Smart Microservice Orchestration in Kubernetes

Mikhail M. Rovnyagin
National Research Nuclear University MEPhI
(Moscow Engineering Physics Institute)
Moscow, Russia
mmrovnyagin@mephi.ru

Klim A. Fedorov
National Research Nuclear University MEPhI
(Moscow Engineering Physics Institute)
Moscow, Russia
klim.fedorov2001@yandex.ru

Artem A. Eroshev
National Research Nuclear University MEPhI
(Moscow Engineering Physics Institute)
Moscow, Russia
aeroshev@mail.ru

Tatyana A. Rovnyagina
National Research Nuclear University MEPhI
(Moscow Engineering Physics Institute)
Moscow, Russia
tanya.plys@bk.ru

Dmitry M. Sinelnikov
National Research Nuclear University MEPhI
(Moscow Engineering Physics Institute)
Moscow, Russia
dsinelnikov96@gmail.com

Alexander V. Tikhomirov
National Research Nuclear University MEPhI
(Moscow Engineering Physics Institute)
Moscow, Russia
avtikhomirov@mephi.ru

Ivan A. Yakovenko
National Research Nuclear University MEPhI
(Moscow Engineering Physics Institute)
Moscow, Russia
ivan.yakovenko@icloud.com

Abstract — Most modern cloud Platform as a Service (PaaS) systems are built on the Docker application containerization technology. In this case, the server application (microservice) is "packaged" together with all its dependencies into a Docker container, which simplifies its transfer from server to server. The servers operate in cluster mode under the control of the Kubernetes container orchestration system. The Kubernetes cluster includes a scheduler that distributes containers across cluster nodes. The schedulers currently used do not have intelligent algorithms for placing containers on nodes. They work on the principle of placing in the "first suitable slot". This paper presents the architecture, technical implementation and experimental results of a «smart» scheduler that places containers while maintaining locality of interaction between microservices, which has a positive effect on interaction latency.

Keywords — ML, Kubernetes, distribute systems, scheduling, containerization, orchestration.

I. Introduction

The Kubernetes cluster includes a scheduler that distributes containers among cluster nodes (servers). The vast majority of applications in the world are containerized [1]. According to Yandex Cloud reports for the first half of 2022, the number of active users reached 23,100 [2]. 45% of them use Kubernetes. If we consider the 5 largest hosting providers in Russia (Yandex, Mail, Selectel, Reg.ru, RU-CENTER), we can talk about tens of thousands of client companies using Kubernetes.

Yandex Cloud has approximately 200,000 servers in 5 data centers [3]. Every year Yandex updates 10% of servers (spends about 15 billion rubles). The average cost of a server is approximately 750,000 rubles. Considering the resources of the largest providers: millions of servers with an average cost of hundreds of thousands of rubles.

Currently used schedulers do not have intelligent algorithms for placing containers on nodes. They work on the principle of being placed in the "first suitable slot". The use of advanced container placement planning algorithms in Kubernetes, which reduce operating costs by 10–15%, will allow cloud providers to achieve significant benefits.

Example: Yandex has 200,000 servers, unit cost is 750,000 rubles releasing 10% of resources saves 15 billion rubles.

Most modern distributed data processing frameworks (such as Apache Spark, Ignite, Flink, etc.) and microservices support Kubernetes as the main execution platform. Applications take advantage of the container orchestration system (such as service discovery, disaster recovery, configuration management). However, operations teams also face problems related to the peculiarities of Kubernetes environments, for example, performance degradation associated with the variability of container infrastructures. Also, a flexible approach to resource management (requests and limits on allocated resources) can itself pose a problem - the difficulty of providing a guaranteed resource quota to several applications in a Kubernetes environment at the same time.

This paper presents the architecture, technical implementation and experimental results of a «smart» scheduler that places containers while maintaining locality of interaction between microservices, which has a positive effect on interaction latency.

II. Related Works

Recently, the percentage of studies devoted to scheduling tasks in a cluster remains consistently high [4]. On the one hand, this is due to the emergence and widespread use of a flexible, convenient, extensible cluster resource management platform - Kubernetes. At the same time, a significant percentage of research is devoted specifically to the AI direction in planning. On the other hand, there is the use of multi-tenant solutions, where it is possible to benefit from the redistribution of resources, prioritizing the use of resources for users [5]. Moreover, as an object for optimization, both global characteristics can be used (for example, a network, the use of which can be improved using the particle swarm method, as in [6]) and the finite response time itself [7]. The

979-8-3503-9759-8/22/$31.00 ©2025 IEEE

largest number of studies is devoted to planning the placement of containers with microservices. However, recently a significant number of publications have focused on building Kubernetes infrastructures for IoT, Fog Computing, Edge Computing, where when planning calculations, it is necessary to solve the old problem of network heterogeneity. The HPC Lab has also conducted several studies on distributed computing scheduling [8-11]. As a result of these studies, it was noted that the accuracy of the scheduler is significantly affected not only by the scheduling algorithm, but also by the speed of appearance of metrics about the work of containers in the decision-making channel, as well as the overhead costs associated with this. This work continues these studies, leaving the general approach unchanged, but adding new tools for its implementation.

III. Optimization Method

The approach proposed in this study is applicable to microservice systems, or distributed systems of streaming and batch computing, in which the locality of interaction of distributed processing components (microservices, executors, etc.) with each other plays a significant role.

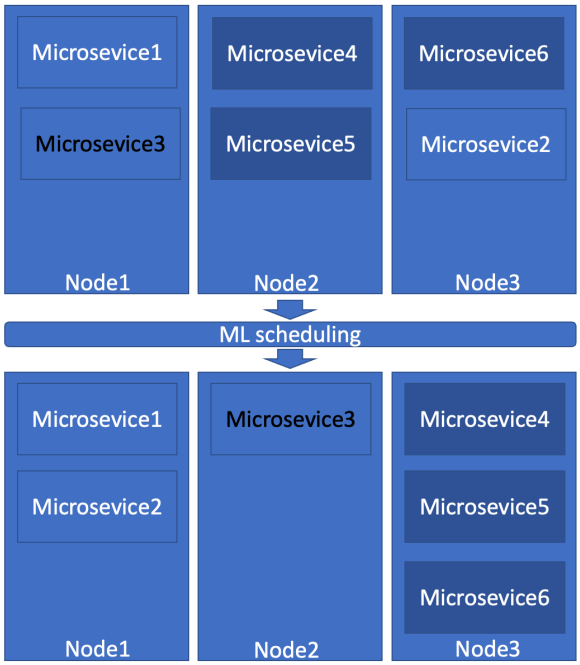

Fig. 1 – ML scheduling for microservices in Kubernetes

Fig. 1 shows an example of using an approach based on ML planning in relation to microservice systems. Before the ML scheduler is triggered, the microservices are distributed across the nodes in an arbitrary manner. The ML scheduler analyzes the state of the cluster nodes and the microservices placed on them and suggests a new placement scheme. A variety of characteristics can be selected as optimization criteria: from the uniformity of temperature distribution inside the data center to the total throughput of all microservices. In this paper, we propose a method for improving the response time of a gateway microservice (a service accessed by an external user with a request, generating the execution of a chain of requests between microservices inside the cluster). In this case, the ML scheduler determines which microservices interact with each other most often (form a chain of calls) and places them, if possible, close to each other on the same cluster nodes. As a clustering algorithm to demonstrate the operability of the method, the algorithm was chosen in this paper as the simplest and most illustrative. The clustering model was packaged as a Kubernetes CronJob and deployed on an experimental k3s cluster. Data collection was set up using the coroot eBPF framework, which collected performance metrics in Prometheus. The model was launched periodically, read metrics about the frequency of interactions between microservices within a certain sliding window, clustered microservices, and recorded data about the microservice belonging to a particular cluster in PostgreSQL. The scheduler was developed in Go as a plugin for the main Kubernetes scheduler. It read information about clusters from the database and built a target placement picture for a specific Pod being placed. The decision was made based on how many microservices that also belong to this cluster were already placed on a particular node. Also, to make the system more "flexible," a de-scheduler was developed that removed the Pod that deviated most from the target placement scheme, thereby initiating its re-placement. An experiment was conducted to demonstrate the viability of the approach.

IV. Experiment

To obtain a controllable and flexibly configurable experimental environment, a botnet was developed (Fig. 2). Each botnet agent deployed in k3s is a python application that receives a list of addresses to which the payload must be transmitted as a configuration. The payload is a parameterized volume of pre-generated data and a list of recipients with whom the recipient must interact. Thus, by sending a request to the gateway microservice, it is possible to generate an entire graph of interactions within the cluster, simulating the operation of real microservice chains.

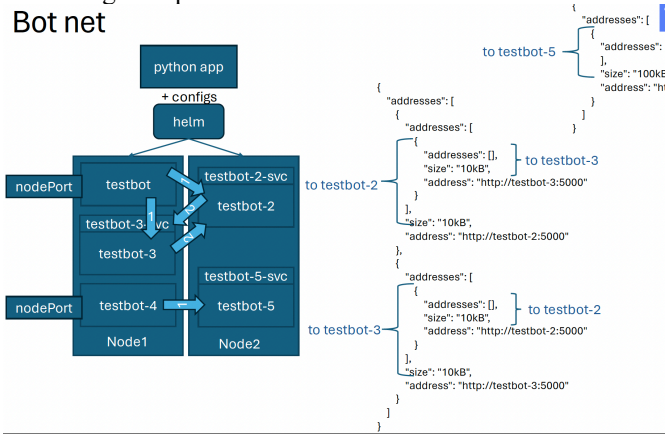

Fig. 2 – Botnet for experimental research

The load on the bot system was created by the k6 load framework. The configurations transmitted in its requests explicitly configured 30 bots for 3 isolated interaction clusters. The load within the experiment was stationary and lasted 15 minutes. The clustering application, implemented as a Kubernetes CronJob, worked every 2.5 minutes. The clustering result is presented in Figure 3.

The experiment used 3 worker nodes of the k3s cluster (Fig. 4).

For comparison, the experiment was conducted in two variants. Variant 1 – the standard Kubernetes scheduler was used. Figure 5 shows that the response time characteristic of the microservice cluster (data for cluster 1 is provided as an example) does not change over time.

```
postgres=# select * from clusters.pod_clusters;
 id |            podname            | cluster
----+-------------------------------+---------
  1 | botnet-8-56fbf5f579-hmjv8     |       0
  2 | botnet-4-798b44b487-27dq8     |       0
  3 | botnet-9-8484f8644-q788c      |       0
  4 | botnet-1-58c6bd85b4-zl88q     |       0
  5 | botnet-2-fd8b9b67c-mf5hf      |       0
  6 | botnet-7-5f84454cdb-q76h4     |       0
  7 | botnet-5-698698f67b-p5jgk     |       0
  8 | botnet-3-75c78688b-tvm6b      |       0
  9 | botnet-6-c64557677-cmpfn      |       0
 10 | botnet-0-844cd44f95-sm8lf     |       0
 11 | botnet-11-6f6958d88c-czjp7    |       1
 12 | botnet-15-5cb5c6d78c-d2sn8    |       1
 13 | botnet-19-5ff5b8b8c5-wnwf8    |       1
 14 | botnet-12-bb9976ff6-tbth7     |       1
 15 | botnet-10-7f76cbb4c4-cj5jv    |       1
 16 | botnet-13-84848764d5-kw5q6    |       1
 17 | botnet-14-574dfc4974-9km4f    |       1
 18 | botnet-18-54564ddb6-994g6     |       1
 19 | botnet-16-9699bd7b8-8rt88     |       1
 20 | botnet-17-d964d446f-k8cwq     |       1
 21 | botnet-26-7868599885-59gll    |       2
 22 | botnet-20-7c44d9c5f5-p68tr    |       2
 23 | botnet-24-db65b65cf-4mc5b     |       2
 24 | botnet-27-54c5dddfbd-s85jf    |       2
 25 | botnet-28-5f98fdd459-btmb6    |       2
 26 | botnet-21-5f88b899d6-8gqbb    |       2
 27 | botnet-22-796568b849-ntbf2    |       2
 28 | botnet-29-6c99bd66-d9cmf      |       2
 29 | botnet-23-5567877f5b-p9ntk    |       2
 30 | botnet-25-6b97f6bb5d-snlqd    |       2
(30 rows)
```

Fig. 3 – Botnet clustering DB

```
NAME                     READY   STATUS    RESTARTS   AGE     IP            NODE   NOMINATED NODE   READINESS GATES
botnet-0-844cd44f95-8fgvn   1/1     Running   0          7m1s    10.42.2.195   ks3    <none>           <none>
botnet-1-58c6bd85b4-9w8jw   1/1     Running   0          61s     10.42.0.199   ks1    <none>           <none>
botnet-10-7f76cbb4c4-cj5jv  1/1     Running   0          41m     10.42.0.191   ks1    <none>           <none>
botnet-11-6f6958d88c-czjp7  1/1     Running   0          41m     10.42.0.183   ks1    <none>           <none>
botnet-12-bb9976ff6-tbth7   1/1     Running   0          41m     10.42.0.192   ks1    <none>           <none>
botnet-13-84848764d5-kw5q6  1/1     Running   0          41m     10.42.0.184   ks1    <none>           <none>
botnet-14-574dfc4974-9km4f  1/1     Running   0          41m     10.42.0.187   ks1    <none>           <none>
botnet-15-5cb5c6d78c-xhrx5  1/1     Running   0          9m      10.42.0.198   ks1    <none>           <none>
botnet-16-9699bd7b8-frs29   1/1     Running   0          3m      10.42.1.145   ks2    <none>           <none>
botnet-17-d964d446f-k8cwq   1/1     Running   0          41m     10.42.0.185   ks1    <none>           <none>
botnet-18-54564ddb6-g5dgq   1/1     Running   0          10m     10.42.0.197   ks1    <none>           <none>
botnet-19-5ff5b8b8c5-wnwf8  1/1     Running   0          41m     10.42.2.145   ks3    <none>           <none>
botnet-2-fd8b9b67c-mf5hf    1/1     Running   0          41m     10.42.2.138   ks3    <none>           <none>
botnet-20-7c44d9c5f5-p68tr  1/1     Running   0          41m     10.42.2.147   ks3    <none>           <none>
botnet-21-5f88b899d6-8gqbb  1/1     Running   0          41m     10.42.1.114   ks2    <none>           <none>
botnet-22-796568b849-ntbf2  1/1     Running   0          41m     10.42.2.148   ks3    <none>           <none>
botnet-23-5567877f5b-p9ntk  1/1     Running   0          41m     10.42.1.115   ks2    <none>           <none>
botnet-24-db65b65cf-4mc5b   1/1     Running   0          41m     10.42.2.143   ks3    <none>           <none>
botnet-25-6b97f6bb5d-7rjvh  1/1     Running   0          10m     10.42.1.131   ks2    <none>           <none>
botnet-26-7868599885-59gll  1/1     Running   0          41m     10.42.2.152   ks3    <none>           <none>
botnet-27-54c5dddfbd-s85jf  1/1     Running   0          41m     10.42.1.118   ks2    <none>           <none>
botnet-28-5f98fdd459-btmb6  1/1     Running   0          41m     10.42.1.119   ks2    <none>           <none>
botnet-29-6c99bd66-d9cmf    1/1     Running   0          41m     10.42.2.153   ks3    <none>           <none>
botnet-3-75c78688b-tvm6b    1/1     Running   0          41m     10.42.2.141   ks3    <none>           <none>
botnet-4-798b44b487-27dq8   1/1     Running   0          41m     10.42.2.150   ks3    <none>           <none>
botnet-5-698698f67b-hgmsm   1/1     Running   0          10m     10.42.2.190   ks3    <none>           <none>
botnet-6-c64557677-4529c    1/1     Running   0          5m35s   10.42.2.197   ks3    <none>           <none>
botnet-7-5f84454cdb-q76h4   1/1     Running   0          41m     10.42.0.188   ks1    <none>           <none>
botnet-8-56fbf5f579-hmjv8   1/1     Running   0          41m     10.42.0.182   ks1    <none>           <none>
botnet-9-8484f8644-q788c    1/1     Running   0          41m     10.42.0.190   ks1    <none>           <none>
```

Fig. 4 – Botnet in k3s infrastructure

In contrast, the response time of the same cluster 1, after placing the bots associated with it using the smart scheduler (variant 2), decreased from 1.07 seconds to 0.729 seconds. In more complex botnet schemes, or in more loaded Kubernetes clusters, optimal placement can occur in several stages (as the de-scheduler works), until the target placement scheme is reached.

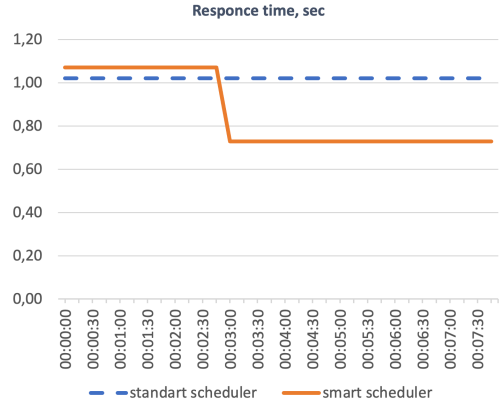

Fig. 5 - Distribution of the number of messages by sessions.

V. Conclusion

The conducted research resulted in the application of machine learning (ML) methodology for clustering and identifying frequently interacting microservices in the k3s environment. Coroot was used as a monitoring tool, which allowed analyzing the interaction of pods within the cluster.

A custom scheduling tool in Go for k3s was developed to manage the placement. It considered the clustering results and ensured a more local location of microservices, which led to a significant increase in system performance. Compared to the standard Kubernetes scheduling algorithm, this solution demonstrated a performance increase of more than 30%.

VI. Future works

The obtained results can be used as a basis for further improvement of resource management and scheduling in container environments.

Future research will focus on developing more sophisticated and accurate clustering methods, as well as optimizing the custom scheduler considering dynamic load changes and microservice composition. In addition, other factors affecting performance, such as network latency and load balancing between cluster nodes, will be analyzed.

VII. References

[1] 451 Research Says Application Containers Market Will Grow to Reach $4.3bn by 2022, Dec. 15, 2024. [Online]. Available: https://www.prweb.com/releases/451-research-says-application-containers-market-will-grow-to-reach-4-3bn-by-2022-885260529.html

[2] Yandex Cloud: results for the first half of 2022, Dec. 15, 2024. [Online]. Available: https://yandex.cloud/ru/blog/posts/2022/08/financial-results-1hy-2022

[3] Billions on Servers: Why Yandex Is Entering the Government, Contract Market Dec. 15, 2024. [Online]. Available: https://www.forbes.ru/tekhnologii/446373-milliardy-na-serverah-zacem-andeks-vyhodit-na-rynok-goszakaza?ysclid=m4q0ft18o1464592751

[4] Senjab, K., Abbas, S., Ahmed, N. et al. A survey of Kubernetes scheduling algorithms. J Cloud Comp 12, 87 (2023). https://doi.org/10.1186/s13677-023-00471-1

[5] A. Beltre, P. Saha and M. Govindaraju, "KubeSphere: An Approach to Multi-Tenant Fair Scheduling for Kubernetes Clusters," *2019 IEEE Cloud Summit*, Washington, DC, USA, 2019, pp. 14-20, doi: 10.1109/CloudSummit47114.2019.00009.

[6] A. Alelyani, A. Datta and G. M. Hassan, "Optimizing Cloud Performance: A Microservice Scheduling Strategy for Enhanced Fault-Tolerance, Reduced Network Traffic, and Lower Latency," in *IEEE Access*, vol. 12, pp. 35135-35153, 2024, doi: 10.1109/ACCESS.2024.3373316.

[7] C. Centofanti, W. Tiberti, A. Marotta, F. Graziosi and D. Cassioli, "Latency-Aware Kubernetes Scheduling for Microservices Orchestration at the Edge," *2023 IEEE 9th International Conference on Network Softwarization (NetSoft)*, Madrid, Spain, 2023, pp. 426-431, doi: 10.1109/NetSoft57336.2023.10175431.

[8] M. M. Rovnyagin, D. M. Sinelnikov, S. S. Varykhanov, T. R. Magazov, A. A. Kiamov and T. A. Shirokikh, "Intelligent Docker Container Orchestration for Low Scheduling Latency and Fast Migration in Paas," 2023 Seminar on Information Computing and Processing (ICP), Saint Petersburg, Russian Federation, 2023, pp. 181-185, doi: 10.1109/ICP60417.2023.10397344

[9] E. F. Mingazhitdinova, D. M. Sinelnikov, V. V. Odintsev, M. M. Rovnyagin and S. S. Varykhanov, "Approach of Program's Concurrency Evaluation in PaaS Cloud Infrastructure," *2022 Conference of Russian Young Researchers in Electrical and Electronic Engineering (ElConRus)*, Saint Petersburg, Russian Federation, 2022, pp. 383-385, doi: 10.1109/ElConRus54750.2022.9755685

[10] M. M. Rovnyagin, S. O. Dmitriev, A. S. Hrapov and V. K. Kozlov, "Algorithm of ML-based Re-scheduler for Container Orchestration System," *2021 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (ElConRus)*, St. Petersburg, Moscow, Russia, 2021, pp. 613-617, doi: 10.1109/ElConRus51938.2021.9396294

[11] M. M. Rovnyagin, A. S. Hrapov, A. V. Guminskaia and A. P. Orlov, "ML-based Heterogeneous Container Orchestration Architecture," 2020 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus), St. Petersburg and Moscow, Russia, 2020, pp. 477-481, doi: 10.1109/EIConRus49466.2020.9039033.