Dynamic Reconfiguration of the Data Aggregation Topology at the Edge

(extended abstract of the MSc dissertation)

Tiago Gonçalves
Departamento de Engenharia Informática
Instituto Superior Técnico
Advisor: Professor Luís Rodrigues

***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 re-configuration of the data aggregation topology, by using a distributed component that manages the hierarchical topology of the system.**

***Keywords*-edge computing, monitoring, flexible topology, reconfiguration, data aggregation.**

I. Introduction

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 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; (ii) A fog/edge layer, that has the ability to pre-process raw data before it is shipped to the cloud. 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.

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 reconfigure the applications based on the observed usage patterns.

II. Related Work

In order to efficiently manage systems with increasing complexity that can be found in the Cloud, Fog and Edge, 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. In order to operate in an infrastructure of this scale and complexity, a monitoring system needs to fulfill different function [7]:

Observation The observation function is the one responsible for tracking remote resources and gathering data for processing. This function can be implemented in various ways. The work of performing the collection of data can come the monitoring system, or be distributed throughout the remote resources (Push vs Pull [8]).

Processing In order to be able to use the data generated by the monitored 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. The analysis can be used to perform optimizations and reconfigurations on the system to increase performance.

Exposition This function is responsible to export the output of the monitoring system to the users, allowing them to visualize the information captured and processed.

A. Fog/Edge Multi-layer Monitoring Structure

Edge computing follows a different structure from cloud computing in the sense that it is divided by layers. Due

to these differences, a monitoring service for the edge needs to be structured differently from one designed for the cloud. This means following the multi-layer structure defined for the edge infrastructure [3] with distinct characteristics that need to be taken into account. Systems are large and massively distributed, the heterogeneity of the nodes that make the system and the fact that the nodes can easily enter and exit the system as most of them are mobile. These specific characteristics fundamentally change the strategies that can be used when monitoring the system [1], [7]. New requirements are necessary to design a monitoring system dedicated to the edge [7]. A edge monitoring system needs to reduce the amount of network traffic, quickly react to dynamic resource changes, have operating system and hardware independence, it should be able improve the application performance and have location and network context awareness.

B. Existing Monitoring Solutions

We researched and read about existing monitoring solution to get a grasp of the techniques used to gather data, aggregate it. We also wanted to assert if those systems were capable of adapting to the different conditions and changes that we might find in the edge. In specific we read about: Astrolabe [9], FMonE [10], Ganglia [11], MonALISA [12], Monasca [13] and SDIMS [14]. All of those systems 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 the topology without having highly 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 is able provide a good overview of the aggregation process (MonALISA and Monasca). In systems like SDIMS where aggregation regions are closely tied to the data structure 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, that has a design oriented to fog/edge architecture, with the flexibility of attributing different settings to each node, it does not provide a mechanism to easily apply or modify those different rules.

Data aggregation can affect the performance of the system by reducing the amount transmitted data on the system and if the topology is well adapted to the scenario. With a system that is deployed on the edge, the number of devices increases, 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 those loads 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 is able to adapt to different ongoing conditions by modifying its data aggregation topology without overheads that could overshadow the benefits of making said change.

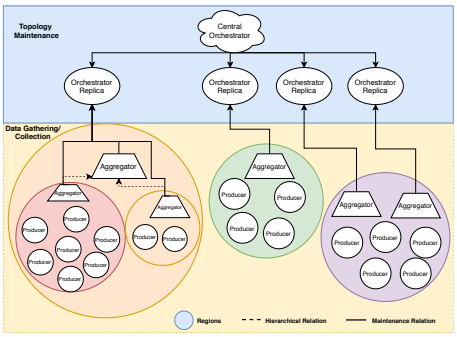


Figure 1: FlexRegMon general structure

III. The FlexRegMon System

FlexRegMon design comes from the necessity of having a system capable of adapting its data aggregation topology to adjust to the constant changes that occur on the Edge. Our design decisions were made in focus with objective of simplifying this process and to increase the performance of management operations in a highly distributed network.

A. System Model

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: (i) *Orchestrator-Replicas* - partial replicas of the Central Orchestrator; (ii) *Aggregators* - nodes that store, process, aggregate, and forward the information generated by the producers; (iii) *Producers* - nodes that produce data and feed it to the their attributed aggregator.

All of these services can be deployed on multiple nodes per region to allow the system to scale. The Central Orchestrator components 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, such as: (i) *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; (ii) *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; (iii) *Edge Devices*, mobile devices with low computation power that can host the *data producers* which generate the data sets that are sent to the system.

B. Data Collection and Processing

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 producers are actively responsible for sending the data to the monitoring system.

Producers connect to the corresponding aggregators and submit their data sets. 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 to resume data. 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).

C. 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 to removing a level.

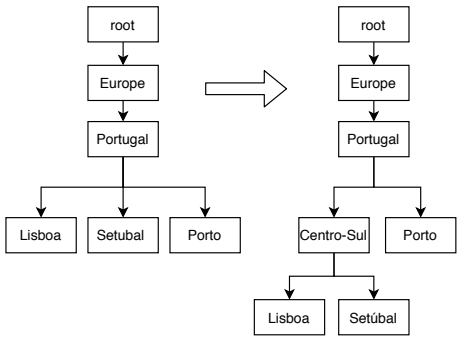


Figure 2: Hierarchical Tree after new level addition

Figure 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 Lisbon/Setúbal regions.

D. 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 these replicas to avoid long-latency links, and still retain the ability to receive topology updates, and query about the topology, and IP of other nodes.

E. 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 that it is associated with) in order to keep receiving topology updates.

F. Producers

Producer role is generally attributed to nodes located on the edge of the network, but the it can be attributed to any other device on the network. This means that any other service can partake as producers on the system. 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 submit the data-sets.

G. Node Bootstrap

The *Control System* 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.

H. 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 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 actions should be pursued.

I. Data Model

1) Hierarchical Structure: The Central Orchestrator and its replicas keep the hierarchical topology information in a *Hierarchical Table* 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. As noted before, the hierarchical topology is maintained by the *Orchestrator* and its replicas. Each one of these nodes keeps an *Hierarchical Table*. 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 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.

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 and parents of regions.

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. To store this information in a *Routing Table* 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 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.

J. Data Sets

Data-sets are sent to aggregators where they are stored, processed and propagated. The data is stored under two constraints, context and region. The first one classifies the value and what it represents (e.g. 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.

IV. Implementation

All components have been implemented in Java (1.8). The Orchestrator and its replicas make extensive use of a Zookeeper instances to store their information and use as a session manager. Each instance is linked to a Zookeeper

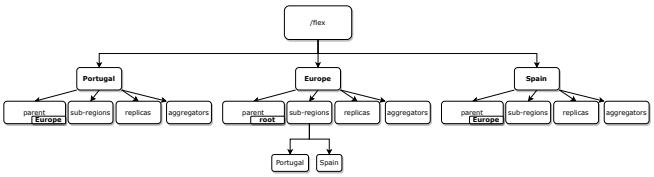


Figure 3: Example of FLEXREGMON Zookeeper Server Hierarchical Structure

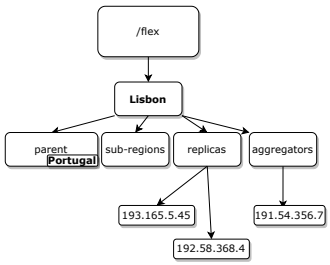


Figure 4: Zookeeper Structure example to store Routing Table information

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 [15], [16] 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.

A. FLEXREGMON *Internal Zookeeper Structure*

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 [17] to store our own hierarchical and routing information. As depicted in Figure 3, 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*.

B. FLEXREGMON *Discovery Service*

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 own discovery service and *Routing Table*.

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 3).

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.

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).

C. Notification System

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 [17]. 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 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.

D. 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.

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*). 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 the Zookeeper instance has a timeout for every session. When a service fails to renew the lease for the ephemeral znode, the Zookeeper service automatically removes the related znode.

E. 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.

F. 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. FLEXREGMON producers uses a push approach to submit data-sets to the system. Aggregator nodes open a server on a well-known port, 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.

G. 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 [18], 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 [15], [16], a decentralized container based network emulator. This emulator allows us to have full control over deployment environment.

V. EVALUATION

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

1) How does FLEXREGMON distributed topology management compare to the centralized solution of Zookeeper [17] 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 a specific region with its characteristic of topology flexibility?

For this purpose it was run a performance evaluation of FLEXREGMON against a centralized Zookeeper adapted to our nodes.

VI. 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 [15] 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.

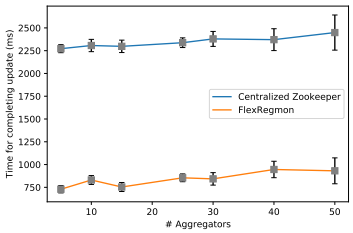
Each FLEXREGMON nodes run a custom docker image containing our system implementation and running on top of Alpine Linux 3.4.6, and Java 1.8.0_111.

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

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

A. 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.



(a)

Figure 5: Time to add hierarchical level

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.

The network scenarios consist of 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 5a 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 of Zookeeper notification mechanism. The notification system 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.

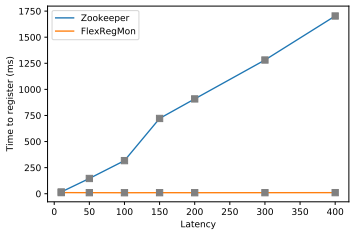
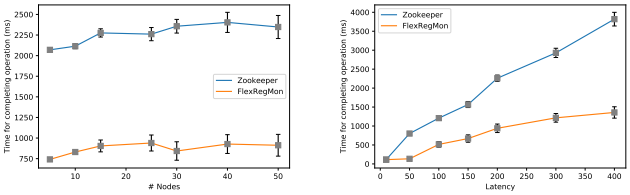


Figure 6: Time to add aggregator



(a) (b)

Figure 7: Time to propagate update (a) varying nodes (b) varying latency

B. Adding and Removing Aggregators

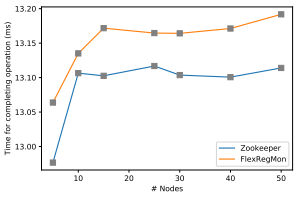
Because FLEXREGMON attributes the management of regions to replicas instead of managing it at the central node aggregators and data producers only are required to talk with the replicas to interact with the system. This allows to have lower latency operations, like joining and exiting the system or receive 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.

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 a topology with 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 6, 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.

C. Notification System

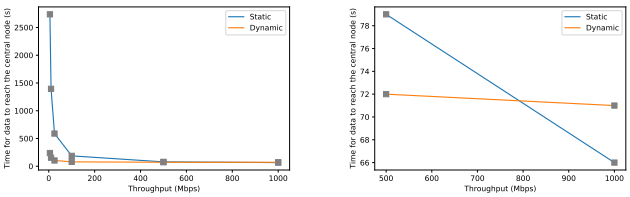
Both FLEXREGMON and the centralized Zookeeper solution rely on a notification system to propagate topology changes. This is a necessary step to update the topology on all necessary nodes, and needs to take into account the condition of the connections between nodes to not hinder the update process. Zookeeper uses the *Watcher* [17] 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 7 shows the results of our experiments to evaluate both notification systems. To evaluate the performance benefits 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 consist on two geographical areas, separated by a Kollaps relay, with the Central Orchestrator in one of the geographical areas and all the other replicas in the other.

In the Figure 7a, 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



(a) Time to update topology on all nodes

Figure 8: FLEXREGMON vs Zookeeper in centralized environment



(a) (b)

Figure 9: Benefits of flexible topology: Dynamic vs Static

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 7b, 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 larger deterioration on performance. This happens because Zookeeper solution requires more 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.

D. 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 and relay 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 used a simple topology with a single geographical area where we allocate every necessary node to run the experiment. 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) on the performance and to achieve similar results to the Zookeeper.

In Figure 8a, 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.

E. *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 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 two extra hierarchical levels to increase the number of times of data aggregation.

In Figure 9a, 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. We see the breaking point for that situation on 9b, where the static topology has better results when the connection has a throughput of $1000Mbps$ and that at around $790Mbps$ 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 aggregate the data more aggressively, and reduce the size of the transmission improving the performance of 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.

VII. 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.

ACKNOWLEDGMENTS

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.

REFERENCES

[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] 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.

[8] 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.

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

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

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

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

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

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

[15] 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

[16] 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.

[17] 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.

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