Evaluation of Container Technologies for an Embedded Linux Device

Harri Manninen

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 12.7.2020

Supervisor

Prof. Jan Blech

Advisor

M.Sc. Vesa Jääskeläinen

Aalto University
School of Electrical
Engineering

Aalto University
School of Electrical
Engineering

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Harri Manninen

Title Evaluation of Container Technologies for an Embedded Linux Device

Degree programme Automation and Electrical Engineering

Major Control, Robotics and Autonomous Systems **Code of major** ELEC3025

Supervisor Prof. Jan Blech

Advisor M.Sc. Vesa Jääskeläinen

Date 12.7.2020 **Number of pages** 51 **Language** English

Abstract

Application containers have become a popular method of deploying software during recent years. Containers are used to virtualize and isolate applications from the underlying host system which provides various benefits, such as increased security and portability. However, containers also introduce overhead to the system which might be a problem in embedded systems, where the availability of computational resources is limited.

The goal of this thesis was to recommend an application container technology to be used in an embedded Linux device. In order to reach this goal, four container technologies were chosen to be evaluated by executing tests and measuring the caused overhead by each technology. Additionally, the security and usability of each technology was evaluated. The technologies chosen for evaluation were Docker, balenaEngine, LXC and Flatpak. The testing was carried out using a well-known embedded device Raspberry Pi 4.

The results showed that while there were some differences between the tested technologies, in most cases the amount of overhead introduced was relatively small. Furthermore, no major issues were found in terms of security or usability with any of the four technologies.

Keywords container, embedded, Linux, Docker, balenaEngine, LXC, Flatpak

Aalto University
School of Electrical Engineering

Aalto-yliopisto, PL 11000, 00076 AALTO
www.aalto.fi
Diplomityön tiivistelmä

Tekijä Harri Manninen

Työn nimi Arviointi konttiteknologioista sulautetulle Linux-laitteelle

Koulutusohjelma Automaatio ja sähkötekniikka

Pääaine Säätötekniikka, robotiikka ja autonomiset järjestelmät **Pääaineen koodi** ELEC3025

Työn valvoja Prof. Jan Blech

Työn ohjaaja DI Vesa Jääskeläinen

Päivämäärä 12.7.2020 **Sivumäärä** 51 **Kieli** Englanti

Tiivistelmä

Ohjelmistokontit ovat saavuttaneet suuren suosion sovellusten käyttöönottotapana viime vuosien aikana. Kontit virtualisoivat ja eristävät ohjelmistoja alla olevasta käyttöjärjestelmästä, mistä on monia hyötyjä, kuten parempi tietoturva ja siirrettävyys. Kontit kuitenkin kasvattavat järjestelmän resurssien kulutusta, mikä voi olla ongelmallista sulautetuissa järjestelmissä, joissa resursseja on saatavilla vain rajallinen määrä.

Tämän diplomityön tarkoituksena oli suositella konttiteknologiaa käytettäväksi sulautetussa Linux-laitteessa. Tämän tavoitteen saavuttamiseksi neljä konttiteknologiaa valittiin arvioitavaksi. Arvionti perustui testien suorittamiseen, joissa mitattiin konttien aiheuttamaa ylimääräistä resurssien kulutusta. Lisäksi teknologioiden tietoturvaa sekä käytettävyyttä arvoitiin. Teknologiat, jotka valittiin tarkasteltaviksi, olivat Docker, balenaEngine, LXC ja Flatpak. Testit suoritettiin Raspberry Pi 4:llä, joka on tunnettu sulautettu laite.

Tulokset osoittivat, että vaikka arvioitujen teknologioiden välillä oli joitakin eroja, suurimmassa osassa tapauksista ylimääräinen resurssien käyttö oli kuitenkin suhteellisen vähäistä. Lisäksi todettiin, että millään testatuista teknologioista ei ollut suuria tietoturvariskejä tai huomattavia ongelmia käytettävyydessä.

Avainsanat kontti, sulautettu, Linux, Docker, balenaEngine, LXC, Flatpak

Preface

First, I want to thank Vaisala for this thesis opportunity. I was allowed to work solely on the thesis for six months, which made it possible for me to complete it in a timely manner. I was also given plenty of support and advice whenever it was needed.

I also want to thank Jan Blech for guidance with the thesis and Vesa Jääskeläinen for helping with the technical aspects of the thesis.

Otaniemi, 25.6.2020

Harri Manninen

Contents

Abbreviations

ABI	Application Binary Interface
API	Application Programming Interface
CLI	Command Line Interface
CPU	Central Processing Unit
GID	Group Identifier Number
I/O	Input/output
IoT	Internet of Things
IP	Internet Protocol
IPC	Inter-process Communication
ISA	Instruction Set Architecture
KiB	Kibibyte
MAC	Mandatory Access Control
MB	Megabyte
OCI	Open Container Initiative
OS	Operating System
PID	Process Identifier Number
REST	Representational State Transfer
SSH	Secure Shell
TCP	Transmission Control Protocol
UID	User Identifier Number
VM	Virtual Machine

1 Introduction

Application containers have become increasingly more popular during recent years in the field of software engineering. A containerized application includes the binaries of the software with all its dependencies in a single, stand-alone package called a container image. Containers are used to virtualize and isolate software from the platform they are executed on, which makes usage of the software more secure. Security can be further improved by restricting the access of a containerized applications to the underlying hardware. Furthermore, the usage of containers makes it easy to control the access to shared resources between applications. Additionally, containers improve portability of applications because container images are stand-alone packages. Thus, they can be easily deployed on many different systems, since containers cause no compatibility issues with required dependencies [1].

However, using containers also has a few drawbacks. The initialization of containers requires time, and application start-up times might be longer than running them natively. Moreover, containers introduce some amount of overhead to the system. For example, containers use more memory than native applications due to their usage of certain operating system (OS) features required for virtualization. Furthermore, they consume more space from the system, since all dependencies are included, whereas executing natively only the binaries would be needed if required dependencies were already installed on the host OS. When considering servers and personal computers, the amount of overhead is often very small in relation to the resources available and does not cause any issues. However, overhead should be taken into account when using containers on embedded systems, where the amount of resources available is limited.

Various container tools and technologies have been developed for different aspects of containerization. A container engine is used to process user input, pull container images from registry servers and call a container runtime. Container runtime is a lower level component that is responsible for starting containerized processes and setting up the security features used in the container. However, container runtimes do not necessarily need to be called by container engines; they can also be used by hand. Some examples of container engines include Docker [8], rkt [9] and LXC [10], while examples of container runtimes comprise runc [11] and Kata Containers [12]. Furthermore, there are also container orchestration tools, such as Kubernetes [13], which focuses on managing and scheduling the execution of clusters of containers. Containers can be further divided into application and system containers. Application containers, for example Docker, virtualize individual applications, while system containers, such as LXC, virtualize an entire OS and create an environment similar to one created by a virtual machine (VM) [2].

Although much research has been devoted to containerization, only a few studies have specifically focused on the usage of containers in embedded Linux devices. Noronha *et al.* [1] compared the performance between LXC containers and native execution with multiple embedded microprocessors. Lammi [22] compared the performance of Docker containers and native execution in an embedded system. However, these studies only focused on the performance of a single container engine

technology.

The goal of this thesis is to recommend an open source application container technology for use by Vaisala Oyj in an embedded Linux device. In order to reach this goal, the study evaluates different available technologies by executing various tests with them measuring the caused overhead. Factors evaluated are CPU, memory and mass storage usage as well as power consumption. Additionally, the thesis also considers other non-measurable factors, such as security as well as ease in developing and deploying. Because the thesis focuses only on evaluating different technologies, implementation of the containerization environment with Vaisala's device will remain outside the scope of this work.

The testing is carried out using a well-known embedded Linux device Raspberry Pi 4 (RPI). There should not be any compatibility issues between these two devices, since containerization on Linux is based on a few Linux kernel features that are available on all recent Linux kernel versions. The Linux distribution that will be used on the RPI will be built using the Yocto Project [38], which is a tool used to build custom Linux distributions.

The remainder of this work is organized as follows. Chapter 2 reviews the concept of virtualization and discusses the most popular virtualization technologies. The chapter explains the most important Linux kernel features utilized in virtualization. Additionally, suitable container technologies for embedded devices are discussed in depth. Chapter 3 explains the methodology used in this thesis. Chapter 4 discusses the implementation of the tests. Chapter 5 presents and evaluates the results acquired from tests. Finally, Chapter 6 provides a summary of the whole thesis.

2 Background

This section explains virtualization and key Linux kernel features used in containerization, and discusses multiple different container engine technologies. Additionally, previous research about virtualization and containers is explored.

2.1 Virtualization

In the context of computing, term virtualization means the creation of a virtual version of a device or a resource. For example, storage devices and network resources can be virtualized. When creating a virtual system, a part of the host system's resources are partitioned and separated to be used by it. Additionally, the virtual part is isolated from the host system, which means that if it crashes, the host system is unaffected [5]. Furthermore, the virtual system is encapsulated which means that it is a complete and independent entity and from within this virtualized part, it seems to be a whole system in itself [22].

Virtualization can be divided into two widely used types: hardware virtualization and operating-system-level virtualization. Hardware virtualization is utilized by virtual machines, while containers use OS-level virtualization. These techniques operate on different interfaces that exist within a computer system. Modern computing stack can be described with machine reference model [6] shown in Figure 1. Hardware virtualization happens at the instruction set architecture (ISA) level whereas OS-level virtualization takes place at the application binary interface (ABI) level.

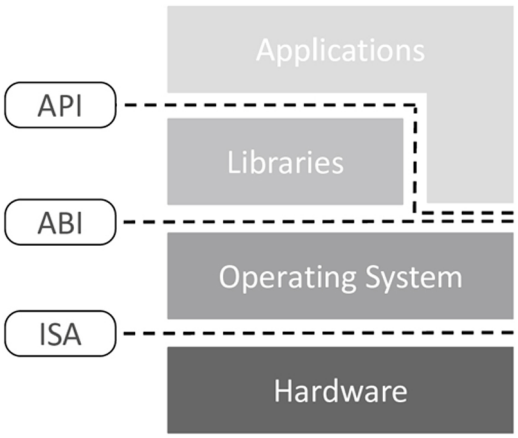

Figure 1: Machine reference model [7].

When VMs are used, a hypervisor is needed that is responsible for managing the VMs. Two kinds of hypervisors exist, which are type-1 and type-2 hypervisors. Type-1 hypervisors (fig. 2A), also known as bare-metal hypervisors, run directly on the physical hardware. This means that the virtualization software is installed directly on the hardware and no OS is needed. Type-2 hypervisors (fig. 2B), on the other hand, are higher-level software layers running on top of operating systems. They are easier to use and manage than type-1 hypervisors, but have larger overhead [7].

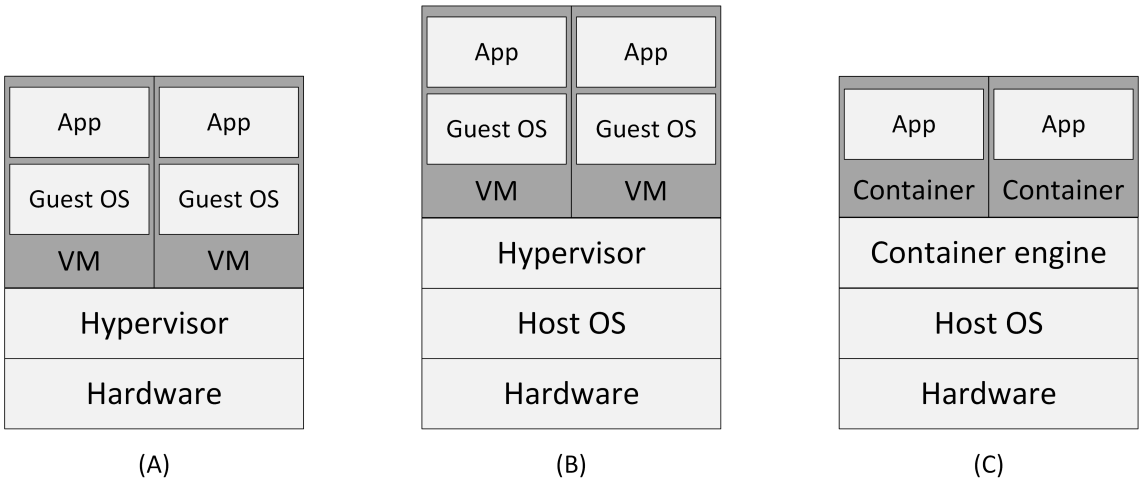

Figure 2: (A) Hardware-level virtualization, type-1 hypervisor; (B) hardware-level virtualization, type-2 hypervisor; (C) OS-level virtualization. Adapted from [7], [42].

Furthermore, type-1 hypervisors are more secure than type-2 hypervisors, since there is no attack-prone operating system needed [28]. With containers, virtualization happens at OS-level and hypervisors are not needed (fig. 2C). Containerization is enabled by a few key Linux kernel features and because hardware does not need to be virtualized [16], containers offer a very light way of virtualization as an alternative to VMs.

Containers excel in many areas in application virtualization when compared to VMs. The size of containers is a lot smaller, usually measured within tens of megabytes, while the size of VMs can be multiple gigabytes. Furthermore, containers use less memory because hypervisors are not needed. The lack of hypervisors also gives containers the edge with application performance, especially with the usage of I/O devices. Additionally, the number of containers that can be run on one system is far greater than the number of VMs. This is because each VM needs its dedicated resources, while container images can be created in a layers, allowing the applications to share some of their dependencies [17].

Development of applications is also faster with containers compared to VMs due to the lightweight nature of container images. Building new container images is quick which reduces the time required for testing, development and deployment. Furthermore, sharing containers is easier between developers and different environments used in the development process, such testing and production environments. Additionally, application developers do not need to support and maintain multiple versions of the software for different Linux distributions, since a single containerized version can run on practically any Linux system [16].

When it comes to security aspects, VMs are better than containers. This is because a VM shares almost no parts with its host system, making it harder to harm from within the VM. Additionally, VMs can also be used run applications from different operating systems, whereas containers are limited to its host OS. For example, a Windows host machine can not execute a Linux container image as it is,

but it can run a Linux VM, which can then be used to execute the image.

2.2 Key Linux kernel features

There are a few important Linux kernel features that enable containers. The most important ones are namespaces, providing a way to abstract system resources, and cgroups, making it possible to limit the resource usage of processes.

2.2.1 Control groups

Control groups (cgroups) is a Linux kernel feature that can be used to limit and monitor the resource usage of processes. As described by the Linux control groups manual pages [29], there are two cgroups implementations: cgroups version 1 and version 2. Version 1 was initially released in Linux 2.6.24 while the development of version 2 began in Linux 3.10 and it was officially released in Linux 4.5. The second version was created because of problems with the first version, and while it is intended as a replacement for cgroups v1, it currently supports only a subset of features that exists in version 1.

Limiting and monitoring the resource usage is done by organizing processes into hierarchical groups. Each of these groups is a cgroup that is bound to a set of limits or parameters which can be defined via the cgroup filesystem. A subsystem, also known as resource controller, is what is used to achieve the limitation of resources of each cgroup. Various resource controllers exist that make it possible to control things such as memory usage, CPU time and freezing and resuming the execution of processes in a cgroup. Each controller's cgroups are arranged in a hierarchy, in which the limitations for a parent cgroup also apply for its child cgroups. For example, if the memory of processes in a cgroup is limited to 100 MB, the maximum memory that can be set for processes in its children cgroups is also 100 MB.

How cgroups work in practice is that first, controllers are mounted to an empty cgroup filesystem, often located in a virtual filesystem at */sys/fs/cgroup*. After mounting e.g. CPU controller there, it would found at */sys/fs/cgroup/cpu*. Next, a new cgroup, e.g. */sys/fs/cgroup/cpu/cg1*, can be added. Processes can then be added to this cgroup by writing their PID to *cgroup.procs* file located at */sys/fs/cgroup/cpu/cg1/cgroup.procs*. After this, the CPU time of these processes is defined by the rules set for the cpu controller. A process can only be a part of a single cgroup within a hierarchy. Writing the PID of a process to *cgroup.procs* will automatically remove the process from the previous cgroup it belonged to.

In cgroup v1 implementation multiple hierarchies can exist simultaneously, meaning that there can be a hierarchy for each resource controller [30]. In cgroups v2, there is only a single hierarchy, which means that all controllers used in v2 are automatically bound to this single filesystem. It is possible to mount different controllers simultaneously under the v1 and v2 hierarchies [31]. The only limitation is that the same controller can not be mounted on both v1 and v2 hierarchies at the same time. There are 13 controllers available in cgroups v1, while cgroups v2 implements only 7 of them. Some examples are:

cpu	Can be used both to guarantee a minimum amount and to define an upper limit of CPU time allocated to processes
memory	For reporting and limiting used process memory, kernel memory and swap
devices	Controls cgroups' read, write, and create permissions of device files
freezer	May be used to suspend and resume the execution of processes in a cgroup
perf_event	For performance monitoring of processes within a cgroup
pids	Can be used to set a limit on the number of processes that can be created within a cgroup and its children

All above controllers except *devices* are also available in cgroups v2. However, not all of them are updated for the version 2 implementation: *pids* and *perf_event* controllers are exactly the same as in cgroups version 1 [29].

2.2.2 Namespaces

The first approaches towards containerization in Linux involved the *chroot* system call which was originally introduced in UNIX Version 7 back in 1979 [34]. *Chroot* can be used to change the root directory of a process and its children. However, it is not very secure since escaping from the new root directory is very easy and there is no isolation between the chroot environment and processes outside of it. To achieve properly isolated environments, namespaces can be used instead.

Linux namespaces provide a way to wrap global system resources in an abstraction layer. Similarly to *chroot*, which allows processes to see any arbitrary directory as the root directory, namespaces make it possible to do process specific modifications to other aspects of the OS. Namespaces are essential in containerization, since they allow different groups of processes to be isolated from each other and provide them different views of the system by making it appear for processes within a namespace that they they have their own isolated instance of the abstracted global resource. Eight different namespaces exist: *cgroup, IPC, network, mount, PID, user, UTS,* and *time* [32].

Namespaces are managed with *nsproxy* structures shown in Program 1. It includes a *count* variable which keeps track of the number of processes associated with it. Additionally, it has pointers to all namespaces, to which the associated process belongs to, except the user namespace, which is managed in a different manner. Each process is associated with a single *nsproxy* instance. Initially, at system startup, there is only a single *nsproxy* structure to which all processes point to. When either an existing or a new process is associated with a new namespace, a new *nsproxy* instance containing pointer to the new namespace is created.

```
1  struct nsproxy {
2          atomic_t count;
3          struct uts_namespace *uts_ns;
4          struct ipc_namespace *ipc_ns;
5          struct mnt_namespace *mnt_ns;
6          struct pid_namespace *pid_ns_for_children;
7          struct net           *net_ns;
8          struct time_namespace *time_ns;
9          struct time_namespace *time_ns_for_children;
10         struct cgroup_namespace *cgroup_ns;
11 };
```

Program 1: *nsproxy* structure source code [35].

Figure 3 illustrates the relationships between processes, *nsproxies* and namespaces. Each process has a structure called *task_struct* that contains information about the process and points to a *nsproxy*, which then has pointers to all namespaces.

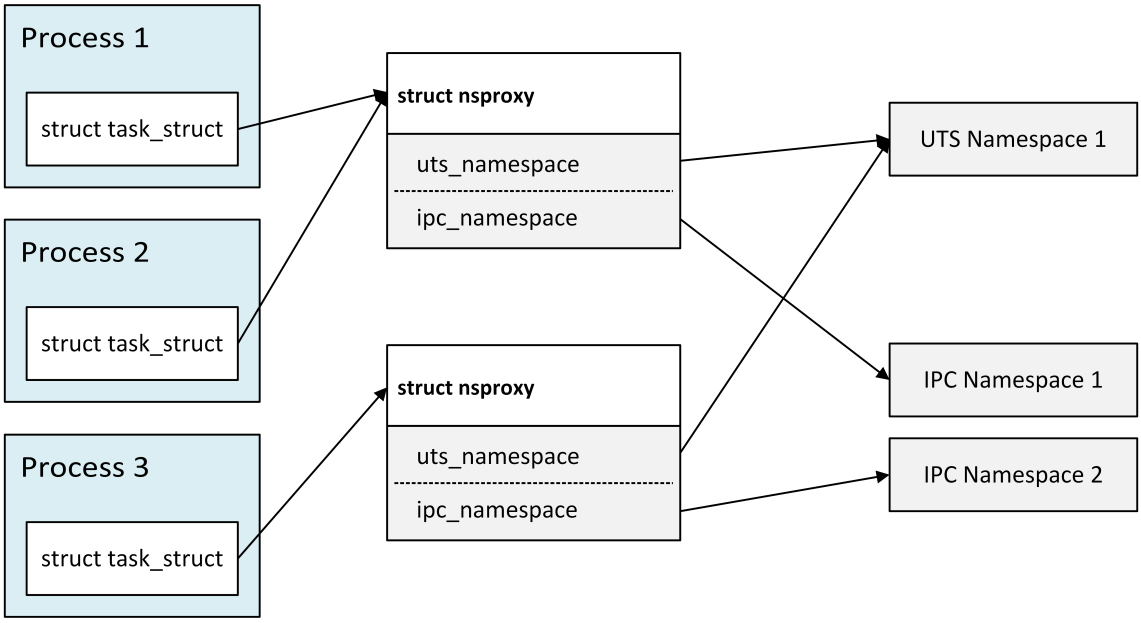

Figure 3: Connection between processes and namespaces. Adapted from [33, pp. 50].

New namespaces can be created with Linux system calls *clone* and *unshare*. *Clone* is used to create new processes, while *unshare* in most cases moves the calling process to a new namespace. One or more new namespaces that should be created can be specified to both of these functions as arguments. If *unshare* is used to create either a PID or time namespace, the calling process will not be made a part of the new namespace. Instead, if child processes are created in the future, they will be moved to the new namespace [71]. These differences can be seen in the *nsproxy* structure in Program 1, where these namespaces have pointers for their children. If the *clone* system call is used to create new namespaces instead, a new child process is created

and made a member of the new namespace. Additionally, *setns* system call can be used to move the calling process to an existing namespace [32].

Information about processes' namespaces can be found from the *proc* filesystem. Each file in */proc/$PID/ns* is a symbolic link to the namespace that the process is in. They can be used to track the namespaces that a process of interest is a part of and to perform operations such as *setns* [34]. Namespace information of a single process can be seen at Program 2.

```
$ ls -l /proc/self/ns/
cgroup -> cgroup:[4026531835]
ipc -> ipc:[4026531839]
mnt -> mnt:[4026531840]
net -> net:[4026531957]
pid -> pid:[4026531836]
user -> user:[4026531837]
uts -> uts:[4026531838]
```

Program 2: Truncated output of *ls -l* command executed in the namespace directory of a single process.

PID and user namespaces are hierarchical while the rest of the namespaces are non-hierarchical [32]. Hierarchical namespaces are nested, which means that all namespaces except the initial root namespace have a parent namespace to which they are related to. The kernel sets a limit of 32 nested levels of namespaces in both PID and user namespaces [36], [37]. Each namespace is only aware of itself and its descendant namespaces, but do not have any information about their parent or any higher level namespaces. This situation is demonstrated with the PID namespace in Figure 5. Non-hierarchical namespaces do not have any relationships between each other [33], meaning that if there are multiple instances of the same namespace, each of them is only aware of themselves.

Cgroup namespaces virtualize processes' view of control groups. All cgroup namespaces have their own set of cgroup root directories, which are the base points for the relative locations of the cgroups that a process is associated with. This information for each process can be found in */proc/$PID/cgroup* file [45].

```
# mkdir -p /sys/fs/cgroup/freezer/containers/app-id
# echo $$ > /sys/fs/cgroup/freezer/containers/app-id/cgroup.procs
# cat /proc/self/cgroup | grep freezer
10:freezer:/containers/app-id
# unshare --cgroup /bin/bash
# cat /proc/self/cgroup | grep freezer
10:freezer:/
```

Program 3: Cgroup namespaces demonstration. Adapted from [31].

Control group namespaces can be used to hide directory paths that exist outside of a container environment. This is demonstrated in Program 3 with the freezer cgroup.

The first two lines create and add the current shell process to */containers/app-id* cgroup. Next, the information available for the process about its freezer cgroup is inspected, and it can be seen that it contains the path */containers/app-id*. This can be considered as information about the host system which should not be visible within a virtualized container environment [31]. However, after creating a new cgroup namespace with the *unshare* command, the process is no longer aware of the whole path, as demonstrated in the last two lines of the program.

Another useful feature of cgroups namespaces is that it allows processes to be isolated from their ancestor cgroups. Without such isolation, each process would need to be provided with the full cgroup pathnames, all of which would also need to be unique [45]. This prevents conflicts with names when, for example, two instances of the same program running simultaneously would try to manage cgroups with same names [22].

Inter-process communication (IPC) namespaces isolate certain resources used for communication between processes. All System V IPC objects, which are message queues, semaphores and shared memory segments can be isolated with IPC namespaces, while the only POSIX IPC mechanism that can be affected is message queues [41]. Because IPC namespaces are non-hierarchical, two processes residing in different namespaces can not use aforementioned IPC methods for communication between each other [22].

Network namespaces provide isolation for resources associated with networking, such as network devices, protocol stacks, IP routing tables and firewall rules [46]. They can be used, for example, to limit or completely remove the access to internet for a namespace and all its processes.

A physical network interface, such as *eth0*, can exist only in a single namespace at a time. If all processes within a namespace terminate, all its physical network interfaces are moved back to the initial network namespace [46]. Separate namespaces can be connected to each other with virtual Ethernet (*veth*) devices. They exist always in interconnected pairs, and when one device of the pair receives a packet, it is transmitted to the other device instantly. Two network namespaces can be connected together by a single *veth* device pair by placing one end of it to the first namespace and the other end to the second namespace. Another way of connecting two different namespaces is to have two pairs of *veth* devices acting as tunnels from the namespaces to a shared bridge [47]. Both cases can be seen in Figure 4.

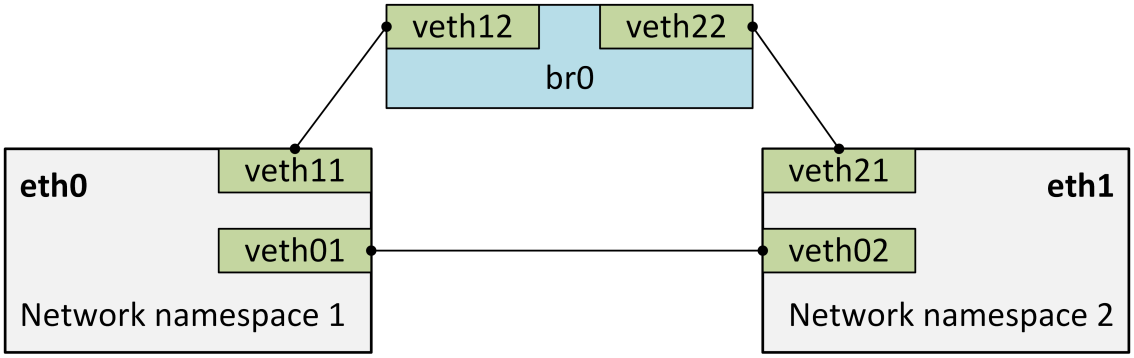

Figure 4: Connections between network namespaces.

Mount namespaces isolate mount points seen by processes within each namespace instance. Mount points seen by a specific process are listed in */proc/$PID/mountinfo* file. All processes that belong to the same mount namespace share the same view of this file [43]. Mount namespaces makes it possible provide different views of the same directory hierarchy to different processes that belong to separate namespace instances. This is demonstrated in Program 4.

```
$ sudo unshare --mount /bin/bash
# mkdir testdir
# mount -n -o size=1m -t tmpfs tmpfs testdir/
# touch testdir/private-file
```

Program 4: Mount namespaces demonstration. Adapted from [44].

```
/
└── testdir
    └── private-file

1 directory, 1 file
```

(A)

```
/
└── testdir

1 directory, 0 files
```

(B)

Program 5: Output of *tree* executed in new mount namespace (A) and outside of the namespace (B).

First, a new mount namespace is created with the *unshare* command. The following two lines create a new directory and mount it. Finally, a new file is created inside the new namespace. Then, *tree* command, which shows the directory structure, is executed inside and outside of the new namespace and the outputs are shown in Program 5. It can be seen that the file created within the new namespace is not visible outside of it.

PID namespaces isolate the process ID (PID) number space, making it possible to have multiple processes with same PID numbers in different namespaces [36]. This means that each PID namespace has an init process with PID 1 which makes the namespace appear as an independent system within itself. This can be seen in

Figure 5, where there is a parent PID namespace with two children. Each of the children have three processes with the same PIDs. Because PID namespaces are hierarchical, the parent namespace is aware of its children processes, but knows them with different PID values. Hence, even though there are only 9 different processes in the system, 15 PIDs are required to represent them, and the value depends on the namespace in which the process is observed [33].

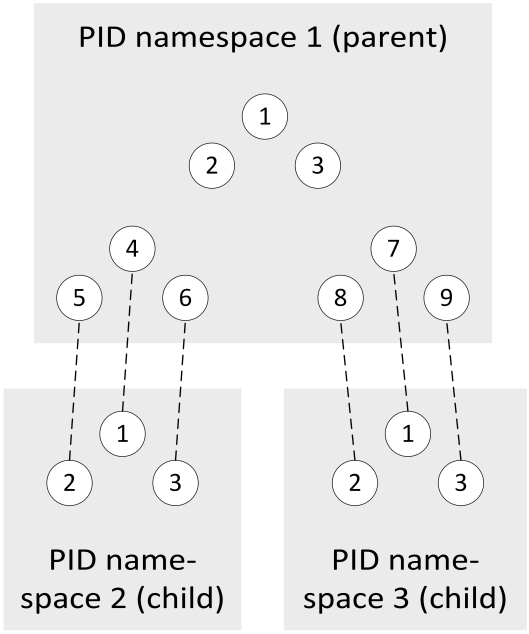

Figure 5: PID namespaces. Adapted from [33, pp. 49]

User namespaces isolate user IDs and group IDs and other security-related features, such as keys and capabilities [37]. User namespace data structures are not a part of the *nsproxy* structure. Instead, all other namespaces include a pointer to a user namespace instance, because all other namespaces are a part of a singular user namespace [22]. When a new user namespace is created by a process, it is given all capabilities in the new namespace by default. Capabilities are a way of giving unprivileged processes permissions to specific tasks. For example, a process with the capability *CAP_CHOWN* is allowed to make arbitrary changes to file UIDs and GIDs [48].

User namespaces are the only namespace that can be created without root privileges, although this behaviour is disabled by default in some Linux distributions. Anyhow, this makes it possible for a normal unprivileged user to have root privileges inside a new user namespace, but their privileges and capabilities remain the same outside of it. This is demonstrated in Program 6.

```
$ id -u
1000
$ unshare --mount /bin/bash
unshare: unshare failed: Operation not permitted
```

```
$ unshare --user -r /bin/bash
# id -u
0
# unshare --mount /bin/bash
# echo "123" > file_owned_by_real_root.txt
bash: root-file: Permission denied
```

Program 6: User namespaces demonstration [22].

First, it is shown that a normal used with UID 1000 can not create a new mount namespace. However, creating a new user namespace is possible, where the user has root UID 0 and can now create a new mount namespace. Nonetheless, the user still has the same privileges outside of the user namespace, since it is not able to write to a file owned by the real root user outside of the namespace as demonstrated in the final two lines of the program.

UTS namespaces isolate hostname and domain name of the system. These names are set with *sethostname()* and *setdomainname()* and can be retrieved with either *uname()* or *gethostname()* and *getdomainnname()* system calls [39]. The name UTS derives from structure called *utsname* passed to *uname()* function which in turn derives from "UNIX Time-sharing System" [40].

Time namespaces are the newest addition to Linux namespaces, added in kernel version 5.6. Time namespaces isolate the boot and monotonic clocks of the system [32]. This is useful e.g. when migrating containers from one host to another, where the system clock values might differ between the hosts. By virtualizing these clocks with the time namespace, the issue can be avoided.

2.2.3 Mandatory access control

Mandatory access control (MAC) is an access control model that enhances the information security of a system. Linux uses discretionary access control (DAC) model by default, in which the owner of a file controls the read, write and execution permissions of groups and users. In MAC, these permissions are controlled on system level by predefined policies, and can not be changed by users [72]. In Linux, MAC is often done with Linux Security Module (LSM) which is a framework that can be used to create various security extensions. LSM is primarily used by MAC extensions such as Application Armor (AppArmor) and Security-Enhanced Linux (SELinux) [73]. Both of them are used to create program specific MAC policies that can define, for example, file permissions and network access.

2.2.4 UnionFS

Union file systems (UnionFS) is a Linux feature that allows different files and directories from separate locations to be merged and seen in a single virtual file system. Each directory that an UnionFS consists of is called a branch, all of which

have a priority level that is used to determine which files are visible in the UnionFS. If there there are multiple files with a same name in different branches, the one with the highest priority level is visible [56].

Union file systems can be useful in containerization. For example, Docker uses them to create container images [52]. Docker's container images consist of layers which are created with union file systems. This makes it possible for multiple container images to share different layers which saves storage space. Additionally, when changes are done to a container image, it is not necessary to build the whole image. Instead, only the layers that are affected by the changes need to be rebuilt.

2.3 Container engines

Container engine is a piece of software that is used to manage containers. It provides a command line interface and handles user input and pulls container images from registry servers. It is also responsible for calling a container runtime, which is a lower level component that actually runs containers [2].

This section takes a closer look on a few specific container engines that have been chosen as the most suitable candidates to use in an embedded device. Additionally, container images are discussed.

2.3.1 Container images

Historically, each container engine had their own non-standard format for container images, but nowadays most engines have started using the format defined by Open Container Initiative (OCI) [2]. The Open Container Initiative is a open source technical community with the goal of promoting a set of common, minimal, and open standards and specifications around container technologies. The OCI image specification is not tied to any particular client, vendor or project and is hardware and OS independent [24]. This means that container images that follow the OCI specification can be run on any OCI compliant container engine.

Images that follow the OCI image format consist of a manifest, a set of filesystem layers, a configuration and an optional image index. The image manifest contains metadata about the contents and dependensies of the image. Specifically, it provides a configuration and points to one or more filesystem layers for a single container image for a specific architecture and OS [26]. Filesystems used by container images are composed of filesystem layers. Each layer is an archived representation of a set of filesystem changes, such as files to be added, changed or removed relative to its parent layer. The configuration can be used to specify things such as username or UID, working directory, and ports that are exposed to the image [27]. The optional image index is a higher-level manifest that contains information about as set of images which can be for multiple different architectures and operating systems [25].

2.3.2 Docker

Docker is one of the most popular container engine technologies. It has been under development for many years, as its initial release was in 2013 [51], making it a

mature technology that supports many platforms and is well documented. The most common use cases for Docker are cloud-native applications but it can also be used with desktops and embedded devices.

The Docker Engine is built on container runtime called containerd [53]. Linux namespaces are used to achieve the isolation of containers and cgroups to limit their resource usage. Docker uses client-server architecture that can be seen in Figure 6. Docker daemon acts as a server and communicates to the Docker client using a REST API. The Docker client provides a command line interface (CLI) to the user with the *docker* command. The CLI can be used to send commands to the daemon to configure and control Docker objects such as containers, images and networks [52].

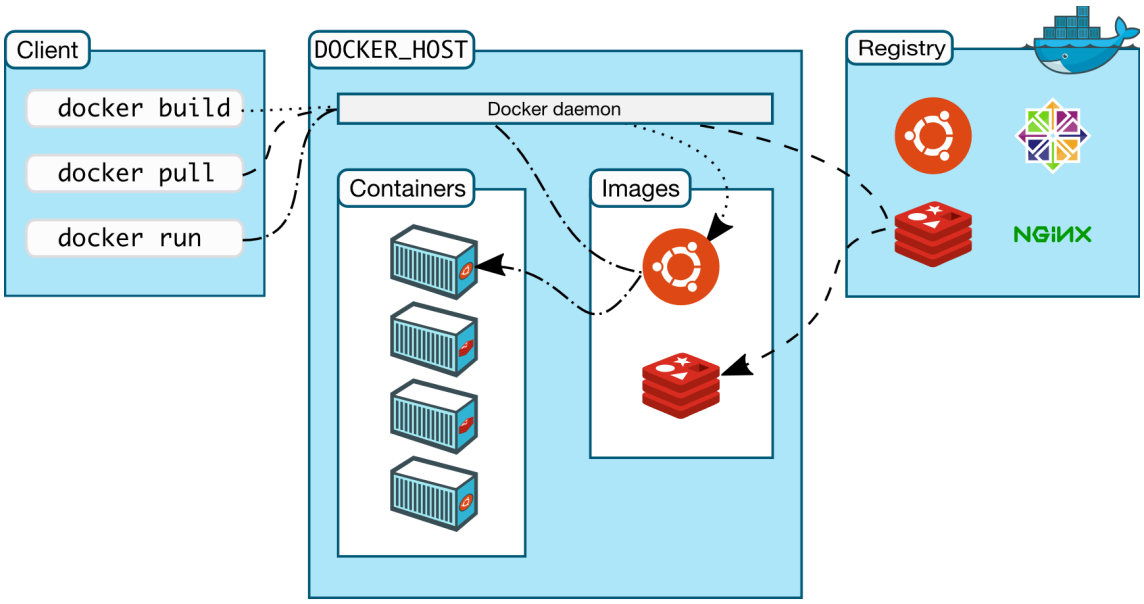

Figure 6: Docker architecture [52].

Docker images can be built using Dockerfiles. A Dockerfile is a text document that contains all instructions that are needed for building a complete container image. There are numerous base Linux distributions available which can be used as a starting point when creating container images. For example, using instruction *FROM ubuntu:latest* at the beginning of a Dockerfile will result in a container image that is based on the latest Ubuntu version. It is also possible to add any Linux packages or other applications to the container image by specifying them in the Dockerfile. Docker uses caching when building containers images from Dockerfiles. After the image is built once, only the instructions that were changed need to be rebuilt, which speeds up the building process. Images are stored in Docker registries. They can be either public or private and they are used in a similar manner as Git repositories. Docker images can be pulled from registries with the *docker pull* command [63].

Docker was chosen as one of the tested technologies because it is widely used and well supported and documented. Additionally, it offers many functionalities and tools that make building and distributing of container images easy. The usage of Docker is straightforward and the access rights of containers can be configured easily.

Lastly, Docker has already been used and tested in various studies and proven to be a feasible solution in embedded systems.

2.3.3 balenaEngine

BalenaEngine is a lightweight container engine specifically built for embedded and IoT use cases. Its features include small footprint, multi-arch support and conservative memory usage. It is based on Moby Project created by Docker, which is a framework providing components and tools that can be used to build specialized container systems [58]. This makes balenaEngine very similar to Docker while being much more lightweight. For example, balenaEngine is a single binary that is 3.5 times smaller in size than Docker binaries. The lightweightness is achieved by dropping some Docker features that are not useful in embedded use cases [57].

BalenaEngine uses the same architecture and CLI structure as Docker shown in Figure 6. The only difference is that instead of using the command *docker*, *balena-engine* is used instead [59]. For example, the command *docker pull* becomes *balena-engine pull*. This makes balenaEngine a drop-in replacement for Docker, as long as the features omitted from Docker are not needed.

BalenaEngine was included in the evaluated technologies because it is essentially a more lightweight version of Docker aimed for embedded systems. Even though balenaEngine is not as mature technology and less used than Docker, it offers many improvements over Docker for embedded systems, making it a good test subject. Additionally, the performance comparison between balenaEngine and especially Docker is an interesting subject, since balenaEngine has not yet been researched much.

2.3.4 LXC

LXC is one of the oldest Linux container technologies having been under active development since 2008, which makes it well-known and heavily tested. LXC is a low-level and rather lightweight technology consisting of *liblxc* library, a set of tools used to control containers, an API, and a set of templates used to create container images [10]. LXC uses a combination of Linux namespaces, control groups and mandatory access control to create a secure and isolated environment for containers [74].

LXC is mainly focused on system containers. Lots of different LXC system container images exist on a public registry server for many different Linux distributions and CPU architectures. These images can be obtained with the *lxc-create* command using a download template: *lxc-create -t download -n container-name*. However, LXC also supports the OCI image format, which makes running also application containers possible [54].

LXC uses configuration files to define the behaviour of containers. There is a system-wide configuration file as well as a container specific configuration file for each container [55], and there are numerous configuration options that can be specified. It is possible to, for example, to specify what namespaces the container should use [74]. This gives the user great control of the containers on a low-level, but some knowledge and effort is required in order to do so.

LXC was chosen to be evaluated in the thesis because it is one of the oldest Linux container technologies and has already been tested and proven to be useful in embedded systems in other studies. Additionally, it is a lower level technology and does not use a daemon like Docker and balenaEngine. Therefore, LXC should be very lightweight and cause only a small amount of overhead.

2.3.5 Flatpak

Flatpak is a portable package manager with the focus on sandboxing and isolating desktop Linux applications. According to [60], Flatpak is not strictly a container technology as it is mostly used as a convenient library bundling mechanism but it can still be used as a container technology. Flatpak is build upon existing technologies that include e.g. namespaces and control groups.

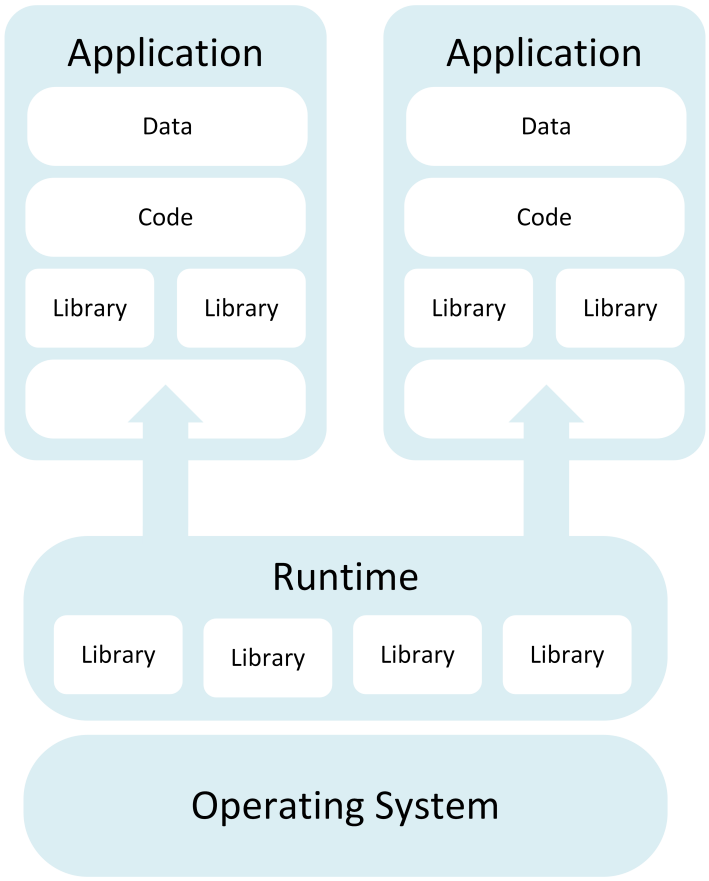

Figure 7: Flatpak architecture. Adapted from [61].

Flatpak architecture can be seen in Figure 7. Flatpak applications run on top of runtimes that are installed on the host OS and provide libraries needed by applications. One runtime can be shared by multiple different applications, which reduces the amount of overhead caused since common libraries needed by multiple applications only need to be included in the single runtime. Additionally, libraries can be bundled

inside applications. This makes it possible for applications to have access to libraries that are not included in the runtime [61].

Flatpak applications and runtimes are typically distributed using repositories in a similar manner to Git repositories. By default, Flatpak applications have very limited access and permissions to the host OS. However, applications can be configured to have access rights to necessary parts of the host environment, such as network or filesystem [62]. The process of creating Flatpak applications is very similar to creating container images with Dockerfiles. However, the information about the application that is being built is stored in a manifest files instead of a Dockerfile. The manifest contains information such as the location of source files, runtime that is being used and e.g. additional Python modules that need to be included [65].

Flatpak was included in the testing because it offers similar functionalities as container engines but with a different approach. It is interesting to compare the performance as well as the ease of use of Flatpak with container engines, since e.g. the method of creating applications for Flatpak is different.

2.3.6 Other technologies

In addition to the four technologies that were selected for testing in the thesis, there are numerous other containerization technologies.

rkt is a application container engine technology that focused on security and is developed for cloud-native environments [9]. It provides a good set of tools for managing the execution of multiple applications simultaneously. Because rkt is developed for cloud environments, it mainly targets the x86_64 CPU architecture and there is a lack of support for other architectures.

LXD is an extension of LXC. Essentially, LXC provides only a basic subset of features that are available in LXD, which is built on top of LXC [69]. LXD provides a daemon and a REST API that make it easier manage the execution of containers and to control the resources available for them. However, LXD only support system containers, whereas LXC supports both application and system containers.

OpenVZ is a containerization technology that is focused on server environments [70]. Its focus is to provide isolation and better resource utilization for operating multiple containers that act like a stand-alone server on a single, physical server.

2.4 Related work

Due to the increased popularity of container technologies, lots of studies have already been devoted to the subject. Many of them have explored the advantages and evaluated the performance of containers. Furthermore, there have been more specific studies around the most used tools in containerization, such as performance analysis of Docker in specific environments [18], [19], Docker security analysis [20] and Kubernetes performance analysis [21].

Performance comparison between containers and virtual machines has been a common topic of research [7], [14], [15], [16], [17]. For example, Felter *et al.* [14] compared the performance between a Kernel-based Virtual Machine (KVM) and Docker containers in a server environment measuring factors such as memory and CPU usage as well as network and I/O bandwidth, and concluded that Docker performance was equal or better than KVM in every test case. Zhang *et al.* [17] performed a similar study by comparing the performance of KVM and Docker in Big Data environment and got results which illustrated that Docker containers achieved higher CPU and memory utilization than KVM. Furthermore, they observed that the usage of Docker was more convenient from a system administrator perspective and the scalability of Docker containers was better than VMs.

There have also been studies that have compared the performance between multiple containerization technologies. Xavier *et al.* [66] evaluated the performance of three containerization technologies and compared them to native performance as well as a VM in a high performance computing environment. Their results showed that all three container technologies had very similar and a near-native performance in terms of CPU, memory, disk and network usage. Morabito [67] investigated the power consumption of Docker, LXC and two VM technologies, comparing them to native power consumption and concluded that the increase in power consumption with all of the tested technologies was almost negligible.

A few studies have focused specifically on embedded systems. Lammi [22] studied the feasibility of application containers in embedded real-time Linux by executing tests with Docker on an embedded Linux device and compared the results to native execution. He concluded that Docker containers could be used in embedded systems, since they introduced only a small amount of overhead. Morabito [68] did a similar research by comparing the performance of Docker containers with native execution on a Raspberry Pi 2, with results showing that the overhead added by Docker containers is almost negligible. Noronha *et al.* [1] evaluated the performance of LXC containers with four different embedded devices. They concluded that with all four devices, the amount of overhead caused by containers was very small and almost negligible compared to native performance.

3 Methodology

Because the research goal is give an recommendation for an application container technology to be used in an embedded device, multiple different container technologies are evaluated using a Raspberry Pi 4 as a platform. The technologies included in the evaluation were chosen based on the preliminary research concluded in the Container engines section. The evaluation is done by first creating a baseline Linux system where the tests and measurements are executed natively providing baseline results. Then, a single container engine at a time is added to the base Linux system and the results are compared to the native execution with the baseline Linux. This approach ensures that measurement results will be comparable, since the only additions to the baseline Linux distribution will be the container engines and container images.

3.1 Measurements

Multiple measurements will be performed with the purpose of evaluating the overhead caused by containerization. The overhead is evaluated by measuring memory, CPU and mass storage usage as well as power consumption. These factors were chosen because they are usually the most important resources in embedded systems, since often there is a small amount of available memory and mass storage, and the CPU has limited processing power. Furthermore, there are many embedded systems that need to be able to operate for long periods of time with limited amount of power. For example, some devices only have a battery or a solar panel as their source of power, making the additional power consumption caused by containers an important consideration.

Memory usage and CPU time measurements can be implemented solely using software available within Linux. Mass storage usage can be measured simply by comparing the size of Linux images that include container engines to the size of the base image. However, power consumption measurements can not be implemented only with software, thus some additional hardware is required.

3.2 Test program design

In order to get meaningful results from the tests and measurements executed and to verify that the evaluated application containers are a feasible solution in real use cases, the tests executed need to be able to perform some common functionalities that require permissions or access rights from the host OS. These functionalities include, for example, serial and D-Bus communication as well as network usage. They need to be tested because a container engine could seem like a good solution when considering only overhead, but if e.g. serial communication is not possible within a container, it would be a large drawback. Furthermore, testing these functionalities gives insight on how difficult it is to configure access rights for each container engine. Additionally, this approach makes the testing I/O intensive which reflects many real use cases for embedded devices.

Three separate programs will be created, each of which will be run in a different

container. Two of the programs will communicate between each other using TCP sockets because it should be ensured that communication between containers is possible. Additionally, running several containers at the same time will give insight on whether the overhead stays small even when scaling the number of containers up. However, there is no need scale the number of containers up to higher than three, since usually there is no need for such use cases in embedded systems. The tests will be executed alongside with the measurements for multiple iterations in order to see if there are any notable differences with the results between individual iterations.

3.3 Other factors

The container engine solution should be open source and secure as well as easy to develop and deploy. Therefore, these factors are also considered when evaluating different containerization options.

Security is a key aspect in containerization because application containers need to be isolated and their access should be limited to the underlying OS and hardware. It is important to ensure that e.g. privilege escalations can not happen within containers. However, exhaustive security testing of containers is difficult and out of scope of the thesis. Therefore, conclusions about the security of each container engine will be drawn from existing literature.

Ease in developing and deploying of each solution are evaluated during the testing process carried out during this thesis. Factors considered are, for example, the availability of Yocto recipes, since existing and maintained recipes make the development process easier. Furthermore, the process of creating and distributing container images is evaluated when considering the ease in deploying.

4 Implementation

This chapter discusses the implementation of the testing setup. It explains how the custom Linux images were built as well as how the measurements and test programs were implemented.

4.1 Test system

Raspberry Pi was chosen in this thesis as the platform used to execute tests because its components are similar to many other embedded devices. The specific model used is Raspberry Pi 4 model B (fig. 8) with 1 GB of random-access memory (RAM). Its CPU has ARMv8 architecture, and one of its features is full backward compatibility with ARMv7 architecture, which is used by most operating systems for the RPI. Additionally, a RS485 Pi board (fig. 9) was used, which enabled the usage of serial communication protocol RS-485 with the Raspberry Pi.

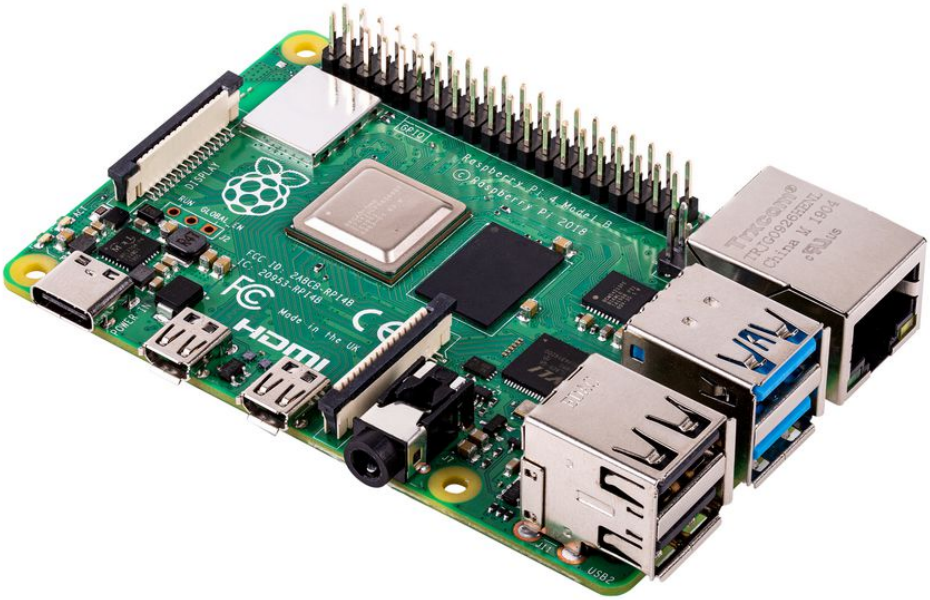

Figure 8: Raspberry Pi 4.

A custom Linux distribution is used and it is cross compiled to the ARMv7 CPU architecture using the Yocto Project. The Yocto Project is an open source collaboration project that provides and maintains a set of tools that can be used to create highly customisable Linux systems aimed for embedded products. Yocto builds everything that is needed for the system, such as kernel and root filesystem, and everything can be customised exactly as needed. Programs are described with *recipe* files which contain instructions on how the program should be built. For example, they contain the location of the program source code, which can be e.g. a git repository, and possible additional instructions or configurations needed. Recipe files are located in repositories called *layers*. Layers contain all recipes and configurations

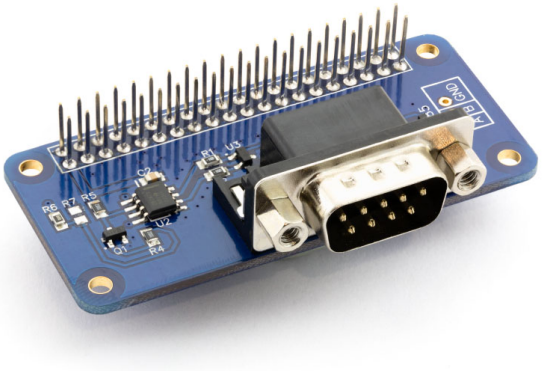

Figure 9: RS485 extension board for Raspberry Pi [75].

for a single logical entity, such as particular piece of hardware. For example, all Raspberry Pi specific configurations are located in *meta-raspberrypi* layer [49].

4.2 Building Linux images

Yocto version *Zeus* was used to build the custom Linux distributions used in the thesis. First, a base image was built using Yocto's *core-image-base* recipe and Raspberry Pi 4 machine configuration recipes that were already available within the Yocto Project as a starting point. The resulting image was a very basic headless Linux distribution that supported the features required for the testing. Kernel version 4.19.93 was used.

The next step was to add container engines to the Linux distribution. A separate image was built for each tested container engine. This process was simple because Yocto allows appending existing recipes, making it easy to create a new image recipe for each container engine where the only addition to the base image was the recipe for the engine itself and the programs used when executing the tests and measurements.

Creating images for each container engine was, for the most part, straightforward. There were existing recipes located in *meta-virtualization* layer for both Docker and LXC. Additionally, to get LXC working, a few Linux utilities needed to be added to the image. In the case of Flatpak, there was an existing recipe for it, but it was a few years old and no longer maintained. It could, however, be used a starting point for creating a custom recipe that was created. For balenaEngine no standalone recipe existed. However, a pre built balenaEngine binaries for various CPU architectures were provided by balena [4]. Therefore, the binary for ARMv7 could simply be added to the Linux image. Version numbers for each technology used were the following: Docker version 19.03.2, balenaEngine version 17.12.0, LXC version 3.2.1, and Flatpak version 1.6.0.

4.3 Test program

Three simple but I/O intensive programs were written in Python. Python was chosen as the programming language due to its ease of use and popularity. The

first program, shown in Program 7, uses serial communication to poll data from a separate Windows PC and writes the responses to a log file. The program runs for five minutes and it uses D-Bus to get the current time which is used to determine when the five minutes has passed.

```
1  import serial
2  import dbus
3
4  pollString=b'SENDDATA\n'
5  port='/dev/serial0'
6  serialPort = serial.Serial(port, 9600, timeout=10)
7
8  bus = dbus.SystemBus()
9  time = bus.get_object('com.rpi.program', '/Time')
10 time_start = time.CurrentTime()
11
12 # run for 5 minutes
13 while ((time_start + 5 * 60) > time.CurrentTime()):
14     f = open("vol/logfile.txt", "a")
15     serialPort.write(pollString)
16     response = serialPort.readline()
17     res = response.decode()
18     f.write(res)
19     f.close()
```

Program 7: Serial communication and D-Bus usage.

Two other programs were used to communicate between separate containers with TCP. The first of them, shown in Program 8 functions as a TCP server, and the other one, shown in Program 9 is a TCP client. The server simply sends random integers as a response to the client every time it connects to the server. After the client receives the response, it sends it to an database that is hosted on a separate Windows PC that is on the same network as the RPI. The server is executed for 5 minutes and the client for 4 minutes. This is because the server should always be on when the client is executing. Therefore, the server must be started first and still run longer than the client which is the reason for it being executed for a longer period of time.

```
1  import socket
2  from random import randint
3  import time
4
5  host = ''
6  port = 9999
7  s = socket.socket()
8  s.settimeout(10)
9  s.bind((host,port))
10 s.listen()
```

```
11
12 time_start = time.time()
13 while ((time_start + 5 * 60) > time.time()):
14     try:
15         conn,addr = s.accept()
16         r = str(randint(0,1000))
17         conn.send(r.encode())
18         conn.close()
19     except socket.timeout:
20         break
21 s.close()
```

Program 8: TCP server.

```
1  import socket
2  import requests
3  import time
4
5  host = '172.18.0.15'
6  port = 9999
7  db_url = http://192.168.1.185:8086/write?db=rpidb
8  time_start = time.time()
9
10 while ((time_start + 4 * 60) > time.time()):
11     s = socket.socket()
12     s.connect((host,port))
13     response = s.recv(1024)
14     res = response.decode()
15     requests.post(db_url, "data value=" + res)
16     s.close()
```

Program 9: TCP client.

4.4 Measurements

A shell script shown in Program 10 was used to log the CPU and memory usage. The script needed one parameter as an argument which was the name of the container engine that was being tested. This was necessary in order to be able to use the same script with all four container engines. A program called *mpstat* (fig. 10) was used to measure the CPU usage and *free* (fig. 11) to measure the memory usage. *Mpstat* can be used to report the system's CPU usage for a defined period of time with a defined interval while also reporting the average of all values after the time period has elapsed. It provides various CPU statistics as shown in Figure 10, and to get the CPU usage, the idle value from last column was subtracted from 100. As can be seen on line 7 of the program, *mpstat* was called as a daemon process for a period of three minutes with and interval of one second. Then, only the average values were

saved to a log file using *grep*. *Free* was used in a similar manner which can be seen on line 10, but as it had no built-in option to take samples for a period of time, it had to be called every second inside a loop that lasted three minutes. Additionally, because *free* reports multiple values, such as total and shared memory, and only used memory was of interest, it was the only value that was logged using program *awk*. Furthermore, the average value of the logged memory had to be calculated manually, so a simple python script was used for this purpose as can be seen on the last line of the script. This python script then saved the average value to a separate log file allowing the memory log file used in the measurements script to be cleared and used again which can be seen on line 6 of the script.

```
1  #!/bin/sh
2
3  MEM_LOG_FILE="log-memory"
4  CPU_LOG_FILE="log-cpu-"
5
6  rm -f ${MEM_LOG_FILE}
7  mpstat 1 180 | grep Average >> ${CPU_LOG_FILE}$1 &
8  for i in `seq 180`
9  do
10     free | grep Mem | awk '{print $3}' >> ${MEM_LOG_FILE}
11     sleep 1
12 done
13
14 python3 calc-avg.py $1 $MEM_LOG_FILE
```

Program 10: A shell script logging CPU and memory usage.

```
root@raspberrypi4:~# mpstat
Linux 4.19.93 (raspberrypi4)    06/14/20        _armv7l_        (4 CPU)

15:46:10     CPU    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest  %gnice   %idle
15:46:10     all    0.16    0.00    0.24    0.05    0.00    0.00    0.00    0.00    0.00   99.54
```

Figure 10: Sample *mpstat* output. The last column value was subtracted from 100 to get the CPU usage percentage.

```
root@raspberrypi4:~# free
              total        used        free      shared  buff/cache   available
Mem:         944588       48908      819728       17008       75952      847964
Swap:             0           0           0
```

Figure 11: Sample *free* output. Third column shows the used memory.

Power consumption was measured using a LabJack T7 data acquisition device [50] which was connected to a circuit with the RPI and a shunt resistor with a known resistance. In order to know the power, voltage and current of the system needed to be measured. The voltage could be measured simply by measuring the voltage

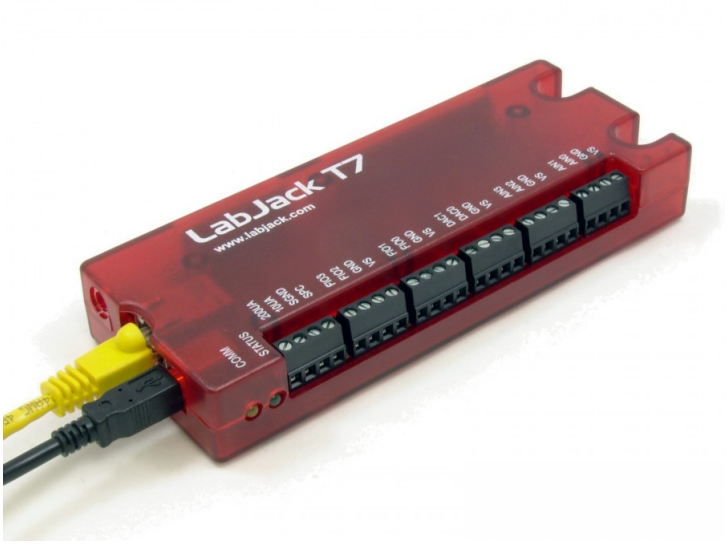

Figure 12: Labjack T7 [50].

over the RPI. The current measurement was done by first measuring the voltage drop over the shunt resistor. Then, the current could be calculated by dividing the voltage drop with the shunt resistor's resistance. The power consumption could then be easily calculated by multiplying voltage and current.

The voltage measurements were done by programming the Labjack T7 with Python using the LJM library [64] provided by Labjack. This was done from an Linux VM running inside an Windows PC. Because memory and CPU measurements script as well as the Python test programs had to be run from the RPI, the commands to execute them were send over SSH from the Linux VM and they were synchronized with the power consumption measurements to have them executing at the same time.

4.5 Container images

A Dockerfile was created for each of the three test programs. All of them were based on a minimal Linux distribution Alpine that contained Python by using instruction *FROM python:3.7.7-alpine3.10*. Then, one of the test Python scripts and the Python modules that it required were added to the Dockerfile.

Docker Buildx was used for cross compiling the Dockerfiles into container images that could be used by Docker, balenaEngine and LXC. Buildx is a plugin for the Docker CLI that can build container images for multiple architectures and supports a variety of building options. For example, building a container image for the ARMv7 using Docker image specification as output format can be achieved with the following flags: *--platform linux/arm/v7 --output=type=docker*. These flags were used to build the container images for both Docker and balenaEngine, since balenaEngine can execute images that are in Docker image format. Building images for LXC was done in a similar manner, as the only change needed was to use *--output=type=oci* as the output format.

Flatpak could not execute the same images, thus its own building mechanism had

to be used. A manifest was created for each application and a runtime containing Python was used. Flatpak's build system flatpak-builder was used to build the applications. It provides the possibility to build applications for various CPU architectures with a flag *arch*. When building applications for the RPI, *--arch=arm* was used. However, flatpak-builder expects that host system is compatible with the specified architecture which was not the case, since flatpak-builder was used from a Linux VM with x86_64 architecture. Fortunately, QEMU could be used to solve this issue by installing *qemu-user-static* package which provides emulated binaries for multiple CPU architectures.

5 Results and Evaluation

The tests and measurements were executed for ten iterations with the baseline Linux system and each container technology. Acquired results are presented and discussed in this section. A bar graph was created for memory and CPU usage as well as power consumption, displaying the minimum, average and maximum measurement values from the ten iterations for each technology. Additionally, plots displaying the results from each iteration are presented, where the x-axis represents the number of iteration and the y-axis shows the measured factor. This section also shows the storage consumption of each technology. Furthermore, tables displaying the mean value as well as the absolute and relative increase to the baseline value for each measured factor results are presented. Finally, the security and usability of the technologies are discussed.

5.1 Memory usage

Figure 13 and Table 1 show that there are large differences with memory usage between the evaluated technologies. It can be seen that LXC has nearly native performance, while the others have significant amounts of overhead. Flatpak has approximately 50% increase in memory usage over the baseline value. However, it can be seen from Figure 14 that Flatpak's memory usage grows linearly as the number of iterations increase. It is unclear whether this is caused by Flatpak itself or something else. If this issue did not exist and the memory usage of all iterations remained similar to the first iteration, the increase over baseline would be significantly lower.

Docker and balenaEngine have the largest increases memory usage, as it can be seen that balenaEngine's value is over two times and Docker's over three times the baseline counterpart. A significant portion of the overhead is explained by the daemon processes that both technologies use. Additionally, both of them are higher level technologies with more functionalities than LXC and Flatpak, which also has an impact on their memory usage. Figure 14 additionally shows that the memory usage of the first iteration is clearly lower than the rest of the iterations with Docker as well as with balenaEngine, albeit in a smaller scale. This suggests that the first initialization of containers slightly increases the system's memory usage even after the execution of containers has finished.

	Baseline	LXC	Flatpak	Balena	Docker
Mean (KiB)	77577.3	82869.9	115947.3	160004.0	252175.7
Absolute increase (KiB)	-	5292.6	38369.9	82426.6	174598.3
Relative increase (%)	-	6.8	49.5	106.3	225.1

Table 1: Mean memory usage values and the overhead with respect to the baseline value.

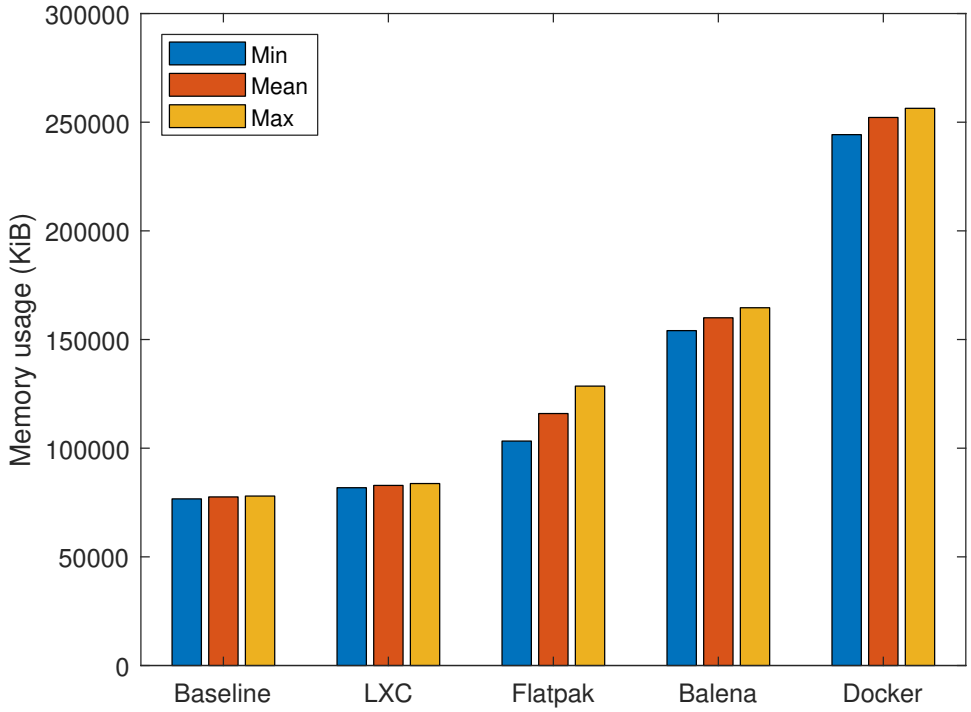

Figure 13: Minimum, mean and maximum memory usage values over ten iterations.

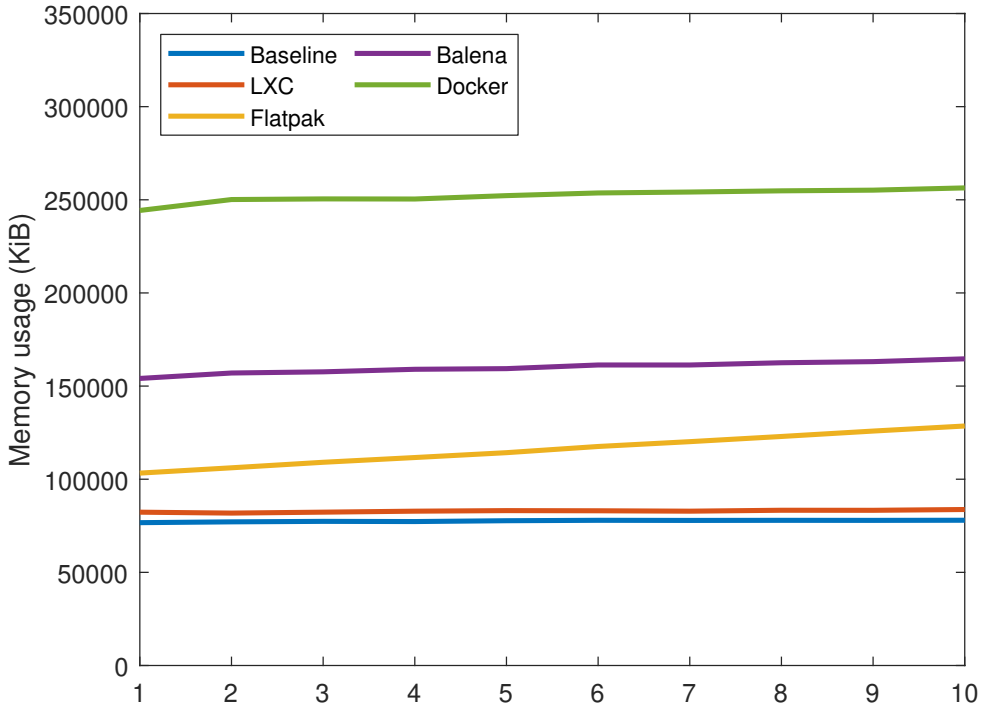

Figure 14: Individual memory usage values of each iteration.

5.2 CPU usage

Figure 15 shows that there are no big differences between evaluated technologies in terms of CPU usage. However, it can be seen from Table 2 that LXC and Flatpak have slightly more overhead than balenaEngine and Docker which is on the contrary to the overhead observed in terms of memory usage. Nonetheless, the CPU usage increase over the baseline is relatively small and can be considered negligible with all four technologies.

	Baseline	LXC	Flatpak	Balena	Docker
Mean (%)	20.8	22.1	22.2	21.6	21.4
Absolute increase (%)	-	1.3	1.4	0.7	0.6
Relative increase (%)	-	6.3	6.7	3.5	2.7

Table 2: Mean CPU usage values and the overhead with respect to the baseline value.

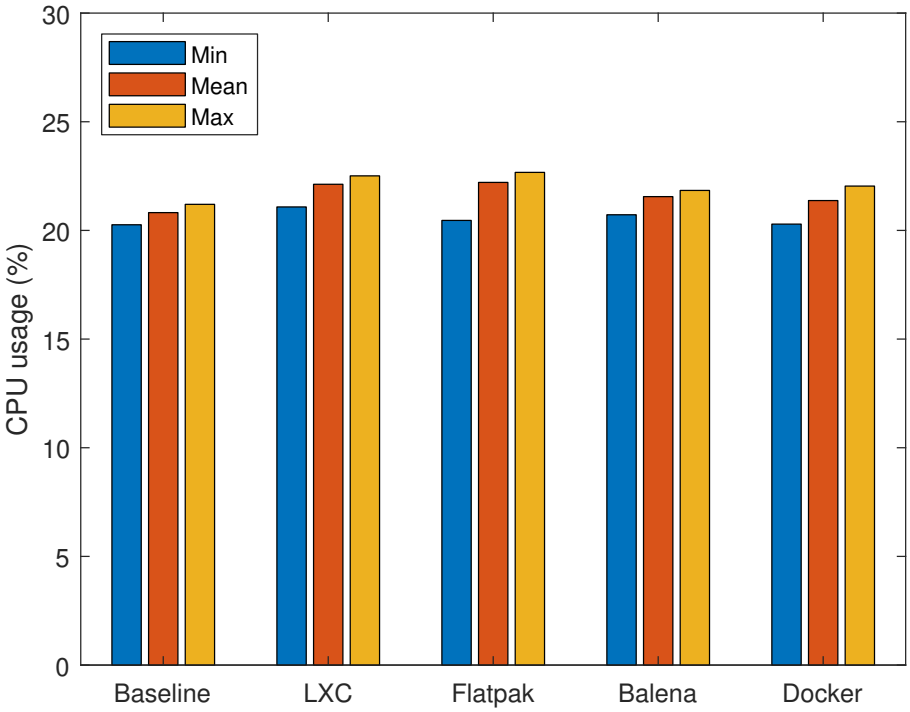

Figure 15: Minimum, mean and maximum CPU usage values over ten iterations.

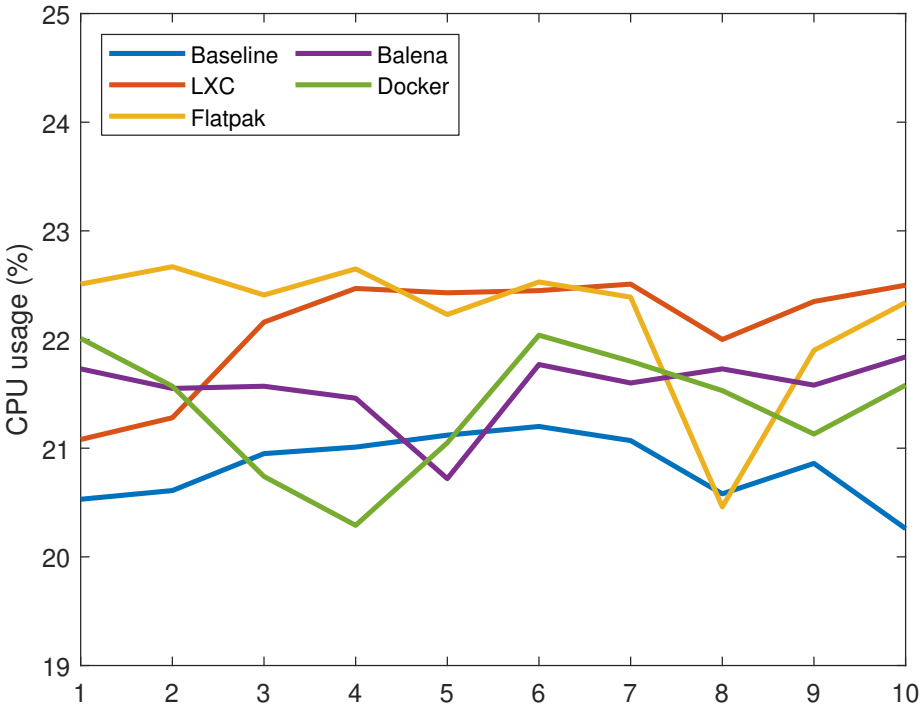

Figure 16: Individual CPU usage values of each iteration.

5.3 Power consumption

It seems that all technologies have almost native performance in terms of power consumption if looking at Figure 17 or the relative increase in Table 3. However, when looking at the absolute increase, it can be seen that LXC has 77.4 mW and Flatpak 34.6 mW higher power consumption than the baseline. These values can be significant in some embedded systems where the power is a critical resource. The higher values of LXC and Flatpak can be partly explained by their higher CPU usage when compared to balenaEngine and Docker.

	Baseline	LXC	Flatpak	Balena	Docker
Mean (mW)	2910.0	2987.4	2944.7	2915.3	2922.5
Absolute increase (mW)	-	77.4	34.7	5.3	12.5
Relative increase (%)	-	2.7	1.2	0.2	0.4

Table 3: Mean power consumption values and the overhead with respect to the baseline value.

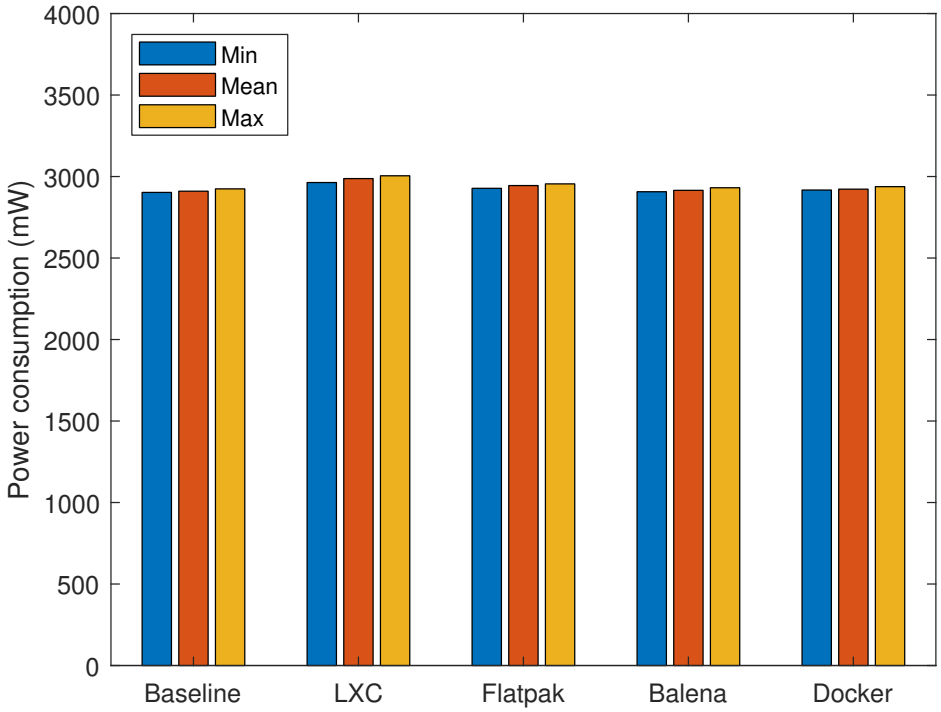

Figure 17: Minimum, mean and maximum power consumption values over ten iterations.

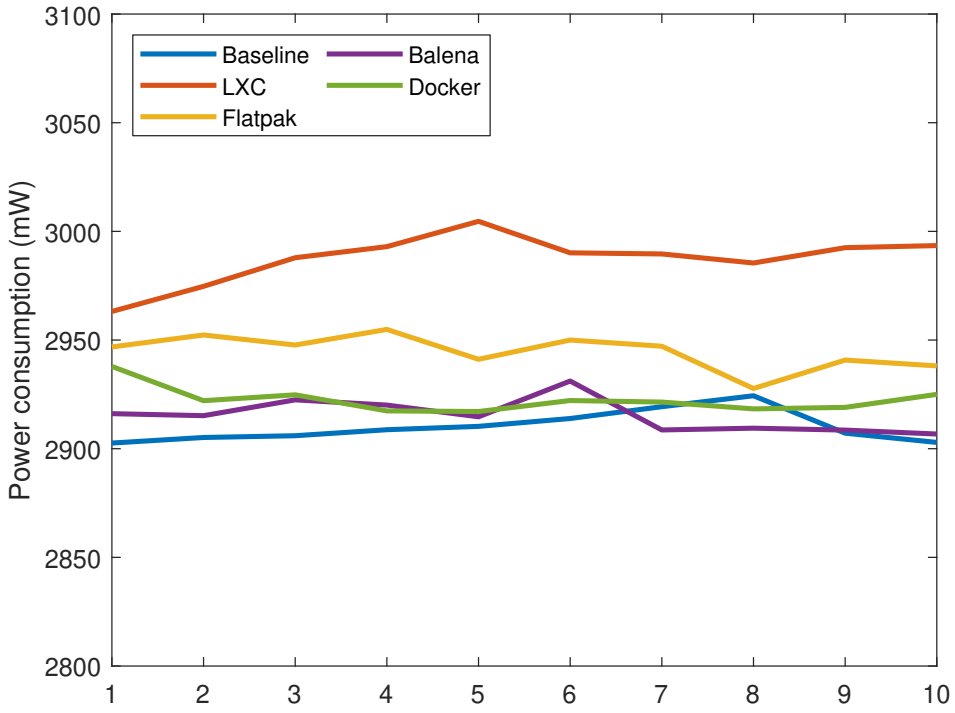

Figure 18: Individual power consumption values of each iteration.

5.4 Mass storage usage

Figure 19 shows that Docker introduces a significant amount of overhead to the system in terms of mass storage usage. The rest of the technologies have rather small amounts of overhead, but there are still slight differences between them. It can be seen from Table 3 that Flatpak increased the baseline image size only by 8.4 MB which can be considered to be a negligible amount. On the other hand, balenaEngine increased the image size by 29.4 MB and LXC by 42.0 MB. These amounts might be significant in some embedded systems but are still generally low.

	Baseline	LXC	Flatpak	Balena	Docker
Value (MB)	276.8	318.8	285.2	306.2	461.4
Absolute increase (MB)	-	42.0	8.4	29.4	184.6
Relative increase (%)	-	15.2	3.0	10.6	66.7

Table 4: Mass storage usage values and the overhead with respect to the baseline value.

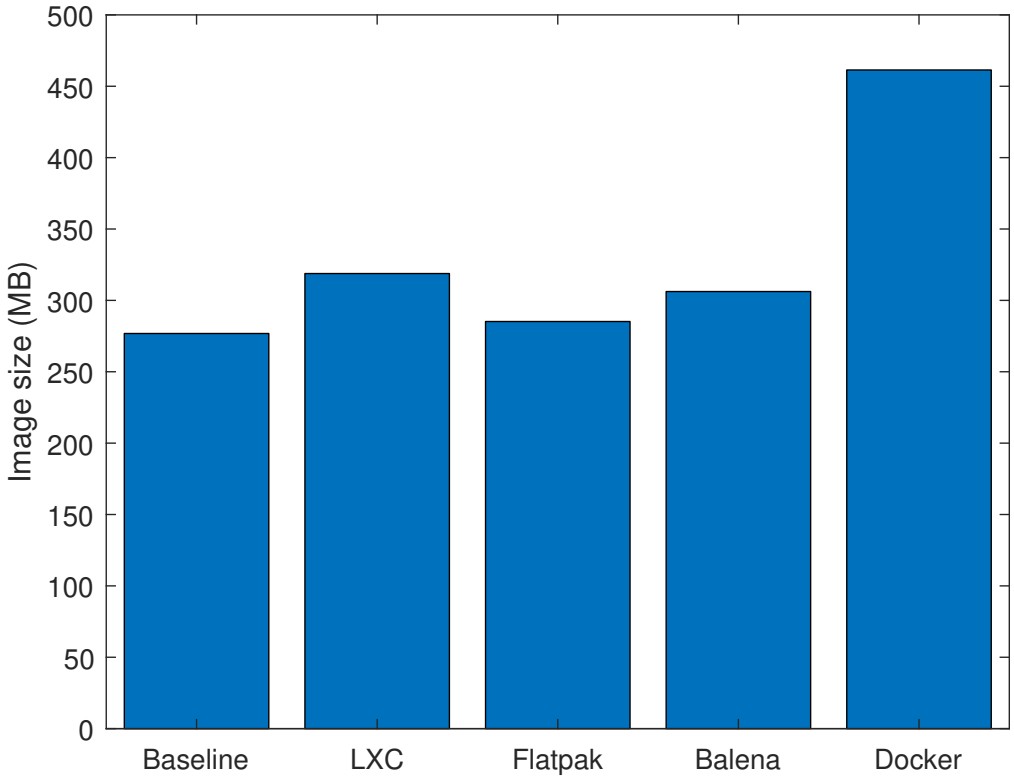

Figure 19: Mass storage usage.

5.5 Security

All evaluated technologies use Linux namespaces and control groups which are the basis for the security of containers. Namespaces provide the isolation of containers while control groups make it possible to limit and monitor their resource usage. Namespaces are used to separate containerized applications from the host OS, making them unable to see or affect other processes that are running in the system. Furthermore, namespaces can be used to isolate other system resources, such as network interfaces and sockets. Control groups, on the other hand, are used to limit e.g. memory and CPU usage of containers, which prevents various denial-of-service attacks. In addition to namespaces and cgroups, other security measures can be used to further enhance the security of containers.

Docker and balenaEngine share many features, since balenaEngine is built on the framework provided by Docker. Both of them use a daemon process that requires root privileges which makes it prone to attacks because e.g. privilege escalations can happen if unwanted parties can access the daemon process. Therefore, it should not be accessible by untrusted users. Furthermore, if users can start containers and thus send commands to the daemon through an interface, the parameter checking should be implemented carefully in order to not allow malicious input to the daemon [76].

There are also a few extra security features that can be used with both Docker and balenaEngine. It is possible to configure the capabilities of containers with both of them. Capabilities are a way of giving processes access to specific tasks on the system. Docker and balenaEngine containers start with a limited set of capabilities by default, and it is possible add or remove capabilities for specific containers. If a container is only given the capabilities that are necessary for it to function, then even if a privilege escalation were to happen inside a container, it would be harder to cause damage to the system. Additionally, Docker and balenaEngine can be configured to only run container images that have a trusted signature. This makes it possible for system administrators to have control over the container images that can be run on a system. Furthermore, it is possible to run other security enhancing systems such as AppArmor or SELinux alongside with the containers [76].

With LXC and Flatpak, there are no attack-prone daemon processes. Unprivileged LXC containers, where the container user id 0 is mapped to an unprivileged user outside of the container, are safe by design. The container is only able to access resources that the user has access to [77]. With Flatpak, all applications have very limited access rights by default. For example, they can not access network or most host files and these access rights have to be configured manually for the applications. Additionally, all processes are run as an user with no capabilities, which further improves the security [78].

Overall, all four technologies seem rather secure. The daemon processes in Docker and balenaEngine introduce an additional attack surface that is not present with LXC and Flatpak, but if the daemon is not accessible by unwanted users, it should not be a security risk.

5.6 Usability

Since Docker and balenaEngine are relatively high level technologies, they have plenty of functions and a CLI that makes the configuration and management of containers simple. Docker was also easy to cross compile to an ARM CPU due to an existing Yocto recipe, and for balenaEngine, a pre built ARM binary was available on the balenaEngine website. Creating container images for ARM was straightforward with the Docker Buildx tool and the same docker image could be used with both Docker and balenaEngine. Additionally, Docker has a comprehensive documentation which makes it easier to use. For example, all possible options that can be included in Dockerfiles are documented in depth. All these factors considered, it can be stated that both Docker and balenaEngine were easy to use and there were no issues identified regarding development or deployment.

Similarly to Docker, there was an Yocto recipe for LXC which made the cross compilation process easy. Additionally, same container images and their build process could be used with only the difference that the images were built as OCI format, which LXC supports. LXC has a CLI that has the basic functionalities and there were no issues in the management of containers. However, configuring access rights for them was harder than with Docker or balenaEngine, since they needed to be specified in configuration files rather than giving them as a parameters to the CLI. Furthermore, the documentation regarding this process was not very comprehensive, which was also the case with LXC's overall documentation. Altogether, there were no major issues with the usability of LXC, but the complexity in configuration of containers and the lack of documentation means that it was not as easy to use as Docker or balenaEngine.

Using Flatpak was not as straightforward as the other three evaluated technologies. There was no up-to-date Yocto recipe for it, thus some work was required to cross compile Flatpak for the RPI. Additionally, same container images could not be used but the process of creating separate Flatpak applications was straightforward, although there were some obstacles in cross compiling them. The Flatpak CLI was good and made the management and configuration of the applications easy. Additionally, the documentation of Flatpak was comprehensive. However, there were some limitations with the access rights management. For example, it was not possible to give an application access to a singular device, since the only option was to give access to all or none devices with Flatpak. This was not an issue with the other technologies, as they had more fine-grained access control options. All in all, it can be stated that Flatpak had some issues in terms of usability, but the issues were minor and Flatpak could still be used with the RPI.

6 Conclusion

The goal of this thesis was to give an recommendation for a container technology that could be used in an embedded Linux device. To reach this goal, different options were investigated and four technologies were chosen for an in depth evaluation. Various tests were executed with each of them and the overhead caused by containers was measured. The chosen technologies were Docker, balenaEngine, LXC and Flatpak.

The testing consisted of three small Python programs that reflected common embedded systems software use cases and behaviour, and a script that measured the usage of various system resources. The factors that were chosen to be measured were memory, CPU and mass storage usage as well as power consumption, since they are the most critical resources for embedded systems. The test programs and the measurements script were first run on a baseline Linux distribution that did not include any of the evaluated technologies. Then, same measurements were executed while the Python test programs were executed inside containers with each technology.

The acquired results showed that there were some differences between the four technologies. In terms of memory usage, LXC performed well having almost a native performance, while the other technologies had significant increases in their memory usage, with Docker having distinctly the worst performance. Regarding CPU usage, all technologies had similar performance and there was no significant overhead with any of them. Docker and balenaEngine performed the best with power consumption, as the overhead introduced by them was negligible. Flatpak had a moderate increase in power consumption. LXC introduced a significant amount of overhead and had the worst performance of the four technologies in terms of power consumption. Regarding mass storage usage, Docker had significant amount of overhead while the other technologies increased the mass storage usage only by small amounts. Overall, it can be stated that Docker had the most overhead across all tested resources, and the while there were some differences between LXC, Flatpak and balenaEngine, their overhead was, on average, relatively low.

In addition to the performance of the container technologies, their security and usability was also evaluated. It was found out that while all of them seem rather secure, the daemon processes of Docker and balenaEngine bring an additional surface to the system that is not present with LXC and Flatpak. In terms of usability, it was concluded that Docker and balenaEngine were the easiest to use. With LXC, the configuration of containers was slightly harder, and with Flatpak there were some minor issues with access rights management. Nonetheless, both of them could still be used with relative ease.

In conclusion, all of the evaluated technologies worked with the Raspberry Pi. Docker seems to be the worst option of them, as it had the most amount of overhead. LXC, Flatpak and balenaEngine had some differences between them, but overall they had relatively low overhead and all of them seem to be feasible solutions as a container technology for embedded systems. As these three container solutions are close to each other in terms of performance, no real recommendation is given about which one should be used in an embedded device. Further analysis on developer experience and practical deployment issues should be done in order to determine the

best technology to use. These are out of scope for the thesis.

This study could also be continued by carrying out the same tests with multiple different embedded devices to see if the hardware of the system has any impact on the test results. Additionally, different test programs could be used to study if e.g. less I/O intensive programs would produce different results. Finally, more container technologies could be tested and more factors, such as network or IPC latencies, could be taken into account when considering the overhead of the containers.

References

[1] V. Noronha, M. Riegel, E. Lang, and T. Bauschert, "Performance Evaluation of Container based Virtualization on Embedded Microprocessors," in *2018 30th Int. Teletraffic Congr.*, Vienna, Austria, Sep. 2018, vol. 1, pp. 79-84, doi: 10.1109/ITC30.2018.00019.

[2] S. McCarty. "A Practical Introduction to Container Terminology." developers.redhat.com. https://developers.redhat.com/blog/2018/02/22/container-terminology-practical-introduction/ (accessed Jan. 23, 2020).

[3] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov, "Microservices: The Journey So Far and Challenges Ahead." *IEEE Software*, vol. 35, no. 3, pp. 24-35, May 2018, doi: 10.1109/MS.2018.2141039.

[4] balena. "A container engine built for IoT." balena.io. https://www.balena.io/engine (accessed Jan. 23, 2020).

[5] P. Sareen, "Cloud Computing: Types, Architecture, Applications, Concerns, Virtualization and Role of IT Governance in Cloud," in *Int. J. Adv. Res. Comput. Sci. Softw. Eng.*, vol. 3, no. 3, pp. 533-538, Mar. 2013.

[6] R. Buuya, C. Vecchiola, and S. T. Selvi, "Virtualization," in *Mastering Cloud Computing*, Waltham, MA, USA: Morgan Kaufmann, 2013.

[7] B. Bardhi, A. Claudi, L. Spalazzi, G. Taccari, and L. Taccari, "Virtualization on embedded boards as enabling technology for the Cloud of Things," in *Internet of Things,* R. Buyya and A. V. Dastjerdi, Eds., Cambridge, MA, USA: Morgan Kaufmann, 2016.

[8] Docker. docker.com. https://www.docker.com/ (accessed Jan. 29, 2020).

[9] rkt. "A security-minded, standards-based container engine." coreos.com. https://coreos.com/rkt/ (accessed Jan. 29, 2020).

[10] Linux containers. "What's LXC?" linuxcontainers.org. https://linuxcontainers.org/lxc/introduction/ (accessed Jan. 29, 2020).

[11] runc Github. github.com. https://github.com/opencontainers/runc (accessed Jan. 29.2020).

[12] Kata Containers. katacontainers.io. https://katacontainers.io/ (accessed Jan. 29, 2020).

[13] Kubernetes. "Production-Grade Container Orchestration." kubernetes.io. https://kubernetes.io/ (accessed Jan. 29, 2020).

[14] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. "An Updated Performance Comparison of Virtual Machines and Linux Containers." IBM. Austin, Tx, USA. Res. Rep. RC25482 (AUS1407-001), Jul. 2014.

[15] P. R. Desai. "A Survey of Performance Comparison between Virtual Machines and Containers," in *Int. J. Comput. Sci. Eng*, vol. 4, no. 7, pp. 55-59. Jul. 2016.

[16] A. M. Joy. "Performance comparison between Linux containers and virtual machines," in *2015 Int. Conf. Adv. Comp. Eng. Appl.* Ghaziabad, India. Mar. 2015. pp. 342-346.

[17] Q. Zhang, L. Liu, C. Pu, Q. Dou, L. Wu, and W. Zhou. "A Comparative Study of Containers and Virtual Machines in Big Data Environment," in *2018 IEEE 11th Int. Conf. Cloud Comput.*, San Francisco, CA, USA, Jul. 2018, pp. 178-185, doi: 10.1109/CLOUD.2018.00030.

[18] P. D. Tommaso, E. Palumdo, M. Chatzou, P. Prieto, M. L. Heuer, and C. Notredame. The impact of Docker containers on the performance of genomic pipelines. *PeerJ*. Sep. 2015.

[19] M. T. Chung, N. Quang-Hung, M-T. Nguyen, and N. Thoai. "Using Docker in high performance computing applications," in *2016 IEEE Sixth Int. Conf. Commun. Electron.* Ha Long, Vietnam. Jul. 2016. pp. 52-57. doi: 10.1109/CCE.2016.7562612.

[20] T. Bui. "Analysis of Docker Security," Aalto Univ., Espoo, Finland, 2014.

[21] V. Medel, O. Rana, J. Á. Bañares, and U. Arronategui. "Modelling Performance & Resource Management in Kubernetes," in *Proc. 9th. Int. Conf.* Utility Cloud Comp. Dec. 2916. pp. 257-262.

[22] Lammi, T. "Feasibility of application containers in embedded real-time Linux," M.S thesis, Dept. Elect. Eng., Tampere Univ. Technol., Tampere, Finland, 2018.

[23] M. J. Scheepers. "Virtualization and Containerization of Application Infrastructure: A Comparison," in *21st Twente Student Conf. IT*. Enschede, The Netherlands. Jun. 2014.

[24] Open Container Initiative. "FAQ." opencontainers.org https://www.opencontainers.org/faq (accessed Jan. 31, 2020).

[25] Open Container Initiative github. "Image Format Specification." github.com. https://github.com/opencontainers/image-spec/blob/master/spec.md (accessed Feb. 6, 2020).

[26] Open Container Initiative github. "OCI Image Manifest Specification." https://github.com/opencontainers/image-spec/blob/master/manifest.md (accessed Feb. 6, 2020).

[27] Open Container Initiative github. "OCI Image Configuration." https://github.com/opencontainers/image-spec/blob/master/config.md (accessed Feb. 6, 2020).

[28] VMware. "Bare metal hypervisor." vmware.com https://www.vmware.com/topics/glossary/content/bare-metal-hypervisor (accessed Feb. 19, 2020).

[29] cgroups man page. "Linux control groups." man7.org. http://man7.org/linux/man-pages/man7/cgroups.7.html (accessed Feb. 20, 2020).

[30] Kernel documentation. "cgroups v1." kernel.org. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt (accessed Feb. 20, 2020).

[31] Kernel documentation. "cgroups v2." kernel.org. https://www.kernel.org/doc/Documentation/cgroup-v2.txt (accessed Feb. 20, 2020).

[32] namespaces man page. "Linux namespaces." man7.org. http://man7.org/linux/man-pages/man7/namespaces.7.html (accessed Feb. 24, 2020).

[33] W. Mauerer. *Professional Linux® Kernel Architecture*. Indianapolis, IN, USA: Wiley Publishing, Inc., 2008.

[34] S. Grunert. "Demystifying Containers - Part I: Kernel Space." Medium.com. https://medium.com/@saschagrunert/demystifying-containers-part-i-kernel-space-2c53d6979504 (accessed Feb. 25, 2020).

[35] Linux source code. "Namespace management." github.com. https://github.com/torvalds/linux/blob/master/include/linux/nsproxy.h (accessed Feb. 25, 2020).

[36] PID namespaces man page. "PID namespaces." man7.org. http://man7.org/linux/man-pages/man7/pid_namespaces.7.html (accessed Feb. 26, 2020).

[37] User namespaces man page. "User namespaces." man7.org. http://man7.org/linux/man-pages/man7/user_namespaces.7.html (accessed Feb. 26, 2020).

[38] Yocto project. yoctoproject.org. https://www.yoctoproject.org/ (accessed Feb. 26, 2020).

[39] UTS namespaces man page. "UTS namespaces." man7.org. http://man7.org/linux/man-pages/man7/uts_namespaces.7.html (accessed Feb. 26, 2020).

[40] M. Kerrisk. "Namespaces in operation, part 1: namespaces overview." LWN.net. https://lwn.net/Articles/531114/ (accessed Feb. 26, 2020).

[41] IPC namespaces man page. "IPC namespaces." man7.org. http://man7.org/linux/man-pages/man7/ipc_namespaces.7.html (accessed Feb. 26, 2020).

[42] A. Javed, "Linux Containers: An Emerging Cloud Technology," Aalto Univ., Espoo, Finland, 2015.

[43] Mount namespaces man page. "Mount namespaces." man7.org. http://man7.org/linux/man-pages/man7/mount_namespaces.7.html (accessed Mar. 4, 2020).

[44] J. Jensen. "Linux unshare -m for per-process private filesystem mount points." endpoint.com. https://www.endpoint.com/blog/2012/01/27/linux-unshare-m-for-per-process-private (accessed Mar. 4, 2020).

[45] cgroup namespaces man page. "cgroup namespaces." man7.org. http://man7.org/linux/man-pages/man7/cgroup_namespaces.7.html (accessed Mar. 5, 2020).

[46] Network namespaces man page. "Network namespaces." man7.org. http://man7.org/linux/man-pages/man7/network_namespaces.7.html (accessed Mar. 5, 2020).

[47] veth man page. "Virtual Ethernet Device." man7.org. http://man7.org/linux/man-pages/man4/veth.4.html (accessed Mar. 5, 2020).

[48] Capabilities man page. "Linux capabilities." man7.org. http://man7.org/linux/man-pages/man7/capabilities.7.html (accessed Mar. 6, 2020).

[49] Yocto Project. "Yocto Project Overview and Concepts Manual." yoctoproject.com. https://www.yoctoproject.org/docs/3.0.2/overview-manual/overview-manual.html (accessed Mar. 13, 2020).

[50] LabJack. "T7." labjack.com. https://labjack.com/products/t7 (accessed Mar. 14, 2020).

[51] Docker blog. "5 years later, Docker has come a long way." docker.com. https://www.docker.com/blog/5-years-later-docker-come-long-way/ (accessed Mar. 24, 2020).

[52] Docker docs. "Docker overview." docs.docker.com. https://docs.docker.com/get-started/overview/ (accessed Mar. 24, 2020).

[53] Docker. "The Industry-Leading Container Runtime." docker.com. https://www.docker.com/products/container-runtime (accessed Mar. 24, 2020).

[54] LXC github. "LXC." github.com. https://github.com/lxc/lxc (accessed Mar. 25, 2020).

[55] LXC configuration man page. "lxc.conf." linuxcontainers.org. https://linuxcontainers.org/lxc/manpages/man5/lxc.conf.5.html (accessed Mar. 26, 2020).

[56] C. P. Wright. "Kernel Korner - Unionfs: Bringing Filesystems Together." linuxjournal.com. https://www.linuxjournal.com/article/7714 (accessed Mar. 30, 2020).

[57] A. Marinos. "Announcing balenaEngine: a container engine for IoT based on Moby technology from Docker". balena.io. https://www.balena.io/blog/announcing-balena-a-moby-based-container-engine-for-iot/ (accessed May 9, 2020).

[58] Mobyproject. "An open framework to assemble specialized container systems without reinventing the wheel." mobyproject.org http://mobyproject.org/ (accessed May 9, 2020).

[59] balena. "Getting started with balenaEngine." balena.io. https://www.balena.io/engine/docs/ (accessed May 9, 2020).

[60] Flatpak. "Frequently Asked Questions." flatpak.org. https://flatpak.org/faq/ (accessed May 10, 2020).

[61] Flatpak documentation. "Basic concepts." docs.flatpak.org. https://docs.flatpak.org/en/latest/basic-concepts.html (accessed May 10, 2020).

[62] Flatpak documentation. "Sandbox permissions." docs.flatpak.org. https://docs.flatpak.org/en/latest/sandbox-permissions.html (accessed May 10, 2020).

[63] Docker docs. "Dockerfile reference." docs.docker.com. https://docs.docker.com/engine/reference/builder/ (accessed May 28, 2020).

[64] Labjack. "LJM Library." labjack.com. https://labjack.com/ljm (accessed May 28, 2020).

[65] Flatpak documentation. "Manifests." docs.flatpak.com https://docs.flatpak.org/en/latest/manifests.html (accessed May 30, 2020).

[66] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange and C. A. F. De Rose. "Performance Evaluation of Container-based Virtualization for High Performance Comput. Environ.," in *2013 21st Euromicro Int. Conf. Parallel, Distrib., Netw.-Based Processing.*, Belfast, UK. Feb.-Mar. 2015, pp. 233-240.

[67] R. Morabito. "Power Consumption of Virtualization Technologies: An Empirical Investigation," in *2015 IEEE/ACM 8th Int. Conf. Utility Cloud Comput.*, Limassol, Cyprus. Dec. 2016, pp. 522-527.

[68] R. Morabito. "A Performance Evaluation of Container Technologies on Internet of Things Devices," in *2016 IEEE Conf. Comput. Commun. Workshop*, San Francisco, CA, USA. Sep. 2016, pp. 999-1000.

[69] Linux containers. "What's LXD?" linuxcontainers.org. https://linuxcontainers.org/lxd/introduction/ (accessed Jun. 3, 2020).

[70] OpenVZ. "Open source container-based virtualization for Linux." openvz.org. https://openvz.org/ (accessed Jun. 3, 2020).

[71] unshare man page. "unshare." man7.org. https://man7.org/linux/man-pages/man2/unshare.2.html (accessed Jun. 4, 2020).

[72] R. Ausanka-Crues. "Methods for Access Control: Advances and Limitations," Harvey Mudd College, Claremont, CA, USA.

[73] Kernel documentation. "Linux Security Module Usage." www.kernel.org. https://www.kernel.org/doc/html/v5.6/admin-guide/LSM/index.html (accessed Jun. 4, 2020).

[74] LXC container configuration man page. "lxc.container.conf." linuxcontainers.org. https://linuxcontainers.org/lxc/manpages/man5/lxc.container.conf.5.html (accessed Jun. 05, 2020).

[75] "RS485Pi." ABelectronicsUK. abelectronics.co.uk. https://www.abelectronics.co.uk/p/77/rs485-pi (accessed Jun. 10, 2020).

[76] Docker documentation. "Docker security." docs.docker.com. https://docs.docker.com/engine/security/security/ (accessed Jun. 22, 2020).

[77] Linux containers. "LXC Security." linuxcontainers.org. https://linuxcontainers.org/lxc/security/ (accessed Jun. 23, 2020).

[78] Flatpak github. "Sandbox." github.com. https://github.com/flatpak/flatpak/wiki/Sandbox (accessed Jun. 23, 2020).