test is running upon time for the current database. Every test comprises a series of requests, such as read and write, from server to database. We set a database client in the server side of the application. In the following sections we will analyze each test separately. The performance of the databases is evaluated by measuring the elapsed time for each request to conclude. Measurements for each test are taken multiple times in order to maintain reliability. Specifically, the values illustrated in the charts that follow have been derived from the average value of a 10-times run of each test. This is a safe way to ensure that the underlying network will not alter the results.

7.4.3 TEST-BED ENVIRONMENT

The whole benchmark was implemented in the server side of our Meteor platform. The server was deployed on a virtual machine instance running 64-bit Ubuntu 14.14 on a Digitalocean instance (droplet) (1 GB memory, 30 GB SSD Disk). The Database editions that we tested are MySQL 5.6.10, MongoDB 2.6.7 and Redis 2.8.9. In order to connect and interact with the database servers, the following libraries and drivers were used for the implementation of the database clients:

- **MySQL:** numtel: mysql, [171]
- **MongoDB:** native Meteor driver
- **Redis:** slava:redis-livedata [172]

In the case of Redis we needed to run a Redis server and a url to connect to it. This is because Redis is not yet shipped with Meteor. Hence, the following command is needed to be passed in the server in order to start the Redis server:

```
REDIS_CONFIGURE_KEYSPACE_NOTIFICATIONS=1.
```

7.4.4 BENCHMARK IMPLEMENTATION

The scope of the benchmark was to run various common operations for databases and record their performance upon big data from sensors. All tests were made on the server side of our middleware platform (located in the cloud). The benchmark comprises 6 separate tests: data insertion test, data reading test, data reading with sorting, data searching, data removal and data aggregation. The visitor of the client page of our platform can easily run the benchmark test by the administrator page we have created. The benchmark test's architecture is shown in Figure 7.9.

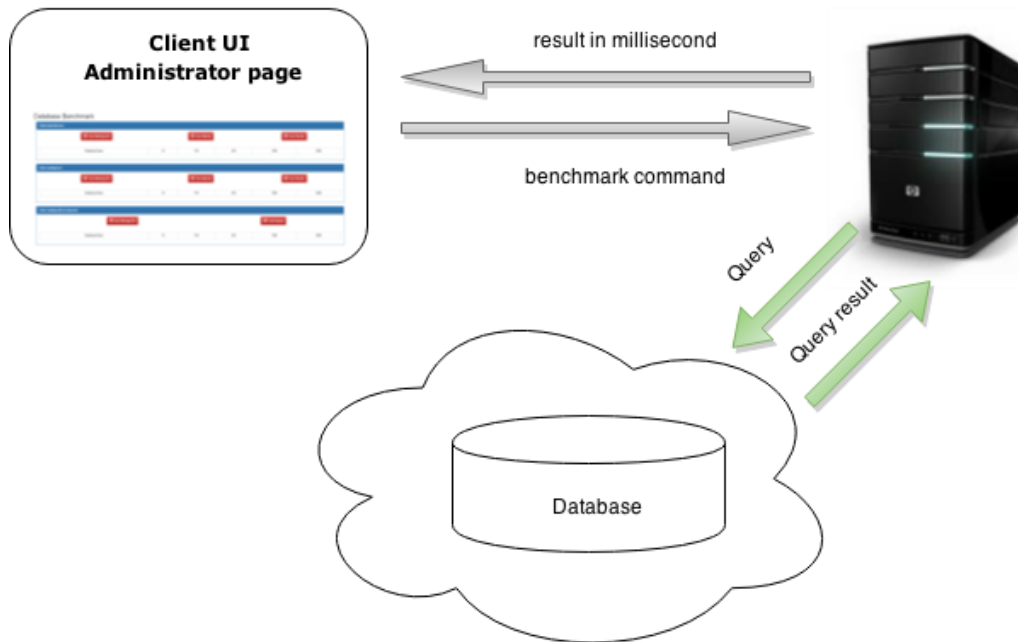


Figure 7-9: Benchmark architecture

All tests are the same for every database. In every test we feed the database with a certain number of documents, namely 10, 100, 500, 1000 and 2000. The scope is to measure the time that elapses between the submission of the query and the time we get the result from the database. We are getting start time upon the submission of the query from the server to the database and we obtain the difference with the timestamp upon the receipt of the results by the server. After that, the server method returns the difference in milliseconds to the client-side function, which displays it. All the tests include a remove and insert method for data. In every

test we start from scratch and this is the reason that it takes some time to proceed. So, for every test, we have two constants:

- The data have the same structure for all records.
- The database collection has not data when we start the tests.

We run the administrator page from the client side of the Meteor platform. The benchmark command enables the query at the server side and then the benchmark process starts. To start the administrator page and run the benchmark tests we need to configure the url of MongoDB driver and enable keyspace notifications for Redis:

- 1) MONGO_URL=mongodb://localhost:27017/noiseserver
- 2) REDIS_CONFIGURE_KEYSPACE_NOTIFICATIONS=1
- 3) ROOT_URL=http://html5platform.tk:3400 meteor --port 3400

7.4.5 DATA STRUCTURE

The common data structure for all records is shown in Table 8.2. Each document has exactly the same structure as the one we obtain from the client application. It has four numbers and one string. The first two numbers represent the location's latitude and longitude and their values derive from a number array with 10 pairs of coordinates. The third number contains the date in unix format and the forth contains the sensor data. Finally, the string represents the locality and it is fed from an array of 10 localities.

Table 7.2: Sensor data structure

Name	Type	Example
Lat	Double	32.8133112
Lng	Double	4.1426565
locality	String	New York City
Hour	Double	1422012856380
Noise	Double	30.5

Table 7.2: Sensor data structure

7.4.6 EXPERIMENTAL RESULTS

In this section, we report the results of all the tests conducted. Figures 7.11 – 7.16 display the charts with the measurements in each category. Figure 7.10 shows the administrator page of the benchmark test. Although NoSQL databases claim that they deliver faster performance with big data than classic RDBMS databases, we reach to the conclusion that MySQL has better performance in our tests. Also, MongoDB, the default database of Meteor, has very good response time in comparison to Redis. However, Redis is not fully supported yet in Meteor. Redis in Meteor does not support documents so it makes for each part of the document one record. In case Redis was fully supported, it would definitely perform much better.

Database Benchmark

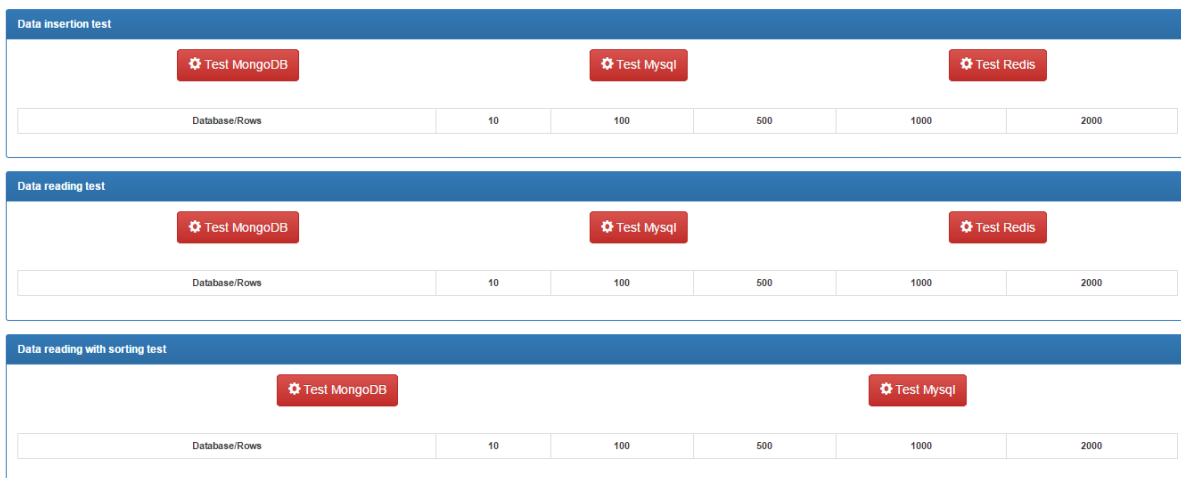


Figure 7-10: Administrator page

A general outcome from the results is that all databases (MySQL, Mongo and Redis) have almost equal response time in case of small number of entries but when this number increases, MySQL and Mongo perform significantly better than Redis. It is worth noting, here, that the initial thought of Meteor developers was to use MySQL as the default database when they started to create Meteor. The reason for choosing Mongo, finally, was that it is consistent with Javascript and the Meteor developers was familiarized with it.

All the tests of the benchmark work with the same way: We are getting the start time at the beginning of each procedure and obtaining the difference with current timestamp at the end. When the procedure is over, the server returns the difference in millisecond. The result denotes server time, since the server sends the query to the database and gets the results back.

Insertion test

Every time we run the benchmark test for a database we consider 5 cases, each with 10, 100, 500, 1000 and 2000 documents, respectively. All tests have the same structure when they start. As we described above, we take some precautions in order to have the same characteristics for every database. When the user presses the button to make a query, we at first remove all the documents from the database and then we insert the necessary documents. Just before the insertion procedure we keep the starting time. When the insertion procedure ends we keep the ending time and we abstract it from the starting time.

```
insertMongoFunc: function(){
    var time=[]; //time =[1,10,50,100,200];
    for(var i=0;i<times.length;i++){ // array with length 5
        var startTime=insertIntoMongo(times[i]); // taking the ending time
        time[i]=new Date()-startTime;// abstract ending to starting time
    }
    return time;
},

insertIntoMongo = function(n){
    testCollection.remove({}); //remove all the records
    var startTime=new Date()-0; // create the starting time
    for (var i=0;i<n;i++)
    {
        for (var k=0;k<coords.length;k++)
        {
            var doc={
```

```
        lat: coords[k].lat,  
        lng: coords[k].lng,  
        locality: locations[k],  
        hour: 1422012856380,  
        noise: k*10  
    }  
    testCollection.insert(doc);  
}  
}  
return startTime;  
}
```

Figure 7.11 shows the results from the insertion test. MySQL and Mongo has very close response times in almost all cases except for 2000 documents. On the other hand, Redis has satisfactory times until the insertion of 1000 documents but in case of 2000 its performance falls in very low values.

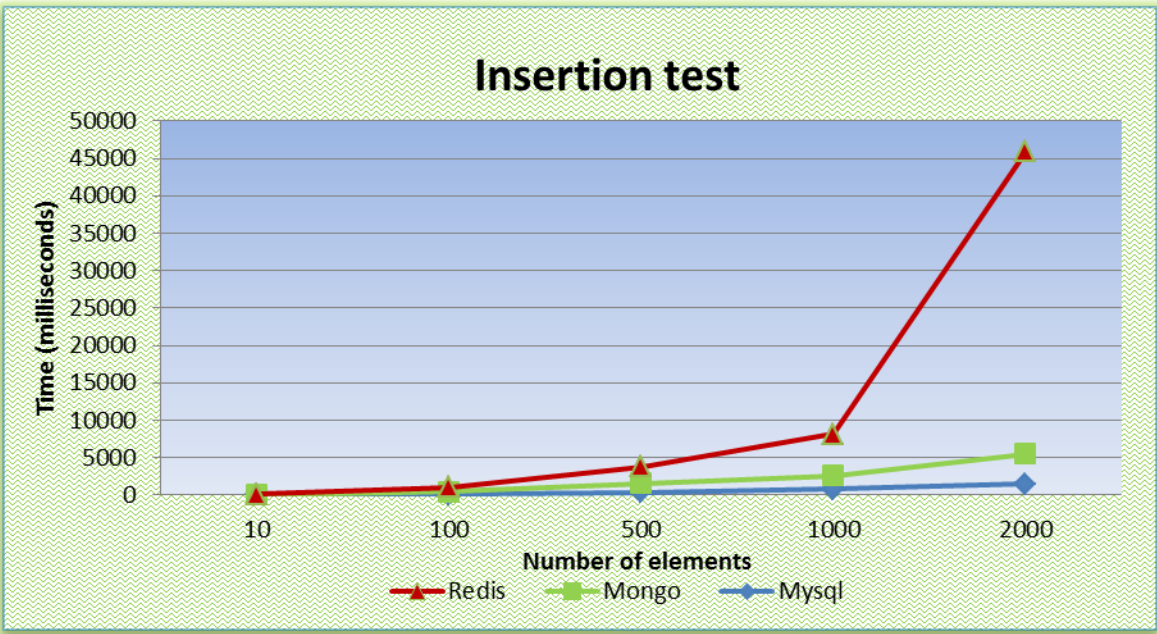


Figure 7-11: Insertion test

Reading test

As we referred above, we also have 5 cases for the insertion of documents. Also, we take the same precautions with insertion test. In fact, hereafter we will have a removing and insertion operation exactly before the starting of the main test. When the insertion job finishes we keep the finishing time. The abstraction of insertion finishing time with the reading finishing time gives us the desired reading time.

Before the reading procedure there is an insertion procedure

```
readMongoFunc: function(){
    var time=[];
    for(var i=0;i<times.length;i++){
        insertIntoMongo(times[i]);
        var startTime=new Date()-0;
        testCollection.find({}).fetch(); //reading command
        time[i]=new Date()-startTime;
    }
    return time;
```

Figure 7.12 shows the results from the reading test. MySQL has a clear superiority against Mongo and Redis.

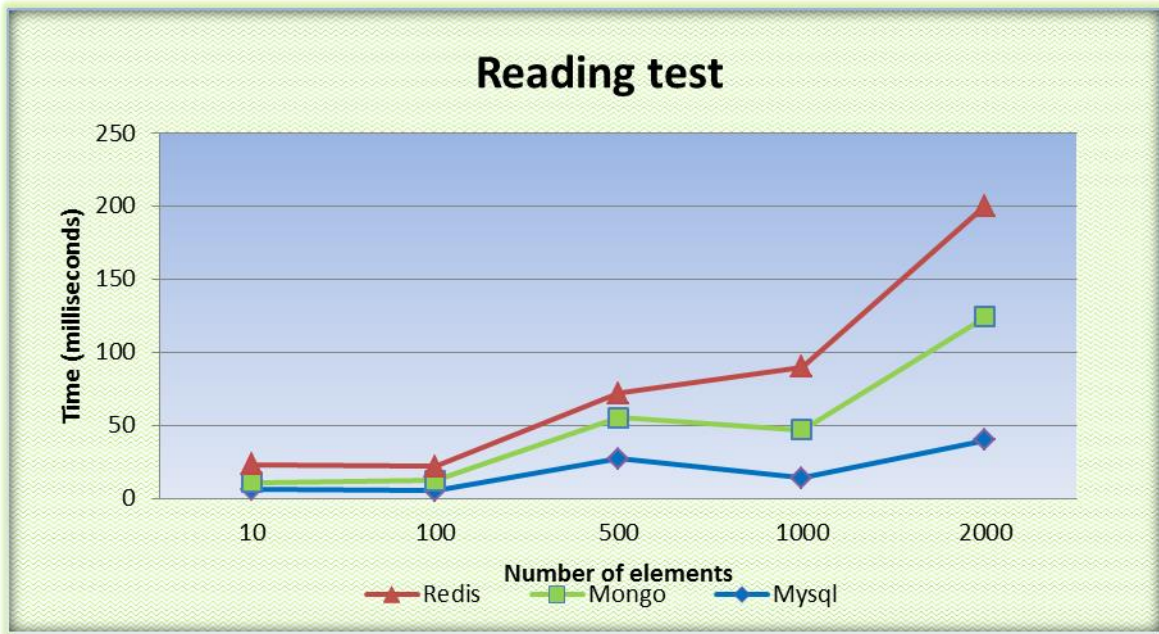


Figure 7-12: Reading test

Reading with sorting test

The "read with sorting" test is the same with the reading one with just ordering of the results in the returned recordset. Recordset is descendingly sorted by the "locality" field. For Mysql we use "order by" command and for Mongo we use "sort". There is no ordering support in Redis driver for the moment.

Before reading with sorting procedure there is an insertion procedure

```
readSortedMongoFunc: function(){
    var time=[];
    for(var i=0;i<times.length;i++){
        insertIntoMongo(times[i]);
        var startTime=new Date()-0;
        testCollection.find({},{sort:{locality:-1}}).fetch();
        time[i]=new Date()-startTime;
    }
    return time;
}
```

Figure 7.13 shows the results from reading test. MySQL has a clear superiority against Mongo in large number of documents. When the procedure starts both databases have similar response times.

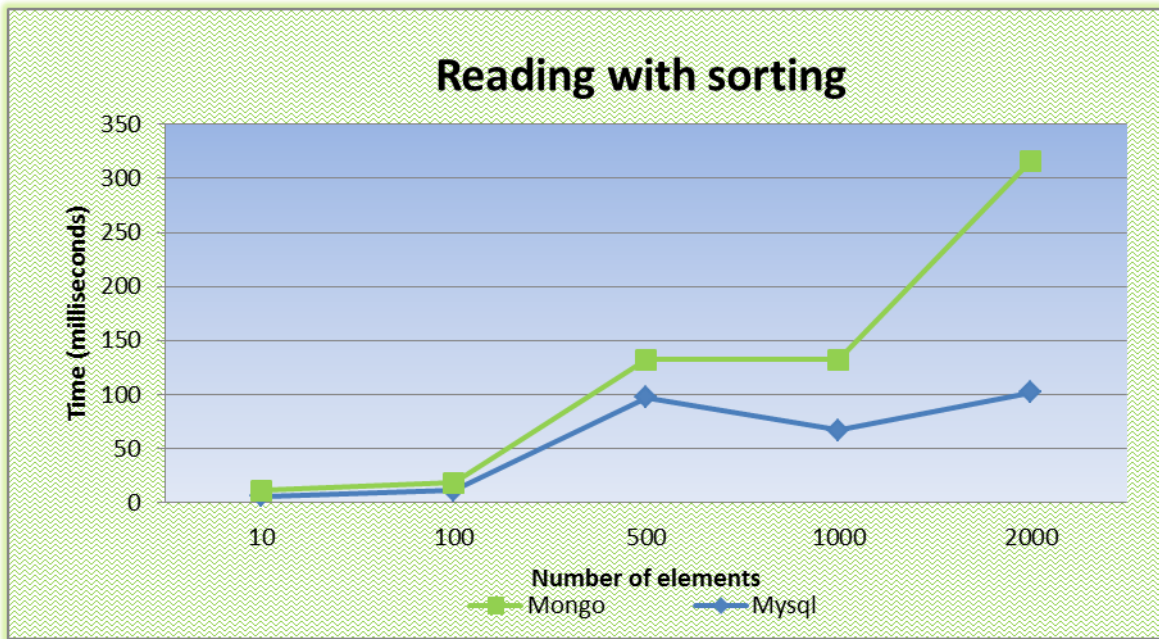


Figure 7-13: Reading with sorting

Searching test

In searching test we search the database by locality “Manila”. Figure 7.14 shows the results of searching test. MySQL along with Mongo has almost equal times. An important notice is that both MySQL and Mongo has the same performance for almost all the procedure. On the other hand, Redis has the same rate with the other two databases until the last set of documents. The searching time rises from 5ms to 324ms when redis searches between 2000 documents. We also tested Redis in searching on more than 2000 documents and the additional time was disappointing.

```

Before searching there is an insertion procedure
searchMongoFunc: function(){
    var time=[];
    for(var i=0;i<times.length;i++){
        insertIntoMongo(times[i]);
    }
}

```



```

var startTime=new Date()-0;
testCollection.find({locality:"Manila"}).fetch();
time[i]=new Date()-startTime;
}
return time;

```

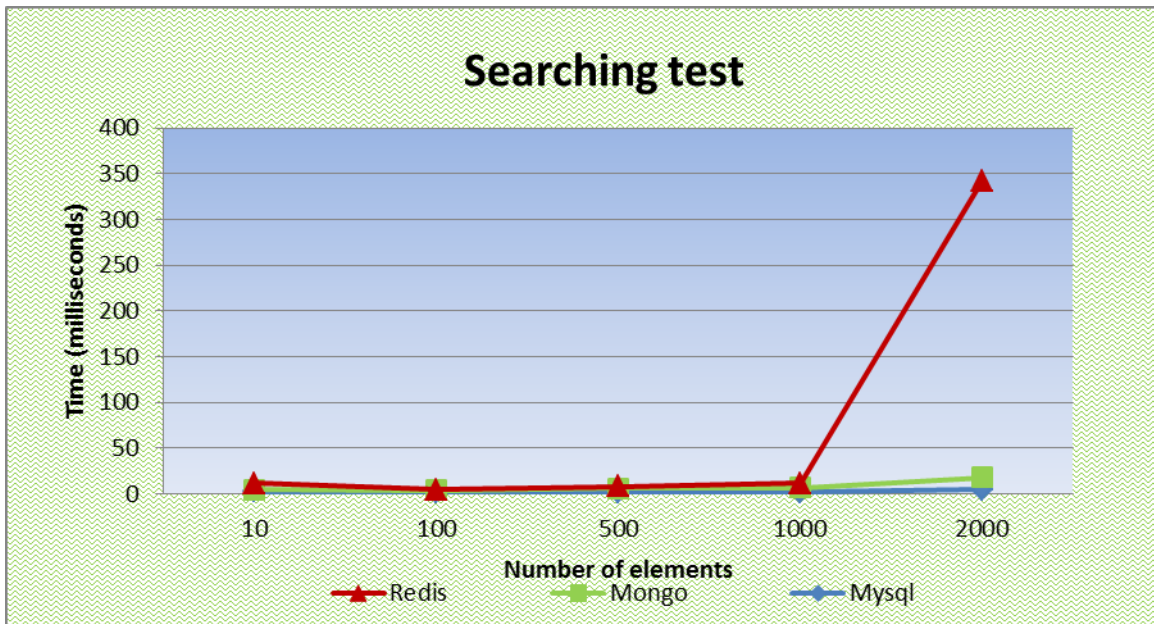


Figure 7-14: Searching test

Removing test

In removing test, MySQL is also the dominant database and Mongo the second. For one more time Redis comes third. All databases have close times which sometimes are equal. For 1000 documents both MySQL and Redis need 6ms to finish the test. Mongo is 1 ms faster. Figure 7.15 illustrates our results.

Before removing procedure there is an insertion procedure

```

removeMongoFunc: function(){
  var time=[];
  for(var i=0;i<times.length;i++){
    insertIntoMongo(times[i]);
    var startTime=new Date()-0;

```



```

testCollection.remove();
time[i]=new Date()-startTime;
}
return time;
}

```

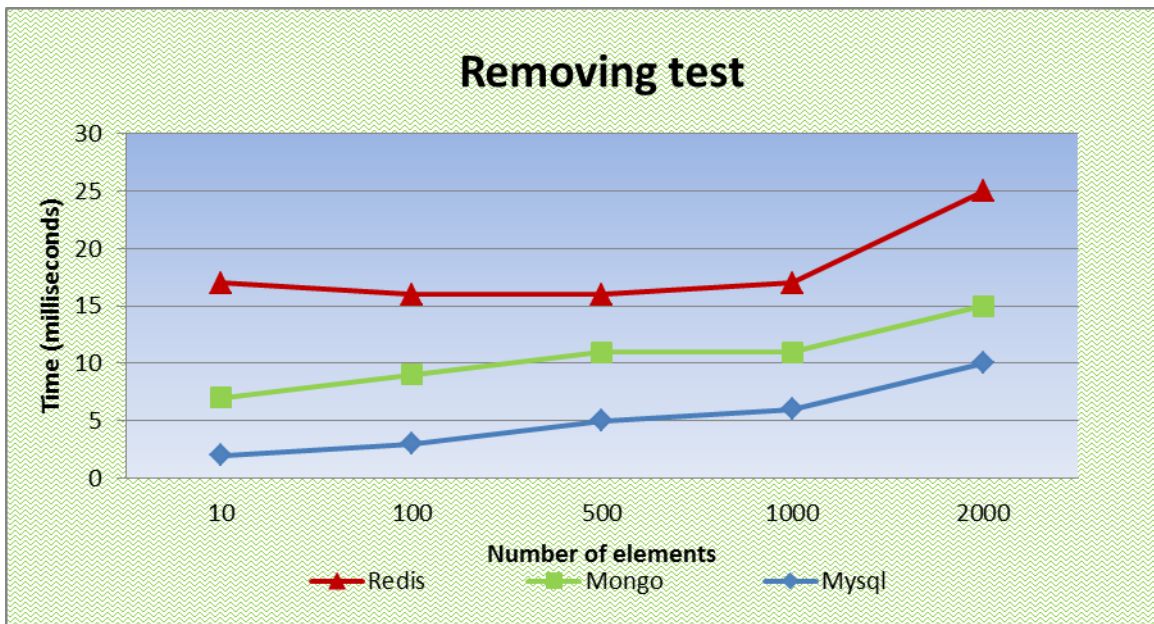


Figure 7-15: Removing test

Aggregation test

The last test is the aggregation test. MySQL and Mongo were tested in aggregation procedure. By the time we made the tests there was not support for aggregation in Redis. For aggregation we used the "group by" command in Mysql and the meteorhacks:aggregate library in Mongo. At first we group by locality and then we summarize the results of each group. Finally we find the average value for each group. The aggregation pipeline provides an alternative to map-reduce and may be the preferred solution for aggregation tasks where the complexity of map-reduce may be unwarranted. Figure 7.16 shows the results of aggregation test. One more time MySQL has better times than Mongo. In both databases it takes more time to aggregate small datasets than large ones.

Before aggregation procedure there is an insertion procedure

```
aggregateMongoFunc: function(){
    var time=[];
    for(var i=0;i<times.length;i++){
        insertIntoMongo(times[i]);
        var startTime=new Date()-0;
        var pipeline = [
            {
                $group: {
                    _id: {locality: "$locality"},
                    total: {$sum: 1}
                }
            }
        ];
        var avgResult = testCollection.aggregate(pipeline);
        time[i]=new Date()-startTime;
    }
    return time;
}
```

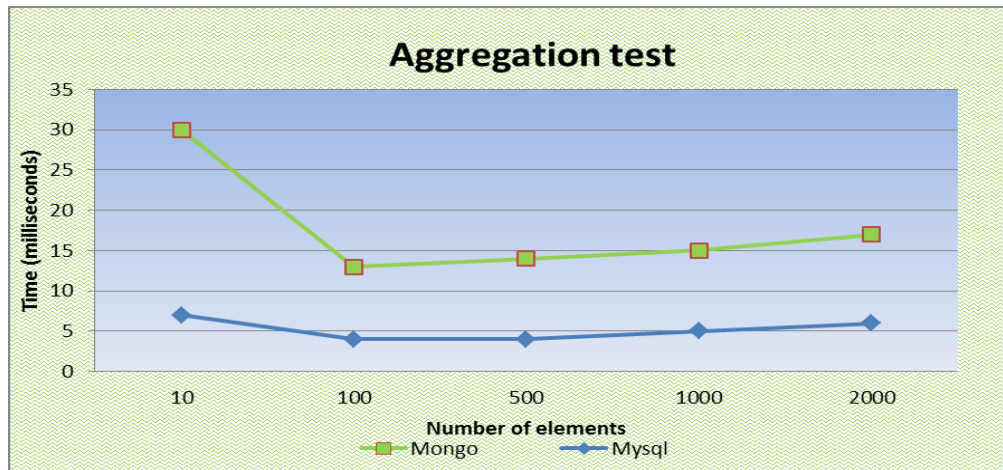


Figure 7-16: Aggregation test

CHAPTER 8: CONCLUSION & FUTURE WORK

In this thesis we created a web middleware platform which is interfaced with the real world through various mobile sensors. HTML5 gives the capability to the developers to interact with mobile and desktop sensors with a web manner. HTML5 sensor APIs offer access to the device hardware with only some lines of Javascript code. Hence, the fundamental functionality of the platform is to gather, process and visualize the initial information that acquires from the device sensors. The scope of the platform is to group and graphically present the retrieved data following statistical processing. The uniqueness of the platform is that it solely uses HTML5 APIs to deliver real-time sensors data to the end users. Google maps and rich-interactive charts are some of the visualization ways that apply. The platform has more advanced capabilities such as a collecting and accessing API which integrates with the database. All these web services are offered to the end users via a 3rd party component.

The sensor data are a very important source of information. An appropriate analysis can offer a better understanding of the environment that surrounds us. Hence, a real-time analysis based on the events from sensors can be a serious help for urban community. Of course, raw sensor data are just numbers. It needs to process, analyze and store the information in some form with human value for future use. A graph or an aggregation can offer multi- information.

Our middleware platform is an intergraded solution for Internet of Things equipment offering services from its components for gathering sensor data along with statistical and visualization services. It contains 3 separated application layers: Presentation layer (split in two parts: Client and 3rd party page), logic layer and database layer. Each part has its own logic which is built in a separate environment following the generic principles of multi-tier software engineering architectures. The components communicate and share data between them with state-of-the-art bidirectional communication protocols such as websockets. The platform also contains an application server, namely the Meteor, which acts as a synchronization layer that keeps client and server databases up to date. Although the client part of platform has been built for users equipped with mobile devices, it can be easily extended to also include devices of any size equipped with ambient and GPS sensors. The only software requirement from such IoT devices is to support HTTP for the transmission of sensor data to our server.

As we refer above, the purpose of the platform is to transfer sensor data from clients to visitors in a real-time manner. For that reason we evaluate the response time of the platform by doing performance tests in various tasks. Performance tests measure the time it takes for delivering sensor data from the initial component (client) to the destination component (visitor). Also, we measure the time it takes for the visitors' page to retrieve data from the database through the server component. The performance tests made under various access communication networks such as Wi-Fi, 2G and 3G. All the networks had similar times with very small differences between them, with Wi-Fi to perform faster followed by 3G and 2G.

Evaluation tests had a second section that measured the speed performance of 3 well-known databases: MySQL, MongoDB and Redis. We tested the databases in basic operations such as read and write, but also in most complicated ones such as aggregation and read with sorting. These tests can be characterized as a competitive test between two different kinds of databases, SQL (MySQL) and NoSQL (MongoDB and Redis), over big sensor data volumes. A general comment that outcome from the results is that all the databases (MySQL, Mongo and Redis) had almost equal times in case of small number of entries, but when this number was increased, MySQL and Mongo had significant superiority over Redis. It worth mentioning that the initial thought of Meteor developers was to use MySQL as the default database when they start creating Meteor.

This platform has a variety of future applications and uses. Using statistical languages like "R", it can help for performing advanced statistical analysis of the measurements and then using them to export conclusions for future changes in society or the environment. A visualization service like google maps can help to create a world map for storing gathered sensor data and presenting statistics. A similar project that records your voice and then adds it to the map has been built in [16]. By using "R" it can create a specific dialect for every place of the world. Also, "R" can be used as analysis tool for more accurate mapping information. For example in [17] it describes the usage of "R" along with Shiny to help farmers and managers to achieve better yields from their crops.

Improvements in the application can be made in many areas. We can add to the client component a personal account that will keep the personal statistics for every user. There it can store time and location statistics with the personal exposure to noise and analogically to the statistics it will reward with budes and other prices the user. Another improvement that we can

make is to insert additional information to the GUI of the client like comments and reviews which will help him to take an easier decision.

Also, the client component can be built with one of the highly promising HTML5 hybrid frameworks like Ionic [162], mobile angular UI [163], Intel XDK [164], Titanium [165] or Phonegap [166]. HTML5 mobile development is evolving day by day and there are always new options emerging. Finally, in the field of usability engineering we can make our web applications responsive to desktop or mobile devices. An optimal viewing experience like easy reading and navigation with a minimum of resizing and scrolling is a desired step.

REFERENCES

- [1] Gubbi, Jayavardhana, et al. "Internet of Things (IoT): A vision, architectural elements, and future directions." *Future Generation Computer Systems* 29.7 (2013): 1645-1660.
- [2] WikiPedia, Internet Of Things, http://en.wikipedia.org/wiki/Internet_of_Things, as visited on March 10, 2015
- [3] Panagiotakis, Spyros, et al. "Towards Ubiquitous and Adaptive Web-Based Multimedia Communications via the Cloud." *Resource Management of Mobile Cloud Computing Networks and Environments* (2015): 307.
- [4] Guinard, Dominique, Vlad Trifa, and Erik Wilde. "A resource oriented architecture for the web of things." *Internet of Things (IOT)*, 2010. IEEE, 2010.
- [5] Tilak, Sameer. "Real-world deployments of participatory sensing applications: Current trends and future directions." *International Scholarly Research Notices* 2013 (2013).
- [6] Ganti, Raghu K., Fan Ye, and Hui Lei. "Mobile crowdsensing: current state and future challenges." *Communications Magazine*, IEEE 49.11 (2011): 32-39.
- [7] Sen, Sougata, et al. "The case for cloud-enabled mobile sensing services." *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*. ACM, 2012.
- [8] Alamri, Atif, et al. "A survey on sensor-cloud: architecture, applications, and approaches." *International Journal of Distributed Sensor Networks* 2013 (2013).
- [9] WikiPedia, Multitier architecture, http://en.wikipedia.org/wiki/Multitier_architecture, as visited on March 10, 2015
- [10] Lee, Uichin, et al. "Mobeyes: smart mobs for urban monitoring with a vehicular sensor network." *Wireless Communications*, IEEE 13.5 (2006): 52-57.
- [11] Anger, Philipp J. *Integration of Mobile Devices in Collaborative Web Applications*. na, 2013.
- [12] Panagiotakis Spyros, *Browser Platform Assessment for X3Dom Graphics'. Rendering Capabilities IJCIT*, 2014.
- [13] Murray, Scott. *Interactive data visualization for the Web*. " O'Reilly Media, Inc.", 2013.
- [14] Kapetanakis, Kostas, Spyros Panagiotakis, and Athanasios G. Malamos. "HTML5 and WebSockets; challenges in network 3D collaboration." *Proceedings of the 17th Panhellenic Conference on Informatics*. ACM, 2013.

- [15] Green Ido. Web Workers, Multithreaded Programms in JavaScript. " O'Reilly Media, Inc.", 2012.
- [16] R Video tutorial for Spatial Statistics, <http://r-video-tutorial.blogspot.gr/2013/07/interfacing-r-and-google-maps.html>, as visited on March 10, 2015
- [17] Jahanshiri, Ebrahim, and Abdul Rashid Mohd Shariff. "Developing web-based data analysis tools for precision farming using R and Shiny." IOP Conference Series: Earth and Environmental Science. Vol. 20. No. 1. IOP Publishing, 2014.
- [18] Amundsen, Mike. Building Hypermedia APIs with HTML5 and Node. " O'Reilly Media, Inc.", 2011.
- [19] Essa, Irfan A. "Ubiquitous sensing for smart and aware environments."Personal Communications, IEEE 7.5 (2000): 47-49.
- [20] Campbell, Andrew T., et al. "The rise of people-centric sensing." Internet Computing, IEEE 12.4 (2008): 12-21.
- [21] Pintos, Antonio, et al. Connecting smart things through web services orchestrations. Springer Berlin Heidelberg, 2010.
- [22] Lane, Nicholas D., et al. "A survey of mobile phone sensing." Communications Magazine, IEEE 48.9 (2010): 140-150.
- [23] WikiPedia, Geocoding process, <http://en.wikipedia.org/wiki/Geocoding/>, as visited on March 10, 2015
- [24] The Google Geocoding API, <https://developers.google.com/maps/documentation/geocoding/>, as visited on March 10, 2015
- [25] Svennerberg, Gabriel. Beginning Google Maps API 3. Apress, 2010.
- [26] Smus, Boris. Web Audio API. " O'Reilly Media, Inc.", 2013.
- [27] Google Maps JavaScript API, <https://developers.google.com/maps/documentation/javascript/examples/geocoding-reverse>, as visited on March 10, 2015
- [28] The WebSocket API - W3C, <https://dvcs.w3.org/hg/audio/raw-file/tip/webaudio/specification.html>, as visited on March 10, 2015
- [29] HTML5rocks, Getting Started with Web Audio API, <http://www.html5rocks.com/en/tutorials/webaudio/intro/>, as visited on March 10, 2015

- [30] HTML5rocks, WebRTC, <http://www.html5rocks.com/en/tutorials/webrtc/basics/>, as visited on March 10, 2015
- [31] Maisonneuve, Nicolas, Matthias Stevens, and Bartek Ochab. "Participatory noise pollution monitoring using mobile phones." *Information Polity* 15.1 (2010): 51-71.
- [32] D'Hondt, Ellie, and Matthias Stevens. "Participatory noise mapping." *Demo Proceedings of the 9th International Conference on Pervasive*. 2011.
- [33] D'Hondt, Ellie, Matthias Stevens, and An Jacobs. "Participatory noise mapping works! An evaluation of participatory sensing as an alternative to standard techniques for environmental monitoring." *Pervasive and Mobile Computing* 9.5 (2013): 681-694.
- [34] Maisonneuve, Nicolas, et al. "NoiseTube: Measuring and mapping noise pollution with mobile phones." *Information Technologies in Environmental Engineering*. Springer Berlin Heidelberg, 2009. 215-228.
- [35] Maisonneuve, Nicolas, et al. "Citizen noise pollution monitoring." *Proceedings of the 10th Annual International Conference on Digital Government Research: Social Networks: Making Connections between Citizens, Data and Government*. Digital Government Society of North America, 2009.
- [36] Drosatos, George, et al. "A privacy-preserving cloud computing system for creating participatory noise maps." *Computer Software and Applications Conference (COMPSAC), 2012 IEEE 36th Annual*. IEEE, 2012.
- [37] Gsmarena, Samsung galaxy S5 specs, http://www.gsmarena.com/samsung_galaxy_s5-6033.php, as visited on March 10, 2015
- [38] Geolocation tutorial, HTML5rocks, http://www.html5rocks.com/en/tutorials/geolocation/trip_meter/, as visited on March 10, 2015
- [39] Movable type scripts, <http://www.movable-type.co.uk/scripts/latlong.html> , as visited on March 10, 2015
- [40] Wikipedia, Haversine formula, http://en.wikipedia.org/wiki/Haversine_formula, as visited on March 10, 2015
- [41] Consolvo, Sunny, et al. "Activity sensing in the wild: a field trial of ubifit garden." *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2008.
- [42] Miluzzo, Emiliano, et al. "Sensing meets mobile social networks: the design, implementation and evaluation of the cenceme application." *Proceedings of the 6th ACM conference on Embedded network sensor systems*. ACM, 2008.

- [43] Mun, Min, et al. "PEIR, the personal environmental impact report, as a platform for participatory sensing systems research." Proceedings of the 7th international conference on Mobile systems, applications, and services. ACM, 2009.
- [44] Thiagarajan, Arvind, et al. "VTrack: accurate, energy-aware road traffic delay estimation using mobile phones." Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems. ACM, 2009.
- [45] UC Berkeley/Nokia/NAVTEQ, "Mobile Millennium", <http://traffic.berkeley.edu/>, as visited on March 10, 2015
- [46] Dutta, Prabal, et al. "Common sense: participatory urban sensing using a network of handheld air quality monitors." Proceedings of the 7th ACM conference on embedded networked sensor systems. ACM, 2009.
- [47] Burke, Jeffrey A., et al. "Participatory sensing." Center for Embedded Network Sensing (2006).
- [48] Phoneygap website, <http://phonegap.com/>, as visited on March 10, 2015.
- [49] Intel XDK developer, <https://software.intel.com/en-us/html5/tools>, as visited on March 10, 2015
- [50] Enyojs, HTML5 Framework, <http://enyojs.com/>, as visited on March 10, 2015
- [51] Mosync, application framework, <http://www.mosync.com/>, as visited on March 10, 2015
- [52] Wikipedia, Node.js, <http://en.wikipedia.org/wiki/Node.js>, as visited on March 10, 2015
- [53] W3C, Geolocation API, <http://www.w3.org/TR/geolocation-API/>, as visited on March 10, 2015
- [54] W3C, Orientation event, <http://www.w3.org/TR/orientation-event/>, as visited on March 10, 2015
- [55] W3C, Battery status, <http://www.w3.org/TR/battery-status/>, as visited on March 10, 2015
- [56] W3C, Proximity, <http://www.w3.org/TR/proximity/>, as visited on March 10, 2015
- [57] W3C, Ambient light, <http://www.w3.org/TR/ambient-light/>, as visited on March 10, 2015
- [58] W3C, Network information API, <http://www.w3.org/TR/netinfo-api/>, as visited on March 10, 2015
- [59] W3C, Resource timing, <http://www.w3.org/TR/resource-timing/>, as visited on March 10, 2015

- [60] W3C, High resolution API, <http://www.w3.org/TR/hr-time/>, as visited on March 10, 2015
- [61] W3C, User timing, <http://www.w3.org/TR/user-timing/>, as visited on March 10, 2015
- [62] W3C, Media capture and streams, <http://www.w3.org/TR/mediacapture-streams/>, as visited on March 10, 2015
- [63] W3C, Vibration, <http://www.w3.org/TR/vibration/>, as visited on March 10, 2015
- [64] W3C, Fullscreen, <http://www.w3.org/TR/fullscreen/>, as visited on March 10, 2015
- [65] W3C, Page Visibility, <http://www.w3.org/TR/page-visibility/>, as visited on March 10, 2015
- [66] W3C, Web Workers, <http://www.w3.org/TR/workers/>, as visited on March 10, 2015
- [67] W3C, Web storage, <http://www.w3.org/TR/webstorage/>, as visited on March 10, 2015
- [68] W3C, Web intents, <http://www.w3.org/TR/web-intents/>, as visited on March 10, 2015
- [69] Campbell, Andrew T., et al. "People-centric urban sensing." Proceedings of the 2nd annual international workshop on Wireless internet. ACM, 2006.
- [70] Yan, Zhixian, and Dipanjan Chakraborty. "Semantics in Mobile Sensing." Synthesis Lectures on the Semantic Web: Theory and Technology 4.1 (2014): 1-143.
- [71] Ra, Moo-Ryong, et al. "Medusa: A programming framework for crowd-sensing applications." Proceedings of the 10th international conference on Mobile systems, applications, and services. ACM, 2012.
- [72] Dimov Daniel, Crowdsensing: State of the Art and Privacy Aspects, <http://resources.infosecinstitute.com/crowdsensing-state-art-privacy-aspects/>
- [73] IDC, <http://www.idc.com/>, as visited on March 10, 2015
- [74] Lewis, Daniel. "What is web 2.0?." Crossroads 13.1 (2006): 3-3.
- [75] Robbins, Jennifer Niederst. HTML5 Pocket Reference. " O'Reilly Media, Inc.", 2013.
- [76] Lengstorf, Jason, and Phil Leggetter. Real Time Web Apps. United States of America: Apress, 2013.
- [77] Freeman, Adam. The Definitive Guide to HTML5. Apress, 2011.

- [78] Pournajaf, Layla, et al. A survey on privacy in mobile crowd sensing task management. Technical Report TR-2014-002, Department of Mathematics and Computer Science, Emory University, 2014.
- [79] Saha, Debashis, and Amitava Mukherjee. "Pervasive computing: a paradigm for the 21st century." *Computer* 36.3 (2003): 25-31.
- [80] Talasila, Manoop, Reza Curtmola, and Cristian Borcea. "Alien vs. Mobile User Game: Fast and Efficient Area Coverage in Crowdsensing."
- [81] Martí, Irene Garcia, et al. "Mobile application for noise pollution monitoring through gamification techniques." *Entertainment Computing-ICEC 2012*. Springer Berlin Heidelberg, 2012. 562-571.
- [82] Wikipedia, Gamification, <http://en.wikipedia.org/wiki/Gamification>, as visited on March 10, 2015
- [83] Zichermann, Gabe, and Christopher Cunningham. *Gamification by design: Implementing game mechanics in web and mobile apps*. " O'Reilly Media, Inc.", 2011.
- [84] Gamification Wiki, Nike, <http://badgeville.com/wiki/Nike>, as visited on March 10, 2015
- [85] Gamification Wiki, Starbucks, <http://badgeville.com/wiki/mystarbucksrewards>, as visited on March 10, 2015
- [86] Evoke, <http://www.urgentevoke.com/>, as visited on March 10, 2015
- [87] Stackoverflow, <http://stackoverflow.com/>, as visited on March 10, 2015
- [88] Richard, B.: *Hearts, clubs, diamonds, spades: Players who suits MUDs* (1996), <http://www.mud.co.uk/richard/hcds.htm>, as visited on March 10, 2015
- [89] Android Developer, http://developer.android.com/guide/topics/sensors/sensors_overview.html, as visited on March 10, 2015
- [90] Rosi, Alberto, et al. "Social sensors and pervasive services: Approaches and perspectives." *Pervasive Computing and Communications Workshops (PERCOM Workshops)*, 2011 IEEE International Conference on. IEEE, 2011.
- [91] Rosi, Alberto, et al. "Social sensors and pervasive services: Approaches and perspectives." *Pervasive Computing and Communications Workshops (PERCOM Workshops)*, 2011 IEEE International Conference on. IEEE, 2011.

- [92] Guo, Bin, et al. "From participatory sensing to mobile crowd sensing." Pervasive Computing and Communications Workshops (PERCOM Workshops), 2014 IEEE International Conference on. IEEE, 2014.
- [93] Goldman, Jeffrey, et al. "Participatory Sensing: A citizen-powered approach to illuminating the patterns that shape our world." Foresight & Governance Project, White Paper (2009): 1-15.
- [94] Goldman, Jeffrey, et al. "Participatory Sensing: A citizen-powered approach to illuminating the patterns that shape our world." Foresight & Governance Project, White Paper (2009): 1-15.
- [95] Mohan, Prashanth, Venkata N. Padmanabhan, and Ramachandran Ramjee. "Nericell: rich monitoring of road and traffic conditions using mobile smartphones." Proceedings of the 6th ACM conference on Embedded network sensor systems. ACM, 2008.
- [96] Eriksson, Jakob, et al. "The pothole patrol: using a mobile sensor network for road surface monitoring." Proceedings of the 6th international conference on Mobile systems, applications, and services. ACM, 2008.
- [97] Bhoraskar, Ravi, et al. "Wolverine: Traffic and road condition estimation using smartphone sensors." Communication Systems and Networks (COMSNETS), 2012 Fourth International Conference on. IEEE, 2012.
- [98] Reddy, Sasank, et al. "Biketastic: sensing and mapping for better biking." Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. ACM, 2010.
- [99] Ryder, Jason, et al. "Ambulation: A tool for monitoring mobility patterns over time using mobile phones." Computational Science and Engineering, 2009. CSE'09. International Conference on. Vol. 4. IEEE, 2009.
- [100] Hicks, John, et al. "AndWellness: an open mobile system for activity and experience sampling." Wireless Health 2010. ACM, 2010.
- [101] Lane, Nicholas D., et al. "Bewell: A smartphone application to monitor, model and promote wellbeing." 5th international ICST conference on pervasive computing technologies for healthcare. 2011.
- [102] Von Kaenel, Michael, Philipp Sommer, and Roger Wattenhofer. "Ikarus: large-scale participatory sensing at high altitudes." Proceedings of the 12th Workshop on Mobile Computing Systems and Applications. ACM, 2011.
- [103] Hasenfratz, David, et al. "Participatory air pollution monitoring using smartphones." Mobile Sensing (2012).
- [104] Hull, Bret, et al. "CarTel: a distributed mobile sensor computing system." Proceedings of the 4th international conference on Embedded networked sensor systems. ACM, 2006.

- [105] Eisenman, Shane B., et al. "Metrosense project: People-centric sensing at scale." Workshop on World-Sensor-Web (WSW 2006), Boulder. 2006.
- [106] Eisenman, Shane B., et al. "BikeNet: A mobile sensing system for cyclist experience mapping." ACM Transactions on Sensor Networks (TOSN) 6.1 (2009): 6.
- [107] Andreas Krause , Eric Horvitz , Aman Kansal , Feng Zhao, Toward Community Sensing, Proceedings of the 7th international conference on Information processing in sensor networks, p.481-492, April 22-24, 2008
- [108] Kim, Sunyoung, et al. "Creek watch: pairing usefulness and usability for successful citizen science." Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. ACM, 2011.
- [109] Reddy, Sasank, et al. "Image browsing, processing, and clustering for participatory sensing: lessons from a DietSense prototype." Proceedings of the 4th workshop on Embedded networked sensors. ACM, 2007.
- [110] Ji, Rongrong, et al. "Mining city landmarks from blogs by graph modeling." Proceedings of the 17th ACM international conference on Multimedia. ACM, 2009.
- [111] Zhao, Qiankun, et al. "Event detection from evolution of click-through data." Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2006.
- [112] Perrig, Adrian, et al. "SPINS: Security protocols for sensor networks." Wireless networks 8.5 (2002): 521-534.
- [113] Zhu, Sencun, Sanjeev Setia, and Sushil Jajodia. "LEAP+: Efficient security mechanisms for large-scale distributed sensor networks." ACM Transactions on Sensor Networks (TOSN) 2.4 (2006): 500-528.
- [114] Karlof, Chris, and David Wagner. "Secure routing in wireless sensor networks: Attacks and countermeasures." Ad hoc networks 1.2 (2003): 293-315.
- [115] Yin, Changqing, et al. "Secure routing for large-scale wireless sensor networks." Communication Technology Proceedings, 2003. ICCT 2003. International Conference on. Vol. 2. IEEE, 2003.
- [116] Deng, Jing, Richard Han, and Shivakant Mishra. "A performance evaluation of intrusion-tolerant routing in wireless sensor networks." Information Processing in Sensor Networks. Springer Berlin Heidelberg, 2003.
- [117] Chan, Haowen, Adrian Perrig, and Dawn Song. "Secure hierarchical in-network aggregation in sensor networks." Proceedings of the 13th ACM conference on Computer and communications security. ACM, 2006.

- [118] Chana, Haowen, et al. "SI A: Secure information aggregation in sensor networks." Security of Ad-hoc and Sensor Networks (2007): 69.
- [119] Tang, Karen P., et al. "Putting people in their place: an anonymous and privacy-sensitive approach to collecting sensed data in location-based applications." Proceedings of the SIGCHI conference on Human Factors in computing systems. ACM, 2006.
- [120] Shin, Minho, et al. "AnonySense: A system for anonymous opportunistic sensing." Pervasive and Mobile Computing 7.1 (2011): 16-30.
- [121] Christin, Delphine, et al. "A survey on privacy in mobile participatory sensing applications." Journal of Systems and Software 84.11 (2011): 1928-1946.
- [122] Dingedine, Roger, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. Naval Research Lab Washington DC, 2004.
- [123] Ghinita, Gabriel. "Privacy for location-based services." Synthesis Lectures on Information Security, Privacy, & Trust 4.1 (2013): 1-85.
- [124] Sabari website, [http://www.sabarimarketing.com/blog/html5-the-fifth-revision-of-the-hypertext-markup language-html](http://www.sabarimarketing.com/blog/html5-the-fifth-revision-of-the-hypertext-markup-language-html) , as visited on March 10, 2015
- [125] W3C, HTML5, <http://www.w3.org/2014/10/html5-rec.html.en>, as visited on March 10, 2015
- [126] W3C, HTML5 recommendation, <http://www.w3.org/html/wg/drafts/html/master/>, as visited on March 10, 2015
- [127] WikiPedia, HTML5, <http://en.wikipedia.org/wiki/HTML5>, as visited on March 10, 2015
- [128] Visualizing raw data, <http://scottizu.wordpress.com/2014/06/24/visualizing-raw-data-samples-from-a-microphone/>, as visited on March 10, 2015
- [129] HTML5rocks, File APIs, <http://www.html5rocks.com/en/tutorials/file/dndfiles/>, as visited on March 10, 2015
- [130] Moniruzzaman, A. B. M., and Syed Akhter Hossain. "Nosql database: New era of databases for big data analytics-classification, characteristics and comparison." arXiv preprint arXiv:1307.0191 (2013).
- [131] Scaledb, data architectures for the Internet of Things, <http://www.scaledb.com/internet-things-database.php#architecture>, as visited on March 10, 2015
- [132] Heroku, NoSQL databases, <https://blog.heroku.com/archives/2010/7/20/nosql>, as visited on March 10, 2015

- [133] <http://kkovacs.eu/cassandra-vs-mongodb-vs-couchdb-vs-redis>, as visited on March 10, 2015
- [134] Websocket official webpage, <https://www.websocket.org/>, as visited on March 10, 2015
- [135] HTML5rocks, websockets, <http://www.html5rocks.com/en/tutorials/websockets/basics/>, as visited on March 10, 2015
- [136] W3C, Standards for Web Applications on Mobile, <http://www.w3.org/2014/04/mobile-web-app-state/>, as visited on March 10, 2015
- [137] W3C, FileAPI, <http://www.w3.org/TR/FileAPI/>, as visited on March 10, 2015
- [138] HTML5rocks, Getusermedia, <http://www.html5rocks.com/en/tutorials/getusermedia/intro/>, as visited on March 10, 2015
- [139] W3C, Websockets, <http://www.w3.org/TR/2012/CR-websockets-20120920/>, as visited on March 10, 2015
- [140] IETF, Websockets, <https://tools.ietf.org/html/rfc6455>, as visited on March 10, 2015
- [141] WikiPedia, Geo-fence, <http://en.wikipedia.org/wiki/Geo-fence>, as visited on March 10, 2015
- [142] Google developer website, Geofences, <https://developers.google.com/maps/documentation/tracks/geofences>, as visited on March 10, 2015
- [143] WikiPedia, Meteor framework, [http://en.wikipedia.org/wiki/Meteor_\(web_framework\)](http://en.wikipedia.org/wiki/Meteor_(web_framework)), as visited on March 10, 2015
- [144] Meteor official website, <https://www.meteor.com/>, as visited on March 10, 2015
- [145] Discovermeteor, <https://www.discovermeteor.com/blog/understanding-meteor-publications-and-subscriptions/>, as visited on March 10, 2015
- [146] WikiPedia, JSON, <http://en.wikipedia.org/wiki/JSON>, as visited on March 10, 2015
- [147] Google maps official webpage, <https://www.google.gr/maps/>, as visited on March 10, 2015
- [148] Google maps website, <https://developers.google.com/maps/>, as visited on March 10, 2015
- [149] WikiPedia, BSON, <http://en.wikipedia.org/wiki/BSON>, as visited on March 10, 2015
- [150] BSON official webpage, <http://bsonspec.org/>, as visited on March 10, 2015

- [151] <http://www.labnol.org/internet/web-3-concepts-explained/8908/>, as visited on March 10, 2015
- [152] Wikipedia, GeoJSON, <http://en.wikipedia.org/wiki/GeoJSON>, as visited on March 10, 2015
- [153] GeoJSON official webpage, <http://geojson.org/>, as visited on March 10, 2015
- [154] GeoJSON live tutorial, <http://geojson.io/#map=2/20.0/0.0>, as visited on March 10, 2015
- [155] Wikipedia, Ext_JS framework, http://en.wikipedia.org/wiki/Ext_JS, as visited on March 10, 2015
- [156] Sensorplatform, <http://www.sensorplatforms.com/sensors-html5/>, as visited on March 10, 2015
- [157] Ext JS, documentation, <http://docs.sencha.com/extjs/4.2.1/#!/guide/charting>, as visited on March 10, 2015
- [158] Blog, D3 and X3Dom example, <http://bl.ocks.org/camio/5087116>, as visited on March 10, 2015
- [159] W3C, HTML5, <http://www.w3.org/html/wg/drafts/html/master/>, as visited on March 10, 2015
- [160] http://www.web3d.org/wiki/index.php/X3D_and_HTML5, as visited on March 16, 2015.
- [161] BEHR, Johannes, et al. X3DOM: a DOM-based HTML5/X3D integration model. In: Proceedings of the 14th International Conference on 3D Web Technology. ACM, 2009. p. 127-135.
- [162] Ionic, framework for HTML5 application, <http://ionicframework.com/>, as visited on March 10, 2015
- [163] Angular framework, <http://mobileangularui.com/>, as visited on March 10, 2015
- [164] Intel XDK framework, <https://software.intel.com/en-us/html5/tools>, as visited on March 10, 2015
- [165] Appcelerator mobile application platform, <http://www.appcelerator.com/titanium/>, as visited on March 10, 2015
- [166] Phonegap official page, <http://phonegap.com/>, as visited on March 10, 2015
- [167] Github example, <https://codeload.github.com/edse/puzzle/zip/master>

- [168] Atmospherejs, Meteorjs packages, <https://atmospherejs.com/aldeed/geocoder>
- [169] Sony lamporatory in Paris official website, <http://www.csl.sony.fr/>
- [170] Noisetube project official website, <http://www.noisetube.net/>
- [171] Reactive MySQL for Meteor, <https://github.com/numtel/meteor-mysql>
- [172] Redis Livedata, <https://github.com/meteor/redis-livedata>
- [173] Wingate, proxyserver, <http://www.wingate.com/download/wingate/download.php>