Challenges in service discovery for microservices deployed in a Kubernetes cluster – a case study

Baasanjargal Erdenebat, Bayarjargal Bud, and Tamás Kozsik

Abstract—**With Kubernetes emerging as one of the most popular infrastructures in the cloud-native era, the utilization of containerization and tools alongside Kubernetes is steadily gaining traction. The main goal of this paper is to evaluate the service discovery mechanisms and DNS management (CoreDNS) of Kubernetes, and to present a general study of an experiment on service discovery challenges. In large scale Kubernetes clusters, running pods, services, requests, and workloads can be substantial. The increased number of HTTP-requests often result in resource utilization concerns, e.g., spikes of errors [24], [25]. This paper investigates potential optimization strategies for enhancing the performance and scalability of CoreDNS in Kubernetes. We propose a solution to address the concerns related to CoreDNS and provide a detailed explanation of how our implementation enhances service discovery functionality. Experimental results in a real-world case show that our solution for the CoreDNS ensures consistency of the workload. Compared with the default CoreDNS configuration, our customized approach achieves better performance in terms of number of errors for requests, average latency of DNS requests, and resource usage rate.**

Index Terms—**Microservice, container, service discovery, CoreDNS, Kubernetes**

I. Introduction

With the continuous development of technology, software systems become larger and more complex, and the architecture of their code must adapt to these changes in order to enable the handling of the increased complexity. The application of a traditional, monolithic architecture is becoming less attractive and less useful in many business scenarios. In a monolithic architecture, any modification made to a small feature necessitates the recompilation and redeployment of the entire application, resulting in long iteration cycles, which is clearly unfavorable. On the contrary, microservices provide an approach to developing a single application as a collection of small services. Each service operates in its own process and communicates through a lightweight mechanism. Transitioning from monolithic applications to microservices requires a significant shift in software design, but it offers numerous advantages, particularly when combined with the introduction of containerization technologies. The microservice-based application code is small, independent, easy to manage, compile and deploy, and allows short development iteration cycles [1], [2]. Due to the small volume of deployment packages, fast service start-up and resource recovery, and easy to achieve flexible scaling, it can well meet application scenarios with high concurrency and large fluctuation in load [3]. The integration of containerization technology and the microservice architecture significantly enhance the performance and efficiency of information systems [4], [5]. We leverage microservice architecture and containerization technology to develop business applications. It allows us to maintain frequent software development and deployment cycles while ensuring consistent and high-quality product delivery. Furthermore, the usage of those technologies facilitates smooth adaptation to the continuous integration and continuous development (CI/CD) methodology and cloud native development [6], allowing for agile adjustments to accommodate evolving business requirements.

In a real world industrial scenario, our team embarked on a Spring Cloud-based DevOps platform project with the objective of gradually transitioning from a monolithic architecture to a microservice-based one. To accomplish this, the entire system needed to be migrated from hypervisor to container virtualization, which allowed us to build a highly available cluster. Over time, we have progressively transitioned our services from a monolithic architecture to a Spring Cloud-based one. In the environment, numerous microservices are constantly being created and perished, while they are making calls to each other. Therefore, a component that specifies the location of a given microservice is needed. This component is called *Service Discovery*. Spring Cloud implements Service Discovery through Eureka [7]. When shifting from VM-based microservice applications to containerized ones, there is a need of using Eureka for service discovery due to the communication between the two different infrastructures: service discovery operates smoothly via Eureka on co-existing infrastructures of VM and Kubernetes. However, it exhibits performance limitations, particularly in the large-scale.

The contributions of this paper are the two-fold. First, we introduce the idea of an easy-to-use dynamic service discovery functionality during the migration from virtual machines to a containerized cluster. Then we provide a detailed evaluation of the service discovery process in container-based clusters. We offer solutions and strategies for service discovery issues and for ensuring smooth operation of Kubernetes clusters.

The rest of the paper is structured as follows. Section II is a brief overview of the background knowledge on service discovery, microservices and containerization technology. Section III introduces the pre-experimental activities including the methodology employed in our study. The detailed investigation on performance overhead is provided in Section IV. In

Baasanjargal Erdenebat and Tamás Kozsik are with Department of Programming Languages and Compilers, ELTE Eötvös Loránd University, Budapest, Hungary (e-mail: baasanjargal@inf.elte.hu, ORCID: 0000-0003-0471-7183; tamas.kozsik@elte.hu, ORCID: 0000-0003-4484-9172.)

Bayarjargal Bud is with National University of Mongolia, (e-mail: bud.bayarjargal@gmail.com)

DOI: 10.36244/ICJ.2023.5.11

Section V, we present our measurement results. Section VI addresses related work. Finally, conclusions are drawn in Section VII.

II. Background

A. Microservices and containerization tools

Quoting from "microservices.io" [8], microservices are an architectural style that organizes an application into a set of services characterized by the following qualities: (1) sufficiently decoupled to ensure high maintainability and testability; (2) loosely coupled from the application glue logic (e.g., orchestration, monitoring, etc.); (3) independently deployable; (4) organized around single business capabilities. Containerization technology plays a crucial role as the key enabler for microservices. From an organizational standpoint, the combination of microservices and containerization empowers the microservice architecture style to facilitate the evolution of an organization's technology and organizational stack in harmony with its architectural structures [9], [10]. In the scope of this work, we focus on Docker as a containerization tool and Kubernetes for the container orchestration. Docker helps to provide a quick and lightweight environment and allows us to build, deploy, run, update and manage container—standardized, executable components that combine application source code with the operating system (OS) libraries and dependencies required to run that code in any environment [11], [5]. On the other hand, Kubernetes is one of the most popular orchestration platforms for automating the deployment of, and managing, containerized workloads and services, as well as for scaling containerized applications across a cluster [12], [13].

B. Service discovery

Service Discovery is a design pattern which can enable clients or API gateways to discover the network information, such as the IP address and port, of microservices [14]. This discovery process relies on a component known as the Service Registry or Discovery Server. The Service Registry is responsible for tracking all individual microservices within the architecture and storing their IP addresses and ports in its database. Whenever a service scales up or down, it sends a message to the discovery service, which updates its database accordingly. The microservices are registered and cancelled through a service registry, with Netflix Eureka serving as an example of such a registry [15], [16].

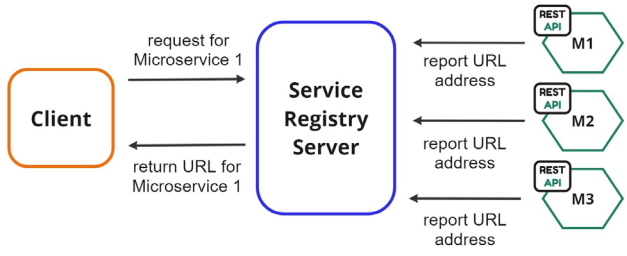

Fig. 1. Client discovery mechanism.

To cater to different business application scenarios, microservices utilize a registration discovery mechanism that combines client discovery and server discovery [17]. The client discovery mechanism can rely on Netflix Eureka technology, where clients query a service registry ("service center") to retrieve a list of available service instances [15], [18] as shown in Figure 1. Using a load balancing algorithm, the client selects one of the available service instances and sends out the request. The registration management and querying of service instances are facilitated through REST API calls provided by Eureka within the application. Eureka's client operates in self-registration mode, requiring it to handle service instance registration, cancellation, and sending regular heartbeats.

III. Migration of microservices from VMs to containers

Two main initiatives were undertaken as part of this study to address service discovery challenges. The first one involved migrating microservices from a virtual machine (VM) based environment to a container-based one. Our microservice system utilizes Spring Cloud development and is deployed on a Docker-based Kubernetes cluster. Over time, we gradually separated services from a monolithic architecture to Spring Cloud. Currently, we are successfully operating approximately 600 instances and 110 application services on DevOps infrastructure and self-hosted Kubernetes cluster. However, to enable service discovery between the VM-based and the container-based infrastructures, a solution was needed. Thus, we designed and implemented a seamless migration process for the microservices, with a primary focus on introducing easy-to-use dynamic service discovery during the transition period.

The second initiative involves leveraging Kubernetes' native service discovery capabilities after the completion of the migration to containerized microservices. The motivation behind this initiative was that we encountered abnormal functionality with Kubernetes DNS and service discovery, particularly in scenarios involving numerous external name services and pods. Therefore, we conducted an investigation to identify the root cause of the problem, to evaluate CoreDNS, and to develop a solution for these challenges.

Performing a migration from legacy system to microservice-based architecture while introducing major architecture changes must be seamless for the product teams and for the end-users. During this study, we performed a gradual migration of front-end applications into containerized microservices deployed within a Kubernetes cluster. However, some of the remaining applications still operate on virtual machines. Until all the applications were completely migrated to a container-based microservice architecture, we need to find a solution to enable service discovery between the Kubernetes cluster and the VM(s). Therefore, we utilized Netflix Eureka as a key component to underpin the service discovery pattern and to provide client-side load balancing. The microservices are able to register themselves with the Eureka server, which stores the microservices' access information, including their respective ports and IP addresses [7].

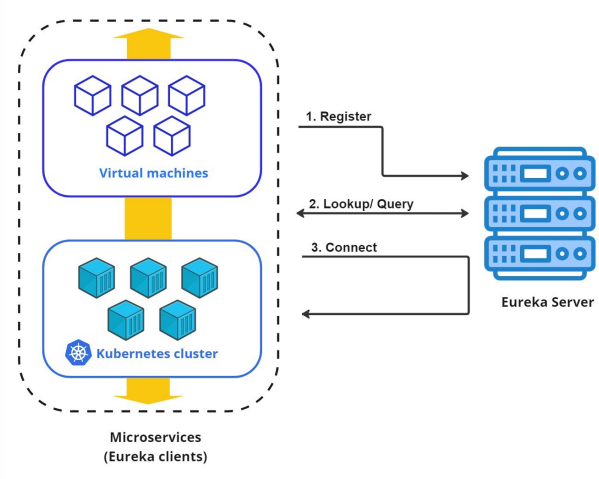

Fig. 2. Service discovery between Kubernetes cluster and VM(s) through Eureka.

For our case, when a microservice running in the Kubernetes cluster needs to invoke another microservice located on a VM, communication completely goes out of the Kubernetes cluster, and reaches out to the Eureka server to discover the location of the called microservice. Then it returns back to the Kubernetes cluster with the desired data as shown in Figure 2. Even when container-based microservices require communication with one another, they rely on Eureka since it is the primary service discovery mechanism within the entire system for now. This approach is not quite efficient from a performance perspective, particularly in large-scale clusters. By transitioning to a container-based architecture, we are able to eliminate certain components from the legacy infrastructure. Once the microservices are fully containerized and deployed in the Kubernetes cluster, we can remove Eureka. This not only enhances performance, but also facilitates a cloud-native deployment. Kubernetes itself can handle both service discovery and load balancing, enabling a more streamlined and efficient approach.

We have identified the necessary steps to smoothly decommission the use of Netflix Eureka during the migration process, with the aim of achieving zero downtime, resolving traffic issues, minimizing effort, and reducing risks and impacts. Our goal was to ensure that microservices should communicate with each other using both Netflix Eureka and Kubernetes service discovery in a seamless manner during the transition phase, simply by switching a "discovery flag". We conducted tests on API endpoints before and after switching the Eureka client flag to verify that the same code base functions properly in both scenarios. The following changes were implemented during the migration process.

A. Code-based changes

- Changes in the `pom.xml` [19] file regarding the dependencies.
- Changes in the main Spring Boot `application.java` file.
- Changes in the `bootstrap.yaml` [20] file of Spring Boot.

B. Adjustments of the cluster configurations

- Modifications in the config-server database – When utilizing Kubernetes discovery, the microservice configuration is automatically updated to include the "Kubernetes" active profile. It is important to note that the default configuration profile is labeled as "development", which differs from the "Kubernetes" profile. To ensure the reliability of microservices, a new profile named "Kubernetes" must be added to the config-server database.
- Add role [21] and role-binding [22] to service account [23] in the Kubernetes cluster – In order for Spring Cloud Kubernetes to retrieve a list of addresses for pods belonging to a specific service, it requires access to the Kubernetes API. To ensure this access, deployment or pod must be assigned to the relevant service accounts, and it is essential to verify that they possess the correct roles.
- Changes in the global configuration file – This modification allows for the convenient switching of the discovery flag between Eureka and Discovery-client, or vice versa.

By implementing these necessary changes in the code base and the configurations, a safe and straightforward transition from Netflix Eureka to native Kubernetes service discovery becomes achievable. After moving dozens of microservices, our implementation proved to be effective and significantly alleviated the burden of the migration process. The focus on simplicity has played a crucial role in enabling a smoother transition while saving effort and time.

IV. Performance Improvements

In large scale Kubernetes clusters, the total number of running pods, services, requests, and workloads can be high, and the increased number of HTTP-requests often result in resource utilization concerns, e.g., spikes of errors [24], [25]. The memory usage of Kubernetes DNS is predominantly affected by the number of pods and services in the cluster [26], [27]. Other factors include the size of the filled DNS answer cache, and the rate of queries received per CoreDNS instance [26].

Upon encountering resource consumption issues and a spike of errors in HTTP-requests, we started to troubleshoot the core pain points, and to solve the issues by fine-tuning the configuration of CoreDNS [24]. Our initial idea was to increase the number of replicas for the application to assess whether it would help enhance performance and mitigate errors. As we delved deeper into the issue with the application developers, we discovered that the majority of failures could be attributed to DNS resolution. This discovery led us to shift our focus towards improving the performance of DNS resolution in Kubernetes.

We have carried out a stress test on service discovery to identify bottlenecks. For our experiment, we used a cluster with one master and 10 worker nodes, which were set up with the default settings of Kubernetes. We executed Java-based

TABLE I
PERFORMANCE BEFORE OPTIMIZATION

Number of pods & services	Max memory (MB)	Max CPU (cores)	Average response time in seconds	Network load: Receive (MB/s)	Network load: Transmit (MB/s)
0	19	0.0001	0.0008	0.0031	0.0047
250	776	2.74	0.67	3.7	6.37
1054	8914	5.27	4.02	8.10	13.98
2000	16664	9.14	8.19	11.25	18.36

front-end applications and microservices as Kubernetes pods and Kubernetes services on that cluster. The figures presented in Table I are based on data collected from the tests using the following setup.

- Master: n1-standard-1 (16 vCPU, 32 GB memory)
- Nodes: n1-standard-2 (32 vCPUs, 125 GB memory)
- Networking: calico-3.19.1
- Kubernetes Version: 1.21.3
- CoreDNS Version: 1.7.0

As shown in Table I, resource consumption and network load drastically increase when the total number of services and pods are raised. We need to emphasize that here the type of most of the services was `externalName` [28], which was one of the reasons of the phenomena that when the total number of services and pods were beyond 1054, the system consumed high amount of resources such as around 16 GB of memory and 9 CPU cores. The average response time for 2000 pods/services was around 8.19 seconds, which resulted in high latency and high error rate for HTTP-requests.

The CoreDNS function with default configuration occasionally crashed when running 500 external services and pods in the Kubernetes cluster. After this incident, we adjusted the memory resource "request/limit" in the CoreDNS deployment up to 8 GB from 170 MB, and increased the total number of instances to four. This high amount of resource consumption indicates that the current implementation may exhibit abnormal behavior, which can lead to malfunctions and failures of the CoreDNS.

According to our experiences, it is insufficient to merely add extra CoreDNS instances or configure Horizontal Pod Autoscaler (HPA) for the cluster based on number of requests, resource consumption, and number of workloads running on the cluster to address the performance and stability problem effectively, especially for large-scale clusters in which numerous projects and environments are being developed simultaneously. Expanding resource utilization continuously in response to increased requests is fruitless, even if the cluster possesses sufficient resources. Therefore, an accurate and appropriate solution is necessary to address these concerns. As we started to investigate more into how the application is making requests to CoreDNS, and troubleshooting the DNS resolution and cluster configurations, we observed most of the outbound requests happening through the application to an external API which leads a high spike of errors. Also, we found out several obstacles that hindered the smooth functioning of the system, including the lack of a logical and well-defined DNS search flow, and misconfigured Kubernetes service objects. These matters had a substantial impact on CoreDNS, leading to failures and excessive load that caused such high resource consumption and latency issues.

Therefore, we developed a dedicated solution at the cluster level, with which organizations can mitigate the previously mentioned service discovery issues and ensure smooth functioning of the Kubernetes clusters.

A. Reconfiguring service object

In our scenario, pods/instances frequently perform external lookups through the `externalName` service object in Kubernetes. We discovered that issues stemmed from unnecessary port definitions in the `externalName` service. However, rectifying this required code modifications on the microservice side to create the `externalName` service without any port definition. As a solution, we made updates to the source code, incorporating the Fully Qualified Domain Name (FQDN) of the client service. This approach prevents invalid DNS lookups resulting from search domains. For example, if the client pod needs to access rate-service, the domain name can be specified as www.rate-service.com. To ensure the effectiveness of these changes, comprehensive end-to-end tests were conducted in the non-production environment.

B. Enabling local DNS Cache

In addition, we implemented a Node Level DNS Cache to enhance the stability and performance of service discovery. This involved optimizing DNS resolution, improving latency, and reducing the burden of CoreDNS. With the current DNS architecture, it is possible that pods with the highest DNS QPS have to reach out to a different node, if there is no local CoreDNS instance [29]. To address this, we deployed local *DNS Caching* agents on nodes as a *Daemonset*, which significantly improved the performance of Cluster DNS. It works as a CoreDNS caching agent and pods will reach out to the agent running on the same node. Thereby it helps to avoid connection tracking and iptables DNAT rules. When the local caching agent encounters cache misses for cluster hostnames, it queries the core-dns service for resolution.

C. Defining search sequence

During our in-depth investigation into how the application sends requests to CoreDNS, we discovered that a significant portion of outbound requests were directed towards an external

API. This led to domain name resolution errors occurring within the cluster. To resolve this issue, we took the necessary steps to optimize the resolv.conf file of the application deployment pod.

```
nameserver 10.10.10.151
search local-namespace.svc.cluster.local \
       global-available.svc.cluster.local \
       svc.cluster.local cluster.local \
       head-zones.local
```

Fig. 3. Defined DNS Search sequence.

We specify the domain name that a client pod needs to access based on the rules as shown in Figure 3. These rules can help minimize the number of attempts for resolving the domain name, and make the DNS resolution service more efficient. When the DNS resolver sends a query to the CoreDNS server, it tries to search the domain considering the search path. If the client pod needs to access a service in the same namespace, the initial search attempt must always be inside a local environment, then the second attempt, in this case, will search in the next specified environment (global available environment). If the domain remains unresolved, the search process proceeds to the current cluster level. If the domain is still not resolved at this stage, the search is expanded to encompass the entire infrastructure of the organization.

If we are looking for a domain `frontapp.io`, the search would make the queries which are presented in Figure 4, and it receives a successful response in the last query.

```
frontapp.io.local-namespace.svc.cluster.local <= NXDomain
frontapp.io.global-available.svc.cluster.local<= NXDomain
frontapp.io.svc.cluster.local <= NXDomain
frontapp.io.cluster.local <= NXDomain
frontapp.io.head-zones.local <= NXDomain
frontapp.io <= NoERROR
```

Fig. 4. DNS query for lookup.

Due to the excessive number of external lookups, an application may receive numerous NXDomain responses for DNS searches. To optimize this, we customized dnsConfig in the Deployment object of the container, which will change `resolve.conf` accordingly on pods. The search is being performed only for an external domain. This reduces the number of queries to DNS servers, and helps mitigate spike errors for the application.

D. Ensuring availability – Optimize resource allocation

During time periods of high DNS query volume and with numerous services/pods in the cluster, CoreDNS tends to consume additional memory and CPU resources. Therefore, the default memory limit can lead to out of memory (OOM) errors. As a result, CoreDNS pods undergo repetitive restart attempts but fail to start successfully. To address this, we fine-tuned CoreDNS and adjusted the resource requirements within the cluster. Specifically, we modified the default values for memory resource "requests and limits" from 170MB to 1GB, and for CPU from 1 milli-core to 500 milli-cores, based on the cluster's status.

E. Enabling scalability – Auto-scale the number of pods - Horizontal Pod Autoscaler

We have implemented the Horizontal Pod Autoscaler (HPA) to dynamically adjust the number of CoreDNS pods. Currently, the cluster is provisioned with five CoreDNS pods. In the event of increased resource utilization leading to overload, we have incorporated a backup configuration within the autoscaler. The HPA is configured with the policy settings illustrated in Figure 5, enabling it to increase the number of CoreDNS pods based on CPU utilization.

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: coredns-hpa
  namespace: kube-system
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: coredns
  minReplicas: 5
  maxReplicas: 10
  metrics:
    - type: Resource
      resource:
        name: cpu
        targetAverageUtilization: 70
```

Fig. 5. HPA configuration based on CPU usage.

V. Results

The combination of the above-mentioned changes and tuning solutions has significantly addressed most of the previously experienced issues, and provided a more optimal resource consumption, minimized the blast radius of CoreDNS crashes, mitigated DNS errors, timeouts, and latency. After the adjustments, the resource utilization of CoreDNS has drastically dropped into a very low amount, as demonstrated in Table II. Even when the total pod and service count reached 2000 (and above), it only consumed 524 MB of memory and 0.2 cores of CPU, which is a remarkable improvement compared to the original state. In the initial measurement data (shown in Table I) we could observe very high latency – around 8 seconds at peak load with 2000 pods/services.

With the new implementation, the average response time has been reduced to less than 2 milliseconds, falling within an acceptable range. This improvement has also helped us minimize spike errors in HTTP-requests. The new measurement proves that the original implementation was functioning in an abnormal way due to design and configuration flaws, resulting in the malfunction and failures of CoreDNS.

VI. Related work

The solution for the increased load and errors of service discovery in Kubernetes requires a multi-faceted approach that addresses both scalability and reliability challenges. To mitigate the issues and ensure smooth operation of the execution environment, a number of techniques can be employed.

TABLE II
MEASUREMENT DATA AFTER THE ADJUSTMENTS.

Number of pods & services	Max Memory (MB)	Max CPU (core)	Average Response time in seconds	Network load: Receive (MB/s)	Network load: Transmit (MB/s)
0	19	0.0001	0.00071	0.0019	0.0025
250	76.480	0.096	0.00104	0.86	0.72
1054	280.72	0.155	0.00125	1.42	1.26
2000	524.09	0.203	0.00172	1.8	1.45

Nguyen et al. [30] proposes the horizontal scaling of the containers using Kubernetes' built-in autoscaling capabilities; in this way resources can be dynamically allocated based on demand, ensuring that DNS pods can adeptly manage surges in traffic and efficiently distribute the workload. We also applied horizontal scaling, but the technique turned out to be insufficient to solve the problems on its own, therefore we had to investigate other solutions as well.

Zhang Wei-guo et al. [31] emphasise that to prevent resource exhaustion and reduce the occurrence of errors or crashes, it is essential to optimize the allocation of resources for DNS pods, including setting appropriate CPU and memory limits. By ensuring accurate resource provisioning, a service discovery pod can gain the necessary capacities to handle the workload efficiently. This approach was still insufficient in our case, thereby we applied it with additional techniques.

Nguyen and Kim [32] argue that implementing a robust load balancing mechanism, either within Kubernetes or by integrating with external load balancers, enables the even distribution of DNS requests across multiple DNS instances. Load balancing helps prevent bottlenecks and ensures high availability. After introducing the changes to our cluster configuration and service discovery, the use of a special load balancer was not necessary. However, it might become so in the future, when the number of system components grow even higher.

Almaraz-Rivera [33] underlines the importance of establishing comprehensive monitoring and alerting systems to track containers' performance, latency, error rates, and resource utilization. Proactive monitoring enables the early detection of potential issues and allows for timely remediation. In align with this approach, we introduced Prometheus and Grafana as monitoring and alerting tools for our Docker-based Kubernetes infrastructure.

Horaleket et al. [34] point out that enabling detailed logging for containers and leveraging logging aggregation solutions can aid in troubleshooting errors and performance issues. Analyzing logs can provide valuable insights into the root causes of problems and guide further optimizations. This technique was a key enabler in our methodology as well. The execution environment and the applications referred to in this paper employ Elasticsearch and Kibana for real-time search, analysis, visualization, and management of massive datasets.

The combination of these techniques can help overcome the challenges related with increased load and errors in CoreDNS within Kubernetes, allowing for the stable and effective operation of DNS resolution. However, they may not be adequate, or sufficient, to handle the arising problems in a variety of scenarios. In our case, the aforementioned approaches were still unsatisfactory to fully address the service discovery challenges. Consequently, we investigated alternative technical and engineering solutions. In this paper, we presented an in-depth explanation of our strategy to tackling the experienced difficulties.

VII. CONCLUSION

As containerization technologies become intensively used, certain challenges and problems arise. This paper proposed a technique to gradually migrate virtual machine based microservices to containerized ones, and solved an issue (which was discovered in a large-scale migration process) in the name service component of a popular cluster management solution.

We introduced a technique to help developers transition from Netflix Eureka based service discovery to a more lightweight native Kubernetes service discovery. This technique is useful when an application is gradually refactored from VM-based to Docker-based microservices, temporarily containing both kinds of components.

We discovered an issue with the default configuration of CoreDNS, the name service of Kubernetes, which causes performance degradation and service failures for high loads. We propose modifications which result improvements in the range of 1–2 orders of magnitude, and drastically increases the stability of CoreDNS.

ACKNOWLEDGMENT

We would like to acknowledge and thank the company and company representatives for their contributions and involvement in the study.

REFERENCES

[1] Blinowski, G., Ojdowska, A. & Przybylek, A. Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation. *IEEE Access*. 10 pp. 1–1 (2022,1)

[2] Thones, J. Microservices. *IEEE Software*. 32 pp. 116–116 (2015,1) DOI: 10.1109/MS.2015.11

[3] Qian, L., Chen, H., Yu, J., Zhu, G., Zhu, J., Ren, C., Mei, Z., Pang, H., Xu, M. & Wang, L. Research on Micro Service Architecture of Power Information System Based on Docker Container. *IOP Conference Series: Earth And Environmental Science*. 440, 032147 (2020,2), DOI: 10.1088/1755-1315/440/3/032147

[4] Shifeng, Z. & Shanliang, P. Application of Docker technology in micro-service. *Electronic Technology And Software Engineerings*. 4, 164 (2019)

[5] Merkel, D. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*., 2 (2014)

[6] Ákos, L., Edina, L., Attila, H., József, V., and László, B., "Closed-loop Orchestration for Cloud-native Mobile IPv6", *Infocommunications Journal*, Vol. XV, No 1, March 2023, pp. 44–54., DOI: 10.36244/ICJ.2023.1.5

[7] Macero, M. Learn Microservices with Spring Boot: A Practical Approach to RESTful Services using RabbitMQ, Eureka, Ribbon, Zuul and Cucumber. *Learn Microservices With Spring Boot*, 1st ed. Berkeley, CA:Apress, 2017, DOI: 10.1007/978-1-4842-3165-4

[8] Microservices: What are microservices, Available online: https://microservices.io/ (Accessed: 2023)

[9] Li, Z., Kihl, M., Lu, Q. & Andersson, J. Performance overhead comparison between hypervisor and container based virtualization. *In Proceedings Of The 2017 IEEE 31st International Conference On Advanced Information Networking And Applications (AINA)*. pp. 955–962, 2017, DOI: 10.1145/1342250.1342261

[10] Germar, S., Paul, P., Matthias, F., Dirk, R., Feryel, Z., and Jerker, D., "Micro Service based Sensor Integration Efficiency and Feasibility in the Semiconductor Industry", *Infocommunications Journal*, Vol. XIV, No 3, September 2022, pp. 79–85., DOI: 10.36244/ICJ.2022.3.10

[11] IBM: Docker, Available online: https://www.ibm.com/topics/docker (Accessed: 2023)

[12] The Kubernetes Authors. Available online: https://kubernetes.io/ (Accessed: 2023)

[13] Brewer, E. Kubernetes and the path to cloud native. SoCC '15: Proceedings of the Sixth ACM Symposium on Cloud Computing (2015,8), DOI: 10.1145/2806777.2809955

[14] Ranjan R, W., A, L. & A, Q. Peer-to-peer cloud provisioning: Service discovery and load-balancing. *Cloud Computing*. pp. 195–217, 2010.

[15] Christudas, B. *Practical Microservices Architectural Patterns: Event-Based Java Microservices with Spring Boot and Spring Cloud*. Apress (2019,1) DOI: 10.1007/978-1-4842-4501-9

[16] Sharp, T. Deploying the Microservice as a Docker Container. *Introducing Micronaut*. 2022, DOI: 10.1007/978-1-4842-8290-8-9

[17] Md, Varadarajan, A., Mandal, V. & Karan. Efficient Algorithm for Identification and Cache Based Discovery of Cloud Services. *Mobile Networks And Applications*. 24, pp. 1–17, 2019, DOI: 10.1007/s11036-019-01256-0

[18] Suryotrisongko, H., Jayanto, D. & Tjahyanto, A. Design and Development of Backend Application for Public Complaint Systems Using Microservice Spring Boot. *Procedia Computer Science*. 124, pp. 736–743, 2017, DOI: 10.1016/j.procs.2017.12.212

[19] Apache Maven. Introduction to the POM. Available online: https://maven.apache.org/guides/introduction/introduction-to-thepom.html (Accessed: 2023)

[20] Spring Cloud Context: Application Context Services. Available online: https://cloud.spring.io/spring-cloud-commons/multi/multi__spring_cloud_context_application_context_services.html (Accessed: 2023)

[21] The Kubernetes Authors. Using RBAC Authorization. Available online: https://Kubernetes.io/docs/reference/access-authn-authz/rbac/ (Accessed: 2023)

[22] The Kubernetes Authors. RBAC Good Practices. Available online: https://kubernetes.io/docs/concepts/security/rbac-good-practices/ (Accessed: 2023)

[23] The Kubernetes Authors: Service Accounts. Available online: https://kubernetes.io/docs/concepts/security/service-accounts/ (Accessed: 2023)

[24] CoreDNS: DNS and Service Discovery, Available online: https://coredns.io/, (Accessed: 2023)

[25] Kubernetes: Using CoreDNS for Service Discovery. Available online: https://kubernetes.io/docs/tasks/administer-cluster/coredns (Accessed: 2023)

[26] Heidari, A., Navimipour, N. Service discovery mechanisms in cloud computing: a comprehensive and systematic literature review. *Kybernetes*. 51, pp. 952–981, 2022, DOI: 10.1108/K-12-2020-0909

[27] Singh, N., Hamid, Y., Juneja, S., Srivastava, G., Dhiman, G., Gadekallu, T. R., Mohd, A.S. Load balancing and service discovery using Docker Swarm for microservice based big data applications. *Journal of Cloud Computing*. 12:4, 2023, DOI: 10.1186/s13677-022-00358-7

[28] The Kubernetes Authors. Services-External Name. Available online: https://Kubernetes.io/docs/concepts/services-networking/service/ (Accessed: 2023).

[29] Using NodeLocal DNS Cache in Kubernetes Clusters. Available online: https://kubernetes.io/docs/tasks/administer-cluster/nodelocaldns/ (Accessed: 2023)

[30] Nguyen, T.-T., Yeom, Y.-J., Kim, T., Park, D.-H., Kim, S. Horizontal Pod Autoscaling in Kubernetes for Elastic Container Orchestration, *Sensors*, 20(16), p. 4621, Aug. 2020, DOI: 10.3390/s20164621.

[31] Wei-guo, Z., Xi-lin, M., Jin-zhong, Z. Research on Kubernetes' Resource Scheduling Scheme. In Proceedings of the 8th International Conference on Communication and Network Security (ICCNS '18). Association for Computing Machinery, New York, NY, USA, 2018, 144–148. DOI: 10.1145/3290480.3290507

[32] Nguyen, N., Kim, T. Toward Highly Scalable Load Balancing in Kubernetes Clusters. *IEEE Communications Magazine*, 58(7):78–83, July 2020, DOI: 10.1109/MCOM.001.1900660.

[33] Almaraz-Rivera, J.G. An Anomaly-based Detection System for Monitoring Kubernetes Infrastructures, *IEEE Latin America Transactions*, 21(3):457–465, March 2023, DOI: 10.1109/TLA.2023.10068850.

[34] Horalek, J., Urbanik, P., Sobeslav, V., Svoboda, T. "Proposed Solution for Log Collection and Analysis in Kubernetes Environment." in *Nature of Computation and Communication*. ICTCC 2022. vol 473. Springer, Cham, 2023 DOI: y10.1007/978-3-031-28790-9_2

Baasanjargal Erdenebat was born in 1993 in Arkhangai, Mongolia. She earned the bachelor and master degrees from the School of Applied Science and Engineering at the National University of Mongolia, in 2013 and 2016 respectively. Currently, she is pursuing a PhD at Eötvös Loránd University, Dept. Programming Languages and Compilers. Her main research interests are containerization technology (i.e., Docker, Kubernetes), cloud computing, DevOps methodology, as well as the toolchains and implementation related to these areas.

Bayarjargal Bud was born in 1987, Mongolia. He obtained the bachelor's degree from the National University of Mongolia (NUM) in 2010, and master's degree from NUM and Riga Technical University in 2021. Currently, he works as Tech Lead at IT delivery service department of the private sector. His research interests include database system, machine learning, and containerization technology (i.e., Docker, Kubernetes).

Tamás Kozsik is associate professor at Eötvös Loránd University, Dept. Programming Languages and Compilers. His fields of interest are formal verification, programming paradigms (i.e., concurrent programming), static analysis, refactoring, and domain specific programming languages. Currently he focuses on researching programming languages and software technology for quantum computing.