MINISTRY OF EDUCATION

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA, ROMANIA
FACULTY OF AUTOMATION AND COMPUTER SCIENCE
COMPUTER SCIENCE DEPARTMENT

MMNS: A MIDDLEWARE BASED APPROACH FOR A MOBILE METEOROLOGICAL NOTIFICATION SYSTEM

LICENSE THESIS

Graduate: **Alexandru Stefan LUPU**

Supervisor: **Assist. Prof. Eng. Cosmina IVAN**

2018

Table of Contents

Chapter 1. Introduction

1.1. Project context

In the information age that we live in, it is more and more common that the real world is in some way linked with the virtual world. Simple things such as maps, shops and even household objects have found an abstraction that allows regular people to have access to informations or services in relation to them faster, from computers or even hand-held devices.

In order to provide services that are of interest to the users, most systems built nowadays will gather some information regarding the context of the user, information that will allow them to approach the user in a most suitable manner and providing useful information to him. The information building up the context however can grow very large as it consists of numerous parameters. However most of the time a subset of these parameters will suffice for a provider in order to profile a user and decide on what information is of interest to him.

The context of a user can also contain factors that cannot be observed by normal means, and are not immediately made available to him in a natural manner - one cannot gather enough information about the surrounding environment by sight, touch, smell, taste or hearing. For some applications it is important to provide information like this to the user in order to make him aware of his context.

The subset of context parameters that this paper focuses to bring to the user is weather, or more exactly information regarding the current weather at the user location. Weather phenomena can impact the user in multiple ways, such as awareness for outdoor activities or alerts regarding dangerous weather conditions, to name a few.

1.2. Motivation

Although information such as time, location, activity and interests, which may have an impact on the context of people, is mostly available to be looked up on the Internet, changes that occur can be of immediate importance to the user, but will only be noticed upon direct interaction. Weather for instance can have an impact on outdoor activities or traffic, and changes that may occur in the current weather cannot always be forecasted accurately. Even when designing an application to notify its user of a change that has occurred in the current weather, it will eventually be necessary to monitor the current weather, meaning that it simply shifts the responsibility of checking the current weather regularly from the user to the application.

MMNS is designed to be the bridge between *information-on-demand* and *information-when-changed*, monitoring the information on behalf of all interested clients, wether human or software. This system eliminates the needs of having infinite polling loops in an application for the purpose of monitoring the data provided by a REST API, freeing the resources that would otherwise be wasted.

1.3. Content

Chapter 2 presents the objectives and specifications of the developed project. In this chapter the problem is thoroughly explained, and based on it, grounds for functional

and non-functional requirements will be established. The non-functional requirements and why they are important are also explained in that chapter.

Chapter 3 approaches some of the literature that is currently available on the subject of user context and awareness, weather information systems and the mechanism behind the method of communication used in this project, namely the Publish-Subscribe architectural pattern, generic ways of implementation and important aspects to be taken into consideration when developing a system based on this pattern.

Chapter 4 discusses the mechanisms of a publish/subscribe system, such as design patterns commonly used within this architectural style and data structures to hold participants of the system. This chapter also performs an analysis in communication methods that are candidates to be used in the system, and presents a conceptual architecture and some use-cases of the system.

Chapter 5 focuses on the implementation of the system, presenting the system architecture and detailing its components. This chapter also describes the implementation of the client library that is used to integrate the functionality of MMNS in the client application, and a proof of concept mobile application that integrates the functionality of MMNS in order to deliver weather updates to the user of the application.

Chapter 6 presents the testing that has been performed on the components of the system and the results of the evaluation of communication between them.

Chapter 7 presents the usage of the MMNS system and MMNS client library, as well as deployment of systems built as part of this project.

Chapter 2. Project Objectives and Specification

2.1. Main Goal

The main goal of this thesis project is to design and implement a middleware that can provide its users with information regarding the current weather at their location as close as possible to the time that changes occur, without the need of the users to ask for it.

Such a system needs to be able to serve a growing number of users potentially at the same time. It should also provide data from a trusted source, such as a service that the users would alternatively use to get information about the current weather. In order to provide the information as close as possible to the moment in which changes are recorded, the middleware needs to monitor the data periodically every quantum of time in order to identify changes. These changes must then be communicated to the users of the system without them initiating the communication by a poll or request. For this to happen, the users should also be able to provide the means for communication, which highlights the need for security in the system: the communication channel that the user can be reached by should not be able to be used by means of hacking to harm or misinform the user.

More generic applications can be found for the main goal, for any data that is subject to change and can be monitored in order to inform interested users. For this project however, the limitation of focusing on current weather data has been placed, and the implemented software will stand as proof of concept for this specific field.

2.2. Secondary Goals

First, in order to make use of the implemented system, a secondary goal is to design and implement a library that can be used by the users of the middleware in order to make it easier for them to communicate with it and use it programatically in order to accomplish more complex functionality that is of their own design.

Another goal is to design and implement a full-stack mobile application made up of server-side and native client application that will use the implemented middleware by integrating the library mentioned above in order to deliver updates in the current weather in the form of push notifications to a smartphone.

2.3. Problem Specification

When information regarding a closed system that is available on the Internet needs to be obtained only when it changes, the currently preferred method of data acquisition presents some problems. This is due to the fact that information on the Web is made available on demand – it has to be requested, meaning that if a user is interested to notice changes in the state of a system, he will need to investigate the data regarding it even if it has not yet changed in order to find differences.

The most common way of accessing data nowadays is by polling a REST API. Most online services are accessible by the use of this method. While this method is widely accepted and currently most used, few issues in some use-cases accompany it.

The problem tackled in this thesis refers to the inefficiency of REST API in the context of real-time monitoring of current weather data. If a good Internet connection is

provided, the communication between the client and API can be completed in as little as a few milliseconds, but the API will respond with redundant data most of the time.

The problem of	REST API polling for weather information
affects	service providers, service clients and end-users
the impact of which is	a cram on the side of the service, irrelevant results on the side of service clients
a successful solution would be	able to deliver information to the clients as soon as it changes, without it having to be requested

The problem when developing a system for monitoring a state accessible through a REST API is the necessity of developing an automated component for polling the service, interpreting each separate response in order to identify changes in the state of the monitored system, and optionally implementing time-related restrictions (such as a limited number of requests, or the knowledge of data refresh time on the backend). Each custom application that relies on the same API will have to develop its own solution to fit these problems, which will usually result in many clients requesting for the same data in an infinite loop. This represents in itself another problem.

Another drawback of this method of information acquisition is that the response of weather services is relative to the caller location, which has a large enough span to cover entire cities. Because of this, multiple instances of the same application used by a large number of users can lead to a multitude of processes polling the same service, with the same expected result, in an infinite loop.

The solution proposed for the specified problems are to build a system that deals with them in a way as efficient as possible, which can be implemented by any client, and feed them the monitored data as it changes. The system acts as a bridge between clients and API, grouping all requests by clustering them as from a singular endpoint. In order to obtain the updates, the clients only need to register once to the system and then wait for updates in the form of notifications.

2.4. Project Objectives

In order to set the objectives of this project in an organised manner, the SMART criterion is used. As thoroughly defined in [1], the SMART criteria is used for setting objectives by project managers, in many fields, including Software Project Management and it is an acronym that stands for Specific, Measurable, Achievable, Relevant and Time-Bound. The letters however may have different meaning for different authors, and in some cases letters have been added to this acronym, for this project the objectives will be defined according to the definition previously referred.

The first and most important objective of this project is to develop a middleware that acts like a bridge between a REST API and its clients, offering data in a Publish / Subscribe manner in order to reduce the number of calls at the API endpoint. This is especially relevant for the Service because it reduces the unnecessary traffic at its end, and for the client because it removes the need of keeping a process in an infinite loop polling the service. This functionality should be obtained by the end of May 2017.

The next objective is to develop a library that aids the users of the middleware to implement its functionality in the applications they develop themselves, thus reducing the

time needed to otherwise poll the system themselves. This objective can only be obtained after the middleware has been completed and tested, but needs to be done by the 10th of June 2017.

The last objective for this project is to create a full-stack mobile application that uses the client library in order to receive messages (publications) from the middleware in the form of notifications, using a platform-specific service in order to deliver the messages received at the application server level to the mobile device. This functionality needs to be available by the 20th of June 2017.

2.5. Functional Requirements

The system features are directly related to the problems it is designed to solve. These are features made directly available to the client or features encapsulated in the functioning of the system.

2.5.1. Registering for Weather Updates

The client is able to register to the system providing location and keeps an open connection waiting to receive weather updates. This is a feature of the system component implemented on the client side, in the form of a library that manages the connection to the middleware notification system remotely.

2.5.2. Receiving Weather Updates

The client is able to receive messages remotely. When a new message is received, the application manages it in a callback function.

2.5.3. Grouping Subscribers

The system efficiently manages subscribers in order to perform correct matching between new publication and user interests, and also with the purpose of grouping their requests into as few as possible in order to satisfy as many subscribers with only one request.

2.5.4. Gathering Data

Data is acquired from the same source as clients would usually do. The data is then interpreted and abstracted in a way that the system may interpret it and be used automatically. What is different is that the data is requested once for multiple users thus allowing for lighter traffic.

2.5.5. Monitoring Weather Changes

Watching the responses in order to identify changes in the weather state is also the responsibility of the system. This is done by keeping track of last weather conditions and looking for differences.

Each parameter in the response must be interpreted differently according with type and semantics, and also it is important to only fire new publications for specific changes (e.g. a 10^{-3} difference in temperature may not be relevant).

2.5.6. Publishing Weather Updates

Communication with the subscribers is maintained by the system as well by the client, and new publications are matched by multiple criteria in order to ensure relevance.

2.6. Non-Functional Requirements

2.6.1. Usability

In software engineering, usability is a term used to denote the degree in which a software can be used by the specific set of users it was designed for and the amount of time they require to learn how to do so.

Due to the fact that the solution is implemented as a middleware, it is important to make it easy to use when developing an application that relies on it. Because of this, the system also includes a component that is available as library, which abstracts the communication with the remote system, an abstraction of the subscriber entity and of the message that is received.

2.6.2. Security

Security in software applications is concerned with the controlled access to user-sensitive information that it manages, the possibility to identify users by authentication and encryption of the communication that is performed as part of the application usage.

The threat to personal information is often brought in discussion these days. Sensitive data such as location is needed in order to provide weather data for that specific location. The system however is designed in such a way that there is no data to link the location to a specific person, thus shielding the real person behind the application from being tracked.

2.6.3. Availability

“Availability of a service” actually refers to the amount of time that the service is live and accessible.

Because the middleware is designed to be used as a remote service, it is important to make sure that it is available constantly. Important factors that need to be considered include the usage of third party services, which influence to a high degree the availability of the service. Another important aspect to be factored in is the communication that is based on sockets that need to be maintained.

Chapter 3. Bibliographic Research

During the research conducted for this project, multiple topics were taken into consideration in order to prepare for the implementation part. First, in order to understand the usage of the system, the concept of User Context needed to be explained. Mobile Notifications are then discussed because it is used as a preferred message delivery system, and because they can be of great importance for delivering important information to the user in a short amount of time. An informative description of systems developed for Weather Information, which are to some extent similarities to the proposed solution, is presented in the next section.

The Publish/Subscribe Principles are also described in order to lay down a foundation for the development of the middleware system and to provide a better understanding of the mechanics of this architectural pattern, as well as some proposed improvements. Also in order to prepare for a better design of the system, the next section discusses the generics of Middleware Design. A short description of REST APIs is provided after that, because it is the most popular interface that online services use and it is important to understand it in order to make correct use of it.

The last section in the Bibliographic Research chapter discusses the important consideration of developing an android application, as well as some platform specific features that must be acknowledged before planning the design of the system.

3.1. User Context

In human-to-human interaction, the understanding of situational information, or context holds an important role. This fluidity is however not maintained in the interaction between human and computer. In order to allow the computer to better communicate with the human agent, the later will have to provide context information in order to train the computer with knowledge in order to improve. Because however this is a limiting solution, it is believed that providing the computer with a broader access to the user context information will allow for better communication between the two. The authors of [2] have provided a synthesis of previous definitions of user context, as well as an original one, referring the importance of context knowledge in the interaction between two agents in the case where these are represented by a human and an application.

In order to build an abstraction of the user context that can be used by the system in order to gain and produce information from it, data must be gathered from the surroundings of the user, and this can be achieved in multiple ways. One such way is presented by the authors of [3] which propose an invention that is comprised of three components: the first one is responsible with collecting the core piece of context information, namely the coordinates from the user location, and send them to a second component, responsible with gathering information that could be of use to the user, such as nearby business and POI (points of interest), and delivering that information to the user by the use of the third component, which is an application that places these points of interest on a map and presents them to the user in an interpretative manner.

User context is not always based on location, however. The creators of [4] have designed a system which relies on context in the form of nearby devices that are accessible for the user of the system in order to deliver messages to said user. They also

mention the fact that sometimes the context of the user is important in order to control the behaviour of the system in the way that user environment may require a different behaviour of the system.

3.2. Mobile Notifications

Mobile notifications are a fundamental feature of smartphones nowadays, and serve a simple yet very important purpose to the user, namely the delivery of messages targeting either the mobile Operating System or an user-installed application. Through this mechanism the user can be informed of new messages received in chat-like applications, activity on relevant posts from social media applications, important updates for software or news and events happening in proximity of the user. The authors of [5] have comprised a large-scale analysis the nature of Mobile Notifications and the way in which they impact users, providing the means of efficient use.

Notifications systems are now implemented in a platform specific manner, as a component of the operating system, and are delivered by the use of a push notification server specific to the platform, such as APNS[1] (Apple Push Notification Service) for Apple devices or FCM[2] (Firebase Cloud Messaging) for Android devices. The way in which the notifications are delivered is managed by the specific systems which are accessible by a specific API implemented in the SDK used when building the Application.

It is important to mention that platforms such as PubNub[3] have been built to allow the development of applications that target multiple device families in order to support a unified method for sending notifications to all devices. This is especially useful because it eliminates the need of keeping track of the specific platform that users of the applications are using and having multiple implementations for each, thus saving time and effort.

Another important note is that before integrating FCM as the default way of notification handling, Google was using GCM[4] (Google Cloud Messaging) in order to send notifications to Android devices. FCM has now replaced GCM, which became deprecated as of April 2018, while retaining the previous feature of also posting notifications to iOS devices (by an extensive use of APNs).

In an attempt to perform an analysis on push notification techniques in the context of social media applications, the authors of [6] explore some metrics focusing on measuring relevance of the delivered message. Relevance is the most common feature that users look for when receiving notifications and a focus on the development of better push notification systems.

3.3. Importance of Mobile Notifications in User Context Awareness

Because of the crowded, fast-moving environment that most people find themselves, the surroundings and dangers have also become more complex and harder to be aware of without close and concentrated observation. It is easy to be taken by surprise

[1] Apple Documentation for communicating with APNs (https://developer.apple.com/)
[2] Firebase Cloud Messaging documentation (https://firebase.google.com/docs/cloud-messaging/)
[3] PubNub: https://www.pubnub.com/
[4] GCM: https://developers.google.com/cloud-messaging/

even by large-scale events, even if they can be predicted. Natural disasters have not changed, and still pose a threat to the passing people.

Many instances of applications have been developed with the purpose of raising awareness of imminent changes in the environment, some with the purpose to warn users and instruct them to take cover, such as [7], which integrates a notification component in an already existing system in order to provide information about approaching threats and how to manage in the aftermath; this application was specifically built for the Virginia Tech Community, but applications with the same purpose have been developed by governmental institutions such as the DSU mobile application for emergency situations developed by the Department for Emergency Situations in Romania [8].

There are also applications for informing the user about situations that develop in his context that do not imply direct threats. The author of [9] for instance has developed a system that delivers context relevant notifications for users of a VOD system, in which case the context is simply related to the usage of that particular system.

3.4. Weather-Based Information Systems

The environment of the user among others represents the user context. In order to deliver real-time information about it, an environmental monitoring system is required. The authors of [10] present the steps for developing such a system and also document the many factors that add up in order to obtain a complete view of the environment. Even when looking at weather as a subset of the environment, too large of a set of parameters need to be observed in order to obtain a complete analysis of the state of the weather conditions.

An example of an application that can seem trivial to the user is CUPUS [11], which is a system built in order to gather data about the air quality using sensors worn by people, with the added functionality of delivering notifications to other users of the system that act as consumers of that information.

In order to deliver the desired information in the most efficient way, multiple options have been used in the past, the most popular in the opening of the digital era being through email, or instant messaging; however in the past few years, a new way to reach users has been developed in the form of Push Notifications, which are available on all smartphones, and according to the survey done by [12] it is also the preferred method of the users.

3.5. Publish/Subscribe Systems

3.5.1. Generic Publish/Subscribe Systems

"Publish/Subscribe" is a communication paradigm that has met many adaptations, as stated in the research [13]. In any variant of the principle of publish/subscribe, systems have met at least these three components:

- Event System;
- Publisher;
- Subscriber.

The interaction between these components is intuitive and simple, inspired from real-life models such as magazine subscription. The Subscriber entity establishes a connection to the Event System in order to register for receiving messages (also referred to as "notifications" in some contexts), either by a topic of interest or by letting the Event System know that there are only certain messages that should reach it, which should be filtered by the content of the publication. The first method of filtering messages is generally known as Topic-Based routing and the later one is known as Content-Based routing.

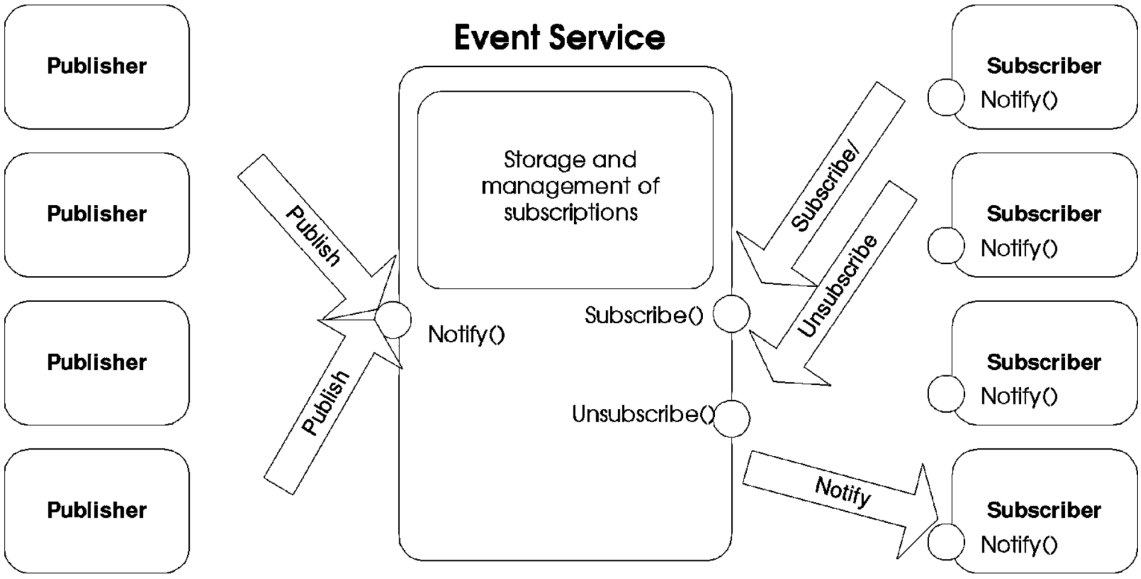


Figure 3.1: A Simple Object-Based Publish/Subscribe System (Extract from [13])

It is the responsibility of the Event System to keep track of active Subscribers, as they register and unregister. Routing messages from publishers to subscriber is also in the responsibility of the Event System, and there are many ways in which this is achieved, the first factor that is taken into consideration being the routing method that is most suitable for the project under development.

Publishers are entities that produce the messages that will need to be forwarded to the subscribers. The data that is received at the Event System end-point will then be interpreted and routed accordingly. Due to the nature of Publishers as separate and highly customisable entities, it is important to take note when designing the Event System that there should exist a highly usable API available for publishers. The beauty of this principle is the fact that it supports the loosely coupled designs that are currently met in the World Wide Web.

3.5.2. Topic-Based and Content-Based Routing

The process of delivering a message from publisher to subscriber is called "Routing", because it is similar to finding the fastest route the message should follow from start to all possible destinations, which are unknown at the beginning. Many routing algorithms and patterns have been developed and out of all of them, the two that stand out are Topic-Based Routing and Content-Based Routing.

In Topic-Based Routing, subscriptions adhere to groups (sets) focused on certain subjects, by registering to a "channel". Through this specialised channel, the messages are to be forwarded by the event system if they belong to the subject set that the channel links to. Depending on the system, subscribers could be able to register to receive messages that are linked to multiple topics.

The general Topic-Based Routing principle has found in itself many implementations, such as [14], wich presents some solutions for general problems that can be found with such a system, such as a dynamic change of topics in the interest of the user when changing the context of the user.

The principle of Content-Based Routing is, as its name implies, based on the actual contents of the published message. Projects such as [15] present the implementation of a publish/subscribe system with content-based routing with the use of a hierarchy of filters, with the use of which messages are interpreted and decisions are made for forwarding to the corresponding subscribers. Most systems like this need a way to filter the messages, but differs from one to another is the way in which the filters are formulated.

The previously mentioned project works by maintaining routing tables in which it stores the filters and links, which is a solution based on managing the route, but many other solutions have been developed, in which for example the subscriptions are reorganised in sets with subsets based on filters. The authors of this paper [16] make a comparison between multiple content-based strategies for subscription management and message routing. The many variants are not limited however and can be used in combination with one another.

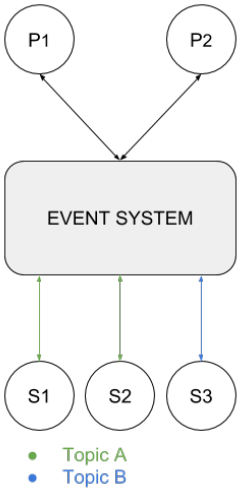


Figure 3.2: Topic-Based Routing

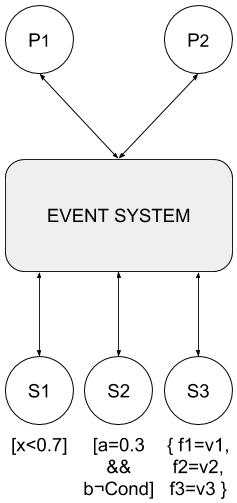


Figure 3.3: Content-Based Routing

Comparing the two routing methods is not of relevance because of the great difference found in the way the message is routed. One method may be better used in a set of problems while the other may be better for another. In [17] the authors present the differences between the two methods of routing messages.

3.5.3. Improvements for Topic-Based Routing

Topic-based routing systems use channels in order to deliver content of interest for the subscribers. In the case in which there are a large large number of publications that need to be delivered to a large number of subscribers, the channel maintenance will need more resources and management overhead. The large number of publications and subscribers and a limited number of channels define a map based on which users are expected to receive only information to which they subscribed for.

From the multiple solutions that have been devised in order to aid in solving what is known as the *channelization problem* one stands out, namely the Dynamic Clustering in Topic-Based Publish/Subscribe [18]. This method proposes a dynamic clustering method, which uses information regarding user subscription to dynamically group topics together into virtual topics. Techniques like this are developed in order to reduce the topic management overhead and are mostly useful in the cases in which a large number of topics exist.

3.6. Middleware Systems

The term “middleware” has been used in the past to describe many products, which is why it has to be explained and understood in context. A simple yet beneficial definition is offered by the authors of [19], who define a middleware as a piece of software that lays between two more software in order to connect them, serving purposes such as communication management, or providing some bridge between the API of the two.

In the same paper, the authors provide a taxonomy of Middleware systems, in terms of integration and application of said system. In terms of the way that the middleware is integrated, the categories presented are: Procedure Oriented, Object Oriented, Message Oriented Middleware, Component Oriented and Agents. Most importantly, it is noticed that the Message Oriented Middleware contains two important sub-categories, namely Publish/Subscribe and Passing & Queuing systems

3.7. REST APIs

A representational state transfer application programming interface (REST API) is, as the name suggests, an application programming interface that uses the hypertext text transfer protocol (HTTP) requests to perform GET, POST, PUT and DELETE methods on data, as described in [20]. REST APIs or RESTful Services as they are also refered rely on the representational state transfer architecture, which is often used in development of web services.

This technique is usually preferred over the SOAP (Simple Object Access Protocol) technology due to the fact that will leverage less bandwidth, thus making it preffered for Internet usage. Because of the rising of Cloud usage, REST APIs have become the preferred way to expose web services.

Weather information is mostly available online, from services that allow access to it by the use of a REST API. While analysing the possible sources for accessing weather data, the author of [21] considered multiple online weather services, which all have this as a common feature. This method of communication is a very popular way of providing access to information or system state, and is used by most online services. Monitoring the

weather is thus made possible by polling such a service, in an infinite loop.

This method however has its flaws, especially rooted in the way it is sometimes inefficiently used. It has been analysed in [22] that up to 98.5% of polls on a REST API return with redundant data or error messages.

Solutions have been built in order to fix this issue, which are best used in situations in which an up-to-date version of certain data needs to be obtained without asking for it in an infinite loop. Such a solution came in the form of REST HOOKS [23], which is presented as a grouping of design patterns that provide an implementation that allows the information to be requested once (hooking) and then updates will be sent as that information changes. The way in which this communication is obtained is by implementing REST API functionality on both the provider and the requester, because the requester hooks by formulating a POST request and the response will come in a similar manner.

3.8. Mobile Application and Android Development Tools

Mobile development has experienced a tremendous growth in popularity since the Apple AppStore has opened in 2008. Mobile applications have been developed before that, but since then market places have been open for other platforms such as Android (Google Play Store) and Microsoft (Microsoft Store) and tools have been made available for developers in order to aid them in the creation of platform specific apps.

In the second chapter of [24], the authors present elements of software engineering through the lens of mobile application development, and identify some differences from the software engineering process for other types of embedded applications out of which some of the most important and worth mentioning are:

- Reading information from sensors – mobile devices are usually equiped with a multitude of sensors that applications can make use of for different purposes;
- Accessing remote services of the internet through mobile network – most applications communicate with some web service in order to display new information;
- Families of hardware and software that need to be taken into consideration when building the application;
- Security, in which we can factor in the sandbox principle adopted by mobile platforms in order to limit the access that application processes can have on core functions of the device;
- User interfaces that need to be similar and familiar to users of a mobile platform.

The conceptual architecture of a generic mobile application is well presented by the authors of [25], comprising of an extensive use of the Layered Architectural style and also showing the communication that takes place between the mobile client application and the application server.

In this model, the HTTP layer is responsible with the communication with the server, and the API layer is responsible with interpreting data that is received from the server. The Generic Data layer covers the responsibilities of the usual Business Layer, implementing functions such as caching, validation and logging. The Platform Dependent Data layer takes the data from the API and stores it in a platform specific way, preparing

it for the user interface. In the Android environment, classes like Adapter, ListView and so on are a few examples of how data is stored in order to be prepared for presentation. The UI layer is of course responsible for holding the views, buttons, and layouts and so on for presentation.

Chapter 4. Analysis and Theoretical Foundation

This chapter focuses on the analysis and theoretical foundation behind the workings of the developed software, starting from an analysis of the Publish/Subscribe Architectural Model and Mechanisms, where some designs patterns most commonly used along with this model are presented and discussed along with some other design patterns that will collaborate well with the designed middleware as well as some model-specific routing algorithms and improvements that are well suited for the design of the proposed project.

An analysis of the communication protocols that are used to provide communication with the weather service used as well as serving information to the client applications that are using the middleware. A conceptual architecture of the middleware is presented afterwards in order to illustrate the basic functions of the middleware as well as the use of the previously discussed design patterns.

The features expected in the client library are discussed next and afterwards a model of a client application is then introduced in order to describe the integration of the client library as well as an example of how the middleware can be of use.

The use-cases of the designed system are then presented in the last section along with the expected flow of the system upon interaction with the multiple client applications that could be using it.

4.1. Publish/Subscribe Architectural Models and Mechanisms

The Publish/Subscribe Architecture is the core blueprint of system that this project focuses on building. As briefly mentioned in the previous chapter, Publish/Subscribe Systems are part of the Message Oriented Middleware category. This of course implies that the Publish/Subscribe Architectural Model is used for the purpose of message delivery but is different in nature than the Passing & Queuing systems that are part of the same large category of middleware systems.

The principle on which Publish/Subscribe systems operate is simple. There are three main classes of entities, the Publishers, the Subscribers and the Event System. Publishers are responsible with the generation of Messages of different types, which are of interest to some consumers. These consumers are called Subscribers, and they exist in a plane logically separated than that of the Publishers, namely their existence does not represent knowledge of the Publishers, and thus they are completely separated. Because of the fact that this separation exists, a need for a third component of the system arrises, which is the Event System. The Event System is responsible with forwarding the publications to the Subscribers thus providing a bridge between the two, and at the same time allowing the Publishers and Subscribers to exist without the need of keeping track of one another.

It is important to explain why so much emphasis is placed on the separation that exists between publishers and subscribers, because it is an important feature of systems that rely on the Publish/Subscribe Model. This feature provides transparency of the subscriber, meaning that the message sent by the Publisher does not contain an address of the subscriber that needs to be reached. This is important especially when considering the privacy of the subscriber, because a Publish/Subscribe system is supposed to be active

most of the time, allowing for the subscribers to be reached whenever a new publication is emitted.

If considered in contrast to the regular messaging systems, we can understand the difference in the fact that no destination address is set on the message and also there is no database of subscriber information on the publishing systems. This allows shielding the subscriber from access to private information that may be needed in order to only select the messages that are considered to be of use. The only component of the system that holds the subscriber information is the Event System, which is also the one responsible with routing the messages based on interest.

4.1.1. Design Patterns for Publish/Subscribe

Design patterns exist in order to make software more comprehensible as well as to reuse programming techniques in similar situations. There exists some design patterns that are more frequently encountered in Publish/Subscribe systems, the most notably being the Observer design pattern and the Event Notifier design pattern.

4.1.1.1. Observer

The Observer design pattern aims to solve the problem of high coupling in systems in which it is desired for multiple objects to monitor the state of a subject. This is accomplished by creating an interface which subjects implement, providing the ability to attach and detach observers to itself, as well as asking the observers to provide the means by which they can be notified. This is accomplished by having the observers provide an interface with the method “notify”.

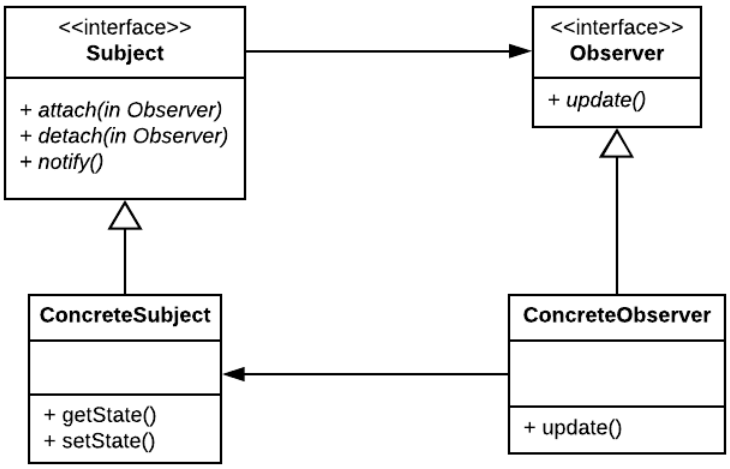


Figure 4.1 The observer pattern (Extracted from [18])

In itself, the Observer pattern provides the most fundamental mechanism of a monitoring system, which is notification upon change of state. In terms of the publishing capabilities however, it lacks the routing capabilities, which is essential for a Publish/Subscribe system.

4.1.1.2. Event Notifier Pattern

The Event Notifier Pattern is also referred to as publish-subscribe because it presents many similarities in terms of purpose to the Publish/Subscribe model. The intent of this pattern is to allow components of the system to respond to events that occur in other parts of the system, without possessing knowledge of one another, as well as to allow the dynamic introduction of new types of events and dynamic participation of other components of the system.

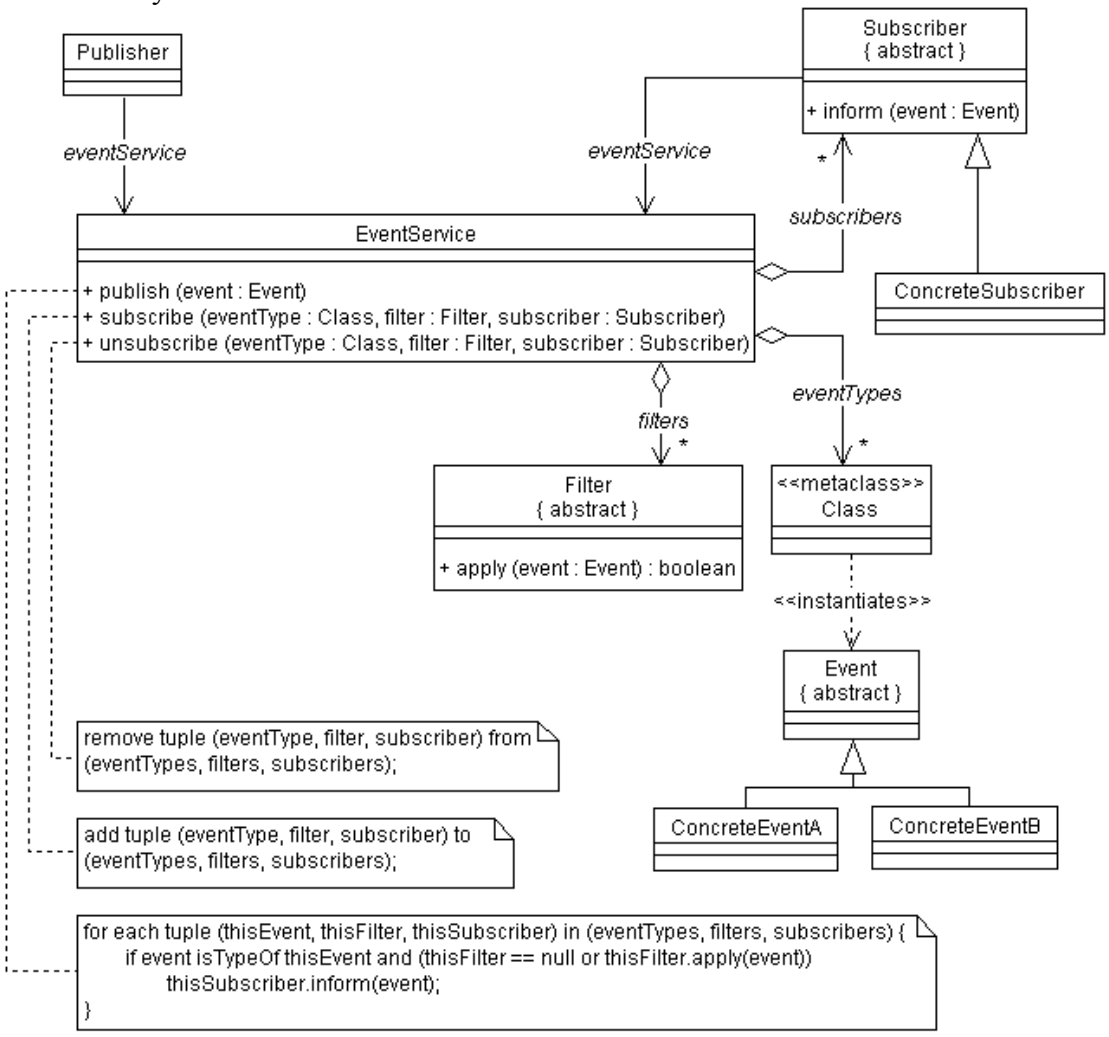


Figure 4.2 Event Notifier Structure (Extracted from [26])

A system with a fixed line of notification in the form of an event line (such as a flag) to some central monitor can be rigid and hard to change when types of events need to be added or when new components are introduced to the system. Instead, the desired system is on that is, from the perspective of the event-responsive parts, unaware of the component set. As it can be seen in Figure 4.2, the Subscriber and Publisher classes are decoupled by introduction of the Event Service class.

This actually solves one of the problems of the Observer pattern, namely the need of maintaining an observer list. This takes away the concern of running through a list (which is of dynamic size) from the Subject, which is observed and instead places this

responsibility on the Event Service class.

This leads to the solution to another problem, the one of targeting subscribers in a transparent manner, without actually maintaining the inter-object knowledge of the observers. Because of this, it is easily to have a dynamically evolving system, in which subscriber registration is possible at runtime, without the concern of the monitored components (namely the Publishers in case of the Event Notifier pattern or the Subject in case of the Observer pattern).

The main components in this pattern are as follows [18]:

- *Event*: describes the structure and functionality that all events should have (in this example, **ConcreteEventA** and **ConcreteEventB**). It holds the information about the data that produced the occurrence of the event;
- *Publisher*: the producer (or emitter) of the Events;
- *Subscriber* (interface): defines the functionality that a **ConcreteSubscriber** should have in order to allow the system to alert instances of a new **Event** that has occurred in the system, namely the public *inform(Event)* method;
- *Event Service*: is the holds the Event routing logic and serves the purpose of decoupling the **Publisher** and **Subscriber** classes;
- *Filter*: is defined by **Subscriber**s in order to inform the **EventService** of the preferences that should be applied in order to select the **Events** that should be forwarded to them.

It is implied in this pattern that the producer of events is the **Publisher** class, even though this is not symbolically represented in the diagram. Because of the fact that **Event**s are structured hierarchicaly, it is made simple for the **Subscriber** to register for a set of **Event** types, assuring that it will be notified for all other subtypes. This means that the set of available Event types needs to be known by both **Publisher** and **Subscriber**, allowing to target subscriptions as narrowly or broadly as desired.

Most systems that adopt this pattern will implement the messaging between the components by using push and pull semantics, meaning that the **Publisher** *push* **Events** to the **EventService**, which will then *push* the **Event** to the **Subscriber**. If the system allows for passive **Publisher**s, the **EventService** may also *pull* mechanics to retrieve the **Event** from the **Publisher**.

4.1.2. Topic-Based Routing and Clustering

Topic-Based Publish/Subscribe is a communication paradigm that relies on message classification based on a subject or **topic** that is recognised both by producers and consumers. These topics allow the selective routing of the messages to the correct destinations.

Topics will usually have channels of communication assigned to them, which are named accordingly, to which subscribers will register to be informed and publishers use to deliver messages. Many industry systems developed with the use of this paradigm allow subscription to multiple such channels, which requires brokers managing the topics and organising the channel in such a manner to ensure efficiency of the system. Helping the orchestration of topics is also the usual hierarchical relationship that is maintained between them.

The qualities of service (QoS) that Topic-Based Publish/Subscribe system provide stand in the degrees of reliability related mostly to the message delivery and persistence, latencies and so on, mostly following the general QoS of Publish/Subscribe Systems. What differentiates in between Topic-Based Systems however is the mechanism that performs the topic management and message routing, since that is the main source of delays in the system.

Management of the topic channels is also known as the *Channelization Problem,* and aims at obtaining the minimum cost to maintain channels. One of the most important conclusions that have risen from studying this problem is that as the number of topics or publications grows to a high extent, it becomes impossible to maintain separate channels for each individual topic.

In order to aid the channel management overhead, one of the proposed solutions is Dynamic Clustering [27], which aims to group topics together into virtual topics, reducing cost by unifying their supporting structures. The user subscriptions are carefully taken into consideration as well as the system state, thus resulting in minimal overhead.

4.2. Communication Protocols

This section explains the communication between services and components of the system, alternatives and choices that are available for the proposed middleware. The parties that need to have communication mechanisms established in between are the Weather service, MMNS and the clients of MMNS.

4.2.1. Communication with Weather Services using HTTP Requests

Most web services are available by use of HTTP that are available either available in the form of a REST API, or by using Simple Object Access Protocol (SOAP) Services described by a WSDL Specification. The latter has been developed in order to allow communication over the Internet over the protocol that is implemented in most systems and Internet browsers, namely HTTP, although it is also available over SMTP or JMS or message queues. It uses XML in order to structure messages that are sent between different processes running on separate systems. The main disadvantages of the SOAP protocol however are the fact that the messages are verbose, and because of the XML structure, hard to parse on some occasions. Also because of the lack of a standardised manner of communication, protocols that make more direct use of the HTTP protocol have become more used, such as REST.

Most weather services that are available online provide access to information by the use of REST APIs, which leads to the conclusion that this is the preferred solution for communication. However, there is no unique way in which the information is structured among these services, and depending on what information there is available. Each service has the API implemented in an original form. It is also important to note that not all services are available freely.

For the purpose of this thesis, the services that were taken into consideration are the following: OpenWeatherMap[5], Weather Underground[6], AccuWeather[7] and Dark

[5] Weather API – OpenWeatherMap (Online) https://www.openweathermap.org/api
[6] API | Weather Underground (Online) https://www.wunderground.com/weather/api/
[7] AccuWeather API (Online) http://apidev.accuweather.com/developers/

Sky[8]. The weather service of choice is OpenWeatherMap because of the compact API and ease of use. The most important features that are freely available from these services are:

- No more than 60 calls per minute
- Access to the current weather API
- 95.0% availability
- Weather API data every two hours or less
- Weather alerts[9]

These are important features to take into consideration when building a system with the responsibility to monitor the current weather in order to report changes to the subscribers, because the mechanics of this communication protocol. In order to analyse a continuous stream of data received from a REST API, requests must be continuously be sent to that API in order to receive the desired data in the form of HTTP responses. Because of this, it is easy to deduce that a process will need to exist with the responsibility to continuously send requests to the API, while it's also important to not step over the limit of 60 calls per minute, which can lead to the blocked access to the API with the used API key.

4.2.2. Communication with Client Applications using Sockets

Since the main goal is to develop a middleware that delivers notifications to clients, it is important to highlight the communication that takes place between the middleware and client applications. Being a Message Oriented Middleware, the two possibilities are of course publish/subscribe interaction and passing & queuing. Due to the nature of the intended result, the first one is a better suited for the situation at hand.

One of the main advantages of Message Oriented Middleware is the fact that asynchronous communication is allowed between parties, unlike for instance RPC-based systems in which the communication is synchronous and the calling entity must wait for the response of the executing entity. In MOM, the message is sent by the process, which can then immediately proceed to continue tasks at hand. Another advantage is the lower coupling that can be obtained in such systems.

The Passing & Queuing methods place emphasis on the delivery of messages in order, keeping them available for consumption in the form of a queue, with one specific direction: from sender to destination. The messaging queue will be consumed by the client systems if and when they desire, which means that sending and receiving ends of the communication need not be active at the same time.

Publish/Subscribe systems, on the other hand, require a more persistent connection between the two ends of the communication, namely the Publishers and the Subscribers, as well as the added necessity for message routing according to the desired method (topic based or content based routing). In this scenario, the middleware often is referred to as a message broker component in the system. The preferred communication for this system will thus be TCP Sockets, in the form of a Client-Server Communication

[8] Dark Sky API (Online) https://darksky.net/dev
[9] Pricing – OpenWeatherMap (Online) https://openweathermap.org/price

protocol.

Sockets allow communication over the Internet at TCP level and are widely used to implement Client-Server interaction. Many kinds of data can be streamed via sockets, since it is broken down to byte data. Because of this, it is easy to design a communication protocol with custom validation and message types. However, because of this raw data transfer in the form of bytes, another problem arises in the form of implementation of the communication mechanism on the client side. The important questions that arise are:

- How will the client know the form of a registration method?
- How will the client know the correct interpretation of a message?

The solution to these problems is to either prepare a very well documented structure of the message formats that are expected and produced by MMNS, or the creation of a library that provides the client with an abstracted, ready to use communication protocol. The latter is preferred because it hides knowledge of the communication protocol from the users of MMNS and also leaves less room for error prone messages.

The connection is maintained between the subscriber threads on each respecting side, meaning that the subscriber thread on the MMNS side acts as a proxy. The message should have a start and end marker that will be used in the validation part of interpretation, and also if an imposed structure of the message will not be respected, the message can be considered to be corrupt and dropped.

4.3. Client Library

Because of the way in which TCP sockets are designed, in order for two parties to communicate, the address and port need to be known in order to obtain a connection. The messages can have any structure, because data is sent as a stream of bytes, so a mechanism for interpreting the data becomes necessary in order to communicate with any service in this form.

In order to provide a solution to these problems and aid the developers of systems that wish to use MMNS to receive weather notifications, a special library is designed. The main purpose of this library is to establish a connection with MMNS, and aims to provide the developers with the main tools of the system, in a descriptive abstraction. Because topic set advertisement is also important in this system, the library also includes an abstraction for the available topics.

Stripped down to its core functions, the library will be able to instantiate subscriber threads which have corespondents on the MMNS and receive messages concerning the selected topics used for subscription. The message is also interpreted by the use of the library, and the contents of the message can be accessed by accessor methods.

4.4. Conceptual Architecture

Figure 4.3 presents the conceptual architecture of the system, featuring the components that are involved in the communication at a large scale: the Weather Service API, MMNS and the client application, highlighting the manner in which the components communicate with one another, namely HTTP between MMNS and the weather service

and TCP sockets between MMNS and the client application.

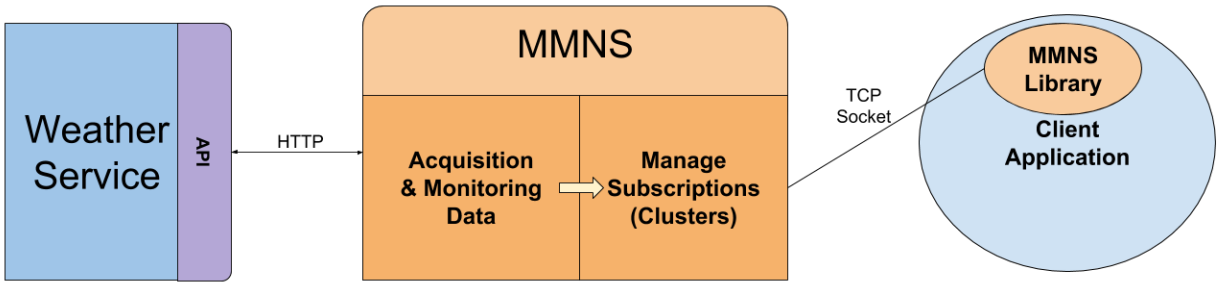


Figure 4.3: Conceptual architecture of the system

The client application will use the specially built client library in order to communicate with MMNS, but the latter is contained in the application and accessible to it, operating under the same parent process.

MMNS presents also the two important parts of the system, namely the acquisition and monitoring of the data that is obtained from the weather service and communicate it to the subscriptions that it manages. At the highest level, the data is to be perceived as flowing from left to right, being monitored and filtered by MMNS, and delivered to the client application in the form of published messages, thus reaching the desired goal of this design.

4.5. Mobile Application

In order to illustrate the use of MMNS, a proof of concept application is also developed as part o the thesis. This application should work well with a publish/subscribe system, so naturally a mobile application is the best option for this. Because the notification systems in smartphones are directly managed by the operating system and their respective services, Android is the mobile environment of choice, especially because the freely available push notification delivery system that is Firebase Messaging.

This section deals with the analysis of the proof of concept: communication with MMNS as well as the analysis of the platform-specific tools and services that will be needed in order to function, namely the push notification service, Firebase, and the geolocation service available in Android.

4.5.1. Communication with MMNS

Following the model of a mobile application, the component that will handle communication with the middleware will be located on the server side of the application. The exchange of messages will be performed by the use of the library developed for the clients of MMNS, which has to be implemented in the Server-side Application.

In order to allow the correct function of the communication mechanism provided in the library is a persistent connection to the Internet and the permission to receive messages from MMNS using the TCP ports, implemented as sockets. Firewalls may be a problem at this point, blocking messages before they reach the application. Internet connectivity is however implied due to the need to communicate with the push notification service of choice.

Since each subscriber has a correspondent in the MMNS, the Server will likely have to keep a set of active subscriber entity that communicate with MMNS. When a subscriber receives a message, the corresponding thread will be brought in the foreground and the message forwarding to the end device will need to be handled.

Because the provided library already handles the communication, the only responsibility that remains to be covered in the client application is the handling of the message.

4.5.2. Push Notification Delivery

Firebase Cloud Messaging (FCM) replaced Google Cloud Messaging (GCM) as the preferred way to deliver push notifications to Android mobile applications. It is a cross-platform cloud solution that can be used to deliver messages to both Android and iOS applications, currently at no cost.

Firebase offers the possibility of creating topics for notifications in order to act as message broker in a Publish/Subscribe Topic Based messaging environment, targeting groups of devices or individual devices with messages. All this can be obtained from the Firebase Messaging console or by automated process using the Firebase Admin SDK. In order to send a message from the server to a single targeted device, the server must have knowledge of the device's Firebase Token, which is automatically generated by Firebase upon device registration, and is known only to FCM and the device. Because of this, it is essential to have the possibility of letting the device communicate with the server in order to register its Firebase token.

Once the Firebase SDK is set up on the mobile application, registration token successfully obtained and is made available to the server side of the application, it can be used in order to send push messages to the device by making use of the Firebase Admin SDK. It is important to note that certain events may trigger the refresh of the Firebase registration token for a particular device. This event will be signalled to the device, but it will have to forward the change to the server.

In order to deliver messages by the use of the Admin SDK, the builder pattern is used in order to provide a simplified way of composing the message. The message has some useful fields such as priority, which can be either **normal** or **high**, a time-to-live duration which allows for the cancelling of messages that become irrelevant after some time, and a **data** field, which is a key-value pair map that contains the actual data contained in the message. The message can also have a notification object attached to it.

In the notification part of the message for the Android platform, parameters that can be set include the title of the notification, the body, the icon which is shown next to the message and the sound. Other important parameters include a **tag**, which allows for replacing already existing, older notifications which have the same tag in the notification drawer. This is especially useful in a system with the intention of keeping the user up to date with the modifications that occur in the weather, because it allows for the automatic replacement of older notifications that concern the same weather parameter[10].

Since the main objective of MMNS is to deliver the updates as soon as they are available, it is important to take the delay added by Firebase into consideration. Messages

[10] Add the Firebase Admin SDK to Your Server | Firebase (Online, retrieved June 21st 2018) https://firebase.google.com/docs/admin/setup

which are sent with **normal** priority will be delivered immediately if the app is in the foreground. If sent with **high** priority, however, the notification will show up even on a sleeping device. FCM intends for high priority messages to result in some user interaction upon arrival, meaning that in the case that a pattern in which the user does not interact with the notification is observed, messages sent from the same source may be de-prioritzed.[11]

4.5.3. Geolocation

Perhaps the most important subscriber data that needs to be taken into consideration in order to route the notifications is the location of the subscriber. The process of obtaining the geographical location of a point in the geographic coordinates system is referred to as **geolocation**. Mobile devices are generally equipped in order to be able to be located using one of two popular methods.

The first method is using a GPS chip that is found in the mobile device, which uses satellite data in order to compute the exact location. The other methods communicates with cell towers or Wi-Fi points and use their information in order to triangulate the location, which results in a less accurate, approximated location data.

Both methods are available as a service on the Android platform, with a very well explained guide that covers the strategies and important considerations when performing location tracking.[12] From the information presented in the guide, it is crucial to note that the application user needs to give the application permission to track location and the fact that depending of the method used for coordinate retrieval (GPS or Cellular Network), coordinate data may be updated at a different pace, and with a different degree of accuracy.

Another potentially harmful event that may occur is known as GPS drift, which means that due to impeding factors to geolocation, the coordinates will be read as if the user would be moving in some direction when he is really standing still, or moving in a different direction. This phenomenon has no current solution, but some workarounds can be devised, like not taking into consideration all readings, or not taking into considerations readings that do not consist into sufficient movement in order to change the location at a level so high that the weather information would actually be different. Happily, GPS drift will usually occur at a small scale, namely meters to tenth of meters, which is in most cases not enough to change the location for which the weather information is monitored.

Factors that can lead to GPS drifting include cloudy weather, which limits the signal strength between the satellite and device. Also the building infrastructure can play a role in signal limitation, but since most buildings nowadays are equipped with Wi-Fi access points and in range of cell towers, we can assume that this alternative method can be used to triangulate the coordinates. The downside of this scenario however comes from the approximated result of the triangulation method, which is not always entirely accurate.

[11] Send Messages | Firebase (Online, retrieved June 25th 2018)
https://firebase.google.com/docs/cloud-messaging/admin/send-messages
[12] Location Strategies | Android Developers (Online, retrieved June 25th 2018)
https://developer.android.com/guide/topics/location/strategies

4.6. Use-Cases

4.6.1. UC1 – Registration

This use-case captures the flow of events that occur when a new subscription is registered to the MMNS service, the context from which the registration is made and the expected behavior of the system.

Primary Actor: the **client application**, which uses the MMNS client library in order to deploy an active subscription for the system.

4.6.1.1. Stakeholders and Interests

1. Application Developer:

Interested in assuring that the end user of the system he is creating will be correctly mapped in the MMNS service, and will be receiving correct notifications automatically.

2. Client Application End-User:

Interested in receiving correct notifications when the client application is running.

4.6.1.2. Basic Flow

This use-case starts when a Subscriber instance is created on the client-side. It is the responsibility of the Application Developer to ensure that the Subscriber is correctly instantiated, even if this process is manual or automated.

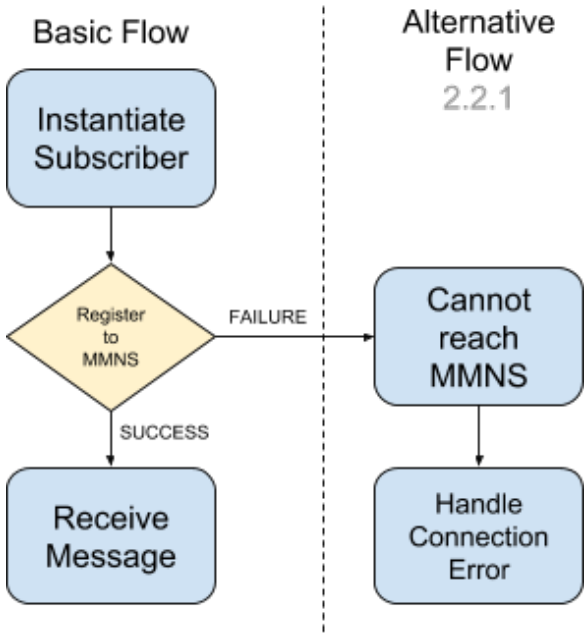


Figure 4.4: Flow Diagram for UC1

1. **Application Developer** instantiates the with the following essential information:

   a. Location coordinates
   b. A set of topics that are characteristic for the application
2. **MMNS client library** will automatically establish connection with the MMNS system, mapping the active subscription and registering it for notifications.
3. The **MMNS System** will respond with an initial set of data representing the last weather update for the location and topics of the Subscriber in the form of a **message**.

This use-case ends after the MMNS System response is interpreted and handled by the Client Application.

4.6.1.3. Alternative Flows

1. AF1: Connection to MMNS System cannot be established

This alternative flow starts at step 2 of the basic flow, when the Client Library will try to establish a connection to the MMNS System but the later is unreachable.

Subscriber thread will end with an exception; Application Developer is responsible with handling connectivity error.

4.6.1.4. Preconditions

1. MMNS can be reached via network
2. End-User starts application
3. End-User location can be obtained

4.6.1.5. Postconditions

1. Subscription is mapped in the MMNS System
2. End-User has received last update
3. Client Application handled connection error (ex. retry, cancel) and resumed execution.

4.6.1.6. Special Requirements

1. Location tracking

Since targeting the notifications is based on end-user location, it is important to ensure that the user knows how his location will be used by the system, and that he must offer his permission for collecting the location.

How the location is obtained is dependent on the platform and is the sole responsibility of the Application Developer.

2. JRE support and dependencies

In order to interact with the MMNS System, it is mandatory to use the Client Library that is available as a JAR, to be used in a Java environment and runs with the support of the JVM.

4.6.2. UC2 – Messaging

This use-case captures the flow of events that occur when a published message is

forwarded by the MMNS service to the active subscribers, the context from which the message is sent and received, and the expected behaviour of the system.

Primary Actor: the client application, which is registered to the MMNS and is waiting to receive messages from the service.

4.6.2.1. Stakeholders and Interests

1. Client Application Developer

Interested in assuring that the implementation of the MMNS client library from the developed application forwards the received message to the end-user.

2. Client Application End-User

Interested in receiving the message on his device.

4.6.2.2. Basic Flow

This use-case starts when a message is forwarded by the MMNS to the client application.

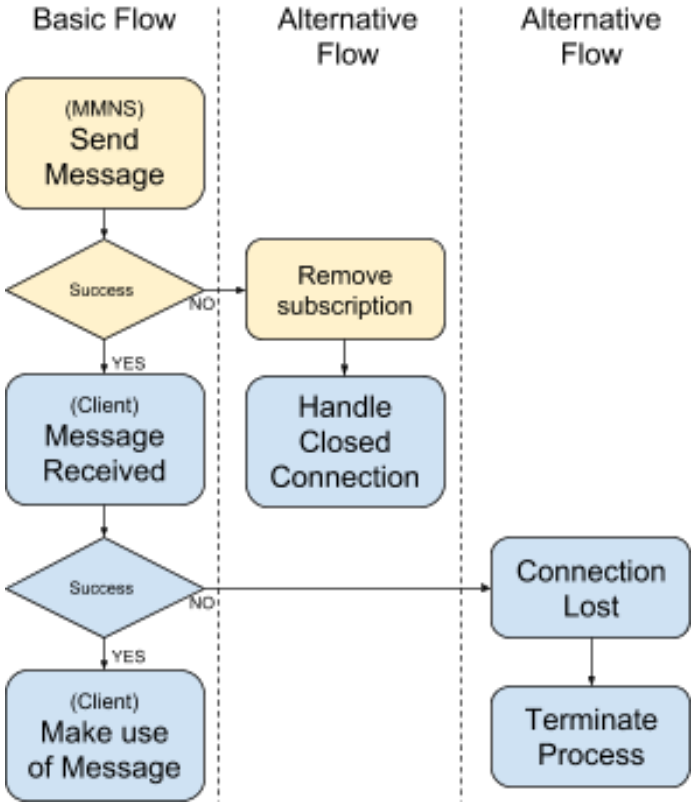


Figure 4.5: Flow Diagram for UC2

1. MMNS sends message to the active subscription;
2. By communication protocol, MMNS ensures that message has been successfully sent;
3. The client application receives the message by a callback method implemented by the application developer;

4. Client application validates the received message;
5. Client application unpacks the message in order to present it to the end-user.

This use-case ends after the Message is handled by the client application.

4.6.2.3. Alternative Flows

1. AF1: Communication failure on MMNS side

This alternative flow starts at step 2, when MMNS cannot send the message.

Subscription will be removed from MMNS and communication channel closed, so the client application has to handle subscription removal in the desired manner.

2. AF2: Communication failure on Client side

This alternative flow starts at step 4, when Client side cannot establish connection with the MMNS. Connection is deemed lost and communication channel is closed. Subscriber thread is terminated.

4.6.2.4. Preconditions

1. Client is registered to MMNS
2. Client is persistently connected to MMNS

4.6.2.5. Postconditions

1. End-User has received message
    a. AF1) Client Application end of the communication channel is closed for respective subscription
    b. (AF2) Client Application subscription thread is joint and respective communication channel closed

4.6.3. UC3 – Unregistering

This use-case captures the flow of events that occur when a subscriber breaks connection with the system, or wishes to unregister from being notified.

Primary Actor: the client application, which is registered to the MMNS and wants to give up on receiving notifications.

4.6.3.1. Stakeholders and Interests

1. MMNS Administrator

Interested in making sure that no unregistered clients are still in the system, consuming resources.

2. Application Developer

Interested in assuring that the client is unregistered from MMNS and will no longer receive messages.

3. Client Application End-User

Interested in no longer receiving the messages on his device.

4.6.3.2. Basic Flow

This use-case starts when a client subscriber process breaks connection to the MMNS.

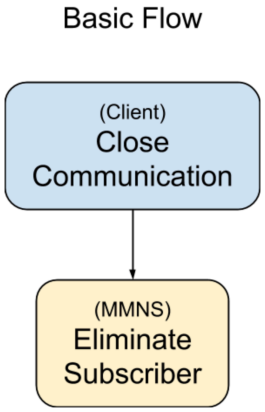


Figure 4.6: Flow Diagram for UC3

1. Client subscriber process closes communication;
2. MMNS eliminates the corresponding subscriber entity from the system.

This usecase ends when the elimitation of the Subscriber from the MMNS system has been completed successfully.

4.6.3.3. Preconditions

1. Client is registered to MMNS
2. Client is persistently connected to MMNS

4.6.3.4. Postconditions

1. Subscriber has finalised execution on client side
2. Subscriber process and data no longer exists on the MMNS
3. Connection between subscriber processes is closed

Chapter 5. Detailed Design and Implementation

In this chapter the detailed design choices are discussed, and the implementation of the different components is explained. Taking from the analysis conduced in the previous chapter, theoretical concepts are translated into working components, aiming for a good design and clear implementation.

A more detailed system architecture is presented first, which also delves into the responsibility of the more complex components of the system. It also will show the physical decomposition of the client application in the case of mobile applications. Upon that system architecture, each component that is developed under this project will be detailed even more in terms of design patterns and functionality.

5.1. System Architecture

Figure 5.1 presents a detailed system architecture that lays out all the components that will be involved in the system at the end, highlighting the extended communication model proposed for the delivery of notification to mobile devices.

The weather service used is OpenWeatherMap, which provides an API in order to allow the retrieval of the current weather, by the use of the HTTP protocol. The next component in the system is MMNS, which has the responsibility to prepare the calls to OpenWeatherMap API accordingly to the subscribers that are managed and their interests, gather that data and manage it in order to send it to the subscribers.

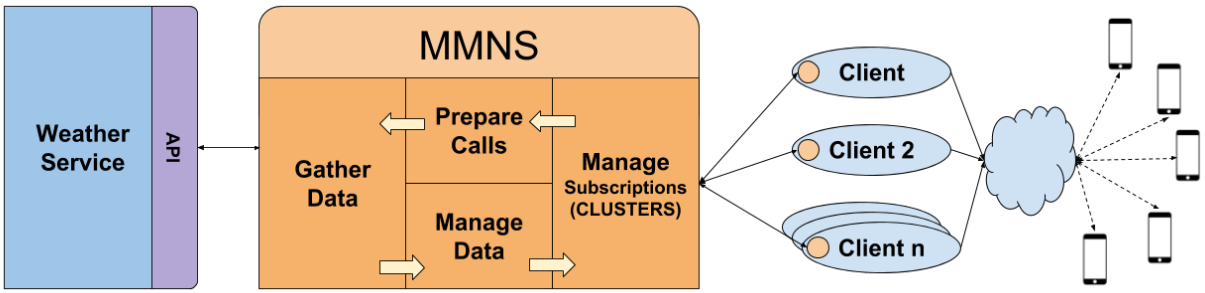


Figure 5.1: System Architecture

The next level covers the applications that are making use of MMNS to receive weather updates. The communication actually takes place between MMNS and the library that clients are to integrate in order to exchange messages with MMNS. Because he application that is developed as proof of concept for this project is a mobile application, this architecture also presents the physical separation in the application components and communication using cloud services: the application server will send push notification using the cloud service available for the used platform, in this case being Firebase Cloud Messaging for Android.

5.2. Middleware Implementation

The core component of the system and the one holding the capability to bring the

functionalities to the consumers is the MMNS middleware. As previously stated, this middleware stands as a bridge between the weather service and its clients in order to translate the communication paradigm from REST API form to the Publish/Subscribe messaging.

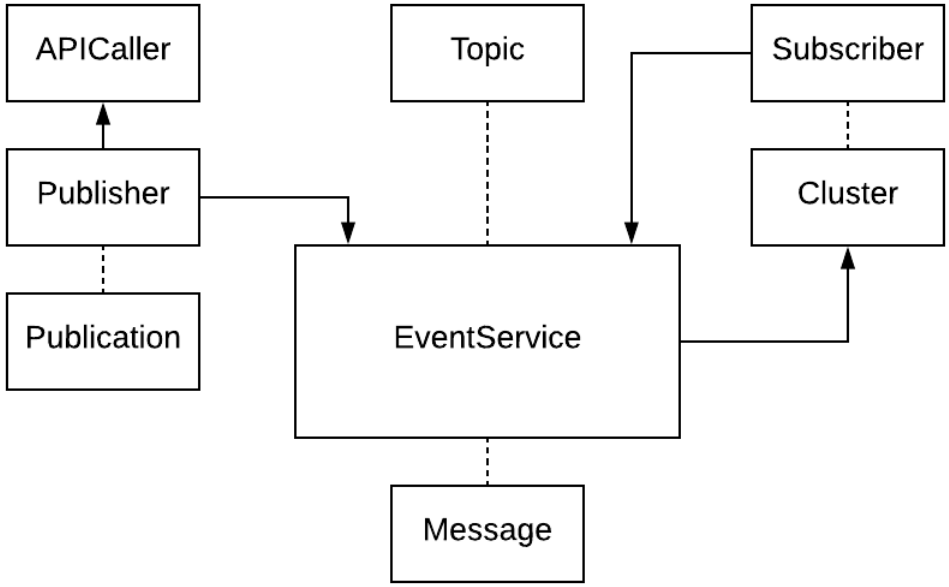


Figure 5.2: Block Diagram of MMNS

5.2.1. Endpoint Abstraction

MMNS needs to perform communication with two separate components, namely the OpenWeatherMap API with the role of data provider, and the multiple client applications, having the role of subscriber. These two endpoints need to be represented in the software in order to make them easier to use. This section presents the way in which the API and Subscribers are abstracted.

5.2.1.1. API Abstraction

OpenWeatherMap uses a REST API to allow access to information about current weather. In order to access that API and poll the data, we need to gain access to an API key. That key is also used by the service in order to identify who is making the calls and restrict the number of calls depending on the type of account linked to the key (free, startup, developer, professional or enterprise).

REST APIs usually respond in JSON form, but the response is shaped differently across services. OpenWeatherMap is no exception, so the keys by which the values should be known. Because there are a multitude of String values that need to be used in order to access data, a frequently used practice will be implemented in this design, namely storing the keys in constants with suggestive names in order to avoid the use of literals in the code.

In the naming of the constants the strategy used is:

PREFIX_MAIN-GROUP_PARAMETER ,

in all-caps. For example, the key for the weather description in the JSON response will be OWM_WEATHER_DESCRIPTION, in which “OWM” is the prefix, “WEATHER” is the main group and “DESCRIPTION” is the parameter the value of which we wish to retrieve.

The constants for the keys, API URL and API key will all be stored in the API class, which also implements the static methods for building a weather condition abstraction from the JSON response as well as building the URL to be used in order to access the API, containing the key and all other path variables needed.

Having the API class implement all functionality that is required in order to extract the response from the OpenWeatherMap API allows for the encapsulation of the API specific data, which also allows for some privacy on the access of fields such as the API key.

5.2.1.2. Publisher

Publishers represent entities that hold data of interest to the subscribers of the event system. Because of this, the Publisher class needs to provide a mechanism of data acquisition, as well as conform to the Event Notifier pattern. Also, the way in which the response of the API is structured, covering the data concerning one location at a time leads to the conclusion that one publisher must exist to cover each location.

Two possibilities are taken into consideration for the gathering of data by the publisher. The first way is to provide the publisher with the functionality of calling the API of OpenWeatherMap, as depicted in Figure 5.3. Because of the existence of multiple publishers as part of the system, this would lead to multiple calls being made from different virtual sources. The problem with this approach comes from the limited number of calls that can be made while using a free account.

The other way for the publisher to gather data is to delegate the responsibility of calling the API to a different component. The way in this is achieved is by using a simplified Observer design pattern and hooking the publishers to an API Caller, which

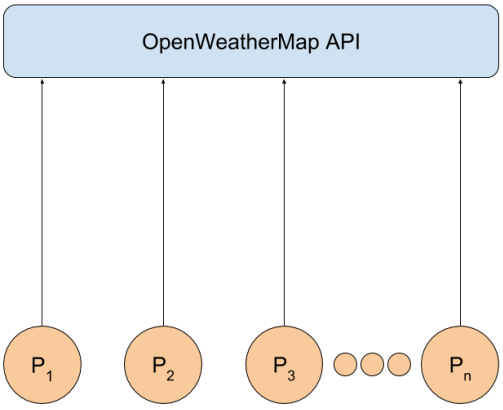


Figure 5.3: Publishers perform the API call directly

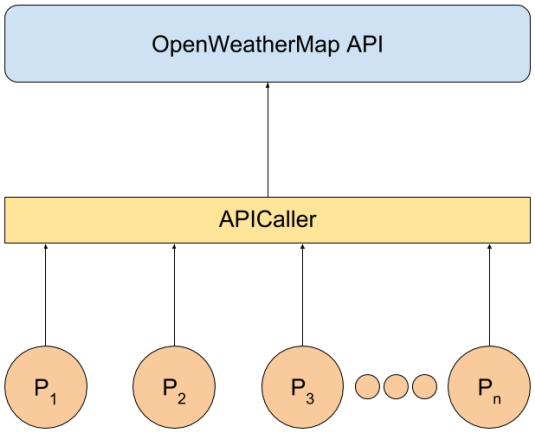


Figure 5.4: Publishers delegate the responsibility of calling the API to the APICaller

will schedule the calls for each publisher and ensure that not more than one call is made per second, as depicted in Figure 5.4.

One thing that is desired from MMNS is to have a dynamic set of Publishers. Since OpenWeatherMap provides service for any location, including over 200,000 cities, it would be unwise to deploy a publisher for all of this locations from the start, even though there is no active subscriber for them. Instead, a Publisher for a location should only be instantiated when at least one subscriber exists for that specific location. In order to obtain this, the component responsible with the instantiation of Publishers will be the EventService.

Since the EventService is designed according to the Singleton design pattern, there is no need for the Publisher to keep a reference to it. The EventService will be referenced by the Singleton method *getInstance()* and messages will be sent to it by the use of public *publish()* method.

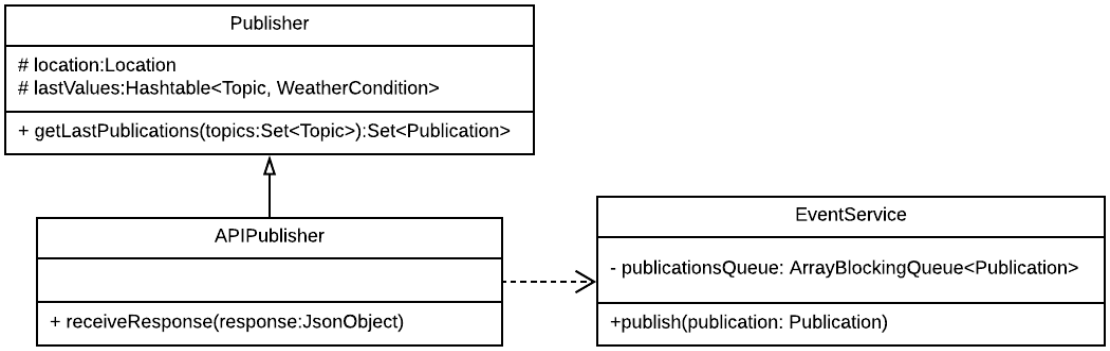


Figure 5.5: Partial Class Diagram for Publisher, APIPublisher and EventService

Because the focus of the project is the current weather data, there is no need to keep the old data updates in memory or store it in a database. However for the situation of a subscriber registering in the system at some time it is useful to keep the last update in memory in order to be able to feed it to new subscriptions so that they do not have to wait for the first update.

Also the Publishers should have the mechanisms needed to filter the new responses in order to only publish changes in weather, not all responses that come from the OpenWeatherMap API. The strategy for this is implemented in the APIPublisher class, which takes the last value and compares it to the new one in order to see if

- a weather condition has been retrieved for the first time, or
- the last value and new value are different,

in which case new value will replace the last value and a new publication will be added to the EventService message queue.

5.2.1.3. APICaller

The free OpenWeatherMap API account has an imposed limit of no more than 60 calls per minute, and by use of these calls data must be gathered in order to serve all subscribers as soon as new data has been made available. Since each publisher covers the

needs of subscribers from a specific location, the priority of the calls should be managed in order to not have some publishers receive the data multiple times while others do not receive any data.

In order to solve both these problems, we create a new class, named APICaller, which keeps track of the publishers that exist in the system and schedules calls for each of them by iterating through the list of publishers. Because of the fact that access to the API is controlled and performed in turn, we can control the timing of the calls, ensuring that no more than one call to the API is made at a given moment in time, and that successive calls are performed in no less than one second, and thus respecting the imposed limit of the API.

Also in order to adhere to the good design practices concerning data transfer over the Internet, the entire APICaller including the calling of the API is managed by a separate thread. This is important because Internet connectivity can sometimes be unpredictable and data may consume time to be gathered, and response times may vary. It is because of this that Internet interaction is commonly managed on a different thread in order not to stahl the main thread.

Data obtained is then passed to the Publishers by the use of the *receiveResponse()* method. Similarly to the Observer pattern, the added difference being that the Publishers will hook to the APICaller instead of observing it, and instead of using the classical *notify()* method, the APICaller uses the *receiveResponse()* method to notify the Publishers.

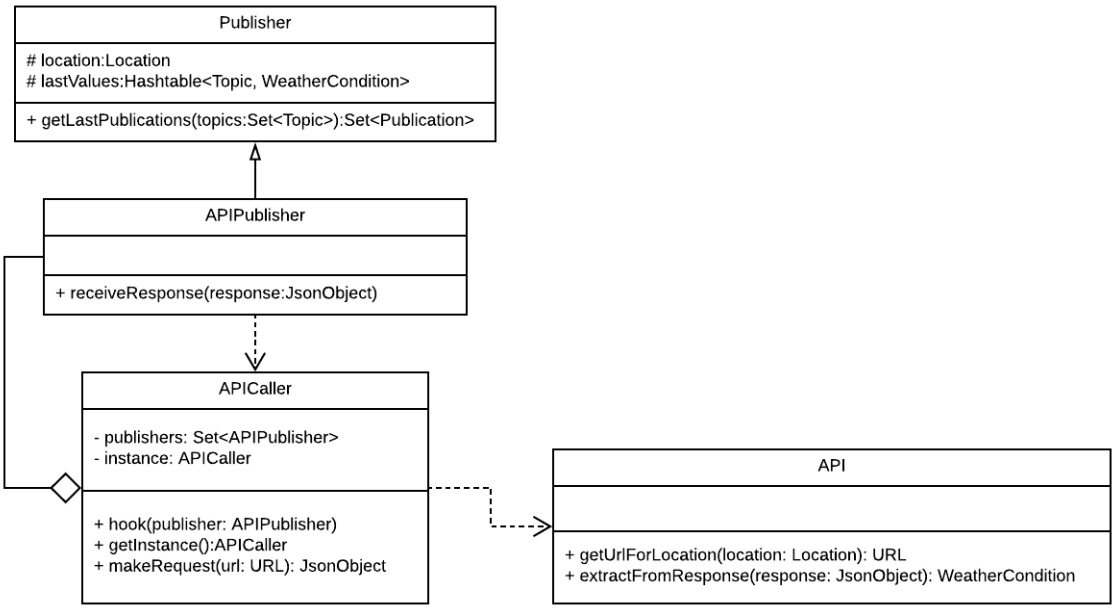


Figure 5.6: Partial Class Diagram for Publisher, APIPublisher, API and APICaller

5.2.1.4. Subscribers

Subscribers represent the recipients of the data provided by the Publishers. In this system, we discuss about two types of subscribers: Subscribers that are physically located with the client, and Subscribers that are running on the MMNS side, acting as a proxy to the real subscribers. Since each Subscriber is mapped in the EventService, an imperative

need arises to ensure that these subscribers will not hold the main execution thread, so each Subscriber will run on an independent thread.

The Subscriber type that resides with the client application will be covered in section 5.3. and the registration and deployment of new Subscriber threads will be covered in section 5.2.2. where the EventService is described more thoroughly.

Before the Subscriber thread is started, the object must be instantiated with the socket that is opened for communication between MMNS and clients. The subscriber thread is then run, and the first operation performed is the retrieval of the input and output streams of the sockets. At this step it is important to check for communication errors in the socket behaviour, in which case the Subscriber thread will be terminated with an error message.

If communication can be successfully established, the Subscriber thread will then wait for a configuration message to be sent from the client side, which is composed of multiple by a series of configuration lines which set the latitude, longitude and topics of the subscriber. Validation of the configuration lines is performed in order to ensure that the parameters are correctly.

The next step is to register to the EventService in order to receive notifications when they are available. For cases in which the Subscriber instance needs to be uniquely identified, the Event service will also generate a unique identification code for each subscriber. For the generation of the identification codes, the UUID class provided in the Java util package will be used.

If the registration is performed successfully, the thread will enter the message receiving loop, which consists of waiting for messages, and then forwarding them to the corresponding client subscription. In order to not use empty cycles of the loop, the subscription thread will go to sleep until a message is received. The Java Concurrent API provides a useful mechanism for this, namely the BlockingQueues.

These queues, as the name suggests, work as regular queues but support concurrency. The most useful part is that if the queue is empty, the consuming thread of the queue will be put in a waiting state until another thread or process adds something to the queue. For this instance, the class ArrayBlockingQueue will be used, and the EventService will route the messages directly to it.

5.2.2. Event Service

The EventService is the core component of MMNS and of the Event Notifier pattern. It is the class responsible with subscription management and message routing. It is because of this that all other objects find themselves having a relationship and interacting with this class.

It can be easily noticed that all objects need to have access to this class at runtime, and being represented as a service, it is a good idea to design this class by the use of the Singleton pattern. Another observation is that concurrent access should be taken into consideration, since subscribers and publishers will mostly perform concurrent access to this class.

The EventService holds the Subscribers and is responsible with their management. The Subscribers are actually held in Clusters of subscribers, which have the role of grouping together the Subscribers that have common features that allow for the messages to reach the correct destinations faster. Because of this, the EventService does

not explicitly have a structure in which to store the Subscriptions, but instead has a register of Clusters. Each time a new Subscriber registers to the EventService, it will have to be correctly placed in one such Cluster.

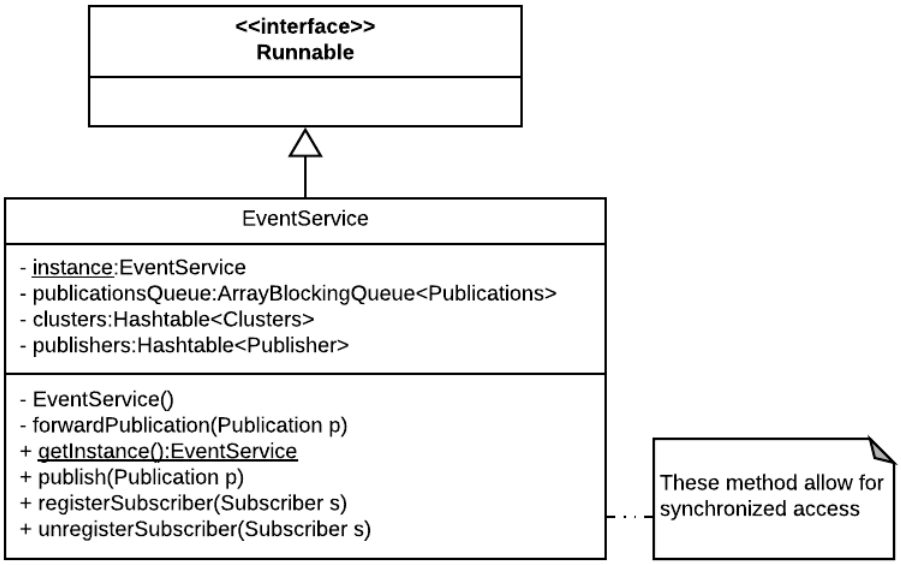


Figure 5.7: EventService Class Diagram

Since MMNS will only have Publishers that are able to serve the set of active subscribers at a specific moment in time, the feature of removing Publishers from the system needs to be available. This is important because it enables the APICaller to only prepare and make calls for the subscribers that are in the system, and if, for instance, one location remains with no active subscribers, one call would be made any way otherwise, consuming the time required with no real need to do so.

5.2.3. Subscription Management. Clustering

Subscription management is the most important function featured in the Event Notifier pattern. The subject of the problem are the Subscriber instances, which must be reached by the publications in a most efficient manner. In order to achieve this, we must take into consideration the following aspects: the routing method, the data structure which holds the subscribers and in this case, an improvement to the Publish/Subscribe model that comes in the form of Clustering.

5.2.3.1. Topics

The routing method chosen for this project is the Topic-based routing. The response that comes from the OpenWeatherApi already hints to a possible breakdown of the weather components into topics. The topics that will be used in this project are as follows:

- Temperature,
- Wind,

- Humidity,
- Pressure,
- Weather.

Subscribers can subscribe to one or more of these topics, in any combination and will receive notifications of events that occur in the respective topics.

The temperature topic will enable users to receive information regarding a current temperature for their location, as well as some possible variation, in the form of maximum variation and minimum variation. Those subscribed under the wind topic will receive updates about the wind speed and direction. The humidity topic only delivers the value of the recorded humidity for the location of the user. Users subscribed to the pressure topic will receive the main pressure reading, as well as the sea level and ground level pressure readings.

Users subscribed to the weather category receive what is perhaps the most desired information, namely a description of the current weather, along with a short title, group and id which serve the purpose to link to more data about the weather found on the OpenWeatherMap documentation. OpenWeatherMap also provides some icons that can be used in order to depict the weather with a graphic form, which can be retrieved from the OpenWeatherMap API using an icon id provided in the initial response.

The best way to realise the topic abstraction was by using an enum simply named "Topic", because they will mostly be used as imposed type codes, or default values that variables can take. The other reason for which Enum values are useful is the fact that it will allow for the topics to be used in conditional statements such as Switch-Case statements, which will most likely be used by Client applications in order to perform operations on received messages. Since their value is constant, Enum values will also reduce compilation and runtime efforts unlike using String constants and provide a form of validation, reducing the risk of passing wrong codes.

5.2.3.2. Clusters

One possibility of managing the subscribers would be to store them into different sets, separated by taking into account the topics that they are subscribed to. Subscribers that are registered for more than one topic will be added to all the corresponding sets; thus, each publication will only require searching through one set of subscribers.

The problem that arrises with this method is generated by the potentially large number of subscribers scattered across multiple locations, and the fact that all publications have the property of targeting one specific location. Because of this, the method of having the subscribers in only six (for the case at hand) sets will generate a potentially long time for finding the subscribers that match the publication location. Also in this case, the routing is no longer topic-based only, it becomes more of a combination of topic- and content-based routing.

In order to reduce the complexity of matching the subscribers we consider the possibility of "clustering", or grouping the subscribers by a characteristic that they share in order to make finding them easier. In this case, the shared characteristic that subscribers may present is the location, that can be easily identified by the location code received upon registration. All location groups (or clusters) of subscribers will be stored in a hash table that can be identified by the unique location code, and inside the group there will be sets of subscribers, divided by their respective topics.

After this, at the cost of one operation the time of searching for the subscribers can be reduced substantially, especially in the cases in which there is a high number of subscribers that are uniformly scattered in multiple locations.

The EventService class will hold the Clusters, adding new subscribers to their respective clusters and creating new clusters when a subscriber for a new location is registered. When a new publication is received, the EventService will delegate the responsibility of finding the subscribers to a new thread, which will go through the corresponding set of subscribers in the specific cluster where they can be found. It is easy to notice that by using this method, the set of subscribers in a cluster represents the complete and correct search result for a publication.

5.2.4. RegistrationService

The RegistrationService is the access point of MMNS. Virtually, it is the place where all clients need to send the registration message. It run continuously starting with de deployment of MMNS, in order to accept incoming connection from new clients.

Being perhaps the simplest class in the project, it implements the Runnable interface in order to support execution on a separate thread and only encapsulates the port number, in terms of class properties. The only implemented method is *run()*, as imposed by implementing the Runnable method.

The thread execution, although trivial, is very important. The thread performs a connection acceptance loop, accepting connection from new sockets and passing them on as access points to new proxy subscriber instances, which will then proceed to the subscription registration process.

This class is also important because it is responsible with the creation and deployment of new Subscriber threads. It does not, however, keep track in any way of the instances it creates. Only after a Subscriber is configured and validated will it be registered to the EventService and a reference to it will be kept in the specific Cluster.

5.2.5. Communication with the Clients. Message Structure

It has already been established that communication with clients will be performed by the use of TCP Sockets. The java.net package supplies two very useful classes that allow Socket communication, namely java.net.Socket and java.net.ServerSocket.

On the MMNS side of the communication, a ServerSocket will be open in order to accept incoming connections from clients. When a client will attempt to open the communication channel in order to establish connection to MMNS, the ServerSocket will accept a new socket representing the client end of the communication. That new socket will be used to create a new proxy subscriber which will then take over the communication with the client.

On the client side of the communication, the address of the MMNS ServerSocket needs to be know in order to attempt connecting to it. The communication resembles the client-server communication method, so the client will be using the regular Socket to open connectivity. Through that socket, the client will receive the messages with the weather updates. All this however is implemented in the library that clients will use to access MMNS.

The first message that is exchanged through the socket will be the configuration message. For this message the direction of communication is from the client to MMNS.

At the beginning, the proxy subscriber will raise a flag to signal that it is under configuration, and be accepting configuration lines in a loop. Each line of configuration will be composed from a parameter name and the value of the respective parameter, separated by the hashtag symbol (#). The parameters to be configured are the latitude, longitude and topics. The order in which the configuration lines are received is not important. Configuration phase ends when the line "END" is received.

After this, the configuration flag will be unset, and the coordinates will be validated. If the coordinates are valid, the Subscriber will try to register to the EventService, upon which a location id and unique identifier will be assigned to it. At this point, communication will only be performed from MMNS to the Subscriber.

The structure of the message sent to the client is again, based on lines. The message begins with a start section:

M_START#<message number>,

and finalises with a pair section:

M_END#<message number>.

This offers validation concerning that messages don't overlap. This is enforced by the fact that the message sending method from MMNS is locked in order to prevent another thread entering and thus causing errors.

Inside the message, parameters of the message are sent in field-value pairs, using the following format:

PARAMETER_NAME#PARAMETER_VALUE,

which are then interpreted by the client and stored into key-value pairs in a hash map.

In MMNS, the message that is published is characterised by both Topic and Location, which are both important for message routing. When reaching the Subscriber however, at least the location is not of interest anymore since it is already recorded in the location field of the subscriber instance. Because of this, two types of abstractions for messages are found in the system, namely the Publication and the Message. The difference between the two is that Publications carry information that is essential for routing, and Messages lose some of the redundant data once the destination has been reached. The Message class also has the logic to translate the content of the message in lines that are directly forwarded by the MMNS Subscriber.

5.3. Client Dependencies

In order to establish communication with MMNS, clients will need to use a library developed specially for this purpose. This section covers the implementation of that library and the functions that it has to offer, as well as some guidelines for integrating the library in a client project.

5.3.1. Implementation of MMNS Client Library

The role of the library is to provide clients with access to MMNS. In order for this to be done, a Connection static class is created, which has the use of encapsulating the connection data and returning a Socket object that Subscriber objects can use to link to MMNS. By having the connection data held as private properties of this class, some

security and access filtering to that data is obtained.

A Subscriber class is also designed in order to provide a working basis upon which clients can add the desired functionality that they ought to have as part of their subscriber entities. This class should offer a partial implementation and force clients to provide message handling functionality, which can easily be obtained by designing the class as an *abstract* class.

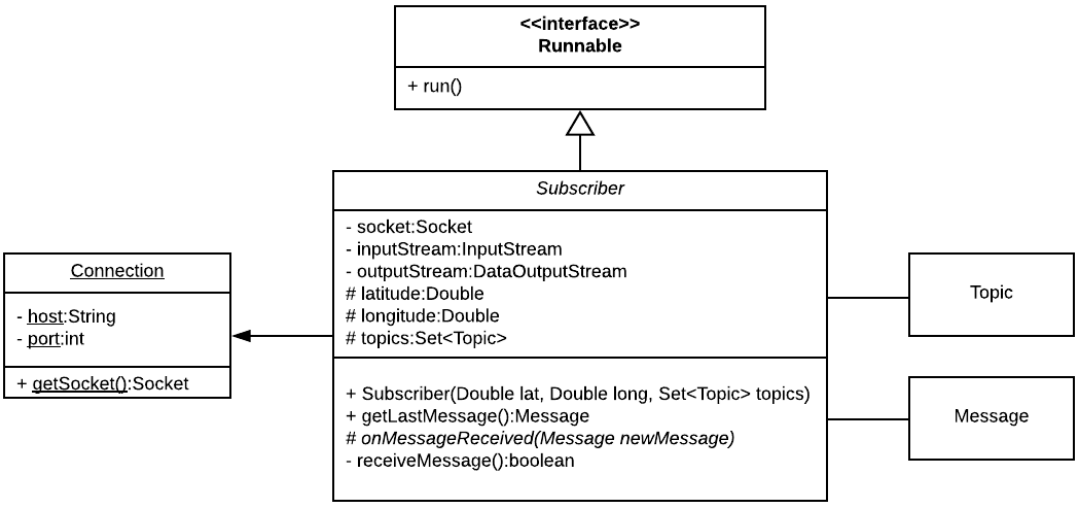


Figure 5.8: Partial Class Diagram of the MMNS Client Library

Since client applications are likely to instantiate more than one subscriber object, it will prove more efficient to also run the subscribers from the client side on independent threads. For this, the Subscriber class implements the Runnable interface, and uses the *run()* method to perform the following tasks: first, the socket instance that is obtained from the Connection class is used to establish communication with MMNS and configure a remote subscriber proxy. After the configuration sequence ends, the subscriber thread enters a loop in which messages are waited upon to be received from MMNS.

The library also contains an abstraction of the Message, class that is used to build Message objects by using the lines that are received from MMNS. The message encapsulates in its fields the message number and the parameters and value received from MMNS, in the form of a hash table. After the message has been fully transmitted between the two sides, the hash table is interpreted in order to be abstracted in a WeatherCondition object, similar to the ones found in MMNS.

In order for a Subscriber to be able to register to the middleware, it is imperative to have information concerning the location and topics that it is subscribed for. In order to ensure this, the default constructor will be replaced with one that requires these fields, in the Subscriber class. In order to start the thread for the Subscriber, it will need to be instantiated first, by the use of said constructor.

An abstract callback method is included in the Subscriber class of the library, which has the use of passing the message to the client once it is received. The client is supposed to access the message by the method parameter of type Message and retrieve the data from it, either by using the key-value pairs or in a more object oriented fashion by using the WeatherCondition classes provided by the library.

5.3.2. Integration of the MMNS Client Library

The library will be made available in **jar** packaging form. In this form it is easy to include it as an external library in any Java project. Clients of MMNS will need to integrate this library in order to register for and receive notifications.

In order to make use of it, clients need to import the jar as an external library in their IDE of choice. The only mandatory step in the usage of the library is to create a class that extends the Subscriber class and implement the callback method *newMessage()* in order to handle the arrival of the new message. Since the Message class defines a *toString()* message, the simplest implementation that a client of MMNS can have is, for example, to print the message to the console.

By the use of inheritance and more coding the functionality of the client library can be adapted to the client need in order to enable a more elaborate and better suited use. What the client wishes to do with the messages that are received is completely up to him; in the proof of case that was built for this project, the client forwards the messages in a modified form to mobile devices by the use of a third party service.

5.4. Mobile Application

In order to provide a working example of how the service provided by MMNS can be used, a mobile application consisting of a application server and mobile client application have been developed as part of this project. This proof of concept application is named WeatherBuddy, and the main goal it has is to use MMNS in order to receive weather updates and deliver them in the form of push notifications to the mobile application.

In the following section the implementation of the application server and mobile client application respectively is described. Implementation of the Application Server begun with the integration of the MMNS library, following with the integration of the Firebase Admin SDK which is needed to deliver the notification to the mobile devices. In order to reach the mobile devices, some information must be provided to the server, which is made possible by having a REST API on the server side which allows the mobile devices to register the respective identification data.

5.4.1. Application Server

The application server is a very simple system, which has the sole responsibility of a low management of the mobile device subscriptions and message forwarding between MMNS and the mobile devices. Because of the platform restrictions of the smartphones today, the server cannot send the notifications directly to the mobile devices; instead, the use of a platform specific service is required for pushing notifications to the mobile devices.

The general architecture of the server application is not complex, since the application is not complex itself. It is a partial layered architecture with two layers, namely the presentation layer (or rather in this case, simply the controller layer) and a business layer. There is no data layer in this application because the data managed is not as complex as to require persistence manager, and information can change so fast that is less complicated to hold a structure at the business layer which manages the data.

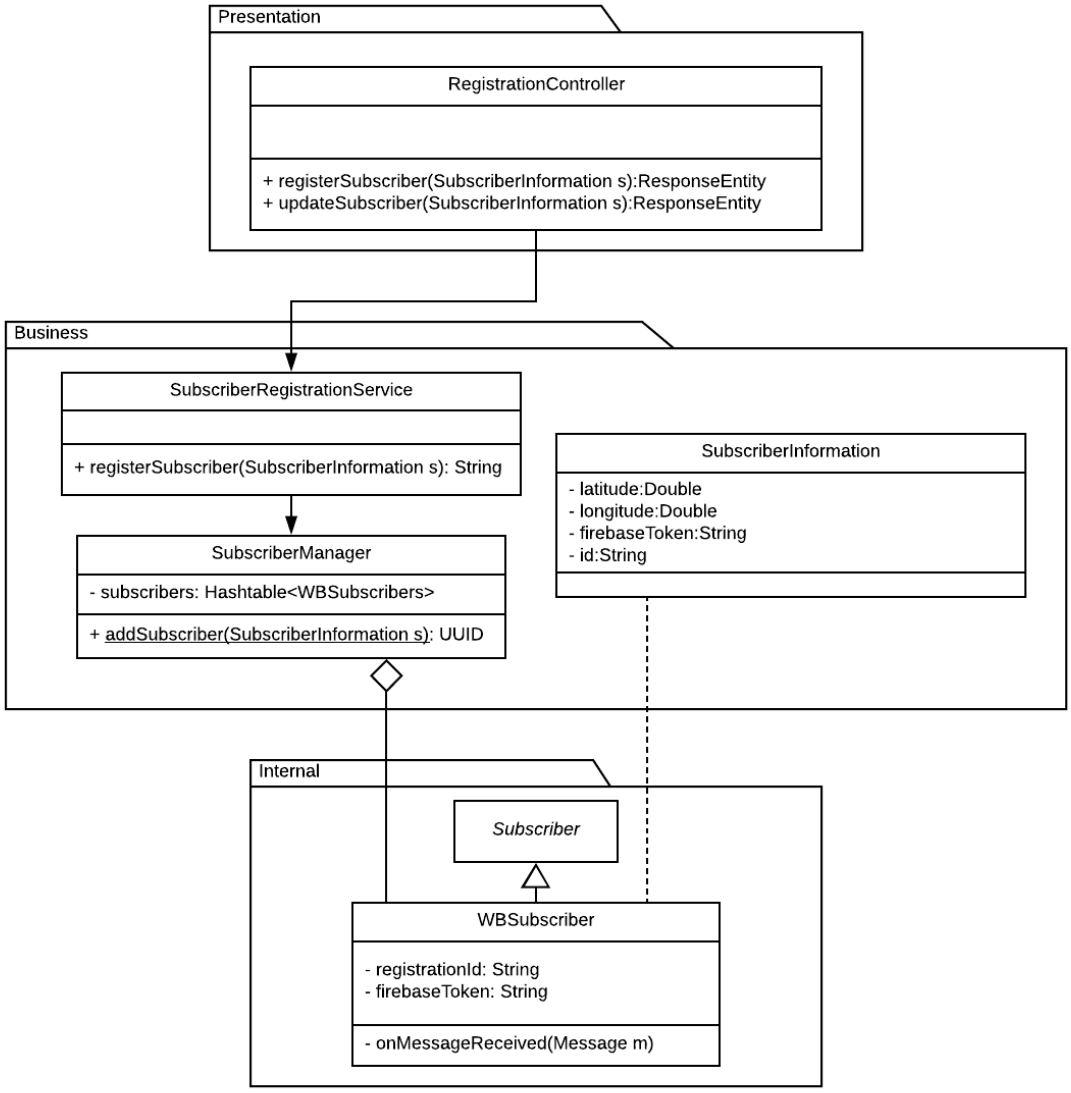


Figure 5.9: WeatherBuddy Server Application Class Diagram

5.4.1.1. MMNS Library Integration

After the project has been created, the MMNS client library is added as an external library to the class path. After this, the server can extend the Subscriber class in order to envelope the mobile subscribers and virtually registering them to MMNS.

The abstract Subscriber class is extended by a class named WBSubscriber (the naming follows the principle of adding a prefix to the parent class name in order to clarify usage and easily identify relationships), which needs to add two more class variables which are very important for the system: a registration id and a firebase token that was given to the device upon registering to firebase. The registration id is given to the subscriber when the mobile application registers to the server in order to receive updates.

Technically, Subscribers could be instantiated and deployed on a thread without keeping a reference to the respective object, and it will run for as long as the socket used

for communication with MMNS is sustained. However, due to the fact that the Subscriber instances are actually virtual representations of the mobile devices, it can be assumed that the mobile will change location at certain times, moving to a place with coordinates that generate a different location id and can therefore have different weather. In order to enable this functionality on the application server side, it will need to keep track of the subscribers in order to receive location updates and change the instance that receives messages from MMNS.

The business layer of the server contains a class developed to aid in this direction, namely the SubscriberManager class. This class holds all the subscribers in a hash table in order to provide fast access for editing, adding and removal. Upon adding a new subscriber, the class returns a unique identifier which will be used to find the subscriber.

The business layer also includes the class SubscriberInformation which is used to pass the data provided by the mobile application to the server using the REST API, as well as the SubscriberRegistrationService which is used by the controller to register subscribers to the SubscriberManager.

5.4.1.2. Integration of Cloud Messaging Service

The Firebase Admin SDK provides programmed access to the services offered by Firebase. Out of these services, the one most important to the project at hand is the possibility of programatically sending messages using the Firebase Cloud Messaging system, which is actually the native push notification platform for Android smartphones.

```
Message message = Message.builder()
        .setAndroidConfig(AndroidConfig.builder()
            .setTtl(3600*1000)
            .setPriority(AndroidConfig.Priority.NORMAL)
            .setNotification(AndroidNotification.builder()
                .setTitle(lastMessage.getParameters().get("TOPIC"))
                .setBody(lastMessage.toString())
                .build())
            .build())
        .setToken(firebaseToken)
        .build();
String response = "fail";
try {
    response = FirebaseMessaging.getInstance().sendAsync(message).get();
} catch (Exception e) {
    e.printStackTrace();
}
```

Figure 5.10: Message Composition using Firebase Admin SDK

Adding the Firebase Admin SDK to the project is performed simply by following the instructions presented in the Firebase documentation[13], under the gradle instruction

[13] Add the Firebase Admin SDK to Your Server | Firebase (Online, retrieved June 30th 2018) https://firebase.google.com/docs/admin/setup

for adding the dependencies to the build.gradle file and the JAVA instructions for implementation steps.

After adding the dependencies to the project and performing the instructions from the documentation, messages can be sent to the mobile devices using the firebase token provided upon registration. In the *onMessageReceived()* function of the WBSubscriber, a Message object is instantiated, this time using the class defined in the package provided by the Admin SDK, com.google.firebase.messaging.Message. The class can be instantiated using the builder pattern, using the Android configuration. The message is then sent asynchronously.

The time in which it reaches the mobile application depends on multiple internal factors, but the factor that can be controlled is the priority, which can either be set to **normal** or **high**.

5.4.1.3. API For Device Registration

The mobile application needs a way to send some data regarding the device in order to register for notifications, namely the location of the device. The way in which the server allows this is by the use of a REST API that provides a POST method which will be used to register new devices.

The Spring frameworks allows for a quick design of a REST controller by the use of annotations. The controller functionality is implemented in the RegistrationController class, which will be automatically be bidden to the servlet (in this case Tomcat) by the use of the @RestController and @RequestMapping annotations. The second annotation also accepts a parameter which indicates a first path component, which in this case will be simply "/api".

Two methods are defined in this class which define the two POST methods that will be supported by the API. The first one is *registerSubscriber(SubscriberInformation info),* which will be mapped to the "/register/" request path, as a POST method intaking Json media type in the request body. The purpose of this method is to allow the mobile application to register to the server in order to receive push notifications. The application returns a ResponseEntity with the CREATED HTTP code if successful and the body will also include the identification code generated by the server for the purpose of identifying the device.

The other method is *updateSubscriber(SubscriberInformation info),* which will be mapped to the "/loc-change/" request path, as a POST method also intaking Json media type in the request body. The application returns a ResponseEntity with the OK HTTP code if successful. This method is used for registering a location change for a subscriber which already has been registered to the server, so the subscriber id will also be provided as part of the request body.

5.4.2. Mobile Application

The purpose of the mobile application is solely to receive push notifications from Firebase. Because of this, the focus will be on the Firebase Cloud Messaging service integration and communication with the application server in order to register devices for receiving push notifications.

5.4.2.1. Push Notification Service Integration

Adding the Firebase Cloud Messaging dependencies to the Android project was the biggest challenge due to the fact that Firebase is still a young project that grows quickly. New releases are brought to the public every few weeks and because of this the documentation available on the website may not be up to date at all times.

Even in this case however, following the instructions available in the Firebase documentation[14] and performing some slight modifications that were available online enabled the application to register to the FCM Service and receive notifications both from the Firebase online console and from the WeatherBuddy server once the device was registered.

After adding the dependencies in the gradle build file, the service has to be registered in the Manifest file. The specification indicates the name of a class that extends the abstract class FirebaseInstanceIdService, which is needed in order to handle the token refresh that may occur at times, especially at the first launch of the application, when the FCM Service releases a new firebase token in order to let Firebase identify the device. The name of the class that provides this functionality is WBFirebaseInstanceIdService, following the same naming principle of adding a prefix to the parent class name. This time the prefix comes from the project name, WeatherBuddy.

The token is made available to the application through the *onTokenRefresh()* callback method of the WBFirebaseInstanceIdService class, and will be managed by the application with the final goal of letting it be known to the server side. That will allow the server to programatically send notification messages to Firebase, which will use the provided token to route it to the right device.

5.4.2.2. Location Services

The first step that needs to be completed in order to access the location of a device running on the Android operation system is to request the permission to do so. For this, the permission needs to be specified in the application manifest file, and the user must grant permission in order to allow the application to collect the location of the device. For this purpose a popup must be presented to the user at the right time to inform that the application needs to use the current location and prompt the user for his permission.

In the case of this application, it is mandatory to prompt the user for permission at application launch, because without the location of the device the application cannot serve its purpose. Once the user agrees for the location to be used, the application will request location updates through the location manager of the context, and will have to supply a LocationListener class that will receive the location updates.

WBLocationListener is the class that extends LocationListener in this project, implementing the *onLocationChanged()* callback method. By the parameter supplied in this method the location of the device can be accessed and forwarded to the server. The method will set the new location to the WBSubscription which will then (if necessary) forward it to the server.

Because of the location accuracy being approximated, small errors may occur sometimes, and location updates may be triggered erroneously. Due to this fact, some

[14] Adding Firebase to Your Android Project | Firebase (Online, retrieved July 1st 2018) https://firebase.google.com/docs/android/setup

logic must be provided in the application that will check if the location change is signifiant, so not all small changes will be reported to the server. In this case, a movement is considered to be significant only if the distance is larger than 0.15 coordinate points in any direction, where the coordinates are considered to adhere to an Euclidean metric space.

5.4.2.3. Registering to the Server

Registering to the Server is performed through the REST API of the server. The challenge with packaging all the information that needs to be delivered to the server in order to post it at once however arises, because the location and firebase token are not obtained in a sequential manner, at a fixed time, but instead they are both obtained asynchronously from a callback method.

In order to solve this problem, a new class is designed that will manage the information received from the callback methods and once every required parameter is known, issue a POST request to the Server API. This class is named WBSubscription because it holds the information needed for posting a new subscription request to the WeatherBuddy server. Since only one instance of this class should exist at all times, the creational design pattern Singleton will be used for it.

For the cases in which the location data and/or Firebase token are not both retrieved in one run of the application, the currently known data will be stored in the SharedPreferences structure. This information will be loaded and saved at the launch of the application and at the exiting of the application respectively.

In order to simplify the server communication, Google's library Volley[15] is used. Calls are made to the server by the WBSubscription class, which will try this every time a new parameter (location or firebase token) is obtained. If the purpose of the call is registering to the server, then a registration id will be expected as a response. If the call is made to update the location or firebase token of the subscriber that is already registered, the call will be made to the update method of the API, posting also the already existing server registration id.

[15] Volley Overview | Android Developers (Online, retrieved July 1st 2018) https://developer.android.com/training/volley/

Chapter 6. Testing and Validation

The goal of this chapter is to present the methods of testing and validation of MMNS and its components. The main scenarios for testing will be created and presented, and an evaluation of the performance of the system will be also discussed.

Manual testing will be performed for the main scenarios and the tools provided by the IDE used for implementation will aid in monitoring the system in order to conclude if the desired changes have occurred.

The main goal of testing is the identification of possible problems or erroneous behaviour of the system, which have not been identified during the design and implementation phase of the system, for the purpose of correcting them.

6.1. Test Cases

6.1.1. Registering a New Subscriber

In this scenario, a client of MMNS tries to register for receiving updates of the current weather at its location by using the MMNS client library. For the purpose of this test, the subscriber will be initialised with the coordinates of the city Cluj-Napoca and will register to all topics. As functionality, it will only print the message to the console.

The expected behaviour of MMNS is the following:

- Receive the connection on the server socket and delegate it to a new proxy subscriber instance;
- The proxy subscriber will receive the configuration consisting of topics and coordinates;
- The subscriber will register to the EventService

```
[RegistrationService]    new connection accepted
[Subscriber Thread-3]       Waiting for configuration...
            CFG LINE: LAT#46.77
            CFG LINE: LON#23.62
            CFG LINE: TOP#PRESSURE
            CFG LINE: TOP#TEMPERATURE
            CFG LINE: TOP#WIND
            CFG LINE: TOP#WEATHER
            CFG LINE: TOP#HUMIDITY
            CFG LINE: END
[Subscriber Thread-3]       Configuration completed.
[APICaller]         Hooked publisher for Cluj-Napoca
Registration completed for Subscriber 97026e33-d597-4855-8bb4-79890e52a62f in Cluj-Napoca
```

Figure 6.1 Console Log of MMNS After Subscriber Registration

The Console of MMNS (Figure 6.1) can be consulted in order to observe the execution of these steps. As it can be seen, after a new connection has been accepted in the RegistrationService, a new proxy Subscriber will be run, in this case on Thread-3.

A correct behaviour of the system will lead to the existence of a new thread running under the process of MMNS, mostly in a WAITING state, unless it received a new message that needs to be forwarded. In order to observe this, a snapshot of the thread dump for the current run can be consulted in the IDE. Figure 6.2 and Figure 6.3 represent the thread dump before and after the registering of a new subscriber.

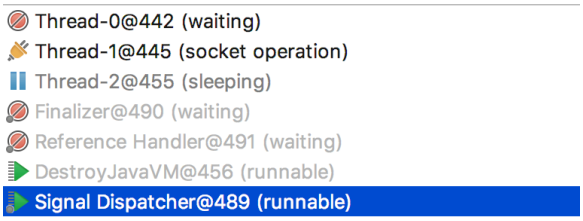


Figure 6.2 Snapshot of the Thread Dump Before the Registration of a New Subscriber

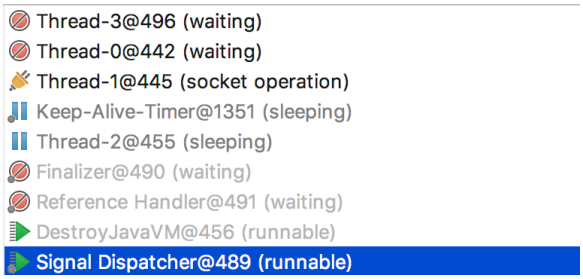


Figure 6.3 Snapshot of the Thread Dump After the Registration of a New Subscriber

6.1.2. Message Sending and Receiving

This scenario considers that a subscriber is already registered in MMNS and a new publication is available for it to consume. For this test the same subscriber instance will be considered. The callback method that signals the receiving of a new message has the simple implementation of printing the message to the console.

The expected behaviour of the client integrating the MMNS library upon receiving a new message is the following:

- The message is first validated and checked for consistency; if something is not right, a message will be printed in the console;
- The current time of the system will be added to the message in order to mark the arrival time;
- The *onMessageReceived(Message m)* method is called from the Subscriber thread;
- The message will be printed in raw form in the console.

The results of this scenario can be observed immediately after the deployment of the Subscriber due to the fact that MMNS will either create a new Publisher entity which will immediately publish the first reading, or send the last publication using the same mechanics that are used for new ones.

As presented in Figure 6.4, the first message that has been received is a message regarding the last update of the Humidity in Cluj-Napoca, listing all message parameters in readable form: the message count, humidity reading, the topic of the message, the time of publication, computation and arrival and the location that the message concerns.

```
M: 1
HUMIDITY: 33.0
TOPIC: HUMIDITY
TOP: 1530708169320
TOC: 1530705600000
TOA: 1530708169323
LOCATION: Cluj-Napoca
```

Figure 6.4 Message Received by the Subscriber

6.2. End-to-end Latency

Since the goal of the project is to deliver data to the clients as soon as it becomes available, an important metric that needs to be considered is the time that passed since the moment when data has been made available and the moment it reaches the subscriber. This time period is referred to as "end-to-end" latency.

The lifetime of the data obtained from the service follows a specific path:

- Processing of the data that is performed by OpenWeatherMap (TOC);
- Retrieval of the data by accessing the OpenWeatherMap API, and the analysis performed in order to see if the weather has changed. If the weather has changed, a publication is created and delivered to the EventService. At this step, the current system time is attached to it in order to mark the time of publication (TOP).
- The publication is received by the EventService which will then route it to all the proxy subscribers which need to be reached by it. The proxy subscribers forward the data to the real subscribers, which upon receiving the data will add the current system time as a marker of the time of arrival (TOA).

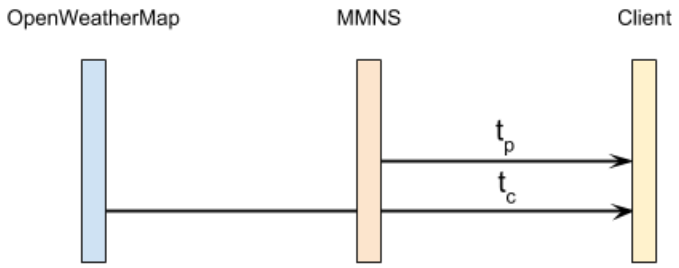


Figure 6.5 Latency Introduced Between the Parts Involved in Data Transfer

The time of the data computation can be obtained from OpenWeatherMap at the same time as the current weather data. The timestamp however does not correspond to the time of data retrieval of the data, but to the time when the data was computed. As the documentation states, depending on the type of account, the data that the API provides will be updated at different rates, with the highest paying accounts having access to data that is refreshed more times and is made available as soon as it has been published. This means that even though the data will be obtained in no more than a few seconds since it has been provided, as much as several minutes may have passed since the time of calculation. The guaranteed time of update of the API data is less than two hours.

Because the time elapsed between the data calculation and arirval at the subscriber (t_c) because of the low refresh rate, it is important to have as little time as possible between the time of publication and time of arrival at the subscriber (t_p). Using the timestamps provided at each step, we can compute the two values by using the following formulas:

$$t_c = \frac{TOA - TOC}{1000} \qquad t_p = \frac{TOA - TOP}{1000}$$

In order to provide a minimal simulation for the purpose of testing this value, multiple groups of subscribers have been deployed:

- City of Deva: 1 subscriber;
- City of Iași: 3 subscribers;
- City of Cluj-Napoca: 7 subscribers;
- City of Bucharest: 9 subscribers.

A summary of the values that the simulation has provided is presented in Table 6-1, where a general overview of the simulation can be viewed.

Table 6.1 Maximum and Average General Values

	t_c (seconds)	**t_p (seconds)**
Maximum values	3080.594	0.074
Average values	1615.145	0.003

Table 6-2 presents the values obtained from the simulation organised by location. This is relevant because of the way in which MMNS manages subscriptions, which is in the form of clusters, organised by location.

Table 6.2 Maximum and Average Values by Location

City	**No. of subscribers**	**No. of messages**	**Maximum t_c (s)**	**Average t_c (s)**	**Maximum t_p (s)**	**Average t_p (s)**
Deva	1	10	597.098	177.856	0.004	0.002
Iași	3	102	2939.358	1492.655	0.008	0.002
Cluj-Napoca	7	304	3040.753	1665.753	0.007	0.002
Bucharest	9	304	3080.594	1657.879	0.074	0.004

Chapter 7. User's manual

7.1. Deployment of the Middleware

7.1.1. Hardware Requirements

Because MMNS is designed to run as a multi-threaded application, it is recommended for the system to support concurrency, either by having more than one CPU or by having a multi-core CPU.

The machine on which MMNS will be deployed also needs to support Internet connection, having a wired or wireless access point available.

7.1.2. Software Requirements

In order to run the MMNS executable file, the latest version of Java (JRE or JDK) needs to be installed.

This manual is written for running MMNS on macOS version 10.13.1, meaning that the command line instructions may not be the same for other operating systems.

7.1.3. MMNS Deployment

In order to run MMNS, the following steps need to be followed:

- Open the Terminal application;
- In the Terminal application navigate to the location of the MMNS.jar location in the file system;
- When in the enclosing directory of MMNS.jar, type the following command in the Terminal application window:

```
java –jar MMNS.jar
```

- After this, the cursor will blink in a new line. This indicates that the application is successfully started and MMNS is waiting for subscribers to register on port 1978.

7.2. Integration of the Client Library

The client library that has been developed as part of this project is available in the form of a JAR file, which means that it can be used to develop Java applications that will use MMNS. In this form it is also easy to add it as an external dependency to the build path.

In this example, the client library will be integrated in a project created using the Eclipse Neon IDE.

7.2.1. Adding the Dependency to the Build Path

The library can be added to a new project or to an already existing one, by right-clicking on the project folder in the "Package Explorer" view, and selecting "Build Path",

"Configure Build Path" from the drop-down menu.

In the "Java Build Path" properties window, select the "Libraries" tab by clicking on it. Select the "Add External JAR…" option at the right side of the window. In the file explorer window that appeared, navigate to the location of the "sub-1.0-SNAPSHOT.jar" file and add it to the build path.

7.2.2. Creating a Subscriber

In the project, create a new class which will extend the common.Subscriber class, representing the subscriber that will be created in the desired system. The class will need to implement two methods:

- A constructor that acquires the geographical coordinates and set of topics that the subscriber uses to register;
- The callback method *onMessageReceived(Message message)*.

The constructor needs to at least make a call to the super-class constructor in order to provide the information needed to register the subscriber to MMNS.

The callback method will allow the application to handle what happens when a new message is received. In this example the message will be printed in the console.

```
public class MySubscriber extends Subscriber {

    public MySubscriber(Double latitude,
            Double longitude,
            Set<Topic> topics) {
        super(latitude, longitude, topics);
    }

    @Override
    protected void onMessageReceived(Message message) {
        System.out.println(message);
    }
}
```

Figure 7.1: Example of Implementing a Subscriber Class

7.2.3. Registering a Subscriber to MMNS

The class that has been designed at the previous step can be instantiated with the constructor and providing the essential data that is needed for a Subscriber to register to MMNS. The subscriber class is designed to work on a separate thread, so in order to function, a new Thread object needs to be created, passing in the instantiated Subscriber as parameter to the constructor.

After that, the *start()* method needs to be called on the thread object that was previously created. That will start the execution of the subscriber thread, which will establish a connection to MMNS assuming it is running on the same host, on port 1978. When MMNS sends a new message to the subscriber, the callback method will be executed.

In the installation description section your should detail the hardware and software resources needed for installing and running the application, and a step by step description of how your application can be deployed/installed. An administrator should be able to perform the installation/deployment based on your instructions.

In the user manual section you describe how to use the application from the point of view of a user with no inside technical information; this should be adorned with screen shots and a stepwise explanation of the interaction. Based on user's manual, a person should be able to use your product.

7.3. Deployment of WeatherBuddy

WeatherBuddy is comprised of two components:

- Server application;
- Mobile application.

In order to run the two components, the following software tools will be used:

- InteliJ-IDEA Ultimate (latest version)
- Android Studio (latest version)

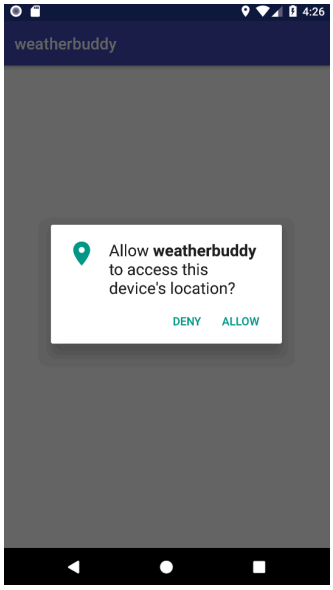
7.3.1. Running the server component

The first step in order to be able to run the Server component of the application is to import the "weatherbuddy-server" project in InteliJ IDEA. This can be done by right-clicking the "pom.xml" file and chosing to open it with InteliJ IDEA.

After the project is important it is important to make sure that the client application is added as an external dependency to the build path; if this is not the case, then the next step is to add it.

After the environment is set up, the project can be run.

7.3.2. Running the mobile application



The "weatherbuddy" project needs to be imported in Android Studio. After the import is successful, start the server application if it is not already running.

After server initialisation is complete, run the mobile application. Android Studio will look for connected devices in order to install and run the application on them. Choose one of the devices or create a virtual device if no physical device is connected to the computer. If choosing the virtual device option, make sure that the configuration of the device allows for the usage of Google Play Services, because these services are used to retrieve the Firebase token.

After the application launches, the user will be prompted to allow the application to collect location data. In the presented popup, select "allow" in order to allow the application execution to proceed. If the coordinates that are set in the emulator are valid, and a Firebase token has been

received, the application will immediately connect to the server. In this case, the application will log a message in the console indicating the successful registration to the server.

If needed, the coordinates of the virtual devices can be set from the emulator settings by clicking on the “location” option and entering some coordinates in the available fields and then clicking “send”.

The notification can be viewed by accessing the notification centre of the device (swiping down from the top of the screen). In this view, the message title and body can be viewed, as depicted in Figure 7.2.

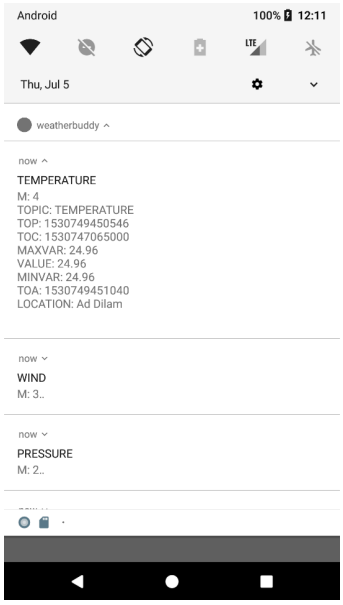


Figure 7.2: Push Notification of Weather Data

Chapter 8. Conclusions

8.1. Obtained Results

The goals described in the second paper have been achieved by the implementation of MMNS and the Client Library. MMNS can monitor the current weather data obtained from a REST API and deliver it to the subscribers, without them needing to ask for it. This establishes the transition from *information-on-demand* to *information-when-changed*, and thus eliminates the need for users to monitor the information by performing polling loops.

During the implementation phase of the project, four components have been successfully developed:

- MMNS
- Client Library
- WeatherBuddy Server Application
- WeatherBuddy Android Mobile App

8.2. Personal Contributions

This project documents the realisation of MMNS and WeatherBuddy application, starting with the choice of architectural and design patterns for each component and ending with the implementation of the middleware, client library and mobile application and server side for WeatherBuddy.

The choice of generic architectural patterns that suits each component has been influenced by research of similar systems and patterns that are used with similar applications. The final design choice as well as coding practices were personal decisions.

The form in which the components of MMNS (middleware and client application) communicate with one another was also a personal decision, especially regarding the message structure, notation and interpretation.

8.3. Future Improvements

Once subscribers are registered to MMNS, communication only is available in one direction, which is from MMNS to its clients. Although for the purpose it was designed this is sufficient, enabling the subscribers to reconfigure some preferences during runtime can allow for a larger number of features of the system.

The most important feature that should be added to MMNS and the Client Library is to allow the update of the subscriber location and topics of interest without the need of restarting the Subscriber thread and removing and re-registering the Subscriber in the EventService. In order for this to happen, MMNS will have to be modified in order to accept incoming messages from subscribers, not only deliver messages.

Currently the time it takes the current weather data is not short enough, as it can be several minutes, and it is essential for the subscriber to be informed as soon as the information is changed. The significant latency is added by the restrictions imposed by the OpenWeatherMap services for a free account, but the quality of service can be improved by using one of the paid versions of the API.

The WeatherBuddy application can also be improved by the addition of a user interface that presents the last update of the current weather at the user location. Other features such as adding multiple locations of interest to be notified about can be added to the application.

Bibliography

[1] Bruno S. Frey and Margit Osterloh, “Successful management by motivation: balancing intrinsic and extrinsic incentives”. Berlin: Springer, 2010, p. 234.

[2] Andid K. Dey, "Understanding and Using Context," *Personal and Ubiquitous Computing*, vol. 5, no. 1, pp. 4-7, February 2001.

[3] John C. Krumm, Eric J. Horvitz, and Ramaswamy Hariharan, "Integration of location logs, GPS signals, and spatial resources for identifying user activities, goals, and context.," Patent no. 7,925,995, April 12, 2011.

[4] Marvin M. Theimer et al., "Selective delivery of electronic messages in a multiple computer system based on context and environment of a user," Patent no. 5,493,692, February 20, 1996.

[5] Niels Henze, Martin Pielot, and Albrecht et al. Schmidt, "A Large-scale assessment of mobile notifications," in *Conference on Human Factors in Computing Systems - Proceedings*, Toronto, 2014.

[6] Luchen Tan, Adam Roegiest, Jimmy Lin, and Charles L. A. Clarke, "An Exploration of Evaluation Metrics for Mobile Push Notifications," in *39th International ACM SIGIR conference on Research and Development in Information Retrieval*, New York, 2016.

[7] Kyle Lynn Schutt, VTGemini: Universal iOS Application for Guided Emergency Response and Notification for the Virginia Tech Community, April 23, 2013.

[8] Ministerul Afacerilor Interne. Departamentul pentru Situatii de Urgenta. DSU - Mobile Application [Online]. HYPERLINK "http://www.dsu.mai.gov.ro/"

[9] Menghao Zhang, An intelligent system for contextual notifications solution, 2016.

[10] Janick F. et al. Artiola, *Environmental Monitoring and Characterization*, Mark L. Brusseau, Ian L. Pepper, and Janick Artiola, Eds.: Elsevier Science & Technology Books, 2004.

[11] Aleksandar Antonić, Martina Marjanović, Krešimir Pripužić, and Ivana Podnar Žarko, "A mobile crowd sensing ecosystem enabled by CUPUS: Cloud-based publish/subscribe middleware for the Internet of Things.," *Future Generation Computer Systems*, no. 56, pp. 607-622, December 2016.

[12] Mohammad H. Alomari, Hesham Abusaimeh, Saadi Shahin, and Rajai Joudeh, "Postat: A Cross-Platform, RSS-based Advertising and Event Notification System for Educational Institutions," *Procedia - Social and Behavioral Sciences*, vol. 73, pp. 120-127, February 2013.

[13] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec, "The Many Faces of Publish/Subscribe," *Computing Surveys*, vol. 35, no. 2, pp. 114–131, June 2003.

[14] Ludger Fiege, Felix C. Gartner, Oliver Kasten, and Andreas Zeidler, "Supporting Mobility in Content-Based Publish/Subscribe Middleware," , vol. LNCS 2672,

2003, pp. 103–122.

[15] Legatheaux J. Martins and Sergio Duarte, "Routing Algorithms for Content-based Publish/Subscribe Systems," Faculty of Sciences and Technology, New University of Lisbon, Lisbon, Technical 2007.

[16] Ying Liu and Beth Plale, "Survey of Publish Subscribe Event Systems," Computer Science Department, Indiana University, Bloomington, Technical 2003.

[17] Toni A. Bishop and Karne K. Ramesh, "A Survey of Middleware," *Computers and Their Applications*, pp. 254-258, March 2003.

[18] Margaret Rouse, Sarah Wilson, and Ed Hannan. (2016, December) techtarget.com. [Online] Available https://searchmicroservices.techtarget.com/definition/RESTful-API

[19] Alberti Jordi Gomez, User-Centered Design of a weather forecast application for smartphones, June 17, 2016.

[20] Tobias Fiebig et al., SoK: An Analysis of Protocol Design: Avoiding Traps for Implementation and Deployment, October 18, 2016.

[21] Rest Hooks. Rest Hooks. [Online]. Avaliable http://resthooks.org/docs/

[22] Antony I. Wasserman, "Software engineering issues for mobile application development," in *Proceedings of the FSE/SDP workshop on Future of software engineering research*, 2010, pp. 397-400.

[23] Suhas Holla and M. Katti Mahima, "Android based mobile application development and its security," *International Journal of Computer Trends and Technology*, vol. 3, no. 3, pp. 486-490., June 2012.

[24] Sasu Tarkoma, *Publish/Subscribe Systems. Design and Principles*, David Hutchison, Serge Fdida, and Joe Sventek, Eds. Helsinku, Finland: John Wiley & Sons Ltd, 2012.

[25] Suchitra Gupta, Jeff Hartkopf, and Suresh Ramaswamy, "Event Notifier, a Pattern for Event Notification," *Java Report*, vol. 3, no. 7, July 1998, [Online]. Available http://www.marco.panizza.name/dispenseTM/slides/exerc/eventNotifier/eventNotifier.html.