Hochschule für Technik
und Wirtschaft Berlin

University of Applied Sciences

System Calls for Containerising and Managing Processes in Linux

A Thesis

Submitted in Partial Fulfillment of the Requirements for the Degree of

Master of Science (M.Sc.) in Applied Computer Science

at the

Berlin University of Applied Sciences (HTW)

First Supervisor:	Prof. Dr. Sebastian Bauer
Second Supervisor:	Dr. Michael Witt
Author:	B.Sc. Atanas Denkov
Matriculation Number:	s0559025
Submission Date:	04.10.2022

Abstract

Linux containers are the de-facto technology underlying modern application sandboxing and orchestration. This work focuses on analysing the primary isolation abstraction that makes containers possible - resource namespaces - and to qualitatively examine their security characteristics by developing a proof-of-concept container runtime that supports rootless containers. Implementation details are provided and the system call interface is discussed. In addition, the isolation overhead of the runtime with respect to networking and filesystem I/O is quantified and presented. The results show that containers achieve native performance when interacting with disk devices, but may suffer from decreased throughput and increased lantencies due to increased variance and packet loss when interacting with each other from different network namespaces over a bridge device.

Keywords: Linux, Namespaces, Containers, System calls, Isolation, Performance

Contents

Chapter 1

Introduction

1.1 Motivation

Primitive support for multiprocessing in the form of basic context switching and dedicated input-output components was introduced in the late 1950s. Multiprocessing allowed for concurrent execution of multiple instructions at the cost of increased system complexity. Interleaved processes had a global unrestricted view of the system which inevitably led to unpredictable program behaviour. For example, programs had the ability to modify each other's memory and monopolise computer resources. Hence, to ensure correctness, every program had to carefully manage its interactions with hardware and all other processes in the system, which resulted in an unsustainable programming model.

The aforementioned issues were addressed by shifting the responsibility of resource management and process protection into a privileged control program that acted as an intermediary between hardware and user programs. This program is most commonly referred to as a kernel. The centralisation of these tasks rendered the kernel an integral part of modern computing systems. The kernel is an essential part of a system's trusted computing base because it acts as a single point of failure for all user programs. This is a particularly important issue for infrastructure providers that sell high-availability execution environments where arbitrary, and therefore, untrusted applications from different tenants coexist. If one application tampers with the kernel, then the whole system may potentially go down. Similarly, applications are able to tamper with each other, despite the protection mechanisms enforced by the kernel. To solve this problem, an additional level of indirection may be introduced between the tenant program and all other programs on the system, including the kernel, that represents an unescapable execution environment that constraints the tenant program to a limited set of resources. This concept is known as virtualisation.

Traditional virtualisation technologies emulate the hardware resources of the system and multiplexe these amongst multiple isolated kernels. Each kernel is allocated to a particular tenant and manages that tenant's application software. The overhead introduced by emulating hardware proved to outweigh the benefits of securely isolating tenant software. The emulation layer drastically impacted the ability to consolidate multiple applications on a single server. See Chapter 3 for a more thorough discussion.

An alternative virtualisation architecture arose in the last decade - operating-system virtualisation. In this architecture, the primitives for creating isolated execution contexts are provided by the operating system kernel and are assembled together by a user-space application. Tenant programs act as standard processes that have a restricted view of the system. The kernel enforces this restriction via the use of resource namespaces - a lightweight layer of indirection in the kernel that associates a process with a limited set of resources that it can access. Resources outside this set are deemed inaccessible by that process. Namespacing is done almost throughout the entire surface of the kernel's system call interface, which is the entrypoint of user-space programs for obtaining system resources. Examining the performance impact of this indirection can give an approximation of the isolation overhead associated with resource namespacing. Small isolation overheads act as proxies for the ability to consolidate multiple applications on a single server.

1.2 Objectives

The primary objective of this thesis is to conduct an experiment consisting of a set of synthetic workloads that measure response times and efficiency within a sandboxed environment created with the help of resource namespaces. The experiment executes the same workloads natively, within the standard noninterference boundary provided by the kernel to user space programs. The differences in response times and efficiency within and outside a container are used as an approximation of the overhead introduced by isolating a process from its environment. The experiment is reproducible and can be executed on any Linux system with a x86_64 or arm64 compatible processing unit.

The second objective of this thesis is to design and implement a container runtime that will be used by the experiment as the primary sandboxing mechanism. The runtime's implementation, including the system calls that it relies on, will be thoroughly discussed and its security characteristics will be qualitatively evaluated.

1.3 Content Structure

Chapter 2 introduces the fundamental axioms, or trade-offs, in virtualisation technologies. These will be referred to throughout the entire document. The same chapter also introduces the concept of resource namespaces - the primary abstraction provided by the kernel to sandbox processes.

Chapter 3 outlines the current state of research in operating-system virtualisation, highlights the relationships between noninterference, isolation and performance, and discusses modern architectures that try to maximise all three.

Chapter 4 provides a detailed description of the requirements and the software architecture of the runtime, benchmark tool, and the workloads. The measurements to be sampled from the kernel are introduced.

Chapter 5 describes the implementation of the container runtime. Furthermore, the security characteristics of the kernel's resource namespacing capabilities are discussed.

Chapter 6 contains the performance evaluation experiment. First, the hardware platform under test is discussed. Second, a set of hypotheses are developed and proven and/or disproven with the help of the benchmark tool.

Chapter 7 contains conclusive remarks, a brief summary of the results of the thesis and future work that can be done by other students in the field.

Chapter 2

Fundamentals

Section 2.1 summarises the concept of virtualisation. Section 2.2 introduces the concept of a namespace as defined in the Linux kernel and describes the individual namespace types and their effects on processes.

2.1 Virtualisation

Virtualisation is the process of abstracting the execution environment of an application into a unit that is secure, manageable and performant. The primary value propositon of a virtualisation platform is the consolidation of independent and untrusted workloads onto a single server which cannot interfere with each other or with the host, and still exhibit optimal performance characteristics. Virtualisation is, in essence, a cost reduction technique for infrastructure providers. Unfortunately, security and performance are concepts that often conflict with each other, thereby rendering virtualisation into a particularly challenging domain.

Section 2.1.1 presents the fundamental axioms that define virtualisation - noninterference, isolation, and performance. The first two axioms will be used to assess the security properties that namespaces have, while the last one will be used to define a benchmark with metrics that measure the performance capabilities of a virtualised application.

2.1.1 Axioms

Noninterference

Codd et al. [Cod59] summarise the fundamental requirements of a multiprogramming system and emphasise the concept of noninterference between processes across space and time. *Spatial noninterference* is represented by all mechanisms that protect references to memory, disk and input-output devices [Cod59]. For example, memory segmentation is a method found in operating system kernels that assigns each process a dedicated portion of physical memory that is invisible to all other processes in the system. The kernel traps any attempt made by a process to access memory outside its allocated memory segment, thereby guaranteeing spatial noninterference [SGG18]. *Temporal noninterference* refers to those mechanisms that allocate execution time and protect against the monopolisation thereof [Cod59]. For instance, CPU scheduling is a technique that decides which process shall run on a core such that the core does not idle and all processes make sufficient runtime progress [SGG18]. The scheduling semantics, paired with an interrupt mechanism that makes sure that no process has hold of the core for too long, guarantee temporal noninterference.

Isolation

Anjali, Caraza-Harter, and Swift [ACS20] define isolation as the level of dependency that a virtualisation platform has towards the host kernel. We generalise this definition and say that *isolation* is the level of dependency that one piece of software has to another. Conceptually, isolation deals with explicit vertical relationships between software, and noninterference deals with implicit horizontal relationships between processes. Isolation can be quantified by counting the lines of external source code that a software executes to obtain a particular functionality. For example, Anjali, Caraza-Harter, and Swift [ACS20] count the lines of kernel code that a virtualisation platform executes when providing services to sandboxed applications. High counts indicate a strong dependency, i.e weak isolation, towards the kernel.

Performance

Randal [Ran20] defines performance as the contention between the overhead associated with isolating a process from its environment and the benefits of sharing resources between processes, i.e fully utilising the capacity of the underlying resource pool. Anjali, Caraza-Harter, and Swift [ACS20] use the same definition and contrast the isolation mechanisms provided by three different virtualisation platforms against processing unit, memory and input-output performance metrics. In particular, the authors define an application that computes prime numbers up to a limit. Since the workload is compute-bound, processing speed is measured and compared to the number of executed lines of code that reside in the `/arch`, `/virt` and `/arch/x86/kvm` subsystems of the Linux kernel. Manco et al. [Man17] use *same-host density* as a performance metric that measures the number of sandboxed applications that can be consolidated onto a single server. In addition, *boot, pause and unpause times* are also considered to be important performance indicators for particular use cases, such as elastic content delivery networks [Kue17] [Man17] and serverless computing.

2.2 Namespaces

A namespace encapsulates a system resource and a collection of processes. Only processes that are part of the namespace can see and manipulate the resource. Namespaces are implemented in the kernel and are therefore a part of the trusted computing base. Container runtimes use them to establish a solid noninterference boundary between processes by restricting their view of the system. The system calls `clone`, `unshare`, `setns` and `ioctl`, as well as the `proc` pseudo file system are the primary interaction points between a user-space program and a namespace (See Figure 2.1).

Every process is associated with a fixed-size set of namespaces, called the *namespace proxy*, that corresponds to the resources that it can access. In user-space, the proxy is contained within the `/proc/[pid]/ns` directory as a collection of symbolic links, each pointing to an inode number uniquely identifying a namespace that wraps the process identified by `pid`. Currently, there are seven different types of namespaces reflected in the proxy. Each will be discussed individually in the upcoming sections.

The lifecycle of a namespace is tightly coupled to the lifecycle of all processes that reside in it. This is reflected in how namespaces are created and destroyed. A namespace and a process can be atomically created via the `clone()` system call. `clone()` creates a child process that begins its lifecycle by executing a function pointer that returns the exit code with which the child process will terminate. In addition, the caller passes a set of configuration options that define the child's execution context. The parent can put the child in a freshly-allocated set of namespaces by ORing a set of flags and injecting the result into the configuration set. It is important to note that the configuration set is very flexible and defines the noninterference boundary between the child and its parent. The parent can use it to share its virtual memory pages, file descriptor table and file system information with the child, thereby weakening the boundary. `unshare()` allows an already existing process to regulate its noninterference boundary by "disassociating parts of its execution context". In other words, a process can reverse the configuration options used when it was created. Hence, the process can detach itself into a new set of namespaces. Alternatively, `setns()` can be used by a process to join a namespace that already exists. The caller must have an open file descriptor referencing a namespace object from the `/proc/[pid]/ns` directory. Notice that a namespace cannot be created without having a process to reside in it. A namespace is implicitly destroyed by the kernel when all processes inside it terminate. However, a user-space process can keep a namespace alive by explicitly referencing it through a file descriptor obtained from the proxy.

Some namespaces are represented as n-ary trees and can be nested up to 32 times in order to create multiple noninterference boundaries. Relationships between namespaces can be queried via the `ioctl` system call that, in combination with the `NS_GET_PARENT` flag, returns an initialised read-only file descriptor pointing at the parent namespace.

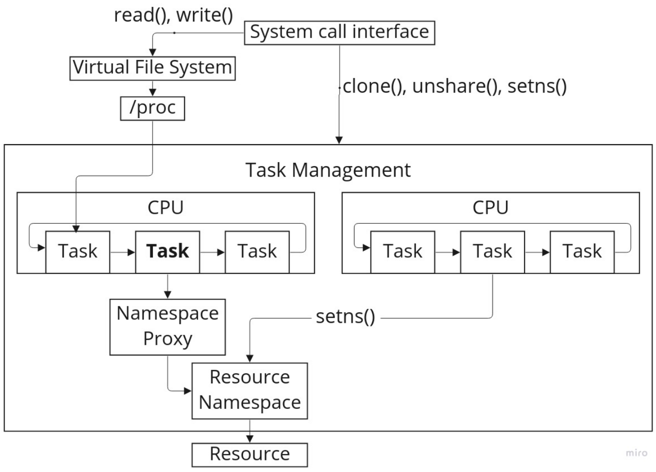

Figure 2.1: Namespace design inside the kernel. Every task is associated with a namespace proxy that can be manipulated through the virtual file system by reading and writing to files, or through the system call interface

2.2.1 User namespace

Every process is associated with a set of credentials. The traditional UNIX security model defines these credentials as a set of unique numerical user and group identifiers. The *real* user and group identifiers represent the user to which the process belongs. They are read from the password database as part of the login procedure [Ker10]. Every process spawned by the user inherits the real user and group identifiers as part of its credentials. The credentials themselves further consist of an *effective* user and an effective group identifier, which are used by the kernel to authorise and execute operations on behalf of the process [Ker10]. For example, when a process attempts to read a file, the kernel checks if the effective user ID of the process matches the user ID of the owner of the file, or if the effective group ID matches the owner's group ID. If any of these predicates holds, then the effective user - and therefore the process - is deemed trustworthy to call the `read` system call on a file descriptor referencing the file. Typically, the effective user and group identifiers match their real counterparts. However, there are programs that can ambiently change the effective identifiers of a process to match those of the user that created them [Ker10]. These programs are known as *set-user-id* and *set-group-id* programs. Every executable file is associated with a set-user-id and a set-group-id permission bit that can be set by that file's owner. If the set-user-id permission bit is set and another user executes the program, the process obtains the effective user identifier of the file's owner, not the user that started the process [Ker10]. A classic example of a set-user-id program is `sudo`, which executes arbitrary commands as the root user. Since the `sudo` executable file is owned by the root user and has the set-user-id bit toggled, any user cam execute any command in a process with full privileges. Another example is the `passwd` utility which allows a user to change her password and therefore requires write access to the password database.

Notice that the UNIX security model differentiates only between privileged and unprivileged processes. The former are not subjected to permission checking and have full control of the system - they can reboot the system, set the system time, kill other processes and so on. If a set-user-id or a set-group-id program exhibits unexpected behaviour, either due to malicious manipulation or programming errors, this coarse-grained security model is unable to prevent a full system exploitation. The Linux capability scheme tries to solve this problem by unbinding privileges from the root user and dividing them into components called *capabilities*. A capability is represented as a single bit that is a part of a 64-bit wide bit mask, called a *capability set*. Every process is associated with a *permitted*, *inheritable*, *effective*, *bounding* and *ambient* capability set. In this model, the effective capability set is used by the kernel to do permission checking. The other sets define what can be stored in the effective set when loading executable files that happen to also have capabilities. The traditional UNIX credentials, paired with the capability sets, represent a process's security context.

A user namespace encapsulates the security contexts of its resident processes. When a process creates a new user namespace, it can map its real user identifier on the host system to the user identifier 0 in the new user namespace. In other words, a non-privileged user on the host system can become `root` within its own user namespace. The kernel associates every resource and thus every other namespace type with a user namespace. A process is granted access to a resource in accordance with its credentials in the user namespace that owns the resource. Hence, on its own, a user namespace is not particularly useful because it does not own any resources. If created in pair with other namespaces, however, it becomes extremely useful.

```
$ id -u && cat /proc/$$/status | grep CapEff
1000
CapEff: 0000000000000000
$ unshare -U -r
$ id -u && cat /proc/$$/status | grep CapEff
0
CapEff: 000001ffffffffff
$ hostname mynewhostname
you must be root to change the host name
$ unshare --uts
$ hostname mynewhostname
$ hostname
mynewhostname
```

Code snippet 2.1: Example of resource ownership semantics with user namespaces

Example (2.1) highlights how the kernel authorises operations in a user namespace. First, the security context of the current process is shown. The user running the shell has no capabilities. The `unshare -U -r` command moves the shell into a newly-created user namespace. We can see that the process has obtained a superuser security context with a full effective capability set. Afterwards, an attempt is made to set the host name of the computer to an arbitrary string. The operation fails, indicating that the process has insufficient privileges, even though the required capability is in its effective set. Why? The kernel uses an additional namespace for encapsulating UNIX time-sharing (uts) operations. The shell's uts namespace is owned by the initial user namespace, in which our user is unprivileged as shown on lines 1-3. To make the new user namespace useful, we encapsulate the hostname resource in a new uts namespace and move the shell process there. Afterwards, the kernel uses the security contexts of the new user namespace to authorise any UNIX time-sharing operations.

Before the advent of user namespaces, a container runtime was required to run with superuser privileges, because the `clone()` and `unshare()` system calls require the `CAP_SYS_ADMIN` capability. Now, if the `CLONE_NEWUSER` flag is set, both system calls guarantee that the process will be moved in a new user namespace first, and its new security context will be used to authorise the creation of the remaining set of namespaces.

2.2.2 Process identifier namespace

Process identiifer (PID) namespaces encapsulate the mechanism of assigning unique identifiers to their resident processes. Every PID namespace localises an associative array, called the identifier registry, capable of allocating up to 2^{22} unique numbers and mapping them to arbitrary pointers. Because of this localisation, processes in different namespaces can be mapped to the same identifier. The first process to join a PID namespace receives the process identifier 1 and acts as the init process for the entire namespace. This process becomes the parent of any orphaned children inside the namespace. Furthermore, if the init process terminates, all processes in the namespace terminate as well. In this case, even if a file descriptor to the namespace is kept open, i.e the namespace is kept alive by the kernel, no process is allowed to join.

Process identifier namespaces are organised into a hierarchy, as shown by Figure (2.2). When a process calls `unshare` with the `CLONE_NEWPID` flag set, the kernel does not detach the process into a new PID namespace. Doing so would require changing the identifier of the process. Instead, the kernel caches the new namespace in the process's namespace proxy and spawns the process's future children into it. The new process namespace is accessible from user-space via the `/proc/[pid]/ns/pid_for_children` file.

We denote N_i as the i-th process in namespace N. In Figure (2.2), A_1 forks two children A_2 and A_3. A_3 calls `unshare` and registers a new PID namespace in the kernel. The process subsequently forks A_4, which the kernel translates into B_1. Note that A_1 and A_2 can, by definition, interact with B_1 because $B_1 = A_4$. Hence, a process is visible to all of its peers in its namespace and to all direct ancestor namespaces. Conversely, B_2 and B_3 cannot see any processes in A. Joining a PID namespace is an irrevirtible operation, i.e if A_2 joins B, then it cannot go back.

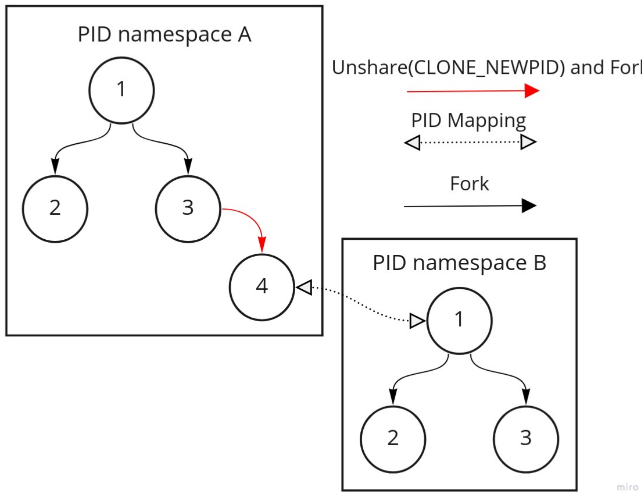

Figure 2.2: PID namespace hierarchy

PID namespaces can be nested arbitrarily up to 32 times. Container runtimes do not utilise this feature, because application workloads are orthogonal to each other. Nesting two application workloads, potentially stemming from two different tenants, in a hierarchy would enable processes resident in the ancestor namespace to kill the init process of the child namespace, which is in direct conflict with the noninterference property.

```
1  /* pid_namespace.c */
2  int err = unshare(CLONE_NEWPID);
3  pid_t child = fork();
4  if (child == 0)
5      child = fork();
6      if (child == 0)
7          execlp("sleep", "sleep", "60", (char *) NULL);
8      else if (child > 0)
9          exit(waitpid(child, NULL, 0) != child);
10 else
11     return waitpid(child, NULL, 0) != child;
```

Code snippet 2.2: PID namespace creation pseudocode

Code snippet (2.2) demonstrates the creation of a new PID namespace with two processes. On lines 1 and 2, a new pid namespace is created through the `unshare` system call and its init process is forked. The init process creates a new child that sleeps for 60 seconds. All processes within this hierarchy await their children's completion. We can run this example in a shell and list the shell's children via the `ps` utility, as shown in Code snippet (2.3). You'll notice that the `sleep` command is visible from the ancestor namespace and can be killed, i.e directly interfered with.

```
$ ./pid-namespace &
[1] 78972
$ ps
PID TTY TIME CMD
62983 pts/1 00:00:00 bash
78972 pts/1 00:00:00 pid-namespace
78973 pts/1 00:00:00 pid-namespace
78974 pts/1 00:00:00 sleep
79027 pts/1 00:00:00 ps
```

Code snippet 2.3: Example of creating a PID namespace and listing all processes in the child namespace from a process in the parent namespace

2.2.3 Mount namespace

Files are multiplexed across multiple devices, each with its own intrinsic implementation details for storing and managing data. The kernel abstracts the underlying idiosyncrasies by arranging all files into a hierarchy that can be accessed through a well-defined programming interface. The process of attaching a device's file system to this hierarchy is called *mounting*. The position in the hierarchy where the file system is attached is referred to as a *mount point*.

Mount namespaces encapsulate the list of mount points that their resident processes can see. When a process detaches into a new mount namespace, it inherits an exact replica of the parent's mount points. This does not necessarily mean that a change made by the parent to an underlying mount point remains invisible to the child, or vice versa. Every mount point is associated with a dedicated *propagation type* that determines whether or not mount and umount events are shared across mount namespaces that reference it or any mount points immediately below it in the file hierarchy. In essence, the propagation type governs the file system's noninterference boundary. Table (2.1) summarises the possible propagation types. The `/proc/self/mountinfo` file displays a multitude of mount point properties with respect to the mount namespace in which the process reading the file resides in.

In addition to mounting device file systems onto the file hierarchy, we can make already mounted subtrees visible at other locations in it. This concept is known as a *bind mount*. Bind mounts are commonly used by container engines to map directories on the host system to directories inside a container. Internally, the kernel copies the data structures of the original mount into a new mount point inside container's mount namespace. Changes in one mount point are reflected in the other because they point to the same underlying data.

Propagation Type	Description
`MS_SHARED`	Mount and umount events are propagated across mount namespaces.
`MS_PRIVATE`	Mount and umount events are local. The mount point does not propagate, nor does it receive mount and umount events.
`MS_SLAVE`	Mount and umount events are propagated unidirectionally from a master set of mount points to a slave mount.
`MS_UNBINDABLE`	Mount point is private and cannot be bind mounted

Table 2.1: Table of mount point propagation types in Linux

2.2.4 Network namespace

Network namespaces encapsulate the full network stack and expose it only to their resident processes. This allows a collection of processes to operate on dedicated network interfaces, routing tables, firewall rules and UNIX domain sockets that are inaccessible to others.

The kernel imposes an important restriction. A physical network device can be attached only to a single network namespace. When the network namespace is released, the phyiscal device is returned back to the initial network namespace, i.e the host. To enable socket-based interprocess communication across network namespace boundaries, the physical devices need to be multiplexed on top of virtual interfaces.

All container engines implement the same default network configuration that allows processes in different network namespaces to communicate with each other. As part of its initialisation procedure, the container engine installs a bridge interface inside the root network namespace. A bridge can be thought of as a virtual network switch. It forwards packets between interfaces, called slaves, that are attached to it. Before detaching a process into a new network namespace, the container engine creates a virtual ethernet device. Virtual ethernet devices are a software equivalent of a patch cable. They are represented by a pair of interconnected endpoints. One end of the virtual ethernet device is moved into the network namespace of the process, and the other remains on the host and is attached to the bridge. Because the bridge acts as a switch, other network devices attached to it can communicate with a namespaced process. Figure (2.3) visualises such a network configuration.

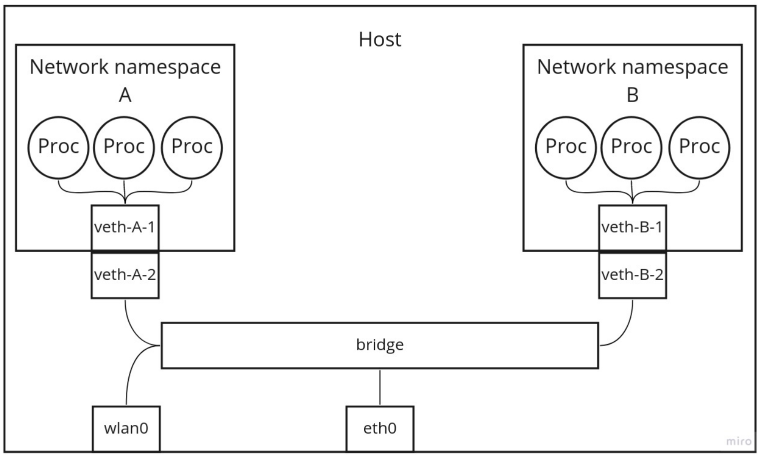

Figure 2.3: Network architecture that enables inter-container communication via bridge and virtual ethernet devices.

In it, two network namespaces, and a wlan interface inside the root namespace, can exchange data over the bridge. Code snippet (2.4) shows how to configure such a network.

```
1  $ ip netns add A
2  $ ip netns add B
3  $ ip link add veth-A-1 netns A type veth peer name veth-A-2
4  $ ip link add veth-B-1 netns B type veth peer name veth-B-2
5  $ ip netns exec A ip addr add "192.168.1.1/24" dev veth-A-1
6  $ ip netns exec B ip addr add "192.168.1.2/24" dev veth-B-1
7  $ ip netns exec A ip link set veth-A-1 up
8  $ ip netns exec B ip link set veth-B-1 up
9  $ ip link set veth-A-2 up
10 $ ip link set veth-B-2 up
11 $ ip link add name br-AB-0 type bridge
12 $ ip link set br-AB-0 up
13 $ ip link set veth-A-2 master br-AB-0
14 $ ip link set veth-B-2 master br-AB-0
15 $ ip addr add 192.168.1.10/24 brd 192.168.1.255 dev br-AB-0
```

Code snippet 2.4: Commands for configuring the network as shown in Figure (2.3)

On lines 1-2, two network namespaces A and B are created. On line 3, a virtual ethernet cable is created, whose ends are represented by `veth-A-1` and `veth-A-2`. The former is injected into namespace A, whilst the latter is kept in the root namespace. The same process is executed on line 4 for namespace B. The isolated ends of the virtual devices are initialised and assigned unique IP addresses that are a part of the same logical network. The other ends are initialised as well. On lines 11-14 a bridge is created and the virtual ethernet cables are attached to it. At this point, A and B can communicate effectively with each other. The host, however, interprets the bridge solely as an L2 packet forwarder that cannot interact with the other namespaces. On line 15, the bridge is made a participant of the isolated network by being assigned an IP address. Now, both namespaces and the host can communicate with each other. However, a mechanism for routing traffic from the bridge to a different node in the host network or the internet is missing.

2.2.5 Interprocess communication namespace

An interprocess communication (IPC) namespace encapsulates System V objects and POSIX message queues and makes them visible only to its resident processes. Every IPC namespace has a dedicated POSIX message queue filesystem that cannot be accessed by processes outside the namespace. When all processes in an IPC namespace terminate, all IPC objects are automatically destroyed.

2.2.6 UNIX time sharing namespace

UNIX Time-Sharing (uts) namespaces encapsulate the system's host name and domain name. When creating a new uts namespace, the kernel simply copies the system identifiers of the previous namespace and exposes the copied versions to all residents of the namespace. The residents are free to change both identifiers without affecting the actual values on the host. Example (2.1) already highlights how to detach a shell into a new uts namespace and set its host name.

Chapter 3

State of research

Operating system virtualisation refers to all mechanisms that enable the creation of secure and isolated application environments that run on top of a shared kernel. Conventionally, these mechanisms are baked into the kernel and are therefore part of the trusted computing base. The kernel may expose these through its system-call interface, thereby allowing a user-space daemon program to provide an automated facility for creating and orchestrating sandboxed environments. This is the only feasible architecture on a general-purpose kernel such as Linux. Alternatively, the kernel may treat every software component, including its own subsystems, as an entity to be wrapped in a sandbox. In that case, the concept of a process itself would have to satisfy all three virtualisation axioms. Examples of such operating systems include the seL4 *microkernel* - the first operating system to be formally verified as free of programming errors [Kle09], and Google's Fuchsia - described by Pagano, Verderame, and Merlo [PVM21].

An application, referred to as a *container*, is defined as an encapsulation of "[...] a software component and all its dependencies in a format that is self-describing and portable, so that any compliant container runtime can run it without extra dependencies, regardless of the underlying machine and the contents of the container" [Ini16c, p. 1]. A *container runtime* is the user-space daemon program responsible for bringing this portable but static representation of an application into execution. In runtime, a container consists of a collection of processes that share a restricted view of the system's resources. For example, every container "believes" it has a dedicated network stack with its own network interfaces, routing tables and packet-forwarding rules. All processes in the container can access and manipulate that network stack, but no other process outside the container has that capability. The container runtime configures this invariant and the operating system enforces it by namespacing system resources. The Open Containers Initiative (OCI) [Ini15] has developed a runtime specification that standardises the operations a container runtime needs to support. Most importantly, it must allow an external process called a *container engine* to hook into the lifecycle of a container. The container engine can use these hooks to manage all the containers on a single host system. In addition, the engine can attach network and storage to containers, thereby allowing processes in different sandboxes to share state and communicate with each other, if required. At the highest level of abstraction sits an orchestration platform that manages containers on multiple hosts by interacting with the container engine on each system. This platform constantly monitors node and container health and dynamically multiplexes workloads based on various system properties of the cluster as to ensure maximum application availability.

It is important to note that, by definition, the kernel is assumed to be trustworthy. It has full control of all hardware resources and can access and modify the execution environment of every process on the system. In other words, noninterference between the kernel and user processes is not guaranteed. Therefore, if the kernel is compromised, all processes on the system become untrustworthy. It follows that if a process compromises the kernel, it transitively interferes with all other processes on the system. Hardening the operating system by implementing various security features such as mandatory access control has been the primary approach for protection against such scenarios. However, the size of a monolithic general-purpose kernel is too large to adequately create a threat model that captures all possible vectors of attack. This problem is of particular concern to infrastructure providers whose entire business model revolves around consolidating hundreds of potentially malicious client applications on the same server, all of which share the same kernel and are allowed to directly interact with it via its system call interface.

Unlike hardware virtualisation, this architecture does not use hardware emulation as an isolation primitive. This means that shadow pages need not be maintained per virtual environment. Input-output operations need only traverse the kernel's stack without any address translations and with the additional performance benefit of direct memory access. As a result, the isolation overhead is lower compared to a virtual machine, which allows more applications to be consolidated onto a single server. Furthermore, guests do not boot up complete operating system images, which reduces start times and memory consumption. Priedhorsky and Randles [PR17] use containers in high-performance computing clusters to run user-defined compute jobs and show that the imposed performance penalties are, at most, negligible compared to vanilla processes that have no additional isolation. Felter et al. [Fel15] show the exact same thing and further conclude that the Docker container engine is resource-friendlier and faster than the Kernel Virtual Machine (KVM) when stressing the "memory, IPC, network and filesystem subsystems" Felter et al. [Fel15, p. 1] of the Linux kernel by running a database inside a virtual environment and evaulating its performance via the SysBench OLTP benchmark [ZK04].

Google's gVisor [Goo18] attempts to sustain the performance advantages of containers whilst introducing an additional isolation boundary between the kernel and each container. The authors implement a substantial portion of the system call interface in a user-space process called Sentry. Their dedicated container runtime calls out to Sentry instead of the kernel when issuing system calls.

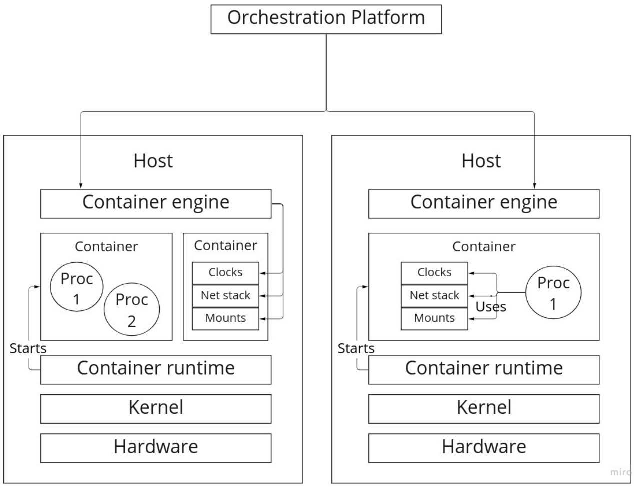

Figure 3.1: Operating system virtualisation architecture using containers. The container runtime starts containers on a single host. A user process can see a bundle of resources allocated to it by the kernel. The kernel guarantees that a process cannot see any other resources. The container engine manages all containers on a single system and allocates storage and networking to create explicit paths between containers. The orchestration platform talks to all engines inside a cluster to provide automatic workload management.

However, Young et al. [You19] show that network and memory allocation performance greatly suffer. This can be partially attributed to the fact that the Sentry process is implemented in a garbage-collected language and lacks the fine-grained optimisations contained in the Linux kernel. Agache et al. [Aga20] take a different approach and try to fuse the security of virtual machines with the performance of containers by programming a custom virtual machine monitor called Firecracker that runs on top of KVM. Firecracker completely relies on the Linux kernel for memory management, CPU scheduling and block I/O. To reduce its attack surface, the virtual machine monitor sacrifices portability by supporting a limited set of emulated network and block devices. To further strengthen the noninterference boundary, the devices have configurable built-in rate limiters that can control the number of operations per second, e.g disk/packets per second. Unlike a traditional container runtime, Firecracker's rate-limiting implementation does not rely on the kernel, which makes its isolation boundary to the kernel stronger. Anjali, Caraza-Harter, and Swift [ACS20] evaluate both gVisor and Firecracker and show that the latter "[...] is effective at reducing the frequency of kernel code invocations, but had a much smaller impact on reducing the footprint of kernel code" [ACS20, p. 12].

Popek and Goldberg [PG74] refer to the control program as a *virtual machine monitor* that ensures isolation and noninterference by providing every program with an environment that is "[...] effect identical with that demonstrated if the program had been run on the original machine directly" [PG74, p. 2]. This definition implies that a running program does not directly use the bare metal resources available. Instead, resources are emulated by the virtual machine monitor at the hardware level and presented as a dedicated physical system. Such an environment is called a *virtual machine*.

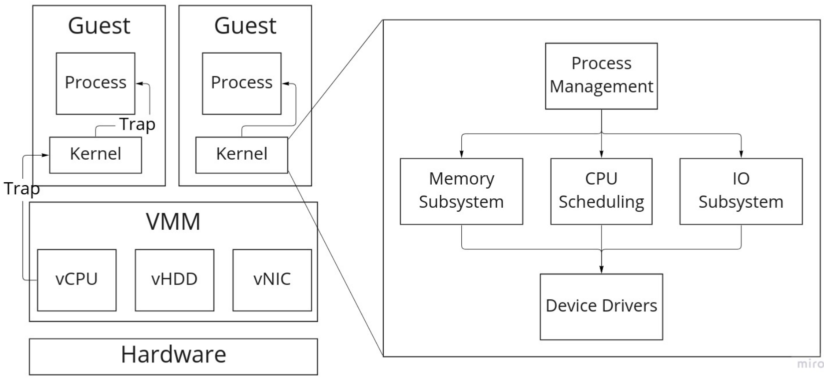

Figure 3.2: Hardware virtualisation architecture. Each guest runs a complete operating system. Privileged operations are trapped by the virtual machine monitor and emulated to provide hardware services.

Popek and Goldberg [PG74] define a requirement that the instruction-set architecture of a computer has to satisfy for it to be virtualisable. The instruction set must be segregated into three groups of instructions - privileged, sensitive and innocuous. An instruction is privileged if it requires changing the mode of execution from user to supervisor mode by means of a trap [PG74]. An instruction i is control-sensitive if, when applied to the current processor state S_1, results in a new state $i(S_1) = S_2$ such that the execution mode of S_2 does not equal that of S_1 or if S_2 has access to different resources than S_1 or both [PG74]. An instruction is behaviour-sensitive if its execution depends on the execution mode or its position in memory [PG74]. An instruction is innocuous if it is not sensitive. Given these definitions, a computer is virtualisable "[...] if the set of sensitive instructions for that computer is a subset of the set of privileged instructions" [PG74, p. 6]. If this criterion is met, the virtual machine monitor can trap all sensitive instructions and emulate each via a homomorphism $i : C_r \to C_v$ that maps the state space of the processor without the virtual machine monitor loaded C_r to the state space with the virtual machine monitor loaded C_v [PG74]. Innocuous instructions do not require protection, i.e a homomorphic mapping, and are directly executed by the processor [PG74].

Given the aforementioned homomorphism, a virtual machine can host a *guest kernel* (Figure 3.2) that runs completely in user mode. Whenever the guest kernel attempts to execute a privileged instruction, the virtual machine monitor traps the attempt and emulates the instruction. Consequently, the guest kernel does not have to be a part of the trusted computing base. Even if it is compromised or encounters an unrecoverable error condition, other virtual machines remain unaffected. As a result, the isolation boundary between user programs running in different virtual machines is stronger compared to processes running on a shared kernel.

In order to fully guarantee spatial noninterference between processes, the virtual machine monitor must be in full control of the host system's memory. There are two primary methods to do this - *shadow paging* and *extended page tables.* The former mechanism is considered first. The virtual machine monitor maintains a nested page table per guest, also called a *shadow page table* [SN05]. In turn, the guest kernel maintains a page table per process. Whenever the guest kernel schedules a new process for execution, it modifies the *page-table base register* to point to the page table for that process [SN05]. The virtual machine monitor intercepts this attempt and transparently updates the page table pointer to point to the guest's shadow page table corresponding to that process [SN05]. Note that the virtual machine monitor has to traverse the shadow page table for that guest in order to find the nested entry corresponding to the process. Afterwards, the memory management unit takes care of translating the virtual memory addresses of the guest and updating the *translation lookaside buffer.* Alternatively, the memory management unit may be "virtualisation-aware" in the sense that it knows there are two page tables it needs to traverse - the page table that maps guest virtual memory to guest "physical memory", and the page table that maps guest physical memory to actual physical memory. The former is maintained by the guest kernel, whilst the latter is maintained by the virtual machine monitor. The extended page table approach is up to 50% faster than shadow paging [Esx06] because table walks are done in hardware - by the memory management unit. Nevertheless, maintaining page table data structures inside the virtual machine monitor and the guests leads to memory pressure, which is further amplified by the fact that guests, their applications and the virtual machine monitor all share the same physical memory [SGG18].

The spatial noninterference property necessitates that the virtual machine monitor manage all input-output devices and their interactions with the guests. This is accomplished by the already introduced trap-and-emulate pattern. When an application within a virtual machine issues a system call requesting some form of input-output, the request is processed by the I/O stack inside the guest. At the lowest level of the stack, the device driver issues a command to the device, typically by writing to memory specifically assigned to the device, or by calling specific input-output instructions [SGG18]. Either way, the virtual machine monitor intercepts this and traverses its own I/O stack, which remaps guest and real input-output addresses and forwards the request to a physical device [WR12]. After processing the request, the physical device triggers an interrupt that is caught by the virtual machine monitor and transformed into a virtual equivalent that is sent to the virtual machine that issued the request. To reduce the overhead associated with interrupt processing, the virtual machine monitor can batch multiple events together and use a single interrupt to notify the guest kernel [WR12]. Still, a request must traverse two input-output stacks. The same holds for the response. In addition, hardware optimisations such as direct memory access are emulated in software, which further degrades performance. This, however, can be mitigated by integrating an input-output memory management unit that remaps all direct memory accesses of a device on the host to an address space in the guest.

The cost of hardware virtualisation becomes apparent when measuring same-host density and boot times. Manco et al. [Man17] consider memory consumption and on-disk image size as the primary limiting factors. The authors measure the time it takes to create and boot virtual machines using the Xen virtual machine monitor and show the negative effects that on-disk image size has by starting images with varying sizes by manually "[...] injecting binary objects into the uncompressed image file" [Man17, p. 3]. As the number of consolidated virtual instances increases and the image size grows, creation and boot times increase linearly. Furthermore, the authors show that creating and starting a process directly on the host is, on average, two orders of magnitude faster. Lv et al. [Lv12] also evaluate Xen and state that processing units spend 25% of their total cycles in hypervisor mode instead of executing guest applications when running "[...] SPEC's first benchmark addressing performance evaluation of datacenter servers used in virtualised server consolidation" [Lv12, p. 2], which includes components such as a web, database and application server.

Chapter 4

Concept

This chapter introduces the functional and non-functional requirements of two applications - the container runtime and the benchmarking tool. In addition, architectural diagrams are provided and discussed. Example usage of both applications is shown. This chapter also contains a thorough justification of the workloads deployed by the benchmarking tool as well as the variables used to measure the performance of the sandboxed workloads.

4.1 Container runtime

The container runtime is the component responsible for wrapping a user-defined binary in a sandbox. It will be used by the benchmarking tool to wrap workloads in sandboxes.

4.1.1 Functional requirements

The Open Containers Initiative standardises the operations [Ini16a] that a container runtime needs to support.

i The container runtime must provide clients with state information for a container given its unique identifier.

ii The container runtime must provide clients with the ability to create a new container. Users must supply the runtime with a unique identifier for the container and a path to a container bundle. The latter consists of a root filesystem and a configuration file that defines, amongst other things, the path to the user-defined binary and the set of namespaces it will reside in.

iii The container runtime must provide clients with the ability to start a container. Users must provide the unique identifier of the container to start. This operation must execute the user-defined binary in the sandboxed environment.

iv The container runtime must allow users to kill the container process. Users are required to provide the unique identifier of the container and the signal to be sent to the container.

v The container runtime must allow users to delete the container. The delete operation must remove all resources allocated in (ii)

vi The container runtime must allow external applications to hook into well-defined points of a container's lifecycle.

It is important to note that the container runtime's only responsibility is to sandbox processes. An external application, such as the benchmarking tool discussed later, must have the ability to interact with the runtime for the purpose of configuring the sandboxed environment. This includes operations such as creating network topologies that interconnect different containers or allocating shared filesystems. From this, requirement (vi) has been derived.

4.1.2 Non-functional requirements

All requirements specified in this section are ranked in order of importance.

i The container runtime must not pollute or damage the host system via its operation. The runtime manages sensible resources, such as mount points and devices. It must in no way cause side effects on the host, leading to operational failure. This is also an important factor for allowing reproducibility of the work. Users must be able to use the runtime without fear of damaging their system.

ii The container runtime must support unprivileged containers, i.e containers that run without root privileges on the host system. This is in and of itself a functional requirement for running containers in multitenant environments.

iii The container runtime must be implemented in a programming language with manual memory management. Satisfying this requirement will ensure that future work aimed at measuring container boot times will not be hindered by unpredictable perturbations introduced by a garbage collector.

iv The container runtime must consist of a library component and an executable component. This will allow users to implement their own abstractions on top of the library for other research-related purposes.

4.1.3 Architecture

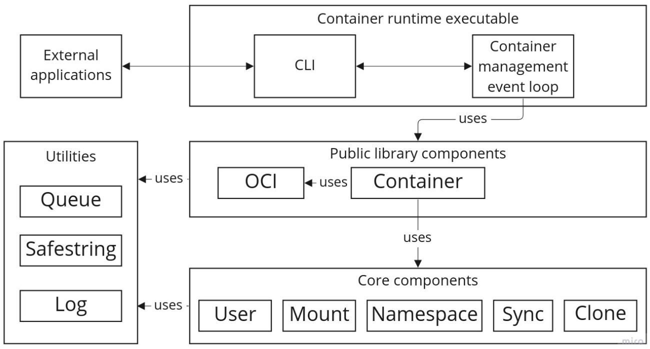

Figure 4.1: High-level overview of the runtime architecture

The architecture of the runtime is shown in Figure 4.1. It consists of a library and an executable that uses it to provide container management services to external applications. The library is split into three parts - utilities, core components and public components.

The utilities provide common data structures such as linked lists and queues. They also contain procedures for safe string manipulation, a logger, and a multitude of helper macros that enable the safe management of resources such as file descriptors and heap memory. The core components directly interface with the kernel. They are responsible for configuring the container, i.e the execution environment of the user-defined application. On top, the public library components use the core components to provide a simple interface for creating, starting, killing and deleting containers. A container is represented as an opaque pointer and is only allowed to be accessed through library functions.

The runtime executable uses the container component to create and manage a single container. It consists of an event loop that monitors state changes of the container and reaps the process when it exits. In addition, it polls a signal file descriptor for itself. When the user desides to kill the runtime executable, the same signal is propagated to the container, the container is killed, and all resources it allocated are deleted accordingly.

When a process is detached into a new user namespace, it's real user identifier is replaced by the kernel with the overflow identifier, also known as the *nobody* user, which has no access to file objects that are not world readable and writable. For this reason, the container runtime must map a range of user identifers from the root user namespace into a range of user identifiers in the user namespace of the process. The user component in Figure 4.1 is responsible for this functionality.

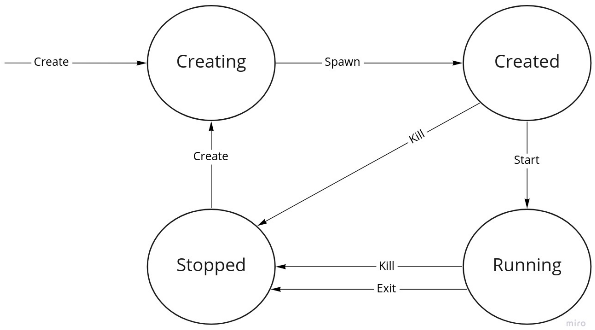

Figure 4.2: State transition diagram of container states. Every state transition is triggered by an operation, whose name is specified in the arrow.

Every container has a dedicated root filesystem with its own set of device nodes, pseudo filesystems, applications and libraries. The mount component in Figure 4.1 atomically changes the container's root mount to a user-defined directory that holds its new root file system. It then creates private mount points for all necessary pseudo filesystems, e.g `proc`, `sys`, and `mqueue`. In addition, this component creates private device nodes for the container such as `/dev/null` and `/dev/urandom`.

The namespace component simply wraps some of the namespace system calls and provides a mechanism to enumerate a contiguous sequence of namespaces.

The clone component wraps the raw `clone3()` system call and the `glibc` wrapper into a portable (at least for `x86_64` and `aarch64`) set of functions.

The Open Containers Initiative (OCI) component is responsible for parsing a JSON configuration file that defines the container's execution environment. Code snippet 8.1 is an example of such a file. In addition, this component provides a mechanism for executing an arbitrary program, called a hook, that receives the container's state through its standard input stream as a JSON string. Hooks are provided by external applications as part of the JSON configuration file and are executed when a container transitions into a new state. This design is defined as part of the runtime specification [Ini16b]. A container's state transition diagram is shown in Figure 4.2. Table 4.1 shows the predefined set of hooks and in which execution context they are executed.

Hooks	Namespace	Triggered By
`on_runtime_create`	Runtime namespace	`EVENT_RT_CREATE`
`on_container_created`	Container namespace	`EVENT_CONT_CREATE`
`on_container_start`	Container namespace	`EVENT_CONT_START`
`on_container_started`	Runtime namespace	`EVENT_CONT_STARTED`
`on_container_stopped`	Runtime namespace	`SIGCHLD`

Table 4.1: Table of hooks, where they are executed and what event they are triggered by.

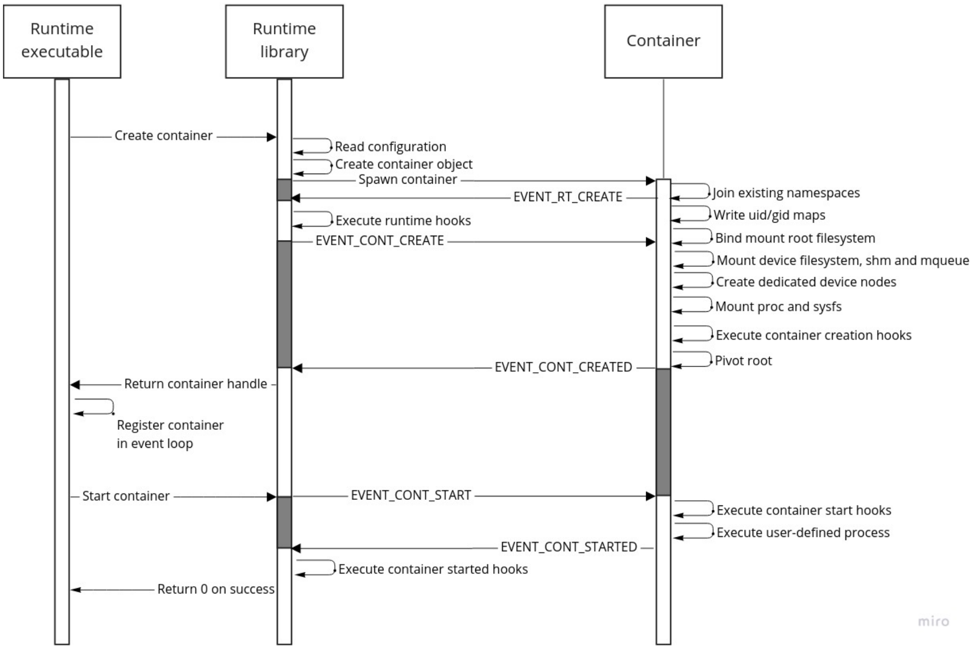

Figure 4.3: Sequence diagram highlighting the container creation and start operations

The `on_runtime_create` hooks are executed in the runtime namespace and prepare the runtime environment on the host. These hooks are triggered after the container process is cloned into a new set of namespaces. The `on_container_created` hooks are executed within the container namespaces in order to prepare the runtime environment inside the container. Note that these hooks are executed before the root filesystem is pivoted. The `on_container_start` hooks are triggered after pivoting the root filesystem, but before loading the user-defined application via `execve()`. The `on_container_started` hooks are executed within the runtime namespaces after the `execve()` is called. The process is then monitored for state changes. Whenever it exits, the `on_container_stopped` hooks are executed, whose primary task is to clean up the runtime environment on the host.

The synchronisation component defines an inter-process communication mechanism between the container runtime and a container. The typical communication flow follows the request-response pattern - one end produces a request and blocks until the other end consumes it and produces a response. However, to enable concurrent work, a peer can “fire” a message and forget about the response. The requests and responses are simple numeric values with semantic meaning.

The container component brings all of the core components together. The sequence diagram in Figure 4.3 shows the procedures executed to create and start a container. The runtime executable invokes the create operation, as defined in requirement (ii). The container component reads the configuration file and allocates an in-memory representation for the container. It then spawns the container process, which creates a freshly-allocated set of namespaces and joins already existing namespaces, depending on the namespace configuration specified in the file.

It then transitions into the "creating" state and notifies the runtime to execute the corresponding hooks by sending an `EVENT_RT_CREATE` through the synchronisation component. The runtime receives the request, executes the runtime hooks, and notifies the child that it should transition into the "created" state by executing the container creation hooks and atomically swapping the old root mount with the new root filesystem. The container process then blocks indefinitely until it is killed or it receives a start request from the runtime executable. The latter will trigger yet another hook execution procedure, it will move the container into the "running" state and will afterwards run the user-defined binary. This is detected by the runtime, a set of post-start hooks is triggered, and the runtime executable is notified of the operation's completion.

After the container is created, the executable is responsible for managing the container process. It does so by registering the container with an event loop that polls container states and reaps processes that have terminated, which happens when the container transitions into the "stopped" state - either by exiting or being manually killed by the executable through a signal.

4.2 Benchmark

Benchmarking is a tool to test performance in a controlled way, allowing different system and software configurations to be compared and analysed.

4.2.1 Functional requirements

i The benchmark must facilitate the collection of performance metrics in native and containerised environments.

ii The benchmark must aggregate and compare metrics in native and containerised environments to enable the detection of isolation overheads.

iii The benchmark must produce observable results, either through visualisations or formatted text outputs, so that they can be analysed.

iv The benchmark must expose a command-line interface that allows users to define their own container and sampling configurations.

4.2.2 Workloads & Metrics

A workload is an application aimed at stressing a particular path of execution. Workloads are executed within an environment, e.g a container or on the host. While the workload is running, a set of tracepoints accumulate performance counters relevant to that path of execution, which are then aggregated into statistics. The tracing procedures may be embedded in the workload itself, or as an external application.

The tracepoints focus on building accurate representations of two variables - latency and throughput. The former is a measure of the time spent waiting for a particular resource. The latter is a measure of the number of operations than a resource can process per unit of time. The benchmark tool, or the workload itself, attaches tracepoints to resources that are namespaced by the kernel and gathers information while the workload is running inside the container. Afterwards, the same statistics are gathered by executing the workload again, but this time as a native application absent of a noninterference boundary constructed by the container runtime. The difference in latency and throughput between both runs acts as an approximation of the isolation overhead introduced by sandboxing the workload. In Chapter 6, this approximation will be used to evaluate a set of hypotheses.

In this section, the workloads and the performance counters associated with them are described.

Network workload

The goal of this workload is to measure the overhead of data transmission between two separate network namespaces. The workload consists of a client and a server that communicate via TCP or UDP. Users have to specify the execution environment of both the client and the server. Hence, both can be deployed in two separate containers, each having a dedicated virtual ethernet cable attached to a bridge device on the host that switches packets between them. Alternatively, one of the components can run in the root network namespace, i.e directly on the host system. A third option is to have both the client and the server run on the host system without a bridge device between them. This is how native applications communicate - by being attached to the same network interface.

The workload begins by starting the server and then the client. The client begins sending out as many data chunks of size b as possible during equally-spaced intervals of n seconds for a total duration of t seconds. During the data exchange procedure, the client keeps track of round-trip time, the number of bytes transfered, the bits transferred per second, and the number of retransmissions for each interval n. The caller configures b, n and t. In additon, the caller is allowed to configure the number of parallel connections betwen the client and the server. In that case, the aforementioned statistics will be tracked for each connection. Increasing the number of connections can be used to determine the degree of saturation, i.e how many connection are necessary to observe increases in latency and decreases in throughput due to throttling. The workload also keeps a summary of the total user and system time it spends on the processing units, both for the client and the server.

Note that the number of retransmissions and the round-trip time are metrics that are tracked only for TCP because they are intrinsic to the protocol implementation in order to detect data loss and retransmit packets accordingly. Conversely, UDP is unreliable and does not maintain such information. Lost data and out of order packets must be handled at the application level.

Round-trip time is this workload's latency metric. It is an approximation of the amount of time it takes to send a packet to the server plus the time it takes to receive the acknowledgement of that packet. The round-trip time is a value maintained by the kernel's TCP implementation. The number of bytes transferred during the interval is this workload's throughput metric. It shows how much data can be forwarded from one network namespace to another via the bridge and its slave devices. This value is accumulated in user-space, whenever the `read()` system call finishes on the receiver's end the counter is incremented. The number of retransmissions acts as a robustness metric for the network itself. It also has a statistical relationship with throughput. Higher retransmission counts may have a negative impact on throughput.

Filesystem workload

The purpose of this workload is to detect and analyse differences in latency and throughput in different mount namespaces. In particular, we want to compare these metrics when running the workload within the root mount namespace and a container mount namespace.

The workload executes operations on a single file. Users can define the I/O pattern issued to the file, e.g performing only sequential reads, writing to the file at random file offsets, using direct I/O in order to bypass the page cache and so on. In addition, the size of the buffers (the block size) can be configured. How the workload should issue the I/O operations must also be specified. There are a plethora of options here. I/O operations can be synchronous, i.e employing calls to `read()`, `write()` and `lseek()` to position file offsets. Alternatively, I/O operations can be asynchronous. That is, the workload registers a set of event handlers for each I/O operation it wants to execute and only submits requests to the kernel. The kernel does not block the calling thread. Instead, it notifies the application whenever the disk driver emits an interrupt that represents the operation's completion. In asynchronous mode, users can configure the number of parallel I/O operations. Another way of controlling parallelisation is by spawning multiple processes that compete with each other. The number of processes is also configurable.

Input-output operations per second (IOPS) is this workload's throughput metric. The metric is heavily dependant on the I/O pattern employed. This work focuses on sequential reads and writes, both in synchronous and asynchronous contexts. For the asynchronous measurements, the number of in-flight I/O operations is controlled by two mechanisms - number of processes issuing I/O operations and number of parallel I/O operations by a single process. These mechanisms are used independently to ease the interpretation of the results. For the synchronous measurements, the level of parallelisation is controlled only through the number of processes.

Total latency is this workload's latency metric. The total latency is derived by summing two variables - the submission latency and the completion latency. The former represents the time it takes to allocate and initialise the data buffers plus a variable a whose interpretation depends on whether or not the underlying I/O mechanism is synchronous or not. If it is, then $a = 0$. Otherwise, a includes the time it takes to submit the I/O operation to the underlying queue of requests. The completion latency is also context-dependant. In the synchronous case, it represents the time from when the input-output operation was submitted to the kernel to when it was completed. In the asynchronous case, it is the time from when the submission operation was completed to when the event handler that reaps the operation was completed. Intuitively, synchronous operations exhibit lower submission latencies but higher completion latencies. Conversely, asynchronous operations exhibit higher submission latencies but lower completion latencies. Completion latencies have a stronger impact on overall I/O performance, making asynchronous mechanisms more favorable for performance-critical workloads.

Chapter 5

Implementation

This chapter focuses on the implementation details of the container runtime and the benchmark tool.

Section 5.1 dives deeply into the system call interface of the Linux kernel that enables the creation and execution of isolated workloads.

Section 5.2 presents the implementation of the benchmarks and the tool that manages them.

5.1 Runtime

The runtime is divided into two parts - a library and an executable. Section 5.1.1 contains information about how the library was implemented as well as a qualitative description of its security characteristics.

Section 5.1.2 shifts the focus towards the runtime executable, also referred to as a *container shim*, and its responsibilities.

5.1.1 Library

The `clone()` system call is the runtime's primary method of isolating a process. This system call creates a new execution context and allows the caller to configure it through a bit mask. In essence, this bit mask defines the noninterference boundary between the new execution context, its parent, and all other processes on the system. If the bit mask is 0, the clone system call is equivalent to a simple `fork()`. The runtime utilises this system call for two purposes - to spawn the container process in a new set of namespaces and to create a pollable process file descriptor for it.

```c
static inline pid_t clone3(struct clone_args *args, size_t args_size)
{
    return (pid_t) syscall(SYS_clone3, args, args_size);
}

static int container_entrypoint(void *arg)
{
    struct conty_container *cc = (struct conty_container *) arg;
    /* Configure container within new namespace set */
    /* At the end, call execve to load the user-defined binary */
}

int conty_container_spawn(struct conty_container *cc)
{
    struct clone_args args = {
        // cc->cc_ns_new contains
        // CLONE_NEWIPC | CLONE_NEWPID | CLONE_NEWUSER | CLONE_NEWNS | CLONE_NEWUTS | CLONE_NEWNET
        .flags = cc->cc_ns_new | CLONE_PIDFD;
        .pidfd = (__u64)(uintptr_t) &cc->cc_pollfd,
    };

    cc->cc_pid = clone3(&args, CLONE_ARGS_SIZE_VER2);
    if (cc->cc_pid == 0)
        /* Within container */
        _exit(container_entrypoint(cc));
    else
        /* Within runtime */
}
```

Code snippet 5.1: Container instantiation within the runtime using clone3

Achieving the former is trivial. The kernel exposes a flag for every namespace type. These flags are simply ORed together and passed into the system call. Internally, the kernel allocates a new namespace proxy and all of its namespaces in a system-wide slab cache and attaches it as a pointer to the new task structure. Whenever a process attempts to operate on a namespaced resource, the kernel follows its reference to the proxy, and the resources' reference to the namespace. If the resource is managed by the same namespace that the process is a part of, then the respective operation is permitted. Code snippet 2.2 shows an example of how clone is used within the runtime implementation. The namespaces are parsed from the configuration file and aggregated into a bit mask `cc_ns_new`. The runtime then calls out to the `clone3()` system call with the namespace mask set, as well as a pointer to a pollable file descriptor for the container that should be populated by the kernel - `cc_pollfd`.

Spawning a process in a new set of namespaces is not enough to provide a secure runtime environment. As a matter of fact, the container's runtime environment after `clone()` is insecure and unusable. The runtime must configure the environment depending on the namespaces that were created.

If the runtime creates a container process with the `CLONE_NEWUSER` flag set, i.e in a new user namespace, the kernel initialises the container's security context to an unprivileged state by setting its user and group identifiers to the overflow value, also known as the "nobody" user. This user has the least privileges on the system, making the new user namespace particularly unuseful. Even if the namespace manages security-sensitive resources, such as network devices and mount points, the container process is considered unprivileged to access them. For this reason, the container runtime has the responsibility of remapping the invalid identifiers to valid user identifiers. It is particularly important to note that the remapped identifiers must refer to users within the root user namespace, i.e on the host. The remapping is done by writing a range of identifiers to the files `/proc/[pid]/uid_map` and `/proc/[pid]/gid_map` where `pid` refers to the process identifier of the container. These files consist of multiple entries separated by newline characters. An entry is represented by three integers - c, h and a. The kernel maps the half-open range of identifiers $[c, c+a)$ to $[h, h+a)$ where c denotes the starting identifier in the container's user namepace and h denotes the starting identifier in the root user namespace.

```
$ cat /proc/self/uid_map
0 1000 50
```

Code snippet 5.2: Example of an identifier mapping

Consider the example in code snippet 5.2. The runtime has mapped the root user in the container's user namespace to the user with identifier 1000 on the host. Similarly, user 1 in the container is mapped to user 1001 on the host. The same logic applies to all users up to 49 and 1049 inside the container and on the host, respectively. If a malicious attacker gains access to a container and manages to break out of its user namespace, protecting the host system depends on the identifier mapping. In the example above, the attacker will still be unprivileged on the host, provided that users $(1000, 1049)$ are unprivileged. If, however, the runtime writes a mapping

$$c = h = 0; a = 1, \tag{5.1}$$

a container escape proves critical. The mapping in (5.1) is the default one for containers created by the Docker engine, necessitating additional configurations to enable rootless containers. The identifier mappings of a container also have an impact on its noninterference boundary with other containers. If the runtime maps the same user identifer on the host to the root user inside the user namespaces of two different containers, an attacker that escapes from one container will be able to enter the user namespace of the other. Upon entering, the attacker will have root privileges. Therefore, for a truly secure environment, two invariants must hold - mapped identifier ranges must not overlap across different containers, and a privileged user on the host must not be mapped to a user inside a container. The kernel imposes several restrictions to user-space for writing to the map files. First, the kernel prohibits more than a single write to such a file. That is, the runtime can write only once to that file. Furthermore, at least one line must be written to the file. An attacker that subsequently attempts to manipulate the mappings would not be able to. Secondly, the writing process must have the `CAP_SETUID` and `CAP_SETGID` capabilities in the root user namespace, or otherwise it can write only a single entry into the file. Deciding which identifier mappings to write is a function of a container engine, not the runtime. The runtime simply reads the identifier mappings from a configuration file like the one defined in code snippet 8.1, buffers them in memory and dumps them to the respective map files from within the container process. This is the very first thing that the container process does to ensure that subsequent operations are properly authorised by the kernel.

Creating a process with the `CLONE_NEWNS` flag puts it in a new mount namespace that inherits a copy of the parent's mount points. This includes the parent's root mount (at `/`), which contains various "submounts" such as the pseudo filesystems `/proc` and `/sys`, the in-memory filesystem `/dev` and all of its resident device nodes, the POSIX message queue filesystem `/dev/mqueue` and the POSIX shared-memory filesystem `/dev/shm`. The container runtime is responsible for setting up all of these filesystems inside the new mount namespace, under the base directory of the container's dedicated root filesystem. Note that upon creation, the container's root filesystem is a simple directory, not a mount point. The runtime must convert it into a mount point so that it can replace the inherited root mount with a new one. Therefore, the runtime bind mounts the aforementioned directory onto itself, using the `mount` system call. Recall from Section 2.2.3 that every mount point is associated with a propagation type. If the parent mount of the directory that holds the root filesystem has a shared propagation type, then bind mounting that directory inside the container will propagate the mount event downstream to the parent mount, causing the bind mount to appear in the set of mount points of the root mount namespace. This is not desirable as it directly interferes with the host environment. For this reason, the container process first remounts the old root filesystem to have a private propagation type and only then performs the bind mount. Note that setting the propagation type to `MS_SLAVE` would also work, if the container wants to receive mount events from the host. This process is shown in code snippet 5.3. The variable `rfs->cro_dst` holds the absolute path to the container's root filesystem. Afterwards, the runtime mounts the device filesystem as a `tmpfs` instance and creates the five device nodes listed in Table 5.1. Even though the container process is privileged in its user namespace and has the `CAP_MKNOD` capability set, trying to create the device nodes with the `mknod()` system call fails with an `EPERM` error. Albeit an important resource inside a containerised environment, device nodes are not namespaced nor owned by a user namespace. Instead, the kernel resorts to authorising the operation with the user credentials in the initial root namespace, which will always fail in the context of a rootless container. For this reason, the runtime falls back to bind mounting the devices from the host. Axiomatically, the noninterference boundary for these devices is nonexistent.

```
/* Error handling deliberately left out */
mount("", "/", "", MS_PRIVATE | MS_REC, NULL);
mount(rfs->cro_dst, rfs->cro_dst, "bind", MS_BIND | MS_REC, NULL);
mount("", rfs->cro_dst, "", MS_PRIVATE, NULL);
```

Code snippet 5.3: Remounting the original root filesystem to disable mount propagation events and bind mounting the container's root filesystem onto itself.

Device	Description
`null`	Data sink. Reads return EOF
`zero`	Data sink. Reads return null bytes
`full`	Data sink/source emulating a full device. Reads return null bytes. Writes return `ENOSPC`
`random`	Random number generator gathering environmental noise from device drivers. Blocks until enough entropy is gathered.
`urandom`	Random number generator gathering environmental noise from device drivers. Has non-blocking semantics

Table 5.1: Table of mounted device nodes within a container

The same semantics apply when trying to mount a block device inside a rootless container. The block device must first be mounted within the root mount namespace, and then bind mounted from within the container's mount namespace. This, however, weakens the noninterference boundary. An attacker that escapes a container may interfere with other containers by manipulating the device's mount point.

The `proc` and `sys` filesystems are mounted conditionally. The former presents information about all processes on the system. If the runtime is instructed to create a container with its own isolated process identifier namespace, then it must mount a dedicated `proc` filesystem that replaces the old one. Otherwise, process information from the host environment will be leaked inside the container. Similarly, the `sysfs` filesystem contains information about the system's network interfaces, routes, firewall rules and so on. If the container inherits this mount from the host, then networking information will be leaked. In addition, containerised processes that rely on the contents of that filesystem will not work properly. Therefore, the runtime replaces the old mount with a new one, provided that the user has requested a new network namespace. The kernel is instructed to disallow the creation of set-user-id programs, executable files and device nodes within these filesystems. Write permissions are disabled for all users that are not mapped to the root user within the container's user namespace. After setting up all the mounts, the runtime pins the root filesystem onto the root directory entry, making all filesystem entries on the host inaccessible from within the container's mount namespace.

```
/* Error handling deliberately left out */
int old_root = -EBADF, new_root = -EBADF;
new_root = openat(-EBADF, rfs->cro_dst, O_DIRECTORY | O_PATH | O_CLOEXEC | O_NOFOLLOW);
old_root = openat(-EBADF, "/", O_DIRECTORY | O_PATH | O_CLOEXEC | O_NOFOLLOW);
fchdir(new_root);
pivot_root(".", ".");
fchdir(old_root);
mount("", ".", "", MS_SLAVE | MS_REC, NULL);
umount2(".", MNT_DETACH);
fchdir(new_root);
```

Code snippet 5.4: Swap the old root filesystem with a new one

This is achieved through the `pivot_root` system call, which moves the old root mount to a location specified by the runtime and mounts the root filesystem at `/`, as shown in code snippet 5.4.

The runtime overlays the old root mount on top of the new one, caching a file descriptor to the former before pivoting. It then switches its current directory by using the file descriptor and unmounts the old root. Note that setting the old root filesystem's propagation type to `MS_PRIVATE`, as shown on line 1 in code snippet 5.3, is a requirement for calling `pivot_root`. Otherwise, unmounting the old root filesystem may propagate downstream to the host, causing it to unmount as well.

Creating a process with the `CLONE_NEWUTS` flag puts it in a new uts namespace that inherits a copy of the parent's system identifiers. The runtime reacts to this by setting the container's hostname to a user-defined string present in the container's configuration file.

The runtime does not enforce a particular network configuration for a container. This is done by the container engine, or in our case, the benchmark tool. Hence, the runtime proceeds to call `execve`. In turn, the kernel installs a fresh virtual address space for the process with the user program loaded. The runtime must ensure that all auxiliary file descriptors are closed before executing the program. Otherwise, these will be preserved in the file descriptor table of the process and leaked to the user program. The most fault-tolerant way to do this is to open every file descriptor with the `O_CLOEXEC` flag. The kernel takes care of closing the file descriptors when `execve` is called. Note that, for debugging purposes, the standard input-output streams are not closed and are therefore inherited.

The runtime process and the container process need to synchronise with each other for two purposes. First, the runtime allows external applications to plug into well-defined points in the lifecycle of a container and execute arbitrary binaries.

Event	Sender	Receiver	Event Description
`EVENT_RT_CREATE`	Container	Runtime	Execute runtime creation hooks
`EVENT_RT_CREATED`	Runtime	Container	Runtime creation hooks completed
`EVENT_CONT_CREATE`	Runtime	Container	Execute container creation hooks
`EVENT_CONT_CREATED`	Container	Runtime	Container creation hooks executed
`EVENT_CONT_START`	Runtime	Container	Start user-defined binary
`EVENT_CONT_STARTED`	Container	Runtime	User-defined binary started

Table 5.2: Table of events exchanged by the runtime and the container

For example, a set of hooks gets executed after the container pivots into its root filesystem but before it executes the user-defined binary. Hence, the container process needs to block and wait for a hook execution event from the runtime. Second, the runtime process must be able to receive error notifications from the container so that it knows when to reap the container and what to report to the user. The kernel provides a variety of inter-process communication mechanisms that can be grouped into two semantical categories - shared-memory and message passing. The runtime uses the latter. In particular, the container and runtime processes interact through a pair of full-duplex UNIX streaming sockets created with the `socketpair()` system call. The runtime and the container keep one end of the socket pair open and close the other end, respectively. Sockets have a natural point of synchronisation - if a process attempts to read any data from the socket, it blocks until data becomes available. Continuing the example above, when the container needs to execute a set of hooks before executing the user-defined binary, it reads from the socket, thereby blocking until the runtime process writes some data. Similarly, if the container process encounters an error, it writes a particular error message into the socket. When the runtime reads from the socket, it will receive the error notification and perform the necessary cleanup operations. The complete summary of messages, as well as the sending and receiving parties are described in Table 5.2. The receiving party in each of the messages always blocks until the sender writes the message to the underlying socket.

5.1.2 Executable

We now turn our focus onto the executable component, also referred to as a *container shim*. The shim sits between a container engine and a container runtime library. The shim is a process dedicated to managing a single container instance. It lives for the duration of the container and takes on the responsibility of redirecting the container's standard input-output streams to log files that can be intercepted by the container engine. This functionality is beyond the scope of this work, albeit trivial to implement. Instead, the container inherits the input-output streams of the shim. In addition, the shim is responsible for tracking and updating the container's state machine, as well as propagating signals sent by the container engine to the container. Both functionalities are implemented in a similar manner - by registering a file descriptor with an event loop and suspending the shim's execution until the file descriptor becomes active. The shim registers two file descriptors - a signal file descriptor that becomes active whenever the shim receives a signal from another process, and the container's pollable process file descriptor, which becomes active whenever the container process terminates. The creation of the latter is done via the `clone()` system call and can be seen in code snippet 5.1. The signal file descriptor is created via a dedicated system call - `signalfd()` - that provides an alternative way of accepting signals besides registering a signal handler incapable of executing functions that are async-signal unsafe. The shim blocks all signals that aren't handled according to their default dispositions and registers the signals it would like to receive events on through a signal mask configured by calls to `sigemptyset()`, `sigaddset()` and `sigprocmask()`. The file descriptor is subsequently registered with an event loop managed by the `poll()` system call. This system call blocks the active thread until a file descriptor is ready. The system call populates a bit mask indicating what type of event occured. The shim is interested in a single event - `POLLIN`, which indicates that the file descriptor is readable. In the context of the container's file descriptor, no data is available. This event simply indicates that the process has exited. In the context of the signal file descriptor, the shim can read a structure from it that holds the requested signal and afterwards propagate that signal to the container process.

The container shim provides a command-line interface to its users. A user is required to specify the path to the container's configuration file, an eample of which is shown in code snippet 8.1. In addition, the user must give the container a unique name. Optionally, the user may specify a timeout. The timeout is used by the shim to wait for the container to exit. If the timeout expires, the container is forcibly killed by the shim. The shim's command line interface is shown in code snippet 5.5.

```
$ conty-runner --help
Usage: conty-runner [OPTION...] NAME
conty-runner -- A program that runs containers

  -b, --bundle=PATH Path to container bundle
  -t, --timeout[=NUMBER] Time to wait for container to exit before killing
  -?, --help Give this help list
      --usage Give a short usage message
  -V, --version Print program version

Mandatory or optional arguments to long options are also mandatory or optional
for any corresponding short options.

Report bugs to htw-berlin.de.
```

Code snippet 5.5: Container shim command-line interface

5.2 Benchmark

The benchmark focuses on measuring network and filesystem latencies and throughput within and outside a container, respectively. The goal is to highlight the differences in response time and efficiency (See Chapter 6). Here, the tools used for measurement sampling and plot generation are discussed. Additionally, implementation details surrounding the workloads are mentioned.

5.2.1 Network workload

The network workload introduced in Section 4.2.2 was implemented with the help of the `iperf3` program - a tool for performing network throughput and latency measurements for TCP and UDP. This tool consists of two launch modes - server mode and client mode. The server can be launched by specifying the `-s` option and the network address and port it should listen on, as shown in code snippet 5.7. By default, the server listens on port 5201.

```
$ iperf3 -s 192.168.8.2
```

Code snippet 5.6: Launching an iperf3 TCP server

The client is launched with the `-c` option and the network address and port of the server it should connect to. In addition, the user can configure the number of parallel connections, the size of the buffers to be transmitted when interacting with the server, and the sampling duration. Furthermore, the user can instruct the tool to output its measurement samples in JSON and to a file.

```
$ iperf3 --logfile out.iperf3 -J --parallel 5 --length 128k --time 60 -c
    192.168.8.2
```

Code snippet 5.7: Launching an iperf3 TCP client that connects to an iperf3 server listening on 192.168.8.2 and writes performance samples gathered for 60 seconds from 5 parallel connections into a file

In the above example, the client writes a total of $60 \times N$ samples, i.e one sample per second, into the file where N denotes the number of parallel connections. Each sample contains, amongst other things, the round-trip time measured in microseconds, the number of bits transfered per second, as well as the number of retransmits encountered during the time interval. To generate the plots, the file is preprocessed by a bash script that groups the aforementioned metrics by connection and generates a folder with files storing the preprocessed data. Afterwards, the results are fed into the `gnuplot` command-line program in order to generate the two-dimensinal plots shown in Section 6.3.

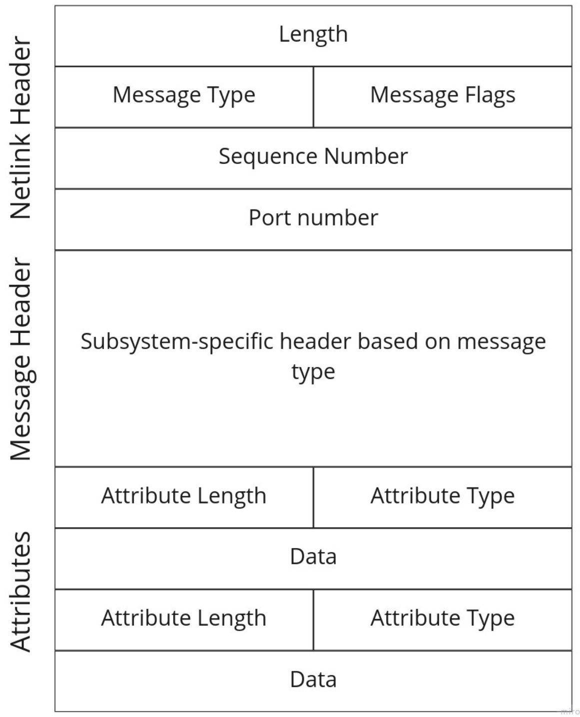

Figure 5.1: Netlink message format

The TCP client and server programs must run in two different containers in order to measure the isolation overhead of the network namespace abstraction. However, they cannot talk to each other unless properly configured. For this reason, a hook program was developed that sets up a virtual ethernet cable for both the client and server. The hook program is referenced within the client and server's container configuration files, under the `on_runtime_create` hooks mentioned in Table 4.1. The container runtime picks up the program and executes it when creating the containers. In addition, the hook program sets up a bridge device (if it does not already exist) and attaches the virtual ethernet interface of the container to it. The interfaces are configured by interacting with the kernel through a datagram socket whose format is dictated by the netlink protocol.

The netlink protocol is based on messages that consist of a header and a payload. Usually, the payload is further deconstructed into a subsystem-specific header that identifies a particular resource and a set of attributes that represent configurable properties of the resource. The hook program specficially targets the routing subsystem, which manages resources like virtual network devices, their addresses and routing tables. The general message format is shown in Figure 5.1.

For the sake of simplicity, the hook program was implemented in the Go programming language and utilised a library that hides the underlying details of the netlink protocol.

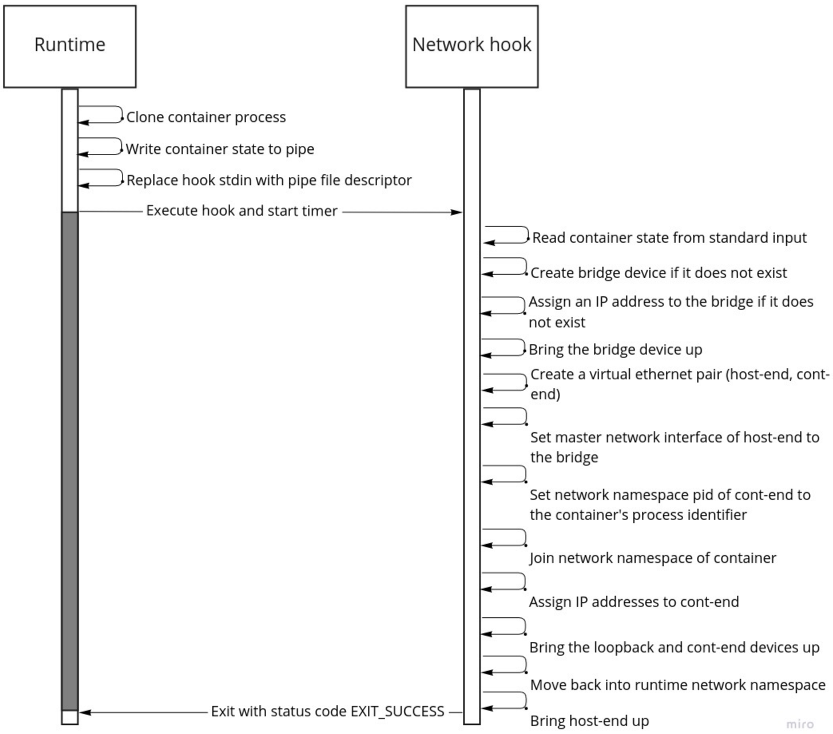

Figure 5.2: Network hook sequence diagram

The hook's sequence of operations is shown in Figure 5.2. Notice that the hook needs to join the network namespace of the container to setup the network addresses of the virtual ethernet cable. This is done by obtaining a file descriptor to the respective namespace and joining it by calling the `setns()` system call. After the configuration is complete, the hook returns back to its original namespace by employing the same mechanism. The source code of this operation is shown in code snippet 8.2.

The runtime ensures that the hook program cannot temporally interfere with its execution environment by starting a timer before executing it. If the timer expires and the hook has not completed its execution, the runtime forcibly kills it. Temporal interference may occur because the runtime is required to reap the hook's process identifier. This is done via the `waitpid()` system call, which suspends the runtime's thread of execution until the hook exits. If the hook does not exit, it will cause a denial of service by indefinitely blocking the runtime.

The network hook program requires capabilities that the container runtime does not have, namely the `CAP_SYS_ADMIN` and `CAP_NET_ADMIN` capabilities. The latter is required to configure the network for the container. The former is needed to join the network namespace of the container. When the hook program is started, it inherits the capability sets of its parent, which are insufficient. For this reason, the program is transformed into a *setuid* binary, as shown in code snippet 5.8. When executed, the hook inherits the capability sets of the root user without elevating the privileges of the runtime.

```
go build -o ./bin/net-hook ./cmd/network/main.go
sudo chmod u+x ./bin/net-hook
sudo chown root ./bin/net-hook
sudo chmod u+s ./bin/net-hook
```

Code snippet 5.8: Building the network hook, changing its ownership and transforming it into a setuid binary by setting the respective permission bit

5.2.2 Filesystem workload

The filesystem workload introduced in Section 4.2.2 was implemented with the help of the `fio` program - a tool for simulating input-output workloads and sampling performance metrics. `fio` can perform input-output operations directly on block devices or on files. In the former case, data is written directly to the device's addressable memory regions from user space. In the latter case, this is handled by the kernel. The first option should be handled with care, because `fio` will completely overwrite the device's contents. Users can configure the size of the test file with the `--size` option. Furthermore, the workloads can be configured to bypass the page cache with the `--direct` option. This makes sure that the kernel's underlying caching mechanisms do not cause any perturbations when assessing disk performance. All input-output operations are directed to the underlying device, regardless if the file's contents are already present in-memory. The size of the input-output buffers can be configured with the `--block-size` option. This option is very important, because it directly impacts the latency and throughput results. Larger block sizes typically result in higher throughputs, and higher latencies, whereas smaller block sizes lead to lower throughputs and lower latencies. Users can also configure the I/O pattern to be employed with the `--mode` option. When employing mixed input-output operations, the number of reads and writes are equally distributed. The available patterns are listed in Table 5.3. Fio populates the file with random data, which may cause perturbations during the data generation process. To mitigate this, the program first generates the buffers and reuses them. This can be bypassed by specifying the `--refill_buffers` option, which refills the buffers

I/O pattern	Description
`read`	Sequential reads
`write`	Sequential writes
`randread`	Random reads
`randwrite`	Random writes
`readwrite`	Mixed sequential reads and writes.
`randrw`	Mixed random reads and writes.

Table 5.3: Table of I/O patterns made available by fio

before every I/O submission. Fio provides two modes of submitting parallel I/O operations. The most straightforward way is to increase the number of parallel processes via the `--numjobs` option. Alternatively, when using an asynchronous I/O engine, the level of parallelism can be increased by specifying the `--iodepth` parameter, which regulates the number of parallel I/O operations that can be submitted to the kernel via a single system call. The I/O engine to use is configured via the `--engine` parameter. It can be set to `sync` and `libaio`. The `fio` command-line tool generates a JSON file containing the performance samples gathered during the workload's execution. To generate the plots, the command-line tool's execution is wrapped in a python script `bench-fio` [lou] that processes the generated JSON file and stores the processed outputs in a format that can be interpreted by another publicly available script called `fio-plot` [lou] that generates a set of plots.

5.2.3 Usage & Reproducibility

Manually reproducing the results of this work is a tedious process. Every workload that is to be wrapped in a container requires a dedicated root filesystem with all dependencies preinstalled, as well as a container configuration file that correctly points to the path of the container's root filesystem on the host, the path to the workload within the container, and to the paths of all hooks, which are also context-dependant. For this reason, a semi-automated facility was developed to aid in reproducing the work. In this section, the steps necessary to reproduce the results are discussed.

First, the repository containing the source code needs to be cloned and the container runtime needs to be installed:

```
$ git clone https://github.com/ubiquitousbyte/lxc-research.git
$ cd lxc-research/src/conty
$ mkdir build
$ cd build
$ cmake ../
$ make --build . --target install
```

Code snippet 5.9: Building and installing the container runtime

The benchmark tool is located in `src/conty/src/bench` relative to the repository's root directory. All succeeding operations must be executed within that directory.

Network workload

To execute the network workload within different network namespaces, both the client and the server need a root filesystem with `iperf3` preinstalled. The benchmark tool leverages another container technology - Docker - to generate such a root filesystem. To create the root filesystem, a custom Dockerfile was created, from wich an image was built. The generated container image is then exported into a tar archive which is extracted into `./rootfs/net`. Our custom container runtime can use the extacted contents as a base directory for the container's root filesystem. In addition, the network hook discussed previously is built and the permissions that it requires are setup accordingly. Furthermore, the template configuration files `net-container-server-template.json` and `net-container-client-template.json` are copied and the paths to the respective resources are automatically adjusted. All of these operations are abstracted away through a Makefile, so running

```
$ make install-net
```

should suffice. Befure running the network workload, make sure to disable docker's mandatory firewall rules via:

```
$ echo 0 > /proc/sys/net/bridge/bridge-nf-call-iptables
```

The docker engine prevents network communication across namespace boundaries by instructing the kernel to drop packets flowing through bridge devices that are not protected by a firewall. Now, the client and the server can be wrapped within containers:

```
$ conty-runner --timeout=120 --bundle=./net-container-server.json srv &
$ conty-runner --bundle=./net-container-client.json cln
```

The server is started as a background container that will be terminated after 120 seconds. The client is started afterwards and runs in the foreground. If the `iperf3` client needs to be reconfigured in any way, the `process.args` property of the JSON configurations generated by `make install-net` can be changed. By default, `iperf3` is configured to write its outputs in a file `/test.iperf3` within the container. This translated on the host to `./rootfs/net/test.iperf3`. The file can be copied and the plots can be generated:

```
$ cd plotters
$ cp ../rootfs/net/test.iperf3 perf-output.json
$ ./iperf-prepare.sh perf-output.json .
$ cd results
$ gnuplot ../../plot_retransmits.plt
$ gnuplot ../../plot_rtt.plt
$ gnuplot ../../plot_throughput.plt
$ ls
RTT.pdf retransmits.pdf throughput.pdf
```

Running the equivalent operations on the host requires that the `iperf3` tool is installed. The listing below shows how to run the workload on the host.

```
# On terminal 1
$ iperf3 -s 192.168.8.2
# On terminal 2
$ iperf3 --logfile out.iperf3 -J --parallel 5 --length 128k --time 60 -c
    192.168.8.2
$ cd plotters
$ cp ../out.iperf3 perf-output.json
$ ./iperf-prepare.sh perf-output.json .
$ cd results
$ gnuplot ../../plot_retransmits.plt
$ gnuplot ../../plot_rtt.plt
$ gnuplot ../../plot_throughput.plt
$ ls
RTT.pdf retransmits.pdf throughput.pdf
```

Filesystem workload

The process of running the filesystem workload is described here. The becnhmark tool takes care of setting up a root filesystem with `fio` preinstalled and replicating the `io-container-template.json` configuration file with all necessary paths set. Running the listing below will take care of the setup.

```
$ make install-io
```

The `process.args` field in the `io-container.json` file needs to be edited multiple times with independently varied settings for the `--numjobs` and `--iodepth` parameters. The values used in the experiment were equal to 2^i where $i = 0 \ldots 6$. The listing below shows an example where `--numjobs` and `--iodepth` are set to 2^0.

```
# Example parameters in io-container.json for the read operation
$ bench-fio --type=file --target=/SanDisk_microSD_UHS-I-Container --size=2G
    --output=/results --block-size=4k --numjobs=1 --iodepth=1 --engine=libaio
    --mode=read --direct=1
# Example parameters in io-container.json for the write operation
$ bench-fio --type=file --target=/SanDisk_microSD_UHS-I-Container --size=2G
    --output=/results --block-size=4k --numjobs=1 --iodepth=1 --engine=libaio
    --mode=write --direct=1 --destructive
```

After every edit, the workload can be run in a sandbox:

```
$ conty-runner --bundle=./io-container.json ioc
```

Note that the `--output` and `--target` properties must remain the same. The `bench-fio` script will populate the `/SanDisk_microSD_UHS-I-Container` directory within the container's root filesystem with the results of every execution.

The same set of operations need to be executed on the host in order to compare container and host latencies and throughput. To do this, a virtual environment is created and all dependencies are installed, as shown below:

```
$ python3 -m venv .env
$ source .env/bin/activate
$ pip install fio-plot
```

Again, the `bench-fio` script needs to be executed multiple times with varying parameters. This time, the results need to be stored in a separate directory on the host:

```
$ touch SanDisk_microSD_UHS-I-Host
$ mkdir fio-host-results
$ bench-fio --type=file --target=./SanDisk_microSD_UHS-I-Host --size=2G --output=./fio-host-results --block-size=4k --numjobs=1 --iodepth=1 --engine=libaio --mode=read --direct=1
$ bench-fio --type=file --target=./SanDisk_microSD_UHS-I-Host --size=2G --output=./fio-host-results --block-size=4k --numjobs=1 --iodepth=1 --engine=libaio --mode=write --direct=1 --destructive
# .. Run the bench-fio script with --numjobs = 1, 2, 4, 8, 16, 32, 64 for read and write while --iodepth=1
# .. Run the bench-fio script with --iodepth = 1, 2, 4, 8, 16, 32, 64 for read and write while --numjobs=1
```

After the workloads are executed, the plots can be generated by doing the following:

```
# Make sure that the virtual environment is activated
$ cd plotters
$ ./fio-plot.sh ../rootfs/io/results/SanDisk_microSD_UHS-I-Container/4k ../fio-host-results/SanDisk_microSD_UHS-I-Host/4k
ls
# Set of PNG files will be generated in the current directory
```

Chapter 6

Experiment

The experiment was conducted on the Raspberry Pi 4 Model B. The Pi's computational capabilities are powered by a Broadcom BCM2711 chip [Ltd22] consisting of four Cortex-A72 processing cores, each having a 48 kB and 32 kB L1 instruction and data cache, respectively. The local caches are connected to a central 1 MB L2 cache, which is also used by an integrated VideoCore 6 GPU. The SoC interfaces with an 8 GB LPDDR4-SDRAM chip through a 32-bit wide bus. Hence, two fetch operations are required to load a 64-bit value from memory into one of the cores' registers. A 64 GB SanDisk UHS-1 microSD card is used for persistent storage. The card is connected to the SoC through an SDIO interface. For networking, the board comes with an ethernet port and a wireless networking module that is connected through an SDIO interface.

A Fedora 36 server image [Fed22] with the 5.17.5 Linux kernel version was flashed on the microSD card and installed on the Raspberry Pi. The disk partitions were resized to fully incorporate the available disk space, ending up with 56 GB of disk space allocated to the logical volume mounted at the root point and 7 GB of swap space. The filesystem mounted at the root point is `xfs`. The device was connected to a hidden local WiFi network through the `nmcli` tool. The ethernet port remained unused.

The device was put in an aluminium case that acts as a heat-sink and dissipates the heat out of the processor. In other words, the case acts as a passive cooling component that protects against potential performance perturbations caused by thermal trottling.

Each of the following sections introduces a question equivalent to a performance problem statement and a hypothesis that tries to answer it. The hypothesis is proven or disproven based on observational data gathered by running the workloads. The data is then analysed. The operational flow is similar for all hypotheses. Observational data is gathered by executing the same workload within and outside the container abstraction. The differences in the data points act as a proxy metric for the isolation overhead of a container.

6.1 Problem A

6.1.1 Problem statement

In order to properly isolate the container's root filesystem from the host, the runtime creates a dedicated mount point that binds the directory on the host to a mount point in the container. The mount point has two important properties. First, it is invisible from the host. Second, it atomically replaces the root mount within the mount namespace of the container. Given this configuration, we can derive the following question:

Does reading/writing data from the underlying device through a bind mounted root filesystem in an isolated mount namespace cause an increase in latency or a decrease in throughput?

6.1.2 Hypothesis

The two potential sources of isolation overhead are the indirection introduced by the mount namespace and the bind mount.

Examining the source code of the `read()` and `write()` system calls shows that the namespace abstraction is used once - to check if the calling process has the necessary permissions to operate on the file. This is done by checking if there is a valid user identifier mapping between the user namespace of the calling process and the owning user namespace of the file. Both are accessed in constant time - by following pointers relative to the currently executing process and the inode that is translated from the file descriptor, respectively. The inode user identifier is translated into a user identifier within the user namespace of the mount namespace where the inode was found. This is also a constant operation, because identifiers in different user namespaces are mapped using simple addition and subtraction [Doc22]. The translated user identifier is compared to the user identifier of the current process and if they match, the read operation is delegated to the underlying fileystem implementation - `xfs`. If the identifiers do not match, the access control lists of the inode are consulted before delegating. The ACL subsystem asseses the permissions of the process through a multitude of bitwise operations, which are also constant.

Setting up the bind mount should not have an effect on these system calls. The overhead of that operation manifests itself during the container creation phase - when a duplicate view of the directory holding the container's root filesystem is created under a new node in the file system hierarchy.

The argumentation above leads to the following hypothesis:

Reading and writing data from a device through a bind mounted root filesystem resident in an isolated mount namespace exhibits the same latency and throughput characteristics as performing the equivalent operations in the root mount namespace without a bind mount.

6.1.3 Test

In order to test the hypothesis, the filesystem workload ran for both the read and write operations within and outside a container. The workload's total runtime was 60 seconds. No concurrent operations were executed - a single process issued a single operation per unit of time. During the workload's runtime, the number of successfully executed input-output operations and the total latency per second were accumulated. The results for the read and write system calls are shown in Figure 6.1, Figure 6.2, Figure 6.3 and Figure 6.4.

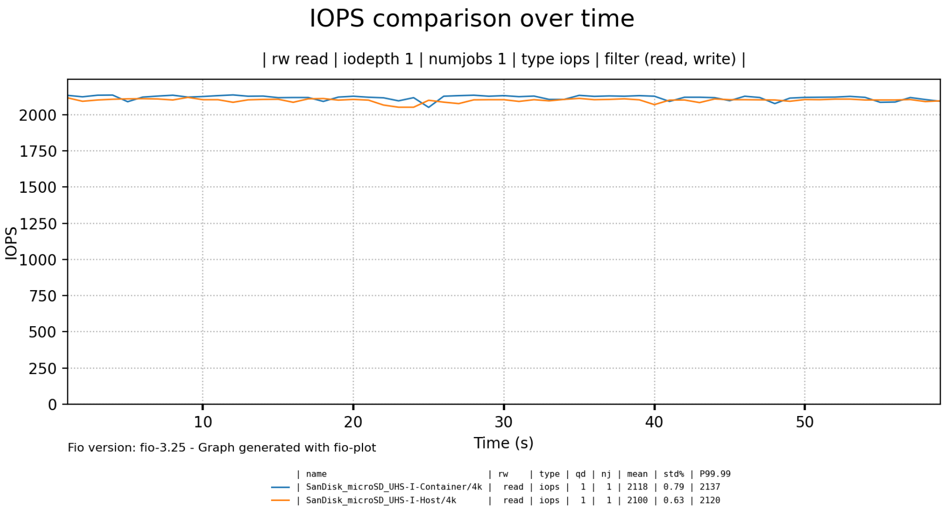

Figure 6.1: Read IOPS on the host and inside a container, compared over time

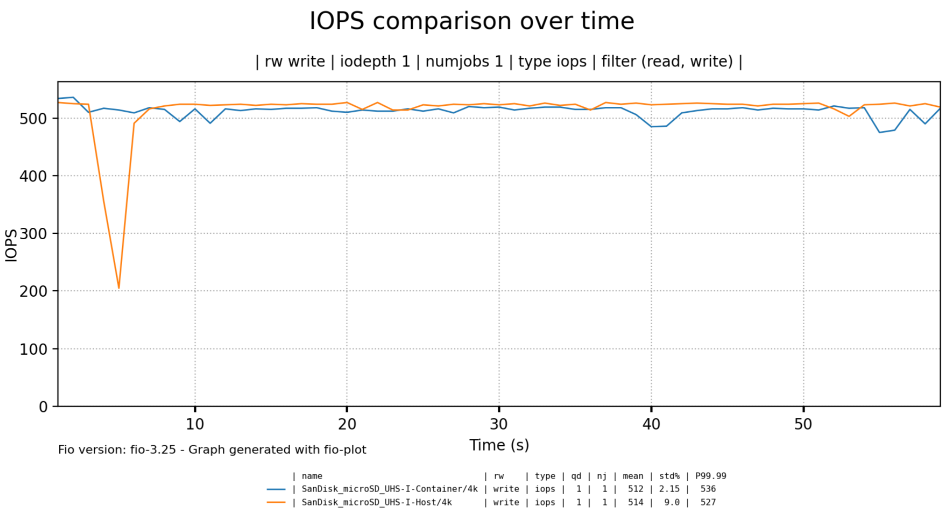

Figure 6.2: Write IOPS on the host and inside a container, compared over time

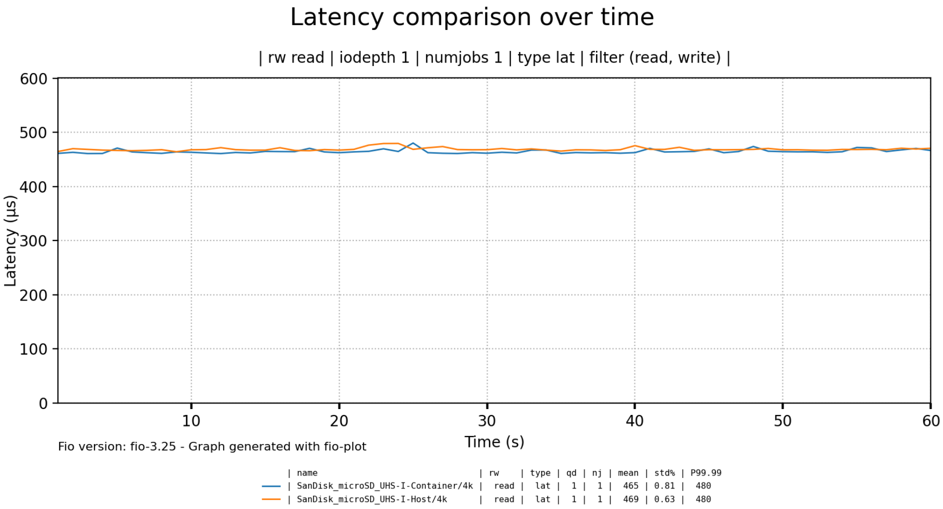

Figure 6.3: Read latency on the host and inside a container, compared over time

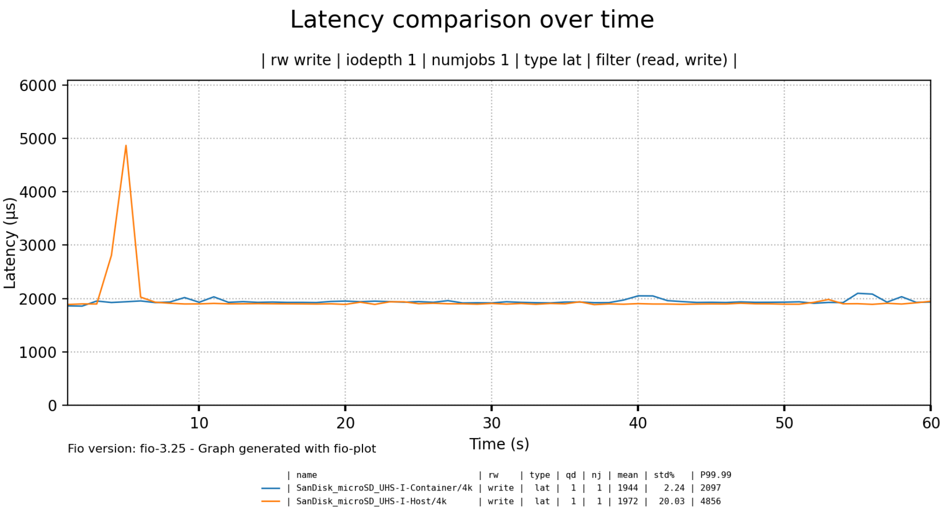

Figure 6.4: Write latency on the host and inside a container, compared over time

6.1.4 Analysis

Figure 6.1 plots the number of read operations per second within and outside a container. The orange line denotes the values accumulated on the host, whilst the blue line highlights the container's accumulations. The results show that reading data through a bind mounted root filesystem within a separate mount namespace has no impact on read throughput. The container handles, on average, 8 more read operations than the host throughout the workload's lifetime. This, however, is arbitrary and varies across workload runs. As can be seen by the graph, the throughput achieved is nearly identical during the entire workload's lifespan.

Figure 6.3 plots the latency (in microseconds) achieved within and outside a container. The graph tells a similar story. Latencies are stable and do not deviate by more than 0.81% from an average of 0.45 milliseconds.

Write operations exhibit higher latencies and lower throughputs. So much so that the difference in read and write performance is a factor of three. In addition, write samples deviate a lot more from their mean value compared to read samples, as can be seen in Figure 6.2 and Figure 6.4. The host has a significant outlier in the beginning but then achieves better throughput and lower latency throughout the end. The container destabilises at the end of the workload. In the end, however, the mean average IOPS within the container are 512, whereas on the host they are 514 - a negligible difference. The same can be said for the latency - 1.94 milliseconds compared to 1.97 milliseconds for the container and the host, respectively. Hence, the hypothesis remains valid.

6.2 Problem B

6.2.1 Problem statement

Efficiently sharing resources between different processes is one of the primary task of a kernel. As the number of processes utilising the same resource increases, a point of saturation may be reached. This happens when the resource reaches 100% utilisation and queuing begins to be frequent and significant. A natural question follows:

Does scaling the number of reads/writes from/to the underlying device issued from a bind mounted root filesystem in an isolated mount namespace saturate the resource faster than performing the equivalent scaling without the noninterference boundary?

This question directly follows the performance axiom in Section 2.1.1. Does the overhead introduced by the container runtime to isolate the filesystem negatively affect the ability of processes in the container to share a given resource?

6.2.2 Hypothesis

From the previous problem, we know that the difference in latency and throughput for a single process outside and within the container's noninterference boundary is negligible and has no statistical significance. Assuming that scaling the number of reads/writes is done by increasing the number of processes that concurrently issue input-output operations, then the point of saturation should be exactly the same as on the host.

6.2.3 Test

In order to test the hypothesis, the filesystem workload ran for both the read and write operations within and outside a container. The workload's total runtime was 60 seconds. The number of processes was scaled n times by 2^i for $i = 0 \dots n$ where $n = 6$. During the workload's runtime, the number of successfully executed input-output operations and the total latency per second were accumulated per process group. The test results are shown in Figure 6.5, Figure 6.6, Figure 6.7 and Figure 6.8.

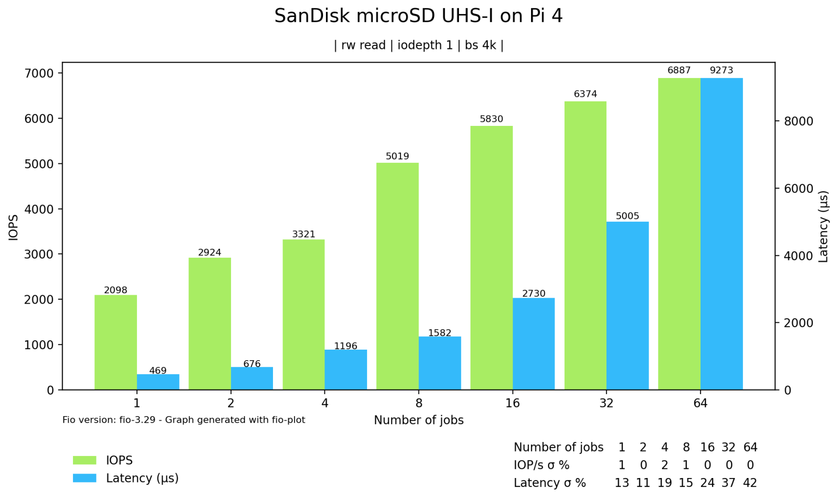

Figure 6.5: Throughput and latency measurements on the host as a function of the number of concurrent processes issuing read operations.

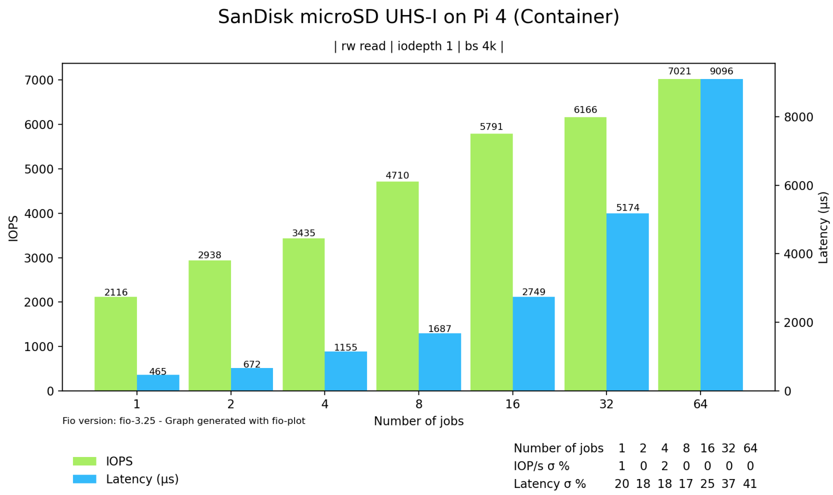

Figure 6.6: Throughput and latency measurements in a container as a function of the number of concurrent processes issuing read operations.

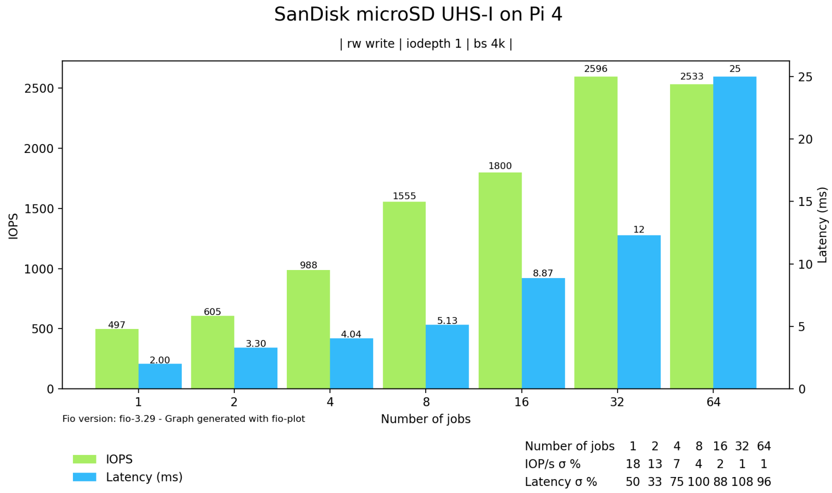

Figure 6.7: Throughput and latency measurements on the host as a function of the number of concurrent processes issuing write operations.

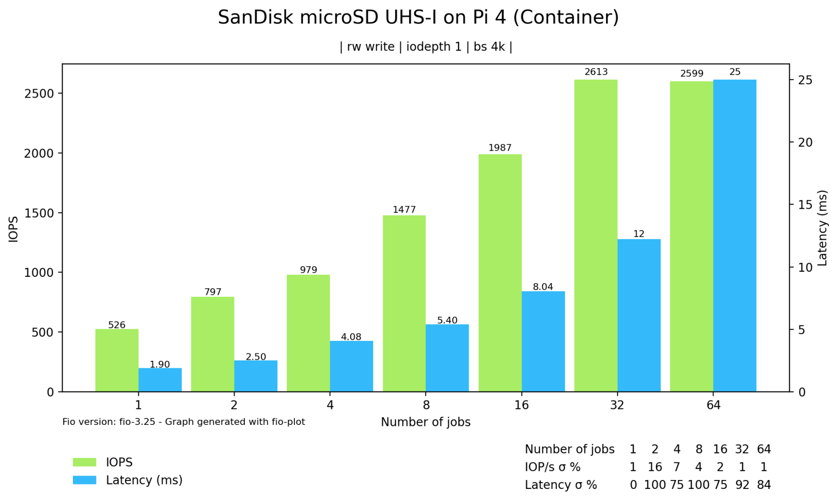

Figure 6.8: Throughput and latency measurements in a container as a function of the number of concurrent processes issuing write operations.

6.2.4 Analysis

Every figure shows the number of concurrent processes issuing the respective operation on the x-axis. The y-axis on the left side represents the number of IOPS conducted. The green bars show the average number of IOPS achieved during the workload's execution. The y-axis on the right side represents the latency (in microseconds) The blue bars show the average latency during the workload's execution.

Figure 6.5 and Figure 6.6 show the number of IOPS and the latency for the read operation on the host and in a container, respectively. In both cases, the same tendency is observed. As the number of processes is increased, the number of successfully executed input-output operations grows. The same holds for latency. The best IOPS-to-latency ratio outside the container is observed when the number of concurrently executing processes equals 8. The point of saturation is reached at approximately 32 concurrent processes where the ratio is worser than running a single process. Within the container, the best IOPS-to-latency ratio is observed when the number of concurrent processes is 16. Both the host and the container have the same point of saturation - at 32 processes. Hence, for reads, the hypothesis holds.

The same graphs are shown for the write operation in Figure 6.7 and Figure 6.8. Latencies for this operation are measured in milliseconds. The container outperforms the host in terms of IOPS as the number of processes increases. The container scales, on average, by 345.5 IOPS, whereas the host scales by 339.3. Conversely, the host latency increases on average by 3.83 milliseconds, whereas in the container the increase is by 3.85. Due to the insignificant differences, the saturation point for writes is the same for the container and the host, i.e the hypothesis holds.

6.3 Problem C

6.3.1 Problem Statement

The container runtime enables communication across network namespace boundaries by setting up a bridge device and attaching a virtual ethernet cable for every container to the bridge. The bridge forwards packets between the virtual ethernet cables of the containers. If the containers were to reside in the same network namespace, then their interaction would be quite simple - by binding all sockets to the loopback interface. This observation leads to the following question:

Does reading/writing network packets through a pair of virtual ethernet devices interconnected through a bridge result in higher latencies and lower throughputs compared to the equivalent operations performed in the same network namespace through the loopback interface?

6.3.2 Hypothesis

The infrastructure necessary to provide communication between different network namespaces increases the number of hops required to transmit a packet from its source to its destination. A packet needs to traverse three devices to reach its destination - the virtual ethernet cable of the sender, the bridge, and the virtual ethernet device of the receiver. Every hop naturally introduces a minor latency, which implies a degradation in throughput. It is important to note that the bridge is an integral part of this process. Its inability to relay packets in time may be interpreted by a container's network stack as an indication of congestion. In such a case, the container will reduce its transmission rate, thereby causing decreases in throughput. Congestion may also lead to packet loss, which is handled by asking for retransmissions of packets that have not been acknowledged by the peer. Retransmissions have a negative impact on throughput, because they waste bandwidth and receivers still need to wait for them. Given all of these considerations, we hypothesise the following:

Inter-process communication through a single interface within the same network namespace exhibits higher throughputs and lower latencies compared to inter-container communication through a bridge device.

6.3.3 Test

The TCP server and TCP client were deployed in two separate containers and ran for 60 seconds. The client consists of four connections that communicate with the server in parallel. The block size was set to 128 kB. The same server and client were deployed on the host, communicating with each other through the loopback interface. During their runtime, the number of retransmissions, the round-trip time and the number of megabytes per second were sampled. Each of these metrics were plotted as a function of the workload's execution time. The number of retransmissions exhibited on the host and from the client container are shown in Figure 6.9 and Figure 6.10, respectively. The round-trip times on the host and from the client container are shown in Figure 6.11 and Figure 6.12, respectively. Figure 6.13 shows the achieved throughput on the host, while Figure 6.14 shows the same metric from the perspective of the client within the container.

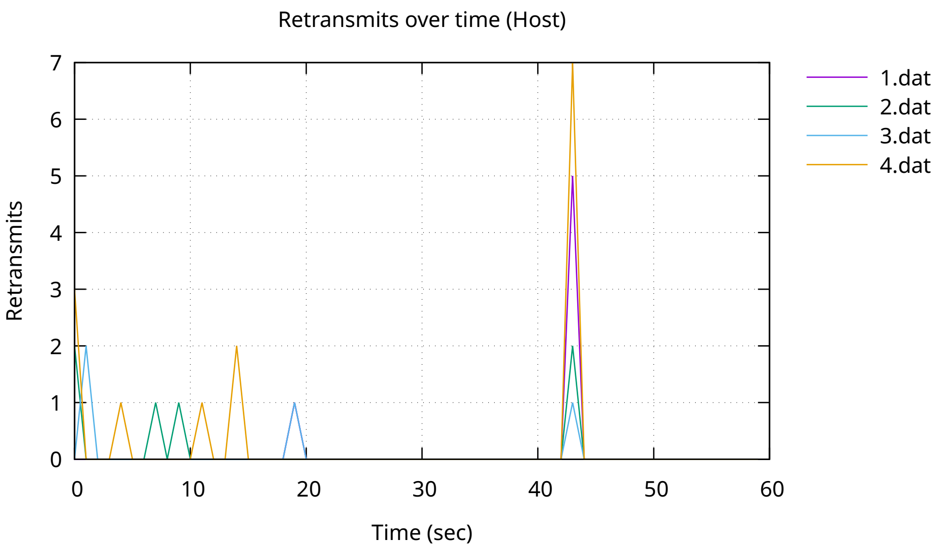

Figure 6.9: Number of packet retransmissions plotted over time on the host

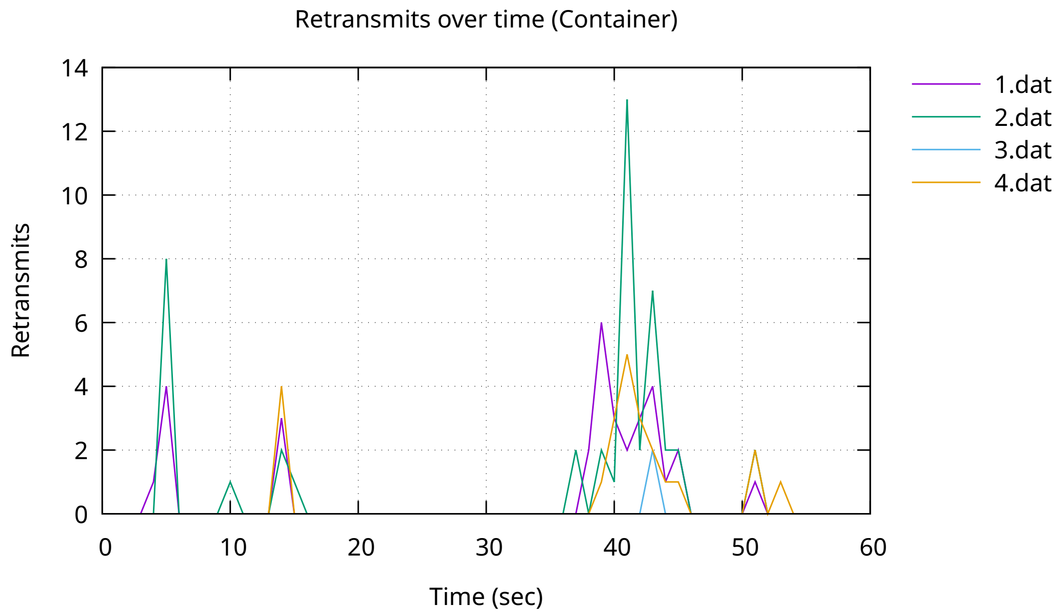

Figure 6.10: Number of packet retransmissions plotted over time within a container

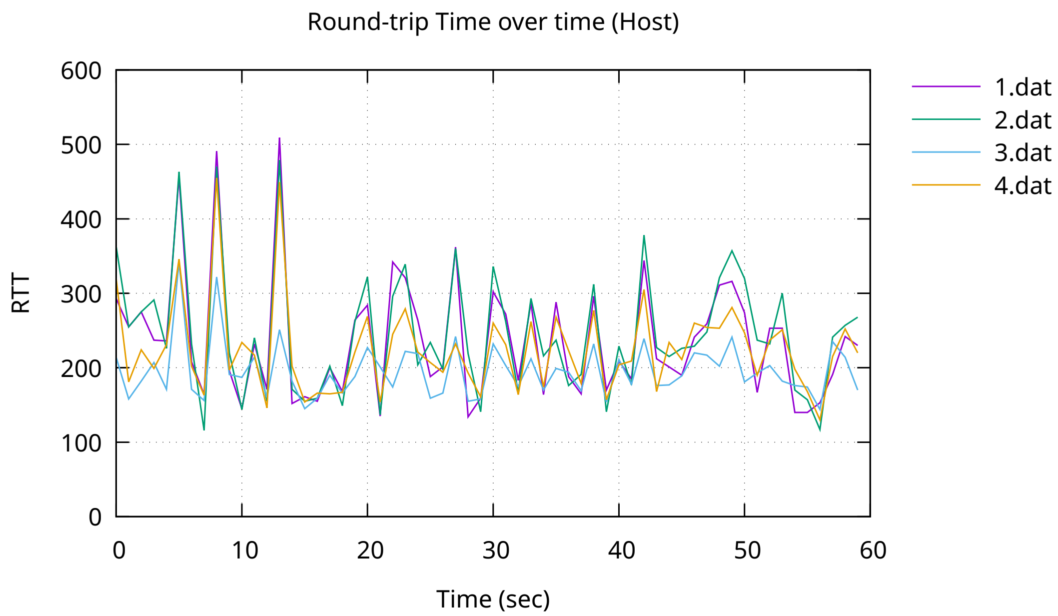

Figure 6.11: Network latency (round-trip time) measured over time on the host

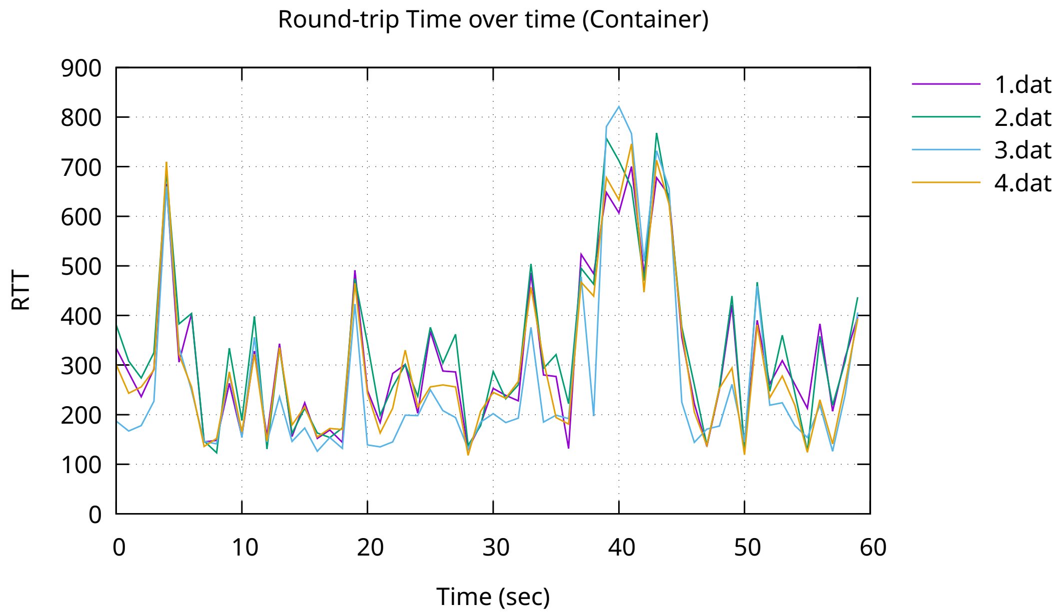

Figure 6.12: Network latency (round-trip time) measured over time within a container

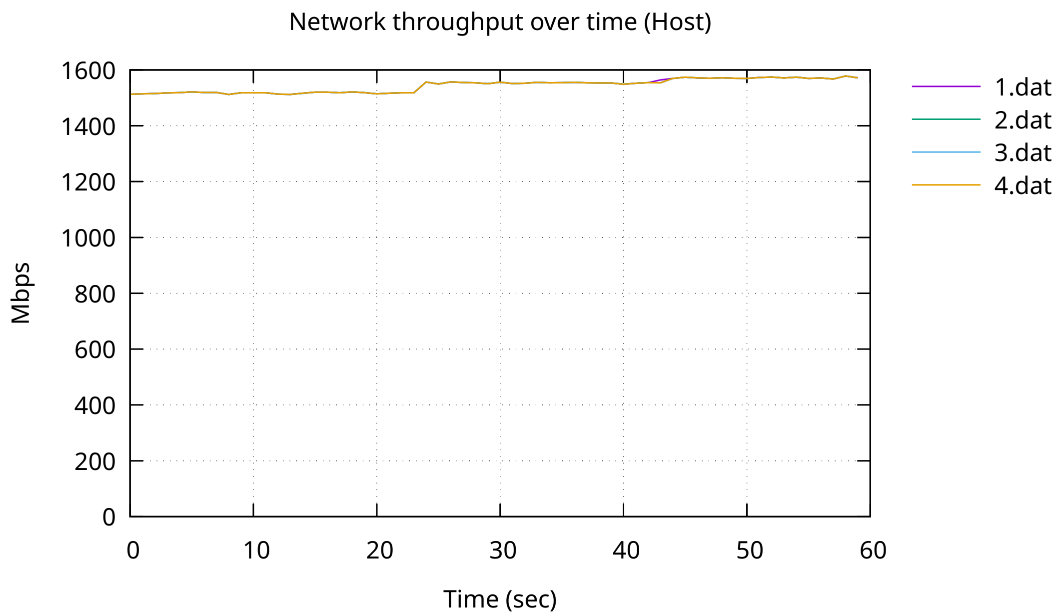

Figure 6.13: Network throughput measured as megabytes per second over time on the host

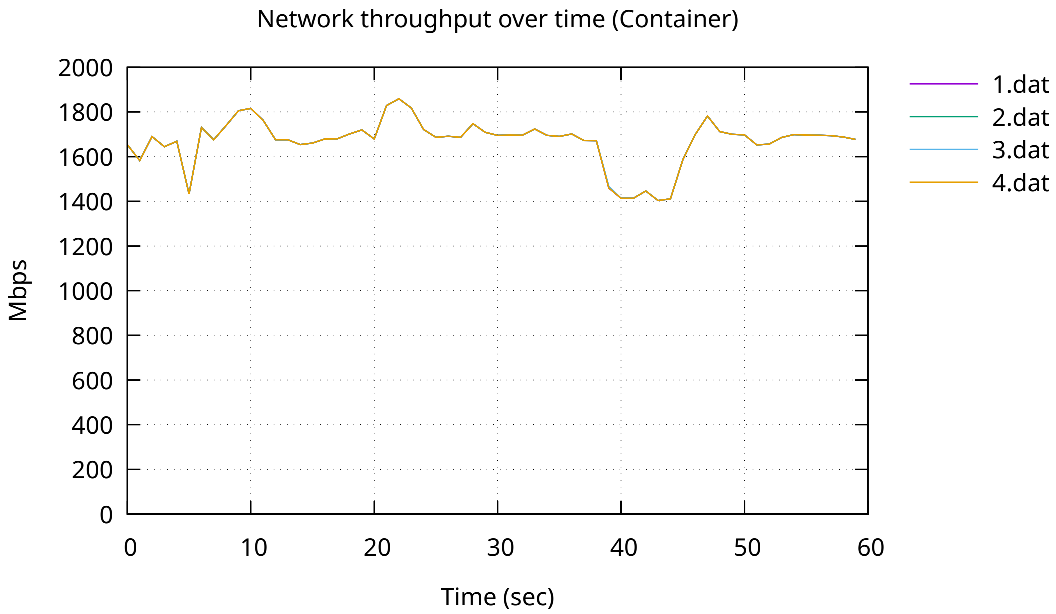

Figure 6.14: Network throughput measured as megabytes per second over time within a container

6.3.4 Analysis

The total number of retransmissions on the host (29) is 2.5 times smaller than the number of retransmissions within the container (76). This has a direct impact on the round-trip time measurements. In Figure 6.11, we can see that the maximum round-trip time on the host is approximately 500 microseconds, but usually fluctuates between 150 and 350 microseconds for all connections. The maximum round-trip time in the container, however, is around 800 microseconds. In addition, it fluctuates in larger strides compared to the host. By examining the time axis, we can see that the largest round-trip times are observed within the same time interval as the highest number of retransmissions for the container. In other words, the workload execution in the container suffered from packet loss, which led to an increased number of retransmissions and higher latencies. Figure 6.13 shows that the achieved throughput on the host was very stable. Starting at around 1.45 GB/s, it slowly grew up to 1.6 GB/s. The throughput within the container environment (Figure 6.14) did not increase linearly. High variances were observed with the smallest achieved throughput sitting at around 1.4 GB/s. The drop in throughput happens within the same time interval as the highest number of retransmissions. Hence, it is also explained by packet loss. Interestingly, the network infrastructure of the container managed to sustain higher throughputs compared to the host in periods where retransmissions did not happen. However, increases in retransmissions on the host did not negatively affect throughput at all, potentially indicating that network communication over a single network interface is more robust than multiple network stacks. From these observations, we can say that the hypothesis holds partially. The container's network infrastructure is less robust, leading to higher retransmission rates and higher latencies. However, in periods where the network does not suffer from packet loss, the container achieves better throughput than the host.

Chapter 7

Conclusion

The results presented in the previous chapter show that the isolation overhead of process containerisation through resource namespacing is negligible. Disk latencies and throughput measurements in containers are on par with the same measurements on the host. The networking infrastructure required to enable inter-container communication may lead to packet loss, higher retransmission counts and therefore increased latencies and decreased throughput. Nevertheless, the average latency and throughput over the lifespan of a workload are almost equivalent to running the application on the host. Furthermore, it is shown that hardware virtualisation mechanisms impact the resulting latencies and throughput far more severely [SLC20] [Zha18]. From a business perspective, the near-native performance of containers makes them highly favourable for consolidating multiple applications on a single server. The primary challenges with containers, however, lie not in performance, but rather security.

7.1 Discussion

Resource namespaces encapsulate a variety of kernel resources, but not all of them. Namespace-unaware kernel subsystems resort to authorising operations with the user identifiers in the root user namespace, i.e those that were mapped by the container runtime into the container. Containers that share these identifiers implicitly share various in-kernel data structures that may lead to interference. For example, the kernel used to (up to v5.13.19) impose per-user resource consumption restrictions such as the total number of file descriptors that can be kept open by that user, the number of pending signals that can be queued among all processes of that user, and the number of processes that the user can spawn. If one container exhausts these limits, other containers poentially owned by a different tenant are directly affected, i.e interfered with. This particular problem has been fixed in the v5.14 release by binding resource limit counters to user namespaces [Gla21]. Another very important example are filesystem implementations. Filesystems associate every file with a unique user identifier that is its owner. Root filesystems for containers residing on the host are typically owned by a single user and group (allocated by the container engine) and are inaccessible by other users to prevent tampering. Hence, to meaningfully mount such a filesystem in multiple rootless containers, each container's identifier map must point to that user on the host, i.e all containers must share the same user identifier. This problem has been tackled by traversing the entire directory hierarchy and changing the ownership of each node to the root user within the container after the mount - an error-prone operation that can take approximately 40 seconds for a large enough root filesystem [Vas20]. Not only is this operation slow, but it affects the ownership values on the host system, thereby necessitating that containers with different identifier mappings must have a dedicated copy of their root filesystem, which wastes storage. This is the primary reason that rootless containers have not gained widespread adoption. The problem was addressed in v5.12 by allowing user-space processes to associate an identifier mapping with every mount point [Edg22] through a new system call `mount_setattr()`. This identifier mapping is used to translate the caller's user identifier to the filesystem user identifier in a localised way [Ker22]. In general, both examples above show that kernel developers are actively working on hardening the noninterference boundary and supporting container software in user-space.

7.2 Future work

This work can be further improved in two areas - the implementation of the container runtime and the benchmarking tool. The former is discussed first. The current implementation of the runtime does not incorporate the kernel's capability framework, which allows for fine-grained control over the container's capability sets. Another unsupported (and very important) feature is resource control groups, also referred to as *cgroups*. Resource control groups are a feature in the kernel that primarily deals with temporal noninterference. It is responsible for restricting the usage of various resources, such as processing units, memory and disk to prevent denial of service attacks. Properly configuring control groups without degrading application performance is a nuanced challenge that, in the authors opinion, is an interesting topic for a thesis. Another innovative technology in the kernel is the eBPF subsystem, which allows for user-space programs to safely inject source code into arbitrary points in the kernel and sample data. This technology can be integrated within the runtime, e.g for sampling behavior data of containerised processes and acting upon it, e.g if malicious intents are detected. The benchmark tool can also utilise eBPF for creating custom performance statistics.

Chapter 8

Appendix A

```
{
    "process": {
        "args": [
        "nsbench-disk-workload"
        ],
        "cwd": "/"
    },
    "root": {
        "path": "./workloads/rootfs/disk",
        "readonly": false
    },
    "namespaces": [
        {
        "type": "user"
        },
        {
        "type": "net"
        },
        {
        "type": "ipc"
        },
        {
        "type": "mnt"
        },
        {
        "type": "uts"
        },
        {
        "type": "pid"
        }
    ],
    "uid_mappings": [
        {
```

```
            "container_id": 0,
            "host_id": 1000,
            "size": 1
            }
    ],
    "gid_mappings": [
            {
            "container_id": 0,
            "host_id": 1000,
            "size": 1
            }
    ],
    "hostname": "nsbench-disk-workload",
    "hooks": {
        "on_runtime_create": [
            {
                "path": "/usr/bin/setup-network",
                "args": ["-a", "192.168.0.101"],
                "env": ["PATH=/usr/bin"],
                "timeout": 2
            }
        ],
        "on_container_created": [],
        "on_container_start": [],
        "on_container_started": [],
        "on_container_stopped": []
    }
}
```

Code snippet 8.1: Open Containers Initiative Configuration File Example

```go
func DoInContainerNamespace(containerPid, namespace int, doer func() error) error {
    runtime.LockOSThread()
    defer runtime.UnlockOSThread()

    ns, ok := namespaces[namespace]
    if !ok {
        return ErrNamespaceUnsupported
    }

    oldNsPath := path.Join("/proc", "self", "ns", ns)
    oldNsFd, err := unix.Open(oldNsPath, unix.O_RDONLY|unix.O_CLOEXEC, 0)
    if err != nil {
        return err
    }
```

```
    defer unix.Close(oldNsFd)

    newNsPath := path.Join("/proc", strconv.Itoa(containerPid), "ns", ns)
    newNsFd, err := unix.Open(newNsPath, unix.O_RDONLY|unix.O_CLOEXEC, 0)
    if err != nil {
        return err
    }
    defer unix.Close(newNsFd)
    if err := unix.Setns(newNsFd, namespace); err != nil {
        return err
    }
    defer unix.Setns(oldNsFd, namespace)
    return doer()
}
```

Code snippet 8.2: Joining an arbitrary namespace, executing a function within it, and returing back to the original namespace

List of Figures

List of Tables

Source Code Content

References

[ACS20] Anjali, Tyler Caraza-Harter, and Michael M. Swift. "Blending Containers and Virtual Machines: A Study of Firecracker and GVisor". In: *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE '20. Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 101–113. ISBN: 9781450375542. DOI: 10.1145/3381052.3381315. URL: https://doi.org/10.1145/3381052.3381315.

[Aga20] Alexandru Agache et al. "Firecracker: Lightweight Virtualization for Serverless Applications". In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 419–434. ISBN: 978-1-939133-13-7. URL: https://www.usenix.org/conference/nsdi20/presentation/agache.

[Cod59] E. F. Codd et al. "Multiprogramming STRETCH: Feasibility Considerations". In: *Commun. ACM* 2.11 (Nov. 1959), pp. 13–17. ISSN: 0001-0782. DOI: 10.1145/368481.368502. URL: https://doi.org/10.1145/368481.368502.

[Esx06] Vmware Esx. "Performance Evaluation of Intel EPT Hardware Assist". In: 2006.

[Fel15] Wes Felter et al. "An updated performance comparison of virtual machines and Linux containers". In: *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Mar. 2015, pp. 171–172. DOI: 10.1109/ISPASS.2015.7095802.

[Ker10] Michael Kerrisk. *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. 1st. USA: No Starch Press, 2010. ISBN: 1593272200.

[Kle09] Gerwin Klein et al. "SeL4: Formal Verification of an OS Kernel". In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. SOSP '09. Big Sky, Montana, USA: Association for Computing Machinery, 2009, pp. 207–220. ISBN: 9781605587523. DOI: 10.1145/1629575.1629596. URL: https://doi.org/10.1145/1629575.1629596.

[Kue17] Simon Kuenzer et al. "Unikernels Everywhere: The Case for Elastic CDNs". In: *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE '17. Xi'an, China: Association for Computing Machinery, 2017, pp. 15–29. ISBN: 9781450349482. DOI: 10.1145/3050748.3050757. URL: https://doi.org/10.1145/3050748.3050757.

[Lv12] Hui Lv et al. “Virtualization Challenges: A View from Server Consolidation Perspective”. In: *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*. VEE ’12. London, England, UK: Association for Computing Machinery, 2012, pp. 15–26. ISBN: 9781450311762. DOI: 10.1145/2151024.2151030. URL: https://doi.org/10.1145/2151024.2151030.

[Man17] Filipe Manco et al. “My VM is Lighter (and Safer) than Your Container”. In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP ’17. Shanghai, China: Association for Computing Machinery, 2017, pp. 218–233. ISBN: 9781450350853. DOI: 10.1145/3132747.3132763. URL: https://doi.org/10.1145/3132747.3132763.

[PG74] Gerald J. Popek and Robert P. Goldberg. “Formal Requirements for Virtualizable Third Generation Architectures”. In: *Commun. ACM* 17.7 (July 1974), pp. 412–421. ISSN: 0001-0782. DOI: 10.1145/361011.361073. URL: https://doi.org/10.1145/361011.361073.

[PR17] Reid Priedhorsky and Tim Randles. “Charliecloud: Unprivileged Containers for User-Defined Software Stacks in HPC”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’17. Denver, Colorado: Association for Computing Machinery, 2017. ISBN: 9781450351140. DOI: 10.1145/3126908.3126925. URL: https://doi.org/10.1145/3126908.3126925.

[PVM21] Francesco Pagano, Luca Verderame, and Alessio Merlo. “Understanding Fuchsia Security”. In: *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications* 12.3 (2021), pp. 47–64.

[Ran20] Allison Randal. “The Ideal Versus the Real: Revisiting the History of Virtual Machines and Containers”. In: *ACM Comput. Surv.* 53.1 (Feb. 2020). ISSN: 0360-0300. DOI: 10.1145/3365199. URL: https://doi.org/10.1145/3365199.

[SGG18] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. 10th. Wiley Publishing, 2018. ISBN: 9781119320913.

[SLC20] Sogand Shirinbab, Lars Lundberg, and Emiliano Casalicchio. “Performance evaluation of containers and virtual machines when running Cassandra workload concurrently”. In: *Concurrency and Computation: Practice and Experience* 32.17 (2020), e5693. DOI: https://doi.org/10.1002/cpe.5693. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.5693. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.5693.

[SN05] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005. ISBN: 1558609105.

[WR12] Carl Waldspurger and Mendel Rosenblum. "I/O Virtualization". In: *Commun. ACM* 55.1 (Jan. 2012), pp. 66–73. ISSN: 0001-0782. DOI: `10.1145/2063176.2063194`. URL: `https://doi.org/10.1145/2063176.2063194`.

[You19] Ethan G. Young et al. "The True Cost of Containing: A gVisor Case Study". In: *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. Renton, WA: USENIX Association, July 2019. URL: `https://www.usenix.org/conference/hotcloud19/presentation/young`.

[Zha18] Qi Zhang et al. "A Comparative Study of Containers and Virtual Machines in Big Data Environment". In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. 2018, pp. 178–185. DOI: `10.1109/CLOUD.2018.00030`.

Online References

[Doc22] The Linux Kernel Documentation. *Idmappings*. 2022. URL: https://www.kernel.org/doc/html/latest/filesystems/idmappings.html (visited on 09/25/2022).

[Edg22] Jake Edge. *ID-mapped mounts*. 2022. URL: https://lwn.net/Articles/896255/ (visited on 09/30/2022).

[Fed22] Fedora. *Fedora 36 Server*. 2022. URL: https://getfedora.org/en/server/download/ (visited on 09/25/2022).

[Gla21] Alexey Gladkov. *Count rlimits in each user namespace*. 2021. URL: https://patchwork.kernel.org/project/linux-hardening/cover/cover.1619094428.git.legion@kernel.org/ (visited on 05/26/2022).

[Goo18] Google. *gVisor*. 2018. URL: https://github.com/google/gvisor (visited on 05/26/2022).

[Ini15] Open Container Initiative. *Open Containers Initiative Website*. 2015. URL: https://opencontainers.org/ (visited on 05/26/2022).

[Ini16a] Open Container Initiative. *Operations*. 2016. URL: https://github.com/opencontainers/runtime-spec/blob/main/runtime.md#operations (visited on 08/20/2022).

[Ini16b] Open Container Initiative. *Operations*. 2016. URL: https://github.com/opencontainers/runtime-spec/blob/main/runtime.md#lifecycle (visited on 08/20/2022).

[Ini16c] Open Container Initiative. *The 5 principles of Standard Containers*. 2016. URL: https://github.com/opencontainers/runtime-spec/blob/main/principles.md (visited on 05/26/2022).

[Ker22] The Linux Kernel. *Idmappings*. 2022. URL: https://www.kernel.org/doc/html/latest/filesystems/idmappings.html (visited on 10/01/2022).

[lou] louwrentius. *Fio-plot*. URL: https://github.com/louwrentius/fio-plot (visited on 09/25/2022).

[Ltd22] Raspberry Pi Ltd. *BCM2711 ARM Peripherals*. 2022. URL: https://datasheets.raspberrypi.com/bcm2711/bcm2711-peripherals.pdf (visited on 09/24/2022).

[Vas20] Mauricio Vasquez. *Experimental: Support idmapped mounts kernel patches*. 2020. URL: https://github.com/containerd/containerd/pull/4734 (visited on 09/30/2022).

[ZK04] Peter Zaitsev and Alexey Kopytov. *Sysbench OLTP*. 2004. URL: `https://github.com/akopytov/sysbench` (visited on 05/26/2022).

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel verfasst habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Berlin, den 04.10.2022

Atanas Denkov