Graph-based Root Cause Analysis for Service-Oriented and Microservice Architectures

Álvaro Brandón[a], Marc Solé[b], Alberto Huélamo[b], David Solans[b], María S. Pérez[a], Victor Muntés-Mulero[b]

[a] *Universidad Politecnica de Madrid, Madrid, Spain*
[b] *CA Technologies, Barcelona, Spain*

Abstract

Service-oriented architectures and microservices define two ways of designing software with the aim of dividing an application into loosely-coupled services that communicate among each other. This translates into rapid development, where each service is developed and deployed by small teams, enabling continuous shipping of new features and fast-evolving applications. However, the underlying complexity of this type of architecture can hinder observability and maintenance by the user. In particular, identifying the root cause of an anomaly detected in the application can be a difficult and time-consuming task, considering the numerous services and connections to be examined. In this work, we present a root cause analysis framework, based on graph representations of these architectures. The graphs can be used to compare any anomalous situation that happens in the system with a library of anomalous graphs that serves as a knowledge base for the user troubleshooting those anomalies. We use the Grid'5000 testbed to deploy three different architectures and inject a set of anomalies. The results show how our graph-based approach is 19.41% more effective than a machine learning method that does not take into account the relationship between elements.

Keywords: SOA, Microservices, Root Cause Analysis, Containers, Graphs

1. Introduction

As many industries increasingly rely on information systems to operate efficiently, software development and architectures have evolved in different directions. Virtualisation has gained momentum in the last decade thanks to the ability of cloud providers to offer Infrastructure as a Service (IaaS) to their clients. This enables customers to use processing power on demand through Virtual Machines (VMs) and it eliminates the burden of maintaining the infrastructure. It also allows the provider to optimize the utilization of the datacenter by means of VM migration and consolidation. The next step in virtualization has been containerisation. A container is an executable unit of packaged software that includes all the dependencies needed. It runs on the host operating system, sharing the kernel and using its own user space, isolating it from other containers. This eliminates any boot time overhead in comparison with VM's, empowering the migration and scaling advantages of virtualisation. A machine can also host dozens of containers at the same time[1] thanks to their lightweight nature. Finally, from a software development point of view, they eliminate the "it works on my machine" problem, since the whole running environment is included inside the container.

Industry has acknowledged all the containers benefits [1], specially thanks to the adoption of technologies like Docker [2], with DevOps [3] and microservice architectures becoming growing trends and common

Email addresses: abrandon@fi.upm.com (Álvaro Brandón), marc.solesimo@ca.com (Marc Solé), alberto.huelamosegura@ca.com (Alberto Huélamo), david.solans@ca.com (David Solans), mperez@fi.upm.com (María S. Pérez), victor.muntes@ca.com (Victor Muntés-Mulero)

[1] https://www.datadoghq.com/docker-adoption/ (last accessed Dec 2018)

practices. The philosophy behind microservices follows the same principles used in service-oriented architectures [4]. It consists of breaking the system logic into different, small units where each one has a single task or responsibility. The different units or microservices communicate and cooperate with each other to provide the system functionality as a whole. Any of the aforementioned units of logic is executed inside a container. In comparison with a monolithic architecture, it allows the user to scale specific components of the application by increasing the number of containers responsible for that part. Besides, it also enables new paradigms like serverless computing, where certain events trigger the allocation of new containers, abstracting the infrastructure needed by the user [5]. Finally, it also facilitates the development process and software reuse [6].

Coordinating and scaling these containers require an orchestrating unit, specially if we want to deploy them in a distributed infrastructure with several machines. Tools like Kubernetes[2] or DC/OS[3] have filled this gap. These platforms have self-healing features, where unhealthy or faulty containers can be relaunched, or containers migrated to a different machine in case one of their hosts dies. But establishing the root cause of these failures can be really complex, specially when we have a network of different services that depend on each other. The troubleshooting process normally involves a tedious search through logs across the different containers, trying to find the faulty piece in the chain.

In this paper, we present a graph-based method to perform root cause analysis (RCA) in service-oriented architectures, such as microservices. We argue that a graph representation of the system is able to capture important information for RCA, like the topology of the architecture or the different connections, both logical and physical, between elements in the system. Based on this observation, we build a framework with two goals: allowing the user to better understand what is the current behaviour of the system (anomalous situations, metrics, communication between elements, etc...) and matching an anomalous situation in the system with a previously diagnosed situation. Our contributions are:

- After reviewing the related work **(Section 2)**, we present a graph-based root cause analysis framework **(Section 3)** for service-oriented architectures.
- We describe a graph comparison engine **(Section 4)** that allows us to match an anomalous graph representation of the system with one that happened previously, in order to establish its root cause.
- By using a set of monitoring tools, we prototype a framework able to build a graph representation of the architecture state **(Section 5)**.
- We evaluate the RCA pipeline with a series of real experiments in a testbed where we deploy, using containers, three service-based architectures widely adopted in Industry. We then inject anomalies into the hosts and containers (**Section 6**). The results show how our approach is able to capture more information about the elements that could influence or could be influenced by the anomaly than other methods that disregard them.
- We wrap up with threats to validity (**Section 7**) and a discussion about the advantages and limitations of our root cause analysis method on service-oriented architectures (**Section 8**).

We finish the paper with a section on future work and open challenges that we have identified during our research.

2. Related work

There are three important research topics we studied to design our system. First, root cause analysis and the different approaches to it. We also surveyed graph similarity and matching techniques, which will be our core techniques for the RCA process. Finally, we reviewed recent work on microservices performance and their tolerance to failures.

2.1. Root Cause Analysis

There is a large body of work related to Root Cause Analysis (RCA) (see [7, 8] for extensive surveys). All RCA techniques can be classified according to different characteristics. One relevant classification to frame

[2]https://kubernetes.io/
[3]https://dcos.io/

our technique is on whether the diagnostic model knows the causality structure of the diagnosed system. They are sometimes referred as *model-based* approaches [9, 10]. For example, Wolfgan et. al [11] present a model-based debugging framework that is able to detect programming errors in software by means of the use of a set of models built through the analysis of source code. Similar techniques are applied by the same authors in the field of web services [12], which is even more relevant for our work. In contrast to these approaches, we do not focus on failure only, but also on performance degradation or other type of problems that might not involve a failed execution.

On the other side of the spectrum, classification-based approaches, e.g., [13, 14] (sometimes called *model-free*), typically generate some internal model using Machine Learning (ML) techniques, trading the need of explicit domain knowledge by having lots of a lighter-weight type of knowledge (i.e., classification labels).

Graphs have been connected to RCA in many ways. Most of the previous attempts are based on the use of Probabilistic Graphical Models (PGMs) [15] as (causal) models for diagnosis [16, 17] or as underlying descriptive model of the system to diagnose. This latter view is the approach taken here, as we consider graphs as entities describing the state of the system in a classification-based approach. To limit the necessity of labeled data typical from classification-based techniques, a nearest neighbors approach has been taken, in which domain knowledge is added in the form of weights altering the metric space. Other uses of graphs in RCA have been rather limited to anomaly detection [18] and building alternative (non-standard PGMs) graph models [19]. The most relevant and similar work to our approach is the proposal of Liu et al. [20], in which frequent subgraphs are extracted from graphs representing a flow of function calls inside a piece of software. The frequent subgraphs can be used to infer the backtrace that was followed before the bug. However the programmer still has to infer the root cause. Additionally, Dot2Dot [21] is a system that is able to group anomalous nodes inside a graph and provide an explanation on why they were grouped. Similar to our approach, they assume that a set of nodes are already labelled as anomalous and then a root cause needs to be found. However, it only considers the connection between nodes and not their attributes.

2.2. Graph Similarity and Matching

Graph similarity and matching are the core techniques used by our RCA approach to find similar graphs in our library of anomalous graph patterns. These two fields have been widely studied in the pattern recognition field through the years. Both disciplines intersect in the sense that the optimal matching between the nodes of two graphs is used to calculate similarity. In our case, we focus on the inexact graph matching problem with multi-attributed graphs. This section is not meant to be an exhaustive study of the state of the art, since this is a very broad field, but rather a general overview of the different methods available to compare two graphs.

The matching problem has been addressed in different ways, as shown in an extensive survey on the topic by P.Foggia et al. [22]. Some techniques are based on a tree search, like Hidovic et al. [23], whose computation time can be improved through heuristics [24]. Another approach is to consider the problem as a continuous optimization instead of a discrete one, using one of the many available optimization algorithms to find the matching [25]. Additionally, spectral methods are based on the idea that the eigenvalues of a matrix are the same, irrespective of the order of rows and columns. Therefore, the matrix representation of the two graphs can be used to compare the eigenvalues and establish a similarity [26]. There are also methods for large graph matching [27], but we focus more on smaller graphs since the comparison will be made with smaller anomalous subgraphs, extracted from the general and larger system graph.

Finally, some approaches are not strictly graph matching techniques, but provide ways of measuring how close two graphs are in a vector space. The first one is graph embeddings, which map either the nodes of a graph or the graph itself onto points in a vector space [28, 29]. The second one is graph kernels, which is a function that maps two graphs into a real number. This technique is used in Kashima et al. [30], where sequences of random walks on the graphs are used to compute the kernel function.

2.3. Microservices

Microservices are catching the attention of both researchers and Industry and are being used extensively in cloud infrastructures [3]. They also have synergies with new types of distributed infrastructures, such as

edge computing and internet of things [31]. Additionally, they have opened new lines of business for cloud providers, such as serverless computing [32].

Ensuring the performance and resilience of microservices has been addressed in recent work. Dullman et al. [33] study the effects related to the high dynamicity of microservice architectures where containers are added or removed regularly. These same authors propose a framework to define topologies for microservices, deploying them and benchmarking their performance and resilience through problem injection [34]. This setup can be monitored afterwards in order to extract insights of the operation of the system. Following the same line of work, Gremlin [35] presents a framework for testing the resilience of microservices, allowing the user to design test scenarios that are translated into anomalies injected in the network. François et al. [36] model an Internet of Things infrastructure as a graph representation and focus on detecting anomalies that happen in the intercommunication between services. However, all of these solutions deal more with the anomaly detection and resilience testing part of microservices than with RCA.

Some other research efforts are the work of Mayer et. al. Namely, a dashboard showing information about running microservice architectures [37] and a model that can be used to choose the right microservice monitoring tool depending on different criteria such as requirements, stakeholders needs or performance [38]. Although monitoring is an important part of our approach, we focus more on giving an explanation of an anomaly extracted from the monitored information.

More related to RCA, we find Sieve [39], which provides a platform that gathers metrics from the elements of the microservice, filters them and finds dependencies between components in the system. They use RCA as one case-study to show the benefits of Sieve, using an Openstack deployment as a scenario and two known bugs of it as the anomaly. In comparison, we do not perform any filtering and use all the available metrics to build a topology of the microservices communication.

As companies are adopting microservices, some industrial tools have been designed to ensure the performance and resilience of these architectures. Netflix uses the Simian Army, [40] a set of processes that inject failures in the microservice architecture to validate its resilience. In any case, this is more a tool to ensure resilience than a RCA approach. Some industrial tools such as Instana[4] or Dynatrace[5] have recently included AI powered root cause analysis. However, the details on how they work are not given and the user cannot provide feedback as with our weights system.

3. Graph based RCA

To motivate why graphs are a good fit to represent microservices and service-oriented architectures, we use a common web serving scenario. In this architecture, we have a number of clients that perform HTTP requests to a load balancer, such as HAproxy[6], which redirects these requests to a set of stateless web servers, for example WordPress[7]. The web servers also store any state information on a backend database such as a traditional MySQL[8] server. Each of these elements is a container running on a machine of the cluster. Every container or host has its own metrics and communicates with other containers or hosts in the system. For example, the load balancer will communicate with all the web server instances in the architecture, but there will be no interaction between the web servers. Graphs allow us to formalize the logical and physical connections between the different elements of the system through edges. In addition, all of the metrics information can be included as attributes on the nodes, as seen in Figure 1. These attributes can be numerical values such as metrics, categories such as a label specifying if the node represents a container or a physical machine or even ontological classes that could help to model the domain, like knowing that WordPress and HAProxy are both frontends. Some other attributes, like vectors or information from logs, can be easily added in order to improve the system.

[4]https://www.instana.com
[5]https://www.dynatrace.com
[6]http://www.haproxy.org/
[7]https://wordpress.com/
[8]https://www.mysql.com/

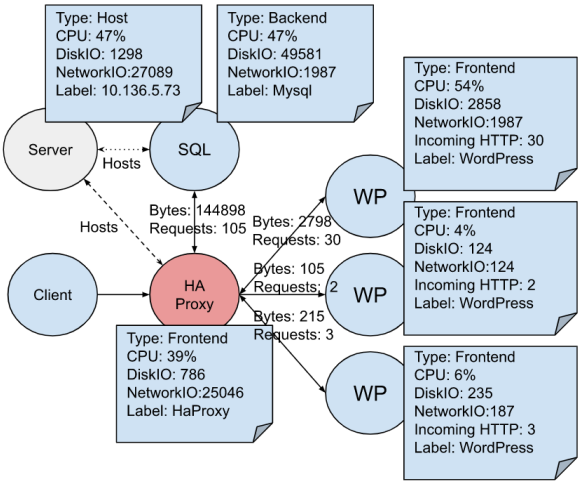

Figure 1: Simplified example of a multi-attributed graph model for an example architecture. An anomaly is generated in the HA Proxy instance due to the unbalanced number of requests that the WordPress instances are receiving

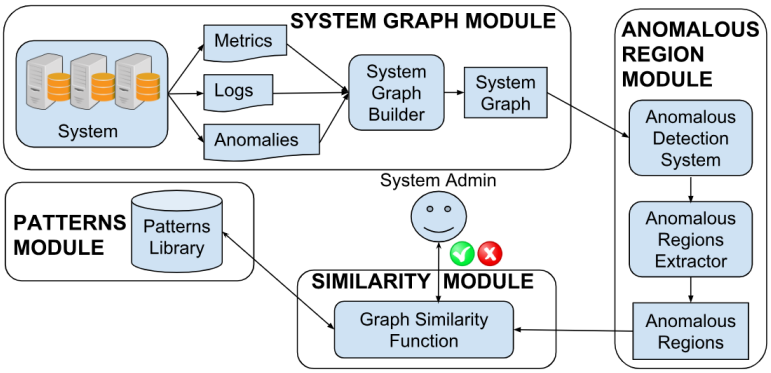

Figure 2: Architecture of the proposed system

It seems logical, and experience tell us so [35], that any anomaly or drop in performance can have a cascading effect. In the case of the load balancer, it might propagate to the webservers as they will see less incoming requests than usual. Similarly, in a container environment, any anomaly in the functioning of the host might be propagated to the containers it hosts. The graph representation allows to identify this propagation effect, since we are able to determine which nodes or elements are connected to an anomalous one.

Based on this graph representation, which we will call system graph from now on, we propose a RCA system that is able to match an anomalous region in the system, represented as a subgraph, with a similar anomalous graph from the past that has already been troubleshooted by an expert. There are several pieces involved in this process represented as a general architecture in Figure 2. In the following subsections we will describe the architecture in more detail. Note that this approach can be applied to other systems, such as local processes that are not containerised. However, and as previously mentioned, the approach excels with microservice and service-oriented architectures, since the different parts of an application are decomposed into units that interact between each other.

3.1. The system graph module

The responsibility of this module is to embed into a graph representation all the collected metrics, network activity, logs and anomalies generated in the system for a given time window (e.g. 15 secs). We can

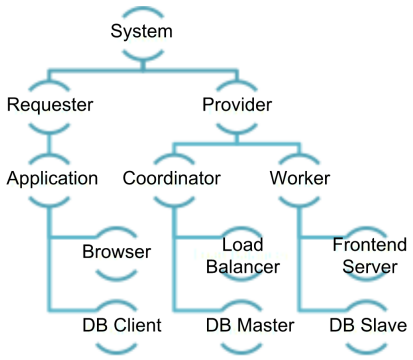

Figure 3: An example of a taxonomy explaining a hierarchy of concepts. When comparing graphs, the taxonomy will allow us to reason those concepts situated close to each other in the tree, concluding that they are more similar. For example a DB master is very similar to a load balancer, since they both are coordinators that distribute some load to their slaves.

think about this graph as a snapshot of the current state of the architecture. The size of the window can be adjusted by the user based on criteria such as the architecture change rate or the monitoring frequency of the agents. There is a trade-off between a large or a too narrow time window. Larger time windows will result in some anomalies not being detected, due to the coarse-grained information of the metrics. Too narrow ones will result in noisy information and a lot of overhead when collecting metrics, logs and anomalies.

The process involved in the graph creation for our particular system graph will be further explained in Section 5. Here, we provide a general overview of the graph elements, which can be applied to any other context besides containers.

Nodes: Nodes represent either containers, hosts or any other element outside the cluster that interacts with our system e.g. an external client accessing a frontend. Nodes may have multiple attributes of different types. We explain the different types further in this section.

Edges: Edges represent any network communication between elements in the system, such as TCP connections or HTTP requests. They also represent a logical connection, such as the relation between a container and the machine hosting it. Same as with nodes, edges may have multiple attributes of different types.

Attributes: Attributes are used to capture the information collected from metrics, logs and anomalies. We provide a list of the types considered in our framework:

- **Numerical**: Here we include metrics that can take any value over a discrete or continuous range. One obvious example will be the cpu usage of a container, but it can also be found in the edges. For instance, the number of bytes sent from one container to another.
- **Categorical**: They take values that are names or labels. One example would be the Docker image for a particular container (e.g. Haproxy v1.8, WordPress v4.9, etc...)
- **Ontological**: This attribute type corresponds to an ontology class and it is used by the system to represent a hierarchy of concepts. A simple example of an ontology in the form of a taxonomy is depicted in Figure 3. These kinds of attributes are useful to generalise problems that are semantically the same but involve different elements. For example in the loadbalancer scenario, it would have been equivalent to have Nginx instances instead of WordPress instances, since they are both *Frontend Servers*.

Anomaly level: This can be considered as a special type of attribute, indicating that a given node or edge is experiencing an anomaly. As we will see later, any anomaly detection mechanism can be used to identify problems and tag the relevant elements of the graph. It will be used afterwards by the anomalous region extractor to build an anomalous subgraph out of the system graph.

3.2. The anomalous region module

The output graph of the previously described system graph module will be analyzed by the anomalous region module to extract the different anomalous elements of the graph. The anomalous region module

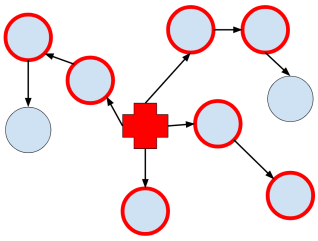

Figure 4: An example function that can be used for the anomalous region extractor. The anomalous node with the cross shape generates an anomalous region including neighbours that are located at 1 or 2 hops from anomalous nodes, representing a cascading effect where the performance of one element depends on the others

consists of two submodules: (i) an anomalous detection system that is able to detect and tag anomalous nodes inside the system graph and (ii) an anomalous region extractor. For the former, any of the state-of-the-art techniques for anomaly detection can be used [41]. Note that we assume the existence of this submodule, as we focus on the problem of finding a root cause for the detected anomaly and not the detection of the anomaly per se.

The anomalous regions extractor is a function that generates an anomalous subgraph for each of the nodes that have been tagged as anomalous. We assume that an anomaly is normally caused by an element that is somehow related or connected with the aforementioned anomaly. This is based on the intuition that a malfunctioning service in the chain of interactions will affect the services depending on it. The objective of the function is to limit the number of nodes and edges that are going to be included in the similarity calculation for scalability purposes. If the system has few computational resources, we can use a function such as the one depicted in Figure 4, where a function creates an anomalous region by taking the neighbours up to a distance of 2-hops for each anomalous node. One obvious drawback of this approach is that the root cause could be left out of the scope of this subgraph. However, if the whole architecture wants to be considered and we have the computational resources, we can use a function that takes all the neighbours without any distance limit. Again, this is customisable since we can insert here any kind of function that is able to extract anomalous subgraphs from nodes or edges that have an anomaly level other than zero.

As this function extracts a subgraph from the general system graph, it will also keep its nodes, edges and attributes.

Nodes, edges and attributes in the anomalous region can additionally have weights assigned, representing the importance of a given element in that anomalous region. For instance, an anomaly involving a malicious process consuming a lot of cpu in one machine, could have its cpu usage attribute weight increased for that node with respect to other attributes of the graph. These weights will be taken into account by the graph similarity module, as we will see in later sections.

The extracted anomalous regions are sent to the similarity module, in order to match them with already troubleshooted anomalies stored in the library.

3.3. The pattern library module

A pattern is an anomalous region that has already been revised and labeled by an expert. These patterns will act as reference templates with which anomalous situations currently happening in the system can be matched. As we will see later in Section 3.4, patterns are added to the library if a new anomalous region is not similar enough (up to a given threshold) to any of the graphs currently present in the library. In addition to this, it can initially be filled manually or as a result of importing the pattern library from a different deployment. The objective is to have an initial set of the most common errors that can affect the system, in order to avoid the cold-start issue. The more patterns, the more complete this library will be, although it will also increase the search time. The user can access the library and visualize the different anomalous graph patterns, browsing their metrics, logs and connections. This visualisation will help the user to enrich the existing patterns by adding metadata through the following actions:

Changing weights: Same as with the anomalous regions, the user can change the aforementioned weights and give more importance to certain nodes, edges or attributes. This will help in the RCA process

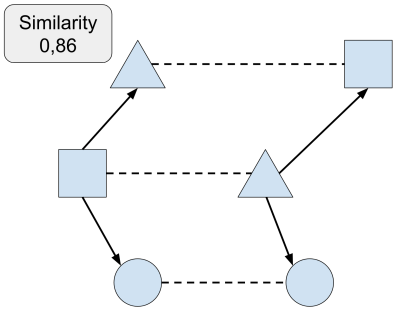

Figure 5: A representation of the mapping between nodes and the similarity score given by the graph similarity function

since the important elements are pinpointed. Consequently, matching the pattern with any new incoming anomaly of the same type will be easier.

Cause label: It is a label that represents the main reason or root cause for the anomalous pattern. Same as with weights, it is normally assigned by an expert in the domain that analyses the graph structure and establishes a root cause. It can also come with a series of steps needed to fix the anomalous situation by the system administrator or with automatic prescriptive actions that can be triggered if the pattern occurs.

Note that we have not implemented this visualisation and interaction with the user part as we wanted to focus more on the effectiveness of the RCA graph comparison method.

3.4. The similarity module

The similarity module relies on an inexact matching function that considers the similarity distance between two multi-attributed weighted graphs. The module receives two inputs: an anomalous region generated by the anomalous region module and a potentially tagged pattern from the pattern library module. The output consists of two elements: a similarity score and a mapping of nodes from graph A to graph B. The former goes from 0 to 1, with a score of 1 meaning that the two graphs are identical and 0 that they are completely different. The latter describes which graph elements from the first graph are matched with the elements from the second graph to achieve this similarity score. A visual example of this function is depicted in Figure 5. Note that the details of the function can be implemented in different ways, applying any state-of-the-art graph matching techniques. We give the details of our function in Section 4.

Every time a new anomalous region arrives, the module tries to match it with one or more graph patterns of the library. We consider that there is a match between two graphs, when the similarity value between them is above a user-specified threshold. If there is no match, the anomalous region is added as a pattern, waiting for an expert to troubleshoot and tag it. If there is one or more, the list of matched patterns can be handled in different ways. We can show it to the user as a ranked list with their root causes (labels). Then the user can validate or reject the different matchings. While the validation means that the matching is aligned with the ground truth, rejection indicates that the comparison did not capture the underlying information in the graphs. Another option is ordering the list by similarity and applying an automatic healing procedure based on the top-similar root cause. We leave this criteria open as it depends on the domain and the criticality of the situation.

The threshold element creates a trade-off between correctly classified anomalies and search time inside the library. Decreasing the threshold will result in having several pattern to explain the same root cause, more options to compare with and higher accuracy in the classification, as we will see later in the evaluation section. However, more patterns will obviously mean longer search times. Some heuristics that can be used are setting a relatively high threshold and verifying if the classification error is acceptable or the threshold should be decreased. We empirically found that a threshold between 0.80 and 0.90 is able to define well the difference between two different anomalous situations. Finally, we would like to point out that correctly tuning the weights will allow the user to set higher thresholds, since the similarity scores will be higher between having their most important elements pinpointed.

4. Graph similarity engine

One of the central pieces of the framework described in the previous section is a function able to find a similarity score between two graphs. In this section, we give the details of our graph similarity function implementation and the particularities that allow introducing expert knowledge from system administrators.

4.1. Problem definition

Definition 1. A graph G is represented by a 5-tuple:

$$G = (V, E, att, C, w) \tag{1}$$

V is a set of vertices. $E \subseteq V \times V$ is a set of edges. att is a function such that $att(V \vee E) = A$, where A is a list of attributes a represented by tuples $a = \{val, weight, c \subset C\}$, with val being its value, $weight$ being the weight or importance of the attribute inside that vertex or edge and c being the context of the attribute. C represents the graph context and is used to explain the attributes that are included in the graph and to give a context in which to compare its values, as we will see later on the section. w will be a function such that $w(x)$ returns either the weight of an edge or node within its graph, or the weight of an attribute within its node or edge.

Definition 2. Given two graphs:

$$G_1 = (V_1, E_1, att, C, w), G_2 = (V_2, E_2, att, C, w)$$

And two injective functions $m_n : V_1 \to V_2$ and $m_e : E_1 \to E_2$, the former returning the matched node of G_1 into G_2 and the latter returning the same for edges, we have an optimisation problem where we want to find the mapping that maximises the similarity between the two graphs, given by the formula:

$$\frac{\sum_{v \in V_1} (w(v)+w(m_n(v))) \cdot sim(v,m_n(v)) + \sum_{e \in E_1} (w(e)+w(m_e(e))) \cdot sim(e,m_e(e))}{\sum_{v \in V_1} w(v) + \sum_{v \in V_2} w(v) + \sum_{e \in E_1} w(e) + \sum_{e \in E_2} w(e)}$$

$$\arg\max_{m_n, m_e} \tag{2}$$

where $sim(x1, x2)$ is a function that represents with a score from 0 to 1 the similarity between two nodes or two edges.

The above formula formalizes that the similarity score is the weighted average of the similarities between the optimally mapped elements of the graph through the functions m_n and m_e. We also have to emphasize the importance of the weights of the nodes $(w(v)+w(m_n(v)))$ and edges $(w(e)+w(m_e(e)))$ in the calculation. Elements that have a larger weight or importance in the graph will contribute more to the global similarity score. This can be used to pinpoint the nodes that are more important within the context of the problem.

The minimization problem can be solved with any of the state-of-the-art techniques available like A* [42]. In our case we use hill climbing [43], which in comparison to A*, is a continuous optimization problem, reducing execution time at the expense of finding a local maximum instead of a global one.

At this point, we have to explain two elements present in the Equation 2: $sim(v, m_n(v))$ and $sim(e, m_e(e))$. In other words, the similarity between one node and its mapped node, and the similarity between one edge and its mapped edge. We detail the implementation of these comparisons in the following subsection. For quick reference, we also include the notation used in Table 1.

4.2. Similarity between nodes and edges

The solution for the optimal mapping found in Equation 2 depends on the similarity between nodes and edges. The similarity between two graph elements (edges or nodes) x_1 and x_2 is given by the following equation:

Table 1: Notation table

Notation	Description
G	a graph represented by a tuple $\{V, E, att, C, w\}$
V	the vertex of a graph
E	the edges of a graph
att	A function that returns a list of attributes a for a given $v \in V$ or $e \in$E
a	an attribute of a given v or e, represented by a tuple $\{val, weight, c \in C\}$
val	The value of an attribute
$weight$	The weight of an attribute inside the vertex or edge it belongs to
c	The context of an attribute. It is used to calculate the similarity between two attributes provided they have the same context
C	The context of the graph. It is used to explain the attributes that can be found in the graph and their different contexts c
w	A function that returns the weight of any given v or e within its graph, or the weight of an attribute inside its v or e
m_n	An injective function that matches the vertex of a graph G_1 to a vertex of a graph G_2
m_e	An injective function that matches the edge of a graph G_1 to an edge of a graph G_2
$sim(x1, x2)$	Function that calculates the similarity score between two vertex, edges or attributes

$$\frac{\sum_{a \in att(v) \cup att(u)} (w(a_1) + w(a_2)) \cdot sim(a_1, a_2)}{\sum_{a \in att(v) \cup att(u)} (w(a_1) + w(a_2))} \tag{3}$$

Where $sim(a_1, a_2)$ is the similarity between the attributes of the elements, which we will explain in the next Subsection 4.3. Similarly to Equation 2, this formula expresses the idea that the similarity between two elements is a weighted average between all of their shared attributes.

Analogously to Equation 2, the importance of the weights $w(a_1) + w(a_2)$ given to each attribute is used to modify the comparison calculation. Increasing the weight of an attribute means that the similarity of two nodes or two edges will increase even more if they have similar values for that attribute. An example of this effect can be seen in Figure 6, where giving a larger weight to an attribute, like the shape of the nodes, prioritizes shape similarity over structure similarity, since the matching node on the right neither has two edges nor is connected to a circle. This is an useful feature that enables including domain knowledge into

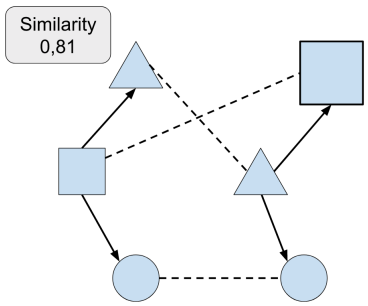

Figure 6: Effect of weights in the matching algorithm. Giving a bigger weight to the shape attribute of the nodes changes the matching in comparison to Figure 5. In this case the structure of the graph is not as important as matching nodes with the same shape.

the graph. An anomalous region where it is clear that the problem is the disk, may have the weights of all the disk-related attributes increased. When searching for patterns in the library, similar values for this metric will be prioritized.

4.3. Similarity between attributes

In this subsection, we explain in detail how we calculate $sim(a_1, a_2)$, in other words, how we compare the multi-type attributes of the elements in the graph. Here, the contexts that we defined in the graph structure (Equation 1) come into play.

When comparing elements, the context helps us to measure the similarity between their different attributes, like ontological concepts (e.g. comparing a backend to a frontend) or metrics that are bounded (e.g. cpu usage going from 0 to 100). The contexts are assigned by the user for each of the metrics or attributes that are used to build the graphs depending on their nature. The representation of this context C will be used in the different similarity calculations. If an attribute does not have a valid context, it will be ignored in the similarity calculation. In this paper, we consider three different types of contexts in our graphs:

Categorical Context: An attribute a_1 with a categorical context will have a label as its value (e.g. *kafka* representing the Docker image used by that container). When comparing two attributes with a categorical context, and since they normally represent labels or names, we use an exact equality function.

$$sim(a_1, a_2) = \begin{cases} 1 & \text{if } a_1 = a_2 \\ 0 & \text{otherwise} \end{cases}$$

Numerical Context: A numerical context is represented as a tuple $c_{numerical} = \{min, max\}$ where min and max represent the minimum and maximum value that an attribute with that context can take (e.g. % of cpu usage, which can range from 0 to 100). When comparing two attributes with a numerical context we use the function:

$$sim(a_1, a_2) = 1 - \frac{|a_1 - a_2|}{max - min} \tag{4}$$

Ontological Context: An ontological context is represented as a tree data structure, where each node represents a concept in the architecture. The goal of this type of context is to enable the comparison between two elements that are similar in nature (e.g. a loadbalancer and a distributed database master perform similar tasks: redirecting requests to a pool of backends) When comparing two attributes with an ontological context, we use a variation of the Wu and Palmer similarity metric [44]: let $c1$ and c_2 be two concepts in the ontology, and C be its closest common ancestor, then $sim(C1, C2) = \frac{2 \cdot d(C)}{d(C1)+d(C2)}$, where $d(x)$ is the number of nodes to traverse from the root of the taxonomy to concept x (we include always x in that number, thus $d(x) \geq 1$).

The advantage of this approach is that we can design new contexts for our attributes. This is also where the usefulness of contexts in the graph comparison engine resides. So far we included these three ones, but we could also create a new context for metrics with different comparison criteria. This is specially useful in the root cause analysis domain. For instance, a stressed system with a value of cpu of 100% compared to a value of 80%, can mean a lot more than just a 20% difference. We can also develop distribution-aware contexts, where attributes with values that fall outside a given distribution (e.g. outliers) will have more weight in the comparison result.

5. Monitoring and building the graphs

In Section 3, we described the architecture of the RCA framework and its different parts. So far, we have explained our own implementation of the anomalous region module in Section 3.2, the design of the pattern library module in Section 3.3 and the graph similarity function in Section 4. In this section, we explain the implementation details to build the system graph representation of the architecture through the system graph module. Our objective is to illustrate the extraction of the different attributes of the nodes and edges and how we define the connections between the different elements of the graph. For this task, we have used Sysdig[9] and Prometheus[10], but other monitoring tools can be used, as long as there is a plugin that transforms the input of metrics into a graph. In the following subsections we explain the technical details on how we build these graphs. Note that the whole pipeline of building the graphs and comparing them with the library of patterns can be used either in a batch fashion, using historical metrics of a system, or in a real-time fashion, ingesting incoming metrics from a continuous stream. As we will see later, we use the batch approach in our evaluation.

5.1. Adding the nodes

We start off by adding the nodes that will form part of the graph, i.e the elements that are alive in the system (containers and hosts). The size of the graphs depends mainly on the number of components involved in the architecture, as we will see later in the evaluation section.

We need to extract metrics at two different levels here: container level and host level. To extract these metrics, we run two containers in each machine: *node exporter*[11] and *cadvisor*[12]. The former will take care of monitoring the host and the latter the containers running on the different hosts.

As a back-end for the metrics collected we choose Prometheus. Prometheus is a monitoring and alerting system where metrics are pulled through HTTP from the monitoring agents, *cadvisor* and *node exporter* in our case. The metrics are stored following a multi-dimensional time series data model. This means that for each metric we will have a series of tuples like $(ts, id, value, labels)$, where ts is a millisecond-precision timestamp, id represents which container or host the metric belongs to, $value$ is the metric value (e.g. 100% for cpu usage) and $labels$ are a set of optional key-value pairs representing particular dimensions for that metric.

As an illustration, for the *node_cpu* metric we could have a data point like $(ts = 1518537794351, id = 10.136.1.9, value = 98.1, cpu = cpu3, mode = user)$, representing that the metric *node_cpu* at timestamp `1518537794351`, for the node `10.136.1.9` had a value of `98.1` in the core `cpu3` in `user` mode. This allows us to slice the data by different dimensions. Since our goal is to build a snapshot of the system for a given time window, we just have to use the time dimension ts to extract the metrics over one period. In our case, we want to extract all the available metrics for the container and hosts alive at every second. We chose this time window since the monitor agents that we use publish their metrics every second and it also gives us enough granularity to classify the different anomalies. Each distinct id is going to be added as a node in the graph and its information contained in the pairs of $(value, labels)$ inserted as attributes for that id.

[9] https://www.sysdig.org/
[10] https://prometheus.io/
[11] https://github.com/prometheus/node_exporter
[12] https://github.com/google/cadvisor

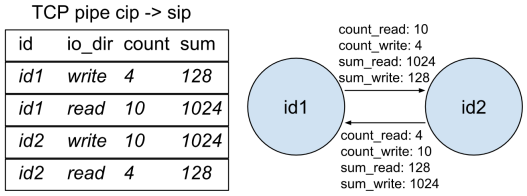

Figure 7: An example on how we group the different TCP requests traced by sysdig. Each group is going to have four entries for each client ip → server ip TCP pipe that will be added as attributes of the edges

5.2. Adding the edges

Once we have the nodes, we can start connecting them through their edges. To achieve that, we need to monitor the network communication inside the architecture. This is specially challenging since containers, unless told otherwise, use the IP of the host and its TCP port space, hindering the process of capturing traffic per container. Luckily, container orchestration tools facilitate the creation of overlay networks [45]. These networks create a separate network namespace, where each container will have its own IP and hostname. We use this feature together with Sysdig in each machine to capture any TCP communication between elements. Sysdig is an open source tool that instruments the Linux kernel and captures every system call and OS event. This trace of events can also be dumped into trace files in a very similar way to tools like tcpdump[13]. Sysdig is also able to differentiate these events per container out-of-the-box.

Ultimately, we will obtain some of the same dimensions from sysdig as those used to slice the Prometheus data in the previous subsection. In fact, and summarizing, for each TCP request we will have a tuple $(ts, id, cip, sip, io\_dir, bytes)$, where ts is the timestamp for that request, id is the container or host in which that communication happened, cip is the client IP, sip is the server IP, io_dir is whether it is reading or writing into the TCP socket and $bytes$ is the amount of information sent or read in bytes. Remember that sip and cip are unique among containers, thanks to the overlay network. Analogously to the Prometheus data processing, we use the ts value to extract all the TCP communications between containers for a given time window. Then we group all of the entries within that window by (cip, sip) and count the number of entries in each group and the sum of its $bytes$, representing the number of $TCPrequests$ and the size of the information sent respectively. As a result, each group is going to have the information sent through the TCP pipe $cip \rightarrow sip$ in the direction io_dir for two different $id1$ and $id2$ elements involved in that pipe. Remember that the ids for the elements were added as nodes in the previous step. Therefore, we just need to add an attributed edge from $id1$ to $id2$ and vice versa, where the attributes are going to be the previously mentioned count of $TCPrequests$ and the sum of $bytes$ in each io_dir. We depict an example of this process in Figure 7.

6. Evaluation

In this section, we evaluate the accuracy of the RCA framework and the effect of its different features, like its weights and thresholds. To do that, we create an experimental environment where we introduce anomalies in three different service-oriented scenarios common in industrial environments. We first explain the scenarios (Section 6.1), followed by the anomalies triggered in them (Section 6.2). Afterwards, we describe the set-up used to run these architectures in a real environment such as the Grid'5000 testbed [46] (Section 6.3). We conclude with an evaluation of the different features of the RCA method.

6.1. Service-oriented scenarios used

To test our RCA graph approach, we have chosen a set of architectures with several connected services. We chose to implement all these service-oriented architectures as Marathon application groups[14]. Marathon

[13]https://www.tcpdump.org/tcpdump_man.html
[14]https://mesosphere.github.io/marathon/docs/application-groups.html

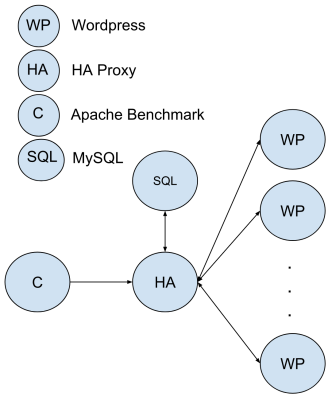

Figure 8: A simple representation of the loadbalancer scenario

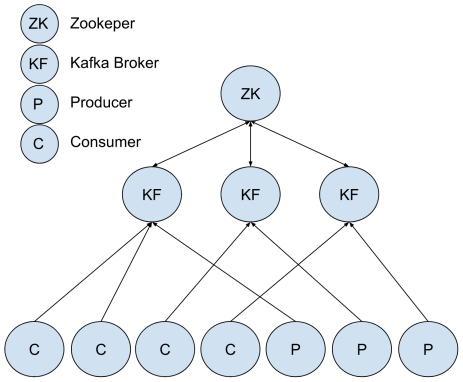

Figure 9: A simple representation of the Kafka scenario

is one of the central pieces of DC/OS[15], a container orchestration tool responsible for starting the containers in the different machines of our distributed infrastructure. Although they do not meet all the requirements of a microservice architecture, like fine grained responsibilities for each service or the large number of containers, the approach could be easily applied on them, as they also form a graph of connected components. We list here the different architectures designed:

Loadbalancer scenario: This is a common scenario used in production where a system receives a high number of HTTP requests that need to be balanced [47]. A loadbalancer (HAProxy) acts as a frontend, receiving HTTP requests from a series of clients represented as Apache Benchmark Docker images[16]. HAProxy balances these requests to a set of 16 WordPress instances that will return a simple blog website. WordPress needs a MySQL database, which is also going to sit behind the loadbalancer. A graphical representation of this setup is shown in Figure 8.

Kafka cluster scenario: Kafka is a distributed messaging system used to collect or send high amounts of data. It is commonly used in many Big Data stacks [48] and its architecture consists of a set of brokers that store the messages, producers sending those messages and consumers that pull the data from the brokers. There are no master nodes as the brokers are coordinated by means of a Zookeeper instance. We implement this architecture with 1 Zookeper instance, 4 Kafka broker containers, 12 producers, which are monitoring agents sending to the brokers system metrics of the host they are running in, and 12 consumers pulling those metrics from the Kafka brokers. A simplified depiction of these architecture is shown in Figure 9.

[15]https://dcos.io/
[16]https://httpd.apache.org/docs/2.4/programs/ab.html

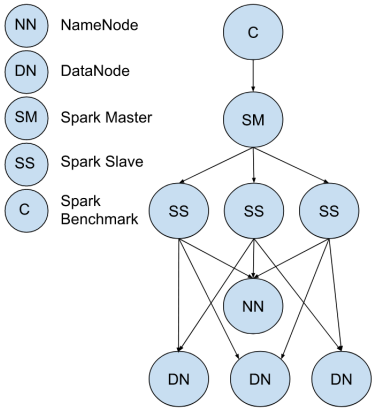

Figure 10: A simple representation of the HDFS + Spark scenario

Spark and HDFS scenario: Same as with Kafka, the combination of Spark and HDFS is a widely deployed architecture in Big Data infrastructures. Apache Spark [49] is an open-source engine for Big Data analytics, widely used in the community. HDFS [50] stands for Hadoop Distributed File System and is used to store large amounts of data using commodity hardware. Both of them have a master-slave architecture, with the master coordinating a series of worker nodes to process and storage data in parallel. In this case, we will have a highly interconnected graph structure with the Spark workers writing/reading data from the different HDFS datanodes and reading metadata from the namenode. Figure 10 represents this mesh of communicating containers. In our case, we will have 4 datanodes, 1 namenode, 4 Spark slaves and 1 Spark master. Additionally, we will use a container that has the SparkBench [51] benchmark, as a means to send data analytics jobs to the Spark master. In the experiments, we use a workload inside this benchmark that generates a dataset in a comma-separated format, writes it to disk and then reads it again to transform it to the Apache Parquet data format[17].

6.2. Anomalies triggered in the scenarios

In order to test the RCA technique, we need to generate different anomalies and evaluate if our system is able to distinguish their root causes. To accomplish this, we use several tools that allow us to stress and change the normal behaviour of the system:

Stressing hosts: *stress-ng* [18] is a tool that can stress several aspects and resources of a machine. Even if this anomaly is only applied directly in the hosts, containers running on those hosts will be affected if they use the same stressed resources. In this case, we will use four different types of stress tests:

- cpu: starts 6 workers that are going to exercise the cpu
- disk: starts 6 workers that will constantly write, read and remove temporary files
- network: starts 6 workers that are going to perform several connects, sends and receives on the localhost, simulating a socket stress
- bigheap: starts 1 worker that is going to grow its heap by constantly reallocating memory.

Connection problems: A drop in the bandwidth available for the machines or delays in communications are two possible problems in a distributed infrastructure [52]. This has a clear impact on performance, specially in data intensive applications, where there is a large amount of data exchanged. We also emulate this kind of conditions through the traffic control utility [19] (tc), a linux tool that allows manipulating the behaviour of the kernel when sending data through the network. Same as with *stress-ng*, this anomaly is only induced in the host, affecting the containers running on top. In our case, outgoing bandwidth is reduced to 75kbps and a delay of 100ms is introduced to all packets leaving that node.

[17] https://parquet.apache.org/
[18] http://kernel.ubuntu.com/ cking/stress-ng/
[19] https://linux.die.net/man/8/tc

Wrong loadbalancing: There are several causes that can force a loadbalancer to unevenly spread the workload [53]. We emulate this condition in the loadbalancer scenario, by login into the container and altering its HAProxy configuration files. By doing so, we overload all of the WordPress instances running on a given host, by redirecting 10 times more requests than the others.

Stressing endpoint: Endpoints can be flooded with requests, overloading the system and affecting the user experience. This can happen because of malicious attacks or just by peak demands [54]. We stress the endpoint of our loadbalancer scenario, by running a set of Apache Benchmark Docker[20] images with several client threads that perform a large number of HTTP requests.

6.3. Experiments set-up

As previously explained, we implemented our architectures in DC/OS. In order to deploy them, we configure a cluster in the Grid'5000 testbed. Grid'5000 provides access to a large amount of computing resources. It is highly customizable and offers a wide range of tools for reproducible experiments. We provision 8 VM's in the Rennes site with 16GB of RAM and 4 vcores. These machines will form a DC/OS cluster with 1 bootstrap node, 1 master node, 1 public node and 5 private nodes. Further information about DC/OS architecture can be found in their website[21].

To build a sequence of system graphs, representing the evolution of the cluster during our experiments, we do as follows:

1. We monitor six executions for each of the explained scenarios. During those executions, we inject the following anomalies:
   - The *stressing hosts* and *connection problems* anomalies for 8 seconds in two random machines of the cluster.
   - The *wrong load balancing* and *stressing endpoint* anomalies for 8 seconds in the HAProxy container for the loadbalancer scenario. For the former, we increase the load for all the WordPress instances that are running on a randomly selected host.
2. With the monitored metrics we build a set of system graphs for each second.
3. We tag the anomalous nodes corresponding to the anomalies we injected. This gives us a set of system graphs with their anomalous nodes.

This process corresponds to the system graph module explained in Figure 2, in which all the metrics and anomalies are gathered from the system in order to build graphs. The maximum size of the graphs built for these architectures are 45 nodes for the Spark scenario, 63 for Kafka and 59 for the loadbalancer. We need to take into account, that the anomaly extractor region's module will limit the anomalous region from all these nodes and edges to a smaller graph. This region will be the graph that will be compared with the ones in the library.

Remember that we assume the existence of an alerting system that is able to raise an anomaly based on the metrics gathered. This could be easily achieved with any anomaly detection technique since *stress-ng* and *tc* have a significant impact on the metrics at system level. In the wrong loadbalancing and stressing endpoint cases, the metrics can be extracted via the Unix socket of HAProxy which includes the number of requests balanced to each backend[22]. We monitor six different executions to have six folds of data and use four of them as a training set and the remaining two as an evaluation set. This setup is further explained in the next subsection. The anomalies are injected into two random nodes to create some variability in the anomalous graphs generated, since each machine is going to host different containers.

6.4. Precision of the system

The objective of this experiment is to evaluate the accuracy of our system when matching the different anomalous regions of the evaluation set into the patterns found in the training set. Remember that we have six folds of data corresponding to six executions in our experimental environment. We perform the evaluation as follows:

[20] https://httpd.apache.org/docs/2.4/programs/ab.html (Last accessed Dec 2018)
[21] https://docs.mesosphere.com/1.10/overview/concepts/ (last accessed Dec 2018)
[22] https://www.datadoghq.com/blog/how-to-collect-haproxy-metrics/

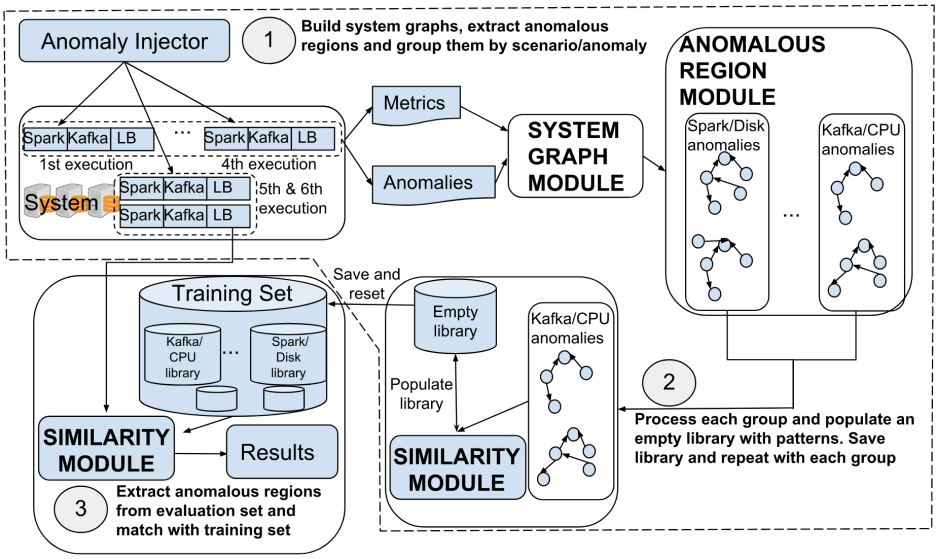

Figure 11: Design of the experiments to evaluate the RCA framework precision. There are two main steps. First, we process the first and second execution of each scenario to build its system graphs, generate its anomalous region and then, for each anomaly, generate a representative library of patterns for that problem. All of these representative libraries will be combined into one that will be used to establish the RCA of the anomalies induced in the third execution, based on the previously generated patterns.

1. For the first four executions, we take their system graphs, and group them by scenario and type of anomaly (e.g. spark cpu stress, kafka disk stress, load balancer stressing endpoint, etc...).
2. We process each of these groups with the RCA framework. In the process, the anomalous regions will be extracted. Remember that the anomalies are induced for 8 seconds in two random nodes for each scenario. That means 16 anomalous regions for each scenario/anomaly group, except for the *wrong load balancing* and *stressing endpoint* anomalies, with 8 anomalous regions, as they are induced in the single HAProxy container. We start with an empty library. The matching threshold is set to 0.9. This means that any anomalous region that is not similar up to 90% with a pattern, will be added to the library as a new pattern. This will generate a set of patterns for that scenario/anomaly group. Once the whole group is processed, we save the library, empty it and process the next group. The idea is to merge afterwards all of these saved libraries into a big library of patterns that will represent the training set, where each pattern is going to use the scenario/anomaly from where it originated as the classification label.
3. The graphs of the remaining two executions will constitute the evaluation set and are processed with the library populated during training. The threshold is lowered to 0 in order to avoid the addition of new patterns and forcing the system to always choose from the ones in the library. The RCA framework will match the different anomalous regions of these two executions with the patterns that were generated through points 1 and 2. From all the possible matched patterns we take the one with the highest similarity and assume the same root cause for the new situation.

We depict the whole process in Figure 11. Since we also want to test the previously explained weight system, we will run this experiment twice. In the first run, we do not tune the weights, while in the second run we tune the weights of certain attributes as an expert system admin would do.

Results are shown in Table 2 in a similar way to a normalised confusion matrix, where each anomalous region is classified with its corresponding scenario/anomaly label (the root cause). For the sake of space, we just use the first letter of the anomaly on the left column. In each cell there are two values, which represent the percentage of correctly classified anomalous regions: the one on the left is the result without tuning weights and the one on the right by tuning them. The sum of the values in each column for both the weights and no weights cells will be 100%. We consider that the distinction between the different scenarios (Kafka,

		KAFKA				SPARK				LB					
		CPU	DISK	BAND	HEAP	CPU	DISK	BAND	HEAP	CPU	DISK	BAND	HEAP	CONF	STRESS
KAFKA	C	**0.43/0.87**	0.12/-	0.03/-	0.18/-	-/0.125	-	-	-	-	-	-	-	-	-
	D	0.18/-	**0.71/0.81**	-	0.15/-	-	-/0.03	-	-	-	-	-	-	-	-
	B	0.03/-	-	**0.90/1**	0.15/0.06	-	-	-	-	-	-	-	-	-	-
	H	0.34/-	0.12/-	0.06/-	**0.5/0.71**	-	-	-	-/0.06	-	-	-	-	-	-
SPARK	C	-/0.12	-	-	-	**0.59/0.84**	-	0.21/-	0.15/-	-	-	-	-	-	-
	D	-	-/0.09	-	-	0.12/-	**0.25/0.87**	-	0.18/-	-	-	-	-	-	-
	B	-	-	-	-	-	0.18/-	**0.34/0.78**	0.06/-	-	-	-	-	-	-
	H	-	-	-	-/0.03	0.28/-	0.56/-	0.21/0.31	**0.59/0.59**	-	-	-	-	-	-
LB	C	-	-	-	-	-/0.03	-	-	-	**0.28/1**	0.28/-	-	0.21/-	-	-
	D	-	0.03/0.09	-	-	-	-/0.09	-	-	0.09/-	**0.12/1**	-	0.09/-	-	-
	B	-	-	-	-	-	-	0.18/0.124	-	0.06/-	-	**0.96/0.96**	0.31/0.28	-	-
	H	-	-	-	-/0.18	-	-	0.03/0.06	-/0.343	0.52/-	0.59/-	0.03/0.03	**0.37/0.71**	-	-
	C	-	-	-	-	-	-	-	-	-	-	-	-	**1/1**	-
	S	-	-	-	-	-	-	-	-	-	-	-	-	-	**1/1**

Table 2: Normalised confusion matrix for the graph based RCA method

Spark and loadbalancer) must be taken into account when performing root cause analysis, even when the cause of the anomaly is the same (cpu, disk, bandwidth or heap). We do so because the different elements involved in the anomaly could change the way to deal with the problem. For instance, even if our cpu stress anomaly is caused for the purpose of this experiment by *stress-ng* in all the scenarios, in a real set-up it could have different causes depending on the architecture and the containers running on that host (e.g. a Spark worker or a Kafka broker). Likewise, the prescriptive action can change, such as stopping non-critical containers, which also depends on the architecture. Therefore, this distinction between scenarios is needed.

There are several observations we can drawn from these results. The diagonal that crosses the table and that is highlighted in grey identifies the correct cells in which the classified anomalous regions should fall into. Any values not in the diagonal represent misclassified anomalous regions. For instance, without tuning weights 56% of Spark Disk anomalies are misclassified as Big Heap anomalies because in both of them there is a high number of bytes written to disk. We can observe that our RCA method is really effective with anomalies that have a significant effect on the elements around the anomalous node. For the BAND (*connection problems*), CONF (*wrong load balancing*) and STRESS (*stressing endpoint*) anomalies, the precision is near 100% without tuning any weights with the exception of the Spark scenario. In the BAND case, limiting the bandwidth of one machine has an effect on the performance of not only the containers running on that host, but also in all the containers in other machines that have to interact with them, creating a propagation effect that is easier to detect with a graph representation. For example, if one of the Kafka brokers has to send data to a consumer and its host has the upload bandwidth limited, the metrics for the whole application are going to be greatly affected. The STRESS and CONF anomalies show the same behaviour. Stressing an endpoint increases significantly the values in the HAProxy container for both the incoming/outgoing traffic attribute in the edges and the cpu usage of all the WordPress servers. Therefore the graph of the system is going to look very different to those of the other anomalies. A wrong load balancing problem means that some servers behind the frontend are going to receive a more significant part of the workload, resulting also in a distinctive graph pattern that the system is able to easily identify.

Another observation is that anomalies affecting a couple of host metrics, like DISK or CPU from the *stressing hosts* group, are difficult to match when weights are not properly tuned. We have to consider that from the whole anomalous region, only the attributes of one host in the graph change with respect to other anomalies. This is a very small part in the whole picture and makes it difficult to establish differences between a big heap problem (HEAP) and a CPU one affecting the same node, since they both intensively use the cpu. Here is where the expert knowledge comes into play in the form of weights. By increasing the weight of the anomalous host respect to other elements in the graph and pinpointing the important metrics of each anomaly in that host, the precision is 100% for some cases, like DISK and CPU for the load balancer scenario. In fact, from the 416 anomalous region generated, 349 are matched with the right pattern resulting in a precision of 83.89% overall when tuning the weights.

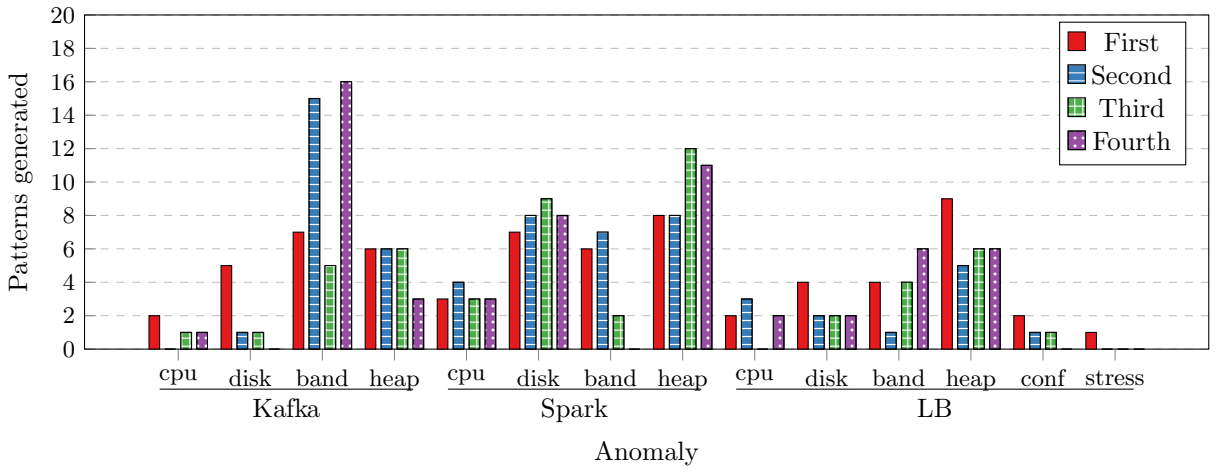

Figure 12: Number of patterns generated for the first, second, third and fourth execution of the training process

6.5. Number of patterns generated

With this experiment we want to measure the amount of different patterns that each anomaly generates. In Figure 12, we show the number of patterns that were generated for the first, second, third and fourth execution of the training phase with the 0.9 matching threshold. For space reasons, we show the results only for the run in which we tuned the weights, although we add a discussion of the results without weights at the end of this section. The number of patterns generated stays low specially for the load balancer scenario and most of the cpu anomalies. This means that the system is able to capture most of the information in the first executions. The *connection problems* and *big heap* anomalies are the ones generating the most number of patterns. The reason behind this is that both anomalies cause intermittent loss of connection between containers, which translates into missing edges from one timestamp to the next. This will create a lot of structural dissimilarity between graphs, hindering the matching with any of the patterns of the library. In fact, this effect can be seen on the bandwidth anomaly for Kafka, in which for the second and fourth execution a large number of patterns were generated since there were no matchings. In addition, for the big heap case, a lot of variability can be seen in different metrics as the anomaly evolves, accentuating this situation even more. The total number of patterns generated are 237 out of 832 anomalous regions that were processed. If we compare this with the execution without weights, 84 patterns are generated for Kafka, 150 for Spark and 70 for the load balancer scenarios. This is a total of 304 patterns and a difference of 67 patterns between the two approaches meaning that tuning the weights not only increases the precision of the classification method, but also reduces the size of the library.

6.6. The effect of thresholds

In the previous experiments we considered that a match between an anomalous region and a pattern in the library happens if the similarity is greater than 0.9. It seems logical that decreasing that threshold will translate into a smaller library of patterns, since the matching requirement is relaxed. However, precision could decrease, since we will have less variety of patterns in the library to match with. The advantage is that a smaller library will mean less patterns to label. To explore this tradeoff between precision and space, we try two different values for the matching threshold: 0.8 and 0.9. In Figure 13, we show the results for the precision as bars and the patterns generated as a line plot. In many cases, like in the CONF, DISK and some BAND anomalies, the higher threshold increases the precision, since we have more information captured in the library thanks to a wider set of patterns. On the other hand, having more patterns decreases the precision of BAND for the load balancer scenario. A finer tuning of the weights would be needed in this case, since some of these anomalous regions are matched with the numerous patterns for the Spark and Kafka scenarios. The maximum number of patterns generated is 43 for the *connection problems* in the

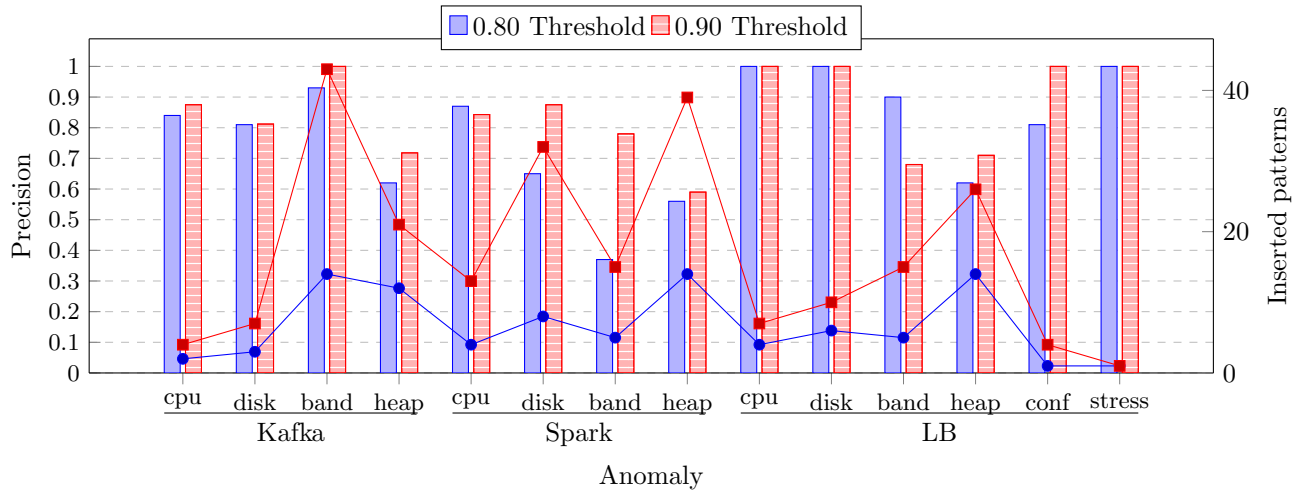

Figure 13: The effect of changing the matching threshold in the system. Precision is depicted as bars and the number of patterns generated as a line.

Kafka scenario. This may seem like an excessive number of cases to troubleshoot by an administrator, but we have to take into account that many of them are generated from graphs representing consecutive time windows. Therefore, the administrator could label the whole sequence of patterns with the same root cause without labeling them one by one.

6.7. Graph approach vs machine learning

We also want to evaluate the benefits of the graph approach, where the topology of the system and the elements around the anomaly are considered, versus a machine learning approach, in which a model is automatically learned to classify the state of an anomalous node into its root cause [14]. To achieve that, we train a boosted classification trees model with a dataset in which each row contains as features, the monitored metrics of a node when an anomaly was injected (cpu usage, bytes written to disk, bytes sent, etc...) and as classification label, the anomaly that was induced in that node. The machine learning algorithm will automatically find which metrics are correlated with each type of anomaly. Note that the difference between the two methods is that in the machine learning approach, we only focus on the metrics of the anomalous node in order to learn the anomaly type, while with our graph-based approach, we also consider the metrics and connections of the surrounding elements. Boosted classification trees were chosen because of their ability to work well with few data and their good accuracy [55]. The comparison between the graph approach and the machine learning one is shown in Figure 14. We also detail its results in a confusion matrix in Table 3. The precision of the graph approach is higher in most of the cases thanks to the weight system, where the expert knowledge embedded in the graphs and the extra information provided by the elements surrounding the anomaly are specially useful in anomalies like CONF or BAND. Classification trees outperform its graph counterpart only in the cpu stress cases, since the problem is well differentiated by the peak experienced in the *cpu usage* metric. The overall precision of the machine learning approach is 70.25%, while our graph-based approach has a precision of 83.89%, a difference of 13.64%. We can conclude that the graph approach, together with the input of the user in the form of weights, enables a more precise and fine-grained root cause analysis, specially in such diverse and interconnected systems such as service-oriented and microservice architectures.

6.8. Graph comparison latency

Latency is an important factor in providing a quick response to fix the anomaly and stabilize the system. In this section, we show the time needed to compare an anomalous region to a pattern in the library. In

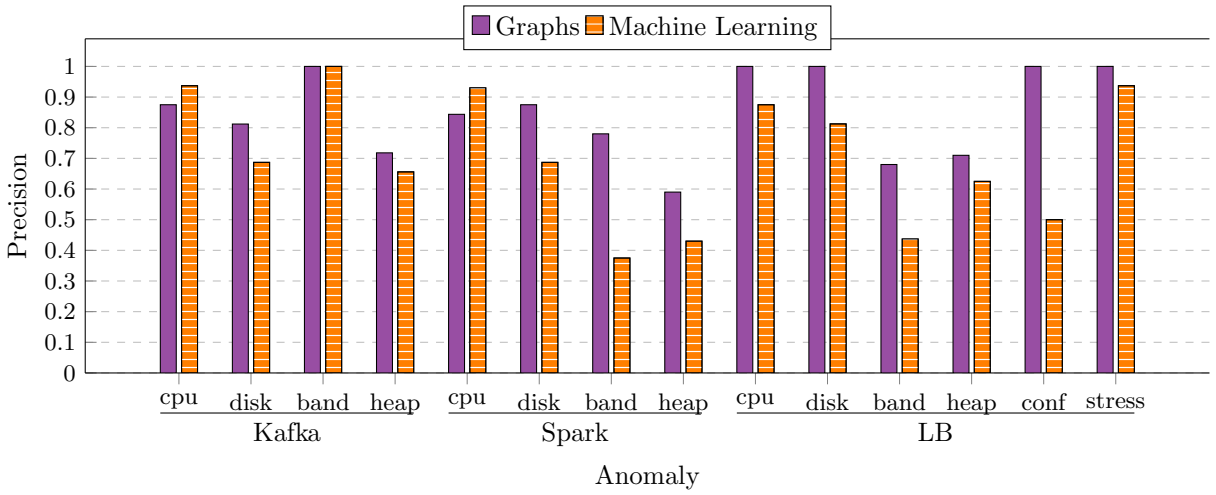

Figure 14: Precision of the graph-based approach compared to a machine learning based approach

		KAFKA				SPARK				LB					
		CPU	DISK	BAND	HEAP	CPU	DISK	BAND	HEAP	CPU	DISK	BAND	HEAP	CONF	STRESS
KAFKA	C	**0.93**	-	-	-	-	-	-	0.18	0.125	-	-	-	-	-
	D	-	**0.68**	-	-	-	-	-	-	-	-	-	-	-	-
	B	-	-	**1**	0.06	-	-	0.40	-	-	-	0.5	0.28	-	-
	H	-	-	-	**0.65**	-	-	-	-	-	-	-	-	-	-
SPARK	C	0.06	-	-	-	**0.93**	-	-	-	-	-	-	0.06	-	-
	D	-	0.25	-	-	-	**0.68**	-	-	-	0.18	-	-	-	-
	B	-	-	-	-	-	-	**0.37**	-	-	-	0.06	0.03	-	-
	H	-	-	-	-	-	-	-	**0.43**	-	-	-	-	-	-
LB	C	-	-	-	-	0.06	0.31	-	-	**0.875**	-	-	-	-	-
	D	-	0.06	-	-	-	-	0.06	-	-	**0.81**	-	-	-	-
	B	-	-	-	-	-	-	0.15	-	-	-	**0.43**	-	-	-
	H	-	-	-	0.28	-	-	-	0.37	-	-	-	**0.62**	-	-
	C	-	-	-	-	-	-	-	-	-	-	-	-	**0.5**	0.06
	S	-	-	-	-	-	-	-	-	-	-	-	-	0.5	**0.93**

Table 3: Normalised confusion matrix for the boosted decision trees RCA method

our experiment, we compare two randomly selected graphs with sizes ranging from 10, 15, 20, 25 and 30 nodes. The experiment is repeated 30 times for each size combination to calculate the mean and variance of the execution time. The results are shown in Figure 15. As expected, the comparison time depends on the number of elements that form part of the graph. The maximum execution time is 0.64 seconds for two graphs with 30 nodes each. Also the variance increases slightly as more nodes are involved in the process. Remember that the framework compares any incoming anomalous region with all the labeled patterns in the library, searching for the most similar one. This results in a sequential search, where a comparison needs to be done *ntimes*, where *ntimes* is the number of graphs in the library. As the library grows, establishing the root cause for that anomaly becomes more expensive. There are techniques that allow to index this search space [56] and we leave as future work their implementation to reduce the total search time in the library. In addition, the number of nodes and edges to be included as part of an anomalous region can be configured depending on the computational resources available, as explained in Section 3.2.

7. Threats to validity

In this section we cover the threats to validity that may arise from our experiments design. We analyse both internal and external threats regarding the accuracy results of our graph-based RCA approach. We focus on the experiment where we compared two different methods: our graph-based method and a different

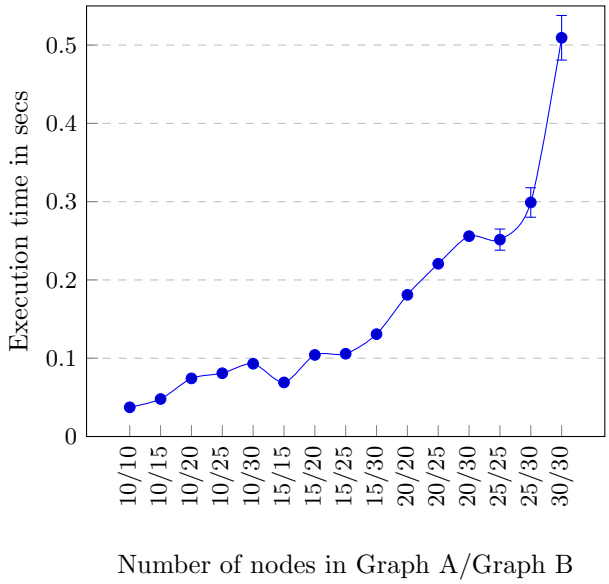

Figure 15: Execution time of the comparison between two graphs with different number of nodes.

one that does not take into account the connections between components and the elements surrounding the anomaly, like the machine learning approach.

7.1. Internal validity

In the context of our experiments, the internal validity refers to how confident we are regarding our graph-based approach being the true cause of the change in the accuracy with respect to other methods [57]. We observed how taking into account the metrics and connections of the surrounding elements, as our graph-approach does, improved the accuracy from 70,25% to 83,89%, in comparison to the machine learning one.

The metrics and events used for both the graph and machine learning approach come from the same experiment in Grid'5000, our chosen testbed. This eliminates many of the threats related to internal validity like:

- History: There were no events that happened in one group and not in the other or viceversa, since the two datasets were identical.
- Extraneous variables: There are no third variables involved, except for the approach to RCA itself.
- Selection biases: The train and test sets were both the same for the machine learning and graph-based approaches.
- Instrumentation: The metrics come from the same monitoring tool.

We can conclude then, that we have strong confidence in our experiments, from the internal validity point of view. The only difference between the machine learning-based results and the graph-based results is that the former do not have the metrics about the edges, but an average of them instead. However, this is the final goal of the experiment: proving that the inclusion of the full detail surrounding the anomaly improves the results of the root cause classification.

7.2. External validity

External validity refers to the degree to which the results can be extrapolated to another group of individuals. It can be divided into two:

- Population validity: It measures how representative is the sample of the population drawn for the experiment. We have included five service based architectures. The number seems good enough to prove that, in architectures where there are multiple connected components, the information about the surrounding elements is key. We would like to prove our approach in microservice architectures composed of a larger number of elements and we leave it as future work. The different anomalies used can also be considered another type of population that could have been enlarged in order to reassure our conclusions.
- Ecological validity: It measures how well these results would have been generalised in a different setting. This is probably the biggest threat to the validity of our experiments, since there are some factors that need to be considered in an environment other than the Grid'5000 testbed. First of all, our experiments were performed in an isolated environment, where all the processes running on the servers are included in the metrics we extracted. In a different environment there could be other factors affecting the resource usage of the hosts, like non monitored processes or occasional spikes that could distort the metric values over time. Regarding the first one, we assume that we have a monitoring system that is able to register the different processes running on the cluster, such as operating system processes, containers or non-container applications to name a few. For the second one, the immediate effect would be a larger library of patterns, as noise in the metrics means a wider range of possible values and more chances of inserting the anomalous region as a new pattern in the library.

8. Discussion

In this section, we discuss some of the advantages and weaknesses of the graph RCA method we presented throughout the paper. In particular, we focus on the advantages of a graph representation, the trade-off between methods that use expert knowledge and those which do not and the additional possibilities that our proposed framework can offer.

Advantages of a graph-based representation of microservice and service-based architectures: Graphs have been widely used to encode knowledge in a way computers understand and that is easily interpretable by humans [58]. Regarding the latter, these types of architectures often involve a great number of interconnected elements and searching for representations that facilitate the exploration of these systems is of vital importance. Many applications for container-observability are using this approach, like Netsil[23] or Sysdig[24], but only focus on the monitoring part. Incorporating a root cause analysis method as the one we presented on top of this graph representation can facilitate the decision making process when supporting complex architectures and ensuring their service level objectives. In addition, we have shown its potential in anomalies like a wrong load balancing configuration or under network bandwidth problems, where the problem is propagated to other elements of the system. As a limitation, we can say that the technique is maybe too complex for any anomaly in which the symptoms are isolated (e.g. a single node failure), while other RCA methods could find it in an easier way.

Incorporating expert knowledge in a RCA system: The current explosion of available data in many areas has brought machine learning techniques to the forefront. These techniques extract the knowledge from the data and do not need any external experts to make decisions, such as establishing a root cause. On the other hand, in expert systems or rule based systems, the knowledge comes from humans that have been exposed to some domain and have acquired expertise in the field. Decisions based on machine learning techniques can be hard to explain or not explainable at all. This is specially important in root cause analysis for critical production systems, where a wrong decision by an automatic system, can go unnoticed and result in great loses. Users need to understand the reasons behind a wrong prediction of the model, in order to prevent and avoid the same mistake in the future. Also machine learning systems need a considerable amount of data to discover the correlations in the metrics behind a given anomaly. This might not be possible in microservice architectures, where applications are developed at a rapid pace and new features

[23] https://netsil.com/
[24] https://www.sysdig.com/

continuously added. A system that is based on the input of an expert, who determines the important factors in an anomalous pattern and assigns labels to it, allows another user to understand the root cause and give valuable context when interpreting the prediction of the model. In our proposed system, this is present in the form of weights.

The potential of semantic graphs in RCA: Modelling the system and its elements as a graph opens several possibilities. One of them is using semantic graphs, which facilitate the incorporation of expert knowledge in the form of an ontology. This can be extended to the edges, with different relations between elements, like a machine *hosting* a container or a master *coordinating* several slaves. New contexts can be defined to specify the way these semantic concepts should be compared. It also enables new user-friendly methods to access the state of the system through natural language processing supported by those ontologies and the graph structure [59]. Questions like *"Which containers connected to the HAProxy instance have a cpu utilisation greater than 90%?"* can be used by the system administrator to troubleshoot an anomaly. We can also use it to define alerts with some semantic meaning and that are not only based on thresholds for certain metrics e.g. raise an alert when the number of requests balanced by HAProxy are not spread evenly. Current state-of-the-art monitoring solutions do not provide these set of tools that a graph based method could.

Using and tuning weights and threshold: Systems where many parameters have to be fine-tuned can be time-consuming. In our case, we have both weights and thresholds that need to be set by the user. However, there are several advantages for both. We have already talked about the weights and how they can be used to capture expert knowledge for a given anomaly, making the graph patterns more understandable. Thresholds can also be used to trigger decisions, based on the similarity score between graphs. For instance, in the scenario where the HAProxy balances the load unequally, we can trigger an action like resetting the configuration file to a default setting if the similarity between the anomalous behaviour and the wrong loadbalancing pattern is above 0.95. This triggering threshold will indicate that we must have a strong confidence in order to activate such an automatic action. Lower thresholds can be used for less intrusive actions, such as killing and resetting a container when its garbage collection time increases. From this point of view, our system can be seen as a decision support system that helps the user to troubleshoot and diagnose problems in microservice and service-oriented architectures.

9. Conclusion and future work

Throughout this paper, we have explained why a graph representation is a good fit to represent the state of microservice and service-based architectures and we have proposed a method based on graph similarity to perform root cause analysis on their anomalies. The evaluation shows how, through the incorporation of expert knowledge, we are able to accurately match past anomalous situation with current ones, facilitating decision making when maintaining the system.

We have also detected a set of challenges that we need to undertake in the future. Firstly, we need to consider the time dimension, where the evolution of the system during a time window can be compared, instead of single snapshots. This is specially important in a serverless environment, where containers come and go, depending on specific events. We have also noticed that the similarity between values of one type of metric must be calculated differently, depending on the distribution of historical data for that metric. Range of values that have been rarely seen before can be given different importance in the comparison function. Another concern is the computation time needed to compare two graphs. Several solutions have been proposed in the graph indexing field to reduce query times. Even though the comparison between graphs can be parallelised through a load balancer, we intend to implement one of these solutions to reduce the search space and consequently the response time of the system. Another option is to use graph embedding methods to transform the graph into a vector that can be fed into a machine learning model. However, this alternative eliminates the option of using user expert knowledge in the system such as the ontological context or the weigths. This reduction of the computation time is also related to the challenges of comparing large microservices architectures, as the responsibilities of the application are decomposed into even smaller units than the ones used in our evaluation. We would like to adapt our system to scale properly and also include this use case. Another line of work is to automatically tune the weights of the graph patterns. If

needed, these weights can be learned or fine-tuned with a properly labeled dataset of anomalous patterns through statistical techniques like machine learning. As in any other type of machine learning process that uses weights, we can use the ground truth (labels) to tune the weights and maximise the amount of correctly classified anomalies. We would also like to try out new scenarios and anomalies that are seen in Industry, in order to test the advantages of our model, specially for anomalies with a propagating effect such as the load balancing problem. Finally, we also need to consider the multi-user aspect of the system in terms of users assigning conflicting labels or being able to edit graphs concurrently.

Acknowledgment

This work is part of the BigStorage project, supported by the European Commission under the Marie Skłodowska-Curie Actions (H2020-MSCA-ITN-2014-642963) and it has been partly funded by the Spanish Ministry of Economy (grants TIN2014-62143-EXP, TIN2015-70799-R and TIN2016-78011-C4-4-R). Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

References

[1] A. Leung, A. Spyker, T. Bozarth, Titus: introducing containers to the netflix cloud, Communications of the ACM 61 (2) (2018) 38–45.
[2] D. Merkel, Docker: lightweight linux containers for consistent development and deployment, Linux Journal 2014 (239) (2014) 2.
[3] A. Balalaie, A. Heydarnoori, P. Jamshidi, Microservices architecture enables devops: Migration to a cloud-native architecture, IEEE Software 33 (3) (2016) 42–52.
[4] M. Endrei, J. Ang, A. Arsanjani, S. Chua, P. Comte, P. Krogdahl, M. Luo, T. Newling, Patterns: service-oriented architecture and web services, IBM Corporation, International Technical Support Organization, 2004.
[5] I. Baldini, P. Cheng, S. J. Fink, N. Mitchell, V. Muthusamy, R. Rabbah, P. Suter, O. Tardieu, The serverless trilemma: function composition for serverless computing, in: Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, ACM, 2017, pp. 89–103.
[6] A. Mouat, Using Docker: Developing and deploying software with containers, " O'Reilly Media, Inc.", 2015.
[7] W. E. Wong, R. Gao, Y. Li, R. Abreu, F. Wotawa, A survey on software fault localization, IEEE Transactions on Software Engineering PP (99) (2016) 1–1. doi:10.1109/TSE.2016.2521368.
[8] M. Solé, V. Muntés-Mulero, A. I. Rana, G. Estrada, Survey on models and techniques for root-cause analysis, arXiv preprint arXiv:1701.08546.
[9] M. Sampath, S. Member, R. Sengupta, S. Lafortune, K. Sinnamohideen, D. C. Teneketzis, Failure diagnosis using discrete-event models, IEEE Trans. Contr. Syst. Tech (1996) 105–124.
[10] K. R. Joshi, M. A. Hiltunen, W. H. Sanders, R. D. Schlichting, Automatic model-driven recovery in distributed systems, in: Reliable Distributed Systems, 2005. SRDS 2005. 24th IEEE Symposium on, IEEE, 2005, pp. 25–36.
[11] W. Mayer, M. Stumptner, Model-based debugging using multiple abstract models, arXiv preprint cs/0309030.
[12] W. Mayer, G. Friedrich, M. Stumptner, Diagnosis of service failures by trace analysis with partial knowledge, in: International Conference on Service-Oriented Computing, Springer, 2010, pp. 334–349.
[13] T. Sorsa, H. N. Koivo, Application of artificial neural networks in process fault diagnosis, Automatica 29 (4) (1993) 843–849.
[14] M. Demetgul, Fault diagnosis on production systems with support vector machine and decision trees algorithms, The International Journal of Advanced Manufacturing Technology 67 (9-12) (2013) 2183–2194.
[15] D. Koller, N. Friedman, Probabilistic graphical models: principles and techniques, MIT press, 2009.
[16] H. Kirsch, K. Kroschel, Applying bayesian networks to fault diagnosis, in: Control Applications, 1994., Proceedings of the Third IEEE Conference on, 1994, pp. 895–900 vol.2. doi:10.1109/CCA.1994.381203.
[17] J. Schoenfisch, C. Meilicke, J. von Stülpnagel, J. Ortmann, H. Stuckenschmidt, Root cause analysis in it infrastructures using ontologies and abduction in markov logic networks, Information Systems.
[18] L. Akoglu, H. Tong, D. Koutra, Graph based anomaly detection and description: a survey, Data Mining and Knowledge Discovery 29 (3) (2015) 626–688.
[19] M. A. Marvasti, A. V. Poghosyan, A. N. Harutyunyan, N. M. Grigoryan, An anomaly event correlation engine: Identifying root causes, bottlenecks, and black swans in it environments, VMware Technical Journal 2 (1) (2013) 35–45.
[20] C. Liu, X. Yan, H. Yu, J. Han, P. S. Yu, Mining behavior graphs for "backtrace" of noncrashing bugs, in: Proceedings of the 2005 SIAM International Conference on Data Mining, SIAM, 2005, pp. 286–297.
[21] L. Akoglu, D. H. Chau, J. Vreeken, N. Tatti, H. Tong, C. Faloutsos, Mining connection pathways for marked nodes in large graphs, in: Proceedings of the 2013 SIAM International Conference on Data Mining, SIAM, 2013, pp. 37–45.

[22] P. Foggia, G. Percannella, M. Vento, Graph matching and learning in pattern recognition in the last 10 years, International Journal of Pattern Recognition and Artificial Intelligence 28 (01) (2014) 1450001.
[23] D. HIDOVIĆ, M. Pelillo, Metrics for attributed graphs based on the maximal similarity common subgraph, International Journal of Pattern Recognition and Artificial Intelligence 18 (03) (2004) 299–313.
[24] K. Riesen, H. Bunke, Approximate graph edit distance computation by means of bipartite graph matching, Image and Vision computing 27 (7) (2009) 950–959.
[25] T. S. Caetano, J. J. McAuley, L. Cheng, Q. V. Le, A. J. Smola, Learning graph matching, IEEE transactions on pattern analysis and machine intelligence 31 (6) (2009) 1048–1058.
[26] H. Qiu, E. R. Hancock, Graph matching and clustering using spectral partitions, Pattern Recognition 39 (1) (2006) 22–34.
[27] Y. Zhu, L. Qin, J. X. Yu, Y. Ke, X. Lin, High efficiency and quality: large graphs matching, The VLDB Journal 22 (3) (2013) 345–368.
[28] M. Neuhaus, H. Bunke, Self-organizing maps for learning the edit costs in graph matching, IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics) 35 (3) (2005) 503–514.
[29] J. Gibert, E. Valveny, H. Bunke, Graph embedding in vector spaces by node attribute statistics, Pattern Recognition 45 (9) (2012) 3072–3083.
[30] H. Kashima, K. Tsuda, A. Inokuchi, Marginalized kernels between labeled graphs, in: Proceedings of the 20th international conference on machine learning (ICML-03), 2003, pp. 321–328.
[31] B. I. Ismail, E. M. Goortani, M. B. Ab Karim, W. M. Tat, S. Setapa, J. Y. Luke, O. H. Hoe, Evaluation of docker as edge computing platform, in: Open Systems (ICOS), 2015 IEEE Confernece on, IEEE, 2015, pp. 130–135.
[32] A. Pérez, G. Moltó, M. Caballer, A. Calatrava, Serverless computing for container-based architectures, Future Generation Computer Systems 83 (2018) 50–59.
[33] T. F. Düllmann, Performance anomaly detection in microservice architectures under continuous change, Master's thesis (2017).
[34] T. F. Düllmann, A. van Hoorn, Model-driven generation of microservice architectures for benchmarking performance and resilience engineering approaches, in: Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion, ACM, 2017, pp. 171–172.
[35] V. Heorhiadi, S. Rajagopalan, H. Jamjoom, M. K. Reiter, V. Sekar, Gremlin: systematic resilience testing of microservices, in: Distributed Computing Systems (ICDCS), 2016 IEEE 36th International Conference on, IEEE, 2016, pp. 57–66.
[36] F.-X. Aubet, M.-O. Pahl, S. Liebald, M. R. Norouzian, Graph-based anomaly detection for iot microservices, Measurements 120 (140) 160.
[37] B. Mayer, R. Weinreich, A dashboard for microservice monitoring and management, in: Software Architecture Workshops (ICSAW), 2017 IEEE International Conference on, IEEE, 2017, pp. 66–69.
[38] S. Haselböck, R. Weinreich, Decision guidance models for microservice monitoring, in: Software Architecture Workshops (ICSAW), 2017 IEEE International Conference on, IEEE, 2017, pp. 54–61.
[39] J. Thalheim, A. Rodrigues, I. E. Akkus, P. Bhatotia, R. Chen, B. Viswanath, L. Jiao, C. Fetzer, Sieve: Actionable insights from monitored metrics in microservices, arXiv preprint arXiv:1709.06686.
[40] A. Tseitlin, The antifragile organization, Communications of the ACM 56 (8) (2013) 40–44.
[41] V. Chandola, A. Banerjee, V. Kumar, Anomaly detection: A survey, ACM computing surveys (CSUR) 41 (3) (2009) 15.
[42] S. J. Russell, P. Norvig, J. F. Canny, J. M. Malik, D. D. Edwards, Artificial intelligence: a modern approach, Vol. 2, Prentice hall Upper Saddle River, 2003.
[43] J. Boyan, A. W. Moore, Learning evaluation functions to improve optimization by local search, Journal of Machine Learning Research 1 (Nov) (2000) 77–112.
[44] Z. Wu, M. Palmer, Verbs semantics and lexical selection, in: Proceedings of the 32nd annual meeting on Association for Computational Linguistics, Association for Computational Linguistics, 1994, pp. 133–138.
[45] V. Marmol, R. Jnagal, T. Hockin, Networking in containers and container clusters, Proceedings of netdev 0.1, February.
[46] D. Balouek, A. Carpen Amarie, G. Charrier, F. Desprez, E. Jeannot, E. Jeanvoine, A. Lèbre, D. Margery, N. Niclausse, L. Nussbaum, O. Richard, C. Pérez, F. Quesnel, C. Rohr, L. Sarzyniec, Adding virtualization capabilities to the Grid'5000 testbed, in: I. Ivanov, M. Sinderen, F. Leymann, T. Shan (Eds.), Cloud Computing and Services Science, Vol. 367 of Communications in Computer and Information Science, Springer International Publishing, 2013, pp. 3–20. doi:10.1007/978-3-319-04519-1_1.
[47] V. Cardellini, M. Colajanni, P. S. Yu, Dynamic load balancing on web-server systems, IEEE Internet computing 3 (3) (1999) 28–39.
[48] R. Estrada, I. Ruiz, Big data smack a guide to apache spark, mesos, akka, cassandra, and kafka (2016), Tech. rep., ISBN 978-1-4842-2175-4 (2016).
[49] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, I. Stoica, Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing, in: Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, USENIX Association, 2012, pp. 2–2.
[50] K. Shvachko, H. Kuang, S. Radia, R. Chansler, The hadoop distributed file system, in: Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on, Ieee, 2010, pp. 1–10.
[51] M. Li, J. Tan, Y. Wang, L. Zhang, V. Salapura, Sparkbench: a comprehensive benchmarking suite for in memory data analytic platform spark, in: Proceedings of the 12th ACM International Conference on Computing Frontiers, ACM, 2015, p. 53.
[52] J. T. Piao, J. Yan, A network-aware virtual machine placement and migration approach in cloud computing, in: Grid and Cooperative Computing (GCC), 2010 9th International Conference on, IEEE, 2010, pp. 87–92.
[53] B. Radojević, M. Žagar, Analysis of issues with load balancing algorithms in hosted (cloud) environments, in: MIPRO,

2011 Proceedings of the 34th International Convention, IEEE, 2011, pp. 416–420.
[54] H. Liu, S. Wee, Web server farm in the cloud: Performance evaluation and dynamic architecture, in: IEEE International Conference on Cloud Computing, Springer, 2009, pp. 369–380.
[55] J. Elith, J. R. Leathwick, T. Hastie, A working guide to boosted regression trees, Journal of Animal Ecology 77 (4) (2008) 802–813.
[56] S. Berretti, A. Del Bimbo, E. Vicario, Efficient matching and indexing of graph models in content-based retrieval, IEEE Transactions on Pattern Analysis and Machine Intelligence 23 (10) (2001) 1089–1105.
[57] M. de Oliveira Barros, A. C. Dias-Neto, 0006/2011-threats to validity in search-based software engineering empirical studies, RelaTe-DIA 5 (1).
[58] M. Chein, M.-L. Mugnier, Graph-based knowledge representation: computational foundations of conceptual graphs, Springer Science & Business Media, 2008.
[59] V. Tablan, D. Damljanovic, K. Bontcheva, A natural language query interface to structured information, in: European Semantic Web Conference, Springer, 2008, pp. 361–375.