ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA
Campus di Cesena

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

Prototyping a scalable Aggregate Computing cluster with open-source solutions

Tesi in
Ingegneria dei Sistemi Software Adattativi Complessi

Relatore:
Prof. MIRKO VIROLI

Co-relatore:
Dott. ROBERTO CASADEI

Presentata da:
CRISTIAN PAOLUCCI

Anno Accademico 2016-2017
Sessione III

Contents

List of Figures

List of Codes

Abstract (italiano)

L'*Internet of Things* è un concetto che è stato ora adottato in modo pervasivo per descrivere un vasto insieme di dispositivi connessi attraverso Internet. Comunemente, i sistemi IoT vengono creati con un approccio bottom-up e si concentrano principalmente sul singolo dispositivo, il quale è visto come la basilare unità programmabile. Da questo metodo può emergere un comportamento comune trovato in molti sistemi esistenti che deriva dall'interazione di singoli dispositivi. Tuttavia, questo crea un'applicazione distribuita spesso dove i componenti sono strettamente legati tra di loro. Quando tali applicazioni crescono in complessità, tendono a soffrire di problemi di progettazione, mancanza di modularità e riusabilità, difficoltà di implementazione e problemi di test e manutenzione. L'*Aggregate Programming* fornisce un approccio top-down a questi sistemi, in cui l'unità di calcolo di base è un'aggregazione anziché un singolo dispositivo.

Questa tesi consiste nella progettazione e nella distribuzione di una piattaforma, basata su tecnologie open-source, per supportare l'Aggregate Computing nel cloud, in cui i dispositivi saranno in grado di scegliere dinamicamente se il calcolo si trova su se stessi o nel cloud. Anche se Aggregate Computing è intrinsecamente progettato per un calcolo distribuito, il *Cloud Computing* introduce un'alternativa scalabile, affidabile e altamente disponibile come strategia di esecuzione. Quest'opera descrive come sfruttare una *Reactive Platform* per creare un'applicazione scalabile nel cloud. Dopo che la struttura, l'interazione e il comportamento dell'applicazione sono stati progettati, viene descritto come la distribuzione dei suoi componenti viene effettuata attraverso un approccio di containerizzazione con *Kubernetes* come orchestratore per gestire lo stato desiderato del sistema con una strategia di *Continuous Delivery*.

Keywords – *aggregate programming, cloud computing, reactive platform, kubernetes, continuous delivery*

Abstract

The *Internet of Things* is a concept that has now been adopted in a pervasive way to describe a vast set of devices connected through the Internet. Commonly, IoT systems are created with a bottom-up approach that focus on the single device, where it is seen as the basic programmable unit. From this method can emerge a common behavior found in many existing systems that derive from the interaction of individual devices. However, this creates often tightly packed distributed application. When such applications grow in complexity, they tend to suffer from design problems, lack of modularity and reusability, deployment difficulties, and test and maintenance issues. *Aggregate Programming* provides a top-down approach to these systems, where the basic unit of computing is an aggregation instead of a single device.

This dissertation consists in the design and deployment of platform, based on open-source technologies, to support Aggregate Computing in the Cloud, where devices will be able to dynamically choose if the computation is located on themselves or in the cloud. Even though Aggregate Computing is inherently designed for a distributed computation, the *Cloud Computing* introduces a scalable, reliable and high-available alternative as execution strategy. This work describes how to take advantage of a *Reactive Platform* to build a scalable application in the cloud. After the application structure, interaction and behavior have been designed, it is described how the deployment of its components is made through a containerization approach with *Kubernetes* as an orchestrator to manage the desired state of the system with a *Continuous Delivery* strategy.

Keywords – *aggregate programming, cloud computing, reactive platform, kubernetes, continuous delivery*

Introduction

In a world where every device is connected to the Internet a coordination model is required for them to operate. Internet of Things devices are nowadays pervasive and often operate on their own or through client-server interaction. To coordinate these devices, which are unreliable and always changing, an adaptive complex system must be deployed.
Today the most elastic, reliable, high-available and scalable method to deploy an application with this requirement is cloud computing. Cloud computing consists of moving the applications from local machines to a centralized data center maintained by a provider that allows elastic self-service provisioning of resources. Cloud computing can be public, when big companies offer this service for a fee depending on how much resources are consumed, or private, when a cluster is owned.

Independently of how the resources are obtained, the application must be designed to scale accordingly to the load, based on the number of devices currently connected, without a re-implementation of the system. To accomplish this result, an old-fashioned vertically scalable system is not sufficient. A horizontally scalable system, unlike a vertically scalable system, is virtually limited only by how many entities can be connected and these entities should be interchangeable on the fly.

Most vendors and open-source software developer focus on these architectures and nowadays there are a lot of different implementations for every component needed that must be analyzed before starting the design phase.

The application infrastructure that has been designed, inspired by [1] and [2], and consists of:

- Database: to store data.

- Message oriented middleware: for communication between components.
- Data processor: where the actual business logic is located.
- Cluster manager: that encapsulates the application for cluster deployment.

For this study, some technologies have been considered that follow a certain criterion. The focus of this application is to support aggregate computing coordination in the cloud with open-source solutions, so these technologies must be:

- open-source;
- widely used and well documented;
- best in class and based on best practices;
- horizontally scalable, reliable and high-available.

This dissertation first presents in Chapters 1, 2, 3, and 4 an analysis on current technologies as a background for the development. Chapter 5 describes aggregate computing. In Chapter 6 the platform analysis begins, where requirements are listed and analyzed in order to create a first system architecture. Right after in Chapter 7 the technologies are analyzed to close the abstraction gap. Then there is an accurate description of the implementation and deployment process in Chapter 8 and 9. Subsequently, in Chapter 10 and 11, testing and evaluation of the system are presented. Conclusive remarks and future perspectives are presented in Chapter 12.

Chapter 1

Databases

Databases are essential to every software system. As a matter of facts correctly modeling a database is essential for the system to function properly. If the database is poorly designed or managed the performance is compromised and future development will become uncertain.
Currently there are two main class of databases: relational and non-relational. Relational Databases save data in tables and all their operations are based on relational algebra and data is accessed through Structured Querying Language (SQL). Relational databases have rigid structure where information is linked through tables thanks to foreign keys, which uniquely identifies them. This connection between data is very useful for data analysis. Non-relational databases, instead, focus on flexible data structures are that mainly: key/value pairs, column, graph or document and usually does not rely on queries, even though in most of them are loosely implemented.

When a big amount of data is processed, relational databases may lack in terms of performance and must be designed with no errors or else transaction may fail. On non-relational databases transactions are created and verified manually so transactions may fail. It is the user duty to correctly manage them. Moreover non-relational databases are designed with performance and scalibility in mind.
Database are also classified by the hardware support in which data is located: in-memory or in-disk. In-memory databases store data inside of the RAM that is faster but is volatile. Storing data on disk is slower, even more with old magnetic

support, but it is persistent. Usually hybrid solutions are easily deployed. For example, in-memory databases can save data on disk for: historicizing with aggregate data, backup as a copy or extra storage capability, while in-disk databases can use RAM as cache.
Some databases have been designed to work in a distributed environment and scale horizontally. However, for every database has been developed a method more or less efficient to achieve scalability and high-availability. Some problem with distributed databases are: the architecture and the access method. Multiple database instances can either divide the data between them to balance load (sharding) or act as master-slave where slaves are read-only copies of their master that reduce reading load and can act as master in case of node failure. Of course, hybrid solutions are often implemented. Once the architecture is established, access mode needs to be defined. Since instances are distributed, data can be in any one of them. The component that defines the sharding policy can be located in different positions, as seen in Figure 1.1.

- Client-side: each client knows data distribution and access the correct instance.
- Server-side: the servers can communicate between them and redirect transaction to the correct node.
- Proxy: knowledge of data distribution is decoupled from clients and servers and is managed by a proxy.

1.1 MySQL

MySQL[25] is an open-source relational database management system. It is one of the most used worldwide and is one of the key component in the LAMP open-source web application software stack. Thanks to the vast community MySQL is one of the best DBMS for small and medium application since is easy to set up and support availability is high. However, there is limited support for clustering in the open-source version. MySQL offers a lot of enterprise solutions with high availability and performance like the cluster solutions, however those are mostly

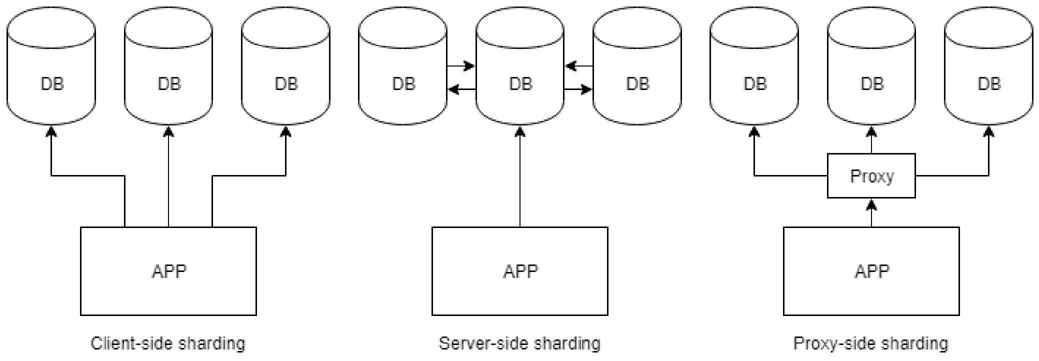

Figure 1.1: Database sharding methods

license-based. Therefore, it needs manual sharding to scale in the open-source version.

1.2 Neo4j

Neo4j[24] is a high-scalable open-source graph database management system. Using a graph, the focus is on connections between data points; in fact, information is stored in label, nodes, edges and properties. Therefore, Neo4J is most efficient when mapping, analyzing, storing and traversing networks of connected data. Graph database main features are:

- Performance: the graph model is designed to maintain performance even with high quantity of data since it does not require costly join operations.
- Flexibility: there is no need for a complete model from the start because data can be added without compromising the current functionality, since the idea of fixed relationships is not inherently present.

  High-availability in Neo4j is obtained with replication of the entire graph on child nodes, distributing read operation. Also, sharding can be performed dividing the graph in chunks that can be accessed in-memory.

1.3 PostgreSQL

PostgreSQL[23] is an object-relational database management system and PostGIS[22] is a spatial database extender. PostgreSQL is a general-purpose object-relational database management system. It allows to add custom functions developed using different programming languages such as C/C++, Java, etc.

Also, PostgreSQL is designed to be extensible and can define new data types, index types, functional languages, etc. Moreover, it allows to develop a custom plugin to enhance any part of the system to meet additional requirements.

Lastly, PostgreSQL have a replication system that creates copies of data on slave nodes. However, there is no sharding policy and clusterization is not trivial.

1.4 Redis

Redis[21] is an open-source in-memory data structure store. Redis uses a key/-value pair to store data in memory and supports a set of data structures on which can perform atomic operation, like classic string value or more complex hashes, sets or lists. With this data structures it can also store locations and perform geospatial queries. Redis can persist data on disk or act as a cache, storing data exclusively on RAM. Data persistence is managed by dumping on disk a log of every command executed, and is primarly used by redis when a node fails and needs to be restarted. Redis is not only fast due to in-memory storage but it can also provide high-availability with Redis Cluster.
Redis Cluster is a way to run multiple instances of Redis with automatic support for data sharding. Each node can have multiple read-only slave that act as replicas of the master node. In case of node failure, the other nodes can still operate normally because they communicate through binary protocol in the cluster bus to share information between them. Finally, Redis Sentinel monitors nodes and manages possible failovers without human intervention. However, Redis Cluster does guarantee strong consistency, because of asynchronous dumping on replicas, and is not really supported by a lot of clients.

1.4.1 Twemproxy

Twemproxy[20][19] is a fast and lightweight proxy for Memcached and Redis protocol. It was built primarily to reduce the number of connections to the caching servers on the backend. This, together with protocol pipelining and sharding enables to horizontally scale a distributed caching architecture.
Basically, Twemproxy can be used for sharding on Redis instances without the incompatible Redis clusterization and it is not a single point of failure because multiple proxies can be deployed.

1.5 MongoDB

MongoDB[18] is an open-source document database that stores data in JSON-like documents with a flexible architecture. Document-based database offer the possibility of a more complex data structure than standard key/value pair, however it is not completely schema-less. Each document is self-describing and there is no fixed central catalog where schemas are declared and maintained. MongoDB can also perform queries based on fields inside the documents to get only the selected documents. With a location field it is also possible to perform geospatial query.
Lastly, with MongoDB, horizontal scaling is made by sharding, which is a method for distributing data across multiple machines. The resulting shards are managed by routers that broadcast queries. If the shard key or the prefix of a compound shard key is known the router will target a subset of the shards to increase performance.

1.6 Cassandra

Apache Cassandra[17] is a free and open-source distributed NoSQL database management system designed to handle large amounts of data across many commodity servers, providing high availability with no single point of failure. Cassandra can scale horizontally through sharding, every node communicates with each other with gossip protocol. Also seed nodes can be defined in the cluster and are needed to bootstrap other nodes but does not possess any other special quality. Cassandra is a wide column store, and, as such, essentially a hybrid between a key-value and a tabular database management system. Its data model is a partitioned row store with tunable consistency. Rows are organized into tables; the first component of a table's primary key is the partition key; within a partition, rows are clustered by the remaining columns of the key. Other columns may be indexed separately from the primary key.

1.7 Ignite

Ignite[16] is a memory-centric distributed database, caching, and processing platform for transactional, analytical, and streaming workloads delivering in-memory speeds. Ignite is not schema-less so it is neither entirely NoSQL nor SQL. ACID transactions are supported only with key/value API and SQL transaction are atomic, but not yet consistent. Ignite instances can be clustered through sharding, so each node contains a set of the data. Moreover, Ignite can distribute queries sent to server to the designated instances and reduce multiple results into one, reducing serialization and transaction cost.

1.8 Comparison

Most of the analyzed Databases have high-availability and sharding policy, however some like Redis, MongoDB, Cassandra, and Ignite have natively a better approach due to the stored data inside of them that have no or little relation. For example, Neo4j, even though it allows sharding, can only divide natural chunks of the graph, or SQL databases must be designed ad-hoc in order to avoid transactions error.

An important feature of these databases is the geospatial API that allows to store locations inside the database. Often this is done with trees[6] that allow for very fast queries for the databases that supports them otherwise the cost of the query may be too high, if, for example, every single record must be checked. Through plug-in or natively most of the analyzed database can perform spatial queries, however if it is supported a level of complexity is removed, like Redis or MongoDB do.

In the end, performance in terms of latency and throughput must be considered a priority because the system may need to execute thousands of requests per second, since IoT devices are growing in numbers. In this field Redis is considered to be one the fastest since it is in-memory and a key/value pair database and its basic operations have a computational cost of O(1). However, if a schema, a complex data structure or persistence is needed on-disk database are a better choice. PostgreSQL, MySQL and MongoDB offer these features and since they have slower performance they can be used for historicizing.

Chapter 2

Message oriented middlewares

Since applications are distributed, a communication system must be deployed. Message oriented middlewares (MOM) are software infrastructure supporting sending and receiving messages between distributed systems. MOMs allow application modules to be distributed over heterogeneous platforms and reduce the complexity of developing applications that span multiple operating systems and network protocols.

Central reasons for using a message-based communications protocol include its ability to store (buffer), route, or transform messages while conveying them from senders to receivers. Another advantage of messaging provider mediated messaging between clients is the ability of monitoring and tuning performance by adding an administrative interface. Client applications are thus effectively relieved of every problem except that of sending, receiving, and processing messages.

2.1 Kafka

Kafka [10] is a distributed streaming platform. Kafka basically can be compared to an enterprise message-oriented middleware where producers and consumers exchange streams of data. When data is being consumed it is stored inside the Kafka cluster in a fault-tolerant way and can be processed.

Kafka core features, as in Figure 2.1, are Producer and Consumer APIs that

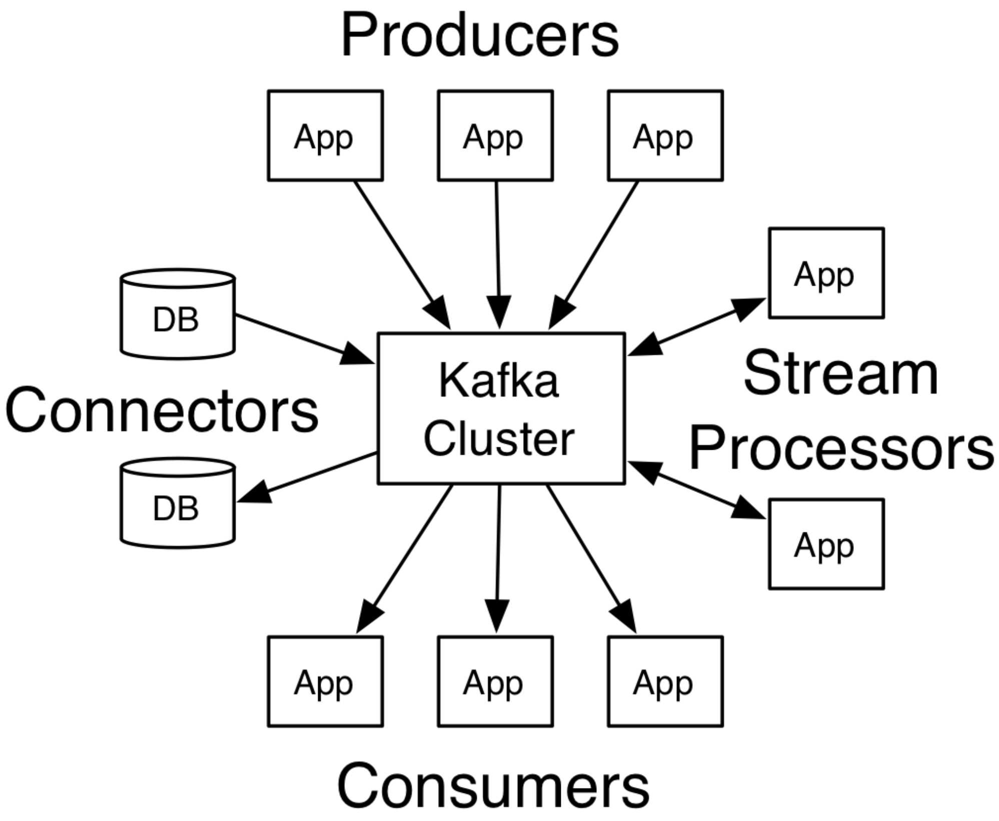

Figure 2.1: Kafka architecture, from [10]

allow applications to publish or subscribe to one or more topic. Connector APIs allow Kafka to move data across systems. Finally, the Streams API allows an application to operate on data by transforming an input stream into a output stream.

The core abstraction of Kafka is the Topic, as in Figure 2.2. A Topic is a category to which records are published and can have zero or many subscribers. A log of published records that are identified by an offset is kept for each Topic. Every consumer can define how to consume data through the offset, like stepping back to an old offset or skipping some to get newer data.
Kafka concept is a mix between a message queue and publish-subscribe model. It allows to divide processing data into more consumer instances, like queueing

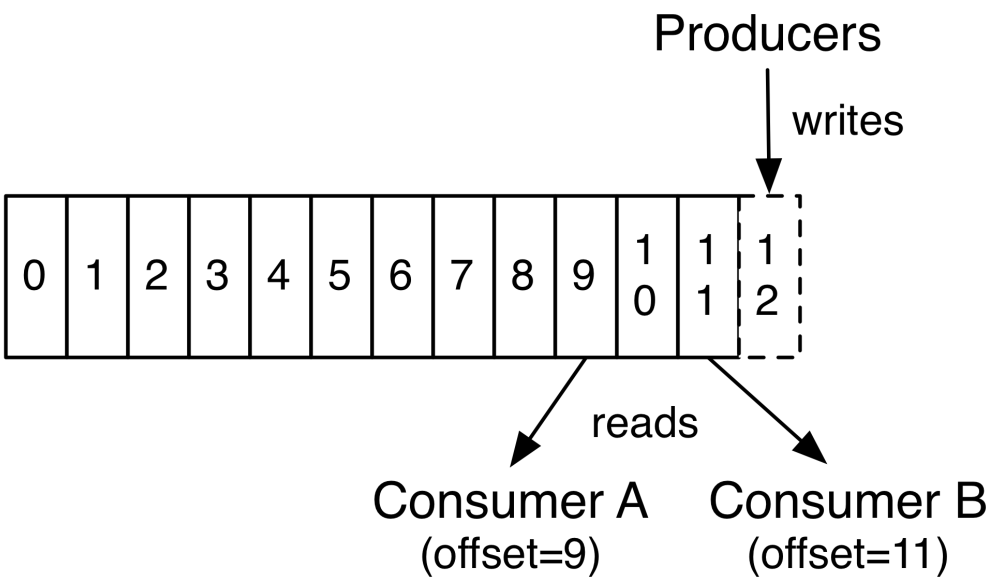

Figure 2.2: Topic architecture, from [10]

does, and allows to broadcast messages to multiple consumer groups, like publish-subscribe does. Every producer distributes messages on partitions within the topic, possibly with semantic partition function, creating a queue with defined offsets based on the order in which messages are sent. Consumers can label themselves with a consumer group name, where a certain record is delivered to one consumer instance in the group.
Kafka can also process and save data because producing and consuming are decoupled. Kafka saves all received messages on disk and replicates them for fault-tolerance. Before being consumed, this data can also be processed transforming input data into more elaborate or simpler output data.

2.2 RabbitMQ

RabbitMQ [11] is a messaging broker based on asynchronous messages. It supports several protocols such as AMQP, MQTT, STOMP etc. RabbitMQ can manage a high throughput of messages and can deploy flexible routing through *exchanges*. Also, a RabbitMQ server can be clustered even though it has not been designed to scale horizontally like Kafka[5]. Scalability can be improved a little with federation, where servers can transmit messages between brokers without clustering reducing the number of interconnection required by the normal cluster. Overall RabbitMQ is a fast and easy to deploy messaging broker with a lot of plugins and a very user-friendly interface. However, as it is, it does not support very high amount of ingress data streams and is not very durable. The RabbitMQ management plugin provides an HTTP API, a browser-based UI for management and monitoring, plus CLI tools for operators. RabbitMQ also provides APIs and tools for monitoring, audit and application troubleshooting. Besides support for TLS, RabbitMQ ships with RBAC backed by a built-in data store, LDAP or external HTTPS-based providers and supports authentication using x509 certificate instead of username/password pairs. These domains pose a challenge for Kafka. On the security front, Kafka added TLS, JAAS role-based access control and kerberos/plain/scram auth, using a CLI to manage security policy.

2.3 Comparison

Summing up, Kafka main characteristics are:

- Kafka has a higher throughput than RabbitMQ, around 100K/s against 20K/s;
- it keeps a history of messages after they are sent;
- allows to transform data through stream processing;
- Kafka is an event source for reactive applications;
- Kafka has been designed to scale efficiently and has been tested by many companies.

By contrast, RabbitMQ main characteristics are:

- it supports: AMQP, STOMP, MQTT;
- it allows finer-grained consistency control/guarantees on a per-message basis (dead letter queues, etc.);

- it offers variety in point to point, request / reply, and publish/subscribe messaging;
- it permits to create complex routing to consumers, integrate multiple services/apps with non-trivial routing logic;
- it offers better security and monitoring tools and API.

Chapter 3

Data processors

Once the system has a method to store and receive data, it needs a processing mechanism. The system should be able to consume data, and this is generally done in: batches or streams. Streaming applications consume data as a flow for continuous computation that can be in real-time. Each computation is supposed to be extremely short and independent, also knowledge of the data can be limited because the execution round depends on a small amount of data or one element. Batches applications, instead, have access to all data and are generally more concerned with throughput than latency of individual components of the computation; each computation may be complex and long.

3.1 Akka

Akka [12] is a set of open-source libraries based on the actor model and was born in response to the need of concurrency and parallelism in modern systems. Each actor encapsulates its own logic and communicates via asynchronous message passing. Instead, sending a message is non-blocking, so the actor can continue working. When a message is received, the actor reacts to it with a set of operations. Reacting to events, the platform can perform both stream and batch computations and is not limited to a set of standard operations, like MapReduce jobs, offering a fully customizable environment.

Akka provides the user with:

- multi-threaded behavior;
- transparent remote communication;
- cluster with high-availability.

Akka Cluster provides a fault-tolerant decentralized peer-to-peer based cluster membership service. Each member of the cluster is a node, which can be composed by zero, one or more actor. Cluster membership is communicated using a Gossip Protocol, where the current state of the cluster is gossiped randomly through the cluster, with preference to members that have not seen the latest version.
To correct possible errors during the gossiping a vector clock is used. If a node does not respond the other nodes can make a guess on its status. The Phi accrual failure detector is a mechanism that compares a series of factors to decide whether a node is up or down with a probability as result. If the Phi value is bigger than the user-defined threshold then the node is down. With the same mechanism, the nodes can detect if a node considered to be unreachable becomes reachable again.

3.2 Storm

Apache Storm [14] is a free and open source distributed real-time computation system. Storm focuses on streams of data and real time processing with a scalable and fault-tolerant environment and guarantees that data will be processed. Storm encapsulates its logic inside a topology, that is representable as an unending MapReduce job. A topology is a graph of spouts and bolts that relate to stream groupings as seen in Figure 3.1. A spout is a source of streams in a topology. Spouts read tuples from an external source and emit them into the topology. Spouts can either be reliable or unreliable. A reliable spout can replay a tuple if it failed to be processed by Storm, whereas an unreliable spout forgets about the tuple as soon as it is emitted. All processing in topologies is done in bolts. Bolts can do anything from filtering, functions, aggregations, joins, talking to databases, and more. A stream grouping defines how that stream should be partitioned among the bolt's tasks. Each spout or bolt executes as many tasks across the

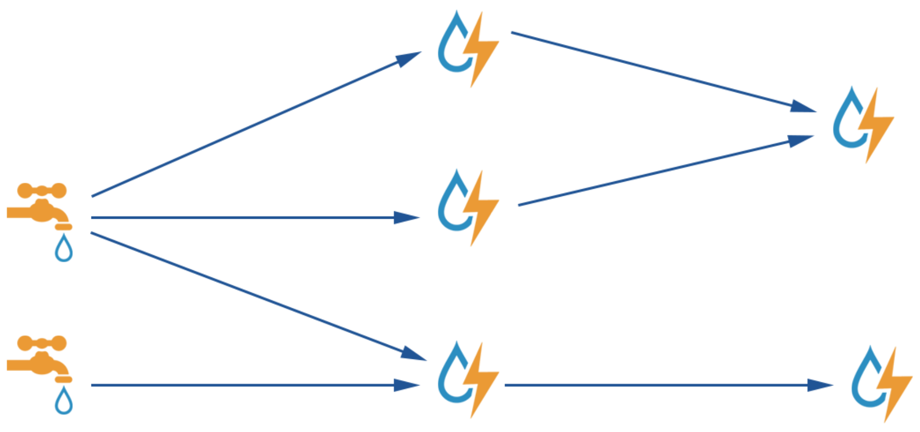

Figure 3.1: Storm flow, from [14]

cluster. Each task corresponds to one thread of execution, and stream groupings define how to send tuples from one set of tasks to another set of tasks. Storm guarantees that every spout tuple will be fully processed by the topology. It does this by tracking the tree of tuples triggered by every spout tuple and determining when that tree of tuples has been successfully completed. Every topology has a ”message timeout” associated with it. If Storm fails to detect that a spout tuple has been completed within that timeout, then it fails the tuple and replays it later.

3.3 Spark

Spark[13] is a general-purpose data processing engine that is suitable for use in a wide range of circumstances. Application developers and data scientists incorporate Spark into their applications to rapidly query, analyze, and transform data at scale. Tasks most frequently associated with Spark include interactive queries across large data sets, processing of streaming data from sensors or financial systems, and machine learning tasks. Spark works with batches of data so it does not work at best for real-time applications. One of Spark’s primary distinctions is its use of RDDs or Resilient Distributed Datasets. RDDs are great for pipelining parallel operators for computation. RDDs are, by definition,

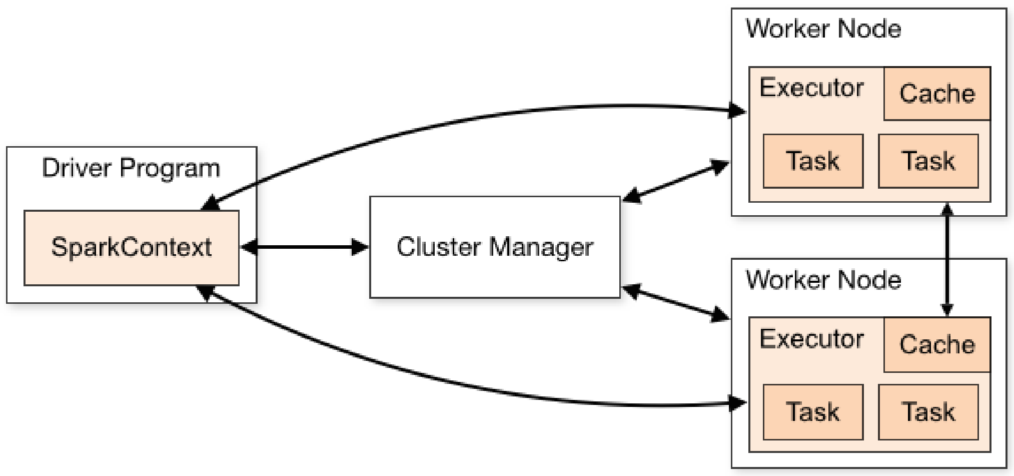

Figure 3.2: Spark architecture, from [13]

immutable, which provides Spark with a unique form of fault tolerance based on lineage information. Spark applications run as independent sets of processes on a cluster, coordinated by the SparkContext object as seen in Figure 3.2. Specifically, to run on a cluster, the SparkContext can connect to several types of cluster managers, either Spark's own standalone cluster manager, Mesos or YARN, which allocate resources across applications. Once connected, Spark acquires executors on nodes in the cluster, which are processes that run computations and store data for the application. Next, it sends application code to the executors. Finally, SparkContext sends tasks to the executors to run.

3.4 Comparison

The main difference between Spark and Storm is that the first computes batches of data, whilst the latter computes each single data received in a stream.

Storm is a good choice if sub-second latency and no data loss are needed. Spark is better if stateful computation is needed, with the guarantee that each event is processed exactly once. However latency is much higher because the computation is made on bigger amount of data than Storm, even when computing micro-batches in streaming mode. Storm has been used in production much longer than Spark. However, Spark has the advantages that it has a company dedicated to supporting it (Databricks) and it is compatible with YARN. YARN basically is a resource manager that splits up the functionalities of resource management and job scheduling/monitoring into separate daemons that were previously paired on Hadoop. The idea is to have a global ResourceManager and per-application ApplicationMaster. Meanwhile Akka is not bound by the computation method and it can perform both, stream and batch processing, being a reactive platform. So, Akka can achieve distributed data and task parallelism, but does not have natively resilience and fault-tolerance. Also, Akka does not guarantee message delivery so it is not the best choice when all the data must be analyzed at least once. Akka cannot really be compared to Storm and Spark, as a matter of facts Akka could be paired with them.

Chapter 4

Cluster Managements

Cluster management systems are tools that coordinate applications inside the cluster. Clusters are often inherently heterogeneous, so managing resources on machines with a single homogeneous interface can drastically reduce development and setup time. Inside the cloud/cluster a knowledge of the machines that compose the infrastructure is not required, so the best cloud architecture to choose is one that either runs application inside a containerized environment or a platform as a service that abstracts from the infrastructure.

4.1 Docker

Docker [8] offers a method to separate applications from infrastructure through containerization. A container is a package that contains an application and includes everything needed to run it. The result is an Image that consists in binary files, when an image is deployed a docker container is created. Every container will execute the same regardless of the environment. Docker running on a machine shares the operative system kernel and creates images using filesystem layers, also in case of pulled images or components, they can be reused. These perks grant optimization in start-up time and storage.
Docker differs from Virtual Machines especially on single machine as seen in Figure 4.1. VM on the same machine are allowed with a Hypervisor, which is computer software, firmware or hardware that manages VMs. Every VM contains a full-copy

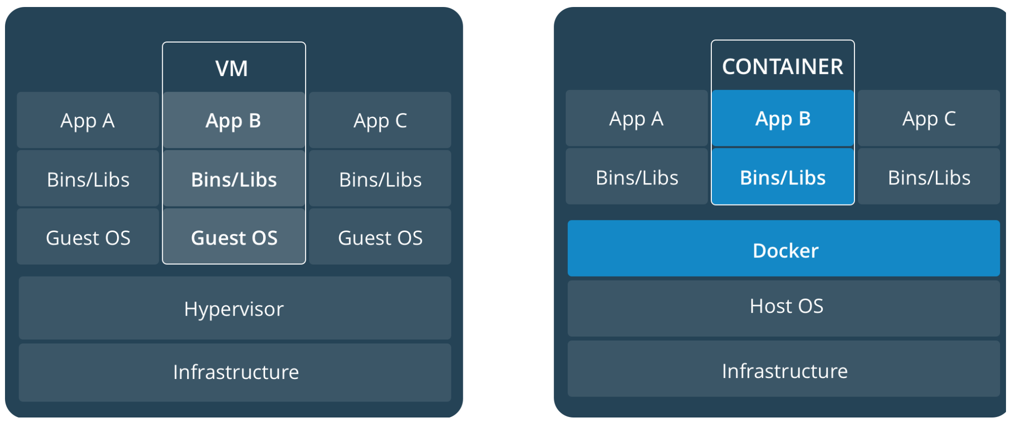

Figure 4.1: Container and VM architectures, from [8]

of an Operative System, where the applications are deployed. This procedure consumes a lot of storage capability and have a longer start-up time because the OS must boot before the application can start. Docker, sharing OS kernel, can directly deploy multiple application faster and without multiple OS copies.

However, Docker does not preclude a mixed approach, such as VMs with multiple container deployed on them.

4.2 Docker Swarm

Docker can be used as a cluster manager with Swarm Mode. With Docker Swarm is possible to deploy a scalable application service on different nodes. Nodes are multiple docker hosts that run in swarm mode and their roles can be worker, manager or both. Manager nodes receive definitions of services and dispatch tasks to worker nodes, while worker nodes execute these tasks. Manager nodes also do cluster management functions required to maintain the desired state of the swarm and elect a single node as a leader to conduct orchestration tasks. Tasks are simply containers with commands to run them and are the atomic scheduling unit. A group of tasks is a service and can be replicated on multiple nodes or executed globally.

4.3 Kubernetes

Kubernetes[9] is an open-source system for automating deployment, scaling, and management of containerized applications. Kubernetes main features are: portability, extensibility and self-healing capability.

Basically, Kubernetes offers a quick and predictable way to deploy applications thanks to containerization. Since containers are complete packages scaling them is only a matter of duplication in the Kubernetes environment. Updates can be performed seamlessly because the system is designed to update one service at the time inside the set of replicas maintaining functionality. Moreover, Kubernetes manages resources of every node.

Kubernetes does not create a new containerization system but relies on Docker. However, it offers a simplified management than docker swarm, since implements its own set of APIs. Kubernetes is not a traditional platform as a service system because because it works at a container level but it offers most of the common PaaS features with the flexibility of a Infrastructure as a Service.

Thanks to labels and annotations the user can define object metadata which can have no semantic meaning to the core system. Labels are key/value pair that users can define to organize and select subsets of object, while annotations can include additional metadata for tools and libraries but cannot be used to select objects. Furthermore, there is the Control Plane, which is a set of components that change according to the node role: Master or Worker, as seen if Figure 4.2. The Master node control plane makes global decision about the cluster and manages cluster events. Worker nodes, instead, wait for assignments from master nodes and manage pods and runtime environment. Usually, a Master node should focus on orchestration, but it can also perform as a Worker node. In this case both control planes components are active.

The core Kubernetes components used to build an application starting from the Docker images are:

- Nodes - as already described nodes are entities with resources that run pods,

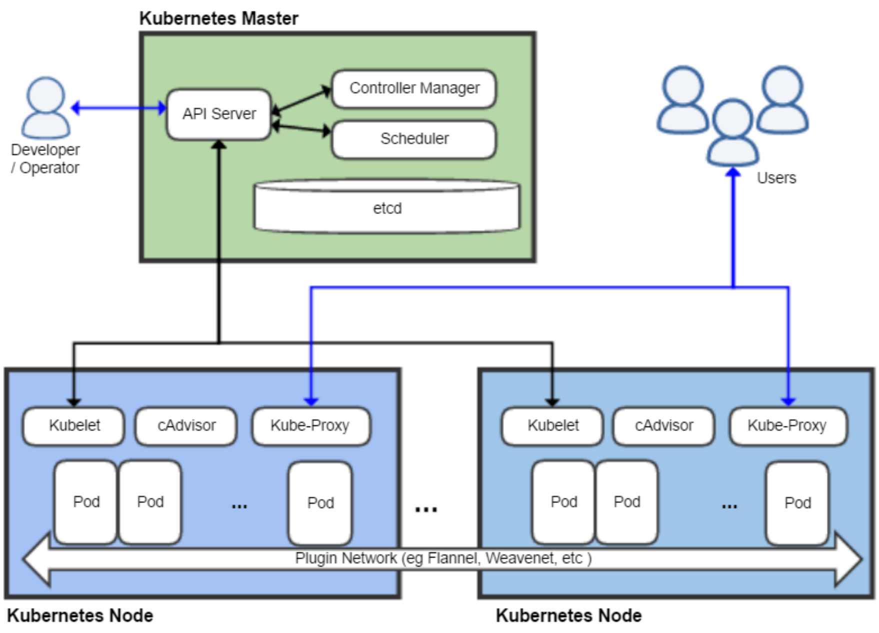

Figure 4.2: Kubernetes architecture, from [33]

which can be virtual o physical machine;

- Services - since pods are supposed to change, mostly when scaling or updating, when new pods are born or terminated services offer stable reference point for another pod that may need to connect. A Service is an abstraction which defines a logical set of pods and a policy by which to access them;
- Persistent Volumes - are abstractions of storage assigned by an administrator with specification of details of how storage is provided and consumed;
- Deployments - are controllers that manage pods and objects that provide a description of the desired state of an application. Inside a deployment are defined, through labels and annotations, all the necessary information to run a container and how the pod should be, including replica set;
- Statefulsets - are a level above Deployments and grant uniqueness and order of a set of deployment. Like a deployment, it describes how a pod should be in its desired state, but assigns a unique network identifier to applications

and provides ordered update, deployment and deletion;

- Pods - A pod is a group of one or more containers (such as Docker containers), with shared storage/network, and a specification for how to run the containers. A pod's contents are always co-located and co-scheduled and run in a shared context. A pod models an application-specific logical host in a pre-container world, they would have executed on the same physical or virtual machine.

4.4 Comparison

Both Docker and Kubernetes: are open-source, offer various storage options, provide networking features such as load balancing and DNS, and have monitoring add-ons like Heapster [4].

However, Kubernetes is based on extensive experience running Linux containers at Google and is deployed at scale more often among organizations. Kubernetes is also backed by enterprise offerings from both Google (GKE) and RedHat (OpenShift). Otherwise Docker swarm does not have many production deployment at scale. Moreover, Docker swarm is limited by the Docker API capabilities, while Kubernetes implements its own to improve the functionality. Also, Kubernetes can perform autoscaling based on factors, such as CPU utilization, while Docker swarm must scale them manually. Lastly, Docker swarm deployment is easier, and it is included inside the Docker Engine, whereas Kubernetes needs external tools for deployment that increase flexibility at the cost of simplicity.

Kubernetes leads on all metrics: when compared with Docker Swarm, Kubernetes has over 80% of the mindshare for news articles, Github popularity, and web searches. However, there is general consensus that Kubernetes is also more complex to deploy and manage.

Chapter 5

Aggregate programming

Commonly, IoT systems are created with a bottom-up approach that focus on the single device, where it is seen as the basic programmable unit. From this method can emerge a common behavior found in many existent systems that derives from the interaction of individual devices. However, this creates often tightly packed distributed application. When such applications grow in complexity, they tend to suffer from design problems, lack of modularity and reusability, deployment difficulties, and test and maintenance issues. Aggregate programming[26] is born as a paradigm in order to create a different approach to how modern systems for the IoT are designed. With this paradigm the basic unit of computing is an aggregate of devices that cooperates, abstracting from the single, so most of the local behaviors are abstracted away.

Aggregate computing needs a model of the domain of computation that comprise a topological model of the environment, the set of situated devices, and the context regulating which part of the environment affects computations at a device. The computational model that operates on this domain can be defined as a cycle as follows:

- Sensing: input phase that consists in the acquisition of information, mostly associated to the space-time structure of the system from the nearby devices or sensors.
- Analysis: the information is stored and analyzed in order to determine the

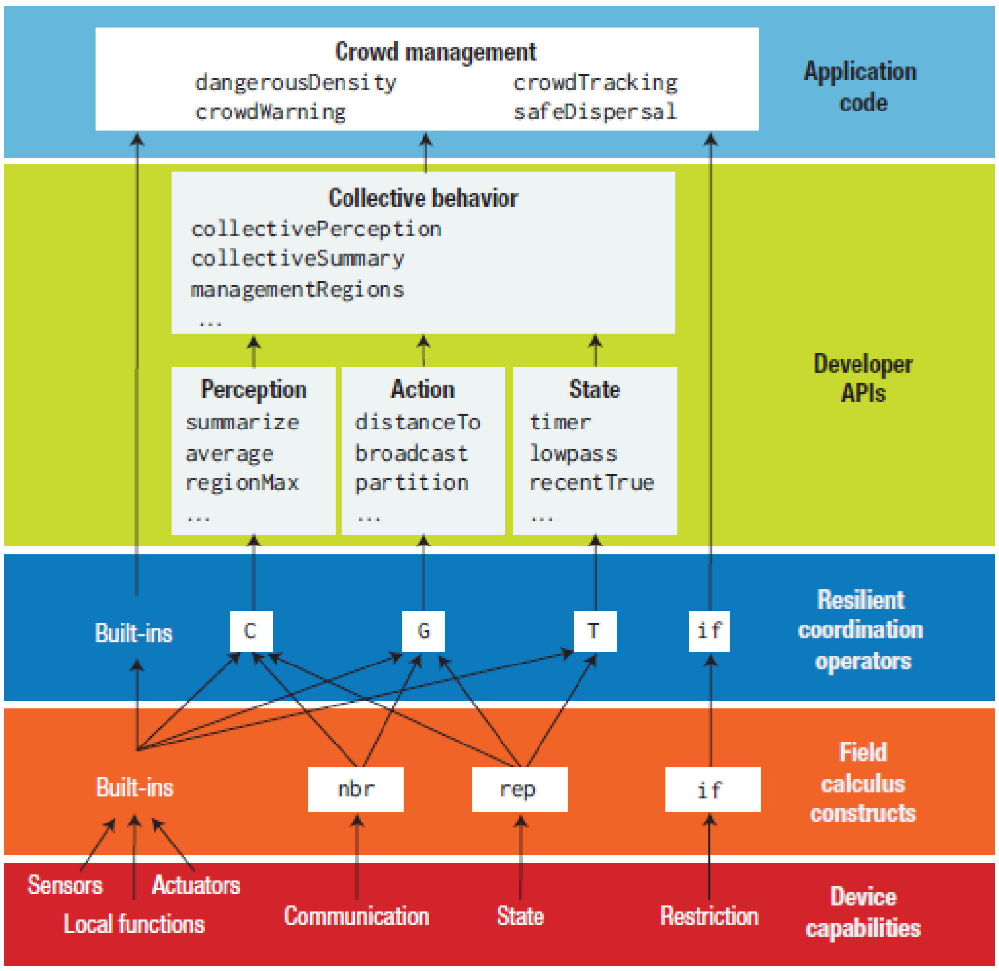

Figure 5.1: Aggregate programming abstraction layers, from [26]

context.

- Computing: creation of a new local state.
- Dispatching information: spread information across neighbors.
- Executing actuation: output phase that consists in a response of the computation.

Aggregate computing is based on the constructs of field calculus and resilient coordination operators, as shown in Figure 5.1.

Field Calculus is a set of operators created to manipulate computational fields. Where computational field is a map from a space into computational values. Interaction inside the field are bounded by the concept of neighborhood, that is given by the physical proximity of devices.

Resilient coordination operators are modular scalable self-stabilizing coordination mechanisms that can be used as building blocks for general purpose coordination patterns.

5.1 Scafi

Scala Fields (Scafi) [3] is a concise, easy-to-use, and modular framework for aggregate programming that includes an internal domain specific language (DSL) built in the Scala programming language. Scafi provides a syntax and the corresponding semantics for field calculus and implements all the basic constructs as Scala APIs. Moreover, Scafi includes a distributed middleware that supports the configuration of aggregate systems, according to different distributed computing scenarios, as well as their runtime execution. Scafi offers a performant Virtual Machine (VM) that implements field calculus semantics and can execute computational round.

In the aggregate program inside Scafi is possible to define behaviors. These can be simple operations on the computational field or high-level behavior defined as Scala methods. High-level behaviors, such as self-organization patterns and coordination algorithms, can be composed of multiple basic field calculus operations, thanks to the functional paradigm and the modularity of the components, creating more levels of abstraction. However, they can also be defined as monolithic methods for complex behaviors.

Chapter 6

Platform analysis

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

6.1 Case study

Even though this dissertation is made for research purposes a real case scenario has been evaluated.

In dangerous situations is often hard to coordinate devices on a large scale in real-time, especially when devices may fail in the local scale due to limited in-site processing capability. The system should focus on gathering data from numerous devices in a hazardous location outside of the local environment in a scalable and reliable method. Once the data from a device has been evaluated with Aggregate computing a response is sent back. The response is composed of an estimate of the risk based on a gradient emitted from the hazard source and a safe direction to follow. How the device reacts to the hazard should be historicized in order to improve the location escape routes.
Anyway a client-server aggregate computing system purpose should be defined by the aggregate program. So the architecture must be able to handle most of the

usual scenario.

6.2 Requirements

The case study does not explicit all the requirements the system should have, and since this application must support most of the scenario suitable for aggregate programming, some more requirements must be explicated.

List of requirements:

- Create a scalable, resilient, high-available and reusable system on cluster/-cloud.
- Collect data from IoT and standard devices in an efficient manner.
- Execute computation of sensor data through aggregate-programming with Scafi.
- Allow multiple configurations for message passing and computation location.
- Collect data on a distributed database with regards for current and historicized data.

6.3 Requirement analysis

Applications deployed in the cloud would be useless if those are monolithic and only vertically scalable. Often cloud and cluster solution are based on multiple virtual machines so making a horizontally scalable system is necessary. One of the main problem with scalable systems is a single point of failure. A resilient application should not have a single point of failure and it should react to the failure of nodes in order to repair the damage without interrupting the normal functionality. Finally, the system must be reusable, because deploying multiple applications for similar purposes increase the cost and time that go into a project.

Collecting data from devices is a primary problem because, without up-to-date normalized sensor data or exports, the system could not create a coherent domain. In a IoT environment, the number of devices connected to the system can be overwhelming and the application must not fail or lose messages because of the load. Also, devices may send data to the system without strict regulation of time and format, moreover the system can evolve and change over time. Obviously, modifying every single device is much more complex, so the data is processed inside the cloud in order to match system necessities. Filtering data allows for a faster and lighter routing inside the cluster. Since needed sensors for computation are known, the excess data from devices can be discarded. Instead, the necessary data can be transformed and aggregated in case it is not done by the device itself.

As said in Chapter 5, Scafi is based on the aggregate programming paradigm, so in order to function properly it has to perform a determined set of operations.

- Sensing;
- Analysis;
- Computing;
- Dispatching information;
- Executing actuation.

In a cloud-based computation, only sensing and actuation are the behaviors that remain on devices, while the others must be moved to the cloud. Then, the execution of a round for a device in the cloud is composed by the following set of operations:

- Gather the exports of the neighboring devices from the database;
- Retrieve contextual information from the sensor data;
- Compute a round;
- Store the export in the database;
- Dispatch the export back to the device.

The fact that the computation in the system is done with Scafi, does not mean that it has to be done only inside the cluster. The system should be flexible on where the computation is located. That means the devices should be able to decide where to locate computation, for example executing the aggregate program on the device or in the cloud.

The sensor data received, and the exports produced, must be stored inside the cloud anyway to keep data available. Since this data consists in the computational field and it must be accessed with little latency. Instead, for historicizing, the database can have higher latency, but data must be persistent. The storage must be distributed and high-available because, even though data is supposed to change rapidly, a node failure takes some time to get up to speed with an aligned computational field. According to the case study a downtime may result in hazardous situations.

Despite performance not being an explicit requirement, it is extremely important. The system must be designed in order to execute computations with a low latency, almost in real-time, otherwise the complexity added overall by a cloud application, with the aforementioned requirements, would not be justified.

6.4 Problem analysis

Since the computation is done with a framework that has already been tested, the most problematic part is how do components interact with each other in a cloud environment and their behavior.

The interaction between the devices and the cloud is the main part of the data ingestion system. This interaction can be managed in a lot of different ways. Messages sent by devices are considered a stream of data, losing one or more does not identify as an error. The system should work prevalently on newer data, so a lost message will be replaced by the next eventually. If a device does not send messages, it must considered down after a determined amount of time. In case the device status become up again the stream will restart and the computation will continue.

The device role can change how the interaction with the cloud happens; it can be either fire and forget or request response or a mix of both. Being the interaction based on messages the system is to be considered event-driven, meaning that computation is based on events created by messages. Event driven architectures have loose coupling within space, time and synchronization, providing a scalable infrastructure for information exchange and distributed workflows.

Specifically, the interaction with the cloud is based primarily on the execution mode of a device:

- distributed local computation;
- client-server support with local computation;
- client-server with server-side computation:
    - synchronous;
    - asynchronous.

Distributed local computation Since the beginning, Aggregate computing has been designed to be distributed on devices. To avoid reducing functionality, distributed computation between devices has not been blocked. However, it would be pointless to build a Cloud computing architecture if no device uses it. Then, being the computation location not static, some other device could use the distributed devices exports for computation. So, these devices only interaction with the server is to send their export, allowing neighboring devices to read them, as seen in Figure 6.1.

Client-server support with local computation While fully distributed, devices contain the computational field within themselves. Some devices could transfer this feature on the server for a lack of a communication protocol between themselves or resources, such as RAM or storage. Basically, these devices request neighbor's exports from the cloud in order to compute their export and then act like the previous case, sending the resulting export to the server, as seen in Figure 6.2.

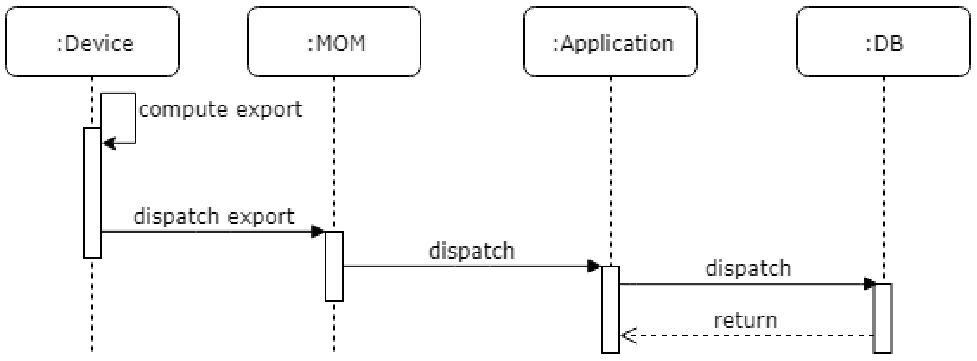

Figure 6.1: The device interaction with the cloud when the computation is executed locally. It sends the export to the server

Client-server with server-side computation With the execution on the cloud, the devices main interaction is sending sensor data to the system and perform actuation when the resulting export is received. However, it is not that easy, some basic sensor devices might not be able to read or does not need the export result. Also, who starts the computation must be decided.

If the device needs the export, the easiest method is a synchronous request reply messaging style, where the device sends sensor data and receive an export back, as seen in Figure 6.3.

However, devices can also send data and execution requests asynchronously, as seen in Figure 6.4, where the sensor data is sent with a fire and forget messaging style. Computation requests are still sent with request reply messaging style because an export result is needed.

Lastly, if the device does not need a response, the sensor data can be just sent with a fire and forget messaging style as well as computation requests, as in the first interaction shown in Figure 6.4. However, in this case the execution, if

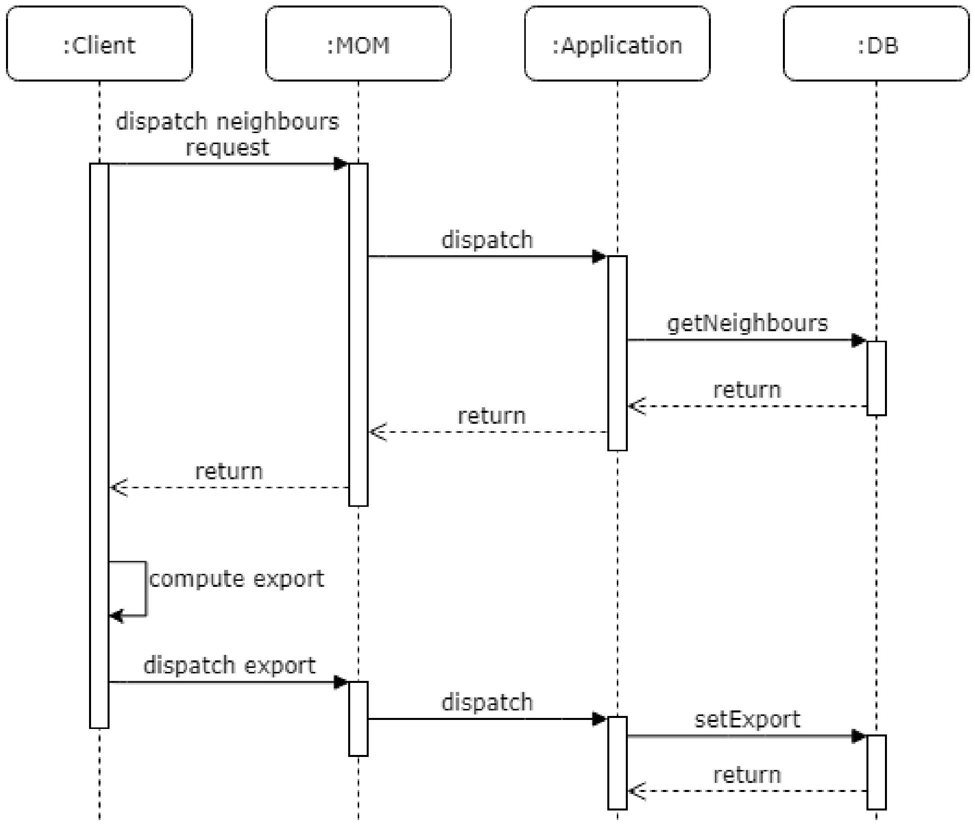

Figure 6.2: The device interaction with the cloud when the computational field is not stored locally. The device requests the neighbor's export from the server then sends the export

needed, can be managed with the following methods:

- the device sends a computation request, as seen in Figure 6.4;
- a node sends multiple computation requests for a set of devices;
- the server schedules a computation internally;
- another device requests computation, in order to compute passive devices if present in the neighborhood.

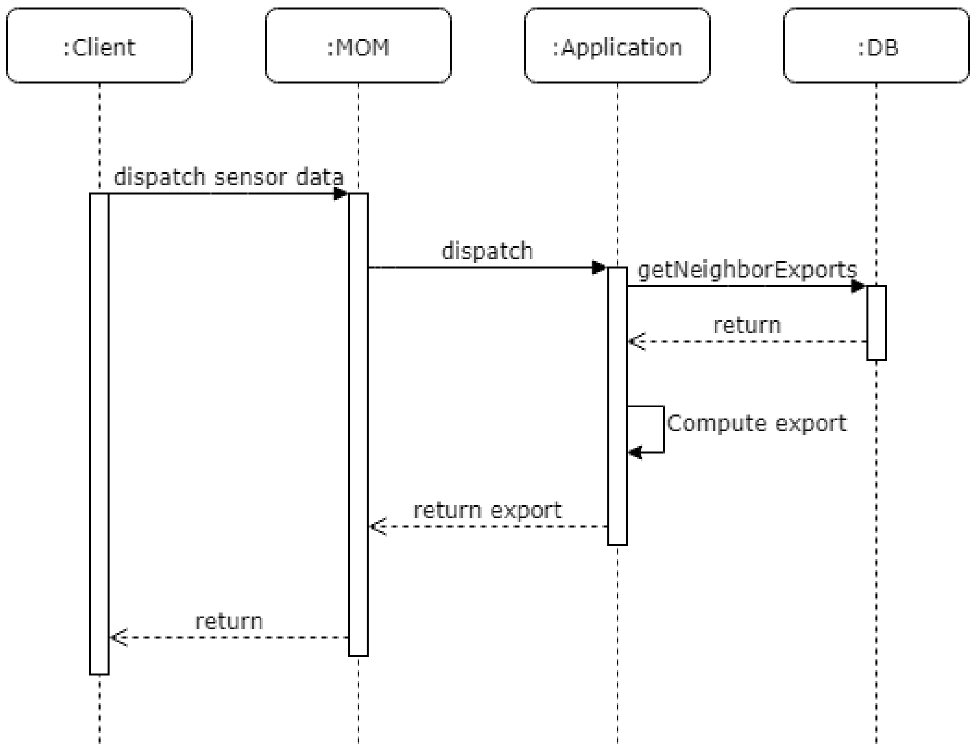

Figure 6.3: The device interaction when the execution is performed on the cloud. The device sends the sensor data and receive the export

6.5 Logic architecture

Since interactions have been described, a logic architecture of the components is defined, as shown in Figure 6.5.

The logic architecture is composed of four major parts:

- Database - that is where the sensor data and exports are stored;
- Application - the component that executes the aggregate program and uses the data storage;
- Data ingestion system - that is basically a message-oriented middleware for the platform;
- Device - a IoT, mobile device or robot that collects sensor data and performs

the actuation.

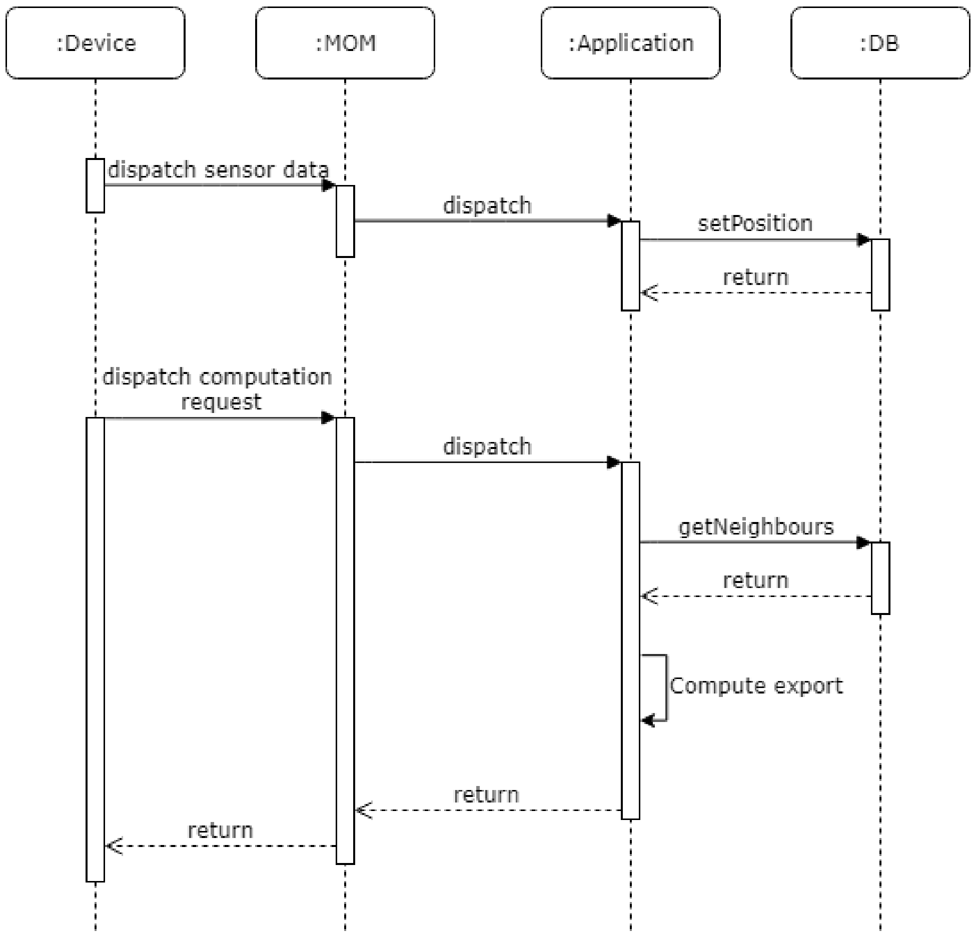

Figure 6.4: The device interaction when the execution is performed on the cloud with asynchronous sensor data storage. The device sends the sensor data then requests the export

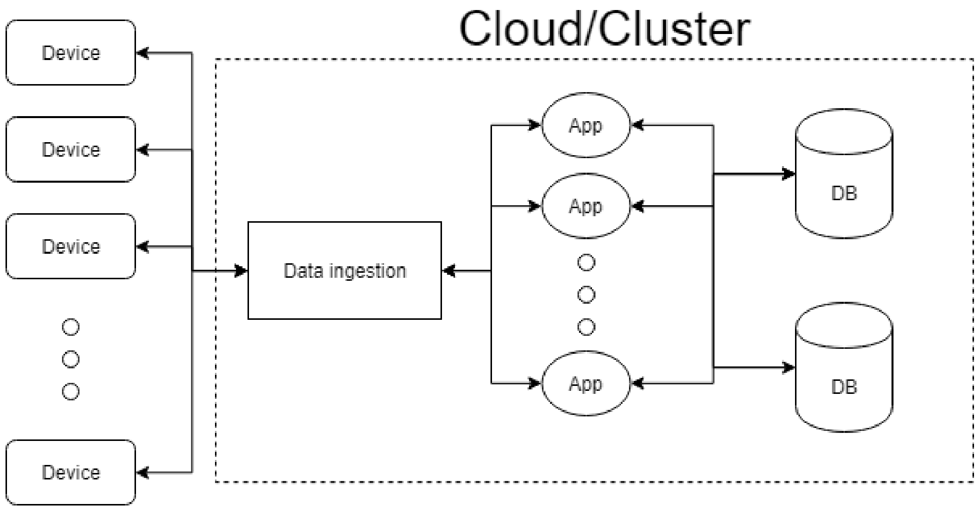

Figure 6.5: Logic architecture of the system structure

Chapter 7

Project architecture

In this chapter the technologies that have been chosen for the system will be described. The project architecture is defined starting from the logic architecture, as shown in Figure 7.1.

7.1 Application

From the requirement analisys has emerged that latency is a key component in the computation, so a stream approach is preferable, even though a stream is not to be considered a proper synonym of real-time it can be designed to approximate it. Since Akka is a reactive platform, it can be designed to react to every single piece of data it receives, performing like Storm does, becoming in fact a stream processor.

In order to satisfy the requirements, the application logic must be able to encapsulate the aggregate computing logic. Akka offers a reactive platform designed to work on a cluster where the application logic is not bound by any given rule. Other frameworks for Big Data analysis, like Storm and Spark, do not allow this freedom since they are designed for specific tasks that focus on map reduce operations. This allows Akka to encapsulate any kind of coordination method for the devices. Since Aggregate Computing is a requirement Scafi can be deployed without limitations. Given these facts Akka cluster is the best choice to satisfy the requirements.

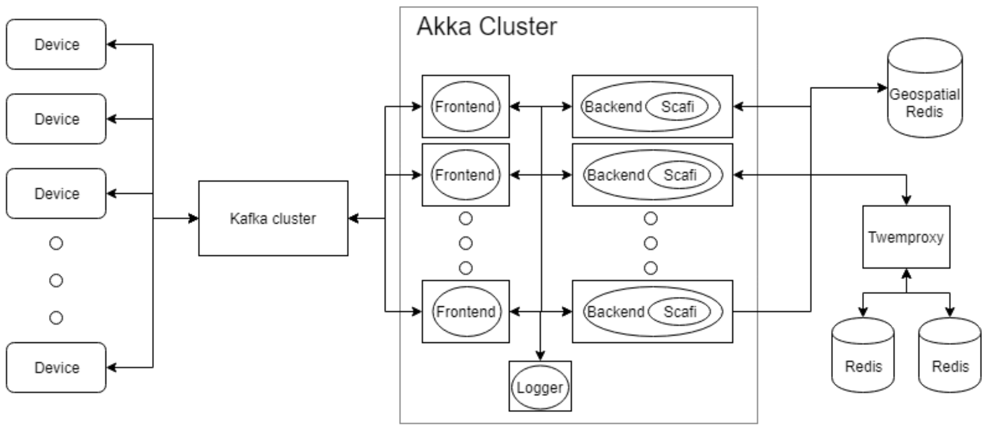

Figure 7.1: Project architecture

Lastly, a language to write the application is needed. Since Scafi was written in the Scala language and Akka provides API for Scala and Java, the application adopted the language. Scala[31], basically, is a modern multi-paradigm statically typed programming language that integrates both object-oriented and functional languages paradigm.

7.2 Data ingestion

For this component there is no explicit requirement regarding the technology in the data ingestion process, apart from the throughput, so Kafka has been chosen for its characteristics. Since the system must handle a high amount of IoT devices, a high throughput is necessary, and Kafka offers a tested architecture that can manage many messages in a distributed environment. With the API Kafka provides, it allows for customization in the system composition and future development. Especially the Stream API that allows for data transformation and cleaning, and the Connector API that can directly connect to databases. Moreover, as an event source Kafka can generate events for the reactive application built with Akka.

RabbitMQ has some other qualities that are not as significant as Kafka's, even though it excels in: applications that need variety in point to point messaging, request/reply messaging, publish/subscribe messaging and in advanced routing.

7.3 Database

The database selection is made accordingly to the system requirements. Two different databases are necessary to fulfil them.

7.3.1 Historicizing

In order to store historicized data permanently a persistent database is required. This database does not necessarily need to be deployed in the cluster with the requiremets set for the system, since it is not subject to heavy operations, at least by the main application. MySQL and PostgreSQL are both an optimal choice for historicizing data if a schema is needed. However, a faster approach is MongoDB which can store aggregate data inside JSON-like documents, like pairing the sensor data and the computation result in a single document.

7.3.2 Computational field storage

The computational field consisting in sensor data and exports of nodes must be rapidly accessed since hundreds of computations are performed per second and every for execution the application needs to build the context for the aggregate program. Redis is the best choice for this scenario, since its performances are best in class. Redis allows for multiple requests in batch or pipelining reducing downtime. Overall with the computational cost O(1), or O(M) in case of multiple requests, it can process a very high amount of transactions per second. Scalability of Redis nodes is managed through Twemproxy, since Redis Cluster is still not mature enough, sharding data between multiple Redis instances.

7.4 Devices

Devices can range from basic IoT devices, that only collect sensors data, to high-tech devices like smartphones or robots. Regarding the basic case study, devices like smartphones can translate the received export from the cloud into a graphic and textual user interface, so users can read the result. From a technological standpoint there is virtually no limitation on devices as long as they have a connection to the Internet.

Chapter 8

Implementation

In this chapter will be shown how the technologies, described in Chapter 7, are implemented and configured. Each component has been designed to be modular and as agnostic of the infrastructure as possible.

8.1 Akka-cluster

The main part of the application, that contains the business logic, is implemented with Akka Cluster.

Akka Remote Akka Cluster allows the creation of distributed actors that communicate between them through Akka Remote. Akka Remote is a configuration driven system that allows actors to send messages transparently with almost no API. Akka Remote can be defined in the configuration file, as shown in Code 8.1.

```
...
  remote {
    netty.tcp {
      hostname = ${HOST_NAME}
      port = 2551
    }
  }
...
```

Code 8.1: Akka-remote configuration

Akka Remote transparency has some problems regarding restrictions on what is possible. First, messages must be serializable. Serialization can be managed in many ways inside the cluster, like Java serializer, but the best approach in term of performance is the already included Protobuf. Protocol Buffers are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data. Protobuf can be set on the application configuration like Akka Remote, as shown in Code 8.2.

```
...
  serializers {
      proto = "akka.remote.serialization.ProtobufSerializer"
    }
    serialization-bindings {
      "model.DeviceMessage" = proto
      "model.ExecutionMessage" = proto
      [...]
    }
  }
...
```

Code 8.2: Serializers configuration

Inside a Protobuf model messages can be defined with fields that are composed of: a data type, a name and an id, as shown in Code 8.3. After the model has been created it must be compiled into Scala code in order to create/read them.

```
message ExecutionOnlyRequestMessage {
        int32 time = 0;
        int32 id = 1;
}

message StoreMessage {
        int32 time = 0;
    int32 id = 1;
    double x = 2;
    double y = 3;
    bool source = 4;
}

[...]
```

Code 8.3: Message model

Another problem of Akka Remote is that everything needs to be aware of all interactions being fully asynchronous. The probability for a message to be lost is higher due to network issues. In order to mitigate network issues two methods have been used. The failure detection system inside Akka cluster and an adaptive load balancing router. Inside the cluster an actor is monitored by the remote death watch, which uses heartbeat messages and a failure detector to generate *Terminated message* from network failures and JVM crashes, in addition to graceful termination. The suspicion level of failure is given by a value called *phi*. The basic idea of the phi failure detector is to express the value of phi on a scale that is dynamically adjusted to reflect current network conditions. The value of phi is calculated as:

$$phi = -log10(1 - F(timeSinceLastHeartbeat))$$

where F is the cumulative distribution function of a normal distribution with mean and standard deviation estimated from historical heartbeat inter-arrival times. If the PHI value is over a threshold the actor is a considered to be down.

Load Balancing Internal load balancing is done through routers inside the Akka Cluster, and every node has its own. These routers are cluster aware, meaning that they know which actor is up. Routers can be linked to a pool of actors with the same role in the cluster and custom routes can be defined. In order to deploy an adaptive load balancing the router must have metrics to analyze which node in under stress. This is done collecting Cluster Metrics. The member nodes of the cluster can collect system health metrics and publish them to other cluster nodes. The metrics can be colleted for the routers or by actors subscribed to the metrics extension. The route selection can be based on different metrics:

- Heap: Used and max JVM heap memory. Weights based on remaining heap capacity; $(max - used)/max$.
- CPU: CPU utilization in percentage, sum of User + Sys + Nice + Wait. Weights based on remaining cpu capacity; $1{-}utilization$.
- Load: System load average for the past one-minute, corresponding value can

be found in top of Linux systems. The system is possibly nearing a bottleneck if the system load average is nearing number of cpus/cores. Weights based on remaining load capacity; $1 - (load/processors)$.

- Mixed: Combines heap, cpu and load. Weights based on mean of remaining capacity of the combined selectors.

Routers can be defined in the configuration file as shown in Code 8.4.

```
...
akka.cluster.metrics.native-library-extract-folder=/app

akka.actor.deployment {
  /frontend/backendRouter = {
    # Router type provided by metrics extension.
    router = cluster-metrics-adaptive-group
    # Router parameter specific for metrics extension.
    # metrics-selector = heap
    # metrics-selector = load
    # metrics-selector = cpu
    metrics-selector = mix
    #
    routees.paths = ["/user/backend"]
    cluster {
      enabled = on
      use-role = backend
      allow-local-routees = off
    }
  }
}
...
```

Code 8.4: Router configuration

8.2 Actors

First, the actors must be started as shown in Code 8.5.

```
object Backend {

  private val CLUSTER_NAME: String = "ActorSystem"

  def main(args: Array[String]): Unit = {
    val config = ConfigFactory.parseString(
                "akka.cluster.roles = [backend]")
                .withFallback(ConfigFactory.load())
    val system = ActorSystem(CLUSTER_NAME, config)
    system.actorOf(Props[Backend], name = "backend")
    ActorMaterializer.create(system)
    val cluster = Cluster.get(system)
    val addresses = System.getenv().get("SEED_NODES")
      .split(",")
      .map(ip => new Address("akka.tcp", CLUSTER_NAME, ip, 2551))
      .toList
    cluster.joinSeedNodes(addresses)
  }
}
```

Code 8.5: Actor bootstrap

The cluster name, the actor role definition, and the actor logic to use are defined by the user directly as an object to bootstrap the actor. The cluster join is performed manually through the environment variable set in the deployment phase.

Now that the basic components are laid down, it is time for the actual business logic. The system is composed by three actor roles:

- Frontend Actor
- Backend Actor
- Logger Actor

And their internal composition, as explained in Section 8.1, is shown in Figure 8.1.

8.2.1 Frontend actor

Frontend Actors are consumers of the data ingestion system and basically manage internal load balance through the adaptive routers. Inside the actor, based on system configuration, messages for the Backend are generated from:

- Internal scheduling;
- Consuming data from the Kafka.

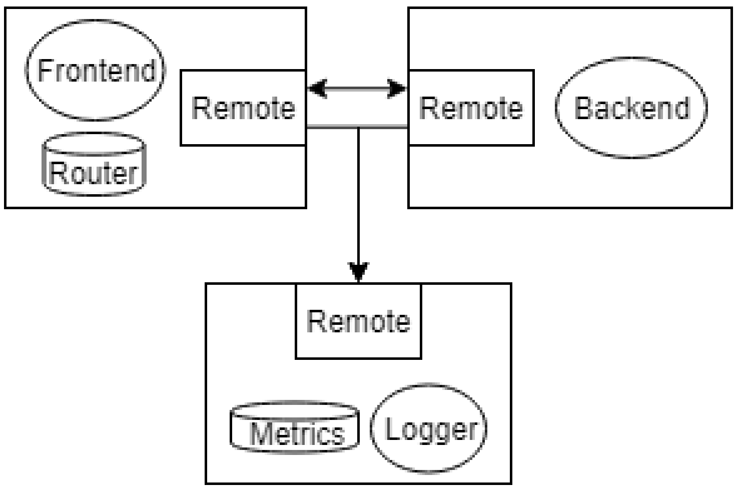

Figure 8.1: Basic actor composition

Scheduling Whenever the server should decide when to compute export for determined devices, scheduling is used as shown in Code 8.6.

```
context.system.scheduler.schedule(2.seconds, 50.millis) {
// Set initial delay and frequency
  val msg = ExecutionOnlyRequestMessage(
      System.currentTimeMillis(),devId)
  self forward msg
}
```

Code 8.6: Scheduler code

In this case the system must know which device should be executed a priori and the data must be already stored in the database. However, this scenario has mostly been used for testing and simulation purposes and it is not how the system has been designed to work. Since the application must be scalable, an equivalent of sharding for requests has to be deployed adding another level of complexity to efficiently run this use case.

Consuming Consuming messages from the data ingestion system, instead, is used to dispatch:

- Store data request - is the data received from device, it varies according to the aggregate program necessities and can be sensor data or an export; for the sensor data the minimum required is the position.

- Execution requests - is a request for the export computation made by a device; includes the device id.
- Execution request with sensor data - is a request for the export computation made by a device zipped with the required sensor data.
- Neighbor Export request - is a request of neighbors exports made by a device for local computation.
- User-defined auxiliary request - is a request defined by the user for external business logic; trigger events, debug, internal status report, etc.

Every message has been modelled to be serialized with Protobuf inside Akka and in the data ingestion system.

Producing When the actor receives the export from the backend, it produces a message for the device and sends it to the message-oriented middleware in the right topic. So the device subscribed to the topic can consume the result.

As seen in Code 8.7, requests that do not imply a computation, like storing data, can be performed directly by the frontend actor. Moreover, if a message is considered too old according to its internal time value, it is discarded. The threshold can be user defined or automatically adjusted based on workload. Lastly, the frontend dispatches messages to the backend through the adaptive router.

```
val backend: ActorRef = context.actorOf(FromConfig.props(),
        name = "backendRouter")
...
def receive = {
   case msg : ExecutionRequestMessage =>
          if(checkMessage(msg.time))
         backend ! msg
   case msg : ExportRequestMessage =>
          if(checkMessage(msg.time))
         backend ! msg
   case msg : StoreMessage =>
          if(checkMessage(msg.time))
         saveToDatabase(msg)
}
```

Code 8.7: Router usage

8.2.2 Backend actor

The Backend actor is the core of the business logic and contains the aggregate program. The aggregate program that is defined in the cluster must be the same on the devices with local computation or else there will be errors in the exports.

The aggregate program, shown in Code 8.8, calculates a gradient based on distance from a source node. The source node, according to the case study, is a source of danger.

```
class BasicProgram extends AggregateProgram
        with BlockG with StandardSensors {
  override def main(): Double = distanceTo(source)

  def source: Boolean = sense[Boolean]("source")
}
```

Code 8.8: Aggregate program

As said in Section 6.3, Scafi needs to perform a set of operations.

- Gather the export of the neighboring devices from the database;
- Retrieve contextual information from the sensor data;
- Compute a round;
- Store the export in the database;
- Dispatch the export back to the device.

Scafi can compute a round for a device through the neighbor's exports, that must be retrieved from the database as shown in Code 8.9.

```
  def getExports(id: Int, n: List[Option[GeoRadiusMember]]) = {
    var exports: Map[BasicSimulationIncarnation.ID,
        BasicSimulationIncarnation.EXPORT] = Map()
    n.collect { case e: Option[GeoRadiusMember]
        if e.get.member.get.toInt != id => e.get.member.get.toInt}
        .zip(getClient.mget(idList.drop(1), idList:_*)
        .getOrElse(List()).map(e => e.getOrElse("")))
        .par.foreach(value => {
          if (!value._2.equals("")) {
             exports += (value._1 -> jsonUtil
                       .createExport(value._2))
          } else {
             exports += (value._1 -> factory.emptyExport())
          }
       })
   exports
}
```

Code 8.9: Export retrieval

As seen in Code 8.10 the sensor data is retrieved from the database or directly from the message, so the context can be created, and the round executed.

```
val id = msg.id
val pos = getMsgPosition(msg)
val source = getMsgSensor(msg)

val neighbors = getNeighbors(id, pos)
val exports = getExports(id, neighbors)
val nbrRange = getNbrRange(id, neighbors)

val ctx = factory.context(id,
      exports,
      lsens = Map("POSITION" -> (pos._1, pos._2),
              "source" -> source),
      nbsens = Map("nbrRange" -> nbrRange))
val c = new BasicProgram()

val export = c.round(ctx)
```

Code 8.10: Scafi round

After that the export is stored in the database and is sent back to the Frontend if the device needs it.

In the case of network issues that results in multiple requests or if the devices do not respect the minimum delay to send requests, the system can decide, through a user defined threshold, that the node export stored in the database is still acceptable and ignore the computation round.

If there are passive nodes in the neighborhood that does not require an export, like a sensor device with no actuator, their export is never computed. The system is designed to include them in the computation. When a device finds some passive

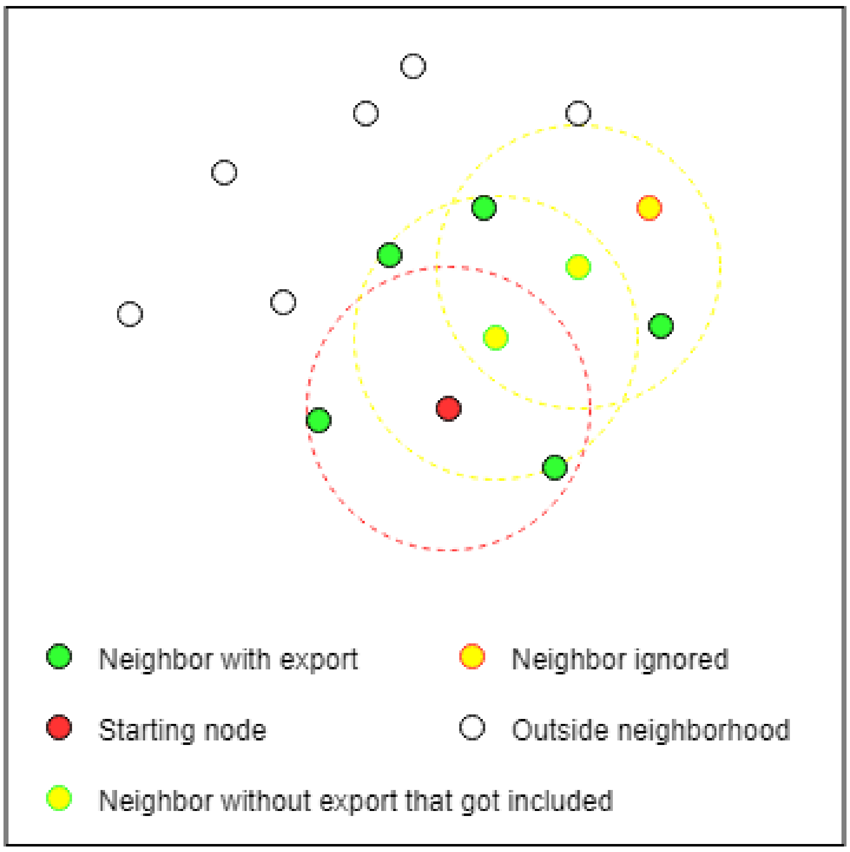

Figure 8.2: Computation of active sensor-only neighbors with an offset of two

node in the neighborhood, an extra round for each can be performed. This can recursively go on indefinitely, so an offset must be chosen, according to the known topology or heuristically, as shown in Figure 8.2 .

The main issue with Scafi in this application is the serialization of the export map. Since the map is not serializable, a method to store the export of a device in a serializable and language independent structure is needed, like JSON [29]. Mapping the export into JSON format allows it to be stored as a string in the database and to be serialized in the message. However, exports use the Scala Tuple class that cannot be created dynamically. So if the aggregate program utilizes tuples with cardinality bigger than three, the JSON conversion must be manually implemented. For the implementation of a JSON parser and mapper has been used json-lift [30], as shown in Code 8.11.

```
class JsonUtilLift {

  implicit val formats: DefaultFormats.type =
              net.liftweb.json.DefaultFormats

  def getJsonExport(export: ExportImpl): JValue = {
    val exportMap = export.getMap
    var list: List[Map[String, String]] = List()
    for ((p, v) <- exportMap) {
      var map: Map[String, String] = Map("path" ->
              p.toString.replace("P:/", ""))
      v match {
        case v: Double if v.isInfinity =>
          map += ("value" -> "Inf")
        case _ =>
          map += ("value" -> v.toString)
      }
      list = list ++ List(map)
    }
    decompose(list)
  }

  def getJsonExportString(export: ExportImpl): String =
              write(getJsonExport(export))

  def formatResult(result: String): Any = {
    if (result.equals("Inf")) {
      Double.PositiveInfinity
    } else if (scala.util.Try(result.toInt).isSuccess) {
      result.toInt
    } else if (scala.util.Try(result.toDouble).isSuccess) {
      result.toDouble
    }else if (scala.util.Try(result.toBoolean).isSuccess) {
      result.toBoolean
    } else {
      if (!result.toString.equals("()")) {
        val plessString = result.replace("(", "").replace(")", "")
        val tupleValues: Array[String] = plessString.split(",")
        if(tupleValues.length == 2) {
          (formatResult(tupleValues(0)),
                      formatResult(tupleValues(1)))
        } else if (tupleValues.length == 3){
          (formatResult(tupleValues(0)),
                      formatResult(tupleValues(1)),
                      formatResult(tupleValues(2)))
        } else {
          //TODO
      }
    } else {
      result
    }
  }
}

def createPath(string: String): Path = {
  var path: Path = factory.emptyPath()
  val pathStrings: Array[String] = string.split("/")
  for (stringElem <- pathStrings) {
    if (stringElem.length > 0) {
      val index: String = stringElem.substring(
                        stringElem.length - 2,
                        stringElem.length - 1)
      val tempString =
                stringElem.substring(0, stringElem.length - 3)
                        .trim
      tempString match {
        case "Rep" =>
          path = path.push(Rep[Any](index.toInt))
        case "Nbr" =>
          path = path.push(Nbr[Any](index.toInt))
        case "FoldHood" =>
          path = path.push(FoldHood[Any](index.toInt))
      }
    }
  }
  path
}

def createExport(input: String):
                BasicSimulationIncarnation.EXPORT = {
  val export: BasicSimulationIncarnation.EXPORT =
                factory.emptyExport()
  val json = parse(input)
  json.extract[List[JObject]].foreach(elem => {
    var result: String = null
    var path: Path = null
    elem.extract[JObject].obj.foreach {
      case field: JField if field.name.equals("path") =>
        path = createPath(field.value.extract[String])
      case field: JField if field.name.equals("value") =>
        result = field.value.extract[String]
    }
    if (result != null && path != null)
      export.put(path, formatResult(result))
  })
```

```
    export
  }

}
```

Code 8.11: JSON export mapping

8.2.3 Logger actor

The Logger actor is not a required element the system needs for normal operativity. This actor purpose is to collect metrics being subscribed to the metrics extension, as shown in Code 8.12.

```
  val cluster = Cluster(context.system)
  val extension = ClusterMetricsExtension(context.system)

  ...

  // Subscribe unto ClusterMetricsEvent events.
  override def preStart(): Unit = {
    extension.subscribe(self)
  }
```

Code 8.12: Metrics extension subscription

The actor then collects the metrics data received and periodically logs them, as shown in Code 8.13.

```
    case ClusterMetricsChanged(clusterMetrics) =>
      clusterMetrics foreach { nodeMetrics =>
        updateHeapMap(nodeMetrics)
        updateLoadMap(nodeMetrics)
      }
    case _: CurrentClusterState => // Ignore.
    case "logMetricsFormat" =>
      val (h, l) = calculateAverages
      log.info(System.currentTimeMillis + "," +
               h.mkString(",") + "," + l.mkString(","))
      heapMap = TreeMap.empty
      loadMap = TreeMap.empty
  }

  def calculateAverages: (Iterable[Long], Iterable[Double]) ={
    val heapAvg = heapMap.values map (nHeapUse =>
        if (nHeapUse.isEmpty) 0 else nHeapUse.sum/nHeapUse.length)
  val loadAvg = loadMap.values map (nLoad =>
      if (nLoad.isEmpty) 0 else nLoad.sum / nLoad.length)
  (heapAvg, loadAvg)
}

def updateHeapMap(metrics: NodeMetrics): Unit = metrics match {
  case HeapMemory(address, _, used, _, max) =>
    val updatedHeapUse: Seq[Long] = heapMap
            .getOrElse(address, Seq.empty) :+
        (used.doubleValue / 1024 / 1024).toLong
    heapMap += address -> updatedHeapUse
    maxHeap = max.getOrElse(0)
  case _ => // No heap info.
}

def updateLoadMap(metrics: NodeMetrics): Unit = metrics match {
  case Cpu(address, _, Some(systemLoadAverage), _, _, _) =>
    val updatedHeapUse: Seq[Double] = loadMap
              .getOrElse(address, Seq.empty) :+
              systemLoadAverage
    loadMap += address -> updatedHeapUse
  case _ => // No heap info.
}
```

Code 8.13: Metrics collection

8.3 Redis

Redis has been chosen due to its performance as a key/value pair in-memory database. Also, it is very easy to deploy for prototyping purposes.

As a matter of facts, the database could be easily changed with another one, as long as it is horizontally scalable and allows geospatial queries like MongoDb or Cassandra.

Spatial data Whenever the Backend actor receives a message, an interaction with the database must happen. The first interaction is always related to the device position in the space. With Redis, the position is stored into the key as a sorted set in a way that makes it possible to retrieve items using a query by radius. Also, different spatial abstraction can be implemented in the database changing the key, allowing multiple aggregate program to work on a subset of nodes.

To add an element to the set the command shown in Code 8.14 is used.

```
GEOADD key longitude latitude member
        [longitude latitude member ...]
```

Code 8.14: Geoadd command

Instead, to retrieve a set of points inside a radius there is the command shown in Code 8.15.

```
GEORADIUS key longitude latitude radius m|km|ft|mi
        [WITHCOORD] [WITHDIST] [WITHHASH] [COUNT count]
        [ASC|DESC] [STORE key] [STOREDIST key]
```

Code 8.15: Georadius command

The results can also be packed with the distance from the origin if it is needed, like in the Code 8.10 when nbrRange is created.

If a device becomes inactive, there is no way to make a spatial record expire after some time. If the device is found to be inactive or disconnected, to avoid useless computations in the next phase, the position is removed with the command shown in Code 8.16 .

```
ZREM key member [member ...]
```

Code 8.16: Zrem command

Sensor/Export data Then, the database is accessed to store: exports and sensor data. The data is added to the database with the command shown in Code 8.17.

```
SET key value [EX seconds] [PX milliseconds] [NX|XX]
```

Code 8.17: Set command

With this command is possible to set an expiration value, so inactive nodes can be detected and removed when a read operation occurs.

Subsequently, with the command shown in Code 8.18, is possible to get data for each device retrieved with the georadius query.

```
MGET key [key ...]
```

Code 8.18: Mget command

As a client, scala-redis [28] allows the connection with the database and the support for every basic command. The connection is enstablished through the database IP and port, as shown in Code 8.19.

```
RedisClient("redis01", REDIS_PORT).set(id, currentDeviceExport)
```

Code 8.19: Scala-redis connection example

Twemproxy Since Redis Cluster is not supported by most of the clients, sharding is entrusted to a proxy. Twemproxy is a proxy for Redis that allows to manage sharding on a single point outside the application. Sharding can be configured deterministically in order to create multiple instances of Twemproxy for extra scalability.

Twemproxy can be configured with a configuration file where Redis servers are listed as in Code 8.20.

```
alpha:
  listen: 0.0.0.0:22121
  hash: murmur
  distribution: ketama
  auto_eject_hosts: true
  redis: true
  server_connections: 1
  server_retry_timeout: 30000
  timeout: 4000
  server_failure_limit: 3
  preconnect: true
  servers:
    - redis02:6379:1 redis02
    - redis03:6379:1 redis03
```

Code 8.20: Twemproxy configuration

Historicization If historicizing is needed, another database can collect changes on the database, possibly without interrupting the normal operation of the system, for example reading dump files on disk. Redis dumps data asynchronously on disk to restore data in case of node failure. Most likely, data could be sent asynchronously after a computation round or in batch directly since no response is needed for the execution to proceed.

8.4 Kafka

Now there is no real implementation of Kafka in the prototype. However, the Kafka client can consume messages through Akka stream. Currently, clients for Kafka are based on Akka for reactive behaviors, for example Akka Streams Kafka [32].

Being an event source, Kafka can trigger the Frontend actor reaction dispatching the request received. In order to efficiently consume all the requests, those must be published to the same topic in order to exploit the automatic load balance given by the consumer groups. This allows the Frontend actor to distribute the requests consumption. The flow of consumption is non-blocking and does not stop when the load surpasses the throughput. It is the device duty to decrease the requests number and the Frontend should detect which request to ignore, based on the request timestamp.

If the system cannot handle all the messages, newer ones are prioritized. Also, if the system is configured with decoupled asynchronous sensor data and execution request messages, Kafka can be configured to send the data directly to the database through the Connector API. In that case Redis can be considered directly as a cache for the latest data.

Since the system is going to evolve and, possibly, change aggregate program, Kafka can transform, aggregate and clean the data received with the Stream API.

8.5 Devices

The devices that have been considered during implementation phase are:

- Smartphones - that send the sensor data to the cluster and await a response that contains the computation export. The frequency on which requests are sent is a result of the response time plus a delay, defined in the system configuration. Moreover, the response can contain a load indicator that increase the delay on less important devices. If a response is not received, after a timeout, the request is dispatched again.
- IoT devices - that send the sensor data and do not require an export since there is no user interface.
- Emergency devices - that are devices used by an emergency response team and are similar to smartphones. However, these devices have a different application logic, for example the device shows the direction to follow in order to reach the emergency, instead of how to avoid it.

8.6 Use cases

In this section are presented some of the complete use cases, defined with sequence diagramas, similarly to the diagrams listed in Section 6.4. These use cases do not cover every single possible interaction, because some are derivable or are simpler versions of the one listed here. For example, the use cases, where no computation export is needed as response, are practically identical to the one where is needed, but without the segment that represents the export response message.

8.6.1 Synchronous client-server

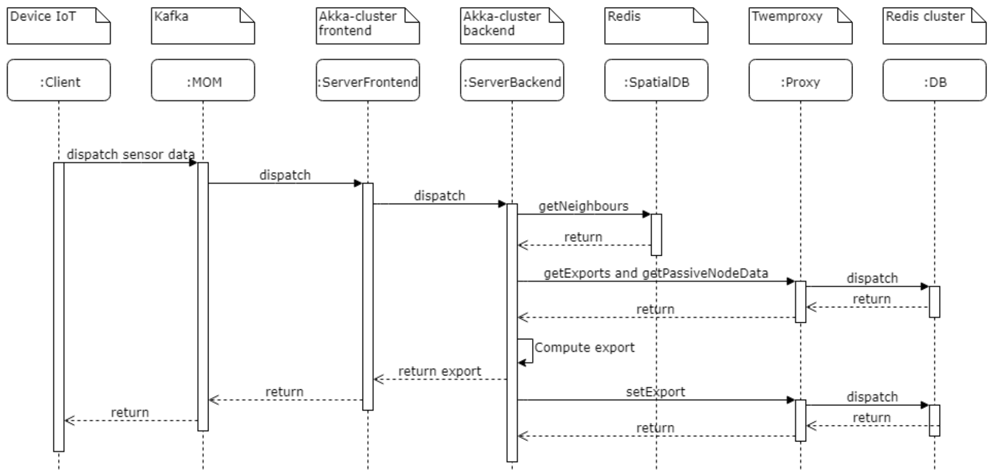

Figure 8.3: Workflow of a request/reply execution with the computation request and data zipped.

8.6.2 Asynchronous client-server

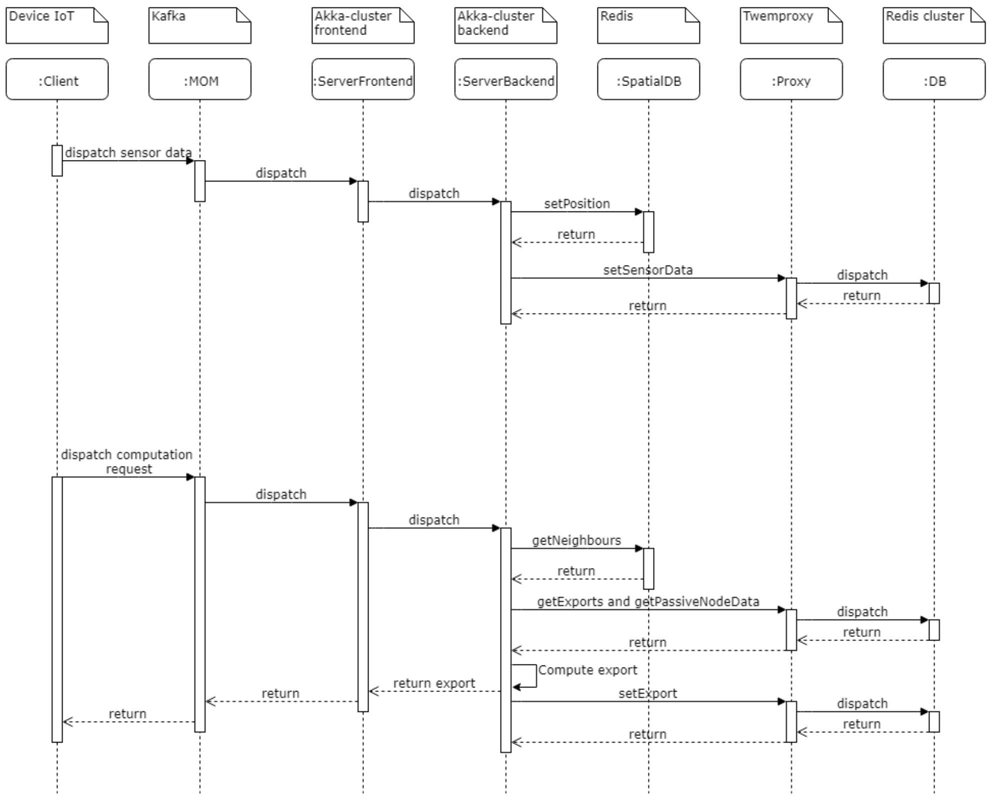

Figure 8.4: The workflow of a request/reply execution with the sensor data sent asynchronously.

8.6.3 Distributed

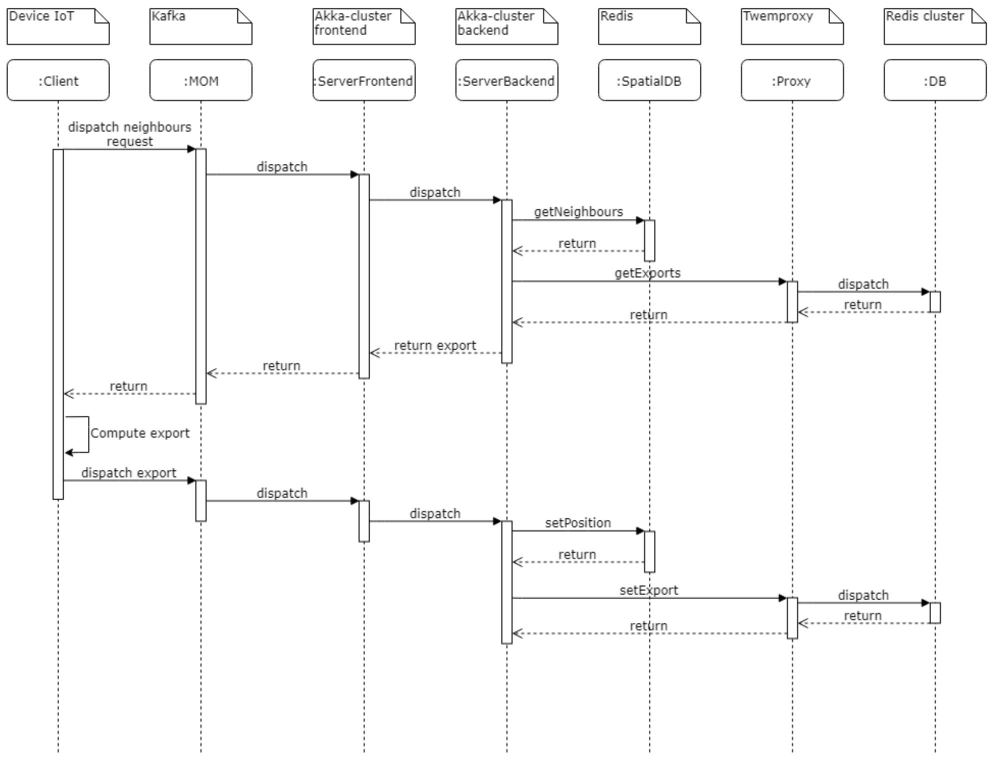

Figure 8.5: The device makes a neighbor's exports request for local computation then sends the export to the server.

If the device is directly connected to the neighbors then only the second part of the Figure 8.5 is executed.

8.6.4 Simulation

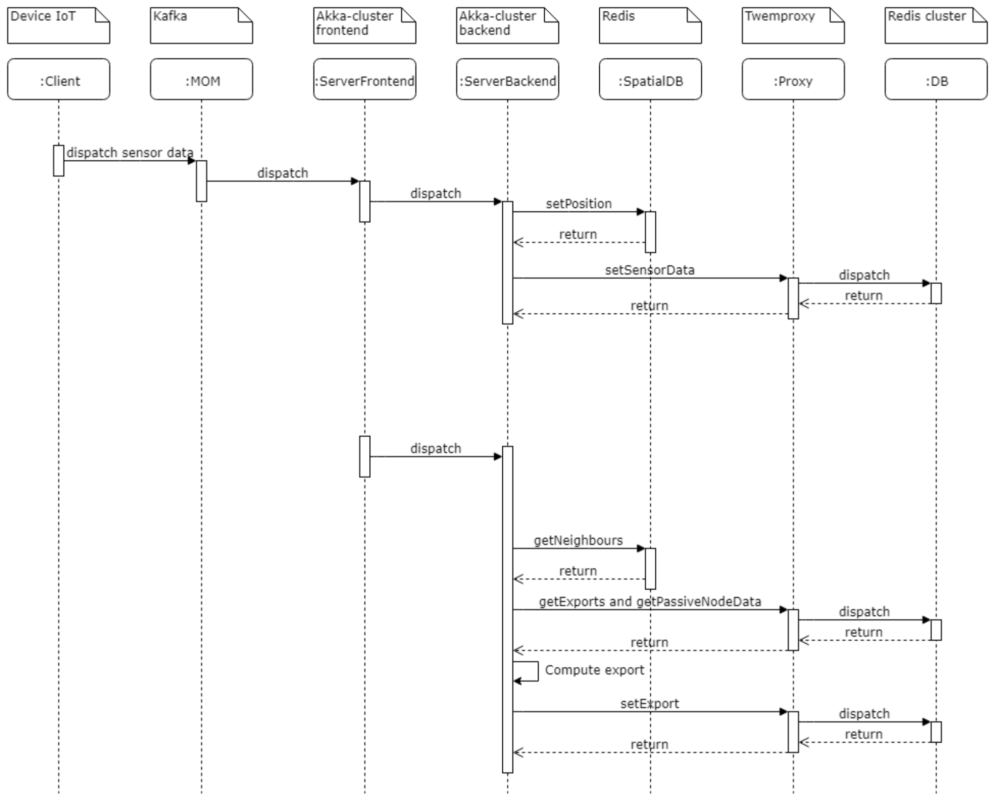

Figure 8.6: Scheduling a device computations on the frontend for simulation purposes.

Chapter 9

Deployment

The deployment phase is going to describe how to wrap the system inside containers and how to deploy them in the cloud environment.

9.1 Infrastructure as code

With Kubernetes as a deployment platform, the infrastructure of the system is definable by code. Infrastructure as Code [27] is an approach to using cloud era technologies to build and manage dynamic infrastructure, which is basically treated like a software system and its elements are compared to data.

The principles of the Infrastructure as code ideology are:

- Reproducibility - the system must be reproducible effortlessly.
- Consistency - the configuration of more elements that implements the same service must be nearly identical.
- Repeatability - every action performed must be repeatable.
- Disposability - the elements are supposed to be destroyed and recreated.
- Service continuity - the service must be available even if some elements are destroyed.
- Self-testing systems - the test must be performed automatically for every change.

- Self-documenting systems - the scripts and tools used already contains the description of the process they are used for.
- Small changes - every small change that can incrementally be made should be tested and pushed into use.
- Version all the things - every version of the system is traceable and can be used to improve the system.

9.2 Continuous delivery

Continuous delivery is a practice that consists in integrating and testing every single version in a production-like environment, with the same tool that would be used in a real production environment. In fact, production-like environment does not mean identical to production. The important point is that the test environment emulates the key aspects of the production environment, including the various complications and complexities. There must be no extra privileges or condition that would not have been allowed in a production environment, otherwise the practice becomes exactly like Continuous integration. Continuous Integration, similarly to Continuous delivery, is a practice that integrates and tests every small change produced.

With Kubernetes, every version pushed is tested in a production-like environment, either recreating the system or through the update mechanism embedded in the platform. This allows for every little change to be pushed to the actual production environment. However, Continuous delivery differs from Continuous deployment, where every change committed is applied to production immediately after passing automated tests. The point of Continuous delivery is not to apply every change to production immediately, but to ensure that every change could be applied.

9.3 Environment

First, a Kubernetes environment must be created and that can be achieved in many ways, but only three methods have been considered for this prototype.

Public cloud Some public clouds already include a Kubernetes environment engine, most notably the Google Cloud Platform since Kubernetes development is based on Google's. Also, Amazon AWS and Microsoft Azure, etc. can support Kubernetes if deployed correctly on the infrastructure. The initialization phase varies for each provider.

Minikube Minikube offers a fast and easy to deploy prototyping environment and is a single node cluster deployed inside a virtual machine. With the command shown in Code 9.1 the virtual machine is started without any configuration needed.

```
Minikube start
```

Code 9.1: Minikube start command

In order to work Minikube needs a docker daemon that must point inside the envirnment and a working instance of Kubectl. Docker can be configured to point to the Minikube environment with the command shown in Code 9.2

```
eval $(minikube docker-env)
```

Code 9.2: Docker environment

However, Minikube does not support multi-node clusters and cannot be used for production, so it can only serve as a development platform.

Kubeadm Kubeadm is a toolkit that helps bootstrap a best-practice Kubernetes cluster in an easy, reasonably secure and extensible way. Kubeadm must be started on the master with the command shown in Code 9.3.

```
kubeadm init
```

Code 9.3: Kubeadm init

The initialization workflow is composed by many phases:

- Pre-flight checks - to validate the system state before making changes.
- Generates a self-signed CA - to set up identities for each component in the cluster.
- Writes kubeconfig files - to connect to the API server.
- Generates static Pod manifests for the API server.
- Apply labels and taints to the master node.
- Generates the token that additional nodes can use to register themselves with the master in the future.
- Makes all the necessary configurations for allowing node joining.
- Installs the internal DNS server (kube-dns) and the kube-proxy addon components via the API server.

After the master node has been initialized, every worker node must join the master with the command shown in Code 9.4. This action is performed bidirectional trust between the master and the worker.

```
kubeadm join --token <token> <master-ip>:<master-port>
                --discovery-token-ca-cert-hash sha256:<hash>
```

Code 9.4: Kubeadm join

Kubeadm does not contain a pod network by default, so one must be installed. Currently six are available: Calico, Canal, Flannel, Kube-router, Romana and Weave Net. For this deployment Calico has been chosen and can be deployed with the command shown in Code 9.5.

```
kubectl apply -f *Calico_Url*/calico.yaml
```

Code 9.5: Initialise Calico

However, Kubeadm has some limitations, the cluster created has a single master, with a single etcd database running on it. This means that if the master fails, the cluster may lose data and may need to be recreated from scratch. Adding HA support (multiple etcd servers, multiple API servers, etc.) to Kubeadm is still a work-in-progress. So Kubeadm is not the best solution for production but is better than a single node cluster like Minikube and can be used as a production-like environment .

9.4 Docker image

The deployment of the application inside the Kubernetes environment is made with Kubectl through declarative yaml files. In order to deploy an application a docker image is required that can be pulled from a local repository or online. First of all, docker images of the components must be created. Docker daemon must be installed in every node to work and if a local repository is used the application can be built with the command shown in Code 9.6

```
docker build -t repository/imagename pathToDockerfile
```

Code 9.6: Docker build

A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image. Using docker build users can create an automated build that executes several command-line instructions in succession.

```
FROM frolvlad/alpine-oraclejdk8

ENTRYPOINT ["java" ,"-jar", "/app/app.jar"]

ADD ./target/scala-2.12/Cluster-assembly*.jar /app/app.jar
```

Code 9.7: Dockerfile

- FROM: The FROM instruction initializes a new build stage and sets the Base Image for subsequent instructions.
- ENTRYPOINT: An ENTRYPOINT allows you to configure a container that will run as an executable.
- ADD: The ADD instruction copies new files, directories or remote file URLs from the source and adds them to the filesystem of the image at the destination path.

9.5 Kubernetes Deployment

Deployments Components of the application are mainly deployed as Kubernetes Deployments. Deployment is a controller that create pods according to a yaml file and maintain the desired state inside the cluster.

```
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  labels:
    run: backend-worker
  name: backend-worker
spec:
  replicas: 5
  selector:
    matchLabels:
      run: backend-worker
  template:
    metadata:
      labels:
        run: backend-worker
    spec:
      nodeSelector:
        role: master
      containers:
      - name: backend-worker
        image: repo/cluster
        imagePullPolicy: Never
        command: ['/bin/sh', '-c', 'java -cp /app/app.jar Backend']
        env:
        - name: SEED_NODES
          value: backend-seed-0.backend-seed,
                 backend-seed-1.backend-seed
        - name: HOST_NAME
          valueFrom:
            fieldRef:
              apiVersion: v1
              fieldPath: status.podIP
        #resources:
        #  requests:
        #    memory: 512Mi
        #    cpu: 1
        readinessProbe:
              tcpSocket:
                port: 2551
              initialDelaySeconds: 10
              timeoutSeconds: 120
            }
        ports:
        - containerPort: 2551
```

Code 9.8: Deployment file

Inside the Deployment are defined a lot information using labels and selector. Firstly, metadata are defined, then the specification. Inside the specification are defined the number of replicas and the template of the pod. Lastly inside the template are defined all the information to run the container and volumes claim. Container labels:

- Image: the docker image created with the Dockerfile and the command to run to start execution
- Environment variables: variables that can be accessed from inside the container.
- Resource request: requested resources and/or limit of resources claimable in order to grant the pods enough resources to run or to block excess usage.
- Readiness/liveness probe: are probes that Kubernetes uses to control readiness and liveness of a container; unhealthy pods are restarted.

Replication in the cluster can be managed with the replica label or directly by kubectl with the command shown in Code 9.9

```
kubectl scale Deployment -replicas=X
```

Code 9.9: Scaling mechanism

Statefulset At startup Akka-cluster nodes need to join the cluster and the most reliable method is through a seed node defined with a static name and IP. Statefulsets offers exactly this deployment method. Seed nodes are paired with a Service. Services name can be resolved by the KubeDNS in order to find the IP of the pods defined in the service selector.

```
apiVersion: v1
kind: Service
metadata:
  name: backend-seed
```

```
spec:
  ports:
  - port: 2551
    protocol: TCP
    targetPort: 2551
  selector:
    run: backend-seed
  clusterIP: None
---
apiVersion: apps/v1beta2
kind: StatefulSet
metadata:
  labels:
    run: backend-seed
  name: backend-seed
spec:
  serviceName: backend-seed
  replicas: 2
  selector:
    matchLabels:
      run: backend-seed
  template:
    metadata:
      labels:
        run: backend-seed
    spec:
      nodeSelector:
        role: master
      containers:
      - name: backend-seed
        image: cris/cluster
        imagePullPolicy: Never
        resources:
          requests:
            memory: 512Mi
            cpu: 1
        env:
        - name: POD_NAME
          valueFrom:
            fieldRef:
              fieldPath: metadata.name
        - name: SEED_NODES
          value: backend-seed-0.backend-seed,
                backend-seed-1.backend-seed
        command: ['/bin/sh', '-c',
                'HOST_NAME=${POD_NAME}.backend-seed
                java -cp /app/app.jar Backend']
        ports:
```

```
        - containerPort: 2551
```

Code 9.10: Stetefulset file

Persistent volume When a pod needs storage, it can claim a part of a Persistent volume.

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0001
spec:
  accessModes:
    - ReadWriteOnce
  capacity:
    storage: 1Gi
  hostPath:
    path: /data/pv0001/
---
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: redis01-pvc
  labels:
    app: redis01-pvc
  annotations:
    volume.alpha.kubernetes.io/storage-class: default
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

Code 9.11: Persistent volume and persisten volume claim files

Inside the template specifications Persistent Volumes Claims can be linked to a container.

```
        volumes:
        - name: "redis-data"
          persistentVolumeClaim:
            claimName: redis01-pvc
        - name: "redis-conf"
          configMap:
            name: "redis-conf"
```

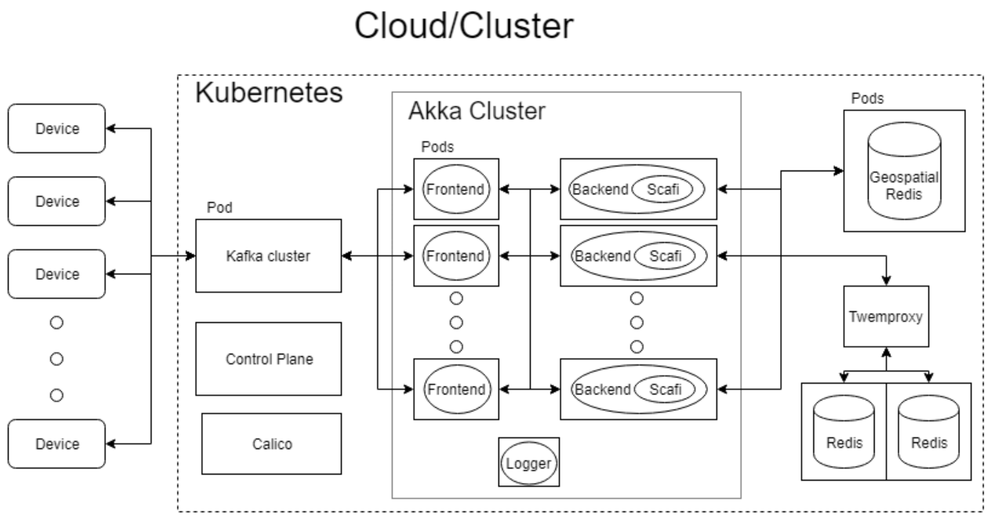

Figure 9.1: The final architecture of the system localized inside Kubernetes.

```
            items:
              - key: "redis.conf"
                path: "redis.conf"
```

Code 9.12: Persistent volume linking snippet

9.6 Final system architecture

Lastly, the system architecture, after the deployment phase, is shown in Figure 9.1.

Chapter 10

Testing

First, each service has been tested before the deployment to verify the correctness of the internal logic. However, the system must be tested inside production-like environment.

The testing has been performed on two local machines, connected through a router. Testing on these heterogeneous machines is, per se, a proof of the Kubernetes platform reliability.

Machine specifications:

- Dell Inspiron 7559:
  - Processor: Intel i7 6700HQ
  - Ram: 16GB DDR3
  - Storage: 500GB SSD
- HP HPE 120it (Kubeadm only):
  - Processor: Intel i7 860
  - Ram: 6GB DDR3
  - Storage: 1TB HDD 5400 RPM

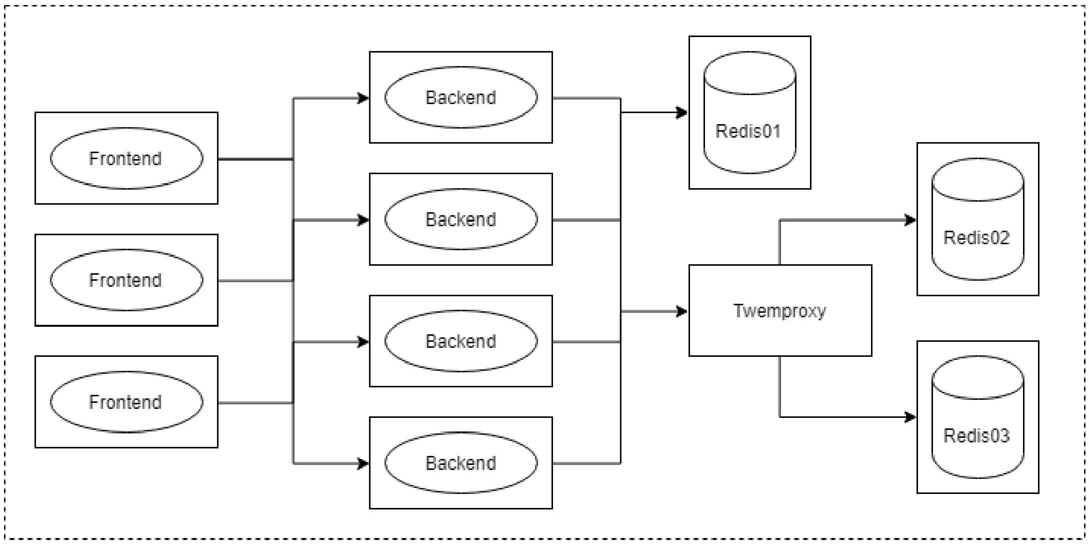

Figure 10.1: System deployment inside Mikikube

10.1 Minikube

The first set of deployment testing has been performed inside the Minikube environment, since it does not need a pod network to work.

This testing scenario has been produced inside the Minikube VM on the first machine as seen in Figure 10.1. This test purpose was to evaluate interaction between the components, inside a sandbox environment with an extremely low latency scenario in order to avoid errors. This method has been used for Continuous integration in the first design phase.

10.2 Kubeadm

In order to test the system on a multi-node cluster, Kubeadm is used to bootstrap the machines. In this scenario, the latency of the system is not approximately zero as with Minikube. Therefore, the system can be deployed as an approximated version of a real case scenario. Deploying the system with Kubeadm creates a production-like environment, which is essential for Continuous delivery.

In this deployment, as seen in Figure 10.2 only the database nodes are bound to the master since it has s faster storage unit, the other pods can be moved from node to node freely.

10.3 Performance

In this section, the performance of the system is analyzed in terms of latency and throughput. The throughput is intended as the number of messages processed by a piece of the system. While the latency is the time that passes from the start to the end of an operation.

10.3.1 Database

The Redis performance, as already said, in terms of latency is very efficient. With the tool provided by the Redis team, is possible to benchmark the minimum latency for connection inside the cluster, with the command shown in Code 10.1.

```
redis-cli --latency -h ‘host‘ -p ‘port‘
```

Code 10.1: Redis performance test

The minimum latency found for the connection is on average 1.54ms. If requests had to be processed singularly the database would be the bottleneck of the system. However, with the multiple get command the network latency is avoided.

As a result, the database scales linearly with the request amount, as shown in Figure 10.3. In any case, scaling the database reduces the overall time parallelizing the requests. With Twemproxy and two Redis instances the latency is more or less halved for a high number of requests. While with a low number of requests, the gain performance is not equally good.

10.3.2 Application

The Akka cluster system does not impact on performance per se. Since it is message driven reactive platform, the factors concerning the performance are network issues and the minimum latency. If the network fails, the overall latency will be greatly increased, as for every other stream processor. However, the asynchronous message passing and the scalability drastically reduce the impact of latency. So the throughput is not limited by the latency.

The bottleneck of the application is not the aggregate programming either, instead, the computation time is irrelevant even with a neighborhood of a thousand nodes (with a relatively simple aggregate program, as shown in Code 8.8). Moreover, it cannot be analyzed properly since the lowest unit of measure for time is the millisecond.

In fact, most of the computation time inside the Backend actor is dedicated to the JSON parsing. The process of parsing a string into a JSON object has a latency extremely low. However, in order to parse thousands of neighbor's exports, the operation has to be executed for each element, with cost O(MxN), where N is the original cost and M is the number of element to parse. The parsing is done with Scala through a parallel collection, so the resulting latency becomes more acceptable. In a scenario with a high number of neighbors, the vertical scalability might become more effective than the horizontal scalability. Actors with very few resources, like cpu time, spend most of their time in the algorithm computatio. This bottleneck is mainly caused by the un-serializability of the export map inside Scafi.

In a low-density scenario, with ten neighbors, the system has been able to process more than a hundred computation requests per second per Backend actor. Every component has been designed to scale linearly with the number of neighbors, starting from a fixed cost given by the minimum round-trip time, caused by the network latency.

10.3.3 Results

Lastly, the results, in terms of latency and frequency, scale linearly with the amount of neighbors, as seen in Figures 10.4 and 10.5.

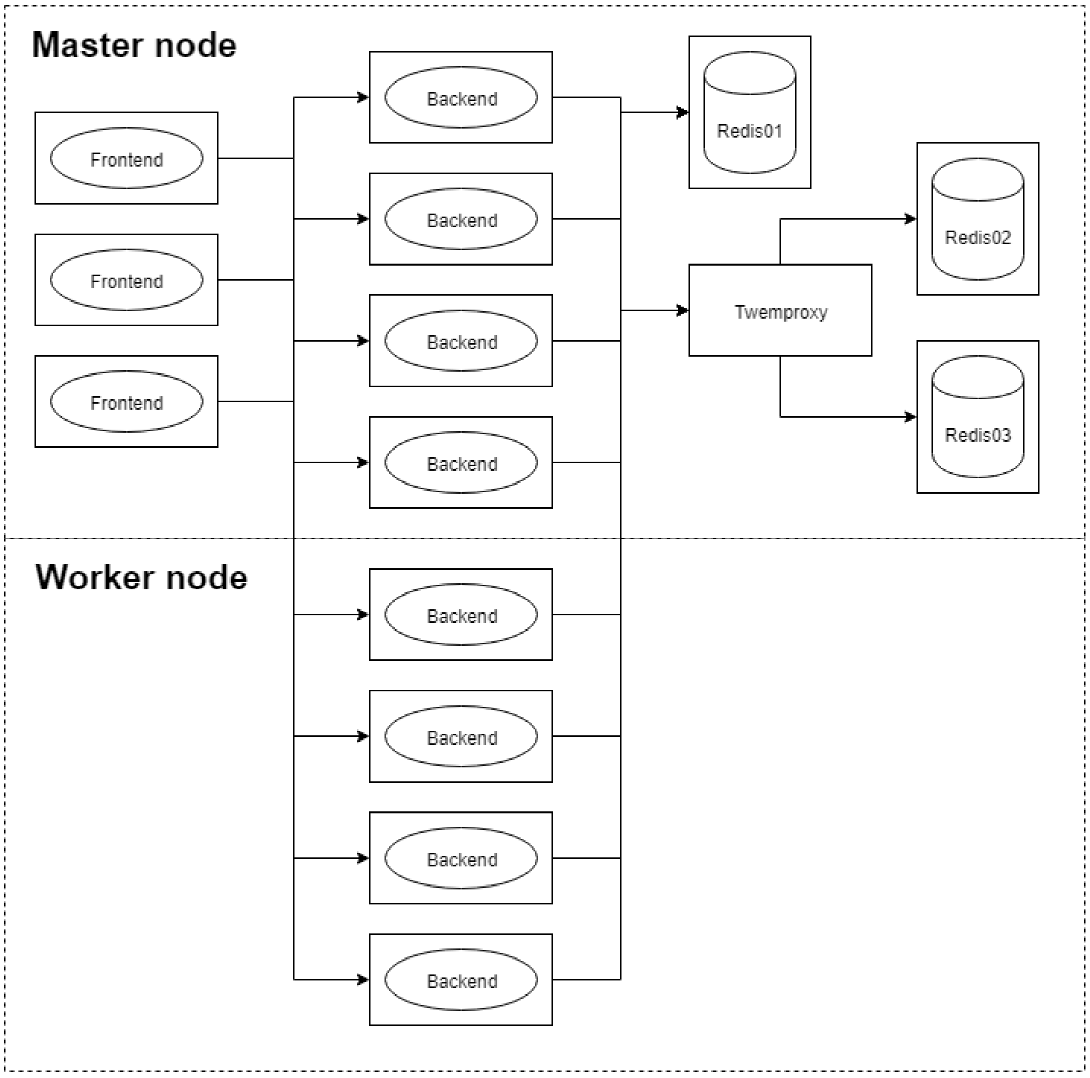

Figure 10.2: System deployment inside Kubeadm with two nodes

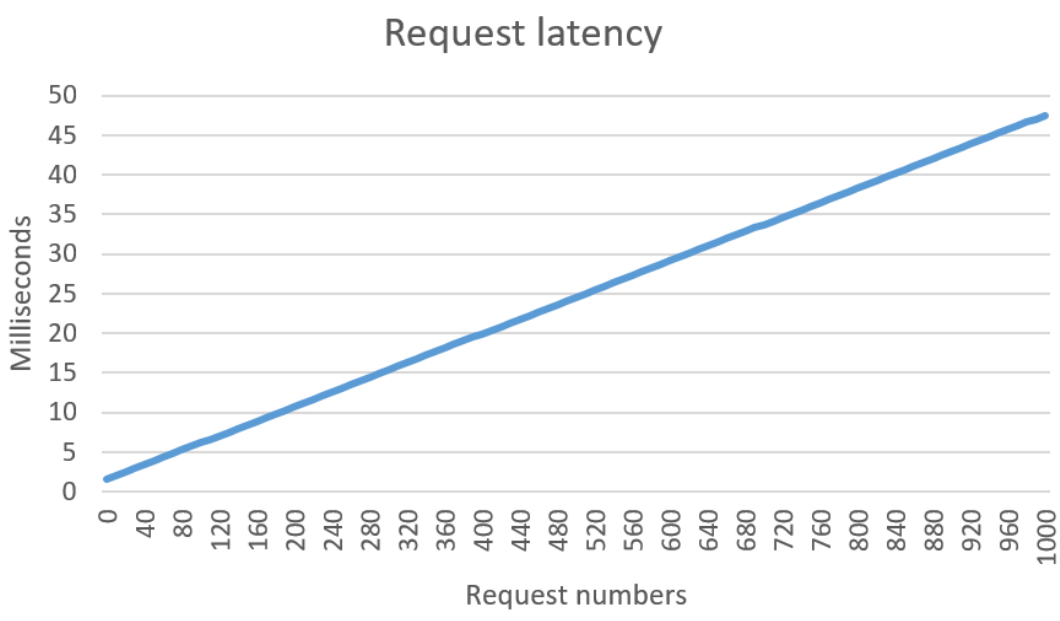

Figure 10.3: Single Redis node latency test

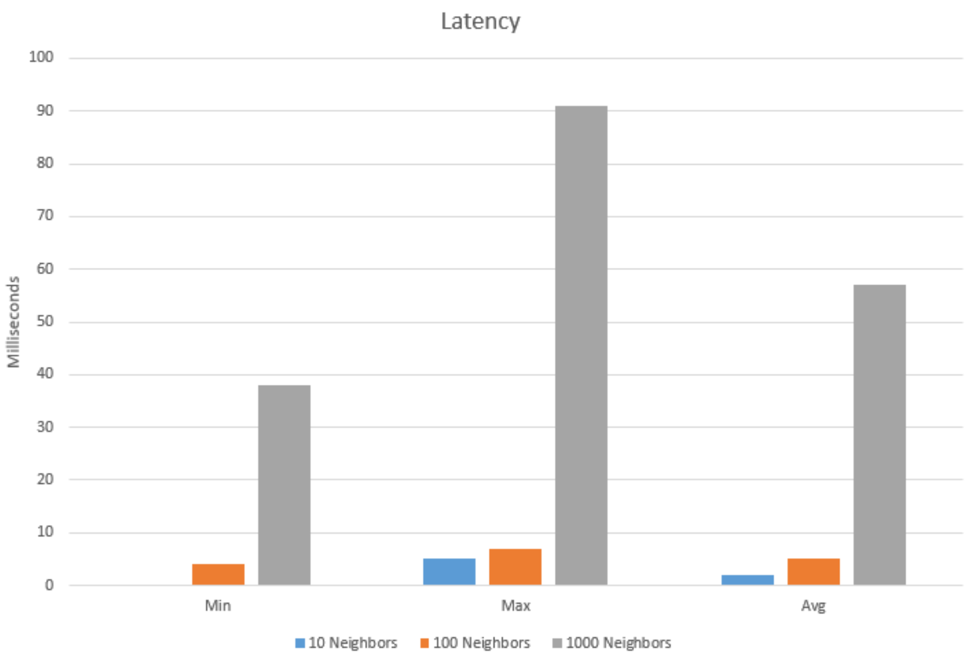

Figure 10.4: The result of the application latency with 10, 100, and 1000 neighbors

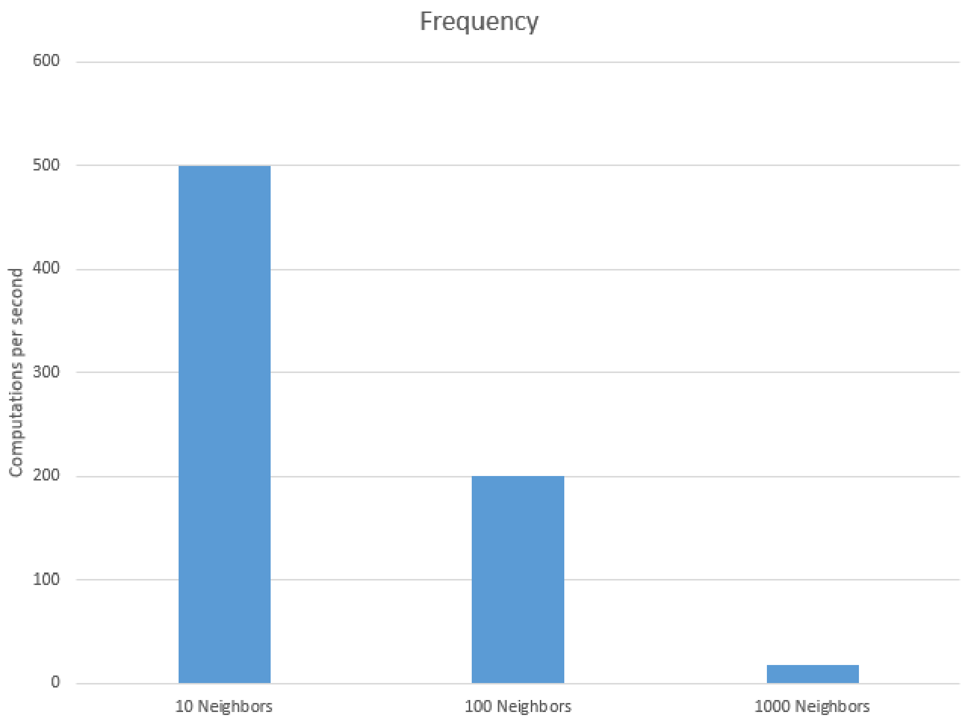

Figure 10.5: The result of the application frequency, derived by the latency, with 10, 100, and 1000 neighbors

Chapter 11

Evaluation

11.1 Functional requirements

The functional requirements of the application have been satisfied as follow:

- Data ingestion - the system has been designed with Kafka as an event source. As a MOM it inherently satisfies the requirement of data ingestion with its specifications;
- Aggregate programming - the computation of the sensor data in the cloud has been achieved with the deployment of Scafi, that provides the aggregate programming platform;
- Computation location - as explained, the computation location can be changed according to the device necessities, from local to cloud based;
- Storage - Redis has been used as main distributed storage and the system is predisposed for historicizing. The system is designed without a schema for data, as demonstrated using a key/value pair database like Redis, so historicizing is only a matter of deployment and configuration.

11.2 Non-functional requirements

Non-functional requirements for the system are: reusability, scalability, resiliency, and high-availability. Most of these are covered, at least in part, from the technologies. With Docker and Kubernetes, these requirements become easily satisfied with the tools provided for replication and self-healing.

11.2.1 Reusability

The system has been designed as a reactive platform and the coordination is performed through Scafi because of the requirements. However, the platform and the coordination model are orthogonal. If a new framework for aggregate programming is realized or another coordination model needs a reactive platform, it can be swapped with very limited knowledge of the system. As a matter of facts, Scafi has been introduced in the application at last and for most of the time the platform iterative testing has been made with ad-hoc algorithms. These algorithms derived the distance from a source point with a complete knowledge of all the nodes location and status.

11.2.2 Scalability

The horizontal scalability of the system is easily managed through Kubernetes automatic or manual scaling. The best method to identify the performance increase, caused by the addition of more Backend actor pods, is to analyze the workload.

Messages and data structures inside the actor are stored inside the RAM, so dividing the workload into more actors should result in a lower resources consumption. Moreover, the ratio of requests has been set to trigger heartbeat alert from the PHI accrual failure detector.

The system is first started in order to allow the stabilization. The actual computation started around point A of Figure 11.1. Increasing the Backend actors number, at point B, the load on the heap decreased and the throughput doubled, resulting in the failure alert stop.

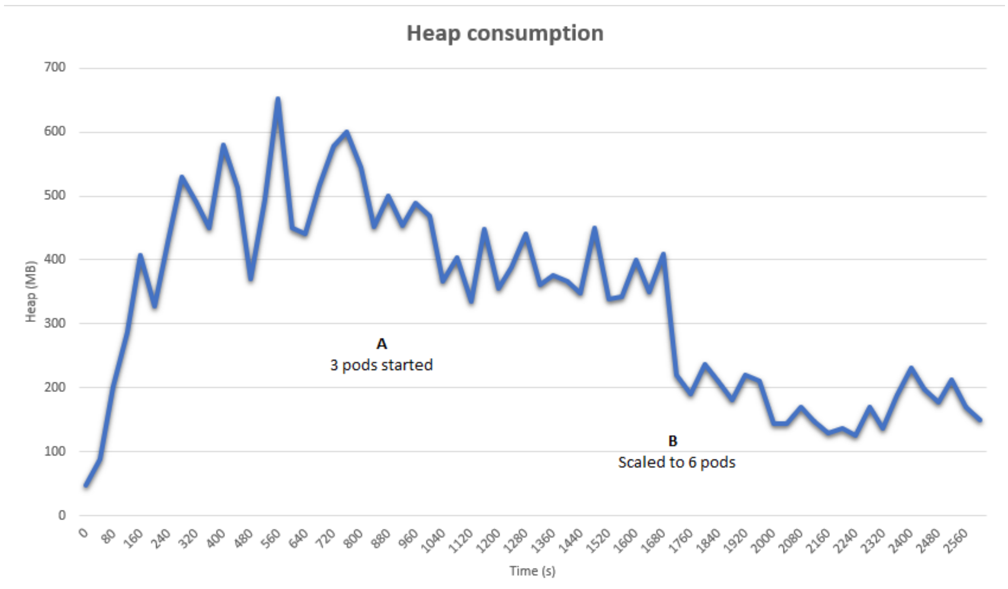

Figure 11.1: Heap consumption for a single Backend actor

11.2.3 Resiliency

The resiliency and the self-healing properties have been tested disconnecting the worker node for a brief amount of time, resulting in the workload to be entirely redirected to the remaining pods. As shown in Figure 11.2, that caused a stress spike in the heap consumption, because of the halved throughput. However, after the node has been reconnected the pods are redeployed and the system stabilizes again in seconds.

Obviously, the same mechanism works for every component, however in the application can be demonstrated with the data collected.

11.2.4 High-availability

The High-availability is a property found on each component of the system, but also in Kubernetes.

Kubernetes supports high-availability, for now, only in public cloud or in made-from-scratch environments. Kubeadm or Minikube, as said in Section 9.3, do not support high-availability. However, single components have their own method to

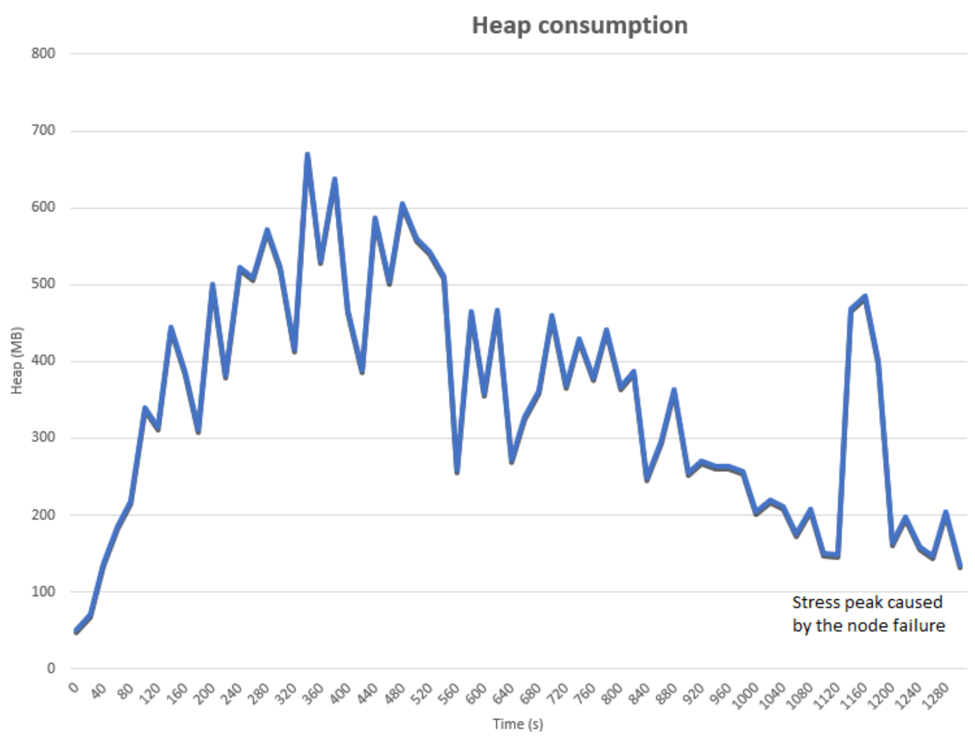

Figure 11.2: Heap consumption for a single Backend actor

achieve high-availability (HA).

Kafka manages HA through replication and leader election. If a Kafka broker goes down a replica is elected to be the new leader, until the broker is up again in a good state, then the leadership is restored.

Akka Cluster, in this context, achieves HA with a pool of stateless actors. If an actor fails it does not interfere with the others and when it is recreated its job is started again. This happens thanks to the failure detection system and the adaptive router, for the Backend actors. While for the Frontend actors distribute the workload with the Kafka consumer group.

Redis, similarly to Kafka, has Redis Sentinel that elects a new master from the replicas. However, this mechanism is supposed to work on a Redis Cluster, so it will really be a HA system when a stable Redis Cluster client is released for Scala. The Twemproxy solution for HA is auto-ejecting hosts in case of failure, so the data that should have been stored inside the failing node, is temporarily stored in other active nodes. Since the data distribution is known once the node becomes up again the normal functionality is restored.

Chapter 12

Conclusion

In this thesis, an approach for aggregate computing in a cloud environment has been presented. The aim of this project was another layer of abstraction between the bottom-up and top-down approaches, of respectively the classic IoT systems and aggregate programming systems, moving the computation in the cloud.

The results seem promising, even though there are many improvements ahead.

DSL The application is composed of many components and their infrastructure is defined by code, however some of the system topology and the application content and configuration must be done by hand. A DSL could allow a user to define the configuration of behaviors and the system topology as code.

Performance The performance of the system has two defined bottleneck that must be analyzed to improve the system throughput; both concern the exports storage. First, the export is mapped in JSON and serialized as a String value. However this bottleneck cannot be overcome until Scafi implements a method to serialize the export. The second bottleneck is how the data is stored in the database. Even though in Chapter 10 the database has been proven to scale linearly with the number of neighbors, multiple requests must be executed in order to get neighbors from the spatial database and then to get their exports. So, a re-engineering process may resolve this problem.

Kafka Obviously the system is currently a prototype and Kafka is not yet implemented. In order to improve the system, it must be deployed and tested with Kafka as a MOM and event source.

Bibliography

[1] Acetozi Jorge, *Pro Java Clustering and Scalability: Building Real-Time Apps with Spring, Cassandra, Redis, WebSocket and RabbitMQ*, 2017, 1st, Apress, Berkely, CA, USA

[2] Thomas Farneti, *Design and Deployment of an Execution Platform based on Microservices for Aggregate Computing in the Cloud*, Master thesis

[3] Roberto Casadei, *Aggregate Programming in Scala: a Core Library and Actor-Based Platform for Distributed Computational Fields*, Master thesis

[4] Docker swarm vs Kubernetes, https://platform9.com/blog/kubernetes-docker-swarm-compared/, Accessed: 03/03/2018

[5] Kafka vs RabbitMQ, https://content.pivotal.io/blog/understanding-when-to-use-rabbitmq-or-apache-kafka, Accessed: 03/03/2018

[6] Spatial query algorithms, https://blog.mapbox.com/a-dive-into-spatial-search-algorithms-ebd0c5e39d2a, Accessed: 03/03/2018

[7] Docker vs Kubernetes, https://platform9.com/blog/kubernetes-docker-swarm-compared/, Accessed: 03/03/2018

[8] Docker, https://www.docker.com/, Accessed: 03/03/2018

[9] Kubernetes, https://kubernetes.io/, Accessed: 03/03/2018

[10] Kafka, https://kafka.apache.org/, Accessed: 03/03/2018

[11] RabbitMQ, https://www.rabbitmq.com/, Accessed: 03/03/2018

[12] Akka, https://akka.io/, Accessed: 03/03/2018

[13] Spark, https://spark.apache.org/, Accessed: 03/03/2018

[14] Storm, http://storm.apache.org/, Accessed: 03/03/2018

[15] Storm vs Spark, http://xinhstechblog.blogspot.it/2014/06/storm-vs-spark-streaming-side-by-side.html, Accessed: 03/03/2018

[16] Ignite, https://ignite.apache.org/, Accessed: 03/03/2018

[17] Cassandra, http://cassandra.apache.org/, Accessed: 03/03/2018

[18] MongoDB, https://www.mongodb.com/, Accessed: 03/03/2018

[19] Twemproxy, https://github.com/twitter/twemproxy, Accessed: 03/03/2018

[20] Twemproxy and Redis, http://antirez.com/news/44, Accessed: 03/03/2018

[21] Redis, https://redis.io/, Accessed: 03/03/2018

[22] PostGIS, https://postgis.net/, Accessed: 03/03/2018

[23] PostgreSQL, https://www.postgresql.org/, Accessed: 03/03/2018

[24] Neo4j, https://neo4j.com/, Accessed: 03/03/2018

[25] MySql, https://www.mysql.com/, Accessed: 03/03/2018

[26] Jacob Beal, Danilo Pianini and Mirko Viroli *Aggregate Programming for the Internet of Things*, Computer 48,2015, pages 22-30

[27] Kief Morris, *Infrastructure As Code: Managing Servers in the Cloud*, 2016, 1st, O'Reilly Media, Inc.

[28] scala-redis, https://github.com/debasishg/scala-redis, Accessed: 03/03/2018

[29] JSON, https://www.w3schools.com/js/js_json_intro.asp, Accessed: 03/03/2018

[30] lift-json, https://github.com/lift/lift/tree/master/framework/lift-base/lift-json, Accessed: 03/03/2018

[31] Scala, http://scala-lang.org/, Accessed: 03/03/2018

[32] Akka Streams Kafka, https://doc.akka.io/docs/akka-stream-kafka/current/home.html, Accessed: 03/03/2018

[33] Wikipedia Kubernetes, https://en.wikipedia.org/wiki/Kubernetes, Accessed: 03/03/2018