ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA
Campus di Cesena

Scuola di Ingegneria e Architettura
Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

Design and Deployment of an Execution Platform based on Microservices for Aggregate Computing in the Cloud

Tesi in
Ingegneria dei Sistemi Software Adattativi Complessi

Relatore:
Prof. MIRKO VIROLI

Presentata da:
THOMAS FARNETI

Corelatore:
Dott. ROBERTO CASADEI

Anno Accademico 2016-2017
Sessione III

Contents

Abstract (italiano)

Il termine *Internet of Things* viene spesso utilizzato per definire oggetti intelligenti, servizi, and applicazioni connessi attraverso Internet. Uno studio redatto da Cisco afferma che la crescita del numero e della varietà di devices da cui collezionare dati è estremamente rapida. Aumentando il numero di devices aumenta conseguentemente anche la complessità, quindi, ci si trova ad affrontare problemi tra i quali: mancanza di modularità e riusabilità, difficoltà nelle fasi di test, manutenzione e rilascio.

La *Programmazione Aggregata* fornisce un'alternativa ai metodi di sviluppo software tradizionali, che semplifica drammaticamente progettazione, creazione, e manutenzione di sistemi IoT complessi. Con questa tecnica, l'unità base di computazione non è più un singolo device ma una collezione cooperativa di devices.

Questa tesi descrive la progettazione e sviluppo di una Piattaforma per Programmazione Aggregata basata su microservizi nel Cloud.

A differenza del modello distribuito della Programmazione Aggregata, il Cloud Computing rappresenta un'ulteriore opportunità per la costruzione di sistemi scalabili e può essere pensato come una strategia alternativa di esecuzione dove le computazioni sono per l'appunto eseguite su Cloud.

Per poter ottenere il massimo dalle tipiche caratteristiche di scalabilità ed affidabilità fornite dal modello Cloud occorre adottare un'architettura adeguata. Questo lavoro descrive come poter servirsi dell'architettura a microservizi costruendo l'infrastruttura richiesta per la comunicazione tra processi dalle fondamenta.

Data la maggiore complessità tecnologica delle architetture a microservizi, l'elaborato descrive come adottare un approccio a "container" alleviando le difficoltà di gestione attraverso un container orchestrator.

Keywords – *aggregate programming,microservices,cloud computing, continuous delivery, docker*

Abstract

The *Internet of Things* has evolved as an umbrella term for smart objects, services, and applications connected through the Internet. A study by Cisco says that the growth of the number and variety of devices that are collecting data is incredibly rapid. Increasing the number of Devices is also increasing the complexity and we have to face many troubles such as: design problems, lack of modularity and reusability, deployment difficulties, and test and maintenance issues.

Aggregate programming provides an alternative to traditional software development methods, that dramatically simplifies the design, creation, and maintenance of complex IoT software systems. With this technique, the basic unit of computing is no longer a single device but instead a cooperating collection of devices.

The work described in this dissertation consists in the design and deployment of an Execution Platform based on microservices for Aggregate Computing in the Cloud.

Despite the distributed model of Aggregate Programming, Cloud Computing represents a further opportunity for building scalable systems and could be thought as an alternate execution strategy where computations are carried out in the Cloud.

In order to take advantage of scalability and reliability provided by the Cloud Model a proper architecture is needed. This work describes how to leverage the Microservices Architecture Pattern, building the required infrastructure for Inter Process Communication from the ground up.

It is known that deploying Microservices could be very difficult. Here we describe how to adopt a containerized approach alleviating management of containers with a mainstream orchestration solution.

Keywords – *aggregate programming,microservices,cloud computing, continuous delivery, docker*

Introduction

The *Internet of Things* has evolved as an umbrella term for smart objects, services, and applications connected through the Internet. Such smart things can also collaborate with other physical and virtual resources available in the Web, thus providing value-added information and functionalities for end-users and/or applications [1].

A study by Cisco [2] says that the growth of the number and variety of devices that are collecting data is incredibly rapid. It estimates that the number of Internet-connected devices overtook the human population in 2010, and that there will be 50 billion Internet-connected devices by 2020. The key for success resides in the interaction between such devices. Complexity has moved from the single devices to the architecture that enables and supports such connectivity.

As the number grows a question arises: Are traditional software development methods ready to support such complex and large-scale interactions in an open and ever-changing environment? Traditionally, we see the individual device as the basic unit of computation. This has influenced many aspects of the design of distributed systems. But as soon as complexity grows, we have to face many issues such as: lack of modularity and reusability, and test and maintenance deployment difficulties.

Aggregate programming provides an alternative that dramatically simplifies the design, creation, and maintenance of complex IoT software systems. With this technique, the basic unit of computing is no longer a single device but instead a cooperating collection of devices: details of their behavior, position, and number are largely abstracted away, replaced with a space-filling computational environment [3].

The work described in this dissertation consists in the design and deployment of an Execution Platform based on microservices for Aggregate Computing in the Cloud.

Cloud computing is a well established model and technology supporting scalability and elasticity through on-demand provisioning of IT resources which are typically visualized. Since it represents a further opportunity for building scalable systems, it is reasonable to think of an alternate execution strategy for aggregate systems where computations are carried out in the cloud [4].

The platform is meant to support large-scale Aggregate Computing Environments, so, a proper architecture is needed in order to take advantage of scalability and reliability provided by the Cloud Model.

The microservices architecture pattern is designed to address these needs offering a way of scaling the infrastructure both horizontally and vertically. The approach is to develop a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms. These services are built around business capabilities and can be independently deployed and scaled based on resource requirements.

Deployment can be really difficult if not managed properly. Mainstream applications relies on Continuous Delivery pipeline that creates Docker Images of these services and deploys them in a Cloud Cluster Environment with the help of an orchestrator.

This dissertation first presents in Chapter 1 and 2 a background about IoT Architectures and microservices design and deployment. Chapter 3 describes the aggregate computing domain. Platform design begins at Chapter 4, which presents a list of minimum requirements with a proper analysis that illustrates the problem, and what has to be done. Then, Chapter 5 describes the key elements of the design of the platform, with a special attention for microservices design. A detailed view of the Implementation is provided in Chapter 6. Since the platform is distributed and runs on Cloud a detailed view of deployment is described in Chapter 7. Finally the evaluation of the platform is carried out in Chapter 8. Conclusive remarks and future perspectives are presented in Chapter 9.

Chapter 1

IoT Reference Architectures

Today, "Internet of Things" (IoT)[1] is used as an umbrella term by many sources. This expression encompasses many solutions somehow related to the world of intercommunicating and smart objects. These solutions show little or no interoperability capabilities as usually they are developed for a specific domain, following specific requirements. Furthermore, as the IoT covers different application fields, development cycles and technologies used vary enormously, thus implementing vertical solutions.

On the long run this situation is unsustainable. As in the networking field, where several solutions emerged to leave place to a common model, the emergence of a common reference model for the IoT domain and the identification of reference architectures can lead to a faster, more focused development and an exponential increase of IoT-related solutions.[5] These solutions can provide a strategic advantage to mature economies, as new business models can leverage those technological solutions providing room for economic development.

Considering only the technical point of view, the existing solutions do not address the *scalability* requirements of a future IoT, both in terms of communication between and the manageability of devices. Additionally, as the IoT domain comprises several different governance models, which are often incompatible.

In the vision of IoT-A[2] of the Internet of Things, the interoperability of solu-

[1]https://en.wikipedia.org/wiki/Internet_of_things
[2]http://www.iot-a.eu/public

tions at the communication level, as well as at the service level, has to be ensured across various platforms. This motivates, first, the creation of a **Reference Model** for the IoT domain in order to promote a common understanding.

Second, businesses that want to create their own compliant IoT solutions should be supported by a Reference Architecture that describes essential building blocks as well as design choices to deal with conflicting requirements regarding functionality, performance, deployment and security. Interfaces should be standardized, best practices in terms of functionality and information usage need to be provided. This chapter will first analyze what are the requirements for a Reference Architecture. Then it will explore the bases of the IoT-A Project that will lead to a concrete Architecture.

References for this chapter are mainly taken from [5]. The concrete reference architecture is based on [6].

1.1 The IoT-A Project

The IoT ARM proposal is based on the establishment of an architectural reference model **ARM** encompassing a reference architecture and a set of key features to construct IoT concrete architectures.

The *IoT Reference Model* provides the highest abstraction level for the definition of the IoT-A Architectural Reference Model. It promotes a common understanding of the IoT domain. The description of the IoT Reference Model includes:

- IoT Domain Model as a top-level description
- IoT Information Model explaining how IoT information is going to be modeled
- IoT Communication Model in order to understand specifics about communication between many heterogeneous IoT devices and the Internet as a whole.

The *IoT Reference Architecture* is the reference for building compliant IoT architectures. As such, it provides views and perspectives on different architectural aspects that are of concern to stakeholders of the IoT.The creation of the IoT Reference Architecture focuses on abstract sets of mechanisms rather than concrete application architectures.

To organizations, an important aspect is the compliance of their technologies with standards and best practices, so that interoperability across organizations is ensured. If such compliance is given, an ecosystem forms, in which every stakeholder can create new businesses that "inter-operate" with already existing businesses. The IoT-A ARM provides best practices to the organizations so that they can create compliant IoT architectures in different application domains. Those IoT architectures are instances from the Reference Architectures with some architectural choices like considering strong real-time or choosing strong security features, etc. They form thus special "flavors" of the IoT Reference Architecture. Where application domains are overlapping, the compliance to the IoT Reference Architecture ensures the interoperability of solutions and allows the formation of new synergies across those domains.

Reference architecture provides high-level architectural views and relevant perspectives for constructing IoT systems. Such architectural views offer descriptions that allow viewing an architecture under different angles and can be used when designing and implementing a concrete architecture. In turn, the perspectives represent set of tasks, tactics, directives, and architectural decisions for ensuring that a given concrete system accomplishes one or more quality attributes shared by one or more architectural views.

Functional View. The Functional View describes nine functionality groups, each one with one or more functional components. Despite this view specifies the basic functional components, it does not specifies the interactions among such components as they typically depend on design decisions that are not made at this level of abstraction, but when developing the concrete architecture.

Information View. One of the main purposes of connected smart objects in IoT is the exchange of information among each other and also with external systems. Therefore, this view is concerned about how representing relevant information in an IoT system in terms of static information structures. Moreover, it describes the components that handle information, the dynamic information flows through the system, and the life cycle of information within the system. The main element of this view is the virtual entity, which models a physical element of interest in the system.

Deployment and Operation View. Connected and smart objects in IoT can be realized in many different ways and can communicate by using many different technologies. In addition, different systems need to communicate with each other in a compliant way. To tackle such issues, this view aims to address how an IoT system can be realized by selecting the proper technologies and making them to communicate and operate, as well as to offer a set of guidelines to drive developers/architects through the different design decisions that they have to face in the system development. Three main elements are encompassed by this view, namely devices, resources, and services. Nonetheless, a complete analysis of all technological possibilities and their combinations goes beyond the scope of this view.

Besides thees views, the IoT ARM offers four perspectives regarded as the most important ones for IoT systems, namely:

- Evolution and Interoperability;
- Availability and Resilience;
- Reliability, Security, and Privacy; and
- Performance and Scalability.

Each of these perspectives has more or less impact over each view and contains:

- the desired quality addressed by the perspective;
- requirements relevant for IoT and related to the perspective;
- applicability of the perspective to (types of) IoT systems;
- activities suggested to achieve the desired qualities;
- architectural tactics related to the perspective and that can be used by an architect when designing the system.

This set of perspectives seems to be relevant for system developers as several quality parameters have to be taken into account, even more for the IoT domain. Moreover, these perspectives are intended to provide a framework for reusing knowledge and fostering the application of a systematic approach to ensure that a given system fulfills the required quality attributes.

1.2 The WSO2 Reference Architecture

In order to achieve standardization it is necessary to create high level reference architectures like the one defined by IoT-A. However, high-level reference architectures are difficult to understand because they are very abstract.

The WSO2 company has proposed a more concrete reference architecture based on its expertise in the development of IoT solutions. Such a reference architecture encompasses devices and both server-side and cloud architectures required to interact with and manage these devices. The main goal is to provide architects and developers with an effective starting point that is able to cover most of the requirements of IoT systems and development projects. However, it does not focus on detailing how a particular client-server, hardware or cloud architecture should work as the reference architecture is independent on specific providers and it is not bound to a specific set of technologies.

References for this section are taken from [6][5] and [7].

The WSO2 company has proposed a reference architecture based on its expertise in the development of IoT solutions [6]. Such a reference architecture encompasses devices and both server-side and cloud architectures required to interact with and manage these devices. The main goal is to provide architects and developers with an effective starting point that is able to cover most of the requirements of IoT systems and development projects. However, it does not focus on detailing how a particular client-server, hardware or cloud architecture should work as the reference architecture is independent on specific providers and it is not bound to a specific set of technologies.

This architecture consists of a set of five layers, each one performing a well-defined functionality:

- Device Layer, in which each device should have a unique identifier and direct or indirect communication with the Internet;
- Communications Layer, which supports device connec- tivity, with multiple potential protocols;
- Aggregation/Bus Layer, which supports, aggregates, and combines commu-

nications from several devices, as well as bridges and transforms data among different protocols;

- Event Processing and Analytics Layer, which processes and reacts upon events coming from the Aggregation/Bus Layer, as well as can perform data storage; and
- External Communications Layer, through which users can interact with devices and access data available at the system.

The WSO2's reference architecture also provides two additional traversal layers, namely:

- Device Management Layer, which communicates with devices through different protocols and allows remotely managing them
- Identity and Access Management, which is responsible for access control and security directives.

It is important to highlight that each of the layers proposed in the WSO2's reference architecture can be instantiated by using specific technologies that better suit the IoT system under construction.

1.2.1 Device Layer

The bottom layer of the architecture is the device layer. Devices can be of various types, but in order to be considered as IoT devices, they must have some communications that either indirectly or directly attaches to the Internet. Examples of direct connections are

- Arduino with Arduino Ethernet connection
- Arduino Yun with a Wi-Fi connection
- Raspberry Pi connected via Ethernet or Wi-Fi
- Intel Galileo connected via Ethernet or Wi-Fi

Examples of indirectly connected device include

- ZigBee devices connected via a ZigBee gateway

- Bluetooth or Bluetooth Low Energy devices connecting via a mobile phone
- Devices communicating via low power radios to a Raspberry Pi

Each device typically needs an identity. The identity may be one of the following:

- A unique identi er (UUID) burnt into the device (typically part of the System-on-Chip, or provided by a secondary chip)
- A UUID provided by the radio subsystem (e.g. Bluetooth identifier, Wi-Fi MAC address)
- An OAuth2 Refresh/Bearer Token (this may be in addition to one of the above)
- An identifier stored in nonvolatile memory such as EEPROM

For the reference architecture it's recommended that every device has a UUID (preferably an unchangeable ID provided by the core hardware) as well as an OAuth2 Refresh and Bearer token stored in EEPROM. The specification is based on HTTP; however the reference architecture also supports these flows over MQTT.

1.2.2 Communications Layer

The communication layer supports the connectivity of the devices. There are multiple potential protocols for communication between the devices and the cloud. The most well-known three potential protocols are

- HTTP/HTTPS (and RESTful approaches on those)
- MQTT 3.1/3.1.1
- Constrained application protocol (CoAP)

HTTP is well known, and there are many libraries that support it. Because it is a simple text-based protocol, many small devices such as 8-bit controllers can only partially support the protocol. The larger 32-bit based devices can utilize full HTTP client libraries that properly implement the whole protocol.

There are several protocols optimized for IoT use. The two best known are MQTT[3] and CoAP[4]. MQTT is a publish-subscribe messaging system based on a broker model. The protocol has a very small overhead, and was designed to support lossy and intermittently connected networks. MQTT was designed to flow over TCP. In addition there is an associated specification designed for ZigBee-style networks called MQTT-SN (Sensor Networks).

CoAP is a protocol from the IETF that is designed to provide a RESTful application protocol modeled on HTTP semantics, but with a much smaller footprint and a binary rather than a text-based approach. CoAP is a more traditional client-server approach rather than a brokered approach. CoAP is designed to be used over UDP.

For a reference architecture MQTT could be chose as the preferred device communication protocol, with HTTP as an alternative option. The reasons to select MQTT and not CoAP are:

- Better adoption and wider library support for MQTT;
- Simplified bridging into existing event collection and event processing systems
- Simpler connectivity over firewalls and NAT networks

However, both protocols have specific strengths (and weaknesses) and so there will be some situations where CoAP may be preferable and could be swapped in. In order to support MQTT an MQTT broker is needed in the architecture as well as device libraries. One important aspect with IoT devices is not just for the device to send data to the cloud/server, but also the reverse. This is one of the benefits of the MQTT specification: because it is a brokered model, clients connect an outbound connection to the broker, whether or not the device is acting as a publisher or subscriber. This usually avoids firewall problems because this approach works even behind firewalls or via NAT. In the case where the main communication is based on HTTP, the traditional approach for sending data to the device would be to use HTTP Polling. This is very inefficient and costly, both

[3] http://mqtt.org/
[4] http://coap.technology/

in terms of network traffic as well as power requirements. The modern replacement for this is the WebSocket[5] protocol that allows an HTTP connection to be upgraded into a full two-way connection. This then acts as a socket channel (similar to a pure TCP channel) between the server and client. Once that has been established, it is up to the system to choose an ongoing protocol to tunnel over the connection.

For the reference architecture once again it's recommended to use MQTT as a protocol with WebSockets. In some cases, MQTT over WebSockets will be the only protocol. This is because it is even more firewall-friendly than the base MQTT specification as well as supporting pure browser/JavaScript clients using the same protocol.

While there is some support for WebSockets on small controllers, such as Arduino, the combination of network code, HTTP and WebSockets would utilize most of the available code space on a typical Arduino 8-bit device. Therefore, it's recommended the use of WebSockets on the larger 32-bit devices.

1.2.3 Aggregation - Bus Layer

An important layer of the architecture is the layer that aggregates and brokers communications. This is an important layer for three reasons:

1. The ability to support an HTTP server and/or an MQTT broker to talk to the devices;
2. The ability to aggregate and combine communications from different devices and to
3. route communications to a specific device (possibly via a gateway)
4. The ability to bridge and transform between different protocols, e.g. to offer HTTP-based APIs that are mediated into an MQTT message going to the device.

The aggregation/bus layer provides these capabilities as well as adapting into legacy protocols. The bus layer may also provide some simple correlation and mapping from different correlation models.

[5]https://tools.ietf.org/html/rfc6455

Finally the aggregation/bus layer needs to perform two key security roles. It must be able to act as an OAuth2 Resource Server (validating Bearer Tokens and associated resource access scopes). It must also be able to act as a policy enforcement point (PEP) for policy-based access. In this model, the bus makes requests to the identity and access management layer to validate access requests. The identity and access management layer acts as a policy decision point (PDP) in this process. The bus layer then implements the results of these calls to the PDP to either allow or disallow resource access.

1.2.4 Event Processing and Analytics Layer

This layer takes the events from the bus and provides the ability to process and act upon these events. A core capability here is the requirement to store the data into a database. This may happen in three forms. The traditional model here would be to write a server- ìside application, e.g. this could be a JAX-RS application backed by a database. However, there are many approaches where we can support more agile approaches. The first of these is to use a big data analytics platform. This is a cloud-scalable platform that supports technologies such as Apache Hadoop to provide highly scalable MapReduce analytics on the data coming from the devices. The second approach is to support complex event processing to initiate near real-time activities and actions based on data from the devices and from the rest of the system. Our recommended approach in this space is to use the following approaches:

- Highly scalable, column-based data storage for storing events
- Map-reduce for long-running batch-oriented processing of data
- Complex event processing for fast in-memory processing and near real-time reaction and autonomic actions based on the data and activity of devices and other systems
- In addition, this layer may support traditional application processing platforms, such as Java Beans, JAX-RS logic, message-driven beans, or alternatives, such as node.js, PHP, Ruby or Python.

1.2.5 Client Communications Layer

The reference architecture needs to provide a way for these devices to communicate outside of the device-oriented system. This includes three main approaches. Firstly, we need the ability to create web-based front-ends and portals that interact with devices and with the event-processing layer. Secondly, we need the ability to create dashboards that offer views into analytics and event processing. Finally, we need to be able to interact with systems outside this network using machine-to-machine communications (APIs). These APIs need to be managed and controlled and this happens in an API management system.

The recommended approach to building the web front end is to utilize a modular front-end architecture, such as a portal, which allows simple fast composition of useful user interfaces. Of course the architecture also supports existing Web server-side technology, such as Java Servlets/JSP, PHP, Python, Ruby, etc. Our recommended approach is based on the Java framework and the most popular Java-based web server, Apache Tomcat. The dashboard is a re-usable system focused on creating graphs and other visualizations of data coming from the devices and the event processing layer. The API management layer provides three main functions:

- The first is that it provides a developer-focused portal (as opposed to the user- focused portal previously mentioned), where developers can find, explore, and subscribe to APIs from the system. There is also support for publishers to create, version, and manage the available and published APIs;
- The second is a gateway that manages access to the APIs, performing access control checks (for external requests) as well as throttling usage based on policies. It also performs routing and load-balancing;
- The final aspect is that the gateway publishes data into the analytics layer where it is stored as well as processed to provide insights into how the APIs are used.

1.2.6 Device Management

Device management (DM) is handled by two components. A server-side system (the device manager) communicates with devices via various protocols and provides both individual and bulk control of devices. It also remotely manages software and applications deployed on the device. It can lock and/or wipe the device if necessary. The device manager works in conjunction with the device management agents. There are multiple different agents for different platforms and device types.

The device manager also needs to maintain the list of device identities and map these into owners. It must also work with the identity and access management layer to manage access controls over devices.

There are three levels of device: non-managed, semi-managed and fully managed (NM, SM, FM). Fully managed devices are those that run a full DM agent. A full DM agent supports:

- Managing the software on the device
- Enabling/disabling features of the device (e.g. camera, hardware, etc.) • Management of security controls and identifiers
- Monitoring the availability of the device
- Maintaining a record of the device's location if available
- Locking or wiping the device remotely if the device is compromised, etc.

Non-managed devices can communicate with the rest of the network, but have no agent involved. These may include 8-bit devices where the constraints are too small to support the agent. The device manager may still maintain information on the availability and location of the device if this is available.

Semi-managed devices are those that implement some parts of the DM (e.g. feature control, but not software management).

1.2.7 Identity and Access Management

The final layer is the identity and access management layer. This layer needs to provide the following services:

- OAuth2 token issuing and validation
- Other identity services including SAML2 SSO and OpenID Connect support for identifying inbound requests from the Web layer
- XACML PDP
- Directory of users (e.g. LDAP)
- Policy management for access control (policy control point)

The identity layer may of course have other requirements specific to the other identity and access management for a given instantiation of the reference architecture.

Chapter 2

Microservices Design and Deployment

Microservices are currently getting a lot of attention and they are rapidly heading towards the peak of inflated expectations on the Gartner Hype cycle. At the same time, there are skeptics in the software community who dismiss microservices as nothing new. However, despite both the hype and the skepticism, the Microservices Architecture pattern has significant benefits – especially when it comes to enabling the agile development and delivery of complex enterprise applications. This chapter will give a brief background on how to design and implement an application that leverage the microservices architecture pattern. References are taken from [8] and [9]

2.1 Monolithic Architecture

Monolithic application has single code base with multiple modules. Modules are divided as either for business features or technical features. It has single build system which build entire application and/or dependency. It also has single executable or deployable binary. For example, Java applications packaged as WAR files and deployed on application servers such as Tomcat or Jetty or other Java applications packaged as self-contained executable JARs. Applications written in this style are extremely common because they are simple to develop since IDEs and tools are focused on building a single application. These applications are also simple to test. The Deployment process is also simpler because we can just copy the whole package to the server. we can scale the application by running multiple copies behind a load balancer.

This approach is simple but applications grows over time eventually becoming huge and after a few years, it will grow into a monstrous monolith. Despite simplicity there are huge limitations:

- Agile development and delivery attempts will flounder because of complexity. As a result, fixing bugs and implementing new features correctly becomes difficult and time consuming tending to be a downwards spiral. If the codebase is difficult to understand, then changes won't be made correctly.
- The larger the application, the longer the start-up time so development will slow down. If developers regularly restart the application server, then a large part of their time will be spent waiting around and productivity will suffer.
- Continuous deployment is extremely difficult to do with a complex monolith, since the entire application must be redeployed in order to update any part of it.
- Scaling can also be difficult when different modules have conflicting resource requirements. For example, CPU-intensive processing module would be deployed in Compute Optimized instances while in-memory database module is best suited for Memory-optimized instances. Because these modules are deployed together it's necessary to make a compromise on the choice of hardware
- Reliability is also a problem because all modules are running within the same process and a bug in any module, can potentially bring down the entire process. Moreover, since all instances are identical, that bug will impact the availability of the entire application.
- There is a huge barrier when adopting new technologies because monolithic applications make it extremely difficult since it use the same technology for the whole codebase. It would be extremely expensive to rewrite the entire application to use the newer framework

So adopting this kind of architecture we will end up with an application that has grown into a monolith, understood by few developers, written using obsolete technologies, difficult to scale and unreliable. As a result, agile development and delivery of applications is impossible.

2.2 Microservices Architecture

To shift from the Monolithic Hell many organizations, such as Amazon, and Netflix, are adopting what is now known as the Microservices Architecture pattern. Instead of building a single monstrous, monolithic application, the idea is to split application into set of smaller, interconnected services.

A service typically implements a set of distinct features or functionality and is a mini-application that has its own architecture. Some microservices would expose an API and might implement a web UI. At runtime, each instance is often a cloud virtual machine (VM) or a Docker container.

Each functional area of a monolithic application will be implemented by its own microservice. Moreover, the web application is split into a set of simpler web applications making easier to deploy distinct experiences for specific users, devices, or use cases. Each backend service exposes a REST API and most services consume APIs provided by other services. Some REST APIs are also exposed to the mobile apps that however don't have direct access to the backend services. Communication is mediated by an intermediary API Gateway that does load balancing, caching, access control, API metering, and monitoring.

With Microservices Architecture rather than sharing a single database schema with other services, each service has its own database. Moreover, a service can use a type of database that is best suited to its needs, the so-called polyglot persistence architecture.

Microservices Architecture pattern has many similarities to SOA because in both approaches, the architecture consists of a set of services. However the Microservices Architecture despite SOA doesn't need the baggage of web service specifications and an Enterprise Service Bus (ESB). Microservice-based applications favor simpler, lightweight protocols such as REST and they implement ESB-like functionality directly in the microservices themselves. Also they don't need a canonical schema for data access.

2.2.1 Microservices Benefits and Drawbacks

The Microservices Architecture has a many of important benefits:

- It tackles the problem of complexity decomposing what would otherwise be a monstrous monolithic application into a set of services. While the total amount of functionality is unchanged, the application has been broken up into manageable chunks or services enforcing a level of modularity that is extremely difficult with the monolithic one. Consequently, services are much faster to develop, and much easier to understand and maintain.
- Each service could be developed independently by a team focused on that service and they are free to choose the best suited technologies provided that the service honors the API contract. Since services are small, it becomes more feasible to rewrite an old service using modern technologies.
- Microservices pattern enables continuous deployment so that each microservice can be deployed independently. Changes can be deployed as soon as they have been tested.
- Finally, each service can be scaled independently, deploying for example just the number of instances of each service that satisfy its capacity and availability constraints. Moreover, we can use the hardware that best matches a service's resource requirements.

Like every other technology, there are no silver bullets and the Microservices architecture pattern has drawbacks:

- One of major drawback of microservices is the complexity that arises from distributed system when implementing an inter-process communication mechanism based on either messaging or RPC. Moreover, we need to handle partial failures, since request destination might be slow or unavailable.
- Another challenge with microservices is the partitioned database architecture. Transactions that updates multiple entities are simple to implement in

monolithic because there is a single database. In microservices applications, we need to update multiple databases owned by different services. Using distributed transactions is usually not an option, first for the CAP theorem and second they are not supported by many NoSQL databases and messaging brokers.

- Testing is also more complex. For example, with a modern framework such as Spring Boot, it is trivial to write a test class that starts up a monolithic web application and tests its REST API. In contrast to a test that starts up a monolithic application and tests it, a similar test for a service would need to launch that service and any services that it depends upon.
- Another challenge is implementing changes that takes multiple services For example implementing changes to services A, B, and C, where A depends upon B and B depends upon C we need to plan and coordinate the rollout of changes
- to each of the services.
- Each service will have multiple runtime instances and that's many more moving parts that need to be configured, deployed, scaled, and monitored. In addition a service discovery mechanism is needed to enable a service to discover the locations (hosts and ports) of any other services it needs to communicate with.

To summarize, it's obvious that building complex applications is inherently difficult. The Monolithic Architecture pattern only makes sense for simple, lightweight applications. The Microservices Architecture pattern is the better choice for complex, evolving applications, despite the drawbacks and implementation challenges.

2.3 Designing Microservices

2.3.1 API Gateway

When building application as a set of microservices, we need to decide how your clients will interact with the services. For example if we are developing a native mobile client for a shopping application when using a monolithic application architecture, the client retrieves data by making a single REST call to the application. A load balancer routes the request to one of several identical application instances. The application then queries various database tables and return the response to the client. In contrast, when using the microservices architecture, the data displayed is owned by multiple microservices.

The first choice could be a Direct Client-to-Microservice Communication where a client makes requests to each of the microservices public endpoint. The endpoint URL would map to the microservice's load balancer, which distributes requests across instances.

Unfortunately, there are challenges and limitations with this option. One problem is the mismatch between the needs of the client and the fine-grained APIs exposed by each of the microservices because it has to make several separate requests. Another problem is that a service could use protocols not web-friendly. Another drawback is that it's difficult to refactor microservices. For example, merge two services or split a service into two or more services could be extremely difficult if clients communicate directly with the services.

Usually a much better approach is to use an API Gateway. An API Gateway is a server that is the single entry point into the system similar to OOP's Facade pattern. The API Gateway encapsulates the internal system architecture and provides an API that is tailored to each client. It might have other responsibilities such as authentication, monitoring, load balancing, caching, request shaping and management, and static response handling.

The API Gateway is responsible for request routing, composition, and protocol translation. All requests from clients first go through the API Gateway. It then routes requests to the appropriate microservice. The API Gateway will often handle a request by invoking multiple microservices and aggregating the results It can translate between web protocols such as HTTP and WebSocket and web-unfriendly protocols that are used internally. The API Gateway can also provide each client with a custom API exposing a coarse-grained API for mobile clients.

Benefits and Drawbacks The first benefit of using an API Gateway is that it encapsulates the internal structure of the application. Clients talk to the gateway rather than having to invoke specific services.

API Gateway also has drawbacks for example it's yet another highly available component that must be developed, deployed, and managed. It could potentially become a development bottleneck because Developers must update the API Gateway in order to expose each microservice's endpoints.

For this reason it's important that the process for updating the API Gateway be as lightweight as possible.

2.3.2 Inter-Process Communication

In a monolithic application, components invoke one another via method or function calls. In contrast, a microservices-based application is a distributed system running on multiple machines with a process for each service instance. When selecting a communication mechanism for a service, it is useful to think about how services interact. There are a variety of client service interaction styles that can be categorized whether the interaction is:

- One-to-one – Each client request is processed by exactly one service instance
- One-to-many – Each request is processed by multiple service instances

or whether the interaction is:

- Synchronous
- Asynchronous

There are the following kinds of one-to-one interactions, both synchronous and asynchronous :

- Request/response. Client makes a request to a service and waits for a response.
- Notification. Client sends a request to a service but no reply is expected or sent
- Request/async response. Client sends a request to a service, which replies asynchronously The client does not block.

There are these kinds of one-to-many interactions, both of which are asynchronous:

- Publish/subscribe. Client publishes a notification message that is consumed interested services
- Publish/async responses. Client publishes a request message, and then waits a certain amount of time for responses from interested services.

Regardless of the choice of IPC mechanism, it's important to define a service's API. A service's API is a contract between the service and its clients. Using an API-first approach lets begin the development of a service by writing the interface definition and reviewing it with the client developers. Only after iterating on the definition the service is implemented.

However in a microservices application it is more difficult to evolve this API, even if all of the consumers are other services in the same application. We cannot force all clients to upgrade in lockstep with the service and probably new versions of a service will incrementally deployed such that both old and new versions will be running simultaneously.

Since clients and services are separate processes we have to deal with partial failures. A service might not be able to respond in a timely way to a client's request or it might be down because of a failure and there are some cases where it might be overloaded and responding extremely slowly to requests.

The strategies for dealing with failures include:

- Network timeouts. The idea is to never block indefinitely use timeouts when waiting for a response so that resources are never tied up indefinitely.
- Limiting the number of outstanding requests. Impose an upper bound on

the number of outstanding requests between a client and a service.

- Circuit breaker pattern. This pattern consist in tracking the number of successful and failed requests. If the error rate exceeds a configured threshold, trip the circuit breaker so that further attempts fail immediately. If a large number of requests are failing, that suggests the service is unavailable and that sending requests is pointless. After a timeout period, the client should try again and, if successful, close the circuit breaker.
- Provide fallbacks. Perform fallback logic when a request fails for example returning cached data or a default value

2.3.3 IPC Technologies

There are lots of IPC technologies that could be used. One example is synchronous request/response-based communication mechanisms such as HTTP-based REST. Alternatively, we can use asynchronous, message-based communication mechanisms such as AMQP. There are also a variety of different message formats. Here we gave a brief summary about most used methodologies.

Asynchronous, Message-Based Communication For messaging, we intend processes communicating asynchronously by exchanging messages. A client makes a request to a service by sending it a message. A service replies by sending a separate message back to the client. Since the communication is asynchronous, the client does not block waiting for a reply. A message consists of headers and a message body. Messages are exchanged over channels between producers and consumers.

There are many advantages to using messaging:

- Decouples the client from the service so that it does not need to use a discovery mechanism to determine the location of an instance.
- Message buffering. With synchronous protocol,such as HTTP,both the client and service must be available. In contrast, a message broker queues up the messages written to a channel until the consumer can process them.

- Explicit inter-process communication. RPC mechanisms attempt to make invoking a remote service look the same as calling a local service however they are quite different and messaging makes these differences very explicit

There are also downsides in messaging:

- Additional operational complexity as the messaging system is another component to be installed and configured
- Complexity of implementing request/response based interaction as each request message must contain a reply channel identifier and a correlation identifier. The service writes a response message containing the correlation ID to the reply channel and the client uses the correlation ID to match the response with the request.

Synchronous, Request/Response When using a synchronous, request/response-based IPC mechanism, a client sends a request to a service that processes the request and sends back a response.

Today it is common to develop APIs in the RESTful style: REST is an IPC mechanism that uses HTTP. A key concept in REST is a resource, which typically represents a business object, or a collection of such business objects and uses HTTP methods for manipulating such resources.

There benefits using a protocol that is based on HTTP:

- HTTP is simple and familiar.
- API can be tested from a browser
- Directly supports request/response-style communication
- Firewall-friendly.
- It doesn't require an intermediate broker

As always there are some drawbacks:

- HTTP only directly supports the request/response style of interaction
- Because the client and service communicate directly they must both be running for the duration of the exchange.

- The client must know the location (URL) of each service instance

There are a few options that could help in what we called interface definition when developing REST API. Some example are RAML and Swagger IDLs that allows to define the format of request and response messages. IDLs typically have tools that generate client stubs and server skeletons from an interface definition.

2.3.4 Service Discovery

Service instances have dynamically assigned network locations and typically the set of instances changes dynamically because of autoscaling, failures, and upgrades. Consequently clients need to use a more elaborate service discovery mechanism.

There are two main service discovery patterns: client-side discovery and server-side discovery.

Client-Side Discovery Pattern In client-side discovery pattern clients are responsible for determining the locations services instances and load balancing requests across them. Client queries a service registry, which is a database of available service instances. Then it select one of the available service instances using a load-balancing algorithm and makes a request.

Network location of a service instance is registered with the service registry when it starts up and it's removed when it terminates. Instance's registration is refreshed periodically using a heartbeat mechanism.

This pattern is relatively straightforward and, except for the service registry, there are no other parts to be managed. Since the client knows about the available services instances, it can make intelligent, application-specific load-balancing decisions.

A significant drawback is that it couples the client with the service registry. Client-side service discovery logic must implemented for each service clients.

Server-Side Discovery Pattern The client makes a request to a service via a load balancer. The load balancer queries the service registry and routes each request to an available service instance. Service instances are registered and deregistered with the service registry

One benefit of this pattern is that details of discovery are abstracted away from the client that simply make requests to the load balancer. This eliminates the need to implement discovery logic Also some deployment environments provide this functionality for free such as Kubernetes or Marathon. The drawback however is that unless the load balancer is provided by the environment it's another that needs to be managed.

The Service Registry Service registry is a key part of service discovery. It is a database containing the network locations of service instances and needs to be highly available and up to date. An example of service registry is Consul. Consul is a tool for discovering and configuring services. It provides an API that allows clients to register and discover services. Consul can perform health checks to determine service availability

Also, as noted previously, some systems such as Kubernetes, Marathon, and AWS do not have an explicit service registry because it's just a built-in part of the infrastructure.

Service instances must be registered and unregistered from the registry. One option is the self-registration pattern where service instances register themselves. One benefit is that it is simple and doesn't require other system components. However it couples the service instances to the service registry we must implement the registration code.

The other option is the third-party registration pattern where another system component that manage the registration of service instances. One example of a service registrar is Registrator project that automatically registers and unregisters service instances that are deployed as Docker containers. A major benefit is that services are decoupled from the service registry and we don't need to implement service-registration logic. Instead, service instance registration is handled in a centralized manner within a dedicated service.

2.4 Deploying Microservices

When we Deploy a monolithic application we usually run one or more identical copies of a single application. We provision N servers and run M instances on each server. A microservices application consists of tens or even hundreds of services written in a variety of languages and frameworks. Each one is a mini-application with its own specific deployment, resource, scaling, and monitoring requirements. Also, each service instance must be provided with the appropriate CPU, memory, and I/O resources What is even more challenging is that despite this complexity, deploying services must be fast, reliable and cost-effective. One option is to use a containerized approach such as docker.

2.4.1 Docker

Docker technology uses Linux kernel and its features to segregate processes so they can run independently. This independence is the intention of containers to run multiple processes and apps separately from one another and make better use infrastructure while retaining the security.

Container tools like docker, provide an image-based deployment model. This makes it easy to share an application, or set of services, with all of their dependencies across multiple environments. Docker also automates deploying the application inside this container environment.

There are many advantages to consider when using Docker approach:

- Modularity. Docker is focused on the ability to take down a part of an application, to update or repair, without unnecessarily taking down the whole app.
- Layers and image version control. Each docker image file is made up of a series of layers. These layers are combined into a single image. A layer is created when the image changes. Every time a user specifies a command, such as run or copy, a new layer gets created.
- Docker reuses these layers for new container builds, which makes the build process much faster. Intermediate changes are shared between images, fur-

ther improving speed, size, and efficiency.

- Rollback. The best part about layering is the ability to roll back.
- Rapid deployment. Docker-based containers can reduce deployment to seconds by creating a container for each process, we can quickly share those similar processes with new apps. And, since an OS doesn't need to boot to add or move a container, deployment times are substantially shorter.

2.4.2 Service Instance as Container

Adopting a Service Instance per Container means that each service instance runs in its own container. To use this pattern service should be packaged as a container image. A container image is a filesystem image consisting of the applications and libraries required to run the service. Once packaged the service as a container it then be scaled launching one or more containers. Multiple containers are usually run on physical or virtual host.

Containers could be manged by cluster manager such as Kubernetes or Marathon A cluster manager treats the hosts as a pool of resources. It decides where to place each container based on the resources required by the container and resources available on each host.

The benefits of containerized deployment are similar to those of virtual machines. They isolate service instances from each other. Containers can easily be monitored and like VMs containers encapsulate the technology used to implement your services. Container management API also serves as the API for managing services. Unlike VMs, containers are a lightweight technology and images are typically very fast to build. There are some drawbacks to using containers. While container infrastructure is rapidly maturing, it is not as mature as the infrastructure for VMs. Also, containers are not as secure as VMs, since the containers share the kernel of the host OS with one another. Another drawback of containers is that you are responsible for the undifferentiated heavy lifting of administering the container images.

Also, containers are often deployed on an infrastructure that has per-VM pricing. Consequently there are extra cost of over provisioning VMs in order to handle spikes in load.

Chapter 3

Aggregate Computing

Nowaday's scenarios of distributed computation such as Internet of Things needs proper abstractions in order to tackle multi-faceted and multi-layered complexity. Aggregate computing is an approach that provides and abstraction layer on top of large scale ubiquitous systems. The idea is to shift from a single device focus to an aggregate viewpoint where all these computations are seen as a single machine. So the target of computation is based on the manipulation of distributed data structures in which sensors are considered as input, and actuation as output.

To characterize these computations we first require a model of the "domain" of computation that comprise a topological model of the environment, the set of situated devices, and the context regulating which part of the environment affects actually computations at a device. Then we need a model of computation that could be performed over such domain. Since the goal is to create programs that specifies how to transform, aggregate, and propagate these data structures such program involves a computational process that spans over a region of space and time.

Such local tasks called computation rounds, comprise the following mechanisms:

1. gathering of messages from nearby devices,
2. perception of contextual information through sensors,
3. storing local state of computation,

4. computing a new local state,
5. dispatching messages to neighbors, and
6. executing actuation.

Aggregate computing could be seen as the definition of global behaviors which are spread into atomic actions repetitively executed through space and time by devices.

This framework is based on the notion of computational field and the associated field calculus. The computational domain model is given by a dynamic neighboring relation that represents physical proximity. A computational field is then a distributed data structure mapping each device to a computational object. The field calculus formalizes the primitives for achieving a global manipulation of these data structures. Global computations are functions from field to field. Traditional functions could be lifted to work on fields and we can identify reusable building blocks to capture recurrent field manipulations.

Field calculus programs are executed by a network of computing devices. Every device compiles the same program expressing the global computation, into a computation round function according to a well-defined operational semantics devices. The result of a round is called an export and is broadcasted to the whole neighborhood. The local execution context consists of the last computed export, the exports of the neighbors, the local sensors and the "environmental sensors," which are provided by the infrastructure. The local context gives to each device a peculiar behavior that results into a global effect. With this approach the importance of individual computing devices fades.

Aggregate Development In Scala

Scafi (Scala fields) is an aggregate programming frame- work, developed in the Scala programming language, that includes: (i) an internal Domain Specific Language (DSL) providing a syntax and the corresponding semantics for the field calculus constructs as Scala APIs, and (ii) a distributed middleware that supports the configuration of aggregate systems, according to different distributed computing scenarios, as well as their runtime execution.[10]

Scafi implements the semantics of field calculus providing a virtual machine for the execution of computational rounds. It is possible to define common operations on fields and high-level behaviors, such as self-organization patterns and coordination algorithms, by simply implementing Scala methods. In fact, the functional character promotes systematic factorization of behavior into building blocks and layers of increasing abstraction.

After defining such building blocks, we could also define expressive APIs such as:

- **randomRegions(d)** creating a partition of the network in regions of diameter d,
- **average(partition,val)** averaging val in each region of partition,
- **summarize(partition,val)** summing val in each region of partition), and
- **localDensity** estimating local density of devices.

There, the advanced type system of Scala is of great help in making such APIs expressive, reusable and modular. Such APIs can be used to implement application-specific aggregate behaviors.

Chapter 4

Platform Analysis

This chapter describes the results of the analysis phase. Despite being described in one chapter the analysis process did not take place in a single instant in time. The analysis phase as that design and implementation were carried out iteratively refining requirements and discovering from time to time the solution.

4.1 Requirements

Requirements can be extracted from [4]. Basically the big picture and consequently the goal of this thesis is to build a platform that can execute Aggregate Computations in the Cloud.

The Aggregate Computing Execution Platform should:

- ingest data from situated IoT devices and store it in the Cloud
- execute aggregate computation in the cloud
- integrate with ScaFi framework [10] which provides Aggregate computing programming model and operational semantics
- be easy to deploy on cloud or on local machine
- provide reliability and scalability typical of cloud solutions.

4.2 Requirements Analysis

4.2.1 Data Ingestion

Data ingestion is the process of collecting raw data from devices while filtering grouping and transforming them in a way that could be processed by the platform. We say "raw" because there's not a priori format for these data. In general data ingested are telemetries that contains values of device's sensors. For device we intend any sort of IoT device that could emit data valuable for aggregate computations.

As the platform is intended to solve real world problems, devices are situated. These means that at least they send their position data. If a device is a fixed device and doesn't needs a GPS sensor it will always send the position which is embedded into the device itself. This could be intended as a sort of device meta-data managed by a dedicated component called "Device Manager".

Received raw data are not ready to be processed. First they need to be filtered, mapped and routed. For filtering we intend purge data that are not meaningful for the platform. If a device has multiple sensors the filtering phase includes an aggregation phase because for example raw data can come from different messages or requests and be all related to the same device. Finally the mapping phase transforms raw data in something that the platform can understand and consequently compute.

All these behaviors can be encapsulated into a piece of the system that i called Ingestion Service. For now a service is a building block of the platform that is responsible of a significant behavior.

4.2.2 Executing Aggregate Computations

Before going deep and define what means doing aggregate computation for the platform i define what is aggregate computing in general.

Aggregate computing is based on field calculus and the notion of computational field. A computational field is a function that maps the points in some discrete or continuous space-time domain to some computational value. The model of the domain of computation is characterized by the topology of computational devices (neighborhood relations) and a context that indicates which neighbors are aligned and consequently affect the computations at a device. So we could define an Execution as function which maps a context to an export that is the result of a computation round. This computation round is executed on each device. The semantic of how to compute an export from a context based on an aggregate program is defined by the ScaFi platform (see section in background about scafi).

From requirements we know that this computation has to be performed by the platform it self. Scafi comes with a distributed actor based platform that reflects the logical spacial model of aggregate programming. Devices are modeled as Actors which are in turn composed them self by child actors that handle specific behaviors:

- computation
- sensing
- actuation
- state store and
- communication

So to carry on computation in the cloud we need to move some of these behavior from the devices to the platform. These behaviors could be enclosed into a service called "Execution Service" that incarnates the execution function stated before.

A computation round performed by an Execution Service comprise the following mechanism:

- gather exports of nearby aligned devices
- gather contextual informations through sensors
- compute a new state
- store local state and
- communicate the resulting actuation back to the corresponding device.

Building the Context for computation An Execution Service needs a context to compute a round and return an export (that is the name of the result of an aggregate computation). The context is composed by: the last device export, the neighbors exports, local sensors values, and neighbors sensors values.

I already said that aggregate computations will be carried on by the platform that will be deployed in cloud, so we need to collect this information from devices and bring them to the cloud.

The necessary information for building the context it's the computation domain of devices. An isolated device it's not so meaningful for an aggregate computation. So there must be another piece of the system that i called Domain Service that is in charge of maintaining updated the devices topology and when asked replies with the neighbors of a given device. As devices are situated it will potentially return all the neighbors found in a given radius. Knowing the neighbors for a given device we are able to gather all the informations needed to build a context.

For what regards locals an remotes sensors we can think of gathering informations directly from Ingestion Service that i have previously defined. But since it's a write heavy service who handles a huge load coming from connected devices it's better to decouple readings and writings. So a new service arises called Sensors service that for a given device replies with its sensors and their last known value.

4.2.3 Platform Domain Model

During requirements analysis has raised some fundamental components that make up what we might call the platform domain model. In particular:

- Ingestion Service: it's the component that is responsible for ingestion, grouping, filtering and projection of data. After processing data it stores them in a way that could later retrieved by other platform components
- Domain Service: it's the component responsible of the topology of devices that is the "Domain" of an aggregate computation
- Sensors Service: it's the component that provides access to sensors and their values
- Ingestion Service: it's the core component of the platform which integrates Aggregate Computing Semantic and uses other component to evaluate a new

state for the field of devices

4.3 Problem Analysis

Requirements analysis provided us a first domain model with the relevant component that made up the the platform. Now it's time to think about interactions between these components and how to couple them in a Logic Architecture.

To better drive the next design phase i anticipate some concepts typical of Domain Driven Design[11]. This choice is justified by the need to tackle complexity and abstract away technology concerns. DDD methodology helps to focus on the core domain with the result of a clear and modular architecture.

One of the first notion of DDD is the need of a Ubiquitous Language. Ubiquitous Language is the practice of building up a common, rigorous language between developers and users. This language should be based on the Domain Model used in the software[11]. So before diving in to the analysis we first report some terminology:

- Domain Entity: An object that is not defined by its attributes, but rather by its identity.
- Domain Service: it's a significant process or transformation in the domain and has its standalone interface.

4.3.1 Ingestion Service

For the ingestion a big role is played by the interaction with IoT devices so we need to define how this interaction takes place.

Telemetry message semantic is intended as fire and forget so it's not important that each message is delivered successfully because the global state will be recomputed when receiving next data. If a device disconnects or for other kinds of problems is unable to communicate with the platform, computation for that device will not carried on. When device reconnects and starts again to send messaged the state will be recomputed based on the current metrics and the global state of the aggregate computation. Again a device manager could spot "broken" devices or manage a "gentle disconnection" based upon a specific protocol.

A this point we need to provide ingested data to the rest of the platform where they could be processed by other services. To better meet scalability requirements services integration is done via messaging so the architecture will be Event Driven. Event driven architectures have loose coupling within space, time and synchronization, providing a scalable infrastructure for information exchange and distributed work-flows.

So device data will be mapped to Domain Events[11]. Domain Events are a special type of Event messages. A Domain Event is something that has happened in the past, that is of interest to the platform. This last distinction means we separate all technical concerns from the domain. Domain Event are processed by Event Processor. An event processor is a component that reacts to a given a domain event with the associated Business Logic.

4.3.2 Domain Service

The Domain Service is responsible for keeping up to date the devices topology so it needs to retrieve informations about their positions.

Since we have an Event Driven Platform we could think that this service has an Event Processor that consumes position events. These information should be stored in a way that could support Spatial Queries for example find all neighbors of a given device within a certain radius. Again we are interested only in the last known position. Obliviously Domain Service should provide an API used by other service to fetch domain informations.

4.3.3 Sensors Service

This service should provide an API for sensors values manly to decouple ingestion writings from services readings. It's obvious that there is concurrency issue but it's not important that we retrieve exactly the last known value. We could leverage the optimistic concurrency model [1]. So when asked this service will

[1] https://msdn.microsoft.com/en-us/library/aa0416cz(v=vs.110).aspx

eventually reply with the updated sensor value or with the last known.

4.3.4 Execution Service

Execution service is the core service of the platform. It needs to integrate with Domain Service and Sensors Service to retrieve data to build context and carry on aggregate computations.

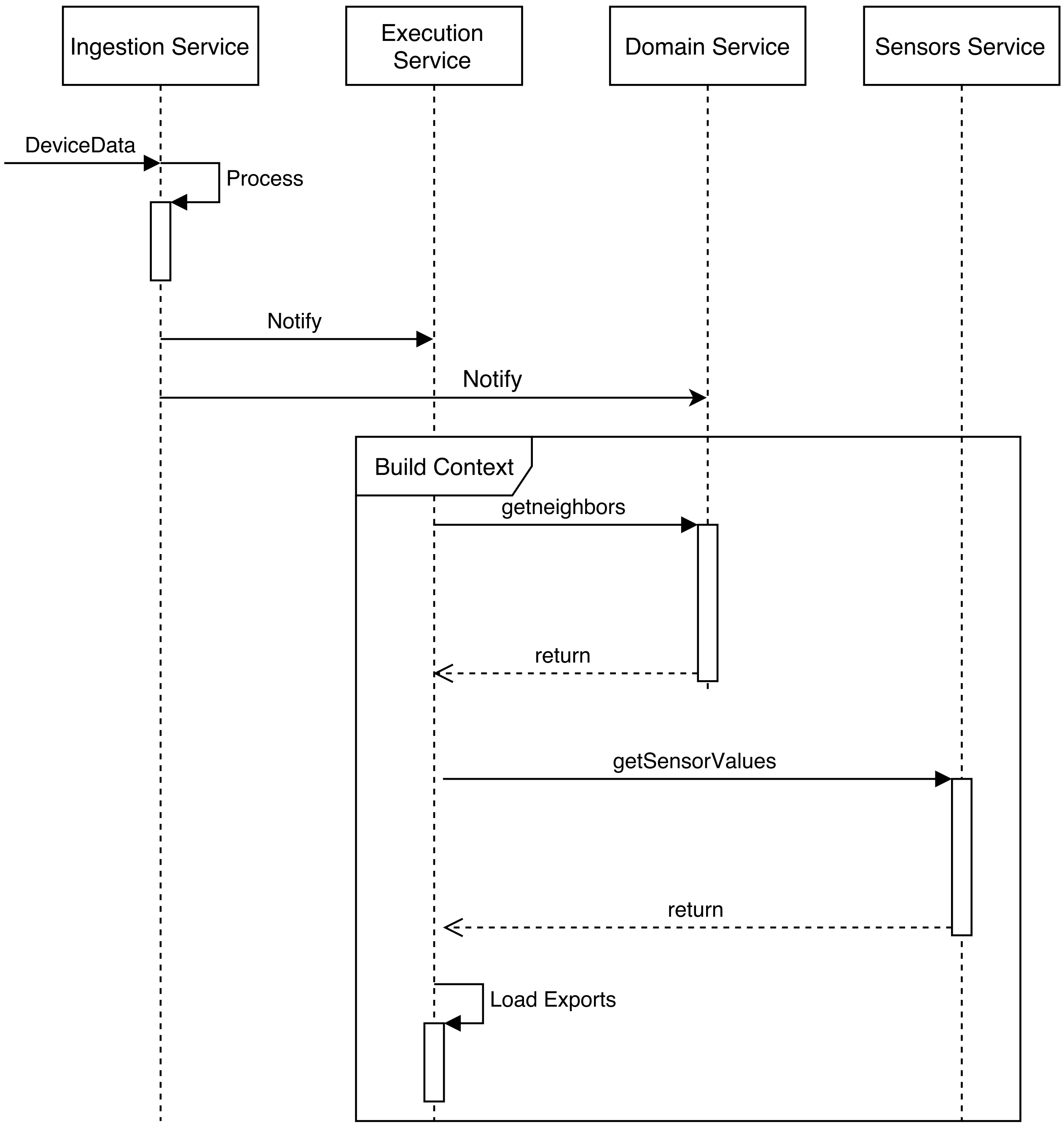
First we need to define what kind of interaction it performs with Domain and Sensors Services. Since the computation can not occur unless it has been first built a context the interaction is of type request response. During design could could be evaluated the implementation an asynchronous response to parallelize the two requests.

Second it's necessary to define how computation should be triggered. ScaFi defines a sort of scheduler that operates as clock and triggers each device computation. For the platform we possibly want to update the field status for each new value of the device's sensors. So there are two kind of interaction. One of type request response with which we can trigger computation. Another of type publish/subscribe where Execution Service has the role of subscriber to Events which could trigger a new computation.

Another thing has to be defined is the status of computation. To build the context for the current computation it's necessary to retrieve the results for current and neighbors device (called Exports) of the previous computation. Domain service are stateless[11] and we want the Execution Service to be stateless too. If not so, we would have one service running for each device. This implies to extract the state and put it into an Export Entity. The identifier of such Entity it's the identifier of the device to which it refers. Implementations details are left to designers.

4.4 Logic Architecture

To summarize what we can call a Logic Architecture the most suitable diagram is the interaction Diagram.


Ingestion Service
Execution Service
Domain Service
Sensors Service
DeviceData
Process
Notify
Notify
Build Context
getneighbors
return
getSensorValues
return
Load Exports

Chapter 5

Platform Design

This chapter is focused on the design phase that is responsible to materialize the *What* into the *How*. The logical Architecture produced in the analysis phase has been evolved in one that matches scalability and reliably non-functional requirements.

5.1 Towards a Microservice Architecture

The analysis phase produced a Logic Architecture that exposes a high level view of the components of the system and how they are interconnected.

During design is crucial to keep in mind non-functional requirements as design choices will impact on performance and maintainability of the platform.

We know from requirements that the platform should be cloud based and support large-scale aggregate applications. This means that we need to consider scalability in therm of resources and performance. Scaling could take place in two directions:

- vertical scaling, when we scale resources dedicated to a single application. For example we expand the memory size of the virtual machine where our application is running.
- horizontal scaling, when we replicate the number of application instance

If we design and subsequently deploy the platform as a monolith we need to scale the whole monolith itself. Another drawback is that it's difficult to evolve and maintain because all the components are coupled with the frameworks and technologies chosen to build the monolith.

Looking at the logical architecture it's possible to observe a kind of modularity within the platform. These modules fulfill a functional division based precisely on the performed functionality from each individual module. This subdivision can be exploited to adopt a Microservices Architecture.

Instead of building a monolithic application i decompose it into a set of services in order to tackle the problem of complexity. Each service has a well-defined boundary. Consequently, individual services are much faster to develop, and much easier to understand and maintain. Finally, the microservices architecture pattern enables each service to be scaled independently.

Design phase should consider this architectural choice defining each service interface with a clear specification of messages exchanged. So that each service can be developed and possibly tested independently.

Technological choices of the underlying infrastructure that enables microservices to communicate should not leak to the service. It has to be infrastructure agnostic. This also helps testability of the service itself.

5.2 Platform Services Design

The microservices architectural approach allows to design each service independently but often they display similar internal structure consisting of some or all of the layers displayed in figure 5.1.

The first component to be designed for each service is what we may call the API Component. This component will have different implementation based on technologies that will provide inter process communication. This components acts as mappers between the application protocol exposed by the service and messages to objects representing the domain.

Almost all of the service logic resides in a domain model. Services coordinate across multiple domain activities, whilst repositories act on collections of domain entities and are often persistence backed. Repository[11] provides a centralized

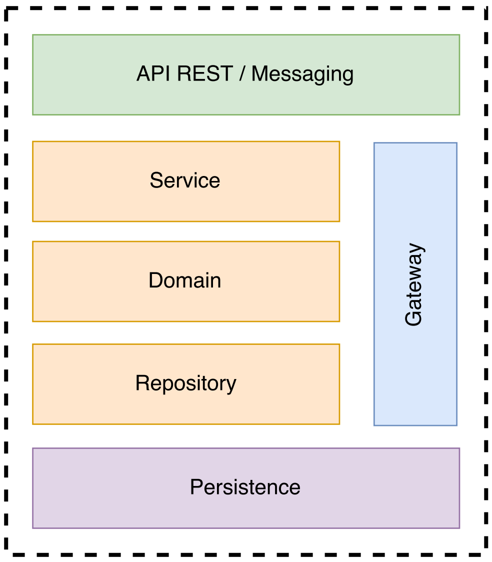


Figure 5.1: Microservice Internal Design

facade over some backing store, whether that backing store is a database, XML, SOAP, REST and so on. Repository component will have different implementation based on the best storage technology for each service.

If one service has another service as a collaborator, some logic is needed to communicate with the external service. A gateway encapsulates message passing with a remote service, marshalling requests and responses from and to domain objects. It will likely use a client that understands the underlying protocol to handle the request-response cycle.

Microservices handle requests by passing messages between each of the relevant modules to form a response. A particular request may require interaction with services, gateways or repositories and so the connections between modules are loosely defined.

Next sections describe single service design defining of which of the previous modules they are made of.

5.2.1 Ingestion Component

Ingestion Service is made of:

- IngestionServiceComponent: Component that encapsulates business logic regarding Filtering Aggregations and Mapping of data coming from Devices
- DomainApiComponent: Component that encapsulates public API that could be used by devices to send sensors data
- IngestionServiceRepository: Component used by Ingestion service to store sensors values for later processing
- IngestionMessagingComponent: Component that uses Ingestion service and integrates with the messaging platform to publish device's events

5.2.2 Domain Service Design

Domain Service is made of:

- DomainServiceComponent: Component that encapsulates business logic regarding neighborhood retrieval
- DomainApiComponent: Component that encapsulates public API that could be used to interact with the service.
- DomainServiceRepository: Component used by Domain service to retrieve neighbors of a specific device from storage
- DomainMessagingComponent: Component that uses Domain service and integrates with the messaging platform to process events from devices and update their positions

5.2.3 Sensors Service Design

The Sensors Service is made of:

- SensorsServiceComponent: Component that encapsulates business logic regarding devices Sensors management
- SensorsApiComponent: Component that encapsulates public API that could be used to interact with the service.
- SensorsServiceRepository: Component used by Sensors service to retrieve sensors values from persistent storage

5.2.4 Execution Serivice Design

Execution Service is made of:

- ExecutionServiceComponent: Component that encapsulates business logic regarding Aggregate Computations
- ExecutionApiComponent: Component that encapsulates public API that could be used to interact with the service. It depends on ExecutionService.
- ExecutionServiceGateway: Component that encapsulates the logic used when calling other services.
- ExecutionServiceRepository: Component used by Execution service to retrieve and store Exports from and to persistent storage
- ExecutionMessagingComponent: Component that uses Execution service and integrates with the messaging platform to publish events when the field is updated

Chapter 6

Implementation

6.1 Microservices Infrastructure Implementation

Individual services have been designed in order to be as possible agnostic to the infrastructure. When dealing with the infrastructure implementation there are a couple of choices that have to be made.

First of all if to make the whole microservices infrastructure or ”buy” it. Vendors such as Microsoft, are proposing microservices infrastructures as a Service. For example Azure Service Fabric abstracts away all infrastructure concerns such ad service discovery, load balancing, etc... and lets focus on the service business logic. The drawback is that the whole architecture would depend on that platform resulting in a lock in with that vendor.

The choice taken is to build the whole infrastructure and to choose step by step the appropriate technology that leads to a better solution.

Next sections presents choices taken and technologies adopted for each microservice infrastructure detail.

6.1.1 Inter Process Communication

As stated in the design phase, services exposes a public API that could be used to interact in a request/response style. We need think on how this API can be implemented.

Since ScaFi has an Actor based platform one choice was to adopt Akka Cluster[1]. Akka Cluster is an Akka module based on Akka Remote that brings the actor model to cluster computing scenarios. This approach simplifies some aspect of development because it has convenient interface for message passing but it comes with a cost. Choosing Akka as a communication protocol for such a significant part of the project means again, losing many of the benefits of a polyglot approach. First of all, to interoperate with Akka all services should run on JVM. Moreover since Akka Remote adopts netty TCP as communication protocol load balancing and service discovery should be done by hand.

A more interoperable choice is to adopt an HTTP REST approach. HTTP is a firewall friendly protocol. Moreover it's possible to adopt mainstream load balancers that most of the time are already provided by cloud vendors.

There are many HTTP frameworks that could be used to build REST services. This time Akka offers a good alternative called Akka HTTP[2]. Akka HTTP is an Akka module build over Akka Streams[3] that implements the HTTP Protocol stack. Akka Streams is the Akka implementation of reactive streams. The most interesting feature of Akka stream is built-in back-pressure. Back-Pressure enables to slow automatically a stream source if the subscriber is too slow and can't process the stream avoiding the risk of overflow. This could be obtained with HTTP by reducing the TCP window. Back-Pressure is very handy in an IoT environment where we have many devices streaming sensors data that needs to be processed.

So each service implements a REST interface written with Akka HTTP. This enables to adopt in future changes a different framework for a particular microservice and if we remain compliant to the REST interface the platform keeps working.

6.1.2 Load Balancing

There are two kinds of load balancing operations that have to be performed: internal load balancing and external load balancing.

[1] `http://doc.akka.io/docs/akka/current/common/cluster.html`
[2] `http://doc.akka.io/docs/akka-http/current/scala.html`
[3] `http://doc.akka.io/docs/akka/current/scala/stream/stream-introduction.html`

Internal load balancing is performed when a platform service needs to communicate with other services. Services are replicated so it should find the right instance. Moreover they could be moved from other hosts and the IP could change.

External load balancing is performed when clients outside the platform need to communicate with internal services.

The general approach is to have a service registry where service instance are registered. Other services use the API provided by the service registry to discover other services. This means to manage and configure another component.

If we adopt a cluster manager an alternative is to use the functions provided by the cluster itself. Docker Swarm for example uses a DNS based approach. To talk with another service it's possible to use it's DNS name. For each request Docker will provide a different IP based on a Round Robin Algorithm.

To keep things simple and open to future extension internal and external load balancing will be performed with the built-in cluster manager's load balancer. So there's no need to write additional code inside services.

Anyway an instance of Consul[4] and Registrator[5] will be provided. This helps other services that integrates with Consul such as monitoring to discover and monitor internal services.

6.1.3 Messagging Infrastructure

The message-based approach allows services to achieve a higher degree of autonomy from other services they depend on. As long as the messaging infrastructure remains operational, messages can be sent and delivered even when any one system is unavailable.

As said in the design phase autonomy is achieved through the adoption of Domain Events. When something of significance happens in one system, it produces an Event about it. There will tend to be several or even many such Events that occur in each system. As Events occur, they are published to interested parties by means of a messaging mechanism

[4]https://www.consul.io/

[5]http://gliderlabs.com/registrator/latest/

The messaging infrastructure is based on a message broker. There are various alternatives in the mainstream. For this platform i choose RabbitMQ[6].

This why RabbitMQ offers a variety of features that lets developers trade off performance with reliability, including persistence, delivery acknowledgements, publisher confirms, and high availability. Moreover several RabbitMQ servers on a local network can be clustered together, forming a single logical broker. RabbitMQ supports messaging over a variety of messaging protocols and clients. Finally it comes with a helpful management UI.

Since Domain Event can be seen as a stream of events they are read and processed from RabbitMQ with a library based on Akka Streams. Once again this brings the advantage of back-pressuring also to messaging.

6.2 On Microservices Implementation

The choices made for the microservices infrastructure will inevitably affect the implementation of services. The technologies chosen, however, must not interfere with the testability of the service. So all the parts that require or interact with an external component or service must be opportunely mocked. It was therefore necessary to evaluate the use of a dependency injection framework that allows to create, based on the context, an appropriate service instance. Having opted for Scala as a programming language, through the adoption of the Cake Pattern it's possible to implement a dependency injection framework through only constructs provided by the language. As drawback there is some boilerplate code to write in order to adopt the Cake Pattern, but in this way we avoid the dependence on external DI frameworks.

As already said each Service must implement a REST API written with Akka Http. So each service has an instance of his API Component that encapsulates the Routing between HTTP requests and the business logic of the service provided by its Service Component.

Components that need to interact with the messaging infrastructure have their

[6] https://www.rabbitmq.com/

own implementation of the messaging component. I adopted an external library that enables to process Rabbit MQ messages with Akka Streams. This component encapsulates only the messaging logic while the business logic is kept in Service Component.

6.3 Execution Service

This service is the core part of the platform and it requires maximum attention especially when implementing Messaging and API components. Afterwards i report some considerations about its components implementation.

6.3.1 Service Implementation

Service component encapsulates the business logic of the microservice. In this case, given a context it must compute the export of the aggregate application.

The semantics of aggregate computation is provided by Scafi framework that need to be integrated into the component. The first step is to create a Scafi Incarnation. In order to work this incarnation should be the same for all the components that make use of Scafi types.

Scafi computation rounds are executed by an instance of the trait AggregateProgram that specifies a function that converts a context to an export. For simplicity we wired a HopGradient Aggregate Program to each instance of the Execution Service. Future works may update the service so that it could execute aggregate program provided from users.

6.3.2 Repository Implementation

The design phase has already defined the repository component interface. For what concerns the implementation phase, we must choose a suitable technology that enables storage of the computed Exports. Since the Exports will be computed very frequently it is not necessary to persist them permanently unless we want to make a snapshot of the state of the computational field. For this reason we have chosen Redis that is an in-memory data structure store, that is used as a database.

Since Redis only accept Strings as data to be stored Exports must be serialized. This proved to be a problem since Scafi has been designed through the family polymorphism, together with Exports are also serialized external classes which contains the Export trait. A future improvement should provide a more suitable solution for serialization or adopt a storage that doesn't require to manage serialization by hand.

6.3.3 Gateway Implementation

The gateway is the component that abstracts the communication with external services. The service's REST API has been realized through the framework Akka HTTP. So also for the gateway we decided to use the Client module provided by Akka Http.

The Client offers different levels of usage based on how many aspects of the http connection we want to handle by hand. In this case we chose to use the so called "Request-Level Client Side API" to let Akka perform all connection management. When the client performs the first request to a server it creates a connection pool. Request are handled by this pool. When there are no request for a for a certain period of time it closes the connection pool. The pool will be restarted after a new connection.

6.4 Domain Service

In general the implementation of the Domain Service is in line with what has been realized for the other sevices. Some considerations could be done on the repository implementation.

The implementation of the Domain Service repository must support the search of the neighbors of a given Device through GPS location. In essence must support Spatial Queries.

We already adopted Redis for Execution Service. Since it supports Spatial Queries we adopted Redis also for Domain Repository. In addition we only need to save the ID of the devices and their position, things that Redis supports natively.

The neighbors are searched by a query that returns the ID of neighbors within a configured radius for example 100 m.

6.5 Ingestion Service

The most important part of this service is the API component because it needs to interact with devices in order to ingest data to the system.

Devices send their sensors data in Json format. They could send a Json per request or they can aggregate data and send them all in one request. This is the case of Field Gateways.

To support the scenario of aggregated Json data the Ingestion Service API uses an Akka Http functionality that is called Json streaming. Json streaming enables to materialize the entity in the http request as an Akka Stream source. So it's possible to process an http request entity in a streaming way

After processing data it will publish events through the messaging infrastructure and store processed data through its repository implementation. Again the repository is implemented using Redis

6.5.1 Sensors Service

The implementation of the Sensors service has no peculiarities. This also implements a REST API component via Http Akka and the sensor data are provided through the repository. The repository implementations it's the same of the Ingestion Service since it reads from the same db.

6.6 Testing Services

In order to test correctness of each service it's important to implement some sort of automated test through the use of ScalaTest, a popular testing framework for Scala.

I implemented some unit test to checks the correctness of domain and execution services. These unit tests use the mocked version of repository and gateway components.

Since this is a distributed platform what that really matters are the integration tests. So introduced integration tests that verify that Domain Sensors and Execution Services properly works together

Chapter 7

Deployment

Since the platform is realized with a microservices architecture hosted in the cloud it's important to put attention on the deployment phase.

7.1 Continuous Delivery Pipeline

References for this sections are taken from [12]

The execution platform is based on a microservices architecture and since each of these services can be developed and deployed separately it is necessary to use tools that simplifies these processes.

Continuous Delivery is the ability to get changes including new features, configuration changes bug fixes into production,safely and quickly in a sustainable way.

The goal is to make deployments of a large-scale distributed system with a complex production environment predictable routine affairs that can be performed on demand.

This practice helps to achieve several benefits:

- Low risk releases.
- Faster time to market. Continuous Delivery to automates: build, deployment and testing processes so that they could be executed daily. This avoids a large amount of re-work that plague a phased approach.

- Higher quality. With automated tools that discover regressions within minutes, teams are freed to focus their effort on other tasks. By building a deployment pipeline, these activities can be performed continuously throughout the delivery process, ensuring quality is built in to products and services from the beginning.
- Lower costs. Any successful software will evolve significantly over the course of its lifetime. By investing in build, test, deployment and environment automation, we substantially reduce the cost of making and delivering incremental changes to software by eliminating many of the fixed costs associated with the release process.

So for this reasons it's been adopted a CD pipeline. Details about this pipeline can be found on the following sections.

7.1.1 Continuous Integration

Continuous integration helps to automate the processes of building and testing. As CI tools we adopted Travis that is free for OSS projects and is easy to configure.

There is a Yaml file that says to Travis what actions it has to take to run test and build a new release. If we already use a build tool like SBT it will simply run the configured steps he founds in the build script.

For this projects it executes the following steps. The code base resides on a Git repository in this case GitHub. Each time a new commit is been pushed to the master branch Travis pulls that commit and runs the build process of build script. If the build step succeeds it will then run the configured test.

If also the test stage succeeds it will publish the artifacts produced. Since the platform is based on microservices there will be one artifact for each microservice. What kind of artifact should be built? For the benefits regarding isolation and simplicity of deploy Docker Image format has been chosen as artifact.

7.1.2 Docker Image as Microservice Artifact

Docker containers can help mitigate many of the challenges with the microservices architecture. Docker make use of Linux kernel interfaces such as cnames and namespaces, which allow multiple containers to share the same kernel while running in isolation . The Docker execution environment uses a module called libcontainer, which standardizes these interfaces. It's this isolation between containers running on the same host that makes deploying microservice with Docker very easy. Docker creates a DockerFile describing all the language, framework, and library dependencies for that service.

The container execution environment isolates each container running on the host from one another, so there is no risk that the language, library, or framework dependencies used by one container will collide with that of another.

The portability of containers also makes deployment of microservices painless. To push out a new version of a service running on a given host, the running container can simply be stopped and a new container started that is based on a Docker image using the latest version of the service code. All the other containers running on the host will be unaffected by this change. Containers also help with the efficient utilization of resources on a host. If a given service isn't using all the resources on an instance, additional services can be launched in containers on that instance that make use of the idle resources.

To build a Docker image for each service Travis will use again SBT with a plug-in called Native Packager. Native packages will look in the build script and execute task regarding Docker Images that specifies for example: name, version and other useful metadata for the image.

Once the image has been built it has to be pushed somewhere so that it could be deployed to production environment. Again Docker helps this process with another tool called Docker-Hub

7.1.3 Docker Hub CD

Docker Hub is a cloud-based registry service which allows to store pushed images, and links to Cloud Environment so it can deploy images to hosts. It provides a centralized resource for container image discovery, distribution and change management, user and team collaboration, and workflow automation throughout the development pipeline.

Once the image built by Travis is pushed to Docker Hub. With the Docker CLI it's now possible to pull the fresh new built image and run it.

Of course, deploying services in containers, managing which services are running on which hosts, and tracking the capacity utilization of all hosts that are running containers will quickly become unmanageable if done manually. So there's the need of a container management solution

7.2 Container Services

As the number of containers grows, it becomes increasingly difficult to manage manually, and this is where containers services can really help.

A Container Service is a cluster management framework that uses optimistic, shared state scheduling to execute processes on Cloud instances using Docker containers.

Each Cloud vendor has his own container service but they support different orchestrator. For example Kubernetes or Marathon. Since the platform should be simple to run also on a local machine the best choice is to use Docker Swarm

7.2.1 Docker Swarm

The cluster management and orchestration feature embedded in the Docker Engine is called Swarm Mode. Docker engines participating in a cluster are running in swarm mode. Swarm mode can be enabled for an engine by either initializing a swarm or joining an existing swarm.

A swarm is a cluster of Docker engines, or nodes, where services are deployed. The Docker Engine CLI and API include commands to manage swarm nodes (e.g., add or remove nodes), and deploy and orchestrate services across the swarm.

When running Docker without swarm mode we execute container commands. When Docker runs in swarm mode, we orchestrate services. Swarm services and standalone containers can run on the same Docker instance.

A node is an instance of the Docker engine participating in the swarm. To deploy an application to a swarm, a service definition is submitted to a manager node. The manager node dispatches units of work called tasks to worker nodes.

Manager nodes also perform the orchestration and cluster management functions required to maintain the desired state of the swarm. Manager nodes elect a single leader to conduct orchestration tasks.

Worker nodes receive and execute tasks dispatched from manager nodes. An agent runs on each worker node and reports on the tasks assigned to it. The worker node notifies the manager node of the current state of its assigned tasks so that the manager can maintain the desired state of each worker.

A service is the definition of the tasks to execute on the worker nodes. It is the central structure of the swarm system and the primary root of user interaction with the swarm.

A service specifies which container image to use and which commands to execute inside running containers.

In the replicated services model, the swarm manager distributes a specific number of replica tasks among the nodes based upon the scale set in the desired state.

For global services, the swarm runs one task for the service on every available node in the cluster.

A task carries a Docker container and the commands to run inside the container. It is the atomic scheduling unit of swarm. Manager nodes assign tasks to worker nodes according to the number of replicas set in the service scale. Once a task is assigned to a node, it cannot move to another node. It can only run on the assigned node or fail.

Load Balancing As said in the implementation phase the platform uses the feature provided by the orchestrator to do Service Discovery and load balancing.

Swarm Mode uses ingress load balancing to expose the services that needs to be available externally to the swarm. The swarm manager can automatically assign the service a Published Port if not configured. If a port is not specified the swarm manager assigns the service a port in the 30000-32767 range.

External components, such as cloud load balancers, can access the service on the Published Port of any node in the cluster whether or not the node is currently running the task for the service. All nodes in the swarm route ingress connections to a running task instance.

Swarm mode has an internal DNS component that automatically assigns each service in the swarm a DNS entry. The swarm manager uses internal load balancing to distribute requests among services within the cluster based upon the DNS name of the service.

7.3 Deployment Specification

After detailing technological choices it is time to provide the specifics of how effectively deploy the platform

7.3.1 Environment Creation

First of all we need to provision one or more hosts running Docker. This could be done by simply installing the Docker Daemon on a local machine (if we want to run the platform locally) or by creating Virtual Machine on a Cloud Provider.

To simplify the process there is a helpful tool for Docker Environments called Docker Machine. Docker Machine is a tool that installs Docker Engine on virtual hosts, and allows to manage the hosts with docker-machine commands. Machine can be used to create Docker hosts on local machines, o company network, in data centers, or on cloud providers.

With docker-machine commands, it's possible to start, inspect, stop, and restart a managed host, upgrade the Docker client and daemon, and configure a Docker client to talk to host. it's also possible to run docker commands directly on a pointed host.

Docker machine implements different drivers for each cloud vendor. So there is no risk of lock in because if we want to change vendor we simply use another driver.

To create a machine we simply choose a driver and run the command:

docker-machine create –driver ”diver” HOST-NAME

Then after the machine is created we connect to the machine using ssh tool provided by docker machine and we enable the Swarm mode by by running:

docker swarm init

We have now created a Swarm Manager. If we want to add Worker nodes we create another host with Docker Machine and then we run:

docker swarm join –token ”specified token”

and we specify the token provided on master creation.

7.3.2 Platform Services Deployment

After creating the cluster it's time time to deploy the platform or rather the services that compose it. This process is very time consuming if done manually. It might be useful to specify the desired status of the system in a declarative way and let the Swarm Manager do the hard job.

This is possible with Docker with the help of Docker Compose. Compose is a tool for defining and running multi-container Docker applications. Compose uses a Compose file to configure application's services. Then, using a single command, it creates and start all the services from the configuration.

There are different version of Docker Compose file. For the platform we use the version 3 that support the definition of services. A service is the definition of how we want to run the platform containers in a swarm. At the most basic level a service defines which container image to run in the swarm and which commands to run in the container. For orchestration purposes, the service defines the "desired state", meaning how many containers to run as tasks and constraints for deploying the containers. When we deploy, each image will run as a service in a container (or in multiple containers, for those that have replicas defined to scale the app).

To deploy the platform, we will run the docker stack deploy command with Docker Compose file to pull the referenced images and launch the services in a swarm. This allows to run the platform across multiple servers, and use swarm mode for load balancing and performance. Rather than thinking about running individual containers, we can start to model deployments as application stacks and services.

The command to deploy the platform stack based on the .yml compose file is:

docker stack deploy –compose-file "compose file name".yml "platform name"

Now we have the platform and consequently its services up and running. The previous command could be executed on a provisioned Cloud environment or simply in a local Docker Swarm instance for test purpose.

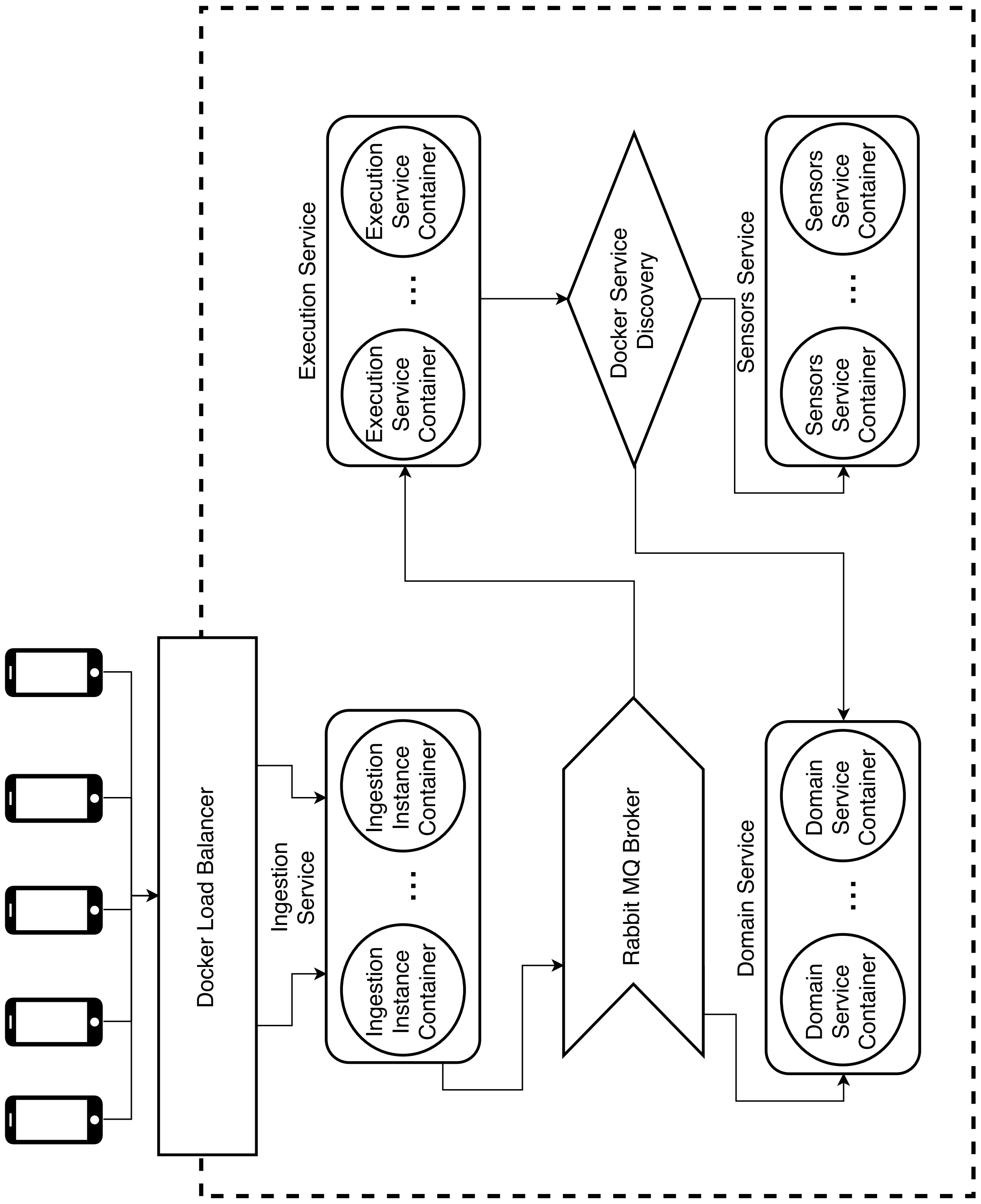


Figure 7.1: Platform Deployment Model

Chapter 8

Platform Evaluation

8.1 Testing Environment

Each service that compose the microservices based platform has been individually tested but now it's time to test the system as whole.

In order to be evaluated the platform must be deployed. Simple tests could be done deploying it on local machine. This requires an installed version of docker daemon suitable for the local OS. With the help of Docker machine and the appropriate driver (Virtual Box for Linux or Hyper-V for Windows) it's possible to create "dockerized" virtual machines and reproduce a little cluster. Than as stated in the previous chapter platform services deployment can be done through docker stack and the related compose file.

Since requirements demands that the platform could execute aggregate computation in the cloud all the evaluation test reported in this chapter has been made with a version of the platform deployed on a Cloud Environment.

8.1.1 Cloud Environment Description

When searching the most suitable Cloud Provider there are three vendors on the top list:

1. Amazon
2. Google

3. Azure

The services offered are pretty much the same, but we must pay attention to what and especially the way we use these services as inevitably there is a risk of lock-in.

So i chose to use only the Virtual Machine Hosting service in order to favor the flexibility to simplicity in setup. As provider i chose Azure. The reason is purely economic and is based on the amount of free credit available for testing.

The specifics of the environment used for evaluation tests are the following:

- 2 Azure virtual machines type DS3V2 : 4 core, 14 GB RAM, 28 GB SSD
- each machine runs the docker daemon in Swarm mode,
- one configured as a Manager the other one as Worker node.

8.2 Functional Evaluation

In this section i report Functional evaluation results. The goal is to evaluate if the platform meets its functional requirements.

Given a number of device with a fixed position i want to run an Aggregate Program that computes the Hop Gradient. The Hop Gradient is the distance intended as the number of intermediate devices through which data must pass between source and destination. We should notice that choose a Source device after a transitory the gradient should stabilize and the value of the gradient of each device should be the hop distance from this source.

To better verify this functional requirement it's helpful to have a GUI where the state of each device can be visualized. For example assigning a color to each value that the Gradient and consequently the state of a device can assume. This is why i created the Visualizer Service.

8.2.1 Visualizer Service

The visualizer service is a web application provided with an HTLM and JavaScript front-end that displays all devices in a map with the use of Open Street

Map APIs.

Devices are represented by a dot. Each dot has a color that maps to the computed value of the Gradient for that device.

The service backend is connected to the event platform and through web sockets streams the computed state to the front-end.

8.2.2 Test Results

As we could see from Figure 8.1 when the Aggregate application starts most of devices are green that is the default value of the Hop Gradient. We could see that some of them near the source start to turn red. After a short period of time we see in Figure 8.2 that devices starts spreading they computed gradient value and more of them becomes red. Finally in Figure 8.3 the gradient is stable. Devices that are still green means that they are too far from neighbors devices and there is not a path from source to them.

8.3 Non-Functional Requirements Evaluation

Rather than the functional requirements, what is really of interest is the verification of non-functional requirements.

Most of the functional requirements are already covered by Scafi so we only need verify if the platform components interact in a way to provide a correct result. For example running a Hop Gradient and verifying that it stabilizes after a short time.

Non-functional requirements cover aspects such as performance, scalability and resiliency that are much more important when building a distributed platform.

As already said the platform is distributed and composed by many services so we need a proper infrastructure to collect performance metrics of these services.

8.3.1 Platform Monitoring System

Since all services are distributed we need centralized monitoring system that collects metrics from services and provides methods to get insight on those metrics.

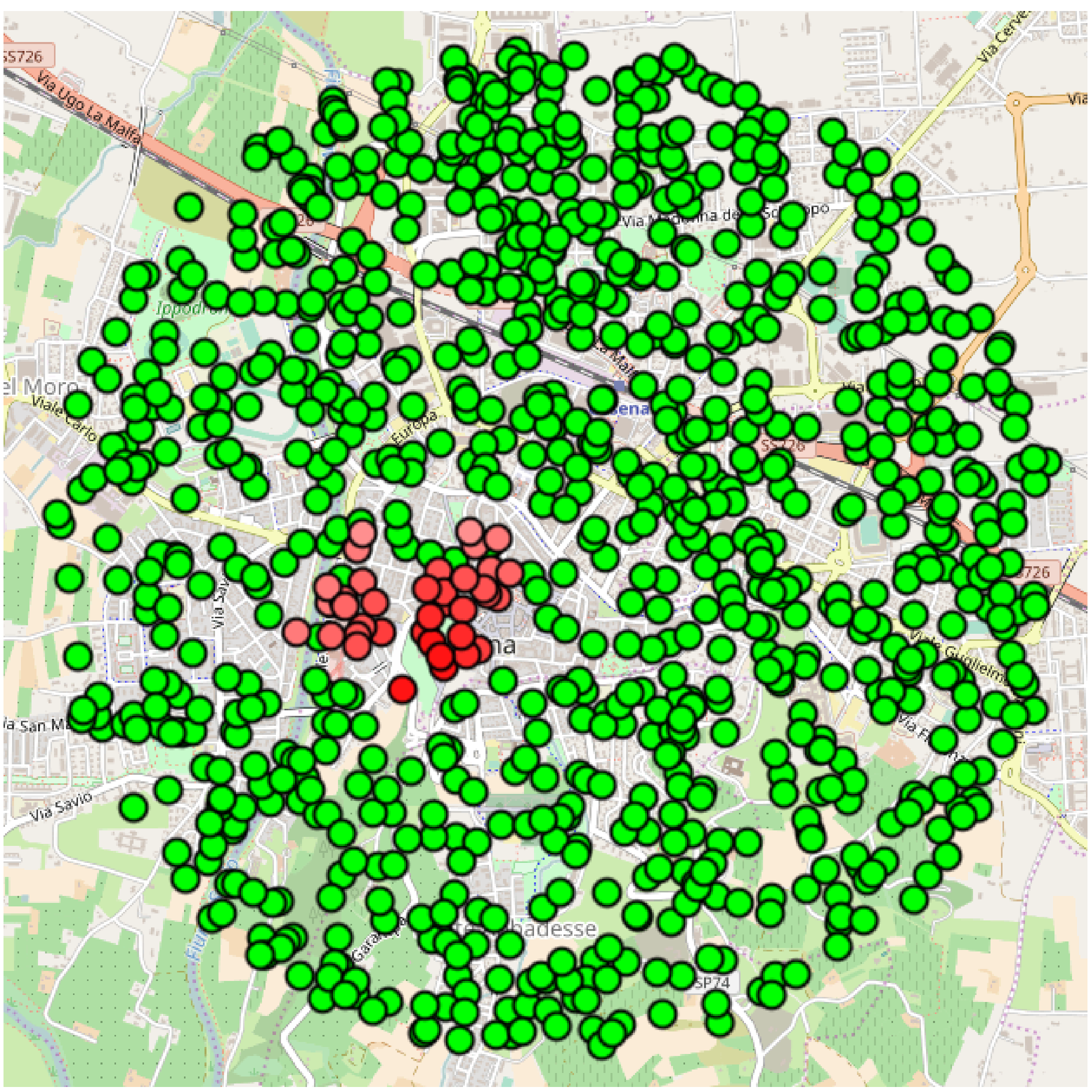


Figure 8.1: Hop Gradient Test Start

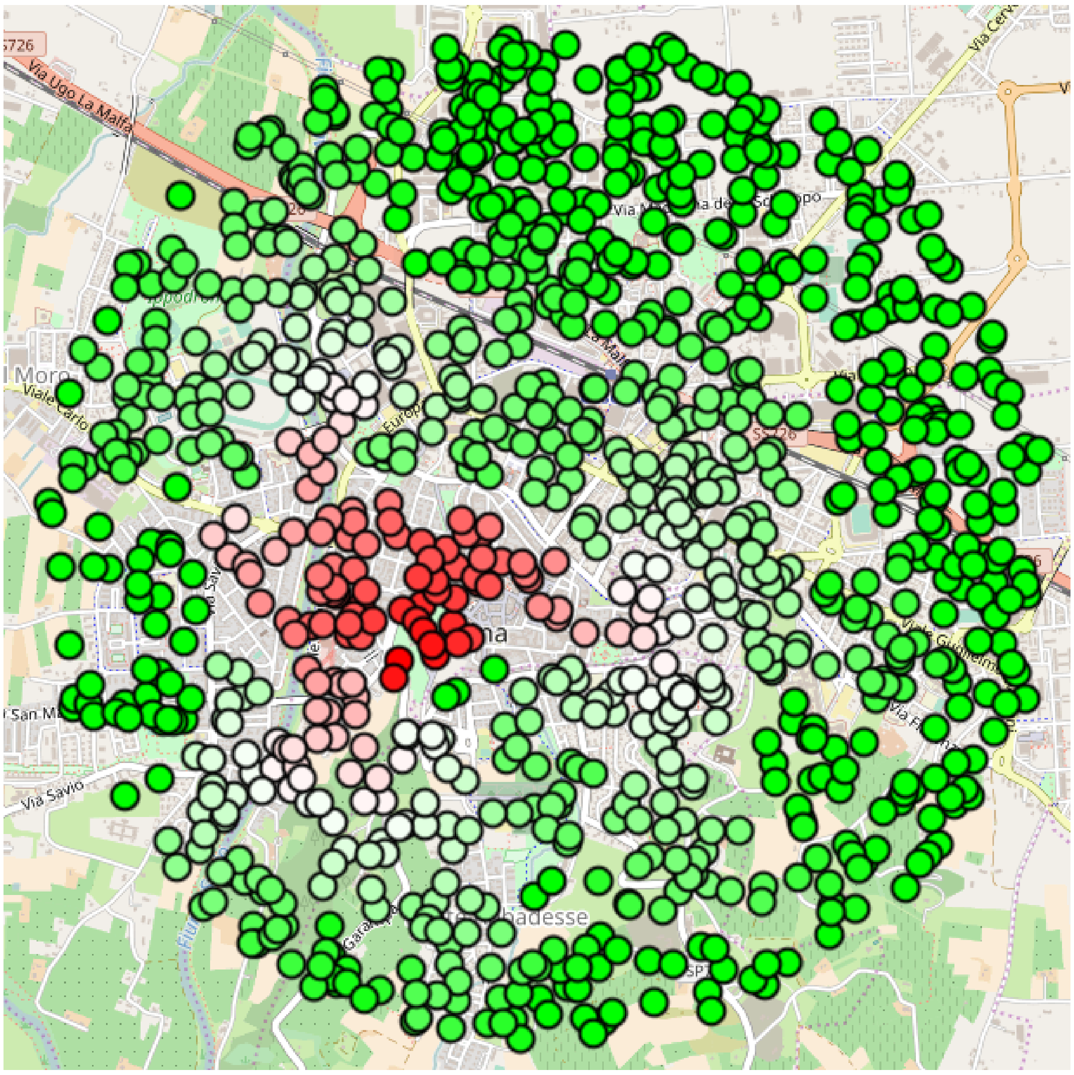


Figure 8.2: Hop Gradient Test Middle

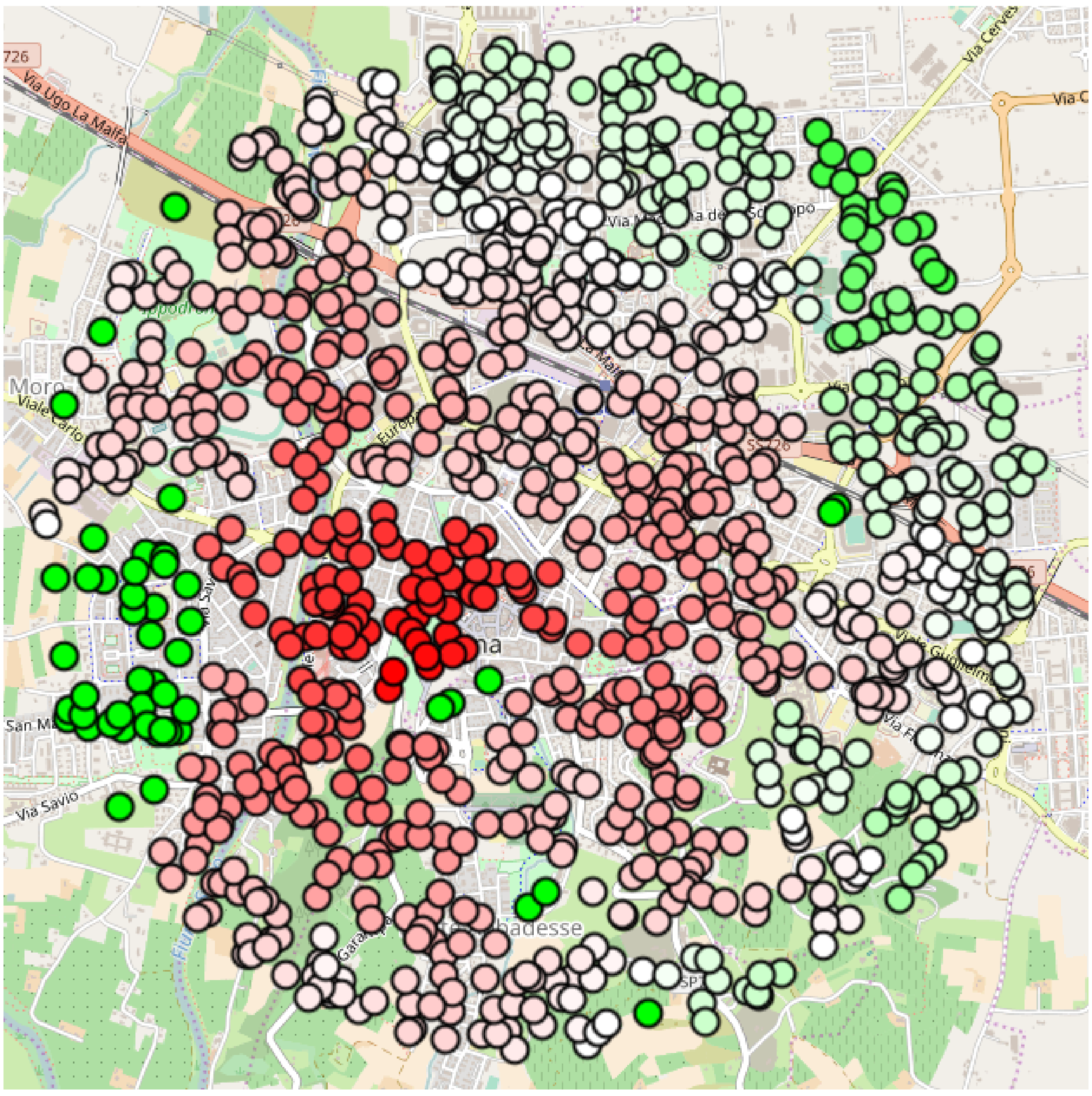


Figure 8.3: Hop Gradient Test End

For this job i chose Prometheus [1]. Prometheus is an open-source systems monitoring and alerting toolkit.

It provides a flexible query language to leverage a multi-dimensional data model and collects time series via a pull model over HTTP. Pushing time series is also supported via an intermediary gateway. Targets are discovered via service discovery or static configuration and provides multiple modes of graphing and dashboarding.

The Prometheus ecosystem consists of multiple components. The most important are the Prometheus server which scrapes and stores time series data and client libraries that are used to instrument application code.

Since each platform's service provides a REST interface Prometheus is configured to automatically scrape metrics querying this interface. Prometheus has its own data model provided trough client libraries. So each service collects metrics, maps them in to a proper Prometheus metrics type and exposes them through its REST interface.

Although Prometheus provides a GUI to visualize and query stored metrics it's quite minimal. To better explore and get insight from data i adopted another tool called Grafana[2]. Grafana is an open source metric analytics and visualization suite. It is most commonly used for visualizing time series data for infrastructure and application analytics.

It provides a configurable dashboarding system that helps to draw graphs and Gauge useful to monitor the status of the system.

8.3.2 Performance Metrics

Before presenting tests and the respective results it's better to define what tests will be performed and which kind of metrics will be taken into account to evaluate the platform.

Since i adopted a messaging architecture the best way to measure platform performance it's to measure its throughput. The throughput is intended as the number o messages processed by a piece of the system.

[1]`https://prometheus.io/`
[2]`http://grafana.org/`

For example the Execution Service reads from the queue of events coming from ingestion computes the new states and writes to the field events queue. The ratio between read rate from the first queue and write rate to the second queue is the throughput. If it's near one it means that the platform computing capacity can satisfy the current load required. If is below one it means that the platform can't afford current load and messages start to be accumulated in the queue.

A second parameter to take into account is the latency. Latency is the time between message ingestion and message processing. It is almost constant if the system is stable and can afford the current load. It grows if messages start to stay in the queue.

To evaluate the platform i tested its scaling capability and its resiliency analyzing how throughput and latency change.

8.4 Scaling Evaluation

In this test i want to evaluate horizontal scaling capabilities of the platform. So for example if the load on the platform exceeds the current processing capacity, replicating services should match the gap.

Figure 8.4 reports the results of the test.

We could see two lines. The Blue one is for the number of messages ingested while the Orange one indicates the number messages processed.

Before time A the system is stable and the Throughput of the system is maximum. Latency is constant and it's near 0,5 ms.

After time A i started to add devices until time E where the number of ingested messages grows linearly again. At time E they are twice the initial number.

We could see that from time A to B the Throughput of the platform grows as the number of ingested messages while after time B the number of processed messages grows linearly as it's has reached the maximum processing capacity. So messages starts to be enqueued and while the latency grows the throughput decreases.

At time D i duplicated the instances of Execution Services and we could see that the Throughput quickly grows and capacity could fit the current load.

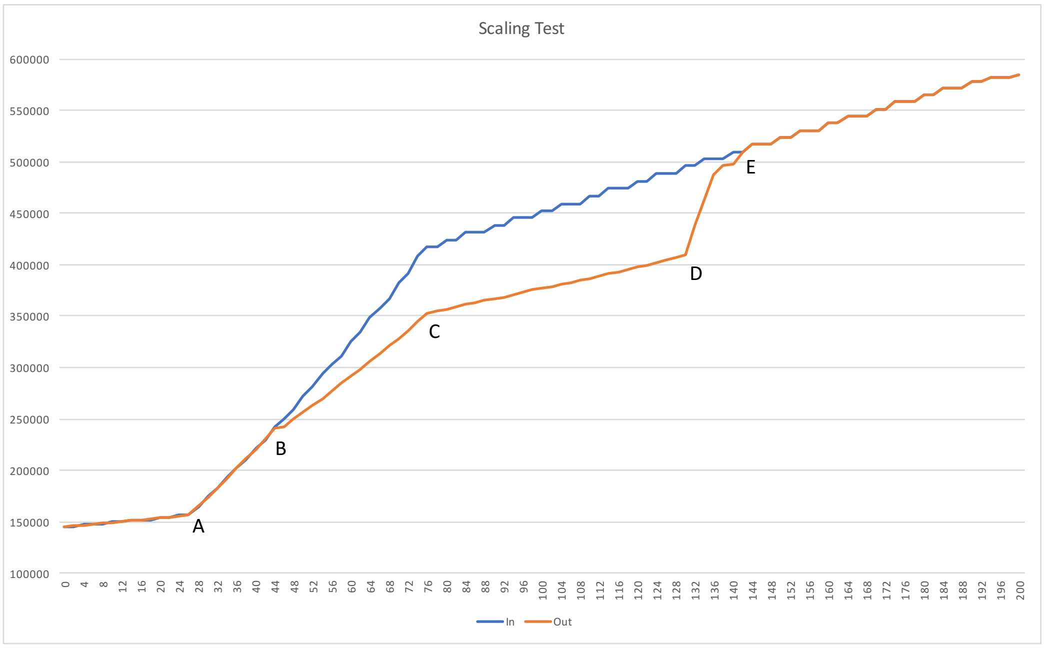


Figure 8.4: Scaling Test Throughput Chart

8.5 Resiliency Evaluation

With this test i want to evaluate the ability of the platform to recover when something bad happens.

Results of the test are reported in figure 8.5.

We could see that before time A the system is stable. At time A i simulated an unexpected event shutting down a virtual machine and as we could see the throughput quickly drops while latency grows exponentially.

The cluster manager realizes that something bad has happened and starts to restore the initial state recreating lost containers.

At time B the platform is fully restored and the throughput quickly grows.

This demonstrates that platform can heal and quickly recover from unexpected failures.

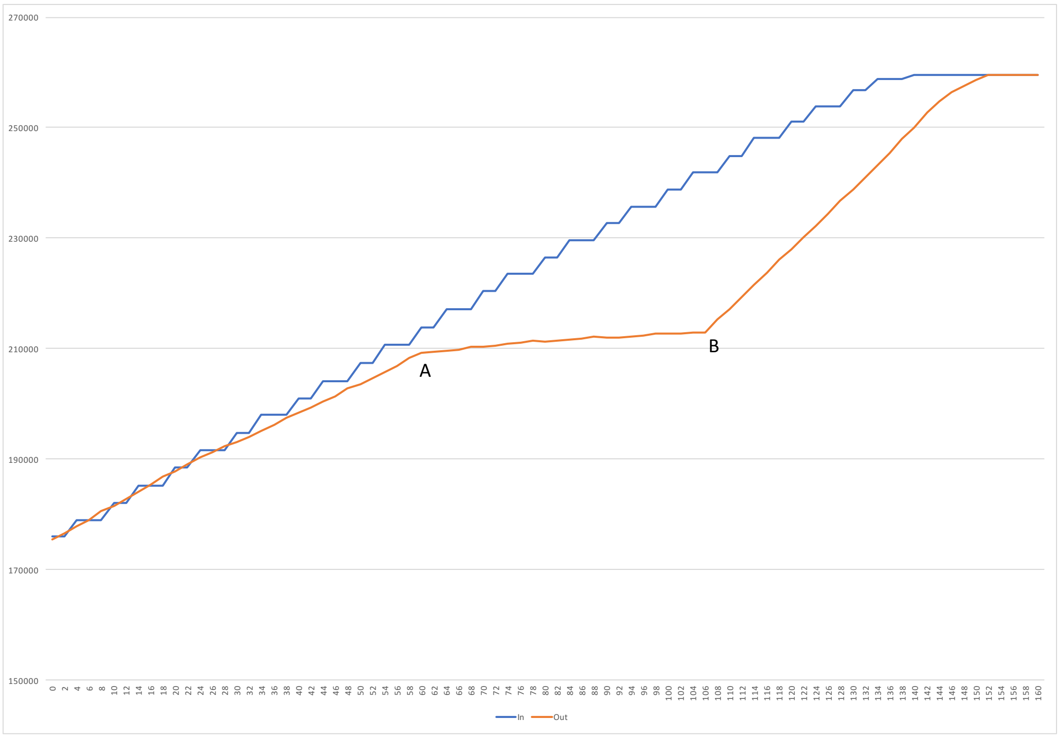


Figure 8.5: Resiliency Evaluation Chart

Chapter 9

Conclusion

The Execution Platform based on microservices developed in this thesis seems promising.

Clearly when dealing with microservices there is some complexity to manage regarding interprocess communication and services deployment.

The adoption of a lightweight framework like Akka Http has helped to quickly build a working prototype, but it is not much documented and requires more extensive testing before using it in production.

Docker has helped to manage deployment complexity. It has an extensive documentation and an active community. With few commands we could have an up and running cluster in few minutes. However Swarm Mode is quite new and lacks some functionality that are present in more established cluster manager like Kubernetes. Since I adopted a general approach to deployment, a future experiment could redeploy the platform on Kubernetes and test how the platform could benefit from autoscaling and an HA load balancer like NGINX.

Another important piece of the platform is the Event Driven Platform. Rabbit MQ is easy to configure but it is very basic and most things need to be implemented by hand. A good alternative could be Kafka which can be the object of future improvements.

Also the cloud approach has a good impact unless it's used only to host virtual machines. A more integrated cluster manager like Kubernetes can not only scale services but also scale the number of connected machines. This operation, currently, has to be done by hand.

Akka Http has proven to be helpful also when dealing with ingestion of data coming from devices. Mainly because of Akka stream back-pressure capabilities. By the way, to be a proper IoT platform, future works should consider to implement features like device management and security.

Finally after talking about technologies adopted we need to make some considerations about the platform as a whole.

As stated in the previous chapter the platform scales well when the load required grows and can self heal when failures happen. These results have been possible thanks to the combination of the microservices architecture and the adoption of cloud computing approach.

However there is work to be done concerning performance. Some empiric tests performed on the same environment used for evaluation tests demonstrated that the platform could process at most 10K devices per second. This number needs to be confirmed by a deeper analysis but clearly does not repay all the complexity introduced by the adoption of a microservices architecture.

Future tests will inspect each service performance in isolation to better point out where are the bottlenecks.

Bibliography

[1] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010.

[2] Dave Evans. The internet of things, how the next evolution of the internet is changing everything. Technical report, 2011.

[3] Jacob Beal and Mirko Viroli. *Aggregate Programming: From Foundations to Applications*, pages 233–260. Springer International Publishing, Cham, 2016.

[4] Mirko Viroli, Roberto Casadei, and Danilo Pianini. On execution platforms for large-scale aggregate computing. In *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct*, UbiComp '16, pages 1321–1326, New York, NY, USA, 2016. ACM.

[5] Alessandro Bassi, Martin Bauer, Martin Fiedler, Thorsten Kramp, Rob Van Kranenburg, Sebastian Lange, and Stefan Meissner. Enabling things to talk. *Designing IoT Solutions With the IoT Architectural Reference Model*, pages 163–211, 2013.

[6] P. Fremantle. A reference architecture for the internet of things. Technical report, 2014.

[7] Everton Cavalcante, Marcelo Pitanga Alves, Thais Batista, Flavia Coimbra Delicato, and Paulo F Pires. An analysis of reference architectures for the internet of things. In *Proceedings of the 1st International Workshop on Exploring Component-based Techniques for Constructing Reference Architectures*, pages 13–16. ACM, 2015.

[8] Chris Richardson. Introduction to microservices. *URL: https://www. nginx. com/blog/introduction-to-microservices*, 2015.

[9] Sam Newman. *Building microservices.* " O'Reilly Media, Inc.", 2015.

[10] Roberto Casadei. *Aggregate Programming in Scala: a Core Library and Actor-Based Platform for Distributed Computational Fields.* PhD thesis.

[11] Eric Evans. *Domain-driven design: tackling complexity in the heart of software.* Addison-Wesley Professional, 2004.

[12] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Adobe Reader).* Pearson Education, 2010.