Software Versioning with Microservices through the API Gateway Design Pattern

Akhan Akbulut*
Department of Computer Engineering
Istanbul Kültür University
Istanbul 34158, Turkey
a.akbulut@iku.edu.tr

Harry G. Perros
Department of Computer Science
North Carolina State University
Raleigh, NC 27606, USA
hp@ncsu.edu

***Abstract*—The microservices architecture is a relatively new approach in implementing service-oriented systems. This cloud-native architectural style enables the implementation of loosely coupled, agile, reuse-oriented, and lightweight services instead of monoliths. It also eliminates vendor and/or technology lock-ins. A modification to a small code segment for monoliths may require the building and deployment of a completely new version of the software. However, the modular form of microservices allows solving software versioning in a polyglot manner. In this paper, we extend the well-known microservice design pattern API gateway with a view to managing the virtual hardware configuration of containers. Specifically, using the proposed approach, the service capacity in the requested version of the service is orchestrated adaptively in compliance with a service-level agreement. In our tests, we found that the proposed version management approach reduced the hosting cost by 27% compared to static or rule-based scaling.**

***Index Terms*—Microservices, Software Versioning, Version Management, API Gateway, Design Pattern, Container Sizing, Scaling**

I. Introduction

As software development practices continue to evolve, the debate of using microservices to migrate traditional monolithic architectures will only become more pronounced. Microservice architectures allow developers to split applications into distinct independent services, each having individual logic that can be maintained and served by the different development team.

RESTian applications are by far the most prominent architectural style used today to expose services for managing requests from multiple channels. It utilizes the power of HTTP instead of more complex protocols like RPC or SOAP. However, updating the services can be a challenge if an accurate versioning method is not employed by the development team. At fined-grained level, software versioning should be supported with intelligent mechanisms.

In the case of governing multiple versions of the same service in the ecosystem, a dynamic management mechanism should be employed. For this reason, we have developed a solution that can dynamically administrate the version control and execute the scaling management of services within the environment. The contribution of this paper is three-fold, as follows:

i. We compiled the state-of-the-art versioning approaches for microservices architecture currently practiced in the software industry.
ii. We showed that version control can also be used to improve the scaling process of microservices.
iii. We proposed an adaptive API versioning scheme that reduces the hosting costs of the microservices ecosystem.

The remaining of this paper is organized as follows. In Section 2, we explain software versioning in microservices architecture. Section 3 presents the key ideas and methodology of our proposed approach. Section 4 reports its performance with thorough experimental tests. Section 5 concludes the paper and discusses future works.

II. Software Versioning with Microservices

The world of today hosts digital systems whose requirements change frequently. Software versioning is used to respond rapidly to these superseding requirements without service interruptions. In addition, version control plays an important role in software projects developed by multiple teams that are constantly changing/updating source codes [1]. In systems that continuously evolve, the provision of different versions for the same service or module is considered an anti-pattern and bad practice [2]. However, in some cases, more than one version of the same service may be required. For example, for users who cannot upgrade their mobile device's operating system to a newer version due to hardware or developer limitations, old generation mobile application services are be offered with the new release services as well. For a variety of reasons, different versions and forms of the same services may be required for different platforms. If the ecosystem with such requirements is not orchestrated by some complex control mechanism, the delivery and maintenance of the service maybe problematic.

The software industry desires to build systems that can be managed at the component level to reach highest degree of maintainability and scalability. Microservices [3]–[5] now recommends the use of lightweight, independently deployable, and API-based services as the most up-to-date presentation of the service-oriented architecture (SOA). This approach also

*A. Akbulut is also with the Department of Computer Science, North Carolina State University, Raleigh, NC 27606, USA, email: aakbulu@ncsu.edu

offers high applicability for version management. Different versions of the same component can be offered with employment of microservices in a coordinated manner. API versioning has two different approaches to meet every aspect of software requirements.

- **Versioning in the URI :** This approach is semantically meaningful since it uses the version information in the Uniform Resource Identifier (URI). A simple example of this might look like *http://v1.example.com/service/* or *http://api.example.com/service/v1/*. The representation of an API is immutable, and a fresh URI space needs to be created, such as, *http://api.example.com/service/v2/*, with the publication of a new version. Netflix uses a different form of URI versioning including query strings like *http://api.netflix.com/catalog/titles/movies/70115894?v=2.0.*. This allows the development team to update a single resource, instead of the full API. The primary disadvantage of using URI versioning is dealing with a very large URI footprint which may become unmanageable in the long run. Also, there is no easy way to simply evolve a single resource which results with inflexibility.
- **Versioning in the HTTP Header :** If the version information in the URI is not intended to be displayed, a version-free URI can be offered by providing custom headers of HTTP like *Request Header: Api-version: 2* or *Accepts-version: 1.0*. With this approach, the URI is clean and not cluttered with versioning parameters as proposed in the URI versioning scheme. Utilizing header versioning allows services to be updated with a high degree of transparency, and end-users can migrate to new versions easily. In the same way the Accept Header spec can be modified for different custom vendor media types, and for parameters to be passed to create a content negotiation action. The most common problem of this approach is dealing with caches and proxies. The *Vary HTTP header* must be used for both client and the server in order to eliminate caching-related problems. Also, if the requests are not carefully constructed, routing faults may arise. Compared with URI-versioned APIs, the header versioning technique outputs less accessible artifacts and it makes it more difficult to test and debug an API using a browser.

Many developers prefer to employ version identifiers in URIs instead of HTTP headers, because of the convenience of using URIs without headers, especially in the browser. But the only thing that does not change is that the services offered in the back-end are accessed through an API gateway pattern as shown in Figure 1.

Each API versioning strategy has its own cons and pros regarding feasibility, deployment plan, client attributes, and server capacity. No matter which approach is preferred, version numbering has a three digit general semantic like $x.y.z$ where x corresponds to major, y minor, and z patch revisions. A major revision is applied when the development team decides to make changes that are not compatible with the previous version. Minor revisions are for improvements or optimization of resources in a backward-compatible manner. Therefore, in case of need, requests can be redirected to services with minor revision differences. Finally, patches are applied to fix bugs or defects of the components.

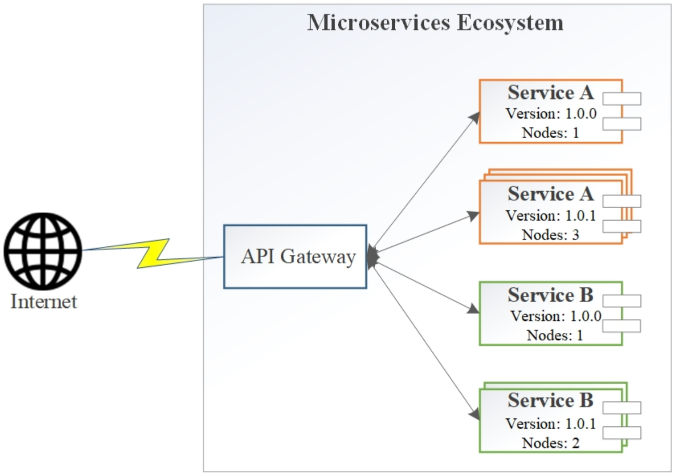

Fig. 1. Microservice API Versioning

III. Methodology

Our proposed methodology extends the API Gateway design pattern of the Microservices architecture along with modifying the Gateway entity by installing several functionalities, such as, intelligent routing, observing other ecosystem entities, and scale up or down services based on fuzzy logic. Traditional Gateway entities are mostly responsible for filtering spam calls, routing the requests to proper back-end services, circuit breaking, and offloading [6], [7]. From this perspective, our proposal increases the load on the Gateway, but it offers an alternative solution to version management.

The fuzzy-based API Gateway operates a logic of two input variables and one output variable. The two input variables are a) the number of requests from the clients and b) the CPU utilization of the back-end microservice that houses the requested version. The first input is a non-negative integer, and the second one is expressed as a percentage. The output variable indicates the action that should be taken, and it takes the values $\langle N^{--}$, N^{-}, N^{0}, N^{+}, and $N^{++}\rangle$. N^{+} and N^{++} means that one respectively two additional instances of the requested service should be deployed in order to cope with the overload. N^{0} means no action should be taken. That is, the load has not changed and the current configuration should not changed. In the opposite direction, N^{-} and N^{--} means that one respectively two unused nodes should be removed from the ecosystem. We note that we only focus on horizontal scaling as a way of adjusting the capacity of microservices. We did not consider vertical scaling achieved by increasing or decreasing the CPU and RAM capacities of existing microservices. Unlike virtual machine virtualization [8], container virtualization can deploy microservices within a few seconds.

The primary input, the CPU utilization, is a parameter that directly affects energy consumption. Barroso et. al. [9] investigated the employment levels of CPU in data-centers and found that processors operate mostly within a utilization range of 10% to 50%. The reason for adopting these levels is to benefit from the use of Dynamic Voltage and Frequency Scaling (DVFS) power management mechanisms that provide significant energy reductions (up to 40%) and power savings (up to 20%) [8]. Based on these findings, we define the membership functions: LI (Light), ID (Ideal), ST (Strong), and IN (Intense), as shown in Fig.2. Since we want to benefit from low level CPU utilization, we have identified the use of more than 40% as a ST and IN situation.

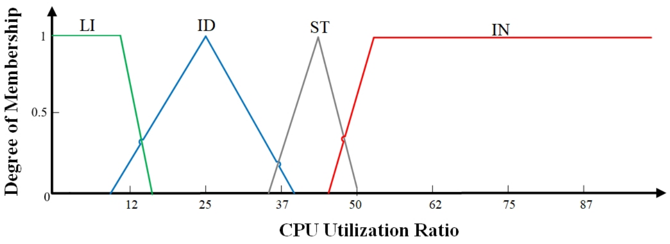

Fig. 2. Membership functions of the input variable CPU utilization

The other system input, the number of requests, vary over time and if the service capacity is not enough to handle the load, then there may be long processing times. On the other hand, keeping a higher number of services than necessary, yields an unwanted hosting cost. To determine the ideal load for a single microservice we conducted several experiments on the Virtual Computing Lab (VCL) of North Carolina State University [10]. With reference to our experiments we defined the membership function of the number of requests as ID (Ideal), HE (Heavy), EX (Extreme), and MA (Maximal), as shown on Fig 3.

With the proposed method, we aim at keeping the service time as committed in the service-level agreement (SLA) while at the same time minimizing the hosting costs.

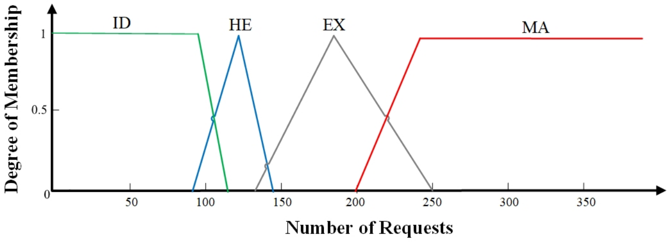

Fig. 3. Membership functions of the input variable number of requests

Table I describes the relationship between two input parameters, CPU utilization and the number of requests, and output parameter, scaling action. Using this table fuzzy production rules can be obtained to run the execution logic.

IV. Results

To evaluate the proposed methodology, we developed two back-end services for a Unity based mobile application. Both services were deployed using the Docker environment [11]. The first version is designed for the Android 6v.0 Marshmallow and the second is for the Android v8.0 Oreo mobile operating system. Since the second service is not compatible with older versions of Android users, their requests are forwarded to the first one. Both services are developed with NodeJS platform supported with MongoDB document-oriented databases. The enhanced API gateway entity was enhanced with the fuzzy-based auto-scaling feature so that to manage the back-end microservices. In order to generate the demand, we have also developed a client application that generates requests to these back-end services with different densities at different times. The proposed fuzzy-based auto-scaling scheme is compared with a static configuration with manual-scaling.

TABLE I
Knowledge base for the Fuzzy-system

	LI	**ID**	**ST**	**IN**
ID	N^0	N^0	N^0	N^+
HE	N^0	N^+	N^+	N^{++}
EX	N^-	N^0	N^+	N^{++}
MA	N^{--}	N^-	N^+	N^{++}

We conducted three experiments, each lasting for 5 hours, for each of the three scaling techniques, ie., manual, rule-based, and fuzzy-based. In the case of manual scaling, an administrator monitors the load of the microservices and scales up or down the services accordingly. For auto-scaling, a rule-based approach is implemented with certain thresholds. Finally, our fuzzy-based scaling technique orchestrates the ecosystem via an API gateway microservice. For each experiment, the client application created the same synthetic demand for the microservices, which was in the form of concurrent HTTP requests. The results obtained are shown in Figure 4. The y-axis gives the capacity of the system, expressed in number of users that it can serve, as it is modified by a scaling technique over time. It is obtained by calculating the number of active containers and then multiplying it by 50. The blue line shows the demand in terms of number of users.

Our experimental scenario is based on a system that serves around 300 users on average and 500 to 600 users at peak-times. Since we used a container with 2 vCPU and 4GB RAM which is capable of serving 50 users at a time, we decided that we wanted no fewer than 7 nodes and no more than 50 in the auto-scaling policy. Ultimately, unlike the manual-scaling, the auto-scaling approach can prevent service loss during the traffic spike which hits around 13:00 hours by launching additional service instances on time. Manual-scaling missed the peak-traffic and caused slow service so that some users could not be served. After the peak, the administrator cannot react on time to reduce the number of instances and produced unnecessary hosting costs. Our proposed scaling technique performed better in scaling back-end microservices

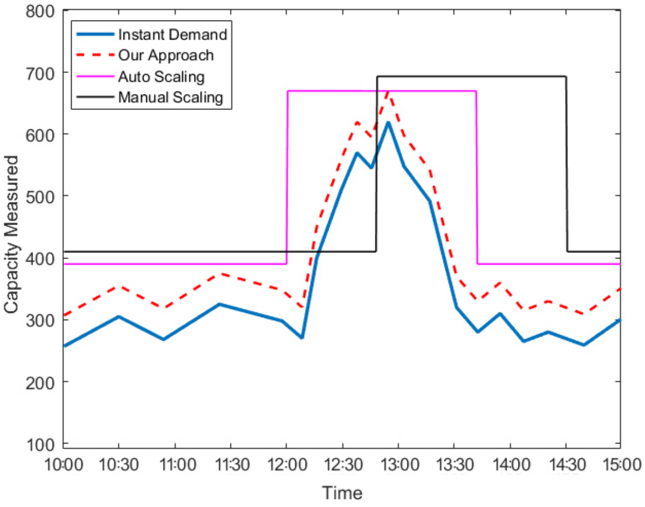

Fig. 4. Scaling of the Microservices Over Time

in changing demand. It scaled services more responsively than the auto-scaling option. The prominent feature of our approach is that it does not need to know the intensity of the load on the back-end microservices. As all traffic passes through the API gateway, it knows instantaneously the service capacity of all entities in the ecosystem. For the auto-scaling or dynamic-scaling technique, certain metrics of all services such as CPU, memory, and network utilization, must be continuously monitored. There is a late reaction of the auto-scaling service, because it has a 5-minute refresh interval. Also, another disadvantage is that the health check process generates an overhead and sometimes it is not possible to work during the time the system is overloaded. This may cause the system to drop that node because the health check is not able to return the result.

During the experiments, we aimed to serve the clients between 20 and 30 ms per request. In accordance with this bound, we designed to operate the ecosystem with minimum hosting costs. As can be seen from Table II, our proposed approach and auto-scaling meet the SLA requirements. However, the average service time is longer with manual-scaling due to its inability to scale up microservices in time. In addition, the energy consumption is increased with aggregated CPU utilization.

TABLE II
EXPERIMENTAL RESULTS

Scaling Technique	Response Time (avg)	Hosting Cost
Manual-Scaling	41ms	48% more
Auto-Scaling	24ms	27% more
Our Approach	23ms	-

In terms of hosting costs, calculated using AWL pricing, our approach offered the lowest run time cost. The auto-scaling technique has a 27% more costly hardware allocation. We observed that the manual-scaling approach is not applicable to systems with a varying demand. It allocated 48% more resources for microservices for the same scenario.

V. CONCLUSION

Because mobile users update their applications at different frequencies, versioning of APIs become more important than others. With several different versions of the application running in the live, the server needs to consolidate and handle the various requests coming in from new and legacy users alike. Sizing the configuration of APIs for different versions is critical and auto-scaling systems should be deployed in order to orchestrate the requests for different versions. Otherwise, systems could crash during irregular traffic patterns or the server load spikes at unexpected times. In this paper, we proposed an API versioning scheme that reduces the hosting costs of the microservices ecosystem with employing fuzzy-logic in adjusting service capacity as needed. The well-known API gateway design pattern is enhanced to orchestrate the requests for different versions of APIs and compared with auto-scaling technique our proposed approach runs the ecosystem with 27% less hosting cost. It provides a truly hands-off approach to scaling while ensuring that demand from the users is met in a timely fashion. Microservices have gained prominence as the most recent form of SOA and with the employment of microservices architecture, the software versioning is easier to implement than ever. But in order to run the different versions together in harmony, scaling and version management should be realized with a resourceful approach. As a continuation of this research, we are planning to implement a neuro-fuzzy [12] routine to dynamically update the knowledge-base as the requirements and systems change over time.

REFERENCES

[1] J. Loeliger and M. McCullough, *Version Control with Git: Powerful tools and techniques for collaborative software development*. "O'Reilly Media, Inc.", 2012.

[2] S. Fowler, *Production-Ready Microservices*. "O'Reilly Media, Inc.", 2016.

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

[4] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: yesterday, today, and tomorrow," in *Present and Ulterior Software Engineering*. Springer, 2017, pp. 195–216.

[5] O. Zimmermann, "Microservices tenets," *Computer Science-Research and Development*, vol. 32, no. 3-4, pp. 301–310, 2017.

[6] I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen, *Microservice architecture: aligning principles, practices, and culture*. O'Reilly Media, Inc., 2016.

[7] S. Newman, *Building microservices: designing fine-grained systems*. O'Reilly Media, Inc., 2015.

[8] B. Familiar, *Microservices, IoT and Azure: leveraging DevOps and Microservice architecture to deliver SaaS solutions*. Apress, 2015.

[9] L. A. Barroso, J. Clidaras, and U. Hölzle, "The datacenter as a computer: An introduction to the design of warehouse-scale machines," *Synthesis lectures on computer architecture*, vol. 8, no. 3, pp. 1–154, 2013.

[10] H. E. Schaffer, S. F. Averitt, M. I. Hoit, A. Peeler, E. D. Sills, and M. A. Vouk, "Ncsu's virtual computing lab: A cloud computing solution," *Computer*, vol. 42, no. 7, 2009.

[11] "Docker - enterprise container platform," [Online]. Available https://www.docker.com/, May 2019.

[12] K. Shihabudheen and G. Pillai, "Recent advances in neuro-fuzzy system: A survey," *Knowledge-Based Systems*, vol. 152, pp. 136–162, 2018.