Approach to Evaluate Scheduling Strategies in Container Orchestration Systems

https://doi.org/10.31713/MCIT.2023.089

Yevhenii Voievodin
Department of Information Technologies
Bohdan Khmelnytsky National University of Cherkasy
Cherkasy, Ukraine
yevhenii.voievodin@vu.cdu.edu.ua

Inna Rozlomii
Department of Information Technologies
Bohdan Khmelnytsky National University of Cherkasy
Cherkasy, Ukraine
inna-roz@ukr.net

Andrii Yarmilko
Department of Automated Systems Software
Bohdan Khmelnytsky National University of Cherkasy
Cherkasy, Ukraine
a-ja@ukr.net

Abstract—**This article offers an in-depth examination of scheduling strategies in container orchestration systems, emphasizing the importance of selecting the most suitable strategy for a system's specific needs. The article explores the evolution of container systems and challenges around cloud reliability and fault tolerance. A structured experimental methodology was introduced to compare different strategies systematically, utilizing various testing methods like packing until first rejection, packing until the cluster is full, and introducing different malfunctions such as liveness and network partitioning. Comparative results between the "binpack" and "spread" strategies highlighted that while the former excels in higher scheduling requests acceptance rate, the latter is superior in offering higher availability. The findings show the need for aligning the scheduling strategy with system objectives, be it resource optimization, fault tolerance or other desired characteristics.**

Keywords—container orchestration system; Kubernetes; Docker; scheduling strategy; cluster resource utilization; availability; fault tolerance; resource allocation

I. Introduction

Container orchestration systems (COS) are essential tools in modern computing environments due to their central role in efficiently managing and coordinating containerized applications. A container image is a self-contained, immutable package that encapsulates an application's code, runtime, libraries, and dependencies, ensuring consistent deployment across various environments [1]. Containers, in turn, are lightweight, portable, and isolated instances created from container images, facilitating the efficient utilization of system resources and ensuring application isolation. While Docker stands as one of the most widely recognized technologies for containerization [2, 3], it is important to note that several alternative containerization solutions exist in the ecosystem.

One of the crucial parts of the COS is a scheduler. Its primary responsibility is to determine the optimal placement of containers within the cluster, which consists of nodes. Nodes represent physical computing resources, encompassing attributes such as available random-access memory (RAM), capabilities of central processing unit (CPU), and storage capacity. The complexity of placement optimization is heightened by the dynamic nature of both container attributes and the fluctuating quantity of containers to be deployed within a cluster.

Among the commonly employed strategies are "binpack" and "spread" [4]. The "binpack" strategy aims to optimize resource usage by efficiently filling up a node with as many containers as possible before considering other nodes. This approach minimizes the initial use of cluster resources, as additional nodes are only brought into play once the initial set of nodes is fully utilized with containers. The "spread" strategy is designed to place containers on nodes with the greatest available physical resources, typically selecting the least occupied node within the cluster. This approach leads to the dispersion of containers throughout the cluster, enhancing the overall fault tolerance [5] and availability [6] of the deployed applications.

The presented strategies show which of the desirable qualities might be expected when the scheduling strategy is selected to be used by COS. It's worth noting that strategies can be more intricate than the ones outlined. For instance, a "spread" strategy could extend beyond selecting merely the least occupied node and instead aim to choose a node located on a distinct physical rack [7], further enhancing deployment resilience and fault tolerance.

The following two use-cases for COS shed light on application deployment patterns that must be taken into account when selecting a scheduling strategy, along with the key metrics to consider.

In the first use-case, an application is designed to allow users to execute custom-written code, such as an online integrated development environment (IDE) [8]. Containers prove advantageous in this context due to their inherent capacity to provide the requisite level of program isolation on individual nodes. Moreover, services like online IDEs often necessitate support for various technology stacks, rendering containers a suitable choice. A scheduling strategy such as "binpack" may be deemed advantageous for optimizing resource allocation during these brief program evaluations.

In the second scenario, an online banking application employs a microservices architecture [9] consisting of 200 microservices, each assigned a specific level of importance. To enhance system robustness, the COS deploys multiple instances of identical containers for the more critical microservices, thereby improving overall system availability. Given the intricate network of interdependencies among microservices, the primary concern revolves around fault tolerance. In this context, where the online banking platform strives for heightened availability, a scheduling strategy like "spread" proves more effective.

II. Related Works

One of the first systems for large-scale cluster management was Google's Borg. Borg provided tools for efficient, reliable management of applications in clusters. Burns et al. presented an insightful analysis of Borg, Omega, and Kubernetes, detailing lessons learned over a decade of container management at Google [10]. This work laid the foundation for Kubernetes, now a dominant COS in the field.

Comprehensive analysis of fault tolerance in the cloud computing [11] that was done by Deepak K. G. et al. shows the variety of faults and mechanisms to prevent them in the cloud, which a typical environment for the COS based systems. The paper also covers the metrics of fault tolerance, such as: reliability, availability, usability etc. Authors make conclusion that there are still challenges associated with building fault tolerant systems.

Machine learning techniques also have use-cases in the COS systems. Zhong Z. et al., describe idea behind using machine learning in scheduling algorithms [12] and provide comparison of different studies in the area of machine learning. Authors make conclusion that there is no systematic approach to build a complete machine learning-based optimization framework. Which implies the need to compare different machine-learning based algorithms effectively.

III. Key Performance Indicators for Scheduling Efficiency

As previously discussed, the choice of scheduling strategy depends on aligning with specific attributes that hold significance for the COS-dependent system. The effectiveness of a scheduling strategy is contingent upon its ability to closely match these chosen attributes:

- Cluster resource utilization. This metric quantifies the efficient utilization of cluster resources. It can be measured as the ratio of used resources to the total resources available. Often likened to fragmentation [13], higher fragmentation levels are unfavorable as they constrain the scheduler's options and result in underutilized resources.
- Container deployment density. The number of containers deployed within the cluster is a pivotal metric. Maximizing this number is desirable, particularly for services reliant on COS that prioritize serving a larger client base.
- Scheduling request satisfaction ratio. This metric offers insight into how effectively scheduling requests are fulfilled. It is calculated as the ratio of rejected scheduling requests to the total scheduling requests in the scheduler queue.
- Availability and fault tolerance. For systems with stringent availability requirements, such as online banking, high availability is paramount and closely linked to fault tolerance.

IV. Experiment Framework for Strategy Evaluation

The proposed methodology outlines a sequence of actions for collecting metrics and evaluating the efficiency of different strategies. An "experiment" represents a series of actions that are systematically applied to a particular cluster configuration and replicated across various strategies. The core objective of a concrete experiment is to determine which strategy best suits a given configuration, with all strategies following the same container sequence and cluster setup.

Experiment consists of multiple iterations. Within a single iteration of the experiment, the following phases unfold:

- Setup iteration phase. Prior to executing the iteration, cluster instances must be created for each tested strategy. Additionally, a stream of scheduler requests is prepared. This stream is essential for distributing identical requests to different clusters, allowing different strategies to schedule the same sequence of containers.
- Packing phase. During this phase, container instances are placed within the cluster until specific conditions are met. These conditions are contingent on the packer's configuration, which will be detailed in subsequent sections.
- Packing results collection phase. After the packing phase is completed, various data points are collected, including metrics such as the overall resource utilization or the number of containers created. These data points are stored in a shared storage repository for subsequent analysis.
- Malfunction phase. If a malfunction is configured, it is introduced to the cluster where resources have been packed.
- Malfunction results collection phase. Similar to the collection of packing results, this phase focuses on specific aspects of availability or fault tolerance in the presence of malfunctions. The results are saved in the shared repository, which is accessible across iterations.

Finally, upon gathering the raw data points, the experiment proceeds to compute the desired metric values. Due to the raw nature of these data points, there exists a multitude of approaches for deriving the final metrics. These may encompass calculations such as computing averages, determining minimum or maximum values, and assessing percentile values.

V. Efficiency Testing Techniques for Scheduling Strategies

A. Halting at Scheduling Request Rejection

The decision of when to cease attempting to allocate containers to the cluster is primarily determined by the packing algorithm, which relies on values generated by the scheduler requests stream. The "until first rejection" packing strategy halts its efforts to assign containers to the cluster as soon as it encounters its initial scheduling request rejection, indicating an insufficient allocation of physical resources for a particular container within the given cluster.

Given that the packer is responsible for managing multiple strategies concurrently, the packing process halts for the specific strategy encountering the rejection while continuing for the others. This approach is instrumental in the experimental process, as it directs attention towards identifying a strategy capable of dynamically allocating resources in a manner that minimizes fragmentation and resource wastage, while aims to avoid schedule request rejections for as long as possible. In this particular context, the fundamental metrics for comparative analysis of strategies comprise the quantification of generated containers and the coefficient denoting resource utilization. Experimental iterations can be executed on clusters featuring distinct node sets.

B. Full Cluster Utilization

The core concept behind the "pack until the cluster is full" method is to prioritize maximum resource utilization by disregarding failures. It's important to note that a cluster is deemed "full" when it lacks the physical resources necessary to accommodate a container configured with the most minimal requirements. The key idea is to stop packing phase after cluster cannot serve scheduling request with minimal possible requirements. Consequently, even with this approach, fragmentation may persist, as it hinges on the specific criteria defining those minimal requirements.

The primary metrics for evaluation encompass the count of containers successfully allocated to clusters, where a higher count indicates better performance, as well as the ratio of successful scheduling requests to the total number of requests. The strategy that excels is one that efficiently places a greater number of containers with a high success rate.

C. Liveness Malfunction

An application deployed within a container is considered available when it can execute its core functions without degradation. Liveness testing is designed to assess both the availability of applications and the fault tolerance characteristics of scheduling strategies. This is achieved by simulating real-world hardware issues that occasionally occur in data centers, such as node or rack failures [14].

As previously mentioned, there are various deployment methods in COS. In some cases, a single application container is duplicated two or more times to ensure that if one instance experiences problems, the others can continue to perform the application's core functions.

The algorithm for testing availability is relatively straightforward: specify a percentage of nodes to be deliberately taken offline, randomly select nodes within the cluster for removal, and then assess how many applications remain available. By repeating this process multiple times, as suggested by the experiment's lifecycle, a comprehensive understanding of availability properties can be obtained.

D. Network Partition Malfunction

Certain applications, like the online banking microservice system used as an example previously, rely on network communication. In such cases, these applications become inaccessible if any of their direct or transient dependencies experience downtime. In this context, a direct dependency is one that an application container directly utilizes to fulfill specific functions, while a transitive dependency is one that the application container relies on indirectly through its direct dependencies.

The malfunction is engineered to disrupt network connections between specific nodes within clusters. After a predetermined number of connections are severed, the collector can calculate how many applications remain accessible. It's important to note that an application is considered available if all of its dependencies are reachable within the isolated network partition [15]. Additionally, the container that directly depends on the container being assessed for connectivity must itself be accessible through a direct connection to a node, simulating a scenario akin to client-side load balancing, where the first container attempts to connect to one of the known nodes. The same principle applies to other dependencies of other containers.

VI. Example of Strategies Comparison

In the first example, the "pack until full" packing algorithm is applied. Containers in this scenario specify only their RAM requirements in Gigabits (GiB). The stream generates random containers in the following sequence with associated probabilities: 1 GiB, 2 GiB, 3 GiB – 50%; 5 GiB, 6 GiB – 33%; 8 GiB, 16 GiB – 12%; 32 GiB – 5%. No malfunctions are introduced into this experiment. Two strategies are under evaluation: "binpack" and "spread". The experiment is repeated ten times, each time with an increment of ten nodes in the cluster. Fig. 1 illustrates the containers creation create. The experiment result implies that "binpack" is preferable when aiming for a higher scheduler acceptance rate.

In the second example, the "pack until first rejection" algorithm is used. The stream configuration remains the same as in the previous example, except that each container is deployed twice. Additionally, this experiment introduces malfunctions by killing 20% of random nodes. Fig. 2 illustrates the application survival rate (the percentage of applications that continue to function), which is notably higher, surpassing 10%, in favor of the "spread" strategy.

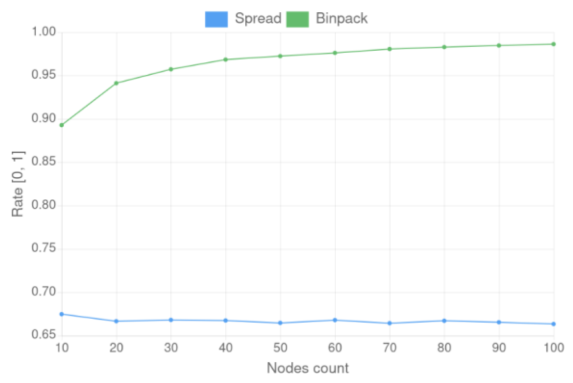

Figure 1. Containers creation rate chart

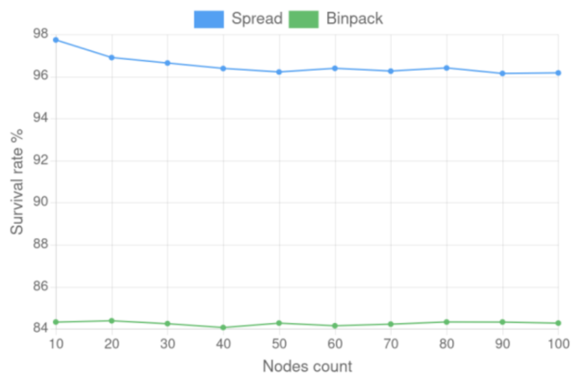

Figure 2. Applications survival rate chart

VII. Discussions

The evolution and growth of COS have opened a new frontier for large-scale application deployment and management. While the introduced experimental framework is comprehensive, real-world scenarios may introduce complexities not captured in our controlled environment. For example, variable network latencies, hardware discrepancies, and unpredictable workloads could influence the effectiveness of a chosen strategy. The framework can be extended to include more performance indicators and testing methods. This includes integrating metrics that are specific to certain types of schedulers. For instance, there might be metrics unique to prediction-based algorithms. While this narrows the range of strategies being examined, it provides additional data to make more informative strategy selection decision. Sharing the findings with the broader community could lead to collaborative improvements and innovative solutions.

VIII. Conclusion

One of the central findings of this article is the critical importance of scheduling in COS systems. The efficiency of an application can be greatly impacted by the chosen scheduling approach, with strategies like "binpack" and "spread" demonstrating that the optimal method varies based on context and workload.

Thorough testing, especially in scenarios that mimic network partitions and other malfunctions, is important. Such a testing framework exposes key characteristics of different strategies and helps to select the most suitable strategy for a specific application. Looking ahead, as applications continue to increase in complexity, the influence and utility of COS will only increase. Innovations, particularly those involving machine learning, could lead to a new era of COS schedulers.

This journey through COS scheduler has provided valuable insights. However, it's essential to underline that this article represents just a fragment of the potential knowledge. Collaboration on a global scale will be the key to unlocking the next stages of advancement in container scheduling algorithms.

References

[1] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in Proceedings of the tenth european conference on computer systems, 2015, pp. 1-17.

[2] C. Anderson, "Docker [software engineering]", Ieee Software, vol. 32(3), 2015, pp. 102-c3.

[3] A. Mouat, "Using Docker: Developing and deploying software with containers," O'Reilly Media, Inc., 2015.

[4] C. Cérin, T. Menouer, W. Saad, and W. B. Abdallah, "A new docker swarm scheduling strategy," in 2017 IEEE 7th international symposium on cloud and service computing (SC2), IEEE, 2017, pp. 112-117.

[5] P. Kumari, and P. Kaur, A survey of fault tolerance in cloud computing, Journal of King Saud University-Computer and Information Sciences, vol. 33(10), 2021, pp. 1159-1176.

[6] M. Nabi, M. Toeroe, and F. Khendek, "Availability in the cloud: State of the art," Journal of Network and Computer Applications, vol. 60, 2016, pp. 54-67.

[7] K. Senjab, S. Abbas, and N. Ahmed, "A survey of Kubernetes scheduling algorithms," Journal of Cloud Computing, vol. 12(1), 2023, pp. 1-26.

[8] S. Bartkova, "Research of online IDE and analysis of directions of development," 2021 [Card-file ontu.edu.ua].

[9] S. Newman, "Building microservices," O'Reilly Media, Inc. 2021.

[10] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes: Lessons learned from three container-management systems over a decade," Queue, vol. 14(1), 2016, pp. 70-93.

[11] D. K. Gaur, and A. Mahalkari, "Comparative Analysis of Fault Tolerance Techniques in Cloud Computing," International Journal of Computer Science and Information Technologies (IJCSIT), vol. 11(4), 2020, pp. 59-64.

[12] Z. Zhong, M. Xu, M. A. Rodriguez, C. Xu, and R. Buyya, "Machine learning-based orchestration of containers: A taxonomy and future directions," ACM Computing Surveys (CSUR), vol. 54(10s), 2022, pp. 1-35.

[13] A. Silberschatz, P. B. Galvin, and G. Gagne, "Operating System Concepts," 10th. ed., John Wiley & Sons, Inc, 2018.

[14] S. M. Ataallah, S. M. Nassar, and E. E. Hemayed, "Fault tolerance in cloud computing-survey," in 2015 11th International computer engineering conference (ICENCO) , IEEE, 2015, pp. 241-245.

[15] M. Kleppmann, "Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems," O'Reilly Media, Inc., 2017.