FH MÜNSTER
University of Applied Sciences

Claus Grewe (Hrsg.)

Abschlussbericht FEP 2022

FH Münster – Master Wirtschaftsinformatik

23. Dezember 2022

Münster

Vorwort

Das Curriculum des Studiengangs Master of Science Wirtschaftsinformatik an der FH Münster beinhaltet das Wahlpflichtmodul „Forschungs- und Entwicklungsprojekt“. Das Modul ermöglicht es Studierenden, sich forschungsorientiert mit innovativen Themen auseinanderzusetzen. Die behandelten Themen werden von Jahr zu Jahr neu gewählt und umfassen sowohl aktuelle Forschungsgebiete als auch Innovationen, die den IT-Markt bzw. die Wirtschaftsinformatik gerade erreichen bzw. noch nicht durchdrungen haben.

Das Forschungs- und Entwicklungsprojekt im Sommersemester 2022 befasste sich inhaltlich mit drei Themenschwerpunkten:

- Leistungsvergleiche von Programmiersprachen, Werkzeugen sowie Plattformen zur serverseitigen Ausführung von WebAssembly-Anwendungen
- Leistungsvergleiche der Programmiersprachen Rust und Go hinsichtlich der Unterstützung von Nebenläufigkeit sowie Rust und Python hinsichtlich der Verarbeitung von Graphalgorithmen
- Circuit-Breaker- und Proxy-Ansätze in Service-Mesh-Architekturen

Das Projektteam umfasste die sechs Mitglieder Jan Föcking, Jonas Brösterhaus, Marcel Niehüsener, Kai-Luka Buchkremer, Lennart Potthoff und Andre Farwick. Die Ergebnisse der Untersuchungen wurden in Form eigenständiger Beiträge verfasst und in diesem Abschlussbericht zusammengetragen.

Münster, im Dezember 2022

Prof. Dr. Claus Grewe

Inhaltsverzeichnis

Performance Comparison of WebAssembly Source Languages and Compilers for Running WebAssembly on the Server

Jan Föcking[1]

Abstract: WebAssembly (Wasm), a portable low-level bytecode format initially designed for targeting Web browsers, is experiencing an increasing adoption in server-side environments. Typically, a WebAssembly binary originates by the compilation of source code from one of many source languages that support Wasm as a compilation target. Unfortunately, insufficient study has been conducted to understand how the choice of a particular source language and compiler impacts the performance of the resulting Wasm binary. In this paper, we compare the performance of Wasm binaries by implementing three different workloads in different source languages, compile them to WebAssembly, and execute them in a standalone Wasm runtime on the server side. We collect several performance metrics and compare them with respect to the different source languages.

Keywords: WebAssembly; WASI; Performance comparison

1 Introduction

WebAssembly (Wasm) is a safe, fast, compact and portable low-level bytecode format. It was initially designed by the four major browser vendors to address restrictions in the standard client-side language JavaScript. Since 2017, all major browsers are supporting this standard. While the initial paper targets the Web as the primary target at first, it also mentions the existence of use cases beyond the Web due to the language-, hardware-, and platform-independence of the bytecode format [Ha17].

Since WebAssembly is an open standard with no Web-specific features or assumptions, the development of standalone runtimes was expected from the beginning [Ha17; Ro19]. In particular, security features (sandboxing and memory-safety), a near-native performance, and characteristics like compactness and portability of the bytecode format make it suitable for use on the server side. However, before the WebAssembly System Interface (WASI) was introduced in 2019, no standardized way was available to perform system calls. The introduction of WASI (including APIs for file I/O access, clocks, random) made it even more attractive to use WebAssembly on the server side [Cl19].

Initially, the focus was on supporting low-level languages, especially C and C++, for targeting the Wasm bytecode format [Ha17]. Today, there are many upcoming use cases

[1] FH Münster, Fachbereich Wirtschaftsinformatik, Wirtschaftsinformatik, jf935878@fh-muenster.de

for WebAssembly, also on the server side, and multiple languages and compilers are able to target Wasm. However, no research is available that investigates the impact of different source languages and compilers to the performance of WebAssembly. This paper is, to our knowledge, the first one that evaluates the performance of different Wasm applications compiled from various source languages and executed in a non-Web, standalone Wasm runtime. Our goal is to propose which source languages should be used for building performant, Wasm based applications for execution in server-side environments.

In the following, section 2 describes backgrounds on Wasm in standalone environments and characterizes related work. Section 3 explains the experimental design to evaluate Wasm performance for different compilers, and section 4 presents and discusses the measurement results. Finally, in section 5 we present our conclusions and outline future work.

2 Background

2.1 WebAssembly basics

Despite its name, WebAssembly is not assembly code, but a safe and portable byte code format for efficient execution and compact representation. Although firstly implemented by Web browsers, Wasm's virtual instruction set architecture is designed to be embedded in other non-browser environments as well [Ro19]. According to the WebAssembly specification, a Wasm binary has the form of a so-called module that consists of multiple components such as functions, instructions, values and linear memory [Ro19]. Values can only be one of four numeric types, either integers (32 and 64 bit), or floating-point numbers. Functions organize code by taking values as parameters and returning a sequence of values as results. Functions contain code in the form of a sequence of instructions that are executed in order based on a stack machine. A Wasm module can load and store values by calling the linear memory, a mutable array of raw bytes, which can grow dynamically. WebAssembly is typically embedded into a host environment, typically a browser or a standalone Wasm runtime. A Wasm module can import functions that are provided by the host environment, the so-called host functions. In addition, a module exports arbitrary functions, which can then be called by the host environment.

For security reasons, WebAssembly does not provide any access to the host environment by default [Ro19]. Consequently, access to resources (like I/O or other operating system calls) can only be achieved if the host environment provides dedicated host functions for these purposes. This sandboxing concept enables the host environment with complete control about the capabilities it offers for Wasm modules. An approach to standardize the set of system-oriented host functions, which can be imported into a Wasm module, is WASI, a runtime-independent, non-Web, and system-oriented API for WebAssembly [Cl19].

Although initially seen as a replacement for JavaScript in the Web browser, Wasm is becoming increasingly popular for non-Browser use cases. Spies et al. [SM21] provide

an overview of some of those use cases, including TruffleWasm, an implementation of WebAssembly running on GraalVm that enables Wasm for use in polyglot programming. Another use case referenced by Spies et al. [SM21] is the execution of Wasm-based Smart Contracts on the Ethereum Blockchain, also known as Ethereum flavoured Wasm. Additionally, WebAssembly is especially suitable for applications requiring low-latency response, or hardware platforms with limited resources, for example in edge computing environments [HR19].

2.2 Runtimes and compilers

Multiple runtimes are available to execute WebAssembly workloads in non-Web environments. Wasmtime is a standalone runtime for WebAssembly and WASI maintained by the Bytecode Alliance. Other well-known standalone Wasm runtimes with WASI support are Wasmer, Wasm3 and WasmEdge.[2]

Support for compiling to WebAssembly is available in many programming languages, compiler toolchains and frameworks. In general, two different strategies can be identified. First, many compiler toolchains compile source code into native Wasm binaries. Second, for some other languages, their language runtime/interpreter is compiled into a Wasm binary in order to interpret ordinary programs written in that language at runtime. Some compiler toolchains are not suitable for compilation to WebAssembly if the resulting binaries are to run in a non-Web, standalone runtime. This is due to the fact that some toolchains assume the resulting Wasm binaries to be executed in a browser environment; these toolchains produce JavaScript glue code and make the resulting Wasm binaries depend on that code. Tab. 1 provides a classification of some well-known compiler toolchains[3].

2.3 Related work

Our work is closely related to other WebAssembly performance measurements. There have been prior studies evaluating the performance of WebAssembly in browser environments. Haas et al. [Ha17] compare the performance of WebAssembly to asm.js and native code by using PolyBenchC, a benchmark suite written in C. Yan et al. [Ya21] provide an understanding regarding the performance of Wasm applications and the impact of different optimization levels of a C-to-WebAssembly-compiler and give an overview of performance measurements prior to their work; this includes comparisons based on common C benchmark suites, PolyBenchC and the SPEC CPU suite, and an evaluation of the performance of WebAssembly based on a Sparse matrix-vector multiplication workload.

There has also been research regarding WebAssembly's performance in non-Web environments. Spies et al. [SM21] measure execution time, startup time, and resulting binary

[2] Overview of Wasm runtimes: https://github.com/appcypher/awesome-wasm-runtimes
[3] Overview of Wasm compilers: https://github.com/appcypher/awesome-wasm-langs

Compiler toolchain	Source language	Compilation strategy	Non-browser support	WASI support
AssemblyScript	TypeScript	Native Wasm binary	Yes	Yes
Blazor Wasm	C#	Language runtime/ interpreter in Wasm	No	No
CRuby	Ruby	Language runtime/ interpreter in Wasm	Yes	Yes
Emscripten	C/C++	Native Wasm binary	Yes	Partial
Go compiler	Go	Native Wasm binary	No	No
Pyodide	Python	Language runtime/ interpreter in Wasm	No	No
Rust compiler	Rust	Native Wasm binary	Yes	Yes
TinyGo	Go	Native Wasm binary	Yes	Yes
WASI SDK	C/C++	Native Wasm binary	Yes	Yes

Tab. 1: Classification of well-known compiler toolchains

size in different runtimes by using the PolyBenchC benchmark suite and the Clang and Emscripten compilers. There are a few more specialized studies evaluating the performance of WebAssembly on the ARM architecture [Me20], or in edge or IoT environments [HR19].

3 Experimental design

In this section, we outline our experimental setup that is made publicly available on GitHub[4]. In general, we performed our measurements in two phases that are shown in Fig. 1. First, we compiled the source code of our workload implementations into WebAssembly binaries. Afterwards, we executed the compiled binaries in Wasmtime (version 0.37), a standalone Wasm runtime, and collected several metrics about the runtime behavior of each binary.

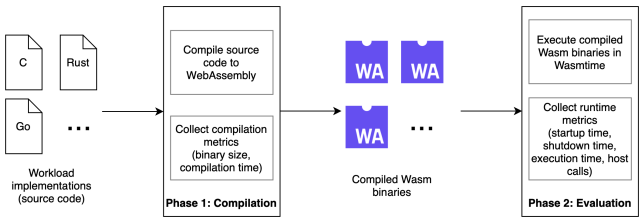

Fig. 1: Overview of the measurement process

As the goal of our measurements is to compare the performance with respect to different source languages and compilers, we used the following compilers from Tab. 1 that support non-browser runtimes and WASI: AssemblyScript (version 0.20.6), CRuby (based on version

[4] https://github.com/jfoe98/comparison-of-wasm-source-languages-for-running-on-the-server

3.2.0 Preview 1), Emscripten (version 3.1.13), Rust compiler (version 1.61.0), TinyGo (version 0.23.0), and WASI SDK (version 16.0 + Clang version 14.0.4). For each compiler setup, we compiled each workload twice, without and with full compiler optimizations enabled, to compare the efficiency of the optimizations.

3.1 Metrics

The first class of metrics, the development metrics, allow us to come to conclusions about the development process. Therefore, the compilation time and the resulting binary size were measured during the compilation process for each source language.

Second, we collected metrics about the runtime behavior of the compiled Wasm binaries. The main metric regarding the performance is the time taken for execution of the Wasm binary. However, since WebAssembly is often used for implementing small and ephemeral applications (like serverless functions in the cloud), the startup (e. g. the time for VM startup, code validation and module initialization) and shutdown time are important and were therefore included in our measurements. Additionally, we captured the amount and kind of WASI system calls to come to conclusions about the efficiency of the I/O implementations of different source languages and compilers.

3.2 Workload selection

To compare the source languages concerning various performance aspects, we implemented three programs, one stack-intensive, one compute-intensive, and one I/O-file-intensive application.

The first application, our stack-intensive program, is a recursive implementation of the Fibonacci sequence. For the compute-intensive workload, we implemented an iterative version of Fibonacci in all source languages. The third workload is intended to compare the I/O capabilities of the different source languages and compilers. As input, a file, containing random numbers from 1 to 100, is provided. The workload consists of reading the numbers from the given input file line by line and appending each number to another file based on the last digit of the number. If the file does not exist, it has to be created first. The output is a set of files, where each file contains all numbers from the given input file ending with the same digit.

We implemented each workload in each of the previously considered source languages. To establish comparability between the implementations, we backed our implementations on the examples from Rosetta Code [5]; for example, our I/O-intensive workload is based on two tasks, `Read a file line by line` and `Append a record to the end of a text file`.

[5] Rosetta Code: http://www.rosettacode.org/wiki/Category:Programming_Tasks

Since the input size of a program affects the amount of performed calculations, we used three input sizes for each workload: Small, Medium, and Large. Tab. 2 depicts the workloads' inputs for each size.

Workload Name	Input Small	Input Medium	Input Large
Fibonacci recursive (`fib-rec`)	5	30	50
Fibonacci iterative (`fib-it`)	5	50	90
Filesplit (`filesplit`)	1,000 lines	10,000 lines	100,000 lines

Tab. 2: Workload sizes for all workloads

3.3 Evaluation setup

Fig. 2 depicts our evaluation setup. We provisioned two virtual machines (VMs) (Ubuntu 22.04 LTS, 4 GB of RAM, Intel Xeon Gold 6150 CPU with one core and 2.7 GHz base frequency), one dedicated to perform the measurements, and the other one for managing the resulting metrics. We deployed multiple artifacts to the Measurement VM: The bash script `measurement.sh` is intended to start and orchestrate our measurements. As a first step, it triggers the compilation of the uncompiled source code to Wasm for all languages and compiler setups. Afterwards, it starts our evaluation tool in order to execute the previously compiled Wasm modules. The metrics that were collected during the compilation and execution of the Wasm binaries, were sent to a dedicated virtual machine, the Metrics VM, via HTTP. The Metrics VM was running two Docker containers, a Rust-based HTTP server we implemented to manage metrics, and a relational database to persist them. We separated out managing metrics into a dedicated VM to minimize overhead and to avoid disruptive impacts to our measurements.

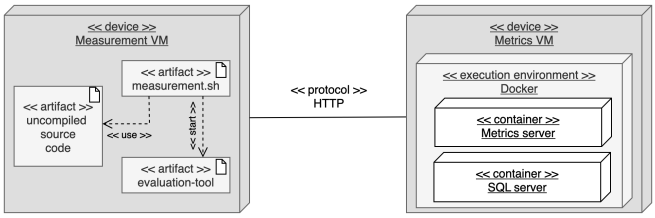

Fig. 2: Overview of the evaluation setup

3.4 Implementation details

3.4.1 Compiling code from source language to WebAssembly

In general, we identified two different approaches for compilation used by the different languages/compilers. On the one hand, most compilers compile to native Wasm binaries,

which can be executed directly by Wasm runtimes. On the other hand, within this study, Ruby with its CRuby runtime follows another approach, so that the CRuby runtime is compiled to WebAssembly. As a consequence, no compilation was necessary for all Ruby-based workload implementations, because the CRuby language runtime interprets ordinary Ruby files at runtime.

The compilation process was automated for each source language by using Docker containers. During the compilation process, the used time and the binary size were measured with the Linux standard tools `date` and `stat`. Each compilation was performed 15 times to achieve more significant results.

3.4.2 Evaluating WebAssembly binaries at runtime

In order to run the compiled Wasm modules and compare its runtime behaviors, we implemented our own evaluation tool in Rust based on the Wasm runtime wasmtime [6], and executed each workload 15 times in order to achieve significant results. The wasmtime Embedding API provides the ability to define and implement host functions that can be imported and called from within a Wasm module. We used this mechanism to define two host functions, `startup` and `finish`. The implementations of our Wasm workloads import the previously listed host functions and call them at the beginning and at the end of the execution. This allowed us to measure the time elapsed for the start and initialization of a module (i. e. the startup time), and the time needed for the termination (i. e. the shutdown time). This procedure is only possible for languages compiling into native Wasm binaries; consequently, we were not able to measure the startup and shutdown time for the CRuby approach. Additionally, we implemented Closures and registered them to the so-called `Call Hooks` of Wasmtime's Embedding API. Call hooks are triggered at certain events and we used them to start or stop our measurements when Wasmtime begins or terminates the execution of a module.

Another important aspect in our evaluation is to compare the efficiency of the WASI-based filesystem implementations between the different source languages. To capture the performed WASI calls, we performed a dedicated evaluation run, where we activated trace logging in order to make Wasmtime log all WASI calls. We collected metrics about the amount and types of performed WASI calls by implementing a tool for parsing the resulting log.

4 Results and discussion

In this section, we present and discuss our measurement results. Additional evaluations and plots are available on GitHub[7].

[6] Crate wasmtime: https://docs.rs/wasmtime/0.37.0/wasmtime/

[7] https://github.com/jfoe98/comparison-of-wasm-source-languages-for-running-on-the-server/blob/main/evaluation/Evaluation.ipynb

4.1 Compile time

In our measurements, WASI SDK and Emscripten stood out with compilation times near one second on average. However, TinyGo and AssemblyScript delivered solid results as well, providing compilation results in nearly 6.5 seconds. The Rust compiler took over two minutes on average for compilation, making it the slowest compiler by far. In our measurements, we did not find any significant impact of the different compiler optimization levels on the compile time; all compilers showed the same or even slightly better compilation performances with higher optimization levels enabled.

4.2 Binary size

As shown in Fig. 3, there were significant differences regarding the resulting binary sizes between the compared compilers. While AssemblyScript (mean binary size of 200 bytes) and Emscripten (600 bytes) produced very small binaries for the Fibonacci workloads, especially Rust's compilation process resulted in much larger binaries (1.9 mb). The source code of our I/O-intensive workload (i.e. `filesplit`) contains calls to system interfaces, like the filesystem. In this case, our results show that the binary size increased significantly because of a huge amount of WASI-specific bytecode added to the resulting Wasm binaries by the compilers. Regarding the different compiler optimization levels, TinyGo (binaries smaller by 55%), Emscripten (27%) and AssemblyScript (24%) produced significantly smaller binaries with high optimization levels enabled; Rust and WASI SDK showed no significant impact when varying the compiler optimization levels.

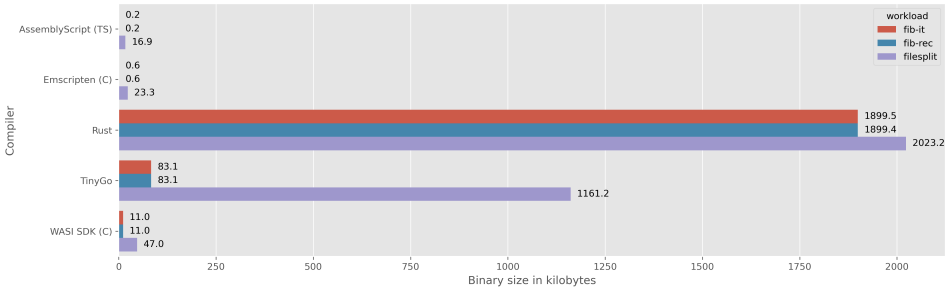

Fig. 3: Comparison of the resulting binary's size of different compilers

4.3 Startup and shutdown time

Fig. 4 shows the results of our measurements regarding the startup and shutdown time. In terms of startup time, the AssemblyScript compiler outperformed all other compilers with a median startup time beneath 1 µs for the Fibonacci workloads and 3 µs regarding the

I/O-intensive workload. Although the Fibonacci modules that were compiled from Go and Rust also delivered a median startup performance lower than 1 µs, the startup time increased massively when looking at the I/O-intensive workload (78 µs for TinyGo and 53 µs for Rust). The WASI SDK performed worst, since even the Fibonacci workloads showed a median startup time greater than 30 µs. An effect we could observe for all compilers is a significant increase of startup time when looking at the Filesplit workload due to larger binary size (as shown in Sect. 4.2) and additional WASI-specific initialization code that is executed during the initialization process of the Wasm module.

In comparison to the previously discussed startup performances, the shutdown of a Wasm module required less time across all compilers. For all compilers, the median shutdown time was under 1 µs for the iterative Fibonacci implementations. Looking at the recursive Fibonacci implementation, we could observe a significant increase in the median shutdown time and in the deviation as well. Due to the recursive behavior, especially for large input sizes, a huge amount of memory is allocated from the stack and needs to be freed when the shutdown occurs. Additionally, an increase of the shutdown time can be seen for the Filesplit workload, since additional system resources have to be released in this case as well.

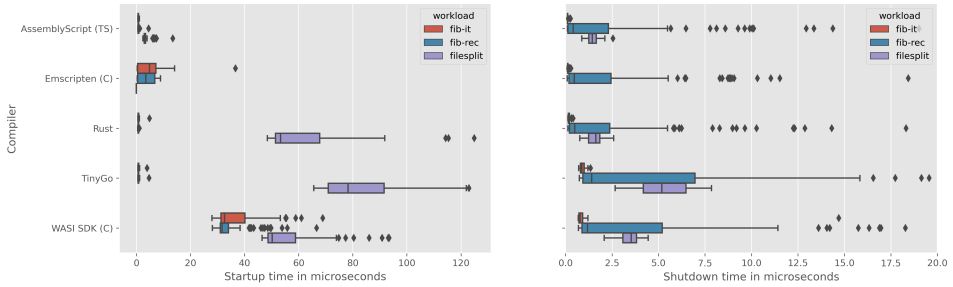

Fig. 4: Comparison of the startup and shutdown time of different compilers

4.4 Execution time

Since we assume high compiler optimizations to be used in production environments, Fig. 5 depicts the execution times we measured regarding the different input sizes for our three workloads that were compiled with full optimizations enabled. As Rust is the most used compiler in real-world use cases, the performance of the Rust-based Wasm binaries was chosen as the baseline and all other results are shown in relation to Rust's results.

Wasm binaries compiled by WASI SDK showed poor performance for workloads (`fib-it`, and `fib-rec` for small inputs) that are executed in the range of microseconds if compiled by other compilers, as WASI SDK's slow startup time (as shown in Sect. 4.3) had a significant effect here. In terms of the `fib-it` workload, the Emscripten-based binaries performed best

for all input sizes, but there were no big differences between the other compilers, except for TinyGo, which took 1.5 times longer. For the `fib-rec` workload, all compilers showed similar results, especially for the large input sizes. With regard to the `filesplit` workload, we identified significant differences between the compilers. While Rust-based binaries delivered the best results, binaries compiled by AssemblyScript and WASI SDK were 1.4 times slower. TinyGo-based binaries showed the poorest execution times by taking up to three times as much time as Rust-based binaries.

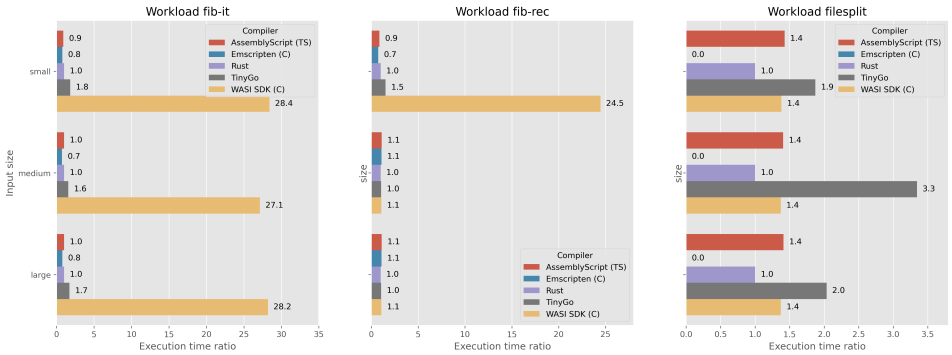

Fig. 5: Comparison of the median execution time for all workloads and input sizes

In Fig. 5, Ruby is not shown because its execution times showed such poor performance that the graph would have been distorted. While Ruby-based executions of the `filesplit` workload were 5 to 17 times slower compared to Rust-based Wasm binaries, the executions of the `fib-rec` workload were 68 times for large input sizes and 16,579 times slower for small input sizes. In median, running the `fib-it` workload took between 18,672 and 19,666 times longer than running the Rust-based binaries for different input sizes.

Depending on the compiler, the optimization level used during compilation had a significant impact on the execution time. On average, the median execution time was across all workloads and input sizes over 6.2 times for Emscripten and 3.9 times for Rust slower when compilation was performed without optimizations. WASI SDK (1.23x slower) and TinyGo (1.2x slower) showed smaller performance losses when deactivating compiler optimizations, while we could not determine any significant differences for AssemblyScript.

4.5 WASI calls

As shown in Tab. 3, the efficiency of the WASI-based implementations of the `filesplit` workload of different compilers differed. While TinyGo, Ruby and WASI SDK made excessive use of the call `fd_fdstat_get`, a call to get attributes of a file descriptor similar to POSIX's `fsync`, binaries compiled by other compilers did not execute this call at all. It is also noticeable that our AssemblyScript implementation called `fd_fdstat_set_flags` to set the append-flag before each write operation. Additionally, while all other implementations

buffered the input file while reading it line by line, binaries compiled with AssemblyScript seemed to be more inefficient and called `fd_read` multiple times for each line. According to the `fd_write` call, Rust-compiled implementations performed two write operations per line.

WASI call	AssemblyScript	Ruby	Rust	TinyGo	WASI SDK
fd_close	100,001	100,002	100,001	100,001	100,001
fd_fdstat_get	0	299,805	0	100,001	300,001
fd_fdstat_set_flags	100,000	1	0	0	0
fd_read	290,901	39	37	73	286
fd_write	100,000	100,003	200,000	100,001	100,000
path_open	100,001	100,050	100,001	100,001	100,001

Tab. 3: Performed WASI calls for the workload filesplit and input size Large

4.6 Reliability

With one exception, all implementations across all compilers always delivered correct results. The exception is the `filesplit` workload in combination with the Emscripten compiler, as it can also be seen in Fig. 5. The reason is that although Emscripten supports WASI in general, reading and writing to/from WASI file descriptors is only implemented for the default file descriptors STDOUT and STDERR at the time of writing[8].

4.7 Discussion

Tab. 4 summarizes our findings on the impact of different source languages and compilers on the performance of compiled Wasm binaries. In our evaluation, AssemblyScript performed well across all criteria and is therefore a good choice in many scenarios. Although Emscripten showed good results as well, it cannot be used for file-based workloads due to a limited WASI support. Due to its large feature set and very good runtime performance results, the use of Rust is recommended if a slow compilation time and a large binary size can be tolerated. TinyGo delivered solid performance results regarding all criteria. While WASI SDK provides a good development experience, its use is not suitable for short-lived, ephemeral tasks that are to be executed in a few microseconds due to the long startup times. The approach of compiling the CRuby language runtime to Wasm to interpret ordinary Ruby code in Wasm proved to be very inefficient and can therefore not be recommended.

5 Conclusions and future work

WebAssembly is experiencing an increasing adoption in server-side environments and more and more source languages and compilers support Wasm as a compilation target. In this

[8] https://github.com/emscripten-core/emscripten/issues/17167

	AssemblyScript	Emscripten	Ruby	Rust	TinyGo	WASI SDK
Compile time	+	++	n/a	--	+	++
Binary size	++	++	n/a	--	-	+
Startup / shutdown time	++	++	n/a	+	+/-	--
Execution time	+	++	--	++	+	-
Reliability	++	+/-	++	++	++	++

Tab. 4: Assessment of compilers regarding the compared performance criteria

paper, we gave an introduction into WebAssembly, presented an overview of use cases and compilers, and referenced related work. Afterwards, we compared the performance of Wasm binaries compiled from different source languages in a standalone, non-Web runtime. Significant differences were found between the compilers with respect to different criteria. We presented our results and made recommendations for the use of the different compilers and source languages.

In our study, we considered six source languages and three different workloads. In future work, the evaluation of additional source languages and compilers, and the measurement of additional, real-world workloads is a promising area.

References

[Cl19] Clark, L.: Standardizing WASI: A system interface to run WebAssembly outside the web, 2019, URL: https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/, visited on: 06/20/2022.

[Ha17] Haas, A. et al.: Bringing the web up to speed with WebAssembly. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM, pp. 185–200, 2017.

[HR19] Hall, A.; Ramachandran, U.: An execution model for serverless functions at the edge. In: Proceedings of the International Conference on Internet of Things Design and Implementation. ACM, 2019.

[Me20] Mendki, P.: Evaluating Webassembly Enabled Serverless Approach for Edge Computing. In: 2020 IEEE Cloud Summit. IEEE, 2020.

[Ro19] Rossberg, A.: WebAssembly Core Specification, https://www.w3.org/TR/wasm-core-1/, 2019.

[SM21] Spies, B.; Mock, M.: An Evaluation of WebAssembly in Non-Web Environments. In: 2021 XLVII Latin American Computing Conference (CLEI). IEEE, 2021.

[Ya21] Yan, Y. et al.: Understanding the performance of webassembly applications. In: Proceedings of the 21st ACM Internet Measurement Conference. ACM, 2021.

Structural Comparison and Performance Evaluation of Platforms and Technologies for Running WebAssembly on the Server

Jonas Brösterhaus[1]

Abstract: Since the release of WebAssembly System Interface (WASI) and standalone runtimes like Wasmtime, many platforms and technologies for running WebAssembly on the server have emerged. We compare these platforms and technologies regarding their structure and propose a simple structural classification system for such platforms and technologies. Furthermore, we evaluate their performance using computation-intensive and HTTP workloads. The results reveal large differences in startup and running time as well as HTTP response time and throughput.

Keywords: WebAssembly; structural classification; Kubernetes; application platform; server-side

1 Introduction: WebAssembly on the server

WebAssembly (frequently and henceforth abbreviated as Wasm) is a standardized bytecode format that was developed for bringing near-native performance to the web browser as an alternative to JavaScript and was proposed in [Ha17]. Similar to the fashion that JavaScript entered the server realm with runtimes like Node.js, Wasm is setting foot on the server with standalone runtimes like Wasmtime and Wasmer. This development is strongly facilitated by the WebAssembly System Interface (WASI) that shall provide access to operating system (OS) features like sockets, clocks, file systems, and random numbers [Go19]. The co-founder of Docker, Solomon Hykes, stated that "If WASM+WASI existed in 2008, we wouldn't have needed to created Docker."[2] This can demonstrate the potential and disruptivity of Wasm + WASI on the server. The WASI specification is however currently at an early stage of maturity and has not yet provided any standardization but only proposals.

Using Wasm on the server can provide several advantages. One main advantage is that each Wasm module is executed in a sandbox that can only interact with its environment via provided system interfaces. Another benefit is that Wasm modules are portable. They can be run in many different environments: on on-premises servers, on the cloud, or on the edge; in the browser, embedded in other applications, on application platforms, as serverless functions, or on standalone runtimes. Wasm modules can be developed in many different source languages including Rust, C, C++, Go, TypeScript, Elixir, and many more.

[1] FH Münster, Fachbereich Wirtschaft, Wirtschaftsinformatik, jonas.broesterhaus@fh-muenster.de
[2] https://twitter.com/solomonstre/status/1111004913222324225

In recent times, several different platforms and technologies for executing Wasm on the server have emerged. We have identified relevant platforms and technologies, and in the following sections, compare their structure and evaluate their performance. The platforms and technologies that are evaluated are: Standalone runtimes (Wasmer, Wasmtime) called via CLI or inside a Container, WasmEdge on Kubernetes, Krustlet, wasmCloud, and Vino. FaaS/serverless technologies are not part of this evaluation, as these have been assessed in other papers such as [Lo21].

2 Structural comparison

2.1 Platforms and technologies: architecture, component model, features

In this section, the platforms and technologies are characterized regarding their structure i. e. internal architecture, component model, and features. Their structure is additionally visualized in Fig. 1.

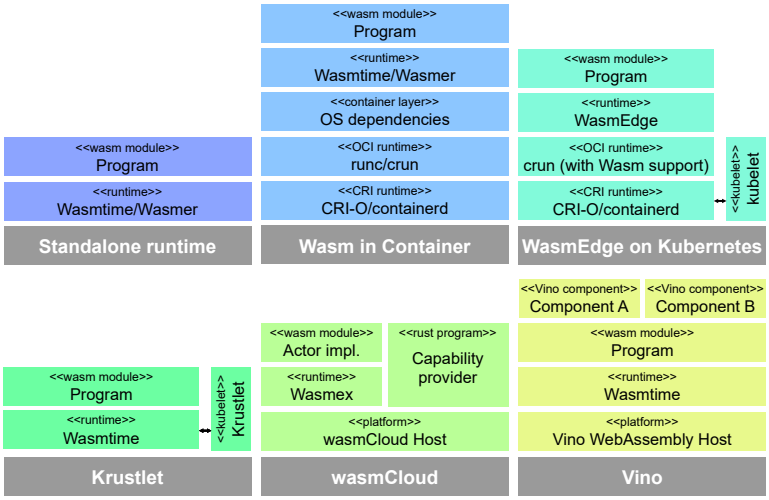

Fig. 1: Structural overview of Wasm execution on evaluated platforms and technologies

Standalone Runtimes (also inside of OS distribution[3]-based containers). The simplest approach to executing Wasm on the server is the usage of a standalone runtime like Wasmtime or Wasmer [By22]. These can be called using a command-line interface and can be used in containers that are based on Linux or other operating systems. These runtimes usually offer support for WASI but no support for network sockets.

WasmEdge on Kubernetes. Wasm modules can be executed on Kubernetes using the WasmEdge runtime [We22]. WasmEdge is supported by the high-level container runtimes

[3] As containers do not contain entire OSs but only userspace OS dependencies, the term *OS distribution* is used.

CRI-O and *containered* and by the low-level container runtime (also called OCI runtime) *crun*. The installation of crun, which is usually part of the installation of Kubernetes, must be configured to include the support for WasmEdge. The containers to be executed do not have to be based on an OS distribution like Ubuntu or Alpine Linux but only need to contain the Wasm modules and must be annotated with `module.wasm.image/variant: compat`. Latter is required to indicate to crun that the container is a Wasm-based container without an OS distribution. These containers can be run alongside non-Wasm-based containers on the same node. Removing the necessity for an OS distribution results in more lightweight containers that should be faster to start up and have a smaller disk and memory footprint.
WasmEdge modules can be written in Rust, JavaScript, Go, Swift, AssemblyScript, Kotlin, Grain, and Python. WasmEdge offers features that exceed the current state of development of WASI such as interfaces for network sockets, access to key-value stores, and database connectors. WasmEdge can also be used as a standalone runtime without Kubernetes.

Krustlet. Another approach to executing Wasm modules on Kubernetes is the use of an alternative kubelet (Kubernetes node agent) implementation, that supports Wasm. Krustlet, such kubelet implementation, can only run Wasm-based pods [De22]. Analogous to WasmEdge, Krustlet pods do not require an OS distribution inside a container. The Wasm modules are executed directly on Wasmtime, thus Krustlet does not even need to use a container runtime like crun. Hence, Krustlet should use less resources than the naive "Linux + Wasm runtime" container approach. To prohibit the scheduling of non-Wasm pods to Krustlet, Kubernetes taints are used. Taints can be understood as negative traits of nodes (e. g. not being able to run non-Wasm pods) and forbid the scheduling of pods that do not have a toleration to these taints. Consequentially, Wasm-based pods must specify these tolerations. When combining Krustlet and other standard kubelet implementations inside a cluster, non-Wasm and Wasm-based pods can be run alongside each other in the cluster, though not on the same (logical) node.
Krustlet is based on the Wasmtime runtime and supports WASI. As it provides no interfaces exceeding Wasmtime's WASI implementation, network sockets are not supported.

wasmCloud. An alternative to Kubernetes-oriented technologies is the application platform wasmCloud [Wc22]. wasmCloud uses an actor-capability component model. Actors are Wasm modules that implement business logic. Actors are strictly reactive in the sense that they can not initiate any action but can solely react to messages by the host runtime. Concurrency inside an actor is not supported, but external concurrency, i. e. multiple actors serving the same type of message, is possible.
Actors do not have access to their environment on their own. They have to rely on capabilities that abstract the environment and adjacent systems like blobstores, message queues, and HTTP-based servers/clients. Capabilities have a unique name and are defined in capability contracts that specify the interface between the actor and the capability provider. Capability providers are concrete implementations of the abstract capability, e. g. the contract `wasmcloud:blobstore` can be implemented based on a file system, using S3 or other blob stores. The capability abstraction ensures that the actor, that shall only implement business

logic, is not tightly coupled to one specific implementation of the capability. Besides capability providers, actors can also communicate with other actors.
Actors can be implemented in Rust or TinyGo, capability providers are to be implemented in Rust. Both can be published to OCI registries. They are run on the wasmCloud Host Runtime implemented in Elixir using the Wasmer-based Elixir Wasm runtime *Wasmex*. Interaction with the Host Runtime can be performed using the wasmCloud Shell (`wash`) or the browser-based wasmCloud Dashboard.
Multiple nodes running the wasmCloud Host Runtime can be connected using the so-called *Lattice* that is realized using the messaging system *NATS*. The lattice is a "self-forming, self-healing mesh network" [Wc22] that provides a flattened logical abstraction over heterogeneous network infrastructure. This enables the transparently decentralized execution of actors and capability providers.

Vino. Vino subdivides applications into small, reusable, and composable components containing business logic that are similar to functions on other platforms [Vi22]. Components have a defined interface for their input and output which must be described in WebAssembly Interface Definition Language (WIDL). From the interface definition, a code skeleton for the component can be generated. Components are written in Rust and compiled to Wasm and are publishable to OCI registries. These small and modular components can be composed to more complex meta components called schematics using a proprietary YAML syntax. Schematics can be used as components themselves. This composability allows for the creation of modular component networks that can be flexibly rearranged to satisfy new or changed requirements. Like wasmCloud actors, Vino components internally do not support concurrency but can be externally scaled to multiple, concurrently running component instances.
Vino components can be run once or served as HTTP or VRPC (Vino RPC; based on GRPC) web services using the Vino WebAssembly Wrapper `vow`. The underlying runtime is Wasmtime. Similar to wasmCloud, Vino components can be run on a NATS lattice to enable distribution. WASI is not supported by Vino.

2.2 Structural classification

The selected Wasm server platforms and technologies can be classified into three proposed classes. The classes are distinguished using the following definitions.

Application platforms (AP) define interfaces between the platform and Wasm modules to provide capabilities for cross-cutting concerns and/or abstracting adjacent systems like databases. They also allow for Wasm modules to be run as services (such as web services). Application platforms are similar to application servers as they externalize such concerns to the platform level, reducing the application/Wasm module to the implementation of the business logic. Some application platforms support the composition of Wasm modules to facilitate small and modular components that can be constellated flexibly. The platforms wasmCloud and Vino can be assigned to this class.

Wasm-enabling extensions (EX) are add-ons to existing platforms that add support for the execution of Wasm modules. The enhanced platforms either do not support Wasm modules at all or rely on other abstraction technologies like containerization to execute Wasm modules. Krustlet and WasmEdge on Kubernetes belong to this class though WasmEdge, when used as a standalone runtime, belongs to the class of the same name.

Standalone runtimes (SR) are basic environments for the execution of Wasm modules that usually do not provide further features for the orchestration of Wasm Modules or capabilities for cross-cutting concerns. They are generally executable via a command-line interface. These runtimes can be containerized and run on orchestration platforms that support containers such as Kubernetes and Docker Swarm. Wasmtime, Wasmer, and WasmEdge (called via CLI or inside a container) are part of this class.

2.3 Summary and discussion

The comparison of platforms and technologies is summarized in Tab. 1.

Platform/Technology	Class	Underlying runtime	Components	Network support
Standalone runtime, called via CLI	SR	*Interchangeable, here: Wasmtime, Wasmer*	Wasm modules	No
Standalone runtime, inside a container	SR	*Interchangeable, here: Wasmtime, Wasmer*	Containers	No
WasmEdge on K8s	SR, EX	WasmEdge	Pods, containers	Yes
Krustlet	EX	Wasmtime	Pods	No
wasmCloud	AP	Wasmex (based on Wasmer)	Actors, capabilities	Yes
Vino	AP	Wasmtime	Components, schematics	Yes

Tab. 1: Overview of platforms and technologies

Despite the promising concept and future of Wasm on the server, the current state of implementation is incomplete, inconsistent, and lacking important features.
Currently, it fails to deliver on the promise of portability as the different platforms and technologies provide different interfaces to the Wasm module, resulting in requiring platform/technology-specific glue-code. This is an implication of the lack of standardization of the system interface. WASI seeks to provide such standardization but is hitherto severely limited as it only provides "Proposed spec texts (Level 2)" for features like I/O and Clocks, and initial "Feature Proposals (Level 1)" for features like Sockets and Parallelism.[4] No feature has reached standardization (Level 5) yet. Platforms and technologies that do not provide their own specification for such features and only implement the WASI specification

[4] https://github.com/WebAssembly/WASI/blob/main/Proposals.md

thus lack these, in many cases critical, features.
Moreover, Wasm on the server currently falls short on the claim of polyglotism as many platforms only offer their required SDK for a limited set of languages, oftentimes only Rust.

3 Performance evaluation

The performance of the platforms and technologies is evaluated experimentally in order to answer the following questions:

- Q1: Do the platforms and technologies significantly differ in run-time performance?
- Q2: How strongly does the overhead of application platforms and containerization affect the run-time performance?
- Q3: Do non-OS-distribution-based containers outperform OS-distribution-based containers regarding startup time?
- Q4: Do relevant differences in HTTP response times and throughput exist between the platforms/technologies?

3.1 Methodology and environment

Prior to the development and execution of the performance evaluation, the experiments were designed by defining variable factors, namely *Platform/Technology*, *Workload type*, and *Workload size*, their levels, and the metrics to be collected. These definitions and further details about the environment and execution of the experiments are described in the following.

Factor: Platform/Technology. Ten platforms and technologies are selected. These are the standalone runtimes `WASMER` and `WASMTIME`, each called via `CLI`, in an Ubuntu-based container run using `DOCKER`, and in an Ubuntu-based container run using Kubernetes (`K8S`) as well as WasmEdge on Kubernetes (`WASMEDGE_ON_K8S`), `KRUSTLET`, `WASMCLOUD`, and `VINO`.

Factor: Workload type. Three levels for workload types are selected: calculation of a Fibonacci number using a recursive algorithm (`fib-r`), using an iterative algorithm (`fib-i`), and an HTTP server implementation that echoes the payload of a `POST` request (`http-echo`). The latter workload type can only be run on `VINO`, `WASMEDGE_ON_K8S`, and `WASMCLOUD` as the other platforms and technologies do not support network sockets.
The focus of this paper lies on the evaluation of differences between the platforms/technologies, hence the workloads are purposefully kept simple. All workloads are implemented in Rust and built including default compiler optimizations by using the `--release` flag.

Factor: Workload size. For the factor workload size, three levels, `SMALL`, `MEDIUM`, and `LARGE`, are selected and are described in Tab. 2. The payload for the workload `http-echo` is a string with a size of 100 bytes. For workload `fib-i` in sizes `MEDIUM` and `LARGE`, integer overflows occur, because 64 bit integers were used, as not all platforms support larger integers.

Workload	SMALL	MEDIUM	LARGE
fib-i	$n = 20$	$n = 40.000$	$n = 80.000.000$
fib-r	$n = 20$	$n = 42$	$n = 46$
http-echo	qps = 250, c = 32	qps = 500, c = 64	qps = 1000, c = 128

n = n-th Fibonacci number, qps = queries per second, c = number of parallel connections

Tab. 2: Workload Sizes

Metrics. For the workloads `fib-r` and `fib-i` the following metrics are collected:

- Startup time: The time between the deployment/invocation of the workload and the execution of the first statement of productive code of the workload.[5]
- Running time: The time between the execution of the first statement of productive code of the workload and the successful termination of the workload execution.

For the workload `http-echo`, the distribution of response times of the Wasm-based service and the maximum number of queries per second is measured.

Automation and tools. The execution of the experiments is automated using a measurement tool developed in Python; the installation of the platforms and technologies is automated via shell scripts. The platforms and technologies under evaluation are sequentially started up, the supported workloads are executed multiple times for "warming up", then repeatedly executed again while collecting the metrics outlined above, and finally shut down. For the HTTP measurements, Fortio is used. In order to prevent Fortio from impacting the performance of the platform or technology under evaluation, Fortio is run on another machine. The execution of the experiments can be configured using a YAML-based configuration file. The collected results are analyzed using Python within a Jupyter Notebook. The measurement tool, scripts, results and their evaluation are made available to the public.[6]

Platform/Technology deployment and execution. Vino components for the workloads `fib-r` and `fib-i` are run using `vow run`. The workload `http-echo` is served as an HTTP service using `vow serve`. The Kubernetes control plane for Krustlet is run on *kind* (Kubernetes in Docker) while Krustlet itself runs outside Docker in order to avoid the performance impact of Docker. WasmEdge on Kubernetes is run on vanilla Kubernetes. The pod for the workload `http-echo` is run in host network mode. wasmCloud actors are invoked using `wash call`.

Configuration and specification. The platforms and technologies under evaluation offer different degrees of configurability. The default configurations are used wherever

[5] As the platform wasmCloud strictly separates deployment of actors and their invocation, and additionally cannot measure the time between invocation and the execution of the first statement of productive Wasm code, only the deployment time is measured as part of the startup time for wasmCloud.

[6] `https://gitlab.com/broester.haus-public/running-wasm-on-the-server-evaluation`

possible. To ensure comparability, some configuration items are adjusted. The detailed platform/technology, system, and environment specifications are listed in Tab. 3.

	Attribute	Value	Attribute	Value
`CLI_*`	Wasmtime version	`0.37.0`	Wasmer version	`2.2.1`
`DOCKER_*` / `K8S_*`	Container base image	`Ubuntu:22.10`	Docker version	`20.10.14`
	Kubernetes configuration/specification is equal to `WASMEDGE_ON_K8S`			
`WASMEDGE_ON_K8S`	Kubernetes version	`1.22.2`	WasmEdge version	`0.9.1`
	Container runtime	`CRI-O`	crun version	`1.4.5`
`KRUSTLET`	kind version	`0.14.0`	Krustlet version	`0.7.0`
`WASMCLOUD`	wasmCloud host version	`0.54.6`	NATS version	`2.8.4`
	HTTP server provider	`httpserver: 0.14.4`	wash version	`0.11.0`
`VINO`	Vino version	`2022-04-08`	Thread count for `vow serve`	`1`
System/ Environment	OS	`Ubuntu 20.04`	CPU clock speed	`3.50 GHz`
	CPU core count	`4`	Memory size	`8192MB`

Tab. 3: Platform/Technology configuration and specification

3.2 Startup time and running time of computation-intensive workloads

The startup time and running time of computation-intensive workloads are evaluated using the workloads `fib-i` and `fib-r`. Each workload is executed for every platform in every workload size and repeated 25 times. To avoid the effects of cold starts, five runs of each workload in each workload size are executed beforehand.

3.2.1 Startup time

The experimentally measured startup times are boxplotted in Fig. 2. The results show large differences between platforms and technologies. The fastest mean startup times were measured for `CLI_WASMER` (23.10 ms) and `CLI_WASMTIME` (23.33 ms). `VINO`, which is based on Wasmtime, introduces a small overhead of about 4 ms over `CLI_WASMTIME`. OS-distribution-based containerization of Wasmer and Wasmtime and execution via Docker increased the startup time by a factor of 15.6× while using Kubernetes increased startup time by about 34×. Using `KRUSTLET` with non-OS-distribution-based containers can sometimes yield faster startup times compared to OS-distribution-based containers, but is on average 20.1% slower than `K8S_WASMTIME` and 24.5% slower than `K8S_WASMER`. `WASMEDGE_ON_K8S` has a slightly faster startup time with a mean of 910 ms than `KRUSTLET` (962 ms) but is still slower compared to OS-distribution-based containers measured on `K8S_WASMTIME` (773 ms) and `K8S_WASMER` (801 ms), answering question Q3. `WASMCLOUD` has the slowest startup by far with a mean value of 3220 ms.

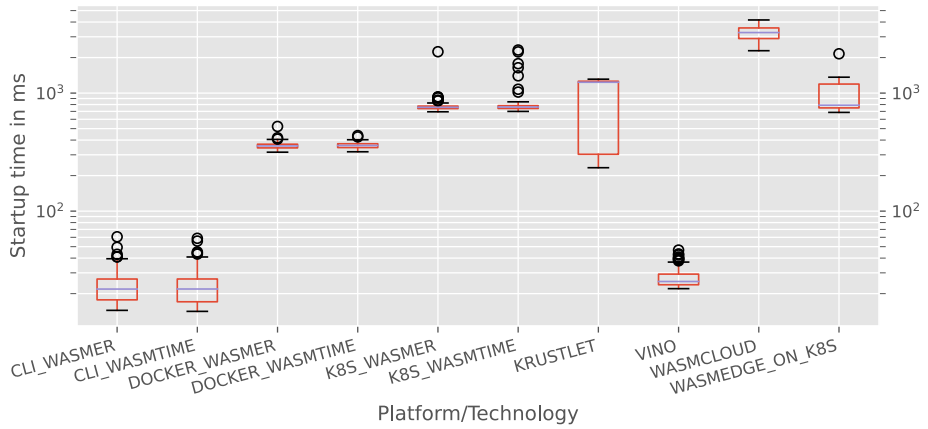

Fig. 2: Startup times of workloads `fib-i` and `fib-r` (max whisker length is 1.5 × IQR)

3.2.2 Running time

The running time of the workload `fib-i` is shown in Fig. 3. For workload `fib-i`, the workload size has no large impact on the running time except for `VINO` which shows a nine-fold increase between the sizes `MEDIUM` and `LARGE`. Averaged over all workload sizes, `CLI_WASMTIME` has the lowest mean running time with a value of 1.21 ms, closely followed by `CLI_WASMER` with 1.48 ms. The other platforms and technologies show significantly longer running times, up to about 1700 ms in the case of `WASMCLOUD`.

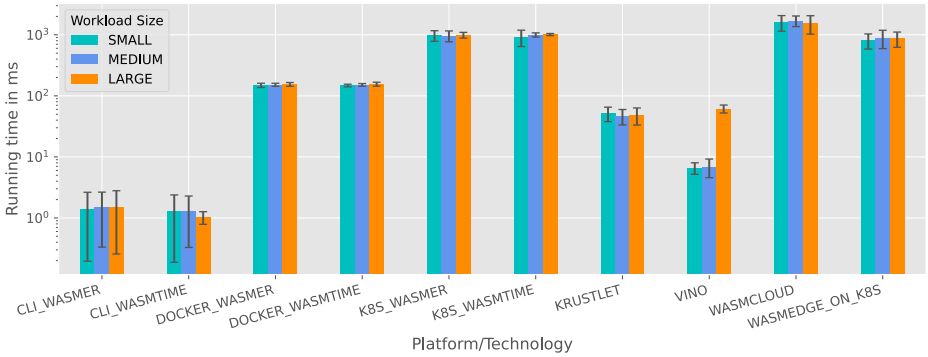

Fig. 3: Running times of workload `fib-i` for different workload sizes (Error bars show SD)

Similar to the startup time, containerization increases running time, here by a factor of about 125× (`CLI_WASMTIME` vs. `DOCKER_WASMTIME`). This factor however shrinks significantly

with larger workloads, which can be seen in the data of workload `fib-r` which is plotted in Fig. 4. For workload size `LARGE` the factor `CLI_WASMTIME` vs. `DOCKER_WASMTIME` is reduced to 0.995×, indicating that the overhead of containerization becomes insignificant with increasing workload size.

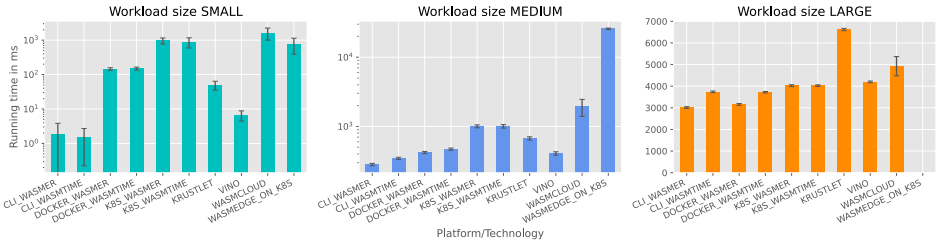

Fig. 4: Running times of workload `fib-r` for different workload sizes (Error bars show SD)

The usage of application platforms, `VINO` or `WASMCLOUD`, introduces a significant overhead for small workloads. `WASMCLOUD` in particular increases running time extremely for small workloads. With large workloads, this overhead is not as influential as with small workloads, but is still present, as made evident in Tab. 4. This provides an answer to question Q2.

Workload size	`CLI_WASMER`	`CLI_WASMTIME`	`VINO`	`WASMCLOUD`
`SMALL`	1.27×	1.00×	4.52×	1104.48×
`MEDIUM`	1.00×	1.23×	1.45×	6.86×
`LARGE`	1.00×	1.24×	1.39×	1.63×

Tab. 4: Increase of running time using application platforms for workload `fib-r`

In general, the differences in running time between platforms/technologies diminish with increasing workload size. This may be caused by considerable termination times of the workloads, which are included in the running time in these experiments, or potential parallel initialization processes that are executed alongside the running Wasm module.
An exception to this is `WASMEDGE_ON_K8S`, whose `fib-r` running time grows rapidly with workload size in comparison with the other platforms/technologies. The workload of size `MEDIUM` runs 91 times longer than `CLI_WASMER`, which is the fastest for this size. The workload size `LARGE` cannot be executed on `WASMEDGE_ON_K8S` in reasonable time and thus is skipped. Another exception is `KRUSTLET`, which has the longest running time for size `LARGE` (77% slower than `CLI_WASMTIME`, despite also using Wasmtime as its underlying runtime) while being mid-range in sizes `SMALL` and `MEDIUM`.

Answering question Q1, it can be summarized that there are significant differences in run-time performance between platforms and technologies, especially for small workloads.

3.3 Response time and throughput of HTTP server implementations

The response time and throughput are evaluated using the workload `http-echo`, run for 20 seconds. The experiments are conducted after "warming up" the platform by executing the workload three times.

The distribution of response times is shown in Fig. 5. Vino performs best and wasmCloud performs worst for all workload sizes. The differences between the platforms/technologies increase with growing workload size. Expectedly, the median response time increases with the number of queries per second.

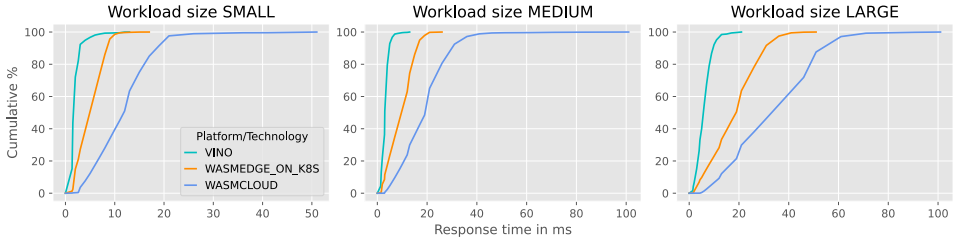

Fig. 5: HTTP response time distributions of platforms/technologies for different workload sizes

For measuring maximum throughput, an unlimited number of requests per second are sent via 512 parallel connections. The results listed in Tab. 5 show that Vino offers significantly higher throughput than WasmEdge on Kubernetes and wasmCloud.

Platform/Technology	Mean	SD
`VINO`	16511	449
`WASMEDGE_ON_K8S`	2221	640
`WASMCLOUD`	1858	26

Tab. 5: Maximum requests per second for workload `http-echo`

Answering question Q4, our results show that there are relevant differences in response times and especially throughput between the examined platforms and technologies.

3.4 Summary and discussion

The results of the experiments presented above are consolidated in Tab. 6. The simplest approach, using standalone runtimes via CLI, performs best overall but does not support HTTP. Vino is a viable alternative if HTTP server functionality is required. Containerizing Wasm increases startup time but does not decrease running time significantly. WasmEdge on Kubernetes and wasmCloud on the other hand offer subpar performance but can offer functionality that is not present in other platforms and technologies.

	CLI_*	DOCKER_*	K8S_*	WASMEDGE_ON_K8S	KRUSTLET	WASMCLOUD	VINO
Startup time	++	+	o	o	o	--	++
Running time (small WL)	++	o	-	-	o	--	+
Running time (large WL)	++	++	+	--	-	o	+
HTTP response time	n/a	n/a	n/a	o	n/a	--	++
HTTP throughput	n/a	n/a	n/a	-	n/a	-	+

Tab. 6: Overview of performance metrics per platform (Workload abbreviated as *WL*)

4 Conclusion

The creation of Wasm standalone runtimes and WASI has evoked the development of several platforms and technologies for running Wasm on the server. We have compared these platforms and technologies regarding their structure and defined a three-class structural classification system. Afterwards, we evaluated the startup and running time as well as HTTP response time and throughput of these platforms and technologies.
Future research in this area may be conducted by evaluating the distributed execution of workloads on the platforms and technologies or by using real-world workloads.

References

[By22] Bytecode Alliance: Introduction - Wasmtime, 2022, URL: https://docs.wasmtime.dev/, visited on: 07/05/2022.

[De22] Deis Labs Team: Krustlet Documentation, 2022, URL: https://docs.krustlet.dev/, visited on: 07/05/2022.

[Go19] Gohman, Dan: WASI: WebAssembly System Interface, 2019, URL: https://github.com/bytecodealliance/wasmtime/blob/main/docs/WASI-overview.md, visited on: 07/11/2022.

[Ha17] Haas, A. e. a.: Bringing the Web up to Speed with WebAssembly. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. Pp. 185–200, 2017.

[Lo21] Long, J.; Tai, H.-Y.; Hsieh, S.-T.; Yuan, M. J.: A Lightweight Design for Serverless Function as a Service. IEEE Software 38/1, pp. 75–80, 2021.

[Vi22] The Vino Team: Vino.Docs, 2022, URL: https://docs.vino.dev/, visited on: 07/03/2022.

[Wc22] WasmCloud Project Authors: wasmCloud Documentation, 2022, URL: https://wasmcloud.dev/, visited on: 06/29/2022.

[We22] WasmEdge Project Authors: WasmEdge in Kubernetes, 2022, URL: https://wasmedge.org/book/en/kubernetes.html, visited on: 06/30/2022.

Vergleich von Nebenläufigkeit in den Programmiersprachen Rust und Go am Beispiel der prototypischen Implementierung eines Webservers

Marcel Niehüsener[1]

Abstract: Die jungen Programmiersprachen Rust und Go erfreuen sich in der letzten Zeit immer größerer Beliebtheit. Liegt Rusts Fokus eher auf Verlässlichkeit und Maschinennähe, fokussiert sich Go mehr auf den schnellen Kompiliervorgang und einen einfachen Sprachumfang. Diese Arbeit beschäftigt sich mit Nebenläufigkeit in beiden Programmiersprachen am Beispiel der prototypischen Implementierung eines Webservers. Dazu werden verschiedene Implementierungen mit mehreren Endpunkten vorgestellt und im Zuge von Messungen miteinander verglichen. Die Endpunkte werden mit und ohne Keepalive und unterschiedlich vielen parallelen Verbindungen aufgerufen.

Keywords: Rust; Go; Nebenläufigkeit; Webserver; Sockets

1 Einleitung

Für die Bereitstellung unterschiedlichster Dienste werden Serveranwendungen benötigt. Einige Beispiele für solche Dienste sind Chats, soziale Netzwerke und Geschäftsanwendungen. Auch für das Internet of Things werden Serveranwendungen benötigt. Häufig werden diese in Form von Webservern bereitgestellt, die per HTTP angesprochen werden.

Eine zunehmend vernetzte Welt führt zu größeren Auslastungen dieser Anwendungen. Dadurch ergeben sich auch größere Anforderungen an die Performance und Skalierbarkeit dieser Dienste. Nicht nur im Zuge des Cloud-Computings führt eine effiziente Ressourcennutzung zu monetären Auswirkungen in Form von Kosteneinsparungen. Serveranwendungen, die viele parallele Verbindungen bei einem geringen Ressourcenverbrauch handhaben können, stellen hier einen Vorteil dar.

Die jungen Programmiersprachen Rust und Go erfreuen sich in der letzten Zeit immer größerer Beliebtheit [St22]. Gleichzeitig behaupten Rust und Go, eine gute Performance für nebenläufige Anwendungsfälle sowie ein hohes Abstraktionsniveau zu bieten.

Im Zuge dieser Arbeit erfolgt ein Vergleich der Nebenläufigkeit in den beiden Programmiersprachen am Beispiel der prototypischen Implementierung eines Webservers. Dazu werden eine Go-Implementierung sowie vier Rust-Implementierungen umgesetzt und miteinander verglichen. In der Literatur gibt es bereits einige Arbeiten, die sich mit dem Vergleich von

[1] FH Münster, Fachbereich Wirtschaft, Wirtschaftsinformatik, mn600659@fh-muenster.de

Nebenläufigkeit in unterschiedlichen Programmiersprachen für verschiedene Anwendungsfälle beschäftigen. Es konnte jedoch keine Arbeit gefunden werden, die Rust und Go am Beispiel der Implementierung von Webservern vergleicht.

Das nächste Kapitel führt zunächst einige Grundlagen zu Rust und Go sowie verwandte Arbeiten an. In Kapitel 3 wird die grundlegende Funktionsweise der Webserver-Implementierungen erläutert, ehe in Kapitel 4 auf Unterschiede zwischen den Implementierungen eingegangen wird. Kapitel 5 beschreibt den Messaubau, während Kapitel 6 die Messergebnisse vorstellt und begründet. In Kapitel 7 wird ein Fazit gezogen und ein Ausblick gegeben.

2 Grundlagen

Nachfolgend werden einige Grundlagen von Rust und Go erläutert. Dabei wird ein besonderer Fokus auf die nebenläufige Programmierung und auf die Thread-Modelle gelegt. Zusätzlich werden verwandte Arbeiten angeführt, die sich bereits mit der Eignung von Programmiersprachen für die Entwicklung von Serveranwendungen befasst haben.

2.1 Rust

Rust ist eine statisch und stark typisierte Programmiersprache, die sich durch Systemnähe auszeichnet und zu Maschinensprache kompiliert wird. Der Entwurf begann 2006 durch Graydon Hoare bei Mozilla, während die erste Veröffentlichung 2010 erfolgte [YSZ19]. Die erste stabile Version erschien 2015. Rust besitzt keinen Garbage Collector, sondern nutzt eine deterministische Speicherverwaltung basierend auf einem Ownership- und Borrowing-System. Dadurch werden Ressourcen automatisch freigegeben, ohne dass dies explizit angewiesen werden muss. [PGF19]

Jeder Wert in Rust besitzt einen Owner und eine Lifetime. Der Owner ist häufig der Scope, in dem der Wert deklariert wurde, kann aber verändert werden, wenn der Wert beispielsweise an eine Funktion übergeben wird (move). Die Lifetime beginnt, wenn der Wert erstellt wird, und endet, wenn der Wert freigegeben wird. Der Rust-Compiler stellt sicher, dass jede Referenz (Borrow) gültig ist, sodass nicht auf Werte zugegriffen werden kann, die bereits freigegeben wurden (use-after-free). Für jeden Wert kann es gleichzeitig nur eine veränderbare Referenz geben, wodurch einige Race Conditions vermieden werden. Mehrere unveränderbare Referenzen sind erlaubt, wenn es gleichzeitig keine veränderbaren Referenzen gibt. [PGF19]

Parallele Verarbeitung kann in Rust mithilfe von Threads realisiert werden. Dabei wird ein Rust-Thread durch genau einen Thread des Betriebssystems abgebildet. Leichtgewichte (green) Threads und einen Scheduler, der diese auf physische Threads verteilt, bietet Rust nicht [YSZ19]. Die Sprache sieht zwar ein *async*-Schlüsselwort vor und besitzt eine

Future-Schnittstelle in der Standardbibliothek, allerdings ist die Verwendung ohne externe Bibliotheken oder eigenem Executor nicht möglich. [Ru18]

Zur Synchronisation von Threads bietet Rust klassische Locks und atomare Datentypen sowie Channel an. Über Channel können Werte mit anderen Threads ausgetauscht werden, wobei gleichzeitig die Ownership übertragen wird. Somit kann der sendende Thread nicht mehr auf den Wert zugreifen. Um threadübergreifend auf dieselben Werte zuzugreifen, besitzt Rust sogenannte Smart Pointer. *Arc* (Atomically Reference Counted) stellt einen Smart Pointer dar, der nach dem Klonen an Threads übertragen werden darf und dabei die Anzahl an Referenzen auf den beinhalteten Wert zählt. Soll threadübergreifend veränderbar auf den Wert zugegriffen werden, erfordert der Compiler zusätzlich, dass ein *Mutex* verwendet wird. [YSZ19]

2.2 Go

Bei Go handelt es sich um eine Programmiersprache, dessen Entwurf 2007 bei Google begonnen wurde. Go ist statisch und stark typisiert, besitzt einen Garbage Collector und wird zu Maschinensprache kompiliert [VCT18]. Die erste stabile Version wurde 2012 veröffentlicht.

Nebenläufigkeit wird in Go mithilfe von Goroutinen abgebildet. Bei Goroutinen handelt es sich um leichtgewichtige Ausführungseinheiten, die durch Go zur Laufzeit auf physische Threads verteilt werden. Je Prozessorkern gibt es standardmäßig maximal einen Thread, sodass die Anzahl an Kontextwechseln reduziert wird. Gegenüber physischen Threads benötigen Goroutinen weniger Arbeitsspeicher, sodass die Anzahl an maximal startbaren Goroutinen größer ist. Gleichzeitig ist das Wechseln zwischen Goroutinen weniger aufwendig als der Wechsel zwischen physischen Threads. [ST12]

Zur Synchronisation von parallelen Verarbeitungseinheiten bietet Go zwei Optionen. Einerseits gibt es den klassischen Ansatz basierend auf Locks. Andererseits gibt es eine auf Tony Hoares *Communicating Sequential Processes* basierende Algebra, die den Austausch von Informationen zwischen Goroutinen über Channel ermöglich. [SM17]

2.3 Verwandte Arbeiten

In [VCT18] wurden zwölf nebenläufige Programmiersprachen – darunter auch Rust und Go – auf ihre Eignung zur Entwicklung von Serveranwendungen verglichen. Der Vergleich fand basierend auf den Sprachkonstrukten statt, ohne dass TCP-Verbindungen aufgebaut wurden. Dazu wurde unter anderem gemessen, wie lange das Starten von Verarbeitungseinheiten dauert, welcher Durchsatz mit untereinander kommunizierenden Verarbeitungseinheiten erreicht werden kann und wie viele Verarbeitungseinheiten maximal gestartet werden

können. Während Go den höchsten Durchsatz erreichte, konnte Scala mit Akka die größte Anzahl an parallelen Verarbeitungseinheiten erreichen.

In [HB20] wurde eine Key-Value-Datenbank in den Programmiersprachen Java, Rust und C++ realisiert. Die Implementierungen horchen im Main-Thread auf eingehende TCP-Verbindungen und starten je Verbindung einen eigenen Thread. Gestartete Threads bleiben so lange bestehen, bis der Client die Verbindung schließt oder ein Timeout auftritt. Während Rust und C++ nahezu gleichauf lagen, wurde festgestellt, dass die Java-Implementierung einen geringeren Durchsatz erreichte.

Schmaus et al. verglichen in [Sc21] Go, Rust (*tokio*) und eine eigene Plattform (EMPER) hinsichtlich nebenläufiger I/O-Operationen. Dazu wurde ein Echo-Server umgesetzt und hinsichtlich des Durchsatzes bewertet. EMPER erreichte den höchsten Durchsatz. Go schnitt besser als Rust ab, wenn die Antworten direkt gesendet wurden. Wurden die Antworten verzögert gesendet, erreichte Rust einen höheren Durchsatz als Go.

3 Funktionsweise der Webserver

Für die Ermittlung, welcher Endpunkt aufgerufen wurde, ist es zunächst erforderlich, die HTTP-Anfrage zu parsen. Hierfür wurde sowohl für Rust als auch für Go eine Bibliothek entwickelt, die diese Aufgabe übernimmt. Diese Bibliothek beinhaltet einen HTTP-Parser, dem über die *append*-Methode mehrfach Bytes zugeführt werden können, die vom Client gesendet wurden. Im Fehlerfall, beispielsweise wenn die *Request-Line*[2] ein ungültiges Format aufweist oder die Länge des Bodys die übermittelte *Content-Length*[3] überschreitet, wird ein entsprechender Fehler zurückgegeben. Zusätzlich beinhaltet die Bibliothek Strukturen, über die HTTP-Anfragen und -Antworten abgebildet werden. Bei der Entwicklung der Bibliotheken wurde darauf geachtet, möglichst identische Implementierungen in Rust und Go umzusetzen, sodass spätere Messungen nicht durch unterschiedliche Implementierungen beeinflusst werden. Abbildung 1 visualisiert die Rust-Bibliothek in Form eines Klassendiagramms.

Die Webserver horchen auf eingehende TCP-Verbindungen und nehmen diese entgegen. Die Verbindungen werden dann an eine eigene Ausführungseinheit übergeben, die die weitere Verarbeitung übernimmt. Innerhalb der Ausführungseinheit wird so lange vom Socket gelesen, bis die Anfrage vollständig ist. Im Anschluss folgt die Ausführung der Geschäftslogik, die je Endpunkt unterschiedliche Aktionen ausführt. Abschließend wird die Antwort über das Socket gesendet. Dieses Verfahren wird so lange wiederholt, bis ein Fehler auftritt (beispielsweise, weil der Client die Verbindung unerwartet schließt) oder das Schließen der Verbindung mit dem Connection-Header[4] angewiesen wurde. Der Pseudocode in Listing 1 veranschaulicht die Funktionsweise.

[2] https://datatracker.ietf.org/doc/html/rfc2616#section-5.1
[3] https://datatracker.ietf.org/doc/html/rfc2616#section-14.13
[4] https://datatracker.ietf.org/doc/html/rfc2616#section-14.10

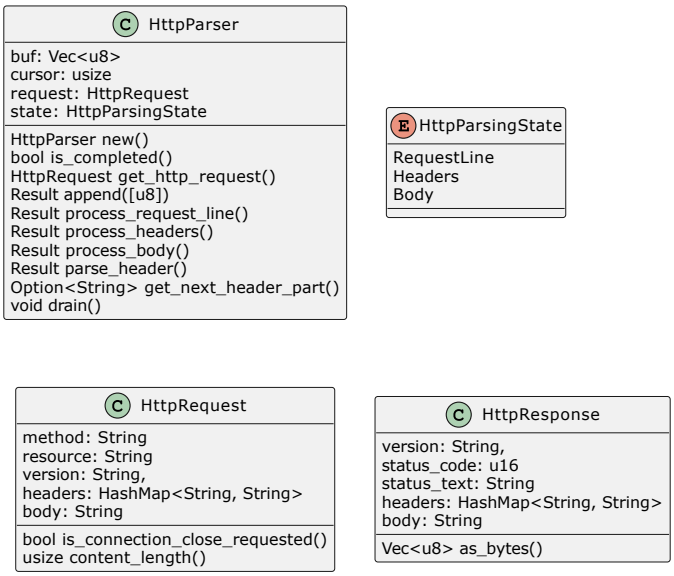

Abb. 1: Klassendiagramm der entwickelten Rust-HTTP-Bibliothek

```
listener := bind("0.0.0.0:8080")
while socket := listener.accept():
  start thread/task:
    while:
      try:
        request := read_and_parse_request(socket)
        response := business_logic(request)
        if request.close_connection():
          response.add_connection_close_header()
        socket.write(response)
        if request.close_connection():
          break
      catch:
        break
```

List. 1: Pseudocode des Webservers

Für die Messungen wurden die drei nachfolgend genannten Endpunkte umgesetzt. Wird ein nicht-existenter Endpunkt aufgerufen, wird mit dem Statuscode 404 (Not Found) geantwortet. Falls ein Endpunkt, der eine Zahl als Parameter erwartet, mit einem Wert aufgerufen wird, der nicht geparst werden kann, wird mit dem Statuscode 400 (Bad Request) geantwortet.

- **/hello** antwortet direkt mit einem statischen Inhalt.

- **/sleep/{ms}** antwortet nach *{ms}* Millisekunden ohne Inhalt (Content-Length: 0).
- **/fibonacci/{i}** berechnet rekursiv die *i*-te Zahl der Fibonacci-Folge und antwortet mit dieser.

4 Vorstellung der Implementierungen

Im Zuge der Messungen werden eine Go-Implementierung (Compiler-Version 1.18.3) und vier Rust-Implementierungen (Compiler-Version 1.61.0) miteinander verglichen. Die Go-Implementierung basiert ausschließlich auf der Go-Standardbibliothek und benötigt keine externen Abhängigkeiten. Die Verarbeitung jeder eingehenden Verbindung wird mithilfe des *go*-Schlüsselworts an eine eigene Goroutine übertragen, sodass parallele Verbindungen ermöglicht werden.

Die ersten beiden Rust-Implementierungen basieren ausschließlich auf der Rust-Standardbibliothek und benötigen ebenfalls keine externen Abhängigkeiten. Die erste Implementierung startet für jede eingehende Verbindung mithilfe von *std::thread::spawn* einen neuen Thread, wohingegen die zweite Implementierung einen Threadpool basierend auf [KN19] mit 10.000 bereits initialisierten Threads verwendet. Die Verarbeitung der eingehenden Verbindungen wird über einen Channel[5] an einen der bestehenden Threads weitergegeben.

Die beiden weiteren Rust-Implementierungen basieren auf asynchronen Streams. Da asynchrone Streams nicht in der Standardbibliothek enthalten sind, ist die Verwendung von externen Bibliotheken erforderlich. Als Bibliotheken wurden *async-std*[6] in der Version 1.11.0 und *tokio*[7] in der Version 1.18.1 ausgewählt. Während sich *async-std* stärker an der Standardbibliothek orientiert, wird *tokio* von einer größeren Anzahl an Projekten verwendet. Neben Strukturen für den nicht-blockierenden Zugriff auf Streams bieten die beiden Bibliotheken einen eigenen Scheduler und eigene Tasks, sodass *N* Tasks durch *M* Threads des Betriebssystems ausgeführt werden können.

5 Messaufbau

Zur Ausführung der Messungen werden zwei virtualisierte Server im VRA-Cluster der Datenverarbeitungszentrale der FH Münster verwendet. Der erste Server dient zur Ausführung der Webserver-Implementierungen, während der zweite Server für das Senden der Anfragen zuständig ist. Hierdurch wird sichergestellt, dass unterschiedliche CPU-Auslastungen der Webserver keine Auswirkungen auf das Senden der Anfragen haben. Der zweite Server übernimmt zusätzlich das Starten und Stoppen der Implementierungen auf dem ersten

[5] https://doc.rust-lang.org/std/sync/mpsc/
[6] https://crates.io/crates/async-std
[7] https://crates.io/crates/tokio

Server. Zum Messen von Metriken wie CPU-Zeit, Arbeitsspeicherverbrauch und Anzahl der Kontextwechsel wird *time*[8] verwendet. Um die Ressourcennutzung im zeitlichen Verlauf analysieren zu können, wird auf *top*[9] gesetzt. Zum Senden von Anfragen wird *fortio*[10] verwendet. Die Ausführung der Messungen wurde mithilfe von Shell-Skripten automatisiert.

Beide Server sind identisch und verwenden Ubuntu 20.04 LTS als Betriebssystem. Sie besitzen 4 virtuelle CPU-Kerne sowie 8.192 MB Arbeitsspeicher. Als Prozessor ist ein Intel Xeon E5-2680 v3 mit einer Grundtaktfrequenz von 2,5 GHz (max. Turbo-Taktfrequenz von 3,3 GHz) verbaut. Das File-Descriptor-Limit wurde auf beiden Server auf 65.536 erhöht, damit für die Messungen ausreichend viele parallele Verbindungen erlaubt werden.

Die Implementierungen werden hinsichtlich Durchsatz, Latenz und Arbeitsspeicherverbrauch verglichen. Die CPU-Zeit und die Anzahl der Kontextwechsel werden lediglich zur Begründung der Ergebnisse, nicht jedoch als eigene Bewertungskriterien verwendet. Für die Messung des Durchsatzes werden die Endpunkte */hello* und */sleep/20* mit 50 – 10.000 parallelen Verbindungen aufgerufen. Die Messungen werden sowohl mit als auch ohne Keepalive ausgeführt, sodass für jede HTTP-Anfrage eine neue TCP-Verbindung aufgebaut wird. Der Endpunkt */fibonacci/37* wird mit 5 – 400 parallelen Verbindungen aufgerufen, wobei ausschließlich Verbindungen mit Keepalive verwendet werden, da der Verbindungsaufbau im Vergleich zur rekursiven Berechnung nur einen Bruchteil der Zeit benötigt. Jede Kombination aus parallelen Verbindungen, Endpunkt und Keepalive wird je Implementierung 15-mal wiederholt. Jede Messung wird für die Dauer von 10 Sekunden ausgeführt. Für die Messung der Latenz wird ausschließlich der Endpunkt */hello* mit einer parallelen Verbindung und einer Anfrage pro Sekunde aufgerufen, sodass die Latenz nicht durch eine hohe Auslastung beeinflusst wird. Die Latenz-Messungen werden für die Dauer von 30 Sekunden ausgeführt und je Implementierung 75-mal wiederholt. Für alle Anfragen wird ein Timeout von 3 Sekunden verwendet.

6 Messergebnisse

Abbildung 2 zeigt die erfolgreichen Anfragen je Sekunde für den Endpunkt */hello* mit Keepalive für bis zu 5.200 parallele Verbindungen. Bei einer höheren Verbindungsanzahl ergeben sich keine nennenswerten Änderungen. Während bei 50 parallelen Verbindungen alle Implementierungen bei etwas über 100.000 Anfragen pro Sekunde liegen, erreicht die *tokio*-Implementierung das Maximum mit über 300.000 Anfragen pro Sekunde bei 400 parallelen Verbindungen und erzielt anschließend dauerhaft die besten Ergebnisse. Mit zunehmender Anzahl an Verbindungen nimmt der Durchsatz etwas ab und pendelt sich bei etwa 250.000 Anfragen je Sekunde ein. Bei 200 – 400 parallelen Verbindungen liegen zudem alle Rust-Implementierungen über der Go-Implementierung. Die Thread-basierten Rust-Implementierungen fallen bei zunehmender Anzahl an Verbindungen am stärksten ab und

[8] https://manpages.ubuntu.com/manpages/focal/en/man1/time.1.html
[9] https://manpages.ubuntu.com/manpages/focal/en/man1/top.1.html
[10] https://github.com/fortio/fortio/

liegen ab 1.200 Verbindungen unterhalb der Go-Implementierung. Das lässt sich durch eine hohe Anzahl an Kontextwechseln zwischen den Threads begründen. So benötigen die beiden Implementierungen bei 10.000 parallelen Verbindungen im Schnitt etwa 1,8 Millionen Kontextwechsel, während die anderen Implementierungen unter 100.000 Kontextwechsel liegen. Dementsprechend verbringen die Thread-basierten Implementierungen auch einige CPU-Sekunden mehr im Kernel- und dementsprechend weniger im User-Modus. Die Implementierung basierend auf *async-std* und die Go-Implementierung liegen ab 4.000 Verbindungen etwa gleich auf.

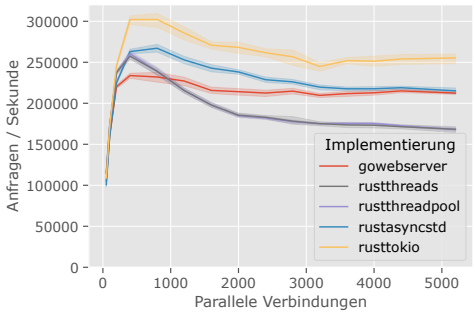

Abb. 2: Durchsatz für den /hello-Endpunkt mit Keepalive

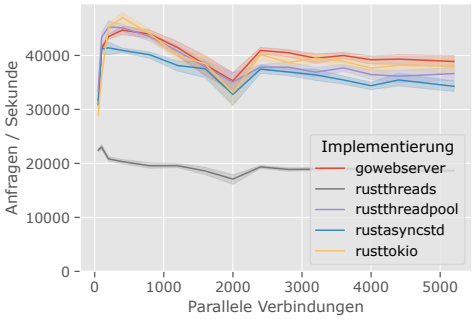

Abb. 3: Durchsatz für den /hello-Endpunkt ohne Keepalive

Die Ergebnisse für die Messungen ohne Keepalive werden in Abbildung 3 visualisiert. Auch hier gibt es bei höherer Verbindungsanzahl keine nennenswerten Änderungen. Erreicht die naive Rust-Implementierung bei bis zu 200 parallelen Verbindungen noch über 20.000 Anfragen je Sekunde, liegt sie anschließend dauerhaft unter 20.000 Anfragen. Die anderen Implementierungen starten bei etwa 29.000 Anfragen. Die *tokio*-Implementierung erreicht maximal 47.000 Anfragen bei 400 Verbindungen. Bei zunehmender Anzahl an parallelen Verbindungen lässt sich erneut ein abnehmender Durchsatz beobachten, wobei die Abnahme geringer als mit aktiviertem Keepalive ist. Die *tokio*-Implementierung und die Go-Implementierung liegen etwa gleich auf, gefolgt von der Threadpool-Implementierung. Auf dem vierten Platz liegt die *async-std*-Implementierung. Die verhältnismäßig schlechten Ergebnisse der naiven Implementierung lassen sich dadurch erklären, dass für jede HTTP-Anfrage ein neuer Thread erstellt wird. Das führt auch dazu, dass diese Implementierung je Messung durchschnittlich 21,3 CPU-Sekunden im Kernel-Modus verbringt, während es bei den anderen Implementierungen 15,3 – 18,7 Sekunden sind.

Abbildung 4 zeigt die Ergebnisse für den Endpunkt */sleep/20*. Auffällig ist zunächst, dass alle Implementierungen bis 2.400 Verbindungen nahezu gleichauf liegen und ein lineares Wachstum besitzen. Aufgrund der Wartezeit benötigt jede Anfrage unabhängig von der Implementierung mindestens 20 Millisekunden. Somit ist maximal ein Durchsatz von $1/0.02 \times Verbindungen$ möglich. Die Thread-basierten Rust-Implementierungen erreichen bei 3.200 Verbindungen ihren maximalen Durchsatz und fallen anschließend ab. Dies lässt

sich erneut durch eine hohe Anzahl an Kontextwechseln begründen. Die Implementierungen basierend auf *tokio* und *async-std* erreichen ihr Maximum im Bereich zwischen 4.000 und 6.000 Verbindungen und fallen anschließend leicht ab. Die Go-Implementierung benötigt hingegen 7.600 Verbindungen, um über 155.000 Anfragen je Sekunde beantworten zu können und liegt abschließend auf dem gleichen Niveau wie die *tokio*-Implementierung, während die *async-std*-Implementierung etwas schlechtere Ergebnisse erzielt. Gleichzeitig benötigt die Go-Implementierung bis 7.600 Verbindungen weniger CPU-Zeit. Das könnte darauf hindeuten, dass der Go-Scheduler länger benötigt, bis die Goroutinen aufgeweckt werden, wodurch der geringere Durchsatz zu begründen wäre.

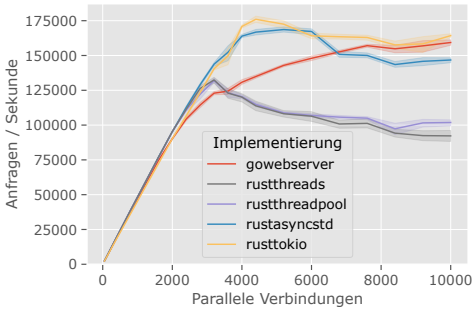

Abb. 4: Durchsatz für den /sleep/20-Endpunkt mit Keepalive

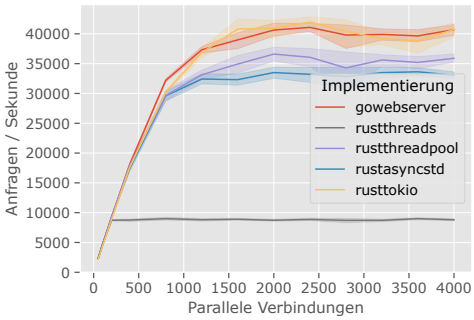

Abb. 5: Durchsatz für den /sleep/20-Endpunkt ohne Keepalive

Mit deaktiviertem Keepalive fallen die Ergebnisse deutlich konstanter aus, wie Abbildung 5 zeigt. Bereits ab 3.200 parallelen Verbindungen ergeben sich keine nennenswerten Änderungen mehr. Im Gegensatz zum Endpunkt */hello* werden mehr Verbindungen benötigt, um den maximalen Durchsatz zu erreichen, was erneut durch die höhere Latenz zu begründen ist. Davon abgesehen erreichen alle Implementierungen einen Durchsatz, der in der gleichen Größenordnung wie beim */hello*-Endpunkt liegt. Lediglich die Rust-Implementierung, die je Verbindung einen neuen Thread startet, erreicht mit 9.000 Anfragen je Sekunde nur noch den halben Durchsatz. Durch das dauerhafte Erstellen neuer Threads verursacht diese Implementierung eine hohe CPU-Auslastung. Im Gegensatz zum */hello*-Endpunkt kann ein gestarteter Thread jedoch nicht direkt auf die Anfrage antworten, sondern wird erneut schlafen gelegt. Durch die hohe Auslastung dauert es länger als 20 Millisekunden, bis die Threads aufgeweckt werden und antworten können, was zu einer höheren Latenz und somit zu einem geringeren Durchsatz führt. Dieses Problem besteht bei der Threadpool-Implementierung aufgrund der geringeren Auslastung nicht.

Abbildung 6 zeigt die Ergebnisse für den Endpunkt */fibonacci/37*. Aufgrund der benötigten Rechenleistung fällt der Durchsatz deutlich geringer aus. Auffällig ist, dass die Go-Implementierung einen deutlich geringeren Durchsatz als die Rust-Implementierungen erreicht. Wie in [Je22] gezeigt wurde, ist Rust bei der rekursiven Berechnung der Fibonacci-Folge effizienter. Somit ist dieses Ergebnis nicht durch die Webserver-Implementierung

zu begründen. Des Weiteren erreichen alle Implementierungen zunächst einen konstanten Durchsatz, bevor dieser deutlich abfällt. Da die rekursive Berechnung der Fibonacci-Folge eine hohe Rechenleistung beansprucht, führt eine höhere Anzahl an parallelen Verbindungen dazu, dass die Dauer jeder einzelnen Anfrage ansteigt. Da die Anfragen während der Messungen einen Timeout von 3 Sekunden besitzen, ist irgendwann der Punkt erreicht, an dem die Dauer je Anfrage diesen Timeout überschreitet. Die Scheduler in *tokio* und *async-std* führen eine weniger gleiche Verteilung der CPU-Zeit auf die Tasks durch, sodass auch bei 400 parallelen Verbindungen noch einige Anfragen innerhalb des Timeouts beantwortet werden. Gleichzeitig führt die Verteilung dazu, dass bereits früher erste Anfragen den Timeout erreichen und der Durchsatz somit abnimmt.

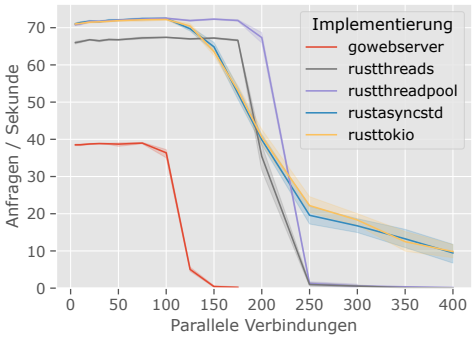

Abb. 6: Durchsatz für den /fibonacci/37-Endpunkt mit Keepalive

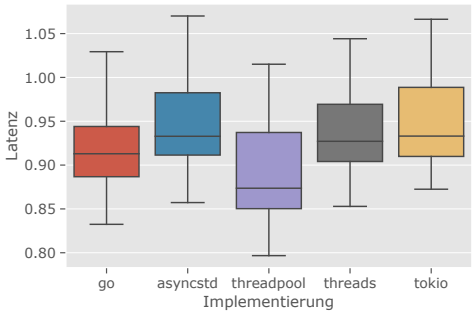

Abb. 7: Latenz für den /hello-Endpunkt bei einem Durchsatz von 1/s ohne Keepalive

Abbildung 7 zeigt die Ergebnisse der Latenz-Messungen in Millisekunden je Anfrage. Die Threadpool-Implementierung erreicht die besten Ergebnisse. Das ist dadurch begründet, dass kein neuer Thread oder Task erstellt, sondern die Verbindung einem bestehenden Thread zugewiesen wird. Die Go-Implementierung erreicht bessere Ergebnisse als die naive Rust-Implementierung, während die asynchronen Bibliotheken die höchsten Latenzen besitzen. Das Starten von Tasks in den asynchronen Bibliotheken benötigt zwar weniger Zeit als das Starten eines Threads, dafür benötigen die Tasks länger, bis sie tatsächlich durch den Scheduler ausgewählt und anschließend ausgeführt werden, sodass es hier bei einer geringen Last zu höheren Latenzen kommt.

Abbildung 8 visualisiert den maximalen Arbeitsspeicherverbrauch in MB in Abhängigkeit der Anzahl an parallelen Verbindungen. Da die Threadpool-basierte Implementierung unabhängig von der Anzahl an Verbindungen 10.000 Threads initialisiert, ist der Arbeitsspeicherverbrauch bei 50 parallelen Verbindungen mit fast 100 MB mit Abstand am größten. Dafür steigt der benötigte Arbeitsspeicher mit der Hinzunahme von weiteren Verbindungen nur marginal an, sodass diese Implementierung bei 10.000 parallelen Verbindungen maximal 113 MB benötigt. Die übrigen Implementierungen besitzen ein nahezu lineares Wachstum. Die Implementierung, die Threads nach Bedarf startet, beginnt bei etwa 7 MB und endet fast auf dem gleichen Niveau wie die Threadpool-basierte Implementierung.

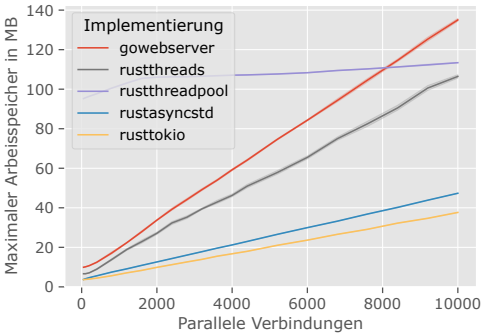

Abb. 8: Arbeitsspeicherverbrauch beim Aufruf des /hello-Endpunkts mit Keepalive

Die Go-Implementierung startet bei etwa 10 MB und benötigt bei 10.000 Verbindungen mit 135 MB somit mehr als die Thread-basierten Implementierungen. Das könnte einerseits dadurch begründet sein, dass Go-routinen einen – zumindest gegenüber den Tasks in *async-std* und *tokio* – größeren Arbeitsspeicherverbrauch besitzen. Andererseits besitzt Go einen Garbage Collector, sodass eingegangene Anfragen und zugehörige Antworten nicht unmittelbar nach dem Senden der Antwort freigegeben werden. Die asynchronen Rust-Bibliotheken starten bei 4 MB und benötigen bei 10.000 Verbindungen 38 beziehungsweise 47 MB. Diese Implementierungen besitzen somit den geringsten Arbeitsspeicherverbrauch, wobei *tokio* etwas ressourcenschonender ist.

7 Fazit und Ausblick

Es wurde gezeigt, dass sowohl Rust als auch Go für die Implementierung von Webservern geeignet ist. Während Go bei ausschließlicher Verwendung der Standardbibliothek auch bei deaktiviertem Keepalive und vielen parallelen Verbindungen gute Ergebnisse erzielt, schneidet die naive Rust-Implementierung, die für jede Verbindung einen neuen Thread startet, deutlich schlechter ab. Hier kann mithilfe eines Threadpools entgegengewirkt werden, der jedoch in der Standardbibliothek nicht enthalten ist. Bei Anfragen mit aktiviertem Keepalive führt die Verwendung eines Threadpools nicht zu besseren Ergebnissen als bei der naiven Implementierung. Zur weiteren Optimierung können externe Bibliotheken wie *async-std* oder *tokio* verwendet werden. Während *async-std* ähnliche Ergebnisse wie die Go-Implementierung erreichte, schnitt *tokio* bei den Messungen am besten ab. Die Verwendung eines Threadpools führt dazu, dass auch bei nur wenigen Verbindungen deutlich mehr Arbeitsspeicher benötigt wird. Die asynchronen Rust-Implementierungen besitzen den geringsten Arbeitsspeicherverbrauch, während die Go-Implementierung mehr Arbeitsspeicher als die naive Rust-Implementierung benötigt.

In dieser Arbeit wurde eine sehr rudimentäre HTTP-Implementierung umgesetzt. Gleichzeitig wurden ausschließlich einfache Endpunkte für die Messungen verwendet. In zukünftigen Arbeiten könnte eine umfangreichere Implementierung umgesetzt oder eine bestehende Bibliothek eingesetzt werden, wobei die Vergleichbarkeit zwischen Rust und Go gegeben sein muss. Zudem könnten auch Verbindungen mit SSL-Verschlüsselung genutzt werden, was die benötigte Rechenleistung je Anfrage erhöhen würde. Ebenso könnten weitere Anwendungsfälle wie POST-Anfragen oder der Aufruf von externen Diensten verprobt werden.

Während unterschiedliche Implementierungen in Rust unter Zuhilfenahme von externen Bibliotheken umgesetzt wurden, basiert die Go-Implementierung auf sehr grundlegenden Sprachfunktionen. Hier könnten Optimierungen vorgenommen werden. So wäre es beispielsweise möglich, zu Beginn eine gewisse Anzahl an Goroutinen zu starten, an die die eingehenden Verbindungen per Channel übertragen werden. Ebenso könnte der Vergleich auf weitere Programmiersprachen ausgeweitet werden.

Literatur

[HB20] Heyman, H.; Brandefelt, L.: A Comparison of Performance & Implementation Complexity of Multithreaded Applications in Rust, Java and C++, 2020.

[Je22] Jensen, D.: Recursive Fibonacci Benchmark using top languages on Github, 2022, URL: https://github.com/drujensen/fib, Stand: 23. 06. 2022.

[KN19] Klabnik, S.; Nichols, C.: The Rust programming language. No Starch Press, San Francisco, 2019, ISBN: 1-7185-0044-0.

[PGF19] Pieper, R.; Griebler, D.; Fernandes, L. G.: Structured Stream Parallelism for Rust. In: Proceedings of the XXIII Brazilian Symposium on Programming Languages - SBLP 2019. ACM Press, New York, New York, USA, S. 54–61, 2019.

[Ru18] Rust Foundation: Asynchronous Programming in Rust, 2018, URL: https://rust-lang.github.io/async-book/, Stand: 30. 06. 2022.

[Sc21] Schmaus, F.; Fischer, F.; Hönig, T.; Schröder-Preikschat, W.: Modern Concurrency Platforms Require Modern System-Call Techniques. Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Erlangen, 2021.

[SM17] Scionti, A.; Mazumdar, S.: Let's Go: a Data-Driven Multi-Threading Support. In (Giorgi, R.; Becchi, M.; Palumbo, F., Hrsg.): Proceedings of the Computing Frontiers Conference. ACM, New York, NY, USA, S. 287–290, 2017.

[ST12] Serfass, D.; Tang, P.: Comparing parallel performance of Go and C++ TBB on a direct acyclic task graph using a dynamic programming problem. In (Smith, R. K.; Vrbsky, S. V., Hrsg.): Proceedings of the 50th Annual Southeast Regional Conference on - ACM-SE '12. ACM Press, New York, New York, USA, 2012.

[St22] Stack Exchange Inc.: Stack Overflow Developer Survey 2022, 2022, URL: https://survey.stackoverflow.co/2022/, Stand: 28. 06. 2022.

[VCT18] Valkov, I.; Chechina, N.; Trinder, P.: Comparing languages for engineering server software. In (Haddad, H. M.; Wainwright, R. L.; Chbeir, R., Hrsg.): Proceedings of the 33rd Annual ACM Symposium on Applied Computing. ACM, New York, NY, USA, S. 218–225, 2018.

[YSZ19] Yu, Z.; Song, L.; Zhang, Y.: Fearless Concurrency? Understanding Concurrent Programming Safety in Real-World Rust Software, 2019.

Comparison of the programming languages Rust and Python using the example of prototypical implementations of graphs in different use cases

Kai-Luka Buchkremer[1]

Abstract: We have often encountered graphs in our daily lives without realising it. Graphs are ubiquitous and yet there are new technologies that have not been looked at closely. In this paper, the graph packages Petgraph, NetworkX and RetworkX are presented, analysed and compared. So far, no comparison of these three graph implementations has been attempted. The question that arises is: Will Petgraph perform better than the supposedly slower Python library NetworkX? And how does the latest innovation in graph libraries RetworkX perform? RetworkX comes from the field of quantum computing and is a novelty in the field of graph implementations. To be able to answer these questions, concrete workload parameters were defined to achieve different measurement results. It was found that RetworkX performs best by far, but also requires the most resources. Petgraph is clearly inferior to the other two graph libraries in terms of performance.

Keywords: Petgraph; NetworkX; RetworkX; Rust; Python; Graphtheory; Graph; Graph-Algorithms

1 Introduction

Nowadays, the importance and size of data volumes are increasing. This importance can also be recognised in the resulting analyses of this data. One possible representation of large amounts of data are so-called networks. These networks can, for example, consist of different numbers, subgraphs or graphs. Graphs exist in a wide variety of forms and differ depending on the use case. Some examples of classic use cases are social networks, recommendations, scientific computing or path optimisation in the form of navigation services, as we know them from Google Maps, for example.[Ma21]
This paper deals with the question, which graph implementation is quantitatively and qualitatively the most suitable for prototypical larger graph applications.

1.1 Distinction from related work

In the research area of graph theory, there are a large number of scientific papers that have dealt with a wide variety of issues. For the purpose of delimitation, some scientific works are

[1] University of Applied Sciences Münster, Department of Economics, Business Information Systems, kb666774@fh-muenster.de

presented below. The first paper deals with the comparison of five graph packages based on the qualitative metric of execution time. One of the graph packages compared is NetworkX. However, it does not consider RetworkX or Petgraph. Real world data sets from Facebook and Amazon, among others, were used to determine benchmarks via meaningful large graphs. The datasets used have hundreds of thousands to millions of nodes and are tested based on different algorithms. This includes not only the analysis of the shortest paths, but also PageRank algorithms, for example. [Li20] Another paper deals with RetworkX. There RetworkX is placed in the technical stack of already existing graph packages. The basic concept of RetworkX is distinguished from NetworkX and other graph implementations. Benchmarks or similar qualitative metrics are not compared. This paper was written by some contributors working on RetworkX and is also intended to serve as an introduction for those interested in RetworkX. [Tr21]
Film stars and their roles in films play a significant role in the next paper. This paper uses the NetworkX library to investigate which actors are the most significant in their business. Different algorithms are used for this purpose. These include calculating the shortest path or determining actors with the most films. Finally, however, no benchmarks or comparisons to other graph libraries are made in this work. [Le20]
There are a number of other papers that deal with different graph implementations, especially NetworkX has been studied many times. [HSC08] However, there are rarely benchmarks, let alone scientific papers, that compare Petgraph, NetworkX and RetworkX quantitatively and qualitatively.

2 Basic graph principals

In this section, the technical basics of graphs are explained first. Then the three graph implementations are briefly presented. A graph G consists of a set of nodes V also called vertices and an edge set E sometimes also called arcs. An edge is an unordered pair of two different nodes from V. The formal definition of a graph then looks as follows: $G = (V, E)$. [To22]
A direct graph or digraph also has d in addition to V and E. d indicates whether V is the source and E is the end. Thus d indicates the direction of the increase in nodes and edge sets. Thus the formal definition of a direct graph would be: $G = (V, E, d)$. [To22]
Graphs can be represented in different forms, a very well known and popular one is the adjacency matrix. If M is the matrix of G , then it is a symmetric 0/1 matrix, with the rows and columns corresponding to the vertices of the graph G .
A path P describes consecutive pairs of vertices which are adjacent. P must always consist of at least one vertex. The formal definition is: $V(P) = x1, , xt \subseteq V(G)$. Here $x1$ or xt are the start and end edges, where the path length is equal to the number of edges. Algorithms that search for the shortest path use the start and end edges and calculate the path length based on the number of edges. [Ul10]

2.1 Brief presentation of the graph implementations

Since NetworkX has already been presented and covered in many scientific papers or blog posts, an introduction will be omitted.
However, since RetworkX is very new and hardly anyone knows it, it will be briefly introduced, as will the Rust Crate Petgraph. Petgraphs most current version is the 0.6.2, published in May 2022. The first commit and the start of the project is dated 7 years ago in 2015. It has reached nearly 23 million downloads (state from June 2022) and is the leading graph data structure library written in Rust. [Pe22]
RetworkX is a graph library for any Python application, but written in Rust. This was made possible by py03 and setuptools-rust.[2]
The origin of RetworkX goes back to Qiskit, an open-source SDK for working with quantum computers.[3] The developers wanted to implement NetworkX more performant and thus reinvent the data structure on which qiskit is based, namely a direct acyclic graph. This idea eventually led to the implementation of RetworkX. [Re22]

3 Structural comparison

In the following sections, the three graph libraries are qualitatively differentiated from each other. For this purpose, they are compared with each other in different areas such as supported graph formats and graph algorithms, as well as the level of support to help developers make progress.

3.1 Graph types

NetworkX has a total of four different graph classes, one class each for distinguishing between undirect graphs and direct graphs. In addition, there are multigraphs that can store multiple edges between two nodes. One last class of graphs is deprecated and will not be supported in the future. [Ne22]
RetworkX distinguishes between two different graph types, one for directed and one for undirected graphs. To ensure backward compatibility, there is an additional class for direct acyclic graphs, called DAG for short. However, this is no longer used explicitly since version 0.4.0 and can be used analogously to PyDiGraph. [Re22] Petgraph distinguishes between several graph types. First, there is the class Graph and StableGraph. N stands for nodes and E for edges, also called weights. The latter ensures that when nodes are deleted, the indices remain stable and move with them. In addition, the maximum size of the graph can be specified. In contrast to the graph, the class GraphMap does not use an adjacency list graph as a data structure, but a hash map that stores the node identifiers as key values. This

[2] //github.com/pyo3/pyo3
[3] https://qiskit.org/

makes it possible that node identifiers are not necessarily required for operations such as Petgraph. A special feature of Petgraph is certainly the Compressed Sparse Row (CSR), which neither of the two graph implementations explicitly has. [Pe22]

3.2 Graph algorithms and functions

Since there are several algorithms on the NetworkX side which are often not implemented in RetworkX or Petgraph, more emphasis is placed on the distinction between Petgraph and RetworkX in the following.
RetworkX does not have an equivalent implementation of a Bellman-Ford shortest path algorithm. Petgraph and RetworkX do, however. [Re22]
Petgraph has several shortest path algorithms implemented. These include Astar, Bellman-Ford, Dijkstra, Floyd-Warshall and k-shortest-path. In addition, Petgraph has four ismorphism functions that check whether a graph or subgraph is identical. In total, there are only eleven algorithm modules that can be applied, so that the possibilities of analyses with Petgraph are very limited. [Pe22] Petgraph is not capable of generating algorithms, nor does Petgraph have any possibilities to import or convert graphs directly. Because Petgraph does not provide native support for importing, generating or reading in a graph, there are no meaningful benchmarks with larger graphs.[4]
RetworkX allows NetworkX graphs to be converted and then processed. NetworkX has many functions to generate graphs. It does not matter whether the graph is very large or rather small. The user also has the option of generating random graphs. It is even possible to generate social networks or graphs from the community. [Ne22] One thing in common, however, is that all three can draw the graphs using the Graphviz library or display them using dot.

3.3 User-friendliness

RetworkX has a much smaller community, hardly any websites or help is available, apart from the documentation. That is why there are only a few scientific works on this topic. This is not least due to the fact that RetworkX is still very new.
NetworkX, on the other hand, has a large community, mainly because the library has been around for over 17 years. This means that the documentation, as well as the help available on the internet, is very large. This makes it easier, especially for beginners, to get to grips with the subject and to make progress.
Petgraph is a popular Rust Crate and has good, but sometimes confusing documentation. There is some information on the internet, but much less than on NetworkX. It can therefore be said that NetworkX was able to gain an advantage in the qualitative metric of user-friendliness.

[4] Github-Issue: https://github.com/petgraph/petgraph/issues/422

4 Measurement concept

The following sections explain the design of the performance analysis, as well as the different parameters and the choice of measurement values.

4.1 Design of the performance analysis

After presenting the basis of graphs, the following figure shows the measurement concept graphically in a UML deployment diagram.

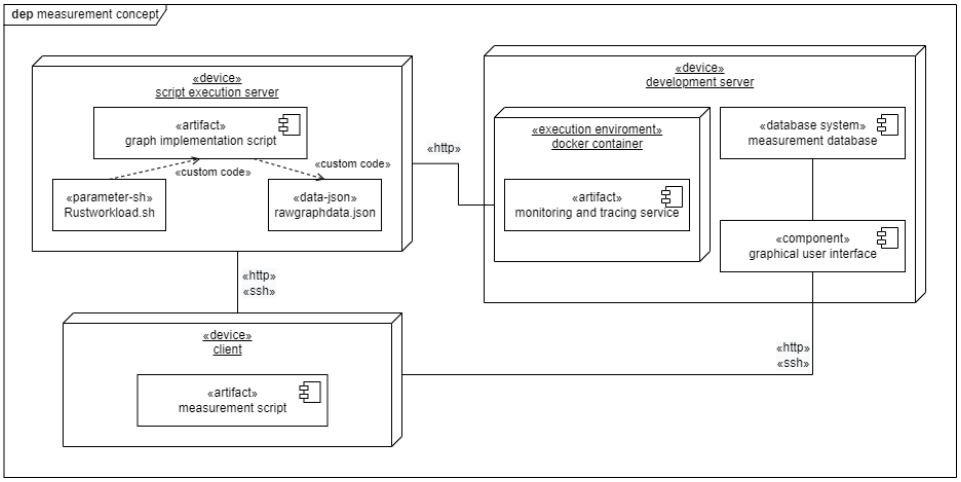

Fig. 1: Deploymentdiagram of the measure conecpt

As can be seen in the diagram, there are two virtual Linux servers that are accessible in the worldwideweb and can communicate via SSH. They are hosted by a cloud provider. The scripts for the graph implementations, including the JSON files to generate the graphs, are located on the script execution server. In addition, shell scripts were created that contain parameters such as the path of the JSON files, the type of workload or even the type of algorithm. These shell scripts allow the measurement script to execute specific code, adapted to the client's request. The measurement script is executed from the client and then triggers the execution of the desired workload. At runtime, the metrics related to RAM and CPU usage are then determined and written away per second to the Mongo DB, which resides on the second virtual server. In addition to these metrics, the required execution time is also determined and stored in the database.
To be able to monitor program execution, a Docker container exists on the script execution server that outputs metrics. On the development server, on the other hand, there is also a Docker container that accesses these metrics via an interface using Prometheus.[5] Grafana

[5] https://prometheus.io/

prepares this data and presents it in a customisable dashboard together with a Docker container running on the virtual machine.[6] Among other things, this enables the analysis of logs to simplify tracing. Furthermore, selected metrics can be monitored on an additional dashboard.
In order to enable the reproducibility of the measurement results shown later, [Ja91] the individual components of the measurement concept will now be briefly described in more detail. Both virtual Linux machines have 6 CPU vCore with 16 GB Ram and an SSD of 500 GB. The operating system used is Ubuntu 20.04 focal. One Linux machine has a GUI called KDE Plasma. IntelliJ 2022.01 was used as the IDE, both on the client side and on the development server. The database is a Mongo DB with the latest version 5.0. Studio 3T was used to display and manage the data in the database.
The measured values of the executions are stored in the database. On the one hand, the total execution time from the start of the programme to the end is to be measured. In addition, CPU and RAM are read out as measured values. For the CPU, a distinction is made between the user and the system. When measuring the RAM values, a distinction is made between free and used memory. In addition, the RAM buffer in cache is measured.

4.2 Workload parameters

In order to be able to carry out a performance measurement successfully, parameters are required that influence the performance of the system as an factor with different levels and thus also the result of the measurement series. The system parameters are kept static and can therefore be neglected for further consideration. The reason for this is that the factors of the workload parameters are influential enough. Otherwise, a larger level of the factors could lead to a malfunction of the executing system. [Ja91]
The following table shows the workload parameters as factors including their levels:

Factor	Level
Implementations	NetworkX, RetworkX, Petgraph
Algorithms	Astar, Dijkstra, Floyd Warshall
Graphs	Flightroutes, Social Network
Size	[50.00,140.000], [100.000, 1.000.000]

Tab. 1: Factors and their associated levels

In the following the size and the workload of the graphs are presented, since the languages, graph implementations and algorithms were discussed earlier. Two use cases were implemented as graphs: social network and flightroutes. These are synthetic workloads whose characteristics are adapted to the real world, but can be repeated again and again. [Ja91] For the generation of the synthetic workloads, real world data was required for the flightroutes as well as for the social network. These were used in the context of this scientific work from

[6] https://grafana.com/

already existing sources of the internet. For the flightroutes, a data source with 140,073 city names including the corresponding longitude and latitude was used.[7] Based on this master data, a JSON was generated using a calculation that can approximately determine the distance between two coordinates, which has all the following characteristics:

```
{"result": [{"name":"Münster","lat":"51.9615","lng":"7.6282","flightroutes"[
{"departure":"Münster","destination":"Düsseldorf","flightdistance":99}]
```

Fig. 2: Example data set from the generated flightroutes JSON file

To save memory space, the keys shown in the figure, e.g. destination, have been abbreviated to "ds". The flight distance between two cities is given in kilometres. This information is used to weight the edges of the graph. It allows path algorithms to calculate and output the flight distance. Based on this JSON, all cities within a radius of 100 kilometres of each other were connected, resulting in a graph with more than a million edges. For the social network graph, real world data from the internet was also used as the basis for the JSON generation.[8] It contains 500 male and 500 female first names and 2000 surnames. These were randomly mixed with each other in order to randomly generate whole names. The persons generated in this way now had a unique name. Now it was determined for this synthetic workload that each of these persons had between zero and twenty friends. This resulted in a graph with over 7.4 million edges considering the biggest size of the workload parameter size.
The factors just described are now sent to the system as a load profile in every possible combination to generate different results.

4.3 Measuring procedure

Before the measurement results are presented and discussed, the following describes what exactly is measured and what exemplary return values are obtained when these workloads are executed. All three workloads deal with the search for the shortest path based on different algorithms. The factors including the levels have already been presented in table 1. The three search algorithms were implemented for all three graph libraries. One exception is the Astar algorithm, which was not implemented in RetworkX. The factor size given in table 1 was executed nine times for each graph library with the Dijkstra and Astar algorithms and the two graphs. The Floyd Warshall algorithm was run with a smaller size factor ten times each, as it previously broke at a larger factor. The reason for this was that Floyd Warshall calculates and returns the length of the shortest paths for all pairs of vertices. To clarify the process, the following is the example command for calling the Astar algorithm in Rust.
The call shown here is exactly identical to the one in the measurement script. It is clear here that the shortest route between the city of Münster in Germany and the city of Enonski in Finland is being searched for. The third parameter in the algorithm call aggregates the edge

[7] https://github.com/lutangar/cities.json
[8] https://github.com/ndsvw/JSON-Namen

```
astar(&depth, "Münster", |finish| finish == "Enonkoski", |e| *e.weight(), |_| 0)
```

Fig. 3: Astar call in Python project

weights. The last parameter is for the estimatecost function and was left empty. Here, one could possibly include a function that estimates the costs when reaching the destination node. If you display the result of the call shown above on the console, you will get the costs for the shortest path as well as an array with the sequence of cities to the destination.
The same applies to the social network graph, but here the names are searched for. The result is again the cost of the path, as well as all the vertices that are traversed to the destination.

5 Measurement results

In the following figures, the total execution time required is given in seconds. On the x-axis are the different influencing factors, consisting of the algorithm and the size of the graph. The words small and large in the ash label refer to the graph size used see table 1. It is assumed that all measurement results are normally distributed, an additional test is waived.

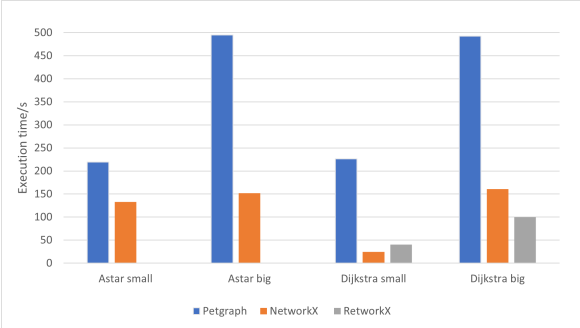

Fig. 4: Measurement results based on flightroutes Graph

The measurement results from nine repetitions for each influencing factor were adjusted by calculating the median value. The bar chart quickly shows that Petgraph has the highest execution time in seconds, even with all influencing factors. With almost 500 seconds execution time for the large workload using Astar and Dijkstra. There, the gap to NetworkX is around 350 seconds. It is also clear that the execution time for RetworkX is significantly lower than that of NetworkX for the large workload with Dijkstra. RetworkX is 61 seconds faster, but 16 seconds slower for the smaller workload.

Overall, it can be said that the execution time correlates with the size of the graph, but this was to be expected.
The following diagram shows the measurement results based on the Social Network Graph.

Petgraph is again the slowest library. However, the distances to NetworkX are significantly smaller than before. For the larger workloads, the median times are 21 and 36 seconds. NetworkX also behaves similarly to before. The gap to RetworkX using Dijkstra and the graph with 1 million nodes is considerable and takes over 65 seconds. All three libraries are close to each other with the smaller Social Network Graph using Dijkstra's algorithm.

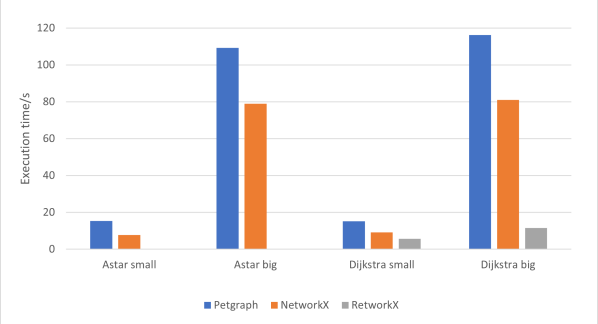

Fig. 5: Measurement results based on Social Network Graph

In addition to the measurements just described, another algorithm was also implemented. The Floyd-Warshall algorithm searches for the shortest path between all pairs of nodes of a graph. As a result, the algorithm's effort is much higher than the previously discussed algorithms. As a result, the graph size had to be adjusted. In concrete terms, this means that there are now only 500 nodes for the social network and 10 for the flightroutes. The number of edges, however, is much higher at a few hundreds to several thousand for the flightroutes. Because all cities within a radius of 100km are connected, the flightroutes graph has significantly more edges and is therefore larger. The result of the measurements with the Floyd-Warshall algorithm is shown below. A total of ten runs were made per graph library and graph.

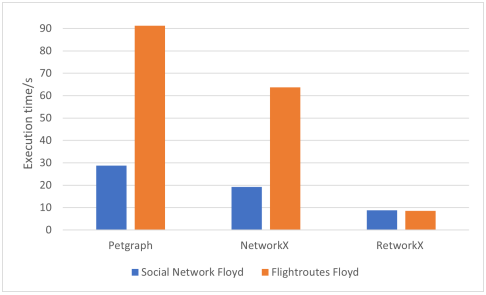

Fig. 6: Measurement results using the Floyd-Warshall algorithm

With the Floyd-Warshall algorithm, a similar picture emerges as with the other measurements. Petgraph performs worst with over 30 seconds more execution time per second in the median than NetworkX for the flightroutes graph. The situation is very similar for the Social Network Graph. Petgraph is about ten seconds slower there. RetworkX is again the fastest.

It is particularly impressive that despite the different sizes of the graphs, the execution time is almost identical.

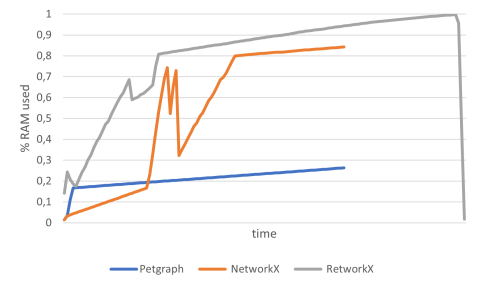

Fig. 7: Percentage of used RAM in flightroutes using the Dijkstra algorithm

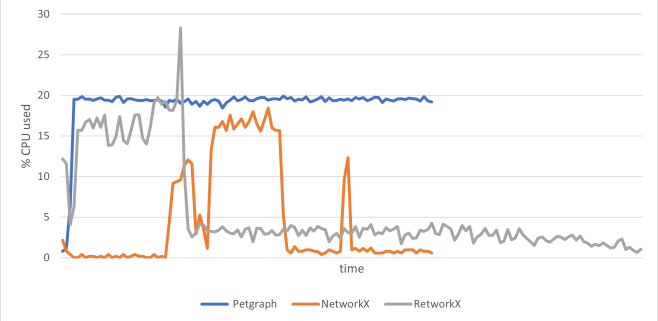

Fig. 8: Percentage of used CPU by the user in flightroutes using the Dijkstra algorithm

The two figures show the RAM and CPU required by the user. The RAM and CPU have both been selected from the same runtime and illustrate the load that RetworkX and NetworkX in particular can cause. RetworkX requires the entire RAM of 16 GB at the end of the calculation. Petgraph requires almost a constant percentage of RAM. In the case of NetworkX it is also clear from the kink, where the graph was loaded, the calculation of the algorithm was running. With the CPU used by the user, you can also see that Petgraph requires a constant amount of power and that there are hardly any peaks to be seen. After an initial high CPU demand, it levels off very quickly at RetworkX and settles between three and five.

6 Discussion of results

This scientific work has dealt with the three graph libraries Petgraph, RetworkX and NetworkX.
The aim was to find out which graph library is suitable for certain use cases, taking into account qualitative and quantitative metrics. The result is that beginners in particular should use NetworkX, as it has clear documentation and many papers or blogposts exist on the internet. Anyone who can do without a little performance is in good hands with NetworkX. A small drawback is the high RAM requirement. Users for whom performance is important can use RetworkX. It is the fastest and most efficient library when it comes to performance measurements. However, RetworkX also requires the most resources. Due to the novelty of this library, not many practical use cases are known yet. Petgraph was not convincing in either qualitative or quantitative metrics.
Due to the limited time, unfortunately not all topics could be implemented. It would have been conceivable to implement and compare even more algorithms. It would also be worthwhile to investigate other types of graphs. Nevertheless, the measurements were successfully planned, executed and evaluated.
It was ensured that all measurements are reproducible for interested parties, inquisitive minds or other students. Thus, further research can be done on the basis of this scientific work.

7 Conclusion and outlook

In summary, this work has highlighted the relevance of the topic and the innovations in the field of graph theory.
The introduction first laid the foundation for an understanding of graphs and their associated algorithms. Then, the three graph libraries were compared structurally. This included, among other things, the algorithms and functions, as well as the different characteristics of the graph types. The measurement concept was then described in detail. The structure was represented by a deployment diagram. The workload parameters for the measurement series were then defined. After an exemplary measurement was illustrated, the measurement results were presented and highlighted. In addition to the execution time, the utilisation of the RAM and the CPU were also taken into account.
In the future, the influencing factors size, graph and algorithms will be expanded. It makes sense to investigate further algorithms on different graphs. In addition, system parameters could also be included as factors, e.g. by implementing parallel computing.

References

[HSC08] Hagberg, A.; Swart, P.; Chult, D.: Exploring Network Structure, Dynamics, and Function Using NetworkX. In. Jan. 2008.

[Ja91] Jain, R.: The Art of Computer Systems Performance Analysis. In: Techniques for Experimental Design, Measurement, Simulation and Modeling. Wiley, Berkeley, CA, 1991, ISBN: 0-471-50336-3.

[Le20] Lewis, R.: Who is the Centre of the Movie Universe? Using Python and NetworkX to Analyse the Social Network of Movie Stars. CoRR abs/2002.11103/, 2020, arXiv: 2002.11103, URL: https://arxiv.org/abs/2002.11103.

[Li20] Lin, T.: Benchmark of popular graph/network packages v2, 2020, URL: https://www.timlrx.com/blog/benchmark-of-popular-graph-network-packages-v2.

[Ma21] Manasvi Aggarwal, M. M. In: Machine Learning in Social Networks. Springer Singapore, 2021, ISBN: 978-981-33-4022-0.

[Ne22] Developers, N.: NetworkX - Network Analysis in Python, 2022, URL: https://networkx.org/documentation/stable/reference/index.html.

[Pe22] Developers, P.: Petgraph Documentation, 2022, URL: https://docs.rs/petgraph/latest/petgraph/.

[Re22] Developers, R.: RetworkX Documentation, 2022, URL: https://qiskit.org/documentation/retworkx/.

[To22] Ton Kloks, M. X. In: A Guide to Graph Algorithms. Springer Singapore, 2022, ISBN: 978-981-16-6350-5.

[Tr21] Treinish, M.; Carvalho, I.; Tsilimigkounakis, G.; Sá, N.: retworkx: A High-Performance Graph Library for Python, 2021, URL: https://arxiv.org/abs/2110.15221.

[Ul10] Ulrich Knauer, K. K. In: Algebraic Graph Theory. De Gruyter, 2010, ISBN: 978-3-11-061612-5.

Vergleich von Circuit-Breaker-Implementierungen in Service-Meshes

Strukturelle und praktische Gegenüberstellung des Circuit-Breaking der Service-Meshes Istio und Traefik im Vergleich zur Bibliothek Resilience4j

Lennart Potthoff[1]

Abstract: Resiliente Mircoservices sind unabdingbar für moderne cloud-native Landschaften. Eines der am meisten verbreiteten Pattern für Resilienz ist der Circuit-Breaker. Service-Meshes sind mittlerweile in vielen Kubernetes-Clustern als Netzwerkabstraktionsschicht anzutreffen. In diesem Paper wird untersucht, in welcher Form die Service-Meshes Istio und Traefik Circuit-Breaking auf Infrastrukturebene umsetzen und wie sich dies vom klassischen Programmbibliothek-Ansatz am Beispiel von Resilience4j unterscheidet. Die Circuit-Breaker-Implementierungen werden in einem Praxistest verschiedenen Szenarien ausgesetzt und deren Reaktion auf Fehler- und Überlastsituationen ausgewertet. Darauf basierend kann eine Entscheidungshilfe für die Auswahl von Circuit-Breaker-Implementierungen gegeben werden.

Keywords: circuit breaker; service mesh; microservice resiliency

1 Einleitung

Moderne Anwendungslandschaften bestehen aus einer Vielzahl von Mircoservices, die über ein verteiltes Netzwerk kommunizieren. Für die Entwicklung und den Betrieb solcher Anwendungslandschaften sind cloud-native Technologien nicht mehr wegzudenken. Techniken, wie Container, Orchestrator oder Services-Meshes *„ermöglichen die Umsetzung von entkoppelten Systemen, die belastbar, handhabbar und beobachtbar sind.“*[CNCF18] In dieser Ausarbeitung soll sich speziell mit der Belastbarkeit – oder Resilienz – und deren Handhabbarkeit, die die cloud-native Technologie Service-Mesh bereitstellt, beschäftigt werden.

Unter dem Begriff Resilienz haben sich Maßnahmen zur Erhöhung der Belastbarkeit, Fehlertoleranz und Widerstandsfähigkeit bzw. zur Aufrechterhaltung der Verfügbarkeit bei Störungen von Softwaresystemen etabliert. Im Laufe der Jahre ist deutlich geworden, dass die alleinige Fokussierung auf die Fehlervermeidung in einem System mit unzähligen Diensten, die über unsichere Netzwerke kommunizieren, nicht zielführend ist [Fr16]. Vielmehr ist ein

[1] FH Münster, Fachbereich Wirtschaft (Wirtschaftsinformatik), Corrensstraße 25, 48149 Münster
lennart.potthoff@fh-muenster.de

resilientes Verhalten - die Fähigkeit des Systems mit unerwarteten Situationen umzugehen - gefordert. Erstmals hat dazu Nygard [Ny18] verschiedene *Stability-Pattern* beschrieben, die die Widerstandsfähigkeit und Stabilität von Softwaresystemen verbessern. In dieser Ausarbeitung wird sich dabei auf das Circuit-Breaker-Pattern fokussiert, welches als ein Schutzschalter fungiert und einen Service davor schützt, neue Anfragen anzunehmen, obwohl dieser bereits überlastet ist.
Konkret werden dazu verschiedene Umsetzungen des Circuit-Breaker-Pattern verglichen und anhand eines Fallbespiel in einem cloud-nativen Technologiestack implementiert und getestet. Die Vielzahl der heute gängigen Techniken bietet dabei verschiedene Ansatzpunkte zur Umsetzung von Circuit-Breaking. Hier sollen die Circuit-Breaker der beiden Service-Meshes Istio und Traefik und die Java-Bibliothek Resilience4j verglichen werden.

In dieser Ausarbeitung werden dabei die drei Implementierungen hinsichtlich struktureller Aspekte, wie Architektur, Funktionsumfang und Konfigurierbarkeit gegenübergestellt. Im zweiten Schritt werden die Implementierungen in einem praktischen Anwendungsbeispiel eingesetzt und die Wirkungsweise in verschiedenen Testszenarien unter Beweis gestellt. Abschließend wird ausgehend von der bisherigen Forschung, der strukturellen Unterschiede und des durchgeführten Praxistests eine Beurteilung und Entscheidungshilfe bei der Auswahl von Circuit-Breaker-Implementierungen gegeben.

2 Grundlagen

2.1 Circuit-Breaking

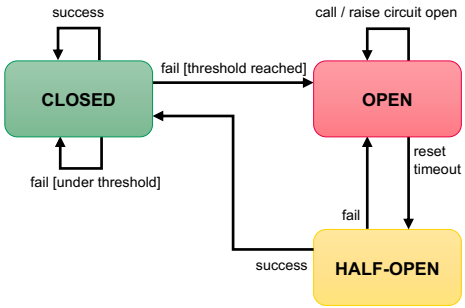

Abb. 1: Zustandsdiagramm des Circuit-Breaker-Patterns. Entnommen aus [Fo14].

Das Pattern Circuit-Breaker schützt davor, dass Clients lange auf fehlerhafte oder in Timeout laufende Anfragen warten. Ein bereits überlasteter oder gestörter Service wird nicht mit weiteren Anfragen überfordert. Eine Fail-Fast-Reaktion soll das Ausweiten der Störung hin zu einem kaskadierenden Fehler über mehrere Services verhindern [Fo14]. Das Pattern wird durch einen Zustandsautomaten mit drei Zuständen implementiert (vgl. Abbildung 1):

- **CLOSED:** Anfragen werden weitergeleitet und die geschützte Operation ausgeführt

- **OPEN:** Anfragen werden abgelehnt und die geschützte Operation nicht ausgeführt
- **HALF-OPEN:** versuchsweise werden Anfragen durchgelassen

Der Circuit Breaker beobachtet die eingehenden Anfragen und deren Fehlerrate. Wird ein vorher definierter Fehlerschwellwert innerhalb eines gewissen Zeitfensters überschritten, so schaltet der Circuit-Breaker in den OPEN-Zustand und ab dann werden keine Anfragen mehr an die geschützte Operation weitergeleitet. In diesem Fall antwortet der Service direkt mit einem Fehlercode oder es wird eine alternative Operation als fachlicher Fallback ausgeführt. Nach einer gewissen Erholungszeit wird in den HALF-OPEN-Zustand gewechselt und versuchsweise Anfragen durchgelassen. Anhand derer wird entschieden, ob der Service wieder störungsfrei funktioniert und wieder Anfragen annimmt oder nicht [Ny18].

Je nach Circuit-Breaker-Umsetzung kann auf unterschiedliche Störungen reagiert werden: Das Schalten des Circuit-Breaker kann bei zu vielen Antworten mit Fehlern (HTTP-Statuscode 5xx), bei Netzwerkfehlern oder bei einem zu langsamen Antwortzeitverhalten erfolgen.

Bei der Umsetzung des Circuit-Breaker-Patterns lassen sich nach Montesi und Weber [MW16] drei Varianten unterscheiden: Der *client-side circuit breaker*, bei dem jeder Client ein eigenen Circuit-Breaker für jeden angefragten Service nutzt. Beim *service-side circuit breaker* ist der Circuit-Breaker innerhalb des Service platziert z. B. durch Nutzung einer entsprechenden Bibliothek. Beim *proxy circuit breaker* ist keine clientseitige oder serverseitige Anpassung erforderlich, sondern der Circuit-Breaker wird zwischen die Kommunikationspartner geschaltet. Dabei kann sowohl jeweils nur ein Proxy für ein Service (vgl. Sidecar-Ansatz in Kapitel 2.2) oder ein Proxy für mehrere Services zuständig sein.

2.2 Service-Mesh

„Ein Service-Mesh ist eine spezielle Infrastrukturebene zur Handhabung der Service-zu-Service-Kommunikation.“[Li19] Diese Netzwerkabstraktion ermöglicht die Kontrolle des Datenverkehrs in modernen containerbasierten cloud-nativen Applikationen mit komplexen Topologien [KA19; Li19; PW+22]. Typischerweise werden dafür neben den Services leichtgewichtige Proxies installiert, um den Netzwerkverkehr - anders als bei eingebundenen Bibliotheken ohne Änderungen des Service-Codes - kontrollieren zu können [Li19; PW+22]. Gängige Kernfunktionalitäten sind Traffic Management, Resilienz, Beobachtbarkeit und Sicherheit [KA19]. Im Weiteren wird sich verstärkt mit dem Aspekt der Resilienz beschäftigt, worunter neben den Circuit-Breaking auch Funktionen, wie Timeouts, Retrys und Fault/Delay Injection fallen [PW+22].

Architektonisch verwenden die meisten Implementierungen das Sidecar-Pattern. Dabei wird neben jedem Service in einem weiteren Container ein Proxy installiert, durch den der Netzwerkverkehr durchgeleitet und kontrolliert wird. Das intelligente Zusammenspiel aller

Proxys wird als Data Pane bezeichnet. Eine weitere Infrastrukturebene ist die Control Pane. Diese erfasst empfange Metriken und steuert die Proxys [Li19].

Die beiden bekanntesten Service-Mesh-Implementierungen sind Istio und Linkerd2. Einen Überblick und Funktionsvergleich über eine Vielzahl an Service Meshes bieten die CNCF-Landscape und servicemesh.es.

2.3 Bisherige Forschung

Die bisherige Forschung lässt sich in die Themenbereiche Service-Mesh und Circuit-Breaker aufteilen:

Im Bereich des Service-Mesh ist das Paper von Li et al. [Li19] zu nennen, was als einer der ersten einen wissenschaftlichen Blick auf das Thema Service-Mesh lieferte.

Wie bereits unter 2.1 erläutert, wurde das Circuit-Breaker-Pattern erstmals von Nygard [Ny18] vorgestellt. Durch Montesi und Weber [MW16] wurde eine architektonische Kategorisierung von Circuit-Breaker-Implementierung erstellt. Falahah et al. [FSS20] geben einen Überblick über die vorhandene Literatur.

Im Zusammenspiel der beiden Themenbereiche ist das Paper von Sedghpour et al. [SKT21] zu nennen, indem ein adaptiver Circuit-Breaker-Algorithmus für Istio / Envoy ähnlich zu dem an das *TCP congestion control* angelehntes Vorgehen von Allen [Al20] entwickelt wurde. Außerdem wurden in einem weiteren Paper von Sedghpour et al. [SKT22] die praktische Anwendung vom Istio-Circuit-Breaker und Retries in einer größeren Mircoservice-Landschaft untersucht.

In Abgrenzung zum bisherigen Forschungsstand steht nicht die Optimierung des Istio bzw. Envoy-Circuit-Breakers im Mittelpunkt. Stattdessen sollen die vorhandenen „Bordmittel“ der jeweiligen Implementierung untersucht werden. Dabei soll entsprechend Montesi's Kategorisierung die Infrastrukturschicht-Umsetzung Service Mesh durch die zwei architektonisch unterschiedlichen Repräsentanten Istio und Treafik mit der Libary-Umsetzung Resilience4j verglichen werden.

3 Struktureller Vergleich

Im Folgenden werden die drei Implementierungen hinsichtlich struktureller Unterschiede verglichen und in Tabelle 1 gegenübergestellt. Als wichtigste Unterscheidungsmerkmale wurden neben den architektonischen Merkmalen die verschieden Störungstypen, vor dessen der Circuit-Breaker schützt, die Fallback-Möglichkeit, die Zustände und die Konfigurierbarkeit einschließlich der wichtigsten Konfigurationsparameter festgelegt. Eine grafische Aufbereitung des architektonischen Unterschiede bietet Abbildung 2.

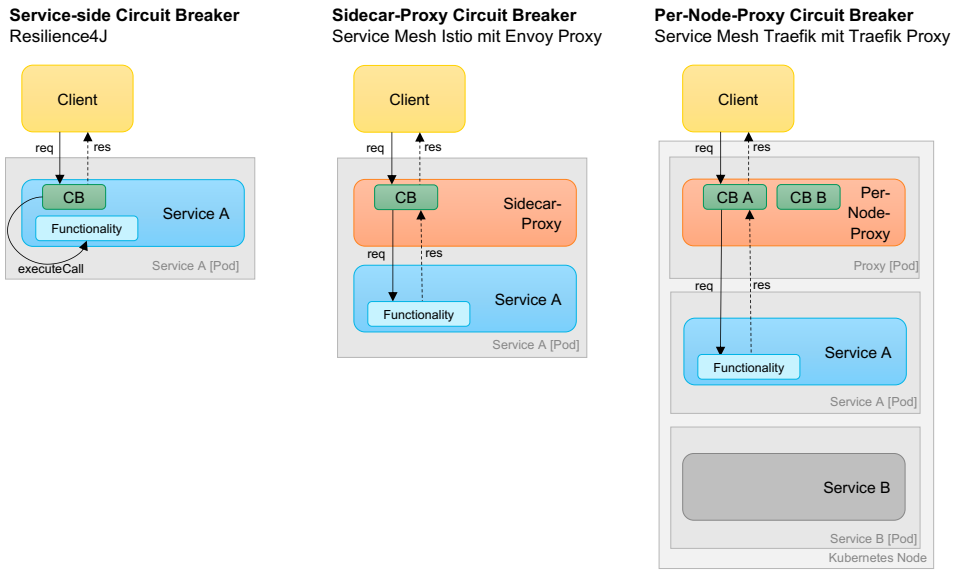

Abb. 2: Architektonische Gegenüberstellung der drei Circuit-Breaker-Implementierungen. In Anlehnung an [MW16].

3.1 Auswahl der zu vergleichenden Circuit-Breaker-Implementierungen

Bei der Auswahl der Circuit-Breaker-Implementierungen wurde versucht, möglichst unterschiedliche Ansätze zu vergleichen. Dabei wurde sich an Montesi's Kategorisierung orientiert, wobei auf einen Repräsentanten der clientseitigen Umsetzung verzichtet wurde, da die Ausstattung jedes Clients mit einem individuellen Circuit-Breaker in der Praxis kaum umsetzbar ist [MW16]. Als *service-side circuit breaker* wird die Java-Bibliothek *Resilience4j* herangezogen und mit den *proxy circuit breakern* zweier Service-Meshes verglichen. Als verbreitetes Service-Mesh wurde *Istio* ausgewählt, welches das übliche Sidecar-Pattern und den häufig anzutreffenden *Envoy*-Proxy nutzt. Zudem wurde ein Service-Mesh ausgewählt, welches einen anderen Proxy nutzt. Dabei viel die Wahl auf *Traefik Mesh*, da dieses - anders als *Linkerd2* - Circuit Breaking mit dem *Traefik Proxy* ermöglicht. Bei diesem Mesh wird kein Sidecar- sondern ein *Per-Node-Proxy* verwendet, wobei ein Proxy für alle Services eines Nodes bereitgestellt wird [PW+22].

3.2 Zusammenfassung des strukturellen Vergleichs

Die Tabelle 1 zeigt die wesentlichen Unterschiede der drei Circuit-Breaker-Implementierungen. Architektonisch bedingt ist die Java-Bibliothek Resilience4j nicht allein auf das Reagieren auf HTTP-Fehlercodes beschränkt, sondern kann auf jegliche

Circuit-Breaker	**Resilience4j**	**Istio**	**Traefik**
Architektur			
Technologie	Java-Bibliothek	Service-Mesh	Service-Mesh
Kategorie nach [MW16]	Service-side Circuit Breaker	Proxy with Circuit Breaker	Proxy with Circuit Breaker
Architektur	CircuitBreakerRegistry basierend auf in-memory ConcurrentHashMap	Sidecar-Proxy Envoy	Per-Node-Proxy Traefik
Einsatzgebiet	Jedes Java-Programm mit schützenswerten Codebereichen	HTTP-Services (in Kubernetes)	HTTP-Services (in Kubernetes)
Störungserkennung			
Fehler	Ja (Excecptions)	Ja (HTTP-Codes)	Ja (HTTP-Codes)
Überlast / Latenz	Ja	eingeschränkt	Ja
Netzwerkfehler	Nein	Ja (nicht getestet)	Ja (nicht getestet)
Fachlicher Fallback	Ja	Nein (nur 503)	Nein (nur 503)
Zustandsautomat	Standard (CLOSED, OPEN, HALF-OPEN)	kein HALF-OPEN, nur CLOSED und OPEN	Standard (CLOSED, OPEN, RECOVERING)
Konfigurierbarkeit			
Konfiguration	im Code / über Spring-Properties-Datei	DestinationRule in Kubernets-YAML	Kubernetes-Annotations
Fehlererkennung	Einstellbar über *failureRateThreshold*. Zu zählende Exceptions auch konfigurierbar.	Einstellbar über *consecutive5xxErrors*. Keine Fehlerrate sondern aufeinanderfolgende Fehler.	Einstellbar über *ResponseCodeRatio*.
Latenzerkennung	Einstellbar über *slowCallRateThreshold* und *slowCallDurationThreshold*.	Kein zeitlicher Schwellwert. Nur statische Parameter, wie *tcp.maxConnections* oder *http2MaxRequests*.	Einstellbar über *LatencyAtQuantileMS*.
Erkennen von Netzwerkfehlern	nicht unterstützt	Einstellbar über *consecutiveLocalOriginFailures* , *consecutiveGatewayErrors* oder statische Parameter.	Einstellbar über *NetworkErrorRatio*.
Timeout	Einstellbar über *waitDurationInOpenState*.	Einstellbar über *baseEjectionTime* (Timeout verlängert sich multiplikativ bei anhaltender Störung).	Einstellbar über *FallbackDuration*. Konfiguration erst seit dem Traefik Proxy Release v2.8 (29.06.2022) möglich. Bei den Tests mit v2.5 konnte der Standardwert von 10s nicht verändert werden.
Auswertungsfenster	Zeitbasiertes als auch anzahlbasiertes Fenster über *slidingWindowSize* konfigurierbar.	Nicht erforderlich, da konsekutive Zählweise.	Zeitbasiert, keine genaue Angabe zum Zeitfenster beschrieben.
Quelle	[R4J22]	[Is22] und [En22]	[Tr22]

Tab. 1: Struktureller Vergleich der Circuit-Breaker-Implementierungen

Fehlersituation, die in Form von Exceptions oder Latenzen im geschützten Codebereichen auftritt, reagieren. Zudem ermöglicht die Nähe zum Code, dass für den Fehlerfall sinnvolle fachliche Alternativen programmiert werden können. Des Weiteren zeichnet sich R4j durch seine umfangreiche Konfigurierbarkeit der Parameter aus.

Dem gegenüber stehen die beiden Service-Mesh-Implementierungen, die den Vorteil haben, dass keine Änderungen des Service-Code erforderlich sind und keine Anforderungen an den Technologiestack des Services gestellt werden. Im Störungsfall werden bei diesen die Nachrichten erst gar nicht an den überlasteten Service weitergeleitet und der Proxy übernimmt das Antworten mit HTTP-Statuscode 503. Die Konfiguration der Circuit-Breaker erfolgt bei beiden mit Kubernetes-Bordmitteln. Bei Istio wird die Fehlererkennung nicht mit einer prozentualen Fehlerrate definiert, sondern es ist die Anzahl aufeinanderfolgender HTTP- bzw. Netzwerkfehler anzugeben. Die Überlasterkennung ist nur eingeschränkt möglich, indem basierend auf Erfahrungswerten statische Lastparameter , wie maximale Requests oder TCP-Connections, für den Service definiert werden. Traefik hingegen ermöglicht das Definieren von unterschiedlichen Schwellwert-Fehlerraten für ResponseCodes, Netzwerkfehler und Latenzperzentils. Jedoch ist aus den Docs nicht eindeutig ersichtlich, welcher zurückliegende Zeitraum *(SlidingWindow)* für die Prüfung der Schwellwert-Raten ausgewertet wird.

4 Praxistest

4.1 System-under-Test des Praxisbeispiels

Zur Evaluierung der verschiedenen Circuit-Breaker-Implementierungen wird als Praxisbeispiel ein Fakultäts-Mircoservice[2], der mittels Java und Spring Cloud realisiert wurde, herangezogen. Neben der funktionalen Anforderung, per REST-Aufruf die Fakultät einer Zahl zu berechnen und auszugeben, wurden nicht-funktionale Anfoderungen (NFA's) hinsichtlich Resilienz formuliert. Die Anwendung soll sich fehlertolerant hinsichtlich Fehlern und Überlast verhalten. Wenn eine Störung erkannt wird, so soll zur Stabilisierung des Services die Ausführung der inneren Funktionalität für 30 Sekunden unterbunden werden. Stattdessen soll im Sinne des Fail-Fast eine HTTP-Antwort mit HTTP-Status 503 ausgegeben werden. Folgende Schwellwerte sind dabei zu berücksichtigen:

- **NFA-1 Konsekutive Fehler:** Nicht mehr als 5 aufeinanderfolgende fehlerhafte Antworten
- **NFA-2 Fehlerhäufung in Zeitfenster:** Innerhalb von 10 Sekunden nicht mehr als 50% fehlerhafte Antworten
- **NFA-3 Durchschnittliche Antwortzeit:** Im Durchschnitt ist die Antwortzeit (der letzten 10 Sekunden) nicht größer als 100 ms

[2] Repository mit Praxisbeispiel, Installations-, Testskripten und Testergebnissen verfügbar unter: `https://gitlab.com/fep22/resilience-comparison`

- **NFA-4 Antwortzeit des 90. Perzentil:** Die 10% langsamsten Anfragen (der letzten 10 Sekunden) sind nicht größer als 200 ms

Das Praxisbeispiel ist so implementiert, dass bewusst Exceptions ausgelöst werden können, die zu einem „500 Internal Server Error“ führen.

4.2 Konfiguration der Circuit-Breaker

Es wurde versucht, die drei Circuit-Breaker so zu konfigurieren, dass sie die nicht-funktionalen Anforderungen erfüllen. Für jeden Circuit-Breaker wurden sechs Konfigurationen erstellt: Die Default-Konfiguration, je eine Konfiguration pro NFA und eine Konfiguration, die versucht alle vier NFA's gleichwertig zu berücksichtigen.

Bei der Konfiguration bestätigen sich die ersten Erkenntnisse aus dem strukturellen Vergleich (vgl. Abschnitt 3). Die umfangreichen Konfigurationsparameter von Resilience4j und Traefik ermöglichen eine einfache Übersetzung der NFA's. Lediglich die Definition von konsekutiven Fehlern in Traefik ist aufgrund des unbekannten Auswertungsfenster nur indirekt möglich. Die Konfiguration von Istio abseits der konsekutiven Fehler ist herausfordernd, da keine Zeitspannen, Fehlerraten oder maximale Antwortzeiten angegeben werden können. Bei Istio's Überlasterkennung mussten deshalb statische Parameter (maximale Anzahl an TCP-Connections oder (ausstehenden) HTTP-Requests) festgelegt werden, ohne auf große Erfahrungswerte zum Lastprofil des neuentwickelten Praxisbespiels zurückgreifen zu können.

4.3 Testaufbau und -druchführung

Die durchzuführenden Tests sollen zeigen, wie gut die Circuit-Breaker die NFA's erfüllen und dadurch nachweisen, ob die Fehler- und Überlasterkennung der Circuit-Breaker funktioniert. Dabei werden drei verschiedene Szenarien von Störungen betrachtet: Permanente Störungen, die während des gesamten Testlaufs anhalten. Transiente Störungen, die nur für einen Ausschnitt des Testzeitraums auftreten. Sporadische Störungen, bei denen Anfragen mit einer gewissen Wahrscheinlichkeit fehlschlagen. Gemäß Äquivalenzklassenbildung ergeben sich daraus sieben Testfälle [SL12]: Ein Positivtestfall (keine Störungen), drei Testfälle zur Fehlererkennung und drei Testfälle zur Überlasterkennung. Zu jedem Testfall wurden ein erwartetes Verhalten hinsichtlich der HTTP-Status-Codes der Antworten und der Antwortzeiten definiert.

Zur Testdurchführung wird das Lasttesttool Fortio verwendet, welches pro Testfall für 120 s mit 10 QPS Anfragen an den Fakultäts-Service sendet. Basierend auf vorbereitenden Messungen wurden konstante Werte für weitere Parameter z. B. Workload festgelegt. Als Testumgebung muss ein MacBook Air mit 2,2 GHz Dual-Core Intel Core i7, 8 GB RAM

und macOS Big Sur 11.6.4 herhalten. Verwendet wird Docker Desktop 4.9.1, Minikube 1.26.0 (2 GB RAM, 2 CPUs), Istio 1.13.2, Traefik Mesh/Proxy 1.4.5/2.5, Java 11.0.15 und R4j 1.7.0. Die sieben Testfälle wurden für 19 verschiedene Konfigurationen durchgeführt. Insgesamt wurden somit über 150.000 Requests in ca. 4,5 Stunden Testdurchführungszeit generiert.

Die möglichen Seiteneffekte und Einschränkungen dieser Testumgebung und -skripte, die Festlegung auf ein konkretes Praxisbeispiel inkl. der damit verbundenen Technologiestack und die geringe Variation der Testparameter limitieren die externe und interne Validität dieser Untersuchung. Da bei diesem Test jedoch kein Performancevergleich der Circuit-Breaker, sondern die Überprüfung der Funktionserfüllung des Patterns im Vordergrund steht, sollte diese Limitierung eine geringe Auswirkung auf die Aussagekraft der Ergebnisse haben.

4.4 Ergebnisinterpretation

4.4.1 Validität: Da die R4j-Default-Konfiguration count-based und nicht zeitbasiert ist, waren bei diesem Testdurchlauf die Testfälle nicht hinreichend unabhängig voneinander, da eine zeitliche Pause allein den Zustand der Circuit-Breakers nicht zurückgesetzt hat. Die Testparameter der sporadischen Überlast wurden nur gering über den Schwellwert gelegt, weshalb dieser Testfall - auch aufgrund der zufällig generierten Verteilung - mit geringer Aussagekraft in die Interpretation einfließt.

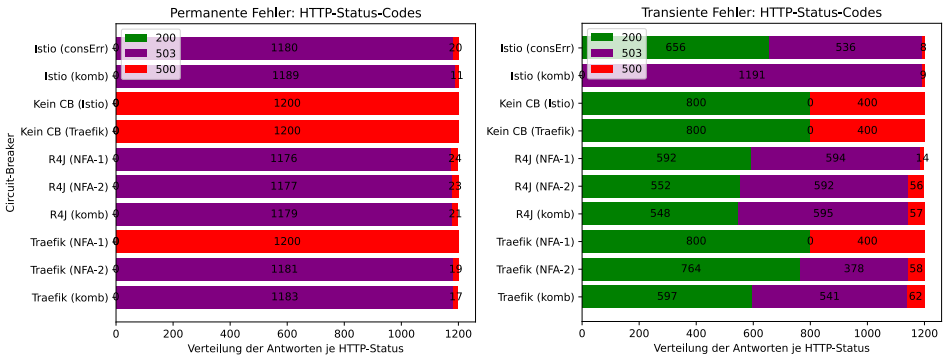

Abb. 3: Testergebnisse zu den permanenten und transienten Fehlern.

4.4.2 Fehlererkennung: Die Testergebnisse zeigen, dass sich für alle drei Circuit-Breaker Konfigurationen finden lassen, die sich zur Fehlererkennung eignen. Bei permanenten oder zwischenzeitlichen Fehleraufkommen eignen sich sowohl der konsekutive Fehlerschwellwert von Istio und R4j, als auch die 50%-Fehlerquote innerhalb von 10s von R4j und Traefik (vgl. Konfigurationen `Istio (consErr)`, alle `R4j`, `Traefik (NFA-2)` und `Traefik (komb)` in Abb. 3). Eine direkte Konfiguration von konsekutiven Fehlern in Traefik, wie auch das Erfassen einer Fehlerquote bei Istio ist nicht möglich. Die Testergebnisse zeigen aber, dass sich beide Konfigurationsarten zum Reagieren auf mehrere Sekunden andauernde

Fehlersituationen eignen. Die Ergebnisse des Testfalls „Transiente Fehler“ zeigen zudem, dass alle drei Implementierungen nach einer Übergangszeit bei Fehlerfreiheit in den CLOSED-Zustand zurückkehren.

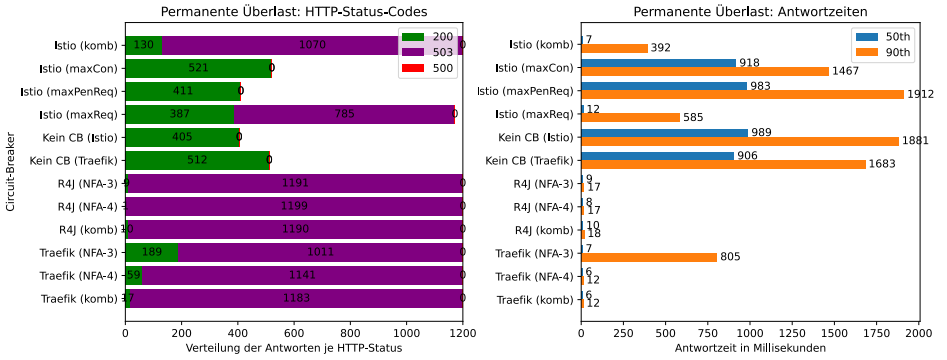

Abb. 4: Testergebnisse zum Testfall „Permanente Überlast“.

4.4.3 Überlasterkennung: Wenn kein Ciruit-Breaker aktiv ist, wird deutlich, dass dies in einer Überlastsituation zu extrem langen Antwortzeiten führt und bei permanenter Überlast sogar mehr als die Hälfte der Rückantworten zu den Anfragen erst gar nicht ankommen (vgl Abb. 4). Wie bereits in Abschnitt 4.2 vermutet, ist die Wahl von der maximalen Requests ohne Erfahrungswerte nicht sinnvoll, sodass die Istio-Konfigurationen `maxCon` und deren abgeleitete Kombination `komb` bereits am Positivtestfall scheitern. Ähnliches gilt für die anderen Istio-Konfigurationen, die sich nicht signifikant vom Verhalten des ausgeschalteten Circuit-Breaker unterscheiden.
Die Konfiguration `R4j (NFA-3)` zeigt eine gute Reaktion bei permanenter und transienter Überlast.
Die Testergebnisse bei Traefik verdeutlichen, dass das Setzen einer Circuit-Breaker-Expressions, wie `LatencyAtQuantileMS(90.0) > 200`, auch die versprochenen Wirkung zeigt: Alle drei Traefik-Konfiguration `NFA-3`, `NFA-4` und `komb` ermöglichen schnelle 503-Antworten bei permanenter, transienter und sporadischer Überlast.

5 Fazit und Ausblick

In dieser Ausarbeitung konnten die wesentlichen strukturellen Unterschiede der Circuit-Breaker-Implementierungen, wie Störungserkennung, Konfigurierbarkeit oder Architektur aufgezeigt werden. Trotz einiger Einschränkungen z. B. bzgl. der Testumgebung bei den durchgeführten Praxistest, konnten wichtige Erkenntnisse erzielt werden:
Für die Erkennung von mehrere Sekunden andauernde Fehlerzustände, eignen sich alle drei Circuit-Breaker-Implementierungen. Sollte z. B. in einem Kubernetes-Cluster bereits Istio aufgrund von anderen Service-Mesh-Funktionalitäten genutzt werden, so sollte die simple Ergänzung der `deployment.yaml`, um ein Schwellwert für konsekutive Fehler, genutzt

werden. Eine adäquate clientseitige Reaktion auf 503-Antworten vorausgesetzt, bietet dies einen deutlichen Mehrwert für die Fehlertoleranz. Die Überlasterkennung von Istio sollte nur genutzt werden, wenn hinreichende Erfahrungswerte für das Lastprofils z. B. den maximalen HTTP-Requests vorhanden sind.
Die Circuit-Breaker von Traefik zeichnet sich im Praxistest als Service-Mesh-Circuit-Breaker mit guter Reaktion sowohl für verschiedene Fehler- als auch Überlastsituationen aus. Zudem ermöglicht er eine einfache Konfiguration über Kubernetes-Annotations.
Anders als die beiden Service-Mesh-Implementierungen hat die Libary-Lösung, repräsentiert durch Resilience4j, den Nachteil, dass sie Zugriff auf den Servicecode benötigt bzw. Anforderungen an die Programmiersprache stellt. Dennoch bietet Resilience4j die umfangreichste Parametrisierung und zudem durch die serviceseitigen Umsetzung auch die Option von fachlichen Fallbacks. Somit bleibt für ein individuell-angepasstes Circuit-Breaker-Verhalten die direkte Umsetzung innerhalb des Services die erste Wahl.

Basierend auf diesen Erkenntnissen bieten sich für das Forschungsfeld „Circuit-Breaking in Service-Meshes“ weitere Untersuchungsaspekte, wie das Ausweiten des Praxistest auf Netzwerkfehler, weitere Konfigurationen, ein größeres Fallbeispiel mit mehreren Nodes und Replicas und weitere Circuit-Breaker-Implementierungen. Interessant wäre beispielsweise ein Vergleich der hier beschriebenen Implementierungen mit der angekündigten Umsetzung von Linkerd2 [PW+22] oder des Algorithmus von Sedghpour et al. [SKT21] zum dynamischen Anpassen von Istio's `HTTPMaxRequest`-Parameters basierend auf der aktuellen Antwortzeit.

Literatur

[Al20] Allen, T.: Envoy, Take the Wheel: Real-time Adaptive Circuit Breaking (KubeCon), 2020, URL: https://youtu.be/CQvmSXlnyeQ, Stand: 05. 07. 2022.

[CNCF18] Cloud Native Computing Foundation: CNCF Cloud Native Definition v1.0, 2018, URL: https://github.com/cncf/toc/blob/main/DEFINITION.md, Stand: 05. 07. 2022.

[En22] Envoy Docs, https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/upstream/circuit_breaking und https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/upstream/outlier, 2022, Stand: 05. 07. 2022.

[Fo14] Fowler, M.: CircuitBreaker, 2014, URL: https://martinfowler.com/bliki/CircuitBreaker.html, Stand: 05. 07. 2022.

[Fr16] Friedrichsen, U.: Resilient Software Design. Informatik Aktuell/, 2016, URL: https://www.informatik-aktuell.de/entwicklung/methoden/resilient-software-design-robuste-software-entwickeln.html.

[FSS20] Falahah; Surendro, K.; Sunindyo, W. D.: Circuit Breaker in Microservices: State of the Art and Future Prospects. In: ICITDA'20. 2020.

[Is22] Istio Docs, https://istio.io/latest/docs/, https://istio.io/latest/docs/reference/config/networking/destination-rule/ und https://istio.io/latest/docs/tasks/traffic-management/circuit-breaking/, 2022, Stand: 05. 07. 2022.

[KA19] Kratzke, N.; Adersberger, J.: Service Meshes in Microservice Architekturen (Fachposter). In. OBJEKTspektrum, 2019.

[Li19] Li, W.; Lemieux, Y.; Gao, J.; Zhao, Z.; Han, Y.: Service Mesh: Challenges, State of the Art, and Future Research Opportunities. In: SOSE'19. IEEE, 2019.

[MW16] Montesi, F.; Weber, J.: Circuit Breakers, Discovery, and API Gateways in Microservices. In. arXiv, 2016.

[Ny18] Nygard, M. T.: Release It! : Design and Deploy Production-Ready Software. In. Pragmatic Bookshelf (2. Ed.), S. 91–98, 2018.

[PW+22] Prinz, H.; Wolff, E. et al.: Service Mesh Comparison, 2022, URL: https://servicemesh.es, Stand: 05. 07. 2022.

[R4J22] Resilience4j Docs, https://resilience4j.readme.io/docs und https://resilience4j.readme.io/docs/circuitbreaker, 2022, Stand: 05. 07. 2022.

[SKT21] Sedghpour, M. R. S.; Klein, C.; Tordsson, J.: Service mesh circuit breaker. In: HAOC'21. ACM, 2021.

[SKT22] Sedghpour, M. R. S.; Klein, C.; Tordsson, J.: An Empirical Study of Service Mesh Traffic Management Policies for Microservices. In: ICPE'22. ACM, 2022.

[SL12] Spillner, A.; Linz, T.: Basiswissen Softwaretest. In. dpunkt.verlag (5. Aufl.), S. 114–125, 2012.

[Tr22] Traefik Docs, https://doc.traefik.io/traefik-mesh/, https://doc.traefik.io/traefik/middlewares/http/circuitbreaker/ und https://doc.traefik.io/traefik-mesh/configuration/#circuit-breaker, 2022, Stand: 05. 07. 2022.

Evaluation der proxybasierten Ansätze von Istio und Cilium zur Zugriffskontrolle mit L7-Netzwerkrichtlinien in Cloudnativen Anwendungssystemen

Andre Farwick[1]

Abstract: Der Einsatz von L7-Netzwerkrichtlinien ist ein wichtiger Bestandteil um Zero Trust Paradigmen in Cloudnativen Umgebungen anzuwenden, und um Workloads, deren Schnittstellen und Daten vor unerlaubten Zugriff zu schützen. In dieser Arbeit wird der strukturelle Aufbau und die Performance der proxybasierten Ansätze zur Umsetzung von L7-Netzwerkrichtlinien von Istio und Cilium verglichen. Bei Istio werden die Netzwerkrichtlinien mit einem Proxy pro Workload (Sidecar Proxy) umgesetzt, wohingegen bei Cilium ein Proxy für alle Workloads eines Rechenknotens (Per-Node Proxy) und die Linux-Kernel Technologie eBPF eingesetzt werden. Die Performance Messungen zeigen, dass Cilium mit dem Per-Node Proxy und eBPF einen höheren maximalen Durchsatz, eine geringere Antwortzeit und eine geringere Speicherplatz Belegung erreicht.

Keywords: Zugriffskontrolle; L7-Netzwerkrichtlinien; Zero Trust; Kubernetes; Cloud; Istio; Cilium; Envoy; Sidecar Pattern; Per-Node Proxy; eBPF

1 Einleitung

Moderne Cloudtechnologien, wie die Containerisierung von Applikationen und deren Orchestrierung, ermöglichen es öffentlichen Cloud Dienstleistern, Workloads von mehreren Kunden auf den selben Servern gleichzeitig zu betreiben. Anders als virtuelle Maschinen teilen sich Container den gleichen Kernel, was die Möglichkeit neuer Angriffsvektoren eröffnet [BLM19, S. 1]. Attacken können nicht nur von außerhalb des Rechenzentrums auftreten, sondern auch innerhalb der Grenzen. Daraus folgt die Frage, wie Workloads sich und ihre Daten vor unerlaubten Zugriff und Exfiltration unabhängig vom Perimeter schützen können.

Ansätze zur Lösung dieses Problems liefern die Konzepte rund um den Begriff *Zero Trust*. Zero Trust beruht auf der Idee, dass jede Anfrage und jeder Zugriff auf ein Workload unabhängig von der Herkunft verifiziert und kontrolliert werden muss. Das steht im Gegensatz zu herkömmlichen Sicherheitsparadigmen, bei denen Workloads und Nutzern, die sich physikalisch auf dem gleichen System oder in dem gleichen Netzwerk befinden, implizit Vertrauen ausgesprochen wird. Des Weiteren basiert Zero Trust auf dem Prinzip der geringsten Privilegien, das heißt es werden maximal nur die Berechtigungen vergeben, die zur Ausführung einer Aufgabe notwendig sind [Ro20, S.1, S.4].

[1] FH Münster, Master Wirtschaftsinformatik andre.farwick@fh-muenster.de

Das Konzept der Zugriffskontrolle des Zero Trust Paradigmas kann mit Hilfe von Netzwerkrichtlinien umgesetzt werden. Eine Möglichkeit diese in Cloudnativen Anwendungssystemen zu definieren und durchzusetzen ist der Einsatz eines *Service Mesh*. Diese nutzen historisch einen Sidecar Proxy, der die Kommunikation zwischen den Workloads abfängt, kontrolliert und steuert [Li19, S. 122f.].

Die Verwendung der Sidecar Proxys erzeugt allerdings zusätzlichen Overhead, der mit der Anzahl der Workload Instanzen steigt. Um dieses Problem zu lösen hat das Open Source Projekt Cilium [2] eine Alternative entwickelt, welche anstelle eines Sidecar Proxys, nur einen Proxy pro Cluster-Knoten (Per-Node Proxy) einsetzt. Zum Abfangen und Weiterleiten des Netzwerkverkehrs an den Proxy wird die Linux Kernel Technologie eBPF verwendet [Gr22].

Das Ziel dieser Arbeit ist die Evaluation der zwei zuvor vorgestellten proxybasierten Ansätze zur Umsetzung von L7-Netzwerkrichtlinien anhand eines strukturellen Vergleichs und eines Vergleichs der Performance. Als Vertreter für den Sidecar Proxy wird die verbreitete Service Mesh Technologie Istio [3] untersucht, und als Vertreter für den Per-Node Proxy Cilium. Bei dem strukturellen Vergleich werden grundsätzliche Unterschiede in der Architektur aufgezeigt und der Funktionsumfang der L7-Netzwerkrichtlinien der zwei Anbieter verglichen. Für den Performance Vergleich werden Messungen für die Antwortzeit, den maximalen Durchsatz, der CPU, sowie der Memory-Belastung durchgeführt. Die Arbeit soll bei der Auswahl eines Sicherheitstools für die Umsetzung von fein-granularen L7-Netzwerkrichtlinien in Cloudnativen Applikationssystemen unterstützen.

2 Forschungsstand

Der Einsatz von Netzwerkrichtlinien zur Umsetzung von Zero Trust Prinzipien ist aktuell Gegenstand der Forschung. In der Literatur finden sich Arbeiten, welche die unterschiedlichen Ansätze zur Umsetzung von Netzwerkrichtlinien in containierisierten und orchestrierten Cloud Umgebungen analysieren und klassifizieren [BLM19][Za19]. Außerdem wurden Container Network Interface (CNI) Plugins, welche Netzwerkrichtlinien auf den Ebenen 3 und 4 des OSI-Referenzmodells umsetzen, hinsichtlich der Funktionsweise und der Performance verglichen [MG19]. Des Weiteren wurde der Einfluss von eBPF auf die Performance untersucht [Sc18] und es wurden neue Ansätze zur Zugriffskontrolle auf Basis von eBPF entwickelt [Za19]. Auch zum Thema Service Meshes und deren Einfluss auf Performance gibt es Untersuchungen [Ga21]. Zum Zeitpunkt des Verfassen der Arbeit gibt es allerdings keine bekannte Forschung, welche die unterschiedlichen proxy-basierten Lösungen von L7-Netzwerkrichtlinien vergleicht und evaluiert.

[2] https://cilium.io/
[3] https://istio.io/

3 Grundlagen

3.1 Zugriffskontrolle mit Netzwerkrichtlinien

Netzwerkrichtlinien sind Sicherheitsrichtlinien mit denen festgelegt wird, welche Workloads miteinander kommunizieren dürfen und ermöglichen somit eine netzwerkbasierte Zugriffskontrolle. Sie können sowohl für eingehende Anfragen (Ingress) als auch für ausgehende Anfragen (Egress) angewendet werden. Die Richtlinien können auf den unterschiedlichen Ebenen des OSI-Referenzmodells (ISO/IEC 7498-1:1994) definiert und durchgesetzt werden. Aufgrund der Schichtenarchitektur des Netzwerkmodells werden die Richtlinien bei steigender Ebene granularer und beinhalten jeweils die Richtlinien der unteren Ebenen.
Je nach angewandtem Prüfmodell werden die Richtlinien entweder in Form von Positiv-, oder Negativlisten verfasst. Bei dem Ansatz von Positivlisten, werden alle Anfragen abgelehnt, die nicht explizit zugelassen worden sind. Negativlisten funktionieren genau entgegengesetzt, sodass die gesamte Kommunikation zwischen Workloads standardmäßig zugelassen wird, außer sie wird explizit untersagt.
Für ein Buchungssystem mit einem Rechnungsservice und einem Bestellservice könnte eine Netzwerkrichtlinie auf der Applikationsschicht des Netzwerkmodells (L7) beispielhaft wiefolgt aussehen: Es dürfen nur HTTP-Anfragen mit der Post-Methode vom Bestellservice and den Rechnungsservice zugelassen werden.
Auf der Vermittlungs- und Transportschicht (L3 und L4) des Netzwerkmodells, können die Richtlinien entweder auf Basis der Netzwerk-Endpunkte oder der Identitäten der Workloads umgesetzt werden [Za19, S.49f.].
Für den ersten Fall werden die Regeln mit Filtern auf Eigenschaften der Netzwerk-Endpunkte wie der IP-Adresse und den Port durchgesetzt. Das ist insbesondere in dynamischen Cloud Umgebungen fehleranfällig und schlecht skalierbar, da diese Eigenschaften nicht die primär bindenden Attribute der Workloads sind und sich ändern können. Bei jeder Änderung müssen die Regeln angepasst werden. Außerdem erhöht sich bei einer steigenden Anzahl von Replikationen die Anzahl der Filter [Za19, S.50].
Identitätsbasierte Richtlinien lösen diese Probleme, indem jedem Workload eine Identität zugewiesen wird und nicht die Netzwerk-Endpunkte, sondern ausschließlich die Identitäten geprüft werden. Mehrere Workloads können sich die gleiche Identität teilen, was die Skalierbarkeit erhöht [Za19, S.50f.].
Netzwerkrichtlinien der Applikationsschicht können mit Proxys umgesetzt werden, welche den Netzwerkverkehr abfangen und die Logik zur Überprüfung der entsprechenden L7-Protokolle enthalten.

Die Abbildungen 1 und 2 zeigen jeweils den Einsatz des Proxys mit dem Sidecar- und dem Per-Node Proxy Pattern.
Beim Sidcear-Pattern erhält jeder Workload einen eigenen Proxy, wohingegen sich im Falle des Per-Node Proxys alle Workloads auf einem Knoten sich die gleiche Proxy Instanz teilen. Der Vorteil des Per-Node Proxys liegt in dem geringeren Overhead, insbesondere bei einer großen Anzahl an Workloads auf einem Knoten, da jeder Proxy CPU-Zeit und

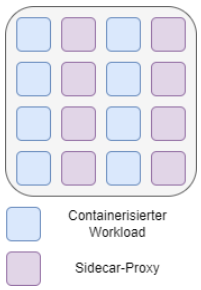

Abb. 1: Sidecar Proxy

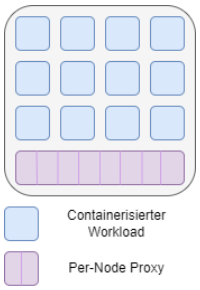

Abb. 2: Per-Node Proxy

Arbeitsspeicher benötigt. Zusätzlich muss eine Anfrage nicht den Netzwerkstack von zwei Proxy-Container durchlaufen, sondern nur von einem [Gr22, S.49-51].
Damit ein Proxy mehrere Workloads bedienen kann, muss dieser allerdings um Logik erweitert werden, um zum Beispiel Ressourcenmanagement zu ermöglichen, damit sichergestellt wird, dass nicht alle Ressourcen von nur einem Workload konsumiert werden [Gr20].

Abschließend werden wichtige Anforderungen an Netzwerkrichtlinien vorgestellt, die im weiteren Verlauf der Arbeit als Bewertungskriterien für den Vergleich von Istio und Cilium herangezogen werden. Die Anforderungen sind angelehnt an die Kriterien aus [Za19, S.51].

Konfigurierbarkeit Je umfassender und detaillierter Netzwerkrichtlinien konfiguriert werden können, umso granularer können die Schnittstellen vor unerlaubten Zugriff geschützt werden. Dazu gehört zum Beispiel die Anzahl und Vielfalt der angebotenen Filterattribute. Außerdem hat die Konfigurierbarkeit Einfluss auf die Breite der Einsatzszenarien, für welche die Netzwerkrichtlinien eingesetzt werden können. Wichtige Aspekte dabei sind die unterstützten Protokolle, die unterstützte Richtung der Netzwerkrichtlinie (Ingress und Egress) und die unterstützten Prüfmodelle (Positiv- und Negativlisten).

Zuverlässigkeit Netzwerkrichtlinien sollen Workloads vor unauthorisierten Zugriff und Exfiltration schützen, sodass es wichtig ist, dass die Netzwerkrichtlinien zuverlässig durchgesetzt werden und nicht umgangen werden können.

Overhead Der Bedarf an zusätzlichen Ressourcen und der Einfluss auf den Durchsatz und die Antwortzeit durch den Einsatz von Netzwerkrichtlinien soll möglichst gering sein. Im Cloudumfeld schlägt sich der zusätzliche Ressourcenbedarf in variablen Kosten nieder. Außerdem erwarten Nutzer von Applikationen möglichst niedrige Latenzen.

3.2 eBPF

eBPF ist eine Linux Kernel Technologie mit der Programme in einer isolierten und geschützten Umgebung innerhalb des Linux Kernelraums ausgeführt werden können. Der

Begriff ist ein Akronym, welches für *extended Berkeley Package Filter* steht und in der Praxis häufig synonym zu dem Vorläufer *BPF* verwendet wird. *eBPF* definiert ein eigenes Bytecode-Format in Form eines RISC-Befehlssatzes. Die maximale Größe des Programms ist auf 4096 Befehle beschränkt (bei Kernel Versionen >=5.1 1.000.000 Befehle). Zum Schreiben der Programme gibt es für die Hochsprachen *C* und *Rust* das LLVM-Compiler Backend *clang*, mit dem der Code in das eBPF-Bytecode-Format kompiliert werden kann. Neben dem eigentlichen eBPF-Programm wird ein Programm benötigt, welches das eBPF-Programm aus dem Benutzerraum in den Kernelraum lädt und ein Event hinzufügt, welches festlegt, wann das eBPF-Programm ausgeführt werden soll. Für diese Events gibt es vordefinierte Einschubmethoden (Hooks), wie zum Beispiel Systemaufrufe, aber es können auch individuelle Einschubmethoden mit Hilfe von *kernel probes* (kprobe) für den Kernelraum, oder *user probes* (uprobe) für den Benutzerraum erstellt werden. Die Ausführung der eBPF-Programme im Kernelraum erfolgt mit Hilfe eines Just-In-Time Compiler, der den Bytecode in Maschinencode übersetzt. Aus Sicherheitsgründen muss der Code vorher allerdings von dem *Verifier* überprüft werden. Der Verifier stellt zum Beispiel sicher, dass die Programme immer beenden, indem unter anderem der Gebrauch von Schleifen verboten wird [22a].

4 Struktureller Vergleich

In dem nachfolgenden Kapitel wird die Funktionsweise und der Funktionsumfang von Istio (v.1.13.2) und Cilium (v.1.11.6) zur Umsetzung der L7-Netzwerkrichtlinien vorgestellt.

Istio ist ein weit verbreitetes Open Source Service Mesh mit einem hohen Reifegrad, welches in der Data-Plane auf dem Sidecar-Pattern aufbaut. Als Proxy wird der Envoy Proxy [4] verwendet, der bei einem Kubernetes Cluster im selben Pod des Applikationscontainers ausgeführt wird. In Istio werden Netzwerkrichtlinien in Form von *AuthorizationPolicies* definiert, welche in Kubernetes mit *Custom Ressources* umgesetzt werden. Die Zugriffskontrolle wird durch den Envoy Proxy des empfangenden Workloads durchgesetzt, sodass ausschließlich Ingress Netzwerkrichtlinien unterstützt werden. Die Richtlinien können sowohl auf Basis der Netzwerk-Endpunkte, über die Angabe von IP-Adressen, oder der Identitäten der Workloads umgesetzt werden. Die Identitäten werden entweder von Kubernetes *Service Accounts* oder von *JSON Web Tokens* (JWT) hergeleitet. Für die Nutzung von Service Accounts als Identität muss in Istio die *Peer Authentication* aktiviert sein, damit die Identität über das mtls-Zertifikat an den Proxy übermittelt werden kann. Als Vorraussetzung zur Nutzung des JWT für die Identität muss in Istio die *Request Authentication* aktiviert sein [22b].
Dadurch, dass der Proxy und der Workload, sich innerhalb des Pod in den gleichen Netzwerk- und Prozess-Isolationsgrenzen befinden, kann die Applikation den Proxy und Weiterleitungsregeln manipulieren. Das ermöglicht es der Applikation den Sidecar Proxy sowohl für den eingehenden, als auch für den ausgehenden Netzwerkverkehr zu umgehen. Als

[4] https://www.envoyproxy.io/

Konsequenz kann nicht garantiert werden, dass die Netzwerkrichtlinien immer geprüft werden [22b].

Cilium ist ein Open Source Netzwerk und Sicherheitstool für Cloudnative Umgebungen wie Kubernetes Cluster, für welches es ein Container Network Interface (CNI) Plugin bereitstellt. Das Projekt verfolgt das Ziel zu einem Service Mesh heranzureifen, indem weitere Funktionen mit Hilfe eines Per-Node Proxys implementiert werden [Gr22]. Die Basistechnologie zur Umsetzung des Netzwerk und der Sicherheitsfunktionen von Cilium ist eBPF. Die eBPF-Programme werden von einem Agenten, der auf jedem Clusterknoten ausgeführt wird, verwaltet und in den Kernel Raum des Knoten injiziert. Innerhalb des Agenten läuft außerdem der von Cilium erweiterte Envoy Proxy, mit dem die L7-Filter der Netzwerkrichtlinie geprüft werden [22a].

Die Netzwerkrichtlinien werden auf Basis von Identitäten von Cilium Endpunkten umgesetzt. Ein Cilium Endpunkt ist eine Abstraktion für Workloads mit gemeinsamer IP-Adresse. Dementsprechend ist in Kubernetes jeder Pod ein eigener Cilium Endpunkt. Die Identität wird mit Hilfe von *Labels* (Schlüssel-Wert Paare) des Workloads identifiziert [22a]. Der Vorteil liegt in der hohen Skalierbarkeit, da beim Hinzufügen eines neuen Workloads lediglich in einem globalen Schlüssel-Wert Speicher die Identität zu dem zugehörigen Label abgefragt werden muss. Falls noch keine Identität vorhanden ist, wird eine neue hinzugefügt [22a]. Problematisch an dem Ansatz ist allerdings, dass dem Label blind vertraut wird und keine weitere Überprüfung der Identität erfolgt, was Spoofing Attacken ermöglicht. Angreifer können schädlichen Workloads die Identität von vertrauenswürdigen Workloads geben, indem sie die gleichen Labels verwenden [Za19, S.51].

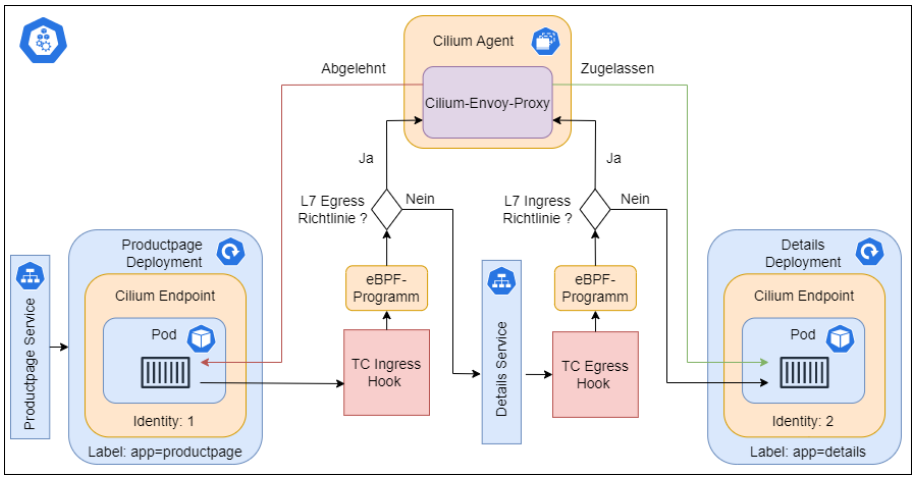

Abb. 3: Funktionsweise L7-Netzwerkrichtlinien mit Cilium am Beispiel eines Auszugs der Microservice Applikation Bookinfo von Istio

Abbildung 3 zeigt anhand eines Ausschnitts der Beispielapplikation *Bookinfo*[5], wie mit Cilium L7-Netzwerkrichtlinien durchgesetzt werden. Dargestellt werden die zwei Microservices *productpage* und *details*, die in einem Kubernetes Cluster ausgeführt werden.
Für Egress Richtlinien wird beim Verlassen einer Anfrage von einem Container der *Traffic Controll (tc) Ingress Hook* des Gastrechners aufgerufen, was die Ausführung eines eBPF-Programms triggert. Das Programm prüft, ob Netzwerkrichtlinien für die Anfrage vorliegen und setzt im positiven Fall L3 und L4 Filter (Identität und Port) durch. Wenn eine Anfrage abgelehnt wird, wird diese fallen gelassen und läuft in einen Zeitüberschreitungsfehler. Falls zusätzlich L7-Attribute in der Richtlinie definiert sind, wird die Anfrage an den Envoy Proxy weitergeleitet. Der Envoy Proxy wendet die L7 Filter an und lehnt entweder mit einer protokollspezifischen Antwort ab, oder leitet die Anfrage weiter [22a].
Für Ingress Richtlinien erfolgt der Prozess ähnlich, mit dem Unterschied, dass der tc Hook beim Eingang einer Anfrage an den Container aufgerufen wird und in dem eBPF-Programm nach Ingress-Richtlinien geprüft wird [22a].

In der Tabelle 1 werden die wichtigsten Funktionen der L7-Netzwerkrichtlinien zwischen Istio und Cilium gegenüber gestellt und zusammengefasst.

5 Performance Vergleich

Um den von Istio und Cilium erzeugten Overhead bewerten zu können, werden Messungen zum maximalen Durchsatz, der Antwortzeit, der CPU- und der Arbeitsspeicherbelastung durchgeführt.

5.1 Aufbau des Experiments

Für die Performance Messungen werden zwei *Deployments* des Loadtestingstools *Fortio*[6] (Version 1.34.1) in einem Kubernetes Cluster erstellt, wovon einer die Rolle eines Clients und der andere die Rolle eines Servers übernimmt. Der Server beantwortet alle Anfragen mit einer Kopie der Anfrage. Der Client und die Server-Instanzen werden auf unterschiedlichen Knoten in dem Kubernetes Cluster ausgeführt, sodass diese nicht um Ressourcen konkurrieren. Der Start der Messungen wird außerhalb des Clusters mit *kubectl* angestoßen, sodass die Ausführung des Testscript auch keinen Einfluss auf die Ressourcenbelastung hat.
Die Messungen zum maximalen Durchsatz und der Antwortzeit werden von Fortio durchgeführt. Für die Messung des maximalen Durchsatzes wird die höchstmögliche Last vom Fortio Client mit 32 Threads erzeugt. Die Antwortzeit wird bei geringer Last gemessen. Für jede CPU wird eine Anfrage pro Sekunde versendet. Die Arbeitsspeicher und CPU-Benutzung werden bei maximaler Last mit Hilfe des Linux-Tools *top* gemessen.
Die in den Messungen angewandten L7-Richtlinien werden für den Fortio Server definiert

[5] https://github.com/istio/istio/tree/master/samples/bookinfo
[6] https://github.com/fortio/fortio

	Cilium	Istio
L7-Protokolle	• HTTP (für REST und gRPC APIs) • Kafka • Erweiterbar um zusätzliche Protokolle, durch Hinzufügen eines Parsers für Envoy Proxy	• HTTP (für REST und gRPC APIs)
Richtung des Netzwerkverkehrs	• Ingress • Egress	• Ingress
Prüfmodell	• Positivliste • Negativliste nur für L3 u. L4	• Positivliste • Negativliste
Peripherie	• Innerhalb des Clusters (East-West Traffic) • Außerhalb des Clusters (North-South Traffic)	• Innerhalb des Clusters (East-West Traffic) • Außerhalb des Clusters nur mit Gateways (North-South Traffic)
Identifizieren der Workloads	• Innerhalb von Cilium gemanagten Ressourcen: Identitäten hergeleitet von Labels • Außerhalb: Domain Name u. IP-Adressraum	• Innerhalb von Istio gemanagten Ressourcen: Identitäten hergeleitet von Service Accounts u. JWTs, sowie Namespaces • Außerhalb: Host Name nur mit Gateways u. IP-Adressraum
L7-Filterattribute (Http)	• Methode • Pfad (inkl. Pattern-Matching) • Request Header	• Methode • Pfad (inkl. Pattern-Matching) • Request Header • JWT Token Claims
Weitere Funktionen	• Multi Cluster fähig	• Multi Cluster fähig • Integration von externen Autorisierung Systemen mit CUSTOM Policies

Tab. 1: Vergleich des Funktionsumfanges zwischen Cilium und Istio [22a][22b]

(Ingress-Richtlinien) und spezifizieren als L7-Filterattribute die erlaubte HTTP-Methode und den erlaubten Pfad. Bei Cilium wird der zugelassene Sender einer Anfrage über das Label identifiziert und bei Istio über einen Service Account.
Alle Tests werden mit 1 aktiven Richtlinie, und wenn nicht anders angegeben mit 1 Server-Instanz durchgeführt. Für die Basismessung wird Cilium als CNI-Plugin ohne den Einsatz von Netzwerkrichtlinien verwendet. Auch bei den Messungen zu Istio wird Cilium als CNI-Plugin verwendet, um auszuschließen, dass Unterschiede bei der Implementierung des Container Netzwerkes die Messergebnisse beeinflussen. Cilium wird mit der Version 1.11.6 und Istio mit der Version 1.13.2 und dem Profil *default* installiert.
Für das Kubernetes Cluster wird die Google Kubernetes Engine (GKE) mit der Version 1.22.8-gke.202 verwendet. Es besteht aus 2 Knoten des Maschinentypen n2-standard-4, der

mit 4 virtuellen CPUs und einen gesamtem Arbeitsspeicher von 16000 MiB ausgestattet ist. Die virtuellen Maschinen haben die Linux Kernel Version 5.10.90. Der gesamte Testaufbau kann in einem öffentlichem Git-Repository[7] repliziert und nachvollzogen werden.

5.2 Antwortzeit und Durchsatz

Auf der linken Seite der Abbildung 4 ist die durchschnittliche Antwortzeit von 100 Anfragen an den Fortio Server zu sehen. Es ist zu erkennen, dass Istio mit Abstand die längste Antwortzeit und ca. 7-mal länger als Cilium benötigt.
Auf der rechten Seite der Abbildung 4 sind die Ergebnisse des maximalen Durchsatzes dargestellt. Dieser wird mit den durchschnittlichen Anfragen pro Sekunde über einen Zeitraum von 30 Sekunden gemessen. Der Unterschied zwischen Istio und Cilium fällt beim maximalen Durchsatz noch deutlicher aus als bei der Antwortzeit. Istio erreicht einen Wert von ca. 5.300 Anfragen pro Sekunde, wohingegen Cilium auf ca 68.500 Anfragen pro Sekunde kommt.

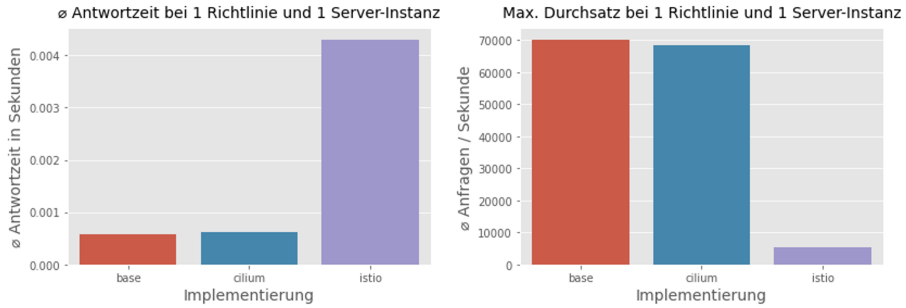

Abb. 4: Messergebnisse zur durchscnittlichen Antwortzeit und maximalen Durchsatz bei 1 aktiven Netzwerkrichtlinie und 1 Server-Instanz

Für die Unterschiede bei der Antwortzeit und dem maximalen Durchsatz zwischen Cilium und Istio kann es mehrere Gründe geben. Zum einen muss bei Cilium eine Anfrage nur durch den Netzwerkstack des Per-Node Proxys gehen, anstelle durch beide Sidecar Proxys. Außerdem wird bei Istio die Anfrage via mtls verschlüsselt, um die Identität mit zu versenden, was zusätzliche Rechenkapazitäten benötigt und die Größe der Anfrage erhöht.
Die Unterschiede zwischen Cilium und den Basismessungen ist gering, woraus sich schließen lässt, dass die Prüfung der Richtlinie sich kaum negativ auf die Latenz und den Durchsatz auswirkt.

[7] https://github.com/codafuerte/netzwerkrichtlinien-messungen

5.3 CPU- und Arbeitsspeicher Belastung

Die linke Seite der Abbildung 5 zeigt den durchschnittlich belegten Arbeitsspeicher bei einer steigenden Anzahl von Server-Replikationen. Bei Istio steigt der belegte Arbeitsspeicher bei wachsender Anzahl an Replikationen schneller, als bei Cilium, sodass Istio bei 30 Replikationen ca. doppelt soviel Arbeitsspeicher belegt. Verantwortlich dafür sind die zusätzlich benötigten Sidecar-Proxys. Auffällig ist, dass der Arbeitsspeicher Bedarf in der Basismessung höher ist als beim Einsatz von L7-Richtlinien mit Cilium. Um Gründe dafür herauszufinden ist weitere Forschung notwendig.

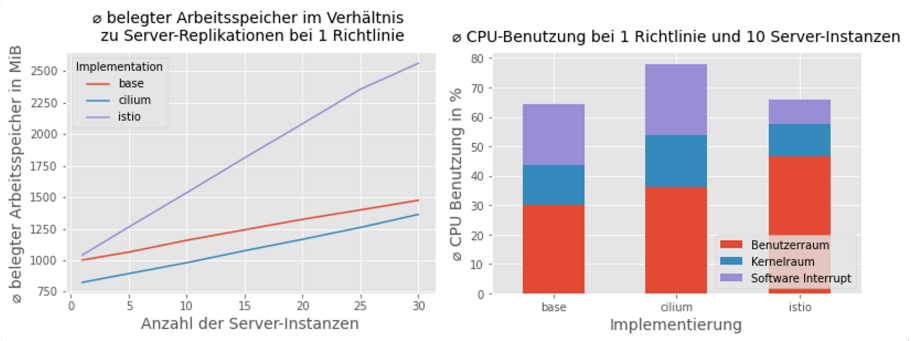

Abb. 5: Messergebnisse zur Ressourcenbenutzung bei 1 aktiven Richtlinie

Auf der Rechten Seite der Abbildung 5 wird die durchschnittliche prozentuale CPU-Auslastung des Server-Knoten bei 10 Server-Instanzen dargestellt. Cilium hat mit ca. 80% die höchste Auslastung. Das liegt vor allem an der hohen Belastung im Kernelraum und an den Software Interrupts, die unter anderem durch Systemaufrufe hervorgerufen werden. Gründe dafür sind die Benutzung der eBPF-Programme, die bei festgelegten Systemaufrunfen im Kernelraum ausgeführt werden und der CPU-Bedarf des Per-Node Proxys.

6 Ergebnisdiskussion

In der Tabelle 2 wird die Evaluation von Istio und Cilium anhand der zuvor definierten Anforderungen an L7-Netzwerkrichtlinien mit Hilfe einer Ordinalskala von 1 bis 3 zusammengefasst. Die Werte stehen aufsteigend für, Anforderung schlecht, teilweise und gut erfüllt.

Konfigurierbarkeit Sowohl Istio als auch Cilium erfüllen die Anforderungen an die Konfigurierbarkeit nur teilweise. Die größten Einschränkungen bei Istio sind, dass Netzwerkrichtlinien nur für eingehenden Netzwerkverkehr (Ingress) eingesetzt werden können und nur HTTP als L7-Protokoll unterstützt wird, was viele Einsatzszenarien ausschließt. Cilium

hingegen unterstützt Egress Richtlinien und ist deutlich weniger protokollabhängig, da es um weitere Protokolle erweitert werden kann.
Auf der anderen Seite unterstützt Istio sowohl Positiv- als auch Negativlisten, welche bei Cilium auf der Netzwerkebene 7 nicht möglich sind. Eine weitere Funktion, welche ausschließlich von Istio angeboten wird, ist die Integration von externen Authorisierungssystemen.

Zuverlässigkeit Bei Istio kann nicht garantiert werden, dass Netzwerkrichtlinien angewendet werden, weil nicht sichergestellt werden kann, dass die gesamte Kommunikation von den Sidecar Proxys abgefangen wird [22b]. Cilium hat dieses Problem nicht, da die Überprüfung der Netzwerkrichtlinien mit eBPF-Progammen im Kernelraum erfolgt, die nicht umgangen werden können. Auf der anderen Seite sind bei Cilium die Identitäten nicht zuverlässig, da diese auf Label basieren, welche auch von Angreifern genutzt werden können [Za19, S.51]. Bei Istio basieren die Identitäten entweder auf Kubernetes Service Accounts oder JWTs. Anders als Label, sind Service Accounts eigene Kubernetes API Objekte. Die Erstellung und Manipulation von den API-Objekten, kann mit Kubernetes *Role Based Access Control* kontrolliert werden [22c].

Overhead Die Messungen zur Performance haben gezeigt, dass Istio einen deutlich geringeren Durchsatz, eine höhere Antwortzeit und einen größeren Bedarf an Arbeitsspeicher im Vergleich zu Cilium hat. Hinsichtlich des benötigten Arbeitsspeicher, skaliert Ciliums Per-Node Proxy deutlich besser, als der Sidecar Proxy. Auch die Label-basierten Identitäten von Cilium wirken sich positiv auf die Skalierbarkeit aus. Ausschließlich bei der CPU-Belastung schneidet Istio besser ab.

	Konfigurierbarkeit	Zuverlässigkeit	Overhead
Istio	2	1	1
Cilium	2	1	3

Tab. 2: Zusammenfassung Evaluation Istio und Cilium

7 Fazit und Ausblick

Sowohl mit dem Sidecar-Pattern, als auch mit dem Per-Node Proxy lassen sich L7-Netzwerkrichtlinien umsetzen. Bei der Konfigurierbarkeit und der Zuverlässigkeit der Richtlinien haben Istio und Cilium unterschiedliche Stärken und Schwächen. Hinsichtlich des zusätzlichen Overheads schneidet Cilium deutlich besser ab als Istio. Es wird weniger Arbeitsspeicher und eine kürzere Antwortzeit benötigt und ein höherer maximaler Durchsatz erzielt. Außerdem skaliert Cilium aufgrund des Per-Node Proxies besser als Istio. Je nach Einsatzszenario und Bedarf, kann Cilium eine Alternative für klassische Service Meshes wie Istio sein, wenn die Hauptmotivation zum Einsatz des Service Mesh die Nutzung der Sicherheits- und Sichtbarkeitsfunktionen ist.
Für zukünftige Forschung können umfangreichere Messungen und Auswertungen wie

zum Beispiel eine Multifaktor Regressionsanalyse interessant sein, um die Performance Unterschiede genauer erklären zu können.
Außerdem wäre es interessant, den Vergleich des Sidecar-Patterns und des Per-Node Proxy Patterns auf weitere klassische Service Mesh Funktionen auszuweiten.

Literatur

[22a] Cilium Documentation, 2022, URL: https://docs.cilium.io.

[22b] Istio Documentation, 2022, URL: https://istio.io/latest/docs.

[22c] Kubernetes Documentation, 2022, URL: https://kubernetes.io/docs.

[BLM19] Bélair, M.; Laniepce, S.; Menaud, J.-M.: Leveraging Kernel Security Mechanisms to Improve Container Security. In: Proceedings of the 14th International Conference on Availability, Reliability and Security. ACM, New York, NY, USA, S. 1–6, 2019, ISBN: 9781450371643.

[Ga21] Ganguli, M.; Ranganath, S.; Ravisundar, S.; Layek, A.; Ilangovan, D.; Verplanke, E.: Challenges and Opportunities in Performance Benchmarking of Service Mesh for the Edge. In: 2021 IEEE International Conference on Edge Computing (EDGE). IEEE, S. 78–85, 2021, ISBN: 978-1-6654-0062-6.

[Gr20] Graf, T.: Envoy Namespaces - Operating an Envoy-based Servicemesh at a Fraction of the Cost, 2020, URL: https://www.youtube.com/watch?v=08opgZkdYIw.

[Gr22] Graf, T.: How eBPF will solve Service Mesh - Goodbye Sidecars, 4.03.2022, URL: https://isovalent.com/blog/post/2021-12-08-ebpf-servicemesh.

[Li19] Li, W.; Lemieux, Y.; Gao, J.; Zhao, Z.; Han, Y.: Service Mesh: Challenges, State of the Art, and Future Research Opportunities. In: 2019 IEEE International Conference on Service-Oriented System Engineering (SOSE). IEEE, S. 122–1225, 2019, ISBN: 978-1-7281-1442-2.

[MG19] Makowski, Ł.; Grosso, P.: Evaluation of virtualization and traffic filtering methods for container networks. Future Generation Computer Systems 93/, S. 345–357, 2019, ISSN: 0167739X.

[Ro20] Rose, S.; Borchert, O.; Mitchell, S.; Connelly, S.: Zero Trust Architecture, 2020.

[Sc18] Scholz, D.; Raumer, D.; Emmerich, P.; Kurtz, A.; Lesiak, K.; Carle, G.: Performance Implications of Packet Filtering with Linux eBPF. In: 2018 30th International Teletraffic Congress (ITC 30). IEEE, S. 209–217, 2018, ISBN: 978-0-9883045-5-0.

[Za19] Zaheer, Z.; Chang, H.; Mukherjee, S.; van der Merwe, J.: eZTrust. In: Proceedings of the 2019 ACM Symposium on SDN Research. ACM, New York, NY, USA, S. 49–61, 2019, ISBN: 9781450367103.