EFFICIENT RESOURCE MANAGEMENT ON CONTAINER AS A SERVICE

by

Paul ChanHyung Park

BSc ChoongAng Univ. Korea, 1989

A thesis

presented to Ryerson University

in partial fulfilment of the

requirements for the degree of

Master of Applied Science

In the Program of

Computer Networks

Toronto, Ontario, Canada, 2019

AUTHOR'S DECLARATION FOR ELECTRONIC SUBMISSION OF A THESIS

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I authorize Ryerson University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

I understand that my thesis may be made electronically available to the public.

Efficient Resource Management on Container as a Service

Paul ChanHyung Park

Master of Applied Science in Computer Networks

Ryerson University, 2019

ABSTRACT

Docker has been widely adopted as a platform solution for microservice. As the popularity of microservice increases, the importance of fine-tuning the efficiency of resource management in the Docker platform also increases. While Docker's out-of-box resource management solution provides some generic management capability, more work is required to improve resource utilization and enforce Service Level Agreement (SLA) for critical services.

In this research, an efficient Docker resource management scheme, called Adaptive SLA Enforcement, is designed and implemented. For the sake of comparison, we also study and implement three simpler schemes: 1) Fixed Number of Containers, 2) Dynamic Resource Management without SLA Enforcement, 3) Strict SLA Enforcement. We found that the Adaptive SLA Enforcement scheme can deliver efficient resource management with SLA enforcement, thus successfully addressing the deficiencies of the other three schemes.

Acknowledgements

Thanks, God. I humbly accept that without His gracious help I wouldn't have been able to come here.

I graciously thank my supervisor and mentor, Dr. Ngok-Wah (Bobby) Ma, for his inspiration, technical and personal insight, and unwavering support throughout this work. His enthusiasm, vision, and experience have been the greatest sources of motivation to make possible the accomplishments herein.

I also thank Dr. Yu Xu and Mr. Roger Ehrlich for their support with computational environment provisioning.

Finally, I would like to thank my family for all the love and support they have provided throughout the years.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1. Introduction

1.1 Overview

The business trend of transitioning to Cloud-based server has been led by the primary motivation of cost cutting. The math to support businesses' decision is evidenced clearly in their budget and spending. Gartner's report on IT Budget of Healthcare Providers shows that an average of 73% of the total IT budget of companies are classified as an operating cost [1]. The expenses are used for technology and currency debt of both hardware and software in order to maintain the availability and reliability of the IT system's Quality of Service (QoS). The burden of this significant portion invested is compounded by the fact that it is often a repeated yearly expense for most companies.

Due to the significant spending on operating costs, the Total Cost of Ownership (TCO) of hardware and its software currency has been an ongoing area of concern and desired reductions for businesses. Adopting Opensource software and transitioning the IT infrastructure onto Cloud have been common options used by organizations to lower the TCO, in most cases freeing companies from vendor locked-in solutions.

In the report from Rackspace, comparisons of TCO of Cloud service providers demonstrate that keeping hardware on-premises results in a much higher TCO and high on-going spending for technology and currency debt [2]. As illustrated through real-world examples, the Rackspace report provides evidence that migrating IT infrastructures to Cloud service leads to significantly lower operating, capital, and indirect business costs.

Soon after a large number of businesses moved to Infrastructure-as-a-Service (IaaS) in order to minimize the operational impact of their IT environment, Docker Platform as a Service (DPaaS) emerged on top of IaaS. While IaaS can provide virtual machines with different sizes, its use of resource management still remains at a coarse-grained level. In comparison, platforms such as DPaaS or Docker on Cloud can provide more efficient and effective use of IT infrastructure by sharing more resources with other containers without impacting each other's QoS. The prevailing adoption of Docker opens another opportunity for efficient resource management with enforcement of the predefined Service Level Agreement (SLA) for each service.

1.2 Problem Statement

While Cloud became the platform of choice for businesses wishing to minimize the TCO of their IT infrastructure, Docker became the most popular technology as the means to realize microservices [1]. Because the Docker out-of-box resource management solutions such as Docker Swarm or Google Kubernetes are insufficient [4], there exists a gap between the level that out-of-box solutions can provide and the level of sophistication that businesses require. Although out-of-box solutions solve QoS issues on scalability, it does not have the SLA enforcement capability[5]. The lack of SLA enforcement can easily lead to the misallocation of resource—a situation which will often escalate to overall resource starvation [6].

Figures 1 and 2 below illustrate an example of resource starvation caused by the lack of SLA enforcement. In this scenario, the available resources are provided to whichever service demands the resources first, without the use of any constraints. If the volume of the service requests for Microservice1 increases, the service will demand more resources. Without SLA enforcement, the demand would be met as long as there are available resources [Figure 1].

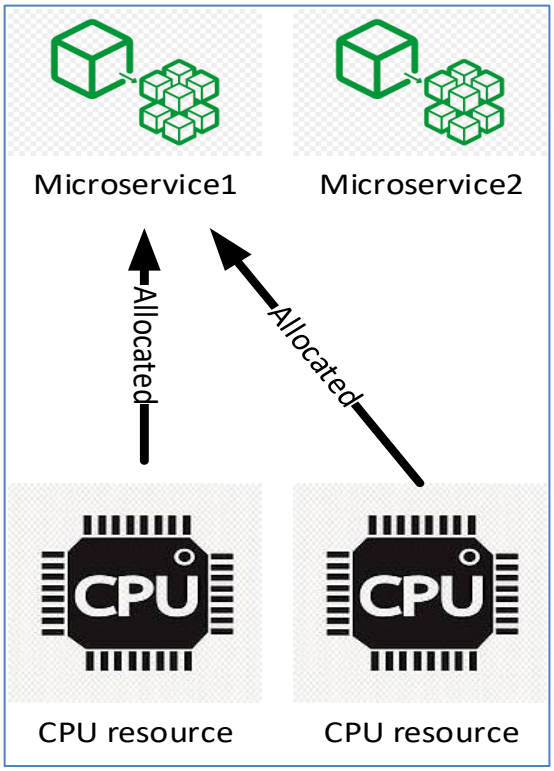

Figure 1 Resource allocation to Microservice 1

Consequently, Microservice1 can consume all available resources leaving none for Microservice2 which requests resources after all have been allocated to Microservice1 [Figure 2].

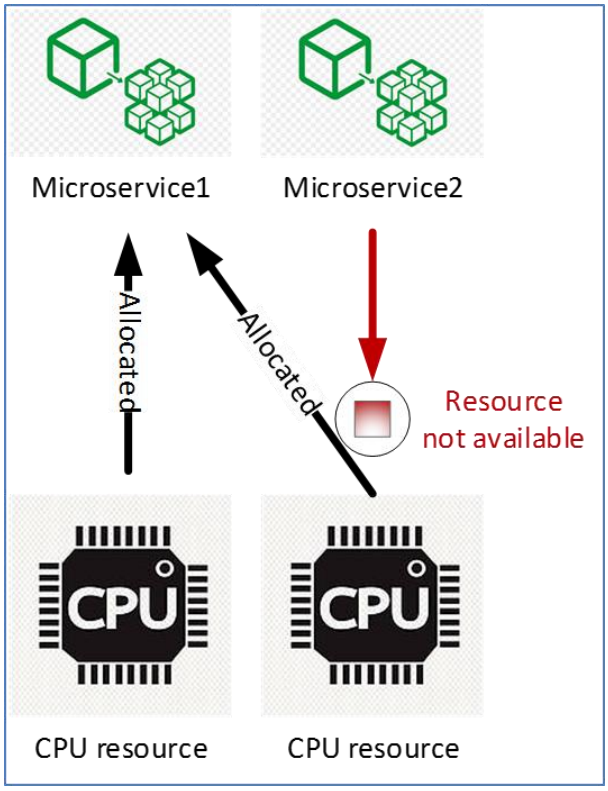

Figure 2 Resource starvation for Microservice2

Since the out-of-box resource management solution lacks the capability to enforce the SLA, unexpected and inundated traffic volume towards a particular service can deplete overall resource availability and may significantly affect the performance of other microservices running in the same Docker environment. Thus, an efficient resource management algorithm must include the enforcement of the SLAs.

One possible method to prevent resource starvation is to use a strict SLA enforcement policy, in which an upper resource allocation limit is imposed on each service. This policy guarantees that each service can consume up to but not exceed a pre-defined amount of resources [Figure 3]. The

main drawback of this approach is that it does not allow services to share their resources, leading to inefficiency in the utilization of resources.

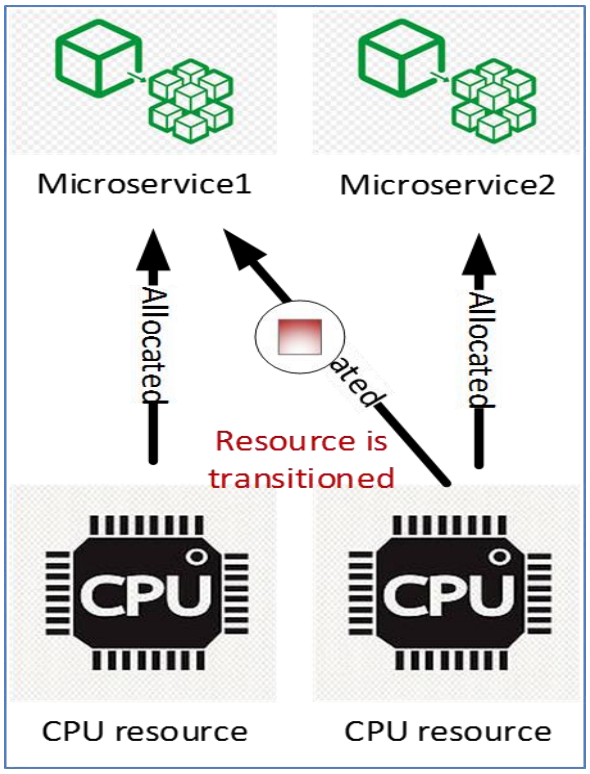

Figure 3: Strict SLA Enforcement Policy

Controlling the resource allocation is essential to meet certain levels of QoS, such as in the case of preventing the excessive resource consumption on exception or failure scenarios [6]. Accordingly, a lot of research has been conducted to address this issue. Resource optimization is one of the solutions often proposed to provide reliable and highly available service provisioning. The early detection of resource starvation is the key in avoiding service outages [7]. Resource starvation is a consequence of competition among services for a finite amount of available resources.

A more efficient approach to address the issue is to allow Resource Manager to redistribute resources among services dynamically. This thesis explores the mechanism of the approach as a remediation solution to alleviate the current inefficient approaches proposed by current research studies and solutions.

1.3 Technology Brief

1.3.1 Docker

Docker resolves many QoS issues in deploying microservices as it provides agility, portability, and control [19]. The Container Management Service (CMS) framework offers increased deployment density, scalability and resource efficiency [20].

Docker is based on container technology and due to the functional similarity, Docker often is compared with Virtual Machine (VM). VM has a full OS with its own memory management installed with the associated overhead of virtual device drivers and valuable resources are emulated for the guest OS and hypervisor, which makes it possible to run many instances of one or more operating systems in parallel on a single machine (or host). Every guest OS runs as an individual entity from the host system [49]. On the other hand, Docker containers are executed with the Docker daemon rather than the hypervisor. Containers are therefore smaller than Virtual Machines and enable faster start up with better performance, less isolation and greater compatibility possible due to sharing of the host's kernel.

The Docker container platform has the following desirable features:

a) Capturing high-level resource requirements: a representation which captures the high-level resource requirements of containerised applications along with CPU, memory and network ports.
b) Efficient containers co-location: techniques to group containers into multi-container (multi-task) units. This grouping serves as a unit of deployment on a container instance,

such that the aggregate resource requirement of a multi-container unit cannot exceed the total resources available to a container-instance.

c) Optimal deployment: a scheduling algorithm which solves the optimal deployment of sets of multi-container units on best fit container-instances across distributed Clouds, thus maximizing all available resources and speeding up the completion time.

1.3.2 Service Level Agreement (SLA)

The SLA is an agreement between the resource management and the microservice that defines the QoS that can be supported by the resource management solution [47]. In this research, SLA refers to the maximum amount of resources that can be provided by Resource Manager to the microservice. The resources are measured by the number of CPU cores.

1.4 Research Objectives and Contributions

This thesis proposes a new scheme that addresses the existing deficiency in out-of-box resource management of microservices deployed in the Docker environment. The proposed scheme introduces SLA enforcement to individual service with dynamic resource allocation. The contributions of the thesis can be summarized as follows:

- We propose a new scheme that improves resource management control in Docker container platform. In particular, an SLA enforcement feature is introduced which will limit resource consumption by an individual service.

- The proposed solution also aims to maximize resource utilization by reallocating resources dynamically among microservices.
- We design and implement a resource management framework for the Docker Container platform. Within this framework, various resource management schemes are studied and implemented.

1.5 Thesis outline

This thesis consists of 5 chapters with this chapter, Chapter 1, serving as the overview and an introduction. The rest of this thesis is organized as follows:

- Chapter 2 explores various resource management tactics and proposals found in the literature. The chapter ends with a summary of the proposed approach.
- Chapter 3 describes the resource management framework that is designed and implemented in this thesis. It also provides details of various resource management schemes. In particular, we propose a resource management scheme called Adaptive SLA Enforcement scheme which provides efficient resource allocation with SLA enforcement.
- Chapter 4 explores the implementation of the schemes introduced in Chapter 3. The performances of these schemes are then analysed and compared.
- Chapter 5 concludes the thesis and introduces potential future works.

Chapter 2. Background and Literature Survey

Many studies have previously researched guaranteed QoS for services. Although these were conducted in the frame of many different fields and aspects, they often fall into two general categories – Overall Resource Provisioning and Differentiated Resource Provisioning on runtime environment. We will explore these categories in-depth in the following chapters.

2.1 Overall Resource Provisioning

The first general category that we explore is Overall Resource Provisioning. The research goal of the studies in this general category is to deliver QoS in a runtime environment hosting multiple services. Although the subject areas have evolved over time from traditional virtual servers to Cloud, and again to Docker, the research goal has remained unchanged. Since the subject area of guaranteed QoS is runtime environment, all services in the environment are provided with the same level of QoS regardless of individual service's requirements. In this chapter, we will review the previously conducted research which aimed to deliver the guaranteed QoS for different runtime environments.

2.1.1 Traditional Objectives of Quality of Service

Traditionally SLA for throughput has been considered the QoS related to IT infrastructure. Architectural principles described by Len et al. [7] show its relevance to IT infrastructure. The authors highlight the importance of the software design principles, stating that it is a software architect's due diligence to consider the SLA when designing software architecture. The authors describe a number of quality attributes such as availability, modifiability, performance, and security, while various methods for fault detection, fault recovery, and fault prevention are reviewed. In addition, performance tactics such as resource demand, resource management, and resource arbitration are studied.

The importance of the QoS has been the primary objective in legacy environment. Leveraging the J2EE clustering feature, Lodi et al. [24] suggest Middleware architecture for enabling SLA-driven clustering of QoS-aware application servers. The authors propose three principles for the QoS:

1) Guaranteeing that the QoS requirements specified in SLAs are met;
2) Optimizing the resource utilization in addressing item 1, above; and
3) Maximizing the portability of the software architecture across a variety of specific J2EE implementations based on a load balancing service and monitoring service with desired SLA property.

2.1.2 Guaranteed QoS on Cloud as Infrastructure

The trend of delivering guaranteed QoS has been continued into the Cloud era. The way of linking the throughput SLA to IT infrastructure remains the same during the transition from the traditional computing environment to the Cloud computing environment. Stantchev [9] proposes a three-step approach to mapping the SLA and QoS requirements of business processes in a Cloud environment. The author describes the formalization of service capabilities and business process requirements. The author briefly explains the design-time non-functional properties (NFPs) and run-time NFPs. The 3 suggested steps to meet SLA are: 1) Formalization, 2) Negotiation, and 3) Enforcement. The primary goal of the research remains replicating technical services in available IT infrastructure to the Cloud environment.

As suggested by Jennings and Stadler [34], many Cloud Users and End Users do not have the expertise to properly exploit dynamic price fluctuations. This may open the field for cloud brokers who accept the risk associated with reserving dynamically priced resources in return for charging higher but stable prices. Moreover, Cloud Users may harness the nested virtualization capabilities of operating systems to directly re-sell computing capacity that they lease from a Cloud Provider. Modelling behaviour and devising pricing strategies for such a Cloud ecosystem are topics that should be investigated.

Giaretta et al. [10] propose the solution of increasing the scalability of services in Docker containers by improving Service Oriented Architecture (SOA) Orchestration using Jolie framework with Docker Kubernetes. Jolie framework is based on the Jolie programming language and is specialized for the orchestration of microservices [11]. Although the primary

subject of SLA is changed from Infrastructure-level QoS to Service-level QoS, the deficiency of the solution lies in the failure to implement the mechanism to prevent resource starvation.

Yamato et al. [18] propose an automatic performance verification technique that executes necessary performance tests automatically on provisioned user environments according to the collection of parameters for the required performance objective. The implementation, called Server architecture recommendation and Automatic verification Functions (SAF), recommends appropriate server architecture and verifies it based on the user's performance requirements. Based on economic factors calculated by the performance verification technique, SAF revises container configuration if required.

Freitas et al. [25] propose an approach for efficient resource utilization of Docker containers. The research shows that a key challenge for service providers is to manage cloud resources efficiently in order to increase profit while maintaining SLAs with customers. The authors define different SLA types by combining response time and reliability. Response time reflects the maximum amount of time for executing a request while reliability reflects the probability of success for a request. Customers can subscribe to three levels of QoS: high, medium and low. The subscribed QoS levels define different level of SLA based on different trade-offs between response time and reliability.

Another methodology of SLA enforcement is proposed by Anuradha and Sumathi [30]. This work presents a study of resource allocation strategies in Cloud computing. The strategies include resource requirements prediction algorithms and resource allocation algorithms. Part of

the strategies in the resource allocation algorithms involve the pre-emption of the currently low priority running task in order to satisfy the SLA of the higher priority tasks.

2.1.3 Service Quality Provisioning using Docker (Microservice)

As Docker container technology becomes the de facto standard for microservice, the subject of QoS guarantee has also been increasingly studied in the Docker environment.

Higgins et al. [12] propose a solution based on the novel cluster-watcher component combined with Software Defined Network (SDN) to improve the scalability of the Docker container's resource management. Although the solution solves the performance limitation through computational resources across Virtual Machines, it still lacks the resource allocation limitation control.

Guan et al. [13] propose an Application Oriented Docker Container (AODC) based on a resource allocation framework to support automatic scaling. The framework creates a pallet container and multiple execution containers for each application. The execution containers are scaled based on application's workload. This solution again does not enforce applications' SLA. The research presents the communication-efficient and scalable resource allocation algorithm to minimize the application deployment cost constrained by capacity and the service delay bound.

Barna et al. [14] propose an Autonomic Management System (AMS) which is a comprehensive resource management solution based on resource utilization. AMS limits the resource allocation based on predefined upper and lower thresholds. The system demonstrates workload distribution

across multiple Virtual Machines. It also demonstrates the resource limited between upper and lower thresholds. However, its deficiency is in its lack of consideration of SLA on resource management capability.

As Dziurzanski and Indrusiak [38] explain that initially Docker containers were executed and managed on a single machine, but soon other orchestration software that manage a number of nodes in a cluster emerged such as Docker Swarm or Google Kubernetes. These systems perform best with the most typical cloud usage patterns, such as Internet services' high availability or load balancing for microservices. It is assumed that Docker Swarm has no priority information regarding the workload or the containers' resource requirement. Since Docker orchestration tools do not consider the priorities of containers, they are not capable of prioritising the container execution without considering the value of the results to the end users' different levels of SLA.

Abdelzaher et al. [23] demonstrate the performance control of a Web server using classical feedback control theory to achieve overload protection, performance guarantees, and service differentiation in the presence of load unpredictability. In addition to achieving performance isolation and QoS guarantees, each virtual server supports request prioritization. Upon overload, lower priority requests are degraded first.

2.1.4 Increase Resource Utilization Efficiency

Sun et al. [16] present a new approach to enable rapid, optimized deployment of software onto a cloud environment by substantially reducing the number of benchmarks required. Based on a heuristic bin-packing algorithm, it guarantees meeting the QoS requirements for all the

applications while minimizing the total resource cost. The goal is to minimize the redundant capacity and packing inefficiency with utilizing different sizes of Virtual Machines which hold container implementation. The authors explore an algorithm called Bin Packing Scheduling, which is adopted in Docker Swarm.

Kaewkasi and Chuenmuneewong [21] propose the solution to improve the resource utilization using Ant Colony Optimization (ACO) to improve the scheduler's optimality. The research is leveraging an ACO-based algorithm to distribute application containers over Docker hosts. The algorithm balances workload leading to a better performance of applications with higher resource utilization by leveraging Docker Swarm.

A task selection and scheduling algorithm based on cooperative game theory is proposed by Kaur et al. [31]. The research is to improve the energy-efficient task selection and scheduling. The research describes the game theoretical models from broker to broker and container to container in order to minimize the overall energy utilization of servers.

Leveraging bidding and allocation framework, Ma et al. [33] provide a theoretical framework for resource management for SaaS providers so providers can efficiently control the service levels of their users, and to easily scale their applications under dynamic user arrivals/departures.

As proposed by Guerrero1 et al. [42], greedy algorithms based on the heuristic process is another approach to solve the issue of finding local optima for resource utilization. The main idea is that global optimum can be achieved by segmenting the optimization problem into smaller problems.

Also, as described in Chang et al. [43], Kubernetes provides a naive dynamic resource-provisioning mechanism which considers only CPU utilization, thus it is not effective. The research aims to develop a generic platform to facilitate dynamic resource-provisioning based on Kubernetes.

2.2 Differentiated Resource Provisioning

Second general category, Differentiated Resource Provisioning takes each service's SLA into consideration, while Overall Resource Provisioning provides equal level of QoS to the services on the runtime environment. Especially in regard to the economic factor, some services are more tolerant to lower level service quality in less expensive runtime environments. In this chapter, we will explore research whose goal is to deliver differentiated service quality for different level of QoS requirements.

2.2.1 Differentiated Level of Service Quality

While the target of guaranteed service quality is focused on overall service throughput, the economic factor cannot be ignored. Accordingly, a price-driven approach must be considered. The definition of importance can vary depending on the research approach. However, all approaches have one thing in common: deliver differentiated levels of service quality depending on each service's importance.

Sekhar et al. [15] call attention to the limitation of performance assurance due to unpredictable end-to-end latency. Based on the consideration of practical issues, the research proposes two schemes: performance monitoring of resources and algorithm for elastic and scalable scheduling. The performance statistics on each host is collected by monitors on each host and by Data collector on the Local Manager host. The collected data is analysed and the Docker resource provision will be adjusted based on cost estimation.

Abdelbaky et al. [22] propose a simple 2 step solution for guaranteed service quality. 1) Environment Description and Resource Filtering and 2) Resource Selection and Workload Allocation. The scheduler is workload-aware in that it selects the most appropriate resources from the current slice which will optimize a given QoS objective such as minimizing budget or data transfer.

2.2.2 Criticality based Resource Provisioning

In order to deliver the guaranteed service quality, monitoring and detecting any service quality violations is critical. Kyriazis [40] proposes an SLA enforcement method that aims at ensuring the quality parameters. The method in the research has service providers exploit monitoring mechanisms in order to obtain both infrastructure and application monitoring data, while adaptable approaches focus on adjusting the monitoring time intervals or the monitoring metrics based on the collected information during runtime. Evaluation tools are deployed to analyse the monitoring data and trigger corrective actions using SLA violation detection mechanisms.

Narayanan et al. [37] propose a resource allocation algorithm for SaaS providers who want to minimize infrastructure cost and SLA violations using Mapping Strategy: mapping of customer QoS requirements to resources (Virtual Machine [VM] type, small, medium, and large). The critical service is allocated in large size VM which larger and a greater number of containers can be deployed, while less critical service is allocated in small or medium size VM.

Also, Wu and Yang [39] propose a new CPU allocation approach called flexible deferrable server (FDS) scheduler to improve the performance of service. In particular, FDS first provides the available CPU capacity to Real Time Containers with higher criticality in order to ensure their timing constraints can be met. Then, the remaining CPU capacity is provided to Non Real Time (or lower criticality) containers dynamically at run-time so that their unpredictable on-line requirements can be met as much as possible.

2.2.3 Priority based Resource Provision

The wide adoption of microservices implemented as Docker containers on a Cloud computing environment requires a better resource management solution in order to avoid unexpected resource starvation. Individual services may also have different throughput SLA. Services with more rigid throughput SLA are expected to consume more resources than services with less rigid SLA. The finite resources in the Cloud computing environment may have to be shared between services unequally based on their SLA. In order to design a comprehensive resource management system, the deficiency of resource management must be resolved.

Implementing the allowable delay as an approach to differentiate service quality is another approach for priority-based resource provisioning. Young Choon Lee et al. [36] propose the approach with the development of a pricing model—using processor-sharing—for Cloud, the application of this pricing model to composite services with dependency consideration and the development of two sets of profit-driven scheduling algorithms. Characterization of allowable delay, a consumer application in this study, is associated with two types of allowable delay in its processing (i.e. application-wise allowable delay and service-wise allowable delay). For a given consumer application, there is a certain additional amount of time that the service provider can afford when processing the application; this application-wide allowable delay is possible due to the fact that the provider will gain some profit.

Tafsiri and Yousefi [41] propose Market-dependent Pricing Model, users pay a fixed price for each period of time and after the end of this period, the paying price is re-set for the next period according to the real-time market conditions such as supply, demand, and revenue of users and providers. In this pricing mechanism, the price is determined by one of the market-based mechanisms such as bargaining and auction.

2.2.4 Adaptive Resource Allocation

Leveraging Docker Swarm for resource management, Tihfon et al. [17] present a resource allocation algorithm that is triggered by resource demand, focusing on building products via Dockerfiles and leveraging the scheduling algorithm of Amazon ECS. Task definition allows for one or more containers to be specified. ECS has another entity called a “service,” which is useful for long running tasks, like web applications.

Makridis et al. [44] present robust dynamic resource allocation mechanisms to allocate application resources meeting Service Level Objectives (SLOs) agreed between Cloud providers and customers. The controllers are self-adaptive, with process noise variances and covariances calculated using previous measurements within a time window. In the allocation process, a bounded client mean response time (mRT) is maintained.

Zhang et al. [45] propose an SLA driven adaptive resource allocation for virtualized servers. When the available resource is insufficient for the demands, the adaptive resource allocation algorithm differentiates the resource allocation to guarantee resource requirements of higher priority applications. However, the algorithm does not force lower priority applications to release resources.

2.2.5 Throttling Resource Allocation

Under the SLA enforcement mechanism, each service is expected to consume resources based on its SLA. The SLA is interpreted as the average throughput measured in terms of Transactions per Second (TPS). The resource allocation may be limited based on the SLA in order to avoid overallocation. One of the techniques to limit the resource overallocation is ‘throttling’.

Wilder [26] defines throttling in his book as “selectively enabling or disabling features or functionality based on environmental signals. Throttling complements instance scaling” [26, p. 50]. Phillips [27] proposes a throttling solution from services competing for resources wherein a maximum number of requests from the services is imposed during the specific time period.

Another approach focusing on meeting the QoS of each service was proposed by Popovici and Wilkes [35]. The description and evaluation of a new family of profit-based scheduling and admission control algorithms for higher-level service providers 1) explicitly address the cost of renting resources; 2) handle variable-shaped jobs (ones that can be run on one or more processors) that scale imperfectly; and 3) explicitly address resource-availability uncertainty. The admission control algorithm is responsible for determining whether it will be profitable for the service provider to accept a job. Essentially, it attempts to determine if the net increase in profit is likely to be positive, at an appropriate level of risk. There are two parameters to consider: 1) the amount of uncertainty (how inaccurate are the resource-provider's estimates); and 2) how much risk a service provider is willing to take, which is expressed as a bound on the expected probability that an undesirable outcome may occur.

2.3 Summary of Proposed Approach

As the traffic volume to a service varies over time, the amount of resources allocated to the service to serve the traffic should be made adaptable to the volume changes.

The solutions examined in this section have one common deficiency: they do not provide a mechanism to limit resource allocation for individual service. As the computing environment is further moved to Cloud, the traditional way of managing throughput SLA requires a change of paradigm, from being associated with IT infrastructure to being associated with service implementation.

In this research, we introduce a resource management scheme with SLA enforcement. In addition to the SLA enforcement feature, the scheme can allocate resources dynamically according to the traffic volume. A resource management framework utilizing a feedback monitoring mechanism is also designed and implemented. The framework will provide a platform for further resource management research.

Chapter 3. Proposed Methodology and Resource Management Schemes

In this chapter, we will first introduce the Resource Management framework to be used to implement and study various resource management schemes. We then describe the four resource management schemes studied in this thesis. But first, we will cover several salient concepts associated with the resource management framework.

3.1 Architecture overview of Resource Management Framework

The architecture of Resource Management framework for this thesis [Figure 3] is comprised of three major components: Monitoring Component (Monitoring Server and Agent), Command Component (Control Command and Control Agent), and Resource Manager. Referring to [Figure 3], machine 1 runs Docker containers for a service that processes requests from service consumers. Monitoring Agent of the machine collects runtime statistics from Docker containers and sends it to Monitoring Server running on machine 2. Runtime statistics are collected by Monitoring Server and provided to Resource Manager. Resource utilization and allocation is determined by Resource Manager and, if needed, the decision is passed to Control Command. Control Command executes Resource Manager's decision by issuing a command to Control Agent. Figure 3 illustrates the overall process including Monitoring, Resource management, and Requirement management.

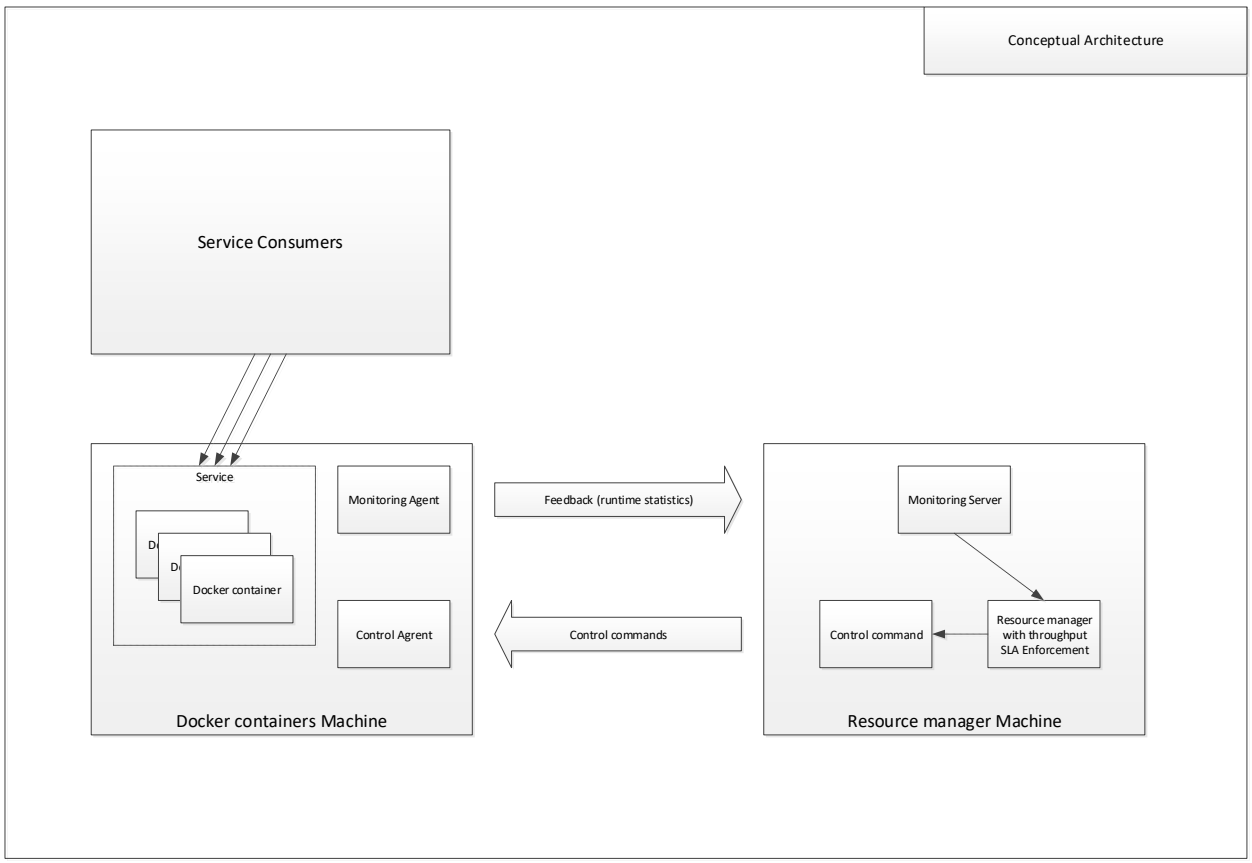

Figure 3 Conceptual solution diagram

3.1.1 Monitoring

The resource utilization of a service is determined based on the number of processed requests within a 60 seconds interval. Additionally, the CPU utilization percentage within a unit time is calculated based on the collected pile of runtime statistics of Docker containers. The information of processed requests and the runtime statistics of Docker containers are stored in the central repository through a 2-tiered process: Monitoring Agent and Monitoring Server [Figure 4].

Monitoring Agent runs on the same machine that Docker containers and web server are running and collects the processed request information and Docker containers' statistics. The web server

is configured to produce the information of processed requests to its log file, while the Docker containers' runtime statistics are available on demand.

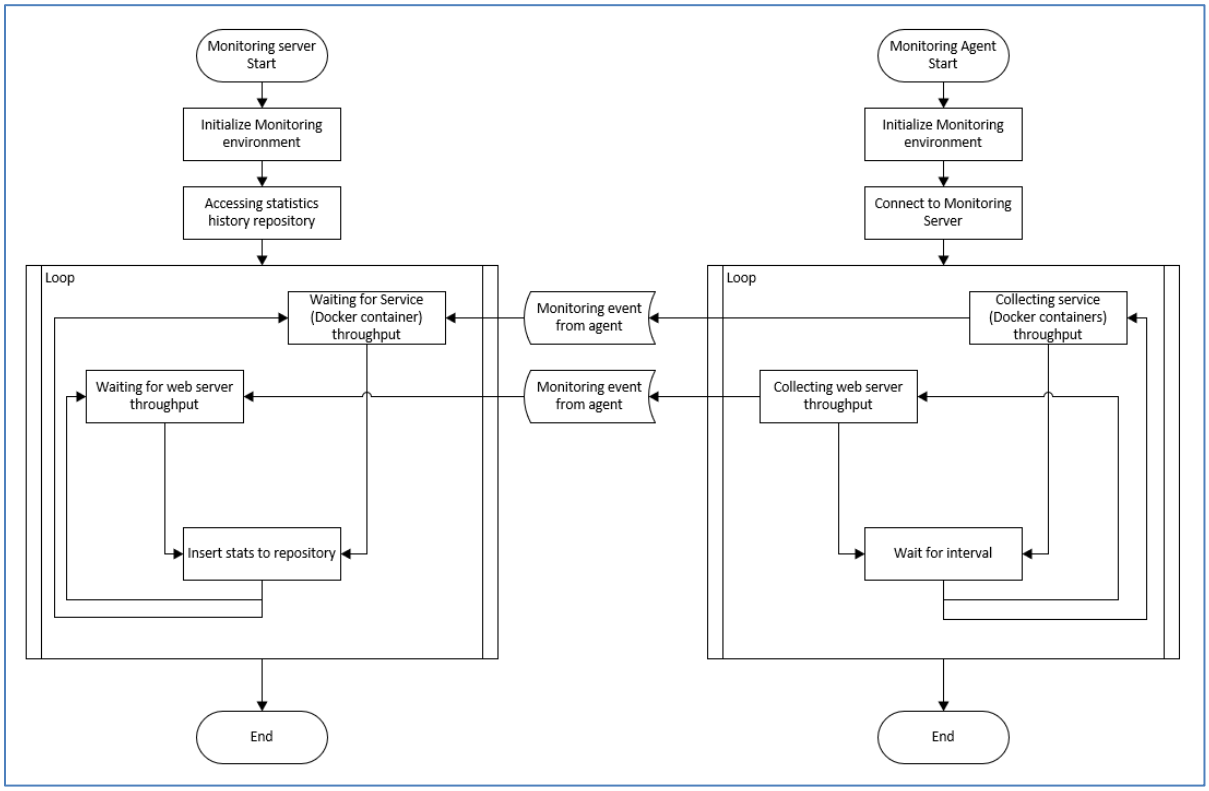

Figure 4 Monitoring process

3.1.2 Control

When the traffic volume of the service requires the change of service capacity, the change requests are, in turn, increasing or decreasing the number of Docker containers. The decision is made by Resource Manager based on various factors depending on the design of Resource Manager. Resource Manager triggers Controlling process [Figure 5] based on decision made through algorithm. Depending on the request type, Controlling Agent increases or decreases the number of Docker containers. The change of the number of Docker containers is applied to web server's port forwarding configuration by the agent, as well.

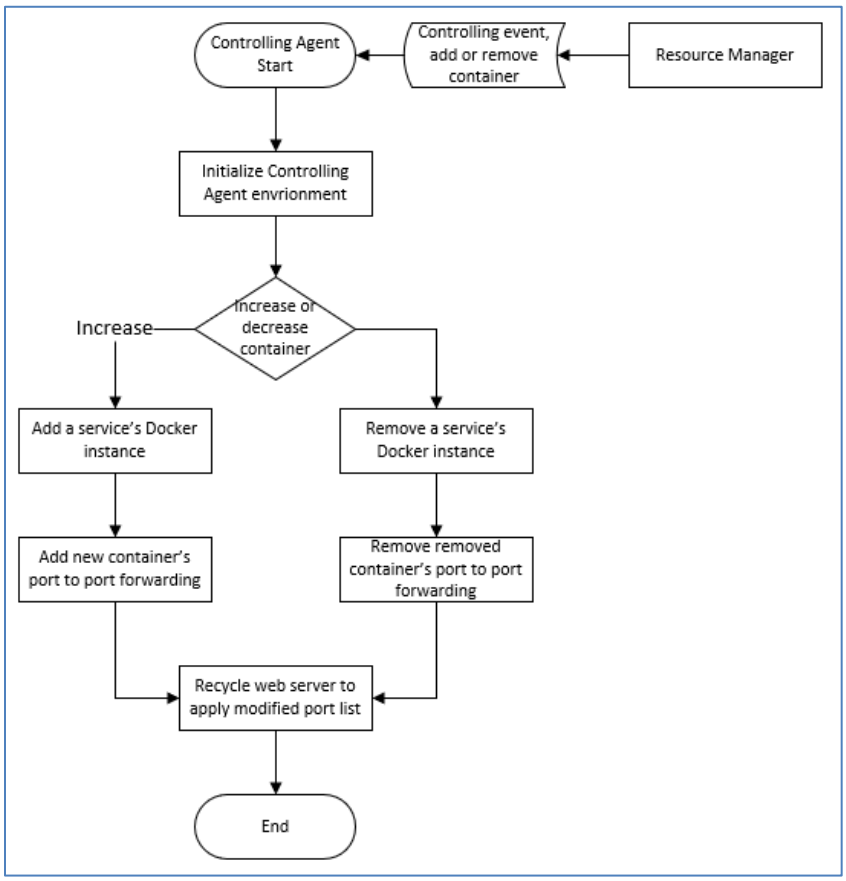

Figure 5 Controlling process - Increase / Decrease Docker containers

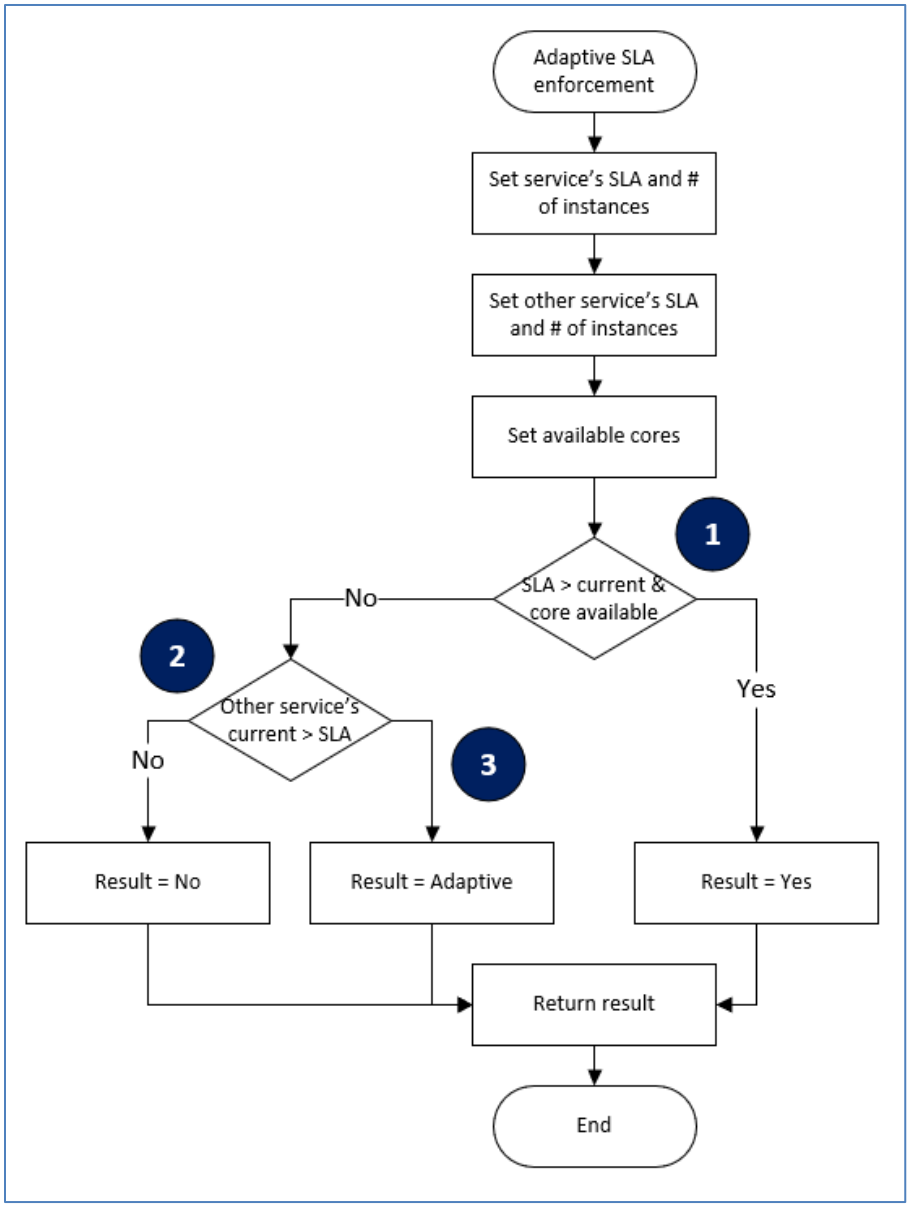

Figure 6 Resource Allocation / Deallocation decision based on SLA

3.2 Resource Manager

3.2.1 Introduction

Resource Manager is in charge of provisioning resources to microservices according to the demand. Two primary functions of Resource Manager in this thesis are:

1) To ensure that the allocation of resource adapts to the demand; and
2) To enforce SLAs of all the services being managed.

The following two subsections describe these functions.

3.2.2 Scalable resource management

The main function of the scalable resource management is to adjust resource allocation according to the demand. In this thesis, the CPU cores are the resources being managed by the resource management. In addition, since we will assign one CPU core to a container, the terms CPU core and container will be used interchangeably. When the traffic volume to the microservice increases significantly, Resource Manager may respond by increasing the number of Docker containers to the service. Conversely, when the traffic volume drops below certain threshold, Resource Manager may decide to reduce the number of Docker containers allocated to the service [Figure 7]. The scalable resource management approach effectively scales the allocated resources according to the traffic volume of the service.

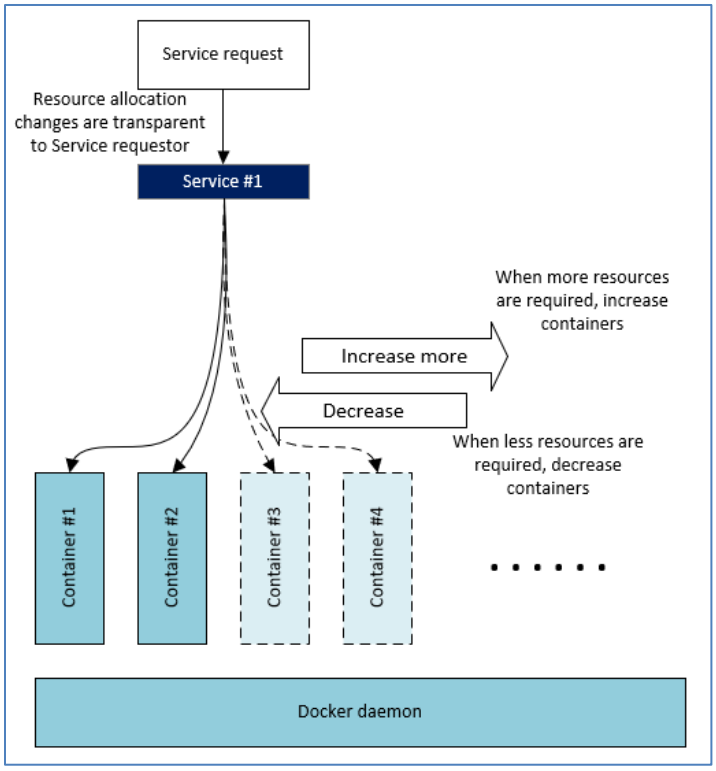

Figure 7 Transparent resource allocation / de-allocation

3.2.3 Enforcement of Service Level Agreement (SLA)

Leveraging the dynamic resource management capability, Resource Manager must ensure that resources are not over-allocated to a particular service. When a service requires more resources, the request must be subjected to a pre-set limit. Additional Docker containers are created only if the current allocated resources are below the upper allocation limit as defined in the SLA [Figure 8]. If the current resource allocation reaches the upper limit, no additional resources will be allocated.

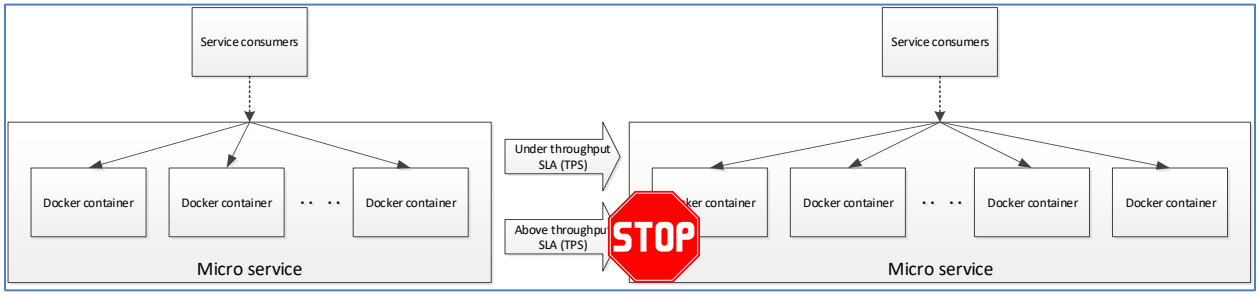

Figure 8 Enforce throughputs Service Level Agreement (SLA)

3.3 Workload Management through Port Forwarding

In order to meet SLA with efficient resource utilization, Resource Manager requires scalable resource management capability to scale the number of Docker containers up or down according to the traffic volume of the service. The fact that there can be multiple Docker containers handling requests for a given service is hidden from the consumer of that service. Service consumers identify the service by its single IP address and port number. In our implementation, we put a single web server at the front end to receive the incoming service requests. The server then distributes each request to one of the available containers running at the backend. In this

thesis, the port forwarding mechanism implemented in the web server is used to distribute the requests.

[Figure 9] illustrates port forwarding mechanism. When the request arrives at the web server at step 1 through its URL, ***http://<machine ip address>:<service port number>,*** the web server routes the request to the service directly. However, if the service is configured as port forwarding, the web server looks up the configuration and routes the request to one of entries in port forwarding configuration (at step 2 of Figure 9). Since multiple entries (containers) are configured in the service's port forwarding configuration, the web server spreads the workload among configured ports (at step 3 of Figure 9) through container URL, ***http://<local ip address>:<container n port number>***.

When the traffic volume increases and an additional container is added, (at step 4 of Figure 9) the newly added Docker container's port number is added to service's port forward configuration in the web server (at step 5 of Figure 9). Once the configuration is effective, workload is shared among configured Docker containers including the newly added Docker container. When traffic volume decreases and the CPU utilization of the active Docker containers is below the lower threshold, excessive Docker containers are removed and service's port forwarding configuration is updated.

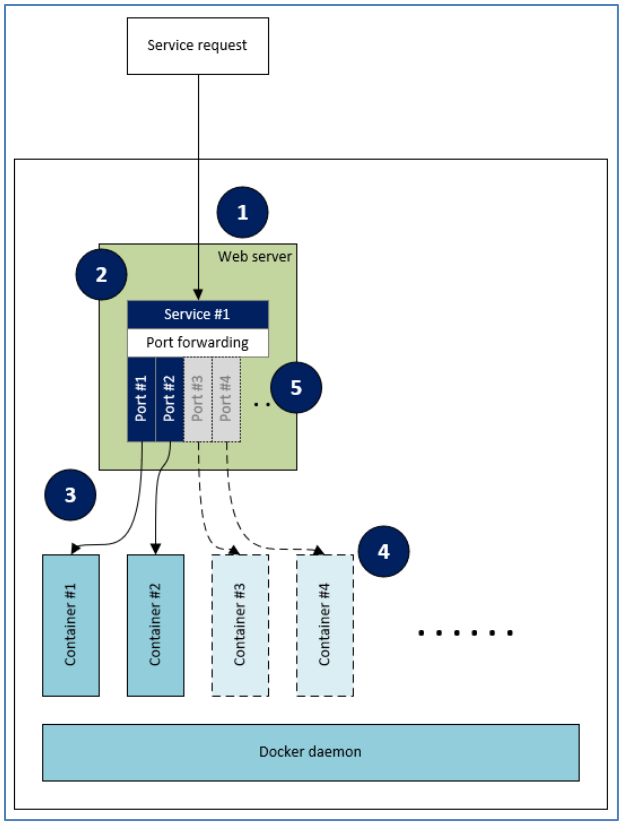

Figure 9 Workload Management through Port Forwarding

3.4 Resource management schemes

With the resource management platform in place, four resource management schemes will be studied. A basic scheme (Scheme 1) based on the native Docker Container platform is served as a benchmark. Three other resource management schemes with or without SLA enforcement (schemes 2, 3 and 4) are then proposed and studied. The four schemes studied in this thesis are briefly described in [Table 1].

Table 1 Different Schemes of Resource Management Control

Scheme No	Name	Description
Scheme 1	Fixed number of Docker containers	Microservice is implemented in 2 Docker containers. Resource Manager does not provide dynamic resource management capability.
Scheme 2	Dynamic Resource Management	Number of Docker containers for the service is maintained dynamically depending on traffic volume change. Resource Manager allocates resources according to the demand as long as the resources are available. Services are competing with each other for the available resources since their SLA is not honored.
Scheme 3	Strict SLA Enforcement	Resource Manager provides resources per service's demand up to predefined number of cores per SLA. Resource Manager declines the demand once the number of Docker containers allocated for a service reaches the upper limit.
Scheme 4	Adaptive SLA Enforcement	Resource management in scheme 4 is more flexible compared to Scheme 3. A service can have more containers than the upper limit specified in its SLA, as long as resources are available. When the resources become scarce, the excessive resources allocated to some services are released to be used by other services in order to meet the SLAs of all the services.

3.4.1 Scheme 1 Fixed number of Containers

Each service is allocated with a fixed number of Docker containers. Resource Manager does not provide dynamic resource management capability; thus, the change of traffic volume of the service does not change the number of Docker containers allocated to the service.

3.4.1.1 How it works

When the service is implemented, Resource Manager creates 2 Docker containers through the management process [Figure 10]. Resource Manager does not take action once the initial service implementation is complete.

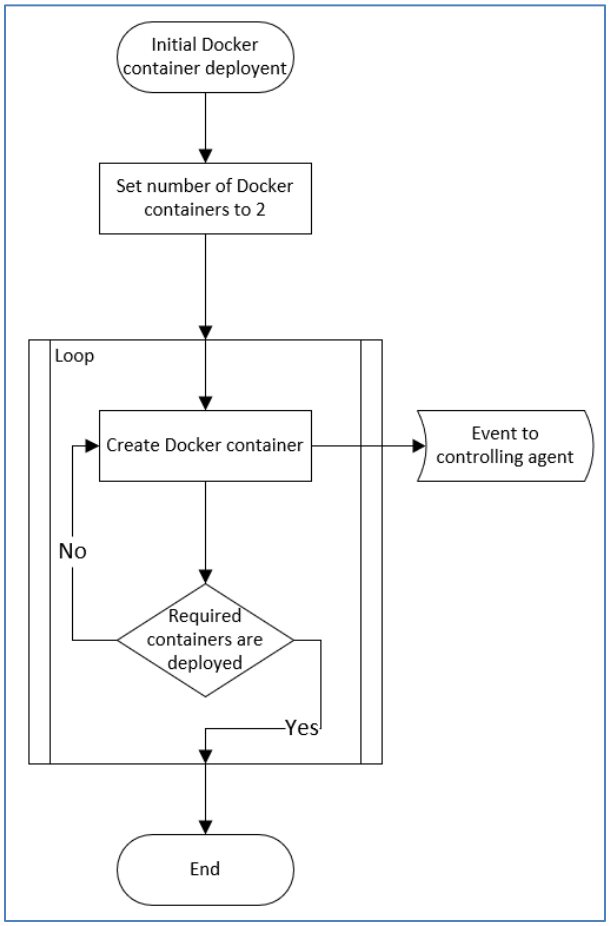

Figure 10 Scheme 1 Fixed number of containers, process flow

In the setup, the maximum number of available cores for the machine is set to 8 cores and 2 services are implemented. Each service has two Docker containers (Input statements of [Algorithm 1]). Each Docker container is allocated with a single core. Resource Manager enforces a fixed number of containers shown in line 5 of [Algorithm 1].

Algorithm 1: SCHEME 1 Each service has a fixed number, two, of containers, one core per container

Input: A finite set $Cores = \{c_1, c_2, \ldots, c_8\}$ of Cores
Input: A finite set $S_1 = \{s_1, s_2\}$ of services
Input: A finite set $s_1 = \{c_1, c_2\}$ of container
Input: A finite set $s_2 = \{c_3, c_4\}$ of container
Output: Two containers are created for each service
// This algorithm is to create fixed number of containers
// for each service. The fixed number is 2
1 $max_s \leftarrow 2$ // total 2 services
2 $max_c \leftarrow 2$ // 2 containers per each service
3 $n \leftarrow 1$ // Next NoOfUsedContainer
4 **for** $i \leftarrow 1$ ***to*** max_s **do**
    // For each service, assign 2 cores
    // one per each container
5   **for** $j \leftarrow 1$ ***to*** max_c **do**
        // Allocate one core to each container
6         $s_i \leftarrow c_n$
7         $n \leftarrow n + 1$
8 **return** n

Algorithm 1 Scheme 1 Fixed number of containers

3.4.1.2 Scheme 1 Summary

Among all the schemes studied in this thesis, Resource Manager of Scheme 1 is the most inflexible. Since there is no dynamic characteristic on resource management, resource planning can be simple. However, the design does not react to the traffic volume changes and it causes either resource waste when traffic volume is low or longer request time when traffic volume is too high.

3.4.2 Scheme 2 Dynamic Resource Management

Resource Manager provides dynamic resource management capability in response to the traffic volume changes. When the traffic volume increases and the service demands more resources, Resource Manager adds a Docker container through the Controlling process if resources are available. Conversely, when the traffic volume decreases to the level at which the CPU of the Docker containers is deemed to be underutilized, Resource Manager reduces the number of Docker containers through Controlling process so that the over-allocated resources are released. In this scheme, Resource Manager does not use the service SLA to impose a limit on the resources being allocated to the service.

3.4.2.1 How it works

When the service is implemented, Resource Manager creates 2 Docker containers through the Controlling process, illustrated in the left portion of the flow in [Figure 11], which is similar to Scheme 1. Once the service is running, Resource Manager monitors the throughput and CPU utilizations for the service, depicted in the right half of the flow in [Figure 11], in a fixed interval. In this research, a 10 second interval was set. When Resource Manager identifies that CPU utilization is above the upper threshold, it requests the addition of another Docker container to the service through the Controlling process. Conversely, Resource Manager may request the Controlling process to remove a Docker container when underutilization of CPU is observed.

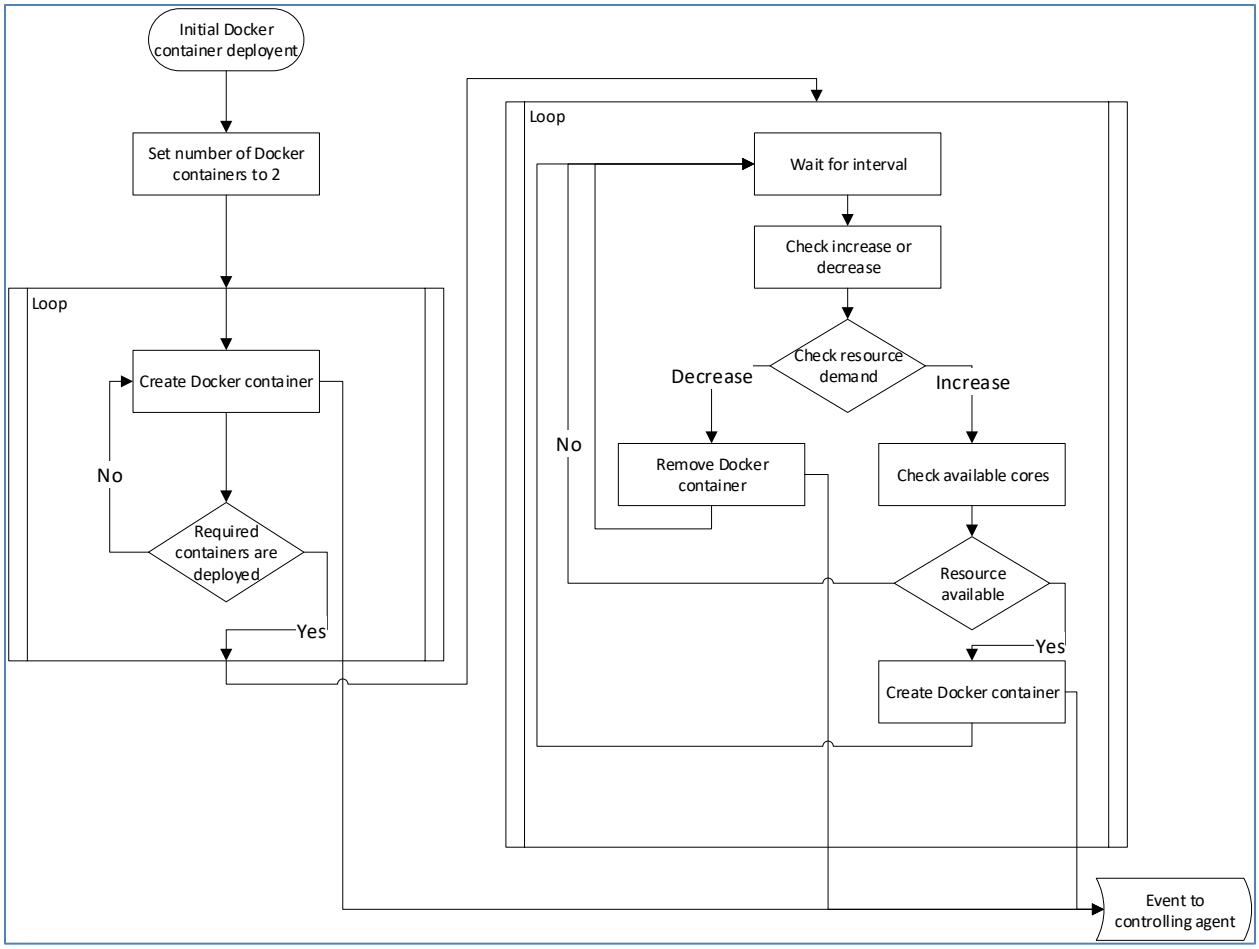

Figure 11 Scheme 2 Dynamic Resource Management process flow

In the setup, the maximum number of cores is set to 8 in line 4 of [Algorithm 2] and two services, s1 and s2, are competing over the available resources. Resource Manager monitors the service activities in a fixed 10 second interval at line 6. Prior to calculating the CPU utilization percentage at line 10, Resource Manager calculates the number of available cores by subtracting the number of allocated cores by the maximum number of cores at line 9.

The Docker containers' runtime statistics are collected by the Monitoring process, which is comprised of the Agent and Server components. The Monitoring Agent inquires Docker daemon for the containers' runtime statistics at each 10 second interval. Once the request is sent to Docker daemon, snapshots of runtime statistics are sent back to the Monitoring Agent, which then sends the information to Monitoring Server. The collected run statistics are saved on a central repository with collected time as an attribute.

The Resource Manager fetches the collected statistics for the past 60 seconds to compute the average. The average CPU utilization is calculated by summing up the CPU utilization of all the entries in the collected statistics and then dividing it by the total number of entries as given by [Equation 1].

Equation 1 Average CPU utilization percentile

$$\frac{\sum_{n=1}^{N} \mu_n}{N}$$

In equation 1, μ_n is the CPU utilization of entry n and N is the total number of entries.

Resource Manager determines whether the average CPU utilization is beyond the upper threshold[1] (in line 11 of [Algorithm 2]). If the upper threshold is crossed, it will check (line 12) if there is a CPU core available. If there is a CPU core available, Resource Manager requests (line 13) to add an additional Docker container to the service through the Controlling process.

[1] The upper threshold is set to 85% in our research.

Resource Manager does not request any additional Docker container if there is no available core (line 15).

Resource Manager removes a container from the service if the removal of a container does not push the utilization of the remaining container(s) over 50%. Let n_s and μ_s be the number of containers currently allocated to the service and their utilization, respectively.

On the assumption that the traffic offered to the service remains constant in the next 10-sec window, the removal of a container will cause the increase of the utilization of the remaining container(s) by $\frac{\mu_s}{n_s-1}$. Thus, the utilization of each remaining containers is $\mu_s + \frac{\mu_s}{n_s+1}$.

Based on this calculation, Resource Manager will remove a container from the service if the following condition applies.

$$\mu_s + \frac{\mu_s}{n_s - 1} < 50\%$$

Algorithm 2: SCHEME 2 Allocate cores to whichever requires more resource. First come first served without considering service's SLA

Input: A Service1 $s1 = \{c_1, c_2, \ldots, c_k\}$ of Cores
Input: A Service2 $s2 = \{c_1, c_2, \ldots, c_l\}$ of Cores
Input: A CPU utilization $P1 = \{p1_1, p1_2, \ldots, p1_n\}$ of Service1
Input: A CPU utilization $P2 = \{p2_1, p2_2, \ldots, p2_n\}$ of Service2
Output: Dynamic resource management, increase or decrease number of containers

```
// It's assumed that both services are implemented with 2
   containers using Algorithm1
// Dynamic Resource Management
// Wait for interval and repeat to check if service demands
// more resource until all available resources are used up
```

1 $interval \leftarrow 180$ `// interval for checking, 180 seconds`
2 $hThreshold \leftarrow 85$ `// high threshold CPU util, 85 percent`
3 $lThreshold \leftarrow 50$ `// low threshold CPU utili, 50 percent`
4 $maxCore \leftarrow 8$
5 **while** $True$ **do**
6 **while** *interval* **do**
   `// sleep for` *interval* `seconds`
7 $T \leftarrow currenttime$
8 $S \leftarrow$T - *interval*
9 $availableCore \leftarrow maxCore - k + l$
10 $avgCpuPercent \leftarrow (\sum_{t=S}^{T} p_t)/$noOfPercentEntries
11 **if** $avgCpuPercent > hThreshold$ **then**
12 **if** $availableCore > 0$ **then**
13 $result \leftarrow$Increase
14 **else**
15 $result \leftarrow$Do Nothing
16 **if** $avgCpuPercent < lThreshold$ **then**
17 $result \leftarrow$Decrease
18 **return** *result*

Algorithm 2 Scheme 2 Dynamic Resource Management algorithm

3.4.2.2 Scheme 2 Summary

Resource Manager of Scheme 2 provides dynamic resource management. However, it does not honour the service's SLA. The resource provisioning is on a 'first come, first served' basis. This may lead to resource starvation to some service due to unbalanced resource allocation. Because the SLA of services are not considered in resource provisioning, low priority services could use up all the available resources while critical services with higher priority do not have enough resources to handle their service requests.

3.4.3 Scheme 3 Strict SLA Enforcement

Scheme 3 incorporates strict SLA enforcement which limits the maximum number of Docker containers allocated based on SLA of the service. By using SLA enforcement, we can control the resource allocation of individual services, thus preventing resource starvation.

Resource Manager provides dynamic resource management capability in response to the traffic volume changes. The behaviour of resource management in response to the traffic volume changes is similar to Scheme 2, with the difference that the upper limit of number of cores is predefined based on the service's SLA.

3.4.3.1 How it works

In the setup, when the service is implemented, Resource Manager creates 2 Docker containers through the Controlling process as illustrated in the left half of the flow in [Figure 12] which is similar to the Scheme 1 process. Once the service is running, Resource Manager monitors the CPU utilizations for the service, as shown in the right half [Figure 12] in 10 second intervals. When Resource Manager needs to increase resources due to the high utilization of the currently allocated containers, it will check if the current resource allocation has reached the upper limit according to the SLA [Figure 13]. Only if the current resource allocation is below the limit will Resource Manager request to create an additional Docker container to the service through the Controlling process. In the case where the resource is deemed to be underutilized, Resource Manager will request to the Controlling process to remove a Docker container [Figure 12].

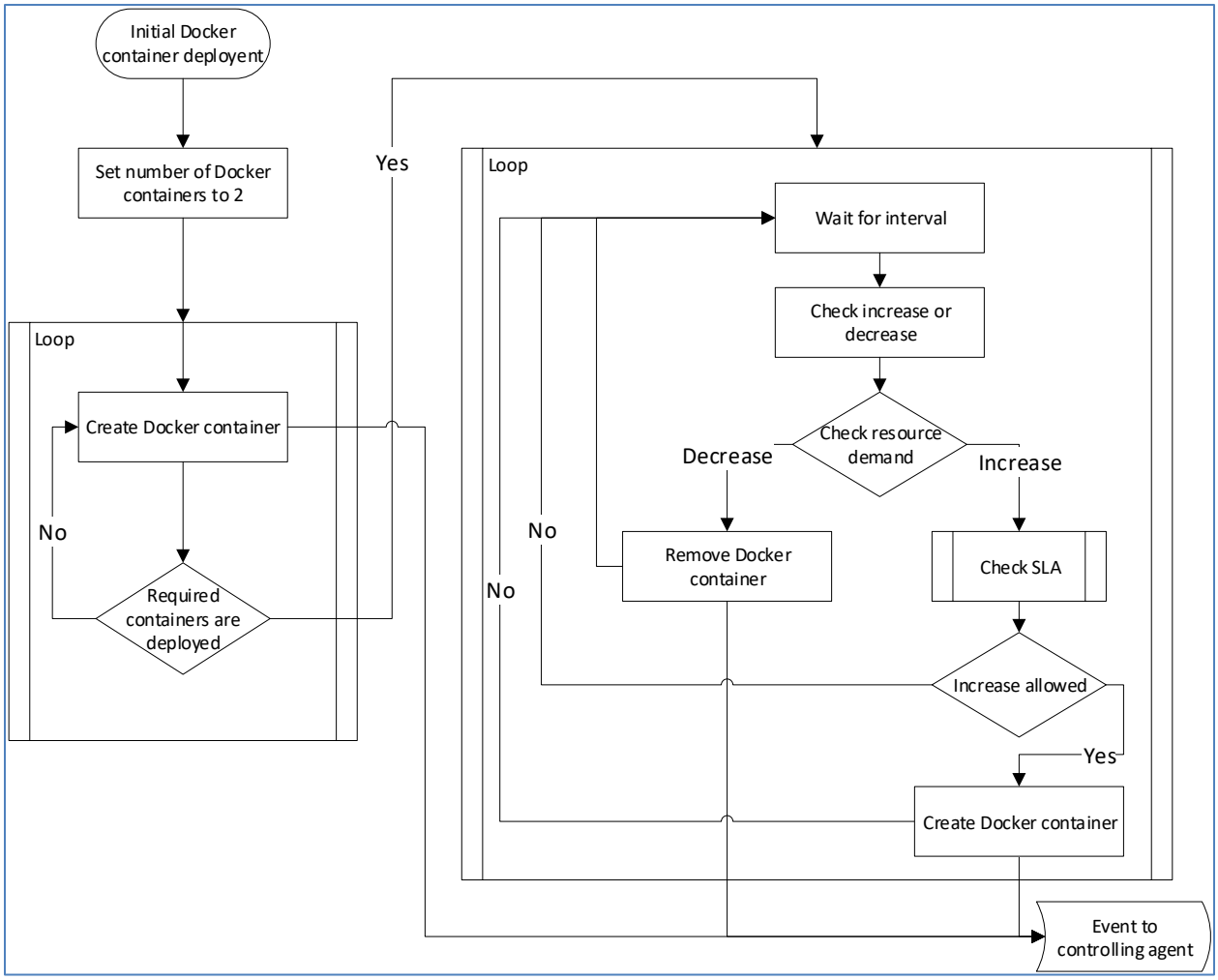

Figure 12 Scheme 3 Strict SLA Enforcement Resource Management process flow

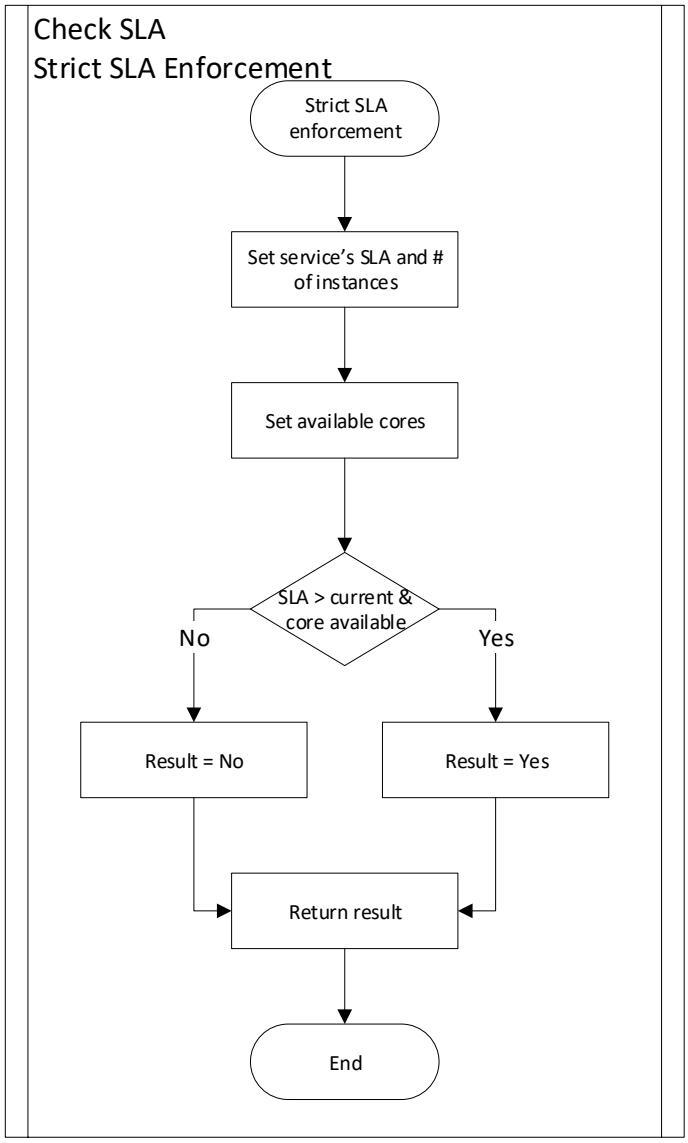

Figure 13 SLA evaluation (scheme 3)

In the setup, the maximum number of cores is set to 8 (line 4 of [Algorithm 3]) and two services, s1 and s2, are competing over the available cores. Resource Manager monitors the service activities using collected runtime statistics of the Docker containers in the same way as Scheme 2 [Algorithm 2].

Resource Manager maintains the predefined upper limit number of cores for each service (lines 5 and 6 in [Algorithm 3]). It obtains the current number of Docker containers for the service (line 13) for the SLA conformance check.

Resource Manager compares the average CPU utilization percentage and upper threshold to determine if any additional resources are required (line 14) and then to the lower threshold to determine if excessive resource is allocated to the service (line 19).

Additional SLA conformance is evaluated (lines 15 through 18) to check if the number of allocated cores reaches the upper limit as specified in the SLA.

Algorithm 3: SCHEME 3 Strict SLA Enforcement; Allocate cores up to allowed number of Core based on SLA

Input: A Service1 $s1 = \{c_1, c_2, \ldots, c_k\}$ of Cores
Input: A Service2 $s2 = \{c_1, c_2, \ldots, c_l\}$ of Cores
Input: A CPU utilization $P1 = \{p1_1, p1_2, \ldots, p1_n\}$ of Service1
Input: A CPU utilization $P2 = \{p2_1, p2_2, \ldots, p2_n\}$ of Service2
Output: Dynamic resource management, increase or decrease number of containers

```
// It's assumed that both services are implemented with 2
   containers using Algorithm1
// Dynamic Resource Management
// Wait for interval and repeat to check if service demands
// more resource until all available resources are used up
```

1 $interval \leftarrow 180$ // interval for checking, 180 seconds
2 $hThreshold \leftarrow 85$ // high threshold CPU util, 85 percent
3 $lThreshold \leftarrow 50$ // low threshold CPU utili, 50 percent
4 $maxCore \leftarrow 8$ // total number of Cores for Machine
5 $maxSlaCore1 \leftarrow 4$ // SLA for Service1, up to 4 Cores
6 $maxSlaCore2 \leftarrow 4$ // SLA for Service2, up to 4 Cores
7 **while** *True* **do**
8 **while** *interval* **do**
// sleep for *interval* seconds
9 $T \leftarrow currenttime$
10 $S \leftarrow$ T - *interval*
11 $availableCore \leftarrow maxCore - k + l$
12 $avgCpuPercent \leftarrow (\sum_{t=S}^{T} p_t)/$noOfPercentEntries
13 $noOfCoreForService \leftarrow$either k for $s1$ or l for $s2$
14 **if** $avgCpuPercent > hThreshold$ **then**
15 **if** $availableCore > 0$ *and* $noOfCoreForService < maxSlaCore$ **then**
16 $result \leftarrow$Increase
17 **else**
18 $result \leftarrow$Do Nothing
19 **if** $avgCpuPercent < lThreshold$ **then**
20 $result \leftarrow$Decrease
21 **return** *result*

Algorithm 3 Scheme 3 Strict SLA Enforcement Resource Management algorithm

3.4.3.2 Scheme 3 Summary

While Resource Manager of Scheme 3 provides a dynamic resource management in a similar manner to Scheme 2, Resource Manager of Scheme 3 predefines a maximum number of allowable Containers that can be assigned to the service. The SLA Conformance Check can deliver the required QoS to the services in a controlled manner.

The main drawback of Scheme 3 is that resources reserved to different services cannot be shared. For example, consider 2 services, Service 1 and Service 2, with 4 as the maximum allowable cores for each service. Suppose Service 1 currently experiences high traffic volume and it cannot handle the high volume even when maximum resources (4 cores) have been allocated to it. In this case, Service 1 needs additional resources. Meanwhile, Service 2 experiences low traffic volume, thus does not need all 4 cores. In this situation, it is desirable for Service 1 to borrow the unused resources of Service 2. However, the strict SLA conformance check in Scheme 3 does not provide this flexibility. In the next section, we introduce the last scheme of the thesis that allows services to share the resources efficiently.

3.4.4 Scheme 4 Adaptive SLA Enforcement

In scheme 4, Resource Manager can allocate additional resources to a service beyond the service's predefined SLA limit. In this scheme, a service can be in one of three states based on the amount of resource allocation. The service is in the under-allocated state if the resources allocated to the service is below its SLA limit. It is in the over-allocated state if the resource allocation exceeds its SLA limit. Finally, it is in the limiting state if the resource allocated is exactly equal to the SLA limit.

A service is allowed to acquire more resources independent of its SLA if unused resources are available. On the other hand, if the overall resources are not sufficient to meet the demands of all the services, Resource Manager will actively reallocate the resources in order to first meet the SLAs of all the services. The redistribution involves moving resources from services in the over-allocated state to services in the under-allocated state.

3.4.4.1 How it works

In this scheme, we assume that the overall resources meet the SLAs of all the services.
In order to provide dynamic resource management, Resource Manager uses the collected Docker containers' runtime statistics as in Schemes 2 and 3. Based on the average CPU utilization percentage of a service within each 10 second interval, Resource Manager takes one of the following three actions [Figure 14]:

1) Requesting to decrease the number of Docker containers of that service when resources are underutilized;

2) Conducting SLA Conformance check when additional resources are required for that service; or
3) Taking no action when the allocated resource is neither underutilized nor overutilized.

If action 2 is selected, Resource Manager performs a conformance check to determine if extra resources should be allocated. The conformance check leads to three possible actions depending on the state of the service and the resource availability [Figure 14]:

1) Increase the number of Docker containers if there are available resources.
2) Do nothing if there are no available resources and the service is in either over-allocated or limiting states.
3) If the service is in under-allocated state and there is no resources available, it can acquire the resources from the service that is in the over-allocated state.

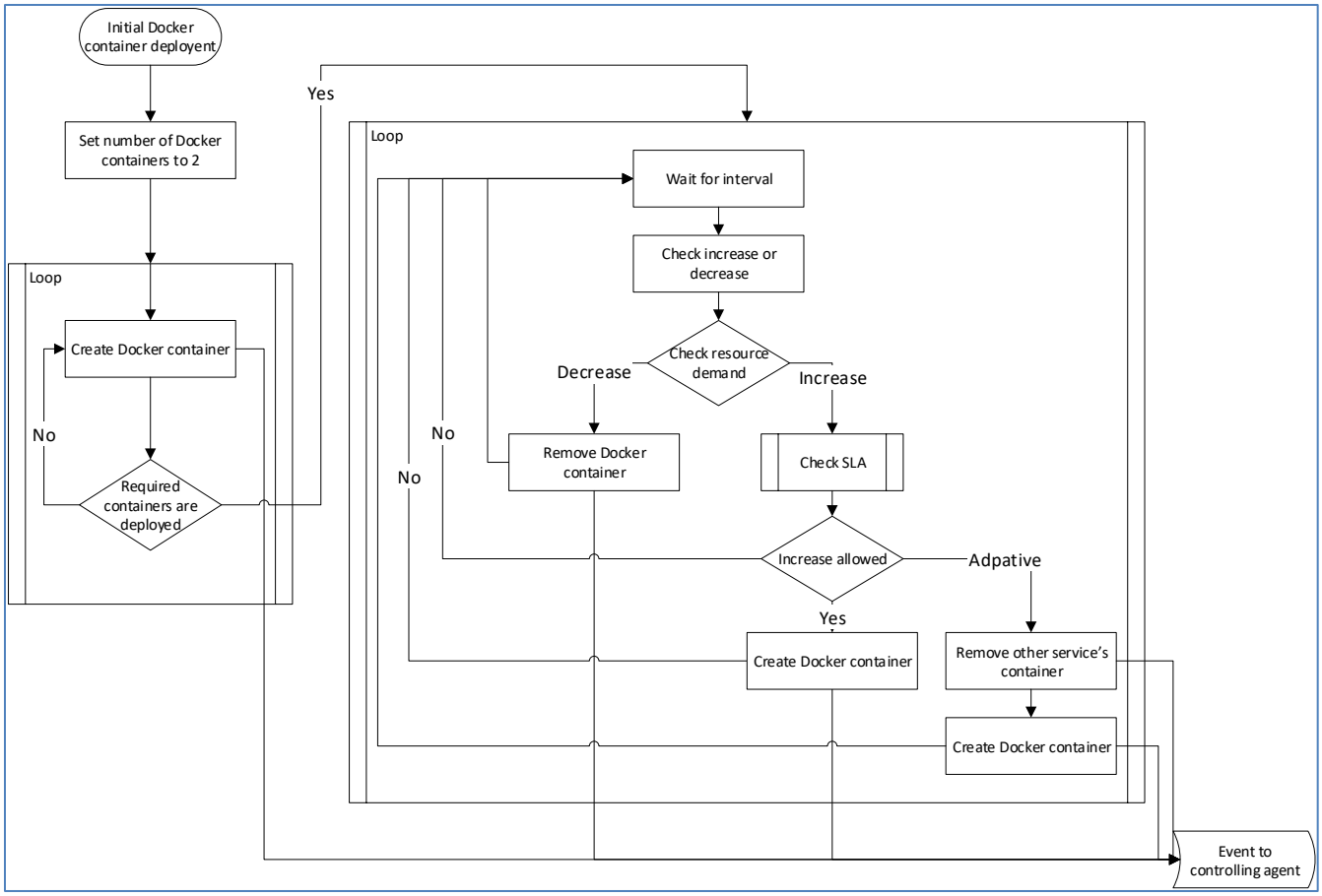

Figure 14 Scheme 4 Adaptive Resource Management process flow

Scheme 4 allows a service to have more resources than its SLA limit given that there are resources available. However, Resource Manager of Scheme 4 can also redistribute the resources from the over-allocated service to the under-allocated service in order to meet the SLA of the under-allocated service [Figure 15].

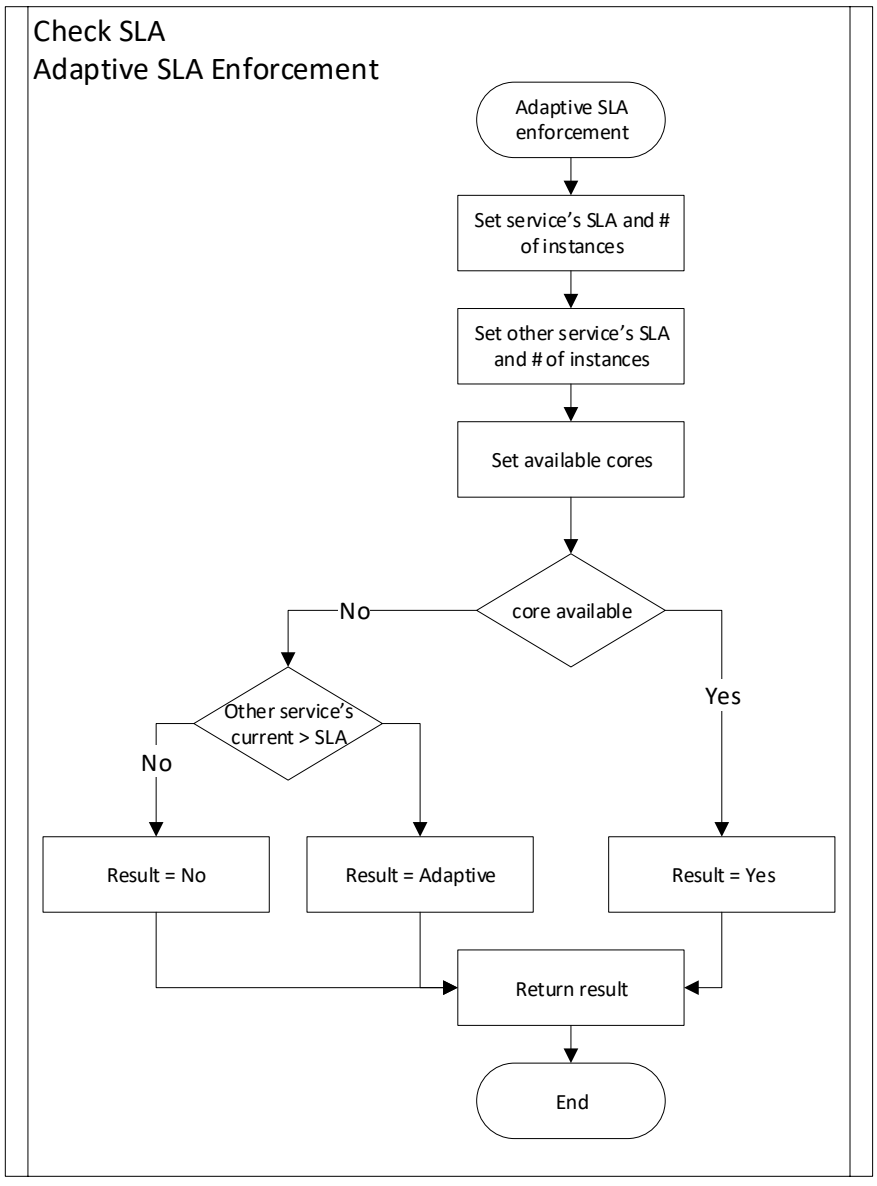

Figure 15 SLA Conformance Check (Scheme 4)

Scheme 4 leverages the same equation used by Scheme 2 [Equation 1] to calculate the resource utilization of all the services. When the service's traffic volume requires less resource than it is entitled, the other service can utilize more than it is entitled.

Algorithm 4: SCHEME 4 Adaptive SLA Enforcement; Allocate cores if resources are available (over allocation); if more resource is required from the other service, reduce the number of containers from over-allocated service to meet SLA

Input: A Service1 $s1 = \{c_1, c_2, \ldots, c_k\}$ of Cores
Input: A Service2 $s2 = \{c_1, c_2, \ldots, c_l\}$ of Cores
Input: A CPU utilization $P1 = \{p1_1, p1_2, \ldots, p1_n\}$ of Service1
Input: A CPU utilization $P2 = \{p2_1, p2_2, \ldots, p2_n\}$ of Service2
Output: Dynamic resource management, increase or decrease number of containers

```
// It's assumed that both services are implemented with 2
   containers using Algorithm1
// Dynamic Resource Management
// Wait for interval and repeat to check if service demands
// more resource until all available resources are used up
```

1 $interval \leftarrow 180$ `// interval for checking, 180 seconds`
2 $hThreshold \leftarrow 85$ `// high threshold CPU util, 85 percent`
3 $lThreshold \leftarrow 50$ `// low threshold CPU utili, 50 percent`
4 $maxCore \leftarrow 8$
5 **while** *True* **do**
6 **while** *interval* **do**
   `// sleep for` *interval* `seconds`
7 $T \leftarrow currenttime$
8 $S \leftarrow$T - *interval*
9 $availableCore \leftarrow maxCore - k + l$
10 $avgCpuPercent \leftarrow (\sum_{t=S}^{T} p_t)/$noOfPercentEntries
11 **if** $avgCpuPercent > hThreshold$ **then**
12 **if** $availableCore > 0$ **then**
13 *result* $\leftarrow$Increase
14 **if** *other service is over-allocated*
15 *and noOfCore for service*$<$*maxSlaCore* **then**
16 *result* $\leftarrow$Adaptive `// Decrease the other service and Increase`
17 **else**
18 *result* $\leftarrow$Do Nothing
19 **if** $avgCpuPercent < lThreshold$ **then**
20 *result* $\leftarrow$Decrease
21 **return** *result*

Algorithm 4 Scheme 4 Adaptive Resource Management algorithm

3.5.4.2 Scheme 4 Summary

Scheme 4 adopts the best features from Scheme 2 for dynamic resource management and from scheme 3 for SLA Enforcement. It manages resources in the most efficient way among all the schemes covered in this thesis.

Chapter 4. Implementation and Result Analysis

4.1 Implementation Overview

We implement the Resource management framework in two Machines [Figure 16]. The Monitor agent and Control agent are installed on the Docker container Machine, while Monitor server, Resource Manager, and Controller are installed on the Resource Manager Machine. Docker containers of the service are also installed on the Docker container Machine.

Web server is installed on the Docker Container Machine to handle service consumers' requests. Port forwarding is also configured to support multiple Docker containers per service. Service consumers are simulated using Apache Benchmark and implemented on a separate Machine.

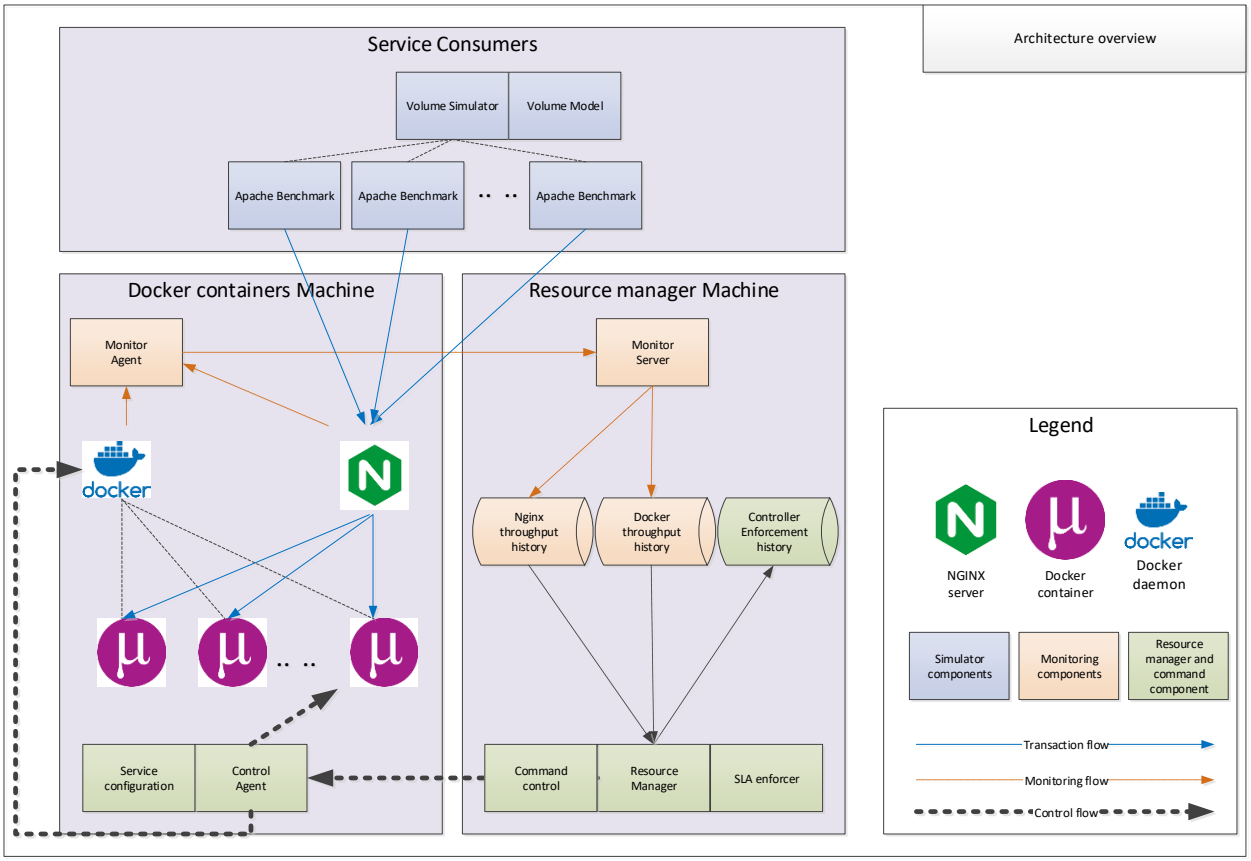

Figure 16 Solution architecture diagram

4.1.1 Traffic and Performance Parameters

In this section, we define the traffic and performance parameters to be used to study and compare the performances of different schemes.

4.1.1.1 Traffic Characteristics

An open source traffic generator, Apache Benchmark, is used to simulate web service traffic. The simulated traffic triggers two services using http protocol with two different service

endpoints[2]. In this research, multiple traffic simulator instances generate traffic volume, with each traffic simulator instance sending the service requests in a single thread.

Each instance can generate sufficient traffic volume to cause a Docker container utilize 50 percent of a CPU core. A single instance of web server receives the service requests of both services. The web server forwards each request to a particular service, then to one of the Docker containers allocated to that service. In this research, each service request and response traffic are considered as a transaction. It is assumed that a request is equivalent to one packet as well as a response.

4.1.1.3 Relation of CPU Cores and Containers

In this research, each Docker container is deployed with one core. The maximum utilization percentage of CPU for each container is 100 percent. Since each service could have multiple Docker containers, its combined CPU utilization can be up to 100 percent multiplied by the number of containers. For instance, when 4 containers are allocated to a service, its maximum combined CPU utilization is 400 percent.

[2] Two microservices are hosted by the same webserver with the same IP address. Each service is identified by different port numbers.

4.2 Simulation Setup and Performance Results

Two web services are simulated with the following parameters:

1. Traffic volume is adjusted every 1-minute interval
2. Traffic volumes of Services 1 and 2 are increased in the first 2 intervals
3. Between the 3rd and the 15th intervals, only Service 1's traffic volume is increased
4. Service 2's traffic volume is increased when Service 1's traffic volume reaches 850 requests per second.

Figure 17 illustrates the traffic patterns of Services 1 and 2.

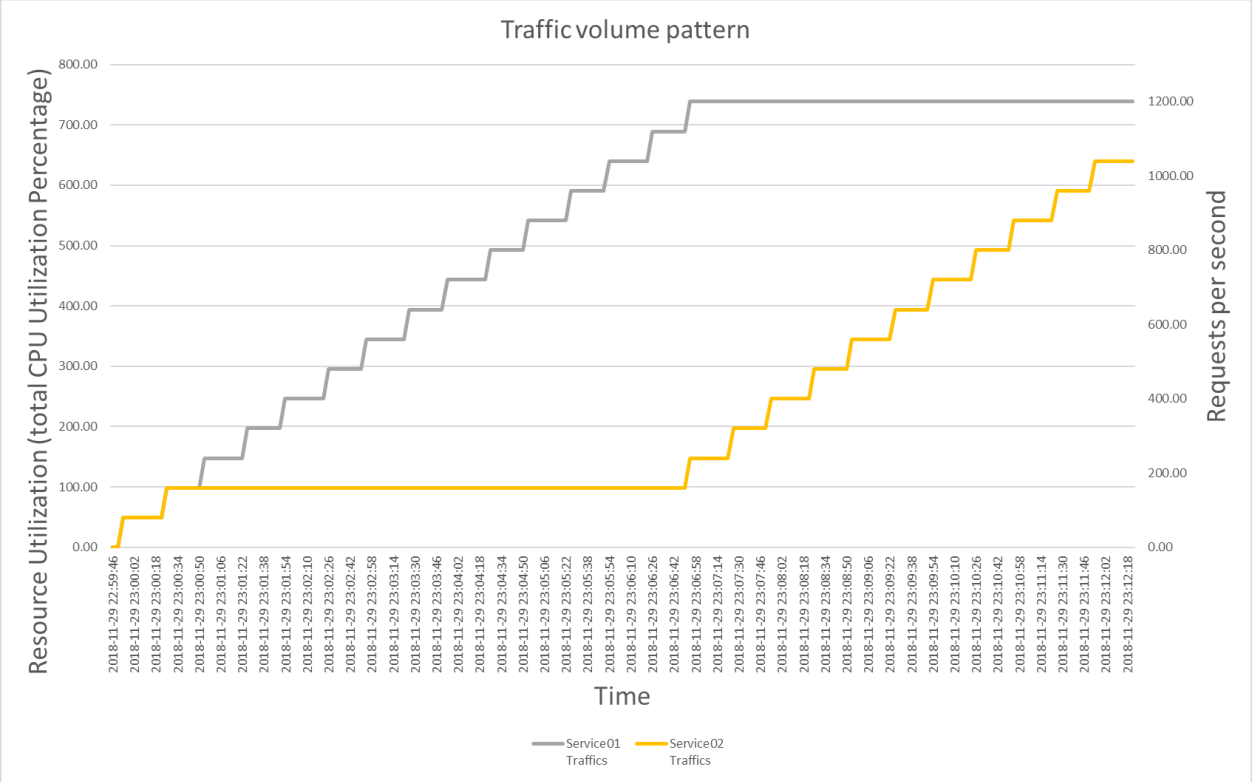

Figure 17 Traffic Volume Graph with Number of Traffic Simulator Instances

4.2.1 Scheme 1 Fixed number of Docker containers

Each service is implemented with two Docker containers. As seen in [Figure 18] below, Service 1's combined CPU utilization quickly reached its max, 200.00 percent (2 Cores times 100 percent each), while Service 2's combined CPU utilization percent is below the maximum due to the lesser amount of traffic volume. Service 2's combined CPU utilization quickly reaches its maximum when its traffic volume is increased.

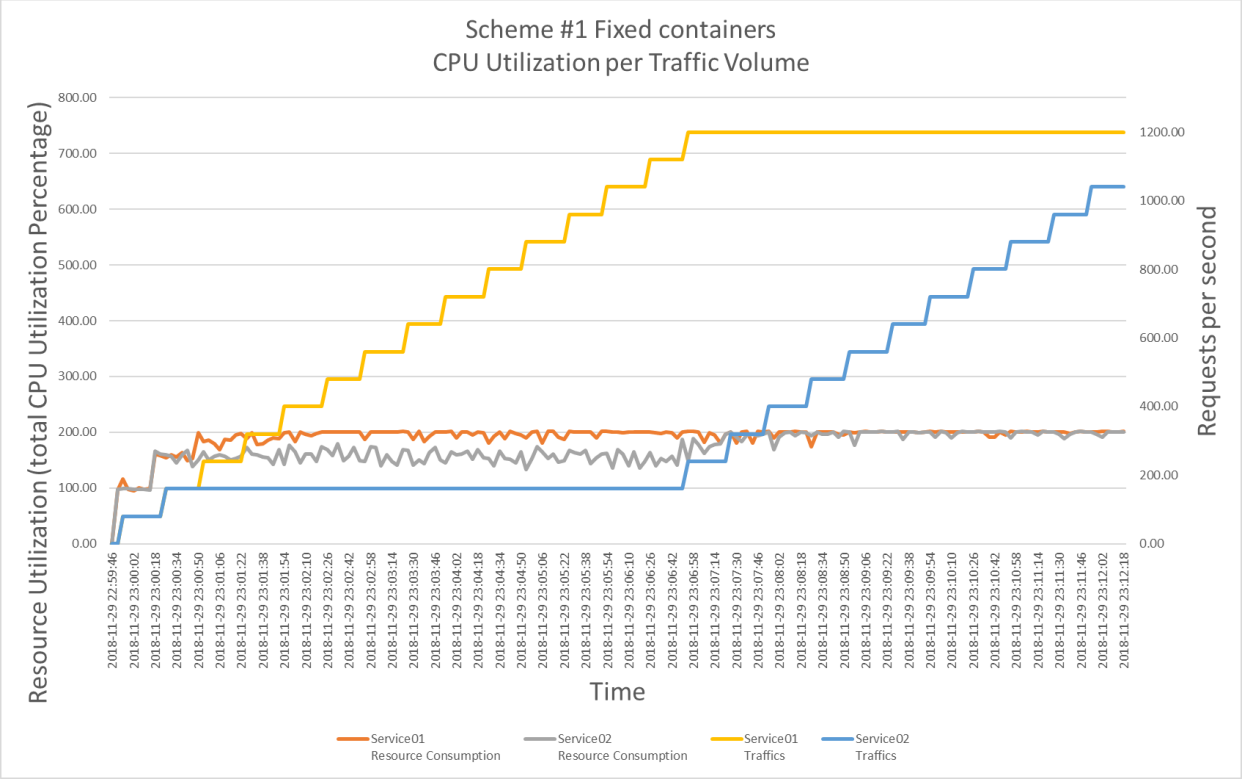

Figure 18 Fixed Resource Number of Docker containers test result

Although incoming traffic volumes for both services are increasing over time, due to the lack of a dynamic resource management capability, the resource allocation of both services is limited to the fixed number of cores. Due to this inflexibility the 4 remaining unused cores are wasted as they remain in an idle state.

Scheme 1 is the most inflexible resource management solution among the four schemes explored here. It leads to inefficiency in terms of resource utilization.

4.2.2 Scheme 2 Dynamic Resource Management

In scheme 2, additional resources can be allocated to the service that demands it. As shown in Figure 19, with the steep increase of traffic volume Service 1 quickly uses up all available six cores. Later, when Service 2's traffic volume starts to increase, there is no available core for Service 2 since Service 1 does not release any resource.

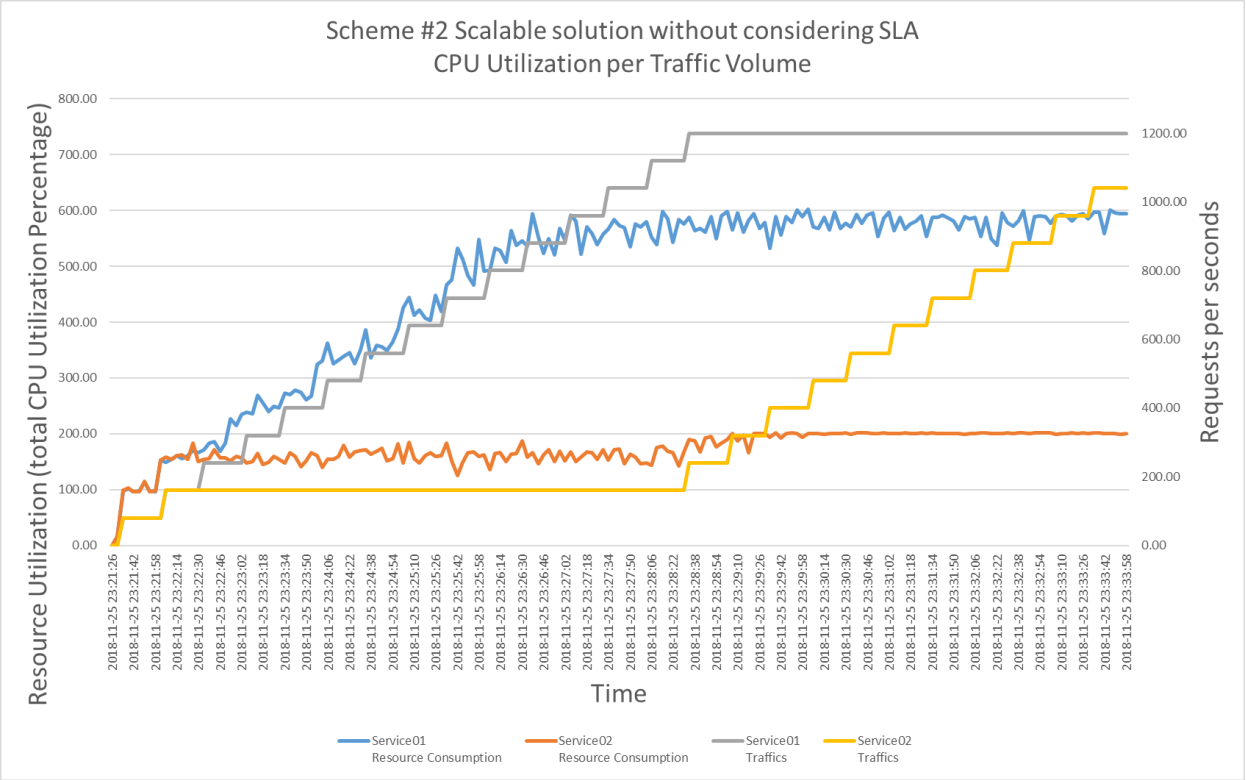

Figure 19 Scheme 2 Dynamic Resource Management test result

Compared to Scheme 1, this solution shows better utilization of resources using dynamic resource management. As the traffic volume of Service 1 increases, additional resources are allocated to the service until all the available resources are exhausted. Thus, no resources remain idle when they are needed unlike in Scheme 1. Scheme 2, however, does not impose limitations on how much resource can be allocated to a service. The consequence is that resources are not fairly distributed among services.

4.2.3 Scheme 3 Strict SLA Enforcement

In the setup for Scheme 3, the upper limit set for allocated resources in the SLA is 4 cores, or 400 percent of combined CPU utilization. As in the previous section, the resources allocated to Service 1 increase as the traffic volume increases. However, since a limit of maximum resource allocation is imposed, Service 1 can only acquire up to 4 cores [Figure 20]. This leaves two cores unused so when Service 2's traffic volume starts to increase; it can acquire the two unused cores. At the end, the resources allocated to both services are equal. This scheme prevents a service from using resources disproportionately. The main drawback of this scheme is that all resources of the machine cannot be fully utilized when needed. For example, 2 cores remain idle in the period between the time when Service 1 hits the limit and the time Service 2's traffic volume starts to increase. These two idle cores could have been used by Service 1 in this period to improve the overall resource utilization.

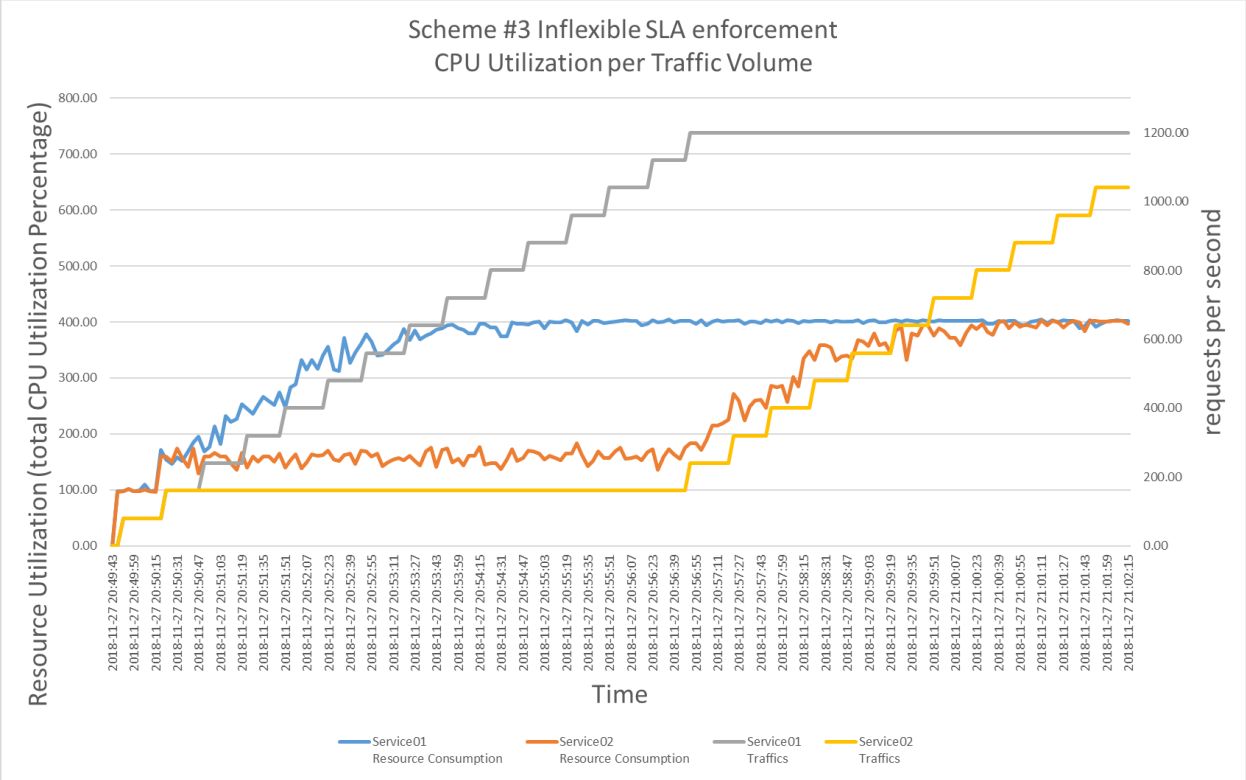

Figure 20 Scheme 3 Strict SLA Enforcement test result

4.2.4 Scheme 4 Adaptive SLA Enforcement

In the setup for Scheme 4, the maximum allocation resource limit for both services is 4 cores, similar to the setup in the Scheme 3 experiment. However, the difference is that each service can acquire resources beyond the limit as long as the resources are available. [Figure 21] shows that Service 1's resource utilization increases until it uses up all available resources.

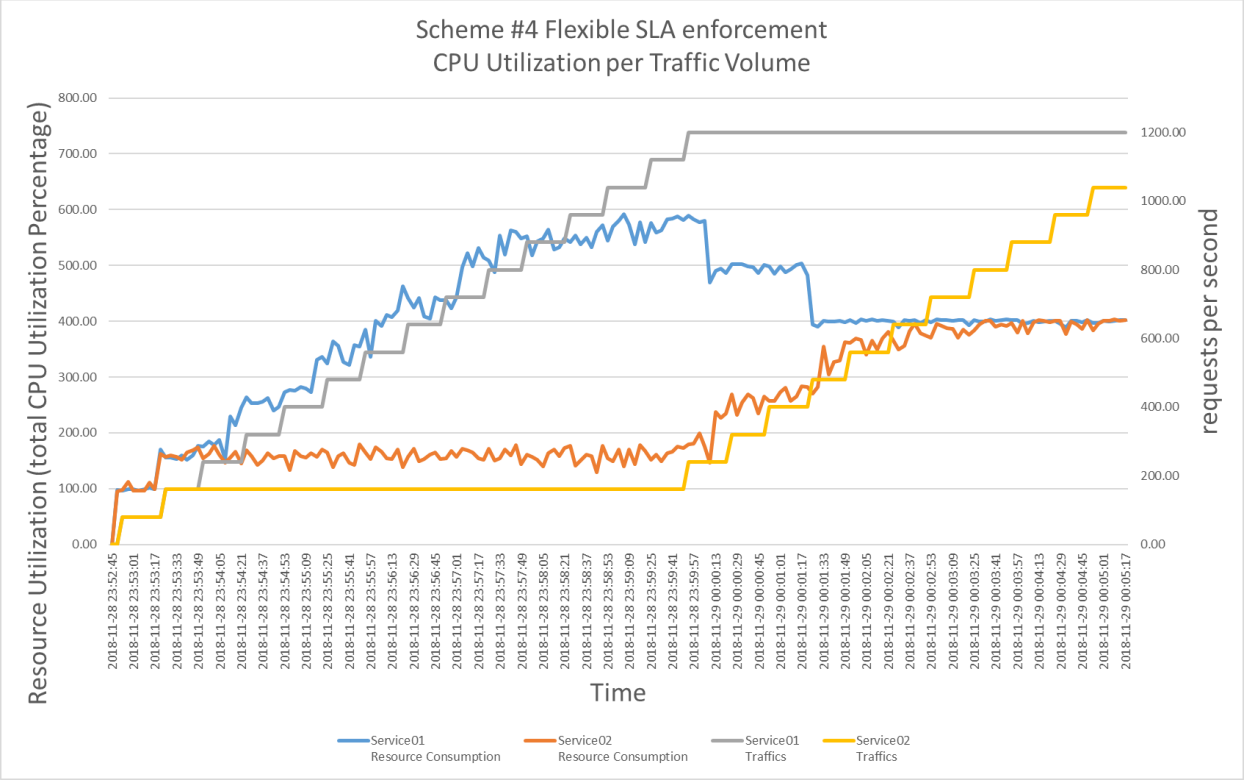

Figure 21 Adaptive SLA Enforcement test result

When all the available resources are allocated to Service 1, the combined CPU utilization is near 600 percent indicating that Service 1 has 6 cores allocated to it. As the traffic volume of Service 2 starts to increase and more resources are needed, Resource Manager redistributes the resource from Service 1, which is in the over-allocated state, to Service 2, which is in the under-allocated state. The shifts of resource allocation are demonstrated by the decrease of the combined CPU utilization of Service 1 and increase of the combined CPU utilization of Service 2. At the end, both services have the same combined CPU utilization of 400 percent, indicating both have reached the maximum limit of their resource allocation (4 cores).

Scheme 4 Resource Manager provides all the capabilities of Scheme 3, while improving the Scheme 3 solution with more efficient resource utilization.

4.3 Practical Considerations

The schemes proposed in this thesis have been implemented and tested in the practical environment. The response time of the resource allocation is found to be quite reasonable. Our study has found that it takes less than 2 seconds to activate a new container and about 11 seconds to deactivate a container. The long deactivated time is due to the default grace period of 10 seconds imposed by the Docker Container system [48]. Thus, the response time to traffic volume changes of Schemes 2 and 3 is quite fast. Since the monitoring window is 10 seconds long, consequently, we expect the response times to traffic volume changes by adding a container and removing a container will be within 12 and 21 seconds, respectively.

In Scheme 4, adding a new container to an under-allocated service may sometimes require removing container from an over-allocated service first. Thus, the response times to the increase of traffic volume of Scheme 4 could be larger than those of Schemes 2 and 3. Our study shows that the longest response time is around 23 seconds, which is still reasonable.

Even though our studies involve two services only, the schemes considered here should be scalable to the situation with a larger number of services. In particular, we expect the response times of these schemes should not increase significantly as we introduce more services. It is because parallel command execution is supported by Docker daemon, thus, the deactivating and activating container for different services can be executed in different threads. Furthermore, if the reduction of the response time is desirable, we can achieve that by either reducing the size of the monitoring window or the grace period of container deactivation, or both.

Chapter 5. Conclusion

The successful adoption of microservices in Docker containers introduces different challenges to resource utilization of the platform. Docker containers that implement microservices share the computing resources of the host machine. While each service may have different SLA requirements, there is no existing solution to enforce the service's SLA in the Docker container platform. As a lack of SLA enforcement on the Docker container may cause unexpected resource starvation, resource planning becomes quite uncertain. In order to cope with the deficiency, we study and compare four resource management schemes. The results of our study show that the proposed Adaptive SLA Enforcement scheme (Scheme 4) has the best performance among the four schemes. It provides dynamic and efficient resource allocation with necessary SLA enforcement.

In this thesis, we also designed and implemented an adaptive Resource Management framework. With this framework in place, we found that all the schemes studied here are readily to be implemented and tested. We expect this framework will be useful for further studies of other resource management schemes.

5.1 Future works

In this research, we designed and implement a Resource Manager framework implemented in a single machine runtime environment. The Docker environment in the real-world is quite complex with multiple machines running in the Cloud. As the next step, the Resource Manager framework should be explored in a more complex environment. Also, the Adaptive Resource

Management services should be improved to handle more than 2 services. This requires the algorithm to be able to select which allocated resources to release in a comprehensive manner.

References

[1] Jamie Guevara, “Gartner IT Budget: Enterprise Comparison Tool,” 2016. [Online]. Available: http://www.gartner.com/downloads/public/explore/metricsAndTools/ITBudget_Sample_2012.pdf: [Accessed Feb 2019].

[2] Gina Longoria, “The Business Value of Cloud-Enabled Managed Hosting,” April, 2017. [Online]. Available: https://go.rackspace.com/should-I-migrate-to-hosted-private-cloud.html: [Accessed Feb 2019]

[3] Hardin, T., “Digital Platform Trends 2018_ Containerization_files,” 2018. [Online]. Available: https://blog.g2crowd.com/blog/trends/digital-platforms/2018-dp/containerization/ : [Accessed Feb 2019]

[4] Docker inc. “Docker documentation, Limit a container's resources,” 2018. [Online]. Available: https://docs.docker.com/config/containers/resource_constraints/ : [Accessed Feb 2019]

[5] 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)*, Kanazawa, 2017, pp. 112-117.

[6] Gaurav, Suraj, and Suren Machiraju. “Hardening Azure Applications,” *Apress L. P.*, 2015. ProQuest Ebook Central, pp. 84

[7] Mohamed Elsabagh, Daniel Barbará, Dan Fleck, Angelos Stavrou, “On early detection of application-level resource exhaustion and starvation,” *Journal of Systems and Software*, Volume 137, 2018, Pages 430-447, ISSN 0164-1212, https://doi.org/10.1016/j.jss.2017.02.043.

[8] Len Bass, Paul Clements, and Rick Kazman, *Software Architecture in Practice 2nd Edition*, Performance Tactics p. 111-115, Addison Wesley, 2010

[9] Stantchev V., Schröpfer C. “Negotiating and Enforcing QoS and SLAs in Grid and Cloud Computing,” In: Abdennadher N., Petcu D. (eds) *Advances in Grid and Pervasive Computing*. GPC 2009. Lecture Notes in Computer Science, vol 5529. Springer, Berlin, Heidelberg

[10] Alberto Giaretta1, Nicola Dragoni1 and Manuel Mazzara 'Joining Jolie to Docker Orchestration of Microservices on a Containers-as-a-Service Layer,' Proceedings of *5th International Conference in Software Engineering for Defence Applications. SEDA 2016. Advances in Intelligent Systems and Computing*, vol 717. Springer, Cham

[11] The Jolie team, “The first language for micro-service,” 2016, [Online], Available: https://www.jolie-lang.org/index.html [Accessed Feb 2019].

[12] Joshu Higgins, Violeta Holmes, Colin Venters, “Autonomous Discovery and Management in Virtual Container Clusters,” *The Computer Journal*, February 2017Volume, 60(Issue2)Page, p.240To-252

[13] X. Guan, X. Wan, B. Choi, S. Song and J. Zhu, "Application Oriented Dynamic Resource Allocation for Data Centers Using Docker Containers," in *IEEE Communications Letters*, vol. 21, no. 3, pp. 504-507, March 2017.

[14] C. Barna, H. Khazaei, M. Fokaefs and M. Litoiu, "Delivering Elastic Containerized Cloud Applications to Enable DevOps," *2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, Buenos Aires, 2017, pp. 65-75.

[15] S. Shekhar and A. Gokhale, "Dynamic Resource Management Across Cloud-Edge Resources for Performance-Sensitive Applications," *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, Madrid, 2017, pp. 707-710.

[16] Sun, Y., White, J., Li, B. et al. "Automated QoS-oriented cloud resource optimization using containers," *Autom Softw Eng* (2017) 24: 101

[17] Tihfon, G.M., Park, S., Kim, J. et al. "An efficient multi-task PaaS cloud infrastructure based on docker and AWS ECS for application deployment," *Cluster Comput* (2016) 19: 1585.

[18] Yoji Yamato. 'Performance-aware server architecture recommendation and automatic performance verification technology on IaaS cloud,' *Service Oriented Computing and Applications*, 01/2017, Volume 11, Issue 2

[19] Docker inc, "Modern App Architecture for the Enterprise," 2017. [Online]. Available: https://goto.docker.com/modern-app-architecture.html [Accessed Feb 2019]

[20] Uchechukwu Awada and Adam Barker "Improving Resource Efficiency of Container-instance Clusters on Clouds," *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 05/2017.

[21] 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)*, Chonburi, 2017, pp. 254-259.

[22] M. Abdelbaky, J. Diaz-Montes and M. Parashar, "Towards Distributed Software-Defined Environments," *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, Madrid, 2017, pp. 703-706

[23] T. F. Abdelzaher, K. G. Shin and N. Bhatti, "Performance guarantees for Web server end-systems: a control-theoretical approach," in *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 1, pp. 80-96, Jan. 2002.

[24] G. Lodi, F. Panzieri, D. Rossi and E. Turrini, "SLA-Driven Clustering of QoS-Aware Application Servers," in IEEE Transactions on Software Engineering, vol. 33, no. 3, pp. 186-197, March 2007.

[25] Lage-Freitas, A., Parlavantzas, N., and Pazat, J. (2017) Cloud resource management driven by profit augmentation. *Concurrency Computat.: Pract. Exper*., 29: e3899, doi: 10.1002/cpe.3899.

[26] Bill Wilder, “Cloud Architecture Patterns”, 2012, Throttling (pp 50), O’Reilly Media, Inc

[27] Chris Phillips, “Go full throttle: The essentials of throttling in your application architecture”, 2017. [Online]. Available: https://developer.ibm.com/articles/mw-1705-phillips/ [Accessed Fb 2019]

[28] Tamas Kiss, Peter Kacsuk, Jozsef Kovacs, Botond Rakoczi, Akos Hajnal, Attila Farkas, Gregoire Gesmier, Gabor Terstyanszky, "MiCADO—Microservice-based Cloud Application-level Dynamic Orchestrator,” *Future Generation Computer Systems*, Volume 94, 2019, Pages 937-946,

[29] Xu, Hong, and Baochun Li. "A study of pricing for cloud resources." *ACM SIGMETRICS Performance Evaluation Review* 40.4 (2013):3. Web.

[30] V. P. Anuradha and D. Sumathi, "A survey on resource allocation strategies in cloud computing," *International Conference on Information Communication and Embedded Systems (ICICES2014)*, Chennai, 2014, pp. 1-7

[31] K. Kaur, T. Dhand, N. Kumar and S. Zeadally, "Container-as-a-Service at the Edge: Trade-off between Energy Efficiency and Service Availability at Fog Nano Data Centers," in *IEEE Wireless Communications*, vol. 24, no. 3, pp. 48-56, June 2017.

[32] Lei Li, Xue Gao, and Lianwen Jin "HCRCaaS: A Handwritten Character Recognition Container as a Service Based on QoS Guarantee Algorithm," Hindawi *Scientific Programming* Volume 2018, Article ID 6509275

[33] Ma, Richard T. B. et al. "On Resource Management for Cloud Users : A Generalized Kelly Mechanism Approach." (2010).

[34] Jennings, Brendan, and Rolf Stadler. "Resource Management in Clouds: Survey and Research Challenges." *Journal of Network and Systems Management* 23.3 (2015): 567-619. Web. 1 Apr. 2019

[35] F. I. Popovici and J. Wilkes, "Profitable services in an uncertain world," SC '05: Proceedings of *the 2005 ACM/IEEE Conference on Supercomputing*, Seattle, WA, USA, 2005, pp. 36-36.

[36] Y. C. Lee, C. Wang, A. Y. Zomaya and B. B. Zhou, "Profit-Driven Service Request Scheduling in Clouds," *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, Melbourne, VIC, 2010, pp. 15-24.

[37] L. Wu, S. K. Garg and R. Buyya, "SLA-Based Resource Allocation for Software as a Service Provider (SaaS) in Cloud Computing Environments," *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, Newport Beach, CA, 2011, pp. 195-204.

[38] P. Dziurzanski and L. S. Indrusiak, "Value-Based Allocation of Docker Containers," *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, Cambridge, 2018, pp. 358-362.

[39] J. Wu and T. Yang, "Dynamic CPU allocation for Docker containerized mixed-criticality real-time systems," *2018 IEEE International Conference on Applied System Invention (ICASI)*, Chiba, 2018, pp. 279-282.

[40] European Commission Directorate General Communications Networks, “Cloud Computing Service Level Agreements: Exploitation of Research Results,” Dimosthenis Kyriazis, July, 2013. [Online]. Available: https://ec.europa.eu/digital-single-market/en/news/cloud-computing-service-level-agreements-exploitation-research-results [Accessed: Feb. 2019]

[41] Seyedeh Aso Tafsiri, Saleh Yousefi. “Combinatorial double auction-based resource allocation mechanism in cloud computing market,” *The Journal of Systems and Software* 137 (2018) pp. 322–334

[42] Guerrero, C., Lera, I. & Juiz, C. “Resource optimization of container orchestration: a case study in multi-cloud microservices-based applications,” *The Journal of Supercomputing*, 07/2018, Volume 74, Issue 7

[43] C. Chang, S. Yang, E. Yeh, P. Lin and J. Jeng, "A Kubernetes-Based Monitoring Platform for Dynamic Cloud Resource Provisioning," *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*, Singapore, 2017, pp. 1-6.

[44] Evagoras Makridis, Kyriakos M. Deliparaschos, Evangelia Kalyvianaki, Argyrios C. Zolotas, and Themistoklis Charalambous. “Robust Dynamic CPU Resource Provisioning in Virtualized Servers,” https://arxiv.org/abs/1811.05533

[45] Wei ZHANG, Li RUAN, Mingfa ZHU, Limin XIAO, Jiajun LIU, Xiaolan TANG, Yiduo MEI, Ying SONG, Yuzhong SUN. “SLA_Driven Adaptive Resource Allocation for Virtualized Servers,” *IEICE Transactions on Information and Systems*, 2012, Volume E95.D, Issue 12, Pages 2833-2843

[46] F. P. Kelly. “Charging and rate control for elastic traffic,” *European Transactions on Telecommunications*, Volume 8, Issue 1 8:33–37, 1998.

[47] Andrew Hiles. *The Complete Guide to IT Service Level Agreements: Aligning IT Services to Business Needs 3rd Edition* pp.11 – 16, Rothstein Associates Inc, 2002 - Business & Economics

[48] Docker inc, “Command-Line interfaces, ‘docker stop’,” 2019, [Online], Available: https://docs.docker.com/engine/reference/commandline/stop/ [Accessed April 2019].

[49] DevOps.com, “Docker vs. VMs,” 2014, [Online], Available: https://devops.com/docker-vs-vms/ [Accessed April 2019].