**Bachelorarbeit**

# Analyse der Echtzeitfähigkeit von Kubernetes-Umgebungen mittels eines Messstands für native Linux SPS in Automobilfabriken

# Analysis of real-time capabilities of Kubernetes environments using a test bench for native Linux PLCs in automotive factories

Harun **Taçlı**

Medieninformatik
Fakultät Elektrotechnik, Medien und Informatik
Ostbayerische Technische Hochschule Amberg-Weiden

| | |
|---|---|
| 1. Prüfer: | Prof. Dr.-Ing. Christoph Neumann |
| 2. Prüfer: | Prof. Dr.-Ing. Dominikus Heckmann |
| Externer Betreuer: | Christian Klaus Müller, B. Eng. |
| Ausgabetag: | 4. Oktober 2021 |
| Abgabetag: | 3. März 2022 |

Ostbayerische Technische Hochschule Amberg-Weiden
Fakultät Elektrotechnik, Medien und Informatik

Ostbayerische Technische Hochschule
Amberg-Weiden

Bestätigung gemäß § 12 APO

Name und Vorname
der Studentin/des Studenten: **Taçlı, Harun**

Studiengang: **Medieninformatik**

Ich bestätige, dass ich die Bachelorarbeit mit dem
Titel:

**Analyse der Echtzeitfähigkeit von Kubernetes-Umgebungen mittels eines Messstands für native Linux SPS in Automobilfabriken**

selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benützt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Datum: 3. März 2022

Unterschrift:

Ostbayerische Technische Hochschule Amberg-Weiden
Fakultät Elektrotechnik, Medien und Informatik

Ostbayerische Technische Hochschule
Amberg-Weiden

Bachelorarbeit Zusammenfassung

| | |
|---|---|
| Studentin/Student (Name, Vorname): | **Taçlı, Harun** |
| Studiengang: | Medieninformatik |
| Aufgabensteller, Professor: | Prof. Dr.-Ing. Christoph Neumann |
| Durchgeführt in (Firma/Behörde/Hochschule): | Audi AG |
| Betreuer in Firma/Behörde: | Christian Klaus Müller, B. Eng. |
| Ausgabedatum: 4. Oktober 2021 | Abgabedatum: 3. März 2022 |

Titel:

**Analyse der Echtzeitfähigkeit von Kubernetes-Umgebungen mittels eines Messstands für native Linux SPS in Automobilfabriken**

## Kurzfassung

Die Virtualisierung von Hardware gewinnt im Zeitalter der Digitalisierung in zunehmendem Maß an Bedeutung. Speicherprogrammierbare Steuerungen in der Fertigung bei der Audi AG werden derzeit als physikalische Hardware ausgeführt. Diese Applikationen mit Echtzeitanforderungen sollen zukünftig virtualisiert im Rechenzentrum betrieben werden. Die Virtualisierungslösung mittels virtuellen Maschinen ist mit einem großen Ressourcenaufwand verbunden, da hierbei auch ganze Betriebssysteme virtualisiert werden müssen. Außerdem ist es notwendig, diese Systeme mit Updates zu versorgen. Dies stellt einen weiteren Kosten- und Zeitfaktor dar. Deshalb werden in dieser Bachelorarbeit Container-Technologien und das Container-Management-System Kubernetes auf die Einsatzmöglichkeit für echtzeitkritische Anwendungen in der Automatisierung untersucht. Hierfür wird zunächst in einem lokalen Kubernetes-Cluster die Echtzeitfähigkeit mittels Latenzzeitmessungen ermittelt. Zudem werden in einer weiteren Untersuchung Datenpakete im Cluster ausgetauscht und ein Ausfall simuliert, um Paketverluste und Verzögerungen beim Neustart zu analysieren. Die Ergebnisse zeigen, dass eine Kubernetes-Umgebung Echtzeitanforderungen untersuchter Applikationen erfüllen kann. Anforderungen hinsichtlich Paketverluste und der Failover-Zeit werden mit der in dieser Arbeit aufgebauten Teststation nicht erfüllt. Beide Ansätze müssen weiterhin aktiv verfolgt und weitere Forschungen betrieben werden.

## Abstract

The virtualization of hardware is becoming increasingly important in the age of digitalization. Programmable logic controllers in production at Audi AG are currently run as physical hardware. In the future, these applications with real-time requirements are to be operated virtualized in the data center. The virtualization solution using virtual machines is associated with a large expenditure of resources, since entire operating systems must also be virtualized. It is also necessary to provide these systems with updates. This represents a further cost and time factor. Therefore, in this bachelor thesis container technologies and the container management system Kubernetes are investigated for their applicability for real-time critical applications in automation. For this purpose, the real-time capability is first determined in a local Kubernetes cluster by means of latency measurements. In a further investigation, data packets are exchanged in the cluster and a failure is simulated to analyze packet loss and restart delays. The results show that a Kubernetes environment can meet real-time requirements of studied applications. Requirements regarding packet loss and failover time are not met with the test station built in this work. Both approaches need to continue to be actively pursued and further research conducted.

## Schlüsselwörter

*Virtualisierung, Container, Virtuelle Maschine, Kubernetes, SPS*

# Danksagung

An dieser Stelle möchte ich mich bei allen Beteiligten bedanken, die mich während der Anfertigung dieser Bachelorarbeit unterstützt haben.

Zuerst gebührt mein Dank Herrn Prof. Dr.-Ing. Christoph Neumann, der meine Bachelorarbeit betreut und korrigiert hat. Für die hilfreichen Anregungen sowie die moralische Unterstützung möchte ich mich herzlich bedanken.

Ich bedanke mich bei Herrn Christopher Kolb, Herrn Christian Klaus Müller und Frau Binnaz Böyükbas, die mich fachlich betreut und mit viel Geduld unterstützt haben. Außerdem bedanke ich mich bei der gesamten Abteilung I/P4-21 der Audi AG in Ingolstadt für die Ermöglichung der Bachelorarbeit.

Besonders bedanken möchte ich mich bei Herrn Thomas Kampa, Herrn Marc Fischer und Herrn Amer El-Ankah für die hilfreichen Ideen und Materialien, die entscheidend dazu beigetragen haben, dass diese Bachelorarbeit in dieser Form vorliegt.

Abschließend möchte ich mich bei meiner Familie bedanken, die mich während meines Studiums finanziell unterstützt und mir den Rücken gestärkt hat.

# Inhaltsverzeichnis

# Abkürzungsverzeichnis

| | |
|---|---|
| BIOS | Basic Input/Output System |
| CPU | Central Processing Unit |
| ICMP | Internet Control Message Protocol |
| IO | Input-Output |
| K8s | Kubernetes |
| OS | Operating System |
| RAM | Random Access Memory |
| RT | Real-Time |
| RTT | Round Trip Time |
| Soft-SPS | Software-SPS |
| SPS | Speicherprogrammierbare Steuerung |
| SSD | Solid State Drive |
| VM | Virtual Machine |
| VMM | Virtual Machine Monitor |

# 1. Einleitung

Die Virtualisierung in der modernen Softwareentwicklung hat in den letzten Jahren zunehmend an Bedeutung gewonnen. Container-Technologien kommen in Unternehmen immer häufiger zum Einsatz und werden ein zentraler Baustein der Digitalisierung. Dabei hat sich Kubernetes innerhalb kurzer Zeit als Container-Management-System durchgesetzt. In Abbildung 1.1 ist das Resultat einer Studie des IT-Research- und Beratungsunternehmens Crisp Research aus dem Jahr 2019 dargestellt. Kubernetes war bei 21 % der Mittelstands- und Großunternehmen bereits im Einsatz. Bei über zwei Drittel war der Einsatz innerhalb der nächsten 36 Monate in Planung. [1]

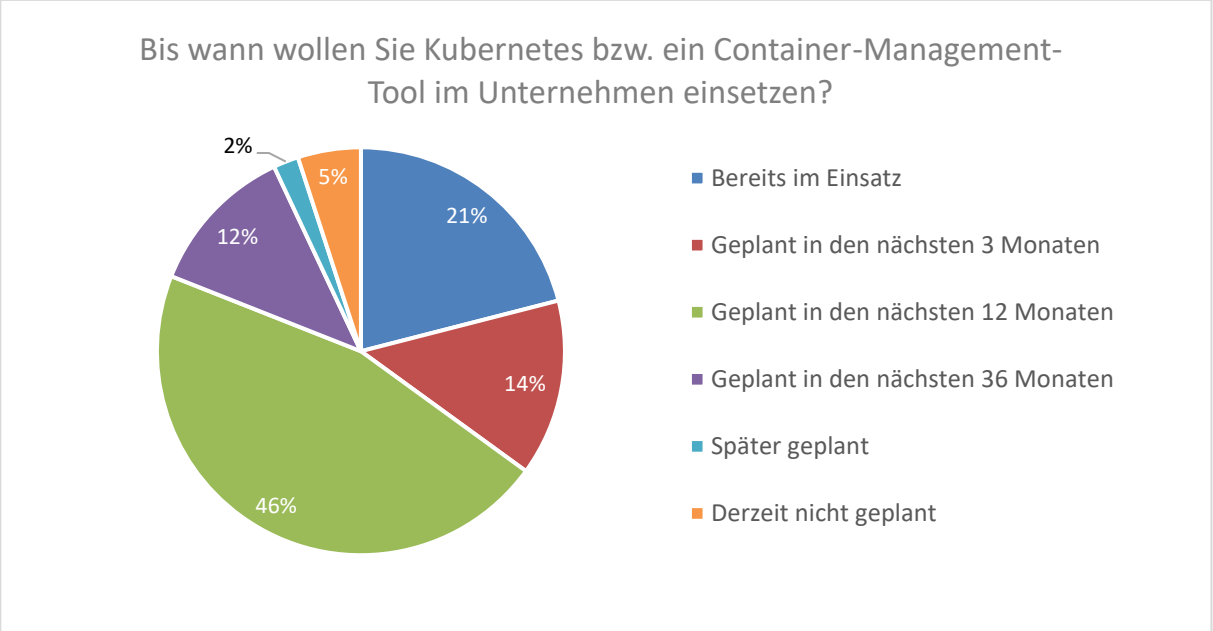



Abbildung 1.1: Ergebnis der Studie zum Einsatz von Kubernetes aus dem Jahr 2019. Eigene Darstellung, Anlehnung an [1]

## 1.1 Motivation

In der modernen Arbeitswelt werden Anforderungen an Software-Entwicklungen immer größer. So sind Schnelligkeit und Agilität von zentraler Bedeutung. Dabei wird Schnelligkeit nicht daran gemessen, wie viele Funktionalitäten in einem bestimmten Zeitraum geliefert werden können. Vielmehr geht es um das Bereitstellen von Anwendungen und die höchst zuverlässige Arbeitsweise eines Dienstes, während die Hochverfügbarkeit weiterhin gewährleistet ist. Zum Beispiel hat es früher weniger ein Problem dargestellt, wenn ein Service zu bestimmten Uhrzeiten aufgrund von Wartungsarbeiten inaktiv war. Jedoch hat insbesondere in der Welt der Automatisierung die durchgehende Verfügbarkeit eine hohe Relevanz. Mit Containern und Kubernetes werden Werkzeuge zur Verfügung gestellt, die benötigt werden, um die oben angesprochenen Anforderungen zu erfüllen. [2, S. 2]

Mit dem Container-Orchestrierungs-Tool soll die Hochverfügbarkeit von Services erreicht werden. So lässt sich die Skalierung eines Dienstes in einer Kubernetes-Umgebung deklarativ umsetzen. Dazu muss in einer Konfigurationsdatei lediglich die gewünschte Anzahl eingegeben werden. Anschließend wird der neue Status übernommen und von Kubernetes aktualisiert. Auf diese Weise kann die Anzahl an Ausführungen je nach Bedarf reduziert oder erhöht werden. [2, S. 6]

Der Einsatz von Kubernetes bringt unter anderem auch einen ökonomischen Vorteil mit sich, indem Maschinen besser ausgelastet und Ressourcen effizienter genutzt werden. Effizienz beschreibt das Verhältnis zwischen den beiden Faktoren Aufwand und Nutzen. Sobald ein Server gestartet wird, entstehen Kosten. Um dabei der Verschwendung vorzubeugen, sollte ungenutzte CPU-Zeit verhindert werden. Dazu müssen Systemadministratoren die Aufgabe übernehmen, die Nutzung von Servern auf einem zweckmäßigen Niveau zu halten. Die Konsequenz daraus ist ein kontinuierliches Management. Auch für diesen Fall liefert Kubernetes entsprechend hilfreiche Tools. Mit diesen lässt sich das Verteilen von Anwendungen in einem Cluster automatisieren. Somit wird im Gegensatz zu herkömmlichen Methoden ein deutlich höherer Grad der Nutzung erreicht. [2, S. 11]

Um die Vorteile von Container-Technologien mit Kubernetes als Orchestrierungsdienst nutzen zu können, muss sich mit dem Thema Virtualisierung beschäftigt werden. Vor allem in der Automobilindustrie wie etwa bei der Audi AG haben sich speicherprogrammierbare Steuerungen seit ihrer Einführung etabliert und sind zu einem unverzichtbaren Werkzeug in der Automatisierung geworden. Eine einfache Implementierung und die zuverlässige Arbeitsweise sind Gründe für den flächendeckenden Einsatz in Fertigungsanlagen [3]. Jedoch bringt die inzwischen hohe Anzahl an speicherprogrammierbaren Steuerungen in Form von Hardware den Nachteil mit sich, dass diese Systeme regelmäßig gewartet und auf Sicherheitslücken überprüft werden müssen. Darüber hinaus steht im Zuge der Digitalisierung auch die Wirtschaftlichkeit im Fokus. Um diesen Problemen nachzugehen, setzen Unternehmen zunehmend auf unterschiedliche Virtualisierungslösungen. Für die Virtualisierung kann neben der Containerisierung auch die klassische Methode mit virtuellen Maschinen verwendet werden. Diese Umstellung soll für den Entfall von Hardware in übermäßiger Anzahl sowie zur leichten Wartbarkeit und Flexibilität sorgen. Im Rahmen der Virtualisierung von zeitkritischen Applikationen wie in diesem Fall speicherprogrammierbare Steuerungen muss insbesondere die Echtzeitfähigkeit gewährleistet werden, da es sonst in der Industrie zu erheblichen Schäden sowohl an Anlagen als auch bei Menschen kommen kann.

## 1.2 Problemstellung

Die schnell fortschreitende Digitalisierung und der Wandel in der Industrie erfordert auch die Weiterentwicklung der Automatisierungstechnik. Speicherprogrammierbare Steuerungen, die in der Produktion für die Steuerung und Verwaltung von Robotern zuständig sind, werden heute als physikalische Hardware ausgeführt. In Zukunft sollen diese Systeme in virtualisierter Form als reine Software betrieben werden. Infolgedessen ergeben sich neue Anforderungen an die Infrastruktur des Rechenzentrums, da diese Applikationen in Echtzeit und deterministisch ausgeführt werden müssen. Um diese Anforderungen zu erfüllen, werden aktuell virtuelle Maschinen eingesetzt. Der Nachteil dabei ist, dass das gesamte Betriebssystem der virtuellen Maschinen ebenfalls virtualisiert werden muss und das zu einem großen Ressourcenaufwand führt. Aus diesem Grund ist für die Virtualisierung der speicherprogrammierbaren Steuerungen eine containerbasierte Lösung zu bevorzugen.

## 1.3 Zielsetzung

Das Ziel dieser Bachelorarbeit ist es, Container-Technologien und den dazugehörenden Orchestrierungsdienst Kubernetes auf die Einsatzmöglichkeit für Echtzeitapplikationen zu untersuchen. Mit Messprogrammen werden im Laufe der Analyse mehrere Testfälle unter diversen Bedingungen abgedeckt. Dabei liegt der Fokus auf Latenzzeiten innerhalb eines Kubernetes-Clusters. Darüber hinaus wird zusätzlich eine Ausfallsituation mit den damit verbundenen Paketverlusten betrachtet. Mithilfe der gemessenen Werte wird eine Aussage über die grundsätzliche Eignung von Kubernetes in der Produktion getroffen.

## 1.4 Aufbau der Arbeit

Die Arbeit gliedert sich in 5 Kapitel. Nach der Einleitung werden in Kapitel 2 die technischen Grundlagen erklärt, die zum Verständnis und für die Umsetzung benötigt werden. Kapitel 3 beschreibt das Test-Setup und die anschließende Durchführung der Messvorgänge. Die Darstellung der Ergebnisse sowie die Validierung erfolgen in Kapitel 4. Abschließend wird die Arbeit in Kapitel 5 zusammengefasst und ein Ausblick für weitere Forschungsgegenstände gegeben.

# 2. Grundlagen

In den folgenden Kapiteln werden die für das Verständnis dieser Bachelorarbeit erforderlichen Grundlagen von Container-Technologien und der dazugehörenden Orchestrierung beschrieben. Zunächst wird allgemein die Bedeutung der Virtualisierung erklärt sowie dessen Vorteile aufgeführt. Anschließend wird auf virtuelle Maschinen und Container-Technologien eingegangen. Die Erklärung von grundlegenden Begriffen aus der Kubernetes-Umgebung und der speicherprogrammierbaren Steuerung sowie die Beschreibung der verwendeten Messmethoden bildet den Schluss des Kapitels.

## 2.1 Virtualisierung und seine Vorteile

Unter Virtualisierung versteht man allgemein eine Technologie, die physisch vorhandene Ressourcen in Form von Hardware, Software oder Netzwerke in abstrakter Weise nachbildet. Das Konzept reicht bis in die 1960er Jahre zurück. Schon damals hat IBM die Hardware-Virtualisierung für eine bessere Auslastung der zu der Zeit sehr knappen Ressourcen benutzt. Bis heute hat sich die Virtualisierung zu einer bedeutungsvollen Technik in der IT-Branche entwickelt. Mit dieser Technologie ist es möglich, mehrere Applikationen mit unterschiedlichen Aufgaben in einem gemeinsamen Server unterzubringen. Die Anwendungen laufen parallel und abgekapselt voneinander, sodass ein im Normalfall störungsfreier Betrieb sichergestellt ist. Dies führt dazu, dass die Ressourcen eines Servers deutlich effizienter genutzt werden können. Abbildung 2.1 veranschaulicht zwei Server, die nicht vollständig ausgelastet werden. [4, S. 8-10]

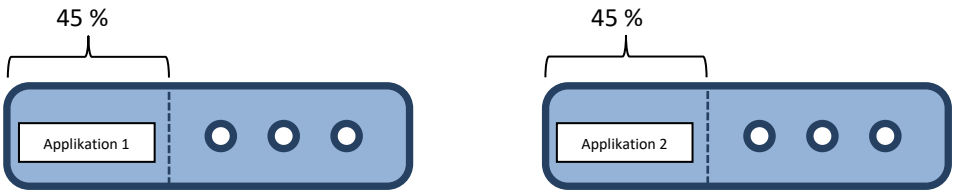



Abbildung 2.1: Beide Server arbeiten nicht effizient. Eigene Darstellung, Anlehnung an [5]

In Abbildung 2.2 werden durch die Virtualisierung zwei Applikationen in einem Server betrieben. Die Rechner, die man auf diese Weise erhält, können entweder für die Ausführung anderer Aufgaben eingesetzt oder komplett abgeschaltet werden. Das Reduzieren von physikalischer Hardware im Rechenzentrum bedeutet logischerweise auch das Einsparen von Strom-, Kühl- und Wartungskosten.


Abbildung 2.2: Applikationen werden in einen gemeinsamen Server übergeführt. Eigene Darstellung, Anlehnung an [4]

## 2.2 Virtuelle Maschinen (VM)

Eine virtuelle Maschine weist dieselben Funktionen wie ein physischer Rechner auf. Diese führen wie herkömmliche Computer Anwendungen und ein Betriebssystem aus. Die Kommunikation zwischen dem Host-System und dem Gastsystem der virtuellen Maschine erfolgt über den sogenannten Hypervisor, welcher im nächsten Abschnitt näher beschrieben wird. Auf einem Host-System können mehrere voneinander isolierte VMs mit unterschiedlichen Betriebssystemen parallel betrieben werden. Die Ressourcen des physikalischen Servers werden dabei untereinander aufgeteilt. Vorzugsweise werden VMs eingesetzt, wenn bestimmte Anwendungen beispielsweise zu Testzwecken unterschiedliche Betriebssysteme erfordern. Mithilfe verschiedener Virtualisierungslösungen wie zum Beispiel Oracle VirtualBox lassen sich virtuelle Maschinen plattformunabhängig erstellen. [4, S. 13] [6, S. 14]

### 2.2.1 Hypervisor

Beim Hypervisor, auch Virtual-Machine-Monitor (VMM) genannt, handelt es sich um eine Softwareschicht, die zwischen Hostsystem und dem virtuellen Gastsystem liegt. Über ihn werden den virtuellen Instanzen Ressourcen wie CPU, RAM oder Festplattenspeicher zugewiesen. Dabei ist die Aufteilung und Zuweisung für die virtuellen Maschinen nicht erkennbar. Sie interpretieren die zur Verfügung gestellten Ressourcen als singuläre Einheit. Der VMM stellt die Laufzeitumgebung für die virtuellen Maschinen dar und sorgt für die Isolation der Gastsysteme untereinander, sodass diese sich nicht gegenseitig stören oder beeinflussen können. [4, S. 11-13]

**Hypervisor Typ-1**

Der Hypervisor wird in zwei Typen unterschieden. Der Typ-1 Hypervisor wird auch als Bare-Metal Hypervisor bezeichnet und läuft ohne ein Betriebssystem direkt auf der Hardware des Hosts. Er weist den Gastsystemen in der Rolle eines Host-Betriebssystems Ressourcen zu. Weil er direkt mit der darunterliegenden Hardware kommunizieren kann, gilt der Typ-1 Hypervisor als der Typ mit dem besseren Performanceverhalten als der Typ-2 Hypervisor. [7, S. 6]

**Hypervisor Typ-2**

Im Fall des Typ-2 Hypervisors ist bereits ein Betriebssystem installiert. Der Vorteil ist, dass der Hypervisor die existierenden Treiber des Host-Betriebssystems für die Kommunikation mit der Hardware nutzen kann. Da hierbei durch das Betriebssystem und eventuell andere Programme Ressourcen verbraucht werden, fällt die Systemleistung jedoch geringer aus.
[7, S. 6]

In Abbildung 2.3 ist der Unterschied zwischen den beiden Typen dargestellt.

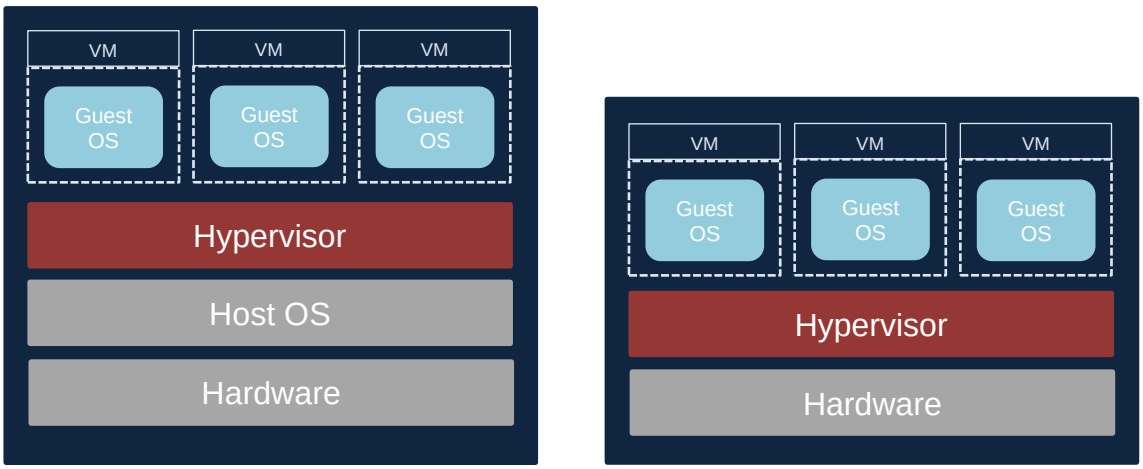



Abbildung 2.3: Vergleich Typ-1 Hypervisor und Typ-2 Hypervisor.
Eigene Darstellung, Anlehnung an [8]

## 2.3 Container-Technologien

Die Digitalisierung soll die Effizienz steigern und Mitarbeiter entlasten [9, S. 44-45]. Jedoch ist mit der hohen Anzahl an Softwareanwendungen, bei welchen auch weiterhin mit einer kontinuierlichen Zunahme zu rechnen ist, auch eine höhere Komplexität verbunden. Um diese steigende Last zukünftig zu bewältigen, sind geeignete Technologien erforderlich. Diese sollen für Flexibilität und Agilität sorgen, auf die immer mehr der Fokus gesetzt wird. So stellen containerbasierte Technologien eine mögliche Lösung und Alternative zu den virtuellen Maschinen dar. In den folgenden Abschnitten wird der Begriff Container und auch der Unterschied zu Hypervisor-basierten virtuellen Maschinen näher erläutert.

### 2.3.1 Container

Der Begriff Container aus dem Industriebereich ist inzwischen auch in der IT angesiedelt. Beispielsweise werden Transportcontainer in der Logistik dazu eingesetzt, unterschiedliche Waren schnell und unkompliziert von einem Ort zum anderen zu verschicken. Analog werden in IT-Unternehmen die gleichen Ziele verfolgt, um Anwendungen auf eine schnelle Art bereitzustellen und allgemein der Komplexität bei der Entwicklung vorzubeugen. Ein Container beinhaltet hierbei eine Anwendung mit allen dazugehörenden Abhängigkeiten, die benötigt werden, damit die Anwendung lauffähig ist.

Vor allem sollen Container Applikationen portierbar und isolierbar machen. Im Gegensatz zu den im Abschnitt 2.2 beschriebenen virtuellen Maschinen kommen Container ohne ein eigenes Betriebssystem aus. Stattdessen teilen sie sich den Betriebssystem-Kernel des Host-Systems. [10, S. 15]
Im Abschnitt 2.3.4 werden Vorteile der containerbasierten Lösung gegenüber der klassischen Virtualisierung aufgeführt.

### 2.3.2 Microservices

In Verbindung mit Container-Technologien fällt auch häufig der Begriff Microservices. Darunter versteht man im Grunde die Modularisierung von Software. Durch die Zerlegung von großen und komplexen Systemen in kleinere Bausteine können diese besser beherrscht und verstanden werden. Zudem erleichtern Microservices die Wartbarkeit und den Austausch von Modulen, um Systeme leicht an spezielle Bedürfnisse anzupassen. Darüber hinaus lassen sich Microservices problemlos skalieren, ohne dass das gesamte System betroffen sein muss. Die Kommunikation untereinander erfolgt über explizite Schnittstellen. Microservices einer Anwendung können unabhängig voneinander ausgeführt werden. Am Beispiel eines Online-Shops ist die Produktsuche oder wie etwa das Ablegen eines Produkts in den Warenkorb ein eigenständiger Service. Auf der Gegenseite zu Microservices stehen die Monolithen. Bei der monolithischen Architektur handelt es sich um ein System, das im Ganzen bereitgestellt wird. Das bedeutet, dass bei Änderungen die ganze Anwendung betroffen ist. In Abbildung 2.4 ist der unterschiedliche Aufbau von Microservices und Monolithen vereinfacht dargestellt.
[11, S. 2-6]

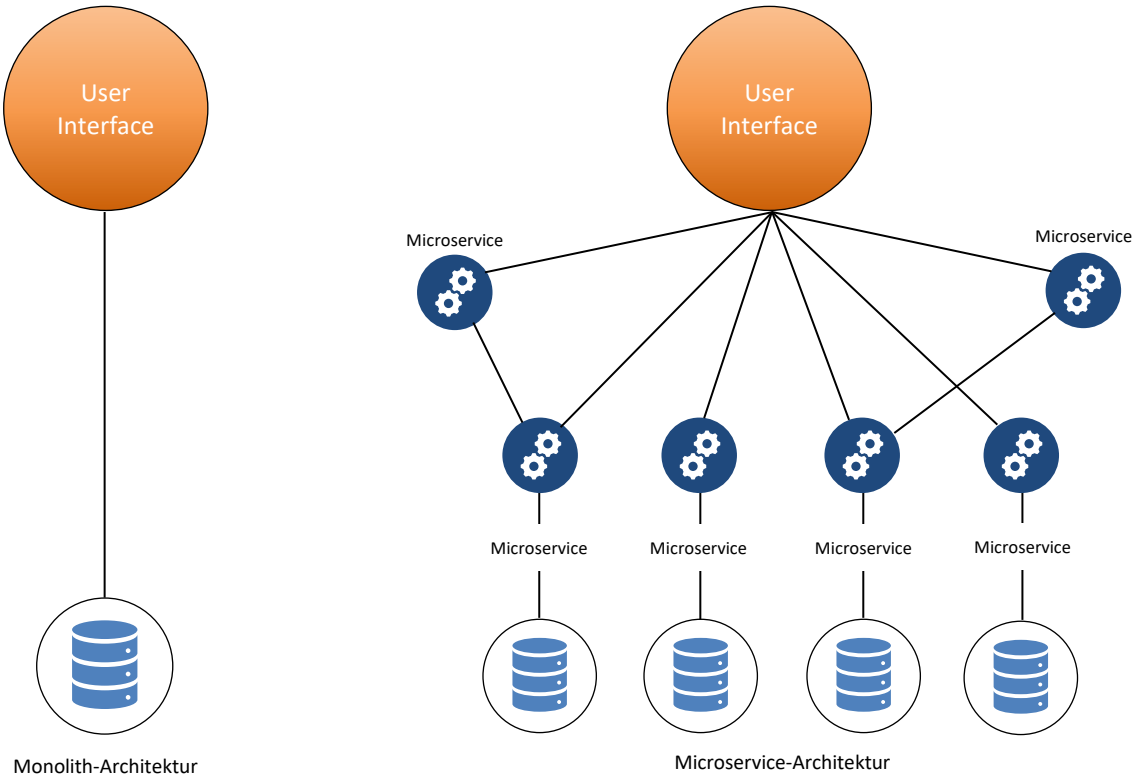



Abbildung 2.4: Monolith-Architektur und Microservice-Architektur.

Eigene Darstellung, Anlehnung an [12]

### 2.3.3 Docker

Docker stellt die Container-Runtime-Engine dar und ist die am weitesten verbreitete Technologie in der Containerumgebung. Sie ermöglicht die Container-Virtualisierung von Anwendungen und bildet die Schnittstelle zwischen dem Hostsystem und den Containern. Mit Docker lassen sich Anwendungen in Containern automatisiert bereitstellen. Die Engine verwendet den Linux-Kernel und kapselt die verpackten Anwendungen ab. Somit ist ein paralleler und ein voneinander unabhängiger Betrieb sichergestellt. Außerdem ist man beim Entwickeln mit Docker an kein bestimmtes Betriebssystem gebunden. Anwendungen können unabhängig davon auf unterschiedlichen Systemen ohne Probleme gestartet werden. Die Software wurde ursprünglich für den Einsatz auf Linux-Systemen entwickelt. Auf der offiziellen Webseite von Docker kann aber die Installation auch auf Windows- und MacOS-Geräten problemlos ausgeführt werden. [13, S. 11-12]

### 2.3.4 Docker-Komponenten

**Dockerfile**

Die Dockerfile ist ein Textdokument, welches die Anleitung zum Bauen eines Docker-Images darstellt. In diesem werden mehrere Schritte definiert, um ein Image zu erzeugen. Jedes Dockerfile baut auf anderen Images auf. Dabei muss die Datei stets *Dockerfile* lauten und mit einem großen Buchstaben beginnen. Im Kapitel 3 wird im Rahmen der Messdurchführungen eine Dockerfile erstellt, auf dessen Aufbau und Inhalt näher eingegangen wird. [14, S. 24]

**Docker-Image**

Ein Image ist ein ausführbares Paket, welches alle notwendigen Dateien zum Ausführen einer Anwendung enthält. Neben dem selbstständigen Bauen können auch vorgefertigte Images aus einer Registry wie *DockerHub* [1] heruntergeladen werden. In diesem Fall ist kein Dockerfile erforderlich. [2, S. 14]

### 2.3.5 Dockerfile-Befehle

In diesem Abschnitt werden einige wichtige Befehle in einer Dockerfile erklärt, die in dieser Arbeit verwendet werden.

**FROM**

Eine Dockerfile beginnt stets mit dem FROM-Befehl. Damit wird das Basisimage definiert, auf welchem das neue Image aufbauen soll. Hier besteht die Möglichkeit, entweder ein Betriebssystem festzulegen oder auch ein Image aus einer Registry zu benutzen. Hinter der Angabe kann auch eine bestimmte Version stehen.

**RUN**

Hinter der RUN-Anweisung werden Shell-Befehle ausgeführt, um zum Beispiel Programme zu aktualisieren und zu installieren. Mit jedem RUN-Befehl wir eine neue Schicht im Image erzeugt, wodurch Speicherressourcen verbraucht werden. Um das zu vermeiden, sollten alle Shell-Befehle mit so wenig RUN-Anweisungen wie möglich ausgeführt werden.

[1] https://hub.docker.com/, Zugriff am 24.02.2022

**CMD**

Die Hauptaufgabe von CMD ist, eine Aktion beim Starten des Containers standardmäßig auszuführen. Zum Beispiel kann es sich hierbei um eine ausführbare Datei handeln. Eine Dockerfile kann nur aus einer CMD-Anweisung bestehen. Im Fall von mehreren definierten CMD-Anweisungen wird nur die letzte CMD berücksichtigt.

**ENTRYPOINT**

ENTRYPOINT fungiert ähnlich wie CMD als Einstiegspunkt beim Starten eines Containers. Der Unterschied ist jedoch, dass ENTRYPOINT im Gegensatz zu CMD nicht überschrieben werden kann, auch wenn der Container mit eigenen Kommandozeilenbefehlen gestartet wird. Beide Befehle können auch kombiniert in einer Dockerfile enthalten sein. Mit ENTRYPOINT kann zum Beispiel eine Datei ausgeführt werden und CMD beinhaltet dafür die Standard-parameter.

In der folgenden Abbildung 2.5 ist der Lebenszyklus von Docker in vereinfachter Form dargestellt.

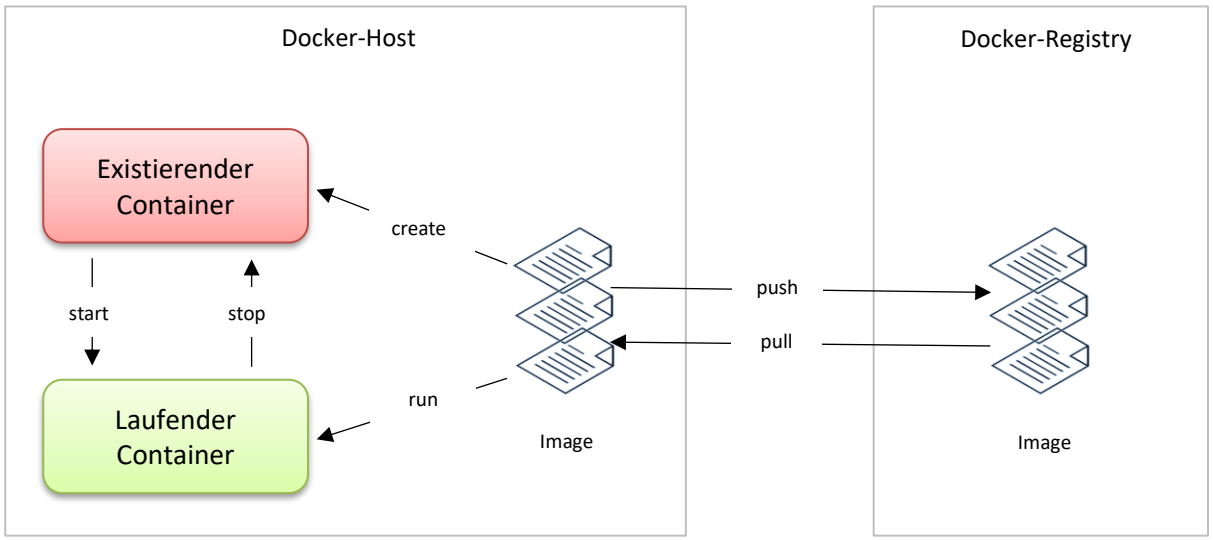



Abbildung 2.5: Lebenszyklus der Container-Technologie.
Eigene Darstellung, Anlehnung an [15, S. 2]

### 2.3.6 Dateisystem

Wie aus den vorherigen Seiten bereits hervorgeht, handelt es sich beim Aufbau von Docker-Images um ein Modell mit mehreren aufeinanderliegenden Schichten. Das Basis-Image bildet auf dem Kernel die erste Schicht. Zum Schluss kann mit den erzeugten Images ein Container gestartet werden. Abbildung 2.6 veranschaulicht das Schichtenmodell in beispielhafter Form.

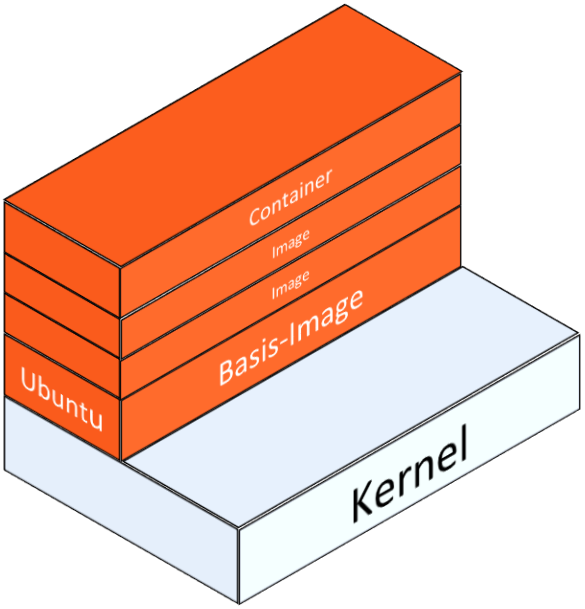



Abbildung 2.6: Docker-Architektur mit den einzelnen Schichten.

Eigene Darstellung, Anlehnung an [16]

### 2.3.7 Vorteile der containerbasierten Virtualisierung

Das Problem bei der Hypervisor-basierten Virtualisierung besteht darin, dass ganze Betriebssysteme mit virtualisiert werden. Das führt zu einem starken Ressourcenverbrauch, was auch als Overhead bezeichnet wird. Hinzu kommt, dass diese virtuellen Maschinen mit Sicherheitsupdates versorgt und verwaltet werden müssen. Ein weiteres Problem sind die Lizenzen von Software und der Betriebssysteme auf virtuellen Maschinen. Im Gegensatz dazu arbeiten Container effizienter und sind deutlich schlanker, weil hier die Virtualisierung des Betriebssystems entfällt. Durch den niedrigeren Ressourcenverbrauch können außerdem deutlich mehr Container auf einem Hostsystem betrieben werden als es bei virtuellen Maschinen der Fall wäre.

Allgemein führen die aufgezählten Gründe dazu, dass mit der Verwendung von Containern eine höhere Flexibilität und schnellere Bereitstellung bei gleichzeitigem Entfall von Mehraufwand ermöglicht wird. [6, S. 13] [17, S. 7-8]

In Abbildung 2.7 wird der unterschiedliche Aufbau von virtuellen Maschinen und Containern schematisch dargestellt.

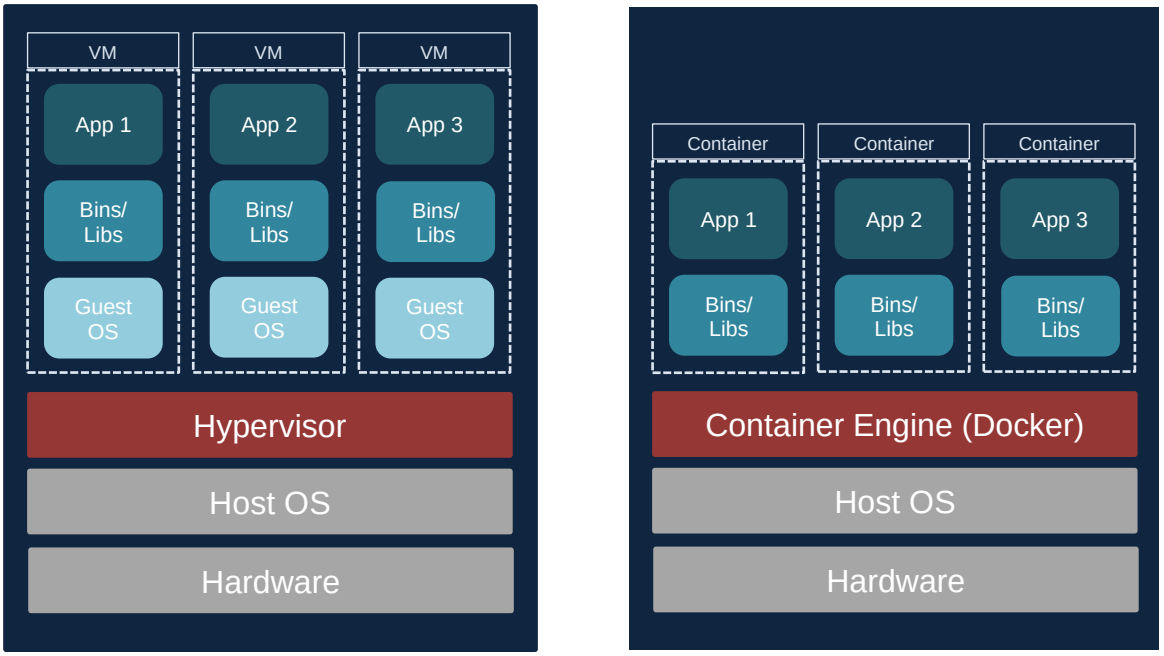



Abbildung 2.7: Bei Containern entfällt die Virtualisierung eines Gast-Betriebssystems. Eigene Darstellung, Anlehnung an [18]

## 2.4 Kubernetes (K8s)

Kubernetes, häufig mit K8s abgekürzt, ist für die Verwaltung von containerisierten Anwendungen zuständig. Das Open-Source-Projekt wurde ursprünglich von Google entwickelt und 2014 der Öffentlichkeit zur Verfügung gestellt. Seitdem wird es von der Community stetig weiterentwickelt. Mittlerweile ist Kubernetes im Bereich der Containerisierung die am meisten genutzte Plattform weltweit. Die Entwicklung des Orchestrierungs-Dienstes war die Antwort auf die Frage, wie eine hohe Anzahl an Containern administriert werden können. Die Menge der täglichen Suchanfragen bei Google ist ein Beispiel dafür, dass ohne den Einsatz von Containern ein sehr hoher Ressourcenaufwand betrieben werden müsste. Mit einer bis heute gut erprobten Infrastruktur bietet Kubernetes eine Software, die containerisierte skalierbare Anwendungen je nach Bedarf schnell und effizient bereitstellen kann. [19, S. 50] [2, S. 1] [20, S. 25-26]

### 2.4.1 Architektur

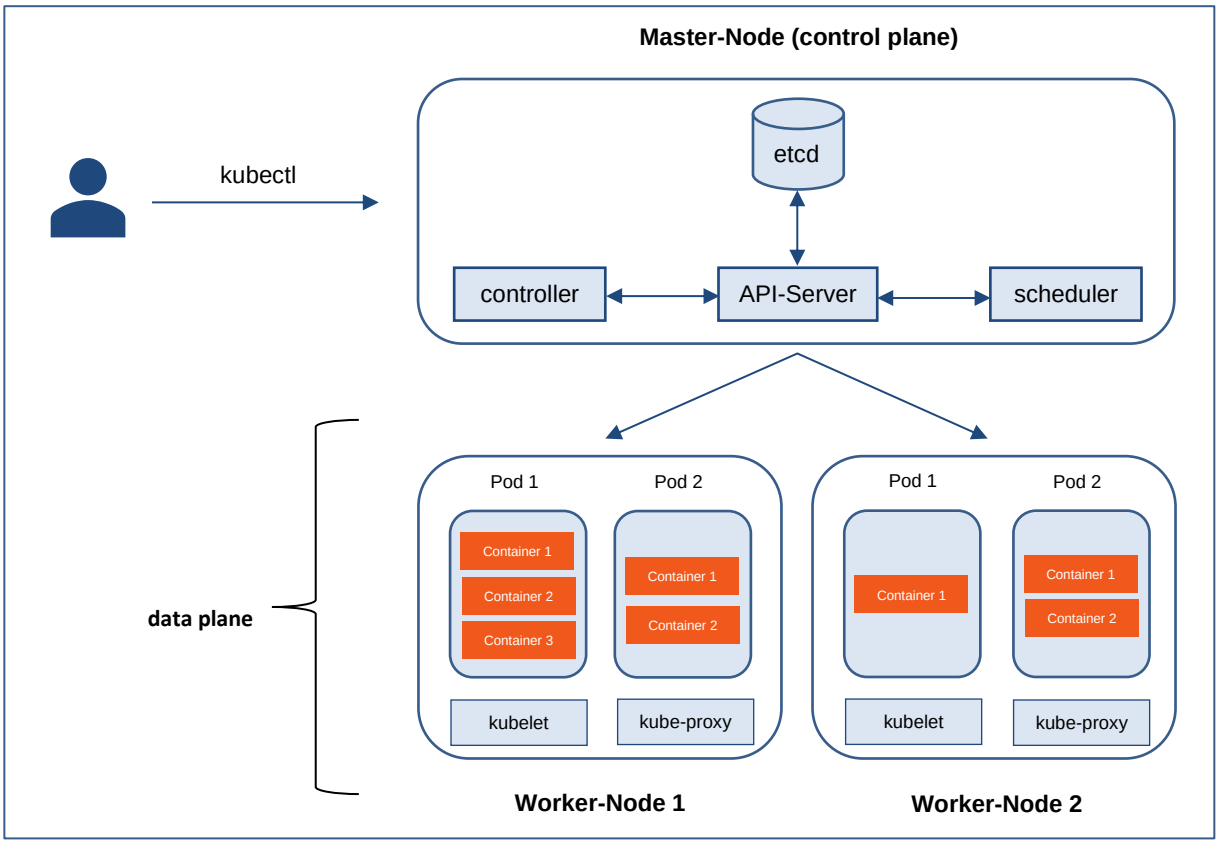



Abbildung 2.8: Kubernetes-Architektur. Eigene Darstellung, Anlehnung an [21]

### 2.4.2 Komponenten und grundlegende Begriffe

In Abbildung 2.8 ist zu erkennen, dass die Architektur von Kubernetes in zwei unterschiedliche Bereiche aufgeteilt ist. Die *control plane* mit dem *Master-Node* und die *data plane* mit den *Worker-Nodes*. Im Folgenden wird auf die einzelnen Komponenten in der Architektur näher eingegangen.

**Kubernetes-Master**

Der Kubernetes-Master steuert und kontrolliert die Daten-Ebene (*data plane*) mit den Arbeitsknoten. Er reagiert zum Beispiel auf Ausfälle von Pods im Cluster und ergreift Maßnahmen. Der Master besteht zudem aus den Komponenten *etcd*, *controller*, *API-Server* und *scheduler*. In Produktionsumgebungen sollten mehrere Instanzen des Masters vorhanden sein, um die Ausfallsicherheit zu gewährleisten. [22, S. 8]

**Nodes**

Ein *Node* ist im Kubernetes-Cluster ein Arbeitsknoten. Der *Node* kann eine virtuelle oder physische Maschine sein, in welchem Pods mit Containern laufen. Ihm können bestimmte Ressourcen mit Obergrenzen zugeteilt werden. Ein Cluster kann aus einer beliebigen Anzahl an *Nodes* bestehen.

**Pods**

Die kleinste Einheit, die in einem Kubernetes-Cluster bereitgestellt werden kann, wird als *Pod* bezeichnet. Alle Anwendungen, die im gleichen *Pod* ausgeführt werden, besitzen die gleiche IP-Adresse. Hingegen sind Applikationen in unterschiedlichen *Pods* voneinander isoliert und haben auch unterschiedliche IP-Adressen. Mehrere Container sollten aber nur dann im gleichen *Pod* laufen, wenn diese auch miteinander verknüpft sind und zum Beispiel auf eine gemeinsame Datenbank zugreifen. In Konfigurationsdateien im YAML-Format werden *Pods* spezifiziert, wie zum Beispiel das Festlegen von Ressourcen-Anforderungen. [2, S. 45-46]

**kubectl**

Bei *kubectl* handelt es sich um den Kubernetes-Client. Mit diesem Befehlszeilentool findet die Interaktion mit der Kubernetes-API statt. Anwender können mit *kubectl* Kubernetes-Objekte wie zum Beispiel Pods oder ReplicaSets erstellen. Mit dem Befehl *kubectl* lassen sich auch diverse Statusinformationen des Clusters abfragen. [2, S. 31]

**etcd**

Der *etcd* ist ein primärer Datenspeicher im Kubernetes-Master. In diesem werden Konfigurationsdaten sowie Zustände des Clusters abgespeichert.

**scheduler**

Der Kubernetes-Scheduler ist eine Komponente auf dem Master-Knoten, der für die Verteilung von neu erstellten Pods auf die einzelnen Knoten zuständig ist. Die maximale Auslastung eines Knotens, was eine gute Verteilung der Pods bedeutet, führt zu einer sehr guten Effizienz im Cluster. [20, S. 26]

**controller**

Der *controller* ist eine weitere Komponente des Master-Knotens. Er verwaltet verschiedene *controller* in der Datenebene. Dazu gehören die *Node-Controller* für das Erkennen und Reagieren bei Knotenausfällen, *Replication-Controller* für das Sicherstellen der korrekten Anzahl von Pods sowie die *Endpoint-Controller* für das Verbinden von Pods mit Services.
[20, S. 26]

**API-Server**

Der *API-Server* ist der Kern der *control plane*. Über diesen zentralen Zugriffspunkt können Anwender und Komponenten außerhalb des Kubernetes-Clusters miteinander kommunizieren. [22, S. 8]

**kubelet**

Beim *kubelet* handelt es sich um einen sogenannten „Agenten“ auf allen Arbeitsknoten im Cluster. Er stellt sicher, dass die Container in einem Pod ordnungsgemäß laufen.

**kube-proxy**

Über den *kube-proxy* wird der Netzverkehr im und außerhalb des Clusters geregelt. Dafür muss der *kube-proxy* auf jedem Knoten vorhanden sein.

**ReplicaSet**

Für eine bestimmte Anzahl an Kopien von einem Pod wird ein *ReplicaSet* verwendet. Statt mehrere Instanzen eines Pods manuell zu erstellen, ist es somit möglich, diese Aufgabe zu automatisieren. Für die Verwendung von mehreren Kopien gibt es einige Gründe. Zum einen wird mit der Redundanz dafür gesorgt, dass bei auftretenden Fehlern ein Toleranzbereich vorhanden ist. Außerdem besteht der Hauptzweck eines *ReplicaSets* darin, Pods zu skalieren, um somit entsprechend viele Requests verarbeiten zu können. Fällt ein Pod aus oder muss ein Update durchgeführt werden, so stellt ein *ReplicaSet* immer die gewünschte Anzahl an Pods zur Verfügung. In Abbildung 2.9 ist zu sehen, dass über einem *ReplicaSet* das *Deployment* steht und Pods hingegen von *ReplicaSets* gesteuert werden. [2, S. 105-106]

**Deployment**

In einem erzeugten Kubernetes-Cluster können containerisierte Anwendungen mit *Deployments* final bereitgestellt werden. In der Deployment-Konfigurationsdatei wird die Art der Erstellung und auch die Aktualisierung der Anwendungen festgelegt. Der Master plant die Verteilung auf die einzelnen *Nodes*. Jede Instanz wird dabei von einem *Deployment-Controller* überwacht. Fällt ein Arbeitsknoten mit den jeweiligen Instanzen aus oder wird gelöscht, wird vom Master ein neuer *Node* ausgewählt, in welchem die Instanzen neu ausgeführt werden. Die Besonderheit ist, dass auf diese Weise ein Selbstheilungsmechanismus geboten wird, um Ausfälle zu vermeiden und eine Hochverfügbarkeit zu erreichen. [22, S. 10-11]

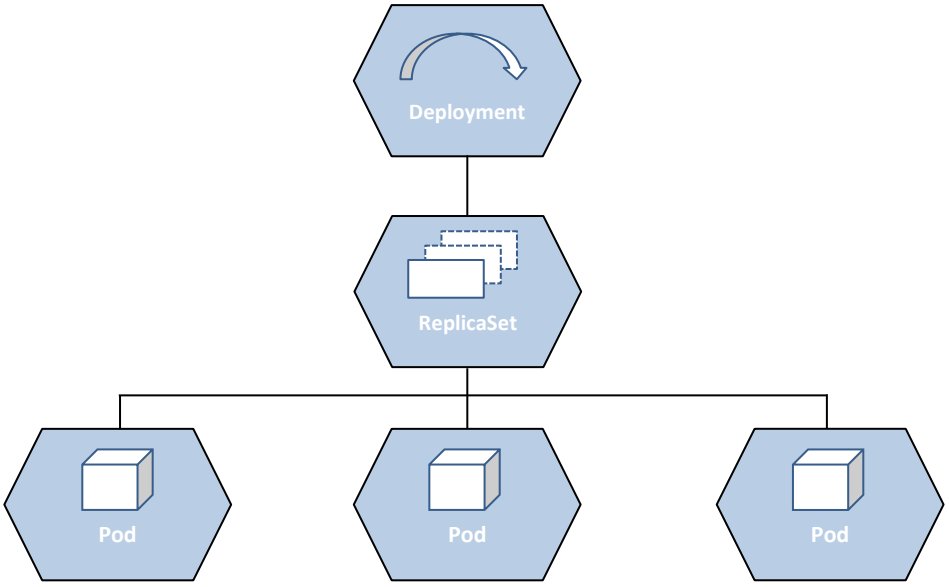



Abbildung 2.9: Logische Verkettung von Deployments, ReplicaSets und Pods.

## 2.5 Speicherprogrammierbare Steuerungen (SPS)

Mit speicherprogrammierbaren Steuerungen werden Roboter in der Produktion angesteuert. Sie haben den gleichen Strukturaufbau wie ein Rechner, bestehend aus einer zentralen Steuerungseinheit (CPU), einer Stromversorgung, digitalen Ein- und Ausgängen sowie einem internen Bussystem. Diese Elemente sind Bestandteil einer einfachen SPS. Bei Bedarf können auch weitere Komponenten hinzugefügt werden. Damit die speicherprogrammierbare Steuerung funktioniert, muss auf diese ein Programm geladen und durchlaufen werden. [23, S. 8]

In der Audi AG ist im konzerneinheitlichen Lastenheft für Anlagenelektrik festgelegt, dass die Zykluszeit der SPS 40 ms nicht überschreiten darf [24, S. 11]. Diese kann je nach Größe des Programms unterschiedlich ausfallen. Da es sich hierbei aber um die Zeit des Programmablaufs in einer SPS handelt und nicht um Anforderungen für Latenzzeiten, wird im weiteren Verlauf dieser Arbeit nicht näher darauf eingegangen.

### 2.5.1 Vorteile einer Soft-SPS

Da die SPSen zukünftig virtuell betrieben werden sollen, werden im Folgenden einige Vorteile aufgezeigt. Die Funktionalitäten der Software- und hardwarebasierten SPS sind identisch. Der Unterschied besteht lediglich darin, dass bei einer Soft-SPS die physikalischen Baugruppen entfallen. Die Reduzierung von dedizierten Hardwarekomponenten kann sich auch auf die Zuverlässigkeit des Systems positiv auswirken. Die Soft-SPS kann auf einem Industrierechner oder auch auf einem gewöhnlichen Computer mit beispielsweise Windows oder Linux laufen. Parallel ist es möglich, dass das Betriebssystem weitere Prozesse ausführen kann. Darüber hinaus stehen den Soft-SPSen ein deutlich höherer Speicherplatz und auch CPU-Leistung zur Verfügung, wenn sie auf einem Industrie-PC im Einsatz sind und somit auf die Rechner-Ressourcen zugreifen können. Die Hardwareressourcen werden hierbei virtualisiert durch beispielsweise den Hypervisor zur Verfügung gestellt. Auch der Bedarf eines zusätzlichen Programmiergerätes zum Ändern der Schaltungen entfällt. Falls ein nicht echtzeitfähiges Betriebssystem, wie zum Beispiel Windows, verwendet wird, muss dieses zunächst für eine Automatisierungsumgebung modifiziert werden. [25, S. 9] [25, S. 29]

## 2.6 Echtzeit

Der Begriff Echtzeit beschreibt allgemein die Eigenschaft, eine Reaktion durch das System in einer definierten Zeitspanne zu erhalten. Während es auf der einen Seite um Geschwindigkeit geht, spielt auf der anderen Seite auch das deterministische Verhalten eine wichtige Rolle, da es vorkommen kann, dass ein System trotz hoher Geschwindigkeit die Zeitvorgaben nicht einhalten kann. Dabei versteht man unter Determinismus die Vorhersagbarkeit eines Systems, also bei gleichen Systemeingängen immer die gleichen Systemausgänge als Resultat.
[10, S. 13-14]

Im Folgenden werden 2 Klassifizierungen bei Echtzeitsystemen näher beschrieben.

- **Weiche Echtzeit:** Hierbei handelt es sich um eine Bedingung, bei der es zu keinen Systemausfällen oder kritischen Konsequenzen bei Mensch und Maschine kommt, wenn das Zeitlimit nicht eingehalten werden kann. Das System läuft zwar weiter, aber die Qualität des Endprodukts sinkt nach der Zeitüberschreitung. [26, S. 26] [27, S. 3]

- **Harte Echtzeit:** In diesem Fall wird eine Überschreitung des Limits nicht toleriert. Das bedeutet, das System muss in einer fest vorgeschriebenen Zeit reagieren. Andernfalls kann dies zu erheblichen Schäden sowohl bei Menschen als auch bei Maschinen führen. [26, S. 26] [27, S. 3]

In Abbildung 2.10 ist eine Grafik dargestellt, die den Unterschied von weichen und harten Echtzeitanforderungen im Zusammenhang zur Qualität eines Produktes veranschaulicht.

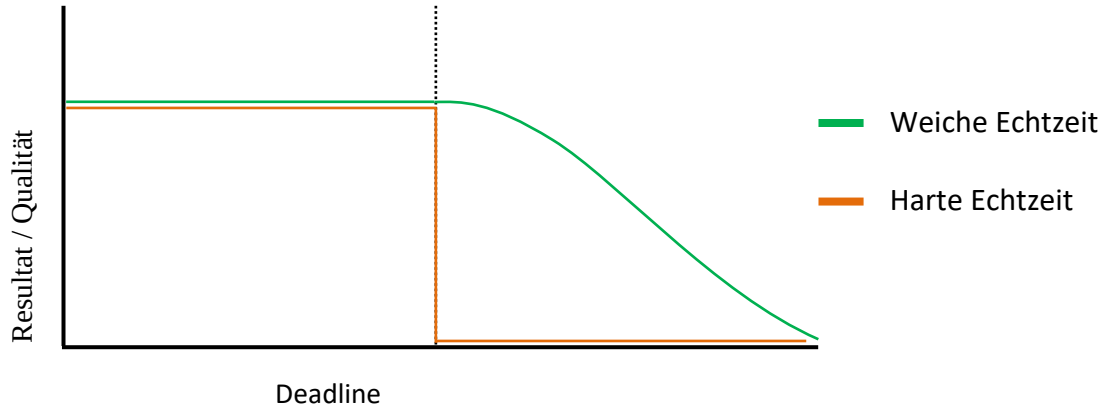



Abbildung 2.10: Gegenüberstellung weiche und harte Echtzeit.
Eigene Darstellung, Anlehnung an [28]

## 2.7 Software für die Messungen

Im Folgenden werden die für die Messdurchführungen verwendeten Programme vorgestellt. Mit den Ergebnissen dieser Messtools erfolgt später die Validierung der Echtzeitfähigkeit in einer Kubernetes-Umgebung.

### 2.7.1 Cyclictest

Mit Cyclictest werden mehrere Tasks gestartet. Die Software misst die Differenz zwischen der beabsichtigten Aufwachzeit eines Threads und der Zeit, zu der der Thread tatsächlich startet. Dabei ist ein Thread ein eigenständiger und von anderen Threads unabhängiger Teil eines Prozesses. Für die Bereitstellung von Statistiken über die Latenzzeiten des Systems werden die Messungen wiederholt durchgeführt. Die gemessenen Latenzzeiten ergeben sich in Abhängigkeit des verwendeten Betriebssystems und der zu Grunde liegenden Hardware.

Mit einer Reihe von Parametern können die Messungen konfiguriert werden. Der Aufruf des Cyclictest mit den verwendeten Parametern wird in Kapitel 3.3 thematisiert. [29]

### 2.7.2 Ping

Das Diagnose-Tool ping kann eingesetzt werden, um zu überprüfen, ob ein bestimmter Host in einem IP-Netzwerk erreichbar ist. Für die Messung wird das Internet Control Message Protocol (ICMP) benutzt. Dabei werden sogenannte „Echo-Request“-Pakete an die Zieladresse gesendet. Der Empfänger sendet daraufhin ein Antwortpaket. Wenn keine Antwort zurückgeschickt wird, kann daraus nicht sofort geschlossen werden, dass das Ziel nicht erreichbar ist. Aufgrund einer bestimmten Konfiguration können ICMP-Pakete nämlich auch ignoriert oder verworfen werden. Als Ergebnis von ping werden Zeitspannen zwischen dem Senden und dem Empfangen eines Paketes sowie Informationen zu erhaltenen Paketen und Paketverlusten angezeigt. [30, S. 37]

# 3. Analyse

Für die Validierung wurde ein Kubernetes-Cluster mit *minikube* erstellt. Damit hat man ein einfaches Cluster mit einem Knoten, welches eine gute Wahl für die lokale Entwicklung und das Experimentieren darstellt. Im Rahmen der Untersuchungen lag der Fokus auf Latenzzeiten.

## 3.1 Test-Setup

Um die Echtzeiteigenschaften der Workstation, welche als Testumgebung dient, herzustellen, wurden einige Modifizierungen vorgenommen. Die technischen Daten des zu Beginn nicht echtzeitfähigen Systems können aus der folgenden Tabelle entnommen werden:

| Komponente | Typ |
|---|---|
| CPU | Intel Xeon(R) W-2175, 2,50 Ghz x 14 |
| RAM | 128 GB |
| Betriebssystem | Ubuntu 18.04.5 |
| Speichermedium | SSD 1 TB |

Zunächst wurde mit einem Realtime-Patch ein echtzeitfähiger Kernel mit der Version 5.10.17-rt32 aufgesetzt. Dafür wurde vom Institut ISW der Universität Stuttgart eine ausführbare Datei zur Verfügung gestellt, sodass die Installation trivial durchgeführt werden konnte. Mit dem Aufsetzen eines Echtzeit-Patches wird eine Linux-Umgebung mit einem echtzeitfähigen Betriebssystem hergestellt. Durch diesen Echtzeit-Patch (PREEMPT_RT) wird die Sperrprimitive des Kernels verbessert. Dies führt zur Maximierung der Anzahl der präemptiven Abschnitte. Ein weiterer Vorteil ist, dass keine speziellen Bibliotheken oder APIs benötigt werden [27, S. 4]. Anschließend wurden Einstellungen im BIOS überprüft und angepasst, wie das Ausschalten der Hyperthreading-Funktion und der Energiesparmaßnahmen. Es wurde sichergestellt, dass die Turbo-Boost-Funktion aktiv ist.

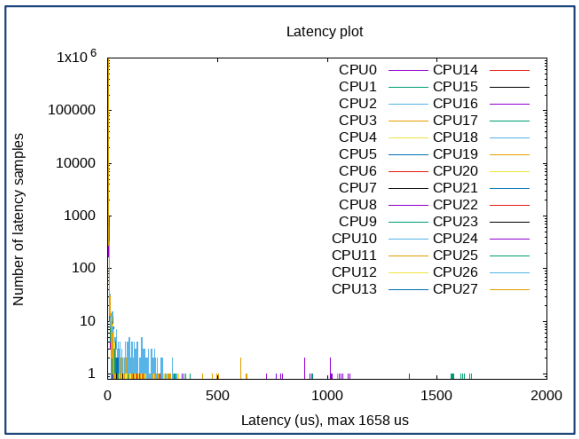
Mit dem Befehl `docker run „Image-name“` und `uname -a` wird im Folgenden außerdem gezeigt, dass der Container den echtzeitfähigen Kernel des Hosts teilt.

```
iiot@iiot:~/Schreibtisch/Dateien-Harun/k8-Test-2$ docker run --name cyclictest-container -it cyclictestimg
root@5e150c15b8ca:/# uname -a
Linux 5e150c15b8ca 5.10.17-rt32 #1 SMP PREEMPT_RT Tue Feb 23 10:07:19 UTC 2021 x86_64 GNU/Linux
root@5e150c15b8ca:/#
```

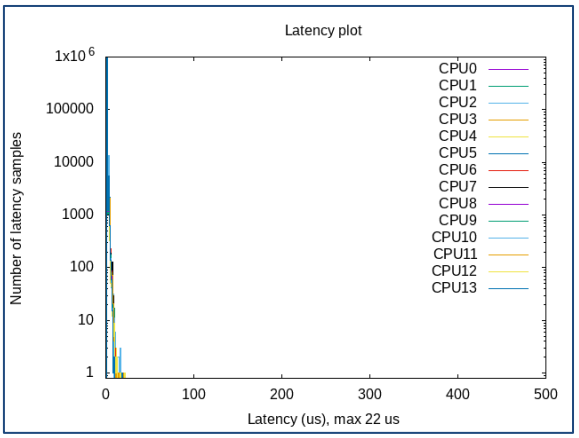
Mit dem folgenden Befehl wurde Cyclictest auf dem Host-System ausgeführt, um die Echtzeitfähigkeit zu überprüfen:

```
sudo cyclictest --smp -l 1000000 -p 95
```

Die folgenden Diagramme zeigen die Ergebnisse des Cyclictests vor und nach der Systemoptimierung. Es ist deutlich erkennbar, dass die Anpassungen zu sehr guten Messwerten geführt haben. Mit dem vorliegenden System wurden die bevorstehenden Tests durchgeführt.




*Host ohne Echtzeiteigenschaften*




*Host mit Echtzeiteigenschaften*

## 3.2 Test-Fälle

Im Rahmen dieser Arbeit fanden die Untersuchungen mittels folgender Test-Fälle statt.

- Latenzzeit-Messungen in einem Kubernetes-Cluster
- Untersuchen von Paketverlusten und der Failover-Zeit

Der Aufbau und die Durchführung der einzelnen Fälle werden im nächsten Abschnitt ausführlich beschrieben. Dabei wird auch auf die verwendeten Parameter der Programme eingegangen.

## 3.3 Messdurchführung Teil 1

Für das Messen der Latenzzeiten wurde in einer Dockerfile die benötigte Software installiert. Im Folgenden wird der Aufbau näher beschrieben.

```
#Auswählen des Basis-Images
FROM debian:buster-slim

#Installation Cyclictest und GNU-Plot
RUN apt-get update && apt-get install -y rt-tests \
    && apt-get update && apt-get install -y gnuplot \
    && rm -rf /var/lib/apt/lists/*

#Aufruf von Cyclictest beim Starten des Containers
CMD ["cyclictest"]
```

Die Dockerfile definiert zu Beginn das Basisimage. Anschließend wird Cyclictest mit Gnuplot installiert. Gnuplot ist ein Programm, welches Messdaten graphisch darstellt. Der letzte Befehl dient zum Aufruf des Programms beim Starten des Containers.

Mit dem Befehl `docker build -t „Image-Name“ .` wird das Docker-Image mit allen enthaltenen Abhängigkeiten erzeugt.

Mit `minikube image load „Image-Name“` muss das erzeugte Image zunächst in das Kubernetes-Cluster geladen werden.

Der verwendete Pod wurde mit dem folgenden Manifest generiert. Hinter `image` wird der Name des gebauten Images angegeben. In der Datei wird auch die Ressourcenzuweisung an den Pod festgelegt. Nach `command` folgt die Programmausführung mit seinen Parametern. Da in dieser Arbeit mit einem Bash-Script gearbeitet wurde, welches die gemessenen Werte graphisch darstellt, wurden `command` und `args` hier auskommentiert und nicht verwendet. Stattdessen wurde im nächsten Schritt das Script in den Pod geladen und dort ausgeführt.

```
apiVersion: v1
kind: Pod
metadata:
  name: cyclictest-pod
spec:
  containers:
    - name: cyclictest
      image: cyclictestimg:latest
      imagePullPolicy: Never
      resources:
        requests:
          cpu: 1000m
          memory: 1G
        limits:
          cpu: 2000m
          memory: 2G
      # command: ["cyclictest"]
      # args:
      securityContext:
        capabilities:
          add:
            - SYS_NICE
```

Schließlich wurde der Pod mit dem Kubernetes-Befehl `kubectl apply -f „pod.yaml“` gestartet.

```
pod/cyclictest-pod created
iiot@iiot:~/Schreibtisch/k8-Test-2$ kubectl get pods
NAME                                READY   STATUS    RESTARTS        AGE
cyclictest-pod                      1/1     Running   0               6s
```

Im Weiteren wurde das vorhin erwähnte Script in den Pod geladen. Dies geschieht mit dem Befehl `kubectl cp /path/to/file pod-name:/path/to/file`.

Um in den Pod zu navigieren, wurde der Befehl `kubectl exec -it „pod-name“ -- /bin/bash` verwendet. Hier können Befehle direkt innerhalb des Pods ausgeführt werden. Mit dem Linux-Befehl `ls` wurde noch überprüft, ob das Script im Pod abgelegt ist.

```
root@cyclictest-pod:/# ls
bin  boot  dev  etc  home  lib  lib64  media  mklatencyplotFinal.bash  mnt  opt  proc
 root  run  sbin  srv  sys  tmp  usr  var
```

Schließlich erfolgte mit `bash „script-name“` die Ausführung des Scripts mit dem Aufruf von Cyclictest:

```
#Aufruf von Cyclictest im Script mit 10⁶ Messungen
# cyclictest --smp -l1000000 -p95 -i200 -h1000 -q
```

**Latenzzeitmessungen im Pod unter Normalzustand**

Im ersten Testfall wurden Latenzzeitmessungen im Pod ohne Auslastung des Systems durchgeführt. Mit dem Parameter `-l` wurde die Anzahl der Messungen, in diesem Fall eine Million, angegeben. Die Messungen erfolgen in Abhängigkeit des Intervalls `-i`. Mit dem Parameter `--smp` wird die Messung auf allen CPU-Kernen mit der gleichen Priorität ausgeführt, welche mit `-p` festgelegt wird. Für die maximal darstellbare Latenzzeit im Diagramm wird der Parameter `-h` benötigt. Mit `-q` wird noch die Anweisung gegeben, dass keine Ausgaben während der Messungen stattfinden sollen.

**Latenzzeitmessungen im Pod mit 100% CPU-Last**

Die gleichen Messungen wie im ersten Fall wurden mit künstlicher Belastung fortgesetzt. Dazu war es notwendig, ein Tool zu installieren, welche die Systemkerne voll auslastet. Hier wurde mit dem Programm stress gearbeitet. Dabei kamen drei Parameter zum Einsatz. Mit `-c` wird die Anzahl der CPU-Kerne angegeben. Für die Belastung des Arbeitsspeichers ist der Parameter `-m` notwendig. Die Erhöhung der Lese- und Schreibvorgänge auf der Festplatte wird mit `-d` eingestellt. In diesem Fall wurde zunächst nur `-c` und `-m` benutzt.

```
#Programm für die Systemauslastung
stress -c 14 -m 60
```

**Latenzzeitmessungen im Pod mit I/O-Last**

Des Weiteren wurden die Latenzzeiten nur mit Festplattenbelastung untersucht. Hierzu wurde während der Messungen stress mit dem Parameter `-d` aufgerufen.

```
#Belastung der Festplatte
stress -d 30
```

**Latenzzeitmessungen im Pod mit 100% CPU-Last und I/O-Last**

Abschließend wurden in einem weiteren Testfall die Messungen mit CPU-Auslastung und gleichzeitiger Last auf der Festplatte gestartet.

```
#Messungen mit CPU-Last und I/O-Last
stress -c 14 -m 60 -d 30
```

Cyclictest wurde bisher mit $10^6$ Messungen durchgeführt. Diese Messungen haben etwa 10 Minuten gedauert. Für Tests über einen deutlich längeren Zeitraum wurde der Parameter `-l` angepasst.

```
#Cyclictest mit 10^8 Messungen
# cyclictest --smp -l100000000 -p95 -i200 -h1000 -q
```

Mit dieser Einstellung wurden die vorherigen Messungen mit den vier unterschiedlichen Szenarien wiederholt. Zuvor wird im Folgenden eine Anpassung im Pod-Manifest durchgeführt. Die Mindestanforderung und die Obergrenze von CPU-Kernen sowie des Arbeitsspeichers werden erhöht.

```
apiVersion: v1
kind: Pod
metadata:
  name: cyclictest-pod-2
spec:
  containers:
    - name: cyclictest
      image: cyclictestimg:latest
      imagePullPolicy: Never
      resources:
        requests:
          cpu: 3000m
          memory: 3G
        limits:
          cpu: 6000m
          memory: 6G
      # command: ["cyclictest"]
      # args:
      securityContext:
        capabilities:
          add:
            - SYS_NICE
```

Das Script wurde nun wieder in den neuen Pod kopiert und dort ausgeführt. Kürzere Tests mit $10^6$ Messungen und unterschiedlichen Systemzuständen sowie die Tests mit $10^8$ Messungen wurden erneut durchgeführt. Die Darstellung und Auswertung aller hier erhaltenen Resultate erfolgt in Kapitel 4.

## 3.4 Messdurchführung Teil 2

Im zweiten Testfall wurde mit dem zuvor beschriebenen Diagnosetool ping gearbeitet. Hierbei lag der Fokus zum einen auf der Untersuchung von Latenzen bei neu gestarteten Pods durch ein ReplicaSet, nachdem diese beendet wurden und zum anderen auf den Paketverlusten. Im Folgenden wird der Aufbau und die Vorgehensweise dieser Messdurchführung beschrieben.

Zunächst wurde das bereits existierende Docker-Image mit der Installation von ping erweitert.

```
#Auswählen des Basis-Images
FROM debian:buster-slim

#Installation Cyclictest, GNU-Plot und ping
RUN apt-get update && apt-get install -y rt-tests \
    && apt-get update && apt-get install -y gnuplot \
    && apt-get update && apt-get install -y iputils-ping \
    && rm -rf /var/lib/apt/lists/*

#Aufruf von Cyclictest beim Starten des Containers
CMD ["cyclictest"]
```

Anschließend wurde ein Manifest für ein ReplicaSet erstellt, in welchem das Image mit der Erweiterung um ping verwendet wurde. Das ReplicaSet sorgt für den automatischen Neustart von beendeten Pods. Dazu wird in der Konfigurationsdatei an der Stelle `replicas` eine gewünschte Anzahl definiert. Kubernetes stellt somit sicher, dass immer die geforderte Menge an Pods laufen.

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: cyclictest-pod-ping
spec:
  replicas: 2
  selector:
    matchLabels:
      app: cyclictest-pod-ping
  template:
    metadata:
      name: cyclictests-pod-ping
      labels:
        app: cyclictest-pod-ping
    spec:
      containers:
        - name: cyclictest-ping
          image: cyclictestimg:latest
          imagePullPolicy: Never
            # command: ["cyclictest"]
            # args:
          securityContext:
            capabilities:
              add:
                - SYS_NICE
```

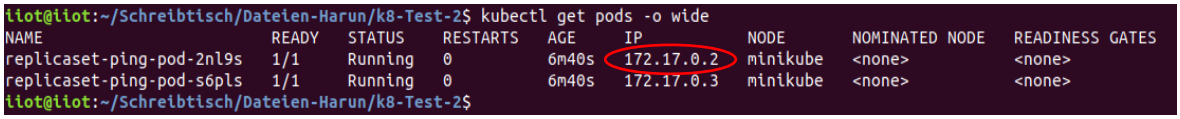
Mit dem bereits bekannten Befehl `kubectl create -f „replicaset.yaml“` wird hierbei das ReplicaSet mit der geforderten Anzahl an Pods erstellt.

```
iiot@iiot:~/Schreibtisch/Dateien-Harun/k8-Test-2$ kubectl create -f ping-replicaset.yaml
replicaset.apps/replicaset-ping-pod created
iiot@iiot:~/Schreibtisch/Dateien-Harun/k8-Test-2$
```

Über `kubectl get rs` lässt sich das erstellte ReplicaSet mit Informationen zu geforderten und den aktuell vorhandenen Pods sowie dem aktuellen Status anzeigen.

```
iiot@iiot:~/Schreibtisch/Dateien-Harun/k8-Test-2$ kubectl get rs
NAME                 DESIRED   CURRENT   READY   AGE
replicaset-ping-pod  2         2         2       3m5s
iiot@iiot:~/Schreibtisch/Dateien-Harun/k8-Test-2$
```

Schließlich werden noch die aktiven Pods ausgegeben. Diese Ausgabe erfolgt erneut mit dem bekannten Befehl `kubectl get pods -o wide`, wobei mit dem Zusatz `-o wide` weitere ausführliche Informationen zum jeweiligen Objekt angezeigt werden können.




Innerhalb eines Pods soll die Diagnose-Software gestartet werden und „Echo-Request"-Pakete an den anderen Pod senden. Dafür wurde zunächst mit dem Befehl `kubectl exec -it „pod-name" -- /bin/bash` in den Pod navigiert, von dem aus die Pakete gesendet werden sollten. In diesem Fall ist es der Pod mit der IP-Adresse 172.17.0.2. Für die Ausführung wurde die IP-Adresse des Ziel-Pods angegeben, welche in der obigen Abbildung umkreist ist. Außerdem wird mit dem Parameter `-i` das Zeitintervall zwischen den gesendeten Paketen in Sekunden definiert. Im Weiteren wird von einer Zykluszeit von 8 ms und einer Failover-Zeit von 24 ms ausgegangen. Das bedeutet, dass alle 8 ms ein Paket von der SPS zur Steuerungsanlage und wieder zurück gesendet wird. Die Failover-Zeit bezeichnet hier die Dauer, bis ein Pod nach einem Ausfall neu gestartet und wieder einsatzfähig ist. Nachdem ein Paket gesendet wurde, startet der WatchDog Timer (WDT). Sobald die festgelegte Zeit im WTD überschritten wurde, befinden sich die Geräte in einem Fehlerzustand und die Verbindung wird unterbrochen. Wenn ein Paket erneut ankommt, wird der WDT zurückgesetzt. [31, S. 13]

Mit der vorausgehenden Beschreibung erfolgte an dieser Stelle die Ausführung von ping.

```
#Aufruf von ping
# ping -i 0.008 172.17.0.2
```

Während in einem Kommandozeilen-Terminal das Tool gestartet wurde, erfolgte in einem weiteren Terminal das Beenden der Pods mit dem Kubernetes-Befehl `kubectl delete pods/"pod-name"`. Das ReplicaSet hat für den Neustart des beendeten Pods gesorgt. Somit konnten Informationen zur Dauer der Unterbrechung und den Paketverlusten gewonnen werden. Damit waren die Messdurchführungen abgeschlossen.

# 4. Ergebnisse und Auswertung

In diesem Kapitel erfolgt die Auswertung der erhaltenen Latenzzeiten mit dem Messtool Cyclictest. Daraufhin werden die Ergebnisse zu Paketverlusten und der Failover-Zeit, die mithilfe des Diagnosetools ping ermittelt wurden, dargestellt. Im Anschluss erfolgt eine Diskussion zu den Messwerten.

## 4.1 Testfall 1: Latenzzeiten im Pod

### Pod mit $10^6$ Messungen ohne Last

Im ersten Testfall wurden Latenzen innerhalb eines Pods und ohne Systemauslastung gemessen. Alle Messungen wurden insgesamt 10-mal wiederholt. Dabei wurden Messwerte zwischen 52 µs bis maximal 73 µs erreicht. Im Folgenden werden je Szenario 4 Resultate mit dem minimalen bis zum maximal gemessenen Wert abgebildet.

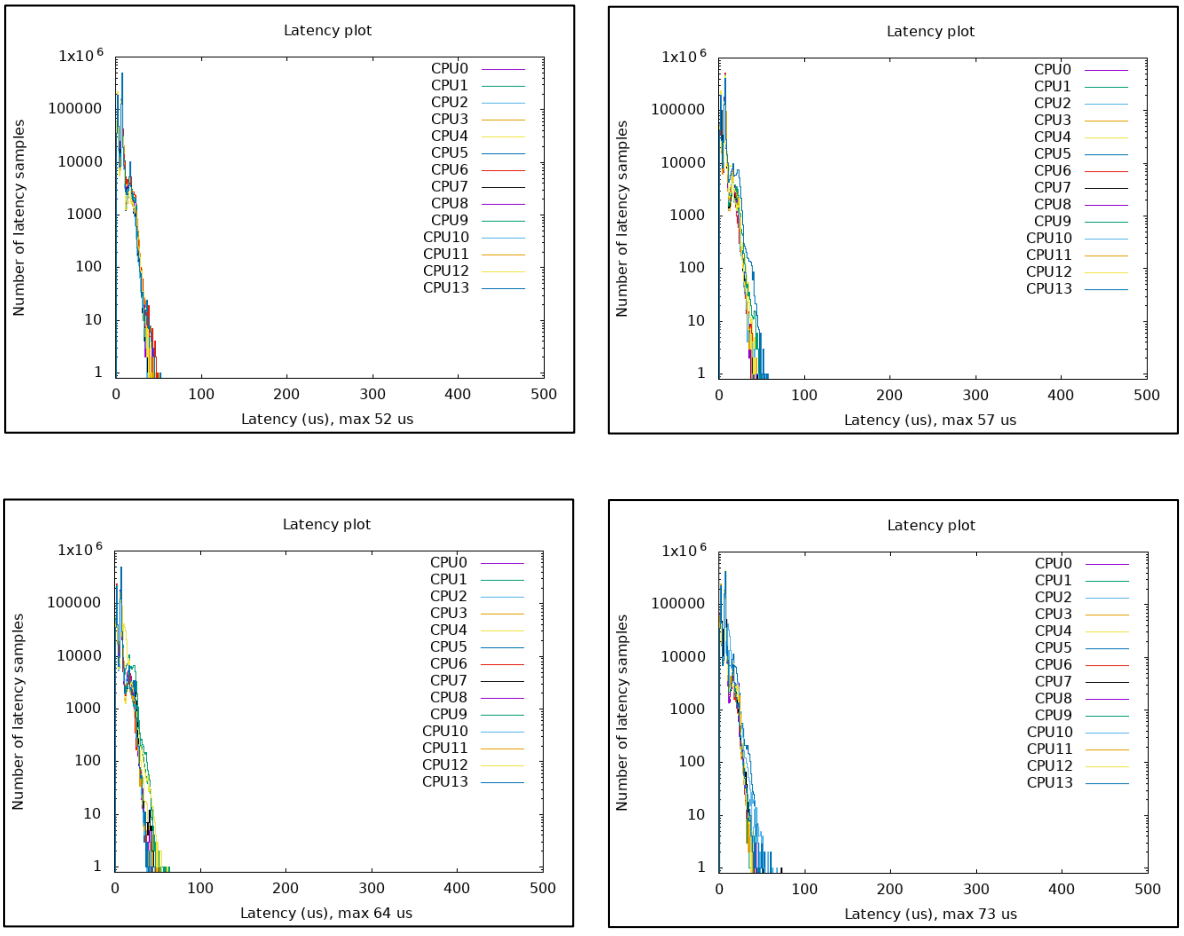



Abbildung 4.1: Latenzzeiten im Pod ohne Systemlast

## Pod mit $10^6$ Messungen und Last

Die Messungen fanden während der Auslastung der CPU-Kerne statt, wobei Ergebnisse zwischen 29 µs und 43 µs erreicht wurden. Verglichen mit dem maximalen Wert aus den Messungen ohne Systemlast hat sich die Latenzzeit unter dieser Bedingung um 30 µs verringert. Allein der niedrigste Wert mit 29 µs fällt im Vergleich zu 52 µs aus der vorherigen Messung um über 20 µs geringer aus. Somit war durch die CPU-Auslastung eine Verbesserung der Messwerte zu beobachten.

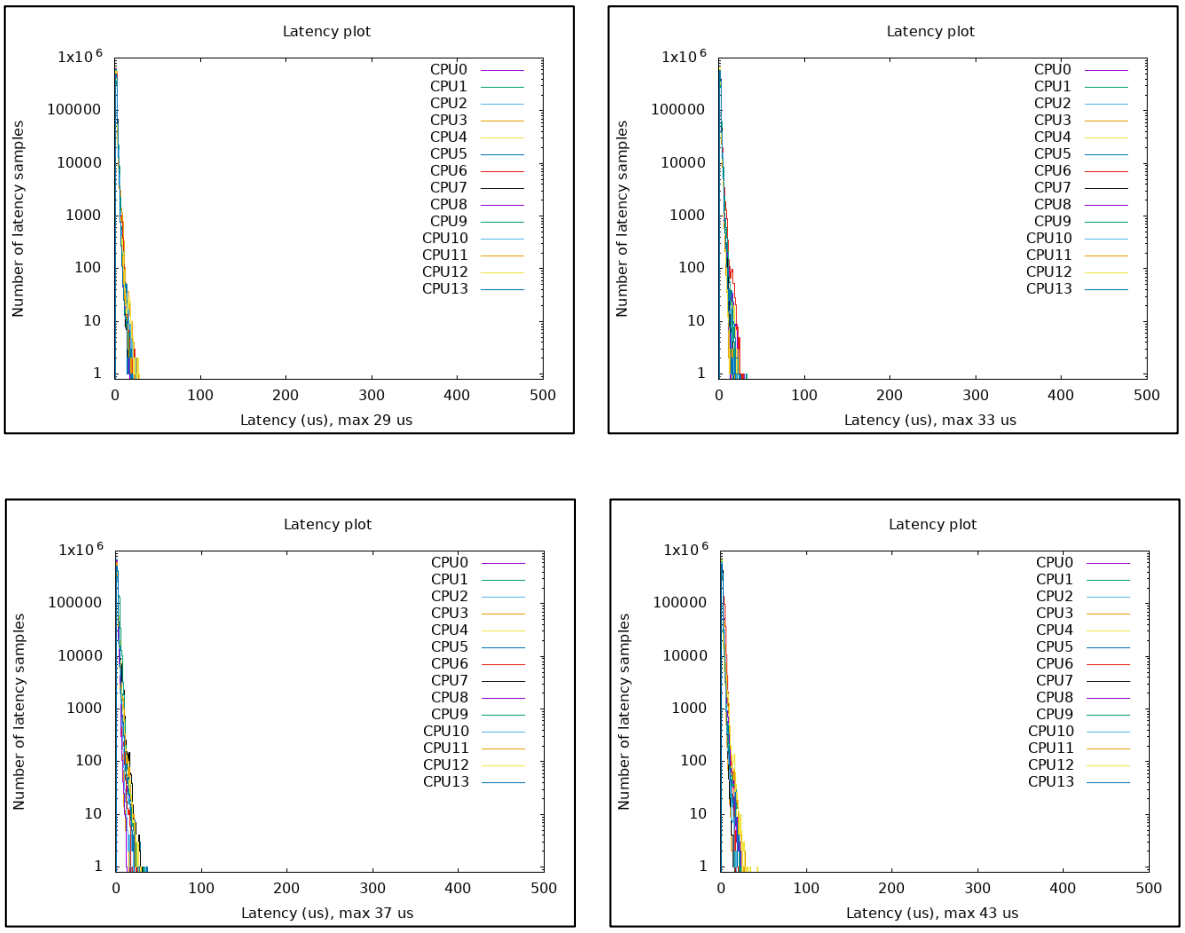



Abbildung 4.2: Latenzzeiten im Pod mit 100 % CPU-Last

## Pod mit $10^6$ Messungen und I/O-Last

In diesem Fall wurden während der Messungen nur die Lese- und Schreibvorgänge auf der Festplatte erhöht. Gegenüber der CPU-Auslastung waren die Messwerte hier zwar ähnlich, jedoch wurden mit 36 µs bis 48 µs minimal schlechtere Ergebnisse erzielt. Lediglich mit der Auslastung der Festplatte fielen die Werte aber auch bei dieser Messdurchführung ebenfalls niedriger aus als im Zustand ohne Systemlast.

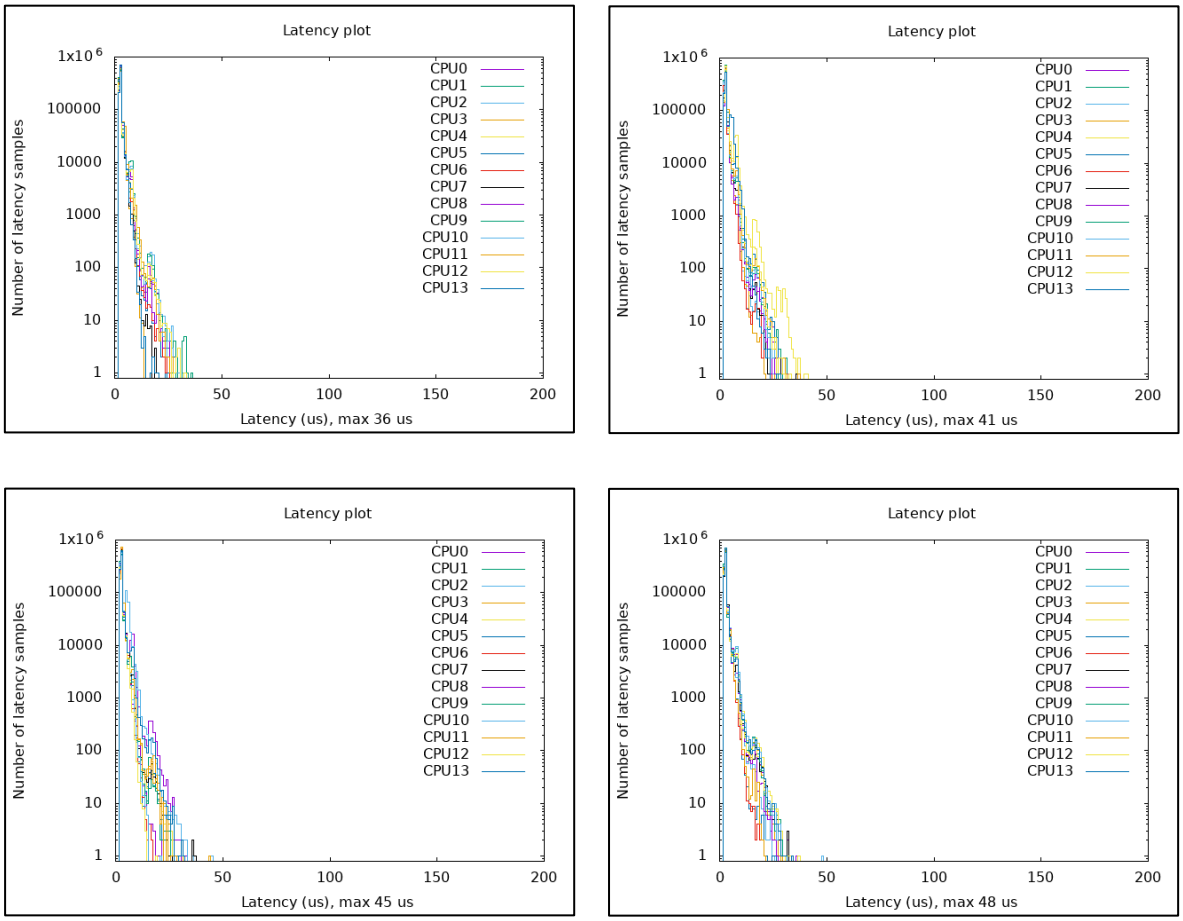



Abbildung 4.3: Latenzzeiten im Pod mit I/O-Last

## Pod mit $10^6$ Messungen und CPU- und I/O-Last

Im letzten Durchgang mit $10^6$ Messpunkten wurde der Fall untersucht, wie sich die Werte unter CPU-Last und gleichzeitiger Erhöhung der Lese- und Schreibvorgänge auf der Festplatte verhalten. Mit 101 µs bis 133 µs verschlechterten sich die Latenzzeiten in diesem Zustand im Vergleich zu den vorherigen Szenarien um ein Vielfaches. Hier fällt auf, dass neben der Auslastung der Prozessorkerne die zusätzliche Festplattenauslastung zu höheren Latenzzeiten führt.

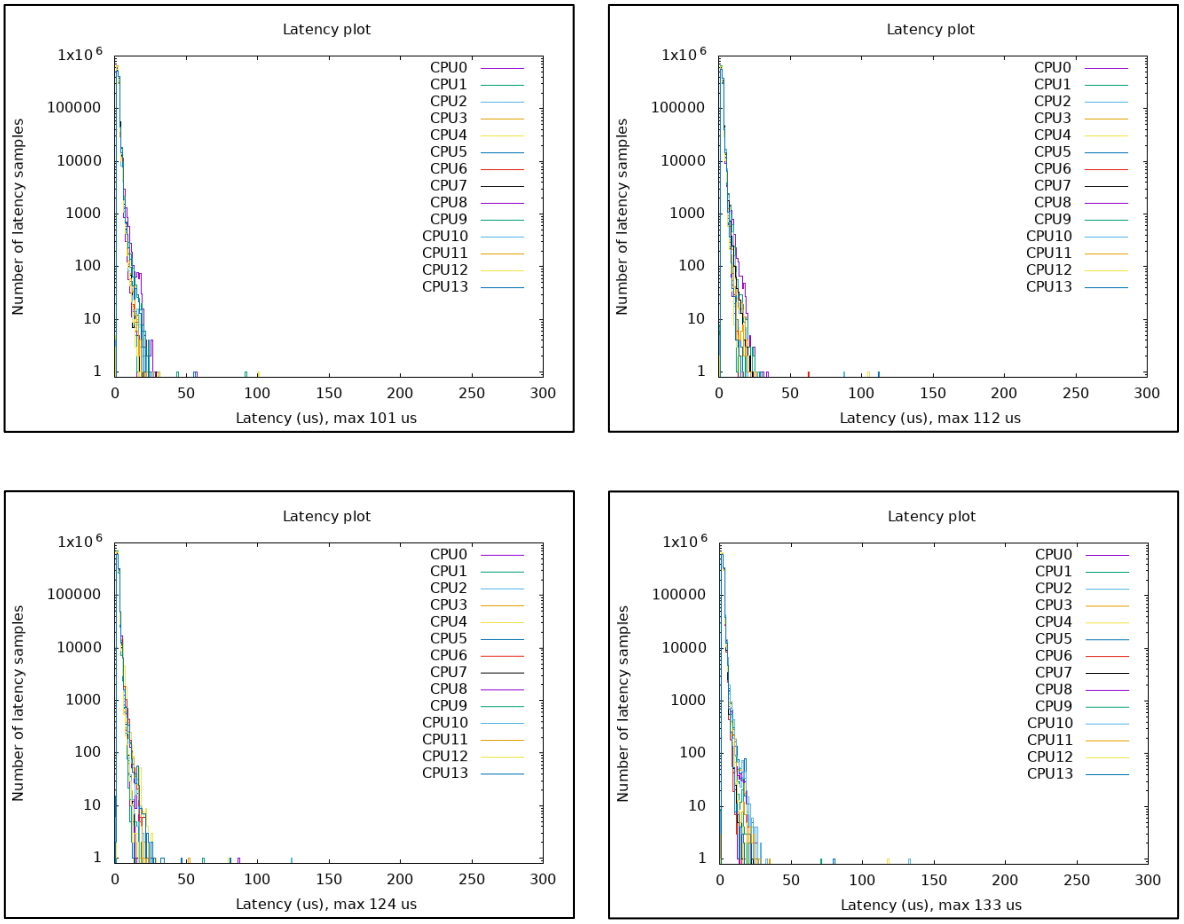
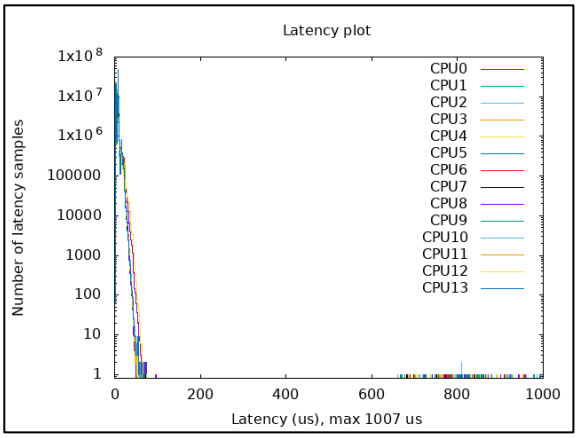


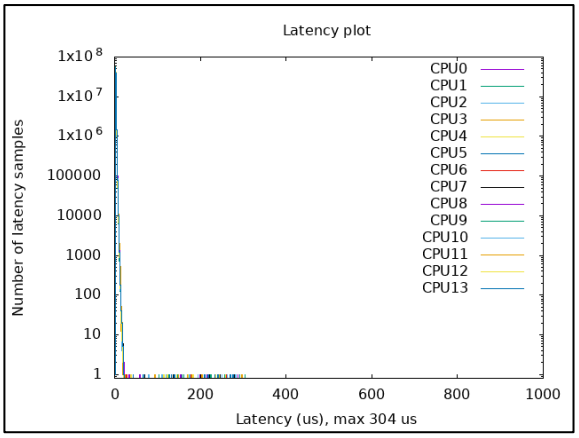

Abbildung 4.4: Latenzzeiten im Pod mit 100 % CPU-Last und I/O-Last

## Pod mit $10^8$ Messungen

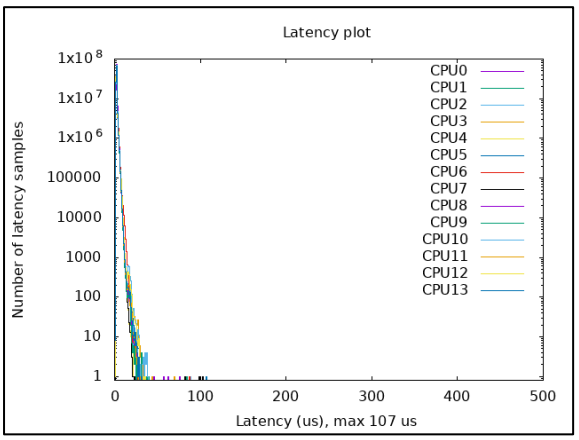
Im Folgenden sind die Ergebnisse von Tests über einen längeren Zeitraum abgebildet. Die Anzahl wurde auf $10^8$ Messungen erhöht, welche ungefähr 8 Stunden Zeit in Anspruch genommen haben. Je Szenario wurde die Messung einmal durchgeführt. Besonders auffallend war hierbei der Wert im Zustand ohne Systemlast. Mit einer Latenzzeit von 1007 µs wurde erstmalig die Grenze von 1 ms überschritten. Das Ergebnis unter CPU-Last ist hier mit 304 µs auch deutlich höher als bei den kürzeren Messdurchführungen. Dennoch fällt auch hier der Wert um mehr als das Dreifache niedriger aus als die Ergebnisse aus den Tests ohne Systemlast. Die niedrigsten Werte wurden in den letzten beiden Szenarien erreicht. Ein Grund für die insgesamt höheren Werte ist die Zeitspanne der Messungen, da mehr Ausreißer abgefangen werden, wenn über einen längeren Zeitraum gemessen wird.



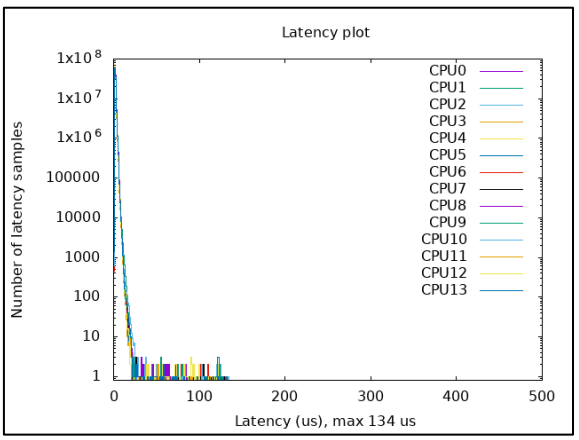

Ohne Systemlast




Mit 100 % CPU-Last




Mit I/O-Last




Mit 100 % CPU- und I/O-Last

Abbildung 4.5: Latenzzeiten im Pod mit $10^8$ Messungen

## Erweiterter Pod mit $10^6$ Messungen ohne Last

Alle Messungen wurden schließlich im angepassten Pod mit erhöhter Ressourcenzuweisung wiederholt. Im Folgenden sind drei Ergebnisse von zehn durchgeführten Messungen abgebildet. Es wurden ähnliche Werte wie im vorherigen Pod erreicht, wobei der maximale Wert mit 63 µs minimal geringer ausfällt als vor der Anpassung. Auch hier wurde insgesamt 10-Mal gemessen. Die gelieferten Ergebnisse zeigen in diesem Fall keine großen Unterschiede.

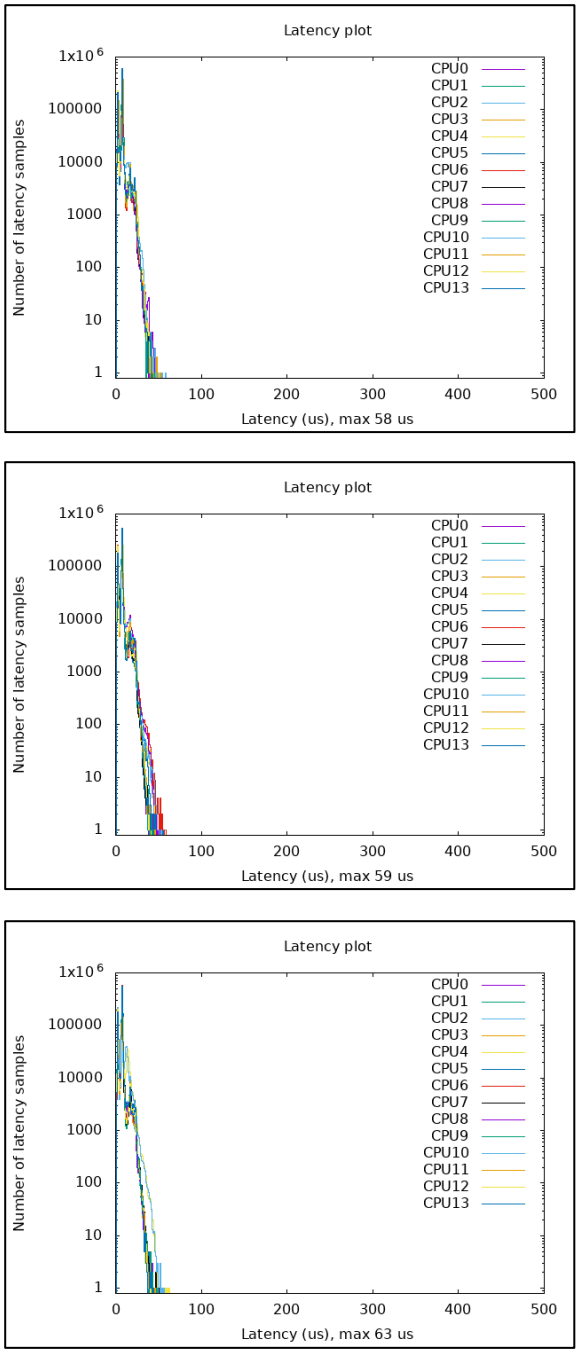



Abbildung 4.6: Latenzzeiten im erweiterten Pod ohne Systemlast

## Erweiterter Pod mit $10^6$ Messungen und Last

Mit 28 µs bis maximal 35 µs sind die Werte im modifizierten Pod und unter Auslastung der CPU-Kerne verglichen mit den Werten aus dem ersten Pod zwar ähnlich, dennoch konnten auch in diesem Fall minimal bessere Ergebnisse beobachtet werden. Außerdem führte die CPU-Auslastung mit einer maximalen Latenzzeit von 35 µs im Gegensatz zu 63 µs aus den zuvor durchgeführten Messungen erneut zu niedrigeren Latenzzeiten.

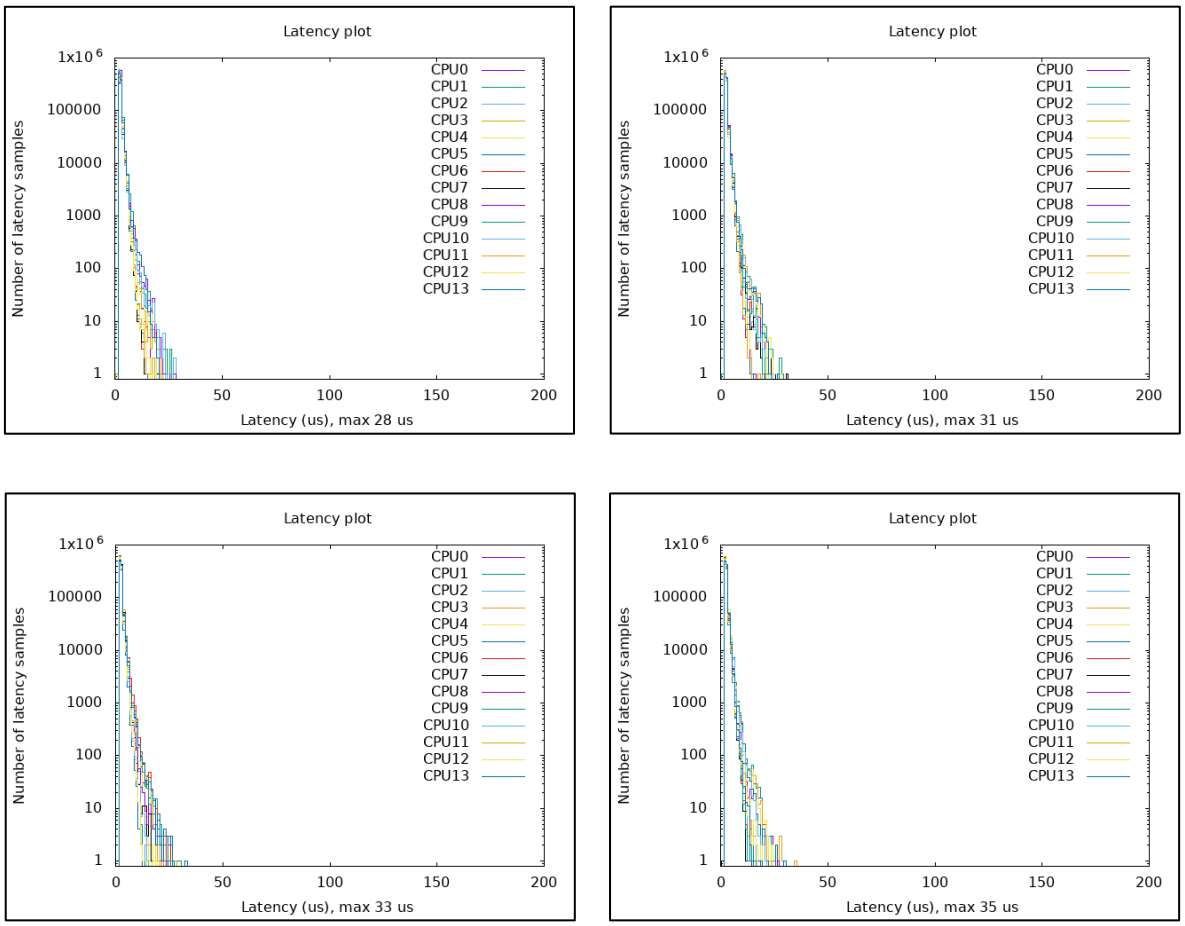



Abbildung 4.7: Latenzzeiten im erweiterten Pod mit 100 % CPU-Last

## Erweiterter Pod mit $10^6$ Messungen und I/O-Last

Während der Zugabe von Last auf die Festplatte konnten Latenzzeiten zwischen 33 µs und maximal 37 µs erreicht werden. Auch hier waren zwar keine großen Unterschiede zu beobachten. Es wurde jedoch ersichtlich, dass die Anpassung der Ressourcenzuweisung zu einer leichten Verbesserung beigetragen hat und die Latenzzeit von 48 µs aus der zuvor durchgeführten Messung mit I/O-Last auf 37 µs gesunken ist.

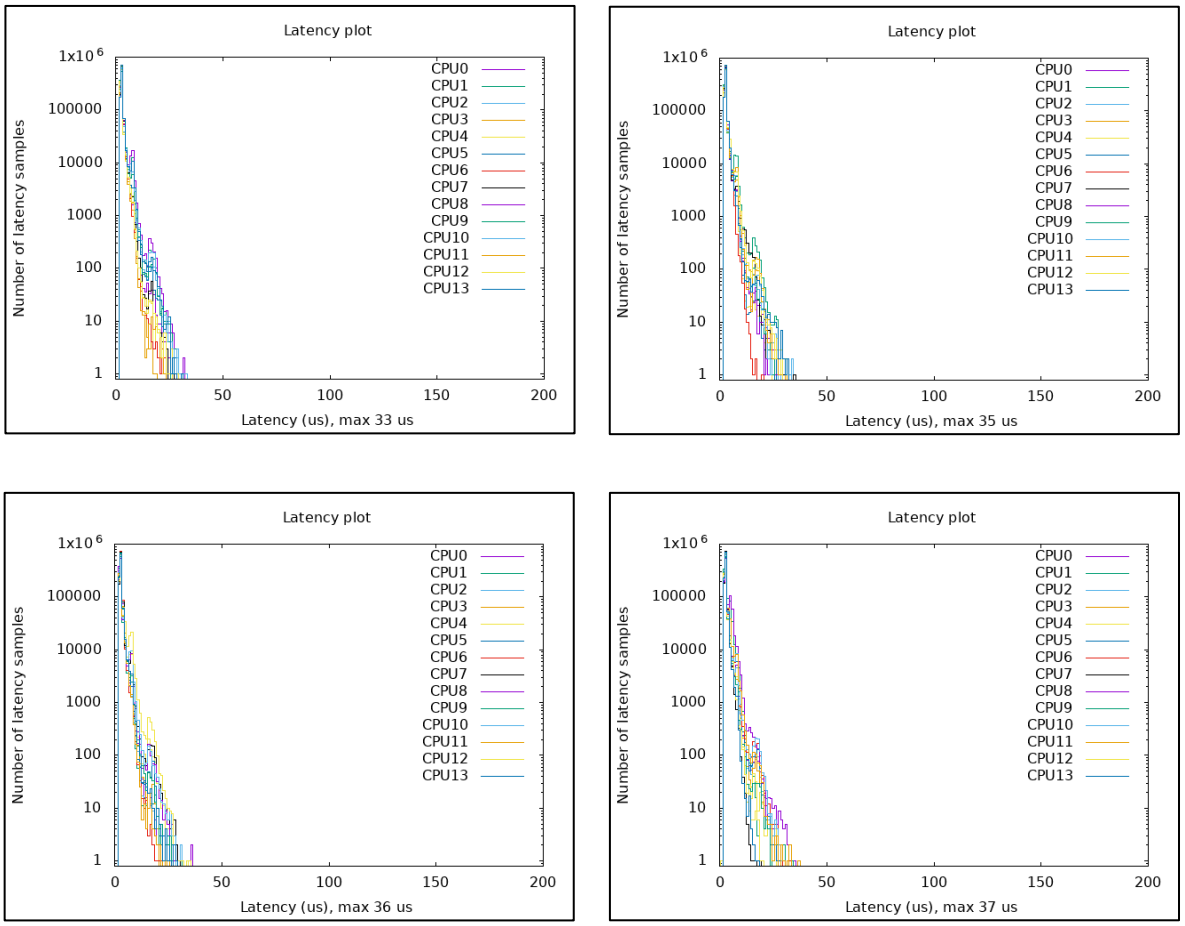



Abbildung 4.8: Latenzzeiten im erweiterten Pod mit I/O-Last

## Erweiterter Pod mit $10^6$ Messungen und CPU- und I/O-Last

Die gleichzeitige Belastung der CPU und der Festplatte lieferte Messwerte zwischen 105 µs und 131 µs. Diese Ergebnisse waren nahezu identisch zu den Messungen im Pod ohne die Erhöhung der zur Verfügung stehenden Ressourcen und konnten somit keine großen Veränderungen aufzeigen. Neben dieser Erkenntnis kann gesagt werden, dass sich auch in diesem Fall die Werte verglichen mit den Ergebnissen aus den unterschiedlichen Systemzuständen verschlechtert haben.

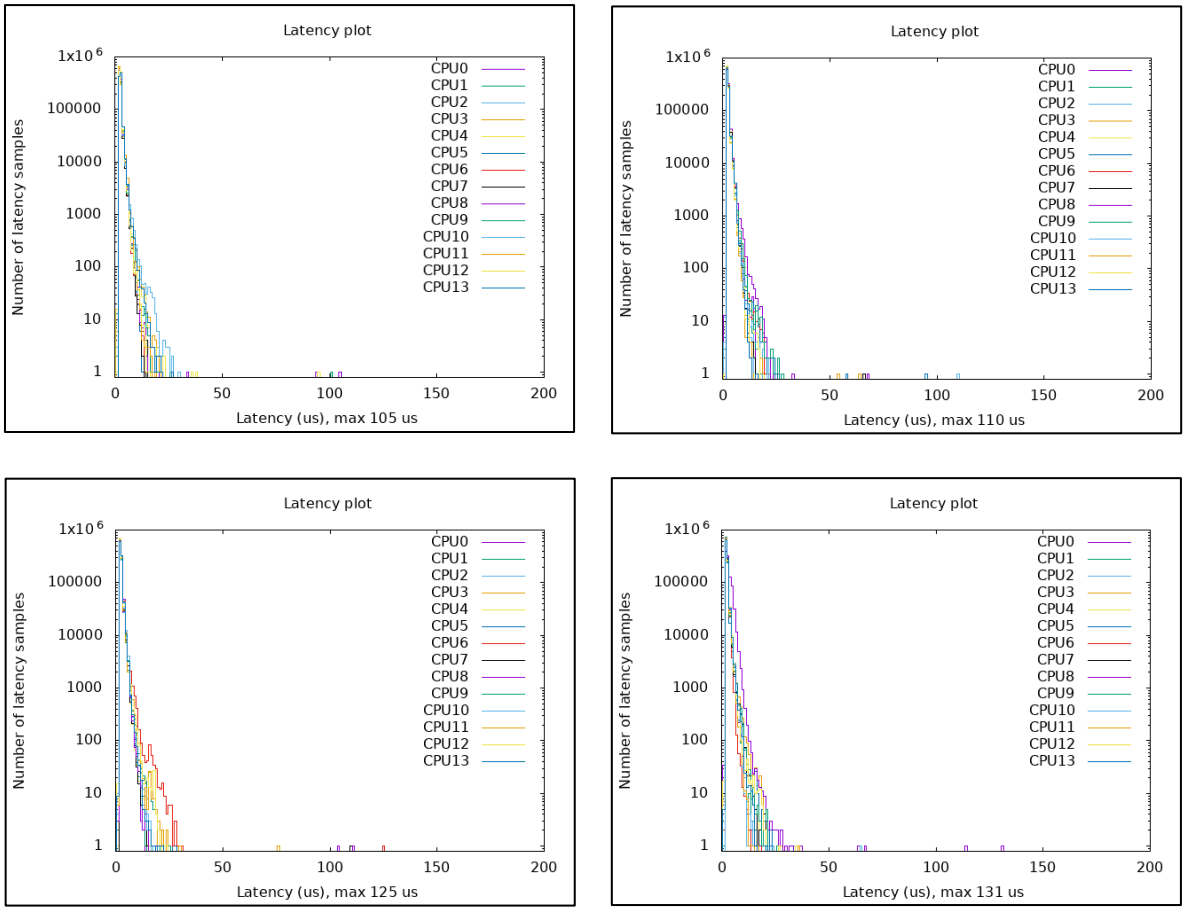
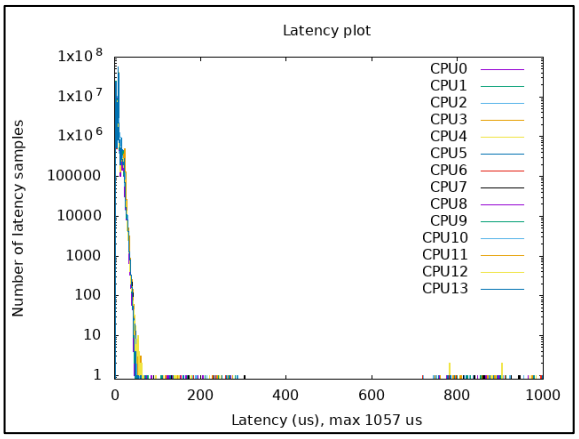


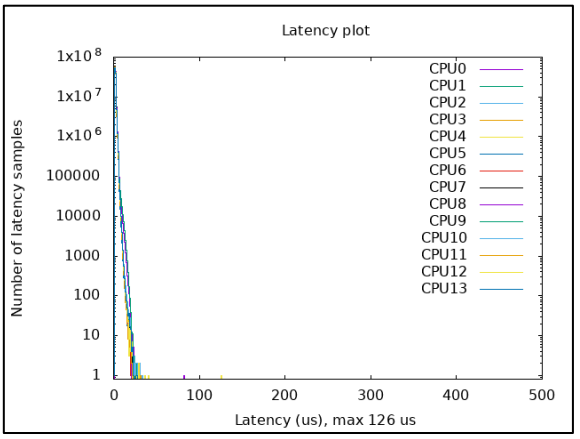

Abbildung 4.9: Latenzzeiten im erweiterten Pod mit 100 % CPU-Last und I/O-Last

## Erweiterter Pod mit $10^8$ Messungen

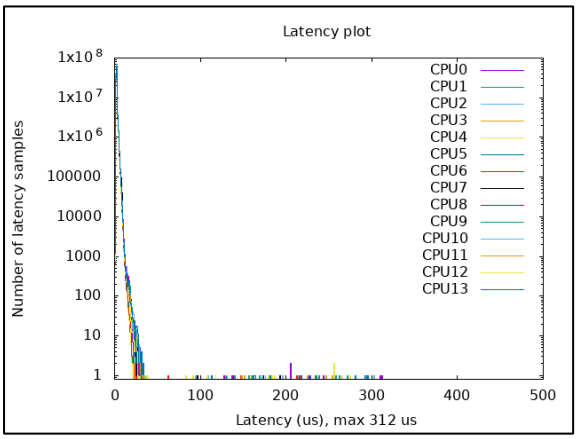
In den letzten vier Testszenarien über einen längeren Zeitraum gab es im ersten Fall ohne Systemauslastung keine großen Veränderungen. Eine Latenzzeit von 1057 µs lag nahe dem Ergebnis aus dem ersten Pod. Im Zustand mit der CPU-Auslastung war aber ein deutlich größerer Unterschied zu erkennen. Mit 126 µs reduzierte sich hier der Wert um knapp 60 % zum Ergebnis aus dem ersten Pod. Mit der Erhöhung der Lese- und Schreibvorgänge auf der Festplatte war wiederum das Gegenteil der Fall. Hier hat sich das Resultat mit 312 µs gegenüber zum alten Wert um knapp das Dreifache verschlechtert. Schließlich lieferte die Messung unter CPU-Last und I/O-Last mit 136 µs beinahe das gleiche Ergebnis wie aus dem Pod ohne Anpassungen.



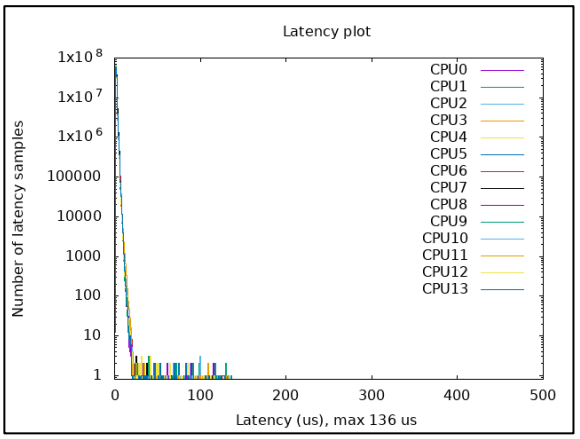

Ohne 100 % CPU-Last




Mit 100 % CPU-Last




Mit I/O-Last




Mit 100 % CPU- und I/O-Last

Abbildung 4.10: Latenzzeiten im erweiterten Pod mit $10^8$ Messungen

## 4.2 Testfall 2: Paketverluste und Failover-Zeit

Untersuchungen hinsichtlich Paketverluste und der Failover-Zeit wurden zunächst mit der Standard-Paketgröße von ping durchgeführt. Diese beträgt bei Linux-Systemen 64 Bytes. Im Folgenden ist ein Ausschnitt von 460 gesendeten Paketen dargestellt. Zwischen der laufenden Nummer 259 und 310 wurde der Pod beendet und neu gestartet. Mit dem Ausfall sind 50 Pakete verloren gegangen. Diese Anzahl der Paketverluste multipliziert mit der Zykluszeit von 8 ms ergibt ein Ergebnis von 400 ms, welches beinahe mit dem Faktor 17 über der geforderten Grenze von 12 ms liegt.

Die Tabelle und auch die linke Achse in der Grafik zeigt die Paketumlaufzeit in Millisekunden (Round Trip Time). Diese Information beschreibt die Zeitdauer, die ein Datenpaket für die gesamte Strecke von der Quelle bis zum Ziel und wieder zurück braucht. Unmittelbar nach dem der Pod wieder verfügbar ist, beträgt die maximale Umlaufzeit 0,130 ms. Anzumerken ist an dieser Stelle, dass es zu unterschiedlichen Umlaufzeiten beim Senden und Empfangen kommen kann. Auch die Bearbeitungszeit am Ziel kann Verzögerungen bewirken und somit die Genauigkeit beeinflussen. [32]

| Min. ms | Avg. ms | Max. ms |
|---|---|---|
| 0,005 | 0,007 | 0,130 |

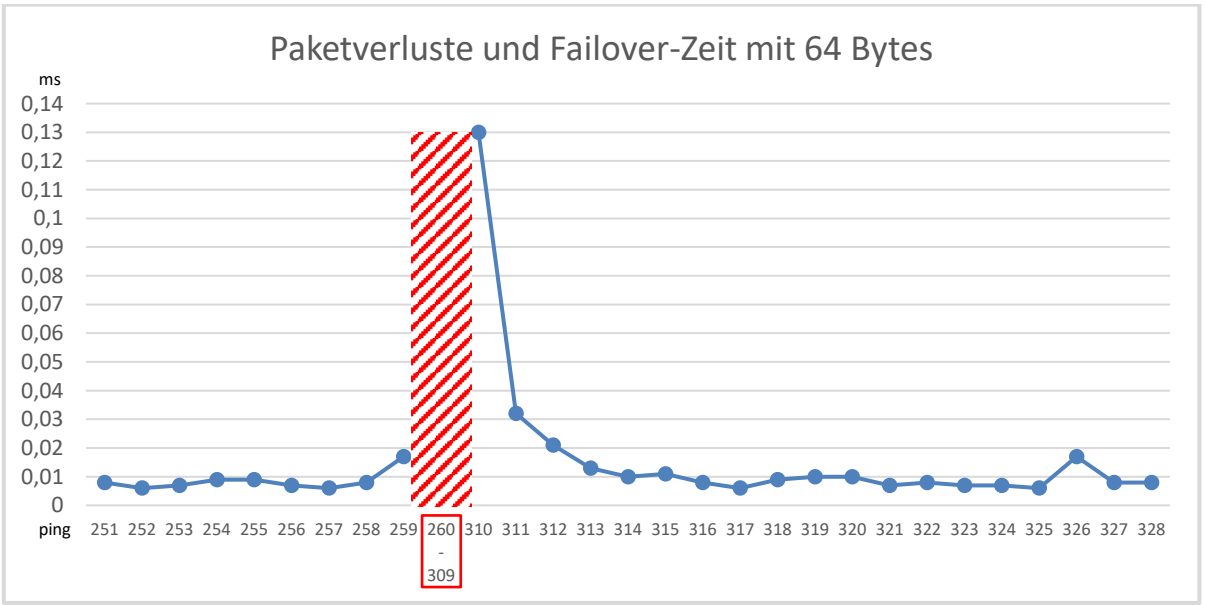



Abbildung 4.11: Paketverluste und Failover-Zeit

In einem weiteren Durchgang wurde die Paketgröße auf 1000 Bytes erhöht und der Test erneut durchgeführt. Dabei sind im Moment des Ausfalls 77 von 556 gesendeten Paketen nicht angekommen. Mit der vorherigen Rechnung ergibt sich daraus eine Failover-Zeit von 616 ms, mit dieser die gesetzte Zeitgrenze um 592 ms überschritten wird.

| Min. ms | Avg. ms | Max. ms |
|---|---|---|
| 0,007 | 0,008 | 0,161 |

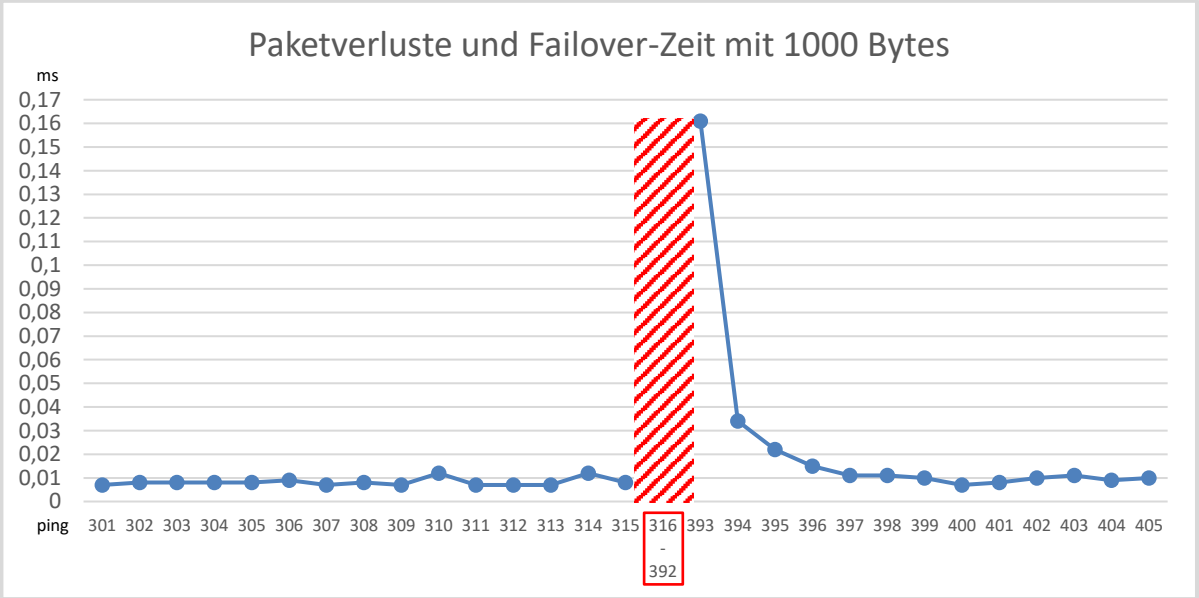



Abbildung 4.12: Paketverluste und Failover-Zeit mit 1000 Bytes

Im Rahmen der Untersuchung wurde die Paketgröße ein letztes Mal erhöht. Mit einer Datengröße von 5000 Bytes führte die Dauer der Unterbrechung zum Verlust von 46 von 488 gesendeten Paketen und zu einer Ausfallzeit von 368 ms. Auch in diesem Fall lag der Wert um 344 ms deutlich über den geforderten 12 ms. Außerdem hatte die Erhöhung der Paketgröße wie erwartet auch die steigende Paketumlaufzeit zur Folge. Prozentuale Angaben zu den Paketverlusten hängen von der Anzahl der insgesamt gesendeten Paketen ab, weshalb diese hier weggelassen wurden.

| Min. ms | Avg. ms | Max. ms |
|---|---|---|
| 0,015 | 0,019 | 0,183 |

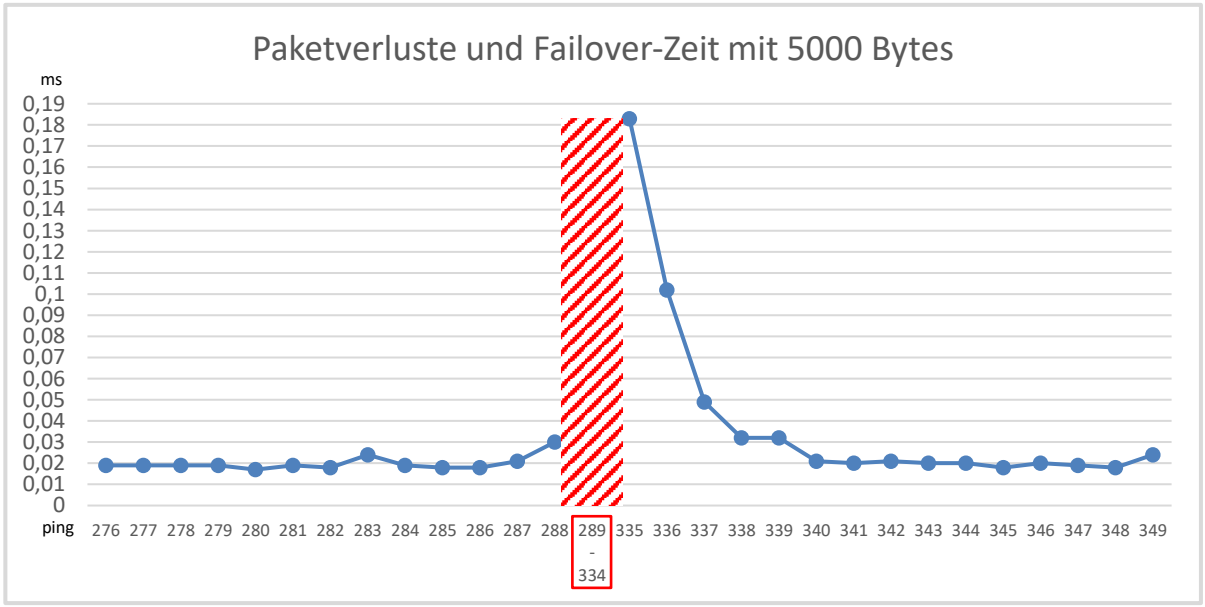



Abbildung 4.13: Paketverluste und Failover-Zeit mit 5000 Bytes

## 4.3 Diskussion der Ergebnisse

Um Aussagen zur Echtzeitfähigkeit von Kubernetes und Container-Technologien treffen zu können, wurde durch eine quantitative Untersuchung die Reaktionsfähigkeit mittels Latenzzeitmessungen und Tests zu Ausfallzeiten analysiert. Im ersten Testfall wurde hierfür ein lokales Kubernetes-Cluster mit einem Arbeitsknoten aufgestellt. Im Verlauf der Analyse wurden zwei Pods erstellt, in welchen die Latenzzeitmessungen stattgefunden haben. Dabei erfolgten die Messungen zum einen mit unterschiedlichen Systemauslastungen. Zum anderen wurden in einem zweiten Durchlauf die dem Pod zur Verfügung stehenden Hardware-ressourcen erhöht.

Die Ergebnisse zeigen, dass in einem Kubernetes-Cluster ein Echtzeitverhalten grundsätzlich möglich ist. Zu Beginn wurde erwartet, dass mit einer erhöhten Ressourcenzuweisung die Latenzzeiten im Pod nochmal geringer ausfallen werden. Die Ergebnisse nach den wiederholten Messdurchführungen zeigen zwar keine beträchtlichen Unterschiede, jedoch wurde diese Erwartung nicht direkt falsifiziert, da diese Anpassung eine leichte Senkung der Latenzen bewirkt hat. Aus diesem Grund ist an dieser Stelle zu empfehlen, weitere experimentelle Tests mit unterschiedlichen Ressourcenzuweisungen durchzuführen.

Mit den Messwerten wird deutlich, dass die Auslastung der CPU eine Verbesserung hinsichtlich der Latenzzeiten herbeiführt. Auffällig war hingegen die Verschlechterung der Ergebnisse nachdem die Lese- und Schreibvorgänge auf der Festplatte erhöht wurden.

Ergebnisse zu den Ausfallzeiten zeigen, dass die maximale Grenze von 24 ms nicht eingehalten werden kann. Im Hinblick auf die niedrigen Latenzzeiten war die Erwartung, dass die Anforderungen auch in diesem Testfall erfüllt werden. Diesem Problem muss auch in Zusammenhang mit mehr Rechenleistung weiterhin nachgegangen werden. In der taktgebundenen Fertigung würde die Überschreitung der kritischen Zeit zu einem Stillstand der Produktionsanlagen mit erheblichen Folgen führen. An dieser Stelle ist auch zu berücksichtigen, dass es sich in den Untersuchungen um synthetische Tests gehandelt hat und bei weiteren Versuchsaufbauten innerhalb einer realen Produktionsumgebung mit einem

zusätzlichen Overhead zu rechnen ist. Diese Anmerkung trifft auch auf zukünftige Latenzzeitmessungen zu.

Im nächsten Schritt können Validierungen mit echtzeitkritischen Applikationen stattfinden, welche containerisiert in einem Kubernetes-Cluster betrieben werden. Hier können die gleichen Latenzzeitmessungen und auch Untersuchungen zu den Ausfallzeiten mit den resultierenden Paketverlusten durchgeführt werden. Eine entscheidende Rolle bei weiteren Forschungsarbeiten werden die Auslastung des Netzwerks und die verfügbaren Hardware-ressourcen spielen.

# 5. Fazit

Mit diesem Kapitel wird die vorliegende Bachelorarbeit abgeschlossen. Zunächst wird der Inhalt zusammengefasst sowie ein Fazit gezogen. Daraufhin folgt ein Ausblick auf weitere Aspekte, mit deren Betrachtung das Thema Virtualisierung und Kubernetes in der Produktion weitergeführt werden kann.

## 5.1 Zusammenfassung

Das Ziel dieser Arbeit war die Untersuchung von Container-Technologien und des Orchestrierungsdienstes Kubernetes auf die Einsatzmöglichkeit für Applikationen, bei denen ein Echtzeitverhalten gewährleistet sein muss. In diesem Fall handelt es sich um speicherprogrammierbare Steuerungen, welche zukünftig virtualisiert in Containern betrieben werden sollen. Hierfür wurde zunächst eine Teststation präpariert und anschließend ein lokales Kubernetes-Cluster erzeugt. Der Fokus bei den darauffolgenden Messdurchführungen lag auf den Latenzzeiten, welche innerhalb eines gestarteten Pods gemessen wurden. Um Latenzzeiten im erstellten Pod ermitteln zu können, kam die Software Cyclictest zum Einsatz. Dabei fanden die Messdurchführungen während unterschiedlicher Systemauslastungen statt, welche mithilfe eines weiteren Programms künstlich erzeugt wurden. Die Deaktivierung der Hyperthreading-Funktion und das Aufsetzen eines echtzeitfähigen Betriebssystems führten zu deutlich besseren Messwerten. Die Ergebnisse zeigten, dass bei den Tests mit $10^6$ Messungen sowohl beim ersten Pod als auch beim Pod mit erweiterter Ressourcenzuweisung mit reiner CPU-Auslastung die besten Werte mit maximal 43 µs erreicht wurden. Gleicheres gilt für die Langzeittests mit maximal 304 µs. Hier fielen die Werte aufgrund der größeren Zeitspanne allgemein etwas höher aus. An dieser Stelle ist zu empfehlen, in weiterführenden Arbeiten Messungen hinsichtlich Latenzzeiten über einen großen Zeitraum durchzuführen, um auch möglichst viele Ausreißer zu detektieren. So zeigen die Ergebnisse aus den $10^8$ Messungen, dass mit 1007 µs und 1057 µs ohne Systemauslastung die Grenze von 1 Millisekunde überschritten wird. Mit der Kombination von CPU- und Festplattenauslastung lagen die Werte in jedem Testfall mit 131 µs bis 136 µs sehr nah beieinander. Im direkten Vergleich mit reiner CPU-Auslastung ist ersichtlich, dass sich die Werte unter dieser Bedingung um mehr als das Dreifache verschlechtern. Allein beim ersten

Pod ohne erweiterter Ressourcenzuweisung kommt es mit 304 µs und 134 µs zu einem Ausreißer nach unten.

Untersuchungen zu den Paketverlusten und der Verzögerung beim Neustart eines Pods ergaben, dass die Werte von allen Messungen mit jeweils unterschiedlicher Paketgröße die Anforderung nicht erfüllen konnten. Mit der Standard-Paketgröße von 64 Bytes betrug die Failover-Zeit 400 ms. Bei 460 gesendeten Paketen kam es hierbei zu 50 Verlusten. Auch bei den weiteren Messungen mit unterschiedlichen Paketgrößen konnte die Anforderung nicht eingehalten werden. Mit einer Paketgröße von 1000 Bytes wurden 556 Pakete versendet. Davon konnten 77 nicht empfangen werden, was zu einer Verzögerung von 616 ms geführt hat. Wie erwartet lag mit einer Größe von 5000 Bytes die Failover-Zeit auch im letzten Fall bei 368 ms und somit deutlich über dem geforderten Limit von 24 ms. Dabei kamen von 488 gesendeten Paketen 46 Pakete nicht an.

Schlussendlich kann die Aussage getroffen werden, dass Container-Technologien und Kubernetes mit einer optimierten Hard- und Softwareumgebung Echtzeitfähigkeiten vorweisen, da die im Rahmen dieser Arbeit erhaltenen maximalen Latenzzeiten im Bereich von nur knapp über 1 Millisekunde lagen. Jedoch muss sowohl im Bereich der Latenzzeiten als auch im Bereich der Netzwerktests zur Ermittlung von Netzwerk-Metriken experimentell nach weiteren Optimierungslösungen geforscht werden.

## 5.2 Ausblick

Da im Rahmen dieser Arbeit mit dem lokalen Kubernetes-Cluster *minikube* gearbeitet wurde, müssen weiterführende Untersuchungen an einem Cluster mit mehreren Servern stattfinden. Zwar werden die Anforderungen hinsichtlich Echtzeitsysteme mit den durchgeführten Messungen erfüllt, jedoch muss im Hinblick auf die Produktion die Analyse der Echtzeitfähigkeit mit weiteren Versuchsaufbauten fortgesetzt werden. Dabei muss eine deutlich höhere Anzahl an Arbeitsknoten und somit ein umfangreicherer Cluster zum Einsatz kommen. In diesem müssen die speicherprogrammierbaren Steuerungen containerisiert in Betrieb sein, in welchem schließlich Inspektionen zur Hochverfügbarkeit und Ausfallsicherheit stattfinden können. Der Fokus in dieser Arbeit lag auf der Qualität der Latenzzeiten, jedoch ist es zwingend notwendig, auch Netzwerktests zur Ermittlung von Paketverlusten weiterhin durchzuführen. Zu bedenken ist auch, dass bei zukünftigen Forschungsarbeiten in einer realen

Produktionsumgebung mehr Einflussfaktoren eine entscheidende Rolle spielen werden. Beispielsweise wird es in einem Cluster mit zahlreichen containerisierten Anwendungen zu einem höheren Ressourcen-aufwand kommen als es in dieser Arbeit der Fall war. Zudem darf nicht vergessen werden, dass Latenzzeiten und die Failover-Zeit durch das Ausmaß des Traffics im Netzwerk einer realen Produktionsumgebung beeinflusst werden können.

Letztendlich kann gesagt werden, dass sich mit der ständigen Weiterentwicklung von Kubernetes durch die breite Community mehr Möglichkeiten ergeben und immer mehr Anwendungsfälle abgedeckt werden. In den Anfängen konnte Kubernetes beispielsweise die Skalierung nur in Abhängigkeit zur CPU-Auslastung durchführen. Inzwischen kam mit regelmäßigen Updates die horizontale Pod-Skalierung, welche eine Erhöhung der Anzahl von Pods basierend auf dem Nutzerverhalten ermöglicht. Das Erstellen und Betreiben eines Kubernetes-Clusters ist zwar mit einer gewissen Komplexität verbunden und stellt eine große Herausforderung für die weitere Verbreitung dar. Für die Nutzung des Systems sind geschulte Mitarbeiter mit Kenntnissen erforderlich. Die Verwendung von Container-Technologien mit Kubernetes und den dazugehörenden Vorteilen überwiegen jedoch. Im Hinblick auf die Zukunft sind Container-Lösungen mit Kubernetes als Standard im Bereich der Container-Orchestrierung zur Automatisierung von Betriebsprozessen auf dem guten Weg, sich als innovatives Werkzeug in der IT-Welt zu etablieren.

# Abbildungsverzeichnis

# Literaturverzeichnis

[1] M. Hille, „Mirantis schnappt sich Docker Enterprise – das Ende von Docker Swarm …und noch viel mehr?“, *Cloudflight*, 20. Nov. 2019, 2019. [Online]. Verfügbar unter: https://de.cloudflight.io/presse/mirantis-schnappt-sich-docker-enterprise-das-ende-von-docker-swarm-und-noch-viel-mehr-35723/. Zugriff am: 10. Februar 2022.

[2] B. Burns, J. Beda und K. Hightower, *Kubernetes: Eine kompakte Einführung,* 2. Aufl. Heidelberg: dpunkt.verlag, 2021. [Online]. Verfügbar unter: https://search.ebscohost.com/login.aspx?direct=true&scope=site&db=nlebk&db=nlabk&AN=2708961

[3] *Ratgeber: SPS und Automatisierung | RS Components.* [Online]. Verfügbar unter: https://de.rs-online.com/web/generalDisplay.html?id=ideen-und-tipps/sps-leitfaden (Zugriff am: 18. Februar 2022).

[4] C. Meinel, C. Willems, S. Roschke und M. Schnjakin, *Virtualisierung und Cloud Computing: Konzepte, Technologiestudie, Marktübersicht*. Universitätsverlag Potsdam, 2011.

[5] *Was ist Virtualisierung?* [Online]. Verfügbar unter: https://www.redhat.com/de/topics/virtualization/what-is-virtualization (Zugriff am: 10. Februar 2022).

[6] *Betrachtung gängiger Softwarelösungen zur Isolierung von Applikationen auf Edge Devices im industriellen Umfeld anhand von Container-basierter …* [Online]. Verfügbar unter: https://www.mivp.tuwien.ac.at/fileadmin/t/ikl/MIVP/one-pager/Bachelorarbeit_ELSNERJohannes_01351461_ABGABE.pdf

[7] A. Baier, „Risikobestimmung von manipulierten Dockerhub-Containern: Bachelorthesis“, Hochschule für Angewandte Wissenschaften Landshut. [Online]. Verfügbar unter: https://opus4.kobv.de/opus4-haw-landshut/frontdoor/index/index/docId/71

[8] Michael Bose, *NAKIVO Blog: Hyper-V vs. VirtualBox-Hypervisor-Typen.* [Online]. Verfügbar unter: https://www.nakivo.com/blog/de/hyper-v-oder-virtualbox-welche-von-ihnen-fuer-ihre-infrastruktur-zu-waehlen-ist/ (Zugriff am: 10. Februar 2022).

[9] *Potenziale der Digitalisierung für kleine und mittlere Unternehmen*, 2020. [Online]. Verfügbar unter: https://hbz.opus.hbz-nrw.de/opus45-kola/files/2020/bachelorarbeit+fabienne+hohl+215202227.pdf

[10] *Konzeption und Realisierung einer Plattform zur Echtzeit-Netzwerkdatenanalyse*, 2017. [Online]. Verfügbar unter: https://reposit.haw-hamburg.de/bitstream/20.500.12738/8154/1/masterthesis.pdf

[11] E. Wolff, *Microservices: Grundlagen flexibler Softwarearchitekturen,* 2. Aufl. Heidelberg: dpunkt.verlag, 2018. [Online]. Verfügbar unter: https://search.ebscohost.com/login.aspx?direct=true&scope=site&db=nlebk&db=nlabk&AN=1856119

[12] *microservices vs monolith - Sekt oder Selters? - ITBALANCE.* [Online]. Verfügbar unter: https://itbalance.de/mvsm-2020-09-17-10/ (Zugriff am: 10. Februar 2022).

[13] *Containisierung von Java Apps mit Docker*, 2018. [Online]. Verfügbar unter: https://www.tschutschu.de/resources/tschutschu/docs/ss2018/ss2018_t18_kaluginvera_studienarbeit.pdf

[14] A. Mouat, *Docker: Software entwickeln und deployen mit Containern*. dpunkt.verlag, 2016.

[15] *Testen und Docker.* [Online]. Verfügbar unter: https://fg tav.gi.de/fileadmin/FG/TAV/40.TAV/3_TAV40_DehlaSokenou.pdf

[16] T. Drilling, „Images und Container in Docker“, *DataCenter-Insider*, 26. Sep. 2018, 2018. [Online]. Verfügbar unter: https://www.datacenter-insider.de/images-und-container-in-docker-a-759636/. Zugriff am: 10. Februar 2022.

[17] *Entwicklung eines Verwaltungssystem für Cluster, auf welchem virtuelle Maschinen und High-Performance-Computing-Anwendungen gemeinsam ausgeführt …* [Online]. Verfügbar unter: http://www.inf.fu-berlin.de/inst/ag-se/teaching/S-BSE/347_goerick-HPCVM-Mngmnt-thesis.pdf

[18] Avi, „Docker vs Virtual Machine - Die Unterschiede verstehen“, *Geekflare*, 15. Sep. 2019, 2019. [Online]. Verfügbar unter: https://geekflare.com/de/docker-vs-virtual-machine/. Zugriff am: 10. Februar 2022.

[19] *Urban Data Platform Hamburg: Integration von Echtzeit IoT-Daten mittels SensorThings API*, 2021. [Online]. Verfügbar unter: https://www.google.de/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&ved=2ahUKEwiomuKouYb2AhWmS_EDHfNAAncQFnoECAwQAQ&url=https%3A%2F%2Fgeodaesie.info%2Fzfv%2Fheftbeitrag%2F8651%2Fzfv_2021_1_Fischer_etal.pdf&usg=AOvVaw1K37P46r91FflN5LtODsfu

[20] F. Mangels, „Analyse der Sicherheit und der automatisierten Bereitstellung eines On-Premises-Clusters auf der Grundlage der Container-basierten Virtualisierung: Kubernetes im Wissenschaftsbetrieb“. [Online]. Verfügbar unter: https://epic.awi.de/id/eprint/52946/

[21] Avi, „Grundlegendes zur Kubernetes-Architektur“, *Geekflare*, 2. Jan. 2020, 2020. [Online]. Verfügbar unter: https://geekflare.com/de/kubernetes-architecture/. Zugriff am: 10. Februar 2022.

[22] M. Oppermann, *Erhöhte Skalierbarkeit durch eine Serverlessimplementierung von Downloadvorgängen im GeRDI Projekt unter dem Einsatz von Kubernetes*, 2019. [Online]. Verfügbar unter: https://oceanrep.geomar.de/48088/

[23] G. Wellenreuther und D. Zastrow, *Automatisieren mit SPS Theorie und Praxis: IEC 61131-3; STEP 7; Bibliotheksbausteine; AS-i-Bus; PROFIBUS; Ethernet-TCP/IP; OPC; Steuerungssicherheit*. Springer-Verlag, 2013.

[24] Volkswagen AG, Hg., *Konzerneinheitliches Lastenheft (KELH): Teil I-B09: Anlagenelektrik*, 2020.

[25] O. Gräfe, „Entwicklung einer EtherCAT Verbindung zwischen einem 32-bit PIC Mikrocontroller und einer Soft-SPS: Entwicklung einer EtherCAT Verbindung zwischen einem 32-bit PIC Mikrocontroller und einer Soft-SPS“, Hochschule für angewandte Wissenschaften Hamburg. [Online]. Verfügbar unter: https://reposit.haw-hamburg.de/handle/20.500.12738/9753

[26] I. Gosetti, „Formale Beschreibung einer IEC 61499- Laufzeitumgebung unter Berücksichtigung von Echtzeitanforderungen und dem unterlagerten Betriebssystem: Formale Beschreibung einer IEC 61499- Laufzeitumgebung unter Berücksichtigung von Echtzeitanforderungen und dem unterlagerten Betriebssystem“. [Online]. Verfügbar unter: https://repositum.tuwien.at/handle/20.500.12708/14222

[27] *Real-time containers: A survey*, 2020. [Online]. Verfügbar unter: https://drops.dagstuhl.de/opus/volltexte/2020/12001/

[28] Thomas Maierhofer, „Linux RT-Preempt Echtzeitkenngrößen ermitteln und testen: Embedded Testing 2017“, 2017. [Online]. Verfügbar unter: https://www.embedded-testing.de/files/cleancode/site/vortraege2017/tag2/D3.7_Vortrag_Thomas_Maierhofer.pdf

[29] *The Linux Foundation: Cyclictest.* [Online]. Verfügbar unter: https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/start. (Zugriff am: 10. Februar 2022).

[30] *Verteiltes Messen der Dienstgüte und Netzwerk-Performance in IP-Netzen*, 2005. [Online]. Verfügbar unter: https://www.repo.uni-hannover.de/bitstream/handle/123456789/6665/495623172.pdf?sequence=1

[31] Thomas Kampa, „Experimentelle Validierung von Lösungsansätzen eines konvergenten Netzwerks als Basis für eine Smart Factory“. Masterarbeit, Technische Universität, Darmstadt, 2021.

[32] S. Luber, „Was ist ping?“, *IP-Insider*, 1. Aug. 2018, 2018. [Online]. Verfügbar unter: https://www.ip-insider.de/was-ist-ping-a-681078/. Zugriff am: 24. Februar 2022.

# Anhang

A1 Cyclictest-Messwerte auf dem Host ohne Echtzeiteigenschaften nach einem 20-minütigen Durchlauf:

| CPU-Kerne | Minimale Latenz [µs] | Ø-Latenz [µs] | Maximale Latenz [µs] |
|---|---|---|---|
| 0 | 1 | 2 | 505 |
| 1 | 1 | 2 | 283 |
| 2 | 1 | 2 | 404 |
| 3 | 1 | 2 | 410 |
| 4 | 1 | 2 | 347 |
| 5 | 2 | 2 | 554 |
| 6 | 2 | 2 | 323 |
| 7 | 2 | 2 | 353 |
| 8 | 2 | 2 | 397 |
| 9 | 2 | 2 | 390 |
| 10 | 2 | 2 | 245 |
| 11 | 2 | 2 | 367 |
| 12 | 2 | 2 | 420 |
| 13 | 2 | 2 | 339 |
| 14 | 2 | 2 | 387 |
| 15 | 2 | 2 | 272 |
| 16 | 2 | 2 | 306 |
| 17 | 2 | 2 | 1350 |
| 18 | 2 | 2 | 370 |
| 19 | 2 | 2 | 513 |
| 20 | 2 | 2 | 380 |
| 21 | 2 | 2 | 260 |
| 22 | 2 | 2 | 684 |
| 23 | 2 | 2 | 130 |
| 24 | 2 | 3 | 367 |
| 25 | 2 | 2 | 741 |
| 26 | 2 | 3 | 407 |
| 27 | 2 | 3 | 131 |

A2 Cyclictest-Messwerte auf dem Host nach Installation des Echtzeit-Kernels über einen RT-PREEMPT-Patch und Systemanpassungen nach einem 20-minütigen Durchlauf:

| CPU-Kerne | Minimale Latenz [µs] | Ø-Latenz [µs] | Maximale Latenz [µs] |
|---|---|---|---|
| 0 | 1 | 2 | 22 |
| 1 | 1 | 2 | 20 |
| 2 | 2 | 2 | 18 |
| 3 | 2 | 2 | 18 |
| 4 | 2 | 2 | 29 |
| 5 | 2 | 2 | 19 |
| 6 | 2 | 2 | 19 |
| 7 | 2 | 2 | 19 |
| 8 | 1 | 2 | 18 |
| 9 | 1 | 2 | 17 |
| 10 | 2 | 2 | 19 |
| 11 | 2 | 2 | 17 |
| 12 | 2 | 2 | 19 |
| 13 | 2 | 2 | 20 |

A3 Cyclictest-Messwerte im Pod ohne Systemlast nach einem 20-minütigen Durchlauf:

| CPU-Kerne | Minimale Latenz [µs] | Ø-Latenz [µs] | Maximale Latenz [µs] |
|---|---|---|---|
| 0 | 1 | 2 | 759 |
| 1 | 1 | 2 | 348 |
| 2 | 2 | 3 | 221 |
| 3 | 2 | 14 | 404 |
| 4 | 1 | 27 | 417 |
| 5 | 2 | 48 | 438 |
| 6 | 2 | 67 | 848 |
| 7 | 2 | 79 | 438 |
| 8 | 2 | 85 | 434 |
| 9 | 2 | 90 | 428 |
| 10 | 2 | 90 | 515 |
| 11 | 2 | 91 | 431 |
| 12 | 2 | 87 | 444 |
| 13 | 2 | 82 | 431 |

A4 Cyclictest-Messwerte im Pod unter 100 % CPU-Last nach einem 20-minütigen Durchlauf:

| CPU-Kerne | Minimale Latenz [µs] | Ø-Latenz [µs] | Maximale Latenz [µs] |
|---|---|---|---|
| 0 | 1 | 2 | 28 |
| 1 | 1 | 2 | 29 |
| 2 | 2 | 2 | 34 |
| 3 | 2 | 3 | 45 |
| 4 | 2 | 3 | 29 |
| 5 | 2 | 3 | 26 |
| 6 | 2 | 3 | 29 |
| 7 | 2 | 3 | 31 |
| 8 | 2 | 3 | 353 |
| 9 | 2 | 3 | 44 |
| 10 | 2 | 3 | 30 |
| 11 | 2 | 3 | 29 |
| 12 | 2 | 3 | 357 |
| 13 | 2 | 4 | 34 |

A5 Bash-Script vom Open Source Automation Development Lab eG (OSADL) zum Messen und graphischen Darstellen der Werte:

```
#!/bin/bash

# 1. Run cyclictest
cyclictest -l100000000 -m -Sp90 -i200 -h400 -q > output

# 2. Get maximum latency
max=`grep "Max Latencies" output | tr " " "\n" | sort -n | tail -1 | sed
s/^0*//`

# 3. Grep data lines, remove empty lines and create a common field
# separator
grep -v -e "^#" -e "^$" output | tr " " "\t" >histogram

# 4. Set the number of cores, for example
cores=14

# 5. Create two-column data sets with latency classes and frequency
# values for each core, for example
for i in `seq 1 $cores`
do
  column=`expr $i + 1`
  cut -f1,$column histogram >histogram$i
done

# 6. Create plot command header
echo -n -e "set title \"Latency plot\"\n\
set terminal png\n\
set xlabel \"Latency (us), max $max us\"\n\
set logscale y\n\
set xrange [0:400]\n\
set yrange [0.8:*]\n\
set ylabel \"Number of latency samples\"\n\
set output \"plot.png\"\n\
plot " >plotcmd

# 7. Append plot command data references
for i in `seq 1 $cores`
do
  if test $i != 1
  then
    echo -n ", " >>plotcmd
  fi
  cpuno=`expr $i - 1`
  if test $cpuno -lt 10
  then
    title=" CPU$cpuno"
   else
    title="CPU$cpuno"
  fi
  echo -n "\"histogram$i\" using 1:2 title \"$title\" with histeps"
>>plotcmd
done

# 8. Execute plot command
gnuplot -persist <plotcmd
```

## A6 Ausschnitt der Messergebnisse von Ping mit der Standard-Paketgröße von 64 Bytes:

```
64 bytes from 172.17.0.2: icmp_seq=250 ttl=64 time=0.006 ms
64 bytes from 172.17.0.2: icmp_seq=251 ttl=64 time=0.008 ms
64 bytes from 172.17.0.2: icmp_seq=252 ttl=64 time=0.006 ms
64 bytes from 172.17.0.2: icmp_seq=253 ttl=64 time=0.007 ms
64 bytes from 172.17.0.2: icmp_seq=254 ttl=64 time=0.009 ms
64 bytes from 172.17.0.2: icmp_seq=255 ttl=64 time=0.009 ms
64 bytes from 172.17.0.2: icmp_seq=256 ttl=64 time=0.007 ms
64 bytes from 172.17.0.2: icmp_seq=257 ttl=64 time=0.006 ms
64 bytes from 172.17.0.2: icmp_seq=258 ttl=64 time=0.008 ms
64 bytes from 172.17.0.2: icmp_seq=259 ttl=64 time=0.017 ms
64 bytes from 172.17.0.2: icmp_seq=310 ttl=64 time=0.130 ms
64 bytes from 172.17.0.2: icmp_seq=311 ttl=64 time=0.032 ms
64 bytes from 172.17.0.2: icmp_seq=312 ttl=64 time=0.021 ms
64 bytes from 172.17.0.2: icmp_seq=313 ttl=64 time=0.013 ms
64 bytes from 172.17.0.2: icmp_seq=314 ttl=64 time=0.010 ms
64 bytes from 172.17.0.2: icmp_seq=315 ttl=64 time=0.011 ms
64 bytes from 172.17.0.2: icmp_seq=316 ttl=64 time=0.008 ms
64 bytes from 172.17.0.2: icmp_seq=317 ttl=64 time=0.006 ms
64 bytes from 172.17.0.2: icmp_seq=318 ttl=64 time=0.009 ms
64 bytes from 172.17.0.2: icmp_seq=319 ttl=64 time=0.010 ms
64 bytes from 172.17.0.2: icmp_seq=320 ttl=64 time=0.010 ms
```

A7 Ausschnitt der Messergebnisse von Ping mit erhöhter Paketgröße von 1000 Bytes:

```
1000 bytes from 172.17.0.2: icmp_seq=300 ttl=64 time=0.007 ms
1000 bytes from 172.17.0.2: icmp_seq=301 ttl=64 time=0.007 ms
1000 bytes from 172.17.0.2: icmp_seq=302 ttl=64 time=0.008 ms
1000 bytes from 172.17.0.2: icmp_seq=303 ttl=64 time=0.008 ms
1000 bytes from 172.17.0.2: icmp_seq=304 ttl=64 time=0.008 ms
1000 bytes from 172.17.0.2: icmp_seq=305 ttl=64 time=0.008 ms
1000 bytes from 172.17.0.2: icmp_seq=306 ttl=64 time=0.009 ms
1000 bytes from 172.17.0.2: icmp_seq=307 ttl=64 time=0.007 ms
1000 bytes from 172.17.0.2: icmp_seq=308 ttl=64 time=0.008 ms
1000 bytes from 172.17.0.2: icmp_seq=309 ttl=64 time=0.007 ms
1000 bytes from 172.17.0.2: icmp_seq=310 ttl=64 time=0.012 ms
1000 bytes from 172.17.0.2: icmp_seq=311 ttl=64 time=0.007 ms
1000 bytes from 172.17.0.2: icmp_seq=312 ttl=64 time=0.007 ms
1000 bytes from 172.17.0.2: icmp_seq=313 ttl=64 time=0.007 ms
1000 bytes from 172.17.0.2: icmp_seq=314 ttl=64 time=0.012 ms
1000 bytes from 172.17.0.2: icmp_seq=315 ttl=64 time=0.008 ms
1000 bytes from 172.17.0.2: icmp_seq=393 ttl=64 time=0.161 ms
1000 bytes from 172.17.0.2: icmp_seq=394 ttl=64 time=0.034 ms
1000 bytes from 172.17.0.2: icmp_seq=395 ttl=64 time=0.022 ms
1000 bytes from 172.17.0.2: icmp_seq=396 ttl=64 time=0.015 ms
1000 bytes from 172.17.0.2: icmp_seq=397 ttl=64 time=0.011 ms
1000 bytes from 172.17.0.2: icmp_seq=398 ttl=64 time=0.011 ms
1000 bytes from 172.17.0.2: icmp_seq=399 ttl=64 time=0.010 ms
1000 bytes from 172.17.0.2: icmp_seq=400 ttl=64 time=0.007 ms
```

## A8 Ausschnitt der Ergebnisse von Ping mit erhöhter Paketgröße von 5000 Bytes:

```
5000 bytes from 172.17.0.2: icmp_seq=280 ttl=64 time=0.017 ms
5000 bytes from 172.17.0.2: icmp_seq=281 ttl=64 time=0.019 ms
5000 bytes from 172.17.0.2: icmp_seq=282 ttl=64 time=0.018 ms
5000 bytes from 172.17.0.2: icmp_seq=283 ttl=64 time=0.024 ms
5000 bytes from 172.17.0.2: icmp_seq=284 ttl=64 time=0.019 ms
5000 bytes from 172.17.0.2: icmp_seq=285 ttl=64 time=0.018 ms
5000 bytes from 172.17.0.2: icmp_seq=286 ttl=64 time=0.018 ms
5000 bytes from 172.17.0.2: icmp_seq=287 ttl=64 time=0.021 ms
5000 bytes from 172.17.0.2: icmp_seq=288 ttl=64 time=0.030 ms
5000 bytes from 172.17.0.2: icmp_seq=335 ttl=64 time=0.183 ms
5000 bytes from 172.17.0.2: icmp_seq=336 ttl=64 time=0.102 ms
5000 bytes from 172.17.0.2: icmp_seq=337 ttl=64 time=0.049 ms
5000 bytes from 172.17.0.2: icmp_seq=338 ttl=64 time=0.032 ms
5000 bytes from 172.17.0.2: icmp_seq=339 ttl=64 time=0.032 ms
5000 bytes from 172.17.0.2: icmp_seq=340 ttl=64 time=0.021 ms
5000 bytes from 172.17.0.2: icmp_seq=341 ttl=64 time=0.020 ms
5000 bytes from 172.17.0.2: icmp_seq=342 ttl=64 time=0.021 ms
5000 bytes from 172.17.0.2: icmp_seq=343 ttl=64 time=0.020 ms
5000 bytes from 172.17.0.2: icmp_seq=344 ttl=64 time=0.020 ms
5000 bytes from 172.17.0.2: icmp_seq=345 ttl=64 time=0.018 ms
5000 bytes from 172.17.0.2: icmp_seq=346 ttl=64 time=0.020 ms
5000 bytes from 172.17.0.2: icmp_seq=347 ttl=64 time=0.019 ms
5000 bytes from 172.17.0.2: icmp_seq=348 ttl=64 time=0.018 ms
5000 bytes from 172.17.0.2: icmp_seq=349 ttl=64 time=0.024 ms
5000 bytes from 172.17.0.2: icmp_seq=350 ttl=64 time=0.021 ms
```