Container-based Service State Management in Cloud Computing

Shubha Brata Nath*, Sourav Kanti Addya†, Sandip Chakraborty‡ and Soumya K Ghosh§
Department of Computer Science and Engineering
*‡§Indian Institute of Technology Kharagpur, India †National Institute of Technology Karnataka, Surathkal, India
Email: *nath.shubha@gmail.com, †souravkaddya@nitk.edu.in, ‡sandipc@cse.iitkgp.ac.in, §skg@cse.iitkgp.ac.in

***Abstract*—In a cloud data center, the client requests are catered by placing the services in its servers. Such services are deployed through a sandboxing platform to ensure proper isolation among services from different users. Due to the lightweight nature, containers have become increasingly popular to support such sandboxing. However, for supporting effective and efficient data center resource usage with minimum resource footprints, improving the containers' consolidation ratio is significant for the cloud service providers. Towards this end, in this paper, we propose an exciting direction to significantly boost up the consolidation ratio of a data-center environment by effectively managing the containers' states. We observe that many cloud-based application services are event-triggered, so they remain inactive unless some external service request comes. We exploit the fact that the containers remain in an idle state when the underlying service is not active, and thus such idle containers can be checkpointed unless an external service request comes. However, the challenge here is to design an efficient mechanism such that an idle container can be resumed quickly to prevent the loss of the application's quality of service (QoS). We have implemented the system, and the evaluation is performed in Amazon Elastic Compute Cloud. The experimental results have shown that the proposed algorithm can manage the containers' states, ensuring the increase of consolidation ratio.**

***Index Terms*—Container, State Management, Service Management, Cloud Computing, Container Migration, Checkpointing.**

I. INTRODUCTION

Containers [1] have emerged as an essential and effective alternative for hypervisor-based virtualization that complements the virtual machines (VMs) with a lightweight orchestration framework while providing an isolated, standalone, and reliable computing environment. Containers [2] can run over bare-metals and within a VM, and therefore, it offers significant computation flexibility for application services deployment. Due to its low overhead, container-based virtualization supports higher consolidation (number of container-based virtual servers that can run over a host) compared to hypervisor-based virtualization [3]. Consequently, it promotes the dense deployment of user applications over a physical host machine [4]–[6].

In a typical virtualized cloud environment, containers are a good candidate for deploying stateful application servers [7] like gaming servers, servers providing machine learning toolboxes like optical character recognition, speech processing, image processing, etc., augmented reality and virtual reality servers, and so on. These types of server applications have less dependency on the underlying operating system. Therefore, a containerized version of those application servers can be hosted on a cloud platform with less overhead. One typical nature of such application servers is that they get executed when some external client requests ask for services; otherwise, they keep running in an idle state. Therefore, the application server states can be checkpointed, and the server's resources can be released temporarily for the best utilization of the hardware resources [8]. The application service can be resumed from the checkpoint when some external client requests come. Although such an approach can significantly improve the hardware utilization and the cloud environment's energy efficiency, dynamic checkpointing for stateful containerized applications has two major *challenges*.

i. Different containers need to be checkpointed dynamically when they are idle, and their operations need to be resumed later from that state with minimum startup latency. Also, any partially processed data for the containerized application needs to be stored.
ii. During the restoration of a checkpointed service, sufficient hardware resources are not available at the original physical host, due to the dynamic application workload for various other services running over the same physical host. Therefore, there is a need to share the checkpointed state of the containers between different computing servers. Moreover, the checkpointed state needs to be migrated to a new server based on the demand for resuming the idle container when external client requests come for that application service.

Therefore, we require to develop a method for dynamic checkpointing and restoration of containerized stateful application services so that the *consolidation ratio* increases. The *consolidation ratio* is defined as average number of virtual instances on each host [9]. In the proposed system, the virtual instances are containers. Accordingly, in this paper, we propose a dynamic container checkpointing and service migration [10], [11] strategy considering the dynamic application demands and workloads of a containerized stateful service. In order to solve the above-mentioned challenges, we formulate an optimization problem and propose a checkpoint based heuristic algorithm to solve the problem of container deployment. The objective function considers the increase of consolidation ratio. The major *contributions* of this work are:

978-3-903176-32-4 © 2021 IFIP

i. The service should go to checkpointed state by freeing the idle container's consumed resources when no request is coming for that service. Other services that need to serve the clients are deployed dynamically after deallocating the idle containers. However, this management is challenging as there is a need to deallocate and allocate containers based on the container state dynamically.

ii. To analyse the system's performance, we have performed experiments in Amazon Elastic Compute Cloud (Amazon EC2) [12]. For this analysis, Amazon EC2 VMs are taken as the servers. We have considered docker [13] as the container engine and Checkpoint/Restore In Userspace (CRIU) [14] as the software to freeze a running container. Container state management is done by a shared storage and the docker checkpoint feature. The shared storage is created using a network file system (NFS) [15].

iii. Whenever the current server can not allocate the checkpointed container, the system supports migration of docker containers by moving the container's state in a new server and starting the container there from the checkpointed state. The experimental results from Amazon EC2 show that the proposed system can increase the *consolidation ratio*.

II. RELATED WORK

A number of works have focused on various aspects of the problem of container based service state management in the recent literature. In [7], the authors have provided a middleware to achieve high availability for cloud applications. The solution can compensate the limitations of linux containers in achieving high availability. They are monitoring the containers that hosts critical components and checkpoint its state. In case of a failure, the computation is resumed from the most recent state. The feasibility of their solution is verified using a case study. The authors in [16] have performed a container migration service. It is a filesystem-agnostic and vendor-agnostic consistent full-system migration service. The work incorporates CRIU-based memory migration and focuses on minimizing the migration time. However, no service response time related analysis is present in this work. In order to do failure recovery of multi-tier stateful applications, Gomes et al. [17] have performed an evaluation of checkpoint services in both virtualized and physical environments. They have considered the checkpoint in application-level as well as in system-level. Though analysis of failover time is given, service response time is not considered. In [18], the authors have proposed elastic provisioning of virtual machines for containers deployment which takes into account the heterogeneity of containers requirements and computing resources. Their approach can determine the container deployment on virtual machines on-demand while optimizing quality of service (QoS) metrics. It evaluates the container deployment at runtime considering the QoS metrics. If it achieves an improvement based on the QoS metrics, a reconfiguration is planned. The authors in [19] have modeled the micro-service based application (container) deployment problem to minimize total cost considering server capacity and service delay as the constraints. They have proposed a communication efficient framework and a suboptimal algorithm to determine container placement. In [20], the authors have developed hybrid autoscaling algorithms and a network scaling algorithm. The hybrid autoscaling algorithm provides high availability by horizontal scaling and fine-grained resource control by vertical scaling. The authors in [21] have solved the problem of container placement across heterogeneous infrastructure to minimize the overall energy consumption and balance the load of the hosts. The problem have been formulated as a multi-objective optimization problem. The authors have solved the problem based on an incremental exploration of the solution space. The works of [22] have introduced a resource allocation framework for the containerized deployment of microservices to reduce operating costs while providing a minimum guaranteed level of service.

From the above discussion, we observe that the existing studies mostly look into the placement of the services along with migration needs. Our objective is to increase the consolidation ratio i.e. increase the number of containers per host.

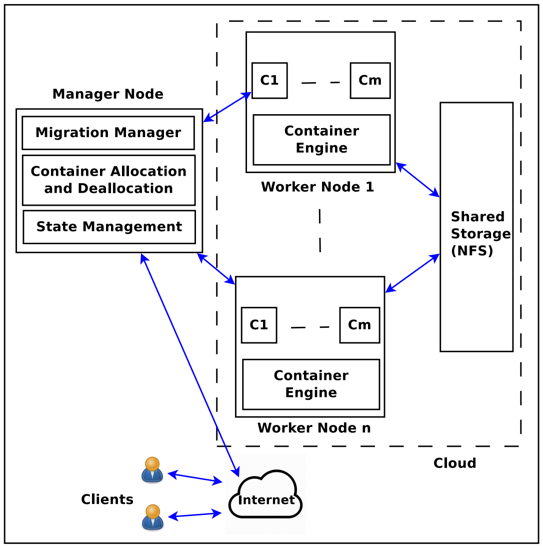

Fig. 1. System architecture

III. SYSTEM ARCHITECTURE

The proposed architecture of the system is discussed in this section, which consists of the following components (Figure 1).

i. **Clients:** Clients send service request to the manager node of the cloud data center.

ii. **Manager Node:** The manager node is connected to all the worker nodes of the cloud data center. The manager node has a *container allocation and deallocation* module to place the services in the cloud data center. This node does the aggregation of the results. Also, the final result is sent to the clients. We need to find the status of the container state (active/idle). This work is performed by the *state management* module. The *container allocation*

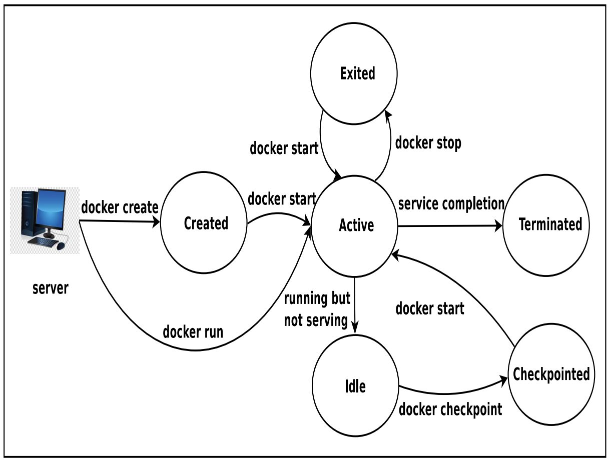

Fig. 2. State transition diagram of a docker container

and deallocation module checkpoints the idle containers. If the current server does not have sufficient resources, we need to migrate the container to a new worker node. This migration is performed by the *migration manager* module of the manager node. *State management* module is responsible for letting the *migration manager* module know about the idle containers to be migrated at each time instant.

iii. **Cloud Worker Node:** The cloud worker node has container engine (https://docs.docker.com/get-started/overview/) to run different services as a container.

iv. **Shared Storage using Network File System:** The cloud worker nodes share a storage to save the state of the checkpointed containers. This shared storage is designed using NFS. NFS server is configured in a worker node that needs to share its directory with other worker nodes. NFS client is configured in the worker nodes that need to access the NFS server's directory.

The management of a container's states is a very challenging task as a container goes through many states in its life-cycle. We discuss the state transition of a docker container in Figure 2. When a docker is running and serving the requests, it goes to an *active state*. The docker container goes to an *idle state* when it is running but not serving any requests. Docker containers can be checkpointed to save the current running state of the container. The docker container goes to a *checkpointed state* when we save the running state of the container. The checkpointed container can be resumed to make it active again. After the complete service completion, the docker container goes to a *terminated state*.

IV. SERVICE STATE MANAGEMENT OF CONTAINERIZED APPLICATIONS

Let us consider c containers and s servers present in the system at a particular time instant. We denote $S = \{S_i : i \in (1, \ldots, s)\}$ and $C = \{C_j : j \in (1, \ldots, c)\}$ as sets of cloud data center servers and containers respectively.

A. *Average Startup Latency of the Services*

We need to start the checkpointed service whenever a request comes. Therefore, the startup latency for the checkpointed container is defined as

$$Startup\_latency_i = Container\_resumption\_time_i + Migration\_time_i$$

where $Container\_resumption\_time_i$ is the time taken by the system to resume the container i and $Migration\_time_i$ is the communication time needed to transfer the state from the source node to the destination node for the container i.

We define the startup latency for the new container as

$$Startup\_latency_i = Container\_deployment\_time_i$$

where $Container\_deployment\_time_i$ is the time taken by the system to deploy the new container i. $Startup\_latency_i$ is the individual startup delay of a service i. Therefore, the average startup latency is defined as

$$Avg_{startup\_latency} = \frac{\sum_{i=1}^{c} Startup\_latency_i}{c}$$

B. *Average Response Time of the Services*

We define the response time of a service as

$$Resp\_time_i = Startup\_latency_i + Service\_running\_time_i$$

where $Service\_running\_time_i$ is the processing time taken for a service in a server. Now, we define the average response time of the services as,

$$Avg_{resp\_time} = \frac{\sum_{i=1}^{c} Resp\_time_i}{c},$$

where $Resp\_time_i$ is the individual response time of a service i.

C. *Constraints*

We define C_j^{CPU}, C_j^{RAM}, and C_j^{BW} as the CPU, RAM, and bandwidth resources needed by a container C_j respectively. The available CPU, available RAM, and available bandwidth of the server S_i are Re_i^{CPU}, Re_i^{RAM}, and Re_i^{BW} respectively.

We find k servers K=(S_1,..., S_k) from the set of all servers S where $K \subseteq S$ such that the following conditions are true.

$$\sum_{j=1}^{TC_i} C_j^{CPU} \times x_{i,j} \leq Re_i^{CPU} \times y_i \tag{1}$$

$$\sum_{j=1}^{TC_i} C_j^{RAM} \times x_{i,j} \leq Re_i^{RAM} \times y_i \tag{2}$$

$$\sum_{j=1}^{TC_i} C_j^{BW} \times x_{i,j} \leq Re_i^{BW} \times y_i \tag{3}$$

where i = 1,...,s and $0 \leq j \leq c$. We define $x_{i,j}$ as the decision variable to denote if a container C_j has been placed in the server S_i. Also, y_i is the decision variable to denote if a server S_i is used for placement or not. TC_i is the total number of containers present in the server S_i.

$$k = \sum_{i=1}^{s} y_i \geq 1 \tag{4}$$

$$\underset{i,j}{\forall} x_{i,j} \in \{0,1\}; \underset{i}{\forall} y_i \in \{0,1\} \tag{5}$$

The following equation states that all the containers should be deployed in the system.

$$\sum_{i=1}^{k} TC_i = |C| \tag{6}$$

where $|C|$ is the cardinality of the set C.

We have another constraint that one container can be placed in exactly one server.

$$\underset{j}{\forall} \sum_{i=1}^{k} x_{i,j} = 1 \tag{7}$$

D. *Problem Definition*

Given a set of containers (C) with resource requirement and a set of physical servers as workers (S) with resource availability, our objective is to deploy the containers dynamically by checkpointing the idle containers so that the consolidation ratio ($Consolidation\_ratio$) increases. Therefore, we present the objective function in Equation (8).

$$\text{maximize} \quad Consolidation\_ratio \tag{8}$$

subject to the constraints given in IV-C.

It can be shown that the bin packing problem [23], a known $\mathcal{NP}$-Hard problem, is polynomial time reducible to our containerized service state management problem given in Equation (8). Therefore, the problem of containerized service state management in the cloud data center is $\mathcal{NP}$-Hard.

V. ALGORITHM DESIGN

We propose a heuristic algorithm called *Service Deployment Algorithm* to find a near optimal solution. Also, we present *Container Checkpointing Algorithm* and *Container Resumption Algorithm* to checkpoint and resume the containers respectively. We discuss the algorithms below.

A. *Service Deployment Algorithm*

We present the *Service Deployment Algorithm* in Algorithm 1. The inputs of the algorithm are container set (C) and cloud server set (S). The output of the $Consolidation\_ratio$. Our aim is to maximize the consolidation ratio. The checkpointed containers are deployed in the same server where it run last time if the server is able to meet the resource requirement (Eq. (1, 2, 3)). Otherwise, a new server is chosen to deploy the checkpointed container. The proposed algorithm deploys the containers in the selected server (S_u). If the resource requirement is not met for some containers, we find whether a checkpoint in the selected server can give enough resources for the container to be deployed. If a checkpoint can give the required resources, we identify the idle containers in the selected server and checkpoint them. This approach frees up more resources that can be allocated to the new containers. Otherwise, a new server is chosen for checking if deployment is possible or not. In this way, the consolidation ratio is maximized by our proposed algorithm.

Algorithm 1: Service Deployment Procedure

```
Input: Container Set ($C$), Cloud Server Set ($S$)
Output: $Consolidation\_ratio$
Function Deployment(C,S):
    Sort the containers in descending order of their resource requirement and store in $Sorted\_con$;
    for all the containers in $Sorted\_con$ do
        $servers\_not\_available$ = 0;
        if the deployment request is for a checkpointed container then
            $k$ = −1;
            /* $S\_last\_run\_Ci$ is server where the checkpointed container run last time */
            $S_u$ = $S\_last\_run\_Ci$;
        else
            $k$ = 0;
            $S_u$ = $S[k]$;
        while the resource requirement of container $C_i$ is greater than the resource available in server $S_u$ do
            if checkpointing in server $S_u$ can deploy the container $C_i$ then
                $resource\_available\_in\_server$ = $Checkpoint\_idle\_containers(S_u)$;
                Break from the while loop;
            if all servers are explored then
                /* $servers\_not\_available$ is set to 1 if no servers are found for deployent */
                $servers\_not\_available$ = 1;
                Break from the while loop;
            else
                $k$ = $k$ + 1;
                $S_u$ = $S[k]$;
        if $servers\_not\_available$ != 1 then
            Deploy the container $C_i$ in the selected server $S_u$;
    Calculate $Consolidation\_ratio$;
    return $Consolidation\_ratio$;
```

B. *Container Checkpointing Algorithm*

The stateful applications remain in idle state for a long duration. We propose an algorithm for container status finding and performing checkpointing of the idle containers. We describe the *Container Checkpointing Algorithm* in Algorithm 2. The input of the algorithm is the server where the containers are

Algorithm 2: Container Checkpointing Procedure

```
Input: Server where the checkpoint is required to be performed
Output: Amount of resource available in the server
Function Checkpoint_idle_containers($Server_i$):
    for all the containers in $Server_i$ do
        Find the maximum value of CPU usage and store it in $max\_cpu\_usage\_cont_i$;
        if $current\_cpu\_usage\_cont_i \leq (\alpha \times max\_cpu\_usage\_cont_i)$ then
            $status\_cont_i \leftarrow Idle$;
            Checkpoint the idle containers;
        else
            $status\_cont_i \leftarrow Active$;
        Calculate $resource\_available\_in\_server$;
    return $resource\_available\_in\_server$;
```

Algorithm 3: Container Resumption Procedure

```
Input: Services or Containers the need to be resumed ($C\_resume$), Container Set ($C$), Cloud Server Set ($S$)
Output: $Consolidation\_ratio$
Function Resume_containers($C\_resume$,$C$,$S$):
    $C = C \cup C\_resume$;
    $con\_ratio$ =Deployment(C,S);
    return $con\_ratio$;
```

required to be checkpointed and output of the algorithm is the amount of resource available in the server. Here, α ($0 < \alpha < 1$) is the parameter to set the CPU usage threshold of the active containers.

C. *Container Resumption Algorithm*

In this subsection, we present the *Container Resumption Algorithm* for resuming the idle containers. This algorithm is described in Algorithm 3. The inputs of the algorithm are the containers that need to be resumed (C_resume), the container set (C), and the cloud server set (S). The output of the algorithm is the $Consolidation\_ratio$.

VI. PERFORMANCE EVALUATION

We performed the experiments in Amazon EC2 (https://aws.amazon.com/ec2/) using python 2.7.13.

A. *Implementation Details*

The cloud servers have Ubuntu 18.04 long-term support (LTS) (https://www.ubuntu.com/) operating system. Also, we use Docker [24] as the container engine in the servers. The docker version taken for the implementation is 17.03.2-ce. To perform the docker checkpoint, the docker *experimental feature* is enabled and CRIU (https://criu.org/Main_Page) is installed in the servers. Docker uses CRIU [25] to manage the lifecycle of processes running inside its containers (https://criu.org/Docker). We set up the shared storage using NFS (https://help.ubuntu.com/lts/serverguide/network-file-system.html) to make the checkpointed files available in different servers. We have taken zookeeper application (https://zookeeper.apache.org/), PHP web application (https://www.php.net/) and redis application (https://redis.io/) for evaluation. We have analysed the output of the *docker stats* command to check the resource status of a container. We show the values of the parameters taken in Amazon EC2 experiments in Table I. The topology considered for these experiments is shown in Figure 3.

TABLE I
PARAMETERS IN AMAZON EC2 EXPERIMENT

Parameter	**Value**
Number of Amazon EC2 virtual machine instances or worker nodes	10
Processor of a worker node	2.5 GHz Intel Xeon Family
Number of vCPU in a worker node	1
Storage of a worker node	8 GB
Operating System used	Ubuntu Server 18.04 LTS
Maximum available RAM of the worker node	680 MB
Maximum available RAM of the manager node	1780 MB

B. *Competing Heuristics*

We consider Execution Container Placement and Task assignment Algorithm (EPTA) [19] as a baseline. In EPTA, the microservice controller queries a small local region for available physical resources. The execution containers are placed in the server, which has less node cost as well as less link cost. The microservice controller requests physical resources and waits for responses from the physical machines (PM). Once the request is approved, an execution container is built on the selected PM. If the request is rejected, the PM is marked as infeasible. When some tasks have not been successfully assigned, the microservice controller queries PMs on a larger scale. We have taken first fit decreasing (FFD) as another baseline. FFD is used in many works in the existing literature [26], and it is a well-known heuristic for bin packing problem. In FFD, the containers are sorted in decreasing order of the size of the required resource. Then, FFD chooses the first server that is large enough to place the containers.

C. *Consolidation Ratio*

The consolidation ratio indicates how many containers the system can deploy per host. It is desired that the consolidation ratio should be higher for a system. We show the consolidation ratio in Figure 4. The baseline algorithms cannot maximize the consolidation ratio as these are unable to deploy containers when the system does not have resources to meet the demand of the container. Our proposed algorithm has higher consolidation ratio than the EPTA algorithm [19] and the FFD algorithm [26] as our system is able to take checkpoint for the idle services and make resources free for the new containers that need to be allocated. Thus, the proposed algorithm maximizes $Consolidation\_ratio$.

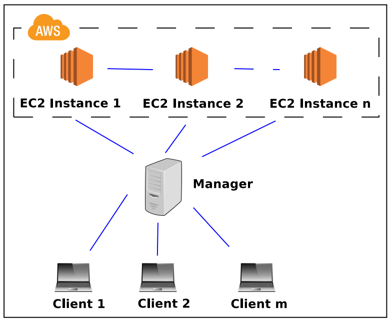

Fig. 3. Topology used in the Amazon EC2 experiments

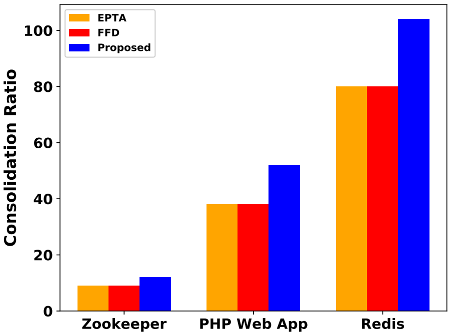

Fig. 4. Consolidation ratio in Amazon EC2

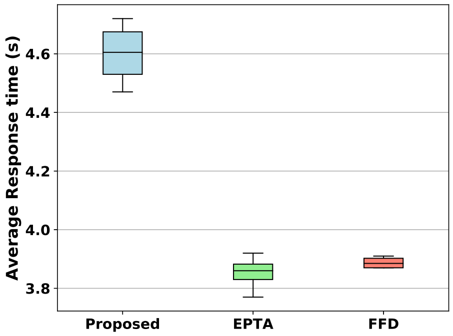

Fig. 5. Average response time of the zookeeper applications

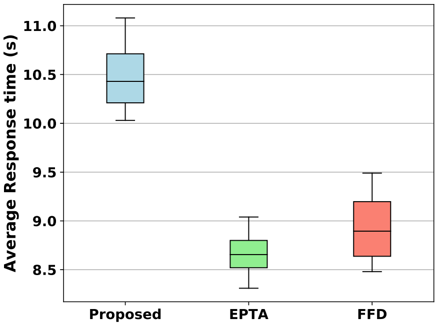

Fig. 6. Average response time of the web app

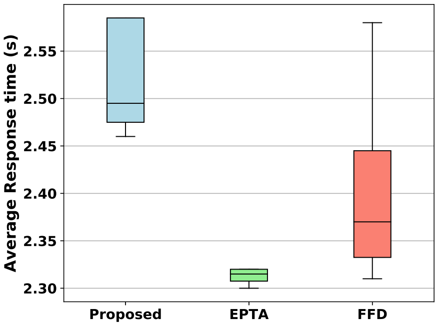

Fig. 7. Average response time of the Redis app

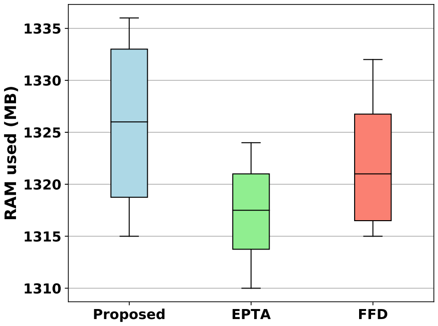

Fig. 8. RAM used by the manager node

D. Average Response Time of the Services

We show the average response time for the proposed approach and the baseline methods in Figure 5 (zookeeper application), Figure 6 (PHP web application), and Figure 7 (Redis application). The EPTA algorithm [19] places the containers of an application in the server with less node cost and less link cost. Whereas, the FFD algorithm [26] places the containers in the first server that is large enough. Our proposed algorithm can deploy more containers by checkpointing the idle containers. Our proposed algorithm has a little more average response time for the containers than the baselines due to this checkpointing.

E. RAM Used by the Manager Node

The manager node performs the deployment, checkpointing, and as well as the migration of the containers, as mentioned in section III. The manager node's RAM consumption is analyzed, and the RAM consumption is shown in Figure 8. Our proposed algorithm is running the checkpoint and resumption procedure, along with the deployment method. This consumes more amount of manager node resources. We find that the proposed algorithm has more RAM usage than the EPTA algorithm and the FFD algorithm.

VII. CONCLUSION

We have performed the container-based service state management to maximize the cloud data center's consolidation ratio. In our proposed approach, the idle containers need to be checkpointed to save the hardware resources. However, after the end of the idle period, the containers need to be resumed from the last saved state. The idle containers are migrated to a new server if the current server does not have sufficient resources to run it. The problem of state management of the containers is formulated as an optimization problem that maximizes the consolidation ratio as the objective function. We have proposed a heuristic algorithm to solve the optimization. In order to save the state of an idle container, we have proposed a container checkpointing algorithm. Also, an algorithm is proposed for the resumption of idle containers when a request comes for these services. The evaluation in Amazon EC2 shows that the proposed algorithm is able to maximize the consolidation ratio ensuring the state management of the containers. Thus, the proposed algorithm can efficiently manage the service state.

REFERENCES

[1] E. Casalicchio, "Container orchestration: A survey," in *Systems Modeling: Methodologies and Tools*. Springer, 2019, pp. 221–235.

[2] F. Rossi, V. Cardellini, and F. L. Presti, "Elastic deployment of software containers in geo-distributed computing environments," in *Proc. of IEEE ISCC'19*, 2019.

[3] K. Suo, Y. Zhao, W. Chen, and J. Rao, "An analysis and empirical study of container networks," in *proceedings of the IEEE Conference on Computer Communications*. IEEE, 2018, pp. 189–197.

[4] A. Celesti, D. Mulfari, M. Fazio, M. Villari, and A. Puliafito, "Exploring container virtualization in iot clouds," in *2016 IEEE International*

Conference on Smart Computing (SMARTCOMP). IEEE, 2016, pp. 1–6.

[5] A. Celesti, D. Mulfari, A. Galletta, M. Fazio, L. Carnevale, and M. Villari, "A study on container virtualization for guarantee quality of service in cloud-of-things," *Future Generation Computer Systems*, vol. 99, pp. 356–364, 2019.

[6] I. M. A. Jawarneh, P. Bellavista, L. Foschini, G. Martuscelli, R. Montanari, A. Palopoli, and F. Bosi, "Qos and performance metrics for container-based virtualization in cloud environments," in *Proceedings of the 20th International Conference on Distributed Computing and Networking*. ACM, 2019, pp. 178–182.

[7] W. Li, A. Kanso, and A. Gherbi, "Leveraging linux containers to achieve high availability for cloud services," in *2015 IEEE International Conference on Cloud Engineering*. IEEE, 2015, pp. 76–83.

[8] D. R. Sangolli, N. M. Ravindrarao, P. C. Patil, T. Palissery, and K. Liu, "Enabling high availability edge computing platform," in *2019 7th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)*. IEEE, 2019, pp. 85–92.

[9] O. Sukwong, A. Sangpetch, and H. S. Kim, "Sageshift: managing slas for highly consolidated cloud," in *2012 Proceedings IEEE INFOCOM*. IEEE, 2012, pp. 208–216.

[10] W. Fan, Z. Han, P. Li, J. Zhou, J. Fan, and R. Wang, "A live migration algorithm for containers based on resource locality," *Journal of Signal Processing Systems*, vol. 91, no. 10, pp. 1077–1089, 2019.

[11] S. Noor, B. Koehler, A. Steenson, J. Caballero, D. Ellenberger, and L. Heilman, "Iotdoc: A docker-container based architecture of iot-enabled cloud system," in *3rd IEEE/ACIS International Conference on Big Data, Cloud Computing, and Data Science Engineering*. Springer, 2019, pp. 51–68.

[12] S. Yang, L. Pan, and S. Liu, "An online algorithm for selling your reserved iaas instances in amazon ec2 marketplace," in *2019 IEEE International Conference on Web Services (ICWS)*. IEEE, 2019, pp. 296–303.

[13] D. N. Jha, S. Garg, P. P. Jayaraman, R. Buyya, Z. Li, and R. Ranjan, "A holistic evaluation of docker containers for interfering microservices," in *2018 IEEE International Conference on Services Computing (SCC)*. IEEE, 2018, pp. 33–40.

[14] S. Pickartz, N. Eiling, S. Lankes, L. Razik, and A. Monti, "Migrating linux containers using criu," in *International Conference on High Performance Computing*. Springer, 2016, pp. 674–684.

[15] D. Jung, D. Lee, M. Kim, and J. Kim, "Efficient data synchronization method on integrated computing environment," *The Journal of Supercomputing*, vol. 75, no. 8, pp. 4252–4266, 2019.

[16] S. Nadgowda, S. Suneja, N. Bila, and C. Isci, "Voyager: Complete container state migration," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2017, pp. 2137–2142.

[17] D. Gomes, G. Gonçalves, M. Bezerra, D. Sadok, P. T. Endo, and C. Curescu, "Failover time evaluation between checkpoint services in multi-tier stateful applications," in *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. IEEE, 2017, pp. 797–800.

[18] M. Nardelli, C. Hochreiner, and S. Schulte, "Elastic provisioning of virtual machines for container deployment," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, 2017, pp. 5–10.

[19] X. Wan, X. Guan, T. Wang, G. Bai, and B.-Y. Choi, "Application deployment using microservice and docker containers: Framework and optimization," *Journal of Network and Computer Applications*, vol. 119, pp. 97–109, 2018.

[20] A. Kwan, J. Wong, H.-A. Jacobsen, and V. Muthusamy, "Hyscale: Hybrid and network scaling of dockerized microservices in cloud data centres," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2019, pp. 80–90.

[21] K. Kaur, S. Garg, G. Kaddoum, F. Gagnon, and D. N. K. Jayakody, "Enlob: Energy and load balancing-driven container placement strategy for data centers," in *2019 IEEE Globecom Workshops (GC Wkshps)*. IEEE, 2019, pp. 1–6.

[22] N. D. Keni and A. Kak, "Adaptive containerization for microservices in distributed cloud systems," in *2020 IEEE 17th Annual Consumer Communications & Networking Conference (CCNC)*. IEEE, 2020, pp. 1–6.

[23] S. Martello, D. Pisinger, and D. Vigo, "The three-dimensional bin packing problem," *Operations research*, vol. 48, no. 2, pp. 256–267, 2000.

[24] D. N. Jha, M. Nee, Z. Wen, A. Zomaya, and R. Ranjan, "Smartdbo: smart docker benchmarking orchestrator for web-application," in *The World Wide Web Conference*, 2019, pp. 3555–3559.

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

[26] J. Son and R. Buyya, "Sdcon: Integrated control platform for software-defined clouds," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 1, pp. 230–244, 2018.