Performance Monitoring on the Edge

Tiago Miguel Calhanas Gonçalves
tiago.miguel.c.g@tecnico.ulisboa.pt

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, 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. For the successful operation of edge computing it its 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. This report addresses the problem of monitoring edge computing infrastructures. We survey existing monitoring solutions for large scale systems and identify some key limitations for the edge computing scenarios. Then we propose to design and implement a monitoring tool that addresses these limitations.

Table of Contents

1 Introduction 3
2 Goals 4
3 Background 5
3.1 Observation Function - Data Collection 5
3.2 Data Processing Function 6
3.3 Data Exposition Function 7
3.4 Fog/Edge Multi-layer Monitoring Structure 8
3.5 Fog/Edge Monitoring Properties 8
3.6 Monitoring Service Specification on Edge/Fog 9
4 Related Work 11
4.1 Existing Monitoring Solution 12
4.2 Analysis of Monitoring Solution 19
4.3 Shortcomings of Existing Solutions for Edge Computing: 21
5 Architecture 21
5.1 Node and Region Organization 22
5.2 Lookup Service 23
5.3 Edge Layer 24
6 Evaluation 25
6.1 Response Latency 25
6.2 Overhead and Resource usage 26
6.3 Elasticity 26
7 Scheduling of Future Work 26
8 Conclusions 27

1 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 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, providing significant bandwidth savings.

Edge computing typically relies on a multi-layer architecture [1][2] 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 preprocess 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 IoT 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[3], cloudlets[4], or fog computing[5]. To ensure that they can provide services with low latency to devices in the IoT 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.

This report addresses the problem of monitoring edge computing infrastructures. We survey existing monitoring solutions for large scale systems and iden-

tify some key limitations for the edge computing scenarios, as we will discuss ahead. Although many designs have previously addressed the problem of system monitoring in different contexts, including in cloud infrastructures, many of these designs are unable to cope with the heterogeneity, elasticity, and massive distribution of resources that can be found on the edge [6]. In this report we survey existing monitoring systems and their implementations; we discuss the problems that might arise when adapting these systems to an edge environment and, based on this analysis, we devise a new architecture to support multi-layer monitoring.

The rest of the report is organized as follows. Section 2 briefly summarizes the goals and expected results of our work. In Section 3 we provide the necessary background and in Section 4 we discuss related work. Section 5 describes the proposed architecture to be implemented and Section 6 describes how we plan to evaluate our results. Finally, Section 7 presents the schedule of future work and in Section 8 the conclusions that we drawn from our findings.

2 Goals

This work addresses the problem of monitoring an edge multi-layer infrastructure. Due to their cluster/datacenter environments, existing solutions assume a more static structure for aggregating data, where all the processing of a specific region can only be done by an elected node of that region or by nodes of that region. We want to allow data aggregation to be made in a more relaxed way where any server can assume the behavior of data collector/aggregator for any region at any time. More precisely:

> *Goals:* We aim at designing and implementing a monitoring infrastructure for edge computing. The monitoring algorithms should be able to provide information regarding the operation of the system with different levels of detail, while avoiding sending all data regarding the operation of edge nodes to a central node.

Our work is inspired by previous works on scalable monitoring infrastructures, such as Astrolabe and SDIMS (these systems will be discussed in Section 4). We will organize the monitoring infrastructure in a hierarchical composition of logical *regions*. Isolated edge nodes exist at the lower layer of the hierarchy. They collect information regarding their own operation and the devices that they serve. Edge nodes can be grouped into a logical region, and coordinate to collect aggregated information regarding the collective performance of nodes in their regions. In a recursive manner, level-n regions can be grouped to create level-$(n+1)$ regions, that further aggregate data. The top level regions that include all the edge nodes and are responsible for computing the aggregated values. With respect to previous works, we aim at providing higher flexibility to where the monitored resources can send their data and where the data is processed. We want to separate the hierarchical logic from the disposition of the servers on the system. In this way, any node can assume another function role at any

time, independently of its location and characteristics while avoiding recreating or changing data structures with complex procedures.

The project will produce the following expected results.

> *Expected results:* The work will produce i) identification of the requirements for a edge-based monitoring system; ii) specification of a monitoring system for the edge; iii) an implementation of the system; iv) an extensive experimental evaluation using Grid'5000 and YCSB to measure our system.

3 Background

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 function [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.

3.1 Observation Function - Data Collection

The *observation* function is the one 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 unregisters 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.

3.2 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.

3.2.1 Aggregation

3.2.1 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.

3.2.2 Transformation Into Events Events are historical records of something that has happened in the monitored system. They can be created when a certain state of the system is reached or when certain actions were performed. With events, we can provide output with behavioral scope instead of relying only on pure data.

3.3 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 either be used to notify an administrator of something going bad in the system or 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 the 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, should be able change the granularity and period of the data being shown.

3.4 Fog/Edge Multi-layer Monitoring Structure

Edge computing follows a different structure from cloud computing in the sense that it is divided by layers [11]. 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 [1,8,11], follows a multi-layer structure that is similar to the edge infrastructure, composed by the following layers:

– **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. Datacenters are the locations where we can perform big data analysis of the whole infrastructure.
– **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 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. It is the source of data for the upper layers.

3.5 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, such as copper, fiber, wireless technologies, each one with its reliability and speed. Both factors, distance and nature of the link, affect latency and bandwidth among the resources, a fact that should be taken into account when designing the 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.

3.6 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]:

3.6.1 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 events 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. It should be able to adapt at the hardware level and at the virtualization level.

- **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.
- **Able to reach all devices in spite of filters and firewalls:** Certain types of network packets are filtered in private administrative domains due to security concerns. The monitoring system should be able to adapt its communications protocols in order to be able to contact nodes that are located in these contained networks.
- **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 data captured and extra attributes to categorize information which will allow to make localized adjustments to the system [12] or end-users' preferences.
- **Minimize energy consumption:** Some edge devices have limited resources and should be able to offload [13,14] 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.

3.6.2 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 if it is virtualized or not.
- **Geo-aware:** An Edge platform has an inherent geo-distributed structure. Nodes can be clustered into regions depending on their location. Nodes can leverage local connections within those regions to get better connectivity among them. These regions create the possibility of deploying a server dedicated to each zone, in order to be able to process local data relatively to its regions' nodes. By using these local servers, we can achieve less latency to the edge nodes. It is also possible to enable these servers to answer requests relative to their own regions or even gather information of other zones and store it in order to be able to answer quickly to the nodes that request that information.
- **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.

4 Related Work

In this section we review the literature on monitoring systems.

4.1 Existing Monitoring Solution

4.1.1 Astrolabe Astrolabe [15] is a hierarchical monitoring system. Astrolabe 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 (no hosts in common). Astrolabe continuously computes summaries of system data using *on-the-fly* aggregation.

The hierarchical distribution of Astrolabe's 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 is 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*[16]. 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. If they belong to different zones, they exchange information associated with their least common 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 [17]. 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's MIB. An aggregation function for a zone is defined as a SQL program. It takes a list of the zone's children MIBs and produces a summary of their attributes. The only attributes that are writable are in the leaf zones. The attributes of these zones are writable and are the sources of the MIBs the higher level zones.

Each agent has access to (keeps a local copy of) only a subset of all the MIBs in the Astrolabe zone tree. 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.

4.1.2 FMonE

FMonE[18] 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 (can be a device or a message queue).
- *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 (still local to the device) 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.

As depicted by the Figure 1, 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[15] and SDIMS [19], 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.

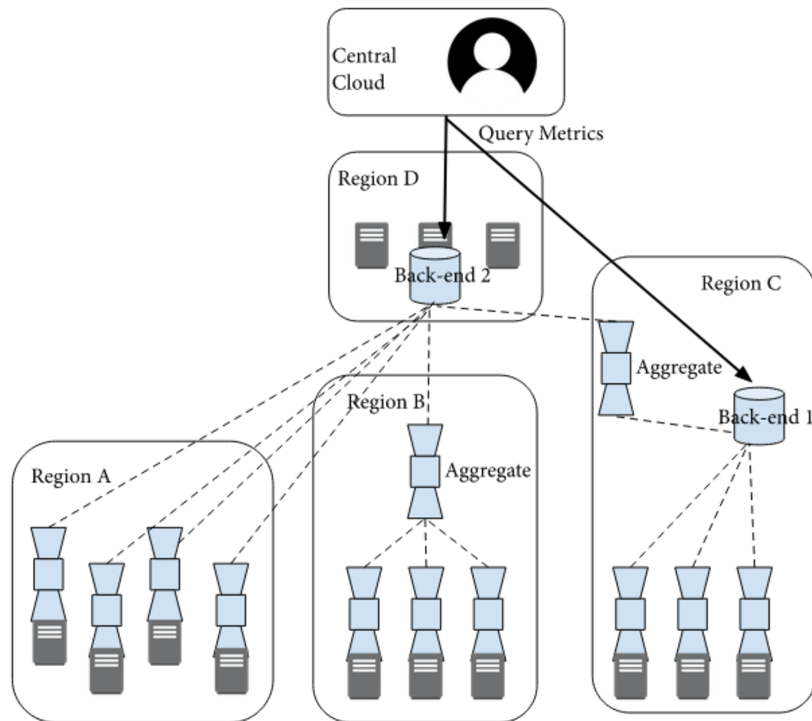

Fig. 1. Pipeline example taken from [18]

4.1.3 Ganglia

Ganglia[20] is a monitoring system for high performance systems such as Clusters and Grids. It is based on a hierarchical architecture, where machines are split between federations of clusters.

Each cluster chooses a representative node. Ganglia creates a tree of point-to-point connections amongst 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.

At each node in the tree (which are cluster's representatives), exists a pull-based service *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 easy, 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 [15], 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.

4.1.4 MonALISA

MonALISA[21] 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 [15], 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.

4.1.5 Monasca Monasca[22], unlike all the other solutions we have seen until now, 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 [23], 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.

4.1.6 SDIMS SDIMS[19] 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 [24] 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 [15], 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 [24], 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.

4.2 Analysis of Monitoring Solution

Due to difficulty of doing a empirical approach evaluation on systems with large heterogeneity, massive distribution and elastic resources, we will follow the strategy used by the authors in [8] and use a qualitative approach to evaluate monitoring solution. We will classify monitoring solutions from each other according to their functional decomposition, architectural model, locality, data collection method, function flexibility and scalability.

- **Functional Decomposition:**
  - **No Decomposition:** In this category, the monitoring function is performed entirely by a monolithic element. It collects all the data from the monitored resources, processes it and exposes it. This model is simple and easy to implement, but restricts the scalability and modularity of the system as it has its functions on a single non scalable element.
  - **Basic Decomposition:** In this category, the *observation* function is deployed on the monitored resources separated from the other monitoring functions which enhances resilience to network failures. Deploying agents on the monitored resources allows the system to adapt the infrastructure and granularity of the measurements.
  - **Fine-grained Decomposition:** Every monitoring function can be deployed on separate locations. This improves modularity and reduces the risk of simultaneous failures of all functions which increases resilience to both servers removals and network failures.
- **Architectural Models:**
  - **Centralized Model:** Only one instance of each monitoring function is deployed except for the *observation* function for which multiple instances may be deployed on the monitored resources. This model limits the monitoring service scalability as it cannot benefit from additional resources. Furthermore, latency between the monitoring service, the monitored resources and the management systems may be significant.
  - **Hierarchical Model:** Deploys a sufficient number of instances for each monitoring function according to the size and requirements of the infrastructure. The scalability is enhanced and latency between instances is reduced. Instances are organized in regions that slotted into levels. Regions in level-n can be composed by hosts or by level-$(n+1)$ regions. This type of infrastructure is more complex and costlier to maintain. Each time a change occurs, either by server removal or network change, the tree organization has to be rebuilt, which is time consuming. The higher the broken node is on the tree, the more significant is the failure impact. This translates to higher levels of locality but it still is insufficient to satisfy the highly elastic nature of Edge environments.

Table 1. Comparison

	Astrolabe	**FMonE**	**Ganglia**	**MonALISA**	**Monasca**	**SDIMS**
Architecture	Hierarchical	Hierarchical	Hierarchical	P2P	Centralized	Hierarchical
Decomposition	None	Fine-grained	Basic	None	Fine-grained	Basic
Data Collection	Mixed	Push	Pull	Push	Push	Push
Locality	+	+	-	-	-	+
Flexibility	-	+	-	+	+	-
Scalability	- [1]	+	+	+	-	+

- **Peer-to-peer Model:** Allows the replication of functions without imposing specific structure to be able organize relations between the nodes. Any instance can connect to any other instance if needed and it still holds the scalability property by allowing multiple instances of a given function. It also solves the expensive structural maintenance issue by removing hierarchical reliance between instances.

– **Locality:** The system allows to differentiate remote resources based on zones. We can query the system to receive information from only specific scopes of the system. This zones can be created based on different criteria like administrative domains (like departments in a University) or geographical distance between the nodes.

– **Data collection method:**

- **Pull:** Every remote resource needs to go through a register process on the monitoring system in order to be found. The system periodically polls the remote resources individually to gather metrics. This puts all the load of data collection on the system instead of splitting it across the remote resources.
- **Push:** Each remote resource is responsible for sending its metrics to the monitoring system. Metrics can be sent based on a time or event condition. It is possible for nodes to do for local aggregation before sending it to the monitoring system, which is beneficial at the Edge level where the the energy consumption of computing that value is lower than transmitting the whole data.

– **Function flexibility:** If it is possible to redefine a node function and if it possible to easily move the function to another node (for example change which node is responsible to aggregate data from a region).

– **Scalability:** If the architecture and protocols used by the system result in a structure that is able to scale efficiently, not only with the number of nodes, but also with the number of metrics.

[1] Although Astrolabe is scalable with the number of nodes, it is not scalable with the number of metrics due to replicating all metrics in a big number of nodes.

4.3 Shortcomings of Existing Solutions for Edge Computing:

The systems mentioned were designed with a specific infrastructure and node organization in mind.

Due to the bigger number of devices in the Edge, data generation is much higher compared to classic Cloud computing systems. Sending all that data through the network is unfeasible. In order to deal with it, we need to aggregate and filter the data with the objective of reducing the volume of information that we need to transmit.

Most of the mentioned monitoring systems use a strict *hierarchical* structure to organize its nodes. Functions are closely tied to the nodes' organization. To perform any change in that structure the system must go through complex processes (like a leader election process). These procedures are expensive and does not allow to easily re-attribute functions (*observation* and *processing*) or create new regions while the system is running without re-configuring the whole structure.

In an Edge system, the environment is always changing with the entrance and exit of nodes. We need to be able to freely attribute a function to any node without diminishing the performance of the system and adapt to the various situations that might arise.

Closely tied to functional flexibility is functional decomposition. It is important to be able to clearly differentiate functions within the monitoring system. Different functions should be performed by different components that do not depend directly on other functions.

With all of this, an *Hierarchical* architectural model with a High functional decomposition and with no strict node organization seems the most viable approach to be able to aggregate and filter data in an Edge system while maintaining locality (due to the existence of regions) and functional flexibility (to adapt to changes in the system).

5 Architecture

In our solution edge nodes are responsible to generate data and push it to the *fog layer* to be processed in cloudlets. Each cloudlet will represent at least one *logical region* from a *hierarchical scheme*. Cloudlets will communicate with each other to synchronize information between zones and gather information from other regions to perform aggregation.

Our goal is to provide a more flexible node organization while maintaining the *hierarchical scheme* provided by systems like Astrolabe[15] and SDIMS[19]. Edge nodes will communicate using a *push* model, while Fog and Cloud nodes will use a *pull* model to gather information from other nodes, as illustrated in Figure 2.

Classic hierarchical schemes like the ones from Astrolabe and SDIMS were designed for grids and datacenters environments. Monitored resources are closer to each other and the monitoring system itself is hosted on the same machines it

is monitoring. In these schemes, data observation and processing functions are closely tied to the node organization. For example, in Ganglia [20] only members of the federation can perform operations over the values of that cluster.

This type of approach results in little flexibility to change nodes disposition and functional responsibilities while the system is running, and when it happens, it relies on complex election protocols to change the nodes assigned to each zone and even after electing the new node it may take a lot of time to change the structure that maintains the hierarchy.

Our solution aims to allow the possibility to having any node taking functional responsibility for any region while maintaining *hierarchical aggregation*.

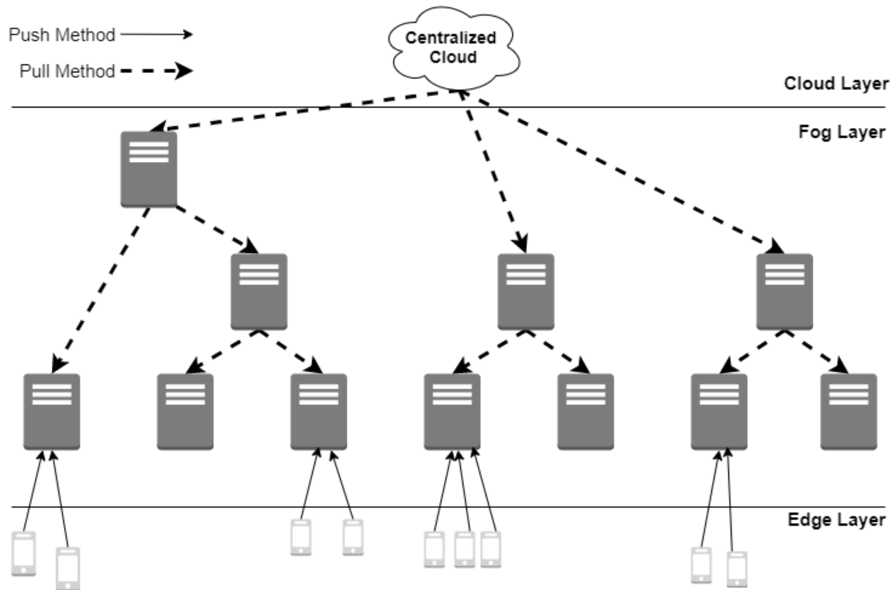

Fig. 2. Communications Model

5.1 Node and Region Organization

The monitoring system will follow Fog/Edge multi-layer architecture, as depicted on Figure 3. At the lower level we will have the *Edge layer*. Edge nodes will belong to at least one *logical region*. They will generate and send data to one fog node *responsible* for its region. The *Fog Layer* will contain the nodes responsible for receiving data from edge nodes and processing it. Each fog node can be responsible for multiple regions. Regions can be composed by edge nodes or by another regions. And at the top level we will have the *Cloud layer* that

will have a global view of the system and will be able to perform data analysis at a global scope.

The *logical regions* need to be specified by administrators and are registered on a *Lookup Service*. The *Lookup Service* will serve as a name service. It can be used to find which node represents which regions and how are the regions composed (see Section 5.2). Each region will have a pool of servers that are responsible for collecting and aggregating information for the region they are assigned to. This list can change at any time with the entrance and exit of servers. The servers will register on the lookup service by declaring which regions they want to be responsible for.

When an edge node wants to join the system, it contacts the lookup service in order to find out the location of one of the servers responsible for its region and starts sending data to it.

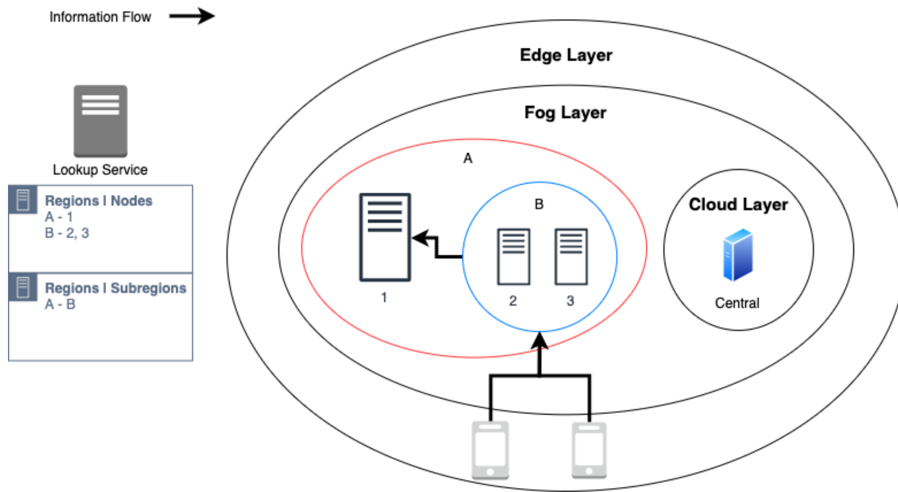

Fig. 3. System Architecture Model

5.2 Lookup Service

Our solution will be based on MonALISA's [21] lookup service. It will be used to find the IP addresses of the responsible servers each region. It will store a list of servers and their location for each zone. It will also store hierarchical levels by having a list with the level-n regions that compose level-$(n+1)$ regions.

Nodes will also store those lists and synchronize them with the Lookup Service. When a node leaves or joins a region, the lookup service announces it to all the other region's nodes so that they can update their local lists. This is done to allow nodes to operate independently from the lookup service after registration.

Just like MonALISA's [21] service, registration will work based on a lease approach. Servers will register themselves on the lookup service by providing their IP address and the regions they want to take responsibility for. Upon failure of renewing the lease the server will be removed from the lookup service.

When joining the system, edge nodes will query the service to receive an IP address for a server that is responsible for its region. With base on metrics that yet to be defined, the lookup service will select the best server from the region's list to give that querying node.

We are still not sure if this component will be centralized in order to have a global notion of the system or if we will try to distribute it. There will also be some kind of replication of this service in order to get resilience in case of failure of the service.

5.2.1 Communication between hierarchical levels: Regions from higher hierarchical levels will *pull* information from lower levels regions in order to calculate the aggregated values. When a node wants to aggregate the values from lower regions it will use its local list of sub-regions and query the lookup server for an IP address of a responsible for those lower level regions. Then the server will *pull* all the necessary data from the lower regions, process it, and aggregate it.

This behavior will work in recursive way until we reach the root node, which is represented by the centralized nodes in the *Cloud layer*.

5.2.2 Synchronization among nodes on the same region: It will be possible to have multiple nodes responsible for each region. In order to have *locality* without every node of the region and also have *resilience* in case of server failures we need to synchronize the servers state. The *responsible* nodes within the same region will periodically (or upon an trigger event) trade their sets of information, in a similar way to Astrolabe's gossip protocol (see 4.1.1).

5.3 Edge Layer

Edge nodes will be the primary data source for the system. Nodes at this layer are highly volatile and dynamic relatively to resource availability and structure (OS, hardware architecture).

5.3.1 Data Collection: We will use an agent based approach. It will implement a *Push* communication protocol between the edge nodes and the processing servers on the *Fog Layer*. We will deploy an agent on each remote resource. It will distribute the load of gathering the information through the edge nodes by periodically, or upon certain condition, send the metrics to the processing servers. The use of push instead of pull allows the system to scale better with the number of monitored resources, and also allows to deal better with edge nodes entering and exiting the system (see Section 3.1).

5.3.2 Agent Deployment:
For the agents to be easily deployed at the Edge level we need to employ a method that is able to be used in a heterogeneous environment, independently from the Operating System and architecture. We plan to use *Containers* [25] due to its ease of deployment in all kinds of OS and architectures. Containers are deployed by having a system (lets say our Lookup Service) send an image of the container to a node and then starting the image. This images are smaller in size than a full fledged VM and require less resources to run.

5.3.3 Distributing edge nodes through regions
Nodes joining the system will query the Lookup Service. Based on the location of the node or other metrics, will give the more adequate region for the node to join. This will allow to group nodes using different rules. For example, it will be possible to group geo-related nodes into regions associated to a specific geographical area. In this case it will enable geo-aware queries simply due to the way the nodes are distributed among regions without extra work from the system.

6 Evaluation

Our main goal is to provide an Edge monitoring solution with a flexible structure to organize the system's nodes while maintaining the hierarchical properties that allow us to reduce the volume of data on the system. The tests will be conducted in the Grid'5000 testbed [26] in order to allow us to set up different scenarios for the system. With this, we will compare our Edge components with Monasca (our agent is based on Monasca's solution) and our fog and cloud components with Astrolabe (in order to compare our hierarchical solution). We will assess the impact that our solution has in respect to response latency; agent overhead and resource usage; and elasticity of the system upon node failure and/or function re-attribution.

6.1 Response Latency

It is important to be able to see how our solution impacts latency to query responses in an Edge Scenario. We will setup both our solution and Astrolabe in Grid'5000 in an Edge-like infrastructure. Our solution will have three layers: edge nodes spread across multiple locations with big distance between them; multiple middle nodes representing the fog layer, these nodes will be distributed to be in various degrees of distance with the edge nodes; and a central node (far from the edge nodes). Astrolabe will have all of its physical nodes hosted in multiple regions (with different distances between them) in order to simulate an edge scenario. Afterwards, we will measure the time it takes to answer to different queries within within different scopes (i.e., global, local and between regions) and compare those values between both systems.

6.2 Overhead and Resource usage

We will monitor the impact that our monitoring system has on resource usage for each node in the edge layer (as it is the more sensitive layer to resource usage). We will also want to measure the time it takes for an edge node to be up and running upon joining the system.

At the edge level, the most interesting comparison we can do is compare our agent with Monasca's, as it is the system we based on to develop our agent. In order to simulate the Monasca architecture, we will setup our solution in a centralized structure, with multiple edge nodes, linked directly to a centralized node that will collect all the metrics. We will then use YCSB [27] to perform these tests. YCSB agents will continuously query the database in the central node until they reach a threshold and measure the amount of operations/sec that the agents are able to perform. Afterwards, we will do the same to Monasca and compare both values.

6.3 Elasticity

Since the biggest improvement our solution brings is the possibility of re-adapting the hierarchical structure and the possibility of re-attributing functions to nodes, we need to measure how quickly the system can adapt to these changes.

We need to assess the time it takes for other nodes to acknowledge a function re-attribution/removal. To do this, we will setup the same topology in our solution and Astrolabe, remove the nodes that are responsible for aggregating a specific zone, and measure the time it takes for that region to start aggregating those values again.

It's also important to measure the time it takes for a node to start running its agent upon joining the system. Because new nodes require a container image, to start up the agent in order to monitor, there is a possible overhead there. We will setup various tests where we will have different number of edge nodes joining the system at the same time (i.e., 10, 15, 20 nodes joining and requiring the container image) and measure the time it takes the nodes to start sending metrics. We will check if there is difference in performance when increasing the number of nodes, and if there is going to be any difference between joining the system while already having the image (which happens when rejoining) versus having to pull the image from the monitoring system.

7 Scheduling of Future Work

Future work is scheduled as follows:

- January 9 - March 29: Detailed design and implementation of the proposed architecture, including preliminary tests.
- March 30 - May 3: Perform the complete experimental evaluation of the results.
- May 4 - May 23: Write a paper describing the project.
- May 24 - June 15: Finish the writing of the dissertation.
- June 15 Deliver the MSc dissertation.

8 Conclusions

There is a need to monitor an edge system to track its behavior and to be able adapt the system to different conditions. The edge brings an increase of the number of highly volatility devices that generate data. Sending all the data to centralized servers to process it may be unfeasible. This motivates the need for a monitoring service capable of aggregating and filtering data to reduce the volume of information transmitted and that gives a flexible structure to construct different aggregation regions to adapt to different applications.

In this report, we surveyed the structure of existing monitoring systems, the requirements imposed by the fog/edge environment, the existing cloud computing monitoring solutions, and their shortcomings in an edge environment. We also elaborated a solution that allows to maintain a hierarchical architecture while allowing functional flexibility. Finally, we discussed several potential strategies for evaluating the proposed architecture.

Acknowledgments We are grateful to Nivia Quental for the fruitful discussions and comments during the preparation of this report.

References

1. Taherizadeh, S., Jones, A.C., Taylor, I., Zhao, Z., Stankovski, V.: Monitoring self-adaptive applications within edge computing frameworks: A state-of-the-art review. Journal of Systems and Software (Feb 2018)
2. Prasad, V., Bhavsar, M., Tanwar, S.: Influence of monitoring: Fog and edge computing. Scalable Computing (May 2019)
3. Greenberg, A., Hamilton, J., Maltz, D., Patel, P.: The cost of a cloud: Research problems in data center networks. Computer Communication Review (Jan 2009)
4. Satyanarayanan, M., Bahl, P., Caceres, R., Davies, N.: The case for vm-based cloudlets in mobile computing. IEEE Pervasive Computing (Oct 2009)
5. Bonomi, F., Milito, R.: Fog computing and its role in the internet of things. Proceedings of the MCC workshop on Mobile Cloud Computing (Aug 2012)
6. Zhang, B., Mor, N., Kolb, J., Chan, D.S., Lutz, K., Allman, E., Wawrzynek, J., Lee, E., Kubiatowicz, J.: The cloud is not enough: Saving iot from the cloud. In: 7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15). (July 2015)
7. Aceto, G., Botta, A., De Donato, W., Pescapè, A.: Cloud monitoring: A survey. Computer Networks (2013)
8. Abderrahim, M., Ouzzif, M., Guillouard, K., Francois, J., Lebre, A.: 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. Martin-Flatin, J.P.: 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. Fatema, K., Emeakaroha, V., Healy, P., Morrison, J., Lynn, T.: A survey of cloud monitoring tools: Taxonomy, capabilities and objectives. Journal of Parallel and Distributed Computing (Oct 2014)
11. Shi, W., Cao, J., Zhang, Q., Li, Y., Xu, L.: Edge computing: Vision and challenges. IEEE Internet of Things Journal (Oct 2016)
12. Zhu, J., Chan, D.S., Prabhu, M.S., Natarajan, P., Hu, H., Bonomi, F.: Improving web sites performance using edge servers in fog computing architecture. In: 2013 IEEE Seventh International Symposium on Service-Oriented System Engineering. (March 2013)
13. Chun, B.G., Ihm, S., Maniatis, P., Naik, M., Patti, A.: Clonecloud: Elastic execution between mobile device and cloud. (Jan 2011)
14. Rudenko, A., Reiher, P., Popek, G., Kuenning, G.: Saving portable computer battery power through remote process execution. Mobile Computing and Communications Review (March 1998)
15. Van Renesse, R., Birman, K., Vogels, W.: Astrolabe. ACM Transactions on Computer Systems (May 2003)
16. Demers, A., Greene, D., Houser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H., Swinehart, D., Terry, D.: Epidemic algorithms for replicated database maintenance. ACM SIGOPS Operating Systems Review (1988)
17. Hare, C.: Simple network management protocol (snmp). (2011)
18. Pérez, M., Sanchez, A.: Fmone: A flexible monitoring solution at the edge. Wireless Communications and Mobile Computing (Nov 2018)
19. Yalagandula, P., Dahlin, M.: A scalable distributed information management system. SIGCOMM Comput. Commun. Rev. (August 2004)
20. Massie, M.L., Chun, B.N., Culler, D.E.: The ganglia distributed monitoring system: design, implementation, and experience. Parallel Computing (2004)
21. Newman, H.B., Legrand, I., Galvez, P., Voicu, R., Cirstoiu, C.: Monalisa : A distributed monitoring service architecture. CoRR (2003)
22. Openstack Project: Monasca wiki. https://wiki.openstack.org/wiki/Monasca Accessed: 2019-12.
23. Kreps, J., Narkhede, N., Rao, J., et al.: Kafka: A distributed messaging system for log processing. In: Proceedings of the NetDB. (2011)
24. Rowstron, A., Druschel, P.: 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)
25. Bernstein, D.: Containers and cloud: From lxc to docker to kubernetes. IEEE Cloud Computing (2014)
26. Cherrueau, R., Pertin, D., Simonet, A., Lebre, A., Simonin, M.: Toward a holistic framework for conducting scientific evaluations of openstack. In: 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID). (May 2017)
27. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with ycsb. In: Proceedings of the 1st ACM symposium on Cloud computing. (2010)