Investigating post-creation port mapping of Docker containers

Gideon Landeman
Abstract

Investigating post-creation port mapping of Docker containers

Gideon Landeman

Container technology allows for applications to be run in loosely isolated virtual environments. Over the past couple of decades, containers have rapidly evolved, and their use has become increasingly prevalent. One of the foundational container principles dictates that a container shall not be reconfigured after its creation. Rather, desired alterations ought to be realised through replacing outdated containers. Consequently, the widely popular container management platform Docker does not support the changing of a container’s port mapping after container creation. In other words, Docker does not allow for post-creation container port mapping without the use of container replacement. Even so, there have not yet been any recorded attempts to evaluate whether adding post-creation container port mapping without replacement might improve the delay associated with changing a container’s port mapping.

This thesis proposes an implementation that adds no-replacement post-creation port mapping capabilities to Docker. Upon evaluating the implementation against the current state-of-the-art method (post-creation port mapping with replacement) the latter appears to be approximately 25% faster. These results favour the continued use of the current state-of-the-art method. Finally, this text also provides resources for how a future implementation of no-replacement post-creation port mapping may be designed. This implementation is hypothesised to be significantly quicker than the current implementation prototype.
Acknowledgements

I would like to thank my supervisor Karl Marklund and reviewer Aletta Nylén for their continuous support and feedback. Also, special thanks go to Linn Barreby, without whom this process would have been considerably less fun. Linn, you are a rock; a perfect pebble; a bracing boulder; any naturally occurring solid mass or aggregate of minerals or mineraloid matter. Thank you for trusting me with being your IT support. Finally, thank you to all my teachers.
Contents

1 Introduction .................................................. 1

2 Background .................................................. 3
   2.1 Virtualisation ............................................. 4
      2.1.1 Virtual machine overview ........................... 4
      2.1.2 Container overview ................................. 5
      2.1.3 Virtual machine networking ......................... 7
      2.1.4 Container networking ............................... 7
      2.1.5 Virtual machines vs containers ................. 9
   2.2 Docker .................................................. 10
      2.2.1 Images and containers ............................. 11
      2.2.2 General networking ................................. 13
      2.2.3 Port mapping of containers ......................... 14
      2.2.4 Open source ........................................ 16

3 Implementation ............................................. 18
   3.1 Requirements .............................................. 18
   3.2 Setting up the development environment ............. 19
   3.3 General design .......................................... 21
   3.4 Verifying the implementation .......................... 23
   3.5 Evaluation and discussion ............................. 24

4 Related work ................................................ 29

5 Conclusions and future work ............................. 29
1 Introduction

In the 1960s, system upgrades often required existing software to be re-written in order to comply with the updated hardware. This time-consuming process sparked a desire for software portability, which would be a way to avoid the perpetual cycle of rewriting the same program over and over again. Fast-forward to the late 1990s and modern virtual machines (VMs) started to gain traction [1]. By building so-called virtual machine images (VMIs), the state of a VM became both distributable and deployable [2]. As a result, companies were able to ensure the homogeneity of employees’ computer environments. All that was required was having employees deploy the same VMI, with the image acting as a blueprint.

In 2008, Linux Containers (LXC) were launched, marking the birth of modern container technology [1]. Having evolved from some similar ideas as VMs, containers still managed to distinguish themselves as viable alternatives. They did so in part by providing a more lightweight form of virtualisation. However, containers also favoured a microservices software architecture. This new perspective on application design, in comparison to the monolithic design patterns from which VM technology had emerged, placed an emphasis on decoupling functionality. Hence, systems were alleviated from having a single point of failure [3]. A real world example showcasing the dangers of single point of failure was Netflix’s database corruption of 2008. As a consequence of the breakdown, the company was unable to service its customers for three entire days. This prompted an organisation-wide shift towards a microservices architecture which took almost eight years to complete [4]. Then, in 2013, Docker was introduced, giving users access to an entire container management platform, featuring tools such as a shared online resource library and a graphical user interface [1].

Today, not only are the number of distributed applications [5], widespread internet access [6], and demands for low latency computing growing [7], but containers are rising to the occasion as the ideal tool in this new microservices world [8]. Simultaneously, demand for fine-grained container networking control is also likely to increase. Despite Docker today being the most prominent container management platform [8], it is still not possible to change the port mapping of a Docker container once it has been created without replacing the entire container [9][10]. Nevertheless, port mapping is an important feature
since it is what allows an application running in a container to be reached and potentially exposed on a broader network [11]. As it stands, the current replacement method results in a post-creation port mapping latency of anywhere from 450 to 1000+ milliseconds [12][13][14].

A hypothetical scenario that would benefit from decreased port mapping delays is as follows. Picture a distributed application with containers making up individual nodes. As an additional safety measure, the port mapping of a container changes each time a message is being sent out, with the new port number being communicated to the other party as a part of the message being sent. That way, in theory, the only agent aware of the new mapping (container 1’s) should be the receiving container (2). When responding, container 2 would then change its own mapping and send it to container 1 along with its response message. Should this theorised communication structure prove to have some merit, lowering latencies would be an important priority. For reference, other highly interactive applications, such as fast action online multiplayer games, are said to perform ideally at ping, or packet round trip times, of below 50 milliseconds. Being able to bring down post-creation port mapping delay from 450+ milliseconds would therefore be beneficial [15].

Despite the potential benefits of adding no-replacement post-creation port mapping to Docker, it is worth mentioning that the implementation goes against the container image immutability principle. This principle dictates that manipulation of deployed containers should be handled exclusively by external orchestrators [16]. One such orchestrator, and the most popular one today, is Kubernetes [8]. Yet, with container technology being less than fifteen years old, challenging assumed principles could be considered an crucial part of furthering its development [1].

Kubernetes circumvents Docker’s lack of no-replacement post-creation port mapping by wrapping containers in additional software layers. Thanks to these layers, it would appear to the Kubernetes user that no-replacement post-creation port mapping of Docker containers exists. However, this is an illusion created by clever software. What actually happens is that Kubernetes keeps the container with the outdated mapping active, while simultaneously launching a new container. Then, once the container with the altered mapping is up and running, traffic is seamlessly rerouted from the old container [17]. Still, even though total downtime might be close to zero due to the or-
chestrated hand over, the latency between the command being given and its execution completing is still present. Additionally, there do not seem to exist any documented attempts of implementing no-replacement post-creation port mapping by means of direct manipulation of the Docker software [10][18].

Nevertheless, having acknowledged the potential usefulness of lower latency post-creation port mapping of Docker containers, this thesis aims to investigate:

- How may a method for no-replacement post-creation port mapping of Docker containers be developed?

- How is performance of post-creation port mapping of Docker containers affected by using the aforementioned no-replacement alternative, as opposed to the current state-of-the-art replacement approach?

Main contribution: This thesis proposes an alternative method for port mapping of created Docker containers. This method is compared to the current state-of-the-art approach and the results are discussed. The results of these tests indicate that the preexisting method is preferable from a latency point of view. Lastly the foundation for an additional port mapping method, on-the-fly port mapping, is laid and a potential design outlined.

The paper is structured as follows. Section 2 introduces the concept of virtualisation, how this technology handles networking and Docker as a specific example of virtualisation technology. Then, Section 3 discusses what implementation attempts have been made in order to answer the thesis questions with Section 3.5 evaluating and discussing the results in depth. Then, Section 4 brings forth some related preexisting approaches before, finally, Section 5 concludes the paper reflecting on its contents and future work.

2 Background

In this section the virtualisation concept is introduced, along with two of its subgroups: VMs and containers. Following this, common networking implementations for both variations are covered. Then, a general presentation of Docker is given together with specifics regarding its images and containers, networking as well as relationship to the open source community.
2.1 Virtualisation

Using virtualisation, resources from one physical node may be distributed among multiple (smaller) virtual ones as seen in Figure 1. All the while, applications running on these virtual nodes believe to be doing so on independent physical servers. The allocation of physical resources to virtual nodes does induce some additional overhead. As a result, given two machines with identical hardware, one using virtualisation and the other not, the machine omitting virtualisation will always perform better [19]. Nevertheless, virtualisation benefits such as infrastructure scalability and flexibility have the potential to tip the scale in favour of deploying the technology [20, p. 1-5].

Figure 1: The physical resources of a host machine get distributed among smaller virtual nodes.

2.1.1 Virtual machine overview

Constituting the first mainstream implementation of virtualisation technology, VMs date back to the 1960s. In order to deploy a VM, a hypervisor is necessary. This hypervisor runs on top of a physical server and allows the VM to interact with the host’s physical hardware, see Figure 2. For exam-
ple, a 'write' operation initiated in the VM will propagate all the way to the physical storage device with the help of the hypervisor. Furthermore, each VM boasts its own operating system (OS) which provides an additional layer of isolation. Having a separate OS ensures the kernel calls by a given VM never reach the OS kernels of the other VMs running on the same hardware. Moreover, sometimes the host machine will run its own operating system. In those cases, the 'host operating system' layer is placed between the 'physical hardware' and the 'hypervisor' layers, as presented in Figure 2 [20, p. 37-43].

![Figure 2: A high-level overview of VM architecture.](image)

### 2.1.2 Container overview

An alternative to VMs is containers. Like VMs, they abstract away the underlying hardware while making it possible to, essentially, split a physical node into multiple virtual ones. However, one large difference between the two virtualisation technologies is that containers always run on top of
the host server’s operating system and all containers share the host’s kernel space \[21\]. Whereas, even when hosts of VMs have an operating system layer between the physical hardware and hypervisor layer, the VMs never share kernel space. Figure 3 showcases container instances running on top of a container engine (a layer which is not always present). This container engine, in turn, runs on top of an operating system and the physical hardware of the host machine.

Two important tools used in order to uphold the perceived separation between containers, even though they run on the same hardware and share an OS kernel, are cgroups, or control groups \[22\], and namespaces \[22\]. The former is a Linux kernel feature used for resource prioritisation and the latter makes it appear as though processes within a namespace have their own isolated instance of a global resource \[23\].

![Figure 3: A high-level overview of container architecture.](image-url)
A consequence of containers sharing kernel space is that the highest achievable level of isolation is technically lower than that of VMs. Another outcome is that running a Windows container on a purely Linux based host, or vice versa, is impossible. The reason behind this is that the calls from the container user space will not be compatible with the kernel space. Subsequently, essential resources such as computer memory and CPU time will not be accessible to the container [21].

2.1.3 Virtual machine networking

One important resource managed by the hypervisor is networking. Whilst certain applications might be content in running locally and never interacting with the outside world, most will at some point want to communicate with other VMs on the host or external devices [20, p. 37-43].

Making up a fundamental part of VM networking are the virtual switches. These come in two flavours: external virtual switches responsible for connecting virtual network interface cards (vNICs) (sometimes called network interface controllers or virtual network adapters) attached to VMs and the network interface card (NIC) of the underlying host. The second variant, internal virtual switches, connect vNICs with each other. Basically, the external switch connects VMs with their physical host (which may or may not be connected to a broader network like the Internet), while internal switches connect VMs with each other. Lastly, these switches are located in the hypervisor. Both types of virtual switches are illustrated in Figure 4 [20, p. 37-43].

2.1.4 Container networking

As modern containerisation technology is rapidly evolving, it is difficult to generalise regarding standard networking implementations [1]. However, in general most configurations that strive to connect containers with an external host utilise either virtual ethernet (veth) pairs or assigns each container a unique mac address. The distribution of mac addresses is facilitated using a technology known as mac address virtual LAN tagging (macvlan) [24].

When a packet is transmitted on any of the two devices in a veth pair, it instantly becomes available on the other device. In this sense, the veth pair
Figure 4: A generic implementation of VM networking using vNICs.

can be seen as an ethernet tunnel [25]. Veth solutions also require a bridge in order to connect the veth pairs attached to containers with the host’s physical ethernet device and each other. Figure 5 shows how veth pairs together with a network bridge may provide connectivity for containers [26].

Using macvlan, it is possible to assign unique mac addresses to containers. This process is based on stacking multiple interfaces with different mac addresses on top of the host’s ethernet interface [26]. Macvlan itself supports various modes, one of which is called bridge mode, demonstrated in Figure 6 [27]. It is the mode which most resembles a veth implementation. In macvlan bridge-mode, aside from the containers being able to communicate directly to the host’s NIC, messages between containers travel on a pseudo-bridge using host memory as a buffer. The containers themselves are assigned unique mac addresses by the host machine’s NIC [25].
Benefits of macvlan compared to veth solutions include performance increases, due to reduced overhead [27]. However, a drawback of macvlan is that it theoretically does not support as many connections as veth does. A limit to the total number of containers on a macvlan network is reached, either as the pool of available IP addresses on the subnet is exhausted or as ‘VLAN spread’ occurs. VLAN spread is when the number of unique mac addresses in a network is ‘unsuitably high’. For macvlan to work, the host’s NIC also needs to support promiscuous mode, allowing it to be assigned multiple mac addresses [28].

2.1.5 Virtual machines vs containers

To summarise, VMs generally provide the user with more control over virtual instances than containers do. In addition, a system running VMs rather than containers have the benefit of being able to deploy its virtual instances with
different OS kernels. The superior customisability of VMs also carry over to what networking features are available by default. Importantly, VMs support on-the-fly port mapping, whereas, going by the preparatory research done for this thesis, most container solutions, including Docker, do not \cite{10,9}. On-the-fly port mapping refers to being able to change the port mapping of a virtual instance without neither stopping nor destroying it. However, containers also have many advantages, two of the most prominent ones being their lightweight nature \cite{21} and microservices architecture friendly design \cite{3}.

### 2.2 Docker

Docker was introduced on the market in 2013 \cite{1}. Since then, although the product line has evolved, the original idea of a container management platform has remained. Terms that easily get confused, however, are containers, Docker containers and Docker. Plainly put, containers come in many shapes and sizes of which two include LXC and Docker containers. Yet, Docker itself can either refer to only the Docker engine \cite{20}, or the entire container man-

Figure 6: Macvlan bridge-mode for containers.
The Docker platform comprises the Docker engine, but also other products such as the Docker client, Docker hub, Docker images and more. The Docker client provides a command line interface (CLI) for sending requests to the Docker engine. The Docker hub is an online library form where Docker images may be shared among users. Lastly, these Docker images act like blueprints and, once supplied to the Docker engine they become Docker containers. Figure 7 shows the relationships between some of these terms.

The Docker engine is a client-server application installed on the host. It contains the Docker daemon used for creating, building and running applications as well as an API used for interacting with the daemon. This API extends to the Docker client which is the primary way for users to send requests to the daemon. The Docker engine and Docker daemon are sometimes seen as one and the same.

2.2.1 Images and containers

A Docker image acts as a blueprint for the Docker daemon, instructing it how to create a specific Docker container. Furthermore, Docker images are often
layered and based on other Docker images. An image might for example use a Linux Mint or Ubuntu image as a base layer. By writing a so-called dockerfile, it is possible to define the steps necessary for creating an image. Such a file might state; this image is based on the Ubuntu 18.04 image; clone this Apache web server repository then step into the cloned directory; install and launch the web server; expose port 80. An example of a simplified dockerfile is displayed in Figure 8. Using docker build, a dockerfile is compiled into a docker image [31].

![Dockerfile](image)

Figure 8: An example of the schematic structure of a layered dockerfile.

To emphasise, a Docker container is the runnable instance of a Docker image and a Docker image originates from a dockerfile. Therefore, the same image can be used to launch multiple identical containers [31]. A good analogy is to view the image as the cake recipe blueprint and the containers as the individual cakes. In the same vein, the most intuitive way of, for instance, reducing sugar content would not be to try to extract sugar from the cake, but rather to bake a new cake according to an updated recipe.
2.2.2 General networking

Docker supports a variety of ways for orchestrating container networking. Still, the underlying foundation of these is found in the Container Network Model (CNM) as described by Figure 9 [32]. Created and maintained by the libnetwork project and proprietary to Docker, the CNM aims to provide an additional abstraction layer for developers when designing Docker networking. On a high-level, libnetwork is connected to the Docker daemon and is customised by drivers. See Figure 9 [33].

![Diagram of the CNM](image)

Figure 9: An overview of the context surrounding the CNM.

The CNM has five main objects: the NetworkController, the Driver, the Network, the Endpoints and the Sandbox. Figure 10 provides an example of a CNM configuration. Each container has its own Sandbox holding the container’s networking details (routing, IP address, mac address, etc). In addition to this, one or more Endpoints connects each Sandbox to one or more Networks. A veth pair could for instance be used as Endpoints. Also, although on a conceptual level the Network is what allows communication between multiple Endpoints, it is actually the Driver that provides the con-
nectivity and isolation for containers. Finally, the NetworkController is what allows a user to bind a Driver to a Network.

As far as specific drivers go, Docker ships with five out of the box. These include; bridge, the default network driver which utilises veth pairs in a similar fashion as can be seen in Figure 5; host, where containers share their networking namespace with the host; overlay, which allows containers running on different Docker daemons to communicate without the need for OS-level routing; ipvlan, which functions in a similar way to the last driver; macvlan, which can be seen in Figure 6. However, whereas macvlan assigns separate mac addresses to each container, ipvlan does not. Instead, ipvlan relies on IP addresses to forward packets to the correct virtual devices.

### 2.2.3 Port mapping of containers

If an IP address is the building address, the port number is the apartment number. Together they are used when directing network traffic. Figure 11
displays a concrete example where a port number together with an IP address can be used to designate a packet’s destination. In this example, the packet is using a destination port 36001 in conjunction with a destination IP address 127.0.0.2, in order to reach its final destination (Application A).

Figure 11: An example of how port numbers and IP addresses may be used to direct network traffic.

By default, any created Docker container will belong to a container network powered by a bridge driver [34]. Although this driver type in theory connects containers to the host machine’s networking namespace, in practice container applications will be available on the host’s local network only if there exists a port mapping between the local container network and the host’s local network [27]. For this mapping to occur, the container must have been created with the following option: `-p <host port>:<container port>`. `docker run --name <containerName> -p 8080:80 <containerImage>` will, for example, map port 80 on the container’s local network to port 8080 on the host’s
Figure 12 provides an overview of how the CNM concepts discussed in Section 2.2.2 are implemented in the default Docker bridge driver using veth pairs and a network bridge [25][27]. Going into more detail regarding Figure 12, the web server application running in Container A is exposed on port 8082 on the local container network. This means that agents on the network, in this case both containers and the host, are able to reach the web server using 172.17.0.2:8082. Additionally, when this container was created, the option -p 8080:8082 was used. As such, the port 8082 of the local container network address 172.17.0.2 was mapped to port 8080 of the host machine’s local network [11]. In practice, this means that in order to reach the web server from either of the two ubuntu shells, 172.17.0.2:8082 or 172.17.0.1:8080 may be used. In case port 8082 of the container would not have been mapped to the host machine’s port 8080, the web server would only be accessible on 172.17.0.2:8082.

Figure 13 presents an example of making a container accessible on an external network, previously labelled ‘Physical network’ in Figure 12. In order for the application in host machine B to reach container A, it will send a request to 198.168.0.3:9000. The reason why this works is because host A has applied port forwarding in such a way that 198.168.0.3:9000 maps to 172.0.0.1:8080, which maps to 172.0.0.2:8082.

2.2.4 Open source

The moby project was created by Docker and is an open source project comprising multiple modules built by the community [35]. These modules together make up components of the official Docker product and the individual repositories serve as upstreams [36]. The intention is for the moby project to allow engineers and enthusiasts to experiment with and modify Docker-like container systems. Notable repositories belonging to moby are libnetwork, used for container networking, containerd, an industry-standard container runtime [37] and buildkit, which is mainly used for building Docker images. Figure 14 showcases how individual open source projects such as buildkit and libnetwork go on to make up components and reference assemblies of the moby project before finally being put together into solutions such as
Figure 12: An overview of container networking. Components highlighted in grayish-blue represent the concepts of the CNM and running processes are colored turquoise. The purple boxes represent ethernet devices (physical and virtual) as well as the exposed and published ports of the web server.

the official Docker engine [35]. Another important open source repository is docker/cli which serves as the official upstream for the Docker CLI. Both
the moby project and docker/cli code bases are mainly written using golang \[35\][38].

3 Implementation

In this section, the implementation accompanying this thesis is discussed. First, the requirements are outlined. Then, the development environment which was set up is described and major design choices highlighted. Finally, the implementation as a whole is evaluated.

3.1 Requirements

The following outlines the basic requirements of the implementation. Provided a container has been created and is currently stopped, it should be possible to execute:

```
docker start <containerName> -p <host port>:<container port>
```
Figure 14: Individual open source projects make up the component library and reference assemblies of the moby project, from which container solutions such as Docker may be built.

from the CLI and have this overwrite the old port mapping. For example, if port 8082 of the container used to map to port 8080 of the host, calling `docker start <containerName> -p 9080:8082` should have the container's port 8082 map to 9080 instead of 8080. This example is depicted in Figure 15. The requirements correspond to adding no-replacement post-creation port remapping to Docker.

### 3.2 Setting up the development environment

In order to work with the source code, two repositories are forked. One was `moby/moby` and the other one `docker/cli`. The moby and cli repositories can be seen as containing the 'server-side' and 'client-side' code, respectively. Worth mentioning, however, is that the 'cli' repository name refers to 'command line interface', rather than 'client'. As to what versions of the repositories were used for the implementation, the information can be
found in the commit history of the public forks that combine to make up the thesis prototype. The moby fork can be found here and the cli fork here.

By compiling the CLI code into a linux binary and then linking to it while launching the server-side, a development container is initiated. Inside the development container, the moby binary is compiled and afterwards the development version of Docker may be started. In practice, changes to the development version of Docker are made by altering the moby and cli local forks and then re-compiling the binaries. Figure 16 describes how the development environment was set up and puts the development container into the context of a laptop. First, Docker is launched on the PC. This release version of Docker is based on stable moby and cli binaries. Then, using the instructions found in the moby [36] and cli [38] official repositories, the local development versions of moby and cli (the local forks) may be compiled into binaries and linked. These development binaries comprise the development

Figure 15: A port mapping example.
version of Docker.

Figure 16: The development environment used for creating the prototype functionality.

### 3.3 General design

In order to implement support for a \(-p\) option for the `docker start` command, three main parts of the original code needed altering. These included the CLI logic, the sending-and-receiving-of-http-requests logic and the daemon handler logic. An overview of the changes made to the preexisting message passing flow of the `docker start` command can be seen in Figure 17. The figure distinguishes what functionality can be found in the cli
and moby repositories, respectively. Notably, the mention of `spf13/cobra` in the image refers to an open source CLI library [39].

Figure 17: A flow chart of the `docker start` command with added features highlighted in red.

The first step towards adding the `-p` option for `docker start` took place in the CLI logic. Here, the port mapping supplied by the user needed parsing. In practice, this corresponded to taking:

```
docker start -p 8080:80 <containerName>
```

and extracting the `8080:80`-part. Following extraction, the `8080:80` was transformed into one structure:

```
PortBindings = map[80/tcp:[{HostIP: HostPort:8080}]]
```

And one structure:

```
ExposedPorts = map[80/tcp:{]}
```
In this way, it was possible to transform user inputted port mappings into the appropriate data structures. These structures were namely, **ExposedPorts**, which mapped port numbers (`nat.Port`) to empty structs (`struct{}`) and **PortBindings**, which mapped port numbers (`nat.Port`) to arrays of port bindings (`[]nat.PortBindings`).

The second step of implementing the `-p` option for the `docker start` command was modifying some of the sending-and-receiving-of-http-requests logic. On the one hand, this encompassed having `PortBindings` and `ExposedPorts` be a part of the `POST` request sent to the Docker daemon from the client. However, the 'start container' post request, unlike the 'create container' post request, did not pass on a request body to the daemon by default. Thus, a request body was added to the start container post request. Equally, it was necessary to modify the decoding of start container post requests, since they now, in some cases, included a request body.

Finally, the last step towards adding a `-p` option to the `docker start` command was to have the newly decoded values for `PortBindings` and `ExposedPorts` be passed to the part of the daemon handling the starting of containers. In overwriting the `config` (container configuration) data structure’s `ExposedPorts` value and the `hostconfig` data structure’s `PortBindings` value, the port mapping had fully propagated from the client to the daemon. To handle the case where a clearing of the current port mapping was desired, passing `-p 0:0` was made to overwrite the old port mapping with a null mapping.

### 3.4 Verifying the implementation

The following method was used to determine the implementation’s success:

1. Launch a container containing a web server exposed on a given port.

2. Attempt to reach the web server from the host machine using the address `<container IP>:<container port>`. The container IP can be found using `docker network inspect <network name>`. The request to the web server should return successfully.
3. Attempt to reach the web server from the host machine using the address **localhost:80**. The request to the web server should **not** return successfully.

4. Stop the container.

5. Start the container using `docker start -p 80:80 <containerName>`

6. Attempt to reach the web server from the host machine using the address `<container IP>:<container port>`. The request to the web server should return successfully

7. Attempt to reach the web server from the host machine using the address **localhost:80**. The request to the web server should return successfully.

The reason why the host should be able to reach the web server using the container IP address, before the mapping takes place, is because the host NIC is directly connected to the container bridge network. This phenomena was depicted in Figure 12 and described in Section 2.2.3.

### 3.5 Evaluation and discussion

An implementation fulfilling the requirements was written. It allows for port mapping of a created container without replacement. Unfortunately, in order to meet the requirements, some backwards compatibility was lost. The reason behind the loss, was that an older version of Docker passed the `hostconfig` datastructure in the docker create post request body. However, the thesis implementation uses the request body to send the port mapping data. Due to a lack of both documentation and time, it was not deemed feasible to write a decoder in the Docker engine that would accommodate both, one of, or neither of the structures being sent.

A performance review comparing the prototype implementation to the current state-of-the-art method (recreating the container) was carried out. Figure 18 describes the timelines for port mapping by, on the one hand, the default solution, in other words using orchestrator such as Kubernetes. The delay associated with this solution stems from launching a new container. In practice, this corresponds to the time it takes to execute `docker create` and
docker start. On the other hand, the stop-start downtime corresponds to the sum of the docker stop and docker start execution times.

By constructing a bash script, it was possible measure the average port mapping execution time of both the current state-of-the-art and the implementation proposed in this thesis. First, a container running a web server was created. It was then destroyed. This step ensured the container image was available locally on the testing device. Following this, a loop was entered. For a given number of iterations two tests were performed in the order listed below. The first test, mimicking the state-of-the-art solution, consisted of:

1. launching a timer
2. launching a container hosting a web server
3. attempting to contact the web server
4. stopping the timer once a response message was supplied by the web server

The second test, replicating the stop-start solution, consisted of:

1. launching a timer
2. stopping the container that was launched in the first test
3. starting the container with the -p option with a new port mapping
4. attempting to contact the web server using the new port address
5. stopping the timer once a response message was supplied by the web server

In both cases, two different container images were used, the results were summarized, and averages from the 100 test runs were taken. These results can be found in [19].

Curiously, the execution times, for both the state-of-the-art and the stop-start solution, seemed to increase as the tests progressed. This is likely to be the result of over taxing of the testing system’s hardware (Intel Pentinum N5000, 1.10 Ghz quad-core, 8GB RAM). In addition to running on relatively weak hardware, the tests also ran in a VM. Thus, in some sense, the
tests performed were separated from the physical hardware by, not one, not two, but three 'levels of virtualisation' (VM -> Stable Docker container -> Development Docker container). During testing, the CPU reported 100% utilisation and since the tests ran for more than twenty minutes this could have contributed to thermal throttling. The exact implementation of the tests can be found in the moby fork under combTest.sh.

The tests indicate that the prototype implementation is slower than the current state-of-the-art. On average the new implementation is 33% slower when testing using a smaller container image (142 MB) and approximately 32% slower using a larger image (3.8 GB). These results are presented in Figure 20. The reason why the percentage slowdown is seen decreasing slightly could be because container creation execution time grows at a more rapid pace than container stopping execution time. Referring back to Figure 18, both methods call on docker start, but the second command call differs (docker create vs docker stop).

The difference between the percentage slowdown using the two container images is relatively small (approximately 5%). Therefore, it would be wise to perform the tests using a wider range of container images. In particular, using more complex container images could potentially give noticeably different results. Such images might, for instance, require a greater number of system calls be carried out and configurations set, before launching the web server. When comparing the two methods, one train of thought is that a larger more complex docker image could result in more equitable performance. Again, this equalization would be hypothesised to stem from docker create being taxed proportionally harder than docker stop by the use of complex images.

Going into details on the particular implementation, it could be argued that the raw input from the option -p <host port>:<container port> ought to have been passed 'intact' from the CLI to the daemon. That is to say, user input should have been parsed into ExposedPorts and PortBindings only once it reached the daemon, rather than in the CLI logic. However, the implementation of the docker start -p option was largely derived from the way docker run -p was implemented. The latter resolved the parsing of the desired new port mapping on the client-side, i.e. in the CLI. Therefore, the same choice was taken for docker start.
Although the implementation that accompanies this thesis uses a combination of `docker stop` and `docker start` to achieve post-creation port mapping of containers, another more elegant solution could also be imagined. By instead using `docker exec` on a running container, the need for both stopping and restarting the container would be removed. Potentially, this would also contribute to a shorter port mapping execution time. This implementation would correspond to on-the-fly no-replacement post-creation port mapping of Docker containers. Such an implementation is, nevertheless, still predicted to reassemble the one depicted in Figure 17. Significant changes when compared to the `docker start -p` implementation would likely only occur in the very last step. Here, in order to avoid writing to a file that is simultaneously being read, the configuration files storing the container port mapping data would first have to be locked.
Figure 19: On average, using the prototype implementation proposed by this thesis is 29% slower.

Figure 20: The percentage slowdown going from the current state-of-the-art port mapping method for Docker containers to the prototype implementation proposed by this thesis.
4 Related work

Besides using an external orchestrator to achieve the resemblance of post-creation port mapping of Docker containers, another method has existed since at least 2016. The method in question involves manually editing two container configuration files. The exact process can be described as follows [10]:

1. Stop the container
2. Stop the Docker engine
3. Grant read and write privileges to folders containing the container configuration files
4. Overwrite the old port mapping
5. Relaunch the Docker engine
6. Restart the container

Additionally, no-replacement post-creation port mapping of LXCs is discussed in some of the official LXC documentation. However, the method involves using `iptables` to redirect network traffic, rather than interacting with the LXC command line interface [18].

5 Conclusions and future work

In this thesis, an implementation of no-replacement post-creation Docker container port mapping is presented and discussed. After comparing the new implementation to the method currently in use (entirely replacing containers with outdated port mappings), the latter is found to be approximately 25% faster. In addition, the prototype implementation removes some backwards compatibility. Therefore, keeping the current state-of-the-art post-creation port mapping method is advised.

This thesis proposes suggestions for how to implement another variety of no-replacement post-creation Docker container port mapping, which is likely to provide significantly lower delays than the current method. Thus, further
investigation of such an implementation would be interesting. Finally, finding new use cases for container port mapping and employing a wider range of tests when evaluating container port mapping methods also constitute desirable future contributions.
References


Container networking. URL: https://docs.docker.com/config/containers/container-networking/. Accessed (2022/03/30).


[34] Networking overview. URL: https://docs.docker.com/network/. Accessed (2022/03/27).


