FogDocker: Start Container Now, Fetch Image Later

Lorenzo Civolani, Guillaume Pierre, Paolo Bellavista

FogDocker: Start Container Now, Fetch Image Later

Lorenzo Civolani
Univ Rennes, Inria, CNRS, IRISA
University of Bologna

Guillaume Pierre
Univ Rennes, Inria, CNRS, IRISA

Paolo Bellavista
University of Bologna

ABSTRACT

Slow software deployment is an important issue in environments such as fog computing where this operation lies in the critical path of providing online services to the end users. The problem is even worse when the virtualized resources are made of modest machines such as single-board computers. This paper leverages the observation that, although Docker images are often very large, only a small fraction of their content is actually accessed by the containers during startup. We therefore propose to reorganize container images and download only the strictly necessary files before starting a container. The remaining image contents can then be downloaded asynchronously while the container is already running. Our performance evaluations show that FogDocker reduces container deployment times in the order of 3-5x on single-board computers and 2-3x on powerful servers, while incurring low runtime overhead and maintaining correctness even in the case a container accesses a file which was not downloaded yet.

KEYWORDS

Fog computing; Docker; containers.

1 INTRODUCTION

The popularity of container-based virtualized infrastructures makes it increasingly attractive to dynamically deploy personalized applications upon the (often implicit) request of a single end user [3]. In such scenarios, the application deployment process cannot be considered as a one-time operation that does not affect the end user's Quality of Experience. Rather, it becomes an integral part of the critical path toward providing the expected service to its end users. To ensure a satisfactory user experience it is therefore important to reduce application deployment times as much as possible.

Docker is by far the most popular container management system [4], both in cloud [24] and fog [18] computing platforms. Its popularity is largely due to its lightweightness compared to virtual machines and its excellent integration with DevOps processes. However, depending on the available hardware and the nature of the deployed application, a user may need to wait up to a few minutes until an application is ready to be used [1]. This is particularly true in fog computing infrastructures which often rely on weak hardware such as single-board computers [2, 15, 27].

When Docker deploys a container, if the required application code and data are not already locally available, it is first necessary to *pull* the necessary content from a public or private image repository. This pull operation of the so-called *container image* constitutes the main part of the container deployment time.

To reduce the container deployment time we exploit the observation that Docker containers usually access only about 6% of their image content before being able to start useful work [17]. We therefore propose to reorganize the deployment process such that a container can be started immediately after its most essential files have been downloaded. The remaining image contents can then be downloaded asynchronously while the container has already started.

Implementing this simple idea requires one to address a number of difficult challenges. First, we need to profile a container's execution, identify the set of essential files which must be installed before starting the container, and build a special image layer with these essential files. Then, we must modify the Docker deployment process to start the container immediately after a new "base" layer has been downloaded, while the other image layers are still being retrieved asynchronously. Finally, we need to maintain application execution correctness even in the case where a started container tries to access a file which has not yet been downloaded.

We show that FogDocker reduces by an order of magnitude the amount of data which needs to be downloaded before starting the container. The pull time, measured as the duration until the container can start, is reduced by a factor between 2 (on powerful cloud computing servers) and 5 (on single-board fog computing servers). When the network bandwidth is limited, as is common in fog computing scenarios, the relative improvement grows up to 18x. At the same time, the runtime overhead remains low, between 1 and 8 ms upon every call to the `open()` or `fopen()` functions. The fact that FogDocker succeeds in such diverse environments demonstrates the general applicability of our approach in environments where container images are deployed frequently. In fog computing, for instance, the geographical properties of the workload create strong constraints on the placement of containers, and workload dynamicity may require frequent re-deployments.

This paper is organized as follows. Section 2 introduces the background and related work. Section 3 details the Overlay

file system. Section 4 discusses our system design. Finally, Section 5 evaluates this work and Section 6 concludes.

2 BACKGROUND

2.1 Docker

Containers are standardized units of software that tie an application with the full environment it needs for execution. A container virtualizes a group of processes that share the hosts's OS kernel but operate inside an isolated environment from other processes and containers.

Although containers preexisted Docker, Docker is by far the most popular framework to build, package, and run software inside containers [7].

A central concept of the Docker framework is the *image*, a read-only template that is used as a basis to instantiate containers. Images provide a stable and reliable execution environment for the enclosed application. An image is a combination of: (1) a file system that will be exposed to future containers, usually including a full Linux distribution, software dependencies and configuration files; (2) configuration data such as environment variables; and (3) a start-up command to be executed at launch time.

A Docker image is built using a *Dockerfile* which specifies the sequence of actions that must be performed during the image creation process. Some crucial instructions are: `FROM`, to define the base image for the subsequent commands; `COPY`, to add new files to the container file system; `ENV`, to set the value of an environment variable; and `CMD`, to define the default execution command.

The popularity of Docker is largely due to its incremental approach for building container images: instead of requiring developers to provide a full monolithic image of each container's file system (as is commonly done with VMs), a new image can be defined by specializing an existing one thanks to the `FROM` instruction at the opening of a Dockerfile. A developer only needs to provide incremental changes, which at build-time create one or more additional *layers* containing the files that must be added, modified, or deleted with respect to the original image. As a result, Docker container images are made of a number of layers stacked on top of each other.

Container image layers are read-only so a running container cannot modify their content. Instead, when a container starts, a new writable layer is dynamically added on top of the layer stack. The purpose of this *container layer* is to keep track of all file system modifications made by the running processes using a copy-on-write (CoW) policy [28]. Although a container's file system is composed of a stack of multiple layers, this structure is not exposed to the container processes. File system layers remain physically separate on disk, but a "union mount" file system such as OverlayFS is used to unify all the layers and expose a single homogeneous file system to the container processes [25].

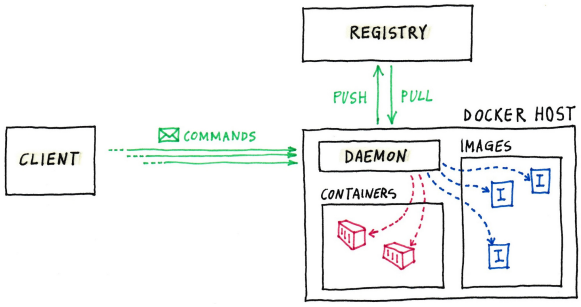


Figure 1: Docker's architecture.

The architecture of Docker shown in Figure 1 comprises three elements. The *daemon* performs the actual management of images and containers. It listens to requests issued by the *client* and performs the requested action. Finally, the *registry* stores a copy of all the container image layers, enabling image sharing between multiple daemons. By default daemons make use of the public "Docker hub" registry [8].

The *pull* operation is performed upon every container deployment to download the necessary layers. Internally to the Docker daemon, a `LayerDownloadManager` object is in charge of identifying the image layers which need to be retrieved, downloading them, and registering them in the local system. The pull operation creates one separate thread per layer to download, decompress and extract them. When all the required image layers have been extracted, the pull operation terminates and Docker starts the container from a merged view of the concerned layers.

2.2 Related work

Optimizing Docker container deployment is an important problem, in particular in fog computing where container deployment is a frequent operation that often lies in the critical path of providing online services to end users [3].

The idea of prioritizing the download of files or blocks that are necessary for the execution of an application, and delaying the others until they are needed, is an old and well-studied idea [29]. Our work however differs from lazy loading and demand paging in that the entire container image gets downloaded it its entirety shortly after the container has started, thus avoiding the cost of page faults during the container's lifetime.

One approach to accelerate the deployment of Docker containers focuses on the image registry to ensure it delivers images as quickly and efficiently as possible. CoMICon is a distributed registry which distributes the image layers across multiple nodes to speed-up deployment time and increase availability [20]. Similarly, Nitro uses deduplication

and network-aware data transfer strategies to reduce the image transfer time over wide-area networks [6]. Although these works result in faster image download, they do not address the performance bottlenecks in the Docker daemon itself and are therefore orthogonal to our work.

Another approach consists of shrinking the container image by removing the useless files from the image [14]. Similar ideas were also used in the context of disconnected operation [19]. However, this operation modifies the Docker semantic of relying on a complete and well-defined operating system. We prefer keeping the full image content including rarely-accessed yet useful files such as as tools for debugging, performance profiling etc.

Wharf proposes to share the caches of multiple Docker servers in a distributed file system to reduce storage utilization and the number of redundant image retrievals [30]. However, it does not change the image pull process and therefore has no performance impact in the case image layers need to be pulled before starting the container.

The only work which exploits the fact that a container accesses only a fraction of its image content before producing useful work is Slacker [17]. Slacker proposes to store the Docker images in a shared NFS file server. With the use of a specialized storage driver in every Docker daemon, only the relevant parts of each image need to be downloaded from the NFS server to the Docker daemon upon every container start operation. However, this assumes that all relevant images are already stored in the NFS file server, and therefore does not impact the performance of the `docker pull` operation.

Finally, Docker-pi proposes to reorganize the download and extraction of image layers in the Docker server itself to better exploit hardware-level parallelism [1]. Although this technique speeds up the image pull process, it still relies on downloading the full image before starting the container, which delays the container startup operation while many non-critical files are being downloaded and extracted to disk. We rather aim at starting the requested container as early as possible and allowing it to provide its intended functionality while the non-critical parts of its image are still being pulled.

3 UNDERSTANDING OVERLAYFS

One of the main challenges in this work is to precisely understand the possibilities and limits of Docker's layered file system structure. In particular, asynchronously downloading and installing content in the lower file system layers after the container has started implies that these layers cannot be considered read-only, which deviates from the canonical usage of the file system layers.

In Docker, all details about the way the layers' content is stored and managed on the disk of the host machine are handled by the storage driver. A number of storage drivers

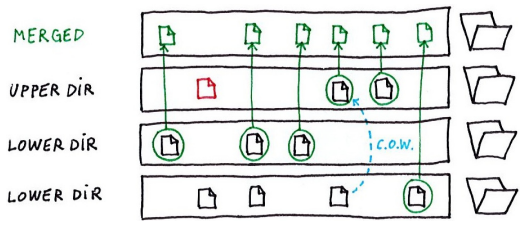


Figure 2: An instance of OverlayFS.

are available to support various usage scenarios. In this paper we focus on `overlay2`, which is the default driver for modern versions of Docker [9].

The `overlay2` storage driver makes use of the Overlay file system [25] to implement Docker's layered architecture for images and containers. OverlayFS sits on top of a host file system with the aim to provide a combined view of contents belonging to multiple directories. An instance of OverlayFS, illustrated in Figure 2, contains the following elements:

- A set of immutable read-only layers, called **lowerdir**, containing the objects that must be joined.
- A writable layer called the **upperdir**, used to keep track of the changes applied through the merged view.
- The **merged** directory, providing a unified view of the lowerdirs and the upperdir. Read operations are redirected to the actual file in the lower layer; write operations are handled with a copy-on-write policy: when a container modifies a file system object, OverlayFS clones it into the upper directory, and modifies the copy.

The `overlay2` storage driver makes direct use of the OverlayFS directories to implement the layered structure of the containers. In particular, when a container is launched, a new instance of the virtual file system is prepared: the directories storing the content of the underlying image layers are registered as lowerdirs; the container layer is mapped to a new dedicated directory, registered as upperdir, which records all the FS updates issued by the container while preserving the integrity of the image layers; finally, the merged directory becomes the root of the file system tree of the container.

The separation between read-only lowerdirs and the writable upperdir provides a natural way for Docker to reuse the read-only layers across multiple containers while keeping each container's file system updates separate. In our case, however, we plan to continue inserting content in the lowerdirs after container execution has started, which temporarily deviates from their self-proclaimed read-only characteristics.

OverlayFS officially does not support run-time modifications of the underlying layers. Its documentation claims that *"changes to the underlying file systems while part of a mounted OverlayFS are not allowed. If the underlying file system is*

changed, the behavior of the Overlay is undefined, though it will not result in a crash or deadlock" [25]. After performing an extensive set of experiments, we however found a different answer which allows us to define the behavior of OverlayFS when the lowerdirs are modified. The virtual file system actually supports certain live changes of the layers, under the following conditions:

- Files are only added into the layers. Renaming, updating or deleting files interferes with the caching mechanisms of OverlayFS and can produce incorrect results.
- Directories are treated differently than files, and the dynamic addition of new directories is not always reflected in the merged directory. As a consequence it is important that the full directory tree is present across all the layers at the time a container is created.
- Whenever a lookup is requested in a merged directory, the operation is redirected to each underlying directory and the results are combined and cached in the appropriate *dentry* ("directory entry cache" [16]) of the merged directory. For this reason, a new file inserted into already-existing lower directory would not appear in the merged view. We therefore need to drop the dentry cache of the underlying file system when the run-time deployment of a lower layer is complete.

We therefore conclude that adding content in the lowerdirs after the container has started is perfectly possible. The main constraint is that the FogDocker "base layer" must contain not only the essential files, but also the whole directory structure of the original file system image. When the base layer has been deployed, the Docker daemon must then clone that directory tree across all the empty layers before mounting the virtual file system and launching the container. In this way, the content that gets injected at run time into the lower layers appears in the merged overall view and is correctly detected by the executing container. Note that, although FogDocker leverages an undocumented feature of OverlayFS, private discussions with the OverlayFS developers indicated that future versions of the file system will remain at least as consistent as in the current one.

4 SYSTEM DESIGN

The objective of this work is to substantially reduce the time spent during the deployment of a container until the container is ready to perform useful functionality. We present FogDocker, which leverages the fact that a container accesses only a small fraction of all its files during its startup phase. As illustrated in Figure 3, the standard Docker policy is to fully download all the container layers before starting the container. The key idea in our work is to identify the essential files, create an additional "base image layer" containing them, and start the container immediately after the base

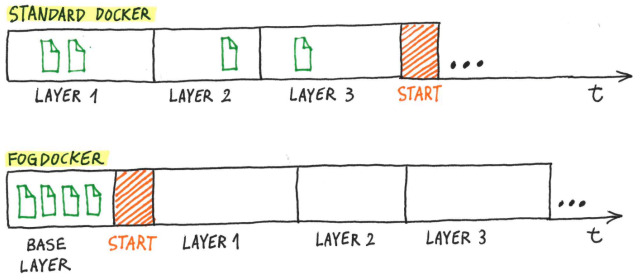


Figure 3: FogDocker's container start operation.

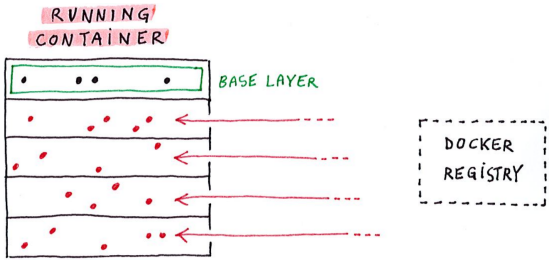


Figure 4: Filling image layers at run time.

layer content is locally available. The container can thus start very early in the deployment process, while the other image layers containing less important files are still being downloaded and installed.

When the Docker daemon initiates the pull operation (for instance, upon a `docker pull` or a `docker run` command), the base layer is downloaded first from the registry. The other layers are created in the OverlayFS virtual file system but kept empty. At this point, the container execution can already begin. As shown in Figure 4, while the container has started running, the Docker daemon proceeds in background with the deployment of the rest of the image: the remaining archives are downloaded, extracted, and the empty layers are filled with their respective content.

To realize a concrete implementation on the basis of this simple idea, we address the following questions in turn:

(1) How can we identify the essential files needed for the execution of a container? (§ 4.1);
(2) Which precise content should be included in the new base layer? (§ 4.2);
(3) What is the best way to integrate the base layer in the new custom image? (§ 4.3);
(4) How should we reorganize the Docker deployment process to follow this new policy? (§ 4.4);
(5) How can we maintain a correct container execution, even in the case where it would try to access a file that is not present yet? (§ 4.5).

4.1 Profiling the container execution

The first challenge that must be addressed to design our solution is identifying the set of essential files that are accessed by a container during the first few dozen seconds of its execution. Naturally, precise file identification is essential for the performance of FogDocker.

The authors of [17] carried out an extensive study based on 57 highly popular images from the Docker Hub. By leveraging block-level tracing they identified that, on average, only 6.4% of the image content is accessed during the container's startup phase.

To monitor the activity of the containerized process, we make use of the *fanotify API* provided by the Linux kernel [21]. This interface creates a system of interception and notification for events that occur in the file system, offering a way to detect the file system objects that a Docker container accesses during its execution. Instead of building a custom file access monitoring program we exploit the open-source `fatrace` (file access trace) tool [22]. fatrace makes use of the fanotify API and reports which files are being accessed by any running process.

We profile the container from within the container itself. To launch fatrace inside the container, the executable must be included into its file system. A simple option consists of keeping fatrace in an appropriate folder on the host machine and make its content accessible from within the container by using a *bind mount* [10]; we are thus able to provide the isolated process with the access to the fatrace executable without the need to create a further Docker image.

When running the container, our solution first launches fatrace in the background and then executes the actual application. Only at that point, the simulation starts. For example, to exercise a Web server container one could issue a number of HTTP requests to retrieve the content of a sample website. In general, we request the application developer — or the user who wants fast deployment for her containers — to provide a script to exercise the container in a way that mirrors a typical usage scenario as closely as possible. For more complex applications, profiling may be integrated in a DevOps workflow and capture the start-up behavior of every subsequent application version. We defer this topic to future work.

After executing the container and exercising it using the developer's script we can simply retrieve the log produced by the tracing mechanism, and extract the list of files that were accessed by the containerized application.

4.2 Building the base layer

As discussed in Section 2, Docker image content is distributed across multiple layers which are retrieved over the network as compressed archives, and then extracted and installed locally by the daemon. We exploit this organization to concentrate the essential files into an additional layer which is then joined to the original stack. As a result, this new version of the image contains an additional *"base layer,"* providing the minimal file system capable of supporting the first few stages of execution of the application.

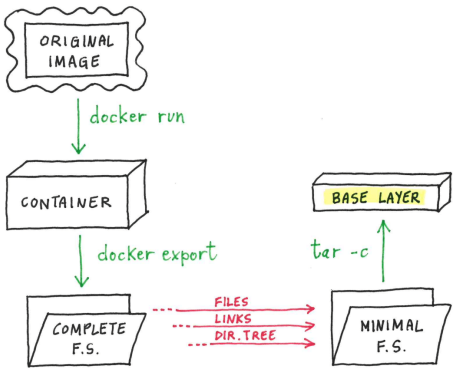


Figure 5: Steps to build the base layer.

As shown in Figure 5, building the base layer requires several operations. First, we launch a sample container from the original image and extract a copy of its entire (merged) file system into an empty folder through the `docker export` command. The information derived from the profiling discussed in the previous section dictates what must be included in the base layer. We copy such content into a new directory which will eventually contain the minimal file system that is needed for the container to start useful work.

Files. Obviously, all the files that have proven to be necessary for execution must be present in the base layer. We use the `rsync` utility [26] to copy files from the source folder, holding the container original file system, to a destination one representing the minimal version of it. The rsync command can be configured to preserve all the files' metadata.

Links. Copying just the files is not enough for the thin version of the container to work. When the monitored application opens a file through a link, only the file itself is reported by the fanotify API and therefore copied into the base layer. As a result, the application inside the thin version of its container would be unable to find what it needs to run. We must therefore also include all the links that lead (directly or indirectly) to any of the files included in the base layer.

Directory structure. As discussed in Section 3, the OverlayFS virtual file system can support the dynamic addition of files in the lowerdir after the mount only if the directory structure is present across every lowerdir. We therefore need to include the full directory structure of the container image

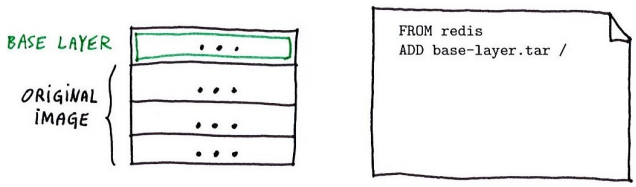


Figure 6: The custom container image and its associated Dockerfile.

in the base layer. This copy can again be realized with the `rsync` utility.

Wrapper library. As will be discussed in Section 4.5, we include a small wrapper library that intercepts calls to the `open()` and `fopen()` system calls in order to handle the case where a container accesses an object that was not downloaded yet. We also include an additional file with the list of all the file names from any of the lowerdir layers.

Once these copy operations are completed, the base layer contains all the file system objects that are necessary to let the container start and operate while the remaining files are asynchronously being downloaded and installed. Note that these objects are copied rather than moved from the main container image, thus creating a certain amount of redundancy. However, this has the advantage of keeping the original layers untouched and therefore potentially reusable in multiple independent images. Layer reusability is a key design principle in Docker that we aim to preserve in Fog-Docker as well.

4.3 Preparing the custom image

We now need to create a new custom image which integrates the original container image with the new base layer. The layered nature of the Docker images naturally suggests a possible organization for the custom image, as illustrated in Figure 6: we can place the base layer on top of the original layer stack.

Writing the Dockerfile of the new image is straightforward. The procedure boils down to the only instructions `FROM` and `ADD` (see Figure 6). Since the image is not built from scratch, it is not even necessary to specify again the procedure for the container start-up with a `CMD` instruction: this information will be automatically inherited from the parent's settings. Neither do we have to locally re-build the image from scratch, because a ready package can be pulled directly from a public repository.

4.4 Re-engineering the pull operation

A central part of this project concerns the idea of starting a container after just a part of its image has been installed. It is necessary to reengineer the image pull operation in order to deploy the topmost layer (i.e. the base layer) first and let the execution start right after. To do so, we implemented a number of modifications in the Docker source code. We discuss them in turn.

Topmost layer download only. The first modification is to instruct the Docker daemon to retrieve and install the topmost image layer first, whereas the others layers are kept empty. The topmost base layer contains the essential files that allow a container to start executing. Because the base layer is much smaller than the entire container image, we expect to cut the amount of data that must be transferred, extracted and installed before starting the container, thereby reducing the delay before being able to start the container.

As discussed in Section 2, the `LayerDownloadManager` object is in charge of retrieving the image layers. In particular, its `Download` method is a blocking function that ensures that all the requested layers are present in the local store. We updated this `Download` method to recognize whether the layer under consideration is the topmost base layer or a regular one. It then passes an extra parameter holding this information to the respective threads it launches to handle the retrieval of each image layer.

When the base layer is being downloaded, the operation proceeds normally, since its content is required for the execution. Conversely, when dealing with regular layers, the thread skips the download of the respective tarballs and simply replaces them with empty ones. In this way, the daemon creates and configures the appropriate image data structure and the container can in principle be immediately started.

Disabling integrity control. Unfortunately, deactivating the retrieval of a layer content is prone to generate runtime errors. Docker verifies the integrity of all layers by computing a hash of their content and comparing it with metadata present in the image manifest. Besides the base layer, which gets downloaded normally, all other layers remain empty and therefore are most likely to fail the integrity test thus aborting the container deployment. We address this issue by disabling the integrity test: we override the calculated hash with the expected value.

An obvious drawback of this solution is that corrupt layers cannot be reliably detected any more. We plan in future work to reinstate integrity checks at the end of the asynchronous layer download phase.

Cloning the directory structure. Once the minimal image has been deployed, the complete directory tree must be cloned from the base layer into the lower ones. This operation must be completed before the daemon mounts the layers together into a unified view through a new instance of OverlayFS. In this way, run-time changes in the lower

layers, taking place while their content is being retrieved in the background, will be made visible in the unified directory serving as the root of the container's file system.

To do so, once again we modify the `Download` method of the `LayerDownloadManager`. Once the threads which download the base layer and create the empty ones have finished executing, we retrieve the paths in the host file system where each layer has been stored. We then iterate over the lower layers and clone the directory tree of the base layer inside each of them by using the `rsync` utility.

Asynchronously downloading the rest of the image. Once the base layer has been downloaded, extracted and installed, and its directory tree has been replicated across all the underlying empty layers, the pull operation can return and the execution of the container can begin. However, before returning, it is necessary to schedule the asynchronous download and installation of the other layers. Since the directory tree is already present in all the layers, the newly-added content will gradually appear in the file system of the running container.

The Go programming language in which Docker is implemented offers a feature called *goroutine* which allows to execute code in the background. Launching goroutines is simple: one just needs to add the keyword `go` in front of a function invocation. This feature helps implementing concurrent applications in a straightforward way.

In FogDocker we use goroutines to schedule the asynchronous download and installation of the original image layers. The code is set in the same loop that we described in the previous step of our solution: for each layer involved by the iteration, in addition to copying the directory tree, we also set up a goroutine with the aim of asynchronously downloading, extracting and installing the layer data in the appropriate directory.

4.5 Handling early access to delayed files

Although the base layer contains all the files that are necessary for a container to start for the first few dozen seconds, one cannot exclude the possibility that during this time the container tries to access a file which has not been installed yet. For example, the user of a redis database may request an unexpected operation needing a resource which was not triggered by the script used to profile the container. Without special measures, in these cases the file access operation would fail with a "file not found" error, making the container's execution incorrect.

To maintain correctness even in this scenario, we transparently intercept all the file-system calls such as `open` and `fopen` issued by the container: if the requested file is part of the full container image but it not been downloaded yet, the call gets blocked until the requested file becomes available.

```
int open(const char *file, int flags) {
  if (!exists(file) && delayed(file)) {
    wait_avail(file);
  }
  if (!real_open) {
    real_open = dlsym(RTLD_NEXT, "open");
  }
  return real_open(file, flags);
}
```

Figure 7: Wrapper open() function to delay access in case the requested file is not present yet.

In the worst case the container gets blocked until the full image is installed. However, assuming that this event is rare thanks to an accurate profiling, in most cases the container will start much faster than regular Docker.

Whenever the container attempts to open a file, we need to distinguish three possible conditions:

- If the requested file is already present, then we want to allow the `open()` and subsequent operations on that file to execute normally;
- If the requested file is not present but we know it will be part of the container image after all the layers have been downloaded, then we need to block the system call until the requested file has arrived locally;
- Finally, if the requested file is not present nor does it belong in the container image, then we want to let the call execute and eventually fail.

To transparently intercept system calls, we use a technique inspired by Cooperman and Rieker [5, 23]. We created a shared library which redefines the `open()`, `fopen()`, and similar functions from the C library with exactly the same signature as the original functions. Thanks to the `LD_PRELOAD` environment variable we can instruct the operating system to load our custom library before any other dynamic library, including the original C library. Any call to one of these functions therefore executes our wrapper function rather than the actual C-library function.

Figure 7 shows a simplified extract from our wrapper library. The `exists()` function verifies the existence of a file. If the file exists, then no waiting is needed and we can call the "real" `open()` function (found thanks to a call to `dlsym()`). Otherwise, the `delayed()` function determines whether the file belongs to the list of objects that are supposed to be downloaded asynchronously at run time. This list is created when building the base layer, and included at a well-known path within the base layer. If the required file does not exist because it is yet to be deployed, a call to `wait_avail()` makes the process iteratively wait for a fixed amount of time and

then recheck if the file has arrived; when the file becomes available, the procedure lets the call proceed.

This technique allows us to remain totally transparent to the application running inside the container. From the container perspective, `open()`, `fopen()` and similar functions are regular system calls that simply occasionally take longer than usual to complete.

Statically-linked executables. We assume that every executable within the container is dynamically linked with the C library. For statically-linked executables a one may either replace them with dynamically-linked alternatives, or recompile them statically with our wrapper library.

Concurrent container deployments. The wrapper library also maintains correctness in the case where multiple containers sharing some or all of their image layers are concurrently deployed. In such cases, Docker will mount the same empty layers as part of multiple instances of OverlayFS. The only requirement is that all the involved images include a base layer with our wrapper library so execution correctness can be maintained when a required file is yet to arrive.

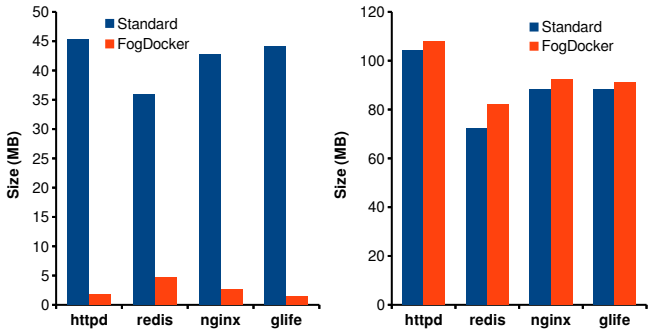
5 EVALUATION

5.1 Experimental setup

We implemented FogDocker as a modified version of Docker CE version 18.01. We therefore compare its performance against that of the same unmodified Docker version. We perform experiments on two different types of machines which represent two extremes in terms of hardware configurations:

- A Raspberry Pi 3 Model B+ is representative of a typical fog computing server [2, 15, 27]. It has a quad-core 1.4 GHz ARMv8 processor, 1 GB of RAM, a 32 GB micro-SD card, and a Gigabit Ethernet interface connected to a high-speed academic network. In this setup we use the standard Docker hub as registry.
- Although we designed FogDocker mostly for fog computing environments, we also evaluate it using a powerful server representative of a typical cloud computing environment. It has two Intel Xeon X5570 CPUs, with 4 cores per CPU running at 2.93 GHz, 24 GB of RAM, a 500 GB HDD and 1 Gbps Ethernet connection to a high-speed dedicated network. In this setup we use a private registry located in a separate cluster to ensure that the connection to the registry does not constitute a performance bottleneck.

We evaluate FogDocker based on the custom version of three popular images: `httpd` [11], `nginx` [12] and `redis` [13] are the official Docker images respectively containing the Apache web server, the Nginx web server and the Redis key-value store. To profile the web servers, we sent a number of sample HTTP requests; for redis, instead, we issued all the commands that are part of the official tutorial. We also set up a custom image, called `glife`, containing a simple application written in Go. In this case, we simply executed the enclosed application.



(a) Data volume to be transferred before container start-up.

(b) Overall image size on disk.

Figure 8: Comparison of image sizes.

5.2 Image layer sizes

We first evaluate the respective sizes of the base layers for all four images. This size indicates the volume of data which needs to be downloaded before a container starts. Figure 8(a) compares the base layer sizes with those of the corresponding original images.

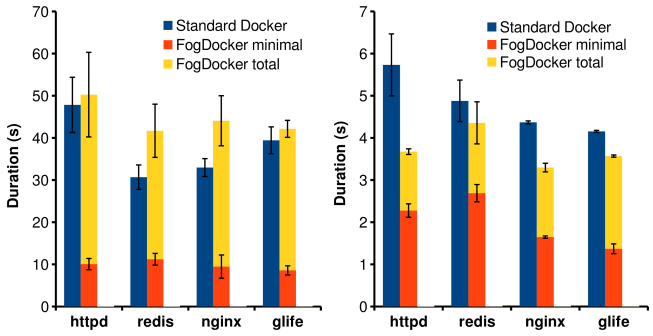
The reduction of necessary download size between regular Docker and FogDocker is evident. The size of the base layer represents between 3% (for glife) and 13% (for redis) of the original image. This is consistent with the results from [17], and confirms the performance improvement potential of FogDocker compared to regular Docker.

Although FogDocker drastically reduces the volume of data to be downloaded ahead of the container launch, the total image size ends up being larger than the original one because the base layer introduces redundancy. Figure 8(b) compares the full image sizes in both versions of Docker. The total data volume to be downloaded by FogDocker exhibits a modest overhead compared to the original images.

5.3 Container startup times

We now evaluate the container startup times in Docker and FogDocker. In the case of FogDocker we measure two delays: "FogDocker minimal" represents the delay from the container launch instruction until the container is ready to start; on the other hand, "FogDocker total" represents the overall duration until the full image has been downloaded and installed.

Figure 9(a) reports the container deployment times for our four images in the fog computing server scenario. In this and the subsequent experiments, every measure reports the



(a) Fog computing scenario. (b) Cloud computing scenario.

Figure 9: Container startup times.

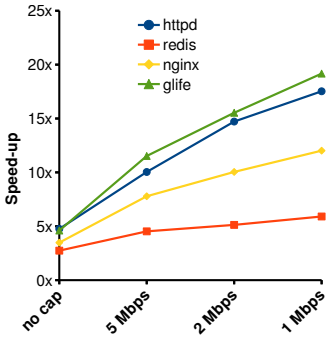


Figure 10: Speed-up with network bandwidth caps.

mean and standard deviation out of 5 identical executions. The performance difference between Docker and FogDocker is crisp: Docker requires between 30 s and 50 s to deploy the different images whereas FogDocker needs only about 10 s before the container can start operating. This represents a speedup between 3x and 5x. Figure 9(b) reports the results of the same experiment on the powerful cloud computing server. Although the performance difference is less spectacular, here as well we observe speedups between 1.8x and 3x.

Note that we report the "total" container deployment time only for reasons of completeness. FogDocker uses the full available bandwidth to download the remaining layers after the base layer has been deployed. However, in many scenarios system administrators may prefer to throttle or postpone this asynchronous image download to reserve more bandwidth for other applications and/or the container itself (e.g., to communicate with clients), at the expense of a slightly longer "total" pull time.

5.4 Performance in resource-constrained networks

The previous experiments were conducted using machines connected to a high-quality academic network. However, in fog computing scenarios, servers may need to be connected to the Internet using various types of commodity networks [2, 27]. In low-bandwidth environments, reducing the volume of data which needs to be downloaded before container startup becomes even more important.

Figure 10 shows the speedup observed between Docker and FogDocker in the fog computing server with increasingly strict network bandwidth limitations. As expected, the speedup grows as the available bandwidth decreases. With a 5 Mbps bandwidth cap the speedups grow up to 12x; with a 1 Mbps the gain reaches 18x for the glife application.

We also observe considerable speedup differences between the four available images. These differences are related to the amount of reduction of necessary download sizes, as previously highlighted in Figure 8(a). The glife image, which exhibits the greater size ratio between the full image and its base layer, benefits the most from FogDocker; on the other hand, redis, which has the largest relative base layer size, benefits "only" from a 5x performance improvement in constraint network conditions.

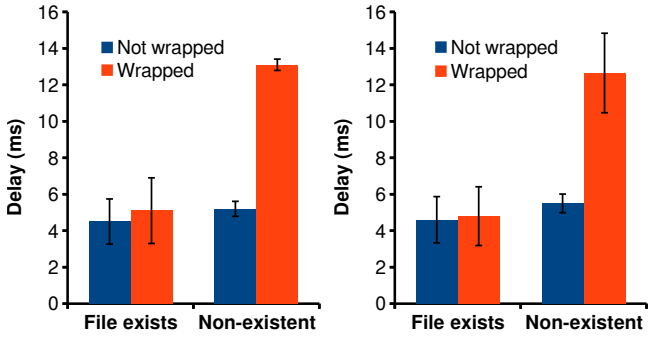
5.5 Performance overhead of wrapped system calls

We finish the evaluation by measuring the overhead of the wrappers which intercept calls to the `open()` and `fopen()` functions. These wrappers are necessary to maintain correctness in case a container deviates from its profiled behavior and tries to access a file which was not downloaded yet.

We prepare a text file containing the paths of 6000 files representing the list of lowerdir files that FogDocker is supposed to download asynchronously at run time. Then, we enable the wrapper by setting the `LD_PRELOAD` environment variable and configure it to perform lookups in our mentioned list. To evaluate the overhead produced by the wrapper, we measure how long it takes to execute a custom program that repeatedly opens and closes a given file.

Figure 11 reports the execution times of the `open()` and `fopen()` calls. The measurements have been done on a slow Raspberry Pi with the aim to evaluate a worst-case scenario. We execute each function call 20 times, and report the mean and standard deviation. When the requested file already exists, the wrapper immediately issues a call to the actual `open()` or `fopen()` functions. The performance overhead introduced by the wrapper, compared to when the library gets called directly, is negligible.

If the requested file does not exist, then the wrapper function also needs to check if it belongs to the full image (in



(a) With the `open()` function. (b) With the `fopen()` function.

Figure 11: System call wrappers' overhead.

which case the call must be delayed until the file arrives), or if it does not belong to the image (in which case the call can be immediately passed to the actual `open()` or `fopen()` function). We do not compare the delays in the case where the file belongs to the image, as the execution time here does not reflect an overhead of the wrapper library but rather the delay until the requested file has been downloaded. We however do measure the delay in case the file does not belong to the image. In this case the wrapper must check the path of the requested file against each entry of the list, just to find out that it is not part of it. Here we observe that the presence of the wrapper introduces a tolerable overhead (between 7 and 8 ms) to the time it usually takes to perform the system call.

Note that, once a file has been opened, the subsequent file system operations such as `read()`, `write()`, `lseek()` and `close()` are not wrapped so they do not experience any performance overhead. Furthermore, once the full image deployment is over, FogDocker could be instructed to remove the list of delayed files. In this way, the wrapper would forward all system calls to the standard library, reducing the overhead almost to zero.

6 CONCLUSION

Slow container deployment is an important issue in environments such as fog computing where this operation lies in the critical path of providing online services to the end users. The problem is even worse when the virtualized resources are made of modest machines such as single-board computers. We presented FogDocker which significantly reduces the time until a container is ready to operate by identifying the core set of files that are necessary to start, and by deploying these files first. FogDocker reduces the container startup time by 3-5x on limited fog computing servers, and 1.8-3x on powerful cloud computing servers while experiencing only modest runtime performance overhead when the container opens a file. When the available network bandwidth is restricted, as is often the case in fog computing, the achieved speedup grows up to 5-18x. At the same time, FogDocker maintains the Docker philosophy of image layer reusability and it requires only minor modifications in the Docker code.

Future improvements of FogDocker include the reintroduction of layer integrity control. A straightforward way to do so is to verify each layer as soon as its download has terminated. In case of failure, we can envision two potential strategies: either deleting the current layer and scheduling a new attempt, or throwing an error and stopping the execution of the container.

REFERENCES

[1] Arif Ahmed and Guillaume Pierre. 2018. Docker Container Deployment in Fog Computing Infrastructures. In *Proc. IEEE EDGE*.

[2] Paolo Bellavista et al. 2017. Feasibility of Fog Computing Deployment based on Docker Containerization over RaspberryPi. In *Proc. ICDCN*.

[3] Luiz F. Bittencourt et al. 2017. Mobility-Aware Application Scheduling in Fog Computing. *IEEE Cloud Computing* 4, 2 (2017).

[4] Eric Carter. 2018. Docker Usage Report. https://sysdig.com/blog/2018-docker-usage-report/

[5] Gene Cooperman et al. 2006. Transparent adaptive library-based checkpointing for master-worker style parallelism. In *Proc. CCGrid*.

[6] Jad Darrous et al. 2018. Nitro: Network-Aware Virtual Machine Image Management in Geo-Distributed Clouds. In *Proc. CCGrid*.

[7] Docker Inc. 2019. Docker: Build, Ship, and Run Any App, Anywhere. https://www.docker.com/.

[8] Docker Inc. 2019. Docker Hub. https://hub.docker.com/.

[9] Docker Inc. 2019. Docker storage drivers. https://docs.docker.com/storage/storagedriver/select-storage-driver/.

[10] Docker Inc. 2019. Use bind mounts. https://docs.docker.com/storage/bind-mounts/.

[11] Docker Official Images. 2019. The Apache HTTP Server Project. https://hub.docker.com/_/httpd.

[12] Docker Official Images. 2019. Official build of Nginx. https://hub.docker.com/_/nginx.

[13] Docker Official Images. 2019. An open source key-value store. https://hub.docker.com/_/redis.

[14] DockerSlim. 2019. Minify Docker Image and Generate Security Profiles. Frictionless! https://dockersl.im/.

[15] Wajdi Hajji and Fung Po Tso. 2016. Understanding the Performance of Low Power Raspberry Pi Cloud for Big Data. *Electronics* 5, 2 (2016).

[16] Halo Linux Services. 2017. https://www.halolinux.us/kernel-reference/the-dentry-cache.html

[17] Tyler Harter et al. 2016. Slacker: Fast Distribution with Lazy Docker Containers. In *Proc. USENIX FAST*.

[18] Saiful Hoque et al. 2017. Towards Container Orchestration in Fog Computing Infrastructures. In *Proc. IEEE COMPSAC*.

[19] James J Kistler and Mahadev Satyanarayanan. 1992. Disconnected operation in the Coda file system. *ACM TOCS* 10, 1 (1992).

[20] Senthil Nathan et al. 2017. CoMICon: A Co-Operative Management System for Docker Container Images. In *Proc. IC2E*.

[21] Eric Paris. 2009. fanotify: the fscking all notification system. https://lwn.net/Articles/339253/.

[22] Martin Pitt. 2012. fatrace: report system wide file access events. https://piware.de/2012/02/fatrace-report-system-wide-file-access-events/

[23] Michael Rieker et al. 2006. Transparent User-Level Checkpointing for the Native Posix Thread Library for Linux.. In *Proc. PDPTA*.

[24] RightScale. 2019. State of the Cloud report. https://www.rightscale.com/lp/state-of-the-cloud.
[25] Miklos Szeredi. 2014. Overlay FS. https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt
[26] Andrew Tridgell and Paul Mackerras. 1996. *The rsync algorithm*. Technical Report TR-CS-96-05. Australian National University. https://rsync.samba.org/
[27] Alexandre van Kempen et al. 2017. MEC-ConPaaS: An experimental single-board based mobile edge cloud. In *Proc. IEEE Mobile Cloud*.
[28] Wikipedia. 2019. Copy-on-write. https://en.wikipedia.org/wiki/Copy-on-write
[29] Wikipedia. 2019. Lazy Loading. https://en.wikipedia.org/wiki/Lazy_loading
[30] Chao Zheng et al. 2018. Wharf: Sharing Docker Images in a Distributed File System. In *Proc. ACM SOCC*.