Docker-pi: Docker Container Deployment in Fog Computing Infrastructures

Arif Ahmed, Guillaume Pierre

Docker-pi: Docker Container Deployment in Fog Computing Infrastructures

Arif Ahmed Guillaume Pierre
Univ Rennes, Inria, CNRS, IRISA

Abstract

The transition from virtual machine-based infrastructures to container-based ones brings the promise of swift and efficient software deployment in large-scale computing infrastructures. However, in fog computing environments which are often made of very small computers such as Raspberry PIs, deploying even a very simple Docker container may take multiple minutes. We demonstrate that Docker makes inefficient usage of the available hardware resources, essentially using different hardware subsystems (network bandwidth, CPU, disk I/O) sequentially rather than simultaneously. We therefore propose three optimizations which, once combined, reduce container deployment times by a factor up to 4. These optimizations also speed up deployment time by about 30% in datacenter-grade servers.

1 Introduction

Fog computing extends datacenter-based cloud platforms with additional resources located in the immediate vicinity of the end users. By bringing computation where the input data was produced and the resulting output data will be consumed, fog computing is expected to support new types of applications which either require very low network latency to their end users (e.g., augmented reality applications) or produce large volumes of data which are relevant only locally (e.g., IoT-based data analytics).

Fog computing architectures are fundamentally different from classical cloud platforms: to provide computing resources in the physical vicinity of any end user, fog computing platforms must necessarily rely on very large numbers of small Points-of-Presence connected to each other with commodity networks, whereas clouds are typically organized with a handful of extremely powerful data centers connected by dedicated ultra-high-speed networks. This geographical spread also implies that the machines used in any Point-of-Presence may not be datacenter-grade servers but much weaker commodity machines. As a matter of fact, one option which is being explored is to use single-board computers such as Raspberry PIs for this purpose. Despite their obvious hardware limitations, Raspberry PIs offer excellent performance/cost/energy ratios and are

well-suited to scenarios where the device's physical size and energy consumption are important enablers for actual deployment [2, 10].

However, building a high-performance fog platform based on tiny single-board computers remains a difficult challenge: in particular these machines have very limited I/O performance. In this paper, we focus on the issue of downloading and deploying Docker containers in single-board computers. We assume that server machines have limited storage capacity and therefore cannot be expected to keep in cache the container images of many applications that may be used simultaneously in a public fog computing infrastructure.

Deploying container images can be painfully slow, in the order of multiple minutes depending on the container's image size and network condition. However, such delays are unacceptable in scenarios such as a fog-assisted augmented reality application where the end users are mobile and new containers must be dynamically created when a user enters a new geographical area. Reducing deployment times as much as possible is therefore instrumental in providing a satisfactory user experience.

We show that this poor performance is not only due to hardware limitations. In fact it largely results from the way Docker implements the container's image download operation: (a) Docker downloads multiple image layers in parallel and then extracts the layers sequentially starting from the first layer. However, downloading the layers in parallel delays the download process of the first layer and therefore, postpones the moment its decompression and extraction phase can start. Therefore, delaying the downloading of the first layer ultimately leads to slowing down the extraction phase; (b) Docker image layers are shipped as compressed tar files. Upon downloading an image file, Docker decompresses it using single-threaded decompression *gzip* which account only for ∼37% CPU utilization of all the machine's cores. A significant amount of deployment time is spent in decompressing the layers; (c) Each image layer is sequentially downloaded, decompressed and extracted to disk. In other words, there is very little overlapping between the three activities of the different hardware resources (i.e. network, CPU and disk) while deploying the image. Docker exploits different hardware subsystems (network bandwidth, CPU, disk I/O) sequentially rather than simultaneously.

We therefore propose three optimization techniques which aim to improve the level of parallelism of the deployment process. Each technique reduces deployment times by 10-50% depending on the content and structure of the container's image and the available network bandwidth. When combined together, the resulting "Docker-pi" implementation makes container deployment up to 4 times faster than the vanilla Docker implementation, while remaining totally compatible with unmodified Docker images.

Interestingly, although we designed Docker-pi in the context of single-board computers, it also provides 23–36% performance improvements on high-end servers as well, depending on the image size and organization.

This paper is an extended version of a previous conference paper [4]. The additional contributions of this article are as follows: (i) We evaluate the resource utilization when deploying three different Docker images and thereby

illustrate the influence of image structure on the performance of each optimization solution (Sections 3-4); (ii) We compare the memory footprint of Docker-pi and standard Docker while deploying container images, and demonstrate that Docker-pi consumes much less memory that regular Docker during image installation (Section 4.4.2); (iii) We evaluate the impact of deploying a Docker image on the performance of an already-running application in the same machine (Section 4.4.3).

This paper is organized as follows. Section 2 presents the background and related work. Section 3 analyzes the deployment process and points out its inefficiencies. Section 4 proposes and evaluates three optimizations. Finally, Section 5 discusses practicalities, and Section 6 concludes.

2 Background

2.1 Docker background

Docker is a popular framework to build, package, and run applications inside containers [7]. Applications are packaged in the form of *images* which contain a part of a file system with the required libraries, executables, configuration files, etc. Images are stored in centralized repositories where they are accessible from any compute server. To deploy a container, Docker therefore first downloads the image from the repository and locally installs it, unless the image is already cached in the compute node. Starting a container from a locally-installed image is as quick as starting the processes which constitute the container's application. The deployment time of any container is therefore dictated by the time it takes to download, decompress, verify, and locally install the image before starting the container itself.

2.1.1 Image structure

Docker images are composed of multiple layers stacked upon one another: every layer may add, remove, or overwrite files present in the layers below itself. This enables developers to build new images very easily by simply specializing pre-existing images.

The same layering strategy is also used to store file system updates performed by the applications after a container has started: upon every container deployment, Docker creates an additional writable top-level layer which stores all updates following a Copy-on-Write (CoW) policy. The container's image layers themselves remain read-only. Table 1 shows the structures of the three images used in this paper.

Figure 1 shows a typical image which consists of 7 layers. The first five layers represent a standard Ubuntu Linux image: the first layer holds most of the contents (46 MB in compressed form), and the other layers are much smaller specializations such as updates of some specific packages and/or configuration files. In this example, we further extended Ubuntu into a so-called "Mubuntu"

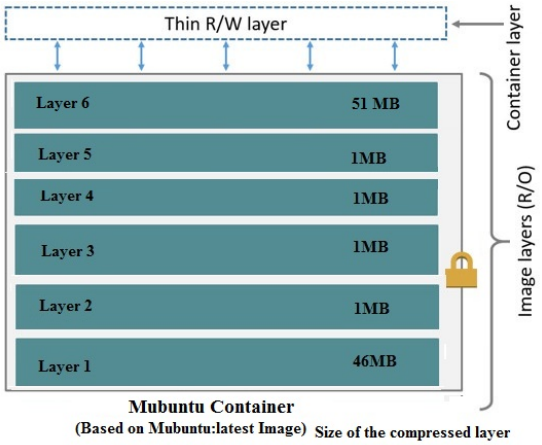


Figure 1: Structure of the "Mubuntu" container image.

image with an extra 51 MB layer representing a hypothetical application's executable and data. Finally, Layer 7 stores any file system update which may be issued by the application during its execution. Each layer is stored in the registry as a simple compressed tar file.

2.1.2 Container deployment process

Docker images are identified with a name and a tag representing a specific version of the image. Docker users can start any container by simply giving its name and tag using the command: `docker run IMAGE:TAG [parameters]`

Docker keeps a copy of the latest deployed images in a local cache. When a user starts a container, there are four possible cases depending on the contents of this cache:

- **The image is not available.** In this case the image must be downloaded from the Docker registry.
- **The image is available but with a different tag.** This is treated the same way as if the image was not available in the host machine: the correct image will be downloaded before being started.
- **The image is already available.** The container can be started immediately.
- **Some of the layers of the image are available.** This case derives from the incremental way images are built, which encourages image layer re-usability. In this case, only the missing layers (uniquely identified with a *sha256* hash of their content) will be pulled from the docker registry. Once all layers are available, Docker will create the unified image of the application, and the container can be started.

The goal of this work is to better understand the different phases of the Docker container deployment process — in particular in terms of hardware resource usage — and to propose alternative techniques to speed up the download and installation of the required image layers. We assume that the image cache is empty at the time of the deployment request: fog computing servers will most likely have very limited storage capacity so in this context we expect that cache misses will be the norm rather than the exception.

2.2 Related work

Many recent research efforts have recognized the potential of single-board devices for building future fog computing infrastructures and have evaluated their suitability for handling cloud-like types of workloads. For instance, Bellavista *et al* studied the practicality of Docker container deployment in fog computing infrastructures for IoT applications [14]. They demonstrated that even extremely constrained devices such as Raspberry PIs may be successfully used to build IoT cloud gateways. They concluded that with proper configuration and optimization, these cost-effective devices can achieve scalable performance with minimal overhead. However, the study was carried out assuming that the Docker container images were already cached in the local nodes. In contrast, we largely focus on the download-and-install part of the Docker container deployment, and show that simple updates in the Docker implementation can significantly improve the performance of this operation.

CoMICon is a distributed Docker registry which aims to reduce the container provisioning time [16]. It distributes layers of an image among multiple nodes to increase availability. The proposed model also allows one to pull an image from multiple registries simultaneously, which reduces the average layer's download times. Similar approaches for distributed docker image downloads using various peer-to-peer protocols [5, 13]. However, distributed downloading solutions rely on the assumption that several powerful servers are interconnected with a high-speed local-area network, and therefore that the main performance bottleneck is the long-distance network to a remote centralized repository. In the case of fog computing platforms, servers will be geographically distributed to maximize proximity to the end users, and they will rarely be connected using high-capacity networks. As we discuss in the next section, the main bottleneck in fog computing nodes is created by hardware limitations of every individual node.

Another way to improve the container deployment time is to propose a new Docker storage driver. Slacker proposes to rely on a centralized NFS file system to share the images between all the nodes and registries [17]. The lazy pulling of the container image in the proposed model significantly improves the overall container deployment time. However, Slacker expects that the container image is already present in the local multi-server cluster environment; in contrast, a fog computing environment is made of large numbers of nodes located far from each other, and the limited storage capacity of each node implies that few images can be stored locally for future use. Another difference is that Slacker

Table 1: Structure of the Docker images.

	Ubuntu image	**Mubuntu image**	**Biglayers image**
6th layer	–	51 MB	-
5th layer	<1 MB	<1 MB	-
4th layer	<1 MB	<1 MB	62 MB
3rd layer	<1 MB	<1 MB	54 MB
2nd layer	<1 MB	<1 MB	64 MB
1st layer	46 MB	46 MB	52 MB
Total size	50 MB	101 MB	232 MB

requires flattening the Docker images in a single layer. This makes it easier to support snapshot and clone operations, but it deviates from the standard Docker philosophy which promotes the layering system as a way to simplify image creation and updates. Slacker therefore requires the use of a modified Docker storage driver (which implements de-duplication features) while our work keeps the image structure unmodified and does not constraint the choice of a storage driver. We discuss the topic of flattening Docker images in Section 5.1.

3 Understanding the Docker container deployment process

To understand the Docker container deployment process in full details we analyzed the hardware resource usage during the download, installation and deployment of a number of Docker images on a Raspberry PI-based infrastructure. The limited hardware capabilities were instrumental in highlighting the inefficiencies of this process, which would be more difficult to pinpoint using faster server machines.

3.1 Experimental setup

We monitored the Docker deployment process on a testbed which consists of three Raspberry Pi 3 machines connected to each other and to the rest of the Internet with 10 Gbps Ethernet [1]. The testbed was installed with the latest Docker version (17.06) This setup also allowed us to emulate slower network connections — which are arguably representative of real fog computing scenarios — by throttling network traffic at the network interface level. We used the `tc` (Traffic Control) command to run experiments either with unlimited bandwidth, or with limits of 1 Mbps, 512 kbps or 256 kbps.

Table 1 depicts the three Docker images we used for this study. The first image simply conveys a standard *Ubuntu* operating system: it is composed of one layer containing most of the content, and four small additional layers which contain various updates of the base layer. The second is the *Mubuntu* image

already presented in Figure 1. Finally, as the name suggests, the *BigLayers* image is composed of four big layers which allow us to highlight the effect of the layering system on container deployment performance.

We instrumented the testbed nodes to monitor the overall deployment time as well as the utilization of important resources during the container deployment process:

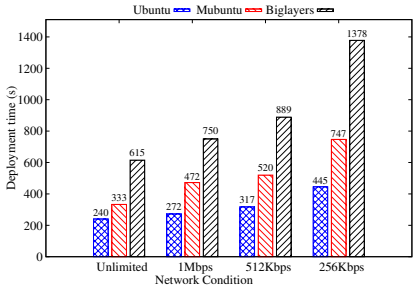
- **Deployment time:** We measured the deployment time from the moment the deployment command is issued, until the time when Docker reports that the container is started.
- **Network activities:** Network activities include incoming data and outgoing data during deployment. We used the *nethogs* tool to monitor the network activities during the whole deployment processes at a 1-second granularity[8]. The script traces the specific network activity of the Docker daemon, and therefore does not take other sources of background traffic into account.
- **Disk throughput:** We monitored the disk activity with the *iostat* Linux command which monitors the number of bytes written to or read from disk at a 1-second granularity.
- **CPU usage:** We monitored CPU utilization by watching the */proc/stat* file at a 1-second granularity.

Unless otherwise stated, every container deployment experiment was issued on an idle node, and with an empty image cache.

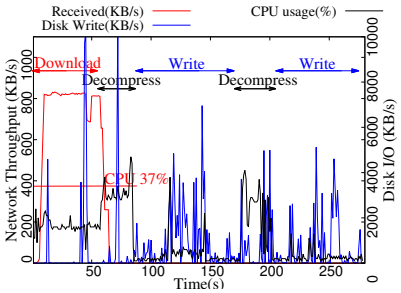
3.2 Monitoring the Docker container deployment process

Figure 2 depicts the results when deploying the three images using regular Docker. Figure 2(a) shows the deployment time of our three images in different network conditions: deploying the Ubuntu, Mubuntu and Biglayers images with unlimited network bandwidth respectively takes 240, 333 and 615 seconds. Clearly, the overall container deployment time is roughly proportional to the size of the image. When throttling the network capacity, deployment times grow steadily as the network capacity is reduced to 1 Mbps, 512 kbps, and 256 kbps. For instance, deploying the Ubuntu container takes 6 minutes when the network capacity is reduced to 512 kbps. This is considerable with regards to the deployment efficiency one would expect from a container-based infrastructure. However, the interesting information for us is the *reason* why deployment takes so long, as we discuss next.

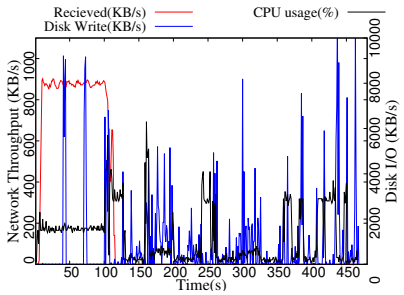
Figure 2(b) depicts the utilization of different hardware resources from the host machine during the deployment of the standard Ubuntu image. The red line shows incoming network bandwidth utilization, while the blue curve represents the number of bytes written to the disk and the black line shows the CPU utilization. The first phase after the container creation command is issued involves intensive network activities, which indicates that Docker is downloading



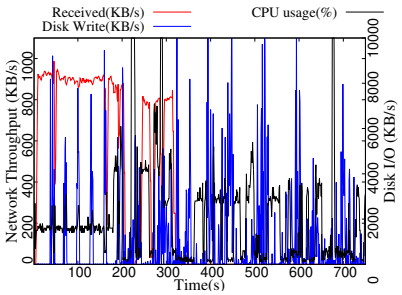
(a) Deployment times.



(b) Ubuntu image with a 1 Mbps network cap.



(c) Mubuntu image with a 1 Mbps network cap.



(d) BigLayers image with a 1 Mbps network cap.

Figure 2: Deployment times and resource usage using standard Docker.

the image layers from the remote image registry. By default Docker downloads up to three image layers in parallel. The duration of downloads clearly depend on the image size and the available network capacity: between 55 s and 200 s for the Ubuntu and Mubuntu images. During this phase, we observe no significant disk activity in the host machine, which indicates that the downloaded file is kept in main memory.

After the download phase, Docker extracts the downloaded image layers to the disk before building the final image of the application. The extraction of a layer involves two operations: decompression (which is CPU-intensive) and writing to the disk (which is disk-intensive). We observe that the resource utilization alternates between periods during which the CPU is busy (∼40% utilization) while few disk activities are performed, and periods during which disk writes are the only notable activity of the system. We conclude that, after the image layers have been downloaded, Docker sequentially decompresses the image and writes the decompressed data to disk. When the image data is big, Docker alternates between partial decompressions and disk writes, while maintaining the same sequential behavior.

We see a very similar phenomenon in Figures 2(c) and 2(d). However, in here the downloading of the first layer terminates before the other layers have finished downloading. The extraction of the first layer can therefore start before the end of the download phase, creating a small overlap between the downloading and extraction phases.

3.3 Critical observations

From the previous experiments we derive a few important observations.

3.3.1 Overall deployment time

The overall deployment of a new container mainly involves three operations: searching for the cached image, pulling the image from the registry and starting the container. Our work assumes that the image is not cached on the machine, so every container deployment involves pulling the image from the registry. As we have seen, Docker takes a significant amount of time for pulling the image from the registry while the other two operations take a negligible amount of time. In this paper, we therefore mainly focus on optimizing the Docker image pull operation.

3.3.2 Pulling image layers in parallel

By default, Docker downloads image layers in parallel with a maximum parallelism level of three. These layers are then decompressed and extracted to disk sequentially starting from the first layer. However, when the available network bandwidth is limited, downloading multiple layers in parallel will delay the download completion of the first layer, and therefore will postpone the moment when the decompression and extraction process can start. Therefore, delaying the downloading of the first layer ultimately leads to slowing down the extraction phase.

3.3.3 Single-threaded decompression

Docker always ships the image layers in compressed form, usually implemented as a *gzipped* tar file. This reduces the transmission cost of the image layers but it increases the CPU demand on the server node to decompress the images before extracting the image to disk. Docker decompresses the images via a call to the standard *gunzip.go* function, which happens to be single-threaded. However, even very limited machines usually have several CPU cores available (4 cores in the case of a Raspberry Pi 3). The whole process is therefore bottlenecked by the single-threaded decompression. As a result the CPU utilization never grows beyond ∼40% of the four cores of the machine, wasting precious computation resources which may be exploited to speed up image decompression.

3.3.4 Resource under-utilization

The standard Docker container deployment process under-utilizes the available hardware resources. Essentially, deploying a container begins with a network-intensive phase during which the CPU and disk are mostly idle. It then alternates between CPU-intensive decompression operations (during which the network and disk are mostly idle) and I/O-intensive image extraction operations (during which the network and CPU are mostly idle). The only case where these operations slightly overlap are images such as Mubuntu and BigLayers when the decompress and extraction process of the first layer can start while the last images are still being downloaded.

This resource under-utilization is one of the main reason for the poor performance of the overall container deployment process. The main contribution of this paper is to show how one may reorganize the Docker deployment process to maximize resource utilization during deployment.

4 Optimizing the container deployment process

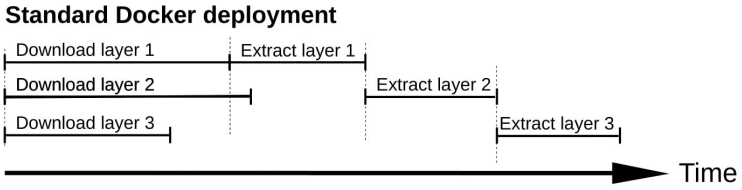
To address the inefficiencies presented in the previous section we propose and evaluate three optimization techniques to speed up the container provisioning time. Each optimization addresses a different issue in the standard Docker container deployment. We can therefore combine them all together, which brings significant performance improvement.

4.1 Sequential image layer downloading

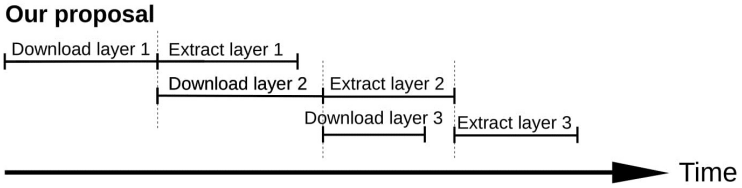
As previously discussed, Docker parallelizes image downloads from the central repository to the local node, with a default concurrency degree of 3. This is a classical technique to maximize the overall network throughput. However, in the specific case of Docker image downloads this strategy has a negative effect because the next phases of the container deployment, namely the decompress and extraction phases, must take place sequentially to preserve the Copy-on-Write policy of Docker storage drivers. The decompress & extract phase can start only after the first layer has been downloaded. Downloading multiple image layers in parallel will delay the download completion time of the first layer because this download must share network resources with other less-urgent image downloads, and will therefore also delay the moment when the first layer can start its decompress & extract phase.

The only cases where the decompression & extraction of one layer overlaps with the download of another image can seen in Figure 2(c) & 2(d). They result from the fact that the download concurrency degree delays the downloading of the last images. We therefore propose to extend this phenomenon, and to reduce the download concurrency degree to one, essentially reverting to a sequential download of the image layers one after another.

Figure 3 illustrates the effect of downloading multiple layers sequentially rather than in parallel, in an example with an image made of three layers. In



(a) Standard Docker pull with parallel layer download.



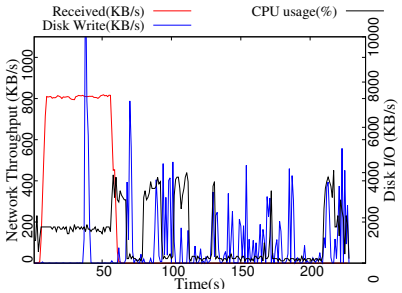
(b) Docker pull with sequential layer download.

Figure 3: Standard and sequential layer pull operations.

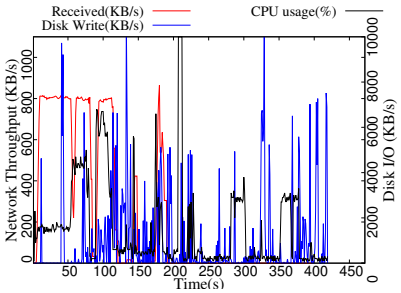
both cases, three threads are created to handle the three image layers. However, in the first option the downloads take place in parallel whereas the only required inter-thread synchronization requires that the decompression and extraction of layer n can start only after the decompression and extraction of layer $n-1$ has completed. In sequential downloading, the second layer starts downloading only when the first download has completed, which means that it takes place while the first layer is being decompressed and extracted to disk. This allows the first-layer extraction to start sooner and it also increases resource utilization because the download and the decompress & extract operations make intensive use of different part of the machine's hardware.

Implementing sequential image downloading requires additional inter-thread synchronization: in this new model the downloading of layer n can start only after the end of the layer $n-1$ download, whereas the decompress & extract of layer n can start only after layer n has been downloaded *and* the layer $n-1$ has finished its extraction. A simple way to implement this is to set the "max-concurrent-downloads" parameter to 1 in the `/etc/docker/daemon.json` configuration file.

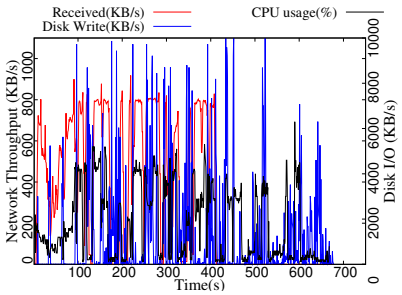
Figure 4 depicts the host machine's resource usage when using sequential downloading of our reference images, and compares the overall deployment times with various network bandwidth limitations. Figure 4(a) shows the resource usage when deploying the Ubuntu image with sequential downloading and a 1 Mbps network capacity. The figure is not much different from Figure 2(a) where the image downloads were done in parallel. The reason is that the Ubuntu image contains only one layer with a significant size, so the downloading and extraction of layers 2–5 is very short compared to layer 1. However, in Figures 4(b) and 4(c), we observe that after the downloading of layer 1 has completed, the



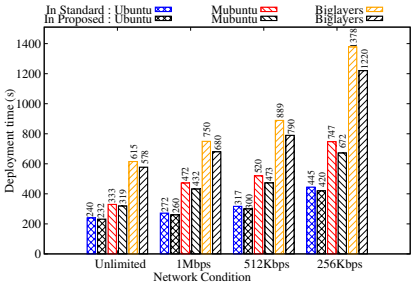
(a) Ubuntu image with a 1 Mbps network cap.



(b) Mubuntu image with a 1 Mbps network cap.



(c) BigLayers image with a 1 Mbps network cap.



(d) Deployment times.

Figure 4: Resource usage and deployment time with sequential layers downloading.

utilization of hardware resources is much greater, with in particular a clear overlap between periods of intensive network, CPU and I/O resources. Also we can observe that the decompression of the first layer (visible as the first spike of CPU utilization) takes place sooner than in Figure 2.

Figure 4(d) compares the overall container deployment times with parallel and sequential downloads in various network conditions. When the network capacity is unlimited the performance gains in the deployment of the Ubuntu, Mubuntu and BigLayers images are 3%, 4.2% and 6% respectively.

However, the performance gains grow steadily as the available network bandwidth gets reduced. With a bandwidth cap of 256 kbps, sequential downloading brings improvements of 6% for the Ubuntu image, 10% for Mubuntu and 12% for BigLayers. This is due to the fact that slower network capacities exacerbate the duration of the download phases and increases the delaying effect of parallel layer downloading.

Sequential downloading therefore provides modest yet non-negligible performance gains. This technique works best when the deployed image contains multiple large layers, and when the network capacity is very limited. These

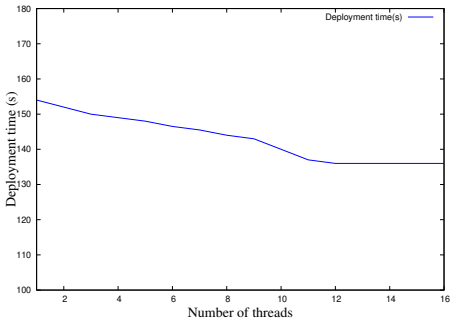


Figure 5: Impact of the number of *pgzip* threads on the deployment time.

conditions happen to match the properties of a container deployment in fog computing environments where non-trivial applications will be deployed in extremely distributed infrastructures which necessarily rely on limited commodity networks.

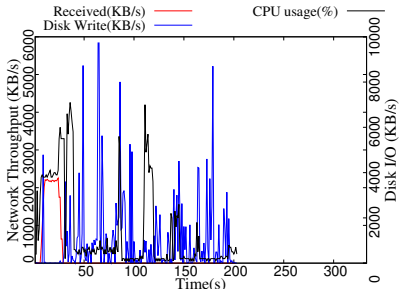
4.2 Multi-threaded layer decompression

Docker image layers are stored and downloaded in the form of a gzipped tar file. After downloading the files from the registry, the compute node therefore needs to decompress every layer before building the image on disk. In our experiments based on Raspberry PI and an unlimited network capacity, the duration of the decompression phase is greater than that of the image download. Increasing the speed of file decompression therefore has the potential to significantly reduce the overall container deployment time.

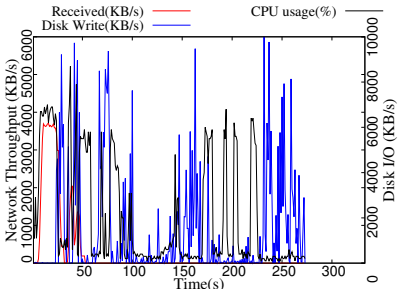
By default, Docker compresses image layers using gzip. Decompression is implemented entirely in the Go language using the standard *gunzip.go* library [18]. However, this function is single-threaded which means that it is unable to exploit multiple cores to speed up decompression. As a result, the CPU utilization during decompression never exceeds 40% of the four available cores in the Raspberry Pi machine.

We therefore propose to replace the single-threaded *gunzip.go* library with a multi-threaded implementation so that all the available CPU resources may be used to speed up this part of the container deployment process. We use *pgzip*, which is a multi-threaded implementation of the standard gzip/gunzip functions [15]. Its functionalities are exactly the same as those of the standard *gzip*, however it splits the work between multiple independent threads. When applied to large files of at least 1 MB, this can significantly speed up decompression.

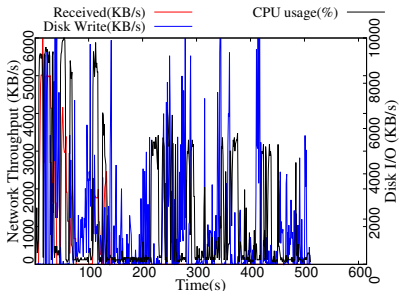
To determine the appropriate number of threads we should allow *pgzip* to use, we deployed a custom container image while varying the available number of threads. We used a very simple image which consists of a single layer of 20 MB in compressed form. This allows us to better isolate the performance gains of the parallel decompression from other effects such as the possible overlap of the



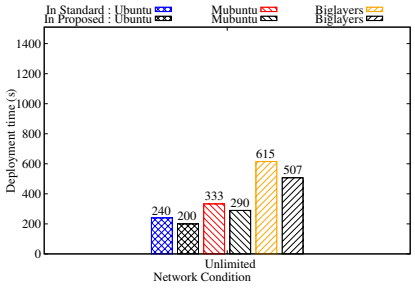
(a) Ubuntu image with multi-threaded decompression.



(b) Mubuntu image with multi-threaded decompression.



(c) BigLayers image with multi-threaded decompression.



(d) Deployment times.

Figure 6: Resource usage and deployment time with multi-threaded image layer decompression.

decompression with the download of other layers. The results are depicted in Figure 5.

When *pgzip* uses a single thread, the performance and CPU utilization during decompression are very similar to the standard *gunzip* implementation. However, when we increase the number of threads from 1 to 12, the overall container deployment time decreases from 154 s to 136 s. At the same time, the CPU utilization during decompression steadily increases from 40% to 71% of the four available CPU cores. If we push beyond 12 threads, no additional gains are observed. We clearly see that the parallel decompression does not scale linearly, as it is not able to exploit the full capacity of the overall CPU: this is due to the fact that gzip decompression must process data blocks of variable size so the decompression operation itself is inherently single-threaded [9]. The benefit of multi-threading decompression is that other necessary operations during decompression (essentially data buffering and CRC verification) can be delegated to other threads and moved out of the critical path.

Figure 6 shows the effect of using parallel decompression when deploying our three standard container images with 12 threads. We observe in Figures 6(a),

6(b) and 6(c) that the CPU utilization is greater during the decompression phases than with standard Docker, in the order of 70% utilization instead of 40%. Also, the decompression phase is notably shorter. The same phenomenon is visible in all three images.

We also notice that the parallel image download phase is fairly CPU-intensive: in the examples of the Mubuntu and the BigLayers images which both have several large layers to decompress, the CPU utilization during downloading grows up to 70% for Mubuntu and even 90% for BigLayers. This clearly indicates that it would be pointless to attempt parallel image layer decompression while simultaneously downloading multiple image layers.

Finally, Figure 6(d) compares the overall container deployment times with parallel decompression against that of the standard Docker. The network performance does not influence the decompression time so we conducted the evaluation only with an unlimited network capacity. The performance gain from multi-threaded decompression is similar for all three images, in the order of 17% of the overall deployment time.

The standard single-threaded *gzip* implementation creates an unnecessary performance bottleneck because it under-utilizes the CPU resources during the decompression of the image layer. Although parallelizing data decompression is very hard and it cannot offer linear speedup, parallel decompression allows one to move the tasks not directly related to decompression to helper threads, which still provides interesting performance benefits. Multi-threading decompression increases the CPU usage, and reduces the overall Docker container deployment times.

4.3 I/O pipelining

Despite the sequential downloading and the multi-threaded decompression techniques, the container deployment process still under-utilizes the hardware resources. The reason is due to the sequential nature of the workflow which is applied to each individual layer. Each layer is first downloaded in its entirety, then it is decompressed entirely, then it is extracted to disk. This requires Docker to keep the entire decompressed layer in memory, which can be significant considering that a Raspberry Pi 3 has only 1 GB of main memory. Also, it means that the first significant disk activity can start only after the first layer has been fully downloaded and decompressed. Similarly, Docker necessarily decompresses and extracts the last layer to disk while the networking device is mostly inactive.

However, there is no strict requirement for the download, decompress and extraction of a single layer to take place sequentially. For example, decompression may start right after the first bytes of the compressed layer have been downloaded. Similarly, extracting the layer may start immediately after the beginning of the layer image has been decompressed.

We therefore reorganize the download, decompression and extraction of a single layer in three separate threads where each thread pipelines data to the next as soon as some data is available. In Unix shell syntax this essentially

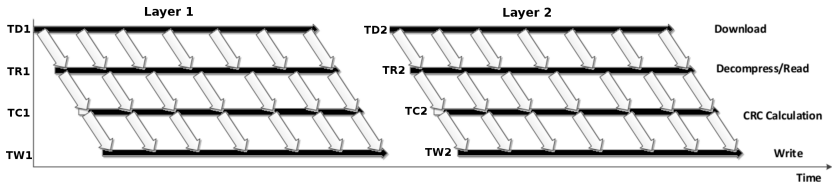
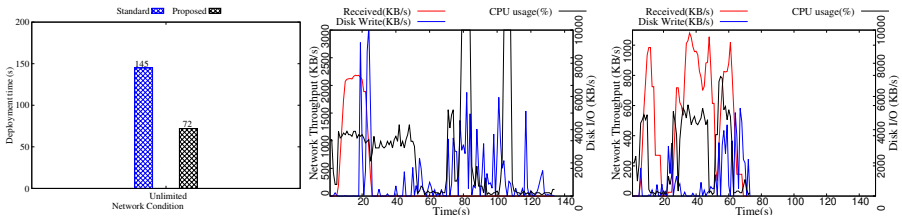


Figure 7: Docker pull operation with I/O pipelining.



(a) Deployment time comparison.

(b) Resource utilization with standard Docker.

(c) Resource utilization with I/O pipelining.

Figure 8: Deployment time and resource usage with I/O pipelining.

replaces the sequential "`download; decompress; crc-check; extract`" command with the concurrent "`download | decompress | crc-check | extract`" command. Figure 7 illustrates this technique with four threads responsible for downloading and decompression a Docker image layer. The thread TD1 downloads the image layer while TR1 performs the decompression, TC1 calculates the CRC, and finally TW1 writes the decompressed data to the disk. Since we stream the incoming downloaded data without buffering the entire layer, the thread TW1 can start writing content to disk long before the download process has completed.

We implemented pipelining using the *io.pipe()* GO API, which creates a synchronized in-memory pipe between an *Input(writer)* and an *Output(reader)*. However, we must be careful about synchronizing this process between multiple image layers: for example, if we created an independent pipeline for each layer separately, the result would violate the Docker policy that layers must be extracted to disk sequentially, as one layer may overwrite a file which is present in a lower layer. If we extracted multiple layers simultaneously we could end up with the wrong version of the file being given to the container. Rather than building complex synchronization mechanisms, we instead decided to rely on Docker's sequential downloading feature already discussed in Section 4.1. When a multi-layer image is deployed, this imposes that layers are downloaded and extracted one after the other, while using the I/O pipelining technique within each layer.

We now evaluate the I/O pipelining technique using a single-layer image only. In the next section we combine all three optimization techniques and therefore show the combined effect of the sequential download and the I/O pipelining. A

single-layered image can be created using a so-called "flatten" operation which creates a single tar file out of a deployed image. Since the flattened version contains a single copy of every file (even though it may have been overwritten multiple times by different layers), the flattened image is usually slightly smaller than the sum of all the initial layers' sizes.

Figure 8 compares the deployment of a flattened image between standard Docker and the I/O pipelining technique. We can see in Figure 8(a) that the pipelined version is roughly 50% faster than its standard counterpart. The reason can be found in Figures 8(b) and 8(c). In the standard deployment, resources are used one after the other: first network-intensive download, then CPU-intensive decompression, then finally disk-intensive image creation. In the pipelined version all operations take place simultaneously, which better utilizes the available hardware and significantly reduces the container deployment time.

4.4 Docker-pi

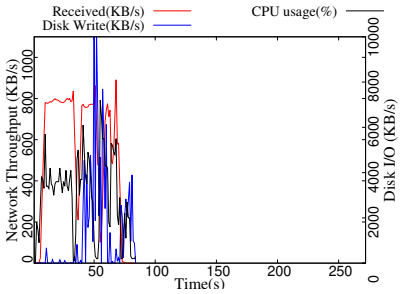
The three techniques presented here address different issues. Sequential downloading of the image layers speeds up the downloading of the first layer in slow network environments. Multi-threaded decompression speeds up the layer decompression by utilizing multiple CPU cores. Finally, I/O pipelining speeds up the deployment of each layer by conducting the download, decompress and extraction processes simultaneously, while avoiding having to keep large amounts of data in memory during the deployment process. We therefore propose *Docker-pi*, an optimized version of Docker which combines the three techniques to optimize container deployment on single-board machines such as Raspberry PIs. The implementation of Docker-pi is available online in the gitlab repository[3].

4.4.1 Deployment time

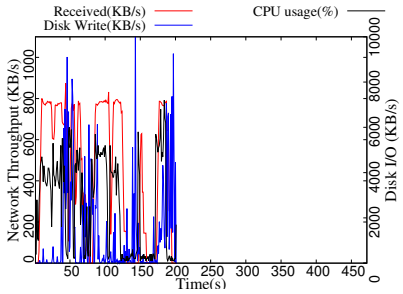
Figure 9 depicts the resource usage and deployment time of our three standard images using Docker-pi. We can clearly see in Figures 9(a), 9(b) and 9(c) that the networking, CPU and disk resources are used simultaneously and have a much greater utilization than with the standard Docker implementation. In particular, the CPU and disk activities start very early after the first few bytes of data have been downloaded.

Finally, Figure 9(d) highlights significant speedups compared to vanilla Docker: with no network cap, Docker-pi is 73% faster than Docker for the Ubuntu image, 65% faster for Mubuntu and 58% faster for BigLayer. When we impose bandwidth caps the overall deployment time becomes constraint by the download times, while the decompression and extraction operations take place while the download is taking place. In such bandwidth-limited environments the deployment time therefore cannot be reduced any further other than by pre-fetching images before the container deployment command is issued.

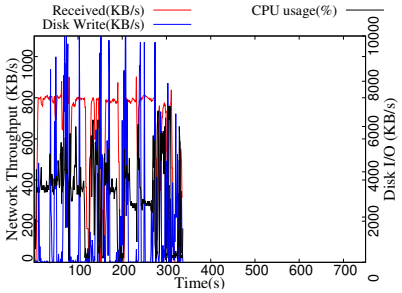
The reason why the gains are lower for the Mubuntu and BigLayers images is that the default download concurrency degree of 3 in vanilla Docker



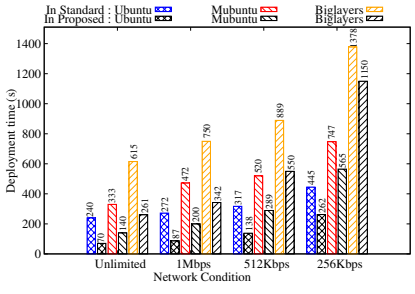
(a) Ubuntu image with a 1 Mbps network cap.



(b) Mubuntu image with a 1 Mbps network cap.



(c) BigLayers image with a 1 Mbps network cap.



(d) Deployment times.

Figure 9: Resource usage and deployment time with Docker-pi.

already makes them benefit from some of the improvements that we generalized in Docker-pi. If we increase the concurrency degree of vanilla Docker to 4, the BigLayers image deploys in 644 s whereas Docker-pi needs only 207 s, which represents 68% improvement.

4.4.2 Memory usage

We now evaluate the memory footprint of Docker and Docker-pi during the deployment process. For simplicity, we deployed a single layer image and extracted memory usage of the node by watching */proc/meminfo* file at a 1-second granularity. Figure 10 clearly shows the effect of pipelining on the memory footprint. Docker-pi starts extracting the layer to local disk immediately after the first few blocks of the layer are downloaded, therefore, memory footprint never exceeds 10 MB while deploying the image. In contrast, standard docker's memory usage is nearly 70 MB. The reason is that it keeps the complete compressed layer in memory, decompresses it entirely in memory again, before writing to disk and releasing both files from memory.

The memory footprint of standard Docker may vary depending on the image size and concurrency degree of parallel download. In a memory-constrained de-

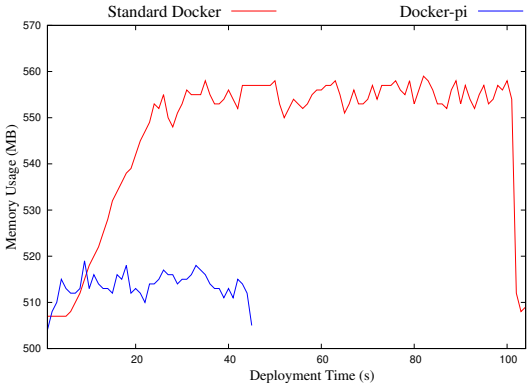


Figure 10: Memory footprint of Docker and Docker-pi during container deployment.

vice like Raspberry PI which has only 1GB of RAM, this may create significant bottlenecks. On the other hand, Docker-pi uses less memory and has a fairly constant footprint during the image pull operation irrespective of image structure, which makes it a better choice in environments such as memory-constraint fog computing devices.

4.4.3 Performance interference with already-running containers

In all experiments presented so far, container deployment took place in an otherwise idle machine. However, this scenario is unlikely in a busy fog computing environment where numerous independent applications share a limited number of physical resources. We therefore now evaluate the impact that container deployment has on already-running containers in the same machine.

We use an Apache Web server container [11] as the already-running container so we can observe its performance while another container with the single-layer image is being deployed. The Apache server serves a constant request workload produced by the http_load HTTP benchmarking tool [12]. We configured it to generate a constant load of 300 requests/second which fetch a single 5kB file. We monitor the Web server's network throughput using *nethogs* before, during and after the single-layer container is being deployed.

Figure 11 compares the upload throughput of the Web server while standard Docker or Docker-pi are deploying a new container image. We observe that in both cases the Web server performance is affected by the simultaneous container deployment taking place in the same machine. This is largely due to the fact that the web server and the container deployment processes need to compete for limited resources such as available network bandwidth. Interestingly, both versions of Docker impose a similar performance reduction to the web server during the time of container deployment. However, Docker-pi deploys the new container faster so the duration of the interference it creates is shorter than

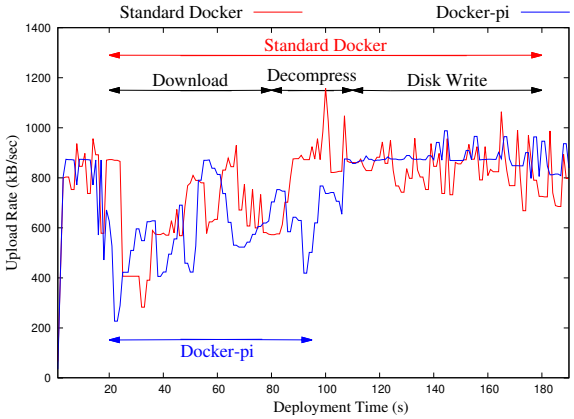


Figure 11: Upload throughput of the Apache web server.

using regular Docker. The fact that Docker-pi generates high resource utilization during container deployment does not seem to affect other containers in greater proportions than regular Docker.

5 Discussion

5.1 Should we flatten all Docker images?

Flattening all Docker images may arguably provide performance improvement in the deployment process. Indeed, multiple image layers may contain successive versions of the same file whereas a flattened image contains only the final version of every file. A flattened image is therefore always a little smaller than its multi-layered counterpart. Systems like Slacker actually rely on the fact that images have been flattened [17]. On the other hand, Docker-pi supports both flattened images and unmodified multi-layer images. We however do not believe that flattening all images would bring significant benefits.

Docker does not provide any standard tool to flatten images. This operation must be done manually by first exporting an image with all its layers, then re-importing the result as a single layer while re-introducing the startup commands from all the initial layers. The operation must be redone every time any update is made in any of the layers. Although this process could be integrated in a standard image build workflow, it contradicts the Docker philosophy which promotes incremental development based on image layer reusability.

In a system where many applications execute concurrently, one may reasonably expect many images to share at least the same base layers (e.g., Ubuntu) which produce a standard execution environment. If all images were flattened this would create large amounts of redundancy between different images, creating the need for sophisticated de-duplication techniques [17]. On the other hand, we believe that the layering system can be seen as a domain-specific form

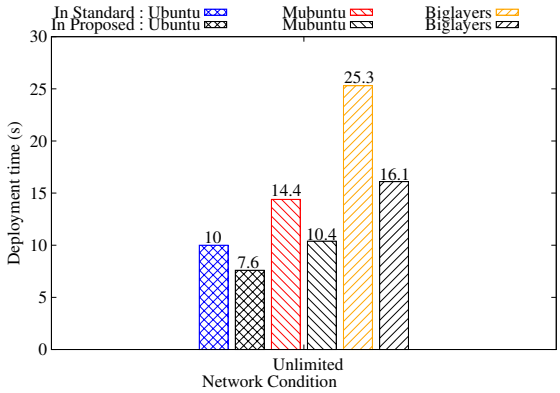


Figure 12: Deployment time of Docker and Docker-pi in Grid'5000.

of de-duplication which naturally integrates in a developer's devops workflow. We therefore prefer keeping docker images unmodified, and demonstrated that container deployment can be made extremely efficient without the need for flattening images.

5.2 Does Docker-pi work also for powerful server machines?

Although we designed Docker-pi for single-board machines, the inefficiencies of vanilla Docker also exist in powerful server environments. We therefore evaluate the respective performance of Docker and Docker-pi in the Grid'5000 testbed which is commonly used for research on parallel and distributed computing including Cloud, HPC and Big Data [6]. We specifically use a Dell PowerEdge C6220 server equipped with two 10-core Intel Xeon E5-2660v2 processors running at 2.2GHz, 128 GB of main memory and two 10 Gbps network connections, and do not cap the network bandwidth.

Figure 12 compares the deployment times of Docker and Docker-pi with our three standard images. Obviously container deployment is much faster in this environment than in Raspberry PIs. However, here as well Docker-pi provides respectable performance improvement in the order of 23% (Ubuntu), 29% (Mubuntu) and 36% (BigLayers). In this powerful server the network and CPU resources cannot be considered as bottlenecks so the sequential layer downloading and multi-threaded decompression techniques bring almost no improvement compared to the standard Docker. On the other hand, the sequential nature of the download/decompress/extract process is still present regardless of the hardware architecture, so the I/O pipelining technique brings similar performance gains as with the Raspberry PI.

6 Conclusion

The transition from virtual machine-based infrastructures to container-based ones brings the promise of swift and efficient software deployment in large-scale computing infrastructures. However, this promise is not being held in fog computing platforms which are often made of very small computers such as Raspberry PIs. In such environments, deploying even a very simple Docker container may take multiple minutes.

We studied the Docker container deployment process in details and identified three sources of inefficiency: (1) Docker downloads multiple layers in parallel; (2) it uses single-threaded decompression; and (3) it sequentially downloads, decompresses and extracts any given image layer. We proposed three optimization techniques which, once combined together, speed up container deployment roughly by a factor 4. Last but not least, we demonstrated that these optimizations also bring significant benefits in regular server environments.

This work eliminates the unnecessary delays that take place during container deployment. Depending on the hardware, deployment time is now basically dictated only by the slowest of the three main resources: network bandwidth, CPU, or disk I/O. As hardware will evolve in the next years the bottleneck may shift from one to the other. But, regardless of the specificities of any particular machine, Docker-pi will exploit the available hardware to its fullest extent.

References

[1] The mobile edge cloud testbed at IRISA Myriads team. YouTube video, January 2017. `https://www.youtube.com/watch?v=7uLkLitiSPo`.

[2] A. van Kempen *et al*. MEC-ConPaaS: An experimental single-board based mobile edge cloud. In *Proc. IEEE Mobile Cloud*, 2017.

[3] Arif Ahmed. Docker-pi. `https://gitlab.com/aahmed/docker-pi-v18.06.git`, December 2018.

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

[5] I. Babrou. Docker registry bay. `https://github.com/bobrik/bay`, 2016.

[6] D. Balouek *et al*. Adding virtualization capabilities to the Grid'5000 testbed. In I. I. Ivanov *et al*, editor, *Cloud Computing and Services Science*, volume 367. Springer, 2013.

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

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

[9] Evangelia Sitaridi *et al*. Massively-parallel lossless data decompression. In *Proc. ICPP*, 2016.

[10] Wajdi Hajji and Fung Po Tso. Understanding the performance of low power Raspberry Pi cloud for big data. *Electronics*, 5(2), 2016.

[11] https://httpd.apache.org/. Welcome! - the apache http server project. `https://httpd.apache.org/`, December 2018.

[12] ACME Laboratories. http-load. `https://acme.com/software/http_load/`, December 2018.

[13] M. Ma *et al*. Dockyard – container and artifact repository. `https://github.com/Huawei/dockyard`, 2017.

[14] P. Bellavista *et al*. Feasibility of fog computing deployment based on Docker containerization over RaspberryPi. In *Proc. ICDCN*, 2017.

[15] Klaus Post. klauspost/pgzip: Go parallel gzip (de)compression. `https://github.com/klauspost/pgzip`, October 2017.

[16] S. Nathan *et al*. CoMICon: A co-operative management system for docker container images. In *Proc. IC2E*, 2017.

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

[18] The Go Authors. Source file src/compress/gzip/gunzip.go. `https://golang.org/src/compress/gzip/gunzip.go`, 2017.