Inform : Jurnal Ilmiah Bidang Teknologi Informasi dan Komunikasi
Vol.8 No.2 July 2023, P-ISSN : 2502-3470, E-ISSN : 2581-0367

Rescheduling Strategy for Container Orchestration System to Improve Application Availability

Akhmad Alimudin[1], Rahardhita Widyatra Sudibyo[2]

[1]*Department of Multimedia Creative Technology, Politeknik Elektronika Negeri Surabaya, Indonesia*

[2]*Telecommunication Engineering, Politeknik Elektronika Negeri Surabaya, Indonesia*

[1]alioke@pens.ac.id (*)

[2]widi@pens.ac.id

Received: 2023-04-27; Accepted: 2023-05-30; Published: 2023-06-05

Abstract— Virtualization technologies such as containers have become increasingly popular and widely used today. Containers offer several advantages, such as flexibility, portability, and scalability. Container-supporting technologies such as container orchestration provide more advantages to container users to maintain the containers. One of the crucial features of container orchestration is the scheduler responsible for assigning containers to the proper server. However, container misplacement often occurs during the scheduling process, which may reduce the application's Quality of Service (QoS), such as availability. Multiple studies on scheduling strategies have been conducted with various goals, such as increasing application availability. To improve the application availability, we can reschedule the containers in the cluster. This study proposes a rescheduling strategy on the Kubernetes container orchestration system to improve the application available in the container. We use the probability approach as a solution for rescheduling actions in dynamic environments against containers that have an unhealthy state. Our experimental results show that our rescheduling strategy improves application availability compared with the Kubernetes default schedule. Rescheduling improves application availability according to experiments. After rescheduling, Kubernetes had a better application success rate. The chart demonstrates that the average success rate of applications for the threshold of 0.9 surpasses the other thresholds on average. In contrast, the availability of applications using the threshold 0f 0.7 is lower than the others. High thresholds improve application availability.

Keywords— Container Orchestration; Scheduling; Stable Matching; Quality Of Service.

I. INTRODUCTION

Cloud computing is a widely used technology for providing IT-based services, such as web services, big data, or Internet of Things applications. Cloud computing offers several advantages to users, such as cost-efficiency, flexibility, and ease of use. Cloud computing technology is closely related to virtualization technology, which is identical to virtual machine (VM) technology, having several advantages over physical machines. In recent years, virtualization technologies other than VMs, such as containers, have become increasingly popular and widely used. Containers have a basic concept of virtualization technology similar to the techniques used in VMs, and containers also offer several advantages, such as resource effectiveness, flexibility, portability, and scalability. Recently, container users have also benefited from supporting container technology, such as container orchestration, designed to manage containers. Container orchestration tools make it easier for users to manage the tens, hundreds, or thousands of containers they maintain. Some well-known container orchestration tools include Kubernetes, Docker Swarm, Marathon, and Amazon ECS.

One of the features offered by container orchestration is a scheduler responsible for assigning containers to proper hosts. The scheduler is an important component of the container orchestration system because it determines the application quality of service (QoS). Each container orchestration tool has a unique strategy for its scheduling process. Improper container placement strategies may negatively affect the QoS of containers. For instance, when the requested resource of the container is larger than the available host resources, the container's application cannot work optimally, thus decreasing the application's QoS or causing the application to stop functioning. This container misplacement problem frequently occurs in the scheduling process because the user cannot define the container resource requirements specifically; therefore, the scheduler does not know the resource requirements to properly place the container.

Moreover, the scheduler does not have historical data regarding the resource requirements of each existing container. The container's dynamic situation and the diversity in host resource capabilities pose a challenge to presenting a good scheduling strategy for container orchestration systems. Several studies regarding strategies in the scheduling process have been performed to achieve a better result.

Several efforts have been offered to improve the QoS on containers, such as Kubernetes, which provides autoscaling features to maintain the QoS. Another way to improve the QoS of containers is rescheduling the container. Rescheduling a container is quite challenging because the situations of the environment change dynamically. For instance, a container requires low resources at certain times but might require very high resources in the next few minutes. Therefore, we need an excellent strategy to obtain the pair's stability between the cluster's container and server. For example, the Kubernetes scheduler mechanism cannot manage the risk of resource overloading when several applications on the same host are competing for resources, such as the CPU [1].

DOI: http://dx.doi.org/10.25139/inform.v8i2.6220

This study offers a strategic solution for scheduling containers using a probability approach and a stable matching to solve this problem. This research aims to improve the application availability by finding the most stable pair between container and server. Our previous research proposed the creation of a scheduler for containers by implementing stable one-to-one matching (known as stable marriage problem (SMP), in a static environment [2]. We also examined a scheduling strategy using the SMP to run in dynamic conditions by rescheduling the current partner [3]. However, we do not use the SMP algorithm in this research, a one-to-one model of stable matching used in our previous research. Our current strategy is to use the hospital/resident problem algorithm [4], a many-to-one model of the stable matching problem applicable in this case, where one node may serve more than one container. We use the probability approach as a solution for rescheduling actions in dynamic environments against containers that have an unhealthy state.

In this study, we attempted to implement our scheduling strategy on Kubernetes, a well-known and widely used container orchestration system today. Our paper is organized as follows. Section 2 briefly explains the research methodology, including the system design. In Section 3, we provide and evaluate our experiments on the rescheduling process. In Section 4, we provide the conclusions of this study.

II. Research Methodology

This section describes the research methodology of this study. The research methodology in this study is described as follows:

1) Literature study: we reviewed several similar studies that had been conducted and related to our research.

2) Problem analysis: the problem is analyzed based on our literature study.

3) System design: we study the Kubernetes architecture and design our new scheduler using the stable matching problem.

4) Evaluation: we evaluate and compare our proposed system with the existing method/system.

5) Conclusion: we provide our conclusion of this study that the reader can consider.

A. Related Work

Table I summarizes the scheduling strategies for virtualization technology related to our study.

Table I
Summary of Related Work

Author	Objective	Platform	Algorithm / Technique
Kaewkasi et al. [5]	Resource utilization optimization	Docker Swarm	Ant colony
Chen et al. [6]	Energy efficiency	Cloud Simulation	Many-to-one stable matching
Cerin et al. [7]	QoS improvement	Rule-based by SLA	Rule-based
Menouer Tarek [8]	QoS improvement and energy efficiency	Kubernetes	TOPSIS multicriteria
Alimudin et al. [3]	QoS improvement	Kubernetes	Reschedule using a stable marriage problem
Rodriguez et al. [9]	Cost-efficient	Kubernetes	Optimizing initial placement, autoscaling, and rescheduling
Xu et al. [10]	QoS improvement and cost-efficiency	Virtual machine	Egalitarian stable matching
Our proposed system	Qos improvement	Kubernetes	Reschedule using a many-to-one stable matching algorithm

B. Literature Study

The main components of our proposed system. A container orchestration system is a technology that helps users manage their container assets. This study offers a scheduling strategy for Kubernetes to improve the application's QoS. Before explaining our concept in detail, we will briefly explain Kubernetes and Stable Matching in this section.

1) Kubernetes Container Orchestration System: Container orchestration is a set of operations offered by cloud providers or application managers, which are used to deploy, monitor, and dynamically manage resources to ensure the QoS [11]. This container orchestration tool enables the application manager to deploy and monitor resources dynamically to produce high availability for services that are being created. Kubernetes is a container orchestration tool widely used today, a Google open-source project previously known as Google Borg [12]. The Kubernetes cluster consists of two main parts: the Kubernetes control plane, the master node, and the Kubernetes nodes, or worker nodes. The control plane manages worker nodes and pods in a cluster. The worker node is a host for the pods to run their containers.

- As shown in Figure 1, a Kubernetes cluster has several components. This section will discuss some of the Kubernetes' components [13] related to this research.
- Kubernetes' components referenced in this work are described as follows: The master node or control plane manages the worker nodes and pods in a cluster. The master node handles the global events in a cluster, such as scheduling.
- The worker nodes are hosts for the pods to run their containers. The worker nodes can be bare metal or virtual machines.
- Pods are the smallest execution units in Kubernetes. A single pod may encapsulate one or more containers. This

DOI: http://dx.doi.org/10.25139/inform.v8i2.6220

study will find the stable pair between pods and worker nodes.

- The kube-Apiserver is a part of the control plane component responsible for exposing the Kubernetes API. The kube-Episerver represents the front end of the Kubernetes' control plane.

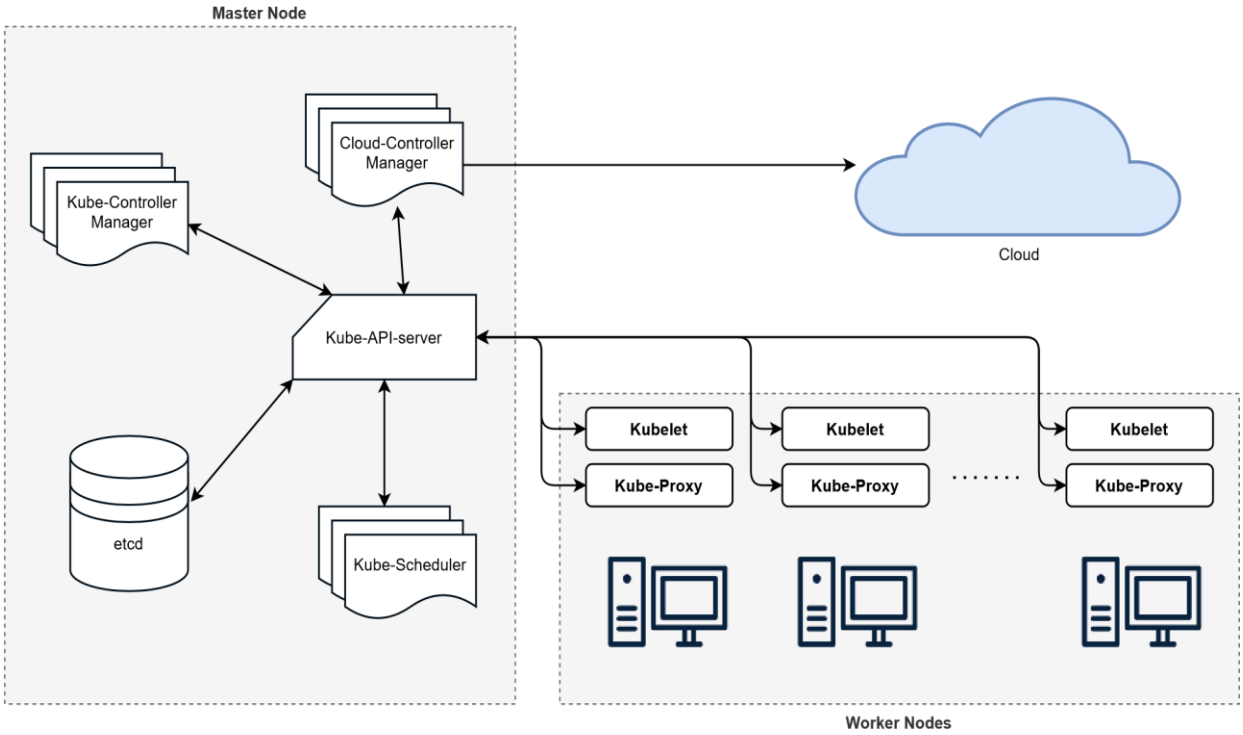


Figure 1. Kubernetes Architecture

The scheduler is one of the components that the container orchestration system must provide. As a well-known container orchestration system, Kubernetes provides a scheduling component called Kube-scheduler, located in the control plane. The Kube scheduler works in two steps [14]:

- Filtering: This step finds nodes that can meet the resource requirements of the pods' request. For instance, the scheduler will filter all available nodes to see if resources are available according to the pods' minimum requirements.
- Scoring: This step ranks nodes that pass the filtering step.

2) Stable Matching: The basic concept of stable one-to-one matching by Gale-Shapley or SMP. The core of SMP is the matching group between n men and n women, where any matched partner has no desire to switch to another partner. In other words, no pair that is better than the existing one exists called a blocking pair. This is the basic concept of the stable matching problem proposed by Gale-Shapley [15]. In Figure 2, a man in m_1 has options of woman preference order (w_1, w_2, w_3, w_4) where the preference order is based on the ranking order (1, 2, 3, 4), which means that the man m_1 likes w_1 the most, then w_2, w_3, and w_4 as the last options. The SMP algorithm matches n-man and n-woman, considered the most stable. If a blocking pair is found, it must be eliminated to achieve stability in the group match. The SMP concept is widely used to solve various economic problems. Another popular stable matching problem is the HR problem [4], a many-to-one model used to solve the placement for internship medical students (residents) in hospitals.

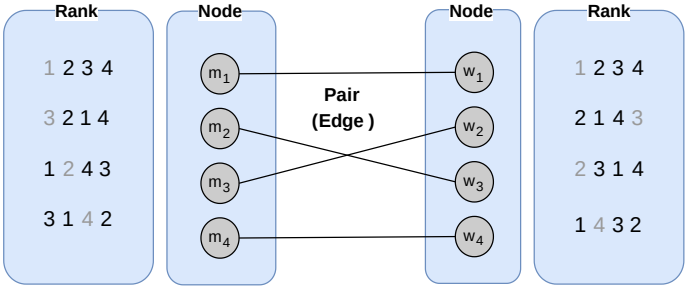


Figure 2. Illustration of SMP

C. Problem analysis

In this study, the HR problem case is suitable to use in the assignment process between containers and servers. Given the preference from the container $P_c = P_{c1}, P_{c2}, P_{c3}, ..., P_{cn}$ and the server $P_s = P_{s1}, P_{s2}, ..., P_{sm}$, we can solve the stable matching problem using the HR problem model. Matching is stable if there are no pairs in the following condition: (i) Container c

DOI: http://dx.doi.org/10.25139/inform.v8i2.6220

prefers server s over the matched pairs, (ii) Server s chooses to add container c and remove the matched container. Several improvements to the algorithm were suggested to be suitable and implementable in many case studies. In the case of SMP, we used deterministic matching, where the preferences of each pair were predefined.

D. System design

Figure 3 shows our proposed system, which is a modification of the original architecture of Kubernetes. Compared with the original architecture shown in Figure 1, the proposed system is modified by adding two components to the master node or control plane: the reschedule controller and matching scheduler. Kubernetes was built based on the Go programming language. Kubernetes client libraries are also available in various programming languages. In this study, we develop our proposed scheduler using *Python*. In the worker node part, we activate liveness probes for each pod to monitor each running pod's health condition. The step for our proposed method is as follows:

- *Reschedule the controller* to monitor the health condition of each container. If the unhealthy container is found under the threshold, then initiate rescheduling,
- *Matching Scheduler* performs the stable matching method based on the trigger of the rescheduled controller.

In contrast, the preferences are dynamic in the real case, so finding a stable partner is difficult. One of the proposed concepts for this case is using a probabilistic approach. The concepts presented by [16] about probabilistic matching use a probabilistic approach to rematch the matched pairs.

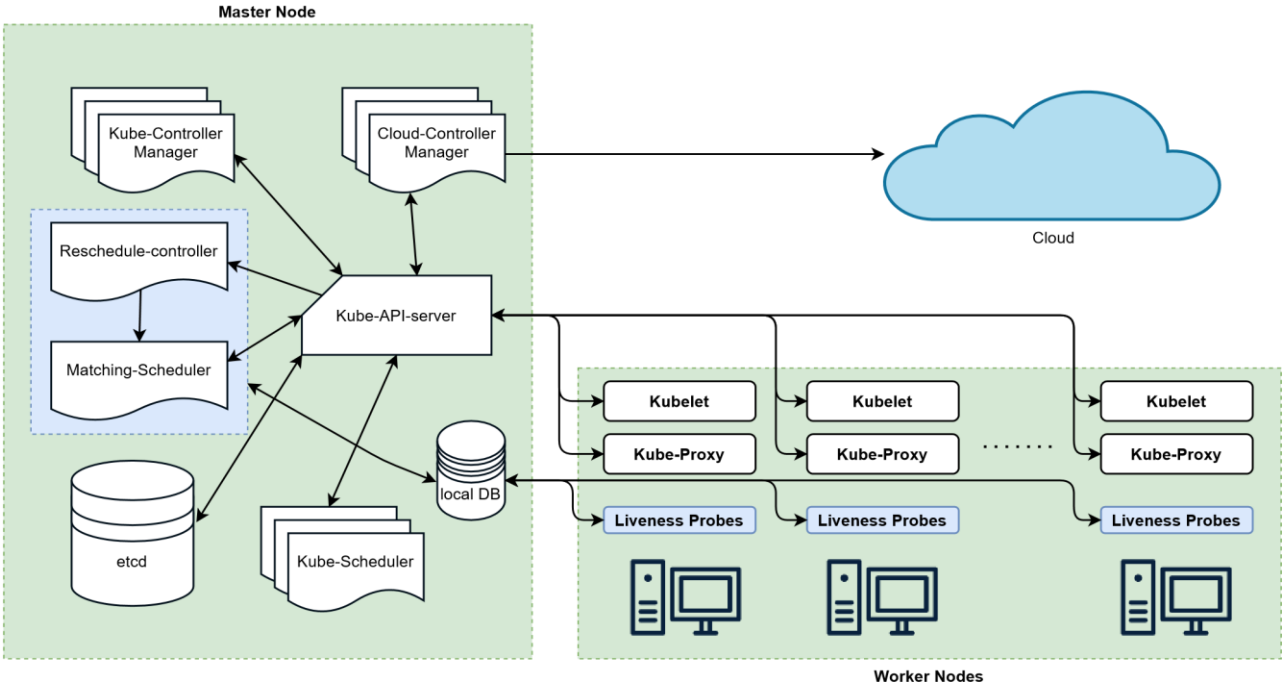


Figure 3. The Proposed Scheduler System on Kubernetes

1) Reschedule Controller: The dynamic situation in Kubernetes' clusters is challenging to find stability using the Gale-Shapley algorithm. Finding a stable pair in dynamic situations when referring to the Gale-Shapley stability concept is difficult. For example, we use CPU and memory resource usage to specify the preference orders for each pair. In dynamic situations, the order of preference in the cluster will always change because the resource request is changed dynamically. Therefore, the process of finding a stable pair will continue when a change in resource request occurs. Finding stable pairs in this dynamic situation is difficult because blocking pairs may occur frequently. To solve these dynamic situations, we offer a rescheduled controller to control the stable matching process in dynamic situations. Therefore, the stable matching process does not run continuously. Reschedule controller is a service that determines whether a running pod needs to reschedule. This service works by periodically observing a pod's activity and health condition. As shown in Figure 3, we activate the liveness probes for every deployed pod in a cluster. The liveness probes are features Kubernetes provides to detect a broken state of pods. For instance, Kubernetes uses liveness probes to decide when to restart a container inside the pod [17]. In this study, the rescheduled controller uses the liveness probes to decide whether or not the Kubernetes cluster needs to be rescheduled. Figure 4 shows how the rescheduled controller works. The liveness probes periodically check each running pod's health condition in a cluster; when the liveness probes find that the service in a container is unable to serve the requests, the liveness probes will mark this as an unhealthy pod event and send the report to the event logger. The rescheduled controller periodically calculates the probability of pod

DOI: http://dx.doi.org/10.25139/inform.v8i2.6220

happiness based on the number of unhealthy events for each pod. The rescheduled controller periodically checks the happiness probability Pr for every deployed pod and compares it with the user-defined minimum probability threshold. When the Pr is less than the threshold, the reschedule controller will initiate the reschedule in a cluster, and the next process will be handover to a matching scheduler. This reschedule-controller allows the user to determine the desired threshold. In the evaluation chapter, we will discuss the impact of the threshold on applications' QoS.

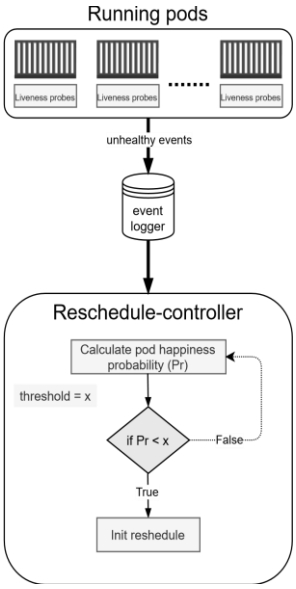


Figure 4. Reschedule controller.

2) Matching Scheduler: The scheduler is one of the Kubernetes control plane's main components responsible for assigning a newly deployed pod to available proper nodes. Kubernetes also comes with a default scheduler called the Kube scheduler. This study proposes a scheduling strategy using a stable matching problem algorithm for the Kubernetes container orchestration system, which we call a matching scheduler. The matching scheduler is a custom scheduler we developed for assigning the pod to the proper nodes in this study. Fortunately, Kubernetes provides a feature for the developer to customize the scheduler component. We develop our custom pod scheduler based on the many-to-one stable matching or HR problem, which is also used to solve medical student assignment problems such as the national resident matching program (NRMP) [18]. The reschedule controller will initiate a reschedule for the pod from the previous process in the reschedule controller when a pod's probability of happiness is less than the predetermined minimum threshold. The reschedules initiation process put the pods in a pending state. The matching scheduler works by monitoring each pending pod and is responsible for finding suitable nodes for that pod. The matching scheduler uses the many-to-one stable matching algorithm in pod assignment to servers. The algorithm requires a preference list for each pod and node to find stable matching. We need historical data from each party to discover the new preference list for each side, such as CPU and memory usage. In the next subsection, we will discuss establishing preferences from the pod to the node and vice versa.

a) Pods to worker nodes' preference

A pod is a place to run a service on the Kubernetes cluster, and a service naturally needs a place that can meet its requirements to run the service properly. Therefore, a service wants a pod to run optimally. To make a pod work optimally, we need a proper worker node to provide resources according to the pod's needs. The correct steps to assign a pod on one of the worker nodes are needed to achieve this goal. We do not have to assign the pods to the most excellent worker node to determine the pod's preference order to the worker node [6]. We simply need to find the closest distance between the worker node's resources and the pod's resource usage based on the historical data of the pod. In this case, we use the Euclidean distance formula to measure the distance. We try to find the distance using the two-dimensional Euclidean distance from the memory and CPU usage of the pod to each worker node's available memory and CPU resource. In this study, we collect the CPU and memory usage for each pod in the local database. Assuming that x is the resource capability of the worker node and y is the number of resources used by the pod, the Euclidean distance Equation (1).

$$d(x,y) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \quad (1)$$

Where the (x_1, $y1$) variable is the coordinates of one point; the (x_2, y_2) variable is the coordinates of the other point; and the d variable is the distance between (x_1, y_1) and (x_2, y_2).

After the distance from each pod to the worker node is obtained, the next step is to run the sorting to determine the preference order. To obtain the preference list order, we need to perform more steps to produce the preference list order, as shown in Algorithm 1.

Algorithm 1: pod to node preference

```
for pod in pods do
    for node in nodes do
        D = euclidean(node, pod);
        if podRequest > availableNode then
            D = D * - 1 ;
        end
        if D > 0 then
            preference.append(D)
        end
    end
    sort.asc(preference)
end
```

Algorithm 1's main iteration is for all pods deployed in the Kubernetes cluster with the default namespaces. Using the two-dimensional Euclidean distance formula, the pod will measure the distance to each worker node. In this case, we use memory and CPU as the measured resources. After the distance between the pod and the worker node has been obtained, the next step is to check whether the worker node's resource is more excellent

DOI: http://dx.doi.org/10.25139/inform.v8i2.6220

than the pod's needs. When the worker node's resource is smaller than the resource expected by the pod, the distance is marked as a negative value. The next step is obtaining the node candidates by filtering the negative value of the distance. The worker node will not be listed on the pod's preference list if the distance is negative. The last step to define the preference list is sorting the available candidates of worker nodes in ascending mode.

b) Worker nodes to Pods Preference

The viewpoint of worker nodes to pods differs from how pods view worker nodes. While pods determine worker nodes' preference based on the resource capabilities of a worker node, worker nodes determine the order of pods from another point of view by looking at resource efficiency. Several studies have revealed the relationship between CPU usage and power consumption. CPU resource usage is linearly related to power consumption [19–21]. Several studies [6, 19] used the CPU variable to show power consumption efficiency in computer devices. This study uses CPU usage as a reference for assigning worker node preferences to pods. We then use the idle CPU usage data history as a variable to determine the preference list order of worker nodes to pods. To determine the priority order, we use sorting from the CPU usage of the pods. The worker node's preference order to the pod will prioritize the pod with the highest resource usage. We prioritize pods with high CPU utility due to the linear relationship between CPU usage and power consumption based on reference [20,22], which reveals that the number of idle conditions in a data center is more frequent than those at full load.

c) Rescheduling Process

The scheduling process aims to ensure that a pod can acquire the proper node. Finding an appropriate and stable pair between pod and node in real conditions is still difficult because the system cannot find the essential information needed, such as each pod's resource usage history. The user is also unable to define the resource requirements of the pod precisely. Improper pod placement may produce several cluster problems, such as power inefficiency in the nodes and decreasing QoS in a pod due to a lack of resources. Based on those situations, a reschedule or rematch process between pods and nodes may sometimes need to be performed to obtain a stable pair between pods and nodes. One technique that can be used for assigning a pod to the node is using the many-to-one model of the stable matching algorithm. The algorithm can work optimally when the preference of each object is a certainty.

More detailed and accurate information corresponds to more optimal results. Hence, stability can be achieved. However, the matching problem we are currently discussing is dynamic, which may cause each object's preferences to fluctuate. For illustration, the current CPU requirements of a pod could differ from that of the next hour. With this dynamic preference situation, achieving a stable pair will be challenging if we use the stability concept of Gale-Shapley, where no pairs are better than the current pair. Furthermore, this situation may trigger the system to perform the matching process continuously and infinitely.

To avoid the repeated rescheduling process due to the cluster's dynamic situation, we use a probability approach to determine when the cluster's rescheduling process occurs. Figure 4 shows the rescheduled controller flow diagram process, which produces rescheduled initiation on the cluster. When a rescheduled controller initiates a rescheduling, Kubernetes gives a signal to reschedule all pods in the cluster. Afterward, our matching scheduler starts rescheduling all pods in the cluster. Figure 5 shows a flow diagram of how our matching scheduler works. The matching scheduler works based on triggers from the rescheduled controller. When triggered, the matching scheduler generates a preference for each pod and node in the cluster. The next process assigns pods to the node when a stable matching pair is obtained. The algorithm used to match the process can be seen in algorithm 2. This algorithm is based on the algorithm from the NRMP case [18]. We make a few modifications to adapt to the dynamic situation in the cluster.

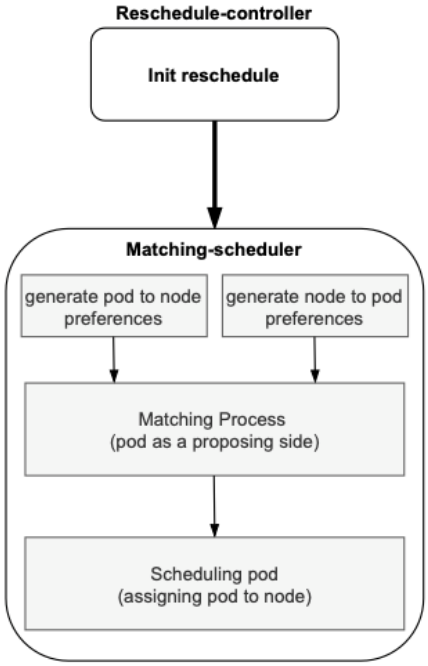


Figure 5. Matching-Scheduler Triggered by The Probability Controller To Perform The Rescheduling In A Cluster.

In the NRMP algorithm, which attempts to solve the problem of residents' placement in a hospital, each hospital can determine the residents' quota at the beginning of the process. However, in our current case, we cannot define each server's quota at the beginning of the process because each pod's resource needs change frequently. It can be seen in algorithm 2; we make modifications to the algorithm to define each server's quota to adapt in dynamic conditions, given that the resource capacity of host is RH and maximum resource usage of pod is MP. To find the quota can be written as $RH > MP + \sum_{i=0}^{n} MPi$, where *n* is the number of matched pods in the host. In this study, we are using the pod-optimal stable matching.

DOI: http://dx.doi.org/10.25139/inform.v8i2.6220

III. Result and Discussion

This section will discuss our experiences with our scheduling strategy for Kubernetes. In this experiment, we created four virtual servers that run on the VirtualBox platform. The four servers were physically located on a single 64-bit architecture physical server, which had a 12core Intel i7 processor with a speed of 3.4 GHz CPU and 64 GB of DDR-III RAM. All experiments in this study were performed on this physical machine, and each virtual server had a different CPU and memory resource configuration on each server. The four servers consisted of one master node and three worker nodes. The main server (physical server) and virtual server were running Ubuntu 18.04.3 LTS. Table II shows the resource specification of each node in the cluster.

TABLE II
NODE'S RESOURCE SPECIFICATION

Hostname	Processor number	Memory
master node	4×VCPUs	12 GB
worker01	12×VCPUs	12 GB
worker02	5×VCPUs	8 GB
worker03	8×VCPUs	16 GB

For the simulation scenario, 15 web applications with different characteristics were deployed into the Kubernetes cluster. We built 15 different PHP applications with different resource usage characteristics into docker images [23]. We built a simple arithmetic PHP application with various iterations. Each application has its unique resource usage, even when it is idle or loaded with traffic.

Algorithm 2: Pod optimal many-to-one Stable matching with dynamic quota

```
for pod in pods do
    for host in podPreferences do
        if hostRemainingResource > sum(PodMaxResourceUsage in hostMatch) then
            if pod in hostPreferences then
                hostMatch(pod)
                  lock(pod)
            else
                preference.remove(host)
            end
        else
            if pod in hostPreferences then
                for podMatch in hostMatch do
                    if rankOfPod < rankOfPairedPod then
                        hostUnmatch(podMatch)
                          unlock(podMatch)
                          hostMatch(pod)
                          lock(pod)
                    end
                end
            else
                preference.remove(host)
            end
        end
    end
end
```

To determine the effectiveness of our proposed system, we evaluate our rescheduled controller and matching scheduler components. We evaluate the rescheduled controller by applying various thresholds to discover its ability to control the occurrence of the rescheduled event. To discover the effectiveness of the matching scheduler, we evaluate the QoS of the 15 deployed applications. QoS requirements are technical criteria that define the quality of the system aspects such as performance, availability, scalability, and serviceability [24]. In this study, we focus on maintaining the availability aspect of QoS, measuring how often a system's resources and services are accessible to end users. We used Vegeta [25] as a load-test tool to measure the system's availability. We performed the load test on the application simultaneously using various load rates. The number of requests an application receives in one second is called its load rate. The application we evaluated is a web application that utilizes the standard configuration of the Apache2 server. As a result, we decided to set our maximum load rate at 150 to avoid any potentially deceptive and unhealthy occurrences that could have been generated by Apache2 itself rather than a lack of resources. Our current stable matching algorithm is pod optimal; thus, we did not evaluate the power consumption of the servers.

A. Rescheduling Evaluation

This research evaluated the rescheduled controller module we developed and attempted to prove the effectiveness of the happiness probability threshold on the scheduler's performance. In this step, we tried to compare three values of the happiness probability threshold to discover the number of reschedules in the system needed to achieve stable matching. The happiness probability threshold was used to control the intensity of the rescheduling process. We evaluated the rescheduling process using three different threshold levels in this scenario. For each

DOI: http://dx.doi.org/10.25139/inform.v8i2.6220

simulation, we gradually performed load tests for 15 deployed pods using Vegeta with various load rates. Each deployed pod was loaded with various load rates simultaneously. We evaluated how many reschedules were required to achieve a stable match for each rate.

As shown in Table 3, a threshold of 0.7 requires three total reschedules to achieve a stable pair at the 150 load rate, whereas the simulation with 0.8 and 0.9 thresholds requires 13 times rescheduling to achieve a stable match condition. The simulations show that the rescheduling process repeatedly occurred during attempts to achieve stability with a high load rate because more resources were needed when the pod attempted to serve numerous requests. Meanwhile, the availability of resources on the worker node was approaching the limit. The trade-off concept is needed to achieve stability and better QoS by removing multiple pods from the cluster. However, when we set the probability threshold to 0.7, the number of reschedules that occurred was not as high as those at thresholds of 0.8 and 0.9, and no pods were dumped for the threshold of 0.7. In the next subsection, we evaluate the performance of the pods based on the new pair generated by the reschedules of each threshold.

B. Evaluation of application availability

This research aims to improve the availability of applications in a cluster using a rescheduling strategy. To demonstrate the effectiveness of our strategy, we performed load tests using Vegeta on the 15 deployed applications. We compared application availability when using the scheduling composition using Kubernetes with three variations of the scheduling composition after rescheduling.

TABLE II
RESCHEDULE PERFORMACE

Vegeta Load rate (rate/s)	Probability Threshold								
	0.7			0.8			0.9		
	Rematch	# of rematch	dumped pods	Rematch	# of rematch	dumped pods	Rematch	# of rematch	dumped pods
50	NO	0	-	NO	0	0	NO	0	-
60	NO	0	-	NO	0	0	NO	0	-
70	NO	0	-	NO	0	0	NO	0	-
80	NO	0	-	NO	0	0	YES	1	0
90	NO	0	-	YES	1	0	YES	1	0
100	NO	0	-	NO	0	-	NO	0	-
110	YES	1	0	YES	2	0	YES	1	0
120	YES	1	0	NO	0	-	NO	0	-
130	YES	1	0	YES	9	4	YES	8	5
140	NO	0	-	YES	1	6	YES	2	5
150	NO	0	-	NO	0	-	NO	0	-

However, Table III shows that a trade-off process occurs by removing some pods from the cluster during the load test process with a rate of 130/s. Therefore, we use the rescheduling results of each threshold variation obtained during the load test with the 120/s rate in this experiment. This research simultaneously performed a load test for 5 minutes on each load rate variant for the 15 applications.

Figure 6 shows the average success rate of these 15 applications during the load test. The experimental results show that rescheduling positively impacts the application's availability. The average success rate of applications after rescheduling was higher than the success rate during the initial scheduling with Kubernetes. Figure 6 also shows the impact of the threshold in the rescheduled controller on the application's availability. The chart shows that the average success rate of applications for the threshold of 0.9 outperforms the other thresholds on average.

In contrast, the availability of the applications for the threshold of 0.7 is always below the other thresholds at a load rate above 70/s. A high threshold corresponds to better availability of the application. However, the users need to consider the trade-off effects when defining a high threshold, such as some pods being removed from the cluster to achieve a better QoS.

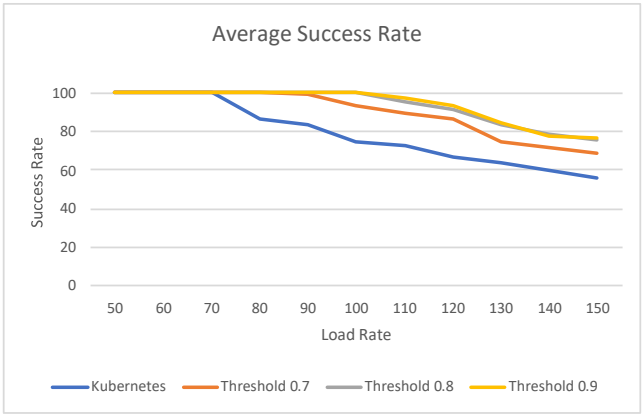
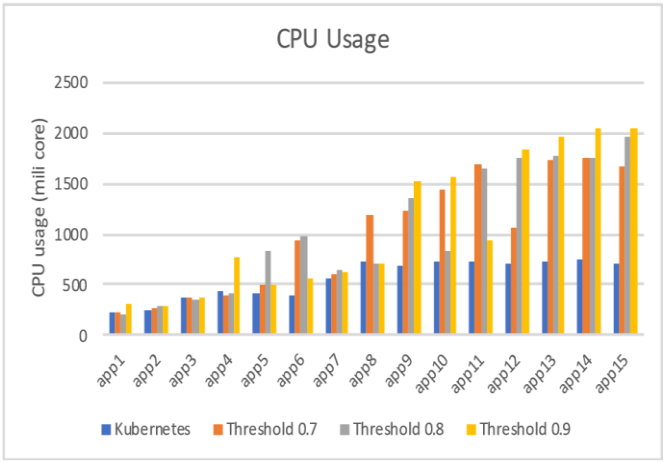


Figure 6. Comparison Of Applications Success Rate For Each Schedule

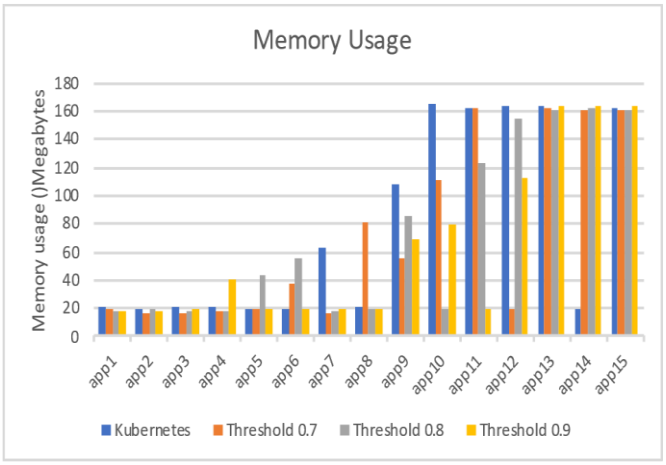
Figure 7 compares CPU and memory usage in each scheduling composition. The average resource usage after rescheduling is higher than that of Kubernetes scheduling. This

DOI: http://dx.doi.org/10.25139/inform.v8i2.6220

situation explains why the application availability while using Kubernetes scheduling in Figure 6 is lower than after rescheduling.



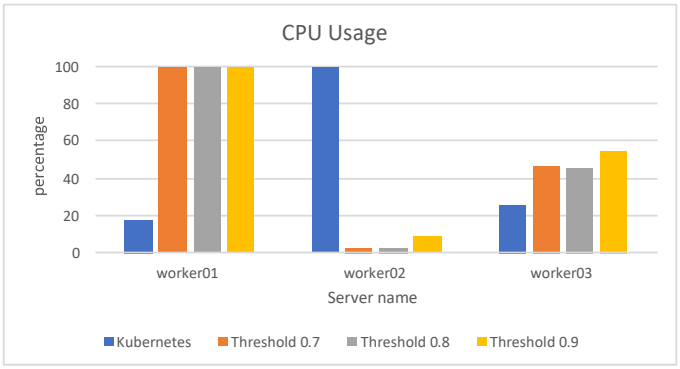
(a) Pods CPU usage



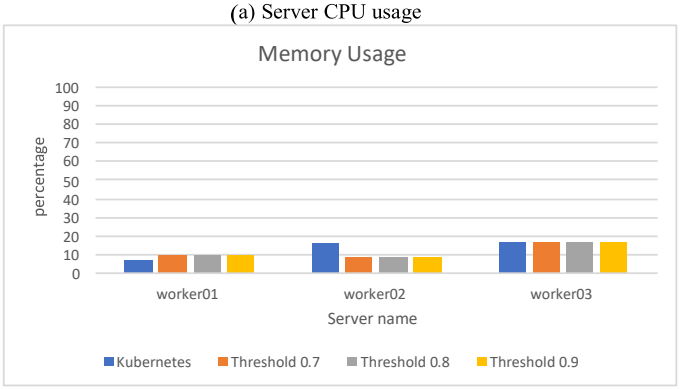
(b) Pods memory usage

Figure 7. Pods Resource Usage

This occurred because the server could not satisfy the pods' needs when the applications requested more resources. Whereas Figure 8 shows the average percentage of the total resource usage of each server. When Kubernetes scheduling was applied, worker02, which has the smallest CPU resource specification on the cluster, used a very high average CPU, close to 100%. Worker01 and worker03, which had more CPU resources than worker02, had many unused resources. This situation shows the misplacement during scheduling with Kubernetes due to a lack of information about the characteristics of the application in the container. The applications can achieve better availability when rescheduling is performed on the cluster. In this experiment, the memory usage in the application did not considerably affect the application's availability because a large amount of memory resources remained.



(a) Server CPU usage



(b) Server Memory Usage

Figure 8. Server Resource Usage

IV. CONCLUSION

Rescheduling is one of the strategies that can be used to increase the QoS of an application in a container cluster. Using a many-to-one stable matching algorithm, we can perform the scheduling process between containers and servers. The probability technique can be used to manage the frequency of the rescheduling process in a cluster, which is necessary to apply a stable matching algorithm to an environment with a dynamic cluster. The experiment results demonstrate that our rescheduling strategy improved the availability of the application. A high value of the happiness probability threshold corresponds to better QoS achieved. However, applying the trade-off concept to clusters, such as removing multiple containers from clusters when the server capacity exceeds the limit, is sometimes necessary to obtain a better QoS.

Several issues can be improved in this study, such as how to manage the trade-off effects. Our next study will enhance our findings by evaluating a trade-off effect to achieve a better QoS. Another issue that needs to resolve is related to energy saving in the cluster.

REFERENCES

[1] Z. Zhong and R. Buyya, "A cost-efficient container orchestration strategy in kubernetes-based cloud computing infrastructures with heterogeneous resources," ACM Transactions on Internet Technology (TOIT), vol.20, no.2, pp.1–24, 2020.

[2] A. Alimudin and Y. Ishida, "Service-based container deployment on kubernetes using stable marriage problem," Proceedings of the 2020 The

DOI: http://dx.doi.org/10.25139/inform.v8i2.6220

6th International Conference on Frontiers of Educational Technologies, pp.164–167, 2020.

[3] A. Alimudin and Y. Ishida, "Dynamic assignment based on a probabilistic matching: Application to server-container assignment," Procedia Computer Science, vol.176, pp.3863–3872, 2020.

[4] D. Gusfield and R.W. Irving, The stable marriage problem: structure and algorithms, MIT press, 1989.

[5] C. Kaewkasi and K. Chuenmuneewong, "Improvement of container scheduling for docker using ant colony optimization," 2017 9th international conference on knowledge and smart technology (KST), pp.254–259, IEEE, 2017.

[6] F. Chen, X. Zhou, and C. Shi, "The container deployment strategy based on stable matching," 2019 IEEE 4th International Conference on Cloud Computing and Big Data Analysis (ICCCBDA), pp.215– 221, IEEE, 2019.

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

[8] T. Menouer, "Kcss: Kubernetes container scheduling strategy," The Journal of Supercomputing, pp.1–27, 2020.

[9] M. Rodriguez and R. Buyya, "Container orchestration with costefficientautoscalingincloudcomputingenvironments," inHandbook of research on multimedia cyber security, pp.190–213, IGI Global, 2020.

[10] H. Xu and B. Li, "Egalitarian stable matching for vm migration in cloud computing," 2011 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), pp.631–636, IEEE, 2011.

[11] B. Di Martino, G. Cretella, and A. Esposito, "Advances in applications portability and services interoperability among multiple clouds," IEEE Cloud Computing, vol.2, no.2, pp.22–28, 2015.

[12] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," Proceedings of the Tenth European Conference on Computer Systems, pp.1–17, 2015.

[13] Kubernetes, "Kubernetes components." https://kubernetes.io/docs/concepts/overview/components. Accessed: 2021-02-21.

[14] Kubernetes, "Kubernetes scheduler." https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/. Accessed: 2021-0221.

[15] D. Gale and L.S. Shapley, "College admissions and the stability of marriage," The American Mathematical Monthly, vol.69, no.1, pp.9–15, 1962.

[16] Y. Ishida and K.I. Tanabe, "Network rewiring in self-repairing network: from node repair to link rewire," 17th International Symposium on Artificial Life and Robotics, 2012.

[17] Kubernetes, "Configure liveness, readiness and startup probes." https://kubernetes.io/docs/tasks/configure-pod-container/configureliveness-readiness-startup-probes/. Accessed: 2021-02-21.

[18] NRMP, "The match, national resident matching program." https: //www.nrmp.org/. Accessed: 2021-02-21.

[19] A. Kella and G. Belalem, "Vm live migration algorithm based on stable matching model to improve energy consumption and quality of service.," CLOSER, pp.118–128, 2014.

[20] X. Fan, W.D. Weber, and L.A. Barroso, "Power provisioning for a warehouse-sized computer," ACM SIGARCH computer architecture news, vol.35, no.2, pp.13–23, 2007.

[21] D. Kusic, J.O. Kephart, J.E. Hanson, N. Kandasamy, and G. Jiang, "Power and performance management of virtualized computing environments via lookahead control," Cluster computing, vol.12, no.1, pp.1–15, 2009.

[22] R. Sinha, N. Purohit, and H. Diwanji, "Energy efficient dynamic integration of thresholds for migration at cloud data centers," IJCA Special Issue on Communication and Networks, vol.1, pp.44–49, 2011.

[23] dockerhub, "Php arithmetic docker image." https://hub.docker.com/r/alioke/hpafix.

[24] Oracle, "Quality of service requirements." https://docs.oracle.com/cd/E19636-01/819-2326/gaxqg/index.html. Accessed: 2021-02-21. [25] Github, "Vegeta load test." https://github.com/tsenart/vegeta.

This is an open access article under the CC–BY-SA license.

DOI: http://dx.doi.org/10.25139/inform.v8i2.6220