Using Distributed Tracing to Identify Inefficient Resources Composition in Cloud Applications

Clément Cassé, Pascal Berthou, Philippe Owezarski, Sébastien Josset

Using Distributed Tracing to Identify Inefficient Resources Composition in Cloud Applications

Clément Cassé
LAAS - CNRS, Université de Toulouse, CNRS,
Orange,
Toulouse, France
clement.casse@laas.fr

Pascal Berthou
LAAS - CNRS, Université de Toulouse, UPS,
Toulouse, France
pascal.berthou@laas.fr

Philippe Owezarski
LAAS - CNRS, Université de Toulouse, CNRS,
Toulouse, France
philippe.owezarski@laas.fr

Sébastien Josset
Orange,
Toulouse, France
sebastien.josset@orange.com

***Abstract*—Cloud-Applications are the new industry standard way of designing Web-Applications. With Cloud Computing, Applications are usually designed as microservices, and developers can take advantage of thousands of such existing microservices, involving several hundred of cross-component communications on different physical resources.**

Microservices orchestration (as Kubernetes) is an automatic process, which manages each component lifecycle, and notably their allocation on the different resources of the cloud infrastructure. Whereas such automatic cloud technologies ease development and deployment, they nevertheless obscure debugging and performance analysis. In order to gain insight on the composition of services, distributed tracing recently emerged as a way to get the decomposition of the activity of each component within a cloud infrastructure. This paper aims at providing methodologies and tools (leveraging state-of-the-art tracing) for getting a wider view of application behaviours, especially focusing on application performance assessment.

In this paper, we focus on using distributed traces and allocation information from microservices to model their dependencies as a hierarchical property graph. By applying graph rewriting operations, we managed to project and filter communications observed between microservices at higher abstraction layers like the machine nodes, the zones or regions. Finally, in this paper we propose an implementation of the model running on a microservices shopping application deployed on a Zonal Kubernetes cluster monitored by *OpenTelemetry* traces. We propose using the flow hierarchy metric on the graph model to pinpoint cycles that reveal inefficient resource composition inducing possible performance issues and economic waste.

***Index Terms*—Distributed Tracing, Cloud Computing, Property Graph, Graph Rewriting, Hierarchical Model**

I. Introduction

Nowadays Cloud-Applications have become the industry-standard way of designing Web-Application running at global scale. Back then, in 2008, the National Institute of Standards and Technology (NIST) defined the term "Cloud Computing" as a ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources [1]. The impact of this new paradigm on software architecture has been massive: Applications are now divided in a multitude of components, running on a numerous virtual machines scattered into data centres around the world. These processes communicate with each other over the network through API calls (often designated as Remote Procedure Calls).

Separating an application into business-centric components is the foundation of the Microservices approach [2]–[4]. The number of services involved in an application can grow up to thousands as evidenced by recent communications from prominent Cloud companies. An engineering blog post from Uber reported that the number of critical microservices composing their application was about 2200 [4]. Also, in [5], Google mentioned that the order of magnitude of RPC involved in a single-user request in an application like Gmail can go from tens to thousands. With this significant increase of network communications involved in the application, the impact of the internal network on the global application response time become a key metric to monitor.

This shift in design has a direct implication on the way monitoring should be done, and what events / metrics should be reported to characterize the overall application performance. By adopting a distributed architecture, Cloud Applications also face a new range of problems like components misconfiguration, cascading errors, hot points / bottlenecks or even noisy neighbours [6], [7].

In the last years, we observed various initiatives introducing a new kind of software monitoring tools related to tracing service composition in Cloud environments. These new tools are often designated as Observability tools; in particular, they focus on creating a visualization of the propagation of a request within a cloud application. This technique, named Distributed-Tracing, has been implemented by most of the major actors of Cloud-Computing for their own monitoring needs [8]–[11]. They reported how they are using traces to detect performance anomalies [9], [11], [12], but also for other scenarios involving a high observability need like running tests in production [5], [13]. Unlike metrics gathering or logging, distributed tracing provides a unified view of the

propagation of a request in a distributed system, crossing the boundaries of its components. This propagation is called a trace and establishes causality between latency measurements of each application components. Traces are displayed as a Gantt chart of the time spent in each component involved in the request. The prevailing approach for building distributed traces is to aggregate measurements is the Google Dapper's *span model* [9]. It has numerous open source derivative implementations, OpenZipkin[1], OpenTracing[2], OpenCensus[3] to quote a few. A recent open source initiative, *OpenTelemetry*, merged the two most mature technologies: *OpenTracing* and *OpenCensus*; this project has a high visibility as it aims to normalize monitoring for Cloud Applications. This project, in Beta version at the time of writing, acts as an element part of the pipeline that uniforms tracing data emitted from the various existing implementations.

In this paper, a focus is put on a way to exploit tracing data from *OpenTelemetry* in order to spot inefficient communication within a Cloud application. Indeed, Cloud orchestrator, like Kubernetes, assign workload to physical resources with an allocation algorithm that do not consider communication between microservices. To illustrate this problem, we will focus on a *Zonal Kubernetes Clusters*. Kubernetes is one of the most popular Cloud Orchestrator, its role is to allocate containers on multiple machines and make them communicate through an overlay network. Once it allocates a Pod (a set of containers) to a machine (called a Node), it does not question this decision any more in the pod lifecycle. A pod is allocated to a machine if that machine has enough memory and CPU resources available. In *Zonal clusters*, nodes are scattered on different "availability zones", this allows handling failure more reliably but has an extra cost as node to node communications across zones are added to the bill. To effectively reduce the bill, having a resource placement that considers the communications across services, Nodes and Zones is required.

Given these constraints of reduced latency and costs, this work aims to exhibit a new generic model and metrics based on the communication monitoring that may help to get a location-aware placement. In the next section, we provide motivating examples where tracing helped to improve a cloud application performance. Then, we propose a generic methodology and a toolset to create a hierarchical property graph leveraged by state-of-the-art cloud monitoring tools. This hierarchical model is motivated by the current trend of pushing computations to the edge, and provides a representation of the overlay network linking microservices as a layered network instead of a flat one. The following section introduces the flow hierarchy metric to detect inefficient composition on the hierarchical model. Finally, this paper concludes with a deployment of this model on a Zonal Kubernetes cluster to illustrate the detection of costly communications.

[1]https://zipkin.io
[2]https://opentracing.io
[3]https://opencensus.io

II. Background

A. Cloud Application Specific Performance Issues

At the scale of a globally used application, made of hundreds of services geographically distributed on multiple data centres, optimizing performance involves minimizing network latency while keeping the utilization of data centres as low as possible. Facebook Engineering published various papers where they detail how they used Traffic Engineering to preserve the balance between latency and data centre utilization.

In [14], authors detail a solution that manages the traffic generated by users into a geographically distributed application. Their custom traffic management improves hardware usage in production by 20%. In another paper [15], authors raised the problem of congestion and bottleneck links in Cloud-Application. They address these problems with the implementation of a routing algorithm dedicated to service-to-service communications, balancing network calls in a more efficient way. In addition, *Maelstrom* [16] applies traffic engineering techniques to disaster mitigation and recovery. Finally, in [17], autors present *Taiji*, an application-level load balancer that pushes more in depth the routing and placement of computing resources to the edge. In this contribution, adding application-level parameters to the routing algorithm reduces the load of back-end servers by 17%.

These various publications made on a real large-scale application (Facebook) highlight the importance of a smarter in-app request routing can greatly improve a resource management in a multi data centre cloud. However, methods used in these publications are not generic and result of years of engineering based on the specificities of the Facebook application. Now, when developping a Cloud-Native Application, Kubernetes acts as a base-component. It abstracts the physical network to creates a flat overlay network for containers. These internal communications hold an important role in global application performance but are often overlooked by both developers and operations. This overlay network undergoes the same challenges for traffic management, in particular when deploying an application on multiple data centres.

B. Monitoring Initiatives for Cloud Native Applications

Whereas monitoring tools tend to become more and more exhaustive in the way they describe Cloud environments, cloud orchestrators and providers, on the contrary, tend to obscure underlying implementations, making debugging more difficult [18]. The recent *OpenTelemetry* initiative aims to normalize Cloud monitoring by providing an open format and production-ready binaries for Cloud-Native Monitoring. While the project is still in Beta at the time of writing, it opens many research opportunities to enhance the quality of tracing data or the way it is processed [19]–[24]

The goal of this paper is to exhibit inefficient allocations made by Kubernetes in the case of a geographically distributed cluster. We use a *Zonal Kubernetes Cluster* as a motivating example for our study. In the later section, we discuss how we leveraged *OpenTelemetry* semantic to create a hierarchical property graph model from tracing data.

III. Modelling Internal Communications Based on Traces

A. Getting Tracing Data With Network Level Measurements

Usually tracing data comes from the code of the application, however, only relying on in-app instrumentation to get traces do not provide the full picture, as it lacks networking data. In order to observe cross services network calls and to cover the latency introduced by services, Kubernetes has been expanded with a Service-Mesh [25]. A Service-Mesh is a Data-Plane made of L4/L7 proxies injected in each microservice to better control and observe service-to-service communications. The configuration of these proxies is made through a control plane that ensures that their configuration is coherent. For the purpose of our experiment, the *Linkerd*[4] service mesh has been used as it is compatible with OpenCensus format (and thus, has a backward compatibility with current OpenTelemetry Beta binaries). In a Kubernetes environment, Linkerd automatically injects and configures HTTP proxies between each microservices instances to allow communications to be observed (through traces) or controlled (through HTTP proxies configuration).

To ensure all the required attributes are set, and the OpenTelemetry semantic is fully respected, another agent is also added to each microservice. This agent captures and reformat tracing data shards (called *Spans*) sent by Linkerd proxies to make them compliant with the OpenTelemetry semantic. For our experimentation, this agent acts as a temporary "hack" and permitted to have a Kubernetes Cloud Application emitting OpenTelemetry formated traces spans. This agent then forwards the trace spans reported by proxies to a central OpenTelemetry Collector.

Finally, OpenTelemetry traces are made available through a Web aplication named Jaeger Tracing, which allow to visualize and query OpenTelemetry traces. Jaeger Tracing is one of the most popular interfaces to display and manage traces: it provides a Web UI but also a gRPC endpoint to query and compute this data. Jaeger gRPC endpoint provides a binary data flow to process traces online.

B. Creating a Property Graph

When considering traces, they appear to be more than a collection of data objects living in a multi-dimensional space independently. The core of this data resides in the interdependencies expressed between latency measurements. Property graphs provide a powerful machinery to represent these traces; indeed, as a graph, relationships are as important as the data represented in nodes. Furthermore, adding the capability to have labels and attributes on both nodes and edges allows the preservation of the original data semantic. There is existing work considering every trace as a Directed Acyclic Graph (DAG) of spans linked by causal relationship [19], [21]. However, considering every trace as independent graphs does not allow a wider view of the application, or the exhibition of observations correlation from other traces. In this work, we consider all traces as a single graph decomposing tracing data into multiple vertices and edges merged with previously observed data, thus establishing correlation over multiple traces.

In *OpenTelemetry* various concepts have been defined: the concept of **Span** holds the description of a latency measurement at a precise point in time. It comes with a variety of attributes aiming to provide an exhaustive description of the action measured. The concept of a **resource** is also defined; it characterizes the executor of the measure. It comes with a variety of attributes that describe the process on which the measurement has been done. In our model, resources are used as common vertices shared by multiple traces. Resources attributes allow the identification of the different abstractions layers involved in the latency measurement: we assume we can identify Kubernetes **Pods**, **Nodes**, **Zones** and **Clusters**.

These terms come from Kubernetes Domain Specific Language (DSL) and describe some abstraction levels involved in representing resource location.

Pods
: They represent the smallest Kubernetes schedulable unit, they are a collection of heavily tighted containers sharing the same network namespace and colocated on the same machine.

Nodes
: In Kubernetes nodes represent machines (either virtual or physical), they are the kernel shared by the containers and are characterized by a number of CPUs and an amount of memory.

Zones
: They are concepts coming from Zonal Kubernetes Clusters, they are often characterizing independent availability areas; Applications developers seeking high availability deploy an application over multiple Availability Zones.

Clusters
: They represent Kubernetes centralized control plane. This level has been defined as it is now common to deploy an application over multiple clusters that are often geographically distributed.

All these concepts follow a containment hierarchy. Indeed Pods are allocated to Nodes, Nodes to Zones, and Zones belong to a Cluster. Whereas these concepts are specific to Kubernetes, the model presented in this paper only considers them as entities part a containment hierarchy [26], [27]. The process described in the following works for every elements part of a predefined containment hierarchy that can be matched from tracing data.

In Figure 1, an example of this transformation is provided where a trace made of seven spans is decomposed in a graph highlighting the resources involved: four Pods scattered on two Nodes. For the sake of clarity of the graph, both Zones and Clusters have been omitted.

This property graph is created by connecting to the Jaeger gRPC endpoint and retrieving data. This data is then processed in order to match a model encoded as a graph following the

[4]https://linkerd.io/ a service mesh for Kubernetes hosted by CNCF

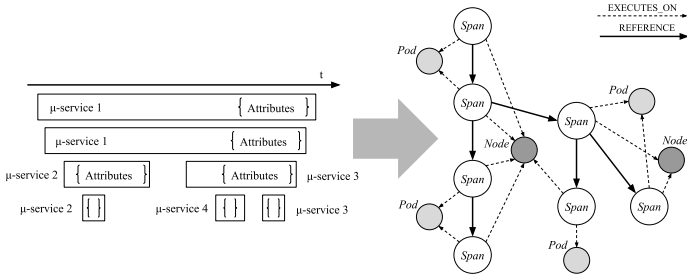

Fig. 1. Transforming a Trace in a Property Graph.

schema described in Figure 2. This meta-model represents two kinds of hierarchies observed in traces, each of this hierarchy is materialized by edges in the graph with different labels.

The first hierarchy represented is the most commonly used to describe traces: according to OpenTelemetry specifications [28, a Trace can be thought of as a DAG of Spans, where the edges between Spans are defined as parent/child relationship]. In this graph-model, the edges labelled REFERENCE are created from the field "reference" from span that points to another span in the same trace.

The second hierarchy expressed in the model corresponds to the containment hierarchy defined earlier for resources. We can define a hierarchical order of service location within a Zonal Kubernetes Cluster by having the following orders of abstraction levels: Pods $\subset$ Nodes $\subset$ Zones $\subset$ Clusters. In the proposed graph-model, this hierarchy materializes as edges labelled IS_CONTAINED that appear between resource vertices. When multiple traces have been processed, traversing the graph of resources by following the IS_CONTAINED relationship will form a pure tree.

Finally edge labelled EXECUTES_ON links a span vertex to their resource vertices (Pod, Node, Zone, Cluster). When processing multiple traces, all spans characterizing the execution of an action in the same Pod will be linked to the same Pod Vertex.

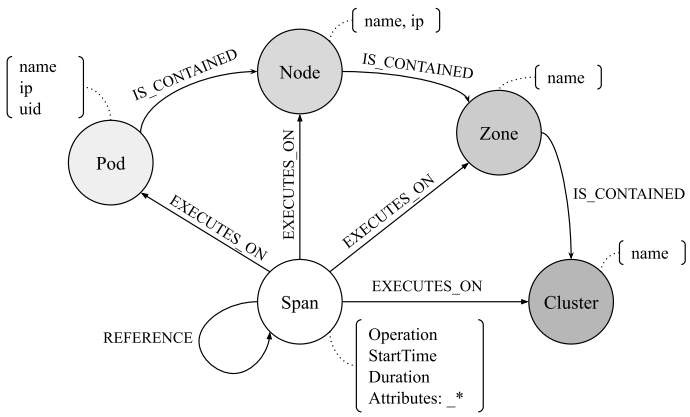

Fig. 2. Transforming a Trace in a Property Graph.

C. Deducing Resources Dependencies

The property graph offers valuable pieces of information regarding how service composition may impact the resources they are executed on. Indeed, a dependency between two resources may be deduced from the observation of dependency in a trace, through the REFERENCE relationship. However the graph scale baldy when ingesting numerous traces as Span vertices accumulates into the graph making it increasingly complex to analyse. A graph rewriting approach has been taken to keep the number of vertices low and to express dependencies observed between two microservices to the level of their respected resources. The rewriting process aims to delete Span vertices and project the REFERENCE relationship to each abstraction level of the containment hierarchy. The rewriting process deletes numerous vertices and edges and adds a new edge labelled PROJECTED_REF.

This graph transformation is formulated with pushout transformations, which are based on concepts from the category theory. There are two main variants of pushouts operations that may be applied to graphs. Both provide a synthetic description graph rewriting, but they differ in particular, in the way they handle suspended edges. As the rewriting process deletes some vertices, there is a chance that it will leave some edges with one of its ends not linked to any vertices, this what is called suspended edges.

The two approaches used for graph rewriting:

- Single Pushout: it allows to add, delete, merge or clone vertices or edges in an attributed graph, deleting any suspended edges.
- Double Pushout: it allows to add, delete, merge or clone vertices or edges in an attributed graph, but blocks if any suspended edges is encountered.

For this purpose, a Single Pushout is convenient to delete Span Vertices and EXECUTES_ON and REFERENCE edges once the PROJECTED_REF has been created. Figure 3 provides a Single Pushout formalization of the rewriting approach and takes the same conventions as in Figure 1. The first line shows on the left side of the operand the pattern that will be searched into the graph, and on the right side the result of the rewriting process (the creation of the new edge) applied on this pattern. On the second line, the left part of the operand is the graph from trace in Figure 1 (with only pods as resources for the sake of clarity), and on the right part the result of the graph rewriting. After the rewriting process, no nodes labelled *Span* remains nor edges labelled EXECUTES_ON. The final graph is only made of resources vertices with IS_CONTAINED and PROJECTED_DEP edges.

With this rewriting approach, we project REFERENCEs observed between spans to any kind of resources present in our tracing data. As a result, for each trace, we build the network topology containing only the network calls of each abstraction layers, of the containment hierarchy. In our example, the constructed graph (right DAG) exhibits the communications between the pods that do not appear in the property graph model (left DAG).

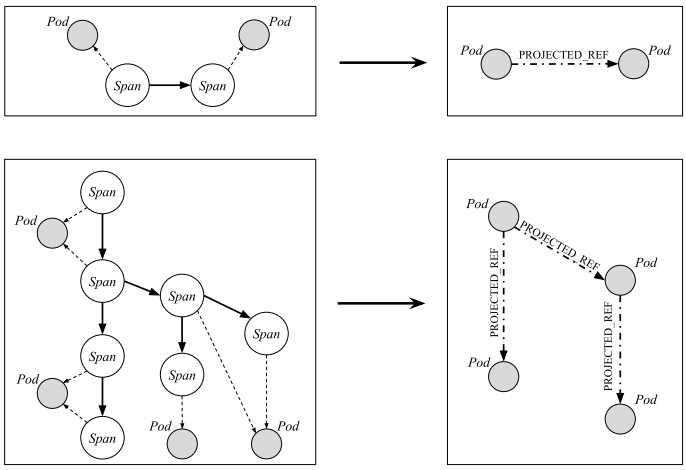

Fig. 3. Graph Rewriting Approach to deduce resources dependencies (applied to Pods) based on a Simple Pushout operation.

D. Graph Hierarchical Model

After the rewriting process, the property graph is not a hierarchical structure by itself. By defining a hierarchy as a DAG whose nodes are graphs and edges are morphisms between elements of these graphs, we can consider our model to match this definition. With this model, we can maintain the view of the different topologies of resources; each of them corresponding to a level of this hierarchy. The hierarchical relationship expressed by the model is the IS_CONTAINED resources relationship: *Pods* $\subset$ *Nodes* $\subset$ *Zones* $\subset$ *Clusters*. In this designation *Pods* is a graph where its vertices are labelled *Pod* and its edges are typed PROJECTED_REFs. The same applies to *Nodes*, *Zones* and *Clusters*, each being a graph of vertices respectively labelled *Node*, *Zone* and *Cluster* linked by PROJECTED_REFs. Figure 4 is a visual representation of a portion of this graph involving the "Pods" and "Nodes" hierarchical vertices where the different network topologies have been reconstructed based on a trace.

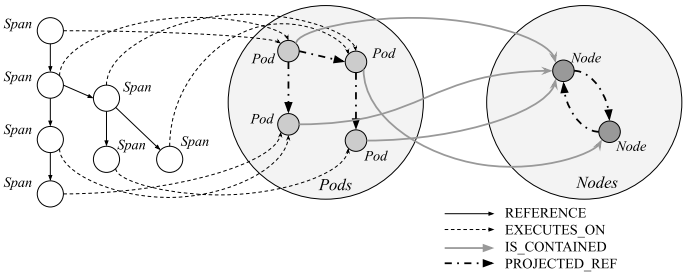

Fig. 4. Hierarchical Graph Representation.

As a result, traces, which are flat graphs where abstraction levels are hidden, have been turned in a multi-level location-aware model that highlights the composition of service and resources.

IV. Detecting Inefficient Communications in Clouds

When projecting the references expressed in traces to upper layers, which originally was a DAG, the new projected graph may exhibit cycles, e.g. Figure 4 shows this kind of configuration where the two nodes within the Nodes Hierarchical layer initiate a communication with each other for a single trace. These cycles show an inefficient placement of services regarding their composition. In this section we focus on the flow hierarchy that may appear among each level of the containment hierarchy. In [27], the flow hierarchy concept is associated with directed networks of vertices of the same entity. Vertices are layered by their influence on each other: higher level vertices influence lower level vertices; the influence is materialized by a relationship between vertices. Authors propose a metric of Flow hierarchy which detects and measure the extent to which all the local flows follow a holistic overall "underlying direction". It may also be defined as the fraction of nodes not involved in a cycle.

A. Using the flow hierarchy metric to assess service composition regarding their placement

To assess resource placement regarding a trace, the key idea is to compute the flow hierarchy metric h in all levels of the containment hierarchy. If a resource topology has cycles, its h metric will be lesser than 1 otherwise it will be 1. As the network of resources does not have numerical attributes on edges, we will consider the following definition of the flow hierarchy:

$$h = \frac{\sum_{i=1}^{L} e_i}{L} \tag{1}$$

where L is the number of PROJECTED_REF in the network and $e_i = 0$ if the PROJECTED_REF relationship i belongs to a cycle or $e_i = 1$ otherwise. Cycle detection is not covered by this formula and is a prerequisit to compute the flow hierarchy metric. In the next section, two methods for identifying cycles in the graph are discussed.

In our case, having cycles within the topology for a single trace spots unnecessary network calls, and thus reveals an inefficient placement of the underlying resources involved in the cycle, and performance degradations. Also, the rewriting process can lead to a projected graph made of a single vertex with no edges, e.g. when all network communications remains contained within the same availability zone. In that case $L = |E| = 0$ and then h is undefined. However, for the purpose of our model, this use case materializes a normal case where network communications are efficient. Therefore when $|E| = 0$ we define $h = 1$.

B. Example

To illustrate a use case where the calculation of the flow hierarchy metric brings valuable feedback regarding the allocation of resources, we will use the example trace we have been using throughout this section. Let's consider the case of Pod load-balancing: Figure 5 represents two traces representing the

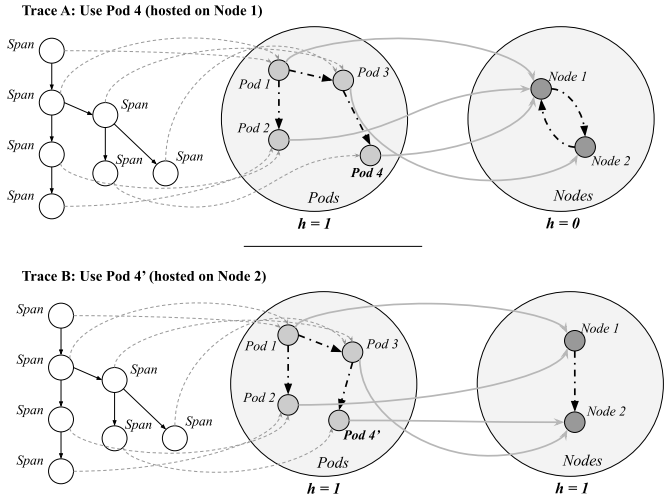

Fig. 5. Examples of Flow hierarchy metric calculation at each layer of the containment hierarchy for two traces

same service composition, but using different resources. In this example Pod 4 has two instances: *Pod 4* and *Pod 4'*; the first one is hosted on Node 1 and the second one on Node 2. Whereas one trace involves the creation of a cycle at the Nodes level the other does not.

V. Implementation and Experimentation

To verify our approach a Proof of Concept (PoC) platform has been developed integrating Linkerd service mesh with an early release of the OpenTelemetry service for trace formatting. This platform has been deployed on a Zonal Kubernetes Cluster, in order to get more depth in the hierarchy representing resource location. Finally, a demonstration application has been deployed on that cluster with an integrated load generator that emulates incoming user requests. The deployment manifests of this platform have been published on GitHub [29].

Both the application own instrumentation and the service mesh proxies have been configured to send traces to a central Jaeger Tracing instance. In the next section, we will provide more details on the implementation of both the pipeline processing traces and of the PoC application that was used for experiments.

A. Creating a Pipeline for Trace Processing

Traces managed by Jaeger Tracing are processed via the *Polynote*[5] data-processing platform. Polynote is a Notebook engine capable of executing code written in Scala; all the steps described in the graph modelling section have been implemented in Scala by using a functional programming approach. We created, in the notebook engine, an real-time processing pipeline for parallel trace computations. This pipeline covers: reading data from a Jaeger gRPC endpoint, applying the model to create graphs, then applying the rewritting process on the graph to generate a hierarchical structure and finally storing this model in a graph database after computations.

The choice of the Scala programming language was further motivated by its compatibility with the Java ecosystem that has a wide set of libraries available. Data retrieval was implemented based on the work published in the *Jaeger Data Analytics Library*[6]; the capability of matching the various resources type has been added in order to create resource vertices in the model. The graph rewriting process has been implemented with the Gremlin language [30] on Tinkerpop In-Memory graphs. In-Memory graphs make computations faster, which is required for the identification of cycles that the flow hierarchy metric depends on. In addition, Tinkerpop provides a solid implementation for processing property graphs. To identify cycles, we took an approach based on the identification of Strongly Connected Components (SCC), as an edge is in a cycle if and only if it is in a strongly connected component. The Tarjan algorithm [31] is in general a good solution to identify SCC because of its low complexity. It has a complexity of $\mathcal{O}(|V| + |E|)$, where $|V|$ represents the number of vertices in the graph and $|E|$ is the number of edges. We favoured this method over the one described in original work on the flow hierarchy metric [27] that was based on exponentiation of the adjacency matrix which has a complexity of $\mathcal{O}(|E|^{|V|})$.

Finally, to store the hierarchical model, a Neo4J database was used. This graph database has a clean and powerful synthax to create or reuse vertices from the database without prior checks. Our graph hierarchical model requires to identify resources vertices already present in the model when stored in the graph database. Using this backend to store the hierarchical model reduced the number of network calls to the graph backend by a factor of three.

This approach is then scalable and allows online trace processing; indeed, all the heavy graph computation and rewriting is done independently for each trace. Only the results of these computations are stored on a graph stored in a Neo4J database. In the current implementation, the model is dependent on Kubernetes as it has the best support for distributed tracing; however the resource containment model may also be used to describe edge computing structure.

B. Test Application Overview

In order to get representative tracing data, a microservices Cloud application was needed. In this paper, a PoC platform has been created by implementing OpenTelemetry data collection on the sample Cloud application provided by Google Cloud for demonstration purpose [32]. This application is made of ten microservices communicating with each other over gRPC and coded in five different languages. Some services have part of their code base instrumented to emit traces spans based on the time spent in some functions; some have no instrumentation. An OpenTelemetry agent has been injected in each of the micro-services in order to convert traces

[5]https://polynote.org A Scala Notebook engine open sourced by Netflix

[6]https://github.com/jaegertracing/jaeger-analytics-java

to a common format and to ensure the common attributes have been set. This agent forwards trace spans to a Jaeger Tracing Collector; this setup builds a complete tracing pipeline representative of production setups.

This application emulates the behaviour of an online boutique. User can basically do five different operations on this web application: (1) *Consult the catalogue* of products: this operation will trigger the execution of five services, (2) *Consult the page of a specific product*: this operation will trigger the execution of six microservices (3) *Consult the cart*: this operation also triggers the execution of six microservices, (4) *change the currency of the boutique*: this operation only triggers one microservice and finally (5) proceed to the *checkout*: this operation involves nine out of the ten services in the application.

Throughout the rest of this section, we will only focus on the *checkout* operation which provides the most complex graph of service composition. As this application follows a common pattern in software architecture (the *API gateway* pattern) the dependencies of these microservices may be represented as a star graph. As we might not observe cycle in a star shaped graph most common actions of the application are not traced. Only the checkout operation does not create a star-shaped graph dependency and adds extra depth to the composition of services.

C. Deployment

Trace spans from *checkout* operation are sent to OpenTelemetry agents and are merged with code-level trace spans into the final trace. With this particular configuration, a trace mixes monitoring measurements from two abstraction layers: the code instrumentation and the network instrumentation. Whereas this exhaustive view is a clear advantage for debugging purpose, it saturates the content of the traces and makes the processing of traces more computation intensive. When applying the rewriting operation, the process automatically extracts the network-related spans and discards the ones focused on code instrumentation.

Our ten microservices application has been deployed on a Zonal Kubernetes Cluster made of four nodes (named $Node_1$, $Node_2$, $Node_3$ and $Node_4$) scattered in two zones (named $Zone_1$ and $Zone_2$). We have $Node_1 \subset Zone_1$, $Node_2 \subset Zone_1$, $Node_3 \subset Zone_2$ and $Node_4 \subset Zone_2$.

With this platform, the application has been deployed with four replicas of each service, each on a different node following the previous topology. The load simulator emulates two concurrent users using the application at the same time.

D. Experimentation

To verify our approach we executed the model on a Zonal Kubernetes Cluster which is, typically, a representative implementation of a hierarchical network model. Our goal was to assess whether Kubernetes allocation could produce bad allocation, or bad load balancing between instances. We applied our model to each *checkout* trace hosted in Jaeger Tracing to build a tuple of three flow hierarchy metrics: $(h_{Pod}, h_{Node}, h_{Zone})$.

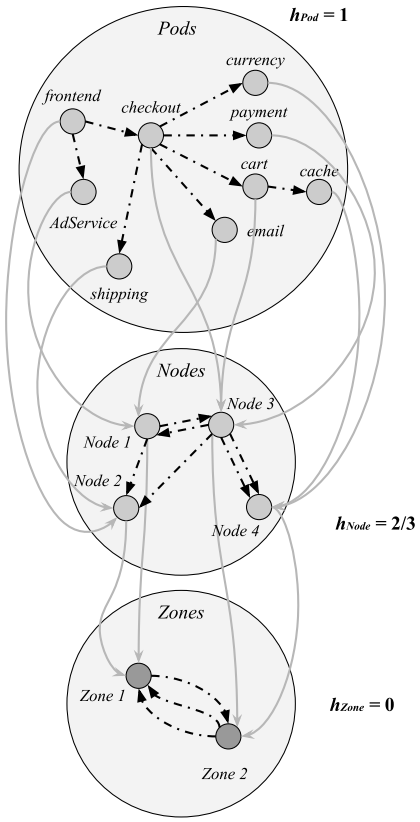

Fig. 6. Graph transformation for a particular trace

In Figure 6, the graph transformation process has been unrolled, showing the three stages of the location hierarchy. Each stage has the flow hierarchy metric provided, calculated as the percentage of edges not involved in a cycle. In that case, the load balancer picked the *email* service hosted on *Node 1*. This decision introduces a cycle between *Nodes* and then between *Zones*. However, by routing the communication initiated by *checkout* to the *shipping* service hosted on *Node 2*, the load balancer introduced a cycle not visible at the *Nodes* hierarchy level. The cycle generated by this costly communication is shown at the *Zones* Level.

When running the model on a flow of traces, we observed that 70% of the traces created a cycle between the two Zones of the cluster. This cycles introduce a communication that could be avoided by having a better placement and communications between zones, as they represent an economic waste. Therefore, there is room for improvement in Kubernetes internal routing so that they can manage a more effective routing between zones. And, to a wider extent, manage a more complex network structure for the pod overlay network.

CONCLUSION

OpenTelemetry is a fast-growing technology in the cloud ecosystem with a high visibility among most prominent industrial like Google, Microsoft or Uber. It provides both an open

format and an implementation to Cloud-Native Monitoring, in particular for tracing. Still, distributed tracing is a young technology and there are few usages of traces today. They are used individually for debugging purpose, but state-of-the-art tools do not provide a wider view of the system. In this contribution we propose a generic model fed by traces that maintain at runtime a hierarchical property graph. This model, based on the semantic defined by *OpenTelemetry*, represents the microservices ecosystem of a Cloud Native Application. Most notably, it highlights service composition which is a topic that is not covered by previous monitoring techniques. With this contribution, we also provide an implementation of this model fed by the most popular tracing tool supported by *OpenTelemetry*: Jaeger Tracing. And finally, we propose a usage of this model by using the flow hierarchy metric at each abstraction layer in order to identify an inefficient placement of resources by Kubernetes. For our experiment we focussed on an existing use case of identification of inefficient communications within a zonal cluster that increase the Cloud bill (cycles in service composition). Note that, both the model and its usage are generic and can adapt to more depth within a placement hierarchy.

REFERENCES

[1] P. Mell and T. Grance, "The NIST Definition of Cloud Computing Recommendations of the National Institute of Standards and Technology," *Natl. Inst. Stand. Technol. Inf. Technol. Lab.*, vol. 145, p. 7, 2011. [Online]. Available: http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf

[2] M. Fowler and J. Lewis, "Microservices, a definition of this new architectural term," 2014. [Online]. Available: https://martinfowler.com/articles/microservices.html

[3] S. Newman, *Building Microservices*. O'Reilly Media, Inc., 2015.

[4] A. Gluck, "Introducing Domain-Oriented Microservice Architecture," 2020. [Online]. Available: https://eng.uber.com/microservice-architecture/

[5] D. Ardelean, A. Diwan, and C. Erdman, *Performance analysis of cloud applications*. USENIX Association, nov 2018. [Online]. Available: https://www.usenix.org/conference/nsdi18/presentation/ardelean

[6] S. C. Kendall, J. Waldo, A. Wollrath, and G. Wyant, "A Note on Distributed Computing," Sun Microsystems, Inc., Tech. Rep., 1994.

[7] O. Ibidunmoye, F. Hernández-Rodriguez, and E. Elmroth, "Performance Anomaly Detection and Bottleneck Identification," *ACM Comput. Surv.*, vol. 48, no. 1, pp. 1–35, 2015. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2808687.2791120

[8] J. Kaldor, J. Mace, M. Bejda, E. Gao, W. Kuropatwa, J. O'Neill, K. W. Ong, B. Schaller, P. Shan, B. Viscomi, V. Venkataraman, K. Veeraraghavan, and Y. J. Song, "Canopy: An End-to-End Performance Tracing And Analysis System," *SOSP 2017 - Proc. 26th ACM Symp. Oper. Syst. Princ.*, pp. 34–50, 2017.

[9] B. H. Sigelman, L. Andr, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper , a Large-Scale Distributed Systems Tracing Infrastructure," *Google Res.*, no. April, p. 14, 2010.

[10] Y. Shkuro, "Evolving Distributed Tracing at Uber Engineering," 2017. [Online]. Available: https://eng.uber.com/distributed-tracing/

[11] Twitter, "Distributed Systems Tracing with Zipkin," 2012. [Online]. Available: https://blog.twitter.com/engineering/en_us/a/2012/distributed-systems-tracing-with-zipkin.html

[12] W. Li, "Anomaly Detection in Zipkin Trace Data," 2018. [Online]. Available: https://engineering.salesforce.com/anomaly-detection-in-zipkin-trace-data-87c8a2ded8a1

[13] A. Gud, "Testing in Production at Scale," in *SRECon19 _ Am. (USENIX Assoc.* Brooklyn, NY: USENIX Association, 2019.

[14] K. Veeraraghavan, J. Meza, D. Chou, W. Kim, S. Margulis, S. Michelson, R. Nishtala, D. Obenshain, and D. Perelman, "Kraken : Leveraging Live Traffic Tests to Identify and Resolve Resource Utilization Bottlenecks in Large Scale Web Services This paper is included in the Proceedings of the," *OSDI'16 Proc. 12th USENIX Conf. Oper. Syst. Des. Implement.*, pp. 635–651, 2016.

[15] P. Kumar, Y. Yuan, C. Yu, N. Foster, R. Kleinberg, P. Lapukhov, C. L. Lim, and R. Soulé, "Semi-Oblivious Traffic Engineering : The Road Not Taken," *15th USENIX Symp. Networked Syst. Des. Implement. (NSDI 18)*, pp. 157–170, 2018. [Online]. Available: https://www.usenix.org/conference/nsdi18/presentation/kumar

[16] K. Veeraraghavan, J. Meza, S. Michelson, S. Panneerselvam, A. Gyori, D. Chou, S. Margulis, D. Obenshain, S. Padmanabha, A. Shah, Y. J. Song, and T. Xu, *Maelstrom: Mitigating Datacenter-level Disasters by Draining Interdependent Traffic Safely and Efficiently*, 2018. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/veeraraghavan

[17] D. Chou, T. Xu, K. Veeraraghavan, A. Newell, S. Margulis, and L. Xiao, "Taiji : Managing Global User Traffic for Large-Scale Internet Services at the Edge," *SOSP '19 Proc. 27th ACM Symp. Oper. Syst. Princ.*, pp. 430–446, 2019.

[18] H. Jayathilaka, C. Krintz, and R. Wolski, "Performance Monitoring and Root Cause Analysis for Cloud-hosted Web Applications," *Proc. 26th Int. Conf. World Wide Web - WWW '17*, pp. 469–478, 2017. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3038912.3052649

[19] J. Mace, R. Roelke, and R. Fonseca, "Pivot Tracing," *ACM Trans. Comput. Syst.*, vol. 35, no. 4, pp. 1–28, 2018.

[20] J. Mace and R. Fonseca, "Universal context propagation for distributed system instrumentation," in *Proc. Thirteen. EuroSys Conf.* New York, NY, USA: ACM, apr 2018, pp. 1–18. [Online]. Available: https://dl.acm.org/doi/10.1145/3190508.3190526

[21] P. Las-Casas, J. Mace, D. Guedes, and R. Fonseca, "Weighted Sampling of Execution Traces," in *Proc. ACM Symp. Cloud Comput. - SoCC '18*, 2018, pp. 326–332. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3267809.3267841

[22] P. Las-Casas, G. Papakerashvili, V. Anand, and J. Mace, "Sifter: Scalable Sampling for Distributed Traces, without Feature Engineering," *SoCC 2019 - Proc. ACM Symp. Cloud Comput.*, pp. 312–324, 2019.

[23] S. Nedelkoski, J. Cardoso, and O. Kao, "Anomaly detection and classification using distributed tracing and deep learning," *Proc. - 19th IEEE/ACM Int. Symp. Clust. Cloud Grid Comput. CCGrid 2019*, pp. 241–250, 2019.

[24] V. Anand, M. Stolet, T. Davidson, I. Beschastnikh, T. Munzner, and J. Mace, "Aggregate-Driven Trace Visualizations for Performance Debugging," *CoRR*, oct 2020. [Online]. Available: http://arxiv.org/abs/2010.13681

[25] W. Li, Y. Lemieux, J. Gao, Z. Zhao, and Y. Han, "Service Mesh: Challenges, state of the art, and future research opportunities," *Proc. - 13th IEEE Int. Conf. Serv. Syst. Eng. SOSE 2019, 10th Int. Work. Jt. Cloud Comput. JCC 2019 2019 IEEE Int. Work. Cloud Comput. Robot. Syst. CCRS 2019*, pp. 122–127, 2019.

[26] A. Zafeiris and T. Vicsek, *Why We Live in Hierarchies?*, ser. SpringerBriefs in Complexity. Cham: Springer International Publishing, 2018, no. July. [Online]. Available: http://link.springer.com/10.1007/978-3-319-70483-8_7

[27] J. Luo and C. L. Magee, "Detecting evolving patterns of self-organizing networks by flow hierarchy measurement," *Complexity*, vol. 16, no. 6, pp. 53–61, jul 2011. [Online]. Available: http://doi.wiley.com/10.1002/cplx.20368

[28] "OpenTelemetry Specification Overview." [Online]. Available: https://github.com/open-telemetry/opentelemetry-specification/blob/v1.0.1/specification/overview.md

[29] C. Cassé, "Itinerix Project PoC platform," 2020. [Online]. Available: https://github.com/clement-casse/itinerix-project/

[30] M. A. Rodriguez, "The gremlin graph traversal machine and language (Invited Talk)," *DBPL 2015 - Proc. 15th Symp. Database Program. Lang.*, pp. 1–10, 2015.

[31] R. Tarjan, "Depth-First Search and Linear Graph Algorithms," *SIAM J. Comput.*, vol. 1, no. 2, pp. 146–160, jun 1972. [Online]. Available: http://epubs.siam.org/doi/10.1137/0201010

[32] GoogleCloudPlatform, "GoogleCloudPlatform/microservices-demo," 2020. [Online]. Available: https://github.com/GoogleCloudPlatform/microservices-demo