My Services Got Old! Can Kubernetes Handle the Aging of Microservices?

José Flora, Paulo Gonçalves, Miguel Teixeira, Nuno Antunes
University of Coimbra, CISUC, DEI
Coimbra, Portugal
jeflora@dei.uc.pt, pgoncalves@dei.uc.pt, mteixeira@student.dei.uc.pt, nmsa@dei.uc.pt

***Abstract*—The exploding popularity of microservice based applications is taking companies to adopt them along with cloud services to support them. Containers are the common deployment infrastructures that currently serve millions of customers daily, being managed using orchestration platforms that monitor, manage, and automate most of the work. However, there are multiple concerns with the claims put forward by the developers of such tools. In this paper, we study the effects of aging in microservices and the utilization of faults to accelerate aging effects while evaluating the capacity of Kubernetes to detect microservice aging. We consider three operation scenarios for a representative microservice-based system through the utilization of stress testing and fault injection as a manner to potentiate aging in the services composing the system to evaluate the capacity of Kubernetes mechanisms to detect it. The results demonstrate that even though some services tend to accumulate aging effects, with increasing resource consumption, Kubernetes does not detect them nor acts on them, which indicates that the probe mechanisms may be insufficient for aging scenarios. This factor may indicate the necessity for more effective mechanisms, capable of detecting aging early on and act on it in a more proactive manner without requiring the services to become unresponsive.**

***Index Terms*—aging, kubernetes, fault injection, microservices**

I. Introduction

A huge number of companies are migrating or developing their systems following a microservice approach [1], taking advantage of cloud infrastructure and the on-demand availability of resources that allows systems to become more adaptive and autonomous [2], [3]. These systems tend to leverage software containers by using their lightweight and easy instantiation to provide more efficient operation and management of the system. Examples of container engines include Docker *(docker.com)* and LXC *(linuxcontainers.org)*. Still, the number of active microservices, and as a consequence containers, tends to grow to manually unmanageable amounts, thus resorting to the utilization of orchestrators is the *de facto*. This approach reduces costs and allows automation of deployment, monitoring, and management of both the microservices and the infrastructure supporting them.

Furthermore, they also provide some resilience properties that increase the ability of systems to withstand design or implementation faults and malicious threats, for instance, tolerating faults, improving availability or performance. There are multiple orchestrators available, **the most popular being Kubernetes, reaching 78% of the companies that use containers** in their infrastructure [4]. Nevertheless, there are others, such as Apache Mesos *(mesos.apache.org)* or Docker Swarm *(docs.docker.com/engine/swarm)*.

There is a relevant problem related to the aging of microservices which is underexplored, and can potentially cause significant damage in systems, most importantly business-critical systems. Software aging [5] has been researched in previous works, mainly focusing on monolithic applications [6], or traditional systems, but also with focus on virtualization platforms such as hypervisors [7] or container engines [8]. Kubernetes provides features that intend to increase the resilience of microservice-based systems but research attention has been limited. An example of such features are probes, that perform health checks on the services, and a study focusing on their capacity in aging detection would contribute to developers awareness regarding their effectiveness.

In this paper, the role of Kubernetes in detecting aging effects is studied and the use of faults to accelerate aging effects in microservices is proposed as a manner to have timely results for studies conducted in the context of microservices. For this, we perform an aging study on a representative microservice-based e-commerce application and monitor its resource consumption over a period of time, according to three scenarios where the load and conditions of the systems vary. We apply a normal load, a stress load, and the stress load with the presence of faults. These intend to provide different viewpoints of the system behavior and analyze the impacts of stressing microservices and using faults to accelerate these effects.

The analysis of the results demonstrate the existence of aging effects in the microservices that are most requested, with the continuous increase of resource consumption, especially memory, over the timeframe of the experiments. Kubernetes probes demonstrate not being very effective in the detection of problems in the system. The practical experiments show that the utilization of faults to accelerate aging in microservices is effective with the faulty services exhibiting faster aging, allowing the time span decrease of aging experiments in the context of microservice-based systems.

The rest of the paper is structured as follows. Section II presents background and the motivation for this work. Section III discusses background on microservice-based systems and Kubernetes probes, while Section IV describes the experimental study: with the experimental campaign and setup

978-1-6654-2603-9/21/$31.00 ©2021 IEEE
DOI 10.1109/ISSREW53611.2021.00042

used. Section V presents and discusses the results observed. Section VI condenses the lessons learned and threats to validity, Section VII reviews previous work and Section VIII concludes the paper.

II. Background and Motivation

Microservices have emerged as a common and well-accepted manner of devising, developing, and deploying systems that support several types of businesses. The main characteristics of their adoption remains connected to the flexibility and ease of utilization and management that is, typically, conducted through the use of orchestration platforms. Kubernetes is the most popular on the market [4], supporting diverse types of applications, from common web applications to business-critical systems. One of its key advantages is that it provides several features that improve and ease management efforts and time consumption with enough autonomy after the initial configuration.

Kubernetes provides by default the capability of detecting failing services and performing a restart so that they can start responding to customer requests again. As this is a very simple approach, and it cannot be configured, Kubernetes also provides Probes which are small snippets of code or commands used to continuously perform health verification and detect the occurrence of malfunctions. There are three types of probes available, discussed in more detail in the next section, that can increase the resilience of microservice-based systems [9]. The **startup probe**, used to assure that the application inside the container has started; the **readiness probe**, which serves to indicate whether the container is ready to respond to requests; and the **liveness probe** that indicates whether the container is running as expected. Upon the identification of a failure, the failing container is restarted according to the selected policy: every time a failure is identified; never; regardless of the failures; or only when the container exits with an error code, thus mitigating unexpected terminations.

The nature of such failures are also an important topic, that is, to discover which kinds of faults are responsible for failures detected by probes implemented in the system. As a kind of fault, the aging of systems has been researched in the past along with its impacts on system performance and the methods of rejuvenating the system affected with minimum impact on its availability. However, the majority of works have been focusing on system that do not follow the microservices approach hence, there is a need to research the topic.

The characteristics of microservices, described in more detail next, require more effective and sensible manners to swiftly detect and report the occurrence of aging. On the one hand, each microservice has a smaller pool of resources available when compared to traditional monolithic applications which requires higher sensitivity in the detection of aging effects to act upon the problem before the affected service becomes a bottleneck for the system. On the other hand, the experiments conducted before deploying the services cannot be extended during large periods of time as in traditional systems, where their duration could take days or even months.

Therefore, our work intends to experimentally evaluate the capacity of Kubernetes to detect and act on aging-related problems in microservices and use fault injection to accelerate the effects of aging in microservices so that it is possible to identify problems without disturbing the use of development and management pipelines.

III. Microservice-based Systems

Microservice-based systems are gaining ground in application design, development, and deployment [10], [11]. Emerging over recent years, they propose to break complex software into indivisible components that potentiate reuse and modularity, through leveraging communication between services. These resolve old concerns related to scalability and configurability of systems that many times were very hard to adapt to sudden modifications in the environment, as for instance user demand. Following, we detail some aspects of microservice architectures and Kubernetes features, which are commonly used to deploy and manage microservices.

A. Microservice Architectures

Microservice architectures intend to overcome limitations of monolithic applications, traditional software systems, characterised very coupled components incapable of executing independently [10], [12], [13]. The monolithic approach eases distribution and deployment as it only requires a package with all the components to be put in production, which results in lower modularity of the components produced as it is easy for them to communicate in a coupled manner and share snippets of code. Further problems arise with the age of the monolith, resulting in harder maintainability over time, with dependencies becoming more difficult to manage and maintain in order to work properly [12].

The resources provided to monoliths are mostly inappropriately handled, given that some components could require more memory while others could demand more disk speed or storage [12]. It becomes cumbersome to satisfy all the deployment configurations requirements in a manner to fit the whole application effectively. This leads to scalability limitations as monoliths were not correctly prepared to deal with replication and would hardly operate effectively under an increase in demand.

On the other hand, microservices emerged with the objective of mitigating the identified limitations. The main idea is to decompose the different parts of the monoliths into indivisible units of software that conduct a very well defined function [10]–[13]. These units, denominated microservices, cooperate to perform complex tasks that are requested to the system.

Aside from more effective management and technology adoption freedom, it is easier to have different views of the system and maintain them. For instance, when a microservice is updated, or a new version is released, it does not imply a complete reboot of the application, as it would with a monolithic approach. As microservices are small, they are easier and faster to instantiate or remove, therefore, potentiating the application of scalability and elasticity to provide resources to the system as they are needed to serve the customers requesting functionalities.

B. Kubernetes and Kubernetes Probes

Kubernetes is an open-source container manager that automates the management and deployment of applications. Based on a master-node architecture, it is through the master machine that manages all the node machines present in a cluster. The master is responsible for deploying all the containers composing the system into the available nodes, according to the configuration defined, and monitor their resource consumption. It can perform diverse operations to assure the healthy status of each component, as auto-scaling or container restart.

With regard to the protection of the services, Kubernetes provides three different probes whose objective is the assurance that applications are delivering the expected service and ready to respond to incoming requests [9]. As these mechanisms are potentially used in critically diverse applications, it is relevant to understand the underlying operation of these mechanisms, their advantages and drawbacks. The two major types of probes present in Kubernetes [9] are:

Readiness Probe: assesses whether or not a container is ready to take requests. On the event of failure, the endpoint controller removes the IP of the pod containing the failing container from all the endpoints that match it. A pod is considered ready when all of its containers are ready. Otherwise, it is removed from the service load balancers.

Liveness Probe: consistently conducts the probing of the container to assess whether it is running as intended. On the event of a failure detection, the container is killed and the restart policy controls the next operation performed. When a container is killed, a new one is created to replace it.

In practice, each probe can be defined using three different approaches: an HTTP request (e.g. a GET request); a TCP connection, where a connection is established to a specified port; and a user-defined command (e.g. check if a file exists).

After the container starts, the startup probe operates to validate the correct initialization of the container. Then, the readiness and liveness probes kick in. The readiness probe checks the ability of the container to receive and process requests. The liveness probe continuously performs the defined checks and acts by restarting the container if it detects that it is in a failing state.

IV. Experimental Study

Kubernetes supports diverse microservice applications, as business-critical systems, that require different levels of resilience and performance. It provides some features that allow to increase the resilience of systems by performing continuous monitoring and detecting malfunctions when they occur. Probes, as discussed, are an example of the features available. Their utilization is advised as they conduct constant verification and act upon the failing container to overcome the problems. Nevertheless, there is little information regarding their detection effectiveness in the identification of service failures when aging affects the microservice.

The objectives of this experimental study is two-fold: i) evaluate the detection effectiveness of Kubernetes probes of aging effects in microservices; and ii) study the acceleration of aging effects in microservices through the utilization of software faults.

For this, we devised an experimental campaign based on a representative microservice-based e-commerce system and applied three different scenarios, with normal operation, with stress testing, and with the utilization of an aging-related fault. The information obtained from this study would contribute to clearer picture of the protection granted by Kubernetes probes. Further, it also provides the possibility of understanding whether aging can be accelerated to usable timeframes within the typical constraints of development and deployment of microservice-based systems.

A. Experimental Campaign

The experimental study in this work is based on an experimental campaign that targets a representative microservice-based e-commerce system, composed of `5` microservices in three different scenarios. The different components of the experimental campaign executed are described and detailed in the following subsections.

1) Target Microservice System: For this work, we selected a representative microservice application that has been used in the past for several research works, the `TeaStore` [14] application, which was developed for the main purpose of serving as testbed in scientific works. Fig. 1 shows the architecture of `TeaStore`, which is composed of `5` services designed to discover themselves through a `Registry` component, to which each running service instance announces itself, as a way for the application to operate regardless of the active service replicas and their distribution.

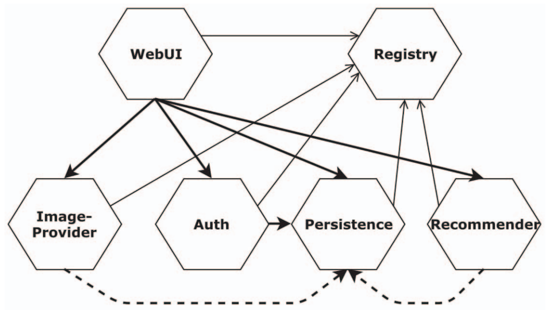

Fig. 1. `TeaStore` service architecture from [14].

The system is also composed by a database that stores all the relevant data and communicates only with the *Persistence* service. All `TeaStore` services are java-based with a common base image.

2) Workload: To exercise the application testbed, we used the Locust *(locust.io)* load testing framework. For the purpose of this work, we generated two workloads, a normal workload, that has an intensity below the maximum the system can respond to, and a stress workload, which oscillates but mainly remains above the system's acceptable levels of demand.

The normal load is a constant load with `9` active users generating requests over the duration of the experiment. The stress load is periodic, repeating itself every hour, and oscillates between `10` and `100` active users. Fig. 2 depicts the two load intensities for a one-hour period.

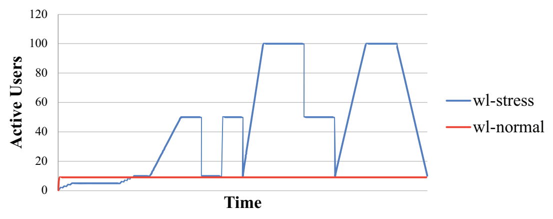

Fig. 2. The load intensities used for the two workloads.

Furthermore, the clients request profile for `TeaStore` was based on the *buy* profile given by the authors and emulates users browsing the store and conducting purchases of selected items. This profile is available in the public repository [15]. The load intensity was combined with the user behaviour profile to generate the workload exercising the system.

Other detail considered in this work is the injection of a fault that accelerates aging. The fault considered starts to manifest over time with the performance deterioration of the service. The service containing the fault is the `WebUI`, responsible for receiving the requests and forwarding them to the appropriate services that will perform the intended work, so that it can assemble the view presented to the user.

This is a medium frequency persistent fault. It is based on the description available on an industrial survey of microservices faults [16]. In microservices environments, many times, faults are complex and propagate across services, which means that a fault slowing down a service can impair the whole system given their reliance on each other.

3) Experimental Scenarios: To analyse the aspects of microservice aging that were the objective of this work, we devised three scenarios of operation, each with their own goal.

The **simple scenario** consisted of a normal operation of the system over `48h` exercised with the normal workload depicted in Figure 2.

The **stress scenario** executed over a period of `48h` with the use of a stress load that is represented in Figure 2, therefore overloading the system in order to provoke aging behavior under stressful conditions.

The **faulty scenario** is a merge of the stress scenario with the presence of a fault in the `WebUI` service of the application. This scenario runs for a shorter period of time, only `24h`, as it is enough to demonstrate the intended use of accelerating the effects of aging in microservices.

Each scenario intended to accomplish the following goals, respectively: i) have the baseline behaviour for the system; ii) obtain the effects of aging for the system under stressful conditions and iii) understand whether faults can be used to accelerate aging effects in microservices.

4) Measurements: To observe the behavior of the system, we collected multiple metrics. To understand the performance of the application under all the different conditions described previously, we collected information regarding the transactions executed by the system using the client load generator Locust. For the purpose of identifying aging in the services, we used Kubernetes Metrics *(github.com/kubernetes-sigs/metrics-server)*, which allows to collect CPU and memory metrics per service to understand the application usage of resources.

Furthermore, Events are a resource type in Kubernetes that are created when other resources have errors, changes in state, or other messages, and are useful to debug malfunctions in the system [17]. Because containers have a status that can be modified, we can use Kubernetes Events to log information regarding whether a probe check detected an erroneous state in its operation, where the container is considered to be *unhealthy*, otherwise it is considered *healthy*.

During the experiments, we used Kubernetes Events to extract information about the pods containing the status of the containers inside each pod and messages that indicate whether the probes have failed. Based upon this information it is possible to compute the total number of times a container became *unhealthy* for each service and the time of each occurrence. This information is useful to understand whether or not the probes were able to detect if and when a service was under problems in its operation.

B. Experimental Setup

We followed the deployment guidelines available at the application's GitHub repository [15] and deployed it into a Kubernetes cluster, which consisted of a master and a node machine. Services are instantiated through one active replica, with the exception of `WebUI` and `Persistence` which have two active replicas each.

The Kubernetes master has a processor with 3 cores and 16GB of RAM, with one worker node which runs with 4 cores and 16GB of RAM. Meanwhile the client that generates the workload has 5 cores and 16GB of RAM. All the machines are running Ubuntu and operate as depicted in Fig. 3.

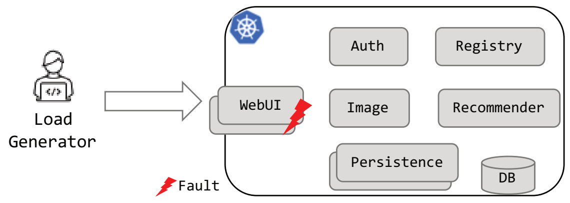

Fig. 3. Experimental Campaign Architecture Setup.

The Kubernetes master orchestrates the e-commerce application while all services are instantiated in the node machine. The application is exercised through the use of Locust load generator that is running in the client machine and communicating with the application.

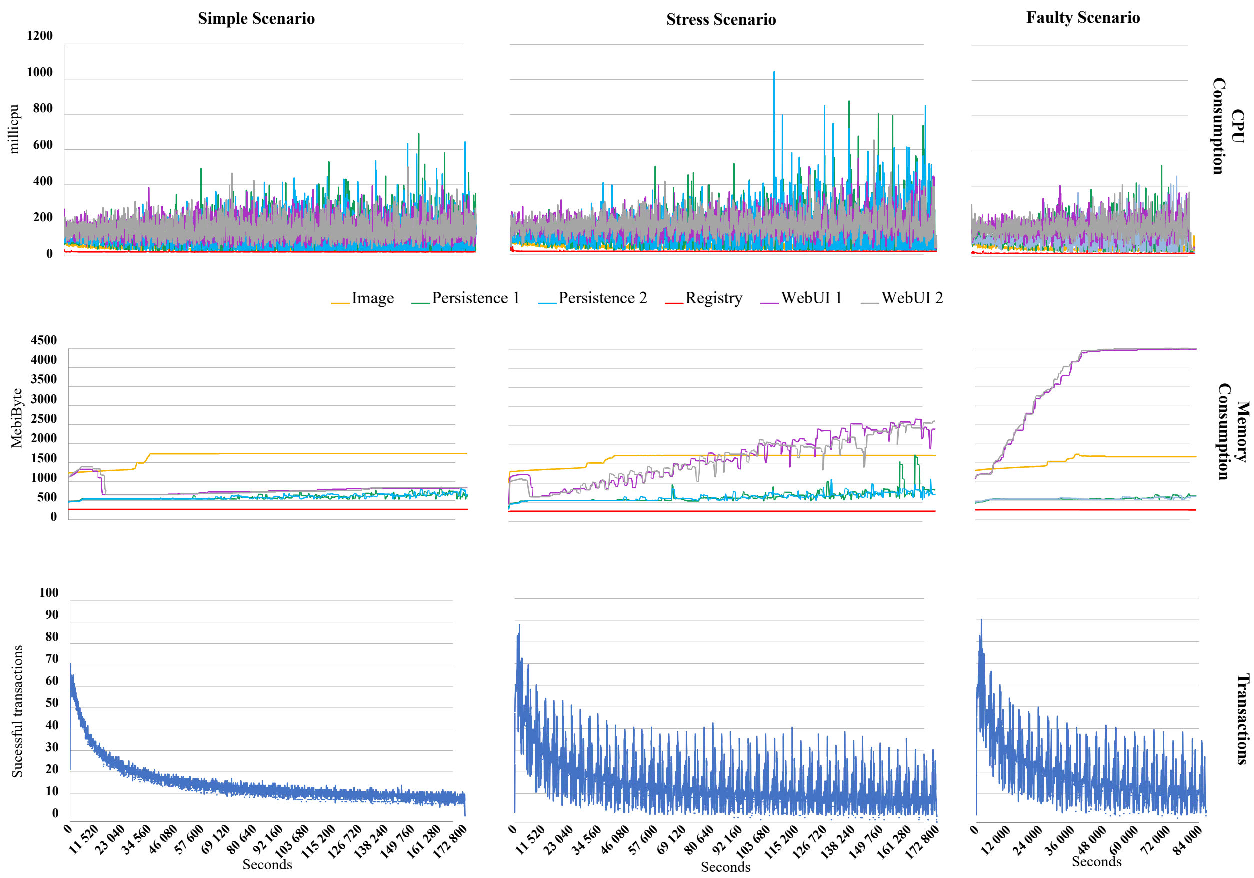

Fig. 4. Charts results matrix with the memory, CPU consumption, and transactions over time for each scenario. Top corresponds to CPU, middle to memory and bottom to transactions executed. From the left to the right we have: simple scenario; stress scenario; and faulty scenario.

V. Results and Discussion

This section presents and discusses the results and observations obtained during the experimental campaign. Fig. 4 presents the complete results of the experimental campaign. `Recommender` and `Auth and DB` services were suppressed as their behavior was similar to `Registry` and `Image` services, respectively, thus not adding much and allowing the visualization to be clearer. The charts presented in the upper line contain the information related to CPU usage, the middle line presents data from the memory consumption, and the bottom line presents data from the transactions executed by the system. The scenarios are presented from left to right: simple scenario, stress scenario, and faulty scenario.

Across all scenarios, the system is exercised with the corresponding workloads, as described in section IV-A. It is possible to observe a consumption of resources that denotes unsteadiness at the beginning and at the end of the experiments, consistent with the warm-up and cool-down phases. Thus, these phases are not considered in the analysis of the results obtained, instead focusing it on the steady phase of the system. Over this period, the microsevices are warmed-up and functioning as normally would in a real environment, receiving and responding to user requests.

With regard to CPU usage, the utilization is clearly very stable for each scenario analyzed, although some outliers and spikes are observed. However, in between scenarios there are clear variations. While the simple and faulty scenarios have all services with an upper bound around `200` millicpu, the stress scenario shows higher oscillations in the utilization of CPU across the services, especially in the final quarter of the experiment.

In the stress scenario, the spikes in some services (mainly `Persistence` and `WebUI`) are constantly increasing, crossing `400` millicpu by the 16^{th} hour of the experiment and `600` millicpu by the 30^{th}. It is noticeable the increasing oscillation of the CPU consumption in these services, which indicates more resources being used owing to the accumulation of user requests.

On the faulty scenario, there are few oscillation, similar to the simple scenario, with the CPU consumption hardly crossing the `400` millicpu value. Still, in the second halt there is a slight increase of oscillation to higher levels, as in the stress scenario, but is taking more time to achieve that range.

Overall, the CPU consumption is clearly affected by the stress load subjected to the system, while it remains more stable and with lower spikes over the simple scenario and with the presence of the fault used during the faulty scenario.

The observations of memory usage depict more dissimilarity between services. That is, there are some services, such as `WebUI` and `Persistence`, that suffer from the accumulation of memory over the period of the experimental campaign. Yet, the services have different degrees of accumulation, as `WebUI` replicas grow faster than `Persistence` replicas.

The analysis of the simple scenario shows very consistent memory usage across all the services. The `Registry` and the `Image` services remain very stable only showing slight increases over the period of experimentation. On the other hand, `WebUI` and `Persistence` have a trend to accumulate memory usage which indicates that these services are more prone to suffer from memory accumulation and aging effects. Still, the variation under normal workloads are slight and manageable as the memory use does not grow to significant values.

For the case of the stress scenario, we observe a similar behavior for all services except for `WebUI` and `Persistence`. When subject to stress, `WebUI` service instances grow their memory consumption from around `600` to close to `2500` MebiBytes each. This represents an increase above `300%`, which is a concerning factor when dealing with microservices that tend to be under stress for a period of time before scaling events take place. Regarding `Persistence`, although the increase is not as large, it still displays a slight increase in memory use. This may indicate that stressed services can suffer from aging if the load faced varies frequently.

With regard to the faulty scenario, although the experimentation period is shorter, we successfully achieve aging effects on microservices with higher intensity. The fault injected in the `WebUI` service increases the memory usage by more than `600%`, from around `600` to close to `4300` MebiBytes, in each replica, in just under `10h` of experimentation. Therefore, the use of faults can significantly reduce the time required to achieve higher levels of aging effects demonstration, which in turn makes the evaluation of mechanisms and tools that should detect aging on microservices more practical.

Regarding the successful transactions responded by the system, these vary according to the workload intensity used and it is possible to see the system having a tendency to respond to fewer transactions successfully over time, which can be a consequence of resource accumulation.

The mechanism for malfunction detection used in this work, in this case focused on aging effects on microservices, fell short, not detecting any problem with the services. Neither Kubernetes probes, nor the standard mechanism of detecting service malfunction, raise alarms for any problem in the system. Both the stress and the faulty scenarios demonstrate significant aging effects in the two service instances of the `WebUI`. In both cases, their memory usage is significantly higher than in the base case (simple scenario).

Overall, the main observations and conclusions of this experiments is that aging also affects microservices and that Kubernetes mechanisms fall short in detecting aging, thus not acting on their correction through service rejuvenation. The experiments demonstrate that a stress load can cause resource accumulation which can be accelerated through fault injection so that it is possible to have faster effects.

VI. Lessons Learned and Threats to Validity

The experiments and analysis conducted with focus on Kubernetes capacity to detect aging provided valuable insight and knowledge that can be very useful in decision-making processes and the experiments conducted in this context. In this section, we provide the lessons extracted from the work performed and also limitations that present threats to the validity of some findings.

A. Lessons Learned

Based on the analysis we performed of the work conducted, there are some lessons that we have drawn and are important findings.

Aging effects can be observed in microservices as in traditional monolithic applications. Although typically smaller in size and computing base, microservices tend to be stressed in periods prior to the activation of auto-scaling mechanisms as they are designed to operate in scalable and elastic environments, adapting to the level of request demand through the use of replication. The increase in the number of active replicas allows to reduce the stress faced by older microservices, still if not addressed the stress effects can accumulate and impair their normal operation.

The utilization of faults in microservices can reduce the time period required to observe the aging effects. The fault used in this work has the possibility to be configured and used to accelerate the microservice resource consumption at different rates, according to need. Faults can therefore accelerate the testing process before deployment and reduce the risk of compromising the performance, and even security, of the complete system. One slow microservice, depending on its criticality to the system, can become a bottleneck or a source of resource waste owing to the aging effects. Thus, the utilization of faults that stimulate aging can be very useful to identify bottlenecks or timely evaluate microservice aging detection mechanisms.

Kubernetes mechanisms were unable to detect the aging of the microservices affected. During our work, Kubernetes did not raise any alarm with regard to the health status of the services that demonstrated aging, mainly through memory usage accumulation. This factor is important to keep in mind when adopting these mechanisms to operate in case of aging and it shows that methodologies and tools to improve microservice aging detection effectiveness are required in the future.

B. Threats to Validity

Even considering the interesting findings, some details are worth noting as threats to the validity of our work.

The experiments executed in this work were not repeated and there are some aspects that could vary among executions as for instance network latency. However, these have small impact on the observations as the duration of the runs for each scenario is quite extended (`24h` and `48h`). Thus, the final results are expected to be very similar, regardless.

A limitation that can be raised is that only one testbed was used to perform the experiments. Other testbed with different technologies, such as SockShop [18], would contribute and improve the generalization and the validity of the results. Nevertheless, TeaStore is a representative testbed, used in multiple contexts, and therefore can be considered adequate for the generalization we are trying to achieve.

Also, in this work only one fault was used, that was focused on memory consumption and management. Yet, it is not an intrusive fault, being realistic and based on what could be found in real world scenarios, and is in fact based on prior work of an industrial survey in the context of microservice-based systems [16]. In this way, we intend to mitigate this concern and present a representative fault.

VII. Related Work

Software aging can be defined as the accumulation of resource utilization owing to faults present in the software released to production [5]. These faults, which are hard to detect during testing, have an effect on the system over time, tackling the performance, resulting in higher response times. It is very improbable to not have faults in large codebases, owing to time or budget constraints, thus aging can occur from the accumulation of faults' effects. Rejuvenation is the process of aging mitigation, or effect reduction, and, for instance, can be performed through restarting an application with the intent of returning the software to a clean state.

Software aging has been researched and several works are available, being applied to some real case scenarios [19]. Web servers were studied with different models used to monitor the system [20]. This work was then extended, focusing on Apache, concluding that when subjected to overwhelming workloads there are traces of aging but also minor rejuvenation [6].

Lately there has been research focused on virtualization platforms. Docker aging effects were evaluated and it was possible to observe aging effects [8]. Other work demonstrated problems not directly connected to Docker but it would also affect the ability to create new containers [21].

Kubernetes has been a topic of research owing to increased use in industry, still mainly focusing on scalability. Several features are provided to address software problems in an autonomous manner, as for instance, the Horizontal Pod Autoscaler (HPA) that allows the autoscaling of services according to demand. Previous to scale up, it is common for the service to be under stress load for a short period of time, to better use the resources available and not frequently perform scaling up or down. HPA has been extensively researched and improvements have been proposed allowing developers for better results, with higher flexibility and effectiveness [22]. Probes are an example of other approaches to monitor services and identify malfunction, which perform periodical health checks on the corresponding services [9].

In microservice applications, various services cooperate to provide the intended goal of the application. Most of the work done focuses on container deployments [23], [24], with a substantial emphasis on Kubernetes; but also on performance and the implications of load balancing and memory reservation size, finding that different setups require distinct attentions [23].

However, micoservice aging has not received significant attention. Previous work performs a preliminary study on the subject, applying known principles of aging to microservices and observing the results [25]. To detect aging, a deep learning model was applied and proved to be effective. Still, it does not go too deep into the root cause, only proving that it is a relevant subject that needs more attention.

VIII. Conclusions

The work presented demonstrates that aging can affect microservices and exacerbate their resource consumption. Also, Kubernetes probes do not show the ability to detect aging-related problems in the services monitored. This factor indicates that work in more effective approaches for early detection of microservices aging is needed.

A more effective approach would be improving the probing mechanism or devising a methodology that can leverage other data sources, such as service data, to proactively identify aging. Furthermore, it would contribute to the prevention of microservice failures that many times can cause problems at the system level. Besides, injecting faults to accelerate aging results in shorter periods of time required to evaluate the effectiveness of such mechanisms and allow timelier conclusions.

Acknowledgments

This work is supported by the Portuguese Foundation for Science and Technology (FCT) through the Ph.D. grant `SFRH/BD/05145/2020`. It is also supported by the project AIDA – Adaptive, Intelligent and Distributed Assurance Platform (`POCI-01-0247-FEDER-045907`), co-financed by the European Regional Development Fund (ERDF) through the Operational Program for Competitiveness and Internationalisation (COMPETE2020) and by the Portuguese FCT under CMU Portugal.

References

[1] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov, "Microservices: The Journey So Far and Challenges Ahead," *IEEE Software*, vol. 35, no. 3, pp. 24–35, 2018.
[2] S. Sultan, I. Ahmad, and T. Dimitriou, "Container Security: Issues, Challenges, and the Road Ahead," *IEEE Access*, vol. 7, pp. 52 976–52 996, 2019.
[3] P. Mell and T. Grance, "The NIST Definition of Cloud Computing," *Computer Security Division, Information Technology Laboratory, National*, 2011.
[4] S. J. Vaughan-Nichols, "Kubernetes jumps in popularity," zdnet.com/article/kubernetes-jumps-in-popularity, 2020.
[5] D. L. Parnas, "Software aging," in *Proceedings of 16th International Conference on Software Engineering*. IEEE, 1994, pp. 279–287.

[6] M. Grottke, L. Li, K. Vaidyanathan, and K. Trivedi, "Analysis of software aging in a web server," *IEEE Transactions on Reliability*, vol. 55, no. 3, pp. 411–420, 2006.

[7] L. Beierlieb, L. Iffländer, A. Milenkoski, C. F. Gonçalves, N. Antunes, and S. Kounev, "Towards testing the software aging behavior of hypervisor hypercall interfaces," in *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2019, pp. 218–224.

[8] M. Torquato and M. Vieira, "An experimental study of software aging and rejuvenation in dockerd," in *2019 15th European Dependable Computing Conference (EDCC)*. IEEE, 2019, pp. 1–6.

[9] Kubernetes, "Configure liveness, readiness and startup probes," kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes, 2020.

[10] Martin Fowler and James Lewis, "Microservices," 2014. [Online]. Available: https://martinfowler.com/articles/microservices.html

[11] C. Pahl and P. Jamshidi, "Microservices: A Systematic Mapping Study:," in *Proceedings of the 6th International Conference on Cloud Computing and Services Science*, 2016, pp. 137–146.

[12] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: yesterday, today, and tomorrow," *arXiv:1606.04036 [cs]*, 2017. [Online]. Available: http://arxiv.org/abs/1606.04036

[13] S. Newman, *Building Microservices: Designing Fine-Grained Systems*. "O'Reilly Media, Inc.", 2015.

[14] J. von Kistowski, S. Eismann, N. Schmitt, A. Bauer, J. Grohmann, and S. Kounev, "TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research," in *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2018, pp. 223–236.

[15] D. Research, "DescartesResearch/TeaStore," github.com/DescartesResearch/TeaStore, 2018.

[16] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding, "Fault Analysis and Debugging of Microservice Systems: Industrial Survey, Benchmark System, and Empirical Study," *IEEE Transactions on Software Engineering*, 2018.

[17] K. Jackson, "Types of Kubernetes Events," bluematador.com/blog/kubernetes-events-explained, 2020.

[18] Weaveworks, "Sock shop : A microservice demo application," github.com/microservices-demo/microservices-demo, 2017.

[19] M. Grottke, R. Matias, and K. S. Trivedi, "The fundamentals of software aging," in *2008 IEEE International Conference on Software Reliability Engineering Workshops (ISSRE Wksp)*, 2008, pp. 1–6.

[20] L. Li, K. Vaidyanathan, and K. Trivedi, "An approach for estimation of software aging in a web server," in *Proceedings International Symposium on Empirical Software Engineering*, 2002, pp. 91–100.

[21] F. Oliveira, J. Araujo, R. Matos, L. Lins, A. Rodrigues, and P. Maciel, "Experimental evaluation of software aging effects in a container-based virtualization platform," in *2020 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, 2020, pp. 414–419.

[22] A. Abdel Khaleq and I. Ra, "Agnostic approach for microservices autoscaling in cloud applications," in *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*, 2019, pp. 1411–1415.

[23] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara, "Serverless computing: An investigation of factors influencing microservice performance," in *2018 IEEE International Conference on Cloud Engineering (IC2E)*, 2018, pp. 159–169.

[24] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar, and M. Steinder, "Performance evaluation of microservices architectures using containers," in *2015 IEEE 14th International Symposium on Network Computing and Applications*, 2015, pp. 27–34.

[25] J. Yue, X. Wu, and Y. Xue, "Microservice aging and rejuvenation," in *2020 World Conference on Computing and Communication Technologies (WCCCT)*, 2020, pp. 1–5.