TÉCNICO LISBOA

Java Network Virtualization

Jorge Catarino Estevão Tróia Godinho

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisor: Prof. Rodrigo Fraga Barcelos Paulus Bruno

Examination Committee

Chairperson: Prof. David Manuel Martins de Matos
Supervisor: Prof. Rodrigo Fraga Barcelos Paulus Bruno
Member of the Committee: Prof. Luís David Figueiredo Mascarenhas Moreira Pedrosa

November 2023

Declaration
I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

First, I would like to thank my supervisor, Prof. Rodrigo Bruno, for his availability and knowledge on this thesis topic. All the meetings with Prof. Rodrigo were very helpful. I want to thank my parents, Liliana and Jorge, for being so supportive and for being there for me when I needed them throughout all these past years. I also want to thank my brother, Alexandre, who has been the person with whom I have had the most contact in the past year and a half when we started living together in Lisbon, and also my grandfather, Jorge, and my grandmother, Maria Antónia, for being there for me in my entire educational progress. Last but not least, I want to thank all my friends with whom I have spent time within the last few years. They have helped a lot making these years the best years of my life.

Abstract

In Function-as-a-Service, an event-driven computing model, functions are invoked in multiple concurrent invocations. The runtime has to be initialized for each invocation, increasing the invocation latency. One way to minimize this time is for functions to share the same runtime. But, by having multiple functions sharing the same runtime, there are some issues, being network isolation one of them. We want to separate functions in order to call them in different ports, for example. The proposed solution for the network isolation issue is to use network namespaces to ensure network isolation and GraalVM Native Image Isolates to ensure memory isolation.

Keywords

Function-as-a-Service; Network Namespaces; GraalVM Native Image Isolates.

Resumo

No modelo *Function-as-a-Service*, um modelo de computação orientado a eventos, funções são invocadas em múltiplas invocações concorrentes. A cada invocação, o *runtime* tem de ser inicializado, o que faz com que a latência de invocação aumente. Uma maneira de minimizar esta latência é haver partilha de *runtimes* entre funções. No entanto, por haver funções a partilhar o mesmo *runtime*, surgem alguns problemas, sendo a isolamento de rede um deles. Algo que se pretende é isolar funções diferentes e poder invocá-las em diferentes portos, por exemplo. A solução proposta para o problema da isolamento de rede é usar *network namespaces* para garantir isolamento de rede e *GraalVM Native Image Isolates* para garantir isolação de memória.

Palavras Chave

Function-as-a-Service; *Network Namespaces*; *GraalVM Native Image Isolates.*

Contents

List of Figures

List of Algorithms

Listings

1

Introduction

Contents

In the early days of cloud computing, users were responsible for maintaining and managing a lot of infrastructure. This led to cloud computing offerings moving towards fine-grained virtualization where applications could automatically scale with minimal intervention from the user. This push for moving more responsibility away from users and into the cloud provider's side led to the appearance of new cloud services such as Function-as-a-Service (FaaS for short).

In FaaS, each function is containerized to run a short set of operations that perform some task. Applications are deployed as short business logic units, they scale automatically and are elastic and the user only pays for the usage of the invoked functions.

1.1 Problem

The existing virtualization techniques are quite expensive to ensure this fine granularity since the functions are very light and fast, and the virtualization becomes a bottleneck [10]. These virtualization techniques should be improved to minimize the cost of having a lightweight virtualization approach.

Some initial steps have already been done, such as using a single runtime to host multiple concurrent function invocations [3]. However, there are still isolation problems, such as ensuring network isolation.

By not isolating the network, there wouldn't be a rigorous way to call different functions. We want to be able to call different functions giving different network ports and have an opened socket for each function listening at those ports.

1.2 Goal

To virtualize Runtime Languages, the network needs to be virtualized. We will focus on how to isolate the network of multiple applications running in the same language runtime to ensure that each application doesn't share any network resource with any of the others that are running in the same environment.

We also need to measure the scalability of this network isolation to ensure it does not add any bottleneck in terms of the creation and deletion time of a function.

1.3 Proposed Solution

To achieve network isolation between multiple functions running in the same language runtime we are going to use network namespaces. Network namespaces are copies of the host network stack that provide isolation of network system resources. For each function, a new network namespace will be created and it will ensure that the function won't be able to communicate with any of the other functions. Although isolating each function's network, we still want to have control over its routes. If we want to have functions that exchange information between them, we should be able to add routes to make sure they can communicate.

We will also take advantage of existing memory isolation techniques, such as Native Image Isolates, and we will piggyback create and tear down to create and delete network namespaces.

This document will start with the background, where some concepts related to this thesis will be introduced. Then comes the state of the art, where some papers with similar goals as our thesis will be presented. After the state of the art comes the solution architecture, where the proposed solution is presented, approaching the network namespaces' and isolates' interface. After the solution architecture, we will present the implementation details. Then, there will be presented the evaluation results, and

finally, there will be a conclusion section.

2

Background

Contents

In this section, we will talk about some concepts that are related to the thesis, such as virtualization technologies, cloud services, memory isolation, and network isolation.

2.1 Virtualization Technologies

Virtualization [11] is the process to run a virtual instance of a computer system in a layer that is abstracted from the actual computer hardware. These virtual instances are called **Virtual Machines [12]**.

Each Virtual Machine runs a separate operating system that operates as an independent computer, although sharing the actual hardware resources.

Hypervisor is the software that is responsible for the coordination and provisioning of the resources between the VMs and the actual hardware.

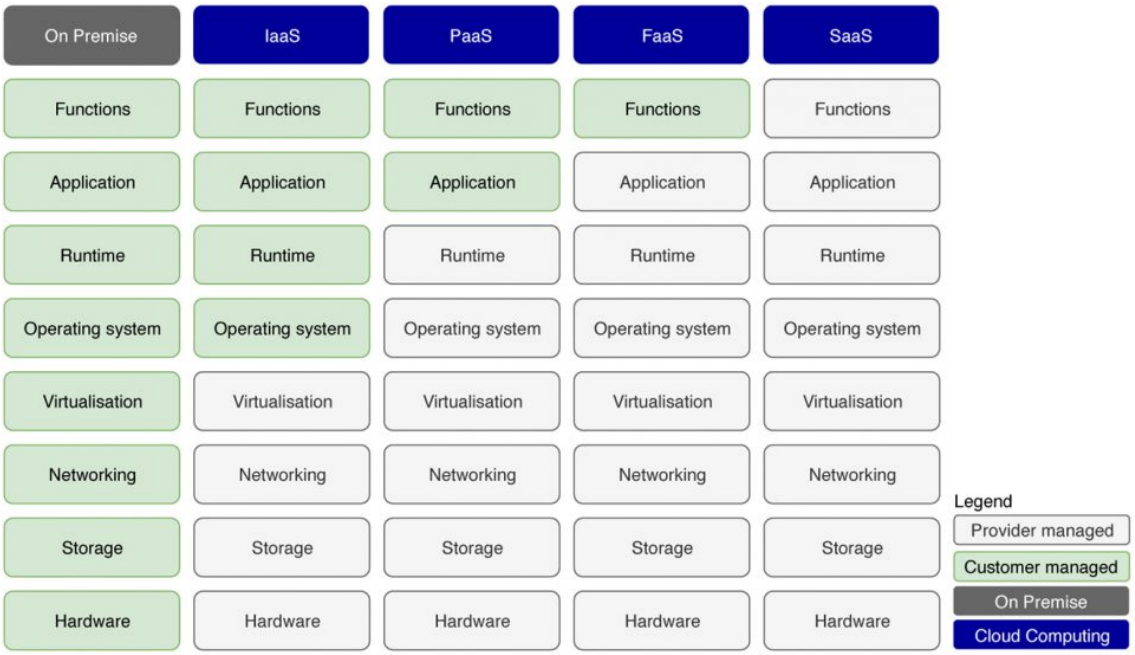

Figure 2.1: User's and cloud provider's responsibility in the different cloud computing models.

Container [13] [14] is an application deployment solution that packages all the software and all the needed dependencies needed to run an application. Containers run anywhere and they are small and fast because they don't include the guest operating system.

In other words, containers virtualize the host operating system, making them faster than Virtual Machines.

MicroVM [15] [16] [17] is a virtual machine that has its hardware isolated having a secure environment to run tasks having untrusted sources. By having this secure environment, MicroVMs have limited access to the operating system resources and can't interact with other processes.

Summing up, both virtual machines and MicroVMs are isolated from the host system, though virtual machines offer a more complete isolation, being MicroVms more lightweight. Containers are the most lightweight of these 3 virtualization techniques since they share the host operating system and kernel, and the isolation is placed at an application level.

We would use virtual machines when there is a complete isolation requirement, MicroVMs when we want strong isolation while being able to run untrusted tasks, and containers when we want to deploy scalable applications across different environments.

2.2 Cloud computing

Cloud computing was created to answer the need for companies to grow over time and expand their local infrastructure to adapt their computational needs to the demands of the customers.

Cloud computing is supported by some virtualization techniques such as Virtual Machines, containers, and MicroVMs, and it differs from traditional computing in the way that using these virtualization techniques, data is stored and services are hosted over the internet instead of locally.

Cloud computing can be exposed through a number of services that will be presented ahead, such as Infrastructure as a Service, Platform as a Service, Function as a Service, and Software as a Service. Figure 2.1 shows the barrier that exists between the user-managed and cloud provider-managed infrastructure in the different cloud computing models.

2.2.1 IaaS (Infrastructure as a Service)

Infrastructure as a Service [18] is a cloud computing model that offers virtualized computing resources. As presented in Figure 2.1, the cloud provider ensures the management of servers, networks, and storage and provides these resources through virtual machines. It also provides some services such as monitoring, detailed billing, load balancing, etc.

One of the main advantages of IaaS is that the user doesn't need to manage and buy the infrastructure behind the services, the provider takes care of all of that.

2.2.2 PaaS (Platform as a Service)

Platform as a Service [19] is a cloud computing model that delivers everything a user needs to deploy an application, that extends an existing platform/framework. Like IaaS, PaaS provides infrastructure through virtual machines, but it also provides development tools, database management systems, etc. as it supports the complete application lifecycle.

It has the same advantages as IaaS, and one of the main advantages over IaaS is that the time to code is lower, since PaaS provides pre-built frameworks that applications can extend to implement their domain logic.

2.2.3 FaaS (Function-as-a-Service)

Function-as-a-Service [20] is a cloud development model that executes some applications in response to events. All the infrastructure, such as physical hardware, virtual machines, operating systems, and web server software management are configured and handled by the cloud provider. This way, the user is only focused on the development of the application's functions.

FaaS has several advantages such as the focus on the code and not on the infrastructure, the user only pays for what he/she uses, so, every time the code isn't being executed, the user isn't charged and the functions are scaled up and down automatically depending on the function's demand.

Serverless [21] is a cloud development model that is mainly focused on the development of the application without the concern of infrastructure. The cloud provider provides the resources to the user and bills him for its computation. This way, FaaS is a subset of Serverless, since serverless has a wider range of development focus (compute, storage, API gateways, etc.), while FaaS has its focus on the event-driven programming model.

Serverless/FaaS is very attached to containers/VMs, and in order to have multiple functions running in the same runtime environment there are some concerns that have to be accounted for, such as memory isolation. Every process should only read and access its own memory space. This is not trivial, so, later in the document, we will introduce GraalVM, a technology that addresses this concern.

2.2.4 SaaS (Software as a Service)

Software as a Service [22] is a cloud computing model that allows users to use cloud-based apps over the internet, like calendaring, email, and office tools. Using SaaS, the user rents the service and the provider handles the infrastructure, middleware, software, and data, and also maintains it.

The main advantage of SaaS is that the user doesn't need to install, maintain, or update anything and the cloud provider handles all of that and provides a functional app for the user.

2.2.5 Function as a Service advantages

Function as a Service is interesting compared to Platform as a Service and Software as a Service due to the fact that Function as a Service is auto-scalable and it is very simple to deploy a function as the user only has to code the logic of the application.

2.3 Lightweight Virtualization

One way to reduce the overheads of virtualization is to allow multiple function invocations sharing the same runtime. In order to guarantee this, memory isolation is needed. Memory isolation can be achieved by using Isolates [23], a technology that will be presented ahead.

2.3.1 GraalVM Native Image

GraalVM Native Image [23] is a technology that ahead-of-time compiles Java code to an executable, called **native image**, which contains the application classes, dependencies' classes, runtime library

classes, and statically linked native code from JDK. This way it has a faster startup time and will most likely use less memory.

2.3.2 GraalVM Truffle

GraalVM Truffle [23] is an open-source library that provides ”language-level” virtualization, i.e. allows multiple languages to run in the same process, without adding any performance cost. It also provides a way for multiple applications to share the same runtime environment (for example the JVM). This is relevant for cloud computing since having a lot of small applications running in a different JVM is more expensive and stacking applications in a JVM can reduce significantly this cost.

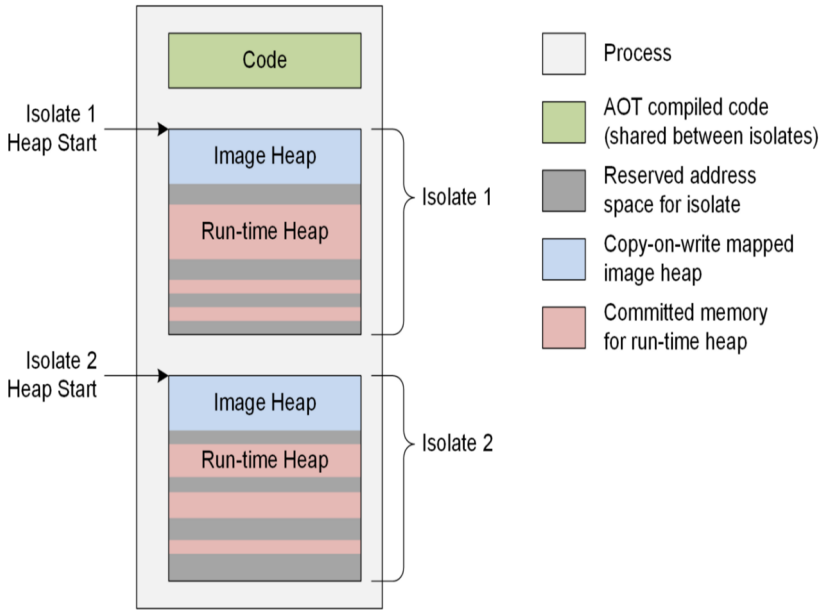

Figure 2.2: Heap divided in isolates.

2.3.3 Isolates

Isolates [23] is also a GraalVM feature. A GraalVM isolate is a disjoint heap. It allows having isolated memory for multiple threads running the same application. This is relevant to cloud computing since if one thread uses a lot of memory it can trigger garbage collection which slows all the other threads using the heap. Having independent heaps for each thread the garbage collection is called for a determined heap and the other threads are not penalized. Isolates have another optimization that is called compressed pointers. Instead of mapping memory addresses through 64-bit pointers, Isolates mapped them using 32-bit pointers. In the context of cloud computing, the isolates should be small. This way, 64-bit pointers would be a bottleneck and 32-bit pointers should be more than enough. Figure 2.2 shows

```
$ ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP mode DEFAULT group default qlen
1000
    link/ether 54:ee:74:c1:19:92 brd ff:ff:ff:ff:ff:ff
3: wlan0: <BROADCAST,MULTICAST> mtu 1500 qdisc noqueue state DOWN mode DEFAULT group default qlen 1000
    link/ether 52:54:00:f0:27:9a brd ff:ff:ff:ff:ff:ff permaddr 94:e9:79:fd:51:5d
```

Figure 2.3: Network interfaces.

two isolates in a process. We can see that each isolate has an image heap copy, that is managed using a copy-on-write mapping, and a run-time heap, independent for each isolate.

2.4 Network Isolation

With Isolates, the memory isolation chapter is closed. Similarly to memory isolation, there is network isolation. Every function in the same runtime environment should have its own network resources. The next **Linux technologies** will address this concern.

2.4.1 Network Interfaces

In Linux, a network interface is a point where the connection between the computer and the network is done. There are two types of network interfaces: Physical Network Interfaces, which represent physical network cards (hardware), and Virtual Network Interfaces, which are an abstraction of a network interface and may or may not be connected to a physical network card. An operating system may have any number of network cards and as many physical interfaces as physical network cards. Figure 2.3 shows the output of running the command `ip link`, which shows the network interfaces available in a computer and their details.

2.4.2 Routing tables

In Linux, a routing table [24] is a configuration that has information to make decisions about where to send data packets. If the destination host is on the local network, the data is sent directly to the destination host, if the destination host is on a remote network but that network is reachable via a local gateway that is in the routing table, the data is sent to that local gateway and if the destination host is on a remote network that is unreachable by any gateway in the routing table, the data is sent to the default gateway. Figure 2.4 shows the output of the command `route -n`, which shows the routing table of the default

```
[root@host1 ~]# route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref
 Use Iface
0.0.0.0         192.168.0.254   0.0.0.0         UG    100    0
 0 eno1
192.168.0.0     0.0.0.0         255.255.255.0   U     100    0
 0 eno1
```

Figure 2.4: Routing table.

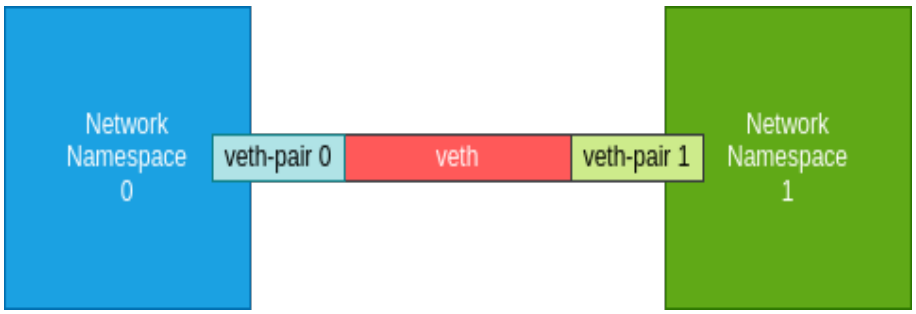

Figure 2.5: Network namespaces connected by a veth.

interface.

2.4.3 Network Namespaces and Virtual Ethernet Devices

Network isolation can be achieved by combining the following concepts: network namespace and veth (Virtual Ethernet Device). Network namespaces are copies of the host network stack that provide the isolation of the system resources such as network devices, IPv4 and IPv6 protocol stacks, routing tables, port numbers (sockets), etc. A virtual network device pair (veth) acts like a tunnel between network namespaces.

Figure 2.5 is a visual demonstration of two network namespaces connected with a veth pair.

A network namespace can be created and deleted using the following shell commands:

```
1 ip netns add <namespace_name>
2 ip netns delete <namespace_name>
```

In order to create a veth and connect it to a namespace we have to use these shell commands:

```
1 ip link add <entry_point_veth_name>
2     type veth peer name <function_veth_name>
```

```
3 ip link set <function_veth_name>
4     netns <function_namespace_name>
```

3

State of the art

Contents

In this section, we will cover some works related to FaaS that try to achieve similar goals as this thesis, i.e. function's isolation in general.

3.1 Groundhog: Efficient Request Isolation in FaaS [1]

In FaaS, each function is executed in its container to prevent concurrent executions of different functions. Consequent invocations of the same function reuse the state of the previous invocations to minimize cold-start delays. Since security is a core responsibility of FaaS, this becomes a huge problem. Groundhog isolates subsequent invocations of the same function in order to have a clean state free of data leaks every time the function is invoked. Groundhog achieves isolation using a lightweight process snapshot/restore mechanism. Each function is encapsulated in a containerized process and a snapshot of each function's fully initialized state is taken. This state is free of any requests, so it should be free of any secrets that could appear by handling requests. Groundhog uses this snapshot to restore the runtime state after a function returns its result. This way it ensures that subsequent requests have the initial state. The Groundhog architecture is shown in Figure 3.1.

With this paper, security is achieved since every invocation will have a clean start state. Network isolation is also achieved, but in a different way than we want. Groundhog uses containers to isolate functions, and our goal is to separate functions running in the same runtime.

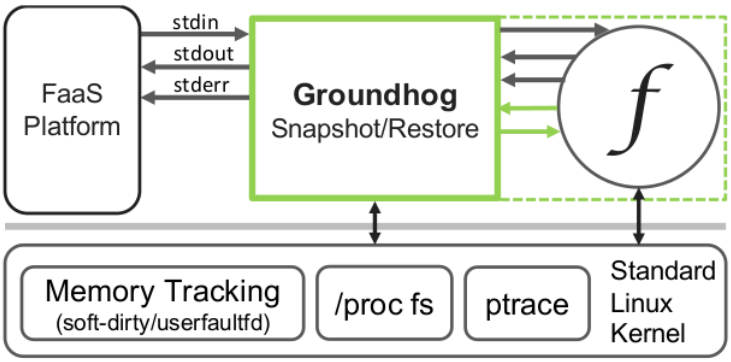

Figure 3.1: Groundhog architecture [1].

3.2 From Warm to Hot Starts: Leveraging Runtimes for the Serverless Era [2]

This paper hints at the fact that serverless functions that run in runtimes like JVM and Node.js run unoptimized code most of the time. After the cold-start, as the function is being invoked, the runtime makes calls to the Just-In-Time compiler to optimize the code (warm-start). When the runtime generates the most optimal code, the function is going to execute with the maximum performance possible (hot-start). Figure 3.2 shows two different code pipelines, the JavaScript Engine (V8) pipeline and the JVM

(HotSpot) pipeline. Due to serverless functions' short-living nature, the function might never reach the best performance. The paper proposes Ignite, a system where runtimes cooperate to generate the most optimized code, sharing the code optimizations between them.

Although the system offers a way to minimize cold starts, invocations are still executed with no network isolation.

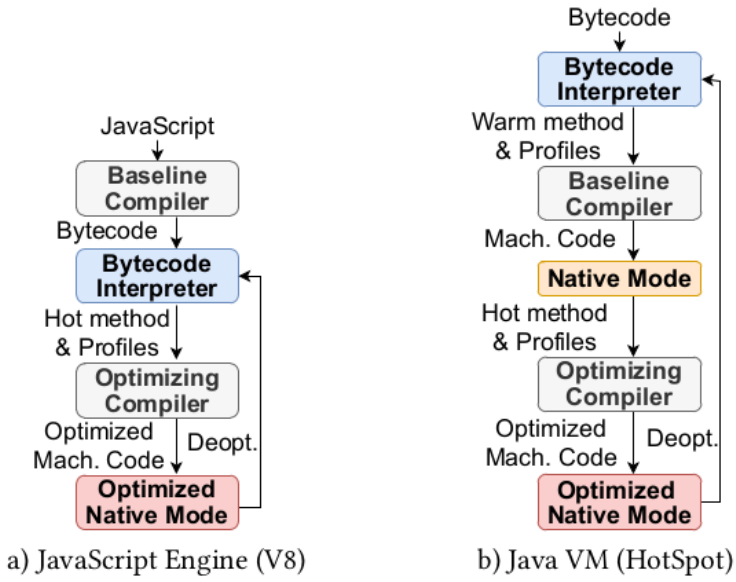

Figure 3.2: Code pipeline JavaScript and Java runtimes [2].

3.3 Photons: Lambdas on a diet [3]

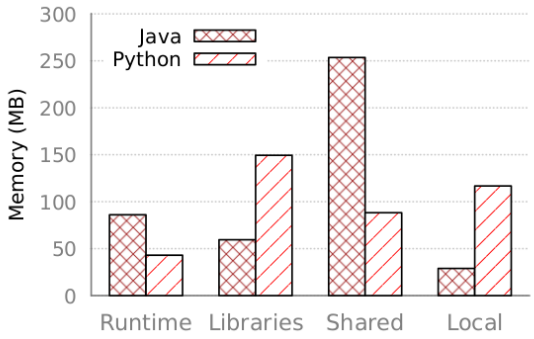

Figure 3.3: Memory usage in a serverless image classification task [3].

Nowadays, serverless platforms initialize and schedule each invocation separately, even when a countless number of invocations use the same code and runtime. There are two pointed inefficiencies.

The first inefficiency is that there is no memory being shared among the invocations, which increases memory utilization. Figure 3.3 confirms this inefficiency, showing that most of the memory used is for the runtime, libraries, and the machine learning model, which becomes the same for all invocations. The second inefficiency is that each invocation has to initialize its runtime imposing a high total execution time overhead. This paper presents Photons, an execution context based on runtime and app-state virtualization for serverless functions, like Figure 3.4 suggests. Photons use a very lightweight isolation layer based on bytecode rewriting while allowing safe runtime sharing to concurrently run many instances of the same serverless function.

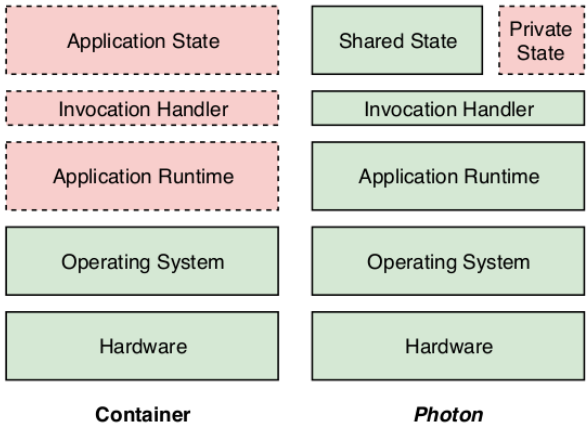

Figure 3.4: Virtualization layers in today's serverless containers versus virtualization layers in Photons [3].

Although Photons share states between runtimes to safely run concurrent instances of functions, it doesn't ensure network isolation.

3.4 Pushing Serverless to the Edge with WebAssembly Runtimes [4]

Serverless computing is ideal for handling unpredictable and bursty workloads, although, cold-start latencies of hundreds of milliseconds impede support for latency-critical IoT services. The authors of this paper argue that OS-level virtualization is unsuitable for serverless edge computing, so they intend to replace it with a technology called WebAssembly (Wasm) [25]. Wasm is a portable, binary instruction format for memory-safe, sandboxed execution. Its functions' creation and deletion time are around 10s microseconds, approximately 10 times faster than usual AWS and Google Cloud Platform functions that lay around milliseconds. Figure 3.5 shows the invocation flow of a Wasm action. Basically, an invoker injects code into a Wasm Executer, which then creates a Wasm Module ready for execution. The invoker then instructs the Wasm Executer to invoke the Wasm Module. The result is then returned to the invoker.

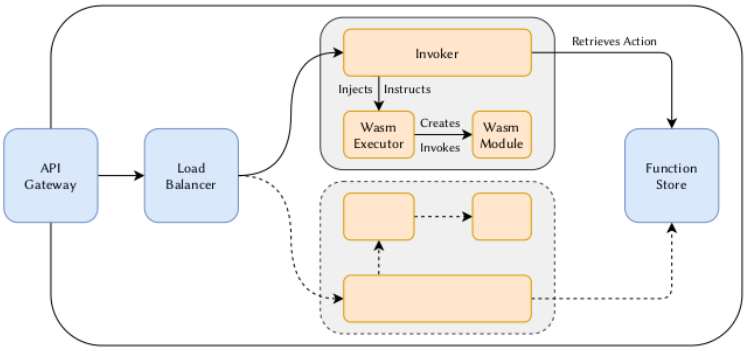

Figure 3.5: Invocation flow of a Wasm action [4].

This paper proposes using WebAssembly [25] which reduces function creation and deletion times to much lower values but it doesn't propose/enforce any network isolation.

3.5 Nightcore: Efficient and Scalable Serverless Computing for Latency-Sensitive, Interactive Microservices [5]

The microservice architecture is a popular software engineering approach for building large-scale online services that are used by many big companies, such as Amazon, Netflix, etc. In Figure 3.6 there is a graph that represents the flow of uploading a new post in a micro-service-based SocialNetwork application [8]. Serverless computing offers a new way of building microservice-based applications. However, FaaS systems have invocation latency overheads ranging from a few to tens of milliseconds, making them a bad choice for latency-sensitive microservice-based applications. Latency-sensitive microservice-based applications also have another performance challenge, due to the high invocation of requests these applications should support. So, the authors state two performance goals to efficiently support interactive microservices, that are: invocation latency overheads must be well within 100 microseconds and the invocation rate must scale to 100K/s with a low CPU usage. Previous studies showed that FaaS runtime overheads could be reduced to a microsecond scale but with the cost of weakening isolation between functions. To follow the microservice architecture approach, isolation should be guaranteed, but achieving the goals written above with proper isolation between the functions is a technical challenge. So, the authors present Nightcore, which is a serverless function runtime designed to reach the goals above, to ensure high performance, and also guarantee isolation between functions.

Nightcore proposes container-level-based isolation of each function with the goal of following the microservice architecture approach. Network isolation is achieved by using containers. The goal of this thesis isn't to achieve network isolation using containers, it is to achieve network isolation for functions

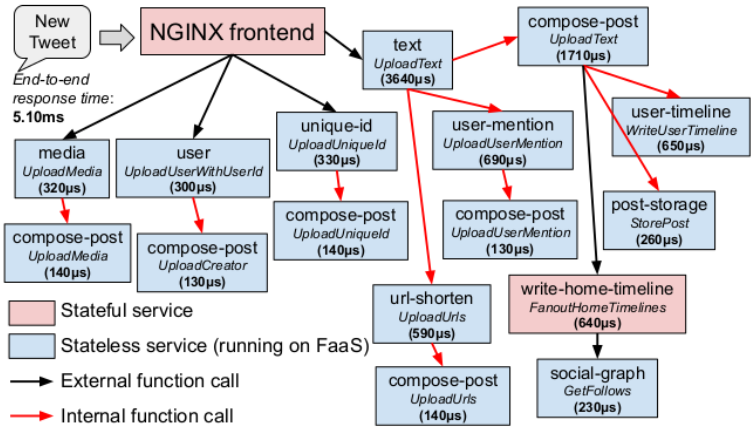

Figure 3.6: RPC graph of uploading new post in a microservice-based SocialNetwork application [8]. This graph omits stateful services for data caching and data storage [5].

running in the same runtime.

3.6 FaasCache: Keeping Serverless Computing Alive with Greedy-Dual Caching [6]

The initialization of a FaaS container (i.e., the cold-start) which involves the container creation and fetching of the necessary code and dependencies adds significant overhead to the function latency. To weaken cold-start latency, a common technique used is to keep the runtime environment warm for a certain duration, so that future function invocations reuse this already initialized runtime environment. Although reducing the cold-start latency, it consumes more resources, which can reduce the overall system utilization and efficiency. The authors state that keep-alive policies can have a critical impact on performance and should be introduced into resource allocation and provisioning. They will focus on how to balance the latency vs. utilization tradeoff by developing new resource management techniques. Keep-alive policies should be based on the functions' usage, so here is a challenge, since there is a wide variety of functions with different request frequencies. The authors correlate functions' resource management with object caching, where keeping a function warm is equivalent to caching an object and a warm invocation is equivalent to a cache hit. Destroying one function execution will originate a cache miss in its next invocation. They use and adapt the Greedy-Dual caching framework and develop keep-alive policies based on it, which will improve the function latency and the system utilization and efficiency.

This paper allows a function to have a more in-depth keep-alive policy in terms of its usage but it ignores the problem of network isolation.

3.7 RunD: A Lightweight Secure Container Runtime for High-density Deployment and High-concurrency Startup in Serverless Computing [7]

In serverless computing, microVMs have started hosting isolated containers. There is a high demand for deployment and concurrency container startup, in order to efficiently manage resource utilization and also to offer a better experience to the user since functions are fine-grained in serverless systems.

Investigations concluded that these software stacks are inefficient in creating rootfs and groups, which results in low container startup concurrency. It also has a high memory footprint, which results in low container deployment density. These bottlenecks are showed in Figure 3.7.

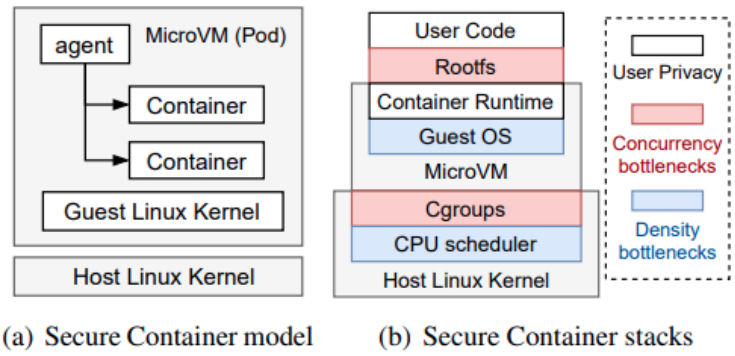

Figure 3.7: The state-of-the-art secure container model, and several bottlenecks in the architecture stacks [5].

The authors propose RunD, a lightweight secure container runtime that resolves the problems of duplicated data across containers, high memory footprint, and high host-side cgroup overhead. The authors could conclude that data user-provided data is read-only to the operating system and operating system-provided runtime files are also read-only for user functions. Addressing the challenge of resolving the high-density and high-concurrency scenario for container rootfs, this information is used in order to separate read-only data from writable data. In order to resolve the memory used by each container, the authors propose two techniques that the deployment density can be increased. These techniques are reducing the guest kernel size by disabling some features that are not going to be used and alleviating the code self-modification, by using a pre-patched microVM template. Lastly, to resolve the bottleneck of the cgroup operations, they conducted an investigation and found out that creating groups concurrently is time-consuming since the kernel can't parallelize cgroup-related operations, and that pre-creating and maintaining groups in a pool could reduce the group creation overhead since only the cgroup rename is used. Using these conclusions, the authors proposed a lightweight cgroup to address cgroup-operations overhead.

3.8 Discussion

This section presented a number of papers related to the central topics of this work: lightweight virtualization and how to improve the performance of serverless applications. Some of them mentioned network isolation and functions running in the same runtime, but none of the papers achieved our thesis goal, which is to ensure network isolation in functions that share the same runtime environment.

Groundhog [1] focused on clearing the state of subsequent invocations of the same function in order to reuse the function's already initialized environment and Nightcore [5] focused on ensuring high performance and isolating functions, two objectives that together were an issue to accomplish. Although these papers achieved network isolation, they used containers in order to do that.

Photons [3] achieved lightweight virtualization by moving functions to the same runtime. However, network isolation wasn't achieved.

Ignite [2] and Pushing Serverless to the Edge with WebAssembly Runtimes [4] offer different ways of minimizing cold starts. In Ignite [2] runtimes cooperate between them to generate the most optimized code and make fewer calls to the Just-In-Time-Compiler and in Pushing Serverless to the Edge with WebAssembly Runtimes [4] it is proposed to use WebAssembly [25] which is gonna reduce the function create and deletion time. Network isolation was achieved in neither of these papers.

Finally, FaasCache [6] proposes a way of predicting cold starts and this way keeps functions alive based on their usage. Again, network isolation wasn't achieved in this paper.

4

Solution architecture

Contents

In this section, we will move to the proposal of the solution architecture. It starts with a brief overview of the solution, then we'll move to how the Linux namespace interface works and how can we use it. Then we will discuss how to connect namespaces to Native Image Isolates, where function invocations will be handled.

4.1 Solution Overview

Figure 4.1 shows the base architecture to achieve network isolation in a Native Image runtime that has multiple functions using isolates. The runtime is composed of:

- **A main thread (entry point thread)**, which is the entry point of our runtime and that has the responsibility of forwarding the incoming requests to another thread according to the function the request is calling. The entry point thread inherits the host's default network namespace;

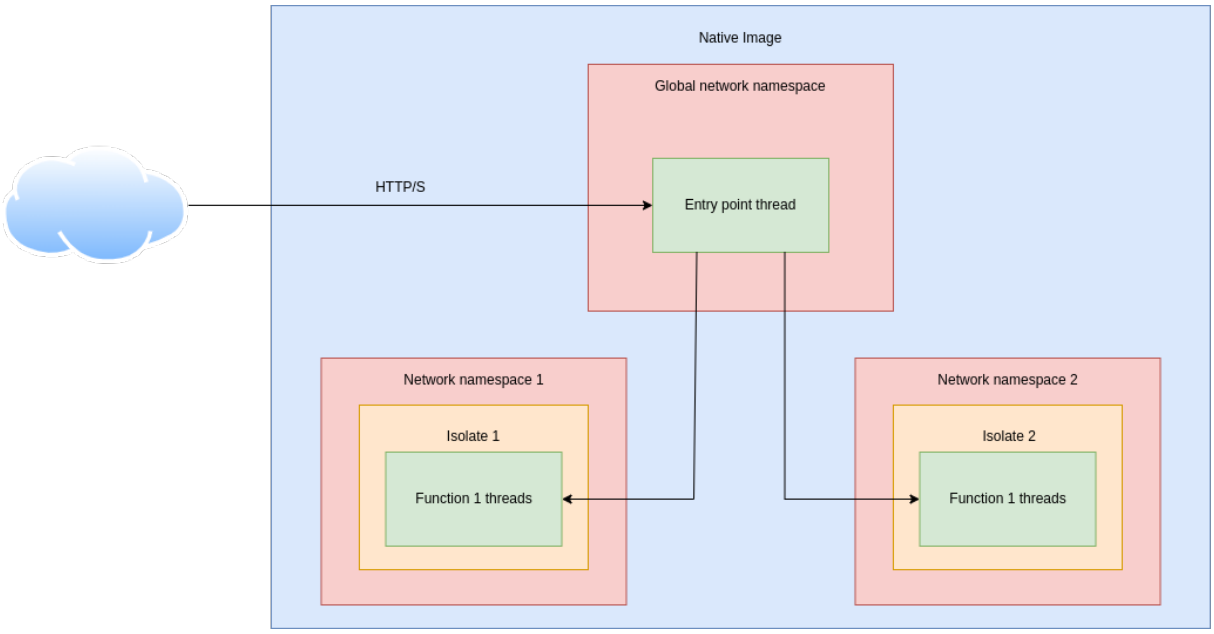

Figure 4.1: System base architecture.

- **Function threads**, that run some function invocation code. Similarly to the entry point, each of these threads executes inside an isolate which ensures that there is one heap for each group of function threads. Each group of function threads will also execute in a network namespace, unique to each of them, to ensure there is network isolation between them in the runtime.

If there were no different network namespaces, all threads would share the default network namespace and, as a consequence, network isolation would not be achieved. With each group of function threads having a unique network namespace attached to it, we can create rules, explained in the subsequent section, that dictate how the network stack of these threads is isolated.

4.2 Linux networking interface

As explained in the previous section, we will have to create some rules between network namespaces to ensure network isolation. In order to do that, we will take advantage of already existing calls from the Linux Kernel that will allow us to create network namespaces, create and enable veths, link veths to network namespaces, set the IP addresses of the devices, set default network gateways and assign a thread to a specific network namespace.

We will follow the isolation flow of a function being created, by ordering the commands that need to be executed, which in the end will result in a function that is completely isolated from the other ones.

So, the first thing we need to do is create the network namespace that is going to be associated with that function. To do that we can simply run the following command:

```
ip netns add <namespace_name>
```

After creating the network namespace, we will need to create a `veth` that will connect the function's network namespace to the entry point's network namespace. Each `veth` has 2 edges, the entry point edge and the function edge, which we call `entry point veth name` and `function veth name`, respectively. A `veth` is created by running the command:

```
ip link add <entry_point_veth_name>
    type veth peer name <function_veth_name>
```

A `veth` is created in the scope of a network namespace, i.e. one of the edges of the `veth` will be automatically assigned to the network namespace where the command to create the `veth` was executed. The `veth` creation will be executed in the scope of the default network namespace (entry point's network namespace), so the next step will be to assign the function's edge to the function's network namespace by running the command:

```
ip link set <function_veth_name>
    netns <function_namespace_name>
```

Now it is time to assign IP addresses to each of the edges of the `veth`. For each function, a subnet will be created. The IPs will be assigned depending on each subnet mask. It can be done by running these commands:

```
ip addr add <entry_point_ip> dev <entry_point_veth_name>

ip netns exec <function_namespace_name>
    ip addr add <function_ip> dev <function_veth_name>
```

After creating and setting up the `veth`, we will need to enable its edges, since its edges are disabled by default when a `veth` is created. To do that, we can simply run the commands:

```
ip link set <entry_point_veth_name> up

ip netns exec <function_namespace_name>
    ip link set <function_veth_name> up
```

After creating a network namespace, its routing table is empty by default, so now we will need to set the default network gateway to the entry point's IP by running the command:

```
ip netns exec <function_namespace_name>
    route add
    default gw <entry_point_ip> <function_veth_name>
```

By default, a network namespace can't connect with the internet, because the network namespaces veths have private static IP addresses assigned. Since the IPs are static, we can't use NAT (Network Address Translation) with the default settings. In order to make possible the translation, we will use iptables to masquerade and accept traffic in the entrypoint veths. In order to do these operation we have to run these commands:

```
iptables -t nat
    -A POSTROUTING -s <entry_point_ip>
    -o <forward_interface_name> -j MASQUERADE

iptables -A FORWARD
    -i <forward_interface_name>
    -o <entry_point_veth_name> -j ACCEPT

iptables -A FORWARD
    -o <forward_interface_name>
    -i <entry_point_veth_name> -j ACCEPT
```

The forward network interface can be obtained by running the following command:

```
ip -o route | grep default | awk '{print $5}'
```

By now, we will have a completely isolated network namespace. All there is left to be done is assign the thread where the function will run to this new network namespace. That can be done using a C function called `setns`, which receives the file descriptor (FD) of the new network namespace and a flag that represents the type of the Process ID given (we should use the flag that indicates that the PID corresponds to a network namespace). After calling this function, the current thread will be now running on top of the given network namespace.

```
int setns(int fd, int nstype);
```

At some point, we will want to delete the namespaces. This can be done by running the following command:

```
ip netns delete <function_namespace_name>
```

We have not found yet a programmatic interface to embed this in the code, with the exception of the setns function. If there is one programmatic interface, the replacement would be trivial.

4.3 Connecting namespaces with isolates

We want to connect the isolate creation and invocation with our serverless system. As our first solution, presented as Algorithm 1, the system will receive requests to invoke functions. In each request, the network will be isolated following the steps of the previous subsection. Now, already in the function's thread which is already assigned to the new network namespace, we will have to create an isolate, copy the necessary objects to that new isolate, invoke the method that executes the code in the new isolate, copy the function's result from the new isolate to the source isolate, tear down the new isolate, and then delete the network namespace. Although this works, it is very inefficient. In every function invocation, we will virtualize the network and also create an isolate, and this is very expensive.

This way, we propose a second solution, presented as Algorithm 2, where we will reuse namespaces and isolates. In this solution, we have a list that has available namespaces, i.e. namespaces that are not running any function. We start by receiving a request and parsing the arguments. Then we verify if there is any available namespace. If there is an available namespace, we will get that namespace, remove it from the list, and attach the current thread to that isolate. If there is no namespace available, we will create one, isolate the network, and then create an isolate. After isolating the network and heap, the function is invoked, but it won't tear down the isolate. Instead of that, we will detach the current thread from the isolate and add the namespace back to the namespace list. By using this algorithm, subsequent requests can reuse namespaces and isolates that were used in previous requests. By detaching the threads from the isolates, the state that was created by them will be cleaned, so it is safe to reuse the isolates and network namespaces.

There will also exist a thread that will run Algorithm 3. This algorithm will delete unused namespaces and isolates. This thread will be scheduled to run within a certain interval of time, which is a threshold for network namespace and isolate non-utilization.

Algorithm 4.1: First solution to integrate isolates with the system.

```
while true do
    request ← getRequest();
    args ← request.getArgs();
    networkNamespace ← createNetworkNamespaceAndIsolateNetwork();
    newIsolate ← createIsolate();
    currentIsolate ← getCurrentIsolate();
    argsHandle ← copyArgsToIsolate(newIsolate, args );
    resultHandle ← invokeFunction(newIsolate, currentIsolate, argsHandle );
    result ← getResultFromHandle(resultHandle );
    teardownIsolate(newIsolate );
    deleteNetworkNamespace(networkNamespace );
    sendResponseBack(result );
```

Algorithm 4.2: Second solution to integrate isolates with the system with namespace and isolate reuse.

```
while true do
    request ← getRequest();
    args ← request.getArgs();
    if availableNetworkNamespaces.isEmpty() then
        networkNamespace ← createNetworkNamespaceAndIsolateNetwork();
        isolateThread ← createIsolate();
        isolate ← getIsolateFromIsolateThread();
        networkNamespace.setIsolate(isolate );
    else
        networkNamespace ← availableNetworkNamespaces.get(0);
        availableNetworkNamespaces.remove(networkNamespace );
        isolate ← namespace.getIsolate();
        isolateThread ← attachCurrentThread(isolate );
    currentIsolate ← getCurrentIsolate();
    argsHandle ← copyArgsToIsolate(isolateThread, args );
    resultHandle ← invokeFunction(isolateThread, currentIsolate, argsHandle );
    result ← getResultFromHandle(resultHandle );
    detatchIsolateThread(isolateThread );
    availableNetworkNamespaces.add(networkNamespace );
    sendResponseBack(result );
```

Algorithm 4.3: Namespace deletion and isolate tear down.

```
while running do
    if availableNetworkNamespaces.size() > MAX_NETWORK_NAMESPACES then
        while availableNetworkNamespaces.size() > MAX_NETWORK_NAMESPACES do
            networkNamespace ← availableNetworkNamespaces.get(0);
            availableNetworkNamespaces.remove(networkNamespace );
            teardownIsolate(networkNamespace.getIsolate());
            deleteNetworkNamespace(networkNamespace );
    sleep (60);
```

5

Implementation

Contents

The network virtualization, using network namespaces, was integrated on a high-performance serverless platform called Graalvisor [26]. Graalvisor combines the use concepts of Native Image, Isolate, and Truffle to have function invocations at a large scale with low latency and low memory footprint, compared with other traditional serverless platforms.

5.1 Graalvisor

As stated before, Graalvisor already makes use of the concepts of Native Image, Isolate, and Truffle in order to have low latency and low memory footprint on function invocations at a larger scale. Since FaaS is executed in a Serverless approach, users don't have control over the environment where the function is running. Traditional environment virtualization techniques are quite expensive, but by using Native Image Isolates it is possible to virtualize the environment where the function runs without any

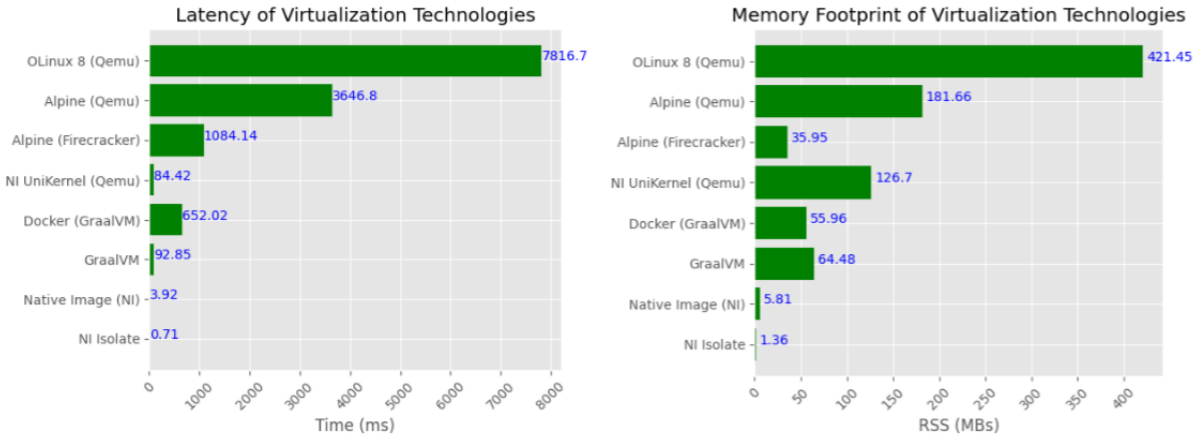

Figure 5.1: Native Image Isolates performance [9].

performance issues as Image 5.1 shows. Like traditional Serverless platforms, Argo offers an endpoint to register new functions and another endpoint to invoke registered functions.

Graalvisor offers serverless function wrappers that offer support for Native Image Isolates and Truffle Languages. A Lambda Proxy is the entry point of function invocations. It is responsible for the register and management of functions and also responsible for function invocations by making use of HTTP endpoints.

As Figure 5.2 shows, Graalvisor has a thread pool that contains threads that will be used to execute invocation requests. If at any time there is no thread available to execute a request, a new thread is created and added to the pool. Threads that are inactive for sixty seconds are terminated and removed from the thread pool. These threads have a record of function cache, that contains the isolate pool for each of the registered functions. Similar to the thread pool, if any time there is no isolate available to perform an invocation, a new isolate is created, and an isolate is destroyed and removed from the isolate pool after sixty seconds of inactivity. Each isolate contains in memory the associated function code and the invocation data needed to execute the function.

5.2 Native Image Isolates implementation

Graalvisor already has support for function execution using Native Image Isolates. There are two Isolate implementations, one is cached and the other one isn't cached. Both can run in the same runtime environment. A specific argument can be passed to the function invocation request in order to run one implementation or the other.

When a function invocation request is received and there is not any cache argument, or the cache argument is set to false, a new Isolate will be created, then the function will be invoked with the arguments passed, and finally, after the invocation ended, this Isolate will be deleted. This Isolate will only

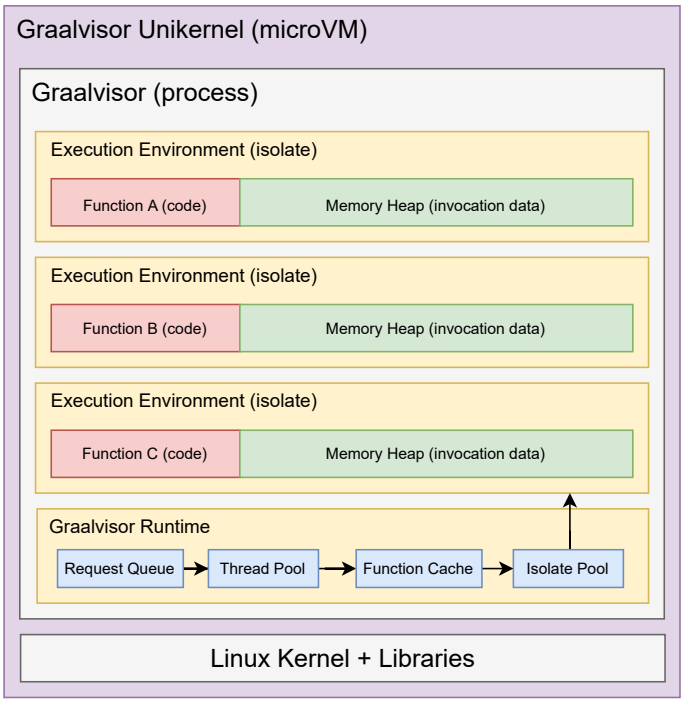

Figure 5.2: Graalvisor architecture.

serve this particular invocation, so each invocation request will create one Isolate and then destroy it.

On the other hand, when a function invocation is received with the cache argument set to true, things get more complex. There are workers that process requests and each worker is associated with a function pipeline that contains a queue of requests associated with a particular function. Workers retrieve requests from this queue in order to process them. A worker is an isolated thread and each has a unique Isolate. The worker thread starts by creating the Isolate, and then it starts an infinite loop processing requests by polling requests from the associated function pipeline. After 60 seconds of not being able to retrieve any request from the function pipeline, the infinite loop ends. The Isolate is destroyed and the worker is killed.

5.2.1 Isolates Java API

Graalvisor uses the Java API for Isolates [23, 27] in order to manage Isolates. `Isolates` is the isolate main type and every thread that is assigned to an isolate is represented by `IsolateThread`. This `IsolateThread` type will be the one that is going to be used to invoke a method in an isolate.

```
1 public final class Isolates {
2
3     public static IsolateThread createIsolate(
```

```
4           Isolates.CreateIsolateParameters parameters);
5
6       public static IsolateThread attachCurrentThread(
7       Isolate isolate);
8
9       public static void detachThread(IsolateThread thread);
10
11      public static void tearDownIsolate(
12          IsolateThread thread);
13
14  }
```

Listing 5.1: Java Isolates API.

The `createIsolate` method, as the name suggests, creates the new isolate. Objects of the calling isolate will not be available in the new isolate and the thread assigned to the new isolate will be the thread where the createIsolate method is called.

The `attachCurrentThread` method will assign the current thread to the isolate passed as an argument, returning an object of type `IsolateThread`, which links the isolate to the current thread. If the thread passed as an argument is already assigned to the isolate, the returning value will be the object that already connects the isolate to the current thread.

The `detachThread` method will detach the thread passed as an argument, discarding any state created by the thread in the heap. When this method is called there cannot be any code being executed in the isolate thread's context.

The `tearDownIsolate` method takes down the isolate associated with the thread passed as an argument. It waits for other threads to detach from the isolate and then discards the isolate's objects, threads, and any other state.

In order to do some computational work in the isolates, a method needs to be invoked, but in a different way. It is different since we are invoking the method in one isolate and executing it in another isolate. These isolates have different heaps, so we cannot pass Java objects as arguments directly. This way, we will need to copy the objects that we need from the calling isolate to the execution isolate. The same thing happens with the return value. In order to copy the objects, we will have to use handles. Handles are an opaque indirection to a Java object and the object they refer to can only be accessed in the isolate in which the handle was created. The `@CEntryPoint` annotation marks the method as isolate-transition, i.e. a method that is going to be executed in a different isolate. That isolate has to be passed as an argument with the `@CEntryPoint.IsolateThreadContext` annotation. To return the result we will have to create an `ObjectHandle` object in order to copy the result to the source isolate, so, if there is any result to be returned, we should pass the source isolate as an argument as well.

Here is an example of the method signatures that would be needed to use a new isolate to perform a concatenation of two strings. The `concat` method would be called inside the `concatInIsolate` method after copying the strings from the source isolate to the target one. The `copyString` methods will create the handle to copy the string. The first method, the one without the annotation, is going to run in the source isolate and will transform the Java `String` object to a C string (`CCharPointer`). The second method, the one with the `@CEntryPoint` annotation, is going to run in the target isolate and will transform the C string into a Java `String` object.

```
1  public class StringConcat {
2
3      public String concatInIsolate(
4          String s1,
5          String s2);
6
7      @CEntryPoint
8      private static ObjectHandle concat(
9          @CEntryPoint.IsolateThreadContext
10             IsolateThread targetContext,
11         IsolateThread sourceContext,
12         ObjectHandle s1Handle,
13         ObjectHandle s2Handle);
14
15     private static ObjectHandle copyString(
16         IsolateThread concatContext,
17         String sourceString)
18
19     @CEntryPoint
20     private static ObjectHandle copyString(
21         @CEntryPoint.IsolateThreadContext
22             IsolateThread renderingContext,
23         CCharPointer cString);
24
25 }
```

5.3 Network namespaces

The network namespace implementation has two parts to it. The first iteration consists of creating, setting up, using, and deleting a network namespace at each function invocation. Using a network namespace means the invocation will not run in the default network namespace but in the created network namespace.

5.3.1 Network namespaces native code implementation

The network namespaces implementation uses the Java Native Interface in order to be able to run native code. There are 6 native methods registered that do the following:

- Create a network namespace
- Delete a network namespace
- Switch the current thread to network namespace
- Switch the current thread to the default network namespace
- Disable veths
- Enable veths

5.3.1.A Create network namespace native method

The create network namespace native method has 3 arguments, the name of the network namespace, and the second and third bytes of the veths' ip addresses. The method starts by creating the network namespace. After creating the network namespace, it creates two veths. When a veth is created, it is linked to the default network namespace by default, so after creating the veths it links one of the veths to the created network namespace.

After linking one of the veths to the new network namespace, it is time to assign an ip address to each of the veths. In our logic, each network namespace has an id which is an integer that is incremented every time a new network namespace is created. Every veth ip will have the format 10.X.Y.Z/24, where X is the whole division of the network namespace id by 256, the Y is the network namespace id modulus 256, and the Z is 1 if the veth is linked to the default network namespace or 2 otherwise.

After assigning an ip address to each of the veths, each one of them has to be enabled, since they are disabled by default when they are created.

In order for a network namespace to have access to the internet, there are some steps needed. The first is to use iptables nat masquerade to hide address translation, and then use iptables nat forwarding to forward packets.

5.3.1.B Delete network namespace native method

The delete network namespace native method has 1 argument, the network namespace name. It starts by deleting the default network namespace veth associated with the request network namespace, followed by deleting the requested network namespace.

5.3.1.C Switch to network namespace native method

The switch to network namespace native method has 1 argument, the network namespace name. It switches the current thread to the requested network namespace.

5.3.1.D Switch to the default network namespace native method

The switch to the default network namespace native method has no arguments. It switches the current thread to the default network namespace.

5.3.1.E Disable veths native method

The disable veths native method has 1 argument, the network namespace name. It brings down both the default network namespace veth associated with the requested network namespace and the requested network namespace veth. By bringing them down, the default network gateway of the requested network namespace is deleted.

5.3.1.F Enable veths native method

The enable veths native method has 1 argument, the network namespace name. It enables the network namespace veth associated with the requested network namespace and the request network namespace veth.

This method is supposed to be called after the disable veths method. As said before, after disabling the veths, the default network gateway of the network namespace is deleted. So, in order to set the network namespace back up, the default network gateway is set again.

5.3.2 Network namespace logic implementation

The network namespaces have their logic centered on a Java class called NetworkNamespaceProvider. The resulting object of this class provides methods that call the native methods mentioned above. It has complete autonomy over which names to give to new network namespaces. There is also a class called NetworkNamespace that contains relevant information about a network namespace.

The network namespace implementation had two iterations. The first iteration consists of creating, setting up, using, and deleting a network namespace at each function invocation. Using a network namespace means the invocation will not run in the default network namespace but in the created network namespace.

There are two possible ways to use Native Image Isolates in an invocation: reuse a cached isolate and create and delete one isolate for each invocation. We considered the network namespace utilization in both of these options, although we are only interested in the Native Image Isolates cache option.

5.3.2.A Creating a network namespace at each function invocation

In this approach, the network namespace provider keeps track of how many network namespaces have already been created. Whenever a new network namespace is created, it increments that variable and that number becomes the id of that new network namespace.

This approach added a significant overhead in terms of latency since the time to create a network namespace and delete it takes some hundreds of milliseconds. Because of this, we had to think of another possibility that could add minimal overhead to invocation latency.

5.3.2.B Network namespaces cache

As we will present further in this document, switching to a new network namespace and switching to the default network namespace takes around dozens of microseconds, which is pretty fast.

In this iteration, considering this fact, we took advantage of having cached network namespaces. In this case, the network namespace provider keeps track of the created network namespaces in a thread-safe queue. New namespaces are added to the queue, and whenever an invocation needs a network namespace, a network namespace is pulled from the queue. There is a command running periodically that manages the network namespaces. When it runs, it ensures there are at least 40 network namespaces and that there are no more than 800 network namespaces. So, if an invocation needs a network namespace and there is not one available, it waits for this command to run and create new network namespaces.

To reduce the overhead of creating network namespaces in the middle of the invocations, before launching the Graalvisor webserver that handles function registration and invocation, network namespaces are created. The number of network namespaces created is configurable. After creating the network namespaces, the command that manages them is set to run periodically and then the webserver is launched.

When invoking a function without isolate caching, we get an available network namespace and switch to that network namespace before actually invoking the function. After the invocation, we switch back to

the default network namespace, free the network namespace, and add it back to the queue so it can be reused later.

In the Native Image Isolates cached option, every time we create an isolate, we start by getting an available network namespace, and this network namespace gets attached to the isolate. We switch to the network namespace we just got and then we perform requests in this cached isolate. After some time in idle, this isolate will be destroyed, but before that, we switch to the default network namespace, free the network namespace attached to the isolate, and add it back to the network namespaces queue.

5.3.2.C Network isolation configuration

The network isolation is configurable. It can be enabled or disabled with the `lambda_network_isolation` environment variable, and, as stated in the previous subsection, the number of network namespaces created on Graalvisor initialization is configurable with the `lambda_initial_network_namespaces` environment variable.

6

Evaluation

Contents

In this chapter, we will present how we measured the impact of adding network isolation to the Graalvisor system. We will present the workloads we used, the metrics we have extracted, what are the experiments we ran, and finally the results.

6.1 Workloads

We chose three workloads that were used in Photons: a simple Hello World function, a File Hashing function, and a Video Processing function. These workloads represent different parts of the application spectrum in terms of resource utilization.

- **Hello World function:** A simple function that returns the String ”Hello World” in the output;

- **File Hashing function:** A function that receives as argument a URL, pointing to a file. Using the Java MessageDigest class, it hashes the file using the MD5 algorithm and returns the hash in hexadecimal base. This workload is mixed on resource utilization since it downloads an input file and then uses CPU in order to hash the file;
- **Video Processing function:** A function that receives as arguments 2 URLs, one pointing to an mp4 video, and one pointing to a ffmpeg file. It uses the FFmpeg library in order to reduce the resolution of the video. This workload is CPU-bound.

6.2 Metrics

The metrics we extracted in order to evaluate our system with the network namespaces integration are latency, throughput, and memory footprint.

- **Latency:** We want to measure the latency in order to see if the use of network namespaces delays in any kind the execution time of a function;
- **Throughput:** We measure throughput to see how the function's requests per second are affected using network namespaces;
- **Memory footprint:** We also measure the memory footprint in order to see if using network namespaces introduces high memory usage.

6.3 Experiments

We ran three experiments with Native Image Isolates cache in order to evaluate our system. The experiments were run the workloads in:

a) A system without network isolation;
   - We want to measure the latency time on a system without network isolation to serve as a base to compare to the other experiments and see if they add any overhead;

b) A system with network isolation and network namespaces cache;
   - This is the second iteration of the implementation, so used it as an experiment to compare it with the baseline;

c) A system with network isolation and without network namespaces cache;
   - As one of the implementation iterations was creating a network namespace for each request, we used it as an experiment in order to compare it with the baseline experiment.

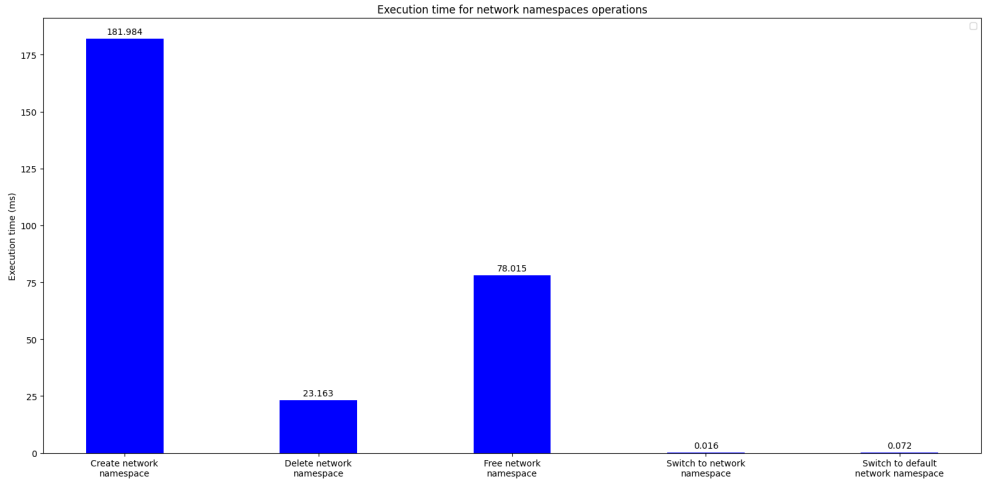

Figure 6.1: Average execution time (ms) for each network namespace operation.

6.4 Evaluation environment

The experiments were conducted in a Linux Ubuntu 22.04.3 LTS machine, with 16GB of RAM and the 11th generation Intel i7-1165G7 processor with 8 cores and 2.80GHz. Results were processed in different ways. The latency is the average latency of all requests performed, the throughput is the total number of requests divided by the benchmark total execution time, and the memory footprint is the average of the last ten records of memory utilization. The experiments were executed in different levels of concurrency in order to see how concurrency affects the use of network namespaces.

6.5 Results

In this section, we will present the results of the network namespace operations, followed by the latency, throughput, and memory footprint overheads on the different workloads. The Video Processing workload is a lot heavier than the other two, so the results on this workload were performed with concurrencies 1 and 2 instead of 1, 2, 4, and 6 like in the other workloads. All these experiments were done using cached Native Image Isolates.

6.5.1 Network namespace operations overhead

We started by measuring the execution time of network namespace operations such as creating a network namespace, deleting a network namespace, freeing a network namespace, switching to a network namespace, and switching to the default network namespace. We can see this represented in Figure

6.1.

Creating a network namespace takes around 180 milliseconds, which is a huge number and adds overhead to functions that perform really low latency operations.

Despite not having a number as high as the create operation, the delete network namespace operation still has a significant execution time of around 23 milliseconds. This execution time can add overhead to the same functions that the create operation affects.

The free network namespace operation also has a considerable execution time of around 78 milliseconds. It is still a high execution time and can also affect the same functions of the previous operations.

Both switch to network namespaces operations have less than 0.1 milliseconds of execution time, which adds minimal overhead to the functions' latency.

6.5.2 Latency overhead

After measuring the network namespace operations execution time, we ran all the experiments in the different workloads and extracted the average latency of a request. These values are presented in Figures 6.2, 6.3, and 6.4.

6.5.2.A Latency overhead in the Hello World workload

In the Hello World workload, there is almost no overhead when comparing the experiment without network isolation with the experiment with network isolation and network namespaces cache. This minimal overhead happens because we are using network namespace caching (network namespaces were previously created). A network namespace gets attached to a cached isolate, and the only network namespace operations performed are the switches between network namespaces, which have a low execution time.

On the other hand, the experiment with network isolation and no network namespaces cache has really high latency compared to the other experiments. It can be explained due to the fact that, in this experiment, at every invocation, a new network namespace is created, both switches between network namespaces are performed, and, in the end, the network namespace is deleted, and these execution times are really high compared to the Hello World function's execution time. If we sum these operations' execution times from Figure 6.1, we get close to the latency overhead at concurrency 1. As the concurrency gets higher, the overhead increases, this is due to Kernel locks performed in network namespace operations which actively increases the execution time.

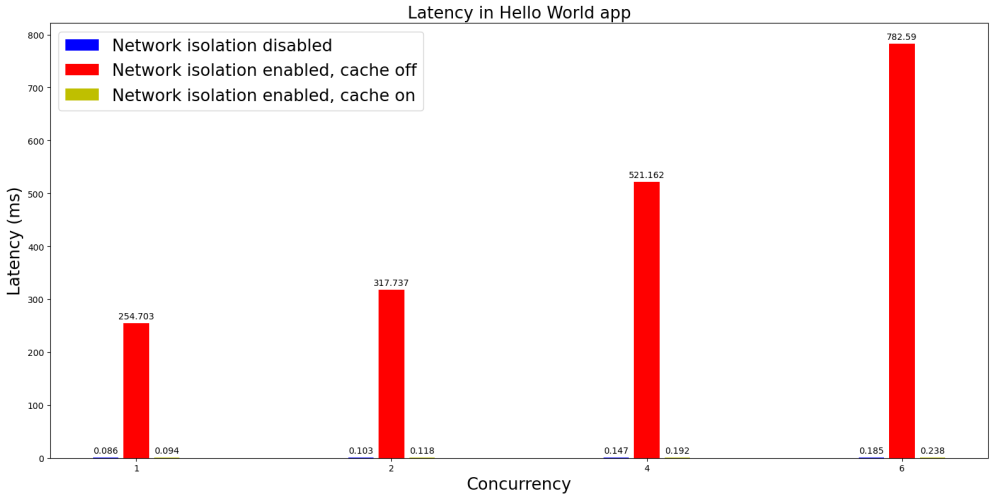

Figure 6.2: Average latency (ms) per request on all experiments for the Hello World workload.

6.5.2.B Latency overhead in the File Hashing workload

Similar to Hello World, the File Hashing workload shows the same minimal overhead in the experiment without network isolation and the experiment with network isolation and network namespaces cache, and the overhead added in the experiment with network isolation and no network namespaces cache as the Figure 6.3 shows. These overheads are similar for the same reason as the Hello World workload, the integration of cached network namespaces in cached isolates for the experiment with network isolation and cached network namespaces, and the overhead added by the network namespace operations in the experiment without network isolation.

6.5.2.C Latency overhead in the Video Processing workload

As we can see in Figure 6.4, this workload is a lot heavier than the others. The latency overhead in this case is a lot smaller since the function invocation latency itself is now much higher than the overhead brought by the network namespace operations. However, we can see that the network namespace operations in the experiment with no network isolation still add an overhead compared to the other experiments.

6.5.3 Throughput overhead

In this section, we measure how the network isolation impacts the throughput of the workloads, i.e. the number of requests per second.

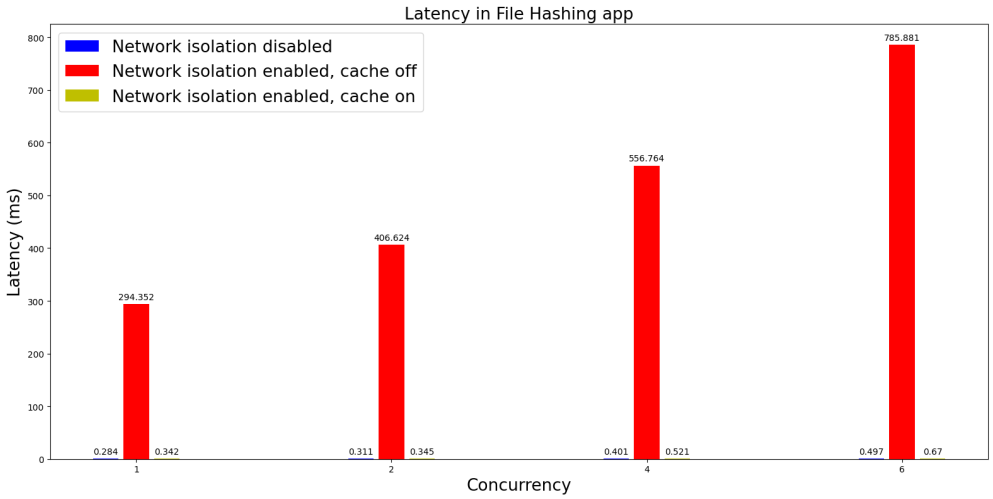

Figure 6.3: Average latency (ms) per request on all experiments for the File Hashing workload.

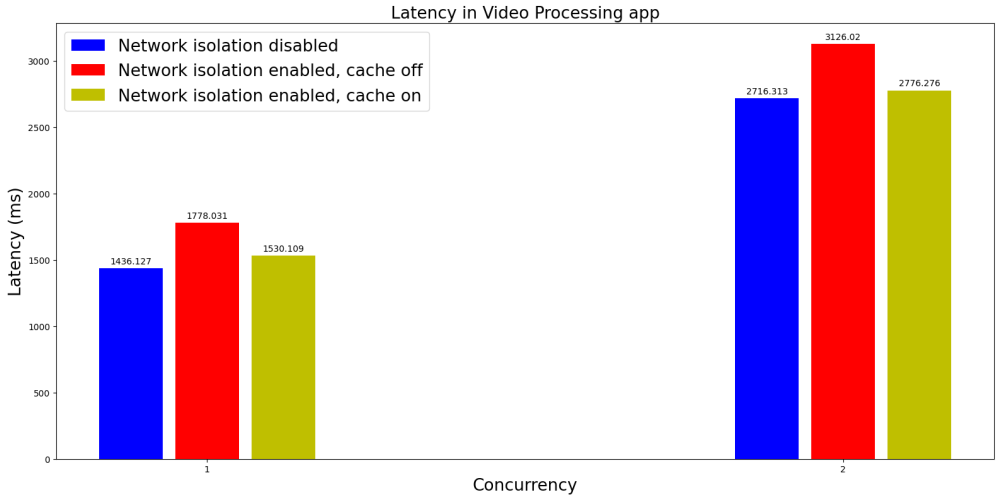

Figure 6.4: Average latency (ms) per request on all experiments for the Video Processing workload.

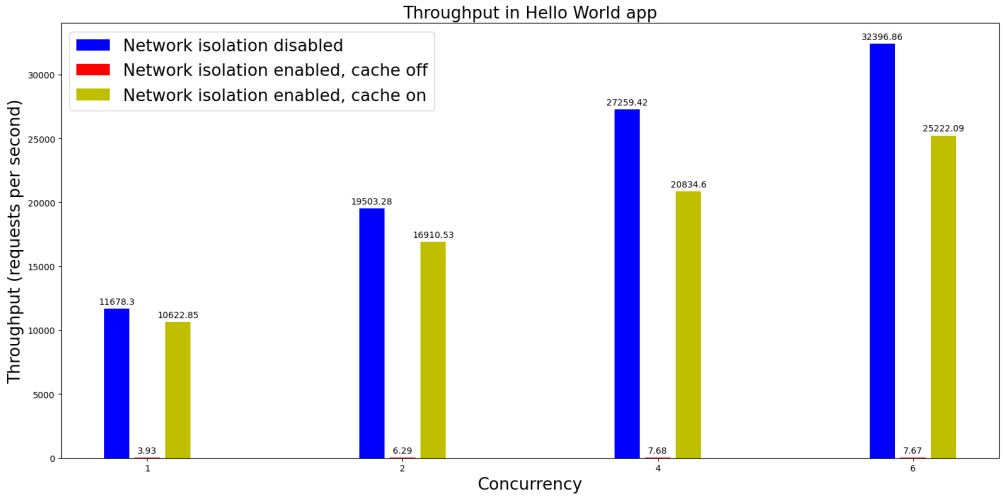

Figure 6.5: Throughput (requests per second) on all experiments for the Hello World workload.

6.5.3.A Throughput overhead in the Hello World workload

As Figure 6.5 shows, introducing network isolation with cached network namespaces added minimal overhead to the throughput. On the other hand, having network isolation without cached network namespaces added a huge overhead. As already explained in the latency section, this happens because the network namespace operations, such as the create and delete, have a high execution time compared to the Hello World function's execution time.

6.5.3.B Throughput overhead in the File Hashing workload

The throughput overhead in the File Hashing workload is very similar to the Hello World workload throughput, as it can seen when comparing Figures 6.5 and 6.6. The Hello World function and the File Hashing function have similar execution times, so it is expected that the overhead of adding network namespaces is very similar on both functions.

6.5.3.C Throughput overhead in the Video Processing workload

In the Video Processing workload, the throughput overhead is not so evident. As we can see in Figure 6.7, throughput values are very alike compared to the other experiments. In contrast with the Hello World and File Hashing functions, the overhead of performing the network namespace operations with the highest execution times in the Video Processing function becomes almost absorbed because the function's execution time is around seven times higher than the network namespaces operations altogether in the experiment with network isolation without cached network namespaces.

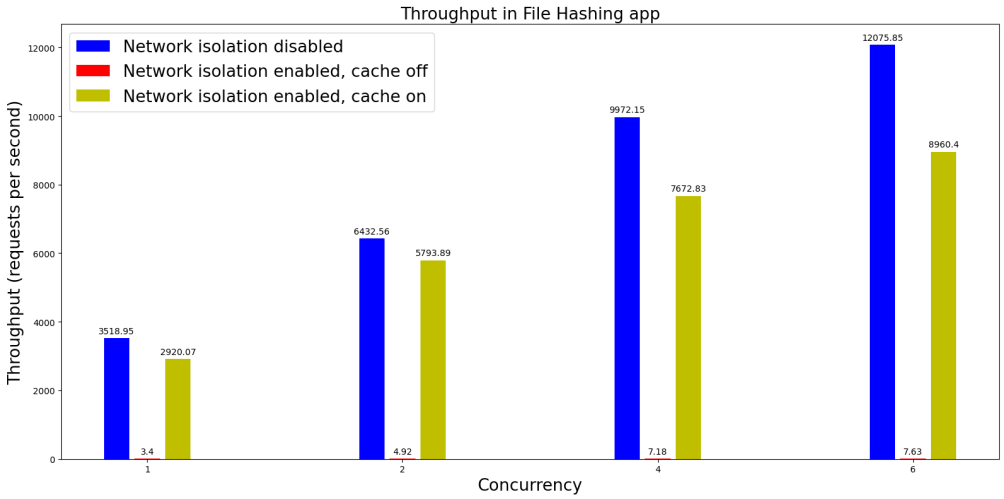

Figure 6.6: Throughput (requests per second) on all experiments for the File Hashing workload.

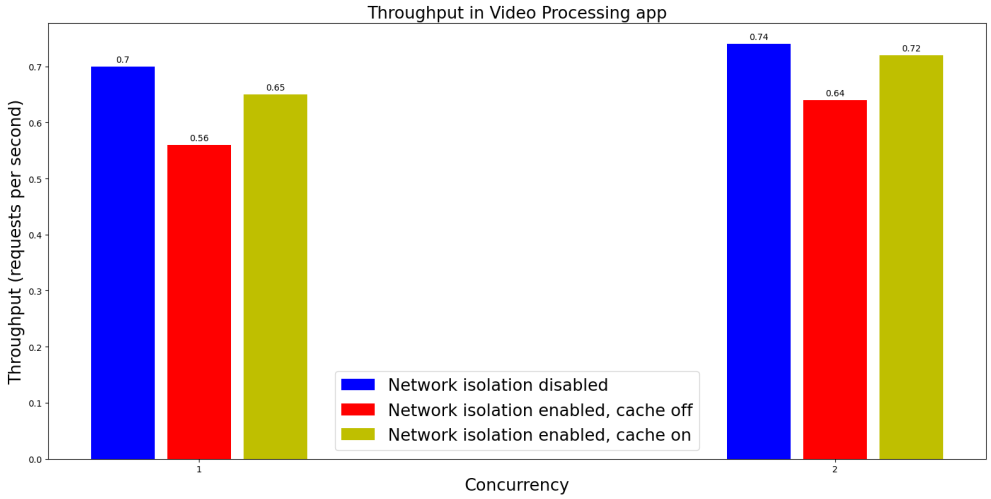

Figure 6.7: Throughput (requests per second) on all experiments for the Video Processing workload.

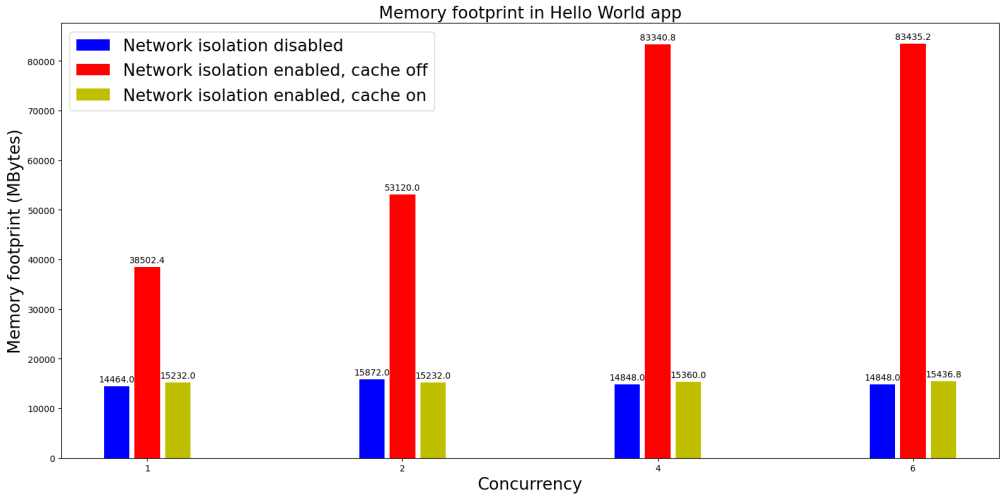

Figure 6.8: Memory footprint (MB) on all experiments for the Hello World workload.

6.5.4 Memory footprint overhead

The memory footprint overhead in the different workloads can be seen in Figures 6.8, **??**, and **??**. It is common in all the workloads having a higher memory usage in the experiment with network isolation and without cached network namespaces. This is also expected, because creating a network namespace will involve create a network namespace object in our code and a new object is created at every invocation.

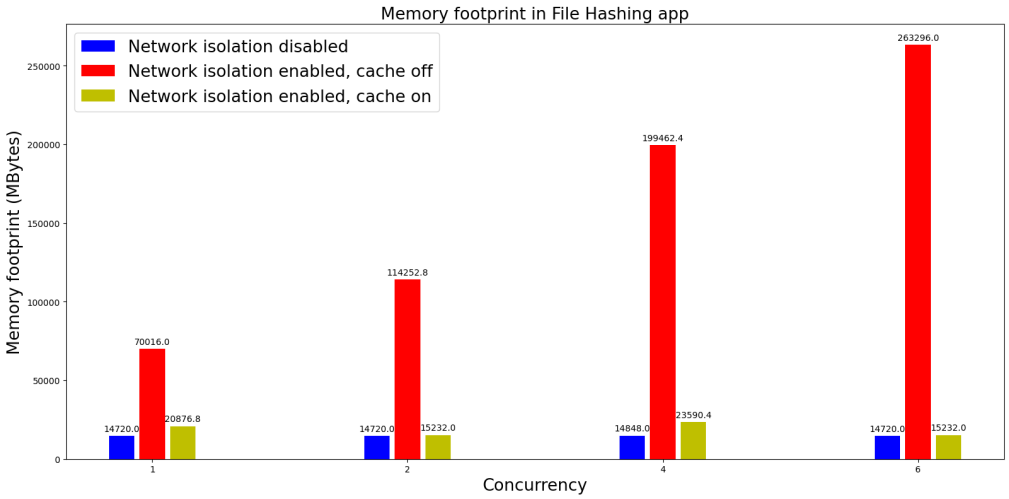

Figure 6.9: Memory footprint (ms) on all experiments for the File Hashing workload.

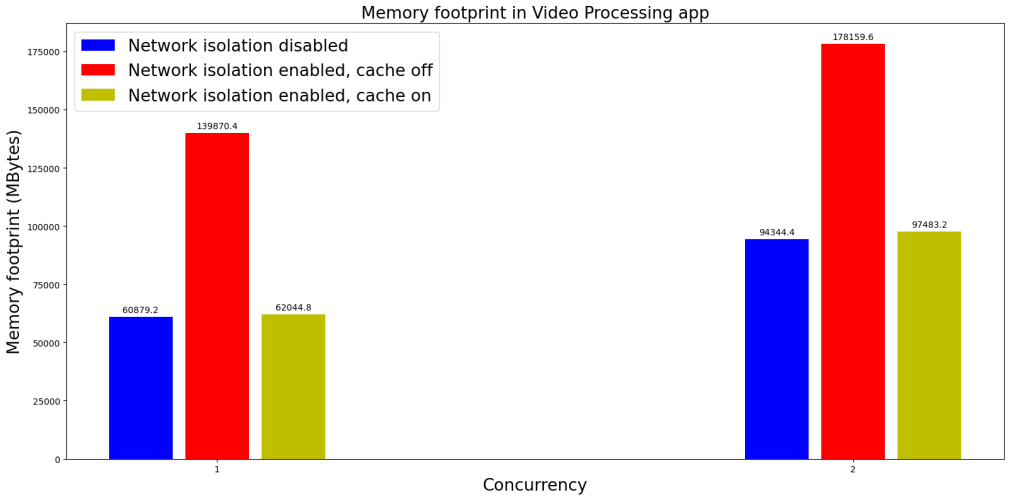

Figure 6.10: Memory footprint (MB) on all experiments for the Video Processing workload.

7

Conclusion

In this project, we aimed to virtualize the network of Function-as-a-Service applications that run in the same runtime environment without adding any kind of performance issues.

In order to achieve this goal, we used network namespaces in order to isolate the network of different function invocations. We also took advantage of integrating the network namespaces in Graalvisor project, which is built to serve function invocations. It already has the advantage of using Isolates to isolate the memory heap and also the advantage of using JNI to be able to run native code.

Our project was tested by running 3 different workloads in 3 different experiments with different levels of concurrency. From this testing, we collected some metrics such as latency, throughput, and memory footprint.

We can conclude that, from the two iterations proposed in the implementation section, the most viable is the implementation that user network namespaces cache. The first iteration, the one that has no network namespaces cache, is not viable for workloads where the request latency is very low. On the other hand, for workloads with higher latency, the overhead is less visible. In terms of throughput, it is similar to the latency comparison. In terms of memory, it uses a lot more memory which makes it not viable as well. The second iteration, the one that has network namespaces cache, is viable for both low and high-latency workloads. It also adds almost no overhead in terms of throughput, and the memory utilization is basically identical.

We could not find any programmatic interface to manage network namespaces, so, in future work, we will try to create one programmatic interface that will help us with network namespace management. We also could not build a Java Shutdown Hook in order to delete the network namespaces when the application ends, so, in future work, we will also work on that. As a more complex future work, we will try to find a way to share network namespaces across different functions and also find a way to clean network namespaces efficiently, since this implementation adds significant overhead to functions that perform low latency operations.

Bibliography

[1] M. Alzayat, J. Mace, P. Druschel, and D. Garg, “Groundhog: Efficient request isolation in faas,” 2022.

[2] J. Carreira, S. Kohli, R. Bruno, and P. Fonseca, “From warm to hot starts: Leveraging runtimes for the serverless era,” in *Proceedings of the Workshop on Hot Topics in Operating Systems*, 2021, pp. 58–64.

[3] V. Dukic, R. Bruno, A. Singla, and G. Alonso, “Photons: Lambdas on a diet,” in *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020, pp. 45–59.

[4] P. Gackstatter, P. A. Frangoudis, and S. Dustdar, “Pushing serverless to the edge with webassembly runtimes,” in *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 2022, pp. 140–149.

[5] Z. Jia and E. Witchel, “Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices,” in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 152–166.

[6] A. Fuerst and P. Sharma, “Faascache: keeping serverless computing alive with greedy-dual caching,” in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 386–400.

[7] Z. Li, J. Cheng, Q. Chen, E. Guan, Z. Bian, Y. Tao, B. Zha, Q. Wang, W. Han, and M. Guo, “RunD: A lightweight secure container runtime for high-density deployment and high-concurrency startup in serverless computing,” in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 53–68. [Online]. Available: https://www.usenix.org/conference/atc22/presentation/li-zijun-rund

[8] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou, “An open-source

benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 3–18. [Online]. Available: https://doi.org/10.1145/3297858.3304013

[9] GraalVM, "Argo," https://github.com/graalvm/argo.

[10] M. Shahrad, R. Fonseca, Í. Goiri, G. Chaudhry, P. Batum, J. Cooke, E. Laureano, C. Tresness, M. Russinovich, and R. Bianchini, "Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider," in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, 2020, pp. 205–218.

[11] "What is virtualization?" [Online]. Available: https://www.ibm.com/cloud/learn/virtualization-a-complete-guide

[12] "What is a virtual machine (vm)?" [Online]. Available: https://www.redhat.com/en/topics/virtualization/what-is-a-virtual-machine

[13] "What is a container?" Dec 2022. [Online]. Available: https://www.docker.com/resources/what-container/

[14] "What are containers?" [Online]. Available: https://www.ibm.com/topics/containers

[15] S. J. Bigelow and S. Shea, "What is a micro vm?" Dec 2021. [Online]. Available: https://www.techtarget.com/searchsecurity/definition/micro-VM-micro-virtual-machine

[16] "Micro virtual machine definition - glossary," Sep 2022. [Online]. Available: https://nordvpn.com/cybersecurity/glossary/micro-virtual-machine/

[17] [Online]. Available: https://www.koyeb.com/blog/what-is-a-microvm

[18] M. Boisvert, S. J. Bigelow, and W. Chai, "What is iaas? infrastructure as a service definition: Techtarget," Nov 2022. [Online]. Available: https://www.techtarget.com/searchcloudcomputing/definition/Infrastructure-as-a-Service-IaaS

[19] "What is paas? platform as a service: Microsoft azure." [Online]. Available: https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-paas/

[20] B. I. C. Education, "Faas." [Online]. Available: https://www.ibm.com/se-en/cloud/learn/faas

[21] "What is serverless computing?" [Online]. Available: https://www.cloudflare.com/learning/serverless/what-is-serverless/

[22] “What is saas? software as a service: Microsoft azure.” [Online]. Available: https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-saas/

[23] C. Wimmer, “Isolates and compressed references: More flexible and efficient memory management for graalvm...” Jun 2019. [Online]. Available: https://medium.com/graalvm/isolates-and-compressed-references-more-flexible-and-efficient-memory-management-for-graalvm-a044cc50b67

[24] D. Both, “An introduction to linux network routing.” [Online]. Available: https://opensource.com/business/16/8/introduction-linux-network-routing

[25] “Webassembly.” [Online]. Available: https://webassembly.org/

[26] R. Bruno, S. Ivanenko, S. Wang, J. Stevanovic, and V. Jovanovic, “Graalvisor: Virtualized polyglot runtime for serverless applications,” 2022.

[27] “Isolates javadoc.” [Online]. Available: https://www.graalvm.org/sdk/javadoc/org/graalvm/nativeimage/Isolates.html