FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Fachhochschule Aachen, Campus Jülich

Fachbereich 09 - Medizintechnik und Technomathematik
Studiengang Angewandte Mathematik und Informatik

Seminararbeit

Analyse der Anforderungen einer Kommunikationsstrecke und Entwicklung einer Lastsimulation zur Validierung derselben

Eingereicht von:
Henrik van Onna, 3521517

Erster Prüfer:
Prof. Dr. rer. nat. Volker Sander

Zweiter Prüfer:
Dr. rer. nat. Simon Becker

Aachen, den 4. Januar 2024

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die Seminararbeit mit dem Thema „*Analyse der Anforderungen einer Kommunikationsstrecke und Entwicklung einer Lastsimulation zur Validierung derselben*“ selbstständig angefertigt, verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Ich verpflichte mich, ein Exemplar der Seminararbeit fünf Jahre aufzubewahren und auf Verlangen dem Prüfungsamt des Fachbereiches Medizintechnik und Technomathematik auszuhändigen.

03.01.2024, Hiam Omra

(Datum, Unterschrift des Studierenden)

Inhaltsverzeichnis

1 Einleitung und Motivation

In einer zunehmend digitalen Welt ist die Verlagerung von Anwendungen in die Cloud für Unternehmen zu einem entscheidenden Schritt zur Steigerung der Flexibilität, Skalierbarkeit und Effizienz geworden. Die Auslagerung von Anwendungen in die Cloud ermöglicht es Unternehmen außerdem, sich auf ihre Kernkompetenzen zu konzentrieren, und weniger Zeit und Ressourcen für die Verwaltung von IT-Infrastrukturen aufzuwenden. Auch das Produkt PRODIS.PlantHUB der DSA Daten- und Systemtechnik GmbH bildet da keine Ausnahme. PlantHUB wird in der Fahrzeugproduktion eingesetzt und ist für die Überwachung und zentrale Steuerung der Endmontage einer Produktionsumgebung (auch Werk) verantwortlich. Aktuell ist PlantHUB auf einem Server in der Produktionsstätte installiert und kommuniziert über ein lokales Netzwerk mit verschiedenen Systemen im Werk, darunter Befüllanlagen und Prüfständen. Mit einem Umzug in die Cloud würde die Kommunikation über das Internet stattfinden und Latenzzeiten und Stabilität der Verbindung könnten sich ändern. Bei Verbindungsabbrüchen oder zu hoher Latenz könnte es zu Problemen kommen, im schlimmsten Fall könnte die Produktion still stehen. Mit jeder Minute und nicht produziertem Fahrzeug entsteht ein Schaden von vielen tausend Euro. Um eine sichere Migration in die Cloud zu ermöglichen, soll in dieser Arbeit eine Lastsimulation entwickelt werden, mit der eine Kommunikationsstrecke getestet und validiert werden kann.
Zunächst müssen die Anforderungen an die Kommunikationsstrecke festgelegt werden. Dazu soll die Kommunikation zwischen PlantHUB und einer Produktionsstätte analysiert werden, um dann eine Lastsimulation zu entwickeln, die die typische Kommunikation über einen längeren Zeitraum simuliert. Gleichzeitig soll die Reaktionszeit von PlantHUB gemessen werden. Die gemessene Zeit kann dann mit den Anforderungen verglichen werden, um zu prüfen, ob die neue Kommunikationsstrecke die Anforderungen erfüllt.
Zu Beginn dieser Arbeit wird PlantHUB und dessen Aufgaben vorgestellt. Darauf folgt eine Analyse der Kommunikation zwischen PlantHUB und einer Produktionsumgebung, um die Anforderungen an die Kommunikationsstrecke zu ermitteln. Danach werden die Idee und Implementierung der Lastsimulation sowie Zeitmessung beschrieben, und es werden die verwendeten Technologien vorgestellt. Darauf folgt die Durchführung der Lastsimulation und die Auswertung der Ergebnisse. Abschließend wird ein Fazit gezogen und ein Ausblick auf mögliche Erweiterungen gegeben.

2 Projektumfeld

2.1 DSA Daten- und Systemtechnik GmbH

Die DSA Daten- und Systemtechnik GmbH wurde 1980 gegründet und hat ihren Hauptsitz in Oberforstbach bei Aachen. DSA beschäftigt mehr als 850 Mitarbeiter in mehreren Niederlassungen weltweit. Das Unternehmen entwickelt Software- und Hardwarelösungen für die Automobilindustrie in den Bereichen Fahrzeugelektronik, Diagnose, Connected-Vehicle und Aftersales. Des Weiteren bietet DSA Qualitätssysteme für die Flughafenlogistik und die Baustoffindustrie an. [DSA]

2.2 PRODIS.PlantHUB

PlantHUB wird in der Fahrzeugproduktion eingesetzt und ist für die Überwachung und zentrale Steuerung der Endmontage einer Produktionsumgebung verantwortlich. Die Endmontage ist der letzte Schritt in der Produktion eines Fahrzeugs. Hier werden die Karosserie und das Interieur zusammen gefügt und das Fahrzeug wird mit Scheinwerfern, Rädern und anderen Teilen ausgestattet. Entlang der Fertigungslinie sind verschiedene Fertigungsbereiche angeordnet, in denen eben genannte Arbeitsschritte durchgeführt werden. Des Weiteren existieren Prüforte, an denen Kalibrierungen, Flash-Vorgänge und Prüfpläne durchgeführt werden können. Die Prüfpläne testen beispielsweise die Funktionalität von Scheinwerfern oder der Elektronik.
Zu Beginn der Produktion wird ein Fahrzeug mit einem Vehicle Communication Interface (**VCI**) verbunden. Damit kann PlantHUB das Fahrzeug identifizieren und mit diesem kommunizieren. Während eines Prüfplans wird über das VCI mit den Steuergeräten im Fahrzeug kommuniziert. Nach der Fertigstellung eines Fahrzeugs wird das VCI wieder entfernt und kann für ein anderes Fahrzeug verwendet werden.
Die Prüfpläne werden von virtuellen Testern (**RTS**) ausgeführt. Diese befinden sich auf einem Server in der Produktionsstätte. Dieser Server ist nicht Teil von PlantHUB und würde bei einem Umzug von PlantHUB in die Cloud weiter in der Produktionsstätte bleiben. Tester sind einem Prüfort zugeordnet und wenn ein Fahrzeug in diesen fährt, reserviert PlantHUB einen der zugeordneten Tester und dieser führt dann den Prüfplan aus.
Außerdem können Werksmitarbeitende mit Hilfe von einem Human Machine Interface (**HMI**) ein Fahrzeug scannen, und so aktuelle Informationen über das Fahrzeug und dessen ausgeführte Prüfpläne erhalten. Bei Fehlern in Prüfablaufen oder anderen Mängeln kann das Fahrzeug auch für die Nacharbeit registriert werden. VCI's, RTS's und HMI's werden im Folgenden auch als **Geräte** bezeichnet. Diese sind auch in Abbildung 2.2 dargestellt.
All diese Komponenten werden von PlantHUB verwaltet und überwacht. In einer Webanwendung wird die Produktionsumgebung visualisiert und die aktuellen Zustände der Fahrzeuge, Tester und Prüfpläne angezeigt. Durch Logging kann der Verlauf der Produktion nachvollzogen werden.

In Abbildung 2.1 ist die Webanwendung dargestellt. Zu erkennen ist die Übersicht einer Linie, welche wiederum in Sektionen unterteilt ist. Entlang der Linie sind Fahrzeuge und deren aktueller Status dargestellt. Der Prüfbereich *Commissioning* (Inbetriebnahme) ist hellblau hinterlegt. Hier werden die Steuergeräte des Fahrzeugs in Betrieb genommen und geprüft.

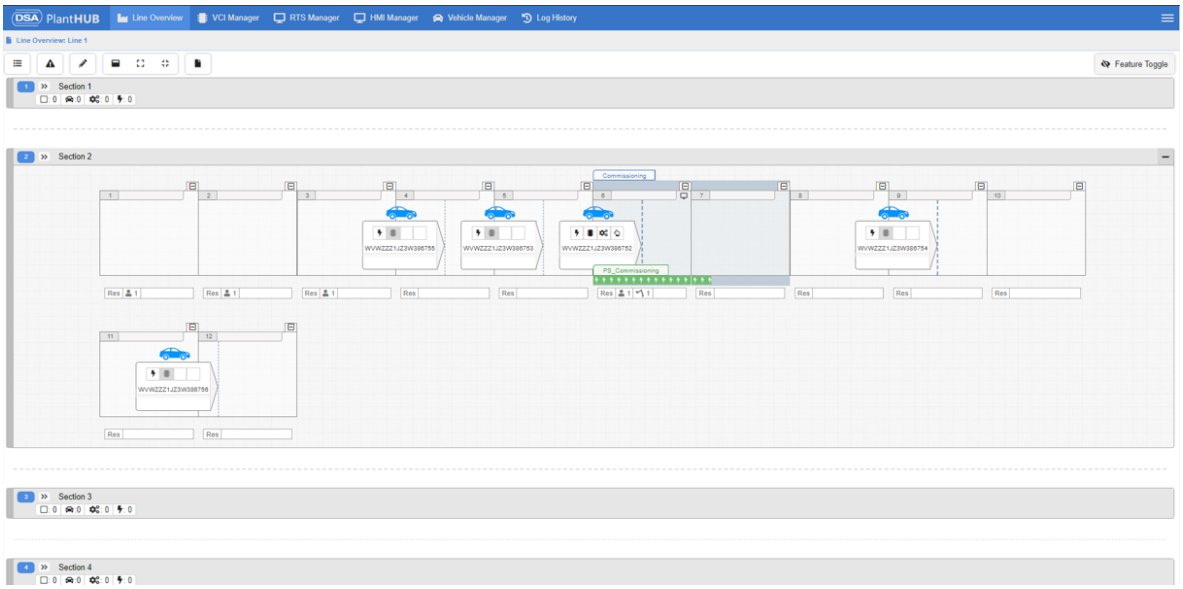

Abbildung 2.1: PlantHUB Webanwendung

2.2.1 Architektur

PlantHUB ist eventbasiert und besteht aus über 20 Microservices, die auf einem Linux-Server ausgeführt werden. Die Microservices lassen sich in drei Kategorien einteilen: Services, Adapter, Agents.
Services sind für die Datenhaltung und -verarbeitung zuständig. Sie halten die digitale Repräsentation der Produktionsumgebung, der Fahrzeuge und der Geräte. Außerdem erzeugen sie Events bei Änderungen der Daten. Die wahrscheinlich wichtigsten Services sind der *ph-product-service*, *rts-service*, *vci-service* und *hmi-service*. Diese sind jeweils für die Fahrzeug-, RTS-, VCI- und HMI-Daten zuständig. Die Geräte melden aktuelle Zustände an den entsprechenden Service, dieser speichert diese in einer Datenbank. Ein HMI meldet beispielsweise, dass ein Fahrzeug gescannt oder zur Nacharbeit registriert wurde. Ein RTS meldet den Status eines Prüfplans, ob dieser z.B. gestartet, abgeschlossen oder abgebrochen wurde. Hier entstehen regelmäßig Nachrichten aus dem Werk und PlantHUB.
Adapter sind oft kundenspezifisch und kommunizieren mit Systemen aus der Produktionsumgebung, beispielsweise erhalten sie Informationen über die Position eines Fahrzeugs, validieren und formatieren diese und senden die Datenänderungen über eine REST-Schnittstelle an den *ph-product-service*. Auch hier herrscht eine regelmäßige Kommunikation zwischen der Produktionsumgebung und PlantHUB.
Agents beinhalten die Geschäftslogik und reagieren auf Events, die von den Services erzeugt werden. Beispielhaft sei hier der *action-trigger-agent* genannt, dieser reagiert auf Events, die von z.B. Fahrzeugspostionsänderungen, HMI-Scans oder Testabbrüchen ausgelöst werden. Der Agent kann dann beispielsweise einen Prüfplan starten, wenn sich das Fahrzeug in einem Prüfort befindet.

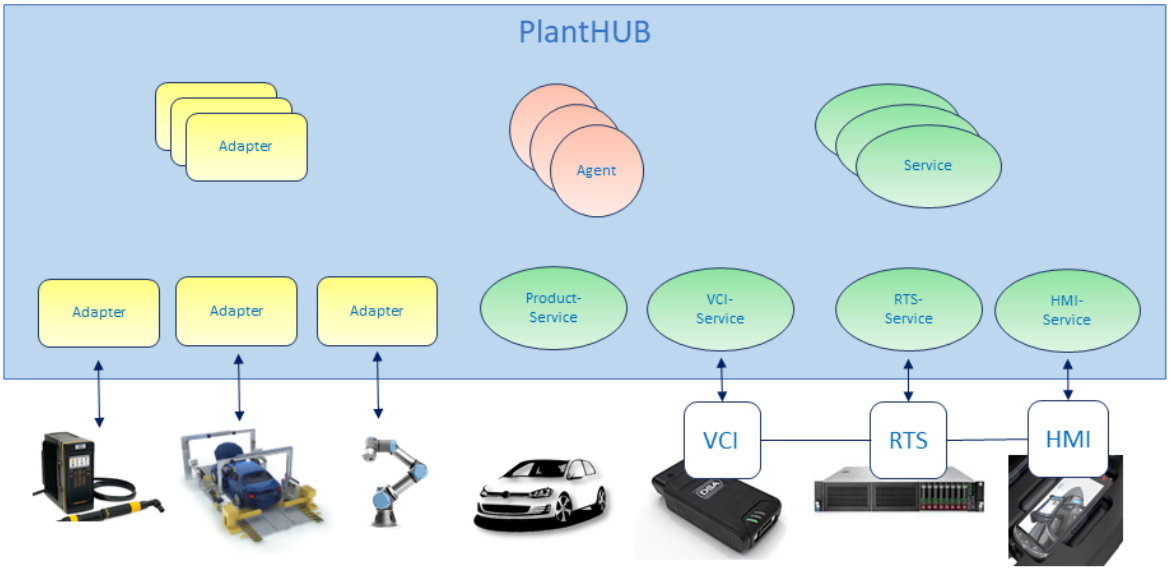

Abbildung 2.2: PlantHUB Architektur

2.2.2 Standard Anwendungsfall

PlantHUB's Standard Anwendungsfall sieht wie folgt aus: Ein Fahrzeug befindet sich vor einem Prüfort, ein VCI ist mit dem Fahrzeug verbunden und ein RTS ist dem Prüfort zugeordnet. Das Fahrzeug fährt dann in den Prüfort. PlantHUB erfährt von dieser Produktionsänderung und löst ein Event aus. Der *action-trigger-agent* reagiert darauf, und weil sich das Fahrzeug jetzt in einem Prüfort befindet, soll ein Tester für die Prüfung reserviert werden. Dazu selektiert der Agent einen der dem Prüfort zugeordneten Tester. Der letzte Schritt ist das Setzen eines Attributes in dem Tester, um dieses für die Prüfung an dem Fahrzeug zu reservieren. Dann ist es die Aufgabe des Testers den Prüfplan zu starten und den erfolgten Start an PlantHUB zu melden.
Während der Selektion oder Reservierung des Testers kann es zu unterschiedlichen Fehlern kommen. Zum Beispiel kann der Tester bereits für eine andere Prüfung reserviert oder offline sein. Falls kein anderer Tester verfügbar ist, kann der Prüfplan nicht gestartet werden.

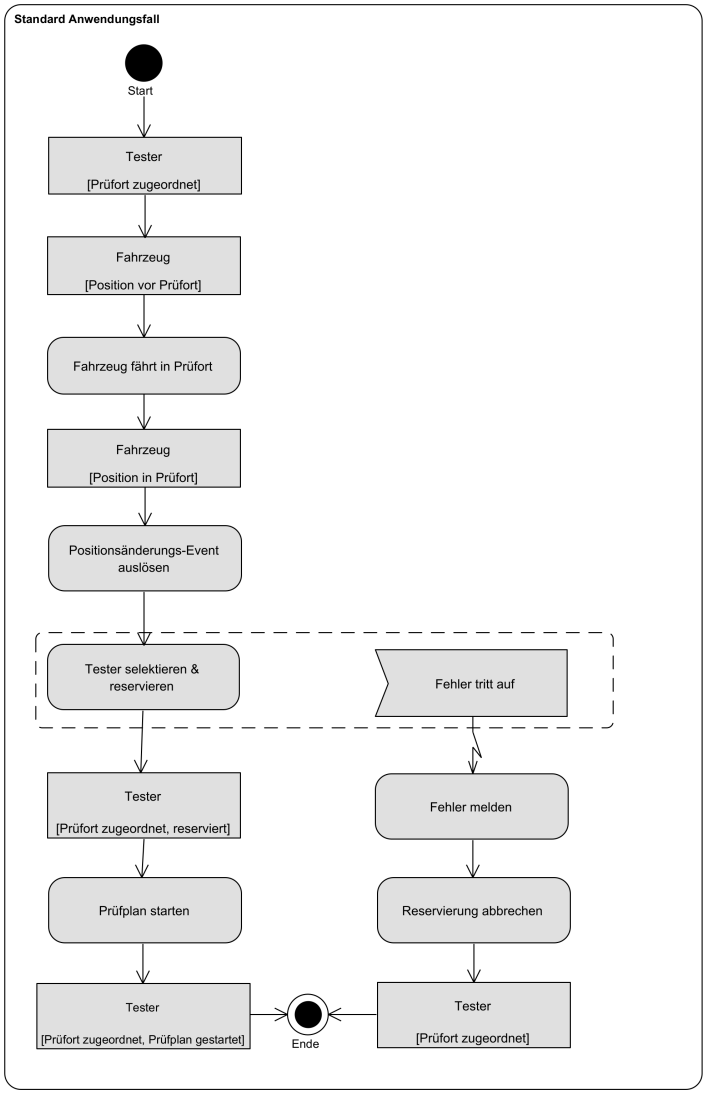

Abbildung 2.3: PlantHUB Standard Anwendungsfall

3 Anforderungsanalyse

3.1 Kommunikation

Die Kommunikation zwischen ausgewählten Services (*ph-product-service*, *rts-service*, *vci-service*, *hmi-service*) und den entsprechenden Geräten in der Produktionsumgebung soll als Richtwert für die zu erzeugende Last dienen. Diese findet in der Regel über HTTPS statt. Dazu wurde die Kommunikation zwischen den Services und den Geräten analysiert und grundlegende Abläufe identifiziert.
Ein Fahrzeug ändert seine Position einmal pro Taktzeit. Die Taktzeit ist werksabhängig und liegt in schnellen Werken bei circa 50 Sekunden. Sie beschreibt, in welchen Zeitabständen ein Fahrzeug einen Teilbereich der Linie (Takt) durchläuft. Das externe Positionssystem sendet ein Positionsupdate pro Takt und Taktzeit an PlantHUB.
Während der Produktion eines Fahrzeuges ist das verbundene VCI in der Regel konstant verbunden und online. In diesem Zustand ändert sich nur noch das Batterielevel sowie die WLAN-Stärke regelmäßig. Beides zusammen ergibt ungefähr 6 Nachrichten pro Minute von VCI an PlantHUB. Weiterhin gibt es Bereiche in der Linie, in der das VCI seine Firmware aktualisiert oder Logdaten hochlädt. Beides wird von PlantHUB über das Setzen eines Attributes im VCI ausgelöst. Das passiert weniger oft, circa alle 20 Takte.
RTS's die nicht für einen Prüfplan genutzt werden durchlaufen ungefähr alle fünf Minuten einen Reload-Zyklus. In diesem fragen sie ein externes System nach Softwareupdates und laden diese bei Bedarf herunter. Danach stehen sie wieder für die Ausführung von Prüfplänen zur Verfügung. Pro Zyklus kommt es zu fünf Änderungen im RTS. Daraus ergibt sich eine Nachricht von RTS an PlantHUB pro Minute. Um einen Prüfplan auszuführen, muss PlantHUB zunächst einen Tester auswählen und reservieren. Dazu sind vier Nachrichten von PlantHUB an das RTS notwendig. Der Tester meldet unter anderem die Reservierung, den Start, den aktuellen Stand und den Abschluss des Prüfplans an PlantHUB. Daraus ergeben sich sieben Nachrichten von RTS an PlantHUB pro Prüfplan.
Ein HMI ändert, ähnlich wie ein VCI, nur das Batterielevel und die WLAN-Stärke regelmäßig, auch ungefähr sechsmal pro Minute. Bei einem Scan eines Fahrzeugs und anschließendem Befehl zur Nacharbeit oder zum erneuten Ausführen eines Prüfplans kommen nochmal vier Nachrichten pro Scan dazu.
Die Menge bzw. Frequenz der Nachrichten ist abhängig von der Taktzeit, Anzahl der Fahrzeuge, Geräte und Prüforte. Diese können je nach Produktionsumgebung variieren, sind aber in der Regel konstant. In der Lastsimulation sollten sie deshalb konfigurierbar sein. Ein beispielhaftes Werk wird in Tabelle 3.1 dargestellt.

Parameter	Wert
Taktzeit (in Sekunden)	50
Anzahl Takte	200
Anzahl Prüforte	6
Anzahl HMI Prüforte	4
Anzahl online VCI	150
Anzahl online RTS	150
Anzahl online HMI	40

Tabelle 3.1: Beispielhafte Werksparameter

An jedem der hier angegebenen Prüforte wird pro Takt ein Prüfplan durchgeführt. Die Anzahl der HMI-Prüforte beschreibt die Menge an Prüforten, an denen ein HMI ein Fahrzeug scannt. Die Anzahl der online Geräte beschreibt die Menge an Geräten, die in der Produktion verbunden sind und Nachrichten an PlantHUB senden. Zusätzlich gibt es VCI's die mit keinem Fahrzeug verbunden und offline sind. Genauso kann es HMI's und RTS's geben die offline sind. Dann senden sie keine Nachrichten an PlantHUB und können zur Berechnung der Last ignoriert werden. In Summe ergibt das Beispiel aus Tabelle 3.1 1637 Nachrichten pro Minute.
Die Größe der Nachrichten wird in dieser Arbeit nicht betrachtet und ist nicht konfigurierbar. Die Nachrichten, die von den Geräten an PlantHUB gesendet werden, beinhalten, wie bereits erwähnt, meist nur das Batterielevel oder die WLAN-Stärke. Nachrichten zur Positionsänderung des Fahrzeugs sind leicht größer, da sie mehrere Attribute enthalten. In beiden Fällen sind die Nachrichten aber recht klein und es ist davon auszugehen, dass die Größe keinen wahrnehmbaren Einfluss auf die Reaktionszeit hat. Andere Nachrichten bei denen größere Datenmengen übertragen werden, passieren selten und werden deshalb nicht betrachtet.

3.2 Systemanforderungen

PlantHUB hat nur begrenzt Zeit, um auf ein Ereignis zu reagieren. Eine sehr hohe Reaktionszeit von PlantHUB kann dazu führen, dass ein Arbeitsschritt länger dauert als es die Taktzeit vorsieht. In diesem Fall kann es zu Produktionsverzögerungen und Staus in der Linie kommen. Die Reaktionszeit sollte daher möglichst gering sein. Ein Kunde der DSA fordert eine Reaktionszeit von unter 1000ms. Diese Zeit soll in der Lastsimulation gemessen werden. Der in Abschnitt 2.2.2 beschriebene Anwendungsfall soll dafür genutzt werden. Der zeitliche Abstand zwischen Erhalten der Information über die Positionsänderungen des Fahrzeugs und setzen des Attributes im RTS wird gemessen und dient als Indikator für die Performance des Systems.

4 Lastsimulation

Die Grundidee ist die folgende: Über einen längeren Zeitraum erzeugt die Simulation Last auf dem System. Diese Last wird durch regelmäßige Nachrichten simuliert. Die Frequenz mit der Nachrichten erzeugt werden, ist abhängig von der Anzahl der Fahrzeuge, Geräte und Prüforte. Diese Variablen werden vorher konfiguriert und sind für die gesamte Simulation konstant. In regelmäßigen Abständen wird der bereits erwähnte Anwendungsfall (siehe Abschnitt 2.2.2) ausgeführt. Die Zeitmessung findet allerdings erst später statt.
Die Lastsimulation soll die Last erzeugen, nicht aber das Verhalten der Geräte, eines Fahrzeugs oder gar des Werkes simulieren. Das bedeutet, dass die generierten Nachrichten keinen realen Ablauf im Werk widerspiegeln, sondern die Menge und Frequenz der Nachrichten simulieren. Im Folgenden werden die verwendeten Technologien und die Implementierung der Lastsimulation beschrieben.

4.1 Robot Framework

Robot Framework ist ein Open-Source Framework zur Automatisierung von Softwaretests. Es ist in Python geschrieben und zeichnet sich durch seine Lesbarkeit und benutzerfreundliche Syntax aus. Es kann mit Bibliotheken erweitert werden, die in Python geschrieben sind. Integrierte Funktionen zur Dokumentation und Zusammenfassung von Testergebnissen sorgen für eine gute Übersichtlichkeit. Eine Datei mit der Endung `.robot` ist eine ausführbare Datei und bildet den Startpunkt eines Robot Tests. Dann gibt es noch `.resource`-Dateien, in denen Robot Keywords (vergleichbar mit Funktionen aus anderen Programmiersprachen) und Variablen definiert werden. Diese können dann in der `.robot`-Datei importiert und verwendet werden. [Robb]

4.2 RCC

RCC ist ein Kommandozeilen-Tool, welches die Entwicklung, Ausführung und Verwaltung von Robot Tests ermöglicht. Es nutzt eine `conda.yaml`-Datei zur Konfiguration einer virtuellen Umgebung, in der die Tests ausgeführt werden. In Beispiel 4.1 ist eine solche Datei zu sehen. Diese enthält eine Liste von Abhängigkeiten, die in der virtuellen Umgebung installiert werden.

Listing 4.1: Conda Yaml Beispiel

```
1 channels:
2   - conda-forge
3
4 dependencies:
```

```
5    - python=3.11.0
6    - pip=23.3
7
8    - pip:
9        - robotframework==6.1.1
10       - robotframework-requests==0.9.6
11       - aiohttp==3.9.1
12       - aiomultiprocess==0.9.0
```

Das Ausführen der Tests wird in einer `robot.yaml`-Datei konfiguriert. In Beispiel 4.2 ist eine solche Datei zu sehen. Hier ist der Befehl zum Starten der `.robot`-Datei festgelegt, darüber hinaus lässt sich eine Log-Datei, das Log-Level und die auszuführende Robot-Datei festlegen. Hier wird auch der Pfad zur `conda.yml`-Datei angegeben. Mit dem Befehl `rcc run` wird die virtuelle Umgebung erstellt und die Tests darin ausgeführt. [Roba]

Listing 4.2: Robot Yaml Beispiel

```
1  tasks:
2    run loadsimulation:
3        command:
4        - python
5        - -m
6        - robot
7        - --nostatusrc
8        - --outputdir
9        - log
10       - --console
11       - verbose
12       - --loglevel
13       - DEBUG
14       - --exclude
15       - Ignore
16       - loadsimulation.robot
17
18 condaConfigFile: conda.yaml
19 artifactsDir: artifacts
```

4.3 Nebenläufigkeit in Python

Um eine große Menge von Nachrichten konstant zu erzeugen und gleichzeitig den Testfall auszuführen zu können, genügt ein einfacher, synchroner Ablauf nicht. Synchrone und damit blockierende Aufrufe würden nur einen Bruchteil der benötigten Nachrichten erzeugen. Deshalb braucht es einen parallelen Ansatz. Hier gibt es in Python mehrere, gängige Möglichkeiten.
Threads sind wohl die bekannteste Möglichkeit. Sie sind die kleinste ausführbare Einheit in einem Prozess. Sie haben eigene Register und einen eigenen Stack, teilen sich aber den Adressraum mit anderen Threads des gleichen Prozesses. Dabei kann es zu Konflikten bei gleichzeitigem Zugriff auf gemeinsame Ressourcen kommen.
Multiprocessing geht einen Schritt weiter und erzeugt mehrere Prozesse. Diese teilen sich keine Ressourcen, dadurch wird das Konfliktpotential minimiert. Allerdings macht das den Datenaustausch zwischen den Prozessen schwieriger, zudem ist das Erzeugen von Prozessen aufwändiger als

das Erzeugen von Threads.
Eine weitere Möglichkeit sind **Coroutinen**, in Python mit dem Schlüsselwort `async` gekennzeichnet. Sie sind leichtgewichtiger als Threads und Prozesse und werden nicht vom Betriebssystem verwaltet. Üblicherweise werden sie von einer `Event-Loop` aus der `asyncio` Bibliothek verwaltet. Eine `Event-Loop` in Python ist ein Objekt, welches die auszuführenden Coroutinen kennt und deren Ausführung koordiniert. In einer Endlosschleife werden die Coroutinen ausgeführt. Währenddessen können weitere Coroutinen hinzugefügt werden. Eine Coroutine kann ihre Ausführung kooperativ unterbrechen, das passiert meist bei blockierenden Aufrufen (z.B. I/O Operationen). Die Coroutine gibt dann die Kontrolle an die `Event-Loop` zurück, die dann eine andere Coroutine ausführen kann. Nach Beendigung des blockierenden Aufrufs ist die erste Coroutine wieder bereit ausgeführt zu werden. Diese kann dann in der nächsten Schleifeniteration der `Event-Loop` weiter ausgeführt werden. Dadurch können Coroutinen parallel ausgeführt werden und die Wartezeit bei blockierenden Aufrufen wird minimiert. [Pyta]

4.4 Python GIL

In Python herrscht eine besondere Situation. In der Python-Referenzimplementierung CPython gibt es einen sogenannten **Global Interpreter Lock** (GIL). Dieser sorgt dafür, dass immer nur ein Thread gleichzeitig Python-Bytecode ausführen kann. Selbst in Anwendungen mit mehreren Threads können Threads deshalb nie simultan ausgeführt werden. Das GIL wird allerdings freigegeben, wenn ein Thread einen blockierenden Aufruf macht, dann kann ein anderer Thread den GIL übernehmen und ausgeführt werden. In diesem Verhalten ähneln sich Python-Threads und Coroutinen. Aus diesem Grund ist echte Parallelität in Python nur mit Multiprocessing möglich. [Pytb]

4.5 Aufbau

Der Startpunkt der Ausführung ist eine bash-Datei. Diese löscht zunächst alle alten Logdateien und startet dann mit dem Kommando `rcc run` die Ausführung. Nach dem Erstellen der virtuellen Umgebung wird der in der `robot.yaml`-Datei definierte Befehl ausgeführt. Dieser sorgt für den Aufruf der `.robot`-Datei, in der die Simulation gestartet wird. Hier werden mehrere Keywords aufgerufen, die die Vorbedingungen, die eigentliche Simulation und die Nachbedingungen ausführen. In den Vorbedingungen wird ein Werk mit einer Linie und Takten erzeugt. In diesem Werk werden Fahrzeuge erzeugt und Tester Prüforten zugeordnet. Des Weiteren werden die VCI's, RTS's und HMI's erzeugt, die später die Last erzeugen. Danach wird der Hauptteil der Simulation gestartet. Dieser erzeugt die Last und führt regelmäßig den Testfall aus. Nach Beendigung des Hauptteils werden die Nachbedingungen ausgeführt. Diese löschen das Werk, die Fahrzeuge und die Geräte, damit der Zustand von vor der Simulation wiederhergestellt wird und keine Datenreste zurückbleiben.

4.6 Implementierung

Für die Implementierung wurde Multiprocessing in Verbindung mit Coroutinen gewählt. Um eventuelle Probleme mit dem GIL zu umgehen, werden mehrere Prozesse erzeugt, in denen die Last erzeugt wird. Innerhalb dieser Prozesse werden Coroutinen verwendet, um Wartezeiten bei blockierenden Aufrufen zu minimieren. Dadurch ist eine hohe Anzahl von Nachrichten pro Zeiteinheit möglich.
In einer `.resource`-Datei sind alle Variablen definiert, dazu gehört unter anderem der Name des Werks, das für die Simulation erzeugt wird, die Anzahl der Takte, Fahrzeuge, VCI's, RTS's und HMI's und die Dauer der Simulation. Diese Variablen werden in der `.robot`-Datei importiert und verwendet. In einer zweiten `.resource`-Datei sind alle Keywords definiert, die wichtigsten sind hier das Erzeugen und Löschen des Werks, der Fahrzeuge und der Geräte. Um zum Beispiel eine Menge von Fahrzeugen zu erzeugen, wird zuerst eine Liste von eindeutigen Nummern (VINs - Vehicle Identification Numbers) erzeugt. Für jede Nummer wird dann ein Fahrzeug über die REST-Schnittstelle des *ph-product-service* erzeugt. Auch für VCI's, RTS's und HMI's wird eine Art von ID erzeugt und dann über die REST-Schnittstelle des jeweiligen Service das Gerät erzeugt. Für das Löschen des Werkes, der Fahrzeuge und Geräte werden ebenfalls die zuvor generierten IDs und die Schnittstellen der Services verwendet.
Der Hauptteil der Simulation ist in einer Python Datei implementiert. In der Klasse `Loadsimulation` wird die Simulation gestartet und die Last erzeugt. Des Weiteren gibt es Client-Klassen für den Zugriff auf die REST-Schnittstellen der Services. Diese beinhalten Methoden die PUT oder POST Anfragen an die Services senden und damit den Zustand eines Fahrzeuges oder eines Gerätes verändern. Die Methoden sind asynchron, dadurch blockieren sie nicht und es können mehr Anfragen pro Zeiteinheit gemacht werden (siehe Coroutinen).
Um die Variablen aus der `.resource`-Datei auch in Python nutzen zu können, werden sie in der Klasse `Variables` eingelesen und gespeichert. Die Klasse `Loadsimulation` enthält die Methoden, die die Last erzeugen. Die Methoden iterieren jeweils über die Liste von Fahrzeug-, VCI-, RTS- oder HMI-IDs und rufen in einer vorher bestimmten Frequenz die entsprechende Methode der Client-Klasse auf. Damit wird der Zustand der Fahrzeuge oder Geräte regelmäßig geändert und dadurch Last generiert.
Eine Top-Level-Funktion startet die eigentliche Simulation über den Befehl `asyncio.run(Loadsimulation().run())`. Die Funktion `asyncio.run` dient als Einstiegspunkt für asynchrone Programme. Sie erzeugt eine Event-Loop, führt die übergebene Coroutine aus und beendet die Event-Loop, wenn die Coroutine terminiert. Die Methode `run` in der Klasse `Loadsimulation` startet die Lastsimulation. Hier wird zunächst eine Instanz der Klasse `Variables` erzeugt. Dann werden mehrere Prozesse erzeugt, die jeweils eine Last erzeugende Methode als Zielfunktion haben. Zusätzlich gibt es einen Prozess, der für den Testfall (siehe Abschnitt 2.2.2) zuständig ist. Die Ausführung des Testfalls findet auch über eine Methode in `Loadsimulation` statt. In dieser Methode wird der Testfall in bestimmten Zeitabständen wiederholt. Mit der gewünschten Gesamtlaufzeit und dem festgelegten Zeitabstand ergibt sich die Anzahl der Durchläufe. In jedem Durchlauf wird ein neues Fahrzeug, VCI und RTS erstellt, das RTS einem bestimmten Prüfort zugeordnet und das Fahrzeug in den Prüfort gefahren. Nach Reservierung des RTS wird das Fahrzeug, RTS und VCI wieder gelöscht, um den Ausgangszustand wiederherzustellen. Das Fahrzeug sowie RTS und VCI werden also nur für die Dauer des Testfalls erzeugt und erfahren keine Zustandsänderungen durch die Lastsimulation. Die `Variables`-Instanz wird als Parameter an die erzeugten Prozesse übergeben, damit diese auf die Variablen zugreifen können. Die Prozesse werden dann gestartet. Der Prozess der den Testfall ausführt, terminiert, wenn alle Durchläufe

abgeschlossen sind. Dann werden auch die Prozesse, die die Last erzeugen, beendet. Damit ist die Simulation beendet und die Nachbedingungen werden ausgeführt.

In Abbildung 4.1 ist ein Klassendiagramm mit den wichtigsten Klassen der Lastsimulation zu sehen. Die Klasse `Loadsimulation` enthält die Methode zum Starten der Simulation und die Methoden, die die Last erzeugen sowie die Methode zum Ausführen des Testfalls. Eine Instanz der Klasse `Variables` wird zu Beginn der Simulation erzeugt und als Parameter an die Methoden, die in einem anderen Prozess ausgeführt werden, übergeben. Sie enthält alle Variablen, die für die Simulation benötigt werden. Weiterhin gibt es vier Client-Klassen, die den Zugriff auf die jeweilige REST-Schnittstelle des *ph-product-service*, *rts-service*, *hmi-service* und *vci-service* ermöglichen. Alle vier Klassen erben von der Klasse `Client`.

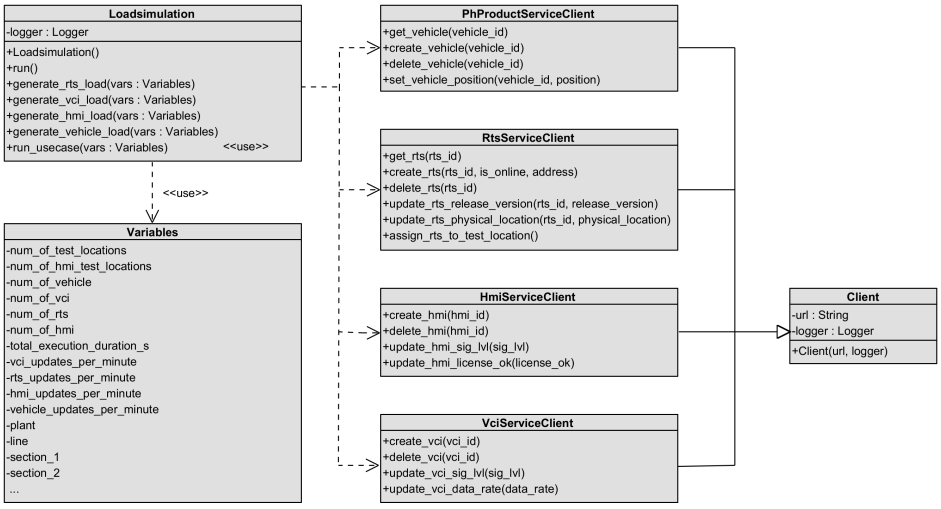

Abbildung 4.1: Klassendiagramm Lastsimulation

5 Zeitmessung

Nach Ausführung der Lastsimulation wird die Reaktionszeit von PlantHUB bestimmt. Dazu werden die während der Lastsimulation erzeugten Logs genutzt. Zunächst müssen die beiden Logs ausgewählt werden, die den Anfang und das Ende der Zeitmessung markieren. Das erste Log einer jeden Anfrage ist ein Log vor der Validierung der Daten der Anfrage, so auch in dem Fall der Anfrage an den *ph-product-service* die die Positionsänderung des Fahrzeugs enthält. Das ist der frühste Zeitpunkt, an dem PlantHUB von der Positionsänderung erfährt. Wenn der *action-trigger-agent* auf das erzeugte Event reagiert, macht er eine Anfrage an den *rts-service* um den RTS zu reservieren. Hier wird natürlich auch ein Log zu Beginn der Validierung der Anfrage erstellt. Später wird dann das Attribut im RTS gesetzt, die Änderung in der Datenbank gespeichert und kurz danach ein weiteres Log erstellt. Zu diesem Zeitpunkt ist das RTS reserviert und der Zeitpunkt des letzten Logs wird als Endzeitpunkt der Zeitmessung verwendet. Die Differenz der beiden Zeitstempel ist dann die Reaktionsdauer auf die Positionsänderung des Fahrzeugs.

5.1 Elasticsearch

Elasticsearch ist eine verteilte, dokumentenorientierte Suchmaschine, mit der sich große Mengen an Daten speichern, durchsuchen und analysieren lassen. PlantHUB nutzt Elasticsearch, um Logdaten zu speichern. Kibana dient als Visualisierungstool für Elasticsearch [Ela].
Elasticsearch bzw. Kibana ist auch auf dem PlantHUB-Server installiert. Die Logs aller Microservices werden in entsprechenden Dateien auf dem Server gespeichert. PlantHUB nutzt Kibana, um diese Logs zu visualisieren. Damit lassen sich die Logs filtern und nach bestimmten Kriterien durchsuchen. Außerdem kann man über den Trace eines Logs den Programmablauf nachvollziehen. Elasticsearch stellt auch eine API bereit, um programmatisch auf die Logdaten zugreifen zu können.
Neben den normalen Logs gibt es noch Audit-Change-Logs. Diese werden von dem *ph-product-*, *rts-*, *vci-* und *hmi-service* erstellt. Bei jeder Datenbankänderung eines Fahrzeuges, RTS, VCI oder HMI wird ein Audit-Change-Log erzeugt. Dieses Log enthält drei Objekte, ein Objekt, welches jeweils den Zustand vor und nach der Änderung beschreibt und ein Objekt, welches die Änderung beschreibt.

5.2 Implementierung

Das Python-Skript besteht aus einer Klasse mit einer Main-Methode. Im Konstruktor der Klasse wird der Start- und Endzeitstempel der Zeitmessung gesetzt. Der Startzeitstempel ist der Zeitpunkt, an dem die Lastsimulation gestartet wurde. Der Endzeitstempel ist dann der aktuelle Zeitpunkt. Außerdem wird eine Elasticsearch-Verbindung zum PlantHUB-Server aufgebaut und eine Datei

zum Speichern der Ergebnisse erstellt.
Dann wird über die Elasticsearch-API nach allen Start-Logs gesucht. Das sind die Logs, die vor der Validierung der Anfrage erstellt werden. Um nur diese Logs zu bekommen, wird nach dem genauen Inhalt der Log-Nachricht gefiltert. Dieser beinhaltet die neue Position des Fahrzeugs und kann nur in diesen Logs vorkommen, da diese Position eindeutig ist und nur während der Ausführung der Lastsimulation existiert. Danach wird nach allen End-Logs gesucht. Das sind die Audit-Change-Logs die die Reservierung des RTS markieren. Diese Logs werden nach dem eindeutigen Identifikator des RTS und dem gesetzten Attribut gefiltert. Dann werden beide Listen von Logs nach ihrem Zeitstempel sortiert. Nun wird über beide Listen iteriert und die Differenz der Zeitstempel der Logs berechnet. Mit dem Trace kann sicher gestellt werden, dass nur zusammengehörende Logs verglichen werden. Das Ergebnis ist eine Menge von Zahlen, welche die Reaktionsdauer in Millisekunden darstellen. Diese werden in die Ergebnisdatei geschrieben.

6 Durchführung und Auswertung

Die Lastsimulation kann immer nur für die Validierung einer einzelnen Kommunikationsstrecke verwendet werden und das Ergebnis kann nicht auf andere Kommunikationsstrecken übertragen werden. Der Standort des Hosts der Lastsimulation und PlantHUB-Instanz, sowie Netzwerkeigenschaften sind entscheidend für das Ergebnis der Lastsimulation, können aber je nach Fall variieren. Um zu zeigen, dass die Lastsimulation zur Validierung einer Kommunikationsstrecke geeignet ist, soll sie gegen eine bereits existierende PlantHUB-Instanz auf einem Server ausgeführt werden. Der Server befindet sich im lokalen Netzwerk der DSA. Während der Ausführung fand keine weitere Kommunikation auf dem Server statt. Die Simulation wurde mit den in Tabelle 3.1 aufgeführten Parametern durchgeführt. Zusätzlich wurde der Anwendungsfall alle 15 Sekunden ausgeführt. Die gesamte Laufzeit betrug eine Stunde. Danach wurde das Skript aus Kapitel 5 ausgeführt, um die Reaktionszeit zu ermitteln und speichern. Als Vergleich wurde die Simulation auch ohne Lasterzeugung ausgeführt. Was bedeutet, dass nur der Anwendungsfall alle 15 Sekunden für eine Dauer von einer Stunde ausgeführt wurde.

Die gemessenen Reaktionszeiten soll nun betrachtet werden. In Diagramm 6.1 sind diese in Relation zur Häufigkeit dargestellt. Die durchschnittliche Reaktionszeit beträgt 864 ms. Der Median liegt bei 743 ms. 83,75 % der Messwerte liegen unter der geforderten Grenze von 1000ms. Besonders auffällig sind die sehr hohen Messwerte, das Maximum beträgt 4699 ms, weitere neun Werte liegen über 2000 ms. Diese sollten in einer Produktionsumgebung nicht auftreten. In der Vergleichsmessung ohne Lasterzeugung sind die Messwerte ähnlich, es treten ebenfalls sehr hohe Messwerte auf. Der Grund für sporadisch hohe Messwerte ist also nicht die Last. Ein genauer Grund konnte in einer vorläufigen Analyse nicht gefunden werden.

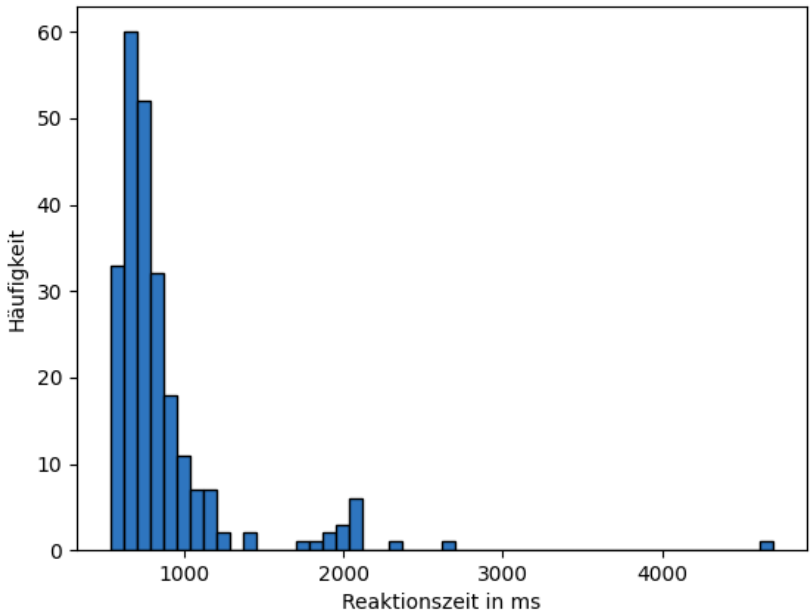

Abbildung 6.1: Reaktionszeit PlantHUB

7 Fazit und Ausblick

Im Rahmen dieser Seminararbeit wurde die Kommunikation zwischen einer Produktionsumgebung und PlantHUB analysiert und typische Parameter für eine Produktionsumgebung ermittelt. Außerdem wurde eine Lastsimulation unter Verwendung von Multiprocessing und Coroutinen entwickelt. Die in der Lastsimulation erzeugte Last basiert auf den vorher ermittelten Parametern. Neben der Lasterzeugung wird ein Standardanwendungsfall von PlantHUB simuliert. Dieser wird genutzt, um die Reaktionszeit von PlantHUB zu messen. Dazu wurde ein Skript entwickelt. Dieses berechnet die Reaktionszeit anhand der Logs, die während einer Ausführung des Standardanwendungsfalls entstehen. Diese Lastsimulation wurde erfolgreich gegen eine PlantHUB-Instanz ausgeführt und kann daher als geeignet zur Validierung einer Kommunikationsstrecke eingestuft werden. Sie hat Probleme der aktuell bestehenden PlantHUB-Instanz aufgezeigt, die in einer Produktionsumgebung nicht auftreten sollten. Nun gilt es diese Probleme genauer zu untersuchen und zu beheben.
Die Lastsimulation kann ab sofort für reale Zwecke verwendet werden. Die Lastsimulation kann und soll von Kunden der DSA verwendet werden, um die Kommunikationsstrecke aus ihrem Netzwerk zu einer PlantHUB-Instanz zu validieren. Hauptziel ist es natürlich, die Verbindung zu der Cloud mit der PlantHUB-Instanz zu validieren, die dann später auch Teil des Produktionssystems sein soll. Damit wäre eine sichere Migration in die Cloud möglich.

Die Lastsimulation kann in verschiedenen Bereichen weiterentwickelt werden. Zum einen ist vorstellbar, neben Fahrzeugen, VCI's, RTS's und HMI's, noch weitere Aspekte einer typischen Kommunikation in PlantHUB betrachten. So gibt es beispielsweise noch andere Geräte die bei der aktuellen Betrachtung aufgrund ihrer geringen Anzahl nicht berücksichtigt wurden. Es ist wahrscheinlich, dass die Anzahl der Nachrichten über einen bestimmten Zeitraum nicht konstant ist und sich die Frequenz der gesendeten Nachrichten leicht ändert und dieses Verhalten in der Lastsimulation abgebildet werden kann. Eine weitere Möglichkeit wäre, mehr Anwendungsfälle zu betrachten. Bei anderen, komplexeren Anwendungsfällen, ist es denkbar, dass mehr Nachrichten und Aktionen von PlantHUB nötig sind, und, dass sich damit die Reaktionszeit von PlantHUB ändert. Zuletzt ist es auch vorstellbar, ein anderes Nachrichtenprotokoll zu verwenden. Neben HTTPS ist auch MQTT ein von PlantHUB unterstütztes Protokoll. Für Kunden die MQTT verwenden, wäre diese Erweiterung also besonders interessant. Mit solchen Weiterentwicklungen würde die Lastsimulation realistischer werden und immer mehr eine Produktionsumgebung simulieren, als nur die reine Last.

Literaturverzeichnis

[DSA] DSA Daten- und Systemtechnik GmbH: *Unser Unternehmen*. URL: https://dsa.de/de/unternehmen.html, abgerufen am 1. Dez. 2023.

[Ela] Elastic: *Elasticsearch*. URL: https://www.elastic.co/de/elasticsearch/, abgerufen am 1. Dez. 2023.

[Pyta] Python Land: *Python Concurrency*. URL: https://python.land/python-concurrency, abgerufen am 2. Feb. 2024.

[Pytb] Python Land: *Python GIL*. URL: https://python.land/python-concurrency/the-python-gil, abgerufen am 2. Feb. 2024.

[Roba] Robocorp: *Robocorp RCC*. URL: https://robocorp.com/docs/rcc/overview, abgerufen am 1. Dez. 2023.

[Robb] Robot Framework: *Robot Framework*. URL: https://robotframework.org, abgerufen am 1. Dez. 2023.

A Tabellenverzeichnis

B Abbildungsverzeichnis

C Listingverzeichnis