Performance behavior of microservice-based cloud applications

Mauro Famà
Politecnico di Milano
mauro.fama@mail.polimi.it

Abstract—**In this paper we explore implications arising from microservice-based cloud applications in terms of architecture and performance, we need to measure and effectively understand this complexity in order to reach a tradeoff between resource usage and performance. We use DeathStarBench [1], an open-source benchmark suite built with microservices that is representative of large end-to-end services, modular and extensible. We first implement the benchmark on physical hardware and then on virtual machines in public cloud infrastructures. We then use DeathStarBench to study the implications of microservices in networking and their challenges respect to cluster management. Finally, we compare the latency distribution effect of microservices with different workloads.**

Index Terms—**cloud computing, microservices**

I. Introduction

Microservices have a distributed architecture, which means that applications, cooperating with each other, reside on several working nodes. Working nodes can execute applications locally and they collaborate for the pursuit of common goals through a communication infrastructure. Nodes are available even with limited resources and they are very flexible, but they need a stable connection between all the nodes and managing the whole infrastructure could get complicated.

To understand how to arrive at a distributed architecture we must start from the monolithic one: we choose a simple monolithic application and to make it more available let's scale it horizontally, that is, replicate it by making multiple identical copies, each host that makes an identical copy will have its own data storage and he will have to take care of it; to face the increased complexity of the application, we vertically scale the application with a functional approach, that is, by breaking down a monolith into a set of services, in this way each service can deal with a specific function, and it is from here that it is defined a microservices architecture.

Microservices fundamentally change a lot of assumptions current cloud systems are designed with, and present both opportunities and challenges when optimizing for quality of service (QoS) and utilization.

II. Application containerization

When we want to deploy an application with virtualization technology, the package that we use is a virtual machine, and it includes an entire operating system in addition to the application. A physical server running three virtual machines would have a hypervisor and three separate operating systems running on top of it. By contrast, a server running three containerized applications runs a single operating system, and each container shares the operating system kernel with the other containers. Containers are the set of data an application needs to run: libraries, other executables, file system branches, configuration files, scripts, etc. A container is an isolated environment, and it is the isolation that allows us to run multiple containers simultaneously within a host. That means the containers are much more lightweight and use far fewer resources than virtual machines. A container may be only tens of megabytes in size, whereas a virtual machine with its own entire operating system may be several gigabytes in size. Because of this, a single server can host far more containers than virtual machines. Containerd [8] is an open source project created with the aim of automating the distribution of applications in the form of containers that can be run on the cloud (public or private) or locally, we will use it to manage applications images of DeathStarBench. Microk8s [3] has its own Containerd suite already installed. Kubernetes [2] is an open-source system for automating deployment, scaling, and management of containerized applications, that we will use to control DeathStarBench. Pods are the smallest deployable units of computing that you can create and manage in Kubernetes. A pod is a group of one or more containers, with shared network resources, and a specification for how to run the containers.

III. Physical Hardware implementation

A. Deployment

We first deployed DeathStarBench on a local machine (1x Intel Core i3-7100 CPU @2.40 GHz, 8GB RAM) deploying the benchmark with Kubernetes on a three nodes cluster (one master node and two nodes) created locally with MicroK8s. Microk8s runs in an immutable container, so Kubernetes itself is fully containerized and we have no moving parts, for better security and simpler operations. An `nginx` [4] pod is the entry point of the cluster for external resources, we use a NodePort Service to allocate a physical port on the K8s Node. This Service type would allow us to expose the Service in a meaningful manner, the Service port will be mapped to a port opened on the interface of every Node.

B. Balancing of the load generator

Running a cluster on a single physical hardware, even if is just for learning, testing and/or developing demonstrates that the performance of microservices depends on the CPU resources and connectivity. Without a good balancing between the number of threads and connections by the HTTP load generator that we use to make qualitative tests, which is wrk2 [5] in this case, a significant number of connections will not be successful, finishing with non-2xx or 3xx response, making the test not relevant for statistics.

In Fig.1 we can observe the values of requests per second and transfer per second completed after three tests that run a benchmark for 30 seconds and a constant throughput of 2000 requests per second. We balanced the number of used threads with the right amount of connections opened, according to the computational power of the physical machine, in order to make the most of the machine power. Pods were correctly scaled in each node to have the best performances.

With a correct balancing of the load generator, requests per second completed by the nodes remain performing even with more connections and more threads, despite the increase in the weight of loads generated. Correct sizing between computational resources and the number of connections allows improving cluster behaviors.

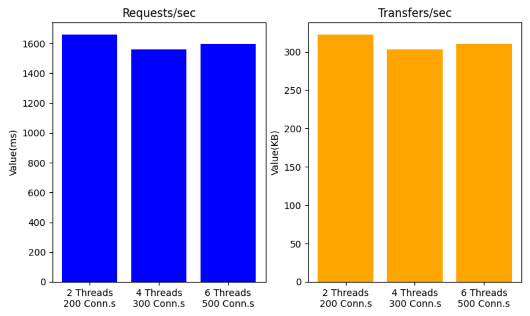

Fig. 1. Values of requests per second and transfer per second completed after three different tests

C. Latency Distribution

Generating workloads with the same number of threads, but with increasing connections, shows that latency increases proportionally with the number of opened connections. This happens because `nginx` load balancer receives a different number of connection requests, but the computational power remains the same, becoming stressed more and more following the increase of requests. In Fig. 2 we can observe maximum latency after three tests that run the benchmark for 30 seconds, and a constant throughput of 2000 requests per second. We increased the number of connections opened in every test but maintaining the same amount of used threads.

D. Horizontal Pod Scaling

A Kubernetes cluster requires to compute resources to run applications and these resources may need to increase or decrease depending on the application requirements. To deal with increasing requirements, such as high traffic, you can scale out your application by running multiple instances. In Kubernetes, this is equivalent to scaling a deployment to

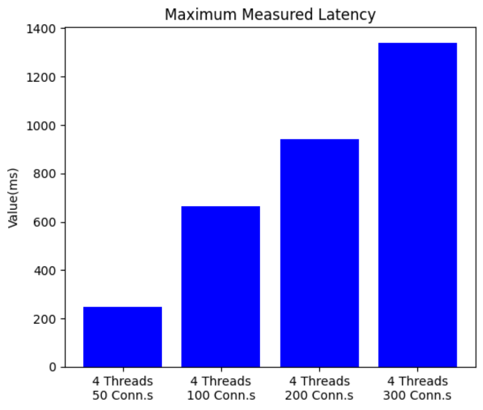

Fig. 2. Maximum Latency measured in three different tests with the same amount of Threads used

add more pods. When you scale an application, you increase or decrease the number of replicas. Each replica of your application represents a Kubernetes Pod that encapsulates your application's container(s). When traffic increases, we will need to scale the application to keep up with user demand.

Running multiple instances of an application will require a way to distribute the traffic to all of them. Services have an integrated load-balancer that will distribute network traffic to all Pods of an exposed Deployment. Services will monitor continuously the running Pods using endpoints, to ensure the traffic is sent only to available Pods. Scaling can have both positive and negative effects: if you increase the number of replicas of a pod that is not the bottleneck of the network and it is not stressed, performances can deteriorate because pods are not working correctly; if you increase the number of replicas of a pod which is the bottleneck and becomes very stressed after a certain workload, its performances can be improved, and latency distribution can be way better. In Fig. 3 we can see both improvements and worsen, a four threads HTTP load generator can be very stressful for certain pods instead of a two threads HTTP load generator, which pods can handle well with no replicas of the stressed pod(s). In this case, we increased at 2 the number of replicas of the most stressed pod(s) during Test A improving latency; and we increased at 3 the number of replicas of the most stressed pod(s) during Test B, despite it was not necessary, demonstrating the bad effect on latency. Test results refer to uncorrected latency, that is, measured without taking delayed starts into account. Choosing which stressed pod should be replicated can have different effects on latency, for example, replicating the `nginx` pod when a lot of connections are incoming, not only does not solve the problem but can make it worse, by admitting even more traffic into the system. Recognize this case is not always easy, as a utilization-based autoscaling scheme would scale out `nginx`, which appears saturated.

We measured the stress levels of pods deploying a metrics-server pod in the cluster, which uses millicores to describe CPU usage. Kubernetes CPU metrics are generally "expressed” in millicores/milliCPU, or 1/1000 of a CPU.

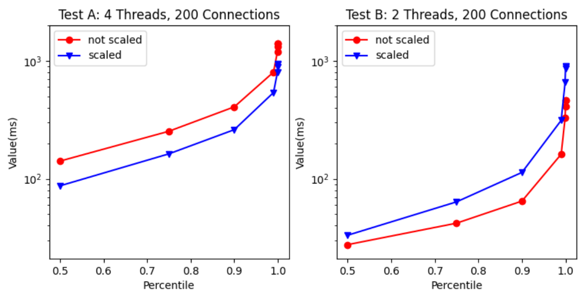

Fig. 3. Superposition of detailed percentile spectrum of the latency distribution

IV. Cloud Infrastructures Implementation

A. Reliability

We wanted to conduct experimental tests using a cloud infrastructure, Amazon Web Services [6] in this case, in order to study latency distribution differences when connectivity is significantly faster. Deploying the benchmark on a powerful cloud solution such as AWS implies that waiting time due to poor connection is not a problem anymore. An example is that downloading packages, downloading container images and container creating become a lot faster than before: with the connection used on the physical machine, container creating after the benchmark deployment takes several minutes; with AWS, nodes take a couple of seconds to create containers and make them ready and running. Now that we do nc have to care about bandwidth, nodes’ compute performance become fundamental in the latency distribution and the cluste performances depend on the singular instances dimension an specifics. AWS also improves security, managing the cluste access with SSH keys.

B. Deployment

We used kOps [7] to create a three nodes cluster (one maste node and two nodes), AWS is officially supported by this too. In order to build a Kubernetes cluster with kOps, we need to prepare somewhere to build the required DNS records. We used a Gossip-based cluster: it uses a peer-to-peer network instead of externally hosted DNS for propagating the K8s API address. This means that an externally hosted DNS service is not needed. Gossip does not suffer potential disruptions due to out of date records in DNS caches as the propagation is almost instant. Gossip is an easy option if you do not want to use Route53 services, so since there is no hosted zone for gossip-based clusters, you simply use the load balancer address directly.

In order to store the state of the cluster, and the representation of the cluster, we need to create a dedicated S3 bucket for kOps to use. This bucket will become the source of truth for our cluster configuration. All instances created by kOps will be built within ASG (Auto Scaling Groups), which means each instance will be automatically monitored and rebuilt by AWS if it suffers any failure.

In the cluster configuration, you can specify the instances types that will host the nodes, for the following studies, we used EC2 T2 instances: they are instances of expandable performance at the base level of CPU performance and can temporarily exceed it. According to AWS, T2 instances are a great choice for a variety of general-purpose workloads, including microservices, low-latency interactive applications. We used “micro” tier, with 1 vCPU and 1 GB of Memory for the master node, and two ”medium” tier, with 2 vCPU and 4 GB of Memory for the nodes. Each vCPU is a thread of an Intel Xeon core.

C. Cloud Resources Implications

With significantly better connectivity provided by AWS, latency behavior is very different from the first physical implementation: now latency distribution after several tests with the same number of threads used is not related anymore to the number of connections. Maximum measured latency remains about the same (it could even decrease) even with an increasing number of connections, although the nodes can still manage all the open connections very well, guaranteeing performance and speed in the computational phase. We can see in Fig. 4 that the cluster based on cloud infrastructures, as the number of connections increases, manages the load optimally, increasing the requests per second completed in proportion to the number of connections, but without increasing the maximum latency. Although the cluster on AWS is very

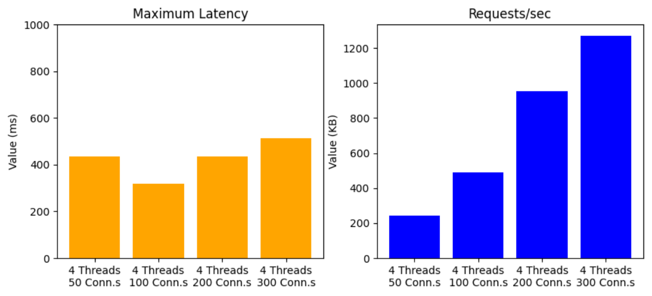

Fig. 4. Comparison between maximum measured latency and requests per second with an increasing number of open connections for each test

performing in terms of maximum latency, we cannot say the same about the latency measured before reaching 50% of the total generated load, that is, in the initial moments of the running test. Latency measured when the requests flow is still at the beginning is significatively high, much more than the measurements taken on the cluster deployed on the physical machine, although the connectivity is worse. In Fig.5 we compare latency at 50% of the load generated with wrk2 in both cloud based cluster and physical based cluster.

We understood that latency on cloud based system remains constant independently from the number of threads and connections opened by the incoming load, but it remains both constantly high or constantly low. We had these test results because the EC2 instances we used had small computational power so they didn't affect connectivity, which has been fully exploited on the basis of available resources.

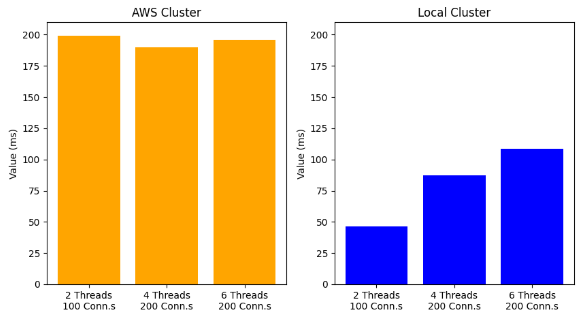

Fig. 5. Comparison between latency measured at 50% of total load on both AWS and local cluster

V. Conclusions

Microservice-based cloud applications can significantly improve the quality of services, in fact, many important companies have already decided to adopt cloud based solutions to develop and distribute their products. The management of microservices needs continuous trade offs to keep the quality of the service high, which over time can be increasingly automated, leading to a significant improvement in terms of computational power as well.

References

[1] DeathStarBench - https://www.csl.cornell.edu/~delimitrou/papers/2019.asplos.microservices.pdf
[2] Kubernetes - https://kubernetes.io/
[3] MicroK8s - https://microk8s.io/
[4] nginx - https://www.nginx.com/
[5] wrk2 - https://github.com/giltene/wrk2
[6] AWS - https://aws.amazon.com/it/
[7] kOps - https://kops.sigs.k8s.io/
[8] containerd - https://containerd.io/