Implications of Real-Time Monitoring in Latency-Critical Microservice Applications

Alexandros Mavrogiannis
Carnegie Mellon University
amavrogi@andrew.cmu.edu

***Abstract*—The microservices architectural style is a paradigm that enables distributed applications to achieve high scalability by fine-tuning the deployment of minimal processes that represent fine-grained application components. We introduce a new monitoring technique for applications structured through the microservices architectural style that allows the logging of individual application-level requests. Using this framework, we demonstrate how the logged requests can be used to generate a weighted graph of microservice dependencies. We explore uses of this graph in existing cluster scheduling algorithms and demonstrate its efficiency for latency-aware scheduling.**

I. Introduction

Media delivery services that offer on-demand video or audio streaming have been growing in popularity over the past years. Specifically, the leading provider of on-demand video in the US, Netflix, accounts for 29.7% of the peak downstream traffic in the US [1]. After acquiring Microsoft MediaRoom in December 2013, Ericsson has been also striving to deliver on-demand, Internet-based media services. Due teir the throughput and latency requirements,media services have historically followed the microservices architectural style.

Under the microservices architectural style, monolithic system components are split into independent language-agnostic services that encapsulate a minimal set of logical features [2]. The fine-grained approach mirrors the Unix philosophy of "Do one thing and do it well" [3]. The microservices architectural style effectively represents a cloud-native approach with an emphasis on availability and scalability.

In the context of a media service delivery service, each external request typically requires the generation of a data-oriented workflow in order to query data from various sources over time. Such media requests often have variable latency requirements, such as in the context of calculating movie ratings or retrieving real-time media content. Essentially, each request generates one workflow which also represents the path of service invocations over a set of microservices.

There are various benefits in the use of microservices in the context of media services, such as the minimization of end-to-end latency, increased reusability of system components and fine-grained scalability. For example, a user profile storage service can be a reusable service in a workflow, which can be also combined with a movie recommendation service in order to serve "browsing" page to a user which offers movie recommendations. These services may have variable latency guarantees and, thus, may be deployed in fewer replicas over a clustered computational system.

Applications developed in the microservices architectural also benefit from more streamlined development cycles, as newer versions of individual application-level components can be gradually rolled into a production environment with no changes in user-perceived latency.

In this work, we explore how we can combine existing application-level and system-level monitoring techniques in order to facilitate scheduling decisions and, thus, affect the perceived latency of requests by an end user. A first step in that direction is the establishment of a mechanism to monitor, track, and record the different types of latency that can be observed within a microservice network. The concept of graph-based representations of system components is not new, as there are existing approaches which utilize call graph analysis in order to partition a graph into separate components [4] [5].

It is worth mentioning that the generation and deletion of microservice instances, or *replicas* may be of non-neglibile latency. In order to increase the efficiency of these microservices, replicas can be dynamically scaled when demand increases, in a closed-loop fashion. More specifically, monitoring data over a historical period of time is utilized to dynamically configure the number of microservices with identical functionality.

II. Microservice Network Formalization

We define a *microservice* as self-containing set of functionality that is logically grouped with an application-level component or constraint. Each microservice encapsulates a set of API functions and is packaged along with all potential dependencies. From a system perspective, a microservice is a process that requires a variable number of system resources, such as CPU, Memory, Network throughput and I/O throughput and it is able to communicate with similar microservices or the external world through well-defined application-layer protocols. Throughout this paper, we will use the terms *microservice* and *service* interchangeably.

In order to monitor and manage microservice latency at various granularity, we diffentiate across the concepts of a microservice instance and a microserivce class. A *microservice class*, C, defines the behavioral definition of a set of microservice instances with common behaviors. A microservice instance *implements* a microservice class if its functional behavior fully matches the behavioral definition that is stated by the microservice class.

We then define a *microservice network*, MSN, as a set of microservice instances that are deployed in an actual computational cloud environment. Each node in a microservice network represents a microservice instance. An edge existing between two nodes indicates that the two services on the two ends collaborate with each other in a workflow (to answer a specific user query).

Within our microservice network, we assume that microservices are able to discover each other using a class-based mechanism, meaning that any microservice instance can query over a predetermined discovery mechanism to retrieve a routable address of a microservice instance that implements a specific microservice class. Some existing microservice-oriented frameworks, such as Kubernetes, have implemented load balancing as part of the service discovery mechanism. Therefore, we will leverage the implementation of discovery and load balancing mechanisms as a defining characteristic of a microservice network.

Finally, we define a *query* in the context of a microservice network as the initial invocation of a singular functionality of a specific microservice, the *recipient*. Recipients can also invoke functions of other microservices within the context of a single query. Based on this definition, we identify two categories of queries that can occur within a microservice network:

- External queries, which are triggered by the invocation of external-facing microservices of the microservice network from an end-user.
- Internal queries, which are triggered within the network itself, such as by periodic background tasks or work queues.

The resulting set of microservices that are utilized to fulfill a specific query will be defined as a *query path*.

A *workflow* is a repeatable pattern of business activity that enables the transformation of information in a predetermined manner that may involve multiple steps. In the context of microservices, we can consider that a query path is an implementation of a workflow.

In this work, we model a single microservice network as a directed weighted graph $G = (N, E)$, with nodes representing individual microservice instances and edges that reflect the intensity of the directional communication path between two microservice instances. Using Graph G, we are able to define graph $G_c = (N', E')$ where nodes represent microservice classes rather than instances.

Each edge in E and E' may hold a value in the range $[0, 1]$, which reflects the statistical probability of a query exiting the originating node. For a given node n, the sum of the probabilities of all outgoing edges of that node always add up to 1 minus he probability that the query path is finalized at n, P_{S_n}, as depicted in Equation 1.

$$P_{S_n} + \sum_i P_{S_i} = 1 \tag{1}$$

Furthermore, the latency of a microservice can be defined as the round-trip time between sending an API request to that microservice and receiving an API response from the microservice. For a given microservice, s, we define its latency, $l(s)$ as per Equation 2.

$$l(s) = t(\text{request}) - t(\text{response}) \tag{2}$$

III. Problem Definition

In this work, we present how monitoring data regarding latency and system load can be extracted from a microservice network deployment and stored in a graph format. We explore how this graph representation can be utilized to make scheduling decisions that lead to latency minimization while also addressing system load distribution across a cluster of physical machines.

We utilize existing techniques in microservice network monitoring such as trace analysis and system-level monitoring. We store the results in a graph-based representation of combined application and system-level mitrics. We show how the assignment of weights to the edges of a graph representation of a microservice network, G, can be generated from real-time observation of the latency across microservice calls and the underlying system load.

We explore potential metrics that can be extracted from the graph G and their use cases and how they can be used as heuristics in existing real-time cluster scheduling algorithms, such as Best-Fit-First, in order to improve the latency of complex application-level requests. Furthermore, we explore how the graph G can be used to also determine the number of microservices that need to be deployed for each microservice class in order to minimize communication bottlenecks of the deployed application. Finally, we identify that the graph G can be used to determine optimal static scheduling policies. For each workflow, we calculate its latency based on the latency of all comprising microservices. In a simplest version when the services are chained in a series, the latency of the workflow is the sum of the latency of comprising services. For example, $L(wfi) = l(S1) + l(S2) + l(S3)$.

When a new query arrives, it triggers the construction of an instance of a workflow template. If there exist workflow instances, then their loads are checked based on the latency records of the workflow instances. If the load of a workflow instance is below the threshold, the query will be assigned to the workflow instance. In other words, a workflow instance can serve for multiple queries simultaneously. If there does exist an instance of an intended workflow, then a workflow instance is created.

A. Call-graph Analysis to Understand SLA Violations

Interaction between microservices are loosely coupled with the consuming service dynamically determining the specific instance of a service to invoke. These *bindings* are transient and can change when the service either fails (i.e., failover) or when the consumer determines that the service is not meeting the desired SLA (we term *soft-failover*). Any such failover increases request handling latency and if a significant number of such failovers occur the SLAs defined for a service interface

may be violated. Having a means to capture and analyse these failures can help debug more significant issues with the system. For instance, knowing which specific segments of the microservice call graph add to the increased overall latency can point to resource underallocations or other transient failures (e.g., application freeze). The information can then be used to allocate more resources or in placement decisions.

IV. Microservice Packaging

Applications that are developed under the microservices architectural style are typically packaged and deployed on an on-premise or public cloud cluster of machines. As a result, the limitations and benefits of the underlying packaging format can significantly affect application-critical attributes of the deployment environment such as latency, throughput or resource availability. Furthermore, the packaging of an applications microservices is directly related to the portability of that application across cloud providers. In this section, we examine why operating-system-level virtualization seems to be an ideal candidate for the packaging of microservices.

Most modern cloud computing providers such as Amazon Web Services, Google Cloud Platform or Microsoft Azure offer their users virtual machines with user-configurable resource constraints. In such cases, full virtualization or paravirtualization are used to generate a new machine. The new machine is populated with a user-configurable operating system image, which may also include the users application to allow for deployment automation.

Operating-system-level virtualization is another form of virtualization that enables lightweight isolated userspace environments, called containers, to co-exist on top of the same physical host. Containers are fundamentally different from virtual machines, as all containers in a host machine share the same Operating System (OS) kernel, rather than spending CPU cycles on a guest OS for each environment. When compared to Virtual Machines, the lightweight nature of containers allows for faster boot times and a capacity for more concurrent environments on each node. At the same time, the applications hardware resource utilization rate will improve, due to the reduce OS overheads. However, one constraint is that containers cannot be used to migrate applications across nodes with different operating systems, such as Linux and Windows.

OS-level virtualization has been thoroughly explored in unix-based systems with examples such as FreeBSD jails and Linux Containers (LXC) [6] and more recently by Docker, which combines the aforementioned portability and isolation with a layered image composition and a reproducible build system [7]. Using this system, it is possible to create an *image* of a microservice in a reproducible manner, through the use of `Dockerfiles`. The image can be then used to generate identical container instances that are easily deployable in a cluster environment.

We can identify that the lightweight nature and added portability of Docker containers are in alignment with the deployment constraints of applications that follow the microservices architectural style. Furthermore, the concept of an *image* as the building block for container instances that Docker follows fits our model of microservice classes and microservice instances. Finally, we can identify that the implementation of networking within Docker containers poses no additional overhead to the latency or bandwidth of communication as when an application is natively running on a host machine. Due to these reasons, we utilize Docker in this work as the standard packaging mechanism for all developed microservices.

V. Application Request Monitoring

In order to capture and measure the cross-dependencies of microservices, we establish a monitoring scheme that is based on logging queries that are invoked at each microservice. Such a scheme allows us to determine the invoked query paths over the specific time duration where monitoring is performed.

Specifically, each query is marked with a Request ID by the microservice that is first invoked to satisfy that query. Each subsequent microservice invocation in the query path also carries the same Request ID. In scenarios where there are multiple externally-visible microservices, a synchronization mechanism is required across the border microservices to assure that IDs are uniquely generated within the deployed application.

The monitoring mechanism stores each microservice invocation as a message that contains S_{in}, the source microservice instance, S_{out}, the invoked microservice instance, ID, the Request ID that corresponds to this message, F, the method of S_{out} that is invoked and t, the absolute timestamp of the message at the time when it is processed by S_{out}. A tuple of these value constitutes the message msg, as presented in Equation 3.

$$msg(S_{in}, S_{out}, \text{ID}, F, t) \tag{3}$$

By counting the number of unique request IDs that were processed by each microservice instance, we are able to calculate a request ratio for each edge of G.

VI. Experimental Setup

For our experimental setup, we implemented a media delivery service application that follows the microservices architectural style.

The architecture of the developed application is composed of the following services:

- A Movies Service
- An Images Service
- A Ratings Service
- A Frontend Service
- A User Service
- A Search Service
- A Service Discovery Service
- An API Gateway Service

For the implementation of the application, we utilized the Spring Cloud framework to develop each individual service,

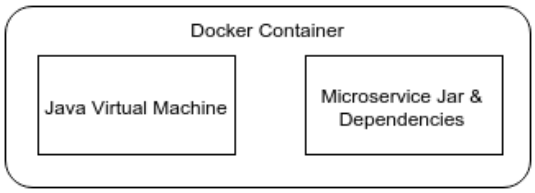

Fig. 1: Packaging of Spring Cloud application within a Docker container

using the Java programming language. For the Service Discovery Service, we utilized the open source Eureka Service Registry that is maintained by Netflix.

A. Target Platform

We applied the suggested monitoring techniques on two separate platforms:

- A Spring Cloud based implementation, deployed over two Virtual Machines in the cloud.
- A configurable simulation developed for the purposes of this experiment over the same physical host.

For the Spring Cloud implementation, a stripped-down version of a media delivery network was implemented as a set of individual microservices. Each microservice was packaged as a Spring Boot application packaged within a Docker container. Service Discovery is performed with the the Eureka service discovery tool, which was deployed as a Docker-packaged microservice itself. The cloud provider that was utilized for the evaluation of this implementation is Amazon Web Services. Each node was provisioned as a `t2.small` instance, featuring one virtual CPU and 2 GB of RAM. Deployment of a specific configuration of microservices is achieved through a set of deployment scripts that handle all necessary links across the deployed containers and the underlying database instances.

For the simulation environment, we implemented a standardized microservice simulation engine, based on virtualization, available at `https://github.com/afein/docker-microservice`. The simulation environment utilizes Docker Swarm, a clustering tool aimed for ease-of-deployment of Docker containers across a cluster of physical nodes. The physical machines that comprise the cluster are deployed using `docker-machine`, a tool that deploys the required nodes on a specific cloud provider, or locally through a hardware virtualization backend. Service discovery is performed through the use of Docker overlay networks, which allow microservices to communicate with each other through DNS names. Deployment of services is achieved, again, through a set of deployment scripts which simply generate instances of Docker containers, as no further linking of containers is required.

Each simulated service can be triggered through a POST HTTP request on its port 80, containing a JSON-formatted object which specifies which services to contact in the same call graph path, as well as how many seconds to spent performing computation and how many seconds being idle. These configuration parameters permit simulations to generate pseudo-realistic system-level metrics from a monitoring perspective. Furthermore, the input format permits probabilistic paths to occur. In order to avoid cycles in these situations, we assume that the input to such a microservice always requests a predetermined path length, and deviations from that should be handled by the caller of the simulation.

B. Implementation of the Monitoring Framework

We divided the implementation of the discussed monitoring framework into two distinct categories: system-level monitoring and application-level monitoring.

1) Application-Level Monitoring: We perform application-level monitoring at the service message level according to the log message specification defined in Equation 3. In order to capture these messages, we utilized a custom-tailored solutions that simulates the behavior o existing tracing frameworks such as *Spring Cloud Sleuth*. Tracing frameworks, such as Google Dapper, aim to generate graph-based representations of system invocations in complex deployments [8]. In tracing frameworks, individual microservice calls are identified and logged with a unique *span* Identifier, as referred to as Request Identifier. A workflow that involves several spans is defined as a *trace* and identified by the framework through Trace Identifiers.

Through our compatible tracing implementation, we are able to extract messages that contain the following fields:

- A Trace ID, reflecting the request ID as assigned by the API Gateway service.
- A Span ID, representing an individual service invocation.
- An identifier of the microservice instance that was invoked.
- An identifier of the microservice endpoint that was invoked.
- An identifier of the *parent* span ID, which is span in the same trace that invoked the current span.

The developed system exports Sleuth Log Entries to a standardized location. One example log entry of that format is the following:

```
2015-11-25 21:36:23.009
INFO 11786 --- [http-nio-80-exec-2] o.s.
    ↪ cloud.sleuth.log.Slf4jSpanListener
    ↪ :
Continued span: MilliSpan(begin
    ↪ =1448487383009, end=0, name=http/
    ↪ movie/1224,
traceId=1d5337fd-1ab1-4776-b218-35
    ↪ a53a8b7d25,
parents=[],
spanId=2960891b-6496-4823-aa11-888468
    ↪ c68695,
remote=false,
```

Metric	Units
CPU Usage	milliCores
CPU Usage (cumulative)	nanoseconds
CPU Limit	milliCores
CPU Usage (per Core)	milliCores
Memory Usage	bytes
Memory Working Set	bytes
Memory Limit	bytes
Network TX Throughput (per interface)	bytes
Network RX Throughput (per interface)	bytes
Network TX Errors (per interface)	N/A
Network RX Errors (per interface)	N/A
Filesystem writes (per block device)	bytes
Filesystem reads (per block device)	bytes
Filesystem size (per block device)	bytes

TABLE I: System-level metrics exposed by CAdvisor and their respective units

```
annotations={/http/request/uri=http://ec2
    ↪ -XXX-XX-XX-XXX.compute-1.amazonaws.
    ↪ com/movie/1224,
/http/request/endpoint=/movie/1224, /http
    ↪ /request/method=GET, /http/request/
    ↪ headers/connection=keep-alive,
/http/request/headers/host=ec2-XXX-XX-XX-
    ↪ XXX.compute-1.amazonaws.com,
/http/request/headers/user-agent=Apache-
    ↪ HttpClient/4.3.3 (java 1.5)}
, processId=null, timelineAnnotations=[])
```

Our use of this tracing technique introduces furtherlatency overheads to the overall microservice network, due to the need for timestamping, buffering and storage of each individual call across microservices. However, this latency can be amortized across requests or absorbed during low-load time periods by adjusting the frequency of filesystem flushes of the underlying log files.

2) System-Level Monitoring: Docker containers achieve resource isolation in Linux hosts through the use of Linux Kernel resource groups, or `cgroups`. We perform system-level monitoring of individual microservices by extracting resource usage metrics from the Linux Kernel cgroups that are allocated for each individual microservice. To that end, we utilize *CAdvisor*, a native cgroup monitoring tool which is able to expose real-time data for individual Docker containers through a RESTful Application Programming Interface. CAdvisor exposes a large set of metrics, presented in Table I.

Using

We perform aggregation of the system-level metrics exposed by CAdvisor from every node in the target cluster using *Heapster*, a cluster-level metric collector. Heapster periodically polls the CAdvisor instance that is operating on each node of the cluster, performs aggregation using labels attached to each metric and exposes them through a set of storage backends, or a RESTful API.

Using the RESTful API of Heapster, we are able to extract all system-level metrics deemed relevant for our implementation through periodic polling. For visualization, we use

Metric	Units
CPU Usage	milliCores
CPU Limit	milliCores
Memory Working Set	bytes
Memory Limit	bytes

TABLE II: System-level metrics that are used for scheduling decisions

Kubedash, a visualization tool that exposes Heapster's internal buffers which hold up to one hour of lossy data. For storage, we configure Heapster to store all metrics in a Timeseries database, InfluxDB, which is also running in the target cluster.

Finally, we identify the set system-level metrics can be used to complement our application-level monitoring graph solution and present them in Table II.

VII. Measurements & Results

VIII. Related Work

Even though the architecture of Netflix is not publicly disclosed, the company often publishes architectural artifacts through its technical blog [9]. Additionally, the externally visible architecture of Netflix has also been documented, primarily in the context of Content Delivery Network (CDN) selection [10]. Using these two sources, we are able to identify that Netflix operates as an international-scale distributed system that operates in patterns that resemble the microservices architectural style.

IX. Conclusions

In this work, we motivated the use of real-time monitoring in the context of applications that follow the microservices architectural style, such as media delivery platforms. We presented a formalization of these applications as *microservice networks* and motivated the use of Docker as a packaging techniques for microservices. We then utilized existing monitoring techniques to examine how application-level and system-level monitoring can be used to generate graph-based representations of a microservice network.

We evaluated the performance of our monitoring environment over a media delivery service application, using both a semantically-accurate implementation as well as simulated implementation. We performed measurements over clusters deployed on Amazon Web Services and demonstrated a technique for the visualization of the results.

Finally, we discussed how the new graph-based representations can be utilized to improve scheduling decisions across variable parameters

References

[1] S. N. Demographics, "Global internet phenomena report: Spring 2011," 2011.

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

[3] L. KRAUSE, *MICROSERVICES: PATTERNS AND APPLICATIONS*. KRAUSE, LUCAS, 2014.

[4] X. Gu, K. Nahrstedt, A. Messer, I. Greenberg, and D. Milojicic, "Adaptive offloading for pervasive computing," *Pervasive Computing, IEEE*, vol. 3, no. 3, pp. 66–73, 2004.

[5] M. G. Nanda, S. Chandra, and V. Sarkar, "Decentralizing execution of composite web services," in *ACM Sigplan Notices*, vol. 39, no. 10. ACM, 2004, pp. 170–187.
[6] M. Helsley, "Lxc: Linux container tools," *IBM devloperWorks Technical Library*, 2009.
[7] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.
[8] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure," *Google research*, 2010.
[9] Netflix, "Technical blog," http://techblog.netflix.com/.
[10] V. K. Adhikari, Y. Guo, F. Hao, M. Varvello, V. Hilt, M. Steiner, and Z.-L. Zhang, "Unreeling netflix: Understanding and improving multi-cdn movie delivery," in *INFOCOM, 2012 Proceedings IEEE*. IEEE, 2012, pp. 1620–1628.