POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Microservice Oriented Pipeline Architectures

Supervisors

Prof. Giovanni MALNATI

Candidate

Eugenio DURANTI

Academic Year 2019/2020

Abstract

Microservice Architecture is an architectural style becoming more and more popular nowadays. It's getting over the old paradigm not only because of the benefits it provides but also because it fits perfectly with the Cloud world, in particular with the capabilities of the network to provide on-demand availability of computer system resources.
The thesis goal is to design a microservice architecture starting from a monolith. The use case is a parsing application for generics semi-structured data coming from a huge variety of devices like IoT devices of a smart city system rather than an industry 4.0 sensor system.
The devices generate data that need to be parsed, classified, processed, aggregated and eventually displayed in a human-readable fashion.
In the context of application pipelines, the present thesis work aims to implement a microservice architecture addressing distributed system properties like scalability, maintenance, high availability, and fallback mechanisms to build a more robust and a more agile architecture from the modularity point of view.
However, Microservice Architectures are not a cure-all. Although they allow many advantageous features, some drawbacks need to be kept in mind when deciding to implement a Microservice Architecture, for instance, the structural complexity that such architectural style adds to the application.
There is not a solution written in stone, this thesis shows which are the main challenges addressed when building Microservices in the context of pipelines and tries to summarize the advantages and disadvantages of the architectural design.

Acknowledgements

ACKNOWLEDGMENTS

Questo lavoro di Tesi è dedicato a mio fratello Alberto,
il faro più luminoso che mi indica la strada.
Eugenio

Table of Contents

List of Figures

Acronyms

AOP
Aspect OrientedProgramming

CaaS
Cache as a Service

CPsubsystem
Consistent and Partition tolerant subsystem

DAO
Data Access Object

ORM
Object Relational Mapping

POJO
Plain Old Java Object

REST
REpresentational State Transfer

SaaS
Software as a Service

SFTP
Secure File Transfer Protocol

SOAP
Simple Object Access Protocol

Chapter 1

Introduction

"Architecture is the decisions
that you wish you could get right early
in a project"
-Ralph Johnson

1.1 The Project

The purpose of this thesis was to implement a Microservice Oriented Pipeline Architecture to parse and aggregate semi-structured data. The thesis project consists basically of three phases:

1. **Architectural design** The first phase consisted of analyzing the monolith application to identify and isolate the functional parts of the software in order to design a balanced separation of concern.

2. **Development** The second phase constisted of integrating the classes extracted from the monolith into the microservices under development while taking care of a series of aspect like Project Structure, Dependency Management and Profiling

3. **Deployment** The last phase is deployment. Once we have Microservices working we need to run them in a real scenario. This means Dockerize the microservices as well as all infrastructure servers, ensuring to meet requirements with respect to network isolation, scalability and High-Availability and have a system that can be deployed in a plug and play approach.

1.1.1 Overall

The pipeline is responsible to parse, decorate and aggregate heterogeneous semi-structured data with the goal to prepare them for a further elaboration and visualization.
Due to the highly heterogeneous kind of data, the elaboration has been organized in pipeline's stages where each of them is responsible to treat the data on a different granularity.
The elaboration starts when some data are stored in an SFTP Server: our data lake[1].
The application is then made up by subsequent elaboration processes, organized as Microservices to guarantee a strong decoupling factor.
In order to carry out their tasks, every microservice acts as consumer with respect to a source, perform its elaboration and in somehow deliver the elaboration's

[1][1]A data lake is a system or repository of data stored in its natural/raw format,usually object blobs or files. A data lake is usually a single store of all enterprise data including raw copies of source system data and transformed data.
A data lake can include structured data from relational databases (rows and columns), semi-structured data (CSV, logs, XML, JSON), unstructured data (emails, documents, PDFs) and binary data (images, audio, video)

product somewhere: typically towards the next microservice in the chain.
To achieve this, as well as to respond to typical distributed system's issues like scalability and high availability, the microservices have been clusterized by means of Hazelcast technology, following an `Embedded Topology`.
It means that each microservice contains an Hazelcast Memeber that will have its own set of threads and will exposes the port 5701.
Hazelcast members perform automatically a discovery of other Hazelcast members to join into an Hazelcast Cluster, allowing to clusterize the entire application.

1.1.2 Ingester

The first stage of the pipeline is called Ingester.
Ingester interact with the data lake to fetch the data and feed them to the pipeline. With this in mind the Ingester is responsible to continuosly poll the SFTP server to fetch the data and to prepare them for the elaboration.
The data are organized in Entities called `RawEntity` and are now ready to be elaborated flowing through the pipeline.

1.1.3 Parser

The subsequent stage is the Parsing.
Typically data are associated with metadata.
Metadata can be separated from associated data in different files and still the data themselves can be also split up and come along out of order.
The Parser is so broken up in two stages: in a first instance we need to identify which are metadata and which data are associated with. This operation is going to transform the more primitive entity `RawEntity` in a semantically more consistent model called `Entity`. This job is still done by the `Ingester`.
The actual parsing is performed in a second stage where we look inside the data, give them a semantic meaning and resolve the dependencies between the various `Entity`.
The output of such operation is an entity called `Item` which groups data with the same semantic meaning (e.g. an audio chopped in chunks that needs to be reassembled) and are now ready to be aggregate in an higher level entity.

1.1.4 Aggregator

The aggregator is the final stage in the pipeline.
The Aggregator's purpose is to semantically aggregate `Item` entities in the `Blob` to let visualize the data on semantic base.

1.2 Pipelines

[2] A pipeline is a set of data processing elements, called stages, chained together, where the output of one element is fed in input to the next element. Pipeline is a pervasive concept in computer science, in this context we will refer to Software pipelines which consist of a sequence of computing processes conceptually executed in parallel, with the output stream of one process being automatically fed as the input stream of the next one.

1.2.1 Why Data Pipelines

The efficient flow of data from one location to another (e.g. from a SaaS application to a data warehouse) is one of the most critical operations for enterprise applications. After all, useful analysis cannot begin until the data becomes available or are prepared for elaboration.
Data flow can be unreliable because there are many things that can go wrong during the migration from one system to another: data can become corrupted, it can hit bottlenecks (causing latency) or data sources may conflict and/or generate duplicates.
As the complexity of the requirements grows and the number of data sources increases, such problems worse in scale and impact over the system.
Data pipelines offer an effective solution for such kind of issues. Pipelines get rid of many manual steps and enables an automated flow of data from one stage to the next. The biggest advantages of a data pipelines are:

1. **It automates the processes** involved in extracting, transforming, combining, validating, and loading data for further analysis and visualization.

2. **It enhances throughput** by eliminating errors and mitigating bottlenecks or latency.

3. **It can process multiple data streams at once**: it is an pretty appealing for today's data-driven enterprise.

A data pipeline views all data as streaming data and it allows for flexible schemas. Regardless of whether it comes from static sources (like a flat-file database) or from real-time sources (such as IoT devices), the data pipeline divides each data stream into smaller chunks that it processes in parallel, ensuring an optimized computing. The data pipeline does not require the ultimate destination to be a data warehouse. It can route data into another application, in our case will be a server responsible to dialogue with a client application in order to visualize such data.

1.3 Microservices

A Microservice Architecture is a software architectural style that consists in developing an application as a set of small services meant as independently deployable components, responsible of serving a particular logical aspect of the entire application. Such services build up the application by cooperating together through a lightweight communication.
This architectural style is opposed to the Monolith Architectures: an application that is made by a single executable unit responsible for serving all the aspects of the application. Monolith applications can be successful because of the simplicity, but present some drawbacks. The first we can notice is that due to the tight-coupled nature of this approach is difficult to mantain a certain level of modularity over time and even a little change will be propagated in the whole application that eventually will have to be retested and redeployed.
Another issue is that a monolith can scale, but you would need to scale the entire application requiring a bigger amount of resources than scaling only a part of it.

1.3.1 Distribution means communication

[3] At the dawn of distributed computing, in the early '80s, the need to have computers communicating to each others brought to the introduction of one of the more widespread technology in the distributed calculus: RPCs. RPC stands for Remote Procedure Call, the idea behind it was that developers didn't have to care anymore whether a procedure call is done locally or remotely, the promise was so that since the computational power was quite limited, with such technology now they were able to design big cross-machine systems able to distribute the elaboration over an high number of machines transparently.
In part because the machines power forced to have distributed elaboration, in part because the idea behind the RPC was to make a transparent usage of remote calls, this led to a situation where the interfaces granularity was very fine-grained.
The problem here is that make a remote call is not the same to perform a local call either from a performance, security and robustness point of views. As the technology grows, the technological limits don't justify anymore the tendency to distribute the elaboration with such granularity. The need to distribute the software came out to be strictly bounded by the need to have standards for communication between applications and this led to the idea of using HTTP to transport, XML to phrase the messages and use these mechanism to to invoking method calls. This approach in fact has been introduced by Microsoft and called *Simple Object Access Protocol*: SOAP.
If we try to identify the biggest advantage of using SOAP architectures, we'd find out that indeed SOAP was able to solve interoperability issues in an effective way:

defining a widely accepted standard protocols. But this approach, again, started to collapse and that's because still treats remote calls as if they were locals and stuff started going worse when system's complexity blew up and developers were asked to write layers and layers to treats additional concept which SOAP didn't was meant for (e.g. Exception handling, transaction support, security, digital signatures).
The industry then started to migrate towards another approach: RESTful Architectures. RESTful architectures turned out to be very popular and this comes from its simplicity: treating HTTP as HTTP.
Rather than layer procedural call semantics over HTTP, REST instead treats the HTTP in the way they were specified in terms of create, read, delete, and update semantics.

1.3.2 From Monolith towards Microservices

The old paradigma in data platform architectures was to have a big monolith application whose goal was:

- Ingest some data: Such data are typically highly hetergeneous.
- Cleanse, enrich, transform and aggregate the source data into trustworthy data that can address the needs of a diverse set of consumers.
- Serve the datasets to a variety of consumers with a diverse set of needs. In the context of an industry 4.0 for example this would mean that data collected can be served to collect statistics, to enhance employer productivity by targeting the informations rather than to enhance controls and security or for organizational purposes.

It's indeed common practice that the monolithic architecture hosts data that semantically belong to different domains: a centralized data platform with no clear domain boundaries.
This centralized model can work for organizations that have a simpler domain with smaller number of different comsumption cases, but will likely fail for applications with rich domains, a large number of sources and a diverse set of consumers, where sources are those entities generating data (e.g. IoT devices, sensors, etc.) and consumers are the final user of such data.
If we look in deep in monolith architecture what we find is an architectural decomposition around the mechanical functions of ingestion, parsing, aggregation, serving, etc.
An approach can be to find a way to scale the system by breaking it down to its **architectural quanta** in order not only to enhance properties like scalability and high availability but also to allow a fast, modular and less painful growth of the system.

An architectural quantum is an independently deployable component with high functional cohesion, which includes all the structural elements required for the system to function properly.
The motivation behind breaking a system down into its architectural quantum is to create independent teams who can each build and operate an architectural quantum and allow a parallelized work across these teams to reach out better operational scalability.
In fact, in a monolithic scenario the development teams needed to communicate and work together along with those responsible for testing and deploying. In the context of monolith application, in fact, they were built following a layered approach. This led to organize the developer teams around a specific layer of the

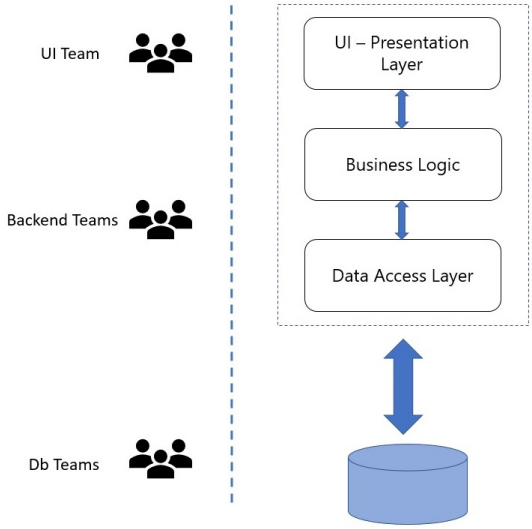


Figure 1.1: Typical Monolith Application's Architecture

software implying that when some change had to be done in some part of the application, this change would have affected the whole application due to high intrinsic level of **connascence** [2].
It's remarkable that Teams were built around skillset base and communication

[2][4] two components are connascent if a change in one would require the other to be modified in order to maintain the overall correctness of the system. In addition to allowing categorization of dependency relationships, connascence also provides a system for comparing different types of dependency. Such comparisons between potential designs can often hint at ways to improve the quality of the software.
Reducing connascence will reduce the cost of change for a software system. One way of reducing connascence is by transforming strong forms of connascence into weaker forms moving away from tightly coupled systems.

among them is not always easy and effective, so teams found out that gaining the ability to build and deploy their applications themselves, in development, test, and production environments was not only faster, but errors was less likely since a lot of errors deriving from environmental inconsistencies were eliminated.
Given the influence of previous generations of data platforms architecture, architects decompose the data platform to a pipeline of data processing stages. A pipeline that implements a functional cohesion around the technical implementation of data processing (ingestion, preparation, parsing, aggregation, serving).
[5] If we look at the Martin Fowler's Microservice definition we can find the reasons to decide for a Microservice Architecture in our specific domain.
In fact he points out some of the issues highlighted above:

1. **Organize around Business Capabilities**:
   If we look at the 1.2, we can notice that a monolith architecture follows pretty much the Teams organization structure, a good example of Conway's Law:
   Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.
   – Melvyn Conway, 1967
   What Martin Fowler proposed is a different approach: Do not organize the architecture around the organization structure: organize the architecture around Business Capabilities. We are now going to build self-consistent broad-stack services, which implies building cross-functional teams with a full range of skillset: from user-experience, databases, project management and so on. Certainly is true that also monolithic can be built around Business capabilities, but in most cases is very difficult for the teams to have clear boundaries for their concerns. The explicit separation of concerns that a Microservice Architecture offers help teams to keep those boundaries clear.

2. **Compose via Service**:
   It's a useful concept divide our application in Components as functional unit independently replaceable and upgradeable. One kind of component is a library, which tends to run in the same memory address as the calling code and communicates via language function call mechanisms. Libraries are usually compile-time dependencies thus static.
   The other type of component, called a service, runs in its own address space and communicates either through low-level protocols like TCP/IP or higher-level ones like SOAP or REST, making these runtime dependencies.
   While the components of a monolith are typically libraries because usually data flows through method calls, the components of a microservice architecture are services because they are indepentently deployable and because often it's only documentation and discipline that prevents that concerns of different areas overlap, leading to overly-tight coupling among components.

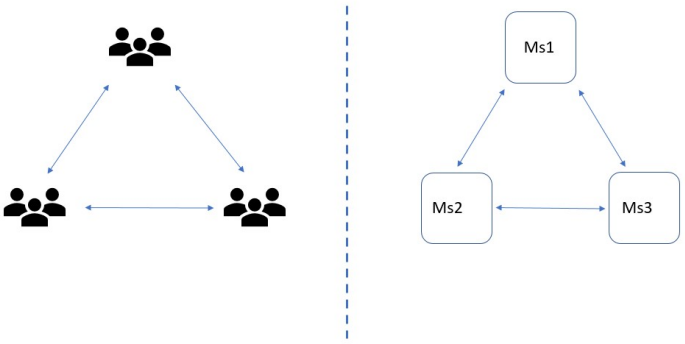


Figure 1.2: Microservice Application's Architecture

Services make it easier to avoid this by using **explicit** remote call mechanisms and by lesseninng the connascence of the system. In our use case we identified such runtime components as processing units with high functional cohesion: in the context of pipelines the stages are quite good candidate.

3. **Smart endpoints and dumb pipes**: While building communication mechanisms there are approaches like SOAP which tend to have heavy communication protocol built typically on top of HTTP and reponsible for complex operations. The approach proposed for microservices is to keep communication lightweight. The idea behind microservices is that each service receives some request, applies some logic and produces a result demanding cross-cutting concerns to infrastructure servers. The communication is performed typically in two ways:

   - RESTful approach: through HTTP requests and response in a synchronous way
   - Event-Driven approach: by means of Message Queues or Topic as asynchronous system to deliver messages.

   All the logic is thus delegated within the services leaving the communication mechanism free of it. With respect to a monolith application the communication pattern must be redesigned though. In our case we chosen an Event-Driven approach, using distributed queues to fit in a Producer/Consumer scenario.

1.4 Goals to achieve

The system needs to be thought to respond to the most common critical issues in distributed architectures.
Such properties are transversal and impact both on the Architectural Design and Deployment.

1. **Scalability**
   Is the property of the system to react to a change of the traffic load in an elastic way.
   We can respond to the load variation in two ways:

   - *Horizontal Scale* or *Scale out*: by adding logical resources to the system (e.g. another server on which the traffic can be spread)
   - *Vertical scale* or *Scale up*: by increasing the physical resources of the machines hosting the service (e.g. incrementing RAM or migrating the system to a more powerful machine).

   Horizontal Scalability is certainly the property impacting most on the Architectural design.
   Of course we can scale the entire system but it brings to waste resources because can happen that the traffic load impacts only over one part of the system: scaling the whole application is a waste of resources and this reflects on the costs. Moreover with monolith we don't have database separation, it means that all the applications will access a single database, ending up to a bottleneck since caching mechanisms are less effective and I/O traffic can result a problem. A Microservice Architecture can be designed to scale on different granularity. Identifying the critical functional units to scale and implementing the architecture accordingly, optimizes the system behaviour when some part of the application is stressed out. A microservice architecture can optimize even the Vertical Scaling: Some part of the system can be more memory greedy or more CPU intensive than other parts.
   The separation of concerns enables the possibility to dedicate specific physical resources on different part of the application.
   The Scalability affects also the Deployment process. In fact adding resources both physical or logical can be an hard task. With use of virtualization technologies like Docker this is simplified, indeed we can add easily logical resources by adding a container and we can bind it physical resources basing on specific needs, limited of course by the machine's power.

2. **Maintainability**
   It is maybe the most crucial part of a software lifecycle. Once the application

is sent to the market, it's time to keep it up and continuously update it either for bug fixing and feature enhancement. Since is typically the longest phase, and then the most expensive, we have to take organizational decisions to reduce this complexity and ease the maintenance process. We can achieve this by leveraging either on Architectural Design, on Modularity and on Deploy technology. Microservice Architecture helps this phase: due to loosely coupling and separation of concerns of the architectural components, the developer teams can be structured to work with one specific component of the system. Since each component has typically a reduced complexity with respect to the whole system, it fits perfectly with agile methodologies and so now we are encouraged to build teams around specific and self-consistent part of the application, instead of letting teams work around a specific layer like in monolith n-tier architectures where teams were strictly coupled and aggregated on skillset base with all the difficulties the teams had to face to work together. Another advantage in using these architectures is that the system can be updated without tear it down, in fact updates can come along by simply adding the new version of one part of the system while the others could not be updated yet. This is possible due to the loosely coupling and isolation of concerns properties. From a different point of view also Deployment choices help to reduce maintenance costs: the use of a virtualization technology allows to have different operating systems on a single machine. This means to reduce the need to have additional hardware resources eliminating all the costs it brings with it such as electricity, space, and hardware maintenance.

3. **Reliability and High Availability**
   High Availability is the property of a system that in presence of faults, independently from their nature, doesn't go down ensuring a certain level of operational performance. Some applications like the *Mission Critical* or *Business Critical* are forced to ensure high availability because of their sensitive nature. Typically High Availability is ensured by eliminating all Single Point Of failures. This is done by adding redundancy, so even if a component goes down the system is still alive. Certainly Microservice Architectures help us to ensure availability through the modularization of the application but it's not enough. In truth redundancy is a slightly different concept from scalability: while scalability is implemented to respond to load variation, thus on the application logic, redundancy is done to respond to failures. For example typically an application works with one database instance: we can of course scale out the database and work with Database's cluster, but if it's not strictly required, this may add a lot of complexity. Here the Database is a single point of failure since the entire application is not able to keep working without it. We need a backup database that takes it over when the primary one crashes or

becomes unavailable. It's common that in redundant and distributed systems are present cross points like Load Balancer or infrastructure server as Service Discovery, Gateway and Configuration Server that often are single point of failure. Introducing redundancy on these points makes our systems more robust ensuring high availability. Ensuring high availability doesn't mean only to be able to respond to failures but also to prevent them to happen. Virtualization technologies comes to our help again: they limit the unreliability of operating systems. If something goes wrong only that Container will be compromised and the host system will keep working regularly.

Chapter 2

State of the art

"Before software can be reusable,
it first has to be usable."
-Ralph Johnson

2.1 Technologies involved

2.1.1 Spring Framework

The Project has been developed using Spring.
Spring is an open-source Framework used for developing Java application.
The key feature making Spring interesting is the infrastructural support at application level, this lets the developers focus only on the application-level business logic delegating other aspects to the Framework.
The Spring Framework provides a lot of feature, well-organized in six modules:

1. **IoC Core**
   It's the hearth of Spring Framework.
   It relies on the concept of Bean which is a wrapper of Java Objects and is managed by the framework so that they can be passed when needed to the rest of the application.
   The main task of the core is to provide an implementation of the Inversion of Control also known as Dependency Injection. IoC is the process whereby objects define their configuration and their dependecies which are the other objects it works with.
   Hence come the name Inversion of Control, in fact are now the beans themselves that manage their own dependencies instantiating and initializing other objects they work with.
   So what does actually do the IoC core is consuming both Business POJO Objects defined by developers and metadata configuration files that instruct Spring container how to instantiate, configure, and assemble the objects in your application.
   Eventually we will have a fully configured system ready to use.

2. **DAO**
   The Data Access Object Support module in Spring aims to simplify the use of data access technologies such JDBC, Hibernate or JDO.
   This module introduces an abstraction layer allowing developers not to worry about specific implementations details of each technology, delegating this job to the framework and making easier switching from one technology to another.

3. **ORM**
   The Object Relational Mapping module in Spring, aims to integrate the support for and abstraction layer responsible to perform CRUD operations from a Object Oriented enviroment in a relational-fashion.
   It's an abstraction layer that manages translations between POJO and relational data.

4. **AOP** Another key component of Spring network is the Aspect Oriented Programming. It's remarkable that IoC core does not depend on Aspects, so its usage is optional.
   AOP provides another way of thinking about the program structure.
   Aspect means a modularization of concerns that cut across multiple classes allowing the developers to manage cross-cutting-concerns in a single point instead to have to spread it over the whole application.

5. **Web** Spring Web packages provide basic web-oriented integration feature such that the initialization of servlet listeners and web-oriented application contenxt in the IoC core.
   They provide also a package for Model-View-Controller implementation for web-application

What Spring basically does thus is that lets you assemble your code together.
The popularity of Spring Framework comes with inclusion of huge number of features and the framework became a lighter weight alternative for enterprise application Java Developers looking a way for building application using the J2EE stack, but avoiding unnecessary and silly mechanism J2EE stack has embedded.
Spring Framework tried to follow the developer tendency to moving away from heavy monolith applications towards a higher distributed model where applications are built around a potentially high number of small services that can easily deployed.
Spring Developer Teams reacted to this shift launching two ambitious projects:

1. **Spring Boot**: it is a revision of the Spring project, it embraces core features depicted above, but cuts out many enterprise features and deliver a lighter REST-based, microservice-oriented Framework.

2. **Spring Cloud**: it aims to drive more the Spring Framework towards Microservices, trying to make simpler to operationalize and deploy Microservice both in public and private cloud infrastructures.

2.1.2 Jhipster-Registry

In a Microservice Architecture the Registry is a fundamental piece: it ties all the components together and allows them to communicate each other.
The Jhipster-Registry is a Open Source, Apache 2-licensed application developed by the Jhipster team.
Jhipster Registry has basically three main purposes. We will see more aboute them later, in a nutshell we have:

1. **Eureka Server**: used as Service Discovery

2. **Spring Cloud Configuration Server**: used as Configuration Server

3. **Administration Server**: integrate a dashboard to monitor and manage applications present in the enviroment.

2.1.3 MongoDB

MongoDb is a general purpose distributed database.
It has been designed for modern applications and optimized for the cloud era.
MongoDb is a Document based Database, which means it stores and handles data in JSON format and allows flexible and dynamic schemas. It does not only allow typical Relational Database features like ACID operations and support for join query, but it also combines the ability to scale out with features such as

1. **Indexing**: MongoDB has support for generic secondary indexes, allowing fast queries and providing unique, compound, geospatial, and full-text indexing capabilities as well.

2. **Aggregation**: MongoDB supports an "aggregation pipeline" that allows you to build complex query from simple ones and allow the database to optimize it.

3. **Special collection types**: MongoDB supports time-to-live collections for data that should expire at a certain time, such as sessions. It also supports fixed-size collections, which are useful for holding recent data, such as logs.

The document-based approach replaces the concept of relations (thus rows and columns) with a more flexible model called Document. There are no predefined schemas, keys and values have not fixed size or type, so working with such flexibility ease the software development helping programmers to model their applications.

2.1.4 Hazelcast

Hazelcast is a distributed cache in-memory platform written in Java. The platform allows to distribute data along a cluster of servers in a secure and consistent way.
An Hazelcast cluster is a network of cluster members that run Hazelcast.
Cluster members automatically join together to form a cluster. This automatic joining takes place with various discovery mechanisms that the cluster members use to find each other. We will rely on the discovery through Eureka Service Discovery.
The use of in-memory and streaming technologies are a necessity for microservices architectures in order to design scalable, modular and easy-to-mantain systems.
In the context of microservices Hazelcast helps to face crucial challenges as high performance and efficient inter-service communications while offering a bunch of features to respond the needs of modern distributed enviroments.
Hazelcast proved to be well suited for orchestrating a number of microservices

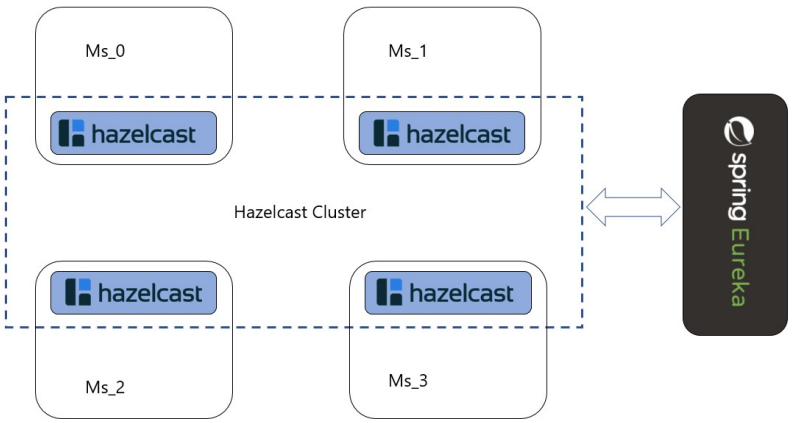


Figure 2.1: Hazelcast Cluster

in a load balanced environment that do not share any resources other than the exposed data structures through Hazelcast.

CPsubsystem

Strong consistency is a crucial requirement for coordination tasks. A distributed coordination tool must keep its consistency in failure cases. However, Hazelcast is mainly designed for high availability. In this regard, Hazelcast gracefully handles server, client, and network failures to maintain availability, but its core replication mechanisms, which are basically built on top of asynchronous replication, do not guarantee strong consistency in every failure scenario. In terms of the CAP principle, Hazelcast is AP thus prefer Availabilty and Partition Tolerance to Consistency.

CAP Principle:
[6] It is impossible for a distributed data store to simultaneously provide more than two out of the following three guarantees:

1. **Consistency**: Every read receives the most recent write or an error.

2. **Availability**: Every request receives a (non-error) response, without the guarantee that it contains the most recent write.

3. **Partition tolerance**: The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes

CPSubsystem contains new implementations of Hazelcast's concurrency APIs on top of the Raft consensus algorithm. As the name of the module suggests, these implementations ensure Consistency and Partition Tolerance with respect to the CAP principle and they live alongside AP data structures in the same Hazelcast cluster, offering consistency and allowing distributed version of java concurrency API such as IAtomicLong. IAtomicReference, ILock, ISemaphore, ICountDownLatch. With the CP subsystem Hazelcast will likely become a good citizen of distributed coordination use cases.

2.1.5 Kafka

[7] Kafka is a stream-processing software platform. The plarform aims to provide a unified, high-throughput, low-latency platform for managing real-time data through a messaging system.
A streaming platform has three key responsabilities:

- Publish and subscribe to streams of records, similar to a message queue or enterprise messaging system.
- Store streams of records in a fault-tolerant durable way.
- Process streams of records as they occur.

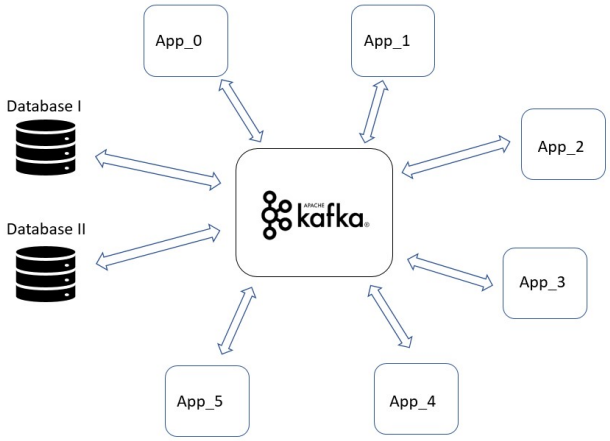


Figure 2.2: Kafka based Application

Kafka offers many features one of the most attractive for our purposes is certainly

its usage as `Message Broker`[1].
Kafka is strongly suggested for such applications that needs to building real-time streaming data pipelines that reliably get data between systems or applications and building real-time streaming applications that transform or react to the streams of data. Kafka ensures strong guarantees as scalability, high availability, persistence as well as low latency and high throughput. Kafka is run as a cluster which instances can run both locally or remotely spanning on multiple datacenter. The Kafka cluster stores streams of records in categories called `Topics`.

2.1.6 Docker

Docker is an open-source virtualization technology used to automate the deployment process. Docker exploits kernel virtualization features to cope with challenges as Dependency hell and versioning by running the application in a light and isolated enviroment. Docker thus provides a platform for developing, shipping and running applications in isolated enviroment called *Container* by exploiting Linux kernel resource isolation properties as *cgroups* and *namespaces*. The container concept not only ensures an isolated enviroment allowing us to run multiple containers on the same host safely but is also lightweight because does not have all the extra load of having an abstraction layer of the virtualization technologies like the hypervisor. The massive usage of virtualization technologies in deployment is not by chance and nowadays is the state-of-art because are able to respond to many common problems in Deployment proposing simple and effective solutions.

[1] A message broker is an architectural pattern for message validation, transformation, and routing. It mediates communication among applications, minimizing the mutual awareness that applications should have of each other in order to be able to exchange messages, effectively implementing decoupling

Chapter 3

Project Design

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."
— Martin Fowler

3.1 Software Purpose

The application aims to parse, classify, decorate, elaborate and eventually aggregate semi-structured data.
This comes from the need not only to allow a future elaboration but also to visualize them on a human readable interface like the one of a client application.
The Architecture is presented as a Pipeline made up by 3 stages: an **Ingester** responsible for fetching data and perform a primitive parsing, a **Parser** responsible to read inside the data, structure and aggregate them basing on their metadata and finally an **Aggregator** responsible for aggregating on a semantic base our well-structured data.
To exploit the advantages of a distributed architecture discussed in the previous chapters accordingly to the goal to achieve, it has been chosen a Microservice Architecture.
The Architecture is presented as a pipeline where each stage is a microservice. The communication between microservices is performed by means of a distributed cache called Hazelcast. Hazelcast is able to clusterize the application allowing the microservices to share a set of data structures used both for communication and synchronization purposes. The communication mechanism uses distributed queues managed by Hazelcast to let flow the data from one pipeline's stage to the next addressing the architecture towards an Event-Driven Architecture.

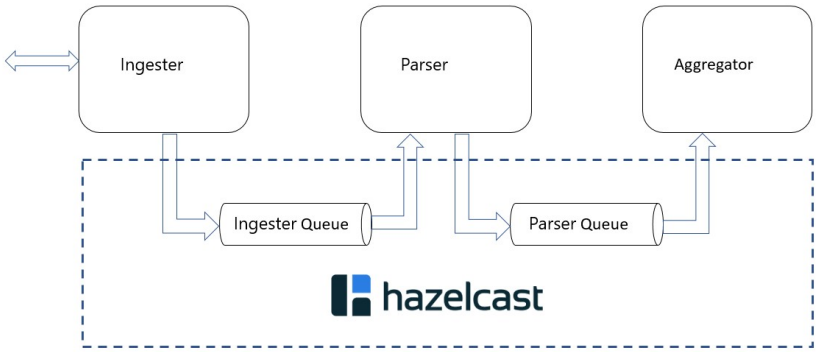


Figure 3.1: Pipeline's Architecture

3.1.1 Infrastructure Servers

The Database

The Database used is MongoDB: a cross platform database NoSQL using document Json-Like format with schema.
The Architecture involves the use of 3 Databases, one for each service composing the application.
Since there are multimedia files involved in the dataset, it will be plausible that files will exceed the maximum data size allowed by `MongoDB`, so, we will rely on `GridFS`. [8] Instead of storing a file in a single document, GridFS divides the file into parts, or chunks , and stores each chunk as a separate document.
By default, GridFS uses a default chunk size of 255 kB; that is, GridFS divides a file into chunks of 255 kB with the exception of the last chunk. The last chunk is only as large as necessary.
Similarly, files that are no larger than the chunk size only have a final chunk, using only as much space as needed plus some additional metadata.
GridFS uses two collections to store file:

1. `fs.chunk`: stores the file chunks
2. `fs.files`: stores file metadata.

When you query GridFS for a file, the driver will reassemble the chunks as needed. You can perform range queries on files stored through GridFS.
You can also access information from arbitrary sections of files, such as to "skip" to the middle of a video or audio file.
GridFS is useful not only for storing files that exceed 16 MB but also for storing any files for which you want access without having to load the entire file into memory.
So, we will end up to have:

1. **Ingester Database** : The database that will serve the Ingester microservices. it will holds the 3 collections:
   - `RawEntity`: holding metadata
   - `fs.files`: holding gridFS related metadata
   - `fs.chunk`: holding the data
2. **Parser Database** The database that will serve the Parser microservices. it will holds the 4 collections:
   - `Entity`: holding metadata of Entity
   - `Item`: holding metadata of Item

- `fs.files`: holding gridFS related metadata both of Entity and Item
- `fs.chunk`: holding the data both of Entity and Item

3. **Aggregator Database** The database that will serve the Parser microservices. it will holds the 4 collections:
   - `Item`: holding metadata of Item
   - `Blob`: holding metadata of Blob
   - `fs.files`: holding gridFS related metadata both of Item and Blob
   - `fs.chunk`: holding the data both of Item and Blob

Service Discovery

The service Discovery is a service in our enviroment that allows server instances to get to know each other throughout a registry that maps service names and ip addresses.
When a microservice is interested in contacting another microservice in the enviroment asks to `Eureka Server` the location of the recipient.
The service discovery keeps update the entries thanks to `Eureka Clients` embedded in each microservice in the enviroment.
Eureka Clients are not only responsible to register themselves in the `Eureka Server`, but are also responsible for sending an healthcheck periodically, so that the Service Discovery is aware of the crashed services and can perform a clean up of the entries. In our application, we'll use Service Discovery to allow Hazelcast nodes to discover each other, in order to join in a cluster.

Configuration Server

The Configuration server is a server that centralizes all the configuration files and responsible to serve the instances when the application boots up.
The Configuration server can be configured either to store the configuration files in a remote repository as GitHub or either in the local file system.
In a microservice architecture, centralize all the configuration files is a very important step to ease maintainability and fasten the configuration process of the entire application. In our case we rely on Spring Cloud framework that offer a very powerful tool called `Spring Cloud Config Server`.
In order to handle as less infrastructure server as possible, the solution proposed uses Jhipster Registry that embeds both `Eureka Service Discovery` and `Spring Cloud Config` in a single deployable unit.

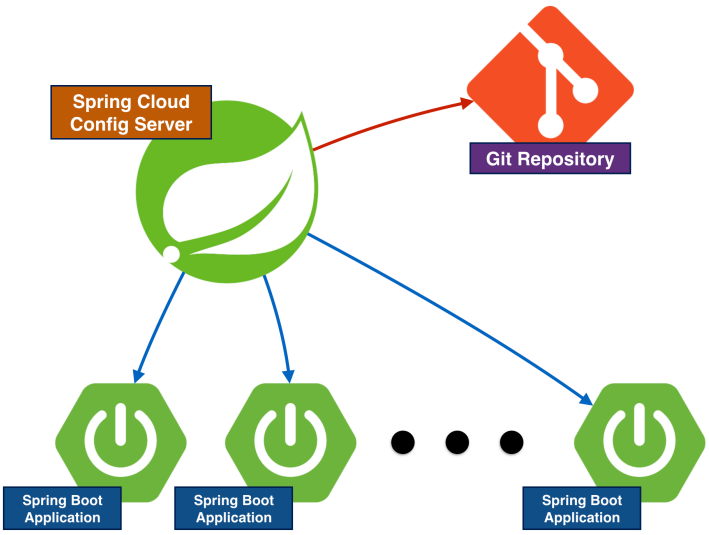


Figure 3.2: Configuration Server

3.1.2 The Data

We can imagine we are treating data coming from large variety of sources like IoT devices deployed around a smart city rather than in a university campus or data coming from sensors of a smart industry. The data source with respect to our system is an SFTP Server on which we will end up to have a folders structure with a root.
Under the root folder can be present a variable number of subfolders that appear dinamically, each of them representing the device that has generated the data contained by the folder itself, thus each folder will be named as the `<Agent_Id>` it refers to.
Due to the large variety of devices, we will have an highly heterogeneous dataset which is formed by actual data and metadata. The data formats we are going to deal with are composed by:

- `Json`: contains data or metadata associated to Zip or Bz2 Files.
- `Xml`: is self-consistent. Contains both metadata ad and data.
- `Bz2`: contains only data, typically multimedia ones.

It's remarkable that data, as well as metadata, can come along fragmented, out-of-order and with any insurance that a fragment of data I'm waiting for, will eventually arrive.

3.2 Ingester

The first stage of our pipeline is called Ingester. The Ingester is responsible for fetching data from a source: in our case an SFTP server.

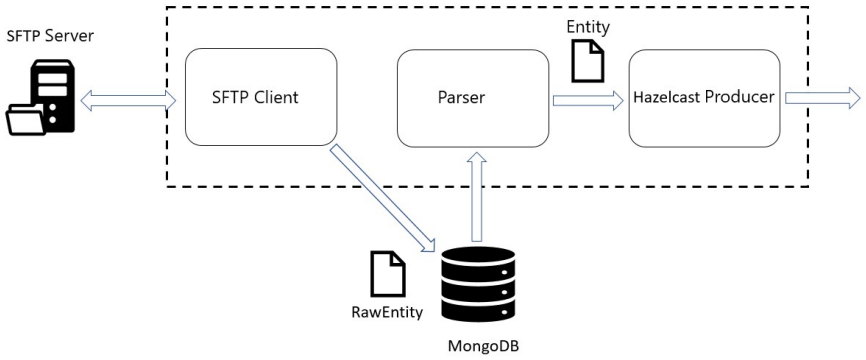


Figure 3.3: Architecture of Ingester Microservice

3.2.1 SFTP Client

In order to fetch the data we will have the first module of the chain: an SFTP Client. This is not a traditional Client, in fact it acts as a Consumer in a Producer/Consumer scenario.
Its role will be to continuously poll the Server to get to know if:

- some files are added under the known folders
- some new folder is present under the root one.

For efficiency purposes we will poll different folders with different threads that we'll call `Watchers`. The parent thread instead will be responsible for polling the root looking for new folders added. We'll call the parent thread `Poller`. The Poller assigns the folders to poll the newly created watchers, so they can check whether there are new files to fetch.
If some new file is added, gets downloaded by a watcher. Each and every file fetched from the server is first structured as a model called RawEntity for being stored into MongoDB. No problem so far. The first issue we can notice is that, since we are

running in multithreading, we should synchronize our threads to prevent some file to be download multiple times. It might seem unnecessary but the network can be a bottleneck, so optimize its usage is fundamental for this application. Synchronize

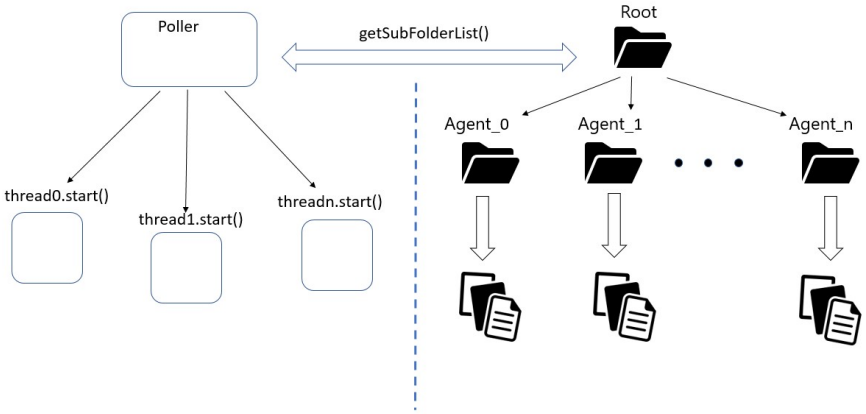


Figure 3.4: Ftp Client's execution model

all the threads is not trivial: we are building a distributed architecture, so we will have multiple instance of Ingester running at the same time, likely on different machines. It's not enough to synchronize the local threads, but we need to do it also with respect to the remote ones, but we will discuss about this later.

3.2.2 The Model: RawEntity

The RawEntity model is represented with a variety of fields, some of them bring informations about its nature, some other is initialized and used by the application for processing purposes. The main fields are:

- `Id` : Unique Id, written by MongoDB
- `content_type`: extension of the file
- `path`: the folder's path where the file has been stored
- `file_name`: the filename
- `metadata`: if it's a file of metadata, is not null
- `payload`: if it's a payload, is not null

- `evidences`: the list of Evidence's models generated from this RawEntity
- `payload_file_name`: if it's a metadata file, tells us how it's called the associated file of data.

3.2.3 Parser Engine

The parser is the functional unit responsible for looking inside the data and actually transform the model from `RawEntity` into an `Entity`. The first thing is to fetch the data from MongoDb.
To keep track of elaboration steps each model has a field called `STATUS`. The parser fetches data having status set to `READY`.
In order to ensure that elaboration cannot happen twice o multiple times on the same data as this can lead to inconsistent situations, we have to synchronize each and every thread in the system that run concurrently to take such data from the Database.
Unfortunately we already said that in our architecture we will have multiple instances of Ingester so a classical in-memory synchronization will not be enough though. We will discuss about this later on.
For now let's focus on the single instance running. At this point the elaboration can end up in three main cases:

1. **The elaboration succeed**: the RawEntity processed is stored in the Database with status `PARSED`, a new Entity model is generated and forwarded to the next stage of the elaboration piepeline.
2. **The Data is malformed**: this will produce an error since is not possible to generate an Entity model from this data, so the RawEntity is store in the DB with Status `ERROR` and it's not propagated through the pipeline.
3. **The current data needs some other RawEntity** to generate the Entity model: The RawEntity's status is set to `POSTPONED` and it will be processed further on.

Once the elaboration succeed, the new Entity is generated and is ready to be wrapped, serialized and pushed into the `ingester_queue`.

3.2.4 Hazelcast Producer

The task to prepare and actually forward the data is demanded to the last module in the chain: the `HazelcastProducer`. This module acts as a Producer with respect to the `ingester_queue`. The `ingester_queue` is indeed a classical Java `BlockingQueue` extended by Hazelcast to ensure it can be distributed among all

the Hazelcast Cluster nodes and to make it redundant for fault tolerance reasons. The advantage of this approach is that we are actually decoupling the services, so that a Producer doesn't know who is going to fetch the data from the queue, it only knows that there is a certain number of consumers responsible to manage the data stored into the queue. Another advantage comes free with this topology: in fact if several microservices crash, that doesn't affect the queue since it is distributed and backed up over all the nodes in the cluster. To ensure the queue stays alive is sufficient that at least one Hazelcast Instance is up. If the system is balanced, so

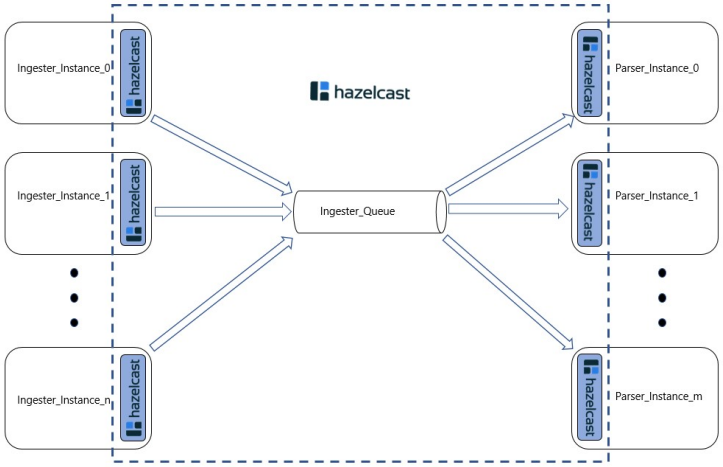


Figure 3.5: Event-Driven model with Hazelcast Cluster

when the producing rate is equal to the consuming rate, the queue doesn't grow in size.
When dealing with real situations the producing rate is asymmetric with respect to the consuming rate, for example if the Consumers are cut out from the cluster or experience network latency, then is possible that the queue may grow out of control.
To address this issue has been implemented a backpressure mechanism that introduces a latency at producer side that depends on the `ingester_queue` size. As the queue grows up, the latency gets bigger, as the queue gets shorter, the latency decreses. If the queue gets full, the producer stops pushing data into it.

3.2.5 Ingester: Hazelcast synchronization

We already introduced that, due to the need to scale the pipeline stages, we are working with multiple instances of Ingester (as well as multiple instances of each pipeline's stage) and since each stage relies on the same Database for

performance reasons, it is necessary to synchronize the microservices to let them work concurrently. We are already using inter-process communication mechanism with the Hazelcast Queue, we can thus exploit Hazelcast also to synchronize the microservices.
We have to synchronize basically two modules:

- The consumer: in this case the SFTPClient
- The parser engine

What we need for the synchronization here is to prevent multiple threads in this distributed enviroment to perform the same operations at the same time: we need to protect our code from ditributed multithreading. With such guarantee we can synchronize our Ingesters.

For what concerning the consumer synchronization we must avoid that different SFTPClient will download the same file. We rely on Hazelcast Distributed Map which is nothing else than a normal Java `HashMap` shared between the ingester instances: we'll call this map `ingesterSyncMap` and we will use it to keep track which files are being processed by the different Ingester instances. The `ingesterSyncMap` is structured as follows:

- `Key`: filename of the file that has been locked.
- `Value`: owner's Id of this lock and a timeStamp.
  The Id identifies who is dealing with such file.
  The TimeStamp is an expiring date for the fault recovery. In case the owner of the lock goes down for some reason while processing the data, such file will not be locked forever.

Hazelcast distributed maps are thread safe objects. They offer lock mechanism that can deal with typical race condition when multiple threads share the same resources. Here we will make use of Pessimistic lock mechanism, the usual way to solve this race issue is using the lock mechanism provided by Hazelcast distributed map: `map.lock` and `map.unlock` methods. You simply lock the entry until you finished with it.

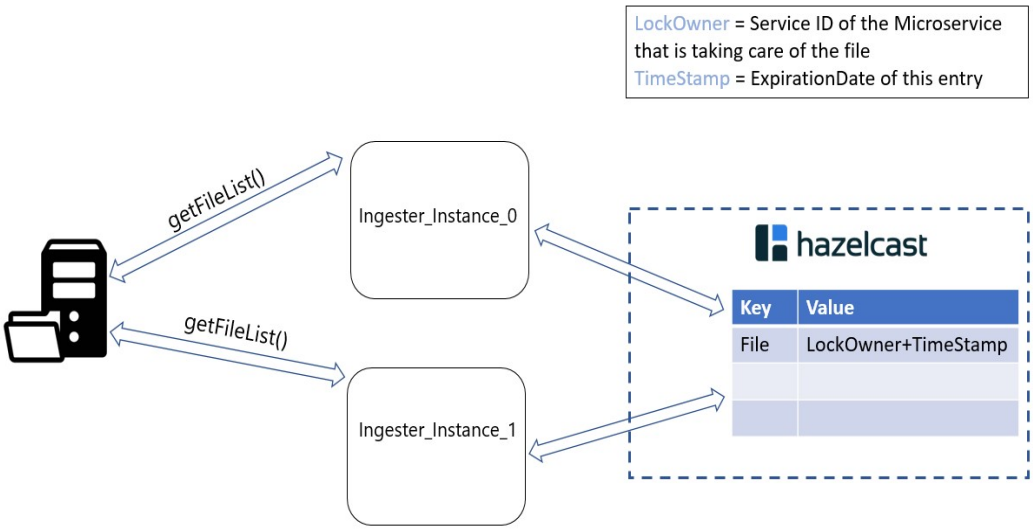


Figure 3.6: Schema for the SFTPClient synchronization

```
IMap<String, HazelcastRecord> syncMap = hazelcastInstance
                                        .getMap("ingesterSyncMap");
syncMap.lock(filePath);
if(checkIfLockExists(filePath)) {
    syncMap.unlock(filePath);
    return false;
}
syncMap.put(filePath, hazelcastRecord);
syncMap.unlock(filePath);
```

So the SFTPClient, before downloading a file, as shown in 3.6:

1. it tries to acquire the lock mechanism provided by Hazelcast distributed map and it waits until it gains the lock.
2. it looks at the `ingesterSyncMap` and checks whether some entry with a `Key` equals to our current file to download exists.
3. if it exists, means that this file is being downloaded by some other SFTPClient instance, if does not exist it writes the record in the map.

4. it unlocks the map entry.

For what concerns the parser engine instead, we must ensure that different `Parser` do not fetch the same data from the Database concurrently.
if this happens, is possible that two identical `Entity` will be generated and forwarded to the next stage. This is not only inefficient, but potentially dangerous as we'll end up to have duplicated files that will eventually be treated as different ones, breaking the system.
We must guarantee that different threads fetch and update data atomically: we need a distributed synchronization primitive.
Hazelcast technology, as already said, offers mechanisms to do atomic operations in a distributed enviroment relying on the CPsubsystem.
Hazelcast offers us an object called `FencedLock` for such purposes. Distributed locks are unfortunately not equivalent to single-node mutexes because of the complexities in distributed systems, such as uncertain communication patterns and independent process failures.
[9] In an asynchronous network, no lock service can guarantee mutual exclusion, because there is no way to distinguish between a slow and a crashed process.
Consider the scenario where a Hazelcast client acquires a FencedLock, then hits a long GC pause. Since it will not be able to commit session heartbeats while paused, its CP session will be eventually closed. After this moment, another Hazelcast client can acquire this lock. If the first client wakes up again, it may not immediately notice that it has lost ownership of the lock. In this case, multiple clients think they hold the lock. If they attempt to perform an operation on a shared resource, they can break the system. To prevent such situations, one may choose to use an infinite session timeout, but this time probably she is going to deal with liveliness issues. Even if the first client crashes, requests sent by two clients can be re-ordered in the network and hit the external resource in reverse order. There is a simple solution for this problem. Lock holders are ordered by a monotonic fencing token, which increments each time the lock is assigned to a new owner. This fencing token can be passed to external services or resources to ensure sequential execution of side effects performed by lock holders. We have to ensure some lock mechanism, to distinguish which data are available for the elaboration and which data are being processed.
In principle the Ingester's parser engine fetches all the data with status set as `READY`. It would be enough to atomically set the status of the database's entities that are going to be processed from `READY` to `PROCESSING` to ensure each data will be processed only once, as shown in 3.7.
After the processing the status will be updated and will fall in one of the status we discussed above.

Figure 3.7: When Ingesters fetch the data let others know that those data don't have to be processed by changing them the `status`

```
hazelcastLockingService.lock(); // myFencedLock.lock();
result = rawEntityService
        .findAllByStatus(RawEntityStatus.READY);
for (RawEntity e: result ) {
    rawEntityService.updateById(e.getId(), rEntity -> {
        rEntity.setStatus(RawEntityStatus.PROCESSING);
    });
}
hazelcastLockingService.unlock(); //myFencedLock.unlock();
```

3.2.6 Fallback

Our architecture should be designed to be scalable, high available, easy to maintain and fault tolerant. To be robust to failures is fundamental having some mechanism to recover when bad things happen. The synchronization mechanisms we discussed about, introduced in the system a couple of criticalities: let's try to analyze the following scenarios:

- What would happen if one Ingester will go down while has some entries in the `ingesterSyncMap` ?
- What would happen if the server goes down when processing is not finished yet?

In both of situations we have "locked" some data and those will stay "locked" forever, in the first case they are the data living in the SFTP Server, in the second case the data "locked" are in the database. The issues have been addressed by introducing a function triggered periodically that try to figure out if some entry is deprecated. Every Ingester periodically will check if there is some entries in `ingesterSyncMap` older than a certain threshold and in case will remove them. This is the reason why every entry in the map has a Timestamp. For what concerning the database entities, they have a timestamp in the field `last_update`. We are going to exploit this information and follow the same argument: The state `PARSED` is set once the elaboration has been concluded, thus once the data has been already pushed into the queue. So if something goes wrong in one of the `Ingester` instances, this will imply that in the DB will be present entries with state `PROCESSING` (while they are marked as `PROCESSING` are not involved in any new elaboration) even though such entries are not under elaboration. Means that every entities with state `PROCESSING` and older than a certain threshold is probably coming from an inconsistent scenario. A periodic triggered function will look at such entities and will recover the state to `READY`, so they can be reprocessed further on.

3.3 Parser

The second stage of our pipeline is called Parser. This stage is responsible to look inside the `Entity` picked from the queue, finds its dependencies, stucture their data into a `Map<String, Object>`, then build a model called `Item` which can be composed either by only one Entity or by many of them depending on its nature. Inside the Parser Microservice, we can identify again three functional units:

1. **Consumer**: it interfaces with the `ingester_queue` and takes the data pushed into it.
2. **Parser Engine**: it is the most important part of the elaboration chain, it processes the data structuring them into the Database.
3. **Producer**: it interfaces with the `parser_queue` to forward the data to the next stage.

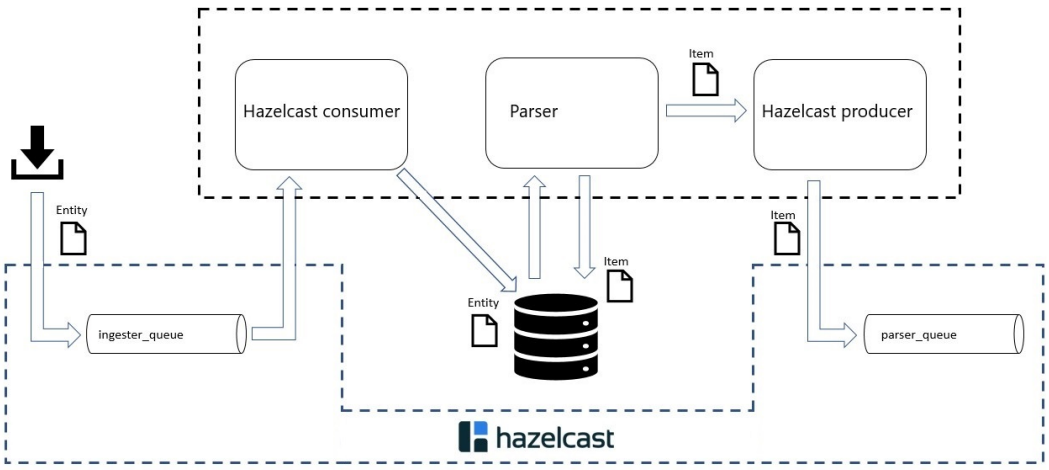


Figure 3.8: Parser Architecture

3.3.1 The Models

There are two main data model which this pipeline's stage works on.

Entity

This model is generated in the previous pipeline's stage, but is stored in this one. It is a model more consistent than `RawEntity`, in fact it brings informations about metadata and data concerning the same file, but while metadata are processed and put into a `HashMap<String, Object>`, the payload is still raw and comes along as `ByteArray`.
The main fields are:

- `id` : is the unique Id generated by MongoDB
- `createdAt`: is the creation timestamp of this Entity
- `agentId`: id of the device where this data comes from
- `contentType`: the data type present in the payload
- `items`: one Entity can be associated to one or more Items, this tracks the items generated by this entity
- `metadata`: the metadata as `HashMap<String, Object>`
- `payload`: the raw payload as `byte[]`

Item

It is an higher level model. The Item has a payload structured in a `HashMap<String,Object>`.
The metadata, from now on are not necessary as one Item can derive from many Entities and the metadata are associated to this lasts. It wouldn't make sense carrying them around.
The Item is designed as follows:

- `id`: : is the unique Id generated by MongoDB
- `blobId`: it's a reference to an higher level model called `blob` this Item will belong to.
- `payload`: is the payload that is now organized in an `HashMap<String, Object>`
- `chunks`

3.3.2 Hazelcast Consumer

This is the Consumer component of our pipeline's second stage. It will poll the `ingester_queue` to pick the data pushed into it.
Hazelcast `DistributedBlockingQueue` is a thread safe object: the data extracted from it are removed atomically, so it won't be possible for two parser instances to fetch the same data twice.
When the consumer picks the `Entity` from the queue, stores it into the Database and triggers the Parser module for processing the new data.

```
@Scheduled(fixedDelay = 300)
public void pickFromQueue(){
    try{
        Record record = hazelcastConsumerService.
   consumeHazelcastMessage();
        if(record != null){
            String tag = record.getKey();
            entityParserService.update(tag);
            Thread.sleep(100);
        }
    }catch (Exception e){
        e.printStackTrace();
    }
}
```

The Consumer is not aware of who's pushing the data into the queue, thanks to this decoupling characteristic, the system can scale very easily: we can add parser instances and this will be completely transparent with respect to the Producer side.

3.3.3 Parser Engine

As discussed above, this module is responsible to transform `Entity` into an higher level model called `Item`.
When the elaboration starts, the parser takes the entities having status `READY`.
Let's highlight that the `entityParserService.update(String tag)` takes one argument and this argument is a string containing:

- `Agent_id`
- `Content_type`

this because of the dependencies among data.
An Item can be generated by many `Entity`, this implies that in order to generate such Item, all the entities involved should be present before the parser starts the

elaboration.
If they are present only partially, will not be possible to generate a complete Item and the elaboration will be `POSTPONED` or depending on the case, is generated an incomplete Item that subsequently will be updated. For this reason, we store also the Items in this pipeline's stage.
The crucial requirement is that all the Entities having the same semantic aggregation key, which are just `Agent_Id` and `Content_type`, have to be processed together by the same thread. We are again in front of a synchronization issue.
We need to ensure that the Entities being elaborated by one Parser, shouldn't be treated by anybody else concurrently or this will end up generating inconsistent data, as for example two different Items semantically equals.
It is also possible that an Entity can concur to create many Items, so things can get even worse if we do not manage such situations.
Once the elaboration is completed the entity status is updated and we can be in one of the following scenarios:

1. **The elaboration succeed**: an entity can produce only one item and in this case the entity will be marked as `PARSED` or can concur to create an Item altogether with other Entity and in this case will be marked as `MERGED`

2. **The elaboration fails**: if something goes wrong then the the Entity is marked as `ERROR`

3. **Some Entity is missing**: then we are waiting for some data to arrive in order to generate the `Item/s`. The `Entity` is thus marked as `POSTPONED`.

The elaboration will then produce `Item` that will not be only forwarded but they will also stored. As mentioned, is possible that to create a complete Item might be required many Entities but we don't know if and when those will eventually arrive. We will start aggregating the Entities in Items even if we do not own all the pieces and we will wait for other pieces to close the incomplete Item in future if possible. The `Item` once generated is passed to the Hazelcast Producer to be forwarded to next pipeline's stage through a queue with same mechanisms described for the previous stage of the pipeline.

3.3.4 Parser: Hazelcast Synchronization

The synchronization between Parser instances are a bit trickier than it was with Ingesters. This because of the strong dependencies between `Entity` models that make mandatory to process them with the same thread.
What does it mean?
When the parser engine takes in charge some data to elaborate, it have to fetch the data Entity from the Database. But while with the Ingester was enough fetching

all data with status `READY`, now it will not work anymore. We should ensure that, given a key (in relational database meaning) composed by attributes `Agent_Id` + `Content_type`, the parser checks whether some other Parser has already taken in charge data with the same key and in case drop them.
Let's consider the following situation:
Some Entity with $\texttt{Key}_0$ = $\texttt{Agent\_Id}_0$ + $\texttt{Content\_Type}_0$ are pushed in the DB. Let's say that such key will identify `N` entities.
At $\texttt{t}_0$ there are `M` with `M<N` Entities stored in the Parser's Database and elaboration for Key_0 is taken charge by $\texttt{Parser\_Instance}_1$.
If $\texttt{Parser\_Instance}_1$ is allowed to parse such data, it would mean that **all** the data with Key_0 (In 3.9 entities in red) will have status `READY`.
When a thread decides that want to process a data, it locks it by setting its status as `PROCESSING`.
If one entity in the database has status `PROCESSING`, it would mean that such key is being elaborated by another thread in the enviroment and $\texttt{Parser\_Instance}_1$ is not allowed to process data with such key.
It's crucial for the parser to process all the Entity with same `Agent_id` and `Content_type` together. Let's assume that **all** the M instance with Key_0 are in status `READY`: then $\texttt{Parser\_Instance}_1$ is allowed to parse them. As the elaboration starts, the data M+1 with $\texttt{key}_0$ is stored in the DB.
How to prevent that this last entity will be involved in another parallel elaboration?

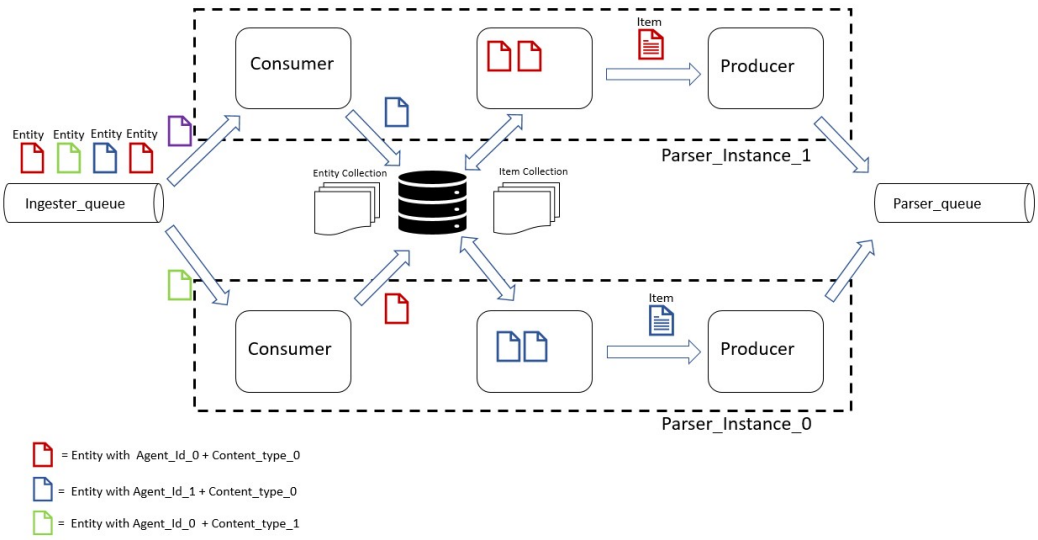


Figure 3.9: The schema represents the data flow through two instances of Parser

The solution presented follows the approach done with Ingester.
We still need to perform atomic operations on Database and ensure that when a thread is querying for data, the other threads wait. So we fetch all data having status `READY`, for each of them we evaluate their Keys for each key we check if there is some entity with state `PROCESSING` and then we filter out all of them. We will end up to have all the keys for which none of the relative entities have status `PROCESSING`. Then we set to those entities the status `PROCESSING` so that another thread will not be able to process data with Key_0 in parallel.

```
if(hazelcastLockingService.lock()){
    try{
        result = entityService.
                    findAllKeysNotHavingState(contentType,agentId,
                    EntityStatus.PROCESSING);

        for (Entity e: result ) {
            entityService.updateById(e.getId(),
            ev -> {
                ev.setStatus(EntityStatus.PROCESSING);
            });
        }
        hazelcastLockingService.unlock();
        return new PageImpl<Entity>(result, page, result.size());
    }catch (Exception e){
        hazelcastLockingService.unlock();
        return new PageImpl<Entity>(result, page, result.size());
    }
}else{
    return new PageImpl<Entity>(result, page, result.size());
}
```

3.3.5 Hazelcast Producer

Again the last functional module inside the single microservice is the one responsible to communicate with the next stage of the pipeline.
Again, this is done by means of Producer/Consumer mechanism through a distributed Java `BlockingQueue<Item>`.
This mechanism has been implemented for the communications between the pipeline stages as it helps us to achieve decoupling and scalability but also to standardize in somehow the communication mechanisms in the Architecture. In fact again, the the parser's producer module is quite similar to the previous stage's one but this time it handles different data. As soon as a new Item is created, it is delivered

to the Producer, responsible to push the data into the queue. This queue is not the same as the one between `Ingester` and `Parser`. This queue will be called `parser_queue` and stands between the `Parser` Microservice and the `Aggregator` Microservice.
It forwards Item models to the `Aggregator` where they will be further aggregated on a semantic base.

3.3.6 Fallback

Let's suppose that something in system goes wrong for any reason and one Parser Microservice goes down.
What will happen to the Application?
How to prevent that a failure affects the system's behaviour?
Let's assume that at a certain point, a Parser fails. Of course it will be scaled, so the system will keep working as usual, but the fail may introduce some inconsistency in the data. It's our duty enstablishing some mechanism to react to such failures. When a Parser goes down without any on-going elaboration, is a safe situation. When instead it goes down while processing data, then the data under elaboration are still marked with status `PROCESSING` and thus will never get to status `PARSED` or `MERGED`. A routine scheduled periodically is responsible to find data marked as `PROCESSING` with `last_update` older than a certain threshold and restore such status to `READY`. Of course can happen that indeed the data has been forwarded, and then the server goes down before changing the state to `PARSED`. In this scenario the data will be restored and reprocessed, but when the `Parser` will try to save the `Item` generated in the DB, this will not be allowed since would be a duplicated `Item` and then will not be forwarded again.

3.4 Aggregator

The aggregator is the last step of our data pipeline.
This stage is responsible for Item aggregation on a semantic criteria.
These criteria are not hardwired but can be configured throughout a configuration file called `aggregator.json` under `src/main/resource/config` folder. this configuration specifies the various aggregation criteria for each and every `content_type`. This file is parsed when the `Aggregator` service boots up and then the service can aggregate `Item` in order to create `Blob` models, nothing more than list of items grouped following semantic criteria. The structure of this service is essentially similar to the previous ones: there is an ingestion stage, an elaboration step but this time, being this the final stage we do not have to propagate the data to another step.
This service will interact with a client application to which the data will be served for displaying them, for this purpose this service will expose some REST Api. The

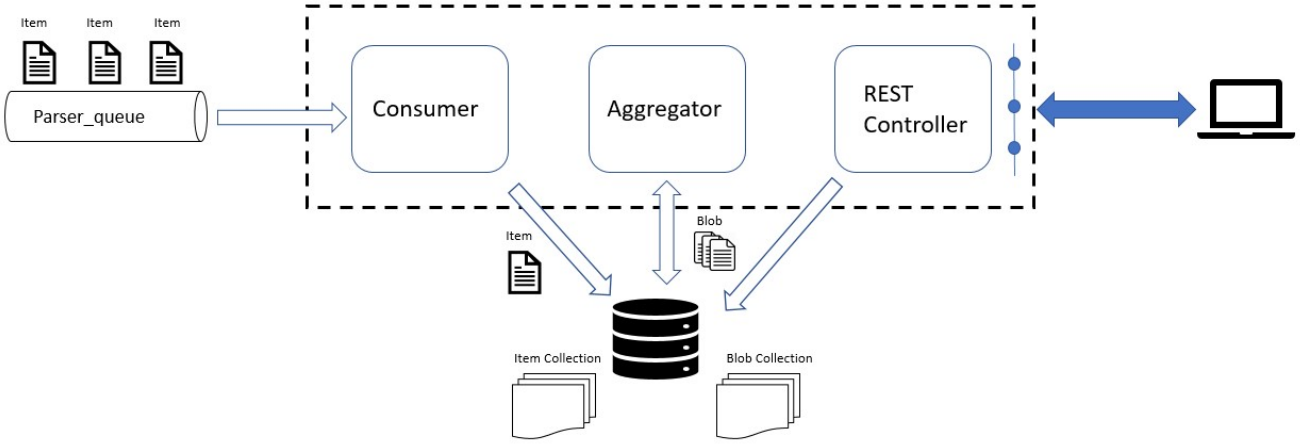


Figure 3.10: Aggregator architecture

purpose of the aggregation is to display the data following semantic criteria, such criteria are passed to the application via the mentioned `aggregator.json`. In fact, we can have many criteria for each content to display and they can change over

the time.
The aggregator acts thus as a sink of the system, where the rest Api are hit by the clients application in order to get and in case post data towards the application. To guarantee high availability, even this microservice is scaled out. In case one Aggregator goes down the application will be still available to serve the data.
Since the Aggregator is scaled, not redundant, we will need a `Load Balancer` in

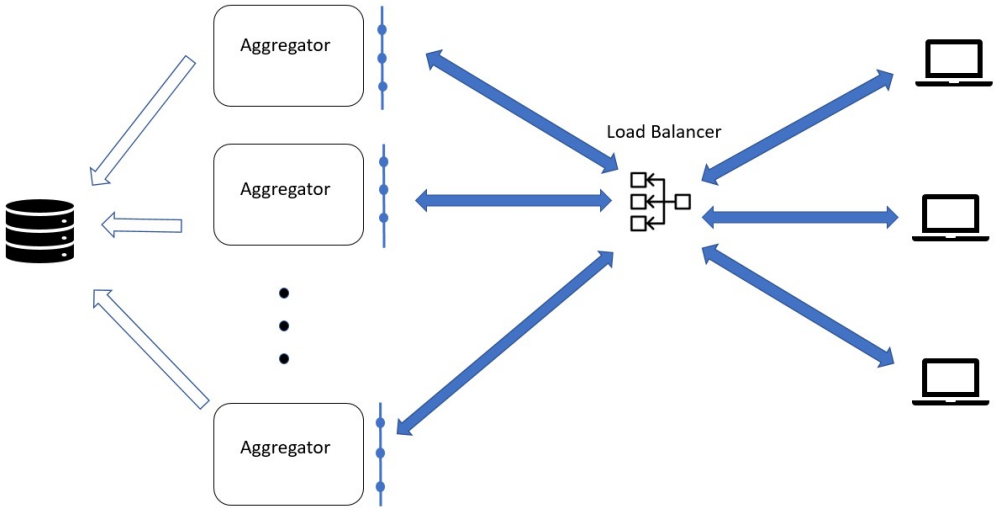


Figure 3.11: Aggregator architecture II

front of the last pipeline stage that will have to forward the traffic to the Aggregator Scaling Group, following some Load Balancing Policy, like, for example, a `Round Robin` policy.
In a real use case scenario, we'll have many endpoints REST responsible for serving quite a lot aspects of the application such as profiling but also administration dashboard and so on.
Let's focus on core, thus the endpoints in charge to handle the data treated by the application.
There are three HTTP methods used to contact such endpoints REST:

1. **GET** : are hit to ask for specific data stored in the DB. Indeed there is some POST methods involved to enrich some requests, but since this doesn't fit with REST specification, in future we'll move such query towards a GET with parameters.

2. **PUT**: used to modify some data present in the DB.

All this methods ask for `Blob` which are the actual data prepared for visualization.

Chapter 4

Deploy

"Currently, DevOps is more like a philosophical movement,
not yet a precise collection of practices, descriptive or prescriptive."
- Gene Kim

4.1 Deploy

Once the system has been developed, it is time to build the production enviroment. For this purpose, as mentioned in the introduction, we will use Docker.

4.1.1 Docker as a virtualization platform

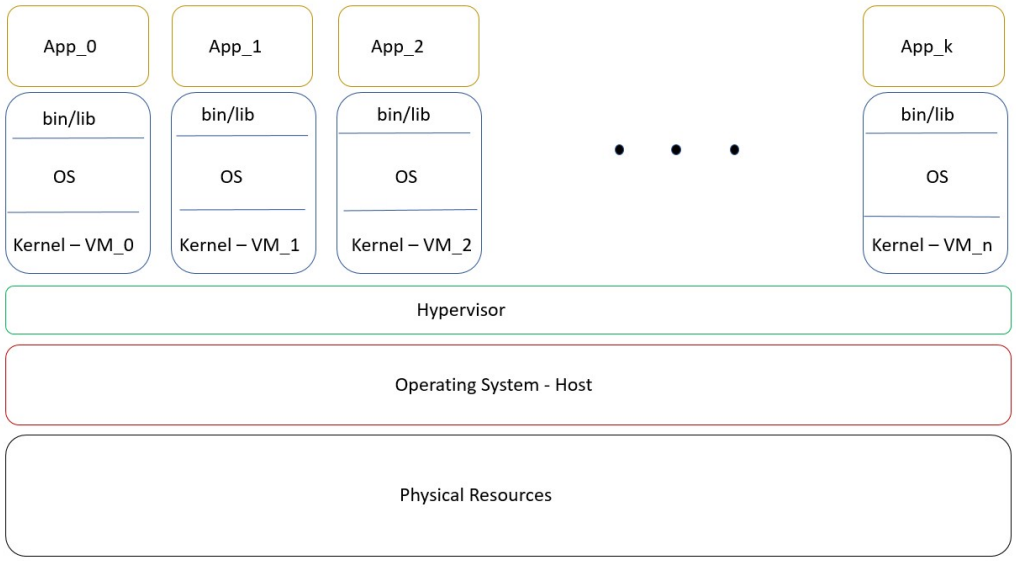


Figure 4.1: Architecture of a virtualized system

Virtualization technologies got into the software industry to optimize physical resources (breaking the old fashion way where to ensure isolation we had to buy new hardware) allowing to run multiple operating systems on the same host that essentially means that allows to run many applications in an isolated enviroment on same host. But virtualization presents some drawbacks, mainly deriving from the fact that virtualization is expensive.
First we have multiple kernel to deal with, and thus many operating systems.
For each operating system we add to our system, we have to allocate resources for it.
Although we do not have physical hardware, we have virtual hardware, so each machine will have its own virtual space, its own virtual RAM and so on as shown in 4.1.
Even if we get many benefits from this approach, is still not the best choice we can take nowadays. Docker is a very good alternative to the previous approach, it

keeps the benefits and eliminate most of drawbacks.
Docker is a virtualization platform that exploit Linux kernel features like *cgroups* and *namespaces* for running applications in an isolated enviroment presenting the following advantages:

- **Lightweight**: we do not have the issue to configure complex virtual machine images hard to set up and hard to deploy
- **Portable**: Docker has images really small and we can use configuration like `Dockerfile` to spin things up instantaneously on almost any system.
- **Fast**: we do not have to boot up an entire operating system and copy all the files, which can be a tricky process, just to make our application running.

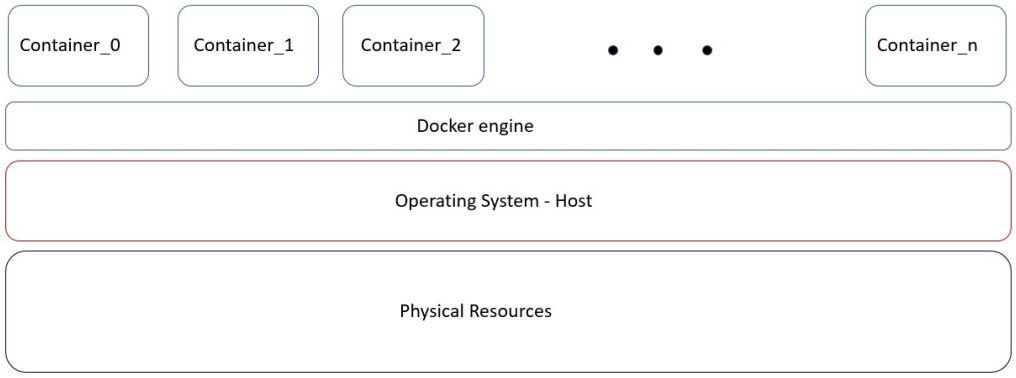


Figure 4.2: Architecture of a Virtualized system through Docker

One of the biggest benefit we can see in Docker is that it doesn't make use of the Hypervisor, that basically means that we do not have anymore separate kernels on top. Docker is still using the same resources as the kernel host and exploits *namespaces* and *control groups* to use such resources in a more efficient way.
Docker has a process that runs directly on the operating system, known as `Docker daemon` which is responsible to manage `Docker Images`, `Docker Containers` as well as many command line utility that comes with Docker. *Docker Containers* are running instances of *Docker Images* that are nothing else than bundles ready to be run.

The application that we deliver with such `Docker Image` basically run directly on the host machine once we start a `Docker Container`. A Docker Container is typically a process running in our host OS, and by using `namespaces` Docker allocates segments of host OS resources as well as indicates how much of resources a container can have by means of `Control Groups`.

Such images are built following a layered approach, thus an image is made up by many tiers stacked on one another. An image is built by running a script called `Dockerfile`, that actually gives instructions on how to build the Docker Image. An example of `Dockerfile` (the one used for building the `Parser Microservice` is reported in 4.1

```
1  FROM openjdk:8-jdk-alpine
2  ADD /target/parser-1.0-SNAPSHOT.jar  parser.jar
3  COPY /wait-for-it.sh /usr/wait-for-it.sh
4  RUN  \
5       apk update && \
6       apk add ffmpeg && \
7       ffmpeg -version
8
9  RUN chmod +x /usr/wait-for-it.sh
10 ENTRYPOINT ["java", "-jar", "parser.jar", "-Xms=4G" , "-Xmx=5G"]
```

Listing 4.1: Example Dockerfile

As we can see it follows a predefined syntax that suggest the layered approach:

1. starts from a base image: the `openjdk:8-jdk -alpine` which provides the enviroment on top of which we can run our application software.

2. our application `.jar` is added to our base image.

3. a shell script is run to configure third party software or plug-in necessary to run our application. In this case the `Parser` Microservice relies on an external software called `ffmpeg`, thus the `RUN` clause arranges the installation.

4. the last step is the `ENTRYPOINT`: defines the command that launches the application software to execute when a Container is run.

Once the images are created, we can now create running containers from them to make run our application. As mentioned before, Docker aims to automate the deploy process providing a script to the Daemon that essentially defines the rules to set up our virtualized enviroment: this script is written following the `YML` declarative syntax and is called `docker-compose`. [10] Compose is thus a tool for defining and running multi-container Docker applications within a context of same

Docker Host.
Docker Compose allows us to define multicontainer application called “stacks” and to run them on a single Docker host o nel cluster; Docker defines stacks as linked services groups sharing software dependencies which are orchestrated and scaled.
Docker Stack allows us to define many application configurations inside a file called `docker-compose.yml` and to run them in an isolated runtime environment managing them in centralized script.
With Compose thus, you use a `YAML` file to configure your application's `services` defined as scalable groups of `containers`.
Then, with a single command, you create and start all the services from your configuration. Compose is basically a three-step process:

1. Define your app's environment with a `Dockerfile` (as in 4.1) so it can be reproduced anywhere.

2. Define the services that make up your app in `docker-compose.yml` so they can be run together in an isolated environment.

3. Run `docker-compose up` to run your entire app.

In the following snippet we show a simplified version of `docker-compose` written to automatically compose our system, from image building to user-defined network definition:

```
version: "2.2"
services:
  ingester:
   build:
    context: .\ingester
    dockerfile: Dockerfile
   networks:
     - service_network
   environment:
     - _JAVA_OPTIONS=-Dspring.profiles.active=prod
   dns: 8.8.8.8
   restart: always
   depends_on:
     - mongodb
     - eureka
   links:
     - mongodb:mongodb
     - eureka:eureka
   command: bash -c "/usr/wait-for-it.sh --timeout=0 eureka:8761"

```

Listing 4.2: Example docker-compose

```
  mongodb:
   image: mongo:latest
   container_name: db_container
   restart: always
   networks:
     - service_network
   depends_on:
     - eureka
  eureka:
   container_name: eureka_container
   image: jhipster/jhipster-registry:latest
   volumes:
         - host_path:container_path
   environment:
    - SPRING_PROFILES_ACTIVE=native
    - JHIPSTER_SECURITY_AUTHENTICATION_JWT_BASE64_SECRET=xxx
    - SPRING_SECURITY_USER_PASSWORD=xxx
    - JHIPSTER_REGISTRY_PASSWORD=xxx
    - SPRING_CLOUD_CONFIG_SERVER_COMPOSITE_0_TYPE=native
    - SPRING_CLOUD_CONFIG_SERVER_COMPOSITE_0_SEARCH_LOCATIONS=file
   :/central-config/Ingester, file:/central-config/Parser , file:/
   central-config/Aggregator
   restart: always
   dns: 8.8.8.8
   networks:
     - service_network
networks:
  service_network:
    driver: bridge
    ipam:
        config:
        - subnet: 10.0.0.0/16
          gateway: 10.0.0.1
```

Listing 4.3: Example docker-compose

The `docker-compose` shows that we created three `services` in terms of `Docker Containers`. We created

1. Service called `Ingester`: our microservice
2. MongoDB: the database
3. Service called `Eureka`: the service discovery

In the compose we specify for each service a bunch of instructions. The main ones are:

- **Where to build the image from**: we have to specify the image or the `Dockerfile` to build it.
- **The restart policy**: sometimes a container can go down due to many reasons, when it happens we can specify a strategy to bring it up again.
- **The network** which the relative container will be attached to.
- **A volume** mapped to host filesystem to have persistence after some container has gone down.
- **The ports** to expose towards the outside world.

With such mechanism we are not only able to build in a very easy way even very complex applications, but we can deploy them just running the command

```
docker-compose up -scale ingester=n -scale mongodb=1 -scale
                                eureka=1
```

from the directory where is the compose file.
With the option

```
-scale <service_name>=<number_of_instances>
```

we are telling the docker daemon that we want to scale the service specified. By default Docker scale one instance for each image. Docker compose presents a limitation: it works only in the local Docker Host. The multi-container application is deployed thus on a single machine limiting quite a lot the architecture features. Docker extends its range also to multiple Docker Host distributed around the Network including a tool called `Docker swarm` for natively managing a cluster of Docker Engines called "swarm". Docker Swarm allows to broaden the arguments discussed so far from Docker Host to a cluster of Docker Engines. This will be discussed in future improvements section.

4.1.2 Set up the Jhipster-Registry

In our scenario we used a preboiled Docker image and set it up through enviroment variables overwriting the default configuration as follows:

- `SPRING_PROFILES_ACTIVE= native`
- `JHIPSTER_SECURITY_AUTHENTICATION_JWT_BASE64_SECRET=<secret_key_base64>`
- `SPRING_SECURITY_USER_PASSWORD=<user_password>`

- `JHIPSTER_REGISTRY_PASSWORD=<service_password>`
- `SPRING_CLOUD_CONFIG_SERVER_COMPOSITE_0_TYPE=native`
- `SPRING_CLOUD_CONFIG_SERVER_COMPOSITE_0_SEARCH_LOCATIONS=<list_of_paths>`

Actually with these we are simply giving the basic security configuration and telling the cloud configuration server module to run in native configuration that means to store and retrieve servers configuration files in the local file system.
Alternatively the Configuration Server can be set to *Composite* to retrieve files from a remote repository such as Git.
Once we run the image as a Docker Container we get the service available and ready to work.

4.1.3 Proposed solution for deployment

Following the requirements, we have to set up a microservices-based pipeline and deploy it by *Dockerizing* the various services needed. In figure 4.3 we can see the Application Deployed.

We can see that there is a user-defined network with subnet range address

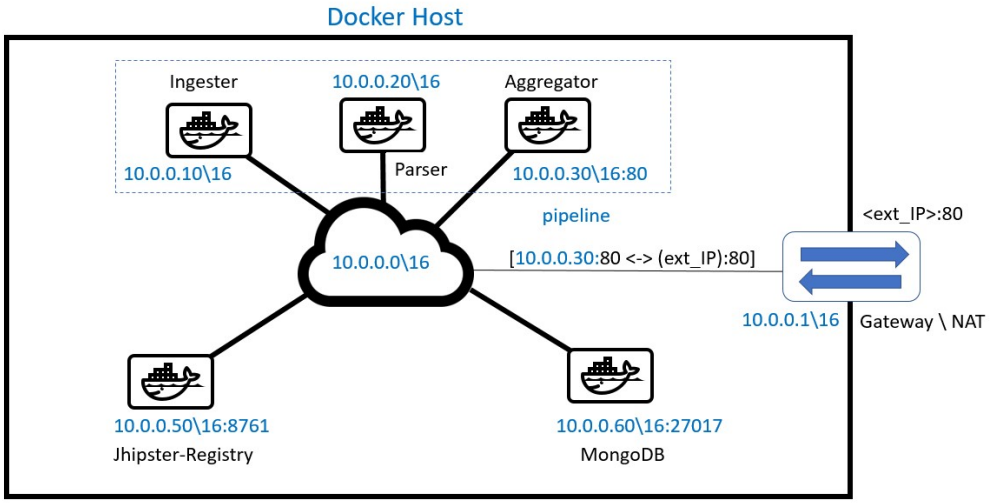


Figure 4.3: Deploy Architecture

`10.0.0.0/16`.
Every services are attached to such network to let containers communicate each other.
Containers expose ports but those are visible only from inside the user-defined network called `service_network`.
The Database, MongoDB, exposes the port 27017 to be reached out by microservices.
Each microservice exposes the port 5701 because Hazelcast members have to communicate in order to join into a Cluster.
The Hazelcast Discovery is performed through Eureka Service Discovery embedded into the Jhipster-Registry.
The Jhipster-Registry exposes only the port 8761 and uses this port both for Service Discovery and for Configuration Server (the default port for the configuration server used - Cloud Config - would be the 8888).
We can think to expose the port 8761 of Jhipster-Registry to the outside world since it offers also a Dashboard where is possible to monitor what's happening with service discovery.

The only port we must expose to the outside world is the 80 port of the Aggregator, as it exposes some REST Api.
This is a partial solution as the microservice should not expose ports, but there will be a gateway like Jhipster-Gateway where pass all the ingoing and outgoing traffic. The Gateway can be a Single Point of Failure, but present some advantages like centralizing cross-cutting concerns for easier maintenance and acts as decoupling point between the application and the outside world, this would be also a module implementing load balancer policy, as the aggregator service is scaled out, to allow a fair spread of traffic incoming from client application among all aggregator instances.

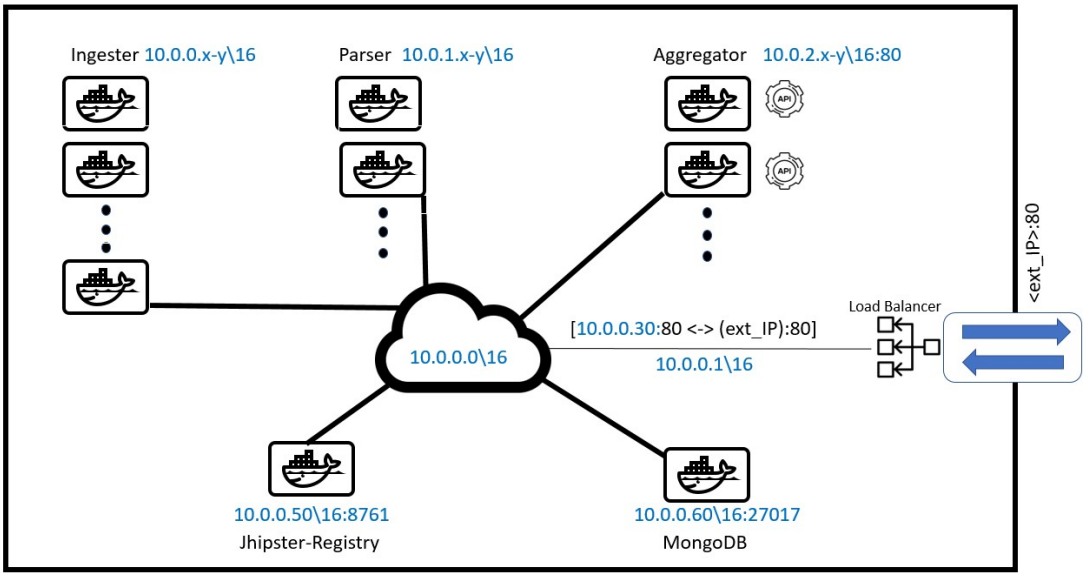


Figure 4.4: Deploy Architecture II

Also the implementation of Apis and Gateway Configuration will be discussed in section future improvements.

Chapter 5

Alternative Solutions

"If you think good architecture is expensive,
try bad architecture."
—Brian Foote

5.1 Kafka as Message Broker

In the early design stages, when defining the architecture have been explored some alternative solutions. The choices were still on Event-Driven solutions but relying on a technology called `Kafka` introduced in section 2.1.5 instead of Hazelcast.
In the architecture we will have 4 Microservices involved and the kafka cluster responsible for the communication acting as Message Broker.

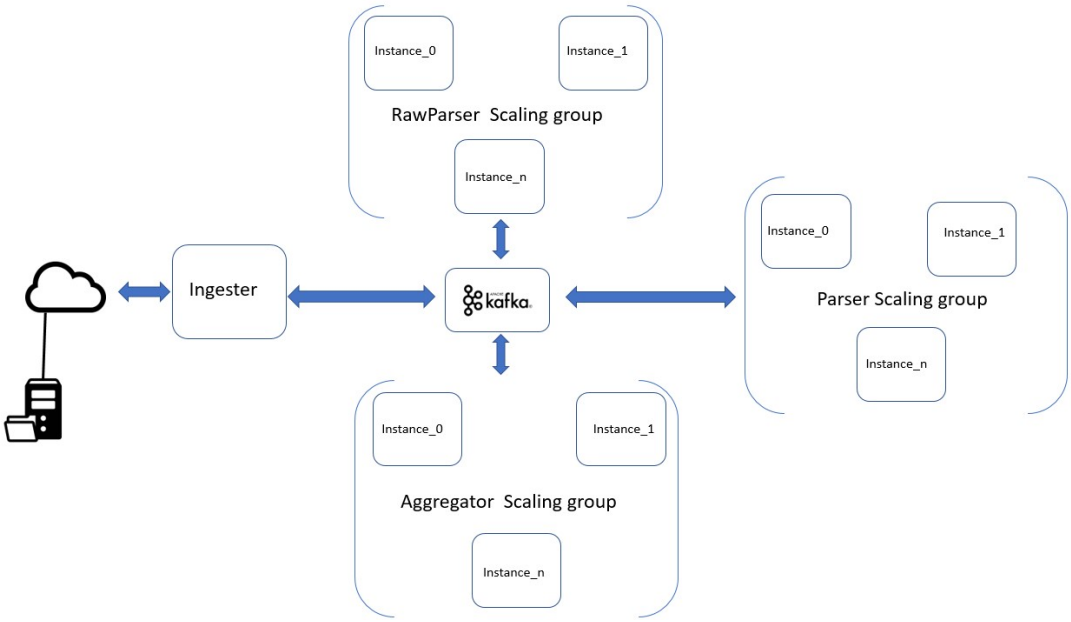


Figure 5.1: Architecture with Kafka

The architecture is still similar to the one presented in section 3.1: we can find

- **An ingester**: this time the ingester is not scaled: it acts as a bare consumer which fetches the data from the SFTP server and push it up into the relative Kafka Topic.

- **A RawParser**: There is a first parser which hold the parsing logic of the 3.2 in the presented architecture. It handles indeed the `RawEntity` and turn them into `Entity`. This Microservice is scaled, we will have thus multiple instances of it.

- **A Parser**: Is the second phase of parsing, is the microservice responsible to process the `Entity` in order to get `Item`. Also this microservice will be scaled and organized as a scaling group.

- **An Aggregator**: Is the last stage. Is the corresponding microservice of 3.4. Also this microservice will be scaled.

The data flow from one stage of the pipeline to the next through kafka topics[1] The microservices instances are organized as a Consumer group with respect to its topic of interest. We will end up to have three topics.

Each Topic is responsible to manage a different type of data model and the

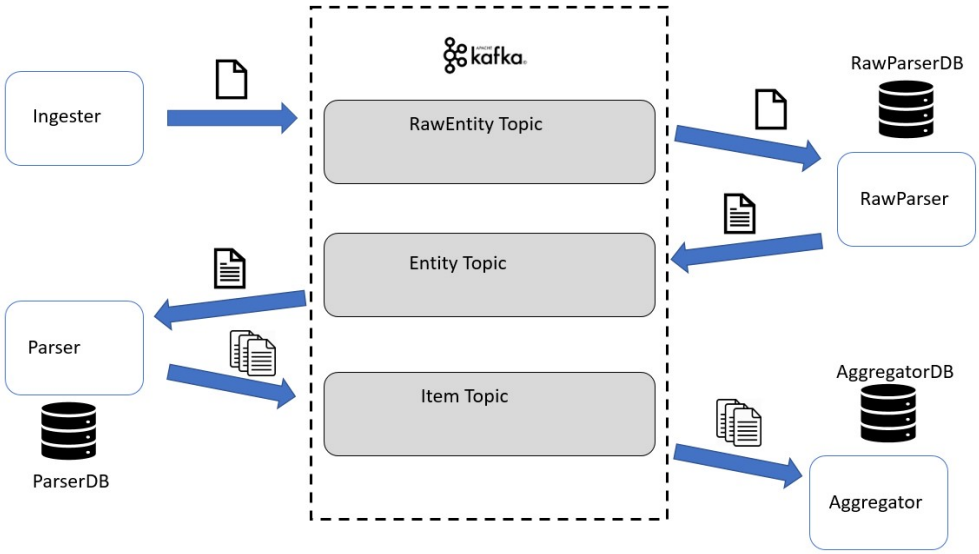


Figure 5.2: Pipeline with Kafka as Message Broker

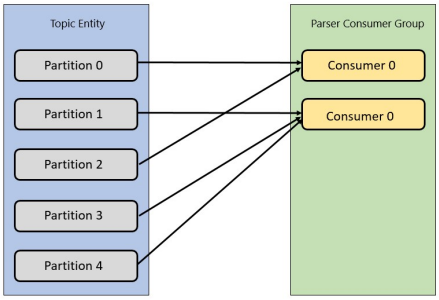
microservices act as Producers and Cosumers with respect to such Topics. Producers publish data to the topics of their choice.

The producer is responsible for choosing which record to assign to which partition within the topic, in our solution is still valid that data comes along fragmented and out of order, so to guarantee that the Parsing for the same data is done by the

[1] [7] A topic is a category or feed name to which records are published. Topics in Kafka are always multi-subscriber; that is, a topic can have zero, one, or many consumers that subscribe to the data written to it.
The Kafka cluster durably persists all published records—whether or not they have been consumed—using a configurable retention period. For example, if the retention policy is set to two days, then for the two days after a record is published, it is available for consumption, after which it will be discarded to free up space. Kafka's performance is effectively constant with respect to data size so storing data for a long time is not a problem.

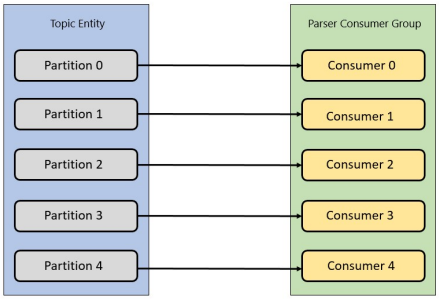
same thread, we have to exploit Topic partitions[2].
Producer will decide target partition to place any message, depending on the partition id that must be specified within the message. So we have to associate fairly a set of `Agent_Id+Content_type` to each partition. Consumers label themselves with a consumer group name, and each record published to a topic is delivered to one consumer instance within each subscribing consumer group.
Consumer instances can be in separate processes or on separate machines.
If all the consumer instances have the same consumer group, then the records will effectively be load balanced over the consumer instances.
If all the consumer instances have different consumer groups, then each record will be broadcast to all the consumer processes.
Partition are not dynamically created, so we have to choose the number of partitions that we want and then Consumers can be created dynamically to be associated to partition accordingly. We put Consumers to listen to specific partions.
Each partition manage a certain number of `Key` in relational database meaning (which for instance at Entity level the key is made out of `Agent_Id+Content_type`) so the producer will send data to a specific partition of a specific topic, depending on the `Key`.
A single consumer is attached to one partition. That consumer is thus responsible for processing all data belonging to that set of `Key` since within the same consumer group each partition will be assigned to one consumer only. We will have three possible scenarios:

1. The consumer group has fewer instances than the number of partition: this will lead to one consumer being assigned to multiple partitions

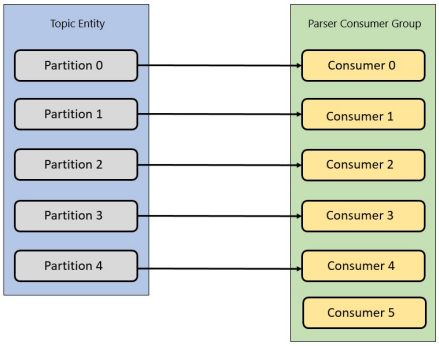


[2]A `Partition` is an ordered, immutable sequence of records that is continually appended to—a structured commit log. The records in the partitions are each assigned a sequential id number called the offset that uniquely identifies each record within the partition.

2. The consumer group has same number of topic partitions, then partition and consumer mapping will one to one.



3. if number of consumer is higher than number of partition then we are wasting resources because some of consumer will not actually consume anything.



This schema is followed for every topic and the relative Producers and Consumers.

5.1.1 Pros and Cons

Let's show the advantages and disadvantages of comparing the architectures presented concerning the goals to achieve and the reasons why it has been adopted the Architecture exploiting Hazelcast.

Scalability

Comparing the two solutions is clear that Hazelcast provides a higher degree of flexibility to our architecture. In fact, with Hazelcast the scaling factor is not bounded by anything. Using Kafka, the scaling factor inside a single scaling group is bounded by the number of partitions chosen during development.
The number of different partitions for a given Topic is an upper bound for the number of consumers we can deploy.
Moreover, if we use Kafka the complexity to coordinate many SFTP Client leads to an architecture where becomes harder to scale the ingester heading to a Bottleneck and Single Point of Failure.

Maintenability

As the project grows in size we might be in the situation where we need to add more content types to our project: while in the kafka solution this growth will impact heavily on the application, in the one presented using Hazelcast will be painless, for example content_type are hardwired to partitions and this will impact whenever we'll have the need to add more `content_type`.
Kafka is a service separated from our pipeline and works following a Server/Client interaction. This introduce in our architecture one more element that needs to be maintained separately. Moreover Kafka makes use of a service discovery called `Zookeeper` and this would have implied to discard Jhipster-Registry and use another technology also for Configuration Server. The Architectural components will be more with a Kafka architecture that would have probably been harder to maintain.

Fault Tolerance

From this point of view Kafka offers persistence unlike Hazelcast that has been is thought as Cache in memory. If the Kafka goes down, when it reboots all the record enqueued are still there because kafka relies on very efficient persistence policies based on `.log` files. With Hazelcast we do not have such feature and the persistence needs to be enabled by using specific interfaces that allow the

implementation of write-through [3] policies.

5.2 Hazelcast solution: Scaling with finer granularity

The architecture presented so far suffers from a certain degree of coupling between the pipeline stages that we cannot get rid of.
In fact, despite the effort to decouple our architecture, pipeline stages are somehow coupled with the fact that they have to work on the same data models creating a dependency between various stages, which means that our services are not completely loosely coupled.
We can think to organize the architecture not to have microservice responsible for serving a single pipeline's stage, but responsible for managing all steps concerning a specific `content_type`. The architecture will be more loosely coupled and with an higher modularity's degree. If we consider the architecture depicted in 5.3,

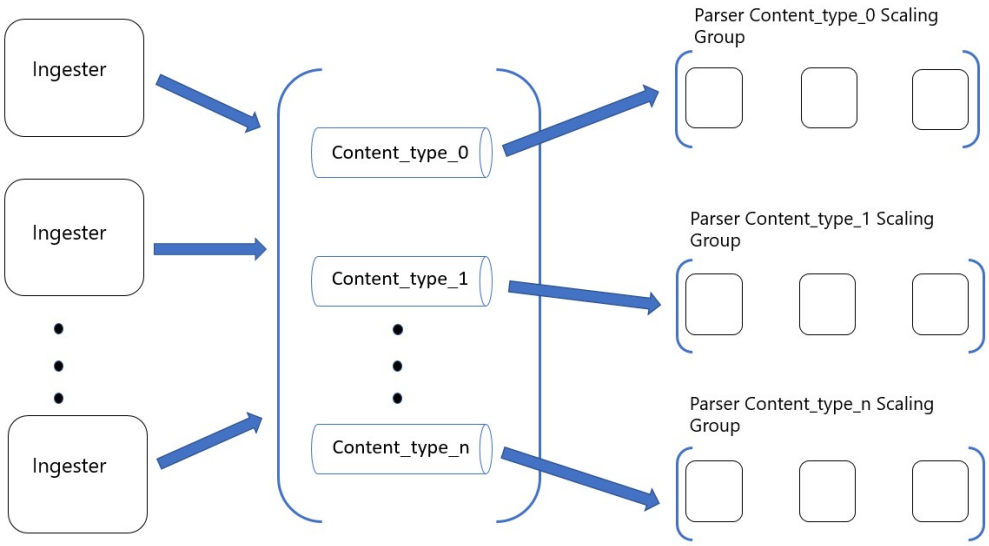


Figure 5.3: Architecture

[3]Write through is a storage method in which data is written into the cache and the corresponding main memory location at the same time. The cached data allows for fast retrieval on demand, while the same data in main memory ensures that nothing will get lost if a crash, power failure, or other system disruption occurs.

we can notice that a lot of complexity has been added to our application. First of all we are now dealing with as many queues as many `Content_types` which number might grow over the time. Moreover, in the 5.3 is not explicit anymore the elaboration pipeline.
We will end up to have only 2 microservices

- **1 Ingester Microservice**: we will have one ingester very similar to the one presented in 3.2
- **Many Parser Microservices**: we will have as many Parser Microservice as many content_types. Each Parser Microservice is responsible to parse, process and aggregate Entity with the same `Content_type`.

In fact, the pipeline is totally contained in every Parser instance (the rightmost part of 5.3). In this architecture the ingester acts exactly as in the solution developed: it's responsible for fetching the data from the SFTP server and performs a first elaboration in order to get the `content_type` of the current `Entity`.
Once we get the `content_type`, the Ingester will forward the `Entity` towards the relative queue.
For each and every queues we will have a service consuming the data pushed into it and responsible to perform all the subsequent steps of the pipeline (Parsing and Aggregation). We will have many instances of such service as it will be scaled out.
In this scenario we will end up to have an ecosystem where each microservice is responsible for treating only one specific kind of data, making possible to extend the application by simply adding a new microservice whenever we add new kind of device.
From the technology point of view we can think to still use Hazelcast and have a single Database within the same scaling group where all the instances access to the same database, in a similar manner we did in the solution developed. In this way we can keep the goals we achieved with the developed solution and without having the limits of that architecture.
We can think of this architecture as an extension and improvement of the architecture developed.

5.2.1 Pros and Cons

Loosely coupling

If we look at this architecture we can notice that the microservices can be developed in different programming languages and by different teams without being strictly bounded one another.
Each microservice can be deployed independently and is almost self consistent with respect the application purposes.
We can find a lot of advantages in this architecture, but in order to implement it there is a lot of complexity that needs to be managed, we are adding a potentially insane number of microservices that need to be developed, deployed and maintained individually.

Scalability

From the scalability point of view this architecture is completely different from the one developed.
We are not scaling anymore on a single pipeline stage, but we can scale on a self-consistent service that handles many stages of processing, but responsible for handling a single kind of data.
Such Architecture allows us to react load variations in a smarter way, in fact, we can bring up and scale out only the microservices responsible for serving the data types present at moment notice.
Such extremely flexible system allows to minimize wasting resources and to maximize the the efficiency.

Maintainability

The maintainability is affected by two factors: In a first instancewe have in fact a loosely coupled system, due to this property we're able not only to manage versioning issues easily but we can evolve our system without affecting what has been already done.
This loosely coupling allows also to build teams around services and make them responsible for the entire lifecycle of a particular application's service.
The solution obviously adds a lot of complexity, indeed the number of service requested depends linearly on the number of content types which could easily blow up.
In order to develop, deploy and to deal with such architecture we have to deal with its intrinsic complexity which ends up to be more expensive.

Chapter 6

Conclusion

"It is not the strongest of the species that survive,
nor the most intelligent,
but the one most responsive to change."
—Charles Darwin

6.1 Results achieved

The architecture that has been implemented addressed some issues about scalability, high availability and maintenance.

The architecture presented doesn't respect precisely all the properties a Microser-

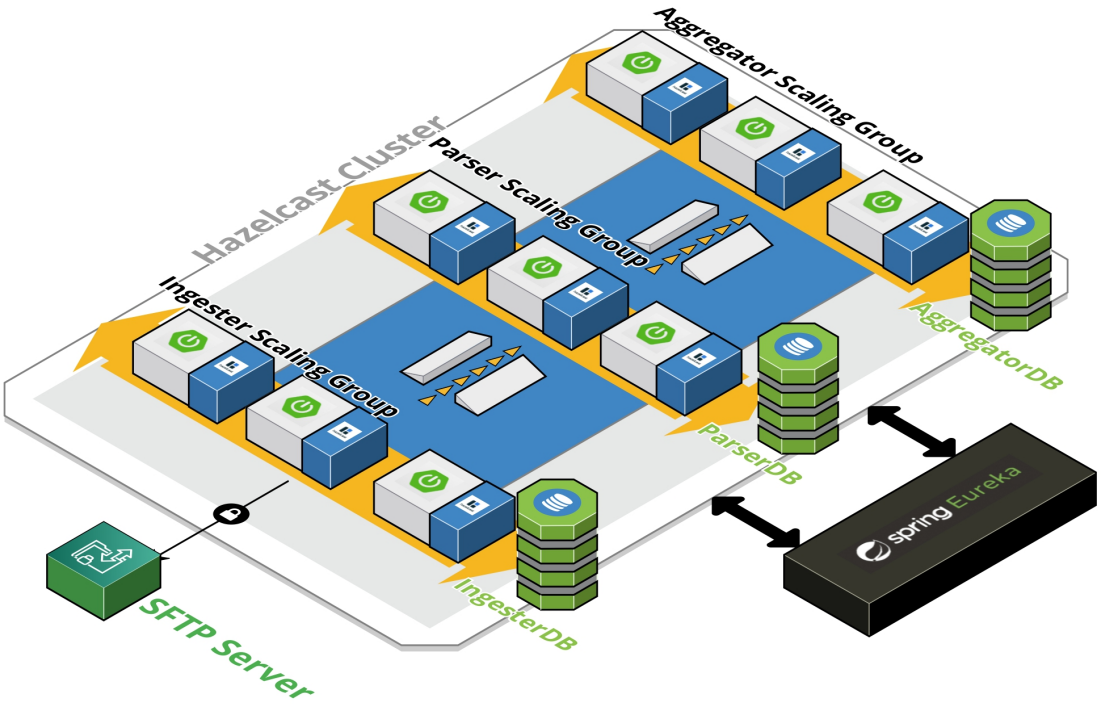


Figure 6.1: Overall Architecture

vice Architecture should have due to the specific nature of the application, but shows that Microservices fit or can be adapted in many use cases very different from each other.

In the context of data pipelines, it is plausible to think that each pipeline's stage would require a different amount of resources. During the design process, the stages are identified to be the elementary functional units: making them independently deployable units allows us to allocate the right amount of resources for each of them as well as giving the chance to scale horizontally at such granularity level.

The use of Jhipster-Registry allows to centralize Service Discovery and Configuration Server uniforming as much as possible the technologies and easing maintenance aspects. The configuration server centralizes all the configuration files, so that when some configuration needs to be changed, it can be done in a single location without going around looking for the right file to change.
The High Availability of the application is guaranteed not by redundancy but thanks to its high scalability that ensures a constant level of computational power even when the application is stressed out.
The fault tolerance is guaranteed by fallback mechanisms implemented along with Hazelcast Clusterization, that keeps all data structures backed up so even if some nodes goes down, the behaviour of the system is not affected. We can see in 6.1 the Architecture of the system developed, it highlights the topology of the system in terms of scaling groups (equivalent to Docker services) and architectural components like the Databases, the jhipster-Registry and the data source.

6.2 Future Improvements

6.2.1 Docker Swarm

As discussed in 4.1, our deploy can be extended to multiple Docker Host in order to exploit the architecture's potentialities.
Docker Swarm is a tool to manage and orchestrate cluster of Docker Hosts called swarm. a A swarm consists of many Docker hosts running as a cluster and act as managers (to manage membership and delegation) and workers (which run swarm services). A given Docker host can be a manager, a worker, or perform both roles. When defining a service we set parameters which in somehow form a state of the service (number of replicas, network resources , storage resources available to it, ports the service exposes to the outside world, and so on), once we run the service Docker works to keep such state.
If some Docker Host node goes down, Docker schedules that node's tasks on other nodes where a task is defined as a running container which is part of a swarm service and managed by swarm manager.
Deploying our application through Docker Swarm we can exploit thus a swarm of docker hosts to exploit a geographical distributed architecture.

6.2.2 Hazelcast Persistent queues

One of the main criticalities in the system developed is that the communication is performed through queue data-structures managed by a distributed in-memory cache. This Architecture is robust with respect to node failures but if the system goes down entirely the data present into the queues will get lost when the system reboots, this is because is not enabled the persistence on the queues.
Hazelcast allow to rise events on data structures that can thus be backed up in a database to enable persistence and make the system more robust to failures.
This improvement enhances **robustness** at the cost of **performance**.

6.2.3 Monitoring system

A possible improvement would be to add a server responsible to collect informations on how the system is behaving.
This server should monitor Hazelcast to see how the queues are growing and optionally establish some healthcheck mechanism towards the microservices by means of the service discovery. Such server will expose REST Api in order to provide such infomations to a client application that allows interaction with the system. The client application should be a kind of Dashboard that can be thus exploited also to react to critical situations and allow a easy deploy within the enviroment.
To achieve this, such monitoring server has to commmunicate with the Docker Engine Api that docker daemons exposes allowing to dynamically create, destroy, run, start and stop containers. We can think to design the system in order to allow the system administrator to use such administration tool not only to provide resource availability to the system, ma also to set up threshold and auto-scaling mechanisms to be automate the system as much as possible.

Bibliography

[1] Wikipedia. *Data Lake* (cit. on p. 2).

[2] Garrett Alley. *What is a Data Pipeline?* Nov. 2018. URL: https://www.alooma.com/blog/what-is-a-data-pipeline (cit. on p. 4).

[3] Kyle Brown. *Beyond buzzwords: A brief history of microservices patterns*. Ed. by IBM. [Online; posted November 30, 2016]. Nov. 2016. URL: https://developer.ibm.com/technologies/microservices/articles/cl-evolution-microservices-patterns/ (cit. on p. 5).

[4] Wikipedia. *Connascence*. URL: https://en.wikipedia.org/wiki/Connascence (cit. on p. 7).

[5] Martin Fowler. *Microservices*. Ed. by martinFowler.com. [Online; posted March 25, 2014]. Mar. 2014. URL: https://martinfowler.com/articles/microservices.html (cit. on p. 8).

[6] Wikipedia. *CAP Theorem* (cit. on p. 17).

[7] Kafka Documentation. *Kafka*. URL: https://kafka.apache.org/ (cit. on pp. 18, 56).

[8] Mongo Documentation. *GridFS*. URL: https://docs.mongodb.com/manual/core/gridfs/ (cit. on p. 22).

[9] Hazelcast Documentation. *FencedLock*. URL: https://docs.hazelcast.org/docs/3.12-BETA-1/javadoc/com/hazelcast/cp/lock/FencedLock.html (cit. on p. 31).

[10] Docker documentation. *Overview of Docker Compose*. URL: https://docs.docker.com/compose/ (cit. on p. 47).