A Tracing Based Model to Identify Bottlenecks in Physically Distributed Applications

Clément Cassé*†, Pascal Berthou*, Philippe Owezarski* and Sébastien Josset†
*LAAS - CNRS, Université de Toulouse, CNRS, UPS, Toulouse, France
Email: {clement.casse,pascal.berthou,philippe.owezarski}@laas.fr
†Orange Labs, Blagnac, France

***Abstract*—The Cloud computing paradigm has become the new industry standard way of designing large scale applications. Over the past years, we observe an increased adoption of this technology on numerous IoT - Edge applications. And while this technology comes with its promises and benefits, considering almost infinite scalability, it also comes along with its drawbacks and challenges. Detecting partial failures or bottlenecks are new obstacles that arose with the adoption of Cloud Applications. Distributed Tracing now allows developers to gain insight on the composition of services within a distributed Application. Today we observe an increased adoption of this technology on numerous cloud-native architectures. The project OpenTelemetry proposes a specification for traces that normalizes this new monitoring data format. In this publication we present an approach that leverages these traces to identify bottlenecks at the scale of a physically distributed application. We propose an extension of our model that builds a hierarchical property graph to exhibit bottlenecks in an application that follows the layered Cloud - IoT network model. Based on OpenTelemetry traces we can maintain a model at runtime of the whole application and compute bottlenecks. Their identification relies on the scores provided by centrality algorithms.**

***Index Terms*—Distributed Tracing, Multi-zones Cloud, Edge-IoT, Property Graph, Graph Rewriting, Hierarchical Model**

I. Introduction

The Cloud Computing has become the industry-standard way of designing large-scale applications. Initially, Cloud Service Providers (CSP) designed a catalogue of services capable of addressing the constraints that rose on Web applications design over the last decade. Although, the Cloud-Native design also spread out to multiple other domains such as Big Data processing pipelines or IoT applications. Indeed, Cloud-Native software architectures are well known to be both scalable and resilient. Many companies reported successful stories on the capacity of their architecture to deliver the service under an extremely high load or under poor conditions.

The way Cloud-Native Applications are built represents a major shift in the way software is developed nowadays. Applications hosted on such platforms have a new model; they are now made of components managed by third parties, hosted in multiple data centres distributed around the world. Cloud Native Applications are usually designed as microservices: the application is separated into business-centric components, communicating with each other via Web APIs. This shift in design has a direct implication on the way monitoring should be done; and, what events / metrics should be reported to characterize overall application performance. These profound structural changes raised several new challenges to Application Performance Monitor (APM):

Adopt the heavily distributed architecture paradigm: Cloud Applications can be made of hundreds or even thousands of components communicating with each other. In the microservices model, each component can be implemented in the language that best fits its functional needs. CSPs now propose multi data centres deployments as well as Edge features that allow an application to be geographically scattered on multiple continents [1], [2].

Handle changes at runtime: Cloud orchestrators can handle automatically the scalability of computing resources: additional resources may be created or deleted to follow the variation of the load generated by users. In addition, it is a common pattern to have ephemeral components in a Cloud Application: recurring tasks are often performed in dedicated computation units [3]. Finally, in the DevOps culture, for each new version of a component, a fully automated pipeline runs a cycle of tests and then deploys the software in production (CI/CD) [4].

Handle various instrumentation levels: Unlike traditional software design, Cloud Application heavily rely on third-party software-defined-services [5]. Whereas it eases development, it also obscures both monitoring and debugging [6], [7]. Indeed, CSP rarely expose their internal metrics; performance of these third-party services is evaluated regarding the Service Level Agreement (SLA) with the client.

Cloud Application monitoring borrows many concepts of distributed system monitoring where elements of the system have different life-cycles and report different metrics. With the environment heterogeneity, each service performance may be evaluated through a different set of metrics; although single component monitoring is a well-known and well-studied problem. But monitoring an application only by monitoring its individual components does not highlight global issues. Indeed, in a distributed system, there is a range of issues specific to their nature: Identifying cascading errors, bottlenecks and noisy neighbours, which are relative to the way component communicate with each other.

In this paper, we propose an approach based on the recent open source initiative OpenTelemetry to detect bottlenecks. The following section provides an history on the use of distributed tracing in cloud applications. Then we present our model and how to derive a hierarchical property graph from OpenTelemetry traces. In a later section, we focus on the use of centrality algorithms on the graph to exhibit bottlenecks. Finally, this paper concludes on running centrality algorithms on a simulation of a distributed application inspired by the Riak database.

II. Background

In the last years, we observed various initiatives introducing a new kind of software monitoring tools related to tracing service composition in Cloud environments. This technique has been named Distributed-Tracing, it grants a unified view of components interactions serving a single request. Distributed tracing enables developers and operators to reason on their system in a global way. Indeed, most of the major actors of Cloud-Computing reported they have been developing internally distributed tracing for various monitoring needs [8]–[11]. They reported how they are using distributed tracing to detect performance anomalies [9], [11], [12], but also for other scenarios involving a high observability need: like running tests in production [1].

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. A single propagation of a request in the application is called a trace: it establishes causality between latency measurements of the components of the application. Traces are often displayed as a Gantt chart of the time spent in each component involved in the request. To ensure the consistency of traces, the prevailing approach to aggregate measurements is the Google Dapper's span model [9]. This technique to create traces has also been used by academics to identify bottlenecks and root cause in application deployed on PaaS [7].

Due to the variety of Distributed-Tracing solutions, a recent open source initiative, *OpenTelemetry*, merged the two most mature technologies: *OpenTracing* and *OpenCensus*. The first one provides a unified API for tracing to avoid Application Performance Monitor (APM) vendor locking whereas the second one focuses on a production-ready default implementation for various languages. This project has a high visibility in both the Cloud-Computing and the Open-Source communities and, by normalizing Tracing in Cloud Applications, aims to provide a reliable data source for traces. OpenTelemetry recently became the reference in terms of semantic and implementation of Cloud-Native monitoring systems and is now hosted by the Cloud Native Computing Foundation (CNCF).

This project, in early-release at the time of writing, acts as an element part of the pipeline transforming tracing data. This project can interface with the various existing implementation ensuring their mutual compatibility. In addition, it defines a semantic of attributes in spans collected, these elements make this technology a promising candidate to go toward normalization of tracing data.

In our previous works on zonal Kubernetes clusters, we put a focus on a way to leverage OpenTelemetry traces to describe the monitored application as a whole entity [13]. In the later, we propose a variation of this model to adapt to a multi-zone AWS application: this pattern exhibits several structural layers and purposes an approach that pushes toward the edge computations units. Both the topic and the hierarchical network structure resonate to actual IoT challenges regarding service placement. In this work, we present how to use our previous model to aggregate traces in a global hierarchical graph. Then we propose using this graph to exhibit potential bottlenecks within the application architecture that cannot be identified otherwise.

III. Modelling a Global Application With Traces

In the following, a focus will be made on leveraging tracing data from *OpenTelemetry* in order to structure a graph representing communication between entities in a distributed Application. In a later section, this graph will be used as support to spot bottlenecks within a distributed application. Throughout the following, we will refer to the terms of *Service Instance*, *Availability Zone* and *Region* as they are defined in the OpenTelemetry specifications [14].

In our previous work, traces were processed independently to spot inefficient communications in a Zonal Kubernetes Cluster [13]. This work proposes an adaptation of the previous model to describe a multi layers networking model for Cloud-Edge deployments inspired by the AWS concepts of location of resources. This type of location hierarchy corresponds to a containment hierarchy [15], where entities are embedded into each other forming a pure tree. This hierarchy can ve represented as Service Instances $\subset$ Availability Zones $\subset$ Regions.

Instead of processing individual traces, this work focuses on merging the trace graphs in a hierarchical global networking model. The resulting graph therefore exhibits behaviours of the whole system, building a global view of components interactions. The first step of our approach has been to transform each traces into a hierarchical property-graph as part of our previous work. Then, we merge all these graphs in order to maintain at runtime a graph representing communications of the application as a whole.

A. Getting Individual Trace Graphs

With the recent adoption of OpenTelemetry as a framework for monitoring distributed applications, we can expect tracing tools to follow a better "normalization". Traces are 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. There is existing work considering every trace as a Directed Acyclic Graph (DAG) of spans linked by causal relationship [16], [17]. In our previous work, we leveraged OpenTelemetry semantic to exhibit the layered structure of the network and add a hierarchical structure to these span graphs.

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.

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 *Service Instances* scattered on two *Zones*; for the sake of clarity of the graph, *Regions* vertices have been omitted.

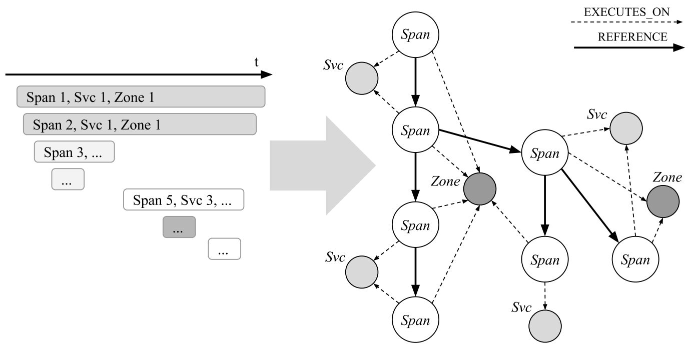

Fig. 1. Transforming a Trace in a Property Graph.

This graph has two different labels for edges. The label REFERENCE comes from the span metadata, the field "reference" within data point to another span within the same trace; traversing the graph by following edges labelled REFERENCE will provide a DAG of Spans. The second edge label: EXECUTES_ON links the Span to the resources (Service instance, Zone, Region) that have been deduced from the resource field from within tracing data.

B. Showing Network Communications with Graph Rewritting

In order to exhibit network communications between services of the application we use a graph rewriting approach. A network call between two services is represented by two spans referencing each other and having different resources attached to them. The goal of the rewriting approach is to add an edge between the communicating entities of the type NET_COMM, the edge would hold important information characterizing the communication: the latency induced by the communication, but also properties from metadata held by the spans. As a result, in our approach, we identify all couples of spans linked via REFERENCE edge but having different execution entities (i.e. Services instances, but also zones and regions) in order to create a single relationship with meaningful properties.

Figure 2 describes the graph rewriting approach, it takes the same conventions as in Figure 1. On the first line, the left side of the operand is 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 with the span vertices deleted. On the second line, the left part of the operand is the graph from trace in Figure 1 (with only Services instances as resource for the sake of clarity), and on the left part the result of the graph rewriting.

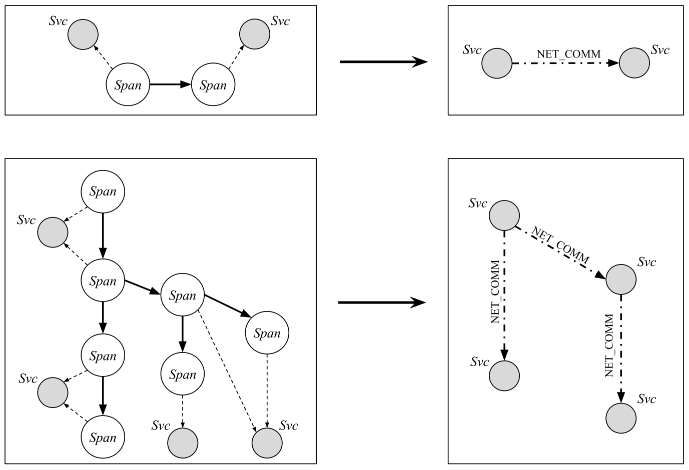

Fig. 2. Graph Rewriting operation to deduce Service instance dependencies.

C. Graph Hierarchical Model

In order to structure the newly created edges NET_COMM, a hierarchical graph model has been used. We define a hierarchical graph model as a Directed Acyclic Graph whose vertices are graphs and whose edges represent morphisms between graphs. With this model, we can maintain the view of the different topologies of Resources. The hierarchy expressed by the model is the containment hierarchy of resources: Service Instances $\subset$ Availability Zones $\subset$ Regions are the vertices of the hierarchical graph model, and the IS_CONTAINED relationships form the morphism describing the containment. Edges within each hierarchy level are the NET_COMMs. The Figure 3 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.

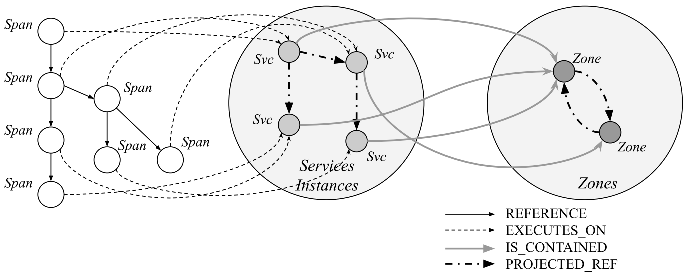

Fig. 3. Hierarchical Graph Representation.

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.

D. Merging Trace Graphs

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. With 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 our model, *Spans* are single instances of a latency measurement observed between two services. *Services Instances*, *Availability Zones* and *Regions* are nodes that may be set in common across multiple trace graphs. Edges, in another hand, may accumulate, as a result, an aggregation method is required to represent communications within the applications. Multiple aggregation methods may be considered based on the data present in traces.

IV. Analysing the Model

The proposed model and techniques leverage the heavily connected nature of traces and their semantic to maintain at runtime a hierarchical property graph. Each layer is a directed graph with vertices being the resource considered at this hierarchy level; edges being the network communications observed between these entities. As one prominent challenge of monitoring a Cloud Application is to identify bottlenecks, the main goal of the final graph is to represent a global view its behaviour.

A. Distributed Applications Bottlenecks

Bottlenecks can take two mains forms, the first one being the resource saturation bottleneck, it manifests when a single component reaches its *limits*. While these the kind of limit may vary according to the type of service described, it can go from CPU or memory usage to disk queue; it always causes significant delays to requests processing. Common methods to handle these bottlenecks are the use of message queues that can handle the back-pressure or the use of a dynamic scaling of such critical resources. The second form of bottleneck is the resource contention bottleneck, it manifests in environments having semaphores, messages queues, buffers and mutexes. It is commonly found in environments having concurrent access to the same resources, which happens a lot in Cloud environments, indeed the over allocation made by providers in data centres causes the *noisy neighbours* issue. This type of bottleneck also causes significant delays but is more difficult to spot as CSPs usually do not provide underlying metrics which may highlight this critical behaviour.

Bottlenecks manifestations causes performance anomalies which usually are contention, saturation, deadlocks or even partial failures of the system [18]. Their causes, however are numerous : misconfigurations or bad system tuning can greatly decrease performance, applications updates and introduction of buggy code can also lead to performances issues on the short term. Finally, underlying transient events or platforms re configurations are common events that may cause bottlenecks within the system. In the following, we propose using the global application network model in order to spot potential bottleneck services in a Cloud application.

As each technologies deployed in a Cloud application may exhibit different bottlenecks metrics or behaviours, our contribution focuses on identifying the potential bottlenecks based on components interactions. To identify specific and critical vertices in a graph, centrality algorithms are often used to spot important vertices within a network. Although various centrality algorithm exist and they all have a different approach for identifying a vertex "importance".

B. Overview of Graph Centrality Algorithm

Degree Centrality Ranking: The degree centrality sorts vertices by the number of edges associated to them, the higher is the number of edges for each single vertex, the more important this vertex is. In the case of a distributed application, a focus on the number of incoming edges would represent how many services depends on each particular services. A degree centrality therefore highlights the most solicited services which are, therefore the inclined of having a contention bottleneck.

Betweenness Centrality Ranking: The betweenness centrality give importance to vertices based on the total number of paths any given vertices is involved into. Betweenness centrality is known to exhibit so called bridges vertices which are nodes that help linking different communities. The entities represented by these vertices do indeed represent bottlenecks. When identifying such vertices with high betweenness score, the underlying service acts as a single bridge between two communities. Scaling up these bridges services would reduce their betweeness score. Considering an IoT application, gateways often acts as central points of collection for many sensors or mobile devices. Betweenness centrality would rank the gateways based not only on their direct neighbours but by their implication in the whole network.

V. Implementation and Simulation

For this implementation we reused the trace processing pipeline initially developped in [13]. We instrumented a simulation software representing Cloud applications inspired by the Netflix architecture named *spigo* [1]. This simulation software uses the actor design pattern to mimic the microservices architecture. It is written in Go and leverages the lightweight implementation of threads in the language to run hundreds of actors simultaneously, each representing a micro-service. For our purpose, actors have been modified to produce traces with a semantic compliant with OpenTelemetry specifications. This simulation therefore produces traces that are ingested by the Jaeger Tracing tool, the de-facto tool to visualize OpenTelemetry Traces.

A. Creating a Pipeline for Trace Processing

The graph modeling, rewriting and merging processes have been written in Scala by using a functional programming approach on the *Polynote*[2] data-processing engine. 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

[1]https://github.com/adrianco/spigo

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

on the work published in the *Jaeger Data Analytics Library*[3]; the capability of matching the various resources type has been added in order to create resource vertices in the model. Trace graph encoding and rewriting processes have been implemented with the Gremlin [19] on Tinkerpop In-Memory graphs. Finally, for the merging stage in the hierarchical model, a Neo4J database was used. This graph database has a clean and powerful syntax to create or reuse vertices from the database without prior checks. Our graph merging stage requires to identify resources vertices already present in the model and took benefits of this feature.

This approach is then scalable and allows an online and parallel 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.

B. Simulation Overview for a Wider Picture

Whereas our preceding work was based on a sample application hosted on a real-world cloud, our last experimentation suffered from its small size and lack of traffic. Indeed, behaviours observed from a full size application running in production are either considered as black swans or may be considered as edge cases [20]. For modeling a complete multi-zone AWS application, we used the tool spigo, a simulator running on a single machine a full hierarchy of actors. These actors represent service instances, although the model does not provide a realistic implementation for managing network delays, latency measurements obtained from traces cannot be exploited to draw numerical conclusions.

In order to represent a real-world Cloud-IoT application, we based our testing on a pre-made architeture from within the simulator: the Riak scenario. Indeed, in this scenario is based on Riak application which is a distributed and resilient database targetted for IoT needs. It offers a way to replicate parts of its architecture to multiple zones, it also implements model for both HTTP communications accross actors but also message queues. The use case of a the Riak application provides an implementation of an heavily hierarchical app, having multiple endpoints scattered accross zones and positioned close to the users. The backend of the app is modeled in the scenario by a multi zones cassandra cluster.

VI. Experimentation

The architecture of the Riak scenario is made of multiple processing stages; detailing all of them in the paper would be out of the scope of this contribution. Therefore we will focus on critical services and make them scale up and down to observe the effect on global graph centrality scores. When building the graph of services it appears that the service designated *ingestMQ* hold an important role bridging the Cloud back-end of Riak to the zonal processing pipelines. In the global architecture, *ingestMQ* is a three-replicas message queue deployed in each region to serve local users. By its role and its design, this service is concerned by risk of resource contention bottleneck. Figure 4 represents a fraction of the graph of Riak services, it is made of 135 services scattered on three regions, each subdivided into three zones. Vertices with a *storage* icon and a label are *ingestMQ* instances.

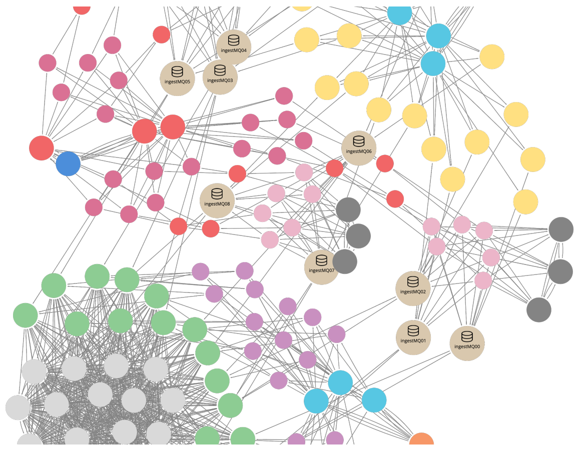

Fig. 4. A sample of the graph of services with a focus in *ingestMQ* services

In our experiment we ran various centrality algorithms on the model. The table I aggregates the results and provides a per service range of the centrality scores obtained by each services for multiple algorithms. $\mathcal{C}_b$ is the designation for the betweenness centrality, $\mathcal{C}_d$ is the designation for the in-degree centrality, it counts only incoming edges to the node.

Service Name	Replicas	$\mathcal{C}_b$	$\mathcal{C}_d$
ingestMQ	9	$[181, 193]$	$[2, 2]$
ingester	18	$[97, 104]$	$[18, 18]$
enrichMQ	9	$[87, 97]$	$[2, 2]$
enricher	18	$[53, 57]$	$[6, 6]$
normalization	18	$[13, 18]$	$[1, 1]$
stream	18	$[3, 12]$	$[1, 1]$
analytics	18	$[5, 9]$	$[2, 2]$
riakTS	18	$[0, 0]$	$[17, 17]$
riakKV	9	$[0, 0]$	$[1, 1]$

TABLE I
Centrality algorithms score range for each group of services in the Riak scenario

These results show how complementary are the two centrality algorithms: they do not exhibit the same service as being important. The betweenness centrality score highlights with a important score the ingestMQ service whereas the in-degree centrality puts massive score on the cloud part of the system where each component communicate with each other without zone separation.

The betweenness centrality score grants a risk indicator of the impact of a failure from these service. As the number of instance increases the impact of a failure of these component should decrease. To illustrate this idea we rerun the simulation by setting the number of ingestMQ services of one per region and of five per region. Table II show the variation of the betweenness centrality score based on the number of instances of the service ingestMQ. In this table, column $\mathcal{C}_{b1}$ give centrality score for the whole graph when there is only one

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

instance of ingestMQ service per zone. Column $\mathcal{C}_{b3}$ keeps the previous values and $\mathcal{C}_{b5}$ provides the score when there are five instances of the service ingestMQ per regions.

Service Name	$\mathcal{C}_{b1}$	$\mathcal{C}_{b3}$	$\mathcal{C}_{b5}$
ingestMQ	$[383, 422]$	$[181, 193]$	$[104, 151]$
ingester	$[187, 219]$	$[97, 104]$	$[118, 123]$
enrichMQ	$[74, 77]$	$[87, 97]$	$[92, 110]$
enricher	$[43, 44]$	$[53, 57]$	$[57, 65]$
normalization	$[14, 16]$	$[13, 18]$	$[13, 25]$
stream	$[3, 7]$	$[3, 12]$	$[8, 12]$
analytics	$[5, 10]$	$[5, 9]$	$[5, 9]$
riakTS	$[0, 0]$	$[0, 0]$	$[0, 0]$
riakKV	$[0, 0]$	$[0, 0]$	$[0, 0]$

TABLE II
BETWEENNESS CENTRALITY SCORE RANGE FOR EACH GROUP OF SERVICES IN THE RIAK SCENARIO WITH A VARYING NUMBER OF INGESTMQ INSTANCES

As a result, we can observe that the betweenness centrality score drastically increases when the service ingestMQ is only present once per regions whereas it confounds with other score when there are fives instances per regions. In another hand, we can also observe that other services preserve their order of magnitude of betweenness centrality score when scaling only one service.

CONCLUSION AND FUTURE WORKS

In our contribution we proposed a model capable of representing a physically distributed application, its components and their interactions as a property graph. It follows the pattern of a multilayered network model that commonly describes Edge computing and IoT resource location. The monitoring data supporting this model creation is OpenTelemetry, a fast-growing technology in the cloud ecosystem. This contribution presents a model and how to derive it from actual real-world monitoring data. A processing pipeline capable of parallel execution of this model is also presented and executed on a simulation program. This simulation use-case, based on the spigo open source project, represents an AWS application made of hundreds of services leveraging the zones and regions placements to bring the service closer to their users.

Once the graph is built, we propose identifying bottlenecks through the use of centrality algorithms. A particular focus is made on resource contention bottlenecks, which may not be detected by standalone component monitoring. The betweenness centrality algorithm appears to be a good candidate to spot services holding a crucial role within the application composition. Further, the betweenness centrality score is greatly influenced by the number of instances of a given service. This makes this centrality algorithm a good candidate for identifying critical services to scale and replicate in a heavy loaded environment.

In [13] we ran the same model to spot inefficient communications within the context of Zonal Kubernetes Clusters by identifying cycles within the graph. For future works, we plan to take advantages of the quantity of metrics and data that can be produced by a Service Mesh in a Kubernetes cluster. Indeed Service Meshes have solid tracing implementations that output latency but also the number of byte exchanged, etc. These new metrics could be used to better describe the network communications observed between entities and, therefore, running centrality algorithms on a weighted graph. Finally, to conciliate these techniques and indicators, a monitoring application is being designed to aggregate these results.

REFERENCES

[1] D. Ardelean, A. Diwan, and C. Erdman, *Performance analysis of cloud applications*. USENIX Association, nov 2018.

[2] 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.

[3] S. Alarifi and S. Wolthusen, "Anomaly detection for ephemeral cloud IaaS virtual machines," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 7873 LNCS, pp. 321–335, 2013.

[4] R. Heinrich, A. van Hoorn, H. Knoche, F. Li, L. E. Lwakatare, C. Pahl, S. Schulte, and J. Wettinger, "Performance Engineering for Microservices," *Proc. 8th ACM/SPEC Int. Conf. Perform. Eng. Companion - ICPE '17 Companion*, pp. 223–226, 2017.

[5] R. Heinrich, "Architectural Run-time Models for Performance and Privacy Analysis in Dynamic Cloud Applications Categories and Subject Descriptors," *Perform. Eval. Rev.*, vol. 43, no. 4, pp. 13–22, 2016.

[6] G. Da Cunha Rodrigues, R. Calheiros, V. Guimaraes, G. Dos Santos, M. De Carvalho, L. Granville, L. Tarouco, and R. Buyya, "Monitoring of cloud computing environments: Concepts, solutions, trends, and future directions," *Proc. ACM Symp. Appl. Comput.*, vol. 04-08-Apri, 2016.

[7] 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.

[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] C. Cassé, P. Berthou, P. Owezarski, and S. Josset, "Using Distributed Tracing to Identify Inefficient Resources Composition in Cloud Applications," in *IEEE 10th Int. Conf. Cloud Netw.*, Virtual, 2021.

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

[15] 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.

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

[17] 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.

[18] 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.

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

[20] M. Marvasti, A. Poghosyan, A. Harutyunyan, and N. Grigoryan, "Identifying Root Causes, Bottlenecks, and Black Swans in IT Environments," *VMWARE Tech. J.*, vol. 2, no. 1, pp. 35–45, 2013.