TÉCNICO LISBOA

Dynamic Reconfiguration of the Data Aggregation Topology at the Edge

Tiago Miguel Calhanas Gonçalves

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Luís Eduardo Teixeira Rodrigues

Examination Committee

Chairperson: Prof. David Manuel Martins de Matos
Supervisor: Prof. Luís Eduardo Teixeira Rodrigues
Member of the Committee: Prof. José Manuel da Silva Cecílio

January 2021

Acknowledgments

I would like to thank my parents and Catarina for their friendship, encouragement and caring over all these years, for always being there for me through thick and thin and without whom this project would not be possible. I would also like to thank my siblings and grandparents for their understanding and support throughout all these years.

I would also like to acknowledge Prof. Luís Rodrigues and Nivia Quental for their insight, support and sharing of knowledge that has made this Thesis possible.

Last but not least, to all my friends and colleagues that helped me grow as a person and were always there for me during the good and bad times in my life. Thank you.

To each and every one of you – Thank you.

This work was partially supported by the FCT via project COSMOS (via the OE with ref. PTDC/EEI-COM/29271/2017 and via the “Programa Operacional Regional de Lisboa na sua componente FEDER” with ref. Lisboa-01-0145-FEDER-029271) and project UIDB/ 50021/ 2020.

Abstract

Edge computing is a paradigm where computation and storage services are offered by nodes that are placed close to devices that constitute the Internet of Things (IoT), as opposed to a pure cloud computing model where these services are provided by large central datacenters. The main advantages of edge computing are twofold: it allows to offer services to the IoT devices with low latency and it reduces the amount of data that needs to be sent to the central datacenters by means of data aggregation, providing significant bandwidth savings. These services are provided by edge nodes, often called fog nodes or cloudlets, that are placed in different geographical locations, close to the users. To ensure low latency the number of these servers will be necessarily high and need to be organized in a structured infrastructure that allows to take advantage of the localization of edge nodes. For the successful operation of edge computing it is crucial to have an infrastructure that is able to adapt to multiple environments and reconfigure itself to a more favorable topology. This thesis presents FlexRegMon, a system that allows to have dynamic reconfiguration of the data aggregation topology, by using a distributed component that manages the hierarchical topology of the system.

Keywords

Edge computing, Monitoring, Data Aggregation, Aggregation Topology, Dynamic Reconfiguration.

Resumo

Computação na periferia da rede é um paradigma onde os serviços de computação e de armazenamento são oferecidos por nós que estão perto de dispostivos que constituem a Internet das Coisas (IdC), ao contrário do modelo puro de computação na nuvem, onde os serviços são oferecidos por grandes centros de dados centralizados. As principais vantagens da computação na perfieria da rede são das seguintes: por um lado, permite oferecer serviços com baixa latência aos dispositivos de IdC e, por outro, permite reduzir a quantidade de dados que é necessário enviar directamente para os centros de dados, recorrendo a métodos de agregação de dados, que permitem reduzir a largura de banda usada no processo de recolha de dados. Estes serviços são fornecidos por nós da periferia, muitas vezes chamadas de nós neblina ou cúmulos, que são colocados em diferentes áreas geográficas, mais perto dos utilizadores. Para garantir a baixa latência dos utilizadores aos serviços será preciso um elevado número de servidores, que necessitam de estar dispostos numa estrutura organizada que permita tirar vantagem dos nós da periferia. De forma a garantir este bom funcionamento da computação na periferia é crucial que a infraestrutura seja capaz de se adaptar a múltiplos ambientes e seja capaz de se reconfigurar. Esta dissertação apresenta o sistema FlexRegMon, um sistema que permite a configuração dinâmica da topologia da rede dos agregadores de dados, através de uma componente distribuída, que gere a topologia hierárquica do sistema.

Palavras Chave

Computação na Periferia da Rede, Monitorização, Agregação de Dados, Topologia de Agregação, Reconfiguração Dinâmica.

Contents

1 Introduction **2**

1.1 Motivation . . . 3

1.2 Contributions . . . 4

1.3 Results . . . 4

1.4 Research History . . . 4

1.5 Organization of the Document . . . 4

2 Background **6**

2.1 Observation Function - Data Collection . . . 7

2.2 Data Processing Function . . . 8

2.2.1 Aggregation . . . 9

2.3 Data Exposition Function . . . 9

2.4 Fog/Edge Multi-layer Monitoring Structure . . . 10

2.5 Fog/Edge Monitoring Properties . . . 10

2.6 Monitoring Service Specification on Edge/Fog . . . 11

2.6.1 Functional Requirements . . . 11

2.6.2 Non-functional Requirements . . . 12

2.7 Existing Monitoring Solutions . . . 13

2.7.1 Astrolabe . . . 14

2.7.2 FMonE . . . 15

2.7.3 Ganglia . . . 16

2.7.4 MonALISA . . . 17

2.7.5 Monasca . . . 18

2.7.6 SDIMS . . . 19

2.7.7 Limits of existing systems in a Edge environment . . . 20

3 Design **22**

3.1 Goals . . . 23

3.2 Design . . . 24

3.3 Data Collection and Processing . . . 25
3.3.1 Orchestrator . . . 26
3.3.1.A Orchestrator-Replicas . . . 28
3.3.2 Aggregators . . . 28
3.3.3 Producers . . . 28
3.3.4 Node Bootstrap . . . 28
3.3.5 Service Discovery and Failures . . . 29
3.4 Data Model . . . 29
3.4.1 Hierarchical Structure . . . 29
3.4.2 Routing Information . . . 30
3.4.3 Data Sets . . . 31
3.5 Protocols . . . 31
3.5.1 Topology Re-Configurations . . . 32
3.5.1.A Steps to Add or Remove a Region . . . 33
3.5.1.B Adding or Removing Hierarchical Levels . . . 35
3.5.2 Aggregator Removal . . . 39
3.5.3 Orchestrator-Replica Failure . . . 41

4 Implementation **44**

4.1 Zookeeper . . . 45
4.1.1 Zookeeper Notification System . . . 46
4.2 FLEXREGMON Internal Zookeeper Structure . . . 46
4.3 FLEXREGMON Discovery Service . . . 47
4.4 Notification System . . . 48
4.5 Orchestrator . . . 49
4.5.1 Orchestrator-Replicas . . . 49
4.6 Aggregators . . . 50
4.7 Deployment . . . 50
4.7.1 Docker . . . 51
4.7.2 Kollaps . . . 52

5 Evaluation **55**

5.1 Evaluation Goals . . . 56
5.2 Experimental Setup . . . 56
5.3 FLEXREGMON vs. Zookeeper - Propagating Modifications on the Edge . . . 57
5.3.1 Adding and Removing Aggregators . . . 59
5.4 Notification System . . . 61

5.5 FLEXREGMON Overhead 64
5.6 Benefits of the Flexible Topology 64
5.7 Discussion 67

6 Conclusions and Future Work **68**

List of Figures

3.1 FlexRegMon general structure 24
3.2 Hierarchical Tree after new level addition 27
3.3 Partial Topology with Portugal and its sub-regions (left) and aggregators available for each region (right) 31
3.4 Sequence diagram for adding a region 34
3.5 Sequence diagram for removing a region 35
3.6 Sequence Diagram for adding a new hierarchical level 37
3.7 Sequence Diagram for propagating hierarchical level removal 39
3.8 Sequence Diagram for aggregator removal in a controlled scenario (System Control kills aggregator) 40
3.9 Sequence Diagram for aggregator removal in a uncontrolled scenario (aggregator crash) 41
3.10 Example of procedure for Replica crash or network partition between *Orchestrator-Replica* and Orchestrator 42
3.11 Network partition between Replica and aggregators 42

4.1 Example of FLEXREGMON Zookeeper Server Hierarchical Structure 46
4.2 Zookeeper Structure example to store Routing Table information 48
4.3 Docker file declaration 52

5.1 FLEXREGMON vs Zookeeper: propagating topology changes on the Edge 58
5.2 Adding and removing aggregators 60
5.3 Notification System: performance while varying number of nodes 61
5.4 Notification System: performance while varying the latency 62
5.5 FLEXREGMON vs Zookeeper in centralized environment 63
5.6 Benefits of flexible topology: Dynamic vs Static 65

List of Tables

3.1 Data Sets: Lisboa and Setúbal aggregator 26
3.2 Example of a data-sets table for the Portugal Region 26
3.3 Example of a partial Hierarchical table for the Portugal Region 30
3.4 Example of a Routing table for the Orchestrator-Replica responsible for Portugal 31
3.5 Example of data set stored in an aggregator 31

List of Algorithms

1 Central Orchestrator Code - Add region 33
2 Orchestrator-Replicas Code - Add region 33
3 Central Orchestrator Code - Remove region 34
4 Orchestrator-Replicas Code - Remove region 35
5 Central Orchestrator Code - Add Hierarchical Level 36
6 Orchestrator-Replica Code - Add Hierarchical Level 36
7 Aggregator Code - Add New Hierarchical Level 37
8 Central Orchestrator Code - Remove Hierarchical Level 38
9 Orchestrator-Replica Code - Remove Hierarchical Level 38
10 Aggregator Code - Remove Hierarchical Level 39

Listings

4.1 Example of FLEXREGMON Kollaps topology file. 53

1

Introduction

Contents

1.1 Motivation 3
1.2 Contributions 4
1.3 Results 4
1.4 Research History 4
1.5 Organization of the Document 4

1.1 Motivation

The number of devices that are connected to the Internet is very large and keeps growing at fast pace. The nature of these devices is very heterogeneous, from powerful laptops and smartphones to small sensors, a plethora of devices have the ability to provide services to end users and to collect and produce data: media servers, smart TVs, consumer appliances, smart watches, smart home sensors and actuators, etc. This reality is known as the Internet of Things (IoT). Edge computing is a paradigm where computation and storage services are offered by nodes that are placed close to devices that constitute the IoT, as opposed to a pure cloud computing model where these services are provided by large central datacenters. The main advantages of edge computing are twofold: it allows to offer services to the IoT devices with low latency and it reduces the amount of data that needs to be sent to the central datacenters, providing significant bandwidth savings.

Edge computing typically relies on a multi-layer architecture [1] [2] [3] with the following components: (i) A centralized cloud computing layer, which includes cloud datacenters. It can be used for long-term storage and big-data analysis and not for time-sensitive data processing; (ii) A fog/edge layer, that has the ability to pre-process raw data before it is shipped to the cloud, for example, aggregating and filtering it. It allows processing data closer to the location of capture which leads to better latency and response times. This layer can have multiple levels, where they can be either closer to cloud or to the edge where end-users are; (iii) the edge layer, composed by sensors/devices that generate the data and execute applications.

The architecture above allows also improves application performance as data is processed closer to the end-user which allows to reduce latency. It provides new approaches to load balancing by introducing new functionalities of service migration such as moving a running service from the cloud layer to the edge computing layer. It also provides awareness of location, network and context information. The edge layer also makes easier to track end-user information and adapt the environment to their needs and preferences. Finally, it also minimizes energy consumption for the end-user devices, as it allows battery-constrained devices to offload heavy tasks to edge nodes which are not as far away as the centralized nodes.

In this work we are mainly concerned with the operation of nodes in the edge/fog layers. These nodes can assume different structures, known as micro-datacenters [4], cloudlets [5], or fog computing [6]. To ensure that they can provide services with low latency to devices in the edge layer, these nodes need to be placed in locations that are physically close to the end-devices. For the successful operation of edge computing it is crucial to have an infrastructure that is able to monitor the status and usage patterns of edge nodes, not only to perform maintenance and repair, but also to re-configure the applications based on the observed usage patterns.

1.2 Contributions

This thesis, implements and evaluates strategies to enforce easy topology manipulation of a running system, in order to allow optimization of load and latency throughout the whole system. The resulting contributions are the following:

- The design of a new system that permits easy and fast topology manipulation
- A new notification system for Zookeeper, which works better in networks with high latency, that allows to use daisy-chained Zookeeper Clusters to maintain information accessible in a distributed environment with low latency response times.

1.3 Results

This thesis produced the following results:

- An implementation of a monitoring system that eases the modification of data aggregation topology in an Edge environment.
- Distribution of management of hierarchical regions that allows reduce latency overhead of operations to fog and edge nodes.

1.4 Research History

This work was developed in the context of the Cosmos research project, that aims at finding techniques to offer causal consistent storage for edge computing scenarios. Techniques to monitor the operation of the edge system are a key component of the Cosmos architecture.

This work was partially supported by the FCT via project COSMOS (via the OE with ref. PTDC/EEI-COM/29271/2017 and via the “Programa Operacional Regional de Lisboa na sua componente FEDER” with ref. Lisboa-01-0145-FEDER-029271) and project UIDB/ 50021/ 2020.

1.5 Organization of the Document

This thesis is organized as follows. Chapter 2 presents the characteristics necessary for a monitoring systems, its requirements, structure and a number of existing systems. Chapter 3 and Chapter 4 describe the design of FlexRegMon and the implementation of the its prototype as well as other components that were required for deployment. Chapter 5 reveals the results of the evaluation and makes some

statements about the differences among the systems. Chapter 6 concludes this thesis by resuming its advancements and possible directions for some future work.

2

Background

Contents

2.1 Observation Function - Data Collection 7
2.2 Data Processing Function 8
2.3 Data Exposition Function 9
2.4 Fog/Edge Multi-layer Monitoring Structure 10
2.5 Fog/Edge Monitoring Properties 10
2.6 Monitoring Service Specification on Edge/Fog 11
2.7 Existing Monitoring Solutions 13

Cloud services can be requested on-demand and are provided by elastic and scalable resources. Some of the main characteristics offered by cloud services are [7]: availability, concurrency, dynamic load balancing, independence of running applications, security, and intensiveness. These characteristics are attained by using datacenters with hundreds of high performing servers.

In order to be able to efficiently manage this type of systems with increasing complexity, there is a necessity of an accurate and fine-grained monitoring solution capable of capturing different types of information regarding the operation of the system, its components, and subsystems.

The ability to monitor the system is essential for both Cloud providers and consumers/users. For providers it helps to have a monitoring system that can gather information to ease control and managements of the infrastructures. While for consumers and users a monitoring service can provide key information about the applications that they are running on the infrastructures.

In order to operate in an infrastructure of this scale and complexity, a monitoring system needs to fulfill different functions [8]: *observation* of monitored resources, *data processing* and *data exposition*. These functions can also be further divided into sub-operations such as aggregation, transformation of measurements into events, and event processing.

2.1 Observation Function - Data Collection

The *observation* function is responsible for tracking remote resources and gathering data for processing. This function can be implemented in various ways. It can be entirely implemented by the monitoring system, or be distributed throughout all the remote resources. There are two strategies approaches to collect data from remote resources [9]:

- **Pull** or polling solutions are usually used in centralized solutions. This type of solution requires a pre-registration of the monitored resources on the system in order to be able to be localized. After registration, the system can request/query the monitored resource for information or actively perform checks on the system, like testing if the node is alive by making ping tests.

  The system needs to be aware of the location of all the nodes and contacts each one individually, which makes it hard to scale.

  Furthermore, in an architecture with a lot of volatility, like the Edge, where we have high number of joins and exits of nodes, we may have a unfeasible number of registration processes, or the system may try to query nodes that already left the system but were not acknowledge as such as there is no way of detecting unless the remote resource de-registers itself. This is hard to guarantee in highly elastic and mobile environments where nodes do not know when they will leave the system.

- **Push** model favors decentralized and highly elastic environments. It does not require resources to

register themselves on the system. Monitored resources are the ones responsible for transmitting the data to the monitoring system according to some time interval or event condition. Furthermore, due to the characteristics of this communication model, an unidirectional channel is created where communication only flows in one way, from monitored resource to the monitoring system. This makes it easier to address security concerns and being capable to communicate with every resource, even if when these are behind middleware boxes and firewalls that, due to security reasons, might block incoming traffic.

The characteristics of the push model make it the more adequate for an Edge environment, due to the volatility of the nodes and the scale of the number of nodes. Using a pull-based solution would be troublesome, as we would have to deal with a high number of registration processes, due to edge nodes being mobile which results in a high chance of leaving the system. Also, while using a pull-based communication, the monitoring tool is not able to immediately recognize the loss of resources, and is going to keep trying to pull data from it, wasting time and resources.

2.2 Data Processing Function

In order to be able to use the data generated by the monitoring resources, there is a need to process and interpret it. This is necessary to evaluate, and provide necessary insight about the system, and its components. This analysis can be used to perform optimizations and adaptations on the system to increase performance.

Data processing operations have different requirements for different objectives [10]. In environments where we answer requests as fast as possible, it is beneficial to use techniques that help us obtaining the relevant data as quickly as possible. For instance, if a request does not require a global view of the system we can consider data from only a subset of all the information on the system. This allows getting the necessary data quicker (in perfect conditions it would only use local data), and in consequence process it quicker. In other scenarios, we may need to have a global view of the system. To achieve that, it may be necessary to process the data from the whole system. The latter type of operations needs to support longer response times, as the volume of data that needs to be processed is much higher and localization can be more sparse.

The data processing/analysis can also be used to create historical data to be later interpreted. For instance, data mining can find patterns and match certain events with problems on the infrastructure, which may reveal that when a specific zone is having issues the whole system performance will go down. This knowledge can then be used to adjust the system.

2.2.1 Aggregation

Aggregation is a type of computation function that can be applied to data sets. It takes multiple sources of data of the same type and resumes their values into a single output. The combined output gives an insight regarding the inputs while avoiding all the individual contributions to be re-processed whenever their values are needed. Aggregation of data provides context around different data points. Typically it occurs at centralized location and provides powerful insight into the behavior of an application or zone of the system.

- **System Aggregation** of data reveals the overall health and state of the system. This type of aggregation is designed to give various views and insight into the system state. In addition, we calculate different aggregates with distinct sets of metrics, in order to predict different scenarios, or to play "what-if" situations, where we can explore potential side effects of changes that might modify or introduce faulty behaviors in the system.

- **Local Aggregation** is an optimization that aims at keeping the data near the nodes that generated it. Many metrics and events can be computed "on the go" locally, instead of requiring data to be sent to another location for analysis. Unfortunately, local aggregation is not always possible, either because the edge nodes do not have enough processing power, or because it would add an unacceptable latency (due to the need to gather the data from other nodes) to the query response.

2.3 Data Exposition Function

This function is responsible to export the output of the monitoring system to the users, allowing them to visualize the information captured and processed. There are two primary types of outputs:

- **Notifications:** Consist on communicating to a user or to another system that an event has occurred. It can also be used to integrate two systems that depend on each other. For example, consider a situation where *system A* needs *system B* to be in a certain state and the monitoring system could be programmed to notify *system A* when *system B* is in that specific state.

- **Visualization:** Can be shown in the form of tables, charts, or graphs, that allow users to analyse and reason about the state and performance of the system. Users should be able to check for different zones of the system and, in case of historical data, change the granularity and period of the data being shown.

2.4 Fog/Edge Multi-layer Monitoring Structure

Edge computing follows a different structure from cloud computing in the sense that it is divided by layers [3]. At its core, has three layers: the *cloud computing layer*, which is similar to the classic cloud computing paradigm where we have big datacenters with a lot of high performance servers; the *fog layer*, that consists on having high performance nodes or even small datacenters called *cloudlets*, deployed in different sites closer to the users; and the *edge layer*, which is populated by the end-users and end-devices, these devices have low performance, power constraints, and are responsible for generating most of the data of these type of systems.

Due to the differences between this structure and the one from classic cloud computing, a monitoring service for the edge needs to be structured differently from a monitoring system for the cloud. The most common structure, proposed in the literature [1, 3, 8], follows a multi-layer structure similar to the edge infrastructure:

- **Centralized cloud computing layer:** Includes cloud datacenters that can be used as long-term storage. Datacenters can execute application-level *data processing* and *exposition* functions that do not have short latency requirements. Big data analysis of the whole infrastructure can be performed on this layer.
- **Fog layer:** In this layer, smaller cloud resources are deployed in order to be closer to end-users and end-devices. It allows for a more localized *processing* and *exposition* of the data and also allows edge nodes to offload some of its computation to it. This layer hosts the entry points to the monitoring system, for the edge nodes. It also hosts data collection services that are able to act on the captured data, for example, by aggregating, filtering or encrypting local data.
- **Edge computing layer:** This layer contains the devices, users, and applications that will implement part of the *observation* function of the monitoring system. Most of the system data origins in this layer.

2.5 Fog/Edge Monitoring Properties

The multi-layer structure makes fog/edge environments very different from the traditional cloud computing environments. Its specific characteristics fundamentally change the strategies that can be used when monitoring the system [1, 8]:

- **Large and massively distributed:** Fog/edge based systems are deployed across a different number of sites that can be widely spread. The distance separating these sites can reach hundreds of kilometers. They can be connected through different types of links, each one with its reliability

and speed. Both factors, distance and nature of the link, affect latency and bandwidth between resources, a fact that should be taken into account when designing a monitoring system.

- **Heterogeneity:** The infrastructure is composed by different types of devices, such as servers with high computing power, storage servers, routers, gateways, middleware appliances, and users devices. All these resources have different characteristics in terms of capacity, reliability, and usage. On top of it, virtualization which is widely used in the cloud environment also adds another level of heterogeneity, due to the possibility of the resources being either virtual or physical.
- **Highly volatile:** Nodes at the edge level may join and leave the system at any time, given that many edge devices are mobile.

With these new properties in mind, we need to provide a set of improvements over the classic cloud paradigm for monitoring systems.

2.6 Monitoring Service Specification on Edge/Fog

The new infrastructure and specific characteristics imposes a new a set of requirements, that need to be taken into account in a monitoring system for a fog/edge environment [1, 2, 8]:

2.6.1 Functional Requirements

- **Reduce the amount of network traffic:** Nodes at the fog computing layer should be able to filter unnecessary data and aggregate information that needs to be streamed to other nodes. By filtering and aggregating the data before sending, it is possible to decrease the amount of data that is being transmitted. This helps dealing with the big volume of information generated by the large and massively distributed number of edge nodes.
- **Tweak monitoring intervals:** The system should be tunable and trigger data collection or processing based on custom intervals or event conditions. By customizing time intervals, we can prevent the network from being congested with too many messages, something that could happen if all nodes would transmit at the same time. Using notifications to trigger monitoring, we can provide immediate response to abnormal circumstances, avoid having to wait for the next time interval and preventing loss of control of the system.
- **Long-term storage:** The monitoring solution should be able store the data in an optimized way that allows for future retrieval of the data. In this way, monitoring data can be used to inform future adaptation strategies.

- **Service migration:** The system should be able to adapt to reallocation of services, something that is common with distributed systems.
- **Independent from underlying cloud infrastructure provider:** The system should be able of inter-operable monitoring and to share information among heterogeneous frameworks.
- **Quickly react to dynamic resource changes:** The monitoring solution should rapidly detect and collect information about the changing environment. Edge computing requires an agile monitoring system, especially at the Edge layer, due to the highly elastic environment where end-devices may frequently join and leave the system.
- **Operating system and hardware independence:** An Edge monitoring solution has to deal with heterogeneous resources and in order to achieve it, needs to interact and capture data independently of the resource operating system, if it is virtualized or not, and of which hardware it is running on.
- **Improve the application performance:** Users and applications at the edge level might need a quick response to a request. The system should be able to provide those fast responses. Therefore, it should not rely exclusively on the cloud computing layer or other centralized components to process localized data. Instead, it should support local data processing on fog nodes, which are closer to the edge. Localized processing reduces the latency and reduces response time (as long as it only uses local data), which results in better application performance.
- **Location and network context awareness:** Edge devices and fog nodes are distributed in a wide geographic area. This is used to track end-users information, such as their location, mobility, network condition, behavior, and environment in order to efficiently provide customized services. This allows to provide context for the captured data and extra attributes to categorize information which will allow to make localized adjustments to the system [11] or end-users' preferences.
- **Minimize energy consumption:** Some edge devices have limited resources and should be able to offload [12,13] tasks to fog nodes that are preferably close (otherwise the power used to transmit the data could be higher than processing the data itself). When applied properly, this technique helps in reducing the energy consumption of edge devices.

2.6.2 Non-functional Requirements

- **Scalability:** The system needs to scale and be able to monitor a large number of resources. It should be able to handle a sudden growth of monitored resources as well as a sudden high load of requests, while maintaining performance across the whole system.

- **Non-intrusiveness:** Edge computing makes use of small devices that need to be efficient due to their energy constraints. This creates an environment where special attention need to be provided to resource usage, by adopting a strategies that use minimal processing, minimal memory usage, and reduced communication. The *observation* function of the monitoring system should also gather metrics from edge devices using a non-intrusive and lightweight implementation.

- **Locality:** The monitoring service should ensure adequate response delay, regardless of the location of the monitored resource. It should allow to deploy the monitoring service on multiple locations and near its users, in order to be able to guarantee low response times.

- **Modularity:** The heterogeneous resources of the fog/edge infrastructure range from high performance servers to low power devices. To offer monitor services more choices its deployment should be made possible in any type of these resources, regardless of their capacities, *OS*, or other hardware characteristics.

- **Robustness:**

  - **Resilience to server additions/removals:** The monitoring system cannot prevent the failure of the servers hosting it. It should be able to adapt if any of its resources is removed or if there is a need for a system migration of any of its modules, an event that is common in a virtualized environment like cloud and fog environments.

  - **Resilience to network changes/failures:** Due to the distributed nature of the fog/edge based architecture, the network is highly vulnerable to network failures. In particular, at the edge level, we have many small mobile devices that can enter and exit the system with ease, and the system should be able to adapt to these changes. Furthermore, monitoring remote resources relies heavily on the network to transmit data. The system should also be able to cope with networks failures, using mechanisms to guarantee deliver of data and alternative routes to reach the system.

2.7 Existing Monitoring Solutions

We searched and read about existing monitoring solution to get a grasp of the techniques used to gather data, aggregate, and to assert if those systems were capable of adapting to the different conditions and the changes that we might find in the edge.

2.7.1 Astrolabe

Astrolabe [14] is a hierarchical monitoring system. It organizes the monitoring nodes into a hierarchy of domains, which are called zones. A *zone* is recursively defined to be either a set of hosts or a set of non-overlapping zones. Astrolabe continuously computes summaries of system data using *on-the-fly* aggregation.

The hierarchical distribution of the zones can be viewed as a tree of nodes, where leaves represent the physical hosts and middle/top nodes are virtual nodes hosted on physical hosts. Each zone has a *local zone identifier*, a string name unique within their parent zone. A zone is identified by its *zone name*, which represents the *name path* of zone identifiers from the root of the tree to the node itself. Representatives from the set of hosts within the zone are elected to take responsibility for running the gossip protocol that maintains the internal zones. If they fail, the zone will automatically elect another node to take the place of the failing node.

Astrolabe propagates information using an epidemic peer-to-peer protocol known as *gossip* [15]. Each node in the system runs an Astrolabe agent and every agent runs the gossip protocol with other agents. It will periodically choose another node at random and exchange information with it. If both nodes are within the same zone, the state exchanged is related to information of that same zone. The use of gossip allows the state of the Astrolabe nodes to converge, as data ages and nodes communicate with each other.

Each zone stores information in an *attribute list*, a form of Management Information Base or MIB, which borrows its terminology from SNMP [16]. Astrolabe attributes, unlike SNMP, are not directly writable, but generated by *aggregation functions*. Each zone has a set of functions that calculates the attributes of that zone. An aggregation function for a zone is defined as a SQL program. It takes a list of the zone's children attributes and produces a summary. The only attributes that are writable are in the leaf zones. Higher level attributes are created from these writable attributes.

Each agent keeps a local copy of only a subset of all the attributes. The subset includes all the zones on the path to the root node, as well as sibling zones of each of those. In particular, each zone has a local copy of the root MIB, and the MIBs of each child of the root.

There are no centralized servers associated with internal zones and all the data is replicated on all agents within the zones it belongs to. Due to the structure of the hierarchy tree and the fact that every node has a subset of the tree's information, it is possible to answer queries and requests for certain zones using only local information.

Astrolabe is also capable of dealing with membership management problems such as failure detection and integration of new nodes. Each MIB has a *representative* attribute that contains the name of the agent that generated that MIB, and an *issued* attribute that contains the time at which the agent last updated that MIB. Each agent keeps track, for each zone and for each representative agent, the last MIBs

from each agents. When an agent has not seen an update for a zone from a particular representative agent for that zone for some time T$fail$, it removes its corresponding MIB. When the last MIB of a zone is removed, the zone itself is removed from the agent's list of zones.

In order to recover from crashes or add new machines, Astrolabe treats node integration as merging two Astrolabe trees. It relies on IP multicast to set up the first contact between the trees. After the initial setup, each tree multicasts a gossip message at a fixed rate leading to an eventually merged state of the two trees and starts using the normal *gossip* protocol.

2.7.2 FMonE

FMonE [17] is a fog monitoring solution aimed at addressing fog-based architecture requirements with its focus on heterogeneity. It relies on a container orchestration system to build monitoring workflows that adapt to the different environments that can be found on a fog architecture.

FMonE main module is a centralized framework that coordinates the monitoring process across the whole fog infrastructure. It is designed to work with container technologies. It uses an orchestrator to coordinate and maintain monitoring agents (responsible for the *observation* and *processing* functions) and the back-ends (used to store metrics). The orchestrator is replicated on multiple instances. In case of failure these replicas can replace the instance and keep the system running.

FMonE organizes groups of nodes into regions. A region is composed internally by FMonE agents which are responsible for gathering the metrics, process them and send to a back-end. FMonE uses a concept called *Pipeline* to match agents and back-ends to its regions. The *Pipeline* also defines the workflow of the agents and how they should behave for each function by defining three set of rules *InPlugin*, *MidPlugin* and *OutPlugin*:

- *Inplugin*: defines how frequently the agent and which data is extracted from a component of the system.
- *MidPlugin*: defines custom functions that are able to filter and aggregate the set of metrics extracted by the agents (applied before the metrics are published).
- *OutPlugin*: defines the time condition to push data and the location to where it will be sent. An agent can dump its data in a back-end to be stored or in a message-queue to be used by another agent.

The customization of these sets of rules allow for extra flexibility on the agent's behavior. It is possible to change the configuration according to each region conditions which allows the system to be used with different types of devices. The collection of the data by the agents is based on a push approach. It starts by extracting the data from the device to the agent memory using *InPlugin* rule set, then it applies all the

filtering/aggregation function declared by *MidPlugin*, and finally pushes the processed data to locations given by *OutPlugin*.

For new nodes to join the system the new nodes simply need to match the *pipeline* rules of the region that it wants to join. The orchestrator will initiate the agent and the node will join and it will start collecting pushing the metrics.

The use of regions along with pipelines to define the workflow of the agents makes the system take a hierarchical architectural approach. While it does not create a strict structure based on trees like Astrolabe [14] and SDIMS [18], it allows the system's regions to take a hierarchical behavior, where they can use aggregated values from other regions instead of taking all the individual values from all devices.

2.7.3 Ganglia

Ganglia [19] is a monitoring system for high performance systems such as Clusters and Grids. It is also based on a hierarchical architecture, with machines divided into federations of clusters.

Each cluster has a representative node. Ganglia creates a tree of point-to-point connections with cluster's representatives nodes in order to aggregate their state and create hierarchical relations between the clusters. This structure needs to be manually configured by the administrator of the system.

In each node in the cluster representatives tree exists a pull-based service called *gmetad*. It periodically polls the child data sources and aggregates them into a single value. The data sources can be other representative nodes (*gmetad* instances) that represent a single or multiple clusters, or at the lower levels of the tree could be the physical machines that compose the clusters.

Every node in a cluster collects and maintains monitoring information for all the other cluster nodes by listening to a well-known multicast address. To collect metrics within each federation, Ganglia uses another service called *gmond* that runs on every node. It monitors the node's local resources and sends multicast packets containing the monitoring data to the cluster-wide multicast address.

The multicast-based listen/announce protocol allows for a swift re-election of cluster's representatives in case of failure of the current one, as all nodes know all the values of the whole cluster. It also makes joining the system easier, as nodes only need to start listening/announcing in order to join the federation.

In order to not flood the network with messages, the broadcast of metrics inside a cluster only occurs when there are significant updates to those values. Ganglia also uses time thresholds to specify an upper bound on the interval of when metrics are sent. Every time a node reaches the threshold, it will multicast its data to refresh the time value, even if there are no new values.

To deal with faulty nodes the system uses heartbeat messages with time thresholds. Each heartbeat contains a timestamp representing the startup time of its *gmond* instance. These values are stored on every node on the cluster like metric data collection. Anytime a *gmond* instance has an altered timestamp (compared with the local value stored in each node) it is immediately recognized by its peers

as having been restarted. A *gmond* which has not responded over some number of time thresholds is assumed to be down.

Although the system has a hierarchical architecture like Astrolabe [14], it cannot provide local scope queries. This happens because federations are not associated with a name space, which makes it impossible to target them. Information is simply collected and sent up the tree to upper *gmetad* nodes. The manual configuration of the tree structure can be a problem at a bigger scale, as it is not possible to manual configure thousands of nodes that we might find in bigger environments.

2.7.4 MonALISA

MonALISA [20] innovates by not focusing on monitoring a single site and instead focusing on monitoring at a global scale. MonALISA uses a service-oriented architecture and is designed to serve large physics collaborations that are spread over multiple data grids composed of hundreds of sites on different locations, with thousands of computing and storage elements.

In order to scale and work robustly while managing global, and resource-constrained Grid systems, MonALISA follows a peer-to-peer approach. Uses a set of *Station Servers*, deployed one per facility or site. All the monitoring functions, much like Astrolabe [14], take place in a single monolithic element.

The system architecture is sub-divided into four logical layers:

- The first layer contains the regional or high-level services (that use the data from other services), data repositories and clients. These are the consumers of information gathered by MonALISA and are able to store data.

- The second layer is composed by *proxies*. They allow for secure and reliable communications, dynamic load balancing, scalability and replication. Clients contact the proxies instead of communicating directly with the services. It allows the proxy service to perform operations over the requests. For example, allowing a service to only send the data once and then the proxy multiplexes it for the all the clients that subscribe to that information.

- The third layer is where *services* are located. The *services* make use of a multi-thread execution engine to perform the data collection and processing tasks. The multi-threaded execution allows the system to monitor a large number of entities, filter and aggregate the monitoring data, store monitoring information for shorter periods of time, manage web services for direct data access, provide triggers, alerts and actions based on monitoring data and control the system using dedicated modules.

  It also allows to perform independent data collection tasks in parallel. The monitoring modules are dynamically loaded and executed on independent tasks which allows to run concurrently a large

number of modules. Due to the use of independent threads, the failure of a monitoring task (due to node failure or delay) will not delay the other tasks.

- The fourth and last layer hosts the *lookup services* (LUS). Consists of a network of services that provide dynamic registration and discovery for all the components described above. MonALISA services are able to communicate and access each other at the global scale by registering themselves with LUS as part of one or more groups along with some attributes that describe themselves. In this way any interested application, service or client can request services based on a set of matching attributes.

The registration uses a lease mechanism. If a service fails to renew its lease, it is removed from the LUS and a notification is sent to all services or other applications that subscribed to such events. The scalability of the system comes from the use of the *multi-threaded execution engine* to host the loosely coupled services, and the use of the lookup service to register and discover services from the proxies that allow the servers to only send information once and then the proxy will multiplex it to all the interested parties.

2.7.5 Monasca

Monasca [21] is a centralized and highly modular monitoring solution. Its functions are isolated from each other and divided into different modules. It follows a micro-services architecture with several services split and responsible for a single function. Each module is designed to provide a discrete service in the overall monitoring solution and can be deployed or omitted according to the operators/customers need.

Instead of using gossip or a hierarchical tree like the other systems, Monasca's communication between processing functions is ensured by topics according to the publish/subscribe paradigm. The central module uses a Message Queue, like Apache Kafka [22], to provide temporary storage for the messages. The message queue has two type of users. Producers that create messages and deliver them, and consumers that connect to the queue and get the messages to be processed. Messages stay on the queue until they are retrieved by a consumer. This type of systems provide an *asynchronous communications protocol* between modules. Publishers do not need an immediate response to continue working, which decouples the different modules from each other as they do not need to interact directly.

Data collection is done using *agents* that execute the observation function on the remote resources. These *agents* capture the metrics from the remote resources and push them to the central Monasca Module, called *Monasca API*. Later, the API publishes the pushed metrics in the message queue under the topic "Metrics" so that they can be used by any other Monasca Modules.

Aggregation is done by the module "Transform Engine". It consumes the data from the "Metrics" topic and transforms it by applying an aggregation or mathematical function to obtain a new value. After

transforming the data it publishes the new values on the same topic “Metrics” so it can be used by the other modules. In order to store the data permanently Monasca has a module called “Persister” that takes the metrics from the Message Queue and puts them in a persistent database.

Monasca also has another modules that are responsible for the creation of alarms, events and notification that consume metrics from the Message Queue.

Due to fact that Monasca uses a centralized architecture and does not have an internal organization of its remote resources, the system is not capable of providing *locality* nor *resilience* against failures.

2.7.6 SDIMS

SDIMS [18] is a generic monitoring system that aggregates information about large-scale networked systems.

It implements the same hierarchical architecture as Astrolabe but instead of exposing all information to all the nodes of a subtree, it allows nodes to only access detailed views of nearby information and summary views of distant and global information.

It uses a modified Distributed Hash Table (DHT) algorithm extended from the Pastry’s protocol [23] to construct a tree spanning across all nodes in the system. Each physical node is a leaf and each subtree represents a logical group of nodes. These logical groups can correspond to *administrative domains* or groups of nodes within a domain (both equivalent to *zones* on Astrolabe).

An internal non-leaf node or *virtual node* is simulated by one or more physical nodes at the leaves of the subtree for which the *virtual node* is the root. The zones on the tree are created by exploiting the fact that each key in Plaxton-based DHT (which Pastry is based on) identifies a tree consisting of the routes from each other node to the root node for that key. The algorithm was modified to have a leaf set for each administrative domain, rather than a single set for the whole tree.

The authors adapted the Pastry’s protocol by changing its routing algorithm. Instead of using a single routing table based on network jumps, they changed the algorithm to have two different proximity metrics when creating the routing tables for the DHT. They use hierarchical domain proximity as its primary metric, which means that domains need to be declared before the tree is formed, and use network distance as a secondary metric.

Each physical node stores its metrics locally. The system associates an aggregation function with each attribute, and for each level-$(n+1)$ subtree in the tree it calculates the aggregated value using the values from level-n aggregated values.

Data collection uses a *pull* method. Values are directly gathered from the source nodes when the system wants to calculate the aggregate of those values.

The aggregated values can be propagated along the tree when calculated in order to provide some degree of *locality*.

While previous systems, like Astrolabe [14], provided a single static strategy for computing and propagating values, SDIMS provides flexible computation and propagation strategies by letting applications customize their propagation patterns to their needs.

This strategy allows the system to provide a wide range of strategies for data propagation in order to match the read-write-ratio of different applications.

SDIMS is able to provide this flexibility by having three operations that manipulate the system configuration:

- *Install()*: installs an aggregation function that defines an operation on an attribute and specifies the update strategy that it will use. It uses two parameters *up* and *down* that define how much the value should be propagated on the tree. The *up* value defines at which levels above the node that the aggregated value should be stored and the *down* parameter determines how much levels it should propagate down to its descendants.

- *Update()*: updates or adds a new value to a leaf node, allowing it to trigger a new aggregation.

- *Probe()*: returns the value of an attribute. An application can specify the level of the tree at which the answer is required for an attribute. It can also specify *up* and *down* parameters in order to ask for re-aggregation of the values taking into account those parameters.

Beyond the strategies already used in Pastry's [23], two more strategies are provided in order to deal with the problem of nodes leaving the system: *On-demand re-aggregation* and *replication in space*.

On-demand re-aggregation is done by using the *up* and *down* attributes of the Probe API application to force a re-aggregation. If an application detects that the aggregated values are stale it can re-issue the probe by increasing the *up* and *down* values, forcing the refresh of those values.

Replication in space is attained by using the *up* and *down* knobs in the Install API. With bigger values on each parameter, aggregates at the intermediate virtual nodes are propagated to more nodes in the system. It reduces the number of nodes that have to be accessed to answer a probe, which lowers the probability of incorrect results due to the failure of nodes that do not contribute to the aggregate.

2.7.7 Limits of existing systems in a Edge environment

Existing systems have dealt with the different characteristics for their intended environments. All of them perform some type of data aggregation, but due to their architecture and the fact that most of these tools were designed to work in centralized sites they either use a static topology to perform the aggregation and do not have the capabilities of modifying it without having to deal with complex mechanisms, manual re-configurations (like in Ganglia) or even restarts of the whole system. Or they are flexible but do not offer a hierarchical structure that can provide a good overview of the aggregation process (MonALISA and

Monasca). And in systems like SDIMS where regions are closely tied to the structure of the DHT it would prove even harder to re-configure a topology. This means that although they do have re-configuration processes they are mostly used to mitigate failures and not re-configure the data aggregation topology. And even FMonE with a design oriented to the fog/edge architecture and has the flexibility of attributing different *Pipelines* does not provide a mechanism to easily apply or modify those different rules.

Data aggregation can affect the performance of the system by reducing by a considerable amount the size of the transmitted data on the system if the topology is well adapted to the scenario. With a system that is deployed on the edge, the number of devices increases, when compared to the classic paradigms, and the load and distribution of nodes is constantly changing. It could prove beneficial to be able to re-configure the topology to adapt to the load that the system is being put under in order to optimize performance. With this, it feels natural that the next step is to create a monitoring system capable of functioning at the edge level, and that is able to adapt to different ongoing conditions by modifying its aggregation topology without overheads that could overshadow the benefits of making said change.

Summary

This chapter introduced the necessary function for a fully working monitoring system. It was also introduced the existent differences between edge and cloud services and a specification that defines the characteristics and necessary requirements for an Edge monitoring system. Several systems were analyzed to get a grasp of the used techniques and to understand how the aggregation structures work in monitoring systems. In the next chapter, it is proposed a system that focus on satisfying some of these requirements, specifically dynamically define the data aggregation topology, and increase performance on the edge.

3

Design

Contents

3.1 Goals 23
3.2 Design 24
3.3 Data Collection and Processing 25
3.4 Data Model 29
3.5 Protocols 31

This chapter introduces the design of FlexRegMon, a system architecture that aims at supporting the reconfiguration of the aggregation topology with low latency. The architecture aims at simplifying the task of adding new level in the hierarchy and also the task of adding new regions to a given level. The architecture relies on a logically centralized component, the *Central Orchestrator* (that oversees the whole topology), whose role is to coordinate the remaining components. The operation of the Central Orchestrator is supported by multiple partial replicas, located in different geographic locations. Each partial replica only keeps part of the state owned by the Central Orchestrator: the part that is relevant to manage the components that execute on the local region. When changes to the topology are performed, the Central Orchestrator delegates the task of coordinating the affected components to its own replicas, such that the coordination activities are performed by the replicas that is closer to the affected components, with the aim of speeding up the reconfiguration. Section 3.1 expresses the goals that we want the system to fulfill. Section 3.2 gives an overview of the system design, its components and how they are connected. Section 3.3 describes how data collection and processing works in the system. Section 3.4 refers to how the data is maintained throughout the system. And finally, Section 3.5 details the different protocols and algorithms for operations that can be performed in the system.

3.1 Goals

In many edge applications, the load on a given region is variable, and depends on the number of users that are located on that region at a given point in time. In some cases, some regions may be subject to temporary spikes in demand. Consider, for instance, scenarios where users can physically can gather in large numbers for short periods (i.e., a musical festival in a city). If the existing servers are statically assigned to geographical regions, it may be hard to ensure that the right amount of resources is allocated to each region. The system may be over-provisioned for normal operation or under-provisioned when an special event makes many users to gather in a given location. When the workload exceeds the capacity of the system, users may experience delays or even loss of information.

Horizontal scaling is useful to address many of the problems that arise from overloaded computing resources [24]. By allocating more resources to a region it is possible to increase the computational power needed to process a higher number of requests. In addition, data aggregation, a technique effectively used in wireless sensor networks [25], can be used to mitigate the stress on the network. This technique allows the information to be divided and summarized into smaller portions, which makes the transmission much faster.

To deal with dynamic workloads we aim at supporting a re-configurable hierarchical composition of the nodes used to process and aggregate incoming data. These nodes, named the *aggregators*, are organized in a logical tree. The number of leaf nodes associated with any given geographical region may

be re-assigned in run-time. Also, the depth of the tree can also changed dynamically, to add intermediate aggregation points when needed. The idea of using hierarchical networks of data aggregators is not new, and has been widely used in monitoring systems [14, 18]. Our work focus on techniques that allow to re-configure this hierarchy in run-time.

3.2 Design

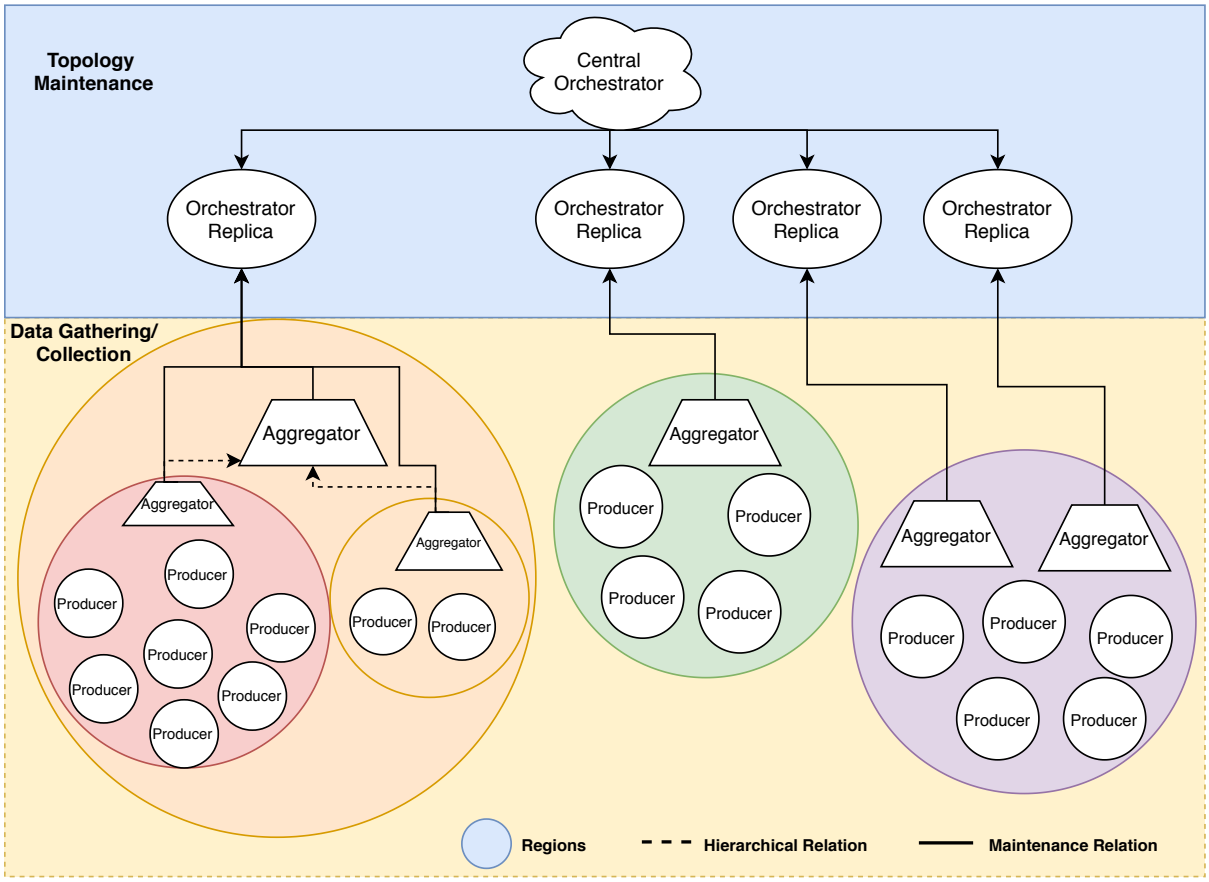


Figure 3.1: FlexRegMon general structure

Figure 3.1 depicts the main architecture of FlexRegMon. We incorporate FlexRegMon nodes into the classic edge architecture, by designing it within the following structure: a central layer, in which we allocate a centralized orchestrator, where we maintain the whole hierarchical topology, and other system maintenance information; an edge/fog layer with wide-spread fog nodes or cloudlets where we are able to allocate the remaining services for the system:

- **Orchestrator-Replicas** - partial replicas of the central orchestrator.

- **Aggregators** - nodes that store, process, aggregate, and forward the information generated by the producers.
- **Producers** - nodes that produce data and feed it to their attributed aggregator.

All of these services can be deployed on multiple nodes per region to allow the system to scale. The Central Orchestrator component keeps record of the whole topology, it is responsible for propagating any changes to its own partial replicas, and acts as the initial point of contact for topology changes to be made by the Control System. Each region is composed by at least one Orchestrator-Replica and one aggregator. Due to the nature of the architecture of the system it is possible to deploy the different types of node on different kinds of devices, as described in Section 2.4:

- *Cloud Devices*, servers located in big datacenters built in specific areas of the globe. Due to their fixed location the communication conditions with the system's nodes might vary depending on the location of said node. Internally the servers have big can act as *Orchestrator*, its replicas, and aggregators nodes;
- *Fog Devices*, located in smaller sized datacenters, called *cloudlets*, distributed across the globe, can act as aggregators for a sub-region or as an Orchestrator-Replica
- *Edge Devices*, mobile devices with low computation power. Can host the data producers which generate the data sets that are sent to the system.

3.3 Data Collection and Processing

As previously described in Section 2.4, in an edge system the number of nodes increases as we approach the outer layers. Because most of the data in these type of systems is produced in these dense regions, we decided to take the approach of submitting data with a *Push* based technique. Here the producers are actively responsible for sending the data to the monitoring system (please refer to Section 2.1 for more details on data collection approaches).

Producers connect to the corresponding aggregators and submit their data sets following the scheme described on Section 3.4. The aggregators keep those data sets and are responsible to relay the information to the upper levels of the hierarchical tree. They achieve this by contacting the upper hierarchical level aggregators, just like a producer, and submit the data sets to those nodes.

Due to the nature of the edge network, the connections between different aggregators have limited bandwidth. Problems such as link congestion, and message loss, can be amplified in environments where large amounts of data are transferred through the network, a scenario that is likely to happen in a system like ours that has a steady flow of information coming from producers.

To ease the process of data propagation, and reduce overhead and possible connection problems between hierarchical levels, we assume that the aggregators are able to use aggregation functions, such as the ones described in Section 2.2.1. These functions help to reduce the size of the data transmitted between levels, and consequently the overhead of sending data up on hierarchical tree. With our data model, we aggregate the values by taking all data sets that have the same context and summarize them into a single value. This aggregation can be performed with just one region, or with multiple regions, the only restriction for aggregation is having the same context. The operation can be any function that combines multiple values into a single one (such as the average).

As an example, consider the topology present in Figure 3.2 and the data storage represented on Tables 3.1 and 3.2. On Table 3.1, we have the collected data sets for the *Temperature* and *Humidity* of *Lisboa* and *Setúbal* regions. *AvgTemp* and *AvgHumidity* are generated from an average aggregation function that summarizes the *Temperature* and *Humitidy* values per region. Eventually, the aggregator relays *AvgTemp* and *AvgHumidity* to the Portugal's region (depicted in Table 3.2). In this case, because the aggregator is composed by multiple sub-regions, it can instead of performing an aggregation per region, create the values for its own region by aggregating the average from both sub-regions (Lisboa and Setúbal).

Table 3.1: Data Sets: Lisboa and Setúbal aggregator

Context	Lisboa	Setúbal
Temperature	22,23,24	26,27,25
Humidity	79,75,78	67,69,71
AvgTemp *	23	26
AvgHumidity *	77.3	69

Table 3.2: Example of a data-sets table for the Portugal Region

Context	Lisboa	Setúbal	Portugal
AvgTemp	23	26	24.5 *
AvgHumidity	77.3	69	73.15 *

* - value is calculated by the aggregator from the other values

3.3.1 Orchestrator

The Orchestrator is a logically centralized component localized in the control layer. This component maintains the record of the entire topology, which regions are replicated in each region, and other information relevant to the maintenance of the system.

We assume the existence of an external *Control System*, that is capable of monitoring the workload and status of each node of the system. It is also capable of deploying additional nodes, provide them

with all the necessary information to enter the system, and re-configure the aggregation tree accordingly. After insertion into the system, all changes to node information are performed in the Central Orchestrator, which then propagates it to its own replicas. In this thesis we do not discuss the policies that may trigger the reconfiguration of the tree. Instead, we focus on the process of notifying the aggregators affected by a reconfiguration of their new parents.

Concerning the topology information, the Central Orchestrator keeps track of all the different regions, their corresponding parent region, children sub-regions, Orchestrator-Replicas nodes and it's associated regions. To create a new hierarchical level, the control system creates on the Central Orchestrator a new register for the new region, sets the parent and children regions for the new region, and then attributes a replica and an aggregator to the newly created region. If the new level is inserted in the middle of the hierarchical tree, it is necessary to change the values of adjacent regions. It needs to remap the parent value of all the sub-regions connected to the previously existent level, and also needs to remap the child values of the region it takes as its parent. The process is similar when removing a level.

Figure 3.2 illustrates this process. The picture on the left shows an initial hierarchical tree with 4 levels. The one on the right shows the resulting hierarchy after adding an intermediate level (the Centro-Sul region) between the Portugal region and Lisboa/Setúbal regions. In this case, the children of the Portugal node should change from [Lisboa, Setúbal, Porto] to [Centro-Sul, Porto], and the parent node of both Lisboa and Setúbal regions should be set to Centro-Sul.

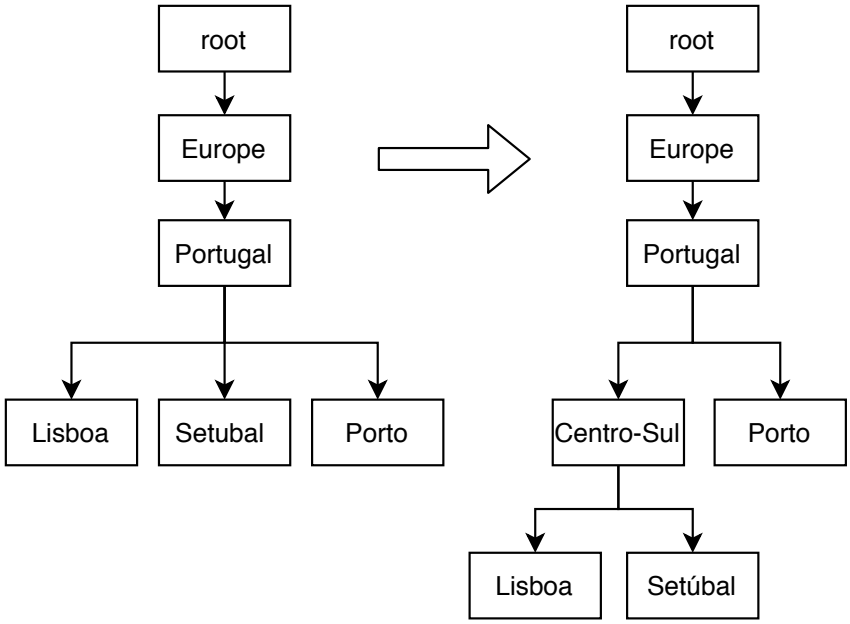


Figure 3.2: Hierarchical Tree after new level addition

3.3.1.A Orchestrator-Replicas

Each replica is assigned to a region by the external Control System and maintains a partial topology of the hierarchical structure that corresponds to the assigned region. When a topology update occurs, these replicas are notified and updated by the Central Orchestrator. The role of these replicas is to interact with aggregators and producers on behalf of the Central Orchestrator. Because those types of nodes only require a partial topology, instead of establishing a connection to the Central Orchestrator, they can connect to replicas to avoid long-latency links, and still retain the ability to receive topology updates, and query about the topology, and IP addresses of other nodes.

3.3.2 Aggregators

aggregators are responsible for running the services that collect and process the data-sets submitted to the system. They receive and store data from end-users, sensors, and information producer nodes in the system. Each aggregator is assigned to at least one region, and those regions are arranged into a hierarchical structure, where data flow up the hierarchical tree (from one aggregator to its parent aggregator). To perform these roles, while being able to adapt to topology changes, every aggregator is required to connect to an Orchestrator-Replica (one per region it is associated with) in order to keep receiving topology updates.

3.3.3 Producers

The producers is generally attributed to nodes located on the edge of the network, but it can be attributed to any other device on the network. This means other services can also partake on the system as producers. Producers join the system by receiving the command from the *Control System*, indicating the region, the aggregator and the Orchestrator-Replica that they should attach to receive updates. Producers connect to aggregators to stream the data-sets following the data model explained in Section 3.4.

3.3.4 Node Bootstrap

As said before, we assume the existence of an external *Control System* that is able to deploy nodes and provide them with the necessary information to integrate the system. For Orchestrator-Replicas this means provision of the region they are representing, their partial topology and the IP address of the Central Orchestrator to maintain the connection to receive updates. For the aggregators, it is necessary the associated region, topology information and which Orchestrator-Replica to connect in order to receive topology updates and obtain the parent level aggregator. Producers only require their region and replica IP address to start functioning.

3.3.5 Service Discovery and Failures

When a failure occurs and a node becomes unavailable, the system needs to make sure that nodes that were connected to the point of failure are re-inserted to the system. To achieve this we make use of the information maintained by the replicas to find a new IP Address for a service that replaces the missing node and restores the connection with the partitioned portion of the system. The task of detecting a node failure is assigned as follows. The Central Orchestrator monitors its replicas and each replica monitors the aggregators in its own region. When a failure is detected, in order to not provide IP addresses of malfunctioning nodes, we rewrite the stored IP address information on the system (see Section 3.4.2 for further details) and remove the failed node's register. Failures that are not directly detected by the Central Orchestrator are relayed to it by the replicas. Then it can feed the external Control System with up to data information regarding the system status; in turn, the control system decides which corrective actions should be pursued (i.e.: create/remove a node, region, level, etc.)

3.4 Data Model

The Central Orchestrator and its replicas keep the hierarchical topology information in a *Hierarchical Table* (Section 3.4.1) that matches regions to their sub-regions and to the parent value. The Central Orchestrator stores the entire topology while the each Orchestrator-Replica keeps only that part of the topology associated with its own region. Replicas also keep a *Routing Table* (see Section 3.4.2) that matches regions to their aggregator IP addresses. These tables are cached on each Orchestrator-Replica and are updated by the Central Orchestrator, that ensures that when any topology modification occurs, the affected replicas are correctly notified and updated.

3.4.1 Hierarchical Structure

As noted before, the hierarchical topology is maintained by the *Orchestrator* and its replicas. Each one of these nodes keeps an *Hierarchical Table* (illustrated in Table 3.1), that matches regions to their sub-regions, and parent. The information on these tables is propagated from the Central Orchestrator to the Orchestrator-Replicas, and subsequently from the Orchestrator-Replicas to the aggregators and producers. Aggregators and producers can receive their topology information from the replicas as these node only require the information for their region, parent value and sub-regions. This reduces the amount of information that each replica needs to maintain to serve a specific region. Upon a topology update from the Control System, the Central Orchestrator sends the updates to the corresponding replicas, and those replicas modify their tables and relay the updates to the affected nodes that are assigned to them. As an example, consider the topology depicted in Figure 3.2. If the Control System were to remove the

Centro-Sul region from the Central Orchestrator, it would trigger the Central Orchestrator to notify the affected Orchestrator-Replicas. After updating each replica would notify their affected aggregators.

In this specific case, because Centro-Sul is a sub-region of Portugal and parent of Lisboa and Setúbal, it would affect Portugal, Lisboa, and Setúbal regions. This means that the Central Orchestrator would need to contact all the Orchestrator-Replicas that maintained information for those regions, so they could change update their own tables. For Portugal region, it would change its sub-regions from [Centro-Sul, Porto] to [Lisboa, Setúbal, Porto], and the parent value of Lisboa and Setúbal would change from Centro-Sul to Portugal.

The query operations that are available to connected nodes provide all the information necessary to propagate and maintain topology information throughout the system. Nodes can determine sub-regions, parent of regions:

- **Parent of *region*** - Check *parent* value column of *region* or reverse-query *sub-regions*.
- **Sub-regions of *region*** - Consults *sub-regions* of *region* and returns the list of sub-regions.

Table 3.3: Example of a partial Hierarchical table for the Portugal Region

Region	Sub-regions	Parent
Portugal	Sul, Centro, Norte	Europe
Norte	Porto, Braga	Portugal
Sul	Faro	Portugal
Centro	Setúbal, Lisboa	Portugal

3.4.2 Routing Information

Like with hierarchical information, the Central Orchestrator and its replicas keep a registry of every node connected that is currently serving each region. The store this information in a *Routing Table* (see Table 3.4) that matches regions to the node IP addresses that serve those regions. If a node is working and is attributed to the region, its IP address will be present in this table. While the Central Orchestrator keeps track of all the routing information of the system, it only directly tracks the status of its replicas. Each replica maintains the *Routing Table* for their own aggregators and provides that information to the central node. Upon joining the system, each aggregator establishes a connection between itself and their corresponding Orchestrator-Replica to have access to topology information updates. This process creates an entry on the Orchestrator-Replica's *Routing Table* entry and while the connection remains intact the node will be kept on the table. If, for any reason, the connection is closed, the Orchestrator-Replica removes the corresponding node from the table, making it unreachable inside the system. The closure of the connection can come from a intentional operation or caused by a failure of the node or network.

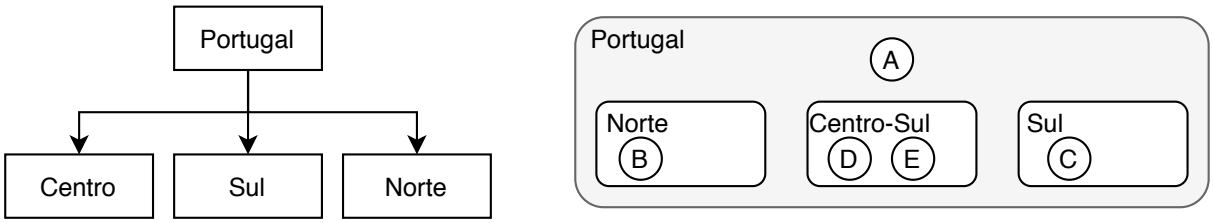


Figure 3.3: Partial Topology with Portugal and its sub-regions (left) and aggregators available for each region (right)

Taking the topology and aggregator distribution depicted in Figure 3.3, and assuming a single replica for all the represented region we would generate the following table (in this table, letters represent IP addresses):

Table 3.4: Example of a Routing table for the Orchestrator-Replica responsible for Portugal

Region	*Aggregators*
Portugal	A
Norte	B
Sul	C
Centro-Sul	D, E

Note: Letters represent IP addresses

3.4.3 Data Sets

As described before (see Section 3.3), data is sent to aggregators where it is stored, processed and propagated. This data is stored under two constraints, context and region. The first one classifies the value and what it represents (for example, temperature, CPU usage, number of nodes). And the second one restricts the scope of the data set, which is defined by the origin of the data collection/production.

Table 3.5: Example of data set stored in an aggregator

Context	Lisboa	Setúbal
Temperature	22,23,24	26,27,25
Humidity	79,75,78	67,69,71

3.5 Protocols

In this section, we provide further details and descriptions of the FlexRegMon topology changes and re-configurations protocols accompanied by their algorithms and message sequence diagrams.

3.5.1 Topology Re-Configurations

Because the main focus of our system is ease of topology re-configurations, we designed different protocols to make use of the hierarchical architecture while allowing re-configurations. A topology re-configuration requires a modification on the Central Orchestrator, which will then be propagated to the corresponding replicas and at then relayed to the rest of the nodes.

The modification is performed by the Control System, which changes the registries on the Central Orchestrator. Then the central node propagates the information to every affected replica. After receiving the information, each replica alters their topology and relays the same message to the connected nodes. Every re-configuration executes the following sequence of steps:

1. Control System changes the Central Orchestractor's registries

2. Central Orchestrator sends the updates to the affected Orchestrator-Replicas.

3. Orchestrator-Replicas receive re-configuration and update their registries.

4. Orchestrator-Replicas send updates to connected nodes (aggregators).

By following this relay strategy, it allows us to send information as quickly and reliably as possible by using lower latency connections that the replicas have to their aggregators, in contrast of using the long-distance connection between aggregators and the central node. This is beneficial to our system as the number of aggregators is much bigger than the number of Orchestrator-Replicas, which would translate in a big overhead due to the number of long distance messages sent by the Orchestrator. With our protocol, this problem is mitigated as the Central Orchestrator only needs to propagate the initial notifications to the Orchestrator-Replicas and let them relay it to other nodes.

With this, our topology change notification system allows us to several operations that we can use to add, remove regions and perform hierarchical re-configurations, with insertions in the middle of the hierarchical tree. This will provide better adaptability of the system to the different situations that it could encounter in different scenarios.

In our system we can have two situations when performing a re-configuration. One where we add or remove region to the end of a hierarchical branch. This scenarios is less disruptive to the hierarchical structure as there is no need reconnect the hierarchical structure. The other situation occurs when we insert/remove regions in the middle of the topology tree. This operation causes the hierarchical structure to break into two different trees which requires extra steps to merge. We differentiate the two situations, by calling the first one *adding and removing a region* and the second one *adding/removing a hierarchical level*. We describe both situations and used protocols in detail in the following sections.

3.5.1.A Steps to Add or Remove a Region

The steps to add a new region to the system's topology are the following:

1. The Control System adds/removes a region to the Central Orchestrator, and changes the topology values to reflect the operations to change the hierarchical tree in the desired place.

2. The Central Orchestrator sends updates to the affected Orchestrator-Replicas.

3. The Orchestrator-Replicas receive re-configuration, update their registries and relay information to affected aggregators.

4. The aggregators adapt to re-configuration and, if necessary, relay necessary information to producers.

The first step is further detailed in the Algorithms 1, 2 and 3, where the Control System invokes those functions. The entire processes is described in the following algorithm descriptions and in Figures 3.4 and 3.5. The major difference between adding and removing a region, consists on the information that is necessary to transmit to the sub-regions and their aggregators, in some cases it's necessary to update the parent value, in other the sub-regions values.

Algorithm 1 Central Orchestrator Code - Add region

```
1:  function ADDREGION(parent_region, new_region, replicas)
2:      hierarchical_table.addEntry(new_region)
3:      hierarchical_table[new_region][parent] ← parent_region
4:      hierarchical_table[new_region][replicas].add(replicas)
5:      hierarchical_table[parent_region][sub_regions].add(new_region)
6:
7:      for orch_replica ∈ hierarchical_table[parent_region][replicas] do
8:          send ⟨ add_region parent_region, new_region ⟩ to orch_replica
9:      for sub_region ∈ hierarchical_table[parent_region][sub_regions] do
10:         for orch_replica ∈ sub_region[replicas] do
11:             send ⟨ add_region parent_region, new_region ⟩ to orch_replica
```

Algorithm 2 Orchestrator-Replicas Code - Add region

```
1:  function ADDREGION(parent_region, new_region, replicas)
2:      hierarchical_table.addEntry(new_region)
3:      hierarchical_table[new_region][parent] ← parent_region
4:      hierarchical_table[new_region][replicas].add(replicas)
5:      hierarchical_table[parent_region][sub_regions].add(new_region)
6:
7:      if self is replica for parent_region then
8:          for aggregator ∈ hierarchical_table[parent_region][aggregators] do
9:              send ⟨ add_region parent_region, new_region ⟩ to aggregator
```

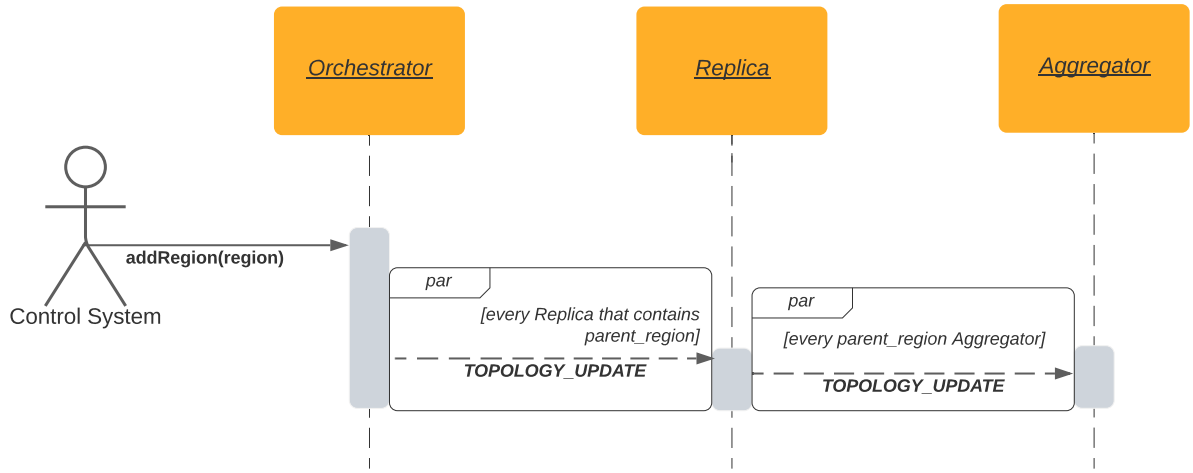


Figure 3.4: Sequence diagram for adding a region

Algorithm 3 Central Orchestrator Code - Remove region

```
1:  function REMOVEREGION(region)
2:      parent_region ← hierarchical_table[region][parent]
3:      hierarchical_table[parent_region][sub_regions].remove(region)
4:      replicas_with_region ← hierarchical_table[parent_region][replicas] + hierarchical_table[region][replicas]
5:
6:      for sub_region ∈ hierarchical_table[region][sub_regions] do
7:          replicas_with_region.add(hierarchical_table[sub_region][replicas])
8:      for replica ∈ replicas_with_region do
9:          send ⟨ remove_region region, parent_region ⟩ to replica
10:
11:     hierarchical_table.removeEntry(new_region)
```

Algorithm 4 Orchestrator-Replicas Code - Remove region

```
1:  function REMOVEREGION(region, parent_region)
2:
3:      hierarchical_table[parent_region][sub_regions].remove(region)
4:
5:      if self is replica for parent_region then
6:          for aggregator ∈ hierarchical_table[parent_region][aggregators] do
7:              send ⟨ remove_sub_region region ⟩ to aggregator
8:      if self is replica for region then
9:          for aggregator ∈ hierarchical_table[region][aggregators] do
10:             send ⟨ remove_region region, parent_region ⟩ to aggregator
11:     for sub_region ∈ hierarchical_table[region][sub_region] do
12:         if self is replica for sub_region then
13:             for aggregator ∈ hierarchical_table[sub_region][aggregators] do
14:                 send ⟨ change_parent sub_region, parent_region ⟩ to aggregator
15:
16:     hierarchical_table.removeEntry(new_region)
```

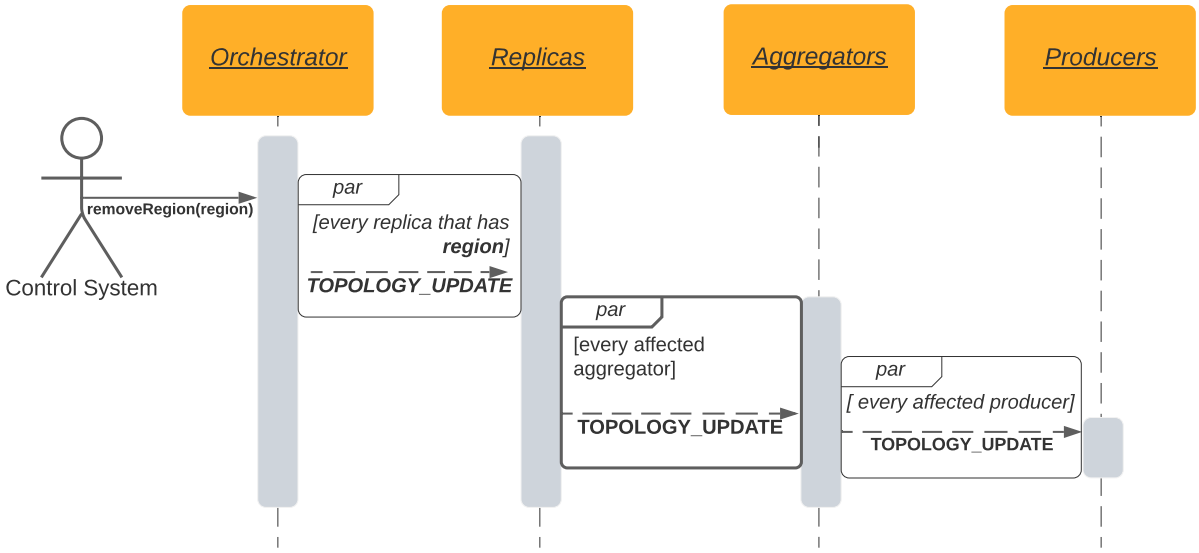


Figure 3.5: Sequence diagram for removing a region

3.5.1.B Adding or Removing Hierarchical Levels

The steps for adding/removing a hierarchical level are as follows:

1. The Control System adds/removes a region in the middle of the hierarchical tree on the Central Orchestrator, and changes the topology values to put the region in the desired place in the hierarchical tree.

2. If necessary, the Control System either creates new nodes or sets new roles to existent nodes, in

order to serve the topology.

3. The Central Orchestrator sends the updates to affected replicas.

4. The Orchestrator-Replicas receive re-configuration, update their registries and relay information to affected aggregators.

5. The aggregators adapt to re-configuration and relay necessary information to producers.

6. If necessary, nodes connect to new *services*.

The initial steps to *add/remove a hierarchical level* to the topology are much similar to the previous example. The main difference comes from having to insert the new region between two existent regions which creates the need to have different internal messages to be sent in the system to complete the re-configuration. The protocol is further detailed in Algorithms 5, 6, 7 and illustrated in Figure 3.2.

Algorithm 5 Central Orchestrator Code - Add Hierarchical Level

```
1:  function ADDHIERARCHICALLEVEL(parent_region, new_region, replicas)
2:      hierarchical_table.addEntry(new_region)
3:      hierarchical_table[new_region][parent] ← parent_region
4:      hierarchical_table[new_region][replicas].add(replicas)
5:      hierarchical_table[new_region][sub_regions]← hierarchical_table[parent_region][sub_regions]
6:      hierarchical_table[parent_region][sub_regions]← [new_region]
7:
8:      for orch_replica ∈ hierarchical_table[parent_region][replicas] do
9:          send ⟨ addHierarhicalLevel parent_region, new_region, replicas ⟩ to orch_replica
10:     for sub_region ∈ hierarchical_table[new_region][sub_regions] do
11:         hierarchical_table[sub_region][parent] ← new_region
12:         for orch_replica ∈ sub_region[replicas] do
13:             send ⟨ addHierarchicalLevel parent_region, new_region, replicas ⟩ to orch_replica
```

Algorithm 6 Orchestrator-Replica Code - Add Hierarchical Level

```
1:  function ADDHIERARCHICALLEVEL(parent_region, new_region, replica)
2:      hierarchical_table.addEntry(new_region)
3:      hierarchical_table[new_region][parent] ← parent_region
4:      hierarchical_table[new_region][replicas].add(replicas)
5:      hierarchical_table[new_region][sub_regions]← hierarchical_table[parent_region][sub_regions]
6:      hierarchical_table[parent_region][sub_regions]← [new_region]
7:
8:      if self is replica for parent_region then
9:          for aggregator ∈ hierarchical_table[parent_region][aggregators] do
10:             send ⟨ addHierarhicalLevel parent_region, new_region ⟩ to aggregator
11:     for sub_region ∈ hierarchical_table[region][sub_regions] do
12:         hierarchical_table[sub_region][parent] ← new_region
13:         if self is replica for sub_region then
14:             for aggregator ∈ hierarchical_table[sub_region][aggregators] do
15:                 send ⟨ addHierarhicalLevel parent_region, new_region, replica ⟩ to aggregator
```

Algorithm 7 Aggregator Code - Add New Hierarchical Level

```
1:  function ADDHIERARCHICALLEVEL(parent_region, new_region, replica)
2:      hierarchical_table[new_region][parent] ← parent_region
3:
4:      if self is aggregator for parent_region then
5:          hierarchical_table[parent_region][sub_regions] ← [new_region]
6:
7:      if self is aggregator for sub_region of new_region then
8:          parent_replica ← replica
9:          hierarchical[sub_region][parent] ← new_region
10:         send ⟨ getAggregator parent_region ⟩ to parent_replica
11:         when
12:         receive ⟨ getAggregatorResponse aggregator ⟩ from parent_replica
13:         do
14:         parent_aggregator ← aggregator
```

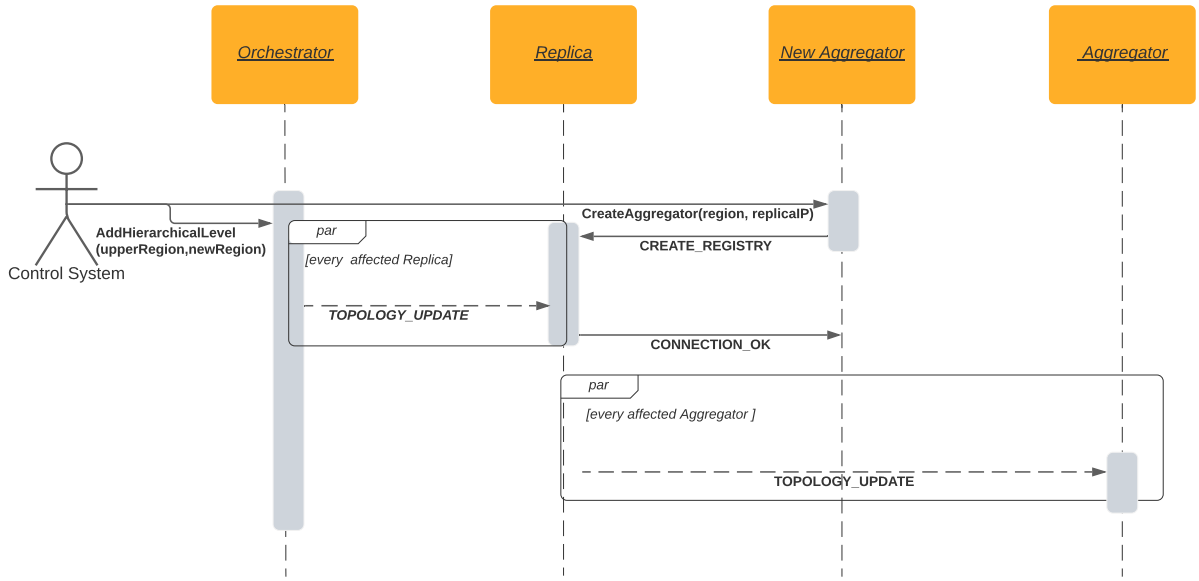


Figure 3.6: Sequence Diagram for adding a new hierarchical level

To remove a hierarchical level, it is necessary to remove every region that belongs to that same level. Only after removing the last region will the level be deleted. The communication pattern of removing a region is similar with the protocol to add a new level. The Central Orchestrator sends the topology changes to affected Orchestrator-Replicas and those relay the change to aggregators. When removing a hierarchical level, we take its upper hierarchical level and its lower level and connect them. The sub-regions change their parent value and the parent adds the sub-regions to its own. Like the previous example, the process is depicted in Figure 3.7 and in the following algorithms:

Algorithm 8 Central Orchestrator Code - Remove Hierarchical Level

```
1: function REMOVEHIERARCHICALLEVEL(region)
2:     parent_level ← hierarchical_table[region][parent]
3:     rem_sub_regions ← hierarchical_table[region][sub_regions]
4:
5:     for replicas ∈ hierarchical_table[parent_level][replicas] do
6:         send ⟨ newSubRegions parent_level, rem_sub_regions ⟩ to replica ▷Notify parent replicas of new sub_regions
7:
8:     for replica ∈ hierarchical[region][replicas] do
9:         send ⟨ removeHierarchicalLevel region ⟩ to replica ▷Notify replicas of region removal
10:
11:    for sub_region ∈ sub_regions do
12:        hierarchical_table[sub_region][parent] ← parent_level
13:        for replica ∈ routing_table[sub_region][replicas] do
14:            send ⟨ removeHierarchicalLevel region ⟩ to replica ▷Notify sub_regions of parent change
15:
16:    hierarchical_table[parent_region][sub_regions].remove(region)
17:    hierarchical_table[parent_region][sub_regions].add(rem_sub_regions)
18:    hierarchical_table.remove(region)
```

Algorithm 9 Orchestrator-Replica Code - Remove Hierarchical Level

```
1: function REMOVEHIERARCHICALLEVEL(region)
2:     parent_level ← hierarchical_table[region][parent]
3:     rem_sub_regions ← hierarchical_table[region][sub_regions]
4:     hierarchical_table[parent_region][sub_regions].remove(region)
5:     hierarchical_table[parent_region][sub_regions].add(rem_sub_regions)
6:
7:     if self is replica for parent_level then
8:         for aggregator ∈ routing_table[parent_level][aggregators] do
9:             send ⟨ removeHierarchicalLevel sub_regions, region ⟩ to aggregator ▷Notify parent aggregators
10:    if self is replica for region then
11:        for aggregator ∈ routing_table[region][aggregators] do
12:            send ⟨ removeRole region ⟩ to aggregator ▷Removes role from region aggregators
13:
14:    parent_replicas ← routing_table[parent_level][replicas]
15:    for sub_region ∈ rem_sub_regions do
16:        if self is replica for region then
17:            for aggregator ∈ routing_table[sub_region][aggregators] do
18:                send ⟨ changeParent sub_region, region, parent_level, parent_replicas ⟩ to aggregator
▷Removes region from sub_regions aggregators and reconnects the them to the parent_level
19:    hierarchical_table.remove(region)
20:    routing_table.remove(region)
```

Algorithm 10 Aggregator Code - Remove Hierarchical Level

```
1:  var aggregator_roles                                  ▷List with all region that aggregator represents
2:
3:  function CHANGEPARENT(sub_region, parent_level, replicas)
4:      hierarchical_table[sub_region][parent] ← parent_level
5:
6:      send ⟨ getAggregator parent_region ⟩ to parent_replicas
7:      when
8:      receive ⟨ getAggregatorResponse aggregator ⟩ from parent_replicas
9:      do
10:     parent_aggregator ← aggregator
11:
12: function REMOVEROLE(region)
13:     aggregator_roles.remove(region)
14:     for producer ∈ routing_table[region][producers] do
15:         send ⟨ regionRemove region ⟩ to producer
16:     routing_table.remove(region)
17:
18: function REMOVEHIERARCHICALLEVEL(region, parent_level, sub_regions)   ▷Necessary update to
    perform aggregations of parent_level
19:     hierarchical_table[parent_level][sub_regions].remove(region)
20:     hierarchical_table[parent_level][sub_regions].add(sub_regions)
```

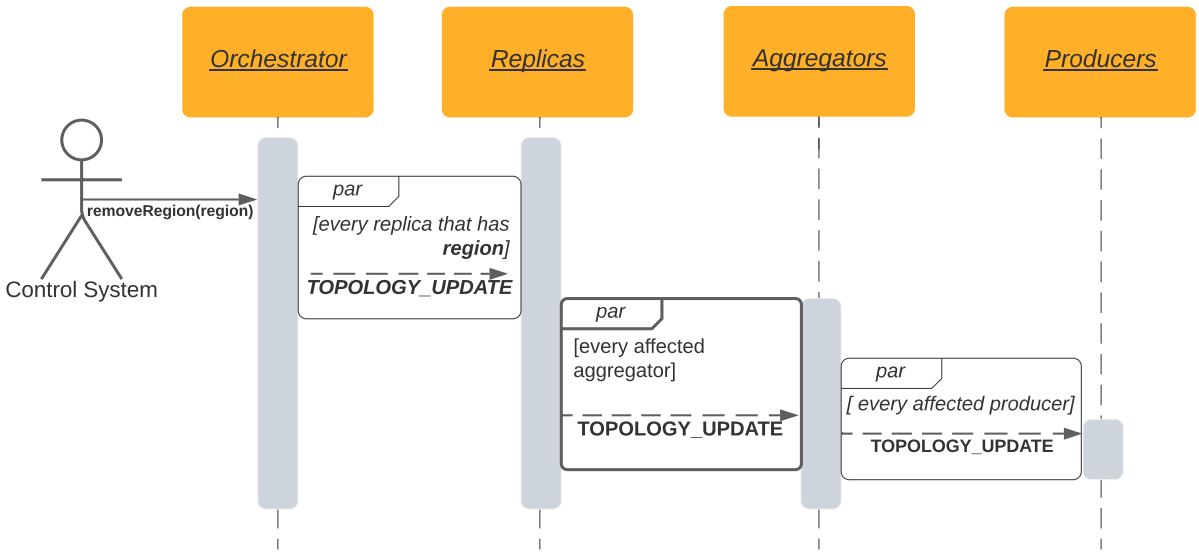


Figure 3.7: Sequence Diagram for propagating hierarchical level removal

3.5.2 Aggregator Removal

As described in Section 3.3.5, the mechanism to automatically remove an aggregator from the system function with base on *Routing Table*. An aggregator is removed from region when its connection with a Replica is closed. This can happen in two different situations: the first one occurs when the Control

System manually removes the aggregator from the system. This forces the aggregator to close its connection with all connected nodes. The second situation can happen when uncontrolled circumstances break the channel between aggregators and their assigned Orchestrator-Replicas, or between aggregators and producers. The channel can break due to network partitions, network failures, or node crashes. Each of these situations can lead to a broken session, which results in the Orchestrator-Replica removing aggregator IP addresses from the *Routing Table*, meaning that the faulty aggregator will no longer be reachable.

While the Orchestrator-Replicas can easily detect the aggregator failures, producers, depending on the reason behind the broken connection, can receive the notification for the failure in two different ways. The first one occurs if the connection is closed due to an aggregator crash/closure. In this case, the connection between aggregator and producers is also severed. Producers are free to contact their Orchestrator-Replica and acquire a new aggregator IP address and keep working. The second happens when the problem comes from a network failure/partition between aggregator and the Orchestrator-Replica, or the extreme of having a Orchestrator-Replica crash (this situation is detailed in Subsection 3.5.3). In this case, from the points of view of the aggregator and producers there is no apparent problem with those services. To verify this, the aggregator tries to contact the *Central* Orchestrator to obtain a new IP for a new Orchestrator-Replica. If not successful it drops all connections with its producers to allow them to connect to other aggregator. Detailed sequence diagrams with examples for both aggregator removal scenarios are depicted in Figures 3.8 and 3.9

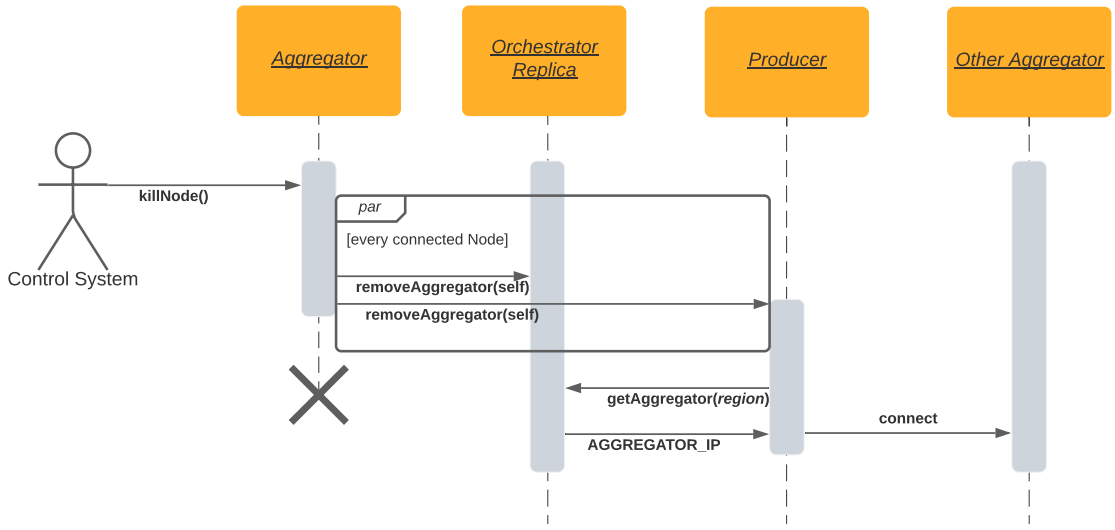


Figure 3.8: Sequence Diagram for aggregator removal in a controlled scenario (System Control kills aggregator)

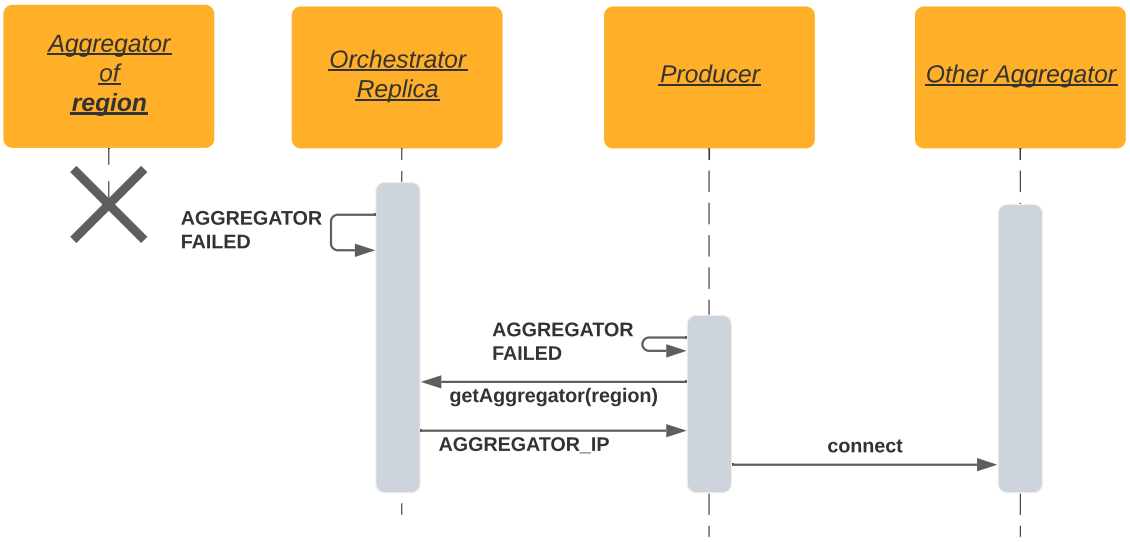


Figure 3.9: Sequence Diagram for aggregator removal in a uncontrolled scenario (aggregator crash)

3.5.3 Orchestrator-Replica Failure

The process to remove a faulty Orchestrator-Replica from the system is similar to removing aggregator nodes. Just like with aggregators, failures can originate either from a node crash or from network problems, like partitions and failures. In a node crash, every connection that the replica had is closed, and every node instantly acknowledges the node failure and proceeds with the necessary protocols to reconnect to the system. In the case of network partitions, it can generate multiple situations, as the Orchestrator-Replica might be able to communicate with part of the network. If the network partition occurs between the Orchestrator-Replica and the Central Orchestrator, the Orchestrator-Replica breaks the connection with every node, in order to release them.

A more complex problems arises from network partitions between a Orchestrator-Replica and its aggregators. As a replica can serve multiple regions, the network partition might only affect just one region and not the other. To deal with this, when a Orchestrator-Replica notices multiple simultaneous aggregators disconnections it will notify the Central Orchestrator, which will trigger the Control System to assess if there is a network partition and if there is, re-adapt the system to mitigate the efects of the failure (create a new node, reallocate roles, etc.). Both network partition situations are depicted in the following sequence diagrams (Figures 3.10 and 3.11).

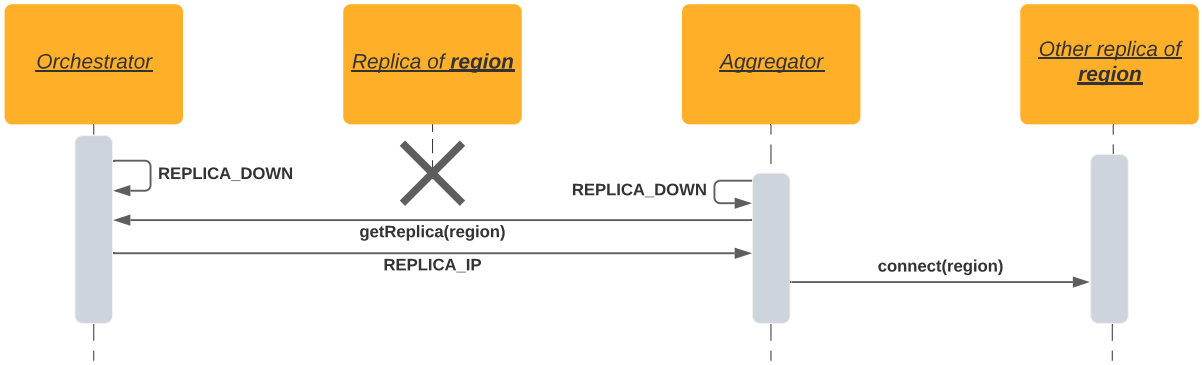


Figure 3.10: Example of procedure for Replica crash or network partition between *Orchestrator-Replica* and Orchestrator

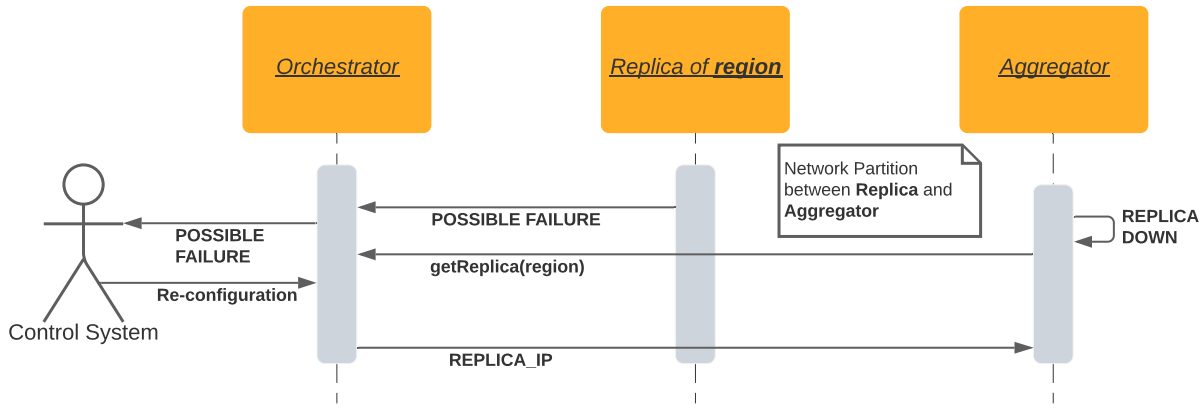


Figure 3.11: Network partition between Replica and aggregators

Summary

In this chapter, it was presented the design of FLEXREGMON, it's components and how they interact with each other. FLEXREGMON presents a distributed management model that is able to provide faster topology updates at the edge and reduce management operations overhead. We also describe all the protocols and algorithms that allow to manipulate the topology and how it propagates across the system.

In the next chapter we present the specific implementation of FLEXREGMON and technologies used.

4

Implementation

Contents

4.1 Zookeeper 45
4.2 FlexRegMon Internal Zookeeper Structure 46
4.3 FlexRegMon Discovery Service 47
4.4 Notification System 48
4.5 Orchestrator 49
4.6 Aggregators 50
4.7 Deployment 50

This chapter describes the implementation of FLEXREGMON. All components have been implemented in Java (1.8). The Orchestrator and its replicas make extensive use of a Zookeeper to store their information and use as a session manager. Each instance is linked to a Zookeeper cluster, where they are able to store, and maintain information about the hierarchical topology and status of the connections between nodes.

To create a realistic wide-spread network and to be able to tweak delays on demand we used Kollaps [26] a Decentralized Container-Based Network Emulator. It allows us to run experiments in a single cluster while simulating a wide-spread network where we can tweak several characteristics of the network to test different scenarios.

The Central Orchestrator maintains all the hierarchical and routing information on its Zookeeper cluster. This allows us to use the central not only as a point of communication for updates but also as a backup for the distributed replicas. If any problem arises, and there is a need to repopulate information to a replica, the Central Orchestrator is able to provide that information.

The Central Orchestrator and its replicas are also able to act as a discovery service for other nodes of the system (see section 3.3.5).

4.1 Zookeeper

Zookeeper [27] is a service for coordinating processes of distributed application. It incorporates elements from group messaging, shared registers and distributed lock services in locks in a replicated, centralized service. Its interface enables a high-performance service implementation with wait-free property, per client guarantee of FIFO execution of requests and linearizability for all the requests that change the Zookeeper state.

Zookeeper excels in heavy read scenarios. It can handle tens to hundreds of thousands of transactions per second which allows it to be used extensively by client application.

Zookeeper has two types of entities: (i) *Zookeeper Service*, which is an ensemble of replicated Servers (where one of them is the *Leader*). These servers are allowed to perform *read* operations, and only the *Leader* is allowed to perform write operations; (ii) *Clients*, which can read data from any server instance or propose a write operation. When connected clients create a session that is open as long as the clients are connected to ZooKeeper.

Zookeeper provides a namespace which is much like a standard file system. In Zookeeper's namespace, a name is a sequence of path elements separated by a slash. Every namespace is identified by a path. Each node in a namespace can have data associated with it as well as children nodes. Data nodes are identified with the term *znode*. Zookeeper also has the concept of *ephemeral znodes*. These *znodes* cannot have children nodes and exist as long as the client session that created the *znode* is

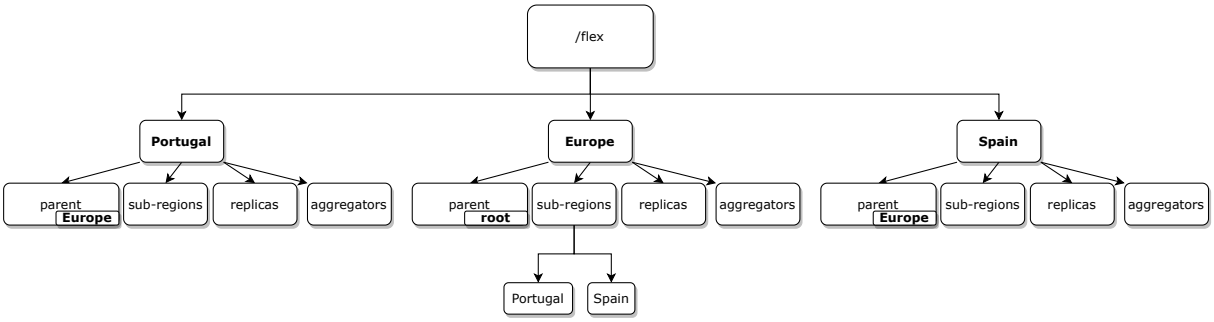


Figure 4.1: Example of FLEXREGMON Zookeeper Server Hierarchical Structure

active. When the sessions end the *znode* is deleted.

Zookeeper characteristics allow us to have a redundant service that allows us to use as a configuration management for the whole system.

4.1.1 Zookeeper Notification System

In order for client to be aware of changes, Zookeeper implements a watcher mechanism where clients subscribe to changes to a specific *znode*. When an event that was subscribed occurs, the Zookeeper Service sends a notification to the subscribers and then those can pull the server to obtain the new data. After a *watch* has been triggered it needs to be set again by the client (this is generally done when requesting the changed data). The process of having to request the data from the server proved to be a problem during our implementation, as we are designing the system to work on the edge (further details in Section 4.4).

4.2 FLEXREGMON Internal Zookeeper Structure

As said before, we associate each orchestrator instance to a zookeeper server. We use the zookeeper to maintain the topological information and connection status with clients for the specific replica.

We take advantage of the Zookeeper Hierarchical Space [27] to store our own hierarchical and routing information. As depicted in Figure 4.1, at the root level of the Zookeeper space we create a branch */flex*, where every child znode of that branch represents a region (e.g. */flex/Portugal*). In each one of those child znodes, we create another four different znodes: (i) */flex/*/parent*, where we keep the value of the parent region; (ii) */flex/*/sub_regions*, where, for each subregion we create a znode with the name of each one of those regions (e.g. *flex/Portugal/sub_regions/lisbon*); and at last (iii) and (iv) which have similar behavior (which is detailed in the following section) are */flex/*/replicas* and */flex/*/aggregators*.

4.3 FLEXREGMON Discovery Service

As described in section 4.1, Zookeeper has the ability to create ephemeral nodes. These znodes remain in the system as long as the client that created those znodes remains connected to the server and renews the lease on the znode (mimicking the behavior of a session). When the session ends the znode is deleted. A limit imposed on the creation of these types of nodes is that they are not allowed to have children znodes.

Due to way these nodes work, one of the documented uses to them is the role of service discovery. If a certain service creates an ephemeral znode with their location (IP address and port) and keeps renewing the lease on that same node, we will be able to find the service on the Zookeeper registries. In this way, we can always have an updated and on-demand list of the current location of machines that are running a said service.

We make use Zookeeper's natural hierarchical structure and ephemeral znodes as the foundation for the implementation for our ow discovery service and *Routing Table* (section 3.4).

We construct the routing table by using the zookeeper hierarchical znodes. As described in the previous section, regions are represented by permanent znodes (with the region name). Within those znodes we create multiple branches that we can use to store the necessary information to maintain the routing table (as can be seen by the *aggregators* and *replicas* branches in Figure 4.1).

These branches are populated by ephemeral znodes, created by the nodes that are running the corresponding service (either as replicas or aggregators). When the connection with the node that created the ephemeral znode is closed, the ephemeral znode is removed and the node becomes unreachable when searching for that service.

The replicas branches are populated when an Orchestrator-Replica connects to the central Orchestrator and they create the ephemeral znode with their location on the corresponding region. The aggregators branch is populated in a different way, depending if we are talking about a replica or the Central Orchestrator. If populating a replica, aggregators nodes connect to those replicas and create the ephemeral znodes to initiate their session. Then the replicas duplicate those znodes to the Central Orchestrator.

As a practical example, take it that we want to store the IP address (let's say 195.63.23.1) for a replica for the Lisbon region. We would store it as a child znode of */flex/lisbon/replicas* and use the location of that znode as the entry, resulting in the following znode */flex/lisbon/replica/195.63.23.1*. This example is depicted in Figure 4.2.

Whenever a node requires to know the location of a service, it simply needs to do a simple query for the children of those znodes (*aggregators* and *replicas*).

This said and following the same example, if any client wishes to know the location of a replica for the Lisbon region, they just have to request to list the children of */flex/lisbon/replicas* which would translate

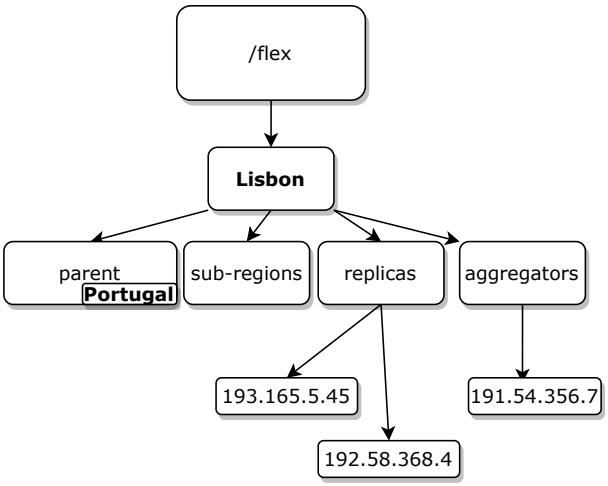


Figure 4.2: Zookeeper Structure example to store Routing Table information

to the return value of [193.165.5.45, 192.58.365.4].

4.4 Notification System

As described in our design chapter (Chapter 3.2), after a topology modification is performed it is necessary to propagate the changes throughout the system. This is important to enable the features that we want our system to have in respect to adaptability to multiple topologies.

We started by using the *Watcher* mechanism already implemented by Zookeeper 4.1.1. This proved to be a problem when sending topology change notifications through long-distance channels that had a higher level of latency. Zookeeper's watch mechanism works by having clients subscribe to changes in a certain node, and notifying subscribers of said data changes and let the client control the operation to perform after notification. This means that to update a single node of a topology change in a region it is required to: (i) report the event to the subscriber (in this case report data change of *x* znode); (ii) the subscriber node will receive the event and decide what to do (in this case request the new data); (iii) send the data from the origin to the subscriber node. In a communication channel where latency is involved, this extra number of messages that is necessary to update the topology results in an excessive amount of overhead due to the time it takes for each message to be transmitted in a long distance channel. We solved this problem by implementing a custom protocol that sends all the necessary information to perform the necessary modification in one single message. When a modification is performed, the Central Orchestrator or replica look at the modified area of the Zookeeper, and send the a single message to the affected nodes with the necessary changes. This allow us to evade the two phase update (notification plus data pull) that is necessary when using the Zookeeper's native watcher mechanism.

This new custom protocol reflects the algorithms and sequence diagrams depicted on section 3.5.

This notification system is also used to report node failures. It works in a similar way to the notification of topology changes. When a connection between a node and its orchestrator breaks, the ephemeral znode that represents the node session is removed. This triggers the notification to the affected nodes (in a single message like a topology change) and perform the necessary operations.

4.5 Orchestrator

The Orchestrator works as the central registry for the hierarchical structure. Each instance is associated with a Zookeeper Cluster, where they store all the necessary data to keep the topology and other necessary information (following the scheme we described in the previous section).

With the central Orchestrator initiated, the distributed replicas can connect to the Central Node, get hierarchical information, and create a ephemeral znode with the replica IP to receive updated (under the */replicas* branch e.g. */Portugal/replicas/IP_ADDRESS*). Just like it explained in sub-sections 3.3.5 and 4.4, these ephemeral znodes allow us to monitor existing services, as the corresponding znodes are removed when the connection are closed. If a connection is closed in a controlled way, either by the orchestrator or by the node itself, the orchestratror deletes the ephemeral znode immediately, effectively removing it from the hierarchical and routing table. In the case of the session being terminated in an uncontrolled way, like network partition or node crash, the Zookeeper instance has a timeout for every session. When a service fails to renew the lease for the ephemeral znode, the Zookeeper server automatically removes the related znode.

4.5.1 Orchestrator-Replicas

In order for individuals Orchestrator-Replicas to take the role of localized topology information maintainers and serve aggregators, we associate each replica to the Central Orchestrator.

Each replica requests the necessary information for its region to the Central-Orchestrator and populates its registries. In order to receive topology updates, the replicas also create an *ephemeral znode* in the central node with their IP for each region they represent. This node is created under the branch: */flex/*region*/replica*. This allows the central node to known which replicas to update when any of these any modification occurs.

After configuration, *aggregators* connect to *replicas* in a similar way that *Orchestrator-Replicas* connect to the central node. They create an *ephemeral znode*, under */flex/*region*/aggregator*, which allows to receive topology information, routing information and updates.

4.6 Aggregators

Aggregators nodes implementation can be divided into two parts: the first part consists of a client that connects to the Orchestrator-Replicas. This connection is what enables aggregators to register themselves for a specific region and create their sessions in replicas. The process is similar to a replica registration on the Central Orchestrator, aggregators connect to a replica and add an ephemeral znode to the region's aggregators branch (e.g. *flex/*region*/aggregators/*AggregatorIpAddress**).

The second part consists on the metric server. As stated in our design chapter (Chapter 3), we use a push approach to submit data-sets to the system. The initiative to submit data comes from the data producer. Aggregator nodes open a server on a well-known port (the default being 1099), where producers or other aggregators can connect to submit data sets.

To submit data to the metric server nodes are required to tag the data-sets with an identifier that contextualizes the data and allows the metric server to sort it. Aggregators store the data sets along with their tags and origin region. Periodically, they perform a set of predefined operations that summarize the values with a specific tag for a region into a single data value. After summarizing the data sets, aggregators can relay the new value to the up in the hierarchical tree to another aggregator. These processes repeats until the data reaches the root of the hierarchical tree.

As a practical example, suppose that the system is monitoring the temperature of Lisbon using thousands of registered sensors. Each sensor will periodically send the measurements along with the tag *"temperature"*, generating the data-set pair: (*temperature*, *temperatureValue*). Eventually, the aggregator will summarize all existing *temperature* sets into an average value (*temperature*, *avgTemperatureValue*) and send it to the Portugal's aggregator.

4.7 Deployment

In respect to the deployment of the system we took extra care with two aspects. The first one is that to be able to run a monitoring system on top of an Edge architecture is highly recommended the ability to deploy software in a heterogeneous environment with different types of machines and architectures and the software should be able to run on different these architecture with various levels of resources availability without high levels of configuration and compatibility problems. And the second problem is that evaluating a large-scale distributed system is a hard, slow, and expensive task. This comes from the large number of components that are involved: system dependencies, libraries, environment heterogeneity, network variability, and difficulty in controlling the network and its conditions to test specific cases.

We solved the first problem by making use of Docker [28], a containerization technology that allows us to run our software in multiple types of environment, as long as they are able to run the docker engine

and support virtualization technology. The second problem is solved by deploying FLEXREGMON on top of Kollaps [26], a decentralized container based network emulator. This emulator allows us to have full control over deployment environment. Both approaches will be described in detail in the following subsections.

4.7.1 Docker

Docker [28] is a technology for app containerization. A container, similar to a lightweight virtual machine (VM), works as a standard unit of software that packages up the code and all its dependencies so that the application can run consistently on different computing environments, making the application portable. It attains those properties by isolating the software from its environment and dependencies.

Containers and VM share similarities to VMs as both are virtualization tools. While VMs, make use of a hypervisor that creates slices of hardware to serve each one. Containers, make available protected portions of the operating system. This results on two containers running on the same operating system now knowing that they are sharing resources because each one has its own abstracted networking layer, processes and other aspects of the OS [28]. Although containers share these OS kernel resources, each process still runs isolated in their own user space and can be launched independently from each other.

Other aspect that limits hypervisor-based virtualization is that it provides access to hardware only, which means you still need to install an operating system for each VM that you want to run. This quickly gobbles resources servers, such as RAM to run the multiple OS, CPU and bandwidth. On the other side of the spectrum, containers rely on the already running OS as their host environment. As said before, containers merely execute in spaces that are isolated from each other, and from the host OS. The gains from this strategy are: resource usage efficiency, if a container is not executing anything, it is not using up resources, and containers can call upon their host OS to satisfy some of their dependencies; and container are cheap and fast to create and destroy, as there is no need to boot and shut down a whole OS. A container just has to terminate the processes running in its isolated space. This means that starting and stopping a container is more akin to starting and quitting an application, and equally as fast.

These characteristics give us benefits that match the requirements to running a system in the Edge (as seen in 2.6.1), like being platform agnostic, the ease of deployment (only requirements is running docker-engine) and the fact this technology is much lighter to run in terms of resources usage. For these reasons, we decided to compile our code into a docker image that allow us to instantiate our system.

Our docker implementation consists on a simple single Docker file (see Figure 4.3) setup to run Java application. We load FLEXREGMON code to the image along with its dependencies (packaged with the .jar to facilitate). These dockers can spawn any type of FLEXREGMON node by providing different inputs to the containers on boot.

```
FROM java:8-jdk-alpine
MAINTAINER  Tiago Goncalves tiago.miguel.c.g@tecnico.ulisboa.pt

COPY Aggregator-1.0.jar Control-1.0.jar Replica-1.0.jar
Orchestrator-1.0.jar Producer-1.0.jar /
```

Figure 4.3: Docker file declaration

4.7.2 Kollaps

As said before, evaluating a large-scale distributed system is hard, slow, and expensive. This comes from the large number of components that are involved: system dependencies and libraries; heterogeneity of the target environment, network variability and dynamics, wanting to control those same attributes in an easy way. We deployed FLEXREGMON on top of Kollaps. Kollaps is a decentralized container based network emulator for large-scale applications. Kollaps effectively allows us to dynamically change link-level emulation properties like, bandwidth, delay, packet loss, and even the layout of the topology.

Kollaps takes the inputted topology and collapses it into a configuration file that setups end-to-end connections between nodes. This allows Kollaps to scale really well with the number of nodes. The program consists on multiple modules, that convert a topology design into a deployment configuration and feed it into Docker Swarm (a docker cluster), which then launches all the necessary to dockers to run the experiment. Further detail about Kollaps can be seen in detail in [26]. Looking at the tests results in Kollaps' paper [26], emulated experiments have very similar results when compared to the corresponding real life situation. This aspect and the overall the Kollaps design and features makes it a very good tool to test our Edge application in multiple simulated environments where we have fine control over the network conditions.

We define the deployment by using Thunderstorm [29], a high-level language to define the deployment of test environments. We define services, networks bridge nodes and point-to-point links between the nodes, where we can define bandwidth, latency. We can also define dynamic behavior during the experiment. To do this we define, in the topology file, actions (e.g.: join and leave) attached to timestamps.

The topology file consists of a xml file, where we define the existing services (nodes), network topology, with along with the dynamic component of the experiment (entrance and exit of nodes). Following the example on Listing 4.1, we have a small topology with five FLEXREGMON nodes (dashboard and puppetmaster exist for debug purposes).

The file is divided into three parts: the node definition, the network definition and dynamic behavior. To declare nodes we use the *services* tag, where we define the associated name, and *docker images* (section 4.7.1) that will launch with the node. Relatively to network we can define *bridges* that create a

network entity that acts as a network switch. And we can also configure each *link* individually in respect to latency, download and upload speeds. The dynamic behavior is defined in the *dynamic* tag and by using the action parameter to define behavior (e.g. join, crash, leave)

Listing 4.1: Example of FLEXREGMON Kollaps topology file.

```
<?xml version="1.0" encoding="UTF-8" ?>
<experiment boot="kollaps:1.0">
    <services>
        <service name="dashboard" image="kollaps/dashboard:1.0" supervisor="true" port="8088"/>
        <service name="orchestrator" .../>
        <service name="lk_portugal0" .../>
        <service name="lk_portugal1" .../>
        <service name="nodeAggregator0" .../>
        <service name="nodeAggregator1" .../>
        <service name="puppetmaster" .../>
    </services>

    <bridges>
        <bridge name="delayed"/>
        <bridge name="portugal"/>
    </bridges>

    <links>
        <link origin="portugal" dest="puppetmaster" latency="0" download="100Mbps" upload="100Mbps"
            network="\systemname_network"/>
        <link origin="orchestrator" dest="portugal" latency="0" upload="100Mbps" download="100Mbps"
            network="\systemname_network"/>

        <link origin="delayed" dest="portugal" latency="200" upload="100Mbps" download="1000Mbps"
            network="\systemname_network"/>
        <link origin="delayed" dest="puppetmaster" latency="0" download="100Mbps" upload="100Mbps"
            network="\systemname_network"/>

        <link origin="nodeAggregator0" dest="delayed" latency="0" upload="100Mbps"
            download="1000Mbps" network="\systemname_network"/>
        <link origin="nodeAggregator1" dest="delayed" latency="0" upload="100Mbps"
            download="1000Mbps" network="\systemname_network"/>
```

```
27
28          <link origin="lk_portugal0" dest="portugal" latency="0" upload="100Mbps" download="100Mbps"
                network="\systemname_network"/>
29          <link origin="lk_portugal1" dest="portugal" latency="0" upload="100Mbps" download="100Mbps"
                network="\systemname_network"/>
30      </links>
31
32      <dynamic>
33          <schedule name="orchestrator" time="0.0" action="join" />
34          <schedule name="lk_portugal0" time="10.0" action="join" />
35          <schedule name="lk_portugal1" time="10.0" action="join" />
36          <schedule name="nodeAggregator0" time="20.0" action="join" />
37          <schedule name="nodeAggregator1" time="20.0" action="join" />
38      </dynamic>
39  </experiment>
```

Summary

In this chapter, it was presented the implementation of FLEXREGMON. We described the systems and technologies that we used to implement each of the FLEXREGMON components. We also detailed the deployment and how we solved the ability to deploy the system in heterogeneous and how we simulated real edge scenarios. The next chapter addresses the evaluation of the implementation and if it brings in comparison to a static topology solution.

5

Evaluation

Contents

5.1 Evaluation Goals 56
5.2 Experimental Setup 56
5.3 FlexRegMon vs. Zookeeper - Propagating Modifications on the Edge 57
5.4 Notification System 61
5.5 FlexRegMon Overhead 64
5.6 Benefits of the Flexible Topology 64
5.7 Discussion 67

This chapter evaluates FLEXREGMON. It starts by describing the goals of evaluation on Section 5.1; Followed by a description of the experimental setup on Section 5.2 with all the scenarios and results on Section 5.3, 5.4, 5.5, and 5.6 ; Lastly the chapter finishes with a brief discussion about the collected results on Section 5.7.

5.1 Evaluation Goals

In the evaluation, we want to address the following problems:

1. How does FLEXREGMON distributed topology management compare to the centralized solution of Zookeeper [27] in edge scenarios?
2. Does the FLEXREGMON distributed management help with operations inside regions?
3. Can FLEXREGMON provide advantages to systems that unexpected loads on specific region with its characteristic of topology flexibility?

For this purpose it was run a performance evaluation of FLEXREGMON against a centralized Zookeeper

5.2 Experimental Setup

All experiments were run on cluster composed by two machines: the first one with a 2.20GHz Intel Xeon Silver 4114 CPU and 128GB of RAM. And the second with a 2.00GHz Intel Xeon Gold 6138 CPU and 64GB of RAM, with all CPU cores locked at running at 50% max load. We used Kollaps [26] to create the virtual networks necessary for the experiments and to launch the nodes instances. The experiment was then deployed on top of a Docker Swarm cluster and inside the virtual network.

Each FLEXREGMON nodes run a custom docker image (described in 4.7.1) that contains our system implementation running on top of Alpine Linux 3.4.6, and Java 1.8.0_111.

Depending on the experiment, we used different topology definitions to match the desired network structure, with different point-to-point network configurations in order to simulate different network structures.

To measure latency and manage all nodes, we also created an extra overseer node, called *Puppet-master*. This node is external to the Kollaps network and is able to communicate with close to 0 ms delay with every other node on the network (which allows us to take the measurements).

To setup the evaluation environments, we have considered the reference Edge/Fog architecture shown in Chapter 2. We envision an architecture where the system can have resources instanced in different geographical regions, that are connected by long distance channels. These connections are simulated by adding higher latency between areas of the network.

5.3 FLEXREGMON vs. Zookeeper - Propagating Modifications on the Edge

In this section, we try to answer the first question and compare, on an Edge environment, our solution to distribute the management of region to orchestrator replicas and compare it to the centralized solution of a Zookeeper cluster. Upon a topology modification, FLEXREGMON relies on the central orchestrator to send the update to a replica that serves the region, and the replica will send the update to the remainder of the nodes. In the case of Zookeeper, the central server is responsible for sending the updates to every node on the system.

We expect that, in scenarios where the central node is in distant geographical areas, as those that can be found in an Edge environment and where the connections have higher latency between nodes, the Zookeeper solution will obtain worse results than FLEXREGMON.

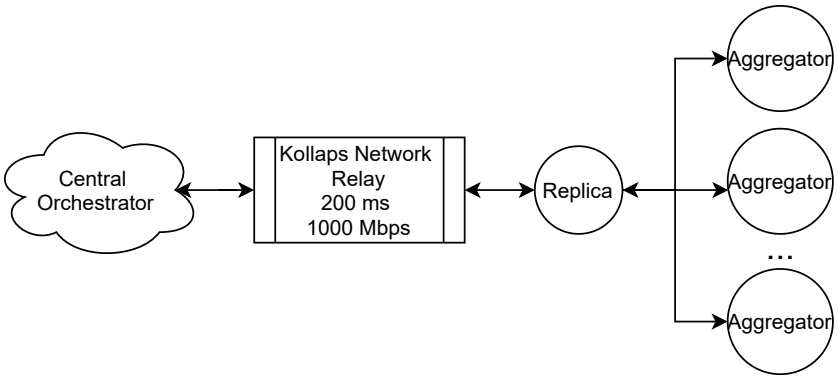
We created the network scenarios, depicted in Figures 5.1(a) and 5.1(b), with two geographical areas, split by a Kollaps relay with $1000Mbps$ of bandwidth and $200ms$ of latency between them. FLEXREGMON deployment keeps the central orchestrator in one of the geographical areas, and all the other nodes required to run the experiment in the second area. Zookeeper keeps the central server on one of the areas and the remainder nodes on the other.

We then measured the time it takes for a modification on the data aggregation topology of a single region to propagate to every affected aggregator. With this, in Figure 5.1(c) we have the x axis where we vary the number of aggregators present in the region and in the y axis the time it takes for all the aggregators to acknowledge the new topology.

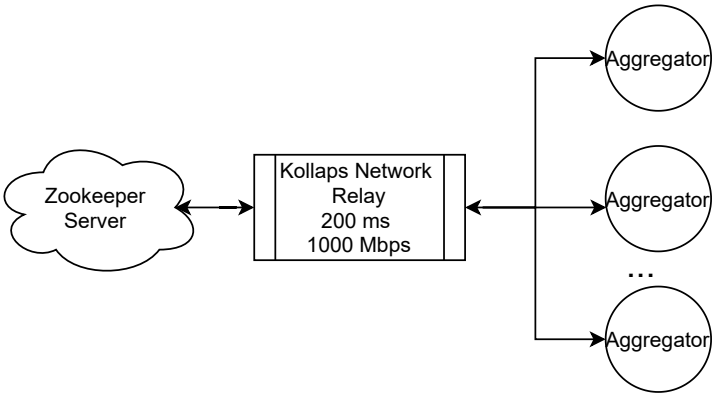
We can observe up to $3.1\times$ lower latency times to update all aggregators in FLEXREGMON when compared to Zookeeper. FLEXREGMON makes use of the geographical proximity of the replicas to leverage lower latency connections, contrasting the Zookeeper solution of updating each aggregator.

Another aspect that might affect these results is the use Zookeeper notification mechanism. The Zookeeper native *Watcher* solution (see Section 4.1.1) in this specific scenario is not optimal. It requires to send n number of notification messages (n being the number of nodes that were affected by the update), plus n data request messages, and n more data answer messages through a channel with a delay of $200ms$. FLEXREGMON design is streamlined to send a single a message with the necessary information to each node. In this case, FLEXREGMON sends a single message to the orchestrator replica, and then that replica relays the message to all the other nodes with a much lower latency. Both these characteristics, the distributed management (Orchestrator-Replicas) and the notification system, allow FLEXREGMON to reduce the time to update the topology by an average of $2.8\times$ in comparison to the classic Zookeeper system in an Edge scenario.

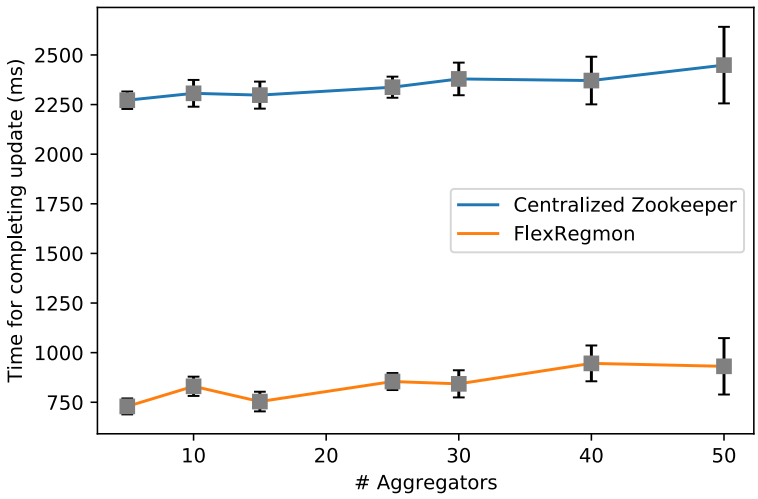
We analyze in a following section the impact that the notification mechanisms has in the performance



(a) Topology for FLEXREGMON with scaling number of aggregators



(b) Topology for Zookeeper with scaling number of aggregators



(c) Time to add new hierarchical level varying number of aggregators

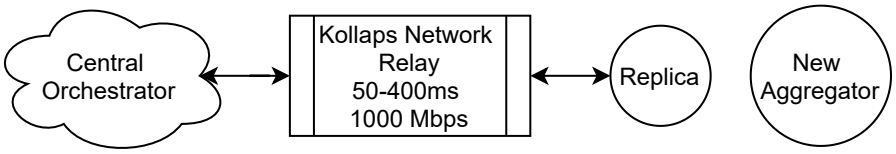
Figure 5.1: FLEXREGMON vs Zookeeper: propagating topology changes on the Edge

of the system.

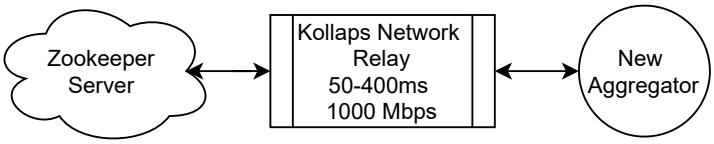
5.3.1 Adding and Removing Aggregators

Because FLEXREGMON attributes the management of regions to orchestrator replicas instead of managing it at the central node. Aggregators and data producers only are required to talk with the replicas to interact with the system. This allows to have lower latency operations, like joining and exiting the system or receive topology updates. It would also be interesting to have a distributed control engine that could give regions the ability to self-regulate, scale as necessary, and even perform topology modifications inside their own scope (e.g. add a hierarchical level to create one more aggregation point).

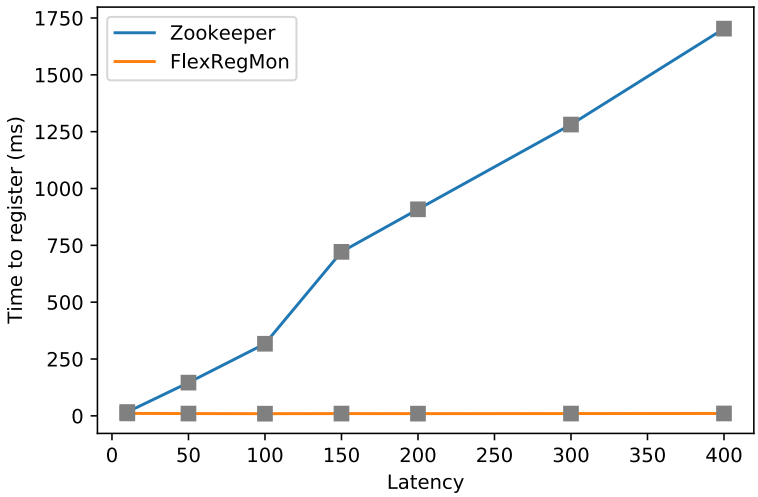
With this we want to measure the effectiveness of our solution to decrease the latency of these types of operations in edge scenarios, where the central node has a high latency connection to the nodes located in the edge. We deploy both FLEXREGMON and Zookeeper in the topology depicted in Figure 5.2(a), where we again have two different geographical areas with a Kollaps relay in the middle where we vary the latency. For FLEXREGMON we keep the Central Orchestrator in one of the areas and on the other we have a single Orchestrator-Replica that is responsible for the unique region that exists in the system. For the Zookeeper solution we keep the server in one of the areas and the remainder nodes on the other. Then we deploy an aggregator that will register on the system and we measure the time it takes for it to integrate the system. We expect FLEXREGMON to have better results. Orchestrator-Replicas reduce drastically the latency from the new node to a management node which leads to an increase of the performance of the system, and reduction of the overhead that would occur by using a centralized management system in an environment with high latency connection such as those found in the edge. In Figure 5.2, it is shown the average time to add an aggregator after instantiating it. In the x axis we have the latency between the geographical areas that we vary from $10ms$ to $400ms$, and in the y axis we have the time it takes for the aggregator to be available to producers. By analyzing the results we can see that the FLEXREGMON maintains a constant value as the system uses the localized replicas to manage the regions, which results in an average of $10ms$ to fully execute the operation and for the aggregator to be available. The Zookeeper performance keeps deteriorating with the increase of the latency between the geographical areas. The distribution of the regions' management makes these types of operations independent from the quality of the connection to the central node. The results show that the FLEXREGMON architecture helps in dealing with local management operations as the necessary nodes are closer, reducing the impact of high latency connections. It also reinforces the idea of possible benefits that having a distributed control mechanism that enable more decision powers in each replica over their region could be beneficial to the system, making the replicas even more independent from the central node.



(a) Topology for FLEXREGMON to measure time top add aggregator

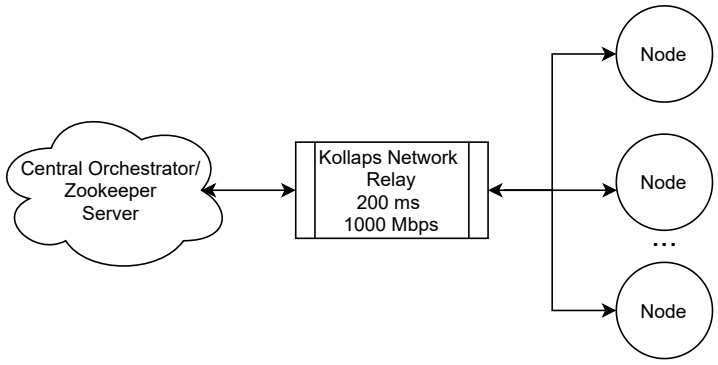


(b) Topology for Zookeeper to measure time to add aggregator

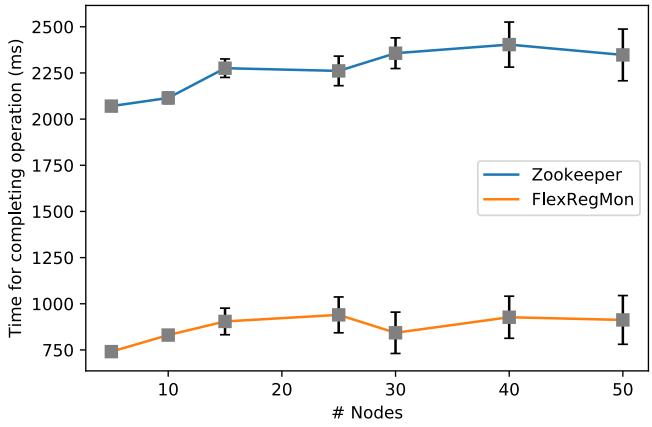


(c) Time to add aggregator depending on the latency (ms)

Figure 5.2: Adding and removing aggregators



(a) Topology for FLEXREGMON to measure notification system performance

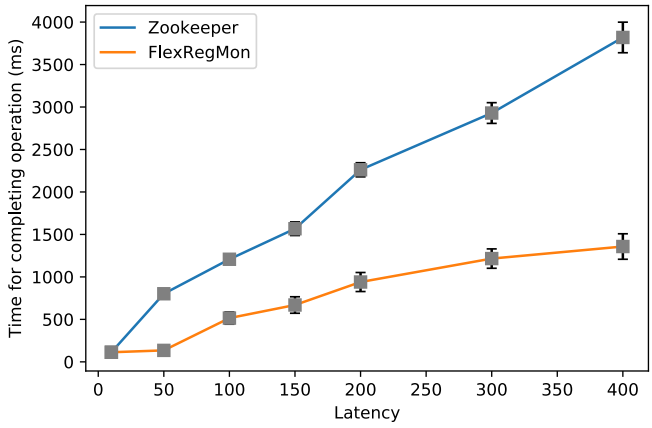


(b) Time to propagate update to all nodes while varying the number of nodes

Figure 5.3: Notification System: performance while varying number of nodes

5.4 Notification System

Both FLEXREGMON and the centralized Zookeeper solution rely on a notification system to propagate topology changes. This is a necessary step to update the topology on all necessary nodes, and needs to take into account the condition of the connections between nodes to not hinder the update process. Zookeeper uses the *Watcher* [27] mechanism. Each node subscribes to changes to a specific znode and are later notified of said changes, having to request the changed data and resubscribe to keep receiving updates. FLEXREGMON uses its own routing table to send the updates automatically upon a modification. In a constrained network with higher latency, as those found between two distant geographical zones, FLEXREGMON saves a lot of overhead due to sending a single message. Figure 5.4 shows the results of our experiments to evaluate both notification systems. To evaluate the performance bene-



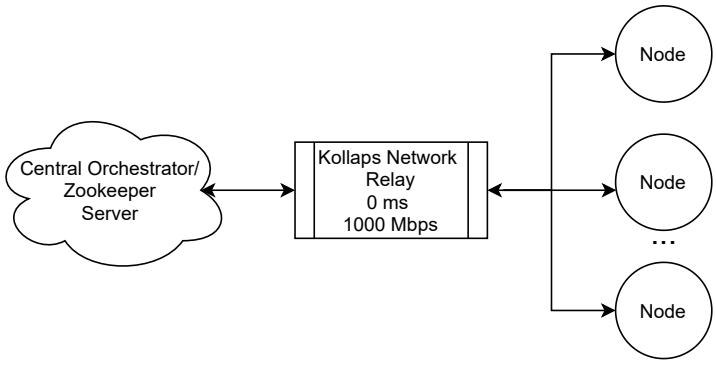
(a) Time to propagate update to all nodes while varying latency

Figure 5.4: Notification System: performance while varying the latency

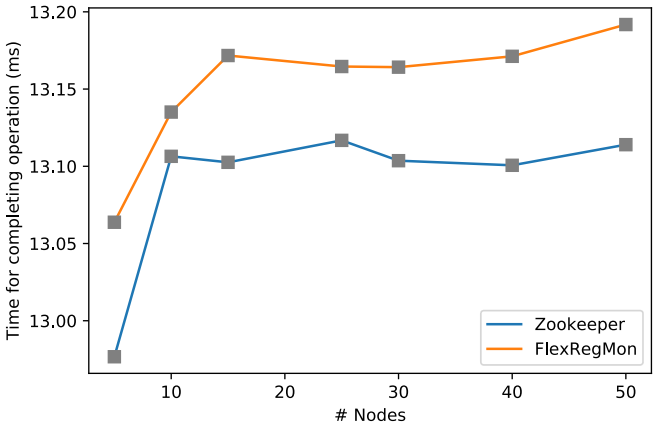
fits of our notification solution, we integrated the Zookeeper notification mechanism into FLEXREGMON and performed two experiments, one measuring the time for both notification mechanisms to update the system while varying the number of Orchestrator-Replicas and maintaining the latency between regions. For the second experiment we took the same measurements but varied the latency between the geographical areas with a fixed number of replicas. The topology for the experiments can be seen on Figure 5.1(b), where we keep the Central-Orchestrator in one of the geographical areas and all the other replicas in the other. To alter the latency between runs we change the latency on the Kollaps relay.

In the Figure 5.3(b), the x axis varies the number of replicas that need to be updated and the y axis shows the time it takes for all nodes to acknowledge the new topology. We kept the latency between the two geographical regions at $200ms$ and vary the number of replicas that required to be updates from 1 to 50. As expected, we can see that both FLEXREGMON and Zookeeper maintain performance when varying the number of replicas that required concurrent updates, with both systems keeping stable results throughout the experiment. This provides assurance that both notification mechanisms are adequate to send concurrent updates to multiple nodes, with no detriment in the performance when increasing the number of notifications.

In Figure 5.4(a), for the x axis we vary the latency between two geographical areas and again in the y axis we have the time in ms for all replicas to receive the update. Here we fixed the number of replicas to 25 and varied the latency between geographical regions, going from $10ms$ to the $400ms$. Looking at the results, it is possible to see that both FLEXREGMON and Zookeeper notification systems performance get worse results as we increase the latency between the geographical areas, with Zookeeper having a bigger deterioration on performance. This happens because Zookeeper solution requires more



(a) Centralized Topology



(b) Time to update topology on all nodes

Figure 5.5: FLEXREGMON vs Zookeeper in centralized environment

messages to perform the data update on the remote nodes, meaning that as we increase the connection latency between geographical regions, the more it will affect the update process. This translates to FLEXREGMON having up to $2.7\times$ better results than the Zookeeper in similar conditions.

The larger performance degradation that occurs in Zookeeper comes from the fact that when using the *Watcher* mechanism, nodes subscribe to be notified of changes for a specific znode (which represents a region). When a modification is performed all the subscribers receive a notification. Then the node need to request that data and resubscribe, which leads to a total of three messages (notification, data read, and read answer) to perform a single update. In an environment where network delay is existent, it leads to a bigger overhead in the communication between the nodes.

5.5 FLEXREGMON Overhead

FLEXREGMON introduces some processing steps to each update, where it is necessary to consult the routing table to assert which nodes need to be updated. Due to the way that FLEXREGMON propagates the updates, it might be necessary to have multiple phases of this process. For example, when making a topology modification that affects aggregators. First the central orchestrator sends the update to the replicas, and only then after processing the update and looking at their routing tables the replicas send the updates to the aggregators. In the case of using the centralized Zookeeper, the server will deliver directly the update directly to each node. By deploying both systems in a centralized scenario we can measure how having to re-process the updates on FLEXREGMON might affect the system. The connection between nodes is no longer a problem and both systems can function just based on execution time.

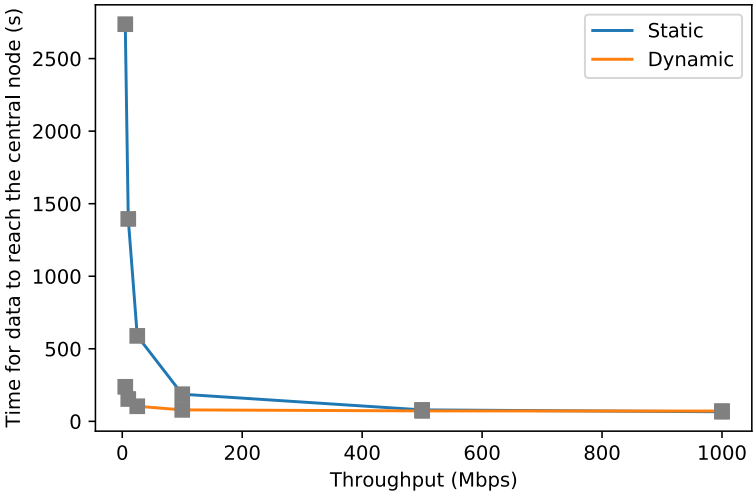
We create a simple topology, as depicted in Figure 5.5(a), with a single geographical area where we allocate every necessary node to run the experiment. Every connection between nodes has no latency added. By varying the number of nodes that need to be updated we can show how it affects each update strategy and if the processing required by our solution affects the performance at all. We expect FLEXREGMON extra steps to have a small impact (or even none) and to achieve similar results to the Zookeeper.

In Figure 5.5(b), we have the x axis where we vary the number of nodes that are updated and in the y the time it takes for every node to acknowledge the update. By looking at the results we can see that the performance is relatively the same between both solutions. They both keep a steady time to update all nodes and no solution is better than the other in this centralized scenarios in terms of update latency. This shows that even with FLEXREGMON added complexity caused by having a distributed region management, it does not cause an excessive overhead over updates in a centralized environment while providing benefits in edge scenarios.

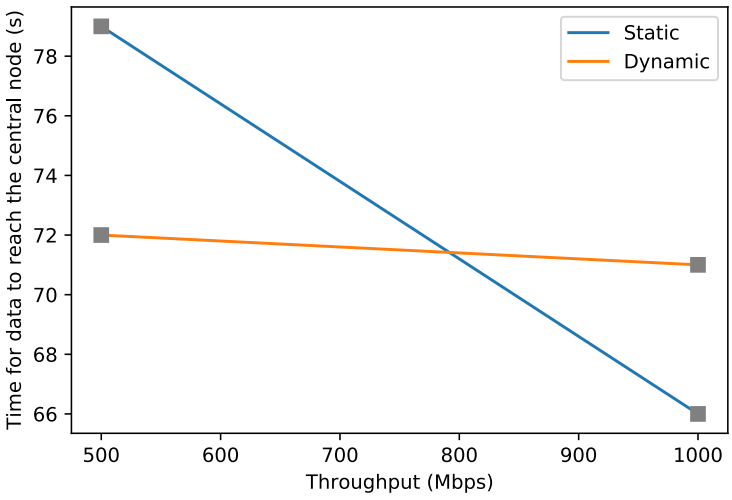
5.6 Benefits of the Flexible Topology

Due to the nature of the edge, it is expected to have constant changes in the load of nodes and regions. With clients having the capability of being mobile, the possibility of big migrations might lead to the degradation of performance in a specific region. To combat this we can re-arrange the topology of the system in order to distribute load and use of data aggregation to reduce the saturation of connections on the system. In the previous sections it was shown that FLEXREGMON is able to apply topology updates in a effective way, even in edge scenarios. Now we wish to prove that the impact of performing those updates does not overbear the benefits of the re-configuration the data aggregation tree.

To measure this we want to run an experiment that compares the performance between a static



(a)



(b)

Figure 5.6: Benefits of flexible topology: Dynamic vs Static

topology vs a dynamic approach where we re-configure the regions topology to adapt to the load. In this experiment both solutions start with two geographical areas separated by a Kollaps relay with no added latency but with variable throughput (from $5Mbps$ to $1000Mbps$). All the other connections have no latency and work with a throughput of $1000Mbps$. The regions divide into two hierarchical levels. With every level being able of resuming the data to a quarter of what it receives. During the experiment we connect a fixed number of clients to the system that generate a total $25GB$ of data-sets and submit them to the system. The system will progressively send the generated data up the hierarchical tree through the aggregators until it reaches the root of the hierarchical tree. For this experiment we set each aggregator to be capable of resuming the received data to a quarter.

We deploy two versions of FLEXREGMON: the first one maintains the topology static and does not create or remove regions or hierarchical levels. The second version applies a topology modification that adds extra two hierarchical levels to increase the number of times that data aggregation that is performed.

In Figure 5.6(a), the x axis varies the throughput of the connections between the geographical areas, and the y axis shows the time it takes for the inserted data (or a resume of it that represents it) to reach the central node. By looking at the results we can see that when we have connections with low throughput and saturate them re-configuring the topology is clearly beneficial to the performance of the system, with much lower times than the static topology due to the reduction of the transmitted data that comes from having the two extra hierarchical levels.

For situations with higher throughput, we need to take into account the objective of a re-configuration. Depending on the objective it might still be effective (e.g. reduce the amount of data transmitted) even in situations of high throughput where we worse results if we perform the re-configurations.

Because the re-configuration has a time cost and introduces extra steps between the central node and the root (due to the increase of hierarchical levels), there are situations where re-configuring the topology leads to worse times. By looking at Figure 5.6(b) (which is just a window adjustment of the first graphic), we can see that the static topology has better results when the connection has a throughput of $1000Mbps$. We can also see the intersection point, at around $790Mbps$ where, for this specific scenario, it starts to be worth to use the dynamic approach instead of the static solution. The smaller the throughput of a connection the worse the system will behave when transmitting the same amount of data. Thus the re-configuration will allow to use a more aggressive aggregation of the data, and reduce the size of the transmission improving the performance of the system.

5.7 Discussion

FLEXREGMON was conceived to be able easily re-configure a data aggregation topology of a system data aggregation topology depending on the conditions of the system. Due to the fact that FLEXREGMON is a system to be deployed on the edge also puts us in face with some challenges, like how to deal with constant entrance and exit of nodes, problem which FLEXREGMON reduces by delegating the role of region management to the orchestrator replicas. Another important aspect to take into account in the evaluation process is to analyze if the benefits of re-arranging the topology are not overshadowed by the an overhead introduced by the system.

Our experimental evaluation shows that FLEXREGMON is capable of modifying the topology of a edge deployed system up to 3.1x faster when compared to a centralized topology management. FLEXREGMON also creates a massive increase in the performance of management operations like joining or exiting a region, with results close to $0ms$ of latency, due to the distributed role of region management that is present in orchestrator replicas. The notification system used by FLEXREGMON also reduces drastically the cost of performing updates on the edge, as the reduce communications cost helps the propagation of updates. And lastly we also validated the utility and viability of dynamically altering the data aggregation topology to increase the performance of the system.

Summary

In this chapter, it was presented the evaluation of FLEXREGMON. The results show that FLEXREGMON is able to reduce the time it takes to perform a modification on the data aggregation topology and propagate said change, while not degrading the performance of the system. It was also noted the benefits of having the management component of the system (orchestrator replicas) distributed as it allows for each region to have lower latency management operations.

6

Conclusions and Future Work

Given that the Edge computing paradigm is spread in a much larger area when compared to the classic Cloud computing paradigm, it is expected to have different types of loads that depending on the situation would benefit from different topologies. In this thesis it was presented FLEXREGMON, a system that by the means of a distributed management orchestrator allows to efficiently modify the topology of regions for data aggregation, to allow low latency operations and increase the performance when propagating information. It was shown that FLEXREGMON allows the system to reconfigure itself and re-arrange its data aggregation topology to adapt to different conditions, leading to performance gains in multiple scenarios. FLEXREGMON combines the use of a new notification system along with the distributed orchestrator replicas to reduce the amount of communication necessary to central node to perform said re-configurations.

As future work, it would be interesting to develop the decision motor to perform dynamic modifications on the topology of the system, and evaluate the benefits of distributing that component. If each region could be self-regulated and modify its topology independently it could enhance the capabilities of the system.

Bibliography

[1] S. Taherizadeh, A. C. Jones, I. Taylor, Z. Zhao, and V. Stankovski, "Monitoring self-adaptive applications within edge computing frameworks: A state-of-the-art review," *Journal of Systems and Software*, Feb 2018.

[2] V. Prasad, M. Bhavsar, and S. Tanwar, "Influence of monitoring: Fog and edge computing," *Scalable Computing*, May 2019.

[3] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, Oct 2016.

[4] A. Greenberg, J. Hamilton, D. Maltz, and P. Patel, "The cost of a cloud: Research problems in data center networks," *Computer Communication Review*, Jan 2009.

[5] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *IEEE Pervasive Computing*, Oct 2009.

[6] F. Bonomi and R. Milito, "Fog computing and its role in the internet of things," *Proceedings of the MCC workshop on Mobile Cloud Computing*, Aug 2012.

[7] G. Aceto, A. Botta, W. De Donato, and A. Pescapè, "Cloud monitoring: A survey," *Computer Networks*, 2013.

[8] M. Abderrahim, M. Ouzzif, K. Guillouard, J. Francois, and A. Lebre, "A holistic monitoring service for fog/edge infrastructures: A foresight study," in *2017 IEEE 5th International Conference on Future Internet of Things and Cloud (FiCloud)*, Aug 2017.

[9] J.-P. Martin-Flatin, "Push vs. pull in web-based network management," in *Integrated Network Management VI. Distributed Management for the Networked Millennium. Proceedings of the Sixth IFIP/IEEE International Symposium on Integrated Network Management.*, 1999.

[10] K. Fatema, V. Emeakaroha, P. Healy, J. Morrison, and T. Lynn, "A survey of cloud monitoring tools: Taxonomy, capabilities and objectives," *Journal of Parallel and Distributed Computing*, Oct 2014.

[11] J. Zhu, D. S. Chan, M. S. Prabhu, P. Natarajan, H. Hu, and F. Bonomi, "Improving web sites performance using edge servers in fog computing architecture," in *2013 IEEE Seventh International Symposium on Service-Oriented System Engineering*, March 2013.

[12] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: Elastic execution between mobile device and cloud," Jan 2011.

[13] A. Rudenko, P. Reiher, G. Popek, and G. Kuenning, "Saving portable computer battery power through remote process execution," *Mobile Computing and Communications Review*, March 1998.

[14] R. Van Renesse, K. Birman, and W. Vogels, "Astrolabe," *ACM Transactions on Computer Systems*, May 2003.

[15] A. Demers, D. Greene, C. Houser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, "Epidemic algorithms for replicated database maintenance," *ACM SIGOPS Operating Systems Review*, 1988.

[16] C. Hare, "Simple network management protocol (snmp)." 2011.

[17] M. Pérez and A. Sanchez, "Fmone: A flexible monitoring solution at the edge," *Wireless Communications and Mobile Computing*, Nov 2018.

[18] P. Yalagandula and M. Dahlin, "A scalable distributed information management system," *SIGCOMM Comput. Commun. Rev.*, Aug. 2004.

[19] M. L. Massie, B. N. Chun, and D. E. Culler, "The ganglia distributed monitoring system: design, implementation, and experience," *Parallel Computing*, 2004.

[20] H. B. Newman, I. Legrand, P. Galvez, R. Voicu, and C. Cirstoiu, "Monalisa : A distributed monitoring service architecture," *CoRR*, 2003.

[21] Openstack Project, "Monasca wiki," https://wiki.openstack.org/wiki/Monasca, accessed: 2019-12.

[22] J. Kreps, N. Narkhede, J. Rao *et al.*, "Kafka: A distributed messaging system for log processing," in *Proceedings of the NetDB*, 2011.

[23] A. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, 2001.

[24] B. P. Rimal, E. Choi, and I. Lumb, "A taxonomy and survey of cloud computing systems," in *2009 Fifth International Joint Conference on INC, IMS and IDC*. Ieee, 2009, pp. 44–51.

[25] L. Krishnamachari, D. Estrin, and S. Wicker, “The impact of data aggregation in wireless sensor networks,” in *Proceedings 22nd international conference on distributed computing systems workshops*. IEEE, 2002, pp. 575–578.

[26] P. Gouveia, J. a. Neves, C. Segarra, L. Liechti, S. Issa, V. Schiavoni, and M. Matos, “Kollaps: Decentralized and dynamic topology emulation,” in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys ’20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3342195.3387540

[27] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, “Zookeeper: Wait-free coordination for internet-scale systems,” in *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIXATC’10. USA: USENIX Association, 2010, p. 11.

[28] D. Merkel, “Docker: Lightweight linux containers for consistent development and deployment,” *Linux J.*, vol. 2014, no. 239, Mar. 2014.

[29] M. Matos, “Kollaps/thunderstorm: Reproducible evaluation of distributed systems,” in *Distributed Applications and Interoperable Systems*, A. Remke and V. Schiavoni, Eds. Cham: Springer International Publishing, 2020, pp. 121–128.