Design and implementation of a distributed Web IoT application using microservice architecture for device control and measuring

Diploma Thesis

By

Christos Theologou

Department of Electrical and Computer Engineering
University of Thessaly

A dissertation submitted to the University of Thessaly in accordance with the requirements of the Diploma degree in the department of Electrical and Computer Engineering.

September 2020

Περίληψη

Τα τελευταία χρόνια,με τη συνεχή ανάπτυξη και εξέλιξη του Internet of Things (IoT), η μονολιθική αρχιτεκτονική εφαρμογών γίνεται πολύ μεγαλύτερη σε κλίμακα και ακόμη πιο περίπλοκη δομή. Αυτό οδηγεί σε κακή επεκτασιμότητα και συντηρησιμότητα. Ανταποκρινόμενοι σε αυτές τις προκλήσεις, η αρχιτεκτονική των microservices έχει εισαχθεί στον τομέα των εφαρμογών IoT, λόγω της ευελιξίας, της ελαφριάς και χαλαρής ζεύξης που προσφέρει.

Η προσφορά της παρούσας εργασίας είναι η περιγραφή και η ανάλυση της αρχιτεκτονικής των μικρο-υπηρεσιών(microservices), παρουσιάζοντας τις λύσεις και τις δυνατότητες που προσφέρει αυτή η αρχιτεκτονική σε σύγκριση με τα μονολιθικά συστήματα. Επιπλέον, παρουσιάζουμε τον σχεδιασμό και την υλοποίηση ενός συστήματος IoT με microservices και δίνουμε έμφαση στην βασική εξυπηρέτηση και επικοινωνία των συσκευών απο επέπεδο υπηρεσίας σε φυσικό επίπεδο.

Abstract

In recent years, with the continuous development and evolution of the Internet of Things (IoT), the monolithic application architecture becomes much larger in scale and even more complex in structure. This leads to poor scalability and maintainability. Responding to these challenges, the microservices architecture has been introduced in the field of IoT applications, due to the flexibility, light and loose connection it offers.

The presentation of this work is the description and analysis of the microservices architecture, presenting the solutions and possibilities offered by this architecture in comparison with the monolithic systems. In addition, we present the design and implementation of an IoT system with microservices and emphasize the basic service and communication of devices from service level at the physical level.

Dedication and acknowledgements

In the first place I want to thank my lead supervisor, Dr. Athanasios Korakis and Dr. Dimitrios Katsaros for their assistance and guidance for the completion of this work. I would also like to thank my friends and my family that were by my side all these years.

To my family and friends.

Author's declaration

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Research Degree Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, the work is the candidate's own work. Work done in collaboration with, or with the assistance of, others, is indicated as such. Any views expressed in the dissertation are those of the author.

SIGNED: ..... DATE: .....

Table of Contents

List of Tables

Table **Page**

List of Figures

Figure **Page**

Chapter 1

Introduction

1.1 Internet of Things

The term IoT (Internet of things) is not very new, there have been several different names and one of such names is “Internet of Infinite Things”, the dreamworld which everything may communicate with each other. Today, we have a large number of physical entities interconnected and integrated into the information space to interchange generated data using communication technologies. Previously a number of research work focused on the connectivity challenges. However, with newly developed systems, the existing structure has given birth to several new research challenges. With the development of technology, the IoT applications have started to consider how to integrate the existing network facilities and the openness and scalability of the system design. Compared with existing services, such as telecommunication services or Internet applications, IoT service faces with new situation. The huge information collected from perception layer (involving a large number of sensors) is further processed and delivered to different applications according to the requirements, then is used to trigger the corresponding collaborative business system. As the IoT technology is widely used in daily life, the events produced by sensors and objects are becoming astronomical. The task is enormous and immense services response to the requests. IoT services system should coordinate man-power and business process to quickly interact with the physical entities across the business domains, even across the organizations based on the processing and integrating of distributed multisource sensing information. Because of the variety of business process, personal and physical entities involved, physical world continuously changes, IoT services are characterized by real time changes. It brings new technical challenges to IoT services such as: heterogeneity of hardware, network and operating system, interoperability between applications and services, fusion of massive heterogeneous data, scalability and continuous integration.

Interoperability between applications and services. In the IoT application, many actors comprising human and nonhuman objects and many systems are designed for specific applications, adopting spe-

cific data standards and communication platforms. Different platforms lead to great inconvenience for achieving interoperability and mutual communication in IoT applications.

Fusion of massive heterogeneous data. While data collected in the IoT generates billion pieces of information with hundreds of data formats, it's hard to classify vast amount of raw data by a complex module in traditional architecture, and it also easily leads to system overload if a large number of redundant data directly goes to the application layer.

Scalability and continuous integration. IoT aims to keep on growing all the time and it has generated a lot of complex code. Developers with different technological skills often concern differently and everyone has their own limitations. So a good and independent architecture can offer a mechanism that the programmers can enjoy the freedom of coding with simplified integration.

In a nutshell, with the development of IoT technology, traditional architecture can't meet the requirements of heterogeneous, interoperable, customizable and scalable systems. To deal with above challenges, we put forward an open IoT framework by decomposing IoT system into microservices to perform different kinds of tasks. By using message driven and registry/discovery mechanism in core service, the framework can easily extend, evolve and integrate third party applications to support interoperability and scalability. Moreover, the system uses device plugins to shield the differences of hardware facilities in order to support more heterogeneous platforms. In particular, as integrated with a series of microservices, the framework delivers strong mechanism for fusing heterogeneous data through hierarchical preprocessing mass sensors data.

1.2 IoT Capabilities

There are 2 billion PCs in use today across the globe. There are over 10 billion mobile phones. By 2020, it is predicted that there will be over 250 billion devices connected to the Internet. Some of these devices will be new products, but most will be existing things we use every day that will be enhanced with sensors, such as thermostats, cars, eyeglasses, wrist watches, clothing, street lamps, cars, buildings...you name it, it will likely become connected.

Each of these devices will be gathering data through sensors and sending data to the cloud. The amount of data that will be collected will be measured in petabytes, exabytes, and zettabytes. In other words, IoT is not just about devices but also about data, a lot of data. The reason that we want to collect all this data is to extract knowledge, to provide real-time visualization and data feeds, and to perform historical and predictive analytics that will drive business decisions at velocity and provide real-time notification and status.

To fully realize an IoT solution, several capabilities will be required. These capabilities include the following:

- **Device Management**: The device, upon initialization, will want to establish a relationship with the cloud environment, usually through its unique identifier, such as a serial number, so that the business is notified that the device is active. The business will also want the ability to send

commands to the device for the purposes of providing software updates or updating local data caches.

- **Telemetry Ingestion**: Devices may be sending multiple messages a second, and there may be hundreds to thousands of devices or more, which would result in 10's of thousands to possibly millions of messages a day. The cloud platform provides high-volume message ingestion using a single logical endpoint.
- **Transformation and Storage**: Once the messages arrive, the cloud provides a mechanism to select, transform, and route messages to various storage mediums for the purpose of archival and staging for downstream processing.
- **Status and Notifications**: The cloud solution will want to provide the ability to visualize the status of the message pool in real time through tabular or graphical UI components. In addition, some messages may contain information of an alert status so the IoT solution must provide a mechanism for real-time notifications.
- **Analytics and Data Visualization**: The value of collecting so much data in a continuous fashion is to build up an historical record for the purpose of performing analytics to glean business insight. Traditional data warehouse techniques or more modern map-reduce and predictive analytics mechanisms can be employed.

The chapters included in this thesis are organized in three parts, as can be seen in table

Organization of Chapters		
Chapters	Parts	Themes
Chapter 1	Introduction	Internet of Things
Chapter 2	Background	Monolithic Architecture Microservices Architecture
Chapter 3	Implementation	System Model
Chapter 4	Evaluation	Tests and Results
Chapter 5	Conclusion	Conclusion and Future Work

Table 1.1: Thesis Chapters

Chapter 2

Background

2.1 Monolithic Architecture

In the monolithic architecture software system is deployed as a single solution, in which functionally distinguishable aspects are all interwoven. The natural advantages of monolithic architecture are module independent, uniform standards and technology stack. Many researches offer solutions from perspective of monolithic architecture. Earlier IoT studies mainly focus on hardware network and low-level software technology. For instance, Wireless Sensor Networks (WSN) is a major approach of IoT, but it has limited capacity and weak scalablity. To address these issues, some researches like ubiSOAP WoT/SDN focus on high-level IoT application programming, which uses standard middleware to implement a layered communication. However, these systems are low reusability and portability. TinySOA , Servilla and a series of typical Service Oriented Architectures (SOA) have been applied in different applica-tions. To improve the scalablity of SOA, an Event-Driven SOA (EDSOA) IoT technology has also been put forward. In addition, OpenIoT expands the concept of SOA, and implements Web of Things (WoT).

However, with the increasing of the system functions, the IoT becomes more and more complex in the distributed environment. Monolithic architecture has some inevitable defects. First, the entire system is a united application; only multiple deployments can improve the system performance, while the overloaded functions create bottleneck, which is a waste of computing resources. Second, in the case of the change and evolution of the system, a change in a function may affect other functions due to high dependencies. This also brings complexity for re-deployment, maintenance and continuous integration. Finally, the whole system uses a sole technology stack and development standards, which in turn limits the methods to solve the problem of physical heterogeneity.

Consider an example of Ecommerce application, that authorizes customer, takes an order, check products inventory, authorize payment and ships ordered products. This application consists of several components including e-Store User interface for customers (Store web view) along with some backend

services to check products inventory, authorize and charge payments and shipping orders.

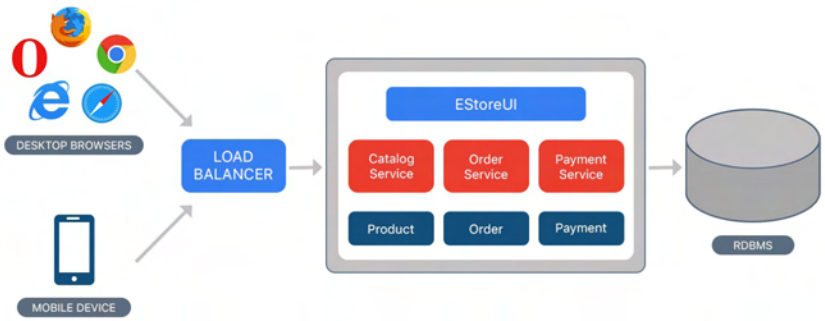


Figure 2.1. Monolithic Architecture (for E-commerce system)

Despite having different components/modules/services, the application is built and deployed as one Application for all platforms (i.e. desktop, mobile and tablet) using RDBMS as a data source. The benefits of this approach are:

- Simple to develop — At the beginning of a project it is much easier to go with Monolithic Architecture.
- Simple to test. For example, you can implement end-to-end testing by simply launching the application and testing the UI with Selenium.
- Simple to deploy. You have to copy the packaged application to a server.
- Simple to scale horizontally by running multiple copies behind a load balancer.

On the other hand, despite the above benefits, deploying an application with the structure of monolithic architecture, also has a lot of disadvantages and issues:

- Maintenance — If Application is too large and complex to understand entirely, it is challenging to make changes fast and correctly.
- The size of the application can slow down the start-up time.
- You must redeploy the entire application on each update.
- Monolithic applications can also be challenging to scale when different modules have conflicting resource requirements.
- Reliability — Bug in any module (e.g. memory leak) can potentially bring down the entire process. Moreover, since all instances of the application are identical, that bug impact the availability of the entire application.

- Regardless of how easy the initial stages may seem, Monolithic applications have difficulty to adopting new and advance technologies. Since changes in languages or frameworks affect an entire application, it requires efforts to thoroughly work with the app details, hence it is costly considering both time and efforts.

2.2 Microservice Architecture

2.2.1 What is microservice

The term microservice can be a bit misleading. The prefix "micro" implies that microservices are either tiny little entities that run around doing tasks on our behalf,like vacuuming the floor or fixing a flat tire, or that form a vast swarm of microscopic insect-like devices that self-replicate through the consumption of matter and energy, and are capable of disintegrating any substance they touch. Microservices do work on our behalf but they are not always tiny.

The "micro" in microservices is actually in reference to the scope of functionality that the service provides. A microservice provides a business or platform capability through a well-defined API, data contract, and configuration. It provides this function and only this function. It does one thing and it does it well. This simple concept provides the foundation for a framework that will guide the design, development, and deployment of your microservices.

Within the context of doing one thing and doing it well, microservices also exhibit a number of other properties and behaviors; it is these elements that differentiate microservices from previous incarnations of service-oriented approaches. These elements affect every aspect of how we develop software today, from team structure, source code organization, and control to continuous integration, packaging, and deployment.

2.2.2 Architecture

Microservices are an approach to application development in which a large application is built as a suite of modular services (i.e. loosely coupled modules/components). Each module supports a specific business goal and uses a simple, well-defined interface to communicate with other sets of services.

Instead of sharing a single database as in Monolithic application, each microservice has its own database. Having a database per service is essential if you want to benefit from microservices, because it ensures loose coupling. Each of the services has its own database. Moreover, a service can use a type of database that is best suited to its needs.

Many researchers begin to provide IoT solution using microservice architecture. For example, Vresk and Tomislav present a microservice based architecture focusing on connecting with heterogeneous devices, where the system confines to data model aspect. An approach of Krylovskiy discusses how to apply microservice architecture to design a Smart City IoT platform. Another approach presents three cloud microservices: contextual triggering microservice, visualization microservice, and anomaly detection and root cause microservice, to accelerate and facilitate the development of context and location

based applications. However, the above mentioned microservice IoT systems only consider a specific application. Therefore, it is necessary to design a more generic and open framework in IoT system.

To summarize, the architecture of microservices has plenty of benefits such as:

- Enables the continuous delivery and deployment of large, complex applications.
- Better testability — services are smaller and faster to test.
- Better deployability — services can be deployed independently.
- It enables you to organize the development effort around multiple teams. Each team is responsible for one or more single service. Each team can develop, deploy and scale their services independently of all of the other teams.
- Each microservice is relatively small
- Comfortable for a developer to understand.
- The IDE is faster making developers more productive.
- The application starts faster, which makes developers more productive, and speeds up deployments.
- Improved fault isolation. For example, if there is a memory leak in one service then only that service is affected. The other services continue to handle requests. In comparison, one misbehaving component of a monolithic architecture can bring down the entire system.
- Microservices Eliminates any long-term commitment to a technology stack. When developing a new service you can pick a new technology stack. Similarly, when making major changes to an existing service you can rewrite it using a new technology stack.

On the other hand using microservice architecture, also has some critical drawbacks, such as:

- Developers must deal with the additional complexity of creating a distributed system.
- Developer tools/IDEs are oriented on building monolithic applications and don't provide explicit support for developing distributed applications.
- Testing is more difficult as compared to Monolith applications.
- Developers must implement the inter-service communication mechanism.
- mplementing use cases that span multiple services without using distributed transactions is difficult.
- Deployment complexity. In production, there is also the operational complexity of deploying and managing a system comprised of many different service types.

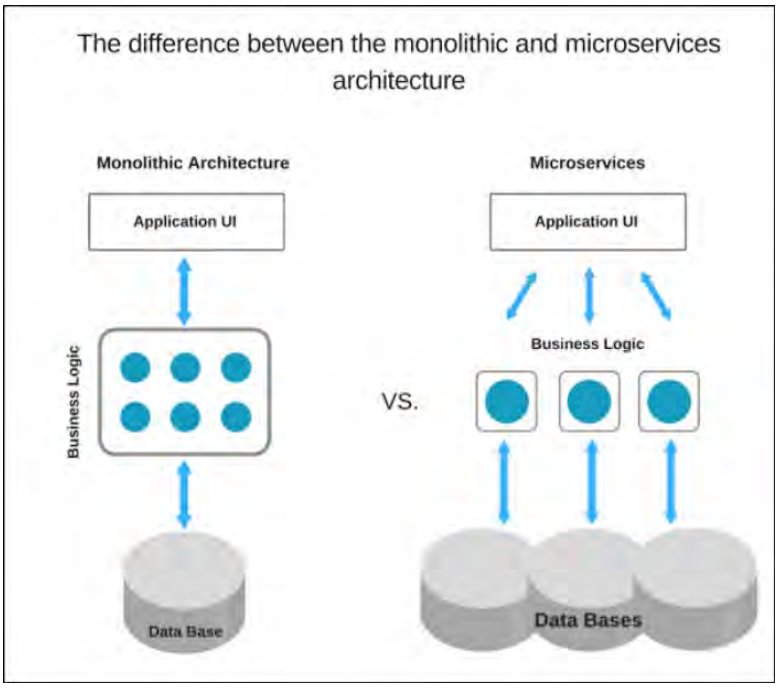


Figure 2.2. Difference between monolith and microservice architectures

Chapter 3

Implementation

In this chapter we are going to demonstrate our distributed web IoT platform for device monitoring and data analysis. We will describe and survey all parts of the application and analyze how each component works in the system.

3.1 System model

The design of a new generation of IoT framework considers the integration and reusability of existing information service system with high cohesion and loose coupling in open and scalable platform design. The core idea of the design is adopting the concept of microservice architecture, based on the analyse of the existing IoT system, reconstructing all the business functions of the system by decoupling them into independent and specific services.The design is using lightweight communication mechanism to interact between services with a minimal overload.

The Web IoT application is designed to handle large data traffic from the edge devices that are spread at the outdoor environment. Moreover, the system goal is to push all these data to a set of frontend applications (user application, admin application) managing to serve a large number of user requests at the same time. Figure 3.1 shows the big picture of our IoT system.

3.2 Edge Devices

Our platform is powered by data that being sent from edge devices. This set of devices consists of sensor and gateways nodes. Each sensor node consists of a set of sensors that measure and detect changes or events in the outdoor environment. Also, each sensor node is mapped to a gateway node. The between communication is achieved by a low power wireless protocol, either Zigbee or LoRa(Long Range), depending on the distance between them.

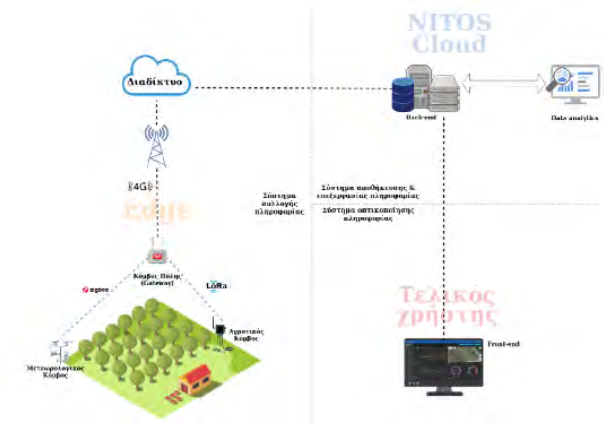


Figure 3.1. The Figure shows our IoT system model.

As we see in the Figure 3.2 the gateway node is responsible to collect all the data from the sensor nodes and forwards them to the cloud through 4g connection. In case of an error, the gateway still keeps the data in its memory and it will retry when the network become stable again.

Each sensor node works on low power consumption. They are scheduled to make a sampling every 30 minutes, and the rest of the time are in sleep mode. So, they only wake up when it is time to make a measurement from the environment. After sampling is finished, sensor nodes enter the transmission mode to send their data to gateways. When transmission process complete successfully, the nodes return again to sleep mode. On the other hand, gateway nodes must be all the time in full power mode. This is necessary because gateways should be ready at any time to catch data from sensor nodes and forward them to cloud.

Figure 3.2. The Figure shows the edge devices and how the can communicate with the cloud.

3.3 Cloud Services

3.3.1 Mqtt Broker

The IoT application is separated in two distinct parts, the Backend and the Frontend part. The Backend is also divided to a set of independent services, as we can see in Figure 3.4. At this point, we assume that data from sensor nodes, reached our web IoT application through the gateway. The service that is responsible to gather all the sensors data, that coming from the gateway, is an Mqtt broker. We use the VerneMq Mqtt broker, as it a distributed service that has high performance and support high number of concurrent devices. The broker service has its own memory and use it to buffer all that data coming from the field devices.

The communication between the gateways and the broker is done through the **MQTT** protocol. **MQTT** protocol it is very efficient and lightweight, allows messaging between device to cloud and cloud to device, it is scalable ,reliable and also offers security as it is easy to encrypt messages using TLS and authenticate clients using modern authentication protocols, such as OAuth. Thus, MQTT broker works under the publish/subscribe architecture as we can observe at the Figure 3.3.

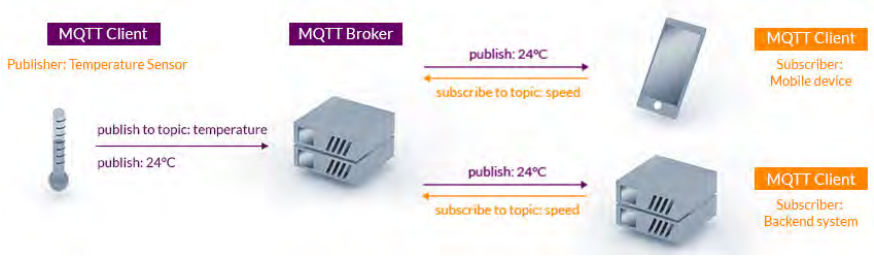


Figure 3.3. The Figure shows the MQTT broker architecture.

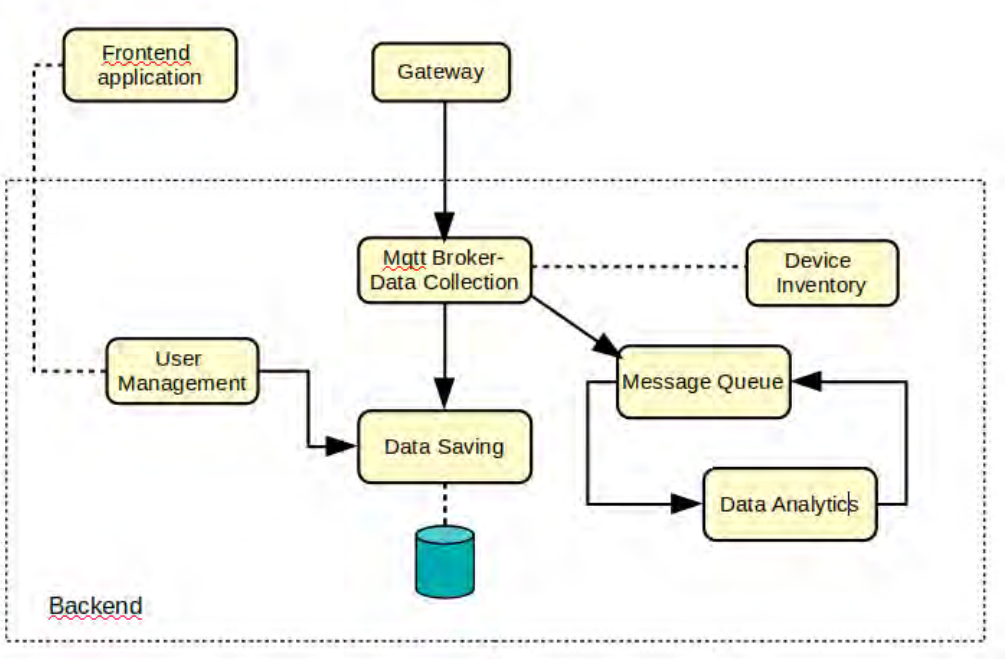


Figure 3.4. The Figure shows the services that run in the backend.

So, when a gateway node receives measurement values, publish each value to a specific Mqtt Broker's topic as we can see at the Table 3.2. Another service responsible for data management, is subscribed in every measurement topic. So, when a new value arrives from gateway, that the service listen to topics, grubs and saves that values to a timescale database, InfluxDB.

MQTT Broker	
Topics	Measurements
$Temperature$	25°C
$Humidity$	98%
$SolarRadiation$	2043K
$Leafwetness$	3%

Table 3.1: Topics in Mqtt Broker service.

3.3.2 Data Analytics

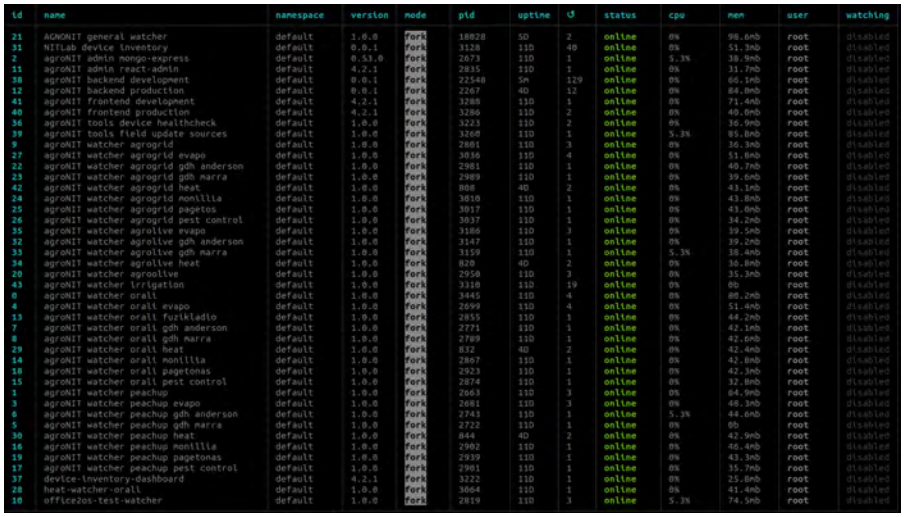
As we mentioned above, our sensor data are located now to our timescale database. Our web IoT application gathers data from sensor nodes and applies to them several algorithms to produce significant and crucial results. For that purpose, we built every algorithm as a standalone service, which fetch data from database and calculates the result. That services are written in the Javascript runtime engine of Nodejs and served via the daemon process manager PM2.

At Figure 3.5 we can see an instance of all standalone services that each one runs a specific algorithm and and calculates its result using the sensors data. Every time sensors send data to gateway and in turn the gateway forwards the data to application, each microservice algorithm, grabs data and recalculate the new algorithms results. Since the sensor nodes sends data every 30 minutes, the algorithms updating the results in 30 minutes too. In case of an error in one of these services, does not generate a conflict in the rest of them. So, these results saved in our database and a database API is responsible to provide them in the Frontend application when a user requests them.

3.3.3 Device Inventory

Our IoT application collects data from the edge devices and offers the end-users a raw data visualization but also a visualization of algorithm results from the data analytics microservices, as we described above. Therefore, the functionality of our application is based on this data. However, there are times when a problem may arise and data from a device does not reach our system. This could be due to either a possible bug in our application or a malfunction of the device or even worse the operation of this device has been interrupted.

So it is a very difficult task for the engineers to figure out where the fault lies, as it is two different parts of the system. The edge devices, sensor and gateway nodes, are placed at the outdoor environment, so there are many possible scenarios that may cause an interrupt of an edge device's functionality. For example, a sensor node's battery may be empty or there is a power supply problem in the area where



id	name	namespace	version	mode	pid	uptime	↺	status	cpu	mem	user	watching
21	AGNONIT general watcher	default	1.0.0	fork	18028	5D	2	online	0%	98.6mb	root	disabled
31	NITLab device inventory	default	0.0.1	fork	3128	11D	40	online	0%	51.3mb	root	disabled
2	agroNIT admin mongo-express	default	0.53.0	fork	2673	11D	1	online	5.3%	38.9mb	root	disabled
11	agroNIT admin react-admin	default	4.2.1	fork	2835	11D	1	online	0%	31.7mb	root	disabled
38	agroNIT backend development	default	0.0.1	fork	22548	5H	129	online	0%	66.1mb	root	disabled
12	agroNIT backend production	default	0.0.1	fork	2267	4D	12	online	0%	84.0mb	root	disabled
41	agroNIT frontend development	default	4.2.1	fork	3288	11D	1	online	0%	71.4mb	root	disabled
40	agroNIT frontend production	default	4.2.1	fork	3286	11D	2	online	0%	40.0mb	root	disabled
36	agroNIT tools device healthcheck	default	1.0.0	fork	3223	11D	2	online	0%	36.9mb	root	disabled
39	agroNIT tools field update sources	default	1.0.0	fork	3268	11D	1	online	5.3%	85.8mb	root	disabled
9	agroNIT watcher agrogrid	default	1.0.0	fork	2801	11D	3	online	0%	36.3mb	root	disabled
27	agroNIT watcher agrogrid evapo	default	1.0.0	fork	3036	11D	4	online	0%	51.8mb	root	disabled
22	agroNIT watcher agrogrid gdh anderson	default	1.0.0	fork	2981	11D	1	online	0%	40.7mb	root	disabled
23	agroNIT watcher agrogrid gdh marra	default	1.0.0	fork	2989	11D	1	online	0%	39.6mb	root	disabled
42	agroNIT watcher agrogrid heat	default	1.0.0	fork	888	4D	2	online	0%	43.1mb	root	disabled
24	agroNIT watcher agrogrid monillia	default	1.0.0	fork	3010	11D	1	online	0%	43.8mb	root	disabled
25	agroNIT watcher agrogrid pagetos	default	1.0.0	fork	3017	11D	1	online	0%	43.0mb	root	disabled
26	agroNIT watcher agrogrid pest control	default	1.0.0	fork	3037	11D	1	online	0%	34.2mb	root	disabled
35	agroNIT watcher agrolive evapo	default	1.0.0	fork	3186	11D	3	online	0%	39.5mb	root	disabled
32	agroNIT watcher agrolive gdh anderson	default	1.0.0	fork	3147	11D	1	online	0%	39.2mb	root	disabled
33	agroNIT watcher agrolive gdh marra	default	1.0.0	fork	3159	11D	1	online	5.3%	38.4mb	root	disabled
34	agroNIT watcher agrolive heat	default	1.0.0	fork	820	4D	2	online	0%	36.8mb	root	disabled
28	agroNIT watcher agroolive	default	1.0.0	fork	2950	11D	3	online	0%	35.3mb	root	disabled
43	agroNIT watcher irrigation	default	1.0.0	fork	3310	11D	19	online	0%	0b	root	disabled
8	agroNIT watcher orali	default	1.0.0	fork	3445	11D	4	online	0%	80.2mb	root	disabled
4	agroNIT watcher orali evapo	default	1.0.0	fork	2699	11D	4	online	0%	51.4mb	root	disabled
13	agroNIT watcher orali fuirkladio	default	1.0.0	fork	2835	11D	1	online	0%	44.2mb	root	disabled
7	agroNIT watcher orali gdh anderson	default	1.0.0	fork	2771	11D	1	online	0%	42.1mb	root	disabled
8	agroNIT watcher orali gdh marra	default	1.0.0	fork	2789	11D	1	online	0%	42.6mb	root	disabled
29	agroNIT watcher orali heat	default	1.0.0	fork	832	4D	2	online	0%	42.4mb	root	disabled
14	agroNIT watcher orali monillia	default	1.0.0	fork	2867	11D	1	online	0%	42.8mb	root	disabled
18	agroNIT watcher orali pagetonas	default	1.0.0	fork	2923	11D	1	online	0%	42.3mb	root	disabled
15	agroNIT watcher orali pest control	default	1.0.0	fork	2874	11D	1	online	0%	32.8mb	root	disabled
1	agroNIT watcher peachup	default	1.0.0	fork	2663	11D	3	online	0%	64.9mb	root	disabled
3	agroNIT watcher peachup evapo	default	1.0.0	fork	2681	11D	3	online	0%	48.3mb	root	disabled
6	agroNIT watcher peachup gdh anderson	default	1.0.0	fork	2743	11D	1	online	5.3%	44.6mb	root	disabled
5	agroNIT watcher peachup gdh marra	default	1.0.0	fork	2722	11D	1	online	0%	0b	root	disabled
30	agroNIT watcher peachup heat	default	1.0.0	fork	844	4D	2	online	0%	42.9mb	root	disabled
16	agroNIT watcher peachup monillia	default	1.0.0	fork	2902	11D	1	online	0%	46.4mb	root	disabled
19	agroNIT watcher peachup pagetonas	default	1.0.0	fork	2939	11D	1	online	0%	43.3mb	root	disabled
17	agroNIT watcher peachup pest control	default	1.0.0	fork	2901	11D	1	online	0%	15.7mb	root	disabled
37	device-inventory-dashboard	default	4.2.1	fork	3222	11D	1	online	0%	25.8mb	root	disabled
28	heat-watcher-orali	default	1.0.0	fork	3064	11D	1	online	0%	41.4mb	root	disabled
10	officeZos-test-watcher	default	1.0.0	fork	2819	11D	3	online	5.3%	74.5mb	root	disabled

Figure 3.5. The Figure shows the list of active microservices in the Web IoT platform.

it is located the gateway node, thus causing an energy device interrupt. Moreover, these devices are exposed to bad weather conditions and device hardware problems may conflict the device's operation.

Hence, it is very useful to know at each time the state of the devices. For that reason, we implemented a device inventory microservice. There are 4 possible states for each device:

- **online** state, in which the device works properly,
- **inactive** state depicts that a device is not working due to a functional error,
- **stopped**, in which the device is intentionally powered off from the system
- and **malfunctioning**, means that the device has stopped sending data for a period of time, but now is working again.

A device being in mulfunctioning state is for example when a sensor node's wireless connection drops due to bad weather conditions and our system lost its measurements during that period of time. But after that time if wireless signal restore, makes the data reception stable again.

Every sensor node transmits measurements every 30 minutes. Device inventory service check every one hour the last data transmission for each declared device in the database. If a device managed to send measurements the last one hour, its state updates to active. In case a device have not managed to send data the last hour, algorithm checks if there are any data from this device the last 3 days. In case we find data within this time period,the device's state update to mulfunctioning. On the other hand, if there are no data from that device the last 3 days, the service update its state to inactive. Finally, if we have deliberately decided to shutdown a device for any reason, we must update its state to stopped.

Moreover, an edge device incorporates in the measurements packet information related to its battery status and its gps location. Hence, the device inventory microservice collects all that data and informs the system with all these information about a device, offering the engineers a very useful tool for remote monitoring edge devices. Also, our microservice keeps track of when a device last send a data packet, helping engineers draw some conclusions about their device deployments.

Device Inventory States				
Devices	States	Coordinates	Remaining Energy	Last Received Packet
$SensorNode1$	Active	(23,4983, 32,4098)	92%	01/09/2020
$GatewayNode1$	Stopped	(26,4983, 34,6778)	70%	23/12/2019
$SensorNode2$	Inactive	(21,42933, 35,0098)	20%	05/03/2020
$GatewayNode2$	Mulfunctioning	(27,3403, 38,9998)	85%	30/08/2020

Table 3.2: Device Inventory microservice.

3.3.4 Web Application

The final part of our system is the frontend which consists of two different applications, each of which addressed to a different group of users. The two groups are the the simple users and the admin users. For that purpose we developed a user management service in order to redirect a each user to the proper web application,depending on the group that belongs to (Figure 3.6).

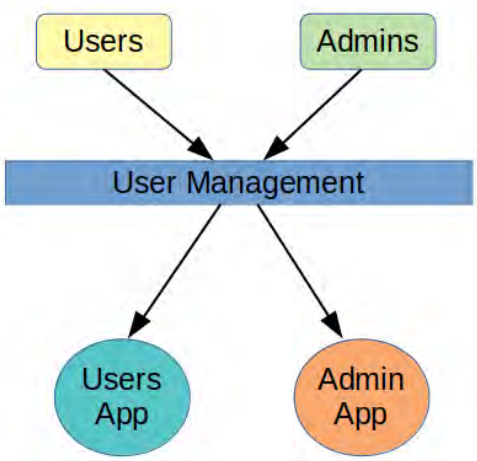


Figure 3.6. The Figure shows user management service.

The first application is addressed to simple users, offers them the ability to monitor their data and also provide them a set of notifications and consultings that our system generates from our data analytics microservices. In Figure 3.7 you can see an instance of the simple user monitoring application.

The second is for admin users also for monitoring data but also devices and users. Admin application offers the ability to set devices into fields and users. Thus, a simple user can see only his device data

Figure 3.7. The Figure shows the frontend application for simple users.

from his field. Furthermore, we can monitor the devices functionality and the devices location inside a map view as we can see in Figure 3.8.

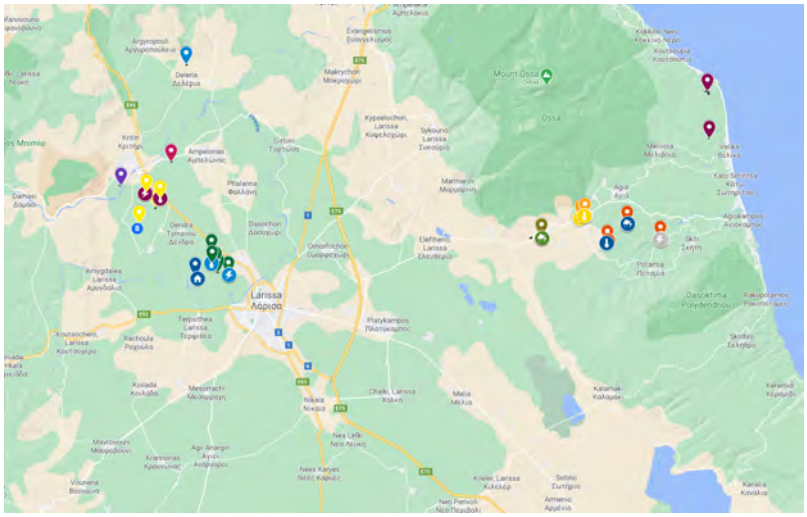


Figure 3.8. The Figure shows the device monitorin and location in map view.

Chapter 4

Evalutation

In this chapter we will evaluate our decision to build the Web IoT platform into separate standalone microservices that communicates to each other.

4.1 Tests and Results

4.1.1 Development methodology and efforts

The development of applications using microservice architectures requires a change in the way that most companies and software vendors have traditionally been developing monolithic applications using single codebases. Microservice architectures allow small teams to work on small applications(microservices) without worrying about how other microservices or teams work. Every team can use different technologies to implement microservices/gateways according to the business and technical requirements; to avoid the use of a lot of technologies that can be difficult to be managed some guidelines should be offered to all teams so they can decide which set of technologies to use in each microservice/gateway. The design, documentation and publication of REST API is very important to allow that services published by microservices can be easily consumed by gateways, and services published by gateways can be easily consumed by end-user applications (browsers, mobile apps, APIs, etc.).

In order to test our implementation, we used Apache Jmeter. Apache JMeter is an Apache project that can be used as a load testing tool for analyzing and measuring the performance of a variety of services, with a focus on web applications. We start our application and we run several tests with a varied number of users making the same requests in a specific time.

In Figure 4.1 we can see how our application responds and serves a total number of 50 HTTP requests in 2 seconds. In this test we simulate 10 users that everyone makes the same 5 HTTP requests. The max response time reach Ĩ.1 sec and the min response time is 300 ms. Also we can see that after

the max upper bound the response time decreases as a result of caching. For a more realistic depiction of how our system works we increase the number of users to 100 but kept the same request per user.

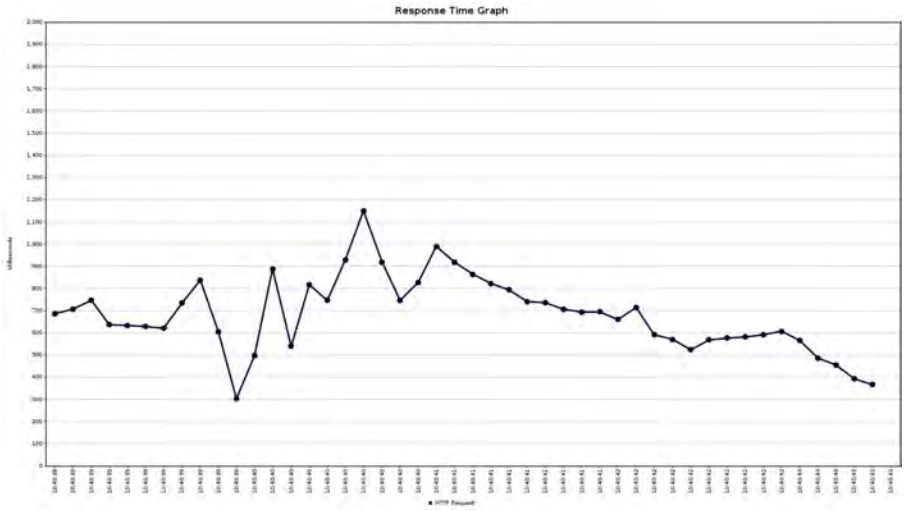


Figure 4.1. The Figure shows a performance test with 10 users making 5 HTTP requests in a 2 seconds time.

In Figure 4.2 we can see the performance of our application and response time. The max response time reached the $\tilde{7}.5$ sec but the average response time is about $\tilde{5}$ seconds. Compared with the first test in which the average response time was 600 ms, here we observe a large increase in response time which is very normal as our system serves 500 total requests in 2 seconds.

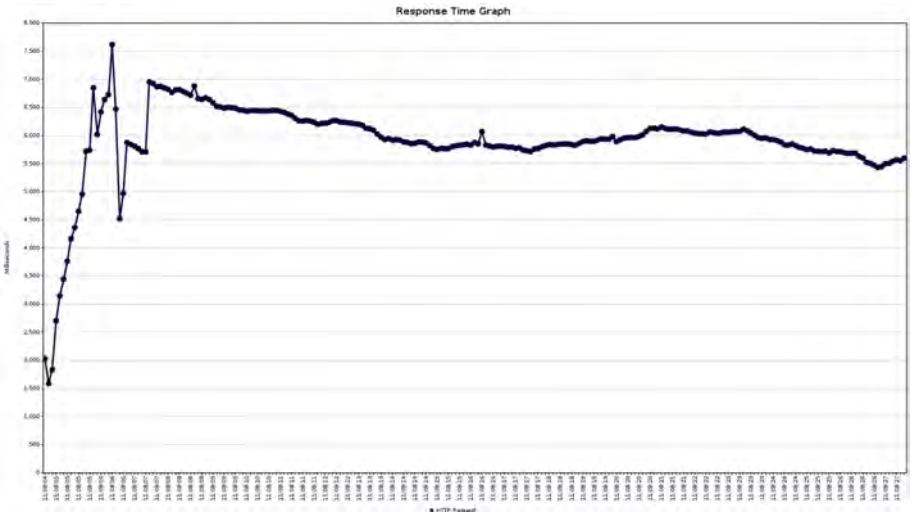


Figure 4.2. The Figure shows a performance test with 100 users making 5 HTTP requests in a 2 seconds time.

In the last test we decided to increase our total number of requests to 1000 HTTP requests. We kept

the serve time to 2 seconds as in the above tests. In Figure 4.3 we have 100 users where each one makes 10 same requests. It can be seen that the max response time is 9̃.5 seconds, the min response time is 7̃00 ms and the average is 6̃.5 seconds.

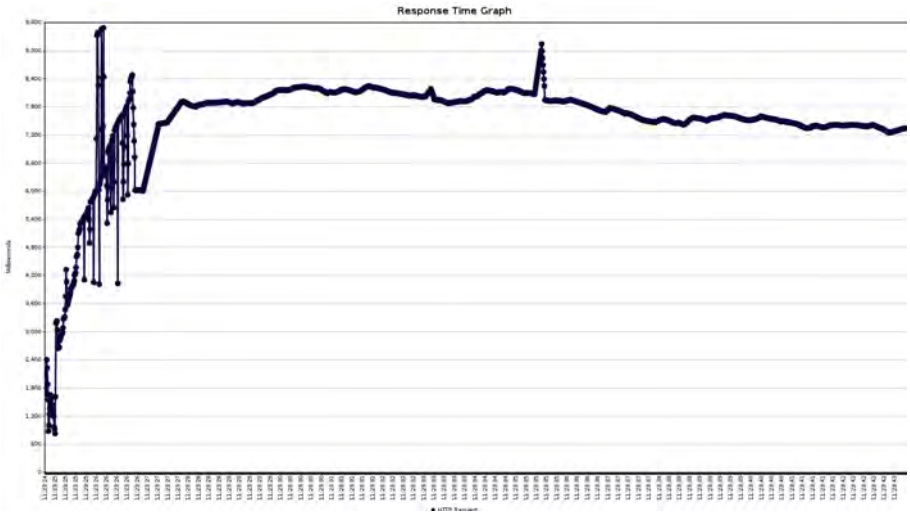


Figure 4.3. The Figure shows a performance test with 100 users making 10 HTTP requests in a 2 seconds time.

Doing these stress tests we observe that using microservice architecture in our system, makes the system more stable and it manages to serve all the requests without losing any data in a very short period of time.

Chapter 5

Conclusion

5.1 Summary

As we mentioned in the introduction the microservices architecture has a lot of benefits and todays it is a ”de facto” choice for building web applications, especially in the sector of IoT in which the system has to manage a large number of data and users.

The contribution of this dissertation is to provide a distributed web IoT platform for IoT deployments with an efficient, scalable and more productive design. Each microservice is responsible for a specific process giving the engineers the ability to make better debugs, better response times and making the application more robust as if one microservice is down the rest system still runs.