Docker Container Deployment in Distributed Fog Infrastructures with Checkpoint/Restart

Arif Ahmed, Apoorve Mohan, Gene Cooperman, Guillaume Pierre

Docker Container Deployment in Distributed Fog Infrastructures with Checkpoint/Restart

Arif Ahmed[1], Apoorve Mohan[2], Gene Cooperman[2], Guillaume Pierre[1]
[1]*Univ Rennes, Inria, CNRS, IRISA (Rennes, France)*
[2]*Khoury College of Computer Sciences, Northeastern University (Boston, MA, USA)*

***Abstract*—In fog computing environments container deployment is a frequent operation which often lies in the critical path of services being delivered to an end user. Although creating a container can be very fast, the container's application needs to start before the container starts producing useful work. Depending on the application this startup process can be arbitrarily long. To speed up the application startup times we propose to snapshot the state of fully-deployed containers and restart future container instances from a pre-started application state. In our evaluations based on 14 real micro-service containers, this technique effectively reduces the startup phase with speedups between 1x (no speedup) and 60x.**

***Index Terms*—Fog computing, Docker, containers, checkpoint/restart.**

I. Introduction

Fog computing extends traditional cloud data centers with additional compute, storage and networking resources located at the edge of the Internet. By bringing the cloud resources in the immediate vicinity of end users, fog computing delivers low end-to-end network latencies for highly-interactive applications such as augmented-reality gaming, and reduced long-distance network resource usage for applications such as IoT analytics where large volumes of transient data can be processed locally.

Fog computing architectures differ from traditional clouds: to maintain close proximity with a large number of users, fog servers must be dispersed across large geographical areas. The fog is therefore often organized into a large number of Points-of-Presence (PoPs) distributed across the covered area. Each PoP may be composed of a small number of weak machines such as single-board computers connected with each other and with the rest of the Internet using commodity networks [8]. Fog platforms typically automate application deployment across the different PoPs using container orchestration systems such as Docker and Kubernetes [14], [23].

Fog users are primarily mobile, which implies that an application dedicated to serving one user may need to be frequently re-deployed in different PoPs [10]. The mobility pattern of a human being is far from being random, and it has been observed that human mobility patterns are largely repetitive and predictable [33]. This means that the same application may be frequently re-deployed in a server from the same PoP every time a user returns to a previously-visited location. Similarly, compute-intensive applications may need to scale horizontally within a PoP. In both scenarios, the same application must be repeatedly deployed in different servers of a PoP.

In Docker, application deployment requires one to create a container, which itself starts the appropriate application. The container boot process terminates when the container's application has been initialized and is ready to serve end-user requests. However, depending on the nature of the application, this boot process may take a large amount of time. This is particularly true for complex applications such as MySQL which need to read configuration data and to undergo complex initialization procedures during their boot phase. Booting a *mysql* container on a fast server requires about 10 s before being available to serve end-user commands [24]. This delay may have a significant impact on the performance of the fog applications as the same application is repeatedly launched, created and booted in a PoP. In this work, we aim to avoid repeating this boot phase when a container is repeatedly deployed in a PoP, which in turn may reduce the average container deployment time.

To speed up the container's boot phase we propose to checkpoint the state of the container after it has started, and to start every subsequent instance of this container by reloading the checkpoint. Process checkpoint/restart is a technique where the state of a running application is saved for later reuse [32]. Restarting from a checkpoint is usually very fast, which potentially delivers significant performance gains compared to starting the application every time the container is created.

However, implementing this simple idea require one to address a number of difficult challenges. Process checkpoint/restart assumes that the application will be restarted in the same environment where it was checkpointed, so it does not save the full environment of a running application during checkpoint. An application environment contains the application's executable, shared libraries and data files, which are stored in the disk. During restart of the application from the checkpoint image, the system expects the exact same application environment to be present in the system to smoothly restart the application. In order to correctly restart an application, we need to capture the whole container environment that was present during the checkpoint operation. This full environment must be re-created in every newly created container before the restart operation takes place.

The first author's internship at Northeastern University was partly supported by a "bourse de mobilité sortante" from Rennes Métropole and by the FogRein associate team from Inria. The work of the second and third author was partially supported by NSF Grant OAC-1740218.

It is also very difficult to anticipate in which server within a PoP a container will be restarted. To maximize the utility of the saved checkpoints it is important to share them across all the servers of a PoP.

We present a new Docker deployment system which integrates checkpoint/restart using the DMTCP tool [7]. The system has two components: (1) a thin container, which enables Docker to checkpoint applications running inside the containers and to capture their container environment. It can also restart an application inside a container from its checkpoint image and the container environment; and (2) a mechanism which leverages Ceph RADOS block devices to share the container environments and checkpoint images across the servers of a PoP [29]. Our performance evaluation shows that this system can deploy containers while skipping the boot phase, which results in boot time speedups between 1x (no improvement) and 60x depending on the type of container.

The remainder of this paper is structured as follows: Section II presents the background and related work. Section III discusses the different issues in developing container deployment with DMTCP; Section IV presents the proposed Docker container deployment using DMTCP tool. Finally the performance evaluation of the proposed container deployment is presented in Section V and Section VI concludes the paper.

II. Background and related work

A. Background

1) Docker: Docker is an open-source tool to package, manage, and run applications inside containers [14]. Docker containers are packaged in the form of images which consist of multiple layers, where each layer contains a part of container's file system that contains application's binaries, libraries, data etc. Docker keeps the downloaded image layers and server configurations in a cache directory located in the local file system. Image layers are always read-only.

Container deployment consists of downloading the image layers from a repository in case they were not in cache yet, merging the read-only image layers with an additional read-/write layer using an overlay file system such as OverlayFS, and creating the container which in turn starts the application.

2) Process checkpoint/restart: Checkpoint/restart is a technique to save the current state of a single process for later restart [32]. It is primarily used to achieve fault-tolerance of an application where the application can be restored to a previous stable state after a crash [21].

Several checkpoint/restart tools are available [11]. The most notable ones are BLCR (Berkeley Lab's Checkpoint/Restart) [16], CRIU (Checkpoint and Restart In Userspace) [26] and DMTCP (Distributed MultiThreaded CheckPointing) [4]. All these tools allow one to save and restore a running application; however they handle differently the way the state of a process is preserved, the type of information of a process state which is preserved in the checkpoint image, how the preserved process information is stored (compressed or uncompressed), APIs, and command-line interfaces.

- **BLCR** captures the state of a process such as its context and allocated memory regions, and saves the checkpoint data in a file [16]. BLCR relies on a modified Linux kernel (as a patch or a loadable module). As a result it needs frequent updates to support new kernel versions. Another shortcoming of BLCR is that it does not checkpoint sockets, which are an essential part of many fog applications. In addition, it does not handle SysV shared memory. SysV shared memory among processes has become very common, for the sake of efficiency. The alternative, BSD shared memory, relies purely on parent-child relationships. BLCR has long been the most frequently-used library to checkpoint/restart processes. However it is now only lightly maintained.
- **CRIU** employs a hybrid approach which combines kernel-space and user-space [26]. It is the most commonly used system to checkpoint and restart Linux containers. During the checkpoint operation, CRIU copies the contents of memory pages, open sockets, open files etc. of a process in a file. Although Docker in its experimental mode supports CRIU-based container checkpointing, it has a number of limitations. In particular, restoration of network connections (both TCP and UDP) is currently not functional [1]. Also, although Docker supports many storage drivers for container file system management, container restoration with CRIU is only possible using AUFS or OverlayFS [27].
- **DMTCP** is a user-level tool which requires no system privileges and operates entirely in user-space [4]. When checkpointing an application, DMTCP relies on the `/proc` Linux file system to capture a map of memory pages, open file descriptors, open sockets etc. of each process of the application. While restoring the application, DMTCP reads the checkpoint image and forks all the processes, which are immediately restored to their previous state.

Some checkpoint approaches rely on customizing kernel modules (BLCR) or exporting kernel internals to the proc interface (CRIU). This type of implementation limits the checkpointing features since the system may restore applications only in machines with exactly the same kernel version (for BLCR) and with a compatible kernel version (for CRIU) [37]. Another disadvantage of such type of checkpointing techniques is that any change in the kernel may require updating the checkpointing system as well. Because of these reasons most of the kernel-based checkpoint/restart approaches do not work with recent kernel versions (for example BLCR was not updated since 2013) [17]. Moreover, CRIU and BLCR cannot fully checkpoint network sockets. Therefore, in our work, we chose to adopt DMTCP, which works entirely in user space and can checkpoint and restart network sockets.

B. Related Work

Docker containers are widely used to deploy applications in Fog computing infrastructures [20], [25]. Bellavista *et al* studied suitability of Docker container deployment in single-board computers such as Raspberry Pis [8]. The authors concluded that these devices can be used as IoT gateways

in fog, with proper configurations and optimizations, which justifies the need for improved solutions.

One way to improve the container deployment is to share the Docker images using distributed data storage. Slacker proposes to share the images with a network file system thanks to the introduction of a customized Docker storage driver [19]. Wharf shares images in high-performance servers but does not require one to change the storage driver [36]. Finally, we proposed an image-sharing framework specifically designed for fog infrastructures [3]. These frameworks improve the hit rate of Docker's image layer cache, but they do not influence the remainder of the container boot process.

Another way to improve the container deployment time is to reduce the cost of a cache miss. Docker-pi re-arranges the Docker image pull process to fully utilize the available hardware resources [2]. Similarly, FogDocker allows Docker containers to be started before their image layers have been fully downloaded [12]. These systems improve the impact of container deployment upon a cache miss; however they retain a normal application boot process and therefore incur the full application startup times.

The closest work to ours is an auto-scaling method for containers in large-scale systems. Nadgowda *et al* proposed a method based on CRIU [26] to checkpoint a running container and efficiently replicate it in other nodes [24]. However, this solution only aims at horizontal scalability scenarios and does not reduce the container initialization latency in the standalone application deployment. Our work aims to eliminate the container initialization phase in the case that the same container is repeatedly launched, created and booted.

III. Challenges

We aim to enable Docker to checkpoint a running container after completing its boot phase, and to later deploy containers from an already checkpointed image. This may remove the container boot phase while deploying the container. However, to successfully implement the above goals, we need to address three challenges, which are illustrated in the next section.

A. Integrating DMTCP with Docker

When deploying a container, Docker pulls the container image if necessary, then creates the container with the given configuration and immediately starts the boot phase. This is done by launching the first process of the application. The boot phase terminates when the application is ready to serve user requests. In contrast, our proposed design rather intends to start the application inside the container from a checkpoint image. This creates two issues. First, the system should enable Docker to use DMTCP to checkpoint an application inside the container in the first deployment. Second, while deploying containers in subsequent deployments, Docker should restart the application from the checkpoint image instead of starting the boot phase normally. We discuss how Docker and DMTCP are integrated to implement these two operations in Section IV-A.

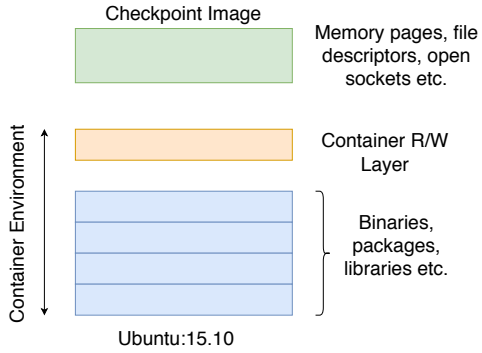


Fig. 1. Contents of a container environment and a DMTCP checkpoint image.

B. Sharing container environments

The components of a container include all the binaries, packages, libraries and application data as depicted in Figure 1. The read-only Docker image layers contain all the necessary files to start the application such as binaries, packages and shared libraries. All the modified files from the image layers, including application data, are stored in the container read-/write layer. During checkpoint, DMTCP captures the memory pages, file descriptors and open sockets of the application, which contribute to the state of the application.

During restart, DMTCP reads the checkpoint image and restarts the application from the checkpoint state. It is usual that at run time an application may access the application data and shared libraries, which are stored in the container environment. Therefore, the container environment captured during the checkpoint must be recreated before the application is restored. We discuss in Section IV-C how container environments (Docker image layers and container read/write layers) are captured during checkpoint and shared among the servers of a PoP.

C. Sharing the checkpoint images

Fog applications are expected to deploy frequently across several servers of a PoP. It is likely that a checkpoint image and its associated container environment will be accessed by several servers when the same container is being deployed repeatedly. Therefore, an efficient mechanism is required to share the checkpoint images and container environments across the servers of a PoP. We discuss how we address this issue in Section IV-C.

IV. System Design

Figure 2 illustrates the Docker container deployment system with DMTCP checkpoint/restart. In this design, Docker uses DMTCP to checkpoint an application after the container boot phase. Later it can deploy any number of containers from the checkpoint image, which eliminates the boot phase. This system relies on two components: (1) a DMTCP lightweight container, which is deployed in all the servers of a PoP; and (2) Ceph RADOS Block Devices (RBD), which use the Ceph distributed storage to share container environments and checkpoint images across the servers of a PoP.

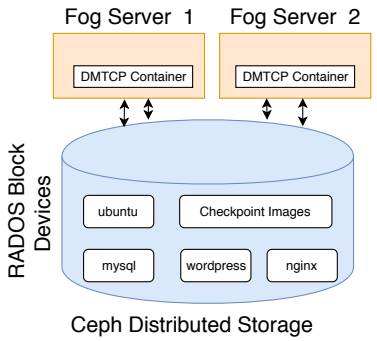


Fig. 2. Container deployment with DMTCP.

A. DMTCP lightweight container

Docker implements many APIs, which handle various functionalities of a container life cycle. For example, `docker ps` shows all the containers currently running in the host machine. However, Docker does not provide any API to integrate external tools such as DMTCP, which can checkpoint/restart applications, while DMTCP itself runs inside the container.

We therefore create a lightweight container image that contains the DMTCP binaries and all the required libraries to perform checkpoint and restart. This allows us to avoid having to modify the Docker source code. Another advantage of the lightweight container is that the same DMTCP container can be used multiple times to checkpoint and restore different applications. Finally, the lightweight container is easy to maintain and distribute across the servers of a PoP with the help of a registry server.

B. Ceph block device

To correctly restart containers from their snapshot, we need to preserve the full application environment in the container file systems. We next explain how application environments are stored, managed and made available to the lightweight container's file system.

1) Snapshotting container environments: When checkpointing a container with DMTCP, Docker needs to save the container's read/write layer as part of the application's environment. This is easily implemented using the `docker commit` command which creates a new image by adding the container's read/write layer on top of the image [15]. The resulting image thus contains the container environment and can be shared across multiple servers through the registry.

Although this simple mechanism can preserve and share a container's environment, it also presents a number of challenges: (1) Docker momentarily pauses the container during `docker commit`, which generates application downtime [9]; (2) this operation is not transparent to the application, as the commit operation has to be performed externally; (3) in order to distribute the container's environment, the image has to be pushed first to a registry server and then pulled multiple times onto different nodes. Both operations take significant network resources and require one to store the container environment redundantly in multiple servers. For these reasons, we propose to share the container environment across all servers of a PoP using the Ceph distributed storage system [28].

2) Ceph distributed storage: Ceph is a highly-scalable distributed storage system that can provide block-level storage [13]. Ceph RADOS Block Devices (RBD) store fix-sized blocks of data (for example 1 MB blocks). The block devices are thin-provisioned and resizable, and the stored data can be striped over multiple object storage drives (OSD). Ceph provides a kernel module and the *librbd* library to create, manage and control the block devices [22], [30].

Ceph provides many features to manipulate block devices, such as snapshot and clone [31]. This feature allows users to create multiple children of a block device using Copy-on-Write (CoW). Once a block device has been snapshot, it becomes read-only. Multiple block devices can then be cloned from the snapshot. All further updates are written in the cloned block devices following CoW. Ceph implements CoW at block-level granularity.

3) Storing container environments in Ceph: We propose to store the full container environment in the Ceph distributed storage system. A container environment contains the Docker image's read-only layers and the container read/write layer. Instead of keeping the Docker images and the container read/write layers in the local file systems, which is the standard option, we propose to store the environment in the shared Ceph RADOS Block Devices (RBD). To implement this, a pool of RBDs is created and an RBD is assigned to each container environment.

Storing container environments in Ceph distributed storage brings many advantages: (1) multiple Docker servers running in different nodes can share the same environment, which enhances storage efficiency; (2) the container environment can be shared without the use of a registry server; (3) the system can efficiently snapshot a container environment and share across servers of a PoP; and (4) Ceph performs CoW at block-level granularity, which is more efficient in terms of performance and space utilization compared to file-based CoW [34].

4) Container environment layering: Although the Ceph RBD system allows us to share container environments across many co-located servers, we also need to make it available to the DMTCP lightweight container. We therefore need to configure the system such that the Ceph block storage is accessible from the DMTCP lightweight containers.

Ceph supports a feature to save the state of a block device using snapshotting [31]. The outcome of a snapshot operation is that the block device becomes read-only. Ceph can then create multiple cloned block devices from any snapshot. All further modifications over the snapshot are performed in the cloned devices following CoW. The obvious advantage of snapshot and clone is that it can re-use the snapshot block devices for multiple purposes. Figure 3 illustrates the working principle of snapshot and clone of a Ceph RBD. The left-side RBD becomes read-only once the snapshot is done, and it is generally referred to as the parent. The right-side RBD is

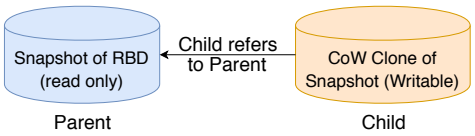


Fig. 3. Snapshot and clone feature of Ceph.

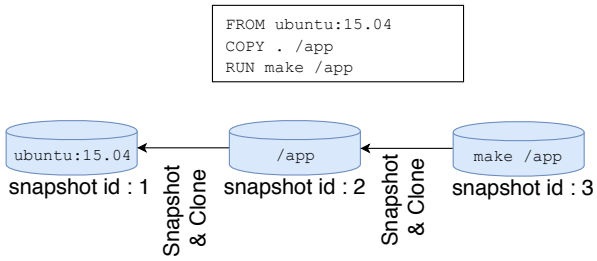


Fig. 4. Container environment layering with snapshot and clone.

created with a clone operation from the parent and it is referred to as the child. Here, all the modifications in the snapshot are done in the child RBD.

We use the Ceph snapshot-and-clone feature to implement container environment layering. The container environment layers are read-only except for the top layer, which is read-/write. This is similar to Docker image layering, which allows one to re-use image layers, where all the layers remain read-only except the top layer, where the file system updates are performed. To implement the layering, we assign a snapshot ID to each new snapshot created from a RBD. This snapshot ID is used to create multiple cloned RBDs where the modification of the snapshot can be performed.

Figure 4 shows a Dockerfile that contains three instructions. The first instruction creates a Ceph RBD, copies the contents of Ubuntu:15.04 in the Ceph RBD, and takes a snapshot. The second instruction creates a clone RBD from the parent snapshot, and copies the current directory to the clone block device. The third instruction compiles the content of a specific directory. Finally, a snapshot ID is generated for the new container environment.

5) Making container environments available to the DMTCP lightweight container: Our design stores the container environments in Ceph RBDs, which helps to share the environments across co-located servers of a PoP. However, in order to enable DMTCP to checkpoint/restart an application, the DMTCP lightweight container must access the container environment. Figure 5 illustrates the operations that the system must perform to make the container environment fully available in the DMTCP lightweight container's file system.

1) **Creating a Clone RBD:** we use the snapshot ID of the container environment to create a clone RBD. The clone RBD is therefore a read/write layer where any update in the container environment will be performed.
2) **Mapping the RBD:** This operation registers the clone RBD to the local kernel block device. With this operation, the kernel assigns a block device identifier such as `/dev/rbd*` to the RBD. The map operation (e.g., `rbd map`) is implemented in the librbd Ceph kernel module.
3) **Mounting the RBD:** When creating a container with `docker run`, the system needs to assign two flags in the container configuration: the `--device` flag to enable the device inside the container, and the `--privileged` flag to mount the block device inside the container file system. The container now can access the block device locally. It then mounts the block device in the local directory of the file system using the standard *mount* command.

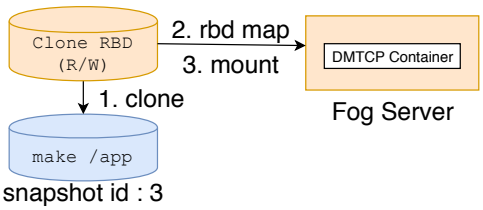


Fig. 5. How a block device is mounted in the container file system.

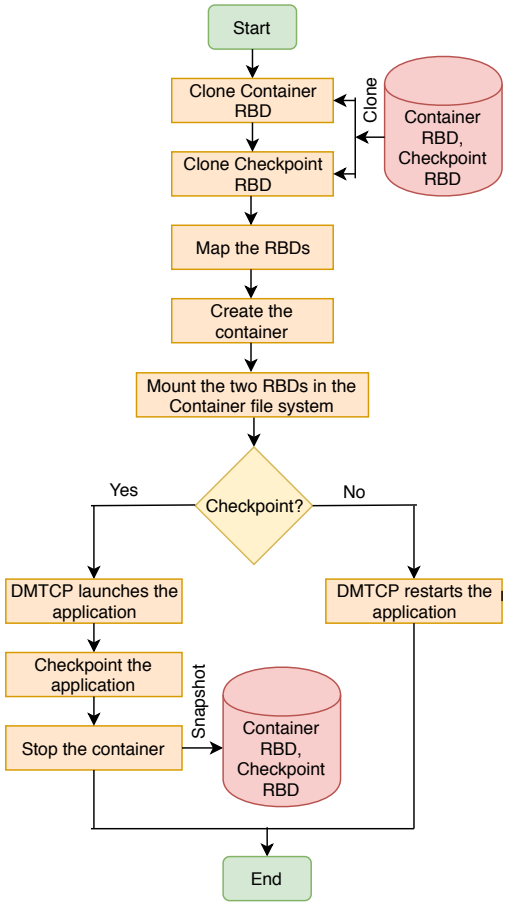


Fig. 6. Flowchart of the Docker container deployment with DMTCP.

C. Container deployment with checkpoint/restart

Figure 6 shows the system design flowchart. Our system relies on two Ceph RBDs to store the container environment

and the checkpoint image respectively. The system first creates the two RBDs by cloning from their respective snapshots: (1) the container RBD is created from the snapshot of the container environment. This RBD is used to store the changes in the container environment; and (2) the checkpoint RBD is cloned from the snapshot of the checkpoint RBD.

Docker then uses the DMTCP lightweight image to deploy the container. When creating the container, the system follows the procedure described in IV-B5 to mount the two cloned RBDs in the local container file system. Once the container is created, the system then launches the application based on the type of deployment:

- Creating a checkpoint image: on the first deployment of a container within a PoP we create a snapshot of the checkpoint image and the container environment. The system uses DMTCP to launch the application and boot the container. When the boot phase is completed, DMTCP checkpoints the application and stops the container. The resulting DMTCP checkpoint image is saved in the checkpoint RBD. Finally, the checkpoint RBD and container RBD are snapshotted. The two snapshot RBDs can be shared across co-located servers of a PoP. In every subsequent deployment, those two RBDs are used to deploy the container.
- Deploying from a checkpoint image: in this case, the system deploys the application from a checkpoint image. The cloned checkpoint RBD and container RBD already have booted the checkpoint image and the container environment respectively. Therefore, the system uses DMTCP to restart the application from the checkpoint image.

V. Evaluation

We evaluate the performance of our container deployment design in a distributed fog environment. The experimental testbed is composed of five virtual machines representing the servers of a PoP. These VMs are created using KVM on a Dell PowerEdge R430 server with two Intel Xeon E5-2620 v4 processors running at 2.10 GHz, with 8 hyperthreaded cores each, and 64 GB of RAM. Each VM has 2 vCPUs, 1 GB RAM and 32 GB disk, and runs Ubuntu 18.04 server with 4.15.0-47-generic Linux kernel.

Building a Ceph cluster requires at least three Object Storage Daemons (OSD) where the objects are stored, plus one monitor and manger, which is responsible for controlling, monitoring and managing the cluster. We set up the Ceph cluster using Ceph version 12.0.4 over the five servers of the testbed. The cluster has one monitor running in one machine, and five OSDs running in a separate fog node. We also deploy the Ceph client in every machine.

We use Docker version 18.04 to deploy containers throughout the experiments. Finally, we build a DMTCP Docker image based on DMTCP version 2.5.2 [7]. While deploying the containers, we make sure all the systems are idle to avoid any interference from other applications.

We carry out the evaluations by deploying the *Edge-ShareLatex* application [35]. This web-based application allows users to edit latex projects collaboratively, compile them and generate output. It is composed of 14 micro-services dedicated to different tasks. This gives us a set of 14 independent containers with different characteristics to evaluate our system.

TABLE I
Boot phase duration of the evaluated services.

Service	Boot phase time			Snapshot size
	Docker	Proposed	Speedup	
redis	0.1 s	0.1 s	1x	5 MB
real-time	0.4 s	0.4 s	1x	9 MB
updater	0.6 s	0.5 s	1.2x	11 MB
notifications	5.2 s	0.8 s	6.5x	35 MB
chats	5.2 s	0.9 s	5.7x	37 MB
filestore	5.3 s	0.9 s	5.9x	33 MB
spelling	5.7 s	1.0 s	5.7x	38 MB
docstore	6.6 s	0.9 s	7.3x	33 MB
tags	6.9 s	0.8 s	8.6x	34 MB
contacts	7.3 s	1.0 s	7.3x	33 MB
tracker	8.3 s	1.1 s	7.5x	39 MB
Mongo	12.1 s	1.5 s	8x	25 MB
clsi	20.2 s	1.5 s	13.5x	42 MB
web	109.0 s	1.8 s	60x	35 MB

A. Checkpointing overhead

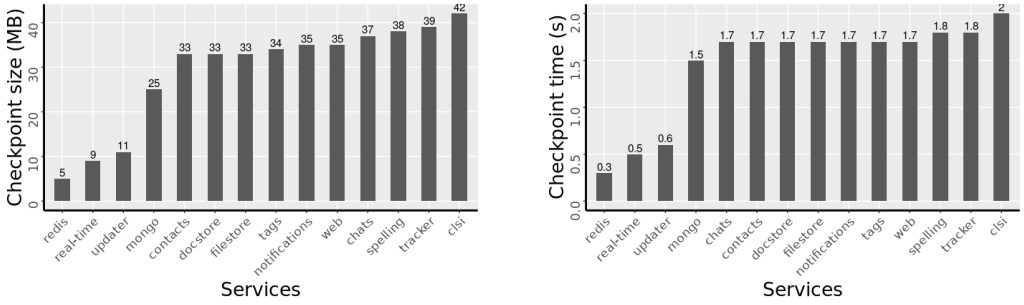
Our system checkpoints containers the first time they are deployed in a PoP. When checkpointing, the system momentarily stops the container and the resulting checkpoint image is stored in the Ceph distributed storage. In this section, we analyze the system overhead while checkpointing the services.

Figure 7(a) depicts the checkpoint image size of the Edge-sharelatex services. The size of the checkpoint images varies from 5 MB to 42 MB depending on the service. We can clearly differentiate the services based on their checkpoint image size: for example, lightweight services such as redis, real-time, updater have image size in the range from 5 MB to 11 MB. The checkpoint image size of the Mongo container is 25 MB. For the other containers that run a script inside the container and for the Mongo database, the checkpoint image size varies from 33 MB to 42 MB.

Figure 7(b) shows the checkpointing time of the services. The checkpointing time varies from 0.3 s to 2.0 s depending on the service. We observe that the checkpointing time is largely proportional to the size of the checkpoint image.

B. Boot phase duration

Table I compares the boot phase duration while deploying with the standard Docker and with container restart. The service boot time with the standard Docker varies from 0.1 s to 109.0 s, whereas with the proposed model, this range is from 0.1 s to 1.8 s. The gains of eliminating the boot phase in the proposed model however depend on the type of the containers and the complexity of their boot phase. For instance, lightweight services such as redis, real-time, updater take a negligible amount of time (less than 1 s) to boot. Deploying such containers with DMTCP takes almost the same amount of time. However, other services such as notifications, chats and filestore take between 5 and 20 s to boot because they deploy a



(a) Checkpoint image size (MB).

(b) Checkpointing time (s).

Fig. 7. Checkpoint image size and checkpointing time of the services.

Mongo database inside the container in addition to their other software. Deploying such services with the proposed model takes only about 1 s, which yields a speedup in the range of 5x to 8x. Finally, the *web* service takes nearly 109 s to boot its container because it compiles some of its scripts before becoming ready to serve end users. In this case, our system delivers 60x improvement. We conclude that the performance gain is largely proportional to the time the container takes to boot. The deployment time from a container snapshot depends on the size of its data rather than the procedure to boot it.

C. Communication within the Ceph cluster

Our system stores the container environments and checkpoint images in Ceph block devices. The system's various Ceph components such as OSDs, monitor and manager running on different nodes constantly communicate with each other to keep the cluster working. To evaluate the impact of the transferred data among the Ceph components, we trace the download and upload network throughput of the cluster nodes.

We use the *nethogs* utility to capture the network throughput for duration of 60 s at 1 s granularity [18]. We capture the network throughput of the machine hosting the monitor+manager, and one which is running an OSD. We did the experiment in three scenarios: first, when the system is in idle condition; and second, when DMTCP is checkpointing the *redis* container; and third, when DMTCP is restarting a container.

Figure 8(a) shows the network throughput of the target machines when the system is in idle state. We observe that both machines (monitor+manager and OSD) exhibit constant low-bandwidth network activities throughout the trace. This may be due to the fact that the monitor+manager periodically communicates with its OSDs to control and monitor the cluster state. The upload throughput of the monitor+manager node is negligible, whereas the download throughput reaches 24 kB/s. The OSD node exhibits both download and upload activities.

Figure 8(b) represents the network throughput of the target machines when the system is checkpointing a container. We observe the same phenomenon except at the time when the system checkpoints the container. During the checkpoint (at t = 6 s), we observe an intense network activity, in particular with the download throughput of the OSD. The download throughput of the OSD goes nearly up to 2900 kB/s. This is due to the fact during the checkpoint, the system writes large chunks of the checkpoint image into that OSD.

TABLE II
THROUGHPUT OF THE HTTP SERVICE.

Concurrency degree	10	100	200	500	1000
Standard (req/s)	3668	3834	4057	4117	4234
Proposed (req/s)	3537	3685	3856	3891	3954
Overhead (%)	3.6%	3.8%	5.0%	5.5%	6.7%

Figure 8(c) shows the network throughput of the target machines when a container is restarting from the checkpoint image. During restart we observe an intense upload throughout in the OSD node. This reflects that the system reads the checkpoint image from the Ceph OSD while restarting the container.

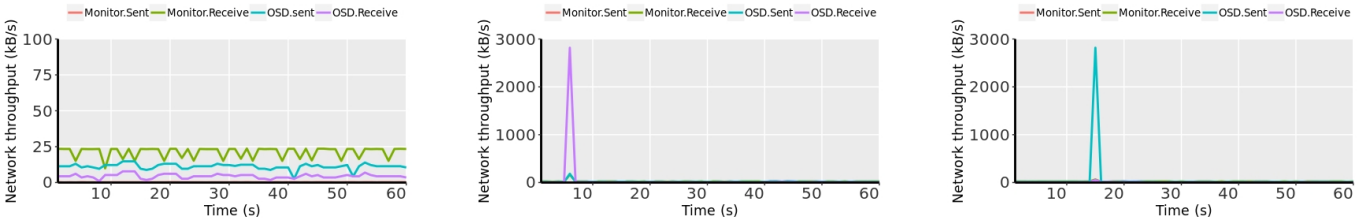
We conclude that Ceph's cluster components perform very little network activity to keep the cluster alive. But we observe intense network activity in the OSD nodes when the system checkpoints and restarts a container.

D. Interference with other applications

Sharing container environments using the Ceph distributed storage clearly has many advantages. However, Docker needs to fetch the application files remotely from the distributed storage, which may in turn impact the performance of other running applications. We therefore study the runtime performance of the already-deployed applications when a new container is being deployed.

We deploy an Apache HTTP server in one cluster node [5]. The server hosts a web page of 5 kB. We then generate an artificial HTTP load from one of the cluster nodes using the standard `ab` benchmarking tool with varying concurrency degrees [6]. This load generates intense I/O activities in the concerned server (as no file system cache was employed). We evaluate the performance interference of concurrent container deployments by measuring the throughput of the HTTP server.

Table II compares the Apache throughput in both scenarios with different concurrency degrees. We observe a small overhead of 3.6% with a concurrency level 10. This is expected due to the fact that the HTTP load generates intensive I/O



(a) System is idle. (b) System is checkpointing. (c) System is restarting.

Fig. 8. Network throughput of the nodes when the system is: (a) idle; (b) checkpointing and (c) restarting.

operations and as a result the proposed system may need to fetch the files from the distributed storage. When we increase the concurrency level, the overhead due to remote access grows gradually up to 6.7% at a concurrency level of 1000. We conclude that the proposed system may exhibit an improved boot phase time with lower overhead. This is due to the distributed storage used to store the container environments.

VI. Conclusion

Booting a Docker container after it has been started requires significant time during the container deployment process and may have an important impact in fog computing environments, since the same container may be repeatedly launched, created and booted. The boot sequence of most containers however always remains the same. This gives us an opportunity to eliminate the boot phase by saving the state of a fully-booted container. In subsequent deployments, the container can be restarted from the saved state, which entirely eliminates the boot phase in the deployment process.

We proposed a container deployment system that uses checkpoint/restart to deploy from a checkpoint image. The system also stores the container environments and checkpoint images in Ceph distributed storage to efficiently share them across the servers of a PoP. Our evaluations show it can yield up to 60x speedup in the container boot phase time depending on the type of container. The overhead of creating checkpoints and storing them across the PoP remain reasonable.

References

[1] X. M. Aguilera et al. Managed containers: A framework for resilient containerized mission critical systems. In *Proc. IEEE CLOUD*, 2018.

[2] A. Ahmed and G. Pierre. Docker container deployment in fog computing infrastructures. In *Proc. IEEE EDGE*, 2018.

[3] A. Ahmed and G. Pierre. Docker image sharing in distributed fog infrastructures. In *Proc. IEEE CloudCom*, 2019.

[4] J. Ansel et al. DMTCP: Transparent checkpointing for cluster computations and the desktop. In *Proc. IEEE IPDPS*, 2009.

[5] Apache Software Foundation. Welcome! - The Apache HTTP Server Project. https://httpd.apache.org/, 2018.

[6] Apache Software Foundation. ab - Apache HTTP server benchmarking tool - Apache HTTP server version 2.4. https://httpd.apache.org/docs/2.4/programs/ab.html, 2019.

[7] K. Arya et al. DMTCP: Distributed multithreaded checkpointing. http://dmtcp.sourceforge.net/, 2019.

[8] P. Bellavista et al. Feasibility of fog computing deployment based on Docker containerization over RaspberryPi. In *Proc. ACM ICDCN*, 2019.

[9] R. Benevides. 10 things to avoid in Docker containers - Red Hat Developer. https://blog.codeship.com/using-docker-commit-to-create-and-change-an-image/, 2019.

[10] L. Bittencourt et al. Mobility-aware application scheduling in fog computing. *IEEE Cloud Computing*, 4(2), 2017.

[11] Checkpointing.org. The home to checkpointing packages. https://checkpointing.org/, 2019.

[12] L. Civolani et al. FogDocker: Start container now, fetch image later. In *Proc. IEEE/ACM UCC*, 2019.

[13] B. Confais et al. Performance analysis of object store systems in a fog/edge computing infrastructures. In *Proc. IEEE CloudCom*, 2016.

[14] Docker, Inc. Docker: Build, ship, and run any app, anywhere. https://www.docker.com/, 2019.

[15] Docker Inc. Docker commit. https://docs.docker.com/engine/reference/commandline/commit/, 2019.

[16] J. Duell et al. Berkeley lab for linux. Technical Report BLCR; 002078WKSTN00, Lawrence Berkeley National Laboratory, 2003.

[17] J. C. Duell. Berkeley lab checkpoint/restart (BLCR) for Linux. https://crd.lbl.gov/departments/computer-science/CLaSS/research/BLCR/, 2019.

[18] A. Engelen. Raboof/nethogs: Linux 'net top' tool. https://github.com/raboof/nethogs, 2017.

[19] T. Harter et al. Slacker: Fast distribution with lazy Docker containers. In *Proc. Usenix FAST*, 2016.

[20] S. Hoque et al. Towards container orchestration in fog computing infrastructures. In *Proc. IEEE COMPSAC*, 2019.

[21] J. Hursey et al. The design and implementation of checkpoint/restart process fault tolerance for Open MPI. In *Proc. IEEE IPDPS*, 2007.

[22] Inktank Storage Inc. Rbd(8): Manage Rados Block Device Images - Linux Man Page. https://linux.die.net/man/8/rbd, 2019.

[23] Kubernetes Authors. Production-grade container orchestration – Kubernetes. https://kubernetes.io, 2019.

[24] S. Nadgowda et al. Comparing scaling methods for linux containers. In *Proc. IEEE IC2E*, 2017.

[25] M. Nardelli et al. Multi-level elastic deployment of containerized applications in geo-distributed environments. In *Proc. IEEE FiCloud*, 2018.

[26] OpenVZ team. CRIU. https://www.criu.org/Main_Page, 2019.

[27] OpenVZ team. Docker external - CRIU. https://criu.org/Docker_External, 2019.

[28] Red Hat, Inc. Ceph. https://ceph.com/, 2019.

[29] Red Hat, Inc. Ceph block device. https://docs.ceph.com/docs/master/rbd/, 2019.

[30] Red Hat, Inc. Librbd (Python). https://docs.ceph.com/docs/jewel/rbd/librbdpy/, 2019.

[31] Red Hat, Inc. Snapshots. https://docs.ceph.com/docs/master/rbd/rbd-snapshot/, 2019.

[32] J. M. Smith et al. *Implementing remote fork() with checkpoint/restart*. Dept of Computer Science, Columbia University, 1987.

[33] C. Song et al. Limits of predictability in human mobility. *Science*, 327, 2010.

[34] V. Tarasov et al. Evaluating Docker storage performance: from workloads to graph drivers. *Cluster Computing*, 2019.

[35] G. Tato et al. ShareLatex on the edge: Evaluation of the hybrid core/edge deployment of a microservices-based application. In *Proc. MECC*, 2018.

[36] C. Zheng et al. Wharf: Sharing docker images in a distributed file system. In *Proc. ACM SOCC*, 2018.

[37] H. Zhong and J. Nieh. Crak: Linux checkpoint/restart as a kernel module. Technical Report CUCS-014-01, Dept of Computer Science, Columbia University, 2001.