Advanced software framework for comparing balancing strategies in container orchestration systems

Yevhenii V. Voievodin[1], Inna O. Rozlomii[1]

[1]The Bohdan Khmelnytsky National University of Cherkasy, 81 Shevchenka Blvd., Cherkasy, 18031, Ukraine

Abstract

This paper introduces a detailed software design for a system that evaluates scheduling strategies in container orchestration systems. Focusing on software architecture, it elaborates on the various components such as dynamic cluster topology and container configuration streams, cluster packing algorithms, metric collectors and a state machine for tracking experiment progress. The system incorporates malfunction scenarios, testing the resilience of different strategies. The system is designed to be flexible and open to be extended with new key performance indicators and test scenarios. The experiment flow is split into independent iterations that can be efficiently run in parallel enabling faster experiment executions. The paper reviews related work, positioning this system as an essential tool in the current research landscape for resource distribution and management in distributed systems. A key aspect of the design is the client-server architecture, which not only ensures scalability and adaptability for various experiments but also includes an API for enhanced interaction and result analysis. This comprehensive design approach makes the designed system a helpful tool for nuanced analysis and informed decision-making in container orchestration, with the potential to advance in the field by speeding up researches and creating a collection of strategy evaluation techniques.

Keywords

container orchestration systems, Kubernetes, Docker, Docker Swarm, software design, distributed systems, resources distribution

1. Introduction

Container orchestration systems (COS) are modern software that provide capabilities to maintain large and complex systems [1]. The primary technology that COS relies on is a container. Containers can be described as applications packed with all the dependencies they require, making the deployment of such applications easy and reproducible across different operating systems and platforms. The convenience of such deployments accelerates the development of applications [2].

There are multiple components that COS consists of, as illustrated in figure 1: a cluster containing nodes, with nodes containing containers, and also a scheduler. The scheduler decides which node to use for deploying the next container in the sequence. To do so, the scheduler uses a scheduling strategy. Typical strategies include “binpack” and “spread”, each aiming for different goals [3]. For example, “binpack” aims to maximize the utilization of nodes, while “spread” allows for better fault tolerance [4] of the deployed application.

The role of the strategy cannot be undervalued, but the choice of strategy is not easy to make. It depends on a variety of factors, such as resilience to failures, usage of resources, and resource locality. Machine learning is one direction where strategy development is heading, which makes the comparison process even more challenging [5].

The key goal of this article is to provide a comprehensive design for an application capable of evaluating the performance of two or more scheduling strategies. Such an application must be flexible enough to compare strategies regardless of their implementation and be easily extendable with new metrics and comparison techniques.

doors-2024: 4th Edge Computing Workshop, April 5, 2024, Zhytomyr, Ukraine
yevhenii.voievodin@vu.cdu.edu.ua (Y. V. Voievodin); inna-roz@ukr.net (I. O. Rozlomii)
https://scholar.google.com/citations?user=AU5G-f0AAAAJ (Y. V. Voievodin);
https://scholar.google.com/citations?user=04ryMCwAAAAJ (I. O. Rozlomii)
0000-0002-6415-8566 (Y. V. Voievodin); 0000-0001-5065-9004 (I. O. Rozlomii)

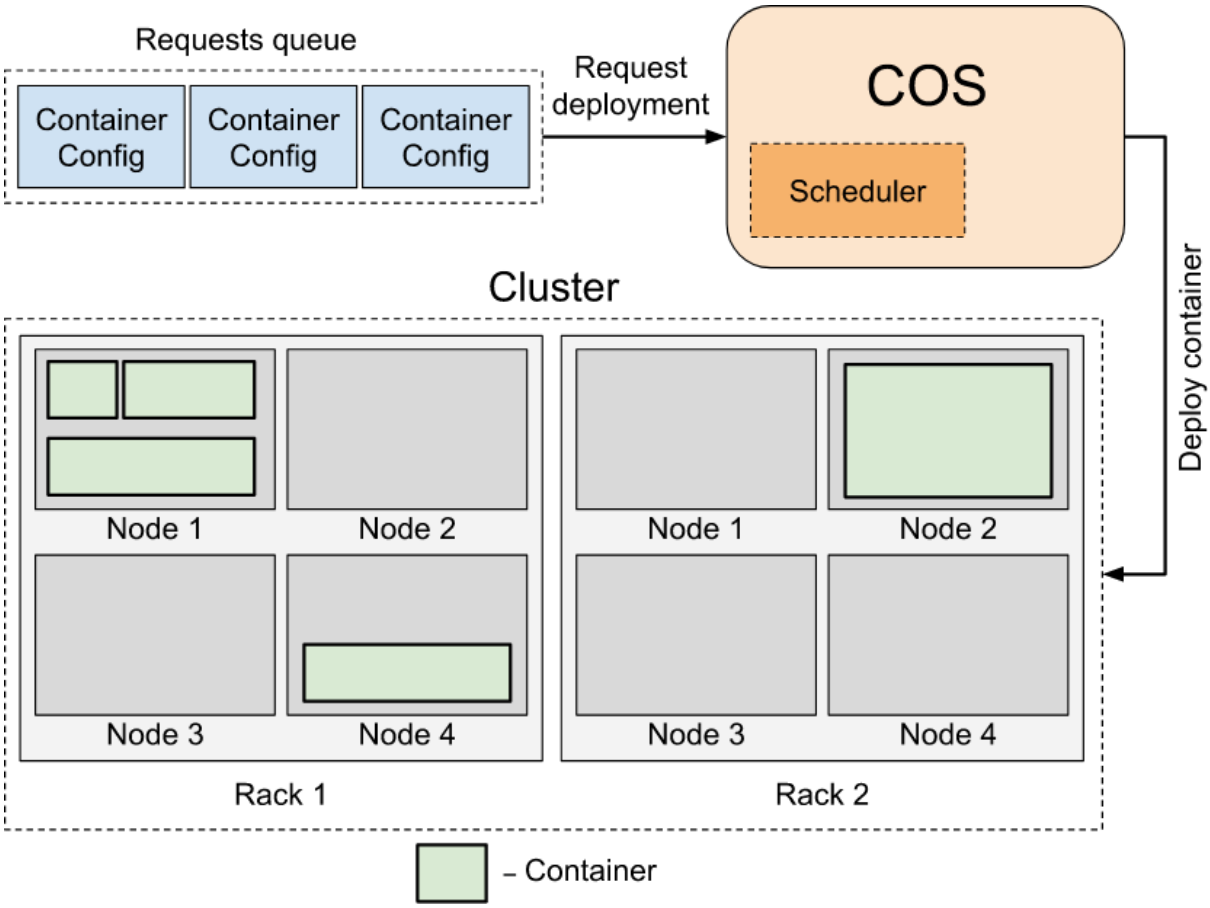


Figure 1: Key container orchestration system components.

2. Related works

The approach described by Voievodin et al. [6] to evaluate scheduling strategies provides a deep dive into the important role of the scheduling strategy choice. It proposes key performance indicators for comparing scheduling strategies and enlists ideas about which algorithms can be used to complete the evaluation from start to end. This includes packing algorithms, fault tolerance testing, and the aggregation of experiment results. This article delves deeper into the topic of strategy evaluation and proposes a more complete and sophisticated design for such evaluation software. While it builds on the proposed approach and algorithms, it extends the topic further by suggesting a concrete system structure and software techniques that can be used to implement such a system.

In the context of distributed systems, particularly those utilizing microservices architecture [7], COS serves as an essential tool, ensuring the efficient and easily scalable operation of independently deployed services across various computing environments at low overhead [8]. Saboor et al. [9] emphasize the importance of resource utilization in such applications, noting that they have gained rapid adoption in the software industry. A study on the aging and fault tolerance of microservices in Kubernetes provides useful insights on how COS, in the representation of Kubernetes, can achieve different fault tolerance properties and what options there are [10]. Gogouvitis et al. [11] discuss how container orchestration can be beneficial to seamless computing in industrial systems, which is software distributed across different computing domains. Akuthota [12] covers a technique of chaos engineering in distributed systems, which involves introducing controlled failures to the system to help make them more robust.

Many scheduling strategies have been developed recently, which higlights the actuality of the topic. An efficient virtual central processor unit scheduling in cloud computing [13]. Container scheduling using TOPSIS algorithm [14]. A combined priority scheduling method for distributed machine learning

[15]. A new container scheduling algorithm based on multi-objective optimization [16]. Improvement of container scheduling for docker using ant colony optimization [17]. A particle swarm optimization-based container scheduling algorithm of docker platform [18]. Contention-aware container placement strategy for docker swarm with machine learning based clustering algorithms [19].

3. Key components of the system

The experiment is a key component and consists of multiple parts, each of which must be separately configured. These parts include: a stream of configurations, a strategy, a packer, an iteration result collector, a malfunction algorithm, and a malfunction result collector. Different phases of the experiment are represented by its state. The state machine includes the states: NEW, RUNNING, COMPLETED, INTERRUPTED, and FAILED (figure 2).

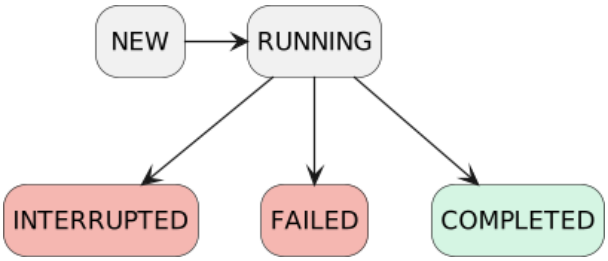


Figure 2: Experiment states and possible state transitions.

- NEW – indicates that the experiment can be configured. It has not been run yet, and new experiments might be incomplete in terms of configuration. Experiment components can be configured step by step.
- RUNNING – indicates that the experiment is currently running, which technically means going through the experiment flow (figure 3). Experiments that are running can no longer be modified in terms of configuration; the only change is that they accumulate data points for each new computed result.
- COMPLETED – indicates that the experiment has successfully completed. Such an experiment has run through all the configured steps and collected the desired metrics, which can now be analyzed.
- FAILED – indicates that the experiment was unable to complete successfully. This could be due to an unexpected error during execution or insufficient resources to finalize the experiment.
- INTERRUPTED – indicates that the execution of the experiment was deliberately interrupted. The reasons might vary, but primarily it could be to save resources when it's clear from the results produced so far that no further executions are necessary.

The class diagram (figure 4) covers the key components of the experiment, offering a detailed look into interfaces and structures. Before the first iteration starts, there is a setup phase, as illustrated in figure 3b. The iteration setup includes the propagation of cluster topology to all the strategies. The topology essentially comprises a set of nodes that have limits and can contain deployed containers. This topology is generated by the topology stream (figure 3a), a crucial first step. The topology stream allows for the definition of virtually any cluster structure, including the placement of nodes in physical racks for further fault tolerance testing. Additionally, the topology stream determines when the experiment stops, as it concludes when there are no more topologies to run the experiment for. The generated topology then serves as a prototype [20] for subsequent experiment iterations.

The stream of configurations is responsible for generating container configurations (requirements) to be placed within a cluster. Firstly, the stream can be either finite or infinite. An infinite stream will continue generating configurations as long as the packer demands it. Finite streams, on the other hand,

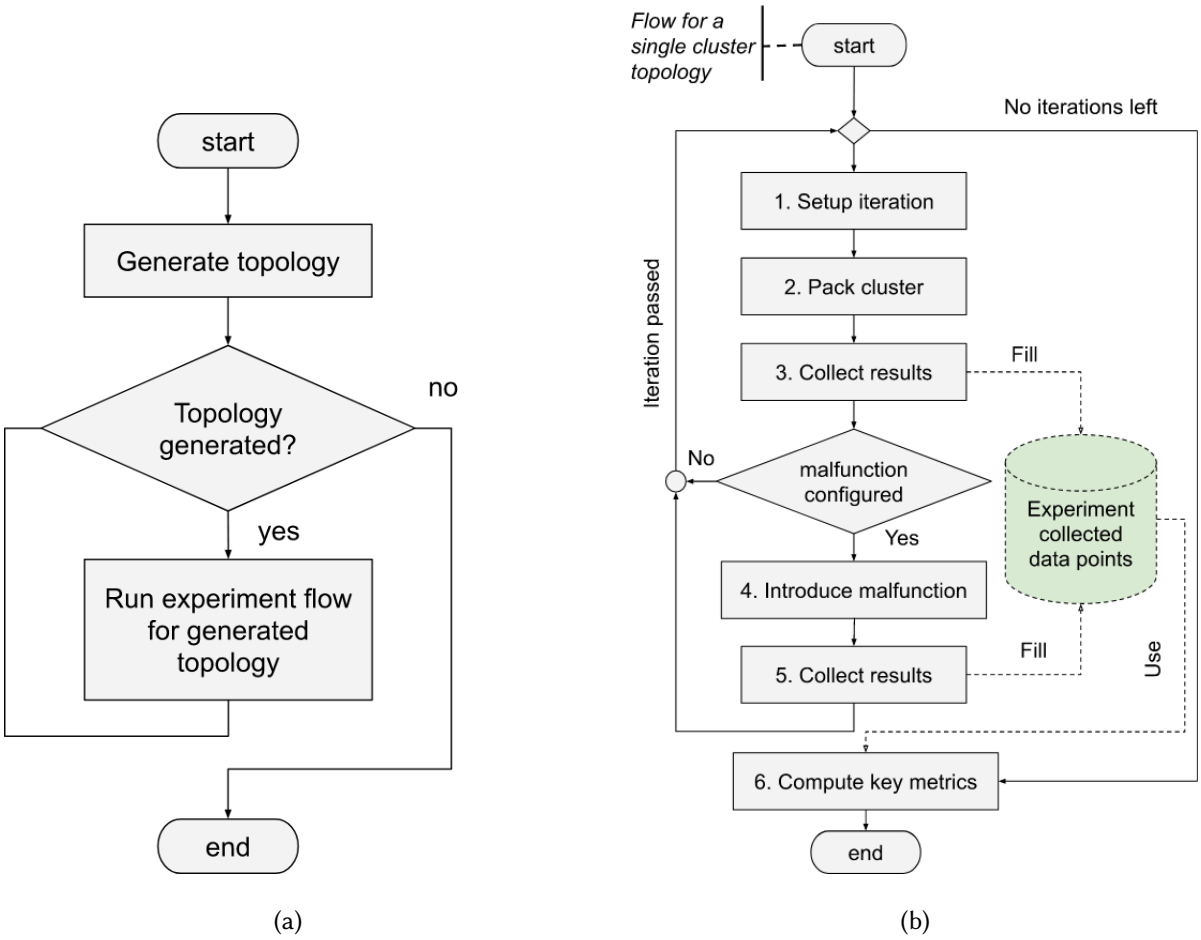


Figure 3: Experiment steps, (a) – topology generation, (b) – experiment flow for a single topology.

can be used to test scheduling strategies for a very specific set of container configurations, to seek a better strategy, or for strategy monitoring purposes. Regardless of whether a finite or infinite stream is used, virtually any sequence of containers can be provided, including random sequences. A crucial aspect is ensuring that the same sequence is fed to different strategies, where each strategy operates its own copy of the cluster to fill.

Secondly, the stream can be used to define application families. For instance, it can generate several configurations that depend on each other and form a larger application, as commonly seen in microservices architecture [21]. The specific implementation of the stream determines how to establish these dependencies, and the container configuration structure allows for such connections to be specified.

Thirdly, the stream can replicate a single container configuration multiple times, for example, if an application must be deployed multiple times within a cluster to ensure better response to failures [22]. Each aspect can be implemented as a separate stream representation. The decorator pattern [20] can be employed to combine these implementations in a desired manner, allowing for a variety of system demands to be covered.

The packer is tasked with making decisions regarding when to stop in the case of an infinite stream of configurations. As discussed by Voievodin et al. [6], such decisions are highly specific to the use-case being tested. For instance, the packer might stop after encountering the first scheduling request error, or once the cluster is full. While it is the packer's responsibility to execute scheduling algorithms, the packer itself does not depend on the specifics of the strategy implementation.

After the packing process is completed, the results aggregator collects the packing results based on the state of clusters filled by different strategies. Firstly, the aggregator's job is to collect important data points, which it does after the execution of every iteration. Secondly, it produces an aggregate that

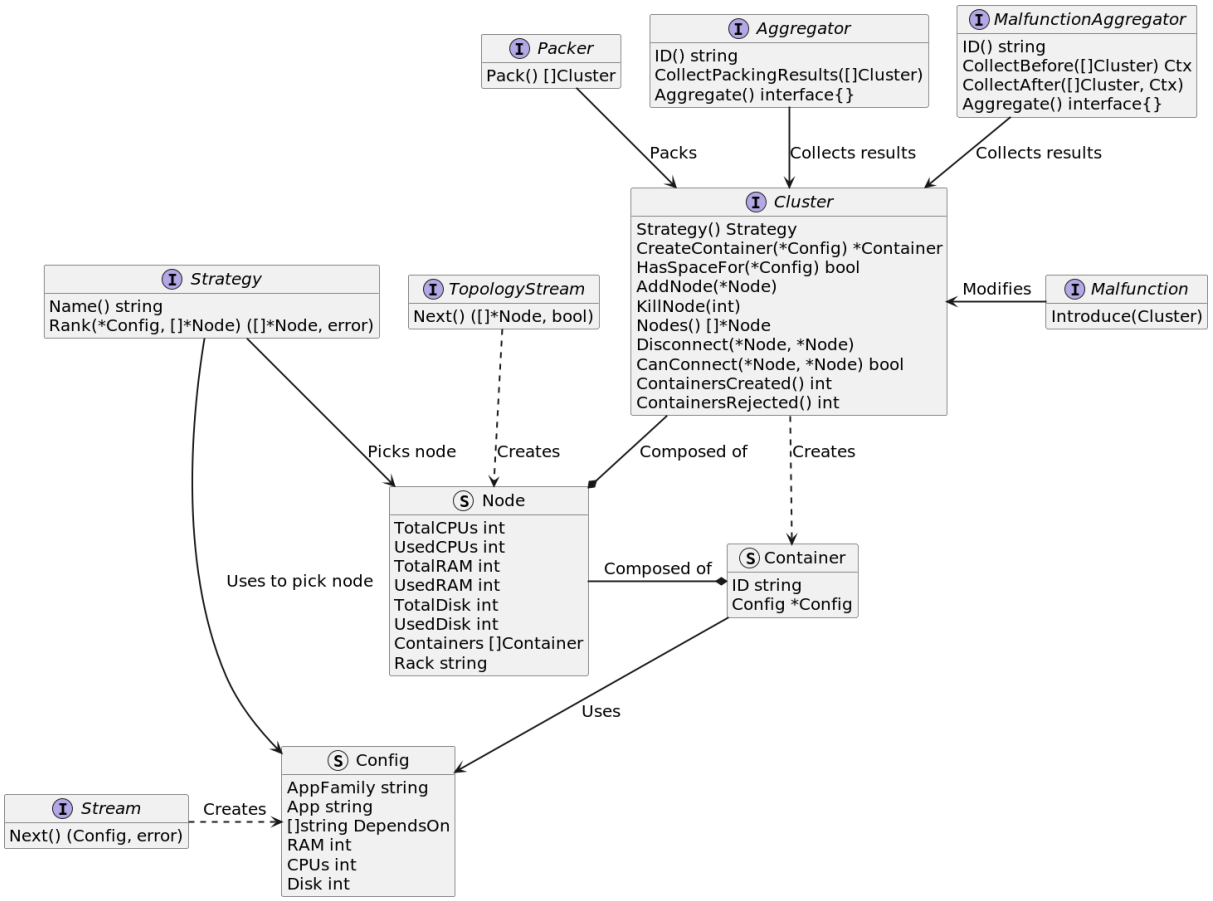


Figure 4: Class diagram of key system components (types notion is Golang specific).

represents these data points. For example, the aggregator might count the number of containers deployed by each strategy and then compute the average number of containers deployed. The aggregation phase occurs after the last iteration for the current topology has been executed. The flexible interface of the results aggregator allows for the production of a wide range of statistical information. For instance, with all the data points collected, an aggregate might include percentile or median values. The aggregated results are stored within the experiment and get associated with the corresponding topology.

One of the desired characteristics of a strategy is its management of the fault tolerance of the deployed system. The "malfunction" component assists in testing this aspect [23]. It's important to describe the malfunction in combination with the malfunction results collector. This collector is similar to the previously described collector, with the primary difference being that it collects results twice: before and after the malfunction is introduced. This approach enables the malfunction results collector to compare changes resulting from the operation of the malfunction algorithm. For instance, it can assess how many applications or application families survived a network partition [24]. The malfunction operates within the cluster, deliberately causing a disruption, such as removing a node from the cluster (figure 5) or reducing the percentage of available connections. Since everything is interconnected by default, the cluster provides a means to disconnect two nodes.

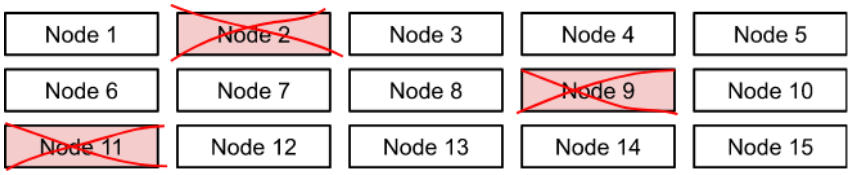


Figure 5: Example of malfunction removing random nodes in the cluster.

4. Parallel execution

The organization of the flow facilitates faster experiment execution in multiprocessor systems [25]. Each experiment iteration is executed in a separate thread, which accelerates the overall experiment execution speed, especially since most of the work occurs in the packer. To effectively collect results between iterations, proper synchronization techniques must be employed. In this context, a common Mutex implementation will suffice. Within the scope of a single experiment, it definitely makes sense to parallelize both iterations and topologies, as they can be executed independently of each other.

Furthermore, the application architecture allows for a higher level of parallelization, presenting additional opportunities. The system can be utilized to identify the most suitable topology for a given sequence of containers. This can be achieved by executing different experiments, each with a distinct topology stream. Virtually any configuration or additional testing techniques can be applied at this higher level, utilizing the existing experiment mechanics. Since each experiment is self-contained, these algorithms can also be parallelized.

5. High level organisation of the system

A client-server architecture [19] is a recommended choice for such an application. Firstly, the implementation of the previously described components is separate from the visualization of the experiment results. The system's flexible state allows for the choice of whether to represent such results with user interface components, or whether another system should simply delegate the execution of experiments to this one while making decisions based on the experiment results. Another advantage of adopting a client-server architecture is the ability to have multiple server instances, thereby enabling high-level parallelization of experiment execution. Additionally, having multiple server instances enhances the overall resilience of the system.

The server component of the system must expose an application programming interface (API) to utilize the previously described features (figure 6). Modern client-server systems typically use REST API [26] or gRPC [27]. The API functions of the proposed software are straightforward and can be implemented using the most preferred approaches. These functions include:

- Create experiment: This function creates a new experiment with all default values set, which cannot be run yet. The created experiment will be in the NEW state.
- Update experiment: This method adds new configurations to the experiment. It allows for step-by-step configuration of the experiment, cloning of experiments, and modification of their parts. It technically facilitates quick testing of various hypotheses.
- Find experiments by id or other attributes: This function enables the retrieval of information on previously run experiments and their results, as the results are part of the experiment data.
- Clone experiment: This creates an identical clone of an experiment, which can then be modified to observe different behaviors. With many configuration options available, it makes sense to change some dimensions and observe how the results vary. For example, adding a new strategy to the list of tested ones and observing the impact on results.
- Execute experiment: This starts the experiment execution, transitioning the experiment to the EXECUTING state and commencing the broadcasting of all previously described events.
- Interrupt experiment execution: This stops the experiment execution and transitions the experiment to the INTERRUPTED state. In cases where it becomes apparent that the experiment is not yielding expected results, continuing the execution would be unproductive and consume more resources. The experiment can thus be interrupted to conserve resources.
- Subscribe to experiment events: Once a subscription is made, the subscriber will receive all the events of interest.

The state of the experiment encompasses all the experiment results, even if the experiment is currently running. Since intermediate results are still useful, they must be distributed over the API to interested consumers. These events include:

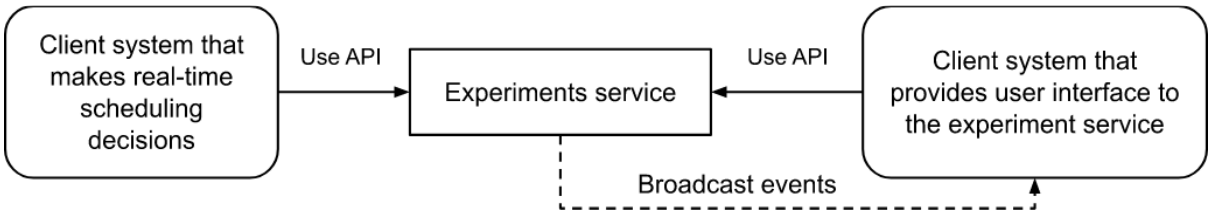


Figure 6: Example of two clients and one server instance.

- Experiment state changes.
- Experiment iteration completion. This event can be throttled to avoid overwhelming the client system.
- Experiment result available. Sent every time the experiment is executed for one of the cluster topologies, indicating that there is a new experiment result entry available, which can then be represented on the client side.
- Experiment execution for a topology started. This event is purely technical and ensures that the client side accurately displays the necessary progress.

Internally, broadcasting is implemented following the event listener pattern [20], where the system's role is to send the event to interested subscribers. Externally, these events will be broadcast over the network to connected clients, enabling them to make quick decisions regarding the progress of experiment execution.

6. Interpretation of experiment results

The aim of results interpretation is to determine which strategy performs better or worse in certain scenarios. Charts and tables are ideal tools for illustrating such comparisons. The comparison itself is based on the experiment results, which include values produced by the aggregators. Each set of values has an identification that allows for differentiating between the aggregates.

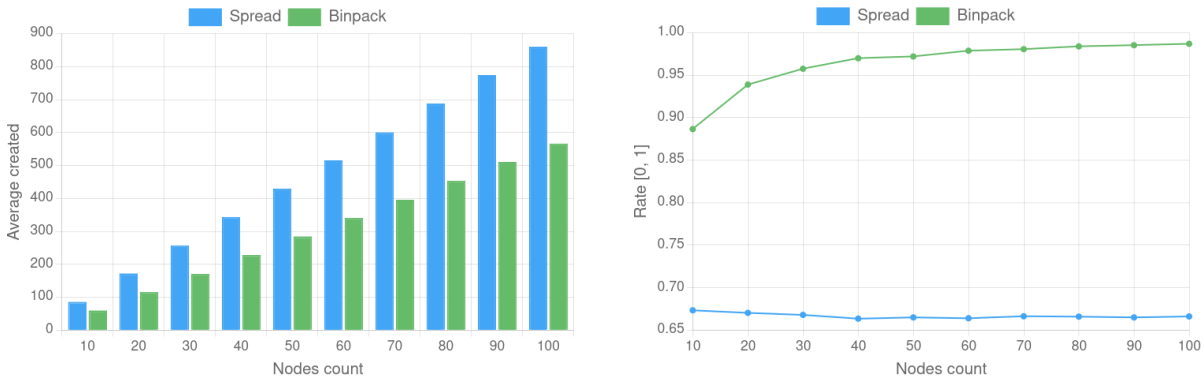
For instance, suppose an aggregator computes the average number of containers created by different strategies. The results are then represented as the average number of containers per strategy for each topology. To analyze these results, a histogram chart can be used [28]. An example histogram (figure 7a) might clearly indicate that, for all topologies, the binpack strategy managed to create fewer containers on average before the packing condition was met in this particular experiment setup.

Additionally, rates, such as the container creation rate (the percentage of successfully satisfied scheduling requests), can be displayed using a line chart. In an example (figure 7b), the "binpack" strategy may be shown to reject significantly fewer container scheduling requests compared to the "spread" strategy.

7. Discussion

While this article comprehensively covers the design of the application, it is important to remember that this is not the software itself. The choice of technology and the discussion around the alternative higher-level organization of components remain open topics. A judicious selection of technology and supporting infrastructure is crucial to ensure that the designed software remains both flexible and scalable. One effective approach to organize such a project is to make it open source, thus allowing contributions from all interested parties.

Another promising direction for this system is the development of a real-time system that relies on experiment results to make further scheduling decisions. It's also important to ensure that the system can be extended with new key performance indicators and strategy algorithms. A potential next step



(a) (b)

Figure 7: Example of charts used to represent experiment results, (a) – average created containers, (b) – container creation rate.

could be exploring the most suitable structure for the cluster, rather than just looking for a strategy that fits a certain setup. By doing so, the experiment could provide even more valuable insights.

8. Conclusion

This paper delves into the design of a system that enables the evaluation of scheduling strategy algorithms within the context of distributed systems, particularly in relation to microservices architectures where COS is extensively utilized. The related works underline the importance of selecting the appropriate strategy and show how such a decision could affect different parts of distributed systems, like resource utilization or fault tolerance.

One of the main goals when designing such a system is to ensure its flexibility. This flexibility allows for the testing of different aspects of distributed systems reliant on COS. The proposed division of responsibilities among different components, such as the topology stream, configuration stream, packer, strategy, cluster, nodes, containers, aggregators, and malfunctions, allows for extensive customization of the experiment flow to achieve the desired behavior. For example, such system can be used to compare the degree of resource fragmentation on the cluster nodes and thus assess the efficiency of resource utilization, measure the rates of containers creation or rejection, evaluate the availability of applications encountering various network partitions or node failures. The client-server organization of these components separates the representation of results from the experiment execution itself. This separation removes any assumptions about how experiment results can be utilized, thereby opening up a variety of other use cases, such as enabling a higher-level system that relies on the experiment's API for making scheduling decisions.

The next step would be the implementation of such a system. This could significantly accelerate further research in the fields of resource distribution and distributed systems. The system would not only offer a platform for experimentation but also become a valuable source of knowledge about scheduling algorithms.

9. Authors contribution

The authors confirm contribution to the paper as follows: study conception and design: Voievodin Y., Rozlomii I.; data collection: Voievodin Y.; analysis and interpretation of results: Voievodin Y., Rozlomii I.; manuscript preparation: Voievodin Y. All authors reviewed the results and approved the final version of the manuscript.

References

[1] M. A. Rodriguez, R. Buyya, Container-based cluster orchestration systems: A taxonomy and future directions, Software: Practice and Experience 49 (2019) 698–719. doi:https://doi.org/10.1002/spe.2660.

[2] T. Siddiqui, S. A. Siddiqui, N. A. Khan, Comprehensive Analysis of Container Technology, in: 2019 4th International Conference on Information Systems and Computer Networks (ISCON), 2019, pp. 218–223. doi:10.1109/ISCON47742.2019.9036238.

[3] H. M. Fard, R. Prodan, F. Wolf, Dynamic Multi-objective Scheduling of Microservices in the Cloud, in: 2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC), 2020, pp. 386–393. doi:10.1109/UCC48980.2020.00061.

[4] P. Kumari, P. Kaur, A survey of fault tolerance in cloud computing, Journal of King Saud University - Computer and Information Sciences 33 (2021) 1159–1176. doi:10.1016/j.jksuci.2018.09.021.

[5] Z. Zhong, M. Xu, M. A. Rodriguez, C. Xu, R. Buyya, Machine Learning-based Orchestration of Containers: A Taxonomy and Future Directions, ACM Comput. Surv. 54 (2022) 217. doi:10.1145/3510415.

[6] Y. Voievodin, I. Rozlomii, A. Yarmilko, Approach to Evaluate Scheduling Strategies in Container Orchestration Systems, in: Modeling, Control and Information Technologies: Proceedings of International scientific and practical conference, 6, 2023, pp. 292–295. doi:10.31713/mcit.2023.089.

[7] V. Bushong, A. S. Abdelfattah, A. A. Maruf, D. Das, A. Lehman, E. Jaroszewski, M. Coffey, T. Cerny, K. Frajtak, P. Tisnovsky, M. Bures, On Microservice Analysis and Architecture Evolution: A Systematic Mapping Study, Applied Sciences 11 (2021) 7856. doi:10.3390/app11177856.

[8] I. M. A. Jawarneh, P. Bellavista, F. Bosi, L. Foschini, G. Martuscelli, R. Montanari, A. Palopoli, Container Orchestration Engines: A Thorough Functional and Performance Comparison, in: ICC 2019 - 2019 IEEE International Conference on Communications (ICC), 2019, pp. 1–6. doi:10.1109/ICC.2019.8762053.

[9] A. Saboor, M. F. Hassan, R. Akbar, S. N. M. Shah, F. Hassan, S. A. Magsi, M. A. Siddiqui, Containerized Microservices Orchestration and Provisioning in Cloud Computing: A Conceptual Framework and Future Perspectives, Applied Sciences 12 (2022) 5793. doi:10.3390/app12125793.

[10] J. Flora, P. Gonçalves, M. Teixeira, N. Antunes, A Study on the Aging and Fault Tolerance of Microservices in Kubernetes, IEEE Access 10 (2022) 132786–132799. doi:10.1109/ACCESS.2022.3231191.

[11] S. V. Gogouvitis, H. Mueller, S. Premnadh, A. Seitz, B. Bruegge, Seamless computing in industrial systems using container orchestration, Future Generation Computer Systems 109 (2020) 678–688. doi:10.1016/j.future.2018.07.033.

[12] A. Akuthota, Chaos Engineering for Microservices, A Starred Paper Submitted in Partial Fulfillment of the Requirements for the Degree Master of Science in Computer Science, St. Cloud State University, 2023. URL: https://repository.stcloudstate.edu/csit_etds/42/.

[13] J. Jang, J. Jung, J. Hong, An efficient virtual cpu scheduling in cloud computing, Soft Computing 24 (2020) 5987–5997. doi:10.1007/s00500-019-04551-w.

[14] A. P. Shriniwar, Container Scheduling Using TOPSIS Algorithm, Msc research project, National College of Ireland, Dublin, 2020. URL: https://norma.ncirl.ie/4551/.

[15] T. Du, G. Xiao, J. Chen, C. Zhang, H. Sun, W. Li, Y. Geng, A combined priority scheduling method for distributed machine learning, EURASIP Journal on Wireless Communications and Networking 2023 (2023) 45. doi:10.1186/s13638-023-02253-4.

[16] B. Liu, P. Li, W. Lin, N. Shu, Y. Li, V. Chang, A new container scheduling algorithm based on multi-objective optimization, Soft Computing 22 (2018) 7741–7752. doi:10.1007/s00500-018-3403-7.

[17] C. Kaewkasi, K. Chuenmuneewong, Improvement of container scheduling for Docker using Ant Colony Optimization, in: 2017 9th International Conference on Knowledge and Smart Technology

(KST), 2017, pp. 254–259. doi:10.1109/KST.2017.7886112.

[18] L. Li, J. Chen, W. Yan, A particle swarm optimization-based container scheduling algorithm of docker platform, in: Proceedings of the 4th International Conference on Communication and Information Processing, ICCIP '18, Association for Computing Machinery, New York, NY, USA, 2018, p. 12–17. doi:10.1145/3290420.3290432.

[19] S. A. Hamid, R. A. Abdulrahman, R. A. Khamees, What is Client-Server System: Architecture, Issues and Challenge of Client-Server System, Recent Trends in Cloud Computing and Web Engineering 2 (2020) 1–6. doi:10.5281/zenodo.3673071.

[20] E. Freeman, E. Robson, Head First Design Patterns, O'Reilly Media, 2020. URL: https://github.com/ajitpal/BookBank/blob/master/%5BO%60Reilly.%20Head%20First%5D%20-%20Head%20First%20Design%20Patterns%20-%20%5BFreeman%5D.pdf.

[21] C. Surianarayanan, G. Ganapathy, R. Pethuru, Essentials of Microservices Architecture Paradigms, Applications, and Techniques, Taylor & Francis, 2019.

[22] L. Abdollahi Vayghan, M. A. Saied, M. Toeroe, F. Khendek, Deploying Microservice Based Applications with Kubernetes: Experiments and Lessons Learned, in: 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), 2018, pp. 970–973. doi:10.1109/CLOUD.2018.00148.

[23] J. Bergstrom, Chaos Engineering, The ITEA (2022) 208. URL: https://itea.org/the-itea-journal/journal-at-a-glance/.

[24] M. Kleppmann, Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems, O'Reilly Media, 2017.

[25] M. Herlihy, N. Shavit, V. Luchangco, M. Spear, The Art of Multiprocessor Programming, Newnes, 2020. URL: https://cs.ipm.ac.ir/asoc2016/Resources/Theartofmulticore.pdf.

[26] H. Subramanian, P. Raj, Hands-On RESTful API Design Patterns and Best Practices: Design, develop, and deploy highly adaptable, scalable, and secure RESTful web APIs, Packt Publishing Ltd, 2019.

[27] K. Indrasiri, D. Kuruppu, gRPC: Up and Running: Building Cloud Native Applications with Go and Java for Docker and Kubernetes, O'Reilly Media, 2020.

[28] S. D. H. Evergreen, Effective Data Visualization: The Right Chart for the Right Data, 2 ed., SAGE publications, 2019.