IJCRT.ORG ISSN : 2320-2882

INTERNATIONAL JOURNAL OF CREATIVE RESEARCH THOUGHTS (IJCRT)

An International Open Access, Peer-reviewed, Refereed Journal

Survey on Container Systems and Their Efficient Orchestration Algorithms

[1]Pratham Jangra, [2]Anuttam Anand, [3]Dr. Amit Kumar Tyagi

[1]Student, [2]Student, [3]Assistant Professor,
[1]Computer Science and Engineering, School of Computer Science and Engineering,
[1]Vellore Institute of Technology Chennai (UGC), Chennai, 600127, Tamil Nadu, India
[2]Computer Science and Engineering, School of Computer Science and Engineering,
[2]Vellore Institute of Technology Chennai (UGC), Chennai, 600127, Tamil Nadu, India
[3]Computer Science and Engineering, School of Computer Science and Engineering,
[3]Vellore Institute of Technology Chennai (UGC), Chennai, 600127, Tamil Nadu, India

Abstract: Containerization is the way toward bundling a software with all of its essentially required runtime-libraries, frameworks and system-configuration files so that it can be executed proficiently in a variety of computing environments [1]. Containers do not put a strain on the system, requiring only the bare minimum of resources to operate the solution without the need to replicate an entire operating system. Since the program requires less resources to run, it can run a greater number of applications on the same hardware, lowering costs as compared to virtual machines which require separate guest OS for isolating the environment for the program libraries. Docker is one of the platforms for containerization and has many orchestration programs for the efficiency. This paper focus on the different algorithms proposed for allocation of resources and talks about their results obtained.

Index Terms - cloud-based computing, Kubernetes Docker-containers, Docker, container-orchestration, virtual machine, microservices

I. Introduction

Containerization is the key to the issue of moving software from one computing environment to another and having it run reliably. This might be from a designer's PC to a test setting, from arranging to yield, or from an actual bodily present data-centre-server to a virtual machine in a private or public cloud. Issues emerge when the auxiliary programming environment isn't correspondent. For example, one might test with Python 2.7 and then run it in production with Python 3 and something won't work, or one might run and execute them for testing on Debian distribution of Linux, but the production is based on the platform of Red Hat organization, and both of these issues arise due to different software and modified versions. A container is a whole runtime environment stuffed into one bundle/package: an application, alongside the entirety of its libraries, conditions, different binaries, dependencies and configuration setting files which are necessary for the execution.

Contrasts in operating system dispersions and basic framework are disconnected away by containerizing the application stack and its conditions/dependencies. A container could be a many megabyte in size, yet a virtual machine with its own working framework could be a few gigabytes. Subsequently, a solitary server will have a lot bigger number of containers than virtual machines. One of the advantages is reduced loading time in containers than in virtual machines and applications can be broken into modules and modules can be run on discrete containers for better execution. Common applications are microservices, batch processing, machine learning, hybrid applications. Google developers also created Cgroups in Linux which can isolate resource use for user processes which can is put into namespaces which are collection of processes that share same resources. Linux Cgroups led to the creation of Linux containers (LXC).

LXC was the first major implementation of what we now call a container, using Cgroups and namespace isolation to construct a virtual environment with distinct process and networking space. In Docker, the container's operating system is in the form of an image. The distinction between this image and the full operating system on the host is that the image only contains the file system and binaries for the OS, while the full OS contains the file system, binaries, and kernel. The image and its parent images are downloaded from the repo, the Cgroup and namespaces are generated, and the image is used to create a virtual environment when a container is booted. The files and binaries listed in the image appear to be the only files on the entire system from inside the container. The main operation of the container is then begun, and the container is considered alive. Users have attempted to deploy large scale applications over several virtual machines since the introduction of Linux containers, with each process running in its own container. This necessitated the ability to effectively deploy tens to thousands of containers across hundreds of virtual machines, as well as handle their networking, file systems, and other resources and led to introduction of orchestration like Kubernetes, Docker Swarm.

Orchestration do schedule, booting the containers, upgrading and rollbacking, responding to failures and restarting the containers. Container-as-a-Service (CaaS) is a model for running containers on an enterprise platform; however, certain extra highlights of these frameworks, like deployment in production and arrangement mechanization, render this stage an undeniable Platform-as-a-Service (PaaS). In spite of the fact that CaaS will execute containers at scale on the computer, PaaS takes input of the source-code, foster it, build containers, and ensure their execution. Organizations are normalizing platforms around technologies which are based and executed on Kubernetes which is one of the major open-source technologies (also known as K8S in short). Google dispatched K8S as an open-source project that is currently managed and developed by various huge organizations which are also acting as platform vendors. Container workloads can also be moved between public clouds using K8S. These are the reasons why Kubernetes is being used by an increasing number of technology companies.

This paper will focus on the algorithms used for different orchestration systems like Kubernetes and find which one is efficient according to the conditions given.

II. Related Works

There are many orchestration tools available and have their own implementation and handling for the containers. The widely used tool is the Kubernetes which was developed by Google. It has a whole wide working system which include pods, deployments, nodes, clusters and services [2]. A Kubernetes pod is a set of containers that Kubernetes manages at the smallest scale and have a single IP address that is assigned to all of the containers in the pod which share the same memory and storage resources. This allows the individual Linux containers within a pod to be viewed as a single programme, as if they were all running on the same host in more conventional workloads. It tends to be a solitary container when the program or process service is a solitary process that necessities to operate, or it tends to be a multi-container pod unit when a few process cycles need to cooperate utilizing same common information data volumes for suitable execution [25].

Kubernetes deployments allow you to specify the scale at which you want to run your application by specifying the specifics of how pods should be replicated across your Kubernetes nodes. Deployments specify the number of identical pod replicas that should be run as well as the chosen upgrade strategy for updating the deployment. Kubernetes can monitor pod health and remove or add pods as required to achieve the desired state for your application deployment. If a pod dies due to a problem, Kubernetes is responsible for replacing it so that the application does not experience any downtime. A service is a layer of abstraction over the pods that serves as the only point of contact for the various application users. Internal names and IP addresses of pods can change as they are replaced. A service maps pods with unreliable underlying names and numbers to a single system name or IP address and guarantees that it appears to be the same to the outside network. The computer (whether virtualized or physical) that performs the provided work is managed and operated by a Kubernetes node. A node gathers entire pods that act together, just as pods collect individual containers that work together. All of the above components are combined into a single cluster [24].

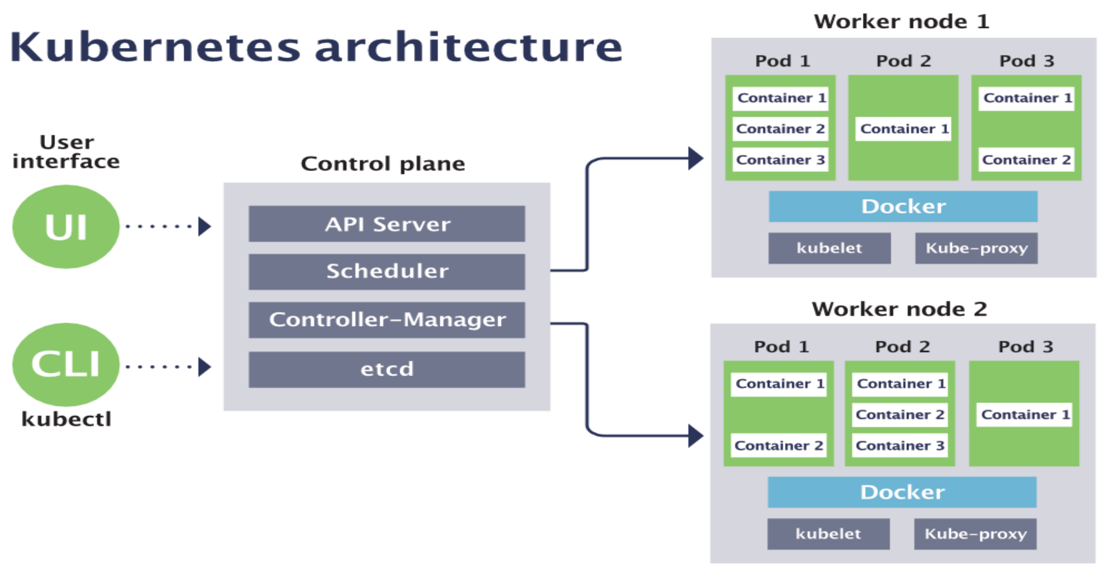

Figure 1. The Kubernetes Architecture.

Next enterprise level orchestration tool is OpenShift, developed by Red Hat. Microservices-primarily based structure of lesser big, decoupled devices that work collectively makes up the OpenShift Container Platform [3]. It's based on top of a Kubernetes cluster band and uses etcd, a reliable grouped key & value store, to store information about the items. REST APIs disclose every of the fundamental objects, and these services are broken down by function. Controllers read APIs, make modifications to rest of the other objects, and either report the status of the object or write back to the object. To alter the state of the device, REST APIs are utilized by the users. Controllers read the end-user's ideal state utilizing the REST API and then attempt to bring the rest of the device into sync.

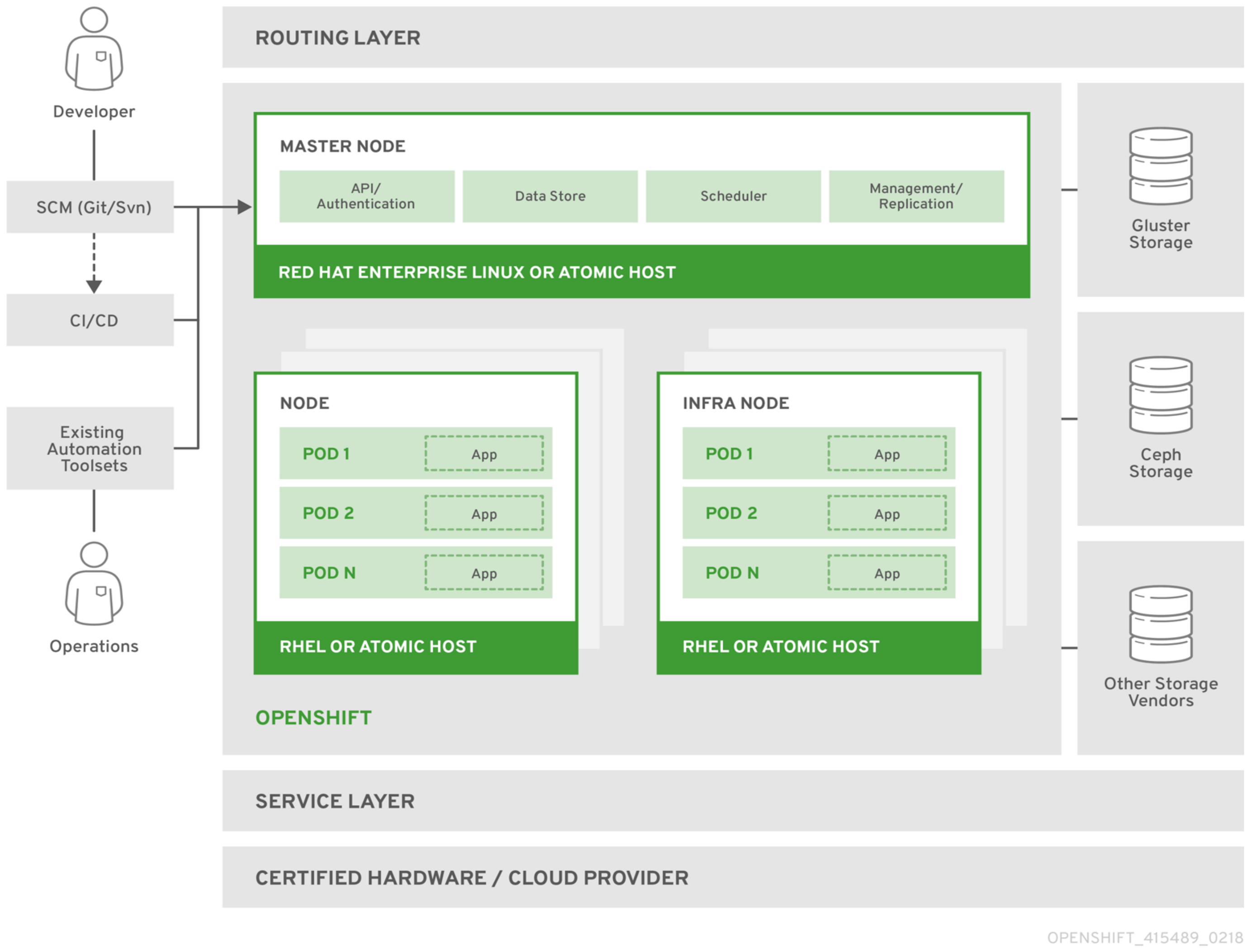

Figure 2. The OpenShift Architecture

The next orchestrator is Nomad developed by HashiCorp. Nomad is a straightforward, adaptable, and simple to utilize responsibility workload orchestrator to send and oversee by managing the containers and non-containerized applications across on-prem and clouds at scale [4]. Nomad executes as a solitary binary with a little asset resource footprint impression (35MB) and natively supported by Windows, macOS, Linux. Machine learning (ML) and artificial intelligence (AI) workloads are supported natively by Nomad (AI). Device plugins enable Nomad to automatically detect and use resources from hardware devices like GPUs, FPGAs, and TPUs. The simplicity, versatility, scalability, and high performance of Nomad set it apart from similar tools. Nomad's synergy and integration points with HashiCorp Terraform, Consul, and Vault make it better suited for quick integration into an organization's current workflows, reducing vital initiative time-to-market. [20] [21]

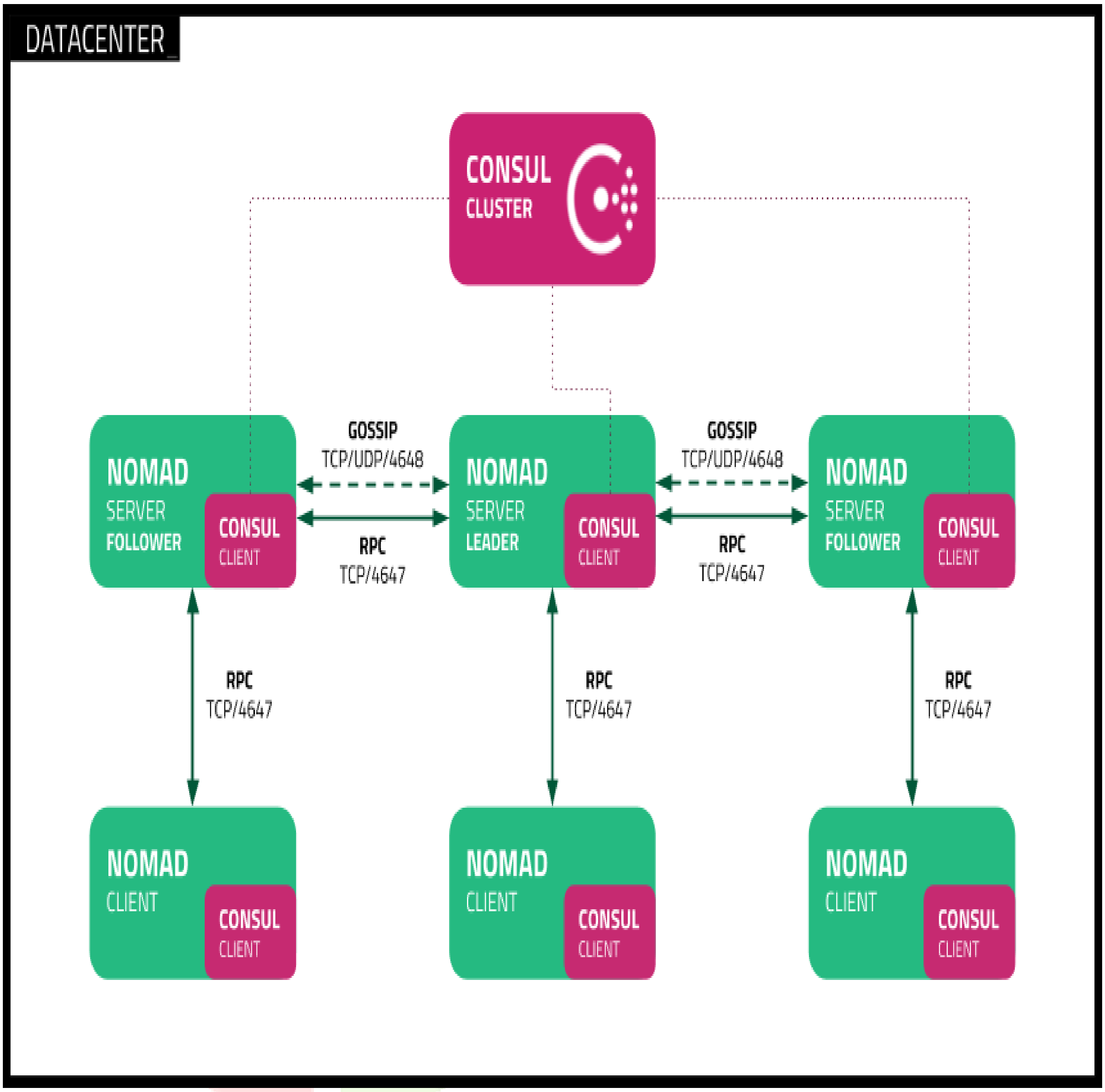

Figure 3. Nomad Architecture

Docker Swarm, built by Docker, is a top tool which is in competition to Kubernetes. Multiple Docker multitudes executes in swarm mode and serve as managers (to deal with delegation and membership) and staff in a swarm (which executes swarm services) [5]. Any Docker host may be a staff, a boss, or both at the same time. You determine the optimum state of a service when you develop it (number of networks, imitations and resource capacity storage assets accessible to it, ports the assistance opens to the rest of the world, and that's only the tip of the iceberg). Docker always strives and attempt to keep the desired state of itself. Docker, for example, schedules an operative worker node's tasks on other nodes if that node becomes unavailable. A job, as opposed to a standalone container, is a running container that is essential part of a swarm service and which is operated by a swarm manager. To expose the resources, one needs to make resources and assets available external remotely to the swarm, for which the swarm manager utilizes the ingress load-balancing [23]. The swarm manager can allocate a PublishedPort to the service automatically, or you can manually configure one. Any unused port may be defined. External elements, for example, such as cloud server load-balancers, are able to get the access to the service through the PublishedPort of any node present in the cluster, regardless of whether that node is currently performing the service's main task/aim/mission. Ingress connections are routed to a running task instance by all nodes in the swarm. [22]

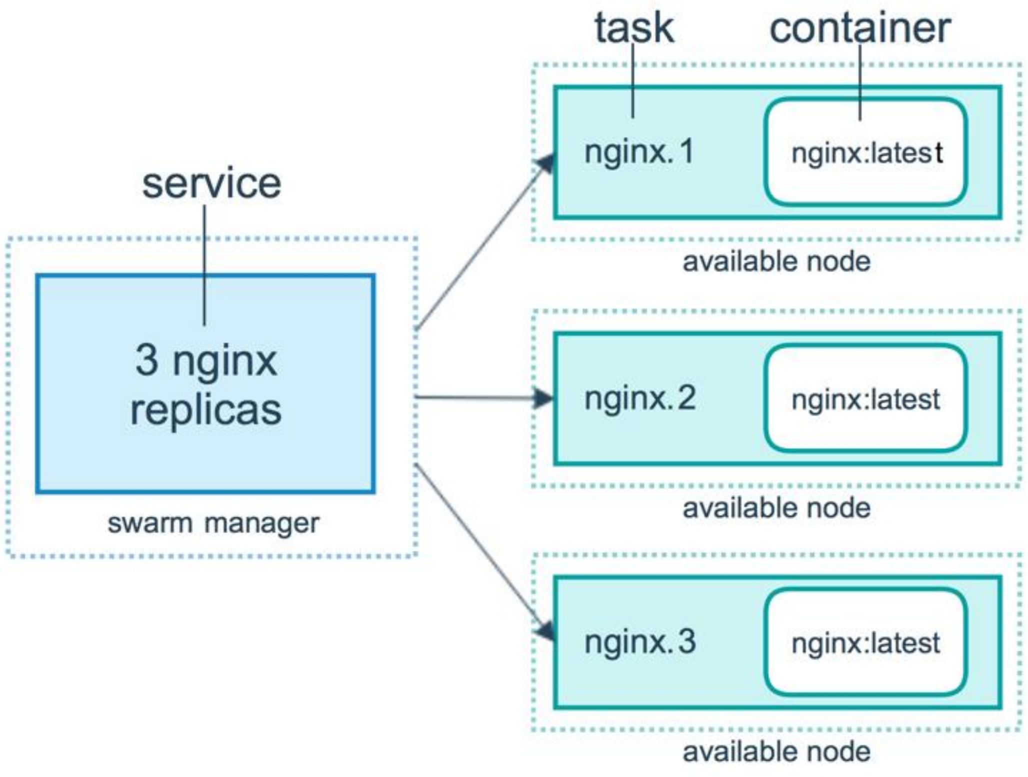

Figure 4. Docker Swarm Architecture

III. Load Balancing Algorithms

Load balancing is a method used by businesses to distribute workload through several servers in a server pool. It works like a virtual traffic cop, routing client requests through servers in order to respond quickly and efficiently. It ensures that no single server is overburdened or suffers performance degradation. The load balancer redirects traffic to the remaining online servers if one of the servers goes down. If a new server is connected to the server pool, the load balancer can send requests to it automatically and will rebalance the load around the pool. To intelligently stack load-balance client's access requests through the server pools, a variety of techniques and algorithms can be used. The technique used will be determined by the kind of administration service or application being dealt with, just as the network and server status at that particular time the solicitation request is made. The algorithms defined below will be consolidated to decide which server is better suited to handle new requests. The system used is also determined by the existing number of requests to the load-balancers. One of the simple load balancing methods will suffice when the load is light. The more complicated approaches are used in times of high load to ensure an even distribution of requests. Following are some of the most used techniques and algorithms for load balancing:

a. **Round Robin:** The most basic essential, broadly and widely utilized load balancing algorithm is round-robin load-balancing. In a pivoting rotating design, client service requests are disseminated to application servers. In the event if we have three application servers, for e.g., the very 1st client request will be sent to the very 1st application server in the stack, the 2nd client request will be sent to the 2nd application server, the 3rd client request will be sent to the 3rd application server, the 4th client request will be sent to the 1st application server, and so on. Round-robin load-balancing algorithm overlooks application server features, assuming that all application servers are indistinguishable in terms of obtainability, computation, and load-handling.

b. **Weighted Round Robin:** Weighted Round Robin represents for different application server features, qualities. Robin builds on the fundamental Round-robin load-balancing calculation algorithm. To exhibit the application server's traffic-handling competence, the admin allocates a load to each application server based on constraints of their choice. If 1st application server is twofold as powerful as 2nd application server (and 3rd application server), 1st application server is provisioned with more weight and 2nd and 3rd application server get the same weight. Suppose there are 5 chronological client requests, the first two go to 1st application server, the 3rd goes to 2nd application server, the 4th goes to 3rd application server and the 5th goes to 1st application server.

c. **Least Connection:** Client requests are distributed to the application server with the least number of dynamic associative connections at the time the client request is submitted utilizing the least connection load-balancing algorithm. In situations where application servers have comparative necessities, a server may be over-burden due to longer-enduring connections; this algorithm considers dynamic connection load.

d. **Resource Based (aka Adaptive):** The Resource Based (Adaptive) load-balancing algorithm necessitates the installation of an agent on the application server that records the load balancer's current load. The application server's availability and resources are monitored by the installed agent. To aid load balancing decisions, the load-balancer queries the yield from the agent.

e. **Fixed Weighing:** Fixed Weighting is a heap-adjusting load-balancing algorithm where the administrator allocates a weight to each application server which depends on set of rules of their electing to exhibit the application servers' traffic-handling competence. The application server with the most elevated weigh will receive all of the traffic. If the application server with the most elevated weight fizzles, all traffic will be coordinated to the next highest weight application server.

f. **Weighted Response Time:** Weighted Response Time is a load-balancing algorithm where the response times of the application servers governs which application server obtains the next request. The application server retort time to a health check is used to calculate the application server weights. The application server that is retorting the quickest obtains the next request.

g. **Source IP Hash:** Source IP hash is a load-balancing algorithm that generates a inimitable hash key by consolidating the client and server's source and destination IP addresses. The key is used to allocate a client to a specific server. The client request is guided to the same server it was using previously so the key can be redeveloped if the session is broken up. This is useful if it's significant for a client to reconnect to an active session after a disengagement.

IV. COMPARISON OF RESOURCE MANAGEMENT ALGORITHMS

Reference	Objective	Platform	Algorithm Used	Results

Maria A. Rodriguez and Rajkumar Buyya [6];
I. Donca, C. Corches, O. Stan and L. Miclea [17];
N. Estrada and H. Astudillo [18];
	Cost-Efficient Autoscaling in Cloud Computing Environments	Implemented in Java	Best Fit Bin Packing Scheduler
			Non-Binding Re-scheduler
			Binding Re-scheduler
			Simple Auto-scaler-Scale Out / Scale In
			Simple Binding Auto-scaler-Scale Out
				For the mixed workload and slow workload, the lowest cost and scheduling duration is obtained by the Non-binding Re-scheduler and Binding Auto-scaler (NBR-BAS)
				The median scheduling time is the fastest for the slow workload and the NBR-NBAS.

Xin Xu; Huiqun Yu; Xin Pei [7]	Minimum Response Time	Java	Resource stable placement algorithm (RSP) Resource Scheduling Approach using Stable Matching Theory	The Strategy causes at most 27.4% degradation compared with the common virtual machine-based clouds which using the MinResTime strategy.
H. Zhang, H. Ma, G. Fu, X. Yang, Z. Jiang and Y. Gao [8]	Improve Resource Utilization	Docker	DRFA Resource Allocation Algorithm	DRFA outperforms the Filter Scheduler and Vector Dot methods, and the utilization ratios of CPU and memory are all over 90%
C. Kaewkasi and K. Chuenmuneewong [9]	Balance Resource Usage	Docker SwarmKit	Ant Colony Optimisation (ACO)	Performance of workloads placed by ACO(A)gained 14.80% better than the greedy algorithm
L. Yin, J. Luo and H. Luo [10]; K. Chaowvasin, P.	Reduce Task Delays	Docker	Task Scheduling Algorithm	Scheduling algorithm can increase the number of accepted

Sutanchaiyanonta, N. Kanungsukkasem and T. Leelanupab [19];				tasks by 5% and the reallocation mechanism can significantly decrease the execution time for each task by 10%
Guerrero, C., Lera, I. & Juiz, C [11]	Reduce Network Overhead	Sock Shop	Genetic algorithm approach, using the Non-dominated Sorting Genetic Algorithm-II (NSGA-II)	The approach obtained values up to 58.1%better for Network Distance, 44.1% for Balanced Cluster. Moreover, 44.1% for System Failure, and 453.9% for Threshold Distances compared with Kubernetes allocation policies. The solution also used a smaller number of physical machines, except for in one of the experiments, with improvement ratios of up to 4.888.
M. Xu, A. N. Toosi and R. Buyya [12]	Reduce energy	Docker	Lowest Utilisation Container First (LUCF) Minimum Number of Components First Policy (MNCF) Random Selection Container Policy (RSC)	LUCF achieves better energy consumption than NPA, BOB and Auto-S. According to response time and SLA violation comparison, LUCF outperforms Auto-S. Compared with BOB, LUCF has better performance when optional utilization percentage is larger than 30 percent.
J. Herrera and G. Moltó [13]	Dynamic Distributed Auto-scaling	Simulation	Bio-inspired algorithms	Sharper load peaks or series with greater number of outliers are solved better with horizontal scaling and repetitive load are conveniently solved with vertical scaling
M. M. Rovnyagin, S. O. Dmitriev, A. S. Hrapov and V. K. Kozlov [14]	Accelerating the Re-scheduler	Docker	Reinforcement Learning based Re-scheduler	The Re-scheduler can work in a variety of very different situations. In addition, the training process of the Re-scheduler core - ML-agent is accelerated and simplified.
Liu, B., Li, P., Lin, W. et al [15]	Optimize Resource Scheduling	Docker Swarm	Multiopt algorithm	Compared to Spread, Binpack, and Random, Multiopt increases the maximum TPS by 7% and reduces the average response time per request by 7.5%

				while consuming roughly same allocation time.

V. Conclusion

In this paper, we talked about the basics of containers and how orchestration tools are applied to ease container management and handling. We saw few examples of industry grade orchestration tools like Kubernetes, nomad and their architectures. We talked about different load balancing algorithms upon which modern algorithms are based and how they determine best performance for websites and applications. Next, we compared some proposed algorithms for different purposes based upon resource allocation and discussed about the results and which platforms they were based on.

VI. Future Works

For future works, one can establish a connection between the different algorithms discussed and can make a super algorithm which reduces the resource space but also is time efficient, which utilizes the resource efficiently and also provides fast connections and server response.

VII. References

[1] Marutitech [online] Available: https://marutitech.com/containerization-and-devops/

[2] Kubernetes [online] Available: https://kubernetes.io/docs/concepts/architecture/

[3] RedHat OpenShift [online] Available: https://docs.openshift.com/container-platform/3.5/architecture/index.html

[4] Nomad [online] Available: https://www.hashicorp.com/resources/how-does-nomad-work

[5] Docker Swarm [online] Available: https://docs.docker.com/engine/swarm/key-concepts/

[6] Rodriguez, Maria and Rajkumar Buyya. "Container Orchestration with Cost-Efficient Autoscaling in Cloud Computing Environments." *Handbook of Research on Multimedia Cyber Security*, edited by Brij B. Gupta and Deepak Gupta, IGI Global, 2020, pp. 190-213. http://doi:10.4018/978-1-7998-2701-6.ch010

[7] X. Xu, H. Yu and X. Pei, "A Novel Resource Scheduling Approach in Container Based Clouds," *2014 IEEE 17th International Conference on Computational Science and Engineering*, 2014, pp. 257-264, doi: 10.1109/CSE.2014.77.

[8] H. Zhang, H. Ma, G. Fu, X. Yang, Z. Jiang and Y. Gao, "Container Based Video Surveillance Cloud Service with Fine-Grained Resource Provisioning," 2016 IEEE 9th International Conference on Cloud Computing (CLOUD), 2016, pp. 758-765, doi: 10.1109/CLOUD.2016.0105.

[9] 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), 2017, pp. 254-259, doi: 10.1109/KST.2017.7886112.

[10] L. Yin, J. Luo and H. Luo, "Tasks Scheduling and Resource Allocation in Fog Computing Based on Containers for Smart Manufacturing," in IEEE Transactions on Industrial Informatics, vol. 14, no. 10, pp. 4712-4721, Oct. 2018, doi: 10.1109/TII.2018.2851241.

[11] Guerrero, C., Lera, I. & Juiz, C. Genetic Algorithm for Multi-Objective Optimization of Container Allocation in Cloud Architecture. *J Grid Computing* **16,** 113–135 (2018). https://doi.org/10.1007/s10723-017-9419-x

[12] M. Xu, A. N. Toosi and R. Buyya, "iBrownout: An Integrated Approach for Managing Energy and Brownout in Container-Based Clouds," in IEEE Transactions on Sustainable Computing, vol. 4, no. 1, pp. 53-66, 1 Jan.-March 2019, doi: 10.1109/TSUSC.2018.2808493.
[13] J. Herrera and G. Moltó, "Toward Bio-Inspired Auto-Scaling Algorithms: An Elasticity Approach for Container Orchestration Platforms," in *IEEE Access*, vol. 8, pp. 52139-52150, 2020, doi: 10.1109/ACCESS.2020.2980852.

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

[15] Liu, B., Li, P., Lin, W. *et al.* A new container scheduling algorithm based on multi-objective optimization. *Soft Comput* **22,** 7741–7752 (2018). https://doi.org/10.1007/s00500-018-3403-7

[16] M. Rostanski, K. Grochla and A. Seman, "Evaluation of highly available and fault-tolerant middleware clustered architectures using RabbitMQ," *2014 Federated Conference on Computer Science and Information Systems*, 2014, pp. 879-884, doi: 10.15439/2014F48. http://ieeexplore.ieee.org/document/6933108

[17] I. Donca, C. Corches, O. Stan and L. Miclea, "Autoscaled RabbitMQ Kubernetes Cluster on single-board computers," 2020 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR), 2020, pp. 1-6, doi: 10.1109/AQTR49680.2020.9129886. http://ieeexplore.ieee.org/document/9129886

[18] N. Estrada and H. Astudillo, "Comparing scalability of message queue system: ZeroMQ vs RabbitMQ," 2015 Latin American Computing Conference (CLEI), 2015, pp. 1-6, doi: 10.1109/CLEI.2015.7360036. http://ieeexplore.ieee.org/document/7360036

[19] K. Chaowvasin, P. Sutanchaiyanonta, N. Kanungsukkasem and T. Leelanupab, "A Scalable Service Architecture with Request Queuing for Resource-Intensive Tasks," 2020 17th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON), 2020, pp. 67-70, doi: 10.1109/ECTI-CON49241.2020.9158114. http://ieeexplore.ieee.org/document/9158114

[20] V. M. Ionescu, "The analysis of the performance of RabbitMQ and ActiveMQ," 2015 14th RoEduNet International Conference - Networking in Education and Research (RoEduNet NER), 2015, pp. 132-137, doi: 10.1109/RoEduNet.2015.7311982. http://ieeexplore.ieee.org/document/7311982

[21] A. M. Beltre, P. Saha, M. Govindaraju, A. Younge and R. E. Grant, "Enabling HPC Workloads on Cloud Infrastructure Using Kubernetes Container Orchestration Mechanisms," 2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC), 2019, pp. 11-20, doi: 10.1109/CANOPIE-HPC49598.2019.00007. http://ieeexplore.ieee.org/document/8950981

[22] C. Link, J. Sarran, G. Grigoryan, M. Kwon, M. M. Rafique and W. R. Carithers, "Container Orchestration by Kubernetes for RDMA Networking," 2019 IEEE 27th International Conference on Network Protocols (ICNP), 2019, pp. 1-2, doi: 10.1109/ICNP.2019.8888116. http://ieeexplore.ieee.org/document/8888116

[23] D. Ermolenko, C. Kilicheva, A. Muthanna and A. Khakimov, "Internet of Things Services Orchestration Framework Based on Kubernetes and Edge Computing," 2021 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (ElConRus), 2021, pp. 12-17, doi: 10.1109/ElConRus51938.2021.9396553. http://ieeexplore.ieee.org/document/9396553

[24] A. Tesliuk, S. Bobkov, V. Ilyin, A. Novikov, A. Poyda and V. Velikhov, "Kubernetes Container Orchestration as a Framework for Flexible and Effective Scientific Data Analysis," 2019 Ivannikov Ispras Open Conference (ISPRAS), 2019, pp. 67-71, doi: 10.1109/ISPRAS47671.2019.00016. http://ieeexplore.ieee.org/document/8990167

[25] R. Eidenbenz, Y. Pignolet and A. Ryser, "Latency-Aware Industrial Fog Application Orchestration with Kubernetes," 2020 Fifth International Conference on Fog and Mobile Edge Computing (FMEC), 2020, pp. 164-171, doi: 10.1109/FMEC49853.2020.9144934. http://ieeexplore.ieee.org/document/9144934