MASTERARBEIT

EVALUIERUNG VON EINSATZMÖGLICHKEITEN AUSGEWÄHLTER MICROSERVICE-ARCHITEKTUREN AUF BASIS VON ARM-PROZESSOR-BASIERTEN HARDWAREKOMPONENTEN

ausgeführt am

Studiengang
Informationstechnologien und Wirtschaftsinformatik

Von: Patrick Lang
Personenkennzeichen: 1510320035

Graz, am 29. Juni 2017 .....

Unterschrift

EHRENWÖRTLICHE ERKLÄRUNG

Ich erkläre ehrenwörtlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die benutzten Quellen wörtlich zitiert sowie inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

.....

Unterschrift

DANKSAGUNG

An dieser Stelle möchte ich mich bei allen Personen bedanken, welche mich bei der Durchführung dieser Arbeit unterstützt haben. Besonderer Dank gilt hierbei meinem Betreuer, Herrn DI (FH) Günther Zwetti für seine Inspiration zu dieser Arbeit, sowie für sein außerordentlich rasches und konstruktives Feedback auf etwaige Fragen.
Weiters gilt mein Dank sämtlichen Lektorinnen und Lektoren des Campus 02, welche mir die notwendigen Grundlagen für die Durchführung dieser Arbeit vermittelt haben.
Ein weiterer Dank geht an meinen Dienstgeber, welcher mir die zur Durchführung dieser Arbeit notwendige Infrastruktur zur Verfügung gestellt hat.
Letztendlich möchte ich mich noch bei meiner gesamten Familie für die Unterstützung in den letzten Jahren bedanken. Insbesondere meiner Schwester Martina gilt hierbei besonderer Dank, da diese des Öfteren ihre Zeit für das Korrekturlesen meiner Arbeiten geopfert hat.

KURZFASSUNG

Der Einsatz ARM-Prozessor-basierter Systeme ist in den letzten Jahren durch den Aufstieg der Smartphones, sowie dem aktuellen Trend zum Internet of Things rasant angestiegen. Viele Unternehmen arbeiten daran, etablierte Systeme durch stromsparendere Varianten mit ARM-SoCs zu ersetzen.

Das Ziel dieser Arbeit ist es, die aktuelle Einsetzbarkeit dieser Systeme anhand des Beispiels des Raspberry Pi 3 Model B in Verbindung mit der Microservices Architektur zu evaluieren.

Zu diesem Zweck wird im ersten Teilbereich dieser Arbeit auf die Besonderheiten der eingesetzten Hardware und auf die technologischen Barrieren eingegangen, die den Einsatz in Unternehmen erschweren und wie diese überwunden werden können.

Aufbauend auf der zuvor erarbeiteten Umgebung wird im darauf folgenden Abschnitt auf die Besonderheiten der Microservices Architektur eingegangen. Hierbei werden Wege aufgezeigt, wie der Einsatz dieser Architektur auch auf dem Raspberry Pi in einer sicheren und redundanten Weise geschehen kann.

Der dritte Abschnitt beschäftigt sich mit der Erarbeitung eines Gesamtkonzepts und dessen prototypischer Umsetzung in Form des Hardwareaufbaus, der gewählten Softwareumgebung, sowie einer beispielhaften Dienstekomposition für den Einsatz in Unternehmen.

Als Ergebnis dieser Arbeit wird ein Konzept vorgestellt, mit welchem es möglich ist, beispielhafte Prozesse in Unternehmen zu realisieren. Zur Demonstration der praktischen Umsetzbarkeit wird abschließend ein Prototyp gezeigt, welcher als Proof of Concept dient.

Aufgrund des erfolgreichen Aufbaus dieses Konzepts wurde die Hypothese H1 bestätigt, beziehungsweise die Hypothese H0 widerlegt.

Aufbauend auf den Erkenntnissen dieser Arbeit kann in einer folgenden Arbeit der Prototyp zu einer vollwertigen Lösung, die den praktischen Einsatz ermöglicht, ausgebaut werden. Weiters können ähnliche Bereiche analysiert und gefunden werden in welchen eine Verwendung von ARM-Prozessoren mit der heutigen Technik bereits praxistauglich möglich ist.

ABSTRACT

The usage of ARM-processor-based systems has increased in recent years due to the growing number of smartphones and the internet of things. Many companies are replacing established systems with more energy-efficient alternatives based on ARM-SoCs.
The aim of this thesis is to evaluate the suitability of these current systems by investigating the option of using the Raspberry Pi 3 Model B in association with the microservices architecture.
The first chapter focusses on the characteristics of the used hardware and the technical barriers to using this technology in corporate environments. This paper then shows potential methods to overcome these barriers.
The following chapter addresses the characteristics of the microservices architecture in order to show how to use this architecture with the Raspberry Pi in a secure and redundant manner.
The development of an overall concept is shown in the third chapter. Its implementation as a prototype proof-of-concept, complete with hardware, the software environment and an ideal service composition serve as an example of how to implement this concept in a company.
The result of this study is a concept for implementing exemplary microservice processes on a Raspberry Pi in a company environment. Thanks to this successfully implemented concept, the hypothesis H1 is proven and the hypothesis H0 is disproven. Further research and tests are necessary to check the performance of this solution with more resource-intense cryptographic protocols to ensure the evolution of this prototype to a full-featured solution. In addition, a study regarding finding further fields of application for ARM processors at their current level of technology would be very interesting.

INHALTSVERZEICHNIS

1 EINLEITUNG

Vor allem durch die in den letzten Jahren im Consumer-Bereich sehr erfolgreich gewordenen Smartphones ist der Begriff der Advanced RISC Machines (ARM)-Prozessor-basierten Geräte jedem technisch interessierten Menschen ein Begriff geworden. Während in den Jahren vor dem Smartphone-Boom die Weiterentwicklung dieser Chips eher schleppend voranging hat sich dies in den letzten Jahren stark verändert. Mittlerweile sind diese stromsparenden Chips in Leistungsregionen vorgedrungen, dass Hersteller wie Cavium, Marvell und Qualcomm daran arbeiten ARM-Prozessoren nicht mehr länger nur in Consumer-Produkten, sondern vermehrt auch im professionellen Serverbereich einzubauen (Tiernan, 2014). Dieser Bereich galt bislang als Intel Monopol mit 99,2% Marktanteil im Jahr 2015 (IDC, 2016). Auch die Bemühungen seitens Microsoft einen Emulator für die Ausführung von x86-Code auf ARM-Prozessoren zu programmieren (Windeck, 2016) zeigen das Potential dieser Architektur – Näheres hierzu wird in Kapitel 2.1 beschrieben.

1.1.1 Inhalt dieser Arbeit

In dieser Arbeit geht es darum zu analysieren ob die bereits am Markt vorhandenen ARM-Computer bereits dazu in der Lage sind ausgewählte für Microservices prädestinierte Aufgaben in Unternehmen in ausreichender Geschwindigkeit zu erledigen. Die zur Veranschaulichung dieser Aufgaben gewählte Microservices-Architektur hat in den letzten Jahren enorm an Popularität gewonnen, weshalb auf diese auch näher in dieser Arbeit eingegangen wird.

Als Hardware wurde der Raspberry Pi 3 Model B als Referenzplattform ausgewählt, da dieser über die notwendige Stabilität, genügend Erweiterungsmöglichkeiten, sowie über die Kompatibilität für zukünftige Softwareentwicklungen verfügt.

> *Der Raspberry Pi behauptet sich gegenüber anderen kostengünstigen ARM-Computern vor allem durch seine gute Software-Unterstützung und die verfügbare Zusatz-Hardware. Die Stabilität der Plattform ist für viele Nutzer (gerade auch im angepeilten Bildungsbereich) der entscheidende Grund, einen Raspi einzusetzen. Entsprechend ist die Raspberry Pi Foundation sehr darauf bedacht, die Kompatibilität zu bestehender Soft- und Hardware zu erhalten.*
> *(Merten, 2016)*

Im ersten Teil dieser Arbeit wird zuerst die ARM-Architektur genauer analysiert sowie ihre Vor- und Nachteile gegenüber konventioneller Hardware angeführt. Weiters wird auf die verschiedenen Möglichkeiten eingegangen wie der Raspberry Pi im Unternehmensumfeld benutzt werden kann um zuverlässig und mit geringem Verkabelungsaufwand eingesetzt zu werden. Zusätzlich geht es in diesem Kapitel auch darum wie die Client-Hardware dazu benutzt

werden kann um mittels Radio Frequency Identification (RFID) diverse Arbeitsabläufe auszulösen, welche auf dem Prinzip der Microservices beruhen.

Der nächste Abschnitt dient dann dazu einen kurzen Überblick über die derzeit am Markt befindlichen Betriebssysteme zu geben, welche mit dem Raspberry Pi 3 Model B kompatibel sind. Hierbei wird wieder auf die jeweiligen Vor- sowie Nachteile eingegangen um die bestmögliche Lösung für den späteren Testaufbau zu finden.

Nach Erarbeitung der Grundlagen wird im dritten Teil auf die Microservices-Architektur genauer eingegangen. Hierbei wird unter anderem auch aufgezeigt wie die Verwendung mehrerer Kleinstrechner wie dem Raspberry Pi dem Konzept zu Gute kommen kann.

Die anschließenden Kapitel erläutern die unterschiedlichen Möglichkeiten der Kommunikation zwischen den beteiligten Komponenten in Unternehmensnetzwerken, sowie die dabei zu beachtenden Sicherheitsaspekte bezüglich Ausfall- und Kommunikationssicherheit.

In den letzten beiden Kapiteln wird auf die prototypische Implementierung des Gesamtkonzeptes in Form der verwendeten Datenbanken und ihrer Struktur, dem Microservices-Konzept, dem RFID-Zugriffs-Konzept, sowie dem grundlegenden Aufbau der Infrastruktur eingegangen und die gewonnenen Erkenntnisse dargestellt.

1.1.2 Forschungsfrage und Hypothesen

Die im Rahmen dieser Arbeit gestellte Forschungsfrage lautet „Wodurch wird der Einsatz von Microservices in (unterschiedlichen) Laufzeitumgebungen ARM-basierter Architekturen beschränkt?“

Zur Beantwortung dieser Frage wurden folgende Arbeitshypothesen aufgestellt:

H1: Im Gegensatz zu konventioneller Server-Hardware ist es mit ARM-Prozessor-basierten Architekturen möglich, kostengünstig und redundant Microservices zur Verfügung zu stellen, die im Vergleich zu konventioneller Hardware keine wesentlichen Performance-Nachteile aufweisen.

H0: Im Gegensatz zu konventioneller Server-Hardware ist es mit ARM-Prozessor-basierten Architekturen nicht möglich, kostengünstig und redundant Microservices zur Verfügung zu stellen, die im Vergleich zu konventioneller Hardware keine wesentlichen Performance-Nachteile aufweisen.

1.1.3 Zielsetzung

Diese Arbeit hat das Ziel eine prototypische Implementierung einer Beispielanwendung von Microservices auf einer Raspberry Pi-System-Architektur durchzuführen, welche den Anforderungen an den Einsatz in Unternehmen genügt. Diese Microservices werden beispielhaft durch die Client-Version einer Zeiterfassung sowie eines Zutrittssystems konsumiert. Anhand der in den nächsten Kapiteln erarbeiteten Anforderungen wird dabei auch auf die in Unternehmen wichtigen Sicherheits- sowie Wartungsaspekte eingegangen. Diese Implementierung soll dazu dienen das mögliche Potential des Austauschs etablierter Systeme durch kostengünstige

Alternativen aufzuzeigen. Es ist nicht Ziel dieser Arbeit ein kommerziell einsetzbares Produkt zu implementieren.

Im nächsten Kapitel geht es darum den grundlegenden Aufbau der eingesetzten Hardware in Form des Raspberry Pi mit seinen Zusatzmodulen und dem verwendeten ARM-Prozessor näher zu erklären.

2 AUFBAU DER EINGESETZTEN HARDWARE

In diesem Kapitel wird näher auf die in dieser Arbeit verwendete Hardware eingegangen. Hierbei geht es einerseits um den mit einem ARM-Prozessor ausgerüsteten Raspberry Pi sowie die für die Veranschaulichung notwendigen Zusatzmodule. Im nächsten Abschnitt wird zunächst auf die Besonderheiten und Unterschiede von ARM- und konventionellen Prozessoren eingegangen.

2.1 Unterschied ARM zu konventioneller x86/x86-64 Architektur

Der Aufbau von ARM-Prozessoren unterscheidet sich grundlegend von dem Aufbau konventioneller x86- beziehungsweise x86-64-Architekturen, welche heutzutage meist als Client/Server-Architektur im Einsatz sind. In diesem Kapitel geht es darum die Unterschiede zwischen den einzelnen Architekturen sowie ihre Vor- und Nachteile aufzuzeigen.

2.1.1 Aufbau konventioneller Prozessoren

Konventionelle Prozessoren, die über den sogenannten x86-Befehlssatz (oft auch IA-32 oder i386 genannt) verfügen, werden heute meist von Intel oder AMD geliefert. Die zugehörige 64-bit fähige Architektur wird x64 oder auch x86-64 genannt. Der erste Mikroprozessor in x86 Architektur wurde von Intel im Jahr 1978 in Form des 8086 auf den Markt gebracht und drei Jahre später durch IBM in Form des 8088 in Consumer-Produkte verbaut. Die Architektur war so erfolgreich, dass sie bis heute in modernisierter Form im Einsatz ist. Wobei sämtlicher Code, welcher für den 8086 geschrieben wurde theoretisch auch noch auf den modernen Prozessoren ausgeführt werden kann, da hierbei eine Abwärtskompatibilität vorliegt (Das, 2010, S. 2-5).

Die x86 sowie x86-64 Architektur beruht auf der Complex Instruction Set Computer (CISC) Architektur, welche erst nach der Erfindung der in Kapitel 2.1.2 beschriebenen Reduced Instruction Set Computer (RISC) Architektur von IBM so benannt wurde um eine Abgrenzung voneinander zu ermöglichen. Die CISC Architektur beruht auf dem Konzept von komplexen Befehlen (Schiffmann, 2006, S. 137-140). Im Vergleich zur RISC Architektur gibt es hierbei auch eine weitaus größere Auswahl an unterschiedlichen Befehlen, welche jeweils mächtiger sind als ihre Pendants in der RISC Architektur. Das Gesamtkonzept des CISC beruht darauf komplexe Befehle direkt in Form von Hardwarekomponenten abzubilden. Die Komplexität des Ablaufs wird somit vom Programmierer versteckt. Dies ist vergleichbar mit der Art und Weise wie moderne Hochsprachen in Maschinensprache übersetzt werden. Der Compiler, welcher für diese Übersetzung zuständig ist, hat bei der CISC Architektur somit weniger Arbeit, da er Hochsprachen-Befehle im Prinzip ohne größere Änderungen übernehmen kann (Chen, Novick, & Shimano, 2006).

Abbildung 1 zeigt das grundlegende Speicherschema eines CISC. Man erkennt die Unterteilung des Hauptspeichers in Zeilen/Spalten sowie die einzelnen Register, welche dazu benötigt werden um Operationen durchführen zu können. Möchte man beispielhaft den Befehl „MUL [A:1, B:3]“

ausführen, der für die Multiplikation der in den jeweiligen Feldern enthaltenen Werte steht, müssen zuerst die einzelnen Werte in die Register (in diesem Fall A bis F) geladen werden. Im Anschluss kann die Recheneinheit den gewünschten arithmetischen Befehl der Multiplikation ausführen und das Ergebnis zum Beispiel in Zelle C:2 zurückschreiben (Stokes, 1999). Man erkennt an diesem Beispiel, dass der Befehl für die Operation selbst zwar kurz, jedoch relativ komplex ist. Die CISC Architektur sorgt dafür, dass dieser komplexe Befehl im Hintergrund jedoch so abgearbeitet wird wie soeben beschrieben und der Programmierer sich somit nicht um das Laden in die Register kümmern muss. Aufgrund des komplexen Aufbaus der Architektur benötigt der CISC hierfür jedoch im Vergleich zum RISC mehr Taktzyklen (Coy, 2013, S. 144). Weshalb die RISC Architektur hierbei so viel schneller arbeiten kann wird im Kapitel 2.1.2 genauer erläutert.

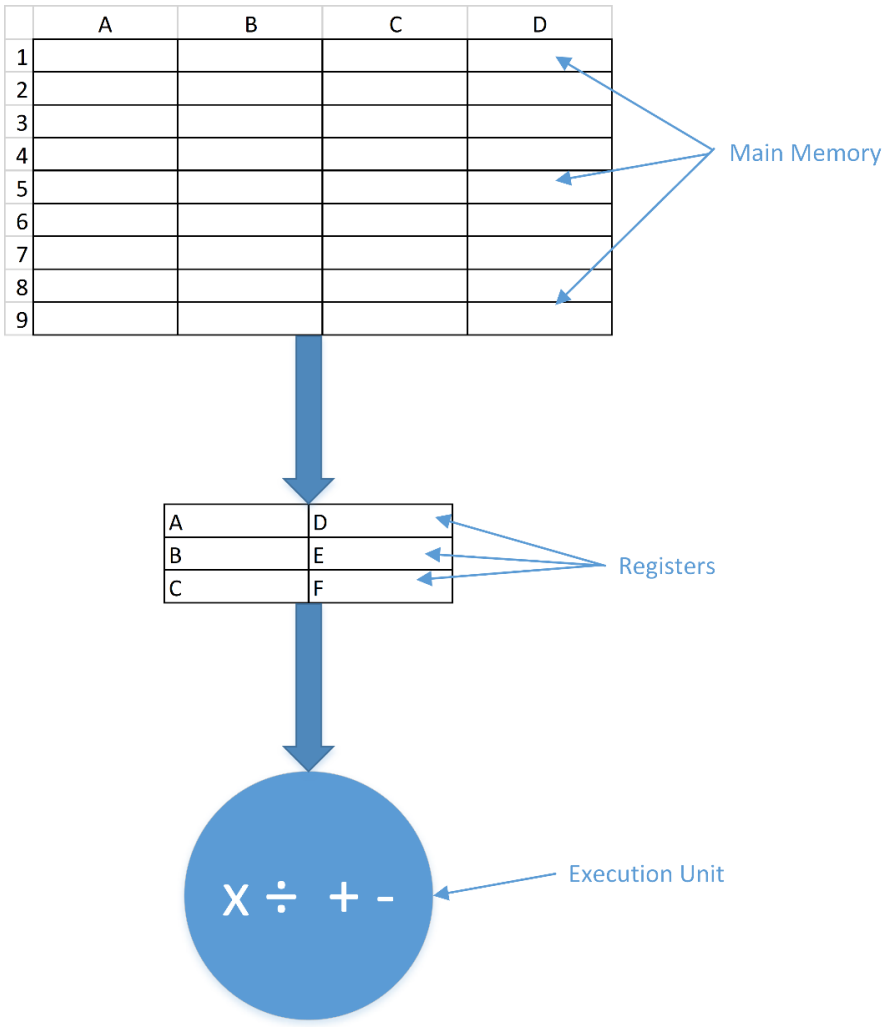

Abbildung 1: Speicherschema CISC Architektur (Stokes, 1999)

Moderne x86-Prozessoren sind heutzutage eine Mischung aus CISC und RISC. Hierbei werden die komplexen CISC-Befehle zuerst in einfachere RISC-Befehle übersetzt und anschließend von der hierfür eingebauten RISC Architektur ausgeführt, sodass man die Vorteile beider Welten vereint (Thompson & Thompson, 2003, S. 154).

Das nächste Kapitel handelt nun davon wie sich der ARM-Prozessor vom soeben beschriebenen Konzept unterscheidet und welche Vorteile dadurch existieren.

2.1.2 Aufbau von ARM-Prozessoren

Die heutzutage meist in mobilen und stromsparenden Geräten verwendeten ARM-Prozessoren haben einen grundlegend anderen Aufbau als die im Kapitel 2.1.1 beschriebenen konventionellen Prozessoren, da sie für vollkommen unterschiedliche Aufgabengebiete konzipiert wurden. Während ARM-Prozessoren daraufhin optimiert wurden so wenig Strom wie möglich zu verbrauchen um ihre Aufgaben zu erledigen (Yiu, 2011, S. 198-199) ging es bei der x86-Architektur mehr darum so viel Leistung wie möglich zur Verfügung zu stellen – der Stromverbrauch ist zwar ebenfalls relevant, jedoch nicht das wichtigste Kriterium für die Entwicklung, da diese Prozessoren meist in Geräten zum Einsatz kommen, welche mit einer dauerhaften Stromversorgung ausgestattet sind (Dear, 2014).

ARM-Prozessoren, welche heutzutage jedoch in den unterschiedlichsten Geräten zu finden sind, müssen dafür sorgen, dass zum Beispiel eine Smartphone-Batterie nicht bereits nach wenigen Stunden leer ist oder ein sogenanntes Internet of Things (IoT)-Device nicht vor dieselben Probleme gestellt wird. Im Gegensatz zu Computern für den Heimgebrauch, welche standardmäßig nur bei Benutzung eingeschaltet werden, arbeiten diese IoT-Geräte im Normalfall 24 Stunden am Tag und 7 Tage in der Woche um Daten zu sammeln oder um mit anderen Geräten zu kommunizieren. Außerdem gehen viele Größen der Wirtschaft davon aus, dass die Anzahl der IoT-Geräte in jedem Haushalt, welche rund um die Uhr aktiv sein werden, in den nächsten Jahren stark steigen wird.

Eine Vorhersage der Firma Ericsson geht zum Beispiel davon aus, dass die Anzahl der IoT-Geräte in den Jahren zwischen 2015 und 2021 jährlich um 23% steigen wird und im Jahr 2021 eine Anzahl von 28 Milliarden mit dem Internet verbundenen Geräten erreichen wird (Ericsson, 2016). Die Analysten von Research and Markets gehen weiters davon aus, dass der Umsatz mit IoT-Geräten von 157,05 Milliarden US Dollar im Jahr 2016 auf 661,74 Milliarden US Dollar im Jahr 2021 steigen wird, was einem jährlichen Wachstum von 33,3% entsprechen würde (Research and Markets, 2016). Ein hoher Stromverbrauch dieser Geräte würde ihre Akzeptanz und die möglichen Einsatzzwecke jedoch stark minimieren. Die International Energy Agency hat im Jahr 2013 festgestellt, dass die zum damaligen Zeitpunkt 14 Milliarden IoT-Geräte für 616 Terawattstunden an Stromverbrauch verantwortlich waren. Zirka 400 Terawattstunden davon wurden jedoch verschwendet, da die eingesetzten Geräte nicht über die notwendigen Stromsparfunktionen verfügten, wie sie zum Beispiel bei ARM-Prozessoren eingebaut sind (International Energy Agency, 2014). Somit herrscht hier enormes Einsparungspotential, wenn man auf sparsamere Architekturen umsteigen würde, welche im Standby-Betrieb weniger Strom verbrauchen als konventionelle Lösungen.

Weiters ist bei ARM-Prozessoren laut Böttcher (Rechneraufbau und Rechnerarchitektur, 2007) neben der begrenzten Energieaufnahme und den niedrig zu haltenden Kosten auch oft der Platz im Gehäuse von Geräten stark begrenzt, was dazu führt, dass auch hier auf die Größe der einzelnen Bauteile geachtet werden muss. Böttcher beschreibt zum Beispiel Chipkarten, welche kleine Prozessoren beinhalten als mögliches Szenario – siehe hierzu Kapitel 2.3, das solche Chipkarten näher beschreibt.

Der ARM-Prozessor selbst beruht auf dem Prinzip des RISC, welcher maßgeblich von Patterson & Sequin (RISC I: A Reduced Instruction Set VLSI Computer, 1981) als Reaktion auf die immer komplexer werdende Architektur des CISC entwickelt wurde. Die beiden Entwickler hatten sich zum Ziel gesetzt die Entwicklungszeit und die Entwicklungsfehler (aufgrund von bisher sehr komplexen Befehlen) zu minimieren, sowie die Ausführungszeit von einzelnen Befehlen signifikant zu verkürzen. Die Notwendigkeit hierfür sahen sie deshalb gegeben, weil die CISC Architektur im Laufe der Jahre je nach Anforderung um weitere Hardware-Bestandteilte erweitert wurde um aufwändigere Befehle innerhalb eines Taktzyklus abarbeiten zu können. Dies führte dazu, dass die Architektur immer komplexer wurde und es somit schwieriger wurde die Technik weiterzuentwickeln.

Patterson & Sequin (RISC I: A Reduced Instruction Set VLSI Computer, 1981) haben sich diese Probleme angesehen und sind zum Schluss gekommen, dass es eine neue, einfachere Architektur benötigt, sodass Weiterentwicklungen wieder einfacher möglich sind. Dabei haben sie den RISC aufgrund folgender eigener Vorgaben konstruiert:

- Abarbeitung einer Anweisung innerhalb eines Taktzyklus – dies ist nicht bei allen Anweisungen möglich (zum Beispiel Lade- und Speicher-Anweisungen benötigen 2 Taktzyklen)
- Alle Anweisungen müssen die gleiche Größe haben (meist 32 Bit) – dies ermöglicht eine Vereinfachung der Programmentwicklung
- Lade- und Speicher-Anweisungen sind die einzigen Anweisungen, welche auf den Speicher zugreifen dürfen – alle anderen Anweisungen können nur mit dafür vorgesehenen Registern arbeiten. Dies vereinfacht das Design der Architektur.
- Unterstützung von Hochsprachen, sogenannten High-Level Languages (HLL) – dient unter anderem dazu die oben angeführten Entwicklungsfehler zu minimieren.

In Verbindung mit der RISC Architektur hat sich der Begriff „Superskalarität“ etabliert, welcher für die gleichzeitige Ausführung mehrerer Befehle auf unterschiedlichen Recheneinheiten steht. Dadurch ist es möglich Programme, welche aufgrund ihrer höheren Anzahl von Befehlen mehrere Taktzyklen zur Beendigung benötigen würden trotzdem innerhalb weniger Taktzyklen abzuarbeiten. Aufgrund des zuvor beschriebenen einfacheren Aufbaus der RISC Architektur ist somit die Superskalarität ein integraler Bestandteil dieser Architektur. Durch die fixe Verdrahtung einfacher Befehle in der RISC Architektur fällt es zudem leichter die Taktfrequenz zu erhöhen, was positive Effekte auf die letztendliche Ausführungsdauer hat (Mueller & Soper, 2001).

Der große Unterschied zwischen der CISC und RISC Architektur wird in Abbildung 2 anhand des dargestellten beispielhaften Ablaufs des Vorgangs einer Türöffnung veranschaulicht. Hierbei ist zu sehen, dass bei der CISC Architektur bedeutend weniger Befehle vorhanden sind, jeder Befehl für sich ist jedoch von hoher Komplexität. Ein Prozessor benötigt hierfür mehrere Taktzyklen um ihn abzuarbeiten. Im Vergleich dazu hat der Ablauf in der RISC Architektur mehr als doppelt so viele Befehle, die es abzuarbeiten gilt. Jeder einzelne dieser Befehle ist jedoch von niedriger Komplexität und kann somit innerhalb eines Taktzyklus abgearbeitet werden, da diese wenig komplexen Befehle fix verdrahtet in der Hardware aufgebaut sind. Letztendlich kann die RISC

Architektur somit den Vorgang in kürzerer Zeit abarbeiten als die CISC Architektur, da diese für die Abarbeitung nicht optimiert ist und viel mehr Overhead bei jedem einzelnen Befehlsschritt zu bearbeiten hat. Das angeführte Beispiel ist angelehnt an ein von Mueller & Scoper (Microprocessor Types and Specifications, 2001) veranschaulichtes Beispiel.

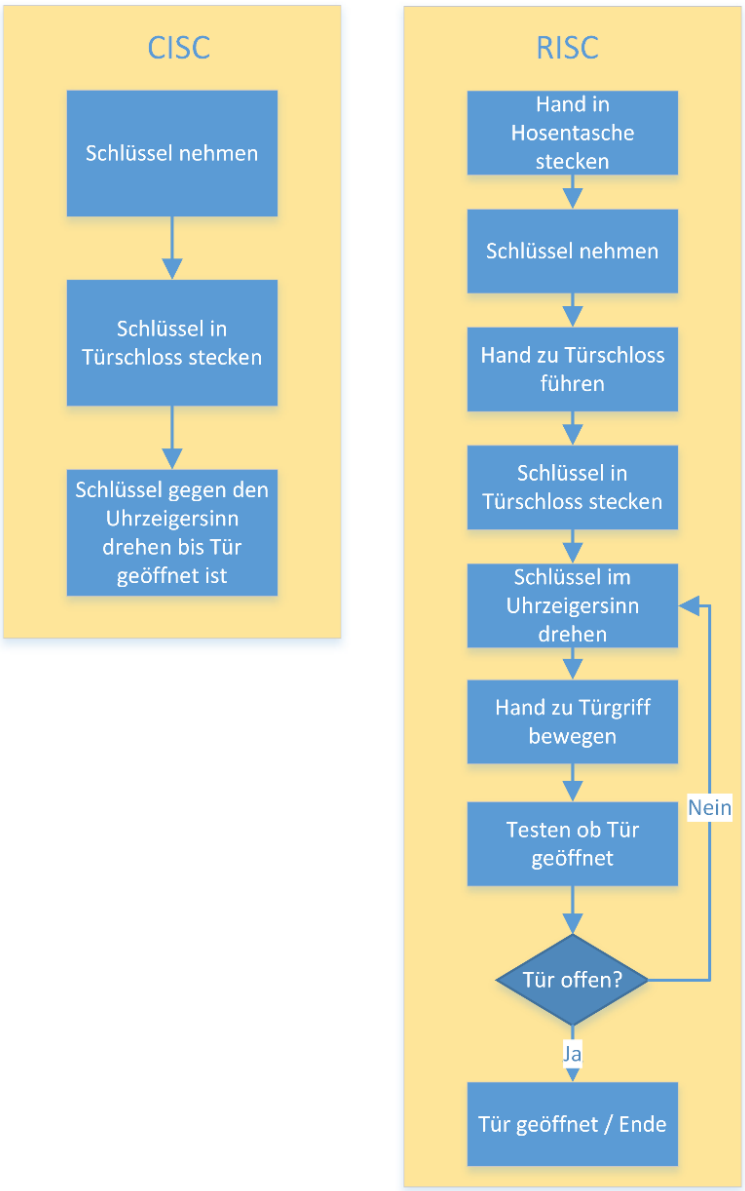

Abbildung 2: Beispielablauf CISC/RISC Architektur – angelehnt an Mueller & Scoper (Microprocessor Types and Specifications, 2001)

Auch aufgrund des enormen Geschwindigkeitsgewinns sind moderne CISC-Prozessoren heutzutage meist Hybrid-Varianten, welche die Vorteile der RISC-Architektur benutzen um gezielt Rechenabläufe zu beschleunigen. Hierbei werden kompatible CISC Befehle zu RISC Befehlen übersetzt und dann durch die hierfür optimierte Hardware abgearbeitet (Mueller & Soper, 2001).

Die Art des in dieser Arbeit eingesetzten ARM-Prozessors wird auch System-on-a-Chip (SOC) genannt, da sämtliche Komponenten auf diesem einen hochintegrierten Chip untergebracht wurden. Die Alternative hierzu – also ein Prozessor, welcher noch auf externe Peripherie angewiesen ist – wird Mikro-Controller genannt (Böttcher, 2007).

Abbildung 3 zeigt den schematischen Aufbau des BCM2837 SOC, welcher im Raspberry Pi 3 eingesetzt wird. Hierbei ist zu beachten, dass ein großer Teil des Chips vom sogenannten VideoCore belegt ist. Dieser beinhaltet laut Merten (Überreife Himbeere: Wie es mit dem Raspberry Pi weitergeht, 2016) unter anderem die sogenannte Vector Processing Unit (VPU), die ähnlich wie der ARM-Prozessor selbst mit RISC-artigen Befehlen angesprochen wird und welcher über einen eigenen Bootloader verfügt, der beim Startvorgang zuerst initialisiert wird. Dies alles geschieht bevor die vom Benutzer festgelegte Bootmöglichkeit angesprochen wird – siehe hierzu Kapitel 2.6.

Genau genommen läuft auf der VPU ein eigenes kleines Betriebssystem, welches dafür zuständig ist die erste Stufe des Bootloaders zu laden, der den Bootvorgang des Raspberry Pi startet (Upton, Duntemann, Roberts, Mamtora, & Everard, 2016).

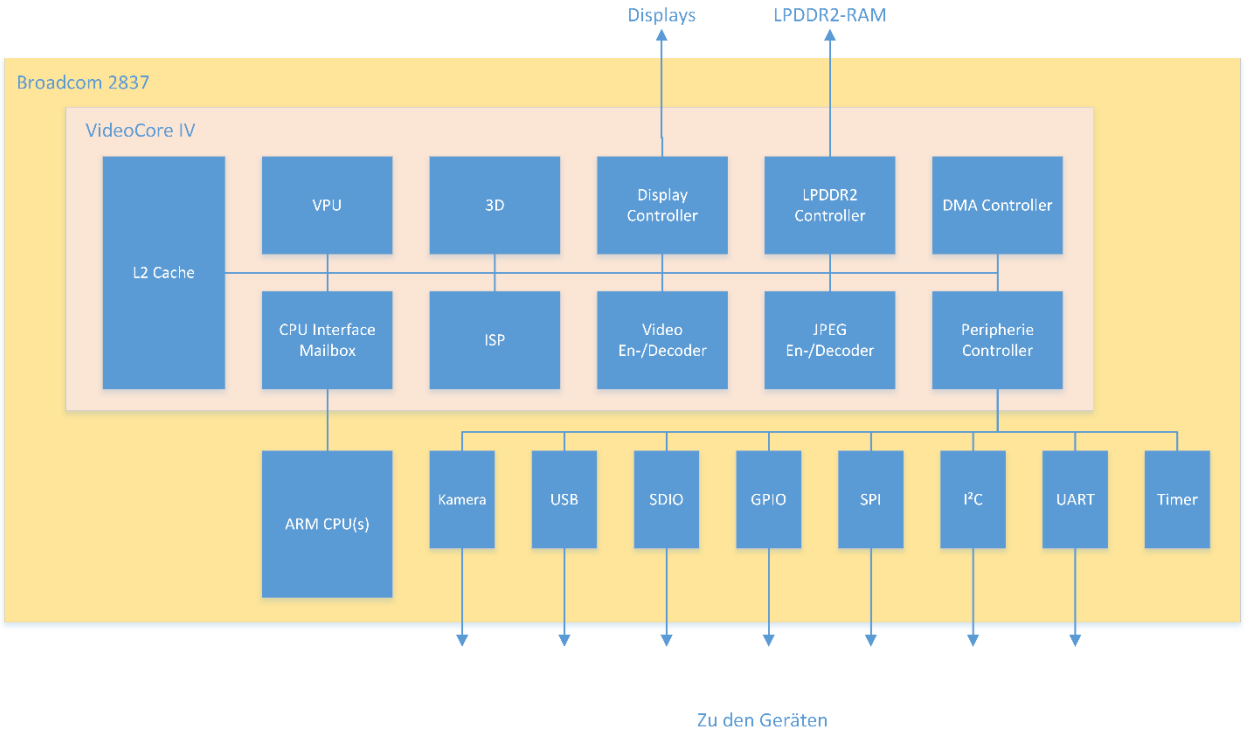

Abbildung 3: Schematischer Aufbau von Broadcom 2837 (Merten, 2016)

Im nächsten Abschnitt wird nun näher auf den für diese Arbeit verwendeten Mini-Computer Raspberry Pi eingegangen, welcher den hier beschriebenen ARM-Prozessor als Recheneinheit an Board hat.

2.2 Raspberry Pi

Der Raspberry Pi ist ein Einplatinencomputer, der in seiner ersten Version im Jahr 2012 auf den Markt gekommen ist. Sein ursprünglicher Zweck war es vor allem Kindern in Entwicklungsländern eine günstige Möglichkeit zu bieten das Programmieren zu lernen. Aufgrund des einfachen Aufbaus und der Verwendung von Standardkomponenten konnte die Raspberry Pi Foundation den Verkaufspreis sehr niedrig halten – was bei dem kleinen Computer zu unerwarteten

Verkaufszahlen führte. Von ursprünglich 10.000 Vorbestellungen sind 2 Jahre später 3 Millionen Stück, der bis heute insgesamt bereits 7 verschiedenen Modelle, verkauft worden (Upton L. , 2014). Im Jahr 2016 wurde bereits die 10 Millionen Grenze überschritten (Upton E. , Ten millionth Raspberry Pi, and a new kit, 2016).

Der Raspberry Pi 3 Model B, welcher im Verlauf dieser Arbeit eingesetzt wurde, ist zum Zeitpunkt dieser Arbeit die aktuellste Revision dieses Computers. Ausschlaggebend für die Verwendung dieses Modells waren die vergleichsweise geringen Kosten, die öffentlich zugänglichen Dokumentationen, sowie die Möglichkeit die Geräte bei möglichen Defekten innerhalb weniger Minuten auszutauschen. Außerdem wurde bereits in der Vergangenheit durch diverse unterschiedliche Projekte bewiesen wie leistungsfähig diese Kleinstcomputer sind und welche teils überraschenden Projekte hiermit realisierbar sind. Als Beispiel für eine solche Lösung sei hier die Docker Challenge aus dem Jahr 2015 erwähnt. Hierbei wurde ein Wettbewerb abgehalten die größte Anzahl von Docker Instanzen eines Webservers auf einem Raspberry Pi 2 Model B auszuführen. Näheres zu Docker wird in Kapitel 4.2 erläutert.

Letztendlich schaffte man es bei dem abgehaltenen Wettbewerb 2.334 Webserver in Docker-Instanzen auf einem einzelnen Raspberry Pi 2 Model B zu betreiben und anzusprechen (Herzog, 2015). Tabelle 1 zeigt die Leistungsdaten der beiden Raspberry Pi Modelle und dient als Anhaltspunkt dafür, dass das in dieser Arbeit verwendete Modell dem im Wettbewerb verwendeten Modell technisch überlegen sein müsste und somit mehr als genug Leistung zur Verfügung steht um eine Umsetzung von Microservices darauf zu testen.

		Raspberry Pi 2 Model B	**Raspberry Pi 3 Model B**
System on a Chip (SOC):		BCM2836	BCM2837
Central Processing Unit (CPU)	**Typ**	ARM Cortex-A7	ARM Cortex-A53
	Anzahl Kerne	4	4
	Takt (MHz)	900	1200
	ARM Architektur	32-bit (v7)	64-bit (v8)
Graphics Processing Unit (GPU)-Takt (MHz)		250	300/400
Arbeitsspeicher		1024	1024
Netzwerk		10/100-Mbit	10/100-Mbit
WLAN		-	2,4GHz WLAN b/g/n (BCM43143)
Bluetooth		-	Bluetooth 4.1 Low Energy
Pins		40	40
General Purpose Input/Output (GPIO)-Pins		26	26
Zusätzliche Schnittstellen		je 1 Mal CSI / DSI / I²S	je 1 Mal CSI / DSI / I²S

Tabelle 1: Raspberry Pi 2 / 3 Model B Vergleich (Raspberry Pi Foundation, 2015) & (Raspberry Pi Foundation, 2016)

Der im Raspberry Pi 3 eingebaute BCM2837 SOC beruht auf der RISC/ARM Architektur, welche im Kapitel 2.1.2 bereits näher beschrieben wurde. Der verwendete ARM Prozessor Cortex-A53 besitzt 4 Prozessorkerne, ist mit 1,2 GHz getaktet und beruht auf der ARMv8 Architektur, welche dafür sorgt, dass 64-bit Befehle ausgeführt werden können. Obwohl mit dem Übergang von 32

auf 64 bit prinzipiell die technische Hürde fallen würde, dass nur maximal 4 GB Random Access Memory (RAM) in den Geräten eingebaut werden können befindet sich beim Raspberry Pi die relevante Einschränkung an ganz anderer Stelle und diese sorgt dafür, dass die Grenze nicht bei 4 GB RAM, sondern bei den derzeit bereits verbauten 1 GB RAM liegt. Wie bereits in Abbildung 3 ersichtlich war befindet sich auf dem SOC neben dem ARM-Prozessor auch eine VPU. Diese VPU ist das erste Modul, welches beim Bootvorgang des Raspberry Pi angesprochen wird und ist laut Merten (Überreife Himbeere: Wie es mit dem Raspberry Pi weitergeht, 2016) das problematische Element in der Weiterentwicklung der Plattform sowie das Hindernis bei der Erhöhung des RAM. Der nächste Abschnitt dient dazu den Bootvorgang des Raspberry Pi besser zu verstehen, um somit die Problematik bei der RAM-Erweiterung zu erkennen.

Merten (Überreife Himbeere: Wie es mit dem Raspberry Pi weitergeht, 2016) beschreibt den Bootvorgang des Raspberry Pi in folgenden Schritten:

- Im ersten Schritt startet die VPU den ersten Bootloader, welcher sich auf dem Read Only Memory (ROM) des SOC befindet – hierbei befinden sich der ARM-Prozessor sowie der RAM weiterhin im Ruhezustand und werden nicht angesprochen
- Im zweiten Schritt wird ein weiterer Bootloader geladen, welcher sich auf dem gewählten Bootmedium befindet – Näheres hierzu wird in Kapitel 2.6 beschrieben. Dieser Bootloader initialisiert den RAM und lädt die Firmware für den gesamten VideoCore IV.
- Im dritten Schritt lädt der VideoCore IV einen dritten Bootloader, welcher letztendlich den RAM zwischen dem VideoCore IV selbst, sowie dem ARM-Prozessor aufteilt und die ARM-Prozessorkerne startet.
- Im letzten Schritt wird nun das eigentliche Betriebssystem vom gewünschten Bootmedium gestartet – Näheres hierzu siehe Kapitel 2.7.

Anhand des schematischen Aufbaus sowie der Beschreibung des Bootvorganges erkennt man nun, dass der ARM-Prozessor selbst nur über die CPU Interface Mailbox mit dem VideoCore IV sowie dem RAM kommuniziert. Das Problem an dieser Vorgehensweise ist jedoch, dass der VideoCore IV nicht mehr weiterentwickelt wird und technisch von ihm maximal 1 GB RAM angesprochen werden kann (Merten, 2016). Möchte man diese Grenze überschreiten und die 64-bit Architektur des BCM2837 zur Gänze ausnutzen ist es notwendig den Aufbau sowie den Bootvorgang des Chips gänzlich zu überarbeiten. Dies könnte jedoch dazu führen, dass die Nachfolgegeneration des Raspberry Pi nicht mehr mit seinen Vorgängern kompatibel ist. Diese Abwärtskompatibilität war einer der Gründe weshalb man auch beim Raspberry Pi 3 den Aufbau wie in der Vorgängergeneration belassen hat (Alasdair, 2016).

Nachdem die grundlegende Hardware nun definiert ist, wird im nächsten Abschnitt auf das für die Clients benötigte Radio Frequency Identification (RFID-) Modul eingegangen, welches dazu dient sogenannte MIFARE Classic Karten auszulesen. Diese werden in weiterer Folge dazu verwendet die beispielhaften Prozesse der Client-Software anzustoßen – Näheres hierzu siehe Kapitel 4.3.

2.3 Radio Frequency Identification Modul

Als Client dient in dieser Arbeit ein Raspberry Pi, welcher zur Interaktion mit dem Benutzer auf ein RFID-Modul zurückgreift. Viele Unternehmen verwenden heutzutage solche RFID-Chips um ihre Mitarbeiter eindeutig zu identifizieren. Dies kann dazu verwendet werden Mitarbeitern und Mitarbeiterinnen Zutritt zu bestimmten Geschäftsbereichen zu geben oder um eine Zeiterfassung durchzuführen. Andere Anwendungsfälle sind die Nachverfolgung von Produkten in Lieferketten oder die Durchführung von Bezahlvorgängen (Glover & Bhatt, 2006).

Bei RFID gibt es diverse unterschiedliche Typen. Abhängig von ihrem Einsatzgebiet gibt es außerdem unterschiedlich große Identifikationsmedien – diese können von wenigen Millimetern (zum Beispiel zur Verwendung in kleineren Produkten oder zur Verpflanzung unter die menschliche Haut) bis zu einigen Dutzend Zentimetern (zum Beispiel als sichtbare Ausweiskarte) groß sein. Die für den Testaufbau verwendete Lösung ist in Abbildung 4 zu sehen. Es handelt sich hierbei um eine passive Transponderkarte, deren integrierter RFID-Chip mit Hilfe der Induktionsspulen vom Lesegerät mit Strom versorgt wird. Über dieses elektromagnetische Feld können somit nicht nur Informationen ausgetauscht, sondern auch in begrenztem Maß Energie übertragen werden (Werner, 2008). Dieses passive System wird laut Walk, et al. (RFID-Standardisierung im Überblick, 2012) auch „Reader Talks First“-Protokoll genannt, da die Initialisierung der Kommunikation vom Lesegerät ausgeht.

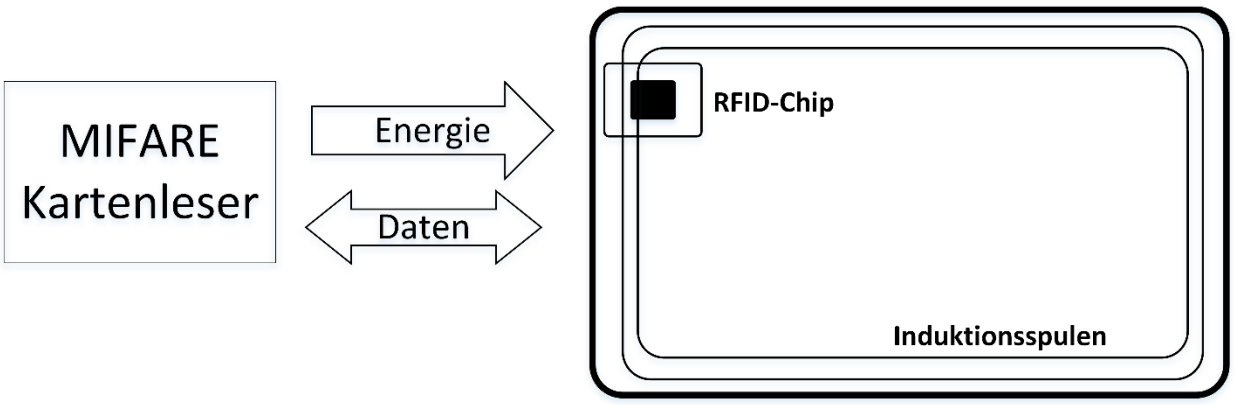

Abbildung 4: Aufbau einer Mifare Karte (NXP, 2011)

In dieser Arbeit wird auf die MIFARE Classic Technologie zurückgegriffen. Neben dieser gibt es noch sehr viele andere Technologien, wie zum Beispiel Hitag oder Legic, welche aber immer auf ähnlichen Konzepten beruhen. Da es bei dieser Arbeit jedoch nicht darum geht die beste RFID-Lösung zu finden, wurde aufgrund mehrerer passender Eigenschaften diese Technologie als beispielhafte Möglichkeit ausgewählt. Diese Eigenschaften werden in den nächsten Abschnitten genauer beschrieben. Für hochsichere Daten sollte die MIFARE Classic Technologie jedoch nicht mehr verwendet werden, da diese Sicherheitslücken enthält, welche dazu genutzt werden können um auf die darauf enthaltenen Daten zuzugreifen ohne die hierfür eigentlich benötigten Schlüssel

zu kennen (Langer & Roland, 2010). Hierbei wäre es ratsam auf die neuere abwärtskompatible Technologie MIFARE Plus S oder MIFARE Plus X zurückzugreifen, welche die ausnutzbaren Sicherheitslücken eliminiert und die Kommunikationssicherheit weiter verbessert haben (NXP, 2011).

Prinzipiell gibt es bei der Identifizierung von MIFARE Classic Transponderkarten zwei verschiedene Möglichkeiten (Bichler, Riedel, & Schöppach, 2013):

- Verwendung des eindeutigen Unique Identifier (UID) – jede MIFARE Classic Transponderkarte verfügt über eine weltweit eindeutige Nummer, welche von den entsprechenden Lesern ausgelesen werden kann. Diese Nummer ist jedoch öffentlich zugänglich und kann somit ohne Wissen eines geheimen Schlüssels ausgelesen werden. Dieses Verfahren wird als Data-on-Network bezeichnet, da sich die eigentlich zu verwendenden Daten in einer Datenbank im Netzwerk befinden.
- Die zu verwendenden Daten werden in einem dafür vorgesehenen Speicherbereich auf der Karte selbst hinterlegt – dies wird als Data-on-Tag bezeichnet. Dies hat den Vorteil, dass die Daten unabhängig von zentraler Serverinfrastruktur gelesen werden können.

Eine MIFARE Classic Transponderkarte hat je nach Variante (Mini, 1K, 4K) unterschiedlich große Datenbereiche, welche beschrieben werden können. Diese reichen von 320 Byte bis zu 4 Kilobyte (Langer & Roland, 2010). Der grundlegende Vorgang des Auslesens ist jedoch unabhängig von den Datenbereichen immer derselbe.

Wenn man als Beispiel die MIFARE Classic 4k Transponderkarte heranzieht, besitzt diese insgesamt 40 Blöcke in denen man Informationen speichern kann. Jeder dieser Blöcke kann mit einem eigenen Schreib- und Lese-Schlüssel gesichert werden. Diese beiden Schlüssel sind jedoch unabhängig voneinander. Es handelt sich hierbei also um kein asymmetrisches kryptographisches Verfahren, sondern um ein symmetrisches. Näheres hierzu wird in Kapitel 4.1 erläutert.

Die Daten, welche man auf dem Chip ablegt kann man somit vor Veränderung schützen indem man den Schreibschlüssel geheim hält. Weiters kann man die Daten auch vor dem Auslesen schützen indem man den Leseschlüssel geheim hält. Natürlich muss dieser jedoch für das Auslesen im hierfür vorgesehenen Lesegerät hinterlegt sein.

Im nächsten Kapitel wird nun auf eine Möglichkeit eingegangen wie man die Verkabelung und auch die Fernwartung des Raspberry Pi vereinfachen kann.

2.4 Power-over-Ethernet Modul

Der Raspberry Pi verfügt dank seiner GPIO-Ports über diverse Erweiterungsmöglichkeiten. Neben der Möglichkeit externe Hardware damit einzulesen oder anzusteuern kann man hiermit auch den Funktionsumfang des Geräts selbst erweitern und mögliche Unzulänglichkeiten für spezifische Einsatzgebiete ausmerzen. Für den Einsatzzweck, dem der Raspberry Pi in dieser Arbeit dient, ist es von Vorteil, wenn man das Gerät aus der Ferne fernsteuern kann, da sonst ein produktiver Einsatz in größeren Umgebungen kaum sinnvoll wäre. Da der Raspberry Pi standardmäßig nur durch das Trennen und Neuverbinden der Stromversorgung nach einem Crash neu gestartet werden kann ist es somit notwendig eine Möglichkeit zu finden, wie diese Stromversorgung aus der Ferne zurückgesetzt werden kann. Hierfür bietet sich ein Power-over-Ethernet (PoE) Modul an. Dieses sorgt dafür, dass das Gerät direkt über den LAN-Anschluss mit Strom versorgt wird und man somit aus der Ferne – vorausgesetzt es wird ein kompatibler Netzwerk-Switch eingesetzt – diesen Reset durchführen kann.

Zusätzlich hat man hiermit den Vorteil, dass nur ein einzelnes Kabel zum späteren Einsatzort gezogen werden muss und man sich somit mögliche Sicherungen und 230 Volt Verbindungen ersparen kann. Somit senkt man mit dieser Vorgehensweise bei der Verkabelung nicht nur die Kosten, sondern ist in weiterer Folge auch flexibler was die Einsatzorte für die Hardware betrifft (Zimmermann & Spurgeon, 2014).

Als Ziel für den PoE Standard wurden laut Zimmermann & Spurgeon (Ethernet: The Definitive Guide, 2014) von der IEEE unter anderem folgende Punkte vorgegeben:

- Energie – Die Endgeräte sollen über die Verbindung mit ausreichend Strom versorgt werden
- Sicherheit – Isolierter Betrieb der Netzwerkkomponenten; Es sollen keine Störspannungen auf den Verbindungen liegen
- Kompatibilität – Der Standard soll mit vorhandenen Kabeln funktionieren; Eine Neuverkabelung soll nicht notwendig sein
- Einfachheit – Die Komplexität soll sich nicht erhöhen; Der Enduser soll nur ein Netzwerkkabel anstecken müssen

Für den Einsatz von PoE gibt es laut Westcott & Coleman (CWNA Certified Wireless Network Administrator Official Deluxe Study Guide: Exam CWNA-106, 2015) und Zimmermann & Spurgeon (Ethernet: The Definitive Guide, 2014) zwei relevante Standards:

- 802.3af – Dieser Standard wurde erstmals im Jahr 2003 verabschiedet und liefert bis zu 15,4 Watt über 10/100/1000 Mbit Verbindungen
- 802.3at – Dieser Standard wurde im Jahr 2009 verabschiedet und baut auf 802.3af auf. Er liefert bis zu 34,2 Watt. Dieser Standard wird auch oft als PoE+ oder PoE Plus bezeichnet

Weiters existieren Erweiterungen zu den Standards, die von diversen Unternehmen eingeführt wurden. So bietet zum Beispiel die Firma CISCO einen PoE Mechanismus, welcher bis zu 60 Watt liefern kann (CISCO, 2016).

Laut Zimmermann & Spurgeon (Ethernet: The Definitive Guide, 2014) existieren bei PoE außerdem zwei Geräte-Rollen:

- Power Sourcing Equipment (PSE)

  Hierbei handelt es sich um das Gerät, welches die benötigte Leistung liefert. In unserem Fall wäre dies der PoE-fähige Ethernet-Switch. Alternativ dazu käme auch ein so genannter PoE-Injektor in Frage. Hierbei wird das vorhandene Ethernet-Kabel, welches von einem nicht-PoE-fähigen Switch zugeleitet wird mittels eines Zwischen-Schalt-Gerätes um die PoE-Funktion erweitert. Diese Vorgehensweise würde jedoch dazu führen, dass man – je nach Aufbau – die Vorteile wieder verliert. Unter anderem wäre es nicht mehr möglich das Gerät aus der Ferne zurückzusetzen.

- Powered Device (PD) oder Data Terminal Equipment (DTE)

  Hierbei handelt es sich um das Gerät selbst, welches mit Strom versorgt wird. In unserem Fall wäre dies das PoE-Modul des Raspberry Pi.

Äquivalent zum PSE in Form eines PoE-Injektors dient das hier beschriebene PoE Modul genau genommen als PoE-Splitter. Das PD wird also nicht direkt mit Strom versorgt, sondern das Vorschaltgerät teilt das Ethernet-Signal wieder in das Daten-Signal sowie die Stromversorgung auf und speist somit den Raspberry Pi über die vorhandenen Pins mit Strom. Abbildung 5 veranschaulicht diesen Aufbau.

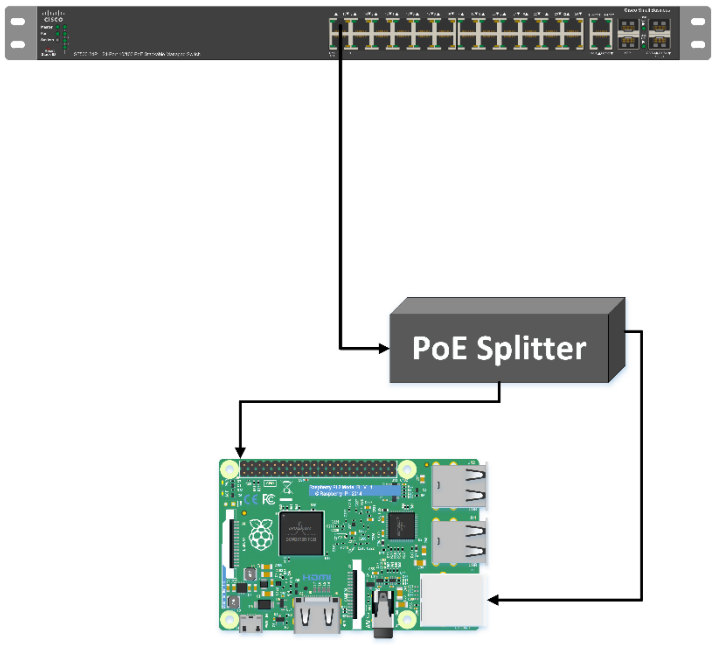

Abbildung 5: PoE Splitter (Zimmermann & Spurgeon, 2014)

Dieser Aufbau kann genutzt werden um den Raspberry Pi über das Netzwerk zurückzusetzen indem man den PoE-Port kurzzeitig deaktiviert und wieder reaktiviert. Je nach Wahl des Bootmediums kann dies jedoch zu fehlerhaften Sektoren führen, was im schlimmsten Fall bedeutet, dass der Raspberry Pi beziehungsweise sein Bootmedium physisch vor Ort ausgetauscht werden muss – Näheres hierzu und wie dies verhindert werden kann wird in Kapitel 2.6 beschrieben.
Der nächste Abschnitt geht nun näher auf die vorhandene Problematik der Zeitsynchronisierung beim Raspberry Pi ein.

2.5 Real Time Clock Modul

Der Raspberry Pi verfügt auch in der dritten Generation noch nicht über ein integriertes Real Time Clock (RTC) Modul, welches dafür sorgt, dass die Systemuhrzeit auch nach einem Reboot erhalten bleibt – was für die meisten Anwendungsfälle zwingend notwendig ist, da die Systemzeit häufig als Grundpfeiler für Programminstruktionen oder Sicherheitssysteme wie Kerberos (Garman, 2003, S. 30-35) dient. Diese Echtzeituhr kann bei den meisten Embedded Systems jedoch nachgerüstet werden (Li & Yao, 2003, S. 167-169). Laut Li & Yao besteht diese Echtzeituhr meist aus einer Batterie, welche das Konstrukt der Zeitmessungseinheit dauerhaft mit Strom versorgt – auch wenn das angeschlossene System seine eigene Stromversorgung verliert. Um die Energiequelle nicht unnötig zu belasten, wird hierbei die Energie, bei vorhandener Stromversorgung, vom Embedded System geliefert. Erst wenn dieses selbst über keine Stromversorgung mehr verfügt wird auf die angeschlossene Pufferbatterie zurückgegriffen – hierdurch ist es möglich die RTC jahrelang laufen zu lassen (Hartl, 2008, S. 356). Somit kann sichergestellt werden, dass man den Zeitstempel – meist als sogenannter Unix-Zeitstempel in Form von vergangenen Sekunden seit dem 01.01.1970 00:00:00 Uhr – nicht verliert, sodass sich das Betriebssystem des angeschlossenen Systems die korrekte Uhrzeit nach dem Neustart wieder auslesen kann.
Technisch wird hierfür heutzutage meist ein sogenannter Uhrenquarz verwendet, welcher mit einer Taktfrequenz von 32.768 Hertz schwingt und hierdurch Aufschluss darüber gibt wieviel Zeit in einer bestimmten Periode vergangen ist (Rost & Wefel, 2016).
Neben der angeführten Lösung mittels einer RTC kann bei einer in einem Netzwerk angeschlossenen Lösung auch auf das sogenannte Network Time Protocol (NTP) zurückgegriffen werden. Hierbei wird laut Mills (RFC 958 - Network Time Protocol (NTP), 1985) die aktuelle Uhrzeit mittels User Datagram Protocol (UDP) über das Netzwerk übertragen, sodass der Client in regelmäßigen Abständen über die aktuelle Uhrzeit informiert werden kann. Hierbei wird auch die Paketlaufzeit berücksichtigt, welche ansonsten eine Abweichung der Uhrzeit bedeuten könnte.

Neben dem NTP gibt es auch eine weitere vereinfachte Variante, welche sich Simple Network Time Protocol (SNTP) nennt und zu NTP kompatibel ist, jedoch laut Mills weniger Ressourcen für die Berechnung der aktuellen Uhrzeit benötigt. Bei SNTP wird hierfür jedoch die Genauigkeit vernachlässigt, da die entsprechenden Algorithmen weniger Komplex ausfallen und somit weniger Variablen im Verfahren berücksichtigt werden (Mills, RFC 4330 - Simple Network Time Protocol (SNTP) Version 4 for IPv4, IPv6 and OSI, 2006). Mills weist außerdem darauf hin, dass bei SNTP nur ein einziger Zeitserver in die Berechnung der Uhrzeit mit einfließt, während es bei NTP mehrere Server sein können. Als NTP-Server können lokale Server im Netzwerk oder auch Server, welche sich im Internet befinden dienen. Häufig wird ein NTP-Server im lokalen Netzwerk zur Verfügung gestellt, welcher sich die Uhrzeit aus dem Internet bezieht und diese Zeitinformation dann an die lokalen Clients im Netzwerk weitergibt. Hierdurch ergibt sich eine zentrale Anlaufstelle für sämtliche Zeitabfragen im lokalen Netzwerk, was auch dazu beiträgt die Komplexität des Aufbaus zu verringern und die Sicherheit zu erhöhen, da nicht allen Netzwerkgeräten der Internetzugriff erlaubt werden muss (Garman, 2003, S. 51-53).

Nachdem nun der grundlegende Hardwareaufbau veranschaulicht wurde geht es im nächsten Kapitel darum welche Speichermöglichkeiten der Raspberry Pi in der Version 3 mitbringt und wie diese genutzt werden können um ein möglichst problemfreies Ausführen des Betriebssystems zu gewährleisten.

2.6 Mögliche Bootoptionen für das Betriebssystem

Aufgrund des Aufbaus der aktuellen Raspberry Pi-Versionen ist es möglich diverse Medien als Startmedium für das Betriebssystem zu verwenden. In den älteren Versionen war dies nur über Umwege möglich. So war es bei Raspberry Pi Version B und Raspberry Pi 2 Version B nicht möglich ohne eine eingelegte Secure Digital (SD) beziehungsweise Micro Secure Digital (MicroSD)-Karte zu booten. Hierbei musste auf der MicroSD-Karte ein Verweis hinterlegt sein, der auf das folgende Bootmedium verweist. Die dritte Generation hat diese Möglichkeit jedoch nachgeliefert und hiermit ist es nun möglich direkt von zum Beispiel Universal Serial Bus (USB)-Laufwerken oder direkt vom Netzwerk zu booten (c't Redaktion, 2016). Die folgenden Abschnitte gehen näher auf die einzelnen Wege und ihre Vor- sowie Nachteile ein.

2.6.1 MicroSD-Karten

Die Möglichkeit von MicroSD-Karten zu booten ist die Standardoption, welche bereits seit der ersten Revision des Raspberry Pi existiert. Die erste Generation hat jedoch im Gegensatz zur darauffolgenden Generation auf die in ihren Abmessungen größeren SD-Karten als Bootmedium zurückgegriffen. Die grundlegenden Vorteile und Probleme hierbei sind jedoch von der Größe der verwendeten Karte unabhängig und beziehen sich – sofern nicht anders angegeben –

hauptsächlich auf die verwendete Speichertechnologie in Form des Flash-Speichers. Somit können die folgenden Zeilen als gültig für beide Speichertypen angesehen werden.

Flash-Speicher kann, im Gegensatz zu RAM, Daten auch im Speicher behalten, wenn dieser nicht mit Strom versorgt wird. Wäre dies nicht der Fall würde ein Betriebssystem, das auf verlässliche Speicherstände zugreifen muss, auf dieser Art von Speicher nicht praktikabel eingesetzt werden können. Laut Benz (Die Technik der Flash-Speicherkarten, 2006) ist die Lebenszeit von Flash-Speicher-Zellen stark begrenzt. Hierbei sind Werte von 10.000 bis 100.000 Schreibvorgänge bis zum Lebensende einer Flash-Zelle die Norm. Lesevorgänge lassen die Speicherzellen jedoch nicht altern und haben dementsprechend keinen Einfluss auf ihre Lebenszeit. Aufgrund des technischen Aufbaus der Flash-Speicher-Zellen, die heutzutage aus Kostengründen sowie aufgrund der maximalen Speicherkapazität meist aus sogenannten Nicht Und (NAND)-Transistoren bestehen, wird jedoch beim Löschen von einzelnen Zellen ein ganzer Speicherblock gelöscht. Im Anschluss an das Löschen können dann die gewünschten Zellen gezielt wieder mit dem gewünschten Wert beschrieben werden ohne ganze Speicherblöcke schreiben zu müssen. Diese Vorgehensweise sorgt jedoch auch dafür, dass die Speicherzellen einer größeren Abnutzung ausgesetzt sind (Zheng, Tucek, Qin, & Lillibridge, 2013).

Benz (Die Technik der Flash-Speicherkarten, 2006) weist außerdem darauf hin, dass für den Fall von defekten Flash-Speicher-Zellen ein sogenanntes Defektmanagement eingesetzt wird. Hierbei kümmert sich der Controller in der MicroSD-Karte darum die gewünschten Daten effizient auf die vorhandenen Flash-Speicher-Chips zu verteilen um somit die Abnutzung von den immer gleichen Zellen so weit wie möglich zu minimieren. Diese Technik wird Wear Leveling genannt (Gay, 2014, S. 89). Dieses Defektmanagement sorgt im Falle eines Defekts auch dafür, dass defekte Zellen nicht mehr länger verwendet werden und ersetzt diese durch eigens hierfür reservierte Ersatz-Speicher-Zellen, sodass es zu keinem Datenverlust kommt. Die Erkennung einer defekten Speicherzelle erfolgt durch Prüfsummen, welche der Controller berechnet und vergleicht (Bertacco & Legay, 2013, S. 221). Weicht die Prüfsumme vom gelesenen Wert ab weiß der Controller, dass die Daten korrumpiert sind, errechnet, wenn möglich, die ursprünglichen Daten und schreibt diese in die Ersatz-Speicher-Zelle.

Diese Prüfsumme wird meist anhand der zyklischen Redundanzprüfung errechnet. Der englische Begriff hierfür lautet Cyclic Redundancy Check (CRC).

Neben eines Defektes der Speicher-Zellen durch Alterung oder übermäßigen Gebrauch kann es auch zu korrumpierten Daten durch unterbrochene oder konkurrierende Zugriffe kommen (Zheng, Tucek, Qin, & Lillibridge, 2013). Hierbei kann eine unterbrochene Stromversorgung während des Schreibvorgangs, ein konkurrierender Schreibvorgang in einer benachbarten Speicherzelle oder ein konkurrierender Lesevorgang in einer benachbarten Speicherzelle dazu führen, dass die geschriebenen Daten unvollständig und somit verloren sind. Dies kann im schlimmsten Fall dazu führen, dass die Benutzerdaten unwiederbringlich verloren sind oder Systemdaten soweit geschädigt sind, dass ein korrekter Start des Betriebssystems nicht mehr möglich ist.

Tests von Tseng, Grupp & Swanson (Understanding the Impact of Power Loss on Flash Memory, 2011) haben ergeben, dass man bei einer unerwarteten Unterbrechung der Stromversorgung –

wie es zum Beispiel bei einem nicht vollständigen Herunterfahren eines Betriebssystems vorkommt – davon ausgehen muss, dass die zuletzt geschriebenen Daten korrumpiert sind.

Im nächsten Abschnitt wird nun darauf eingegangen inwiefern sich die Möglichkeit von USB-Laufwerken zu booten davon unterscheidet MicroSD-Karten zu verwenden.

2.6.2 USB-Laufwerke

Nachdem im vorherigen Kapitel darauf eingegangen wurde, welche Probleme bei der Verwendung von Flash-Speicher existieren, wird in diesem Abschnitt eine Alternative zur MicroSD-Karte aufgezeigt. Prinzipiell beruhen moderne USB-Laufwerke auf die in diesem Kapitel eingegangen werden soll, auf einem ähnlichen Konzept wie die MicroSD-Karten – nämlich ebenfalls auf den zugrundeliegenden Flash-Speicherzellen. Insofern existieren hierfür ähnliche Probleme wie im letzten Kapitel, aufgrund des unterschiedlichen Aufbaus und der Komplexität der USB-Laufwerke können diese Probleme jedoch umschifft werden. Natürlich existieren auch nicht Flash-basierte USB-Laufwerke, wie zum Beispiel magnetische Festplatten, diese kommen jedoch für den in dieser Arbeit beschriebenen Einsatzzweck nicht in Frage, da diese Festplatten mehr Strom benötigen (zirka 6 Watt im Vergleich zu durchschnittlich 2 Watt bei SSDs) und aufgrund ihres technischen Aufbaus nicht in Frage kommen.

Das Hauptproblem bei magnetischen Festplatten in diesem Beispiel ist die Tatsache, dass die darin enthaltenen Magnetscheiben zuerst auf ihre benötigte Drehzahl beschleunigt werden müssen um von ihnen Daten lesen zu können – dieser Vorgang dauert bei modernen magnetischen Festplatten mehrere Sekunden. Weiters bremst die Seek Time den Festplattenzugriff weiter ein, da eine konventionelle magnetische Festplatte für den Zugriff auf ihre einzelnen Datenblöcke einige Zeit benötigt um den Lesekopf entsprechend mechanisch auszurichten (Shanks, 2013). Der Raspberry Pi erwartet jedoch – sofern man unmittelbar von einem USB-Laufwerk booten möchte – bereits nach wenigen Millisekunden Daten um das Bootmedium zu akzeptieren. Weiters benötigt dieser Vorgang auch bedeutend mehr Strom als die USB 2.0 Spezifikation über einen einzelnen USB-Anschluss liefern kann. Die Spezifikation sieht hier maximal 500 Milliampere (mA) vor (USB Implementers Forum, Inc., 2000). Typische mechanische 2,5 Zoll Festplatten benötigen als Anlaufstrom jedoch 600 bis 1100 mA.

Aus oben genannten Gründen wird in diesem Kapitel deshalb nur auf USB-Flash-Sticks sowie hauptsächlich auf USB-Solid State Drives (SSD) eingegangen, da für Flash-Sticks im Prinzip die gleichen Bedingungen gelten wie unter Kapitel 2.6.1.

Bei SSDs unterscheidet man zwischen den sogenannten Single-Level Cell (SLC) sowie den Multi-Level Cell (MLC) SSDs. Beide SSD-Typen verwenden die unter Kapitel 2.6.1 beschriebenen NAND-Speicherzellen. Beim SLC Aufbau wird ein einzelnes Bit für die Speicherung der Information verwendet, während bei MLC-SSDs mehrere Bits gemeinsam in einer einzelnen Speicherzelle gespeichert werden (Lawton, 2014). Während es somit bei SLC-SSDs nur die Information 0 oder 1 in einer Speicherzelle gibt können diese Werte bei MLC-SSDs auch zum Beispiel den Wert 00, 01, 10 oder 11 annehmen. Diese Zwischenstufen erfordern es jedoch, dass auf mögliche Ladungsverluste beziehungsweise defekte Zellen mehr Rücksicht

genommen wird als bei SLC-SSDs, da sich ein Verlust hier wesentlich stärker negativ bemerkbar macht und somit mehr Information auf einmal verloren gehen kann.

Während bei einer SLC-SSD die Information in einer Zelle mittels eines einzigen Schreibvorganges abgeschlossen ist kann dieser bei MLC-SSDs mehrere Schreibvorgänge mit unterschiedlichen Spannungen erfordern – je nach angelegter Spannung wird eine andere Information geschrieben. Eine solche Zelle wird also bedeutend stärker abgenutzt als ihr Pendant, welche nur ein einzelnes Bit speichern muss. Durch diese Mehrbelastung kann die Zelle schneller defekt werden und somit auch mehr Informationen verlieren, da nicht nur 1 Bit verloren geht, sondern gleich mehrere (Hruska, 2016). Zur Kompensation dieses Problems müssen somit mehr Wiederherstellungsinformationen für die gespeicherten Daten zusätzlich abgelegt werden um somit auftretende Fehler gezielt eliminieren zu können. Diese zusätzlichen Informationen schmälern somit den Platzgewinn, welchen man durch die Verwendung von MLC-Technologie hätte, etwas ab. Weiters sorgt diese Technologie dafür, dass die Schreib- und Lesegeschwindigkeit im Vergleich zu SLC-Speicher niedriger ausfällt (Micheloni, Marelli, & Eshghi, 2012, S. 70).

Im Gegensatz zu den in Kapitel 2.6.1 beschriebenen MicroSD-Karten existiert bei SSDs standardmäßig ein schnellerer Cache Zwischenspeicher, der dafür sorgt, dass bei schreibintensiven Vorgängen wie zum Beispiel der Aktualisierung von Routing-Tabellen der Flash-Speicher geschont wird, indem statt auf diesem die Zwischenspeicherung auf sogenanntem Double Data Rate Synchronous Dynamic Random Access Memory (DDR-SDRAM) durchgeführt wird, bevor die Daten zur Langzeitarchivierung in den Flash-Speicher geschrieben werden (Micheloni, Marelli, & Eshghi, 2012, S. 21).

Das nächste Kapitel beschreibt eine weitere Möglichkeit der Speicheranbindung bei Geräten wie dem Raspberry Pi in Form von direkt auf der Platine aufgebrachten Speicherzellen.

2.6.3 Embedded Multimedia Card

Die embedded Multimedia Card (eMMC) beruht wie bereits die beiden zuvor vorgestellten Möglichkeiten auf einzelnen Flash-Bausteinen, welche fix direkt auf der Platine des elektronischen Geräts aufgebracht werden. In diesem Kapitel wird nur ein grober Überblick über diese Möglichkeit gegeben, da der verwendete Raspberry Pi 3 Model B über keinen eMMC Speicher verfügt. Aufgrund der angekündigten Markteinführung des sogenannten Raspberry Pi Compute Module 3, welches großteils auf der gleichen Hardware wie das in dieser Arbeit verwendete Gerät beruht, wird hierbei ein kurzer Überblick über die potentiellen Vorteile und Nachteile gegeben.

Das Compute Module 3 verfügt über die gleichen Leistungsparameter wie der Raspberry Pi 3 Model B. Dies betrifft sämtliche in Kapitel 2.2 angeführten Bestandteile, wie den verwendeten Prozessor, die Arbeitsspeicherausstattung sowie die Anbindung des Gesamtkonstrukts – ein Unterschied existiert jedoch zum Beispiel bei den vorhandenen Anschlüssen sowie der in diesem Kapitel beschriebenen verwendeten Speicherarchitektur in Form eines eMMC Speichers mit einer Größe von 4GB (Raspberry Pi (Trading) Ltd., 2016).

Die Verwendung von eMMC Speicher hat den Vorteil, dass hierbei TRIM Unterstützung vorhanden ist, sowie sicheres Löschen der Flash-Speicherzellen ermöglicht wird (Upton, Duntemann, Roberts, Mamtora, & Everard, 2016). TRIM Unterstützung erlaubt es dem Betriebssystem Speicherbereiche zu markieren ohne sie tatsächlich zu löschen (Labs & Spier, 2016). Dies sorgt dafür, dass der Flash-Speicher geschont wird und die Schreibgeschwindigkeit zunimmt, da nicht die gesamte Speicherzelle sofort überschrieben werden muss, sondern erst wenn diese mit neuen Daten befüllt wird.

Laut Upton et al. (Learning Computer Architecture with Raspberry Pi, 2016) erlaubt eMMC Speicher außerdem die Verwendung mehrerer Boot-Partitionen sowie einem getrennten Replay-Protected Memory Block (RPMB). Dieser erlaubt es das Betriebssystem und sämtliche Daten darauf zu verschlüsseln, während die Entschlüsselungsinformationen auf dem RPMB gespeichert bleiben und somit während des Bootvorgangs zugänglich sind.

Ein Nachteil dieser Lösung ist jedoch, dass die Größe des eMMC-Speichers meist vom Hersteller des Geräts vorgegeben wird, da dieser den Speicher bereits fix auf die Platine aufbringt. Im Falle des Compute Module 3 sind dies nur 4GB Speicherplatz. Die Variante L des Compute Module 3 (CM3L) wird jedoch die Möglichkeit bieten die Speichergröße selbst festzulegen, da der hierfür vorgesehene Platz auf der Platine freigehalten wird, sodass er vom Benutzer selbst bestückt werden kann (Raspberry Pi (Trading) Ltd., 2016).

Im nächsten Kapitel wird nun eine Möglichkeit aufgezeigt wie man ohne direkt angeschlossenes Speichermedium ein Betriebssystem auf dem Raspberry Pi starten kann.

2.6.4 Pre-boot Execution Environment

Das sogenannte Pre-boot Execution Environment (PXE) ist eine Möglichkeit ein Betriebssystem über das angeschlossene Netzwerkinterface zu starten. Hierfür werden am Client keine lokal angeschlossenen Speicher benötigt, da das Betriebssystem direkt über das Netzwerk geladen und im RAM zwischengespeichert wird. PXE benutzt hierfür die Protokolle Transmission Control Protocol (TCP), UDP, Dynamic Host Configuration Protocol (DHCP) sowie das Trivial File Transfer Protocol (TFTP).

Laut Nemeth, Snyder & Hein (Linux-Administrations-Handbuch, 2007) bietet diese Vorgehensweise nicht nur den heutzutage vernachlässigbaren Kostenvorteil aufgrund nicht benötigter Client-Speichersysteme, sondern auch den Vorteil, dass sämtliche relevanten Daten nicht nur am Client, sondern direkt am Server gespeichert sind, sodass ein Ausfall des Clients keinen direkten Datenverlust bedeutet. Außerdem ist es bei dieser Lösung möglich auch bei Ausfall oder ausgeschaltetem Client die Konfigurationsdateien desselbigen zu verändern ohne dass man sich physisch zu dessen Aufstellungsort begeben müsste.

Der Ablauf des Netzwerk-Boot-Vorganges ist in Abbildung 6 veranschaulicht und erfolgt laut Reiter (Minimyth – ein Diskless Client für MythTV, 2011) in folgender Weise:

- Der Client fragt im Netzwerk nach einem Proxy DHCP, der PXE-Boot unterstützt
- Der Server antwortet mit der Information wo die benötigten PXE-Boot-Dateien zu finden sind und weist dem Client eine Internet Protocol (IP)-Adresse zu
- Der Client greift mittels TFTP auf die PXE-Boot-Dateien zu um sich das sogenannte Network Bootstrap Program (NBP) in den Arbeitsspeicher zu laden und dieses auszuführen
- Der Client greift mittels TFTP auf die Konfigurationsdateien sowie den benötigten Kernel zu und lädt diese in seinen Speicher
- Der Client startet den Kernel und somit das Betriebssystem

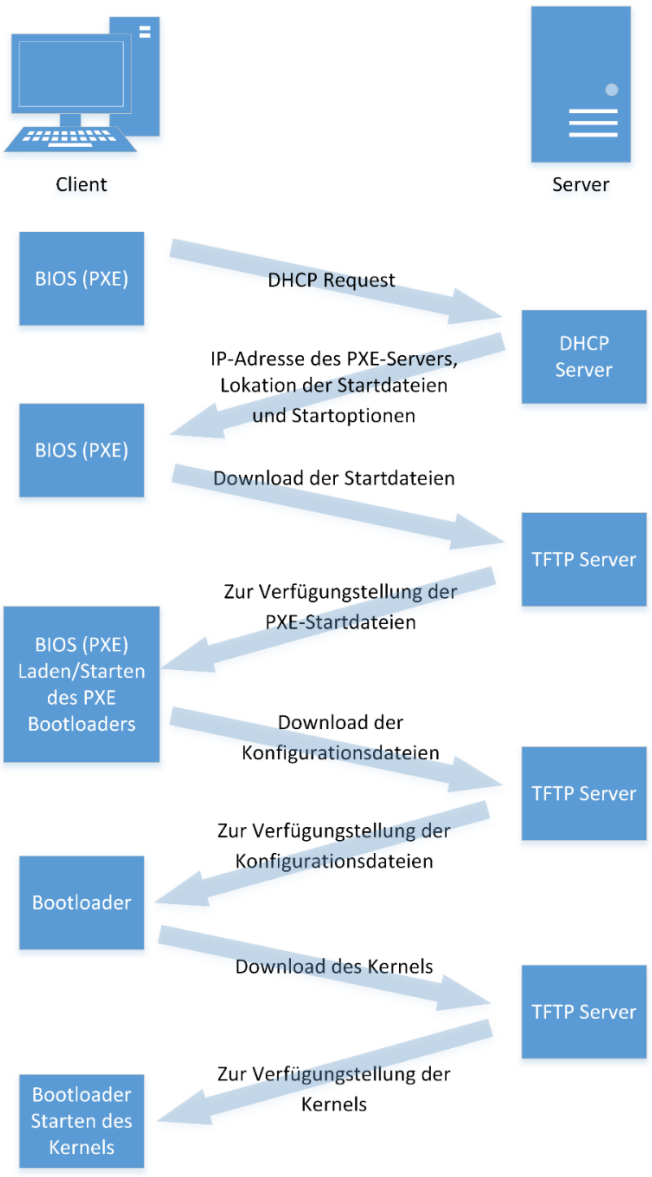

Abbildung 6: PXE Bootvorgang (Reiter, 2011)

Sofern benötigt können hierbei natürlich auch die unter Kapitel 4.1 näher beschriebenen Sicherheitstechnologien, wie zum Beispiel 802.1X sowie Active Directory Authentifizierung verwendet werden.
Nachdem nun die technischen Hardware-Optionen veranschaulicht wurden, wird im nächsten Kapitel darauf eingegangen welche Betriebssysteme derzeit für den Raspberry Pi existieren.

2.7 Betriebssystem-Varianten

Für den Raspberry Pi 3 existieren im Jahr 2016 diverse Betriebssystem-Varianten, welche jede für sich diverse Vor- und Nachteile bieten und somit für bestimmte Einsatzzwecke prädestiniert sind. Aufgrund des unterschiedlichen Aufbaus und der unterschiedlichen Befehlssätze des verwendeten ARM-Prozessors im Vergleich zur x86-Architektur – Näheres hierzu siehe Kapitel 2.1 – sind laut Dembowski (Raspberry Pi - Das technische Handbuch: Konfiguration, Hardware, Applikationserstellung, 2015, S. 123) standardmäßig auf Computern eingesetzte Betriebssysteme wie die x86-Version von Microsoft Windows oder x86-Versionen von Linux nicht auf dem Raspberry Pi lauffähig. In diesem Abschnitt wird näher auf die auf ARM-Prozessoren lauffähigen Betriebssysteme eingegangen um einen groben Überblick hierüber zu geben. Dabei wurden ausschließlich offizielle Distributionen und Projekte angeführt hinter denen Firmen stehen, welche den Support und die Weiterentwicklung für diese Systeme gewährleisten sollten. Betriebssysteme, welche nur aufgrund von Community-Initiativen auf dem Raspberry Pi lauffähig gemacht wurden – wie zum Beispiel ChromeOS oder FreeBSD – blieben also unberücksichtigt.

Den Beginn macht hierbei Raspbian, das wohl bekannteste Betriebssystem für den Raspberry Pi, welches für den Einsatz auf diesem optimiert wurde.

2.7.1 Raspbian

Raspbian ist laut Harrington (Learning Raspbian, 2015, S. 10) das im Jahr 2015 beliebteste und meistverwendete Betriebssystem für den Raspberry Pi. Es ist als Open Source verfügbar und beruht auf dem unixoiden Betriebssystem Debian, welches für die Verwendung auf dem Raspberry Pi optimiert wurde. Auch das in Kapitel 2.7.4 beschriebene Betriebssystem stammt von UNIX ab und ist somit im Grundaufbau sehr ähnlich. Laut Dembowski (Raspberry Pi - Das technische Handbuch: Konfiguration, Hardware, Applikationserstellung, 2015, S. 41) handelt es sich bei Raspbian um ein „Allround-Betriebssystem“, welches für keinen speziellen Anwendungsfall hin optimiert wurde außer darauf auf dem Raspberry Pi optimal ausgeführt zu werden. Dembowski weist außerdem darauf hin, dass Raspbian eines von wenigen Betriebssystemen ist, welches die im SoC enthaltene Floating Point Unit (FPU) unterstützt und somit diesbezügliche Berechnungen hardwarebeschleunigt durchführen kann, während andere Betriebssysteme durch die softwaremäßige Berechnung Potential verschenken und sie somit länger für das gleiche Ergebnis rechnen müssen – Näheres zum Aufbau des SoC wird in Kapitel 2.1.2 beschrieben.

Raspbian bietet folgende Grundbestandteile (Harrington, 2015):

- Bootloader – siehe hierzu Kapitel 2.1.2
- Linux Kernel – das Herz des Betriebssystems, welches dafür zuständig ist die Ausgabe auf dem Bildschirm oder auch Eingaben zu regeln. Weiters enthält er unter Linux die notwendigen Treiber um die meisten Geräte ansprechen zu können (Love, 2010, S. 1-6). Verbindet man zum Beispiel einen USB-Stick, so ist der Kernel dafür zuständig diesen richtig zu initialisieren.
- Daemons – Bestandteile, welche dafür sorgen, dass mit dem Kernel interagiert werden kann. Hierbei existieren diverse Daemons, welche unter anderem zum Beispiel für die Erkennung von USB-Sticks oder dem automatischen Ausführen von Befehlen in bestimmten Zeitabständen – oft Cronjobs genannt – zuständig sind (Nemeth, Snyder, & Hein, Linux Administration Handbook, 2006, S. 150-157).
- Shell inklusive Utilities – unter Raspbian wird standardmäßig auf die Bourne-again shell (Bash) zurückgegriffen. Es handelt sich hierbei im Prinzip um ein Kommandozeilen-Tool mit welchem das System gesteuert werden kann. Die Utilities erweitern das Tool um zusätzliche Möglichkeiten wie zum Beispiel Datei-Verwaltungs-Tools.
- X.Org graphischer Server – hierbei handelt es sich um die Schnittstellen, welche die graphische Oberfläche des Betriebssystems benötigt um zum Beispiel Maus-Interaktionen abzuarbeiten.
- Desktop Umgebung – hierbei handelt es sich um die eigentliche Oberfläche des Betriebssystems, welche auf dem X.Org aufbaut. Bei Raspbian kommt hierbei LXDE zum Einsatz, was für den ressourcenschonenden Einsatz auf Geräten wie dem Raspberry Pi konzipiert wurde.

Nahezu alle UNIX basierten Betriebssysteme greifen zur zentralen Aktualisierung und Installation von Betriebssystem-Bestandteilen sowie Third-Party-Programmen auf ein Paket-Management-System zurück. Laut Robbins (Unix in a Nutshell: A Desktop Quick Reference - Covers GNU/Linux, Mac OS X,and Solaris, 2005, S. 467-520) handelt es sich bei den Debian Derivaten – wozu, wie bereits beschrieben, auch Raspbian zählt – um den Paket Manager genannt Advanced Packaging Tool (APT). Dieser bietet laut Robbins die Möglichkeit weitere Paketquellen hinzuzufügen, sodass man zum Beispiel über einen selbst definierten zentralen Server Updates sowie Programme automatisiert verteilen kann. Durch die Verwendung von Prüfsummen in Form von MD5-Hash-Werten sowie Verschlüsselungsalgorithmen wie Pretty Good Privacy (PGP) oder GNU Privacy Guard (GPG) kann sichergestellt werden, dass die zur Verfügung gestellten Installationsdateien fehlerlos übertragen wurden und von keiner dritten Person auf diesem Übertragungsweg manipuliert wurden. Erst nach der standardmäßig aktivierten Verifizierung der Signaturen werden die gewünschten Pakete installiert (Krafft, 2005, S. 373-379). Weiters kümmert sich APT laut Bauer (Automating UNIX and Linux Administration, 2003, S. 253) auch darum gegebenenfalls Abhängigkeiten in Form von fehlenden Bibliotheken automatisiert zu installieren, sodass gewünschte Programme lauffähig sind ohne hier selbst Hand anlegen zu müssen.

Neben den bereits angeführten Funktionen bietet APT laut Robbins (Unix in a Nutshell: A Desktop Quick Reference - Covers GNU/Linux, Mac OS X,and Solaris, 2005, S. 468) folgende Vorteile:

- Deinstallation nicht mehr benötigter Software
- Rückgängig machen von Installationen oder Updates – APT führt hierfür Protokoll darüber was wann installiert wurde und belässt auch alte Versionsstände sowie Konfigurationsdateien standardmäßig auf dem Speichermedium, sodass im Falle eines Fehlers auf die alte funktionierende Konfiguration zurück gewechselt werden kann
- Führen eines Protokolls über die installierten Pakete, deren Abhängigkeiten sowie deren Versionsstand

Robbins weist außerdem darauf hin, dass für die Installation oder das Entfernen von Software über APT in den meisten Fällen root-Rechte benötigt werden, sodass es also für einen Dritten nicht ohne weiteres möglich ist – trotz Zugriff auf den Client – Programme zu manipulieren.

Zur Fernwartung des Systems existiert die Secure Shell (SSH). Hierbei handelt es sich um eine Client/Server Architektur, welche dafür sorgt, dass sämtliche Kommunikation zwischen den Kommunikationspartnern verschlüsselt erfolgt (Barrett & Silverman, 2001, S. 1-2). Diese Shell ermöglicht es dem Administrator Befehle über eine Ende-zu-Ende verschlüsselte Verbindung auf dem Server auszuführen ohne sich physisch in dessen Nähe befinden zu müssen. Für diese Verschlüsselung wird bei der aktuellen Version SSH-2 auf das Diffie-Hellman Schlüsselaustauschverfahren zurückgegriffen (Barrett & Silverman, 2001, S. 59-60), welches in Kapitel 4.1.1 genauer beschrieben wird.

Im nächsten Kapitel wird nun auf ein System eingegangen, welches für den Einsatz auf RISC-Plattformen optimiert wurde.

2.7.2 RISC-OS

Das Betriebssystem RiscOS ist im Gegensatz zu dem zuvor angeführten System kein Linux Derivat, sondern ein gezielt für den Einsatz auf der RISC-Plattform programmiertes System der Firma Acorn Computers. Näheres zur RISC-Plattform wird in Kapitel 2.1 beschrieben.

Laut Upton & Halfacree (Raspberry Pi User Guide, 2014, S. 49) handelt es sich bei RiscOS um ein System, welches bezüglich seines Feedbackverhaltens performanter arbeitet als alle anderen für den Raspberry Pi erhältlichen Betriebssysteme. Aufgrund der Tatsache, dass das System nicht auf Linux beruht werden jedoch eigens hierfür entwickelte Programme benötigt. Diese Programme sind laut Upton & Halfacree jedoch in den letzten Jahren stetig seltener anzutreffen, da das ursprüngliche Entwicklungsunternehmen Acorn im Jahr 1998 aufgelöst wurde und das System selbst somit nur noch von Drittentwicklern weiterentwickelt wird. Diese Drittentwickler waren auch der Grundstein dafür, dass das System überhaupt erst seinen Weg auf den Raspberry Pi gefunden hat.

Bauer (The Raspberry Pi Computer, 2013, S. 14) schreibt, dass RiscOS innerhalb von 10 Sekunden betriebsbereit ist. Der gesamte Bootvorgang – näher beschrieben in Kapitel 2.1.2 – ist somit in dieser Zeit abgeschlossen und die Einzel-Benutzer-Umgebung ist somit innerhalb

kürzester Zeit dazu bereit Aufgaben abzuarbeiten. Laut Bauer unterstützt RiscOS Multitasking, was dazu führt, dass Programme gleichzeitig ausgeführt werden können. Prinzipiell ist es Bauer wichtig zu erwähnen, dass das gesamte Betriebssystem nur sehr wenige Ressourcen benötigt und auch der Speicherbedarf sich in Grenzen hält. Problematisch für den Einsatz ist jedoch die fehlende Unterstützung von Grafikbeschleunigung durch die GPU, was dazu führt, dass zum Beispiel die Videowiedergabe nur mit wenigen Frames pro Sekunde (FPS) stattfinden kann.

Aufgrund der bereits beschriebenen Einzel-Benutzer-Umgebung ist eine Fernwartung oder der Einsatz des Systems als Headless-Lösung (also den Einsatz ohne graphische Ausgabe auf einem Bildschirm) mit diesem System nahezu unmöglich (Horan, 2013, S. 205-207). Horan merkt außerdem an, dass das Multitasking System auf welches RiscOS zurückgreift grundlegend anders arbeitet als die in den anderen Kapiteln beschriebenen Systeme. Es setzt auf das sogenannte kooperative Multitasking (CMT). Hierbei erhält das gerade ausgeführte Programm die Aufgabe nach Abschluss seiner Tätigkeit das nächste auszuführende Programm darüber zu informieren, dass dieses nun ausgeführt werden kann. Gibt es jedoch ein Problem bei der Ausführung – zum Beispiel in Form einer Endlosschleife – kann dies dazu führen, dass das gesamte System nicht mehr länger seiner Tätigkeit nachkommen kann (Horan, 2013, S. 205-207).

Im nächsten Kapitel wird nun näher auf das in Unternehmen eher verbreitete Betriebssystem Windows eingegangen, welches in abgeänderter Form ebenfalls für den Raspberry Pi existiert.

2.7.3 Windows 10 IoT Core

In diesem Kapitel wird näher auf die Möglichkeit eingegangen Windows 10 IoT Core auf dem Raspberry Pi 3 zu nutzen. Hierbei werden auch die Vor- und Nachteile dieser Lösung aufgezeigt.

Microsoft hat Windows 10 IoT Core erstmals im Jahr 2015 für die zweite Generation des Raspberry Pi vorgestellt. Aufgrund des aufkeimenden Themas des Internet of Things wollte Microsoft hierbei den Benutzern eine Möglichkeit bieten ihre bekannten Tools weiterzuverwenden während sie Software für das IoT entwickeln (Dallas, 2015).

> *Die Verfügbarkeit einer Windows-Plattform ebnet den Weg in das professionelle Umfeld und heraus aus der Ecke des belächelten Bastelrechners. In Kombination mit dem erprobten Entwicklungswerkzeug Visual Studio wird ein großes Feld für die Softwareentwicklung von morgen erschlossen. (Hüwe, 2010, S. 8)*

Zuallererst ist es wichtig zu erwähnen, dass der Einsatz von Windows 10 IoT Core laut derzeitigem Stand nur dann möglich ist, wenn man bereits andere Rechner mit Windows 10 im Unternehmen im Einsatz hat (Teixeira, 2015). Im Gegensatz zu den anderen angeführten Lösungen in den vorhergehenden Kapiteln, welche unabhängig vom Betriebssystem des Entwicklungsrechners eingesetzt werden können, muss hierbei also mit Kosten für entsprechende Windows-Lizenzen kalkuliert werden – wenn auch der Einsatz von Windows 10 IoT Core selbst kostenlos ist (Membrey & Hows, 2015, S. 257).

Es ist außerdem wichtig zu erwähnen, dass Windows 10 IoT Core auf dem Raspberry Pi über keine Hardwarebeschleunigung verfügt, da die hierfür notwendigen Treiber für die VPU/GPU – Näheres hierzu siehe Kapitel 2.1.2 – nicht zur Verfügung stehen. Animierte Oberflächen oder Videowiedergabe sind somit nur mit sehr niedrigen Frameraten möglich (Microsoft Corporation, 2016). Ein Einsatz dieses Systems ergibt somit nur im sogenannten Headless-Betrieb – also der Verwendung ohne angeschlossenen Bildschirm zum Beispiel zur Verwendung als Server, Sensor oder Aktor – oder bei automatisierten Anzeigen, welche keine direkten Interaktionsmöglichkeiten für den Benutzer bieten, Sinn.

Windows 10 IoT Core bietet weiters den Vorteil, dass hiermit eine problemlose Integration der Microsoft Azure Cloud Plattform möglich ist. Außerdem existiert laut Membrey & Hows (Learn Raspberry Pi 2 with Linux and Windows 10, 2015, S. 259) ein vollwertiges Application Programming Interface (API), welches es ermöglicht mit der Software-Plattform Microsoft .NET Programme für dieses System zu entwickeln. Microsoft .NET bietet laut Lowy (Programming .NET Components: Design and Build .NET Applications Using Component-Oriented Programming, 2005, S. xi-xii) die Möglichkeit programmiersprachenunabhängig vielseitige Programme zu entwickeln, welche auf diverse Frameworks wie zum Beispiel ADO.NET oder ASP.NET zurückgreifen können um hiermit Services zur Verfügung zu stellen oder um diese zu konsumieren. Diese Services können hierbei auf unterschiedlichsten Grundlagen, wie zum Beispiel dem Simple Object Access Protocol (SOAP) oder Representational State Transfer (REST), beruhen – Näheres hierzu siehe Kapitel 3.2.

Neben diesen Services ist es natürlich trotz der bereits erwähnten fehlenden Hardwarebeschleunigung auch möglich Programme mit graphischen Benutzeroberflächen zu realisieren. Windows 10 IoT Core setzt hierfür auf die sogenannte Universal Windows Application API (UWP) mit der es möglich ist Programme plattformübergreifend einzusetzen. Dies bedeutet, dass man die Logik seines Quellcodes nur einmal implementieren muss und im Anschluss daran die fertige Applikation für sämtliche Geräte auf denen Windows 10 läuft ausführen kann. Dies beinhaltet neben der in dieser Arbeit angeführten Windows 10 IoT Core, welche für die Ausführung auf ARM Geräten ausgelegt ist, auch Desktop-Rechner, welche auf x86 Prozessoren beruhen – Näheres hierzu siehe Kapitel 2.1.2. Weiters ist es mit UWP möglich die Applikationen auf der Microsoft Konsole Xbox One oder Smartphones mit dem Betriebssystem Windows 10 Mobile auszuführen. In diesem Fall ist es jedoch notwendig die Oberfläche für das zu verwendende Endgerät zu adaptieren, sodass eine sinnvolle Bedienung und Anzeige ermöglicht wird (Whitney, Jacobs, Weston, & Satran, 2017).

Applikationen für Windows 10 IoT werden laut derzeitigem Entwicklungsstand immer im Vollbildmodus ausgeführt – ein Fenstermodus, wie von der Desktop-Betriebssystem-Version von Windows 10 bekannt, existiert nicht. Die einzelnen Applikationen werden hierbei mit Hilfe der Windows Powershell auf den Raspberry Pi übertragen und ausgeführt (Schmidt, 2016, S. 21-22). Allgemein erfolgt die gesamte Konfiguration des Betriebssystems über die Windows Powershell, welche standardmäßig bei Windows 10 vorinstalliert ist. Ein Administrator, der in seinem Firmenumfeld also hauptsächlich mit Windows-Systemen konfrontiert ist, hat hiermit den Vorteil, dass dieser sich nicht auf neue Programme oder Befehle einstellen muss, sondern das Monitoring und Warten der Geräte über seine bereits bekannten Tools stattfinden kann.

Seit 2016 ist die Windows Powershell außerdem als Open Source und für diverse Fremdsysteme verfügbar. Laut dem PowerShell Team (PowerShell on Linux and Open Source!, 2016) existieren lauffähige, jedoch teilweise im Testzustand befindliche Implementierungen für folgende Systeme:

- Linux (Ubuntu 16.04, Ubuntu 14.04, CentOS 7)
- macOS (10.11)
- Windows (7, 8.1, 10, Server 2012 R2, Server 2016)
- Docker – Näheres hierzu siehe Kapitel 4.2.

Im nächsten Kapitel wird nun auf ein Konkurrenzprodukt vom Android-Entwickler Google eingegangen, welches ähnliche Voraussetzungen und Eigenschaften bietet wie das soeben beschriebene System.

2.7.4 Android Things

Neben vielen Community-Versionen des für Smartphones und Tablets beliebten Android-Betriebssystems gibt es für den Raspberry Pi seit Ende 2016 eine erste offizielle Testversion des Google-Betriebssystems namens Android Things. Hierbei handelt es sich um ein für IoT-Geräte konzipiertes Betriebssystem, das, ähnlich wie das zuvor beschriebene Windows 10 IoT, ohne Betriebssystemoberfläche ausgeliefert wird. Applikationen können somit ebenfalls nur automatisiert beim Starten des Betriebssystems ausgeführt werden oder über eine Netzwerkverbindung in Verbindung mit einem PC gestartet oder ausgetauscht werden. Ein Wechseln der Applikationen durch den Benutzer über direkte Eingaben auf dem Gerät wie dies zum Beispiel bei Smartphones der Fall ist, ist nicht vorgesehen.

Die Entwicklung von Applikationen für Android Things findet äquivalent zur Entwicklung von Applikationen für Android selbst statt – und zwar über das Tool Android Studio mit dessen Hilfe man das Android Software Development Kit (SDK) verwenden kann. Auch die Nutzung von Google Play Services, sowie der Google Cloud Plattform sind mit Android Things möglich (Piekarski, 2016).

Wang (Google 'Android Things' - An Operating System for the Internet of Things, 2016) erwähnt jedoch, dass laut derzeitigem Entwicklungsstand des Betriebssystems keine Benutzerinteraktivität in Form von Authentifizierung mittels Anmeldedaten möglich ist.

Die Kommunikation zwischen dem Entwicklungsrechner und dem Android Things-Gerät erfolgt über die sogenannte Android Debug Bridge (ADB), welche in diesem Fall als Äquivalent zur Windows Powershell aus dem vorherigen Kapitel verstanden werden kann.

Die ADB besteht laut Yener & Dundar (Expert Android Studio, 2016, S. 196,197) aus den folgenden Bestandteilen:

- Adb-Server
- Adb-Daemon

Nachdem nun auf einige Betriebssysteme eingegangen wurde, welche für die Verwendung auf dem Raspberry Pi entwickelt wurden, wird im nächsten Kapitel auf die darauf aufbauende Ebene der Microservices näher eingegangen.

3 MICROSERVICES

In diesem Kapitel geht es nun um den Einsatz von Microservices in Unternehmen. Es wird hierbei genauer erklärt worum es sich bei Microservices handelt, was deren Vor- sowie Nachteile sind und wie diese sinnvoll in Unternehmen eingesetzt werden können.

Bei Microservices handelt es sich um eine Softwarearchitektur, welche unter anderem dafür sorgt, dass einzelne Services unabhängig voneinander betrieben werden können. Hierbei ist es möglich Bestandteile von größeren Projekten auszutauschen oder weiterzuentwickeln ohne Gefahr das restliche Konstrukt der anderen Bestandteile dieser Software unbrauchbar zu machen. Jeder einzelne Service verfügt daher über definierte Schnittstellen und ist somit in der Lage autark den von ihm geforderten Dienst zu verrichten (Newman, 2016, S. 15-19). Im nächsten Abschnitt wird näher auf die grundlegende Architektur von Microservices eingegangen, sodass ein genaueres Bild über deren Eigenschaften vermittelt werden kann.

3.1 Grundlegende Architektur

Wie bereits zuvor erwähnt unterscheidet sich die grundlegende Architektur von Microservices stark von dem bekannten monolithischen Design. In Kapitel 3.2 wird näher auf diese Unterschiede eingegangen.

Ein Microservice besteht aus allen für seinen Dienst notwendigen Komponenten und ist – bis auf etwaige Kommunikation zwischen diesen – vollkommen unabhängig von anderen Services (Newman, 2016, S. 15-19). Das führt dazu, dass Microservices in den unterschiedlichsten Programmiersprachen programmiert und parallel zueinander betrieben werden können (Puglisi, 2015, S. 103). Weiters sind diese Dienste überschaubar konzipiert, sodass nur vergleichsweise wenige Codezeilen in deren Implementierung fließen. Meist ist es so, dass ein einzelner Microservice nur von einem kleinen Team implementiert und gewartet wird – das restliche Unternehmen wird über die Schnittstellen informiert und erhält somit sämtliche Informationen, welche wichtig sind um mit dem Dienst kommunizieren zu können. Näherer Einblick für Außenstehende ist bei dieser Art der Implementierung nicht notwendig. Man kann einen Microservice somit in gewisser Weise als eine Art Blackbox verstehen, welche bei einem definierten Input einen definierten Output liefert. Der Weg wie dieser Output generiert wird, ist für den Konsumenten irrelevant (Fowler, 2016, S. 28).

Microservices können natürlich für sich betrachtet auf diverse externe Ressourcen zugreifen. So ist es zum Beispiel möglich, dass ein Dienst, welcher dafür zuständig ist eine Zeitbuchung durchzuführen zuvor bei einem anderen Dienst nachfragt ob der gewünschte Vorgang vom Antragsteller erlaubt ist oder nicht. Erst nach positiver Rückmeldung des externen Prozesses erfolgt die tatsächliche Buchung – in Form des Aufrufs eines weiteren Dienstes oder zum Beispiel durch das direkte Speichern der Information in einer Datenbank, welche autark oder am selben Server betrieben werden kann. Ersteres hätte hier den Vorteil, dass sämtliche Prozesse auf den gleichen Datenbestand zugreifen können, letzteres dass der Dienst tatsächlich vollständig

unabhängig von externen Einflussfaktoren betrieben werden kann. Eine Mischung beider Möglichkeiten bietet hierbei zum Beispiel die Lösung mittels Docker-Containern, welche die Datenbank und den Service in einem gemeinsamen Container vereint – Näheres hierzu siehe Kapitel 4.2. Jeder Microservice sollte auf seine eigene für ihn spezifisch ausgewählte Datenbankarchitektur zurückgreifen damit etwaige Änderungen an der Datenbankstruktur für die Anpassung eines der Services keine Auswirkungen auf andere Services haben kann (Hofmann, Schnabel, & Stanley, 2017, S. 43).

Abbildung 7 zeigt einen beispielhaften Aufbau einer Microservices-Architektur (Richardson, Introduction to Microservices, 2015). Hierbei erfolgt der Zugriff auf die einzelnen Services mittels einer REST-Webservice-Implementierung – Näheres hierzu siehe Kapitel 3.3.2. Man erkennt, dass nicht näher auf die für die einzelnen Services verwendeten Programmiersprachen eingegangen wird, alles was für den Konsumenten des Dienstes zählt ist die Art wie der Dienst angesprochen werden soll. Weiters ist zu erkennen, dass die Dienste über unterschiedliche Endgeräte und Technologien angesprochen werden können. In der Abbildung wird hierbei auf ein eigenes API zurückgegriffen, welches den Zugriff mittels Smartphone erlaubt. Diese API lässt sich jedoch auch von anderen Endgeräten verwenden. Der Zugriff auf die Web-Oberflächen erfolgt im angeführten Beispiel mittels eines Desktop-PCs, welcher die Services über einen Webbrowser anspricht.

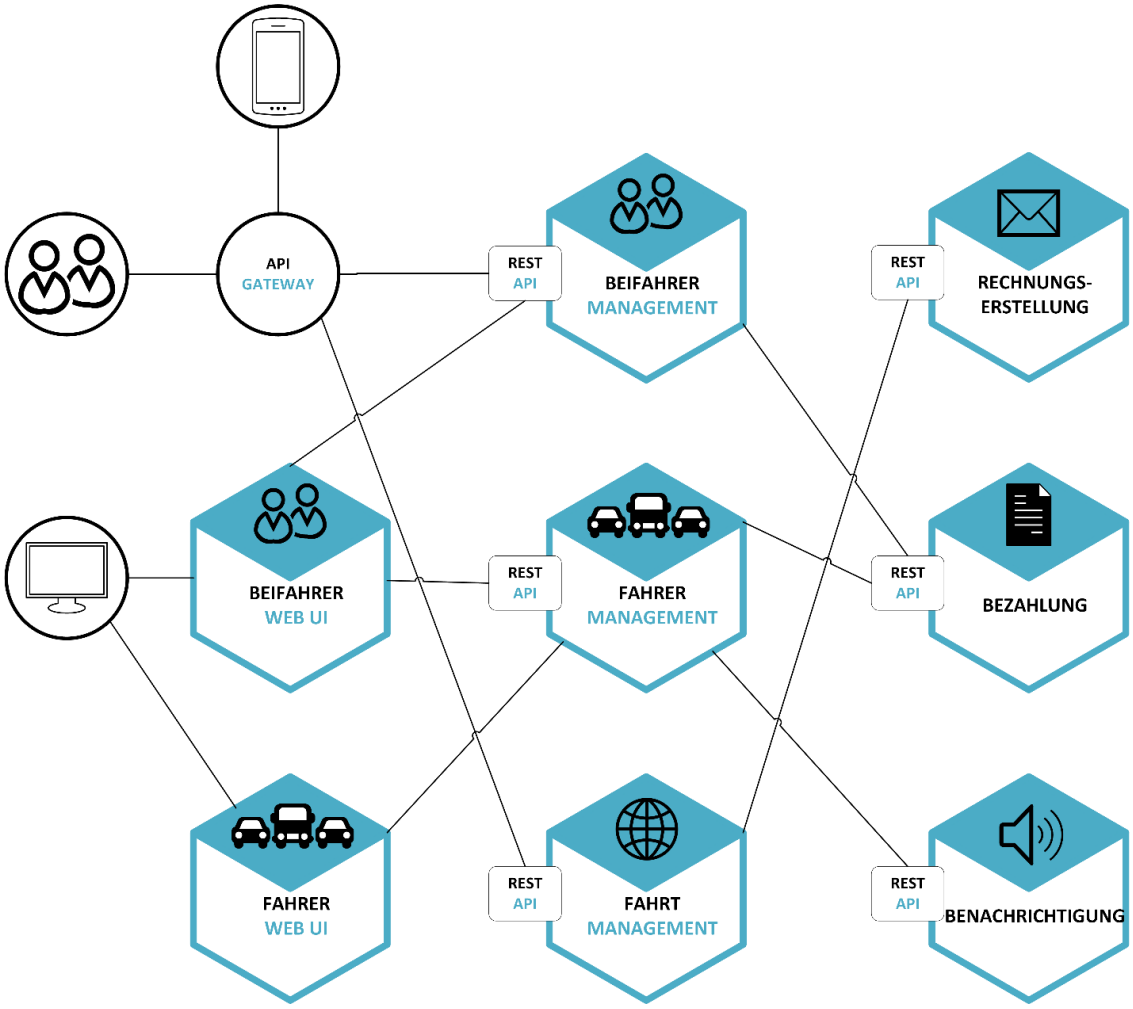

Abbildung 7: Beispielhafter Aufbau von Microservices (Richardson, Introduction to Microservices, 2015)

Dieses angeführte Beispiel zeigt die Vorteile, welche bei einer Implementierung mittels einer Microservice-Architektur existieren (Wolff, Microservices: Flexible Software Architecture, 2016):

- Die Skalierung von einzelnen Microservices kann unabhängig voneinander stattfinden, zum Beispiel, wenn die Funktion Fahrer-Management bedeutend öfter aufgerufen wird als die Funktion Bezahlung – Näheres hierzu siehe Kapitel 3.5.
- Veraltete Services lassen sich einfach gegen neue Services austauschen ohne Gefahr, dass die restlichen Services hierdurch beeinträchtigt werden könnten.
- Erweiterung vorhandener Systeme mit neuen Funktionen ohne den Quellcode des ursprünglichen Systems verändern zu müssen (altes System läuft weiter, neue Funktionen werden parallel entwickelt).
- Implementierung der Services kann von einzelnen Personen beziehungsweise kleineren Teams durchgeführt werden, da sich der Implementierungsaufwand für die Einzelkomponenten in Grenzen hält.

Weitere Vorteile sind laut Carneiro & Schelmer (Microservices From Day One: Build robust and scalable software from the start, 2016, S. 9-12) und Wolff (Microservices: Flexible Software Architecture, 2016):

- Es kann zu keinen unbeabsichtigten Abhängigkeiten zwischen den Services kommen, da diese nur über genau definierte Schnittstellen miteinander kommunizieren können. Eine etwaige Abhängigkeit kann somit nur beabsichtigt entstehen.
- Unabhängigkeit von Programmiersprachen. Existiert zum Beispiel ein Framework nur in einer bestimmten Programmiersprache kann der gewünschte Microservice in eben dieser Sprache implementiert werden, ohne die gesamte Entwicklung aller anderen Dienste auf diese Sprache umstellen zu müssen.
- Continuous Delivery kann im Vergleich zu monolithischem Design ohne größeren Aufwand gewährleistet werden, da es sich bei allen Bestandteilen nur um kleinere Programme handelt.

Nachteile von Microservices sind:

- Bei einer größeren Anzahl von Microservices erhöht sich der Aufwand für Softwaretests
- Bestehende Systeme sind nur mit viel Aufwand in Microservices änderbar, da sich die grundlegende Art der Kommunikation zwischen den Bestandteilen, sowie auch der involvierten Teams, ändern muss.
- Die Verteilung von Software kann durch die größere Anzahl von Microservices mehr Aufwand bedeuten als dies bei monolithischem Design der Fall wäre.
- Es muss dafür gesorgt werden, dass die Services auch ohne Kommunikation zu anderen Services weiterarbeiten können ohne das gesamte Konstrukt zu destabilisieren. Dies beinhaltet auch die Berücksichtigung von Signallaufzeiten oder Antwortzeiten. Wenn ein Microservice stark frequentiert ist und dadurch seine Antwortzeiten steigen kann dies ohne entsprechende Vorkehrungen dazu führen, dass der konsumierende Service in ein

Timeout verfällt und dieser seinen Dienst somit erst wieder aufnehmen kann, wenn der überbeanspruchte Dienst wieder in gewohntem Maß zur Verfügung steht.

Im nächsten Kapitel wird nun auf die Unterschiede zwischen der soeben beschriebenen Architektur der Microservices und dem altbekannten monolithischen Design eingegangen.

3.2 Unterschied zwischen monolithischem Design und Microservices

Nachdem im vorhergehenden Artikel die Vor- und Nachteile von Microservices aufgezeigt wurden soll in diesem Kapitel veranschaulicht werden, wie sich diese Art der Softwareentwicklung von bisher in vielen Unternehmen eingesetzten Monolithen unterscheidet.

Unter einem monolithischen Design versteht man die Entwicklung einer Applikation, welche alle ihre Funktionen und Bestandteile innerhalb eines einzelnen Projektes vereint (Annett, 2014). Möchte man hierbei einen Fehler beheben, neue Funktionen hinzufügen oder alte Funktionen entfernen ist es meist notwendig das gesamte Projekt neu zu compilieren und die alte in Betrieb befindliche Version der Software durch die neue Version zu ersetzen. Hierbei ist es notwendig die Applikation temporär stillzulegen, sodass diese ersetzt werden kann. Im Zeitraum des Updates ist es somit nicht mehr länger möglich etwaige Funktionen der Applikation zu nutzen, da diese vollständig deaktiviert sind. Hierbei gibt es jedoch Wege diesen Ausfall teilweise zu kompensieren indem zum Beispiel die Aufrufe in der Zwischenzeit gecached werden und im Anschluss an die erneute Inbetriebnahme der Reihe nach abgearbeitet werden – zum Beispiel über Message-Queues (Johansson, 2014). Hierfür muss die Applikation jedoch bereits auf diese komplexe Vorgehensweise hin entwickelt worden sein.

Eine solche frühe Entwicklung hin zur Kompensation von Ausfällen hilft zwar in diesen Fällen, verringert jedoch keineswegs die Komplexität des Monolithen, sondern trägt auch noch zu dieser bei. Moderne Software ist heute meist zu komplex als dass all ihre Bestandteile und deren Zusammenarbeit von einzelnen Personen oder auch ganzen Entwicklerteams verstanden werden können. Aufgrund ihrer enormen Größe ist es oft unmöglich ungewünschte Nebeneffekte vorherzusehen, welche bei der Änderung eines kleinen Teils des Programms auftreten können. Durch ihre kleine Größe bieten hierbei Microservices einen großen Vorteil, da sie dazu beitragen die Komplexität enorm zu verringern (Wootton, 2014). Etwaige Seiteneffekte sind somit weitestgehend ausgeschlossen. Auch die Entwicklung neuer Funktionen oder der Austausch vorhandener Funktionen kann ohne Ausfall der gesamten Applikation stattfinden, da nur der eine Dienst, welcher direkt betroffen ist, kurzzeitig beendet werden muss oder – je nach Aufbau – sogar im laufenden Betrieb ausgetauscht werden kann.

Ein großes Problem bei Monolithen ist außerdem oft die Lastverteilung sowie die Skalierbarkeit von Applikationen (Richardson, Monolithic Architecture pattern, 2017). Vor allem wenn Applikationen wachsen und somit komplexer werden, werden ihre Funktionen auch öfter konsumiert, was dazu führt, dass eine größere Last auf dem ausführenden Server verursacht wird als dies früher möglicherweise der Fall war. Bei einem Monolithen ist es in so einem Fall sehr schwer möglich Vorkehrungen zu treffen um auf einen solchen Lastanstieg zu reagieren. Neben leistungsfähigerer Hardware bleibt häufig kein weiterer Weg als eine Neuentwicklung oder

das Auslagern von Funktionen an zusätzliche Applikationen, welche auf anderen Servern ausgeführt werden – ein Schritt, welcher bereits in Richtung Microservices zeigt. Leistungsfähigere Hardware, welche jedoch nur in einem Bruchteil der Zeit voll ausgelastet wird, ist jedoch wirtschaftlich nicht sinnvoll. Fällt diese hohe Last jedoch nur an wenigen Stunden des Tages an müssen trotzdem unnötig Ressourcen hierfür bereitgehalten werden, welche in den anderen Stunden des Tages brachliegen. Somit fällt es bei einem Monolithen ebenfalls schwer zu skalieren. Die in Kapitel 3.4 beschriebene Lösung des Load Balancing erwähnt Richardson als möglichen Ausweg für eine bessere Skalierbarkeit, wobei hier nicht nur die Funktionen des Monolithen skaliert werden würden, welche häufig Verwendung finden, sondern der gesamte Monolith.

Aufgrund des Aufbaus von Microservices ist es hier bedeutend einfacher auf solche Spitzenauslastungen zu reagieren. Zusätzliche Performance in Form weiterer Server (oder in dieser Arbeit in Form von weiteren Raspberry Pi-Computern) kann dynamisch hinzugefügt oder entfernt werden, wenn diese nicht mehr benötigt wird. Dienste, welche selten beansprucht werden oder keine unmittelbare Rückmeldung benötigen können dank des Aufbaus auf leistungsschwächeren Clustern betrieben werden, während stärker frequentierte Dienste entsprechend leistungsfähig zur Verfügung gestellt werden können (Delgado, 2016).

Nachdem nun die Unterschiede zwischen monolithischem Design und der Microservice-Architektur beschrieben wurden, wird im nächsten Kapitel darauf eingegangen welche Möglichkeiten existieren um die Kommunikation zwischen einzelnen Microservices zu ermöglichen.

3.3 Webservice-Architektur

Für die Kommunikation zwischen den einzelnen Microservices existieren diverse Möglichkeiten. Diese unterscheiden sich unter anderem durch ihre Komplexität, den erzeugten Rechenaufwand, Erweiterbarkeit, sowie die grundlegende Ausrichtung auf ihren Haupteinsatzzweck. In diesem Kapitel werden die beiden bekanntesten Möglichkeiten – SOAP und REST beschrieben. Außerdem wird ein kurzer Einblick auf die neueste Technologie in diesem Bereich gegeben, dem Apache Thrift.

3.3.1 SOAP

Unter dem Simple Object Access Protocol versteht man die Kommunikation über einzelne Objekte, in die die gewünschte Anforderung in sich verpackt ist. Diese Verpackung erfolgt über das Extensible Markup Language (XML) Format, welches eine strukturierte Form der Anordnung von Informationen in Textform darstellt (Snell, Tidwell, & Kulchenko, 2001, S. 15). Hierbei existiert eine grundlegende Hierarchie, die die Zusammenhänge zwischen den angeführten Informationen vermittelt. Ein XML-Dokument beziehungsweise das XML-Paket, welches bei SOAP übermittelt wird, ähnelt also einem Baum (Ray E. T., 2003, S. 64-66). SOAP unterstützt neben der Übermittlung mittels XML-Dokumenten jedoch auch die Übermittlung mittels anderer Formate.

Listing 3-1 zeigt den beispielhaften Aufbau eines XML-Dokuments, das der Übermittlung von Buchungszeiten in einem Zeiterfassungssystem dient. Hierbei werden sämtliche Einträge auf hierarchisch gleicher Ebene direkt unter dem Buchungsverzeichnis selbst geführt. Jeder Eintrag enthält die für sich notwendigen Informationen.

```
<?xml version="1.0" encoding="UTF-8"?>
<buchungsverzeichnis>
    <eintrag id="1">
          <datum>01.01.2017</datum>
          <uhrzeit>06:00</uhrzeit>
          <benutzer>42</benutzer>
          <art>abfrage</art>
     </eintrag>
     <eintrag id="2">
          <datum>01.01.2017</datum>
          <uhrzeit>07:30</uhrzeit>
          <benutzer>16</benutzer>
          <art>kommen</art>
    </eintrag>
</buchungsverzeichnis>
```

Listing 3-1: Beispielhaftes XML-Paket

Wenn eine Kommunikation zwischen zwei automatisierten Systemen über XML-Dokumente stattfinden soll ist es wichtig die erhaltenen Informationen vor ihrer Verarbeitung auf ihre Gültigkeit zu überprüfen. Diese Überprüfung beinhaltet die Kontrolle der sogenannten Grammatik – also ob sich das Dokument an bestimmte Vorgaben hält – sowie die Analyse auf Wohlgeformtheit.

Unter wohlgeformt versteht man bei XML-Dokumenten laut Becher (XML: DTD, XML-Schema, XPath, XQuery, XSLT, XSL-FO, SAX, DOM, 2009, S. 20-24) insgesamt über 100 Regeln, welche eingehalten werden müssen. Folgender Auszug dieser Regeln soll als Veranschaulichung dienen:

- Jeder Start-Tag muss über ein Ende-Tag verfügen (<datum></datum> beziehungsweise bei einem leeren Element <datum/>) – wichtig ist hierbei auch, dass zwischen Groß- und Kleinschreibung unterschieden wird.
- Es darf nur ein einziges Wurzel-Element existieren (<buchungsverzeichnis></buchungsverzeichnis>).
- Attribute dürfen pro Element nur ein einziges Mal vorkommen und müssen mittels Hochkomma angegeben werden (<eintrag id=“1“>).
- Es ist möglich, dass mehrere Elemente ineinander verschachtelt werden (<datum> als Unterelement von <eintrag>), diese Elemente müssen jedoch mit dem Ende-Tag verschlossen werden bevor das Mutter-Element geschlossen wird.

Für die Kontrolle der Grammatik dient zum Beispiel die Dokumenttypdefinition (DTD) oder die XML Schema Definition (XSD). Neben diesen beiden Definitionen gibt es noch weitere Möglichkeiten der Überprüfung, wie die RELAX NG oder Schematron (Gulbransen, 2002, S. 408).

Listing 3-2 zeigt eine Dokumenttypdefinition für das in Listing 3-1 angeführte XML-Dokument. Hierbei wird das Dokument auf das Vorkommen aller angeführten Elemente überprüft und ob deren Werte im richtigen Format vorliegen. „#PCDATA“ steht für beliebigen Text, es darf sich jedoch nicht um weitere Unterelemente handeln. Somit wird hierdurch ein weiteres verschachteln unterhalb des Elements „datum“ verhindert. Der Wert „#IMPLIED“ gibt in diesem Beispiel an, dass das Attribut „id“ nicht unbedingt vorhanden sein muss – es ist somit optional anzugeben.

```
<!ELEMENT buchungsverzeichnis (
<!ELEMENT eintrag (
<!ELEMENT datum ( #PCDATA ) >,
<!ELEMENT uhrzeit ( #PCDATA ) >,
<!ELEMENT benutzer ( #PCDATA ) >,
<!ELEMENT art ( #PCDATA ) > ) >
<!ATTLIST eintrag
  id CDATA #IMPLIED > ) >
<!ATTLIST buchungsverzeichnis
  id CDATA #IMPLIED >
```

Listing 3-2: Beispielhafte Dokumenttypdefinition

Listing 3-3 dient als Beispiel für eine XSD Datei, welche das in Listing 3-1 angeführte XML-Dokument auf seine Grammatik kontrollieren kann. Hierbei wird unter anderem überprüft ob alle benötigten Elemente im XML Dokument in der richtigen Anzahl enthalten sind und ob diese den Vorgaben in Form ihres gewünschten Datentyps entsprechen (Skonnard, 2003). Das Schema sieht in diesem Fall vor, dass mindestens ein Eintrag enthalten sein muss. Eine maximale Anzahl ist jedoch nicht vorgesehen. Weiters ist die Angabe des Attributs „id“ im Element „eintrag“ optional, muss jedoch, sofern es vorhanden ist, vom Typ Integer sein. Man könnte somit im XML Schema festlegen, dass ein XML Dokument mit dem Attribut „id“ und dem Wert „first“ automatisch verworfen wird, da das Dokument falsche Informationen enthält. Man sieht an diesem Beispiel, dass es mit XSD möglich ist genauere Einschränkungen beziehungsweise Vorgaben zu definieren als dies mit DTD allein der Fall wäre.

```
<xs:schema attributeFormDefault="unqualified"
elementFormDefault="qualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="buchungsverzeichnis">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="eintrag" maxOccurs="unbounded" minOccurs="1">
          <xs:complexType>
            <xs:sequence>
              <xs:element type="xs:string" name="datum"/>
              <xs:element type="xs:string" name="uhrzeit"/>
              <xs:element type="xs:byte" name="benutzer"/>
              <xs:element type="xs:string" name="art"/>
            </xs:sequence>
            <xs:attribute type="xs:integer" name="id" use="optional"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Listing 3-3: Beispielhafte XML Schema Definition

Neben der Möglichkeit Informationen direkt auszutauschen existiert mittels SOAP auch noch die Möglichkeit des Aufrufs von entfernten Prozeduren, dem Remote Procedure Call (RPC). Hierbei wird vom Client eine Verbindung zum Server aufgebaut und diesem wird mitgeteilt welche Funktion mit welchen Parametern ausgeführt werden soll (Laurent, Johnston, & Dumbill, 2001, S. 1-4). Der Server führt anschließend die entsprechenden Routinen aus und liefert die benötigten Informationen zurück. Nachdem der Client diese Informationen empfangen hat kann er dann mit dem weiteren Ablauf seiner Programminstruktionen fortfahren. Es wird somit tatsächlich ein Funktionsaufruf an einen externen Server ausgelagert, der hierfür möglicherweise besser ausgestattet ist – sei es betreffend seiner Rechenleistung oder auch seiner Sicherheitsarchitektur. Denkbar wäre hier zum Beispiel, dass sich der Server in einer hoch sicheren Umgebung befindet und nur wenige spezielle Funktionen der Außenwelt über eine Firewall zur Verfügung gestellt werden. Somit würde ein Kompromittieren des Clients nicht dazu führen, dass man auf mehr Informationen zugreifen kann als vom Entwickler vorgesehen.

In Abbildung 8 ist der grundlegende Aufbau einer SOAP-Anfrage zu sehen. Alle Anfragen werden als sogenannter SOAP-Envelope verpackt und beinhalten den SOAP-Header, sowie den SOAP-Body. Im SOAP-Header befinden sich grundlegende Instruktionen. Der SOAP-Body enthält Instruktionen wie zum Beispiel der Name der Prozedur, welche ausgeführt werden soll, sowie die zugehörigen Parameter des Funktionsaufrufs. In der Response-Nachricht wird der SOAP-Body im Anschluss an diese Anfrage mit Daten befüllt und enthält am Ende die entsprechenden Informationen, welche angefordert wurden (Burghardt, 2013, S. 51).

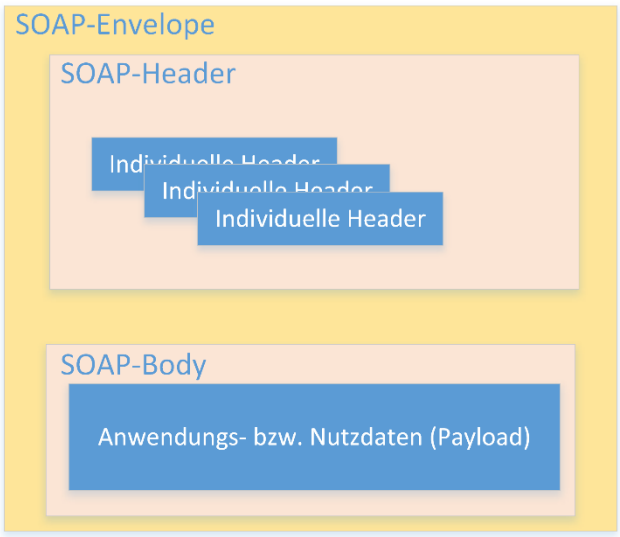

Abbildung 8: Aufbau einer SOAP-Envelope (Burghardt, 2013, S. 51)

In Listing 3-4 ist nun ein beispielhafter Funktionsaufruf zu sehen, bei dem das zuvor gezeigte SOAP-Envelope mit Daten befüllt ist. Das Beispiel zeigt einen RPC mit leerem Header und der Funktion „BuchungenVonBenutzer“ als aufzurufende Funktion. Als Parameter wird der Wert „42“ übergeben.

```
<?xml version="1.0"?>
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope">
    <s:Body>
        <m:BuchungenVonBenutzer xmlns:m="http://localhost/soap">
            42
        </m:BuchungenVonBenutzer>
    </s:Body>
</s:Envelope>
```

Listing 3-4: Beispielhafter Funktionsaufruf mittels SOAP

Die Antwort aus Listing 3-5, welche vom Server im Anschluss an die Anfrage geliefert wird, enthält im SOAP-Header einen eindeutigen Identifier für die Anfrage, sowie die angeforderten Informationen in Form der gespeicherten Einträge für den Benutzer mit der Id „42“.

```
<?xml version="1.0"?>
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope">
    <s:Header>
        <m:RequestID
xmlns:m="http://localhost/soap">f382ac9011</m:RequestID>
    </s:Header>
    <s:Body>
        <m:Response xmlns:m="http://localhost/soap">
            <m:id value="42">
                <m:eintrag>01.01.2017 06:00 - Abfrage</m:eintrag>
            </m:id>
        </m:Response>
    </s:Body>
</s:Envelope>
```

Listing 3-5: Beispielhafte Antwort mittels SOAP

Die Übermittlung all dieser Informationen kann über ein beliebiges Transportprotokoll erfolgen. In den Beispielen handelt es sich hierbei um das Hypertext Transport Protocol (HTTP). SOAP ist jedoch nicht auf die Übermittlung mittels Transmission Control Protocol/Internet Protocol (TCP/IP) und HTTP beschränkt, sondern kann auch zum Beispiel über das Simple Mail Transfer Protocol (SMTP) übermittelt werden (Skonnard, Understanding SOAP, 2003). Die Übermittlung kann außerdem über mehrere Stationen erfolgen, welche jeweils mit unterschiedlichen Protokollen arbeiten. Das bedeutet, dass man zum Beispiel eine Nachricht mittels HTTP absendet, diese ab der ersten Zwischenstation mittels SMTP weiter übermittelt wird und bei der letzten Zwischenstation dann abschließend mittels File Transfer Protocol (FTP) zum Server gesendet wird. Die Antwortnachricht kann dann äquivalent in umgekehrter Richtung zurückgesendet werden – oder wenn erforderlich über andere Protokolle.

SOAP ist auch dazu in der Lage Nachrichten über sichere Kanäle zu versenden, sodass diese nicht von Dritten mitgelesen werden können. Hierbei ist es zum Beispiel möglich das Hypertext Transfer Protocol Secure (HTTPS) zu verwenden (Newcomer, 2002, S. 224). Zur Wahrung der Integrität einer Nachricht sowie ihrer Verschlüsselung gibt es außerdem den WS-Security Standard, welcher hierfür entwickelt wurde (Morrenth, 2014, S. 49-51).

Nachdem nun die Eigenschaften von SOAP erklärt wurden, wird im nächsten Abschnitt näher auf die REST Architektur eingegangen, welche als einfachere Alternative zu SOAP gilt.

3.3.2 REST

In diesem Kapitel wird nun die Representational State Transfer Architektur (REST) genauer beschrieben, bei welcher es sich, im Gegensatz zu SOAP aus dem vorherigen Kapitel, um keinen offiziellen Standard handelt, sondern mehr um einen nicht näher durch Spezifikationen definierten Architekturstil. Während es sich also bei REST um keinen Standard handelt, greift die Architektur selbst sehr wohl auf Standards zurück um die gewünschte Funktionsweise sicherzustellen.

Die Funktionalität von REST war prinzipiell bereits in den frühen Jahren des Internets gegeben, wurde jedoch erst später durch Fielding (Architectural Styles and the Design of Network-based Software Architectures, 2000) genauer definiert und niedergeschrieben. Fielding hat hierbei die Machine-to-Machine Kommunikation näher betrachtet und die ihr zugrunde liegenden Eigenschaften und Voraussetzungen genauer festgelegt. Hierbei beschrieb er die Verwendung von eindeutigen Identifizierungszeichen und die Verwendung von einer begrenzten Menge an Befehlen (Webber, Parastatidis, & Robinson, 2010, S. 12).

Durch diesen grundlegenden Aufbau wollte er eine Definition erarbeiten durch welche die Kommunikation zwischen vernetzten Informationssystemen über simple Wege möglich ist. Wenn man dieses Prinzip mit den Anforderungen von Microservices vergleicht stellt sich schnell heraus, dass das Konzept von REST diese Anforderungen erfüllt und es somit mit diesem bereits einige Jahrzehnte alten Ansatz möglich ist vergleichsweise neue Designphilosophien umzusetzen.

Hinter dem Begriff REST steht die Funktionalität der Datenüberführung von einem Zustand in den nächsten. Diese Zustandsänderung erfolgt mit Hilfe der empfangenen Informationen über das verwendete Protokoll. Voraussetzung für eine solche Möglichkeit der automatisierten Verarbeitung der empfangenen Informationen ist jedoch, dass diese in einer Form aufbereitet sein müssen, welche den Empfänger vor keine Interpretationsprobleme dieser Daten stellt. Das bedeutet, dass sämtliche Daten, welche über REST zur Verfügung gestellt werden einer zuvor genau definierten Struktur entsprechen müssen. Eine dynamische Änderung der Inhalte, welche die automatisierte Interpretierbarkeit der Daten verschlechtert, ist somit nicht erlaubt. Die Übermittlung einer Website, welche das aktuelle Datum zur Verfügung stellt ist somit REST-konform, solange diese das Datumsformat nicht beliebig verändert. Ein REST-Service könnte zum Beispiel bei der Interpretation des Formats „01.03.2017“ nicht plötzlich auf das Format „1.März 2017“ wechseln, da dies unweigerlich zu Kommunikationsschwierigkeiten führen würde. Grundsätzlich benötigt die Machine-to-Machine Kommunikation also immer verlässliche Formate in denen die Daten zur Verfügung gestellt werden. Weiters erfolgt die Kommunikation mittels REST zustandslos. Das bedeutet, dass die einzelnen Kommunikationspartner keine Zustandsinformationen speichern müssen, sondern sämtliche Informationen, die für den Wechsel eines Zustands benötigt werden in der Nachricht übermittelt werden. Um hier auf das vorhergehende Beispiel des Datums Bezug zu nehmen würde dies bedeuten, dass es mittels REST nicht erlaubt ist das Jahr getrennt von den Informationen Monat, sowie den Tag zu übertragen, sofern diese Informationen vom Client über den selben Aufruf angefordert werden, jedoch der Server diese Informationen in alternierender Reihenfolge zurück liefert. Eine solche Art der Übertragung würde eine Speicherung der ersten Information (des Jahres) sowie die Speicherung des Zustands der Übertragung erfordern (wenn die letzte Übertragung das Jahr war, dann sende mir nun das Monat und den Tag) bevor die zweite Information angefordert wird.

Aufgrund dieser zustandslosen Kommunikation ist es mit relativ einfachen Mitteln möglich einen Service, welcher auf REST beruht, zu skalieren. Da keinerlei Informationen am Server über die Kommunikation selbst gespeichert werden muss ist es für den jeweiligen Client irrelevant mit welchem Server dieser tatsächlich kommuniziert. Auch direkt aufeinander folgende Abfragen können somit innerhalb weniger Sekunden von unterschiedlichen Servern beantwortet werden, sofern diese auf die gleichen Algorithmen sowie den selben Datenspeicher zurückgreifen.

Fielding (Architectural Styles and the Design of Network-based Software Architectures, 2000, S. 76-85) beschrieb insgesamt sechs verschiedene Eigenschaften, welche ein Service bieten muss um der REST Architektur zu entsprechen, wobei die letzte angeführte Eigenschaft als optional gilt:

- Client-Server

  Ein REST-Service muss der Client-Server-Architektur entsprechen, was bedeutet, dass der Server eine Ressource zur Verfügung stellt, welche von einem oder mehreren Clients angesprochen werden kann.

- Zustandslosigkeit

  Es ist nicht notwendig den Zustand der Datenübertragung auf Seite des Servers zu speichern. Jede Anfrage des Clients enthält alle notwendigen Informationen, welche der Server wissen muss um diese Anfrage zu beantworten. Es ist somit nicht möglich, dass ein Client sich mit einer Anfrage auf eine vorhergehende eigene Anfrage bezieht ohne diese zuvor angeforderten Informationen erneut zu übermitteln. Wie im vorherigen Abschnitt angesprochenen Beispiel kann ein REST-Service somit seine gelieferten Informationen nur von den übermittelten Parametern der Anfrage abhängig machen, jedoch nicht zum Beispiel von dem in einer zuvor abgesetzten Anfrage angeforderten Datumsformat.

- Zwischenspeicher (Cache)

  Der REST-Service kann bei der Übermittlung der angeforderten Informationen angeben ob diese vom Client gecached werden dürfen oder nicht. Wenn dies möglich ist kann der Client somit bei seiner nächsten Anforderung zuvor überprüfen ob er die erforderlichen Informationen bereits im Cache hat. Ist dies der Fall entfällt die Notwendigkeit einer erneuten Anforderung der Informationen. Dies führt dazu, dass der Server weniger stark belastet wird, da die Anzahl der Anfragen minimiert wird und dazu, dass der Client aufgrund des geringeren Kommunikationsbedarfs seinen Programmcode schneller abarbeiten kann ohne auf Antworten zu warten. Im schlimmsten Fall kann dies jedoch dazu führen, dass der Client mit Informationen weiterarbeitet, welche in dieser Form nicht erneut vom Server gesendet worden wären, da dieser möglicherweise bereits über neuere Informationen verfügt hätte.

- Einheitliche Schnittstellen

  Die Zurverfügungstellung der einzelnen REST-Services erfolgt über einheitliche Schnittstellen, welche von diversen Komponenten angesprochen werden können. Diese Art der Kommunikation hat den Vorteil, dass die Kommunikation zwischen den Services möglich ist, führt jedoch auch dazu, dass die Kommunikation selbst aufwändiger ist als dies durch programmspezifische Schnittstellen der Fall wäre. Diese Abstraktion der programmspezifischen Aufrufe führt dazu, dass die von Microservices geforderte Eigenschaft der Austauschbarkeit erfüllt wird. Eine solche einheitliche Schnittstelle kann somit einfach 1:1 ausgetauscht werden ohne die restlichen Komponenten im System zu

beeinträchtigen. Um all dies jedoch zu ermöglich müssen die einheitlichen Schnittstellen weitere Eigenschaften erfüllen:

- Kennzeichnung von Ressourcen

  Jede Schnittstelle muss in Form eines Uniform Resource Identifier (URI) im Netz zur Verfügung gestellt werden. Unter dem URI versteht man eine eindeutige Adresse der angebotenen Ressource im Netzwerk. Eine solche Adresse wird in Abbildung 9 veranschaulicht (Berners-Lee, Fielding, & Masinter, 2005).

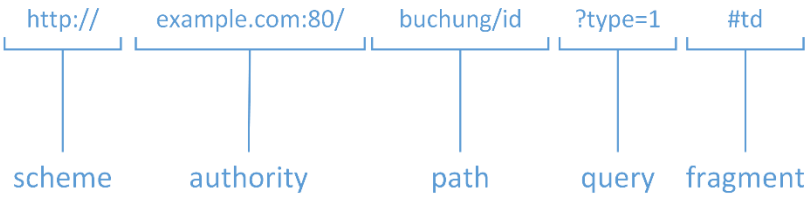

Abbildung 9: Uniform Resource Identifier (URI) Beispiel (Berners-Lee, Fielding, & Masinter, 2005)

- Repräsentation zur Veränderung von Ressourcen

  Es ist möglich bei der Anforderung von Informationen durch den Client mit anzugeben in welcher Form dieser die Informationen gerne entgegennehmen würde. Hierbei kann zum Beispiel angegeben werden, dass der Server die Informationen in Form eines XML-Dokuments oder in Form einer JavaScript Object Notation (JSON)-Nachricht übermitteln soll. Auch das Festlegen einer gewünschten Sprache ist hiermit möglich.

- Selbstbeschreibende Nachrichten

  Die Nachricht selbst enthält alle notwendigen Informationen. Dies beinhaltet die zuvor beschriebene Information darüber ob die enthaltenen Informationen gecached werden dürfen oder nicht oder wie die Daten codiert sind.

- Hypermedia as the Engine of Application State (HATEOAS)

  Die Navigation der einzelnen REST-konformen Services erfolgt ausschließlich über vom Server zur Verfügung gestellte URIs.

- Mehrschichtiges System

  Der Service soll mehrschichtig aufgebaut sein, was dazu führt, dass die Komplexität vom Client verborgen bleibt. Dieser fordert nur eine bestimmte Information über eine Ressource an. Im Hintergrund arbeitet der REST-Service über mehrere Schichten um an die angeforderten Informationen zu gelangen. Diese Schichten können im Beispiel von Microservices auch andere Services sein, welche befragt werden. Dadurch ist es zum Beispiel möglich Dienste, welche seltener benutzt werden auf einem gemeinsamen Server zusammen zu fassen, während stark frequentierte Dienste in mehrfacher Ausführung auf dezidierter Hardware zur Verfügung gestellt werden. Dieses mehrschichtige System hilft somit dabei eine Skalierbarkeit zu gewährleisten – Näheres hierzu siehe Kapitel 3.5.

- Code-On-Demand

  Fielding beschreibt außerdem die Möglichkeit der Übertragung von Quellcode zur Laufzeit des Programms. Hierbei werden also nicht nur Informationen in Form von Text übertragen, sondern ausführbarer Code, welcher später vom Client ausgeführt wird. Dieser Code kann zum Beispiel JavaScript Code sein, welcher nur in bestimmten Fällen tatsächlich benötigt wird. Somit kann der Client vergleichsweise klein gehalten werden und nur wenn die Funktionalität tatsächlich in Anspruch genommen wird, wird die hierfür notwendige Ressource übertragen und ausgeführt. Unter anderem aufgrund der Sicherheitsrisiken dieses Vorgehens gilt diese Eigenschaft jedoch als optional. Eine solche Vorgehensweise sollte laut Fielding nur in Unternehmensnetzwerken durchgeführt werden. Hierbei kann der Download von fremdem Code von externen Ressourcen über eine Firewall unterbunden werden.

In Listing 3-6 ist nun ein beispielhafter Funktionsaufruf zu sehen, bei dem der zur Verfügung gestellte REST-Service konsumiert wird. Hierbei wird der Service „BuchungenVonBenutzer“ aufgerufen und der Parameter „42“ übergeben.

```
GET http://localhost/rest/BuchungenVonBenutzer/42
```

Listing 3-6: Beispielhafter Funktionsaufruf mittels REST

Abbildung 10 zeigt ein Beispiel für die REST-Architektur. Hierbei handelt es sich wie bereits bei den Beispielen in den vorherigen Kapiteln um ein Zeitbuchungssystem. Es ist möglich mit der Übermittlung einer Mitarbeiter-Id sämtliche oder nur einzelne Buchungen eines Mitarbeiters einzusehen. Weiters kann man Listen anfordern, welche alle Buchungen eines Tages enthält oder eine Zeitguthaben-Abfrage eines Mitarbeiters anfordern, welche wiederum die notwendigen Informationen von einem anderen Service anfordert und somit das Guthaben berechnet. Hierbei ist zu sehen, dass die zuvor angeführten Eigenschaften vorhanden sind. Es handelt sich um ein Client-Server Konzept bei dem es nicht notwendig ist Zustände zu speichern. Bei jeder einzelnen Anfrage werden alle benötigten Informationen erneut übermittelt. Es ist nicht notwendig einen bestimmten Service zuvor aufzurufen bevor man einen nachfolgenden Service aufruft, sofern man alle benötigten Informationen bereits weiß. Es ist außerdem möglich, dass der Client Informationen zu einzelnen Buchungen zwischenspeichert, da sich diese nicht mehr ändern können. Führt ein Mitarbeiter nun eine Abfrage aus kann der Client für einzelne Buchungen auf die Nachfrage beim Server verzichten, da er die notwendigen Informationen bereits besitzt. Die einheitlichen Schnittstellen sorgen dafür, dass die Dienste untereinander kommunizieren können. Diese bauen somit ein mehrschichtiges System auf, welches dazu in der Lage ist einzelne Schichten auszutauschen. Hierdurch ist es auch möglich öfter konsumierte Aufrufe, wie zum Beispiel Abfragen eines Mitarbeiters, auf ein leistungsstärkeres Cluster auszulagern. Andere Auswertungen, wie zum Beispiel Gesamtauswertungen über einen Wochentag, können von schwächeren Rechnern durchgeführt werden, da bei diesen Auswertungen keine direkte Benutzerinteraktion stattfindet und es somit irrelevant ist ob diese innerhalb weniger Millisekunden oder erst nach einigen Sekunden vorliegen.

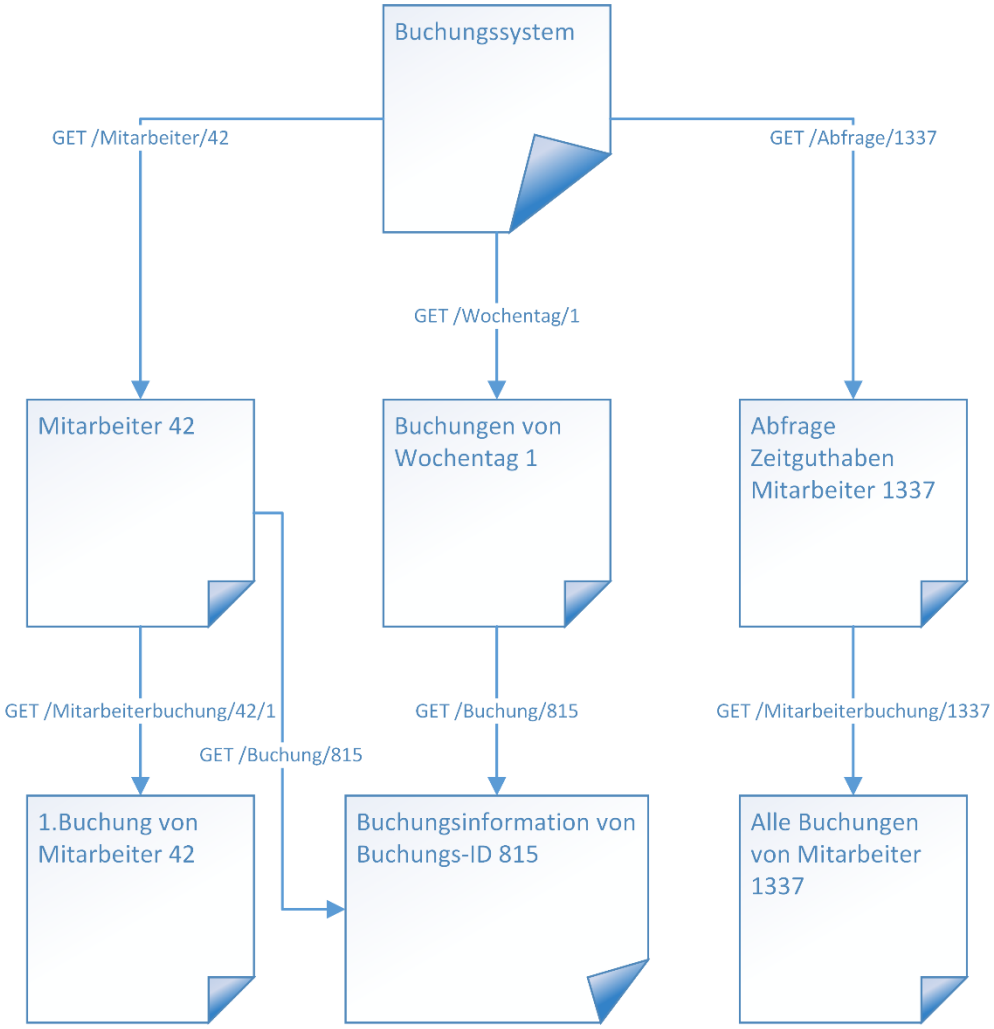

Abbildung 10: Beispiel einer REST-Architektur

Nachdem nun die beiden bereits etwas älteren Prinzipien von Webservices nähergebracht wurden, wird im nächsten Kapitel auf das erst kürzlich veröffentlichte Thrift näher eingegangen.

3.3.3 Apache Thrift

Die dritte Möglichkeit einer Webservice-Architektur auf welche in dieser Arbeit eingegangen werden soll ist Apache Thrift. Hierbei handelt es sich um eine von Facebook im Jahr 2007 unter dem Namen Thrift entwickelte Möglichkeit Services zu generieren, welche programmiersprachen-unabhängig miteinander kommunizieren können. Heute ist Apache Thrift als Open Source verfügbar (Rakowski, 2015, S. 1).

Zu den unterstützten Programmiersprachen zählen unter anderem C, C++, C#, Go, Haskell, Java, Node.js, Objective-C, Perl oder PHP. Wobei nicht jede Sprache sämtliche Funktionen unterstützt (Prunicki, Apache Thrift, 2009).

Im Gegensatz zu den zuvor beschriebenen Lösungen SOAP und REST unterstützt Thrift standardmäßig nur einen einzigen Service pro Server. Dieser Service wird, wie in Listing 3-7 veranschaulicht, mittels einer Interface Definition Language (IDL) beschrieben und im Anschluss daran von Thrift für die gewünschten Programmiersprachen generiert. Hierbei kümmert sich Thrift

um sämtliche Eigenheiten jeder Programmiersprache. Dies betrifft unter anderem Neuinitialisierungen von Datentypen, welche zum Beispiel in PHP: Hypertext Preprocessor (PHP) möglich sind, jedoch in C++ nicht vorgesehen. Bei ersterer Programmiersprache ist es somit möglich eine Variable, welche zuvor als „String“ definiert wurde später als „Integer“ weiterzuverwenden. Thrift nimmt auf diese und andere Eigenheiten Rücksicht und generiert dementsprechend die Vorlagen für den Client und den Server. Der Entwickler hat nur noch die Aufgabe die definierten Funktionen auszuformulieren, sodass die gewünschte Funktionalität hergestellt wird.

```
service BuchungenVonBenutzer 42 {
    results find(1: string name)
}
```

Listing 3-7: Beispielhafte Funktionsdefinition mittels Thrift

In Abbildung 11 ist der Aufbau einer Apache Thrift Client/Server Architektur zu sehen. Hierbei erkennt man die von Thrift generierten Bestandteile und wie diese aufeinander aufbauen. In den nächsten Abschnitten wird auf jeden einzelnen Bestandteil näher eingegangen.

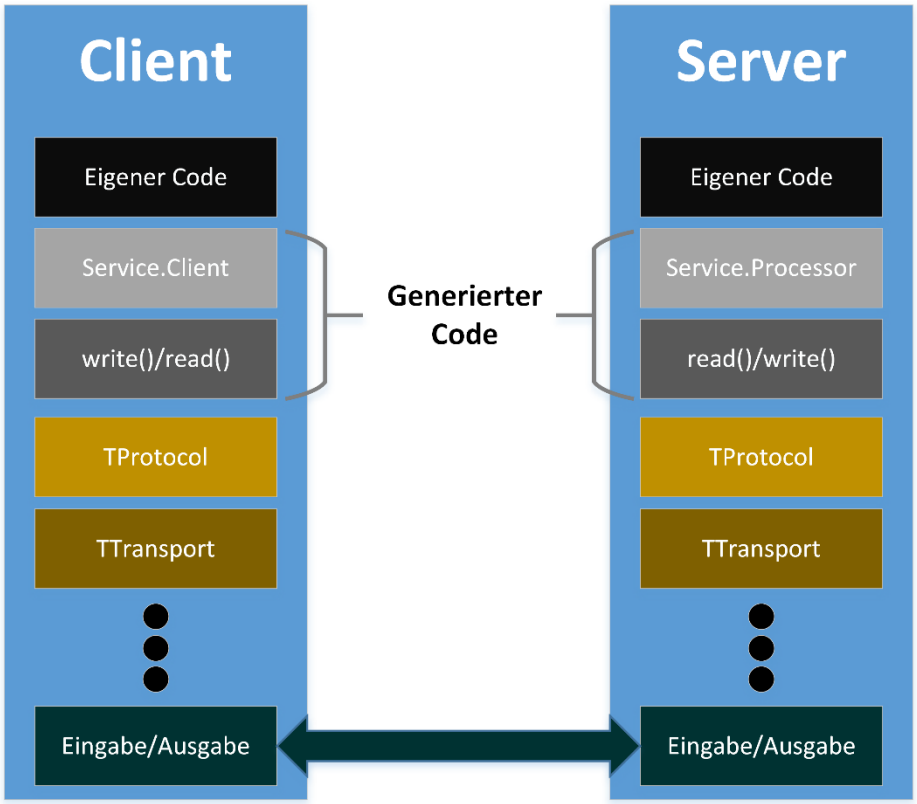

Abbildung 11: Thrift Architektur (Prunicki, Apache Thrift, 2009)

Wie schon bei den zuvor beschriebenen Webservice-Architekturen existieren auch bei Thrift mehrere unterstützte Protokolle, welche für den Informationsaustausch genutzt werden können. Unter anderem stehen folgende Protokolle zur Verfügung:

- TBinaryProtocol – Die zu übertragenen Daten werden binär codiert. Dies führt dazu, dass der Datenaustausch ohne großen Rechenaufwand stattfinden kann. Binäre Daten sind jedoch vom Menschen nicht lesbar und somit ist es schwierig etwaige Fehler zu finden. Dieses Protokoll ist laut Rakowski (Learning Apache Thrift, 2015, S. 57) im Allgemeinen den anderen Protokollen vorzuziehen, da es das universellste Protokoll darstellt.
- TCompactProtocol – Im Vergleich zur binären Übertragung spart dieses Protokoll Speicherplatz, da Thrift die Daten optimiert (Rakowski, 2015, S. 57).
- TDebugProtocol – Dient hauptsächlich zum Debuggen von Informationen, da die übertragenen Daten vom Menschen lesbar bleiben und somit von diesem leichter interpretiert werden können.
- TJSONProtocol – Die Daten werden in Form eines JSON-Objektes übermittelt.

Je nach gewählter Programmiersprache existieren für Thrift mehrere Möglichkeiten des Informationstransports zwischen Client und Server (Rakowski, 2015, S. 55-56):

- TBufferedTransport – Die übertragenen Daten werden gepuffert übertragen. Das bedeutet, dass mehrere Pakete gemeinsam hin- und zurückgeschickt werden können.
- TFileTransport – Die Übermittlung erfolgt mittels Dateien.
- TPHPStream – Ein Beispiel für einen programmiersprachenspezifischen Transport. Hierbei wird ein bereits vorhandener Server, welcher PHP unterstützt für den Transport benutzt ohne einen eigenen Server starten zu müssen.
- TSocket – Es wird die Übermittlung mittels eines einzelnen geöffneten Sockets durchgeführt. Dies führt dazu, dass nur eine einzige gleichzeitige Verbindung möglich ist, da sämtliche parallele Übertragungen blockiert werden.

Neben dem zuvor beschriebenen TPHPStream, welcher auf vorhandene Serverkomponenten zurückgreift um Daten zur Verfügung zu stellen, existieren bei Apache Thrift drei verschiedene Möglichkeiten für das Ausführen einer Serveranwendung (Rakowski, 2015, S. 58):

- TSimpleServer – Dieser Servertyp kann nur eine einzelne Verbindung gleichzeitig aufbauen und wird daher hauptsächlich für die Entwicklung sowie Tests benutzt.
- TNonblockingServer – Dieser Servertyp unterstützt mehrere gleichzeitige Verbindungen und greift auf mehrere Threads zurück um seine Dienste zur Verfügung zu stellen.
- TThreadPoolServer – Dieser Servertyp unterstützt ebenfalls mehrere gleichzeitige Verbindungen, und greift auf mehrere Threads zurück um seine Dienste zur Verfügung zu stellen. Er bietet einen größeren Datendurchsatz, erkauft diesen jedoch durch einen viel höheren Ressourcenverbrauch.

Nachdem nun auf verschiedene Möglichkeiten der Realisierung von Microservices eingegangen wurde, wird im nächsten Kapitel erläutert, welche Auswirkung diese Varianten auf die Lastverteilung von Microservices haben und wie diese Lastverteilung aussehen kann.

3.4 Load Balancing

Ein großer Vorteil von Microservices ist die Möglichkeit der Lastverteilung durch den Einsatz mehrerer parallel geschalteter Server, unter welchen die ankommenden Anfragen aufgeteilt werden können. Dies führt zu Ausfallsicherheit, da bei Verlust eines Servers die Anfragen an andere beteiligte Server weitergeleitet werden können.

Um eine solche Lastverteilung zu ermöglichen existieren mehrere Möglichkeiten. In diesem Kapitel soll auf zwei dieser Möglichkeiten genauer eingegangen werden:

- Load Balancer – Es wird ein Server vorgeschaltet, welcher die eintreffenden Anfragen analysiert, gegebenenfalls mit Informationen anreichert und an einen von mehreren Servern zur weiteren Abarbeitung weiterleitet (Bourke, 2001, S. 3-4).
- Domain Name System (DNS) Server – Jener Server, welcher für die Auflösung der Internet Protocol (IP)-Adresse zuständig ist, liefert in alternierender Reihenfolge eine der möglichen IP-Adressen, sodass die unterschiedlichen eingehenden Anfragen gleichmäßig auf die Server verteilt werden (Membrey, Plugge, & Hows, Practical Load Balancing: Ride the Performance Tiger, 2012, S. 53-55).

Beim Load Balancer dient die Anreicherung mit weiteren Informationen dazu, dass aufeinander folgende Anfragen eines Benutzers, soweit möglich, an denselben Server weitergeleitet werden können (Bourke, 2001, S. 3-4). Diese Vorgehensweise hat den Vorteil, dass etwaig notwendige Schlüsselaustausch-Szenarien für die Verschlüsselung der Kommunikation minimiert werden können, da der Handshake aufgrund der Kommunikation mit demselben Server bereits stattgefunden hatte und somit direkt kommuniziert werden kann, während dies bei unterschiedlichen Servern zuerst von neuem beginnen müsste. Ein Load Balancer mit SSL Offloading hat außerdem die Möglichkeit die Pakete selbst einzusehen um diese im Anschluss daran an einen der Server weiterzuleiten. Der Client spricht in diesem Fall direkt mit dem Load Balancer und nicht mit den Servern dahinter (Assmann, 2013) – Näheres zum Thema Sicherheit der Kommunikation siehe Kapitel 4.1.1.

In Abbildung 12 wird der Aufbau eines typischen Netzwerks mit Load Balancer veranschaulicht, wobei im gezeigten Beispiel insgesamt 3 Server zur Verfügung stehen, welche die jeweiligen Microservices anbieten. Der vorgeschaltete Load Balancer verschleiert hierbei die Information über die insgesamt vorhandenen Server. Für den Konsumenten des Service gibt es nur einen Single Point of Contact. Der Load Balancer hat hierbei laut Bourke (Bourke, 2001, S. 4) mehrere Aufgaben zu erfüllen:

- Abfangen der Anfragen, welche für einen bestimmten Service abgesetzt werden
- Entscheiden welche Anfrage an welchen Server weitergeleitet wird
- Führen einer Liste über die vorhandenen Server inklusive ihrer angebotenen Services
- Überprüfen der Erreichbarkeit von einzelnen Servern, sodass diese bei Nichterreichbarkeit aus der Liste entfernt werden

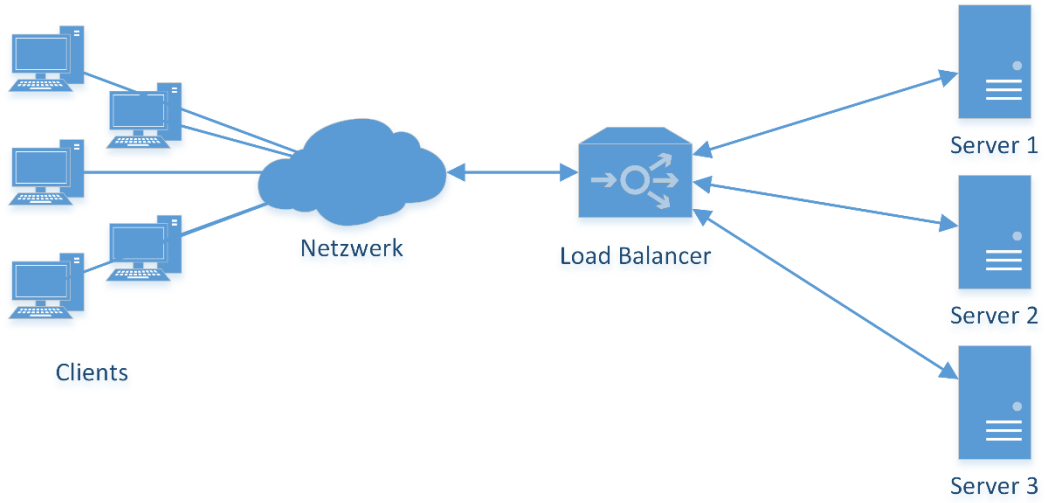

Abbildung 12: Veranschaulichung eines Netzwerks mit Load Balancer (Kopparapu, 2002)

Damit jedoch der Client beziehungsweise der Load Balancer über die Anzahl der zur Verfügung stehenden Server und Services Bescheid weiß gibt es mehrere Möglichkeiten (Morris, 2016, S. 91):

- Client-Side Discovery
- Server-Side Discovery

Bei der Client-Side Discovery ist es dem Client überlassen sich einen gewünschten Kommunikationspartner auszusuchen. Sämtliche Microservice-Instanzen müssen sich bei einer zentralen Liste anmelden, welche wiederum vom Client nach den vorhandenen Instanzen befragt wird. Diese Liste wird regelmäßig geleert, sodass etwaige nicht mehr zur Verfügung stehende Services eliminiert werden. Die Services müssen sich somit regelmäßig bei der Liste neu registrieren um weiterhin konsumiert werden zu können. Abbildung 13 veranschaulicht die Vorgehensweise bei der Client-Side Discovery Lösung. Hierbei befragt der Client zuerst das Service Registry nach allen vorhandenen Microservices, wählt einen für ihn passenden Service aus und kommuniziert im Anschluss daran direkt mit dem gewählten Service (Morris, 2016, S. 91).

Diese Lösung hat laut Richardson (Pattern: Client-side service discovery, 2016) den Vorteil, dass der Kommunikationsablauf simpler stattfindet, da das Service Registry nur dafür zuständig ist eine Liste von Services zu führen. Als Nachteil dieser Lösung gilt jedoch, dass die Auswahl des angesprochenen Service auf der Seite des Clients geschieht und dies dazu führt, dass jeder Client in der für ihn gewählten Programmiersprache sämtliche Funktionalität implementiert haben muss um diese Auswahl treffen zu können. Clients sind somit komplexer aufgebaut als ihre Pendants beim Server-Side Discovery Pattern. Weiters können Clients in diesem Fall gezielt bestimmte Services überbeanspruchen, sodass das Load Balancing umgangen werden kann. Sind die bereitgestellten Microservices somit öffentlich zugänglich ist eine solche Lösung riskant, da böswillige Clients das System überlasten können.

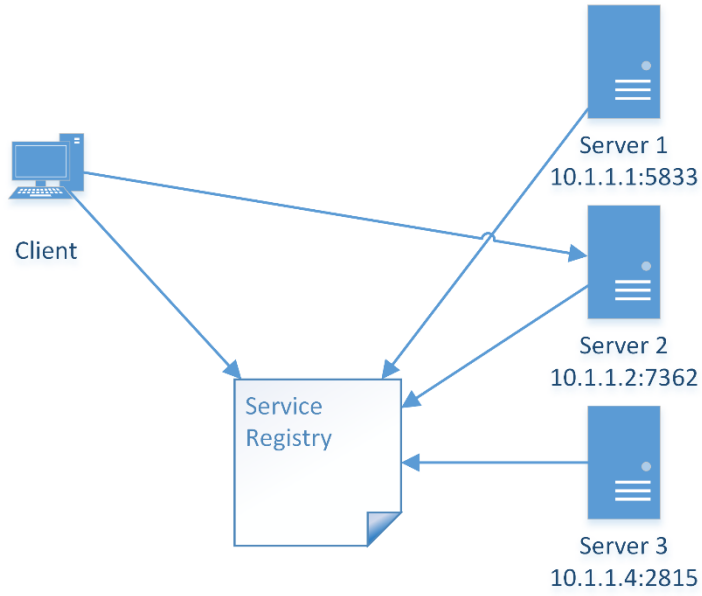

Abbildung 13: Client-Side Discovery (Richardson, Pattern: Client-side service discovery, 2016)

Bei der Server-Side Discovery existiert, genauso wie bei der Client-Side Discovery, ein zentrales Service Registry bei welchem sich alle Microservice-Instanzen registrieren müssen. Im Unterschied zur zuvor beschriebenen Lösung kommuniziert in diesem Fall jedoch der Client nicht direkt mit dem Service Registry, sondern mit einem dazwischen geschalteten Router, welcher als Load Balancer dient (Morris, 2016, S. 91). Abbildung 14 veranschaulicht den Ablauf bei der Server-Side Discovery. Hierbei setzt der Client eine Anfrage ab, welche direkt an den Router gestellt wird. Dieser befragt das Service Registry nach den vorhandenen Services, wählt einen weniger beanspruchten Service aus und leitet die Anfrage an diesen weiter. Diese Vorgehensweise hat den Vorteil, dass der Client nicht direkt mit den Services kommuniziert und dieser somit keine böswillige Überlastung eines bestimmten Servers hervorrufen kann. Weiters ist der Client in diesem Fall bedeutend simpler aufgebaut, da sämtliche Load Balancing Funktionalität vom Router durchgeführt wird und der Client somit nicht in diese Tätigkeit involviert ist. Ein Nachteil dieser Lösung ist jedoch, dass ein weiterer Partner in die Kommunikation eingreift und somit eine weitere Fehlerquelle existiert, welche entsprechend abgesichert sein sollte, sodass nicht der Router selbst zum Problem werden kann. Hierbei empfiehlt es sich den Router ausfallsicher auszulegen.

Laut Richardson (Pattern: Server-side service discovery, 2016) existieren außerdem diverse Lösungen bei denen es sich bei Router und Service Registry um ein gemeinsames Gerät handelt, was die Anzahl der involvierten Hardware wieder verringert.

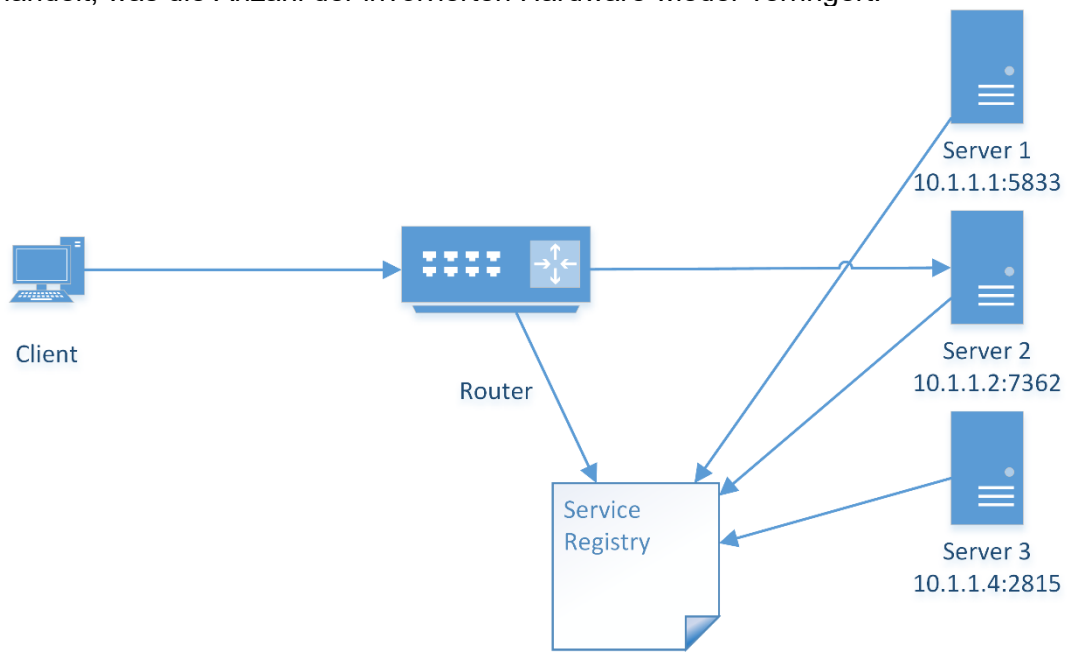

Abbildung 14: Server-Side Discovery (Richardson, Pattern: Server-side service discovery, 2016)

Bei der zweiten Möglichkeit zur Lastverteilung auf welche in dieser Arbeit eingegangen werden soll handelt es sich um die Lastverteilung per DNS. Bei dieser Lösung wird das Prinzip der Hostnamen-Auflösung durch DNS-Server benutzt um Clients dynamisch auf unterschiedliche Server zu lenken (Membrey, Plugge, & Hows, Practical Load Balancing: Ride the Performance Tiger, 2012, S. 53-58). Ein Client, welcher einen bestimmten Dienst im Netzwerk in Anspruch nehmen will spricht diesen meist über eine Domain in Form von „api.localnetwork.at“ an. Die Kommunikation in einem IP-basierten Netzwerk erfolgt jedoch über IP-Adressen, nicht über diese Domains. Deshalb existieren DNS-Server, welche dafür zuständig sind den Domainnamen „api.localnetwork.at“ in eine von Maschinen besser interpretierbare IP-Adresse in Form von „10.1.1.1“ aufzulösen (Membrey, Plugge, & Hows, Practical Load Balancing: Ride the Performance Tiger, 2012, S. 53-54). Bei der Lastverteilung per DNS wird nicht nur eine einzige IP-Adresse für eine bestimmte Domain hinterlegt, sondern mehrere gleichzeitig. Dies führt dazu, dass je nach gewähltem Verfahren unterschiedliche IP-Adressen zurückgeliefert werden (Membrey, Plugge, & Hows, Practical Load Balancing: Ride the Performance Tiger, 2012, S. 67).

Standardmäßig liefert ein DNS-Server alle hinterlegten IP-Adressen gleichzeitig zurück, sodass der Client sich prinzipiell eine dieser Adressen aussuchen kann um mit dem dahinterliegenden Server zu kommunizieren, standardmäßig wird jedoch der erste Eintrag gewählt. Die Reihenfolge der Einträge kann am Server festgelegt werden. Hierbei gibt es laut Albitz & Liu (DNS und BIND, 2002, S. 300-301) drei verschiedene Möglichkeiten:

- Fest: Die Reihenfolge der hinterlegten IP-Adressen wird beibehalten. Der erste Eintrag ist immer der erste zurückgelieferte, der letzte Eintrag immer der letzte.
- Zufällig: Die IP-Adressen werden zufällig durcheinandergemischt und im Anschluss daran an den Client zurückgeliefert.

- Zyklisch: Der erste Client erhält die erste IP-Adresse zurückgeliefert, der zweite Client die zweite und so weiter. Sobald die Anzahl der Clients die Anzahl der IP-Adressen für eine Domain überschreitet wird erneut von vorne begonnen, sodass zum Beispiel bei 3 Servern und 4 Clients der 4. Client wieder die IP-Adresse für den ersten Server erhält.

Listing 3-8 zeigt die Resource Record Einträge für einen DNS-Server, welche für das in diesem Kapitel gezeigte Beispiel notwendig wären. Hierbei ist der Hostname, die Information, dass es sich um eine Internet-Adresse handelt, sowie die zugehörige IP-Adresse selbst zu sehen. Das A steht dafür, dass es sich bei der IP-Adresse um eine IPv4 Adresse handelt.

```
api.localnetwork.at   IN    A     10.1.1.1
api.localnetwork.at   IN    A     10.1.1.2
api.localnetwork.at   IN    A     10.1.1.4
```

Listing 3-8: Resource Record Einträge für Lastverteilung per DNS

Prinzipiell wäre es mit dieser Lösung auch möglich, dass unterschiedliche DNS-Server, welche im Firmennetzwerk auf unterschiedlichen Standorten weltweit zum Einsatz kommen, auch unterschiedliche präferierte IP-Adressen liefern. Dies sorgt dafür, dass ein Client in Amerika standardmäßig zu einem Microservice in Amerika geleitet wird, während ein europäischer Client den für ihn lokalen Microservice in Anspruch nimmt. Bei Ausfall eines Servers kann der Client somit weiterhin mit Übersee kommunizieren. Der Ausfall eines Servers wird bei Lastverteilung per DNS jedoch nicht automatisch erkannt, sondern muss mit diversen Vorkehrungen überprüft werden. Der Client muss auf diesen Ausfall vorbereitet sein, sodass er bei Nichterreichbarkeit nach einer neuen IP-Adresse beim DNS-Server nachfragt. Der DNS-Server selbst muss wiederum von einem meist externen Script, welches dafür zuständig ist die einzelnen Server in regelmäßigen Intervallen auf ihre Funktionsfähigkeit zu überprüfen, darüber informiert werden, dass eine bestimmte IP-Adresse nicht mehr ausgeliefert werden soll.

Aufgrund der grundlegenden Funktionalität der Namensauflösung stellt diese Vorgehensweise auch sicher, dass ein Client bei aufeinander folgenden Anfragen immer mit demselben Server kommuniziert, da die Abfrage der IP-Adresse nur beim erstmaligen Verbindungsaufbau bei unbekannter IP-Adresse stattfindet. Weiß der Client über die IP-Adresse zu einer bestimmten Domain Bescheid speichert er diese zwischen und verwendet sie weiter ohne jedes Mal beim DNS-Server nachfragen zu müssen (Mitchell, 2017).

Diese Lastverteilung per DNS zählt du den simpelsten Methoden und führt dazu, dass nur eine rudimentäre Lastverteilung stattfinden kann. Haben die eingesetzten Server zum Beispiel unterschiedliche Leistungsmerkmale ist es mit dieser Methode nicht möglich diese zu berücksichtigen. Ein Server mit der doppelten Leistung wird genauso oft angesprochen wie ein Server mit der halben Leistung. Diese Einschränkungen könnte man umgehen indem man zum Beispiel den Server mit mehr Leistung zwei Mal in die Liste der verfügbaren Server einträgt, was dazu führt, dass dessen IP-Adresse doppelt so oft als zu präferierende IP-Adresse an die Clients weitergereicht wird als dies sonst passieren würde. Auch diese Vorgehensweise würde jedoch nicht berücksichtigen welcher Client die Services wie stark beansprucht. Weiters benötigt diese

Lösung Server, welche dieselben Microservices zur Verfügung stellen – es ist nicht möglich, dass ein Server mehr oder weniger Services zur Verfügung stellt als ein anderer Server mit demselben Hostnamen, da der DNS-Server nicht über den angesprochenen Service informiert wird und dieser somit nicht darüber Bescheid weiß welchen Service welcher Server zur Verfügung stellt, beziehungsweise auf welchen Service der Client zugreifen möchte.

Ein weiteres wichtiges Thema, das stark mit der Lastverteilung zusammenhängt, stellt die Skalierbarkeit dar. Auf diese wird im nächsten Kapitel näher eingegangen.

3.5 Skalierbarkeit

Die Skalierbarkeit von Systemen stellt vor allem bei größeren Softwareprodukten heutzutage oft ein Problem dar. Ein Großteil der dieses Thema betreffenden Einflussfaktoren wurde bereits in Kapitel 3.4 näher erklärt. In diesem Kapitel soll nun ein kurzer Einblick über die Möglichkeiten der Skalierbarkeit in Bezug auf Microservices und den Einsatz der in dieser Arbeit verwendeten Hardware gegeben werden.

Unter Skalierbarkeit versteht man die Steigerung der Leistungsfähigkeit von Hard- und Software durch das Hinzufügen weiterer Komponenten (meist Hardware). Skalierbarkeit wird in zwei Gruppen unterteilt (Kaul, 2011):

- Vertikale Skalierbarkeit – hierunter versteht man das Hinzufügen von stärkerer Hardware zu einem vorhandenen System. Zum Beispiel das Hinzufügen weiterer Prozessoren, größeren und schnelleren Speicherplatzes oder die Erweiterung des Arbeitsspeichers.
- Horizontale Skalierbarkeit – hierunter versteht man das Hinzufügen weiterer Rechner parallel zur bereits vorhandenen Infrastruktur um somit die Rechenleistung des Gesamtsystems zu steigern.

Für diese Arbeit in Frage kommt nur die horizontale Skalierbarkeit, da die vertikale Skalierbarkeit ein Aufrüsten der Hardware erfordern würde, was mit dem gewählten System des Raspberry Pi 3 Model B nur in stark begrenzten Maße möglich ist. Wie bereits in Kapitel 2.2 beschrieben ist der Arbeitsspeicher, die verwendete CPU, sowie der grundlegende Aufbau mit der GPU/VPU vorgegeben und kann nicht geändert werden. Einzig die Speicherausstattung kann in begrenztem Maße angepasst werden – Näheres hierzu wurde in Kapitel 2.6 beschrieben.

Betreffend der horizontalen Skalierbarkeit ist es wichtig die unterschiedlichen Erfordernisse der eingesetzten Komponenten zu berücksichtigen um eine Skalierung zu ermöglichen. Aufgrund des Aufbaus der einzelnen in dieser Arbeit beschriebenen Systeme in Form von Microservices sollten diese autark funktionieren können und somit nur über genau definierte Schnittstellen voneinander abhängig sein. Um jedoch eine horizontale Skalierbarkeit bei solchen Systemen zu ermöglichen ist es oft notwendig Daten doppelt abzulegen, sodass die Services auch bei Ausfall eines anderen Service in bestimmten Grenzen weiter funktionieren.

Dies betrifft zum Beispiel die Datenhaltung in Datenbanken. Meist ist in Datenbanksystemen wie MySQL der Lesezugriff um ein Vielfaches öfter notwendig als der Schreibzugriff. Dies sorgt dafür, dass es möglich ist Datenbankinhalte in bestimmten Intervallen zwischen den einzelnen Servern

zu spiegeln. Am Beispiel eines Zeiterfassungssystems ist es zwar notwendig, dass die Buchungen sofort stattfinden (Schreibvorgang), die Auswertung aller Buchungen kann jedoch auch mit einiger Verzögerung geschehen. Dies ermöglicht es, dass der Inhalt der Buchungstabelle in der Datenbank zum Beispiel nur stündlich mit den anderen Servern synchronisiert wird. Der für die Auswertung zuständige Service hätte in diesem Fall nur zu jeder vollen Stunde aktuelle Werte zur Verfügung, diese Vorgehensweise würde jedoch die Kommunikation zwischen den Services und somit die Auslastung um ein Vielfaches verringern. Eine solche Vorgehensweise ist jedoch nur zum Beispiel bei solchen Auswertungen möglich, da eine ähnliche Vorgehensweise bei schreibintensiven Vorgängen dazu führen kann, dass Informationen verloren gehen oder die Datenbestände auf unterschiedlichen Servern zu stark voneinander abweichen ohne dass dies repariert werden könnte. Zum Beispiel ein Zähler, welcher durch einen Microservice auf den Wert 2 gesetzt wird, gleichzeitig auf einem anderen Server von einem Microservice auf den Wert 3, würde hier ein großes Problem darstellen, da nicht festgestellt werden könnte was der tatsächlich zu speichernde Wert sein soll. Man könnte in so einem Fall den Wert heranziehen, welcher zuerst geschrieben wurde – oder den, welcher zuletzt geschrieben wurde. Dies würde jedoch nicht die Problematik eliminieren, dass in der Zwischenzeit mit dem jeweils falschen Wert weitergearbeitet wurde – zum Beispiel bei einem Zähler, welcher das noch zur Verfügung stehende Guthaben anzeigt. Der eine Server würde dem Kunden beziehungsweise der Kundin noch 2 Euro zum Konsum zur Verfügung stellen, während der andere Server 3 Euro anbietet. In so einem Fall muss somit sichergestellt werden, dass der Kunde, beziehungsweise die Kundin, immer mit demselben Server kommuniziert und der Datenabgleich zwischen den Servern muss in einer Weise geschehen, dass die jeweils aktuellen Daten aller Kunden und Kundinnen zentral gesammelt werden können. Diese Aufteilung von Datenbankeinträgen auf unterschiedliche Server wird auch Fragmentierung genannt (Geisler, 2014, S. 386).

Bei der Fragmentierung wird zwischen zwei verschiedenen Arten unterschieden:

- Horizontale Fragmentierung (Rahm, 1994, S. 62-66) – Die einzelnen Datensätze werden in unterschiedlichen Tabellen gespeichert. Beim oben angeführten Beispiel würde dies bedeuten, dass zum Beispiel alle Kunden und Kundinnen aus einem bestimmten Postleitzahl-Bereich in einer gemeinsamen Tabelle abgelegt werden. Diese Tabelle kann dann auf andere Server übertragen werden.
- Vertikale Fragmentierung (Rahm, 1994, S. 66-67) – Die einzelnen Datensätze sind zwar in einer gemeinsamen Tabelle gespeichert, jedoch werden ihre einzelnen Informationsbestandteile in andere Tabellen ausgelagert. Das bedeutet, dass zwar alle Kunden und Kundinnen in einer gemeinsamen Tabelle abgelegt sind und somit der Zugriff von allen beteiligten Servern aus funktioniert, jedoch sind einige Bestandteile der Datensätze in andere Tabellen ausgelagert. Am Beispiel der Microservices würde dies also bedeuten, dass ein Service nur auf die Tabelle „Guthaben“ schreibt, dass der Kunde/die Kundin mit der ID 415 soeben 1,50€ seines Guthabens verbraucht hat. Die anderen gespeicherten Informationen des Kunden/der Kundin bleiben unangetastet und können somit parallel von einem anderen Service gepflegt werden.

Letztere Form der Fragmentierung würde im Prinzip einer Normalisierung der Datenbank entsprechen. Hierbei werden sämtliche Informationen zu einem Datensatz soweit zerteilt und in einzelne Tabellen ausgelagert, dass sie keinerlei redundante Informationen mehr enthalten. Das bedeutet, dass die Zusammenhänge zwischen einzelnen Einträgen in den jeweiligen Tabellen nur noch über den Primär- sowie Fremdschlüssel hergestellt werden können (Ray, 2009, S. 5-6). Eine solche Normalisierung hat außerdem den Vorteil, dass hierdurch der Zugriff auf die Daten beschleunigt wird. Dadurch, dass die Informationen jedoch in viele Tabellen aufgeteilt gespeichert werden, ist es notwendig aufwändige JOIN-Operationen durchzuführen um zusammenhängende Informationen zu erhalten. Diese Befehle erfordern jedoch bedeuten mehr Rechenleistung als dies eine denormalisierte Datenbankstruktur erfordern würde (Raýman, 2006, S. 320-336).

Nachdem nun die Grundlagen für die hardware- sowie softwarespezifischen Bestandteile angeführt wurden, wird im nächsten Kapitel eine prototypische Implementierung durchgeführt, welche veranschaulicht wie die bisher angeführten Punkte zu einem Gesamtkonzept verschmolzen werden können.

4 PROTOTYPISCHE IMPLEMENTIERUNG

In diesem Kapitel werden die zuvor theoretisch erarbeiteten Konzepte zu einem Gesamtkonstrukt geformt um somit einen Prototyp zu erstellen, welcher die theoretische Einsetzbarkeit von Microservices auf dem Raspberry Pi 3 Model B im Unternehmensumfeld beweist.

Für die Einsetzbarkeit in Unternehmen müssen die in dieser Arbeit angeführten Microservices und der prinzipielle Aufbau jedoch diverse Sicherheitsaspekte erfüllen. Auf diese wird zuerst im nächsten Kapitel genauer eingegangen.

4.1 Sicherheitsaspekte

In größeren Unternehmensnetzwerken herrschen andere Vorgaben als dies bei kleinen privaten Netzwerken der Fall wäre. Die nächsten Unterkapitel sollen hierüber einen kurzen Überblick bieten. Den Anfang hierbei macht das Thema Kommunikation, was die physischen sowie protokolltechnischen Sicherheitsaspekte der Kommunikation zwischen den beteiligten Servern und Services betrifft.

4.1.1 Netzwerkinfrastruktur und Kommunikation

Für die Kommunikation der einzelnen Raspberry Pi 3 Model B gibt es diverse Möglichkeiten um diese vom restlichen Netzwerkverkehr abzusichern. Eine solche Absicherung ist im prototypischen Beispiel notwendig um den Datenschutz zu gewährleisten, sowie Datenmanipulation zu verhindern. Eine solche Manipulation könnte sonst dazu führen, dass ein Mitarbeiter oder eine Mitarbeiterin in Bereiche vordringen kann für welche dieser/diese keine Zutrittsberechtigung besitzt – Näheres zum Prototypen für das Zutrittssystem siehe Kapitel 4.3.3.

Die in Firmennetzwerken wohl einfachste Möglichkeit die Kommunikation von anderen Netzwerkteilnehmern zu verbergen ist der Aufbau eines eigenen Netzwerks, das nur dem Zweck dient eine Kommunikation zwischen den Services zu gewährleisten. Neben dem physischen Aufbau eines Netzwerks gibt es in den meisten Firmennetzwerken auch die Möglichkeit ein virtuelles Netzwerk – ein sogenanntes Virtual Local Area Network (VLAN) einzurichten.

Über ein Gateway und eine zwischengeschaltete Firewall ist es jedoch weiterhin möglich für die Administratoren beziehungsweise andere Abteilungen der Firma gezielt den Zugriff auf einzelne Services freizuschalten, sodass diese von ihren Office-Rechnern aus Auswertungen durchführen können oder Microservices warten können. Durch den Aufbau, welcher die sichere Kommunikation zwischen den getrennten Netzwerken ermöglicht wäre es somit auch möglich regelmäßig einen Datenabgleich mit einem Datawarehouse im Unternehmensnetzwerk durchzuführen, sodass eine Zugriffsmöglichkeit für andere Zwecke als die Wartung vollkommen entfallen könnte. Diese Art des Datenaustauschs ohne weitreichendere Absicherung würde jedoch dazu führen, dass Teile der Kommunikation der Netzwerkteilnehmer abgehört oder sogar verändert werden könnte. Um dies zu verhindern ist es somit notwendig den Netzwerkverkehr,

welcher in ein öffentliches Netz geroutet wird, zu verschlüsseln. Die Kommunikation der Teilnehmer im gleichen Netzwerk kann theoretisch auch unverschlüsselt erfolgen, solange sichergestellt ist, dass keine Zugriffsmöglichkeit für Dritte existiert, welche sich in das Netzwerk einbuchen könnten. Eine solche Absicherung des Netzwerkzugriffs durch Dritte kann zum Beispiel mittels des Authentifizierungsstandards 802.1X erfolgen (Brown, 2006, S. 2-6).

In Abbildung 15 ist der grundsätzliche Aufbau eines Netzwerks mit 802.1X Standard dargestellt. Hierbei versucht sich der Client laut Brown (802.1X Port-Based Authentication, 2006, S. 2-6) am Netzwerk anzumelden indem er zuerst dem sogenannten Authenticator seine Authentifizierungsinformationen übermittelt. Dieser verpackt diese Informationen wiederum und fragt bei einem zentralen Authentication Server nach ob diese Daten korrekt sind. Nach erfolgreicher Überprüfung und Freigabe durch den Authentication Server gewährt der Authenticator dem Client den Zugriff und setzt gegebenenfalls den Port des eingesetzten Switches auf das gewünschte VLAN, welches ihm ebenfalls durch den Authentication Server mitgeteilt wurde. Hierbei ist es somit möglich unterschiedliche Netzwerkteilnehmer automatisch in entsprechende abgetrennte Netzwerke zu unterteilen, wenn sich diese authentifizieren (Geier, 2008, S. 45-49). Erst nach erfolgreicher Authentifizierung wird dem Client der Zugriff gewährt und es ist für ihn möglich anhand der definierten Netzwerkrouten mit den anderen Netzwerkteilnehmern in Kontakt zu treten. Hierbei ist wichtig zu erwähnen, dass der Authentication Server bereits Teil des gesicherten Netzwerkbereichs ist und nur der Authenticator mit diesem direkt kommuniziert. Der anfragende Client hat keinerlei Kenntnis über den Authentication Server und auch keine Möglichkeit mit diesem direkt in Kontakt zu treten – siehe Aufbau in Abbildung 15. Der Authenticator dient somit als Türsteher und Sekretär, welcher die Anfragen sammelt und weiterleitet.

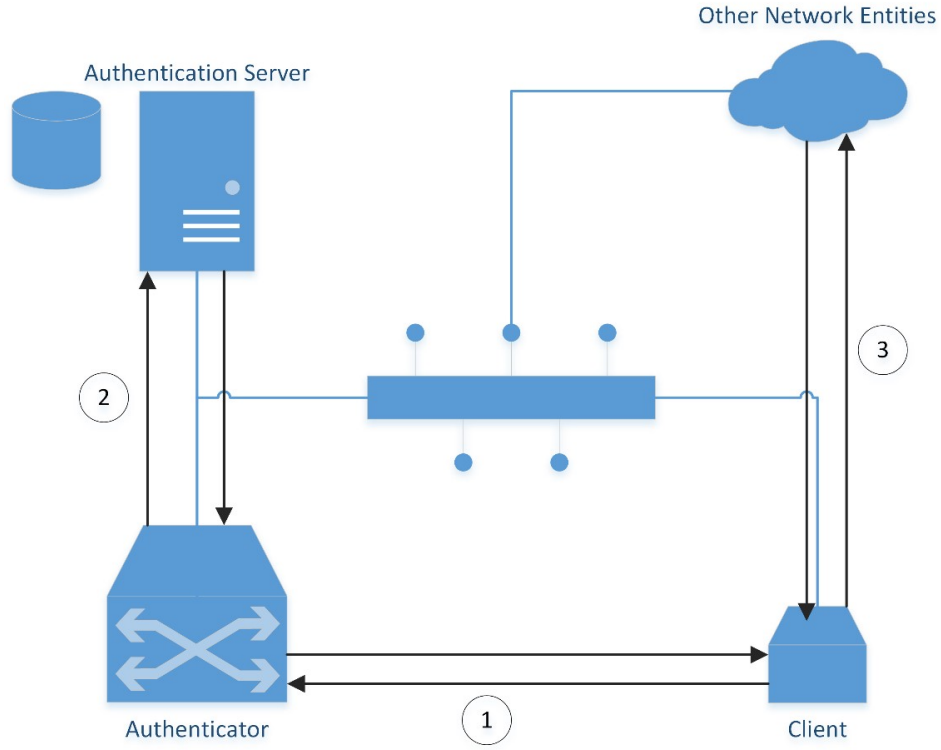

Abbildung 15: 802.1X Authentifizierungsprozess (CISCO, 2013)

Da es meist jedoch erforderlich ist von außen auf diverse Informationen des Netzwerks zuzugreifen wird in der Praxis meist auf eine Firewall zurückgegriffen, welche getrennte Netzwerke miteinander verbinden kann (Noonan & Dubrawsky, 2006). Hierbei wird das mittels 802.1X gesicherte Netz mit einem oder mehreren anderen Netzwerken verknüpft, welche jeweils für sich ebenfalls über ähnliche Sicherheitsmerkmale verfügen können.
Über die Firewall, welche als software- und hardwarebasierte Variante existiert, kann gezielt der Zugriff für einzelne Geräte in den Netzwerken untereinander freigegeben werden. Hierbei ist es zum Beispiel möglich, dass der zuvor beschriebene Datenaustausch mit einem Datawarehouse über einen genau definierten Port freigegeben wird oder den Administratoren der Managementzugriff auf die im gesicherten Netz befindlichen Raspberry Pis erlaubt wird, sodass diese aus der Ferne auf die einzelnen Komponenten einwirken können. Hierbei kann natürlich ein Übertragungskanal gewählt werden, welcher eine Verschlüsselung vorsieht um den Zugriff Dritter auf die Kommunikation zu verhindern – ein Beispiel hierfür ist die in Kapitel 2.7.1 erwähnte SSH-Verbindung.
Abbildung 16 veranschaulicht den Aufbau eines solchen Netzwerks mit einer Firewall. Es ist hier zum Beispiel möglich Client 1 den Zugriff auf die Microservices auf Server 1 zu erlauben, während die restlichen Geräte im Netzwerk für diesen Zugriff gesperrt bleiben.

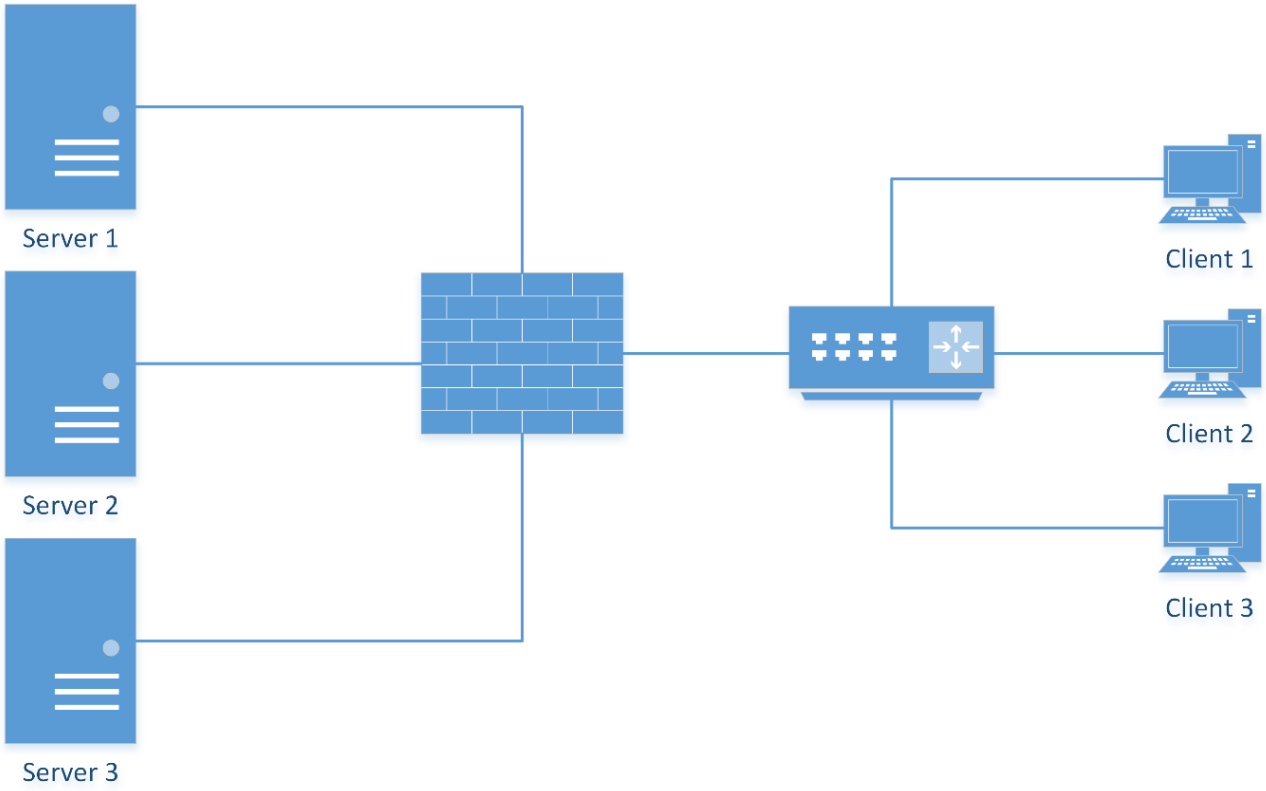

Abbildung 16: Veranschaulichung Firewall-Aufbau

Aufgrund der Verwendung eines Switches erhält auch nur der Client, welcher die Informationen angefordert hat, die benötigten Informationen zurückgeliefert, da hierbei die Informationen nur zum tatsächlichen Empfänger zugestellt werden (Geier, 2008, S. 7-8). Theoretisch ist somit kein Zugriff durch andere Clients auf diese Informationen möglich. Aufgrund diverser praktischer Angriffsszenarien ist es jedoch möglich das Routing so zu verbiegen, dass Informationen an Dritte weitergereicht oder von diesen mitgelesen werden können (Eckert, 2014). Routing in Verbindung mit der angeführten Firewall allein ist somit nicht ausreichend um die übertragenen Daten zu schützen.

Hierfür müssen somit andere Schutzvorkehrungen eingeführt werden, sodass es Dritten nicht mehr ohne Weiteres möglich ist Daten im Klartext mitzulesen. Eine solche Schutzvorkehrung ist die Verschlüsselung der Daten mittels Transport Layer Security (TLS), welche dafür sorgt, dass die Übertragenen Daten vor dem Zugriff durch Dritte geschützt werden, sowie dass diese Dritten die übertragenen Daten nicht manipulieren können (Javvin Technologies, 2005, S. 117). Wie der Name bereits vermuten lässt setzt dieser Ansatz auf der Transportschicht des Netzwerkprotokolls an und dient somit als Grundlage für viele andere Anwendungsprotokolle, welche sich diese verschlüsselte Übertragung zu Nutze machen können. Als Beispiele für eine solche Übertragung auf Anwendungsebene seien die Protokolle HTTPS, File Transfer Protocol Secure (FTPS) oder Simple Mail Transfer Protocol Secure (SMTPS) erwähnt (Oppliger, 2009, S. 79).

Bei TLS handelt es sich um ein hybrides Verschlüsselungsprotokoll, welches für den Austausch des geheimen Schlüssels ein asymmetrisches Verfahren verwendet, während die spätere Kommunikation zwischen den Teilnehmer verschlüsselt über einen symmetrischen Schlüssel stattfindet.

TLS in der derzeit aktuellen Version 1.2 unterstützt für die Verschlüsselung diverse Verfahren, wie zum Beispiel den Advanced Encryption Standard (AES), Data Encryption Standard (DES) oder Triple Data Encryption Standard (3DES) (Oppliger, 2009, S. 37-39). Empfohlen wird jedoch der Einsatz von AES in Verbindung mit dem Schlüsselaustauschverfahren Ephemeral Diffie-Hellman (Sheffer, Holz, & Saint-Andre, 2015).

Das Schlüsselaustauschverfahren Diffie-Hellman dient dazu ein gemeinsames Geheimnis zwischen zwei Teilnehmer auszuarbeiten, während die gesamte Kommunikation darüber über ein fremdes, möglicherweise kompromittiertes Netzwerk stattfindet. Mit diesem Verfahren ist es somit möglich, dass, trotz der Tatsache, dass jemand die Kommunikation bereits beim Aufbau der Verbindung belauscht, eine sichere Kommunikation aufgebaut werden kann. Beim Diffie-Hellman Verfahren handelt es sich um ein sogenanntes asymmetrisches Verfahren, was bedeutet, dass zwei eindeutige zueinander passende Schlüsselpaare existieren. Der geheim gehaltene private Schlüssel dient dazu seine eigene Identität belegen zu können, sowie die übermittelten Informationen zu entschlüsseln. Der öffentliche Schlüssel wird den Kommunikationspartnern zur Verfügung gestellt, damit diese ihre Informationen mit diesem Schlüssel verschlüsseln können. Der private Schlüssel verbleibt somit immer im eigenen Besitz, während der öffentliche Schlüssel veröffentlicht wird. Jemand, der den privaten Schlüssel nicht

besitzt hat jedoch keine Möglichkeit auf Informationen zuzugreifen, welche mit dem zugehörigen öffentlichen Schlüssel verschlüsselt wurden. Das Konzept hinter diesem Prinzip stammt aus der Mathematik, welche vereinfacht dargestellt Operationen verwendet, die nicht umkehrbar sind. Es ist in der Mathematik zum Beispiel nicht möglich aus der 1. Ableitung einer Differentialgleichung auf diese rück zu rechnen, da benötigte Informationen fehlen. Der umgekehrte Weg eine 1. Ableitung einer Differentialgleichung zu berechnen stellt jedoch kein Problem dar. Beim Diffie-Hellman Verfahren wird als mathematischer Hintergrund auf die Berechnung einer diskreten Exponentialfunktion zurückgegriffen, welche sich ebenfalls nicht durch einen diskreten Logarithmus rückrechnen lässt.

Diffie-Hellman nutzt dieses Prinzip zur Berechnung des gemeinsamen Geheimnisses, welches später als symmetrischer Schlüssel dient. Laut Beutelspacher, Neumann & Schwarzpaul (Kryptografie in Theorie und Praxis: Mathematische Grundlagen für elektronisches Geld, Internetsicherheit und Mobilfunk, 2012, S. 134-135) einigen sich hierzu beide Kommunikationspartner auf eine gemeinsame Primzahl p, sowie eine gemeinsame ganze Zahl g. Diese beiden Informationen können über das öffentliche Netz ausgetauscht werden, da diese allein kein Einfallstor bieten. Im Anschluss daran wählt jeder Kommunikationspartner eine eigene geheime zufällige Zahl (der private Schlüssel) und berechnet den Wert für $g^{eigenerprivSchluessel} * mod(p)$, welcher ebenfalls wieder dem jeweils anderen Kommunikationspartner öffentlich mitgeteilt wird. Dieser nimmt den empfangenen Wert und berechnet die folgende Formel: $empfangenerWert^{eigenerprivSchluessel} * mod(p)$

Die beiden Kommunikationspartner erhalten durch diese Rechnung das gleiche Ergebnis und können dieses Ergebnis fortan für die verschlüsselte Kommunikation verwenden. Im Vergleich zum soeben beschriebenen Diffie-Hellman Verfahren wird beim Ephemeral Diffie-Hellman Verfahren bei jeder Sitzung mit neuem Schlüsselpaar der Schlüsselaustausch mit neuer Primzahl p und ganzer zufälliger Zahl g durchgeführt (Vaudenay, 2005, S. 306). Dies führt dazu, dass dieses Verfahren Perfect Forward Secrecy (PFS) bietet. Das bedeutet, dass bei späterer Kenntnis des Schlüsselpaares keine alten Nachrichten entschlüsselt werden können, da diese mit anderen Schlüsseln und Parametern verschlüsselt wurden (Boyd & Mathuria, 2013, S. 50).

Die soeben beschriebene Vorgehensweise sorgt also dafür, dass zuerst die benötigten Informationen über das Public/Private Key-Verfahren (asymmetrische Verschlüsselung) zwischen den beiden Kommunikationspartnern ausgetauscht wird, während die tatsächliche Datenübertragung in verschlüsselter Form über den berechneten Schlüssel in symmetrischer Variante stattfindet. Diese Vorgehensweise wird deshalb gewählt, da die Übertragung von Daten mit asymmetrischer Verschlüsselung um ein vielfaches langsamer ist als die Übertragung mittels symmetrischer Schlüssel, da die unterschiedlichen Berechnungen der mathematischen Formeln vergleichsweise lange dauern (Canavan, 2001, S. 58-59). Diese Vorgehensweise wird im Allgemeinen als hybride Verschlüsselung bezeichnet. Neben den Sicherheitsaspekten hebt Canavan (Fundamentals of Network Security, 2001, S. 59) außerdem hervor, dass dieses Verfahren dazu führt, dass die Kommunikation besser skaliert werden kann – Näheres hierzu siehe Kapitel 3.5.

Nachdem nun mit Hilfe der angeführten Komponenten die Kommunikationssicherheit sichergestellt wurde, wird im nächsten Abschnitt darauf eingegangen wie man dafür sorgen kann, dass etwaige Hardwareausfälle kompensiert werden können.

4.1.2 Redundanz

In diesem Kapitel wird die Redundanz der eingesetzten Komponenten genauer betrachtet. Hierbei geht es darum, dass Ausfälle einzelner Bestandteile des Gesamtsystems nicht dazu führen, dass das gesamte System funktionsunfähig wird.

Aufgrund des technischen Aufbaus von Microservices sind diese bereits per Design dazu in der Lage redundant betrieben zu werden – Näheres hierzu siehe Kapitel 3.4 und 3.5. Die Client Infrastruktur muss, wenn dies gewünscht ist, doppelt ausgeführt werden um den Ausfall der Hardware kompensieren zu können. Hierbei ist es für die in dieser Arbeit beschriebenen Client-Applikationen notwendig, dass im Falle des Zeiterfassungssystems der gesamte Client in Form des Raspberry Pi mit seiner gesamten Peripherie redundant ausgeführt wird. Im Falle des Zutrittssystems ist es nur notwendig, dass der Raspberry Pi selbst redundant ausgeführt wird, da die Hardware für die Türöffnung in diesem Beispiel eine externe Komponente darstellt. Die für diesen Fall eingesetzten Raspberry Pi können mittels ihrer GPIO Ports parallel geschaltet werden, sodass die Ausgangsports den potentialfreien Kontakt des Türöffners direkt ansprechen. Solange alle für diesen Fall eingesetzten Raspberry Pi korrekt funktionieren würde dies bedeuten, dass der Türöffner von allen Komponenten gleichzeitig das Öffnungssignal übermittelt bekommt, während bei Ausfall eines Raspberry Pi dieses Signal nur noch von den übrig gebliebenen, sich in Betrieb befindlichen, Raspberry Pi übermittelt wird. Aufgrund der Potentialfreiheit der Ausgangsports kann dies zu keinen Kurzschlüssen führen. Wenn kein potentialfreier elektronischer Öffner beim Zutrittssystem geschaltet wird, könnte es notwendig sein einen Optokoppler zu verwenden, welcher die Ausgangsports der Raspberry Pi von den mit Spannung versorgten Schaltkontakten der Tür galvanisch abtrennt.

Abschließend wird nun darauf eingegangen wie man den Übertragungsweg zwischen den eingesetzten Komponenten redundant ausführen kann. Hierbei empfiehlt es sich die Übertragung der Signale über das Netzwerk über mehrere örtlich voneinander getrennte Wege durchzuführen, sodass bei Ausfall eines Teilstücks durch Defekte oder unvorhergesehene Umwelteinflüsse ein alternativer Übertragungsweg zur Verfügung steht. Würde man ein solches Netzwerk jedoch in der Form eines Rings aufbauen ohne weitere Vorkehrungen zu treffen würde dies dazu führen, dass das Netzwerk durch die Übermittlung der Daten überlastet wird, da ein Netzwerkknoten die Informationen an den anderen übermittelt, dieser diese entsprechend weitergibt und über einen anderen Weg wieder zurück zum ursprünglichen Übermittlungsknoten finden. Dieser Übermittlungsknoten würde erneut von vorne beginnen und die Daten entsprechend im Netzwerk verteilen. Es existiert für einen solchen Fall zwar eine Vorkehrung im Internet Protokoll selbst, diese würde jedoch nicht ausreichen um das Netzwerk nicht innerhalb kürzester Zeit vollkommen zu überlasten. Diese Vorkehrung ist das Time to Live (TTL) Feld im IP-Paket, welches vom Absender meist mit einem Wert von 255 versehen wird und von jeder an der Übertragung beteiligten Zwischenstation um eins verringert wird. Beim Erreichen des Wertes null wird das

Paket automatisch verworfen (Clark, 2003, S. 177). Da im zugrundeliegenden Ethernet-Frame eine solche Vorkehrung jedoch nicht vorhanden ist würde ein solches Paket unendlich lange im Netzwerk kreisen und dessen Übertragungskapazitäten somit stilllegen.

Aus diesem Grund existiert das Spanning Tree Protocol (STP). Dieses sorgt dafür, dass in regelmäßigen Abständen der Aufbau des Netzwerks genau analysiert wird, sodass die kürzesten Wege zwischen den einzelnen Knoten gefunden werden können. Ein in der Übertragung beteiligter Switch wird in diesem Fall immer den kürzesten Weg zum Empfänger wählen. Ein möglicherweise zweiter verfügbarer Übertragungsweg wird deaktiviert, sofern er einen längeren Weg mit höheren Kosten darstellt. Dadurch kann es zu keinen Schleifen im Netzwerk kommen, die die Übertragung lahmlegen (Javvin Technologies, 2005, S. 219-220). Wird aufgrund eines Fehlers nun der Hauptübertragungsweg gekappt, wird dieser nicht mehr länger zur Verfügung stehende Pfad beim nächsten Aktualisieren des Aufbaus des Netzwerks entfernt und stattdessen der alternative Übertragungsweg aktiviert. Dieses Aktualisieren des Aufbaus erfolgt beim STP jedoch nur zirka alle 30 Sekunden, was dazu führt, dass ein Ausfall eines Netzwerksegments die Übertragung für diesen Zeitraum unterbricht (Dean, 2009, S. 274). Ein Mitarbeiter oder eine Mitarbeiterin, welche/r das in dieser Arbeit beschriebene Zutrittssystem verwendet müsste somit im schlimmsten Fall 30 Sekunden lang warten um eine positive oder negative Rückmeldung über seine Zutrittserlaubnis zu erhalten. Aus diesem Grund wurde das Rapid Spanning Tree Protocol (RSTP) entwickelt, welches eine Unterbrechung innerhalb weniger als einer Sekunde erkennen und beheben kann (Hartpence, 2012, S. 88-92). Hierbei eliminiert das Protokoll einige Probleme von STP, bleibt zu diesem jedoch abwärtskompatibel (Badach & Rieger, 2013, S. 194).

Grundsätzlich muss bei der Inbetriebnahme vom RSTP jeder einzelne Port der Netzwerkteilnehmer konfiguriert werden, sodass den Netzwerkteilnehmern bekannt ist über welchen Port andere Netzwerkknoten erreicht werden können und über welchen Port nur Endgeräte angeschlossen sind. Durch diese genaue Konfiguration und die in einem Abstand von zwei Sekunden gesendeten Status-Pakete, welche dazu dienen zu überprüfen ob der Nachbar noch erreichbar ist, kann der Netzwerkausfall auf wenige hundert Millisekunden reduziert werden. Weiters hat RSTP den Vorteil, dass im Gegensatz zu STP nur der abgeschnittene Netzwerkbereich neu analysiert werden muss, sodass das restliche Netzwerk ohne Ausfall weiter betrieben werden kann während die Analyse der kürzesten Wege für das ausgefallene Netzwerksegment stattfindet (Badach & Rieger, 2013, S. 194). All diese Informationen werden über das Bridge Protocol Data Unit (BPDU) zwischen den Netzwerkteilnehmern übertragen, sodass das gesamte Netzwerk über die kürzesten Wege zu ihrem jeweiligen Nachbarn Bescheid weiß (Javvin Technologies, 2005, S. 219).

Bezugnehmend auf das in Kapitel 4.1.1 beschriebene Prinzip der VLANs existiert eine Erweiterung des RSTP um auf der Ebene dieser VLANs operieren zu können. Diese Erweiterung nennt sich Multiple Spanning Tree Protocol (MSTP) und dient dazu pro VLAN ein eigenes STP zu verwenden, welches über den jeweiligen Aufbau des virtuellen Netzwerks Bescheid weiß (Badach & Rieger, 2013, S. 194-195). Somit ermöglicht dieses Protokoll die Redundanz und Schleifenvermeidung auf virtueller Ebene, während STP/RSTP auf physischer Ebene arbeiten.

Abschließend empfiehlt es sich auch auf die Stromversorgung der einzelnen Komponenten einzugehen. Während Österreich im gesamten Jahr 2015 im Durchschnitt einen Ausfall der Stromversorgung für eine Dauer von 42,31 Minuten (E-Control, 2016), dies entspricht 0,008% der Gesamtzeit, hinnehmen musste, ist diese Ausfalldauer und Ausfallrate in anderen Staaten dieser Erde beträchtlich höher. Als Beispiel dient hier Rumänien, das im Jahr 2015 einen Ausfall der Stromversorgung für eine Dauer von 308 Minuten hinnehmen musste (Romanian Energy Regulatory Authority, 2016) – eine um den Faktor 7 höhere Ausfallrate als in Österreich. Um solchen Ausfällen vorzubeugen, ist es somit notwendig dafür zu sorgen, dass die Stromversorgung auch für den Zeitraum eines Netzausfalls gewährleistet ist, beziehungsweise die noch zu speichernden Daten in sicherer Weise geschrieben werden können, sodass es zu keinen korrumpierten Dateien aufgrund eines unvorhergesehenen Herunterfahrens kommen kann – Näheres zur Problematik mit korrumpierten Dateien siehe Kapitel 2.6.1.

Hierfür gibt es mobile und stationäre unterbrechungsfreie Stromversorgungen (USV). Diese bestehen in der Regel aus Batterien, welche für einen bestimmten Zeitraum den Betrieb aufrechterhalten können. Hierbei ist auch die Sicherheit gegeben, dass bei einem zu langen Ausfall der Netzstromversorgung die Geräte in vorgesehener Weise heruntergefahren werden können, was Datenverlust vorbeugt. Laut Joos (Planungsbuch Microsoft-Netzwerke: der Praxisleitfaden für Unternehmen, 2006, S. 30) unterscheidet man drei Arten von USV-Systemen:

- Voltage and frequency dependent from main supply (VFD) – Die Verbraucher werden standardmäßig mit dem Netzstrom versorgt, erst bei dessen Ausfall wird auf den Batteriebetrieb umgeschaltet. Dadurch kann es jedoch zu einer kurzen Unterbrechung kommen, da dieser Umschaltvorgang einige Millisekunden dauert und somit einige Verbraucher bereits kurzfristig stromlos gemacht wurden.
- Voltage independent from main supply (VI) – Die Verbraucher werden genauso wie bei der VFD-USV standardmäßig mit dem Netzstrom versorgt, erst bei dessen Ausfall wird auf den Batteriebetrieb umgeschaltet. Durch die kontinuierliche Messung der Ein- und Ausgangsspannung wird jedoch dafür gesorgt, dass die Umschaltzeit so kurz ist, dass meist keine Verbraucher den Ausfall wahrnehmen.
- Voltage and frequency independent from main supply (VFI) – Die Verbraucher werden durchgehend von der integrierten Batterie gespeist, welche wiederum gleichzeitig vom Netzstrom geladen wird. Hierdurch kann es zu keinen Ausfällen von Verbrauchern kommen.

Alle diese angeführten USV-Systeme haben gemeinsam, dass sie nicht nur Stromausfälle, sondern auch Spannungsschwankungen und Blitzschläge kompensieren können, wodurch durch ihren Einsatz ein zusätzlicher Nutzen entsteht (Joos, 2006, S. 30).

Bei einer Überlastung der USV (zum Beispiel durch zu viele Verbraucher oder einen Defekt der USV selbst oder eines Verbrauchers) kann es jedoch noch immer dazu kommen, dass die damit versorgten Geräte abrupt vom Strom getrennt werden. Hierfür wird meist auf Geräte zurückgegriffen, welche mindestens zwei Netzteile besitzen um einmal über die Netzspannung, sowie einmal über USV versorgt zu sein. Aufgrund des relativ geringen Verbrauchs eines Raspberry Pi kann dieser auch über eine mobile Powerbank betrieben werden, sodass dieser

über mehrere Tage autark weiterarbeiten kann ohne vom Netz mit Strom versorgt zu werden (Hüwe, 2010). Diese Methode entspricht im weitesten Sinne der VFI-USV, kann jedoch nur dann direkt eingesetzt werden, wenn nicht auf die Versorgung mittels PoE – Näheres hierzu siehe Kapitel 2.4 – zurückgegriffen wird, da in diesem Fall die Stromversorgung über den eingesetzten Switch erfolgt und somit dieser über entsprechende Absicherungen verfügen muss.

Die Verwendung einer Powerbank hat jedoch den Nachteil, dass ein Verbraucher nicht über einen Netzausfall informiert wird. Somit würde der Raspberry Pi ohne entsprechende zusätzliche Absicherung weiterhin abrupt heruntergefahren werden, wenn der Netzausfall länger dauert als dies die Überbrückungsbatterien kompensieren können. Die professionellen Systeme informieren die Verbraucher über den Netzausfall und etwaige kritische Batteriestände, sodass diese sich über eine automatische Prozedur selbstständig in definierten Abläufen herunterfahren können bevor die Stromversorgung kritisch wird (Joos, 2006, S. 31-32). Auch für den Raspberry Pi existieren entsprechende Lösungen.

Eine Möglichkeit Applikationen redundant auszuführen ist die im nächsten Kapitel beschriebene Software Docker, welche die Verteilung der Applikationen auf mehrere Hosts erlaubt.

4.2 Docker

Laut Turnbull (The Docker Book: Containerization is the new virtualization, 2014) ist Docker eine Möglichkeit Applikationen virtuell innerhalb von abgeschlossenen Containern zur Verfügung zu stellen. Ursprünglich entwickelt wurde diese Open-Source Technologie durch die Mitarbeiter der Firma Docker Inc., welche zuvor im Platform-as-a-Service (PaaS) Markt aktiv waren. Im Vergleich zu Virtualisierung, welche bereits seit Jahren im Einsatz ist, benötigt die Verwendung von Containern weniger Ressourcen und bietet auch noch weitere Vorteile. Diese Vorteile sind laut Turnbull (The Docker Book: Containerization is the new virtualization, 2014):

- Applikationen können mit Hilfe des Containers direkt auf andere Systeme kopiert werden.
- Applikationen können sehr einfach gemeinsam programmiert, getestet und ausgerollt werden.
- Applikationen sind performanter als bei der Ausführung in einer virtuellen Umgebung, da der Aufwand vom Hypervisor für die virtuelle Instanz entfällt.
- Applikationen können innerhalb von Sekunden gestartet und beendet werden, da hierfür kein ganzes virtuelles Betriebssystem gestartet werden muss.

Die Funktionsweise von Docker beschreibt Goasguen (Docker Cookbook, 2015) so, dass die zugrundeliegende Architektur des Host-Betriebssystems irrelevant ist, da sich die notwendige Docker-Engine auf allen bekannten Architekturen installieren lässt. Hierzu zählen unter anderem Windows, Linux (einschließlich der Varianten für den Raspberry Pi) sowie macOS, welche sich auf Clients sowie Server installieren und ausführen lassen. Goasguen weist jedoch darauf hin, dass es wichtig ist die verwendeten Docker-Container anhand ihrer Kompatibilität auszuwählen, da die Programme mit dem verwendeten Prozessor kompatibel sein müssen – x86, x86-64 oder

wie in unserem Fall ARM-Prozessor-basierte Systeme benötigen somit die für den Einsatzzweck entsprechend richtig kompilierten Lösungen.

Abbildung 17 zeigt den Aufbau der Docker Architektur mit den Docker Clients, dem Docker Daemon, welcher für die Zuweisung der Zugriffe an die einzelnen Docker Instanzen zuständig ist, sowie den Docker Containern, welche das letztendlich auszuführende Programm beinhalten. Der Docker Host ist der im vorherigen Absatz beschriebene Server auf dem die Docker Installation läuft.

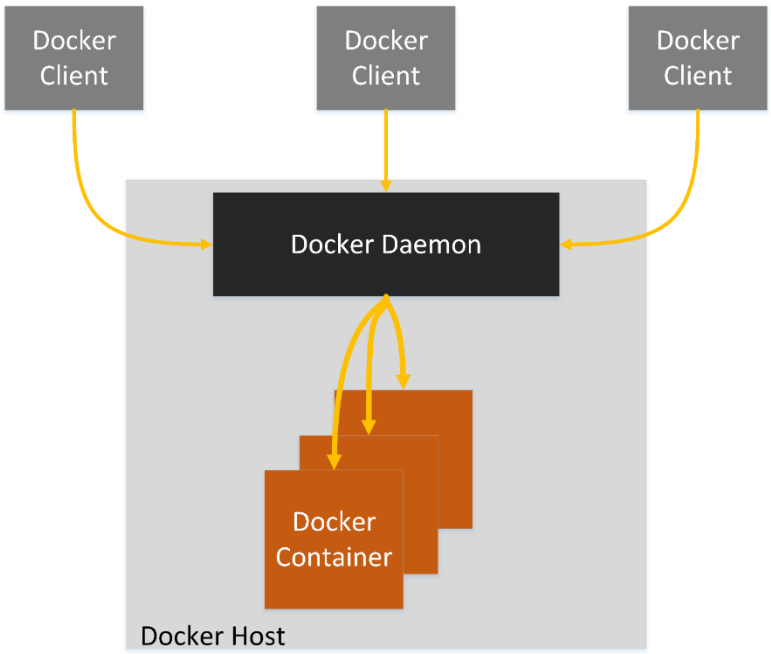

Abbildung 17: Docker Architektur (Turnbull, 2014, S. 10)

Die Ressourcen-Zuteilung des Docker Hosts kann laut Matthias & Kane (Docker: Up & Running, 2015) auf verschiedene Arten erfolgen:

- CPU-Pinning – Die Zuteilung eines Docker-Containers auf eine (oder mehrere) bestimmte CPUs. Hierdurch erfolgt die Zuteilung der Rechenzeit durch den Scheduler nur noch auf die festgelegten CPU-Kerne. Somit kann die Last entsprechend auf der Hardware des Servers verteilt werden. Einzelne Kerne können somit für andere Container oder das Host-Betriebssystem freigehalten werden.
- CPU-Shares – Die einzelnen Docker-Container werden einer bestimmten Anzahl von CPU-Shares zugewiesen. Bei Docker entspricht ein voller CPU-Pool 1024 Shares, dies entspricht der maximalen Anzahl an Shares, die man einem Docker-Container zuweisen kann. Würde man einem Docker-Container zum Beispiel 512 Shares zuweisen würde dieser im Vergleich zu einem Container mit voller Share-Zuweisung nur die Hälfte an Ausführungszeit durch den Scheduler zugeteilt bekommen. Dies zeigt bereits, dass es nicht möglich ist die Ressourcen einer CPU exklusiv für einen Docker-Container zu reservieren. Obwohl die maximale Anzahl an Shares bereits zu Container 1 zugeteilt sein würde erhält Container 2 trotzdem noch immer CPU-Ressourcen. Insofern ist dieser Wert

mehr als Priorisierung zu verstehen. Ein Docker-Container mit mehr Shares erhält mehr Ressourcen als ein Docker-Container mit weniger Shares. Hat die CPU insgesamt 100 Millisekunden an Rechenzeit zur Verteilung zur Verfügung würde in diesem Beispiel Container 1 zwei Drittel der Zeit zugeteilt bekommen. Container 2 müsste sich mit einem Drittel zufriedengeben (1024 Shares insgesamt aufgeteilt auf 2 Container mit insgesamt 1536 benötigten Shares – Prozentuell entsprechend zugeteilt). Weiters erhält Container 2 vom Scheduler automatisch mehr Rechenzeit sofern Container 1 diese nicht benötigt. Somit existiert eine dynamische Lastverteilung zwischen den Containern.

- Arbeitsspeicher – Wie meist bei virtuellen Maschinen ist auch in Docker-Containern der Arbeitsspeicher fix zugeteilt. Es ist somit möglich Container 1 zwei Gigabyte RAM und Container 2 einen Gigabyte RAM zuzuteilen. Diese Zuteilung ist im Gegensatz zu den CPU-Shares jedoch fix. Somit kann sich Container 2 im Falle von Ressourcenhunger nicht einfach mehr Arbeitsspeicher genehmigen und somit diesen von Container 1 entziehen.

Seit August 2016 ist die Docker-Integration ohne größere Umwege auf dem unter Punkt 2.7.1 beschriebenen Betriebssystem Raspbian möglich. Durch die Verwendung von Docker Swarms ist es außerdem möglich eine Lastverteilung zu realisieren. Hierbei dient der Dockerhost als Loadbalancer und verteilt entsprechend die Anfragen an die an ihm angemeldeten Dockerclients.

4.3 Dienstekomposition

Um die in den vorhergehenden Kapiteln beschriebenen Lösungen auf ihre Funktionsfähigkeit hin zu überprüfen wird in diesem Kapitel nun ein Konzept aufgezeigt wie der Aufbau der Microservices sowie der Clients aussehen kann. Hierfür wird in den nächsten Unterkapiteln der grundlegende Aufbau der Dienste und Datenbanken beschrieben, sowie die Zusammenarbeit dieser näher gebracht.

4.3.1 Grundlegender Aufbau der Dienste

Grundsätzlich wird für den prototypischen Aufbau auf die im Kapitel 4.2 beschriebene Lösung mittels Docker-Containern zurückgegriffen, da diese dafür sorgt, dass die in den Kapiteln 3.4, 3.5 und 4.1 angeführten Voraussetzungen bezüglich Lastverteilung, Skalierbarkeit und Sicherheit betreffend Ausfall einiger Komponenten, erfüllt werden.

Diese Arbeit beschreibt zwei in Unternehmen häufig anzutreffende Einsatzgebiete, welche mittels der beschriebenen Hardware der Raspberry Pi sowie dem Einsatz von Microservices vom konventionellen Ansatz der x86-Server (und entsprechend hohem Stromverbrauch) hin zu energiesparenden Lösungen geführt werden sollen.

Das erste Einsatzgebiet ist hierbei die elektronische Zeiterfassung, auf welche im nächsten Kapitel näher eingegangen wird. Das zweite Einsatzgebiet ist ein Zutrittssystem, welches dafür sorgt, dass Mitarbeiter und Mitarbeiterinnen nur zu den für sie vorgesehenen Bereichen Zutritt

erhalten. All diese Lösungen greifen hierbei auf das in Kapitel 2.3 näher beschriebene Konzept der RFID-Chips zurück, welche zur Authentifizierung an den entsprechenden Lesegeräten dienen.

Abbildung 18 zeigt einen möglichen Aufbau der hierfür notwendigen Tabellen in der verwendeten MySQL-Datenbank. Das Konzept sieht vor, dass jeder Mitarbeiter und jede Mitarbeiterin ein einziges Mal in der Datenbank existiert, jedoch mehrere Authentifizierungskarten besitzen kann, welche ihm/ihr zugeordnet sind. Die elektronische Zeiterfassung beruht hierbei darauf, dass die eindeutige „PersonID“ zusammen mit der durchgeführten Transaktion (Kommen/Gehen) in einer eigenen Tabelle gesichert wird, welche den zugehörigen Zeitstempel enthält. Hierdurch ist es später einfach möglich Auswertungen für eine bestimmte Person durchzuführen oder ihren aktuellen Status zu überprüfen. Die in der Abbildung links unten angeführten beiden Tabellen dienen dem Zutrittssystem. Hierbei enthält die Tabelle „Doors“ die Information über alle im System vorhandenen Türen mit ihrer zugehörigen eindeutigen Nummer, sowie der Einstellung ob es bei dieser Tür erforderlich ist laut Zeiterfassungssystem anwesend zu sein oder nicht. Dies dient dazu, dass Mitarbeiter und Mitarbeiterinnen nicht im Unternehmen unterwegs sind ohne dass sie offiziell arbeiten. Um den Zutritt bis zum Zeiterfassungsterminal zu ermöglichen müssen jedoch etwaige Türen auf dem Weg zu diesem Terminal auch ohne offizielle Anwesenheit geöffnet werden können. Über die Tabelle „AccessToDoor“ kann außerdem festgelegt werden welche Bereiche ein Mitarbeiter oder eine Mitarbeiterin betreten darf. Hierbei ist es unerheblich mit welcher Zutrittskarte des Mitarbeiters/der Mitarbeiterin die Authentifizierung durchgeführt wird. Durch den Aufbau der Datenbank kann man somit eine verlorene Karte mittels der Löschung eines Eintrags sperren, sowie einen Austritt eines Mitarbeiters/einer Mitarbeiterin aus dem Unternehmen schnellstmöglich durchführen ohne dass auf etwaige Zusatzkarten vergessen werden kann. Der Mitarbeiter/die Mitarbeiterin ist somit gleichzeitig in beiden angeführten Systemen deaktiviert.

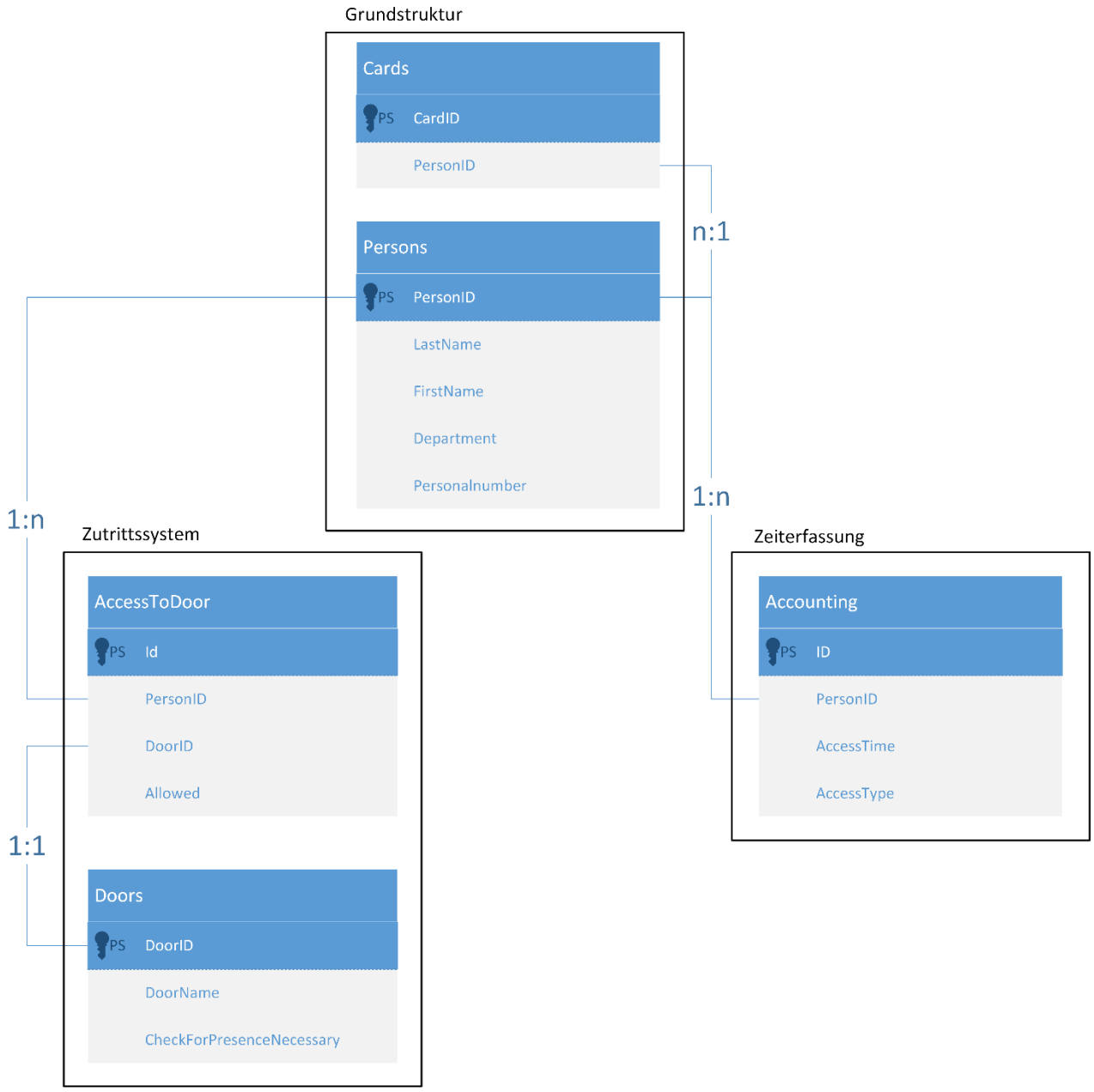

Abbildung 18: Tabellenstruktur

Die einzelnen Tabellen sind hierbei nicht in der selben Datenbank vorhanden, sondern getrennt auf drei unterschiedliche Docker-Instanzen verteilt. Die Verteilung erfolgt anhand der zugehörigen Einsatzgebiete, welche in der zuvor beschriebenen Abbildung näher eingegrenzt wurden.

Abbildung 19 veranschaulicht die prototypisch implementierten Microservices dieser Lösung. Hierbei ist zu sehen, dass der jeweilige Client immer nur einen Service aufruft – je nach Client handelt es sich hierbei um die Abfrage der erlaubten Türöffnung, die Kommen-Buchung oder die Gehen-Buchung. Die Microservices selbst kümmern sich wiederum darum weitere Informationen von anderen Microservices einzuholen, sofern diese erforderlich sind. Nähere Informationen hierzu werden in den nächsten Kapiteln angeführt.

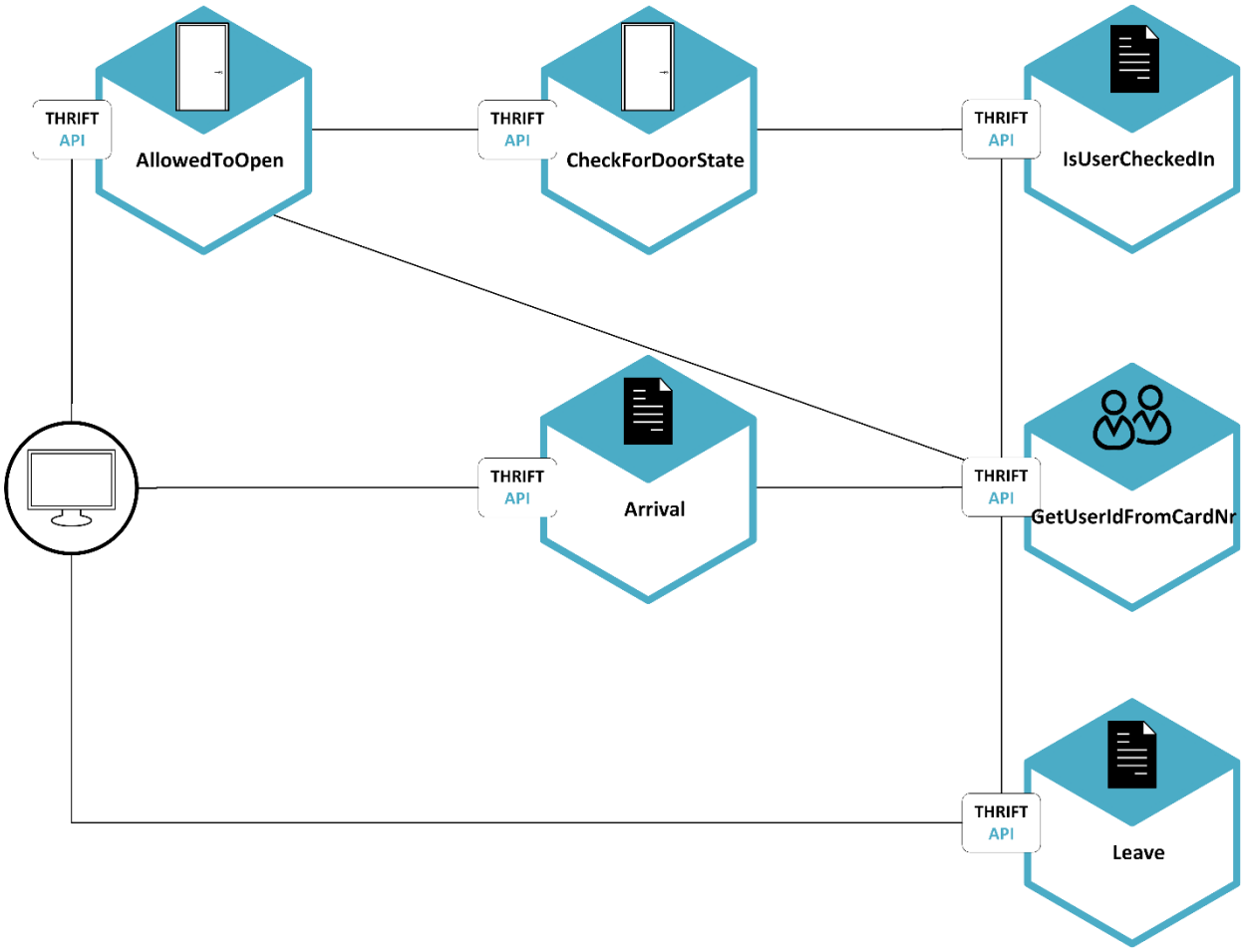

Abbildung 19: Übersicht der implementierten Beispiel-Services

Im nächsten Abschnitt wird nun die Realisierung der Services für die Zeiterfassung näher erläutert.

4.3.2 Zeiterfassung

Bei der prototypischen Implementierung der Zeiterfassung wurde, wie im vorhergehenden Kapitel erwähnt, der Aufbau mittels Microservices durchgeführt indem diverse Schnittstellen zu anderen Komponenten zur Verfügung gestellt und genutzt wurden. Die grundsätzliche Datenhaltung erfolgt in einer Docker-Instanz, welche eine MySQL-Datenbank enthält. Die prototypisch implementierten Microservices dienen dazu von extern aufgerufene Funktionen auszuführen, welche die Kommen/Gehen-Buchungen durchführen. Hierbei erfolgt der Aufruf des Microservices mittels Übergabe der vom Client ausgelesenen eindeutigen ID des RFID-Chips – Näheres hierzu siehe Kapitel 2.3. Der aufgerufene Dienst fragt wiederum bei einem weiteren Microservice an um herauszufinden welcher Person das soeben präsentierte Medium zugeordnet ist. Im Anschluss daran wird die kumulierte Information in die Datenbank gespeichert.

Weiters wäre es möglich anhand der gespeicherten Informationen Auswertungen in Form von Microservices zur Verfügung zu stellen, welche der Human Resources Abteilung Daten liefern mit denen diese die monatliche Zeitabrechnung durchführen können. Diese Übermittlung muss in diesem Fall über eine mittels TLS verschlüsselte Verbindung erfolgen, sofern der Empfänger

sich in einem anderen, öffentlich zugänglichen Netzwerk befinden sollte – Näheres hierzu wurde in Kapitel 4.1.1 beschrieben.

Neben den schreibenden Diensten stellt ein weiterer Microservice eine Möglichkeit zur Verfügung um den aktuellen Anwesenheitsstatus eines Mitarbeiters oder einer Mitarbeiterin abzufragen. Dies wird neben anderen Kriterien, welche im nächsten Kapitel näher beschrieben werden, für die Freischaltung von Türen im Zutrittssystem verwendet.

4.3.3 Zutrittssystem

Als zweites Veranschaulichungsbeispiel dient in dieser Arbeit ein Zutrittssystem, welches als Prototyp implementiert wurde. Hierbei erfolgt das Auslesen des Authentifizierungsschlüssels äquivalent zum Zeiterfassungssystem über die eindeutige ID des RFID-Chips. Zuerst wird hierbei wieder die zugehörige „PersonID“ anhand der ID des RFID-Chips über einen Microservice ausgelesen. Bevor die Türöffnung autorisiert wird, wird mittels Aufrufs eines weiteren Microservices überprüft ob die betroffene Tür zu einem internen Bereich zählt, welcher es erfordert eingestempelt zu sein. Ist dies der Fall, erfolgt die Nachfrage mittels des vom Zeiterfassungssystem zur Verfügung gestellten Microservice, der überprüft ob der Mitarbeiter oder die Mitarbeiterin eingestempelt ist oder nicht. Bei positiver Rückmeldung erfolgt die Überprüfung der Zutrittsberechtigung über die ermittelte „PersonID“ und die eindeutige „DoorID“, anderenfalls wird der Zutritt verweigert. Wenn der Microservice die Erlaubnis des Zutritts zurück meldet öffnet der Client (in Form des Raspberry Pi) mittels der Schaltung eines seiner GPIO-Ports – Näheres hierzu siehe Kapitel 2.2 – die angebundene Tür und gibt somit den Zutritt frei.

Wichtig zu erwähnen ist, dass im implementierten Prototypen keine Dokumentation der durchgeführten Abfragen oder Öffnungen erfolgt. Diese würde jedoch im späteren Produktiveinsatz notwendig sein um etwaige Bewegungen und Abläufe nachvollziehen zu können.

Nachdem das Konzept der Datenbank und Microservices näher beschrieben wurde, wird im nächsten Kapitel nun abschließend auf den Aufbau der Hard- und Software eingegangen.

4.4 Grundlegender Aufbau der Infrastruktur

In diesem Kapitel wird nun der grundlegende Aufbau der Hard- und Software, für das in dieser Arbeit beschriebene Konzept, näher beschrieben. Hierbei wird auf die in den vorhergehenden Kapiteln beschriebenen Hard- und Softwarekomponenten zurückgegriffen um die Realisierbarkeit zu beweisen. Im nächsten Abschnitt wird hierfür zuerst auf den Aufbau des Netzwerks sowie der eingesetzten Hardware eingegangen.

4.4.1 Hardware

Für die Realisierung der beiden für diese Arbeit ausgewählten Beispiele in Form einer Zeiterfassung und eines Zutrittssystems ist es notwendig die zuvor theoretisch beschriebenen

Eigenschaften des Aufbaus in ein Gesamtkonzept zu verstricken. Abbildung 20 veranschaulicht dieses Konzept. Die involvierten Netzwerkswitches werden mittels USV versorgt und verfügen insgesamt über vier verschiedene VLANs welche untereinander nicht direkt kommunizieren dürfen. Eine VLAN-übergreifende Kommunikation ist nur über die Firewall möglich. Die Switches sind untereinander in einer Ring-Form miteinander verbunden, sodass eine Schleife entstehen würde. Durch den Einsatz von RSTP wird jedoch in dem gewählten Beispiel die Verbindung zwischen Switch 2 und 3 temporär deaktiviert. Switch 1 ist in diesem Beispiel der Root-Switch. Bei Ausfall der Verbindung zwischen Switch 1 und einem der anderen beiden Switches würde das RTSP automatisch die zuerst temporär deaktivierte Verbindung wieder öffnen, sodass der gesamte Netzwerkverkehr weiterhin möglich ist.

VLAN-intern können alle Geräte miteinander kommunizieren, sodass der Zugriff ohne Einschränkung auf einen oder mehrere Microservices möglich ist. Möchte ein Client aus VLAN1 auf Daten oder einen Microservice zugreifen, können diese nur über die zwischengeschaltete Firewall erreicht werden. Bei dieser muss der Zugriff explizit erlaubt werden, anderenfalls wird die Kommunikation blockiert.

Die einzelnen Raspberry Pis werden aufgrund der in Kapitel 2.4 beschriebenen Vorteile mittels PoE von den Switches versorgt, sodass ein Administrator einen Neustart über das Netzwerk durchführen kann auch wenn die Software des Raspberry Pi nicht mehr reagieren sollte. Aufgrund der in Kapitel 2.6 beschriebenen Problematik mit korrumpierten Dateien durch unerwartete Neustarts wird in diesem Fall auf die Option des PXE zurückgegriffen (Näheres hierzu siehe Kapitel 2.6.4), wodurch der Administrator aus der Ferne gegebenenfalls ein anderes noch funktionsfähiges Betriebssystem starten kann oder das gesamte Betriebssystem im ReadOnly-Modus ausführen kann, sofern dies gewünscht ist.

Über die beiden Load Balancer ist es möglich in jedem der beiden Beispiel-VLANs dynamisch weitere Microservices auf physischen Raspberry Pis oder als weitere Instanz auf einem vorhandenen Raspberry Pi hinzuzufügen. Als Load Balancer diente im Testaufbau die im Kapitel 4.2 beschriebene Variante der Docker Swarms.

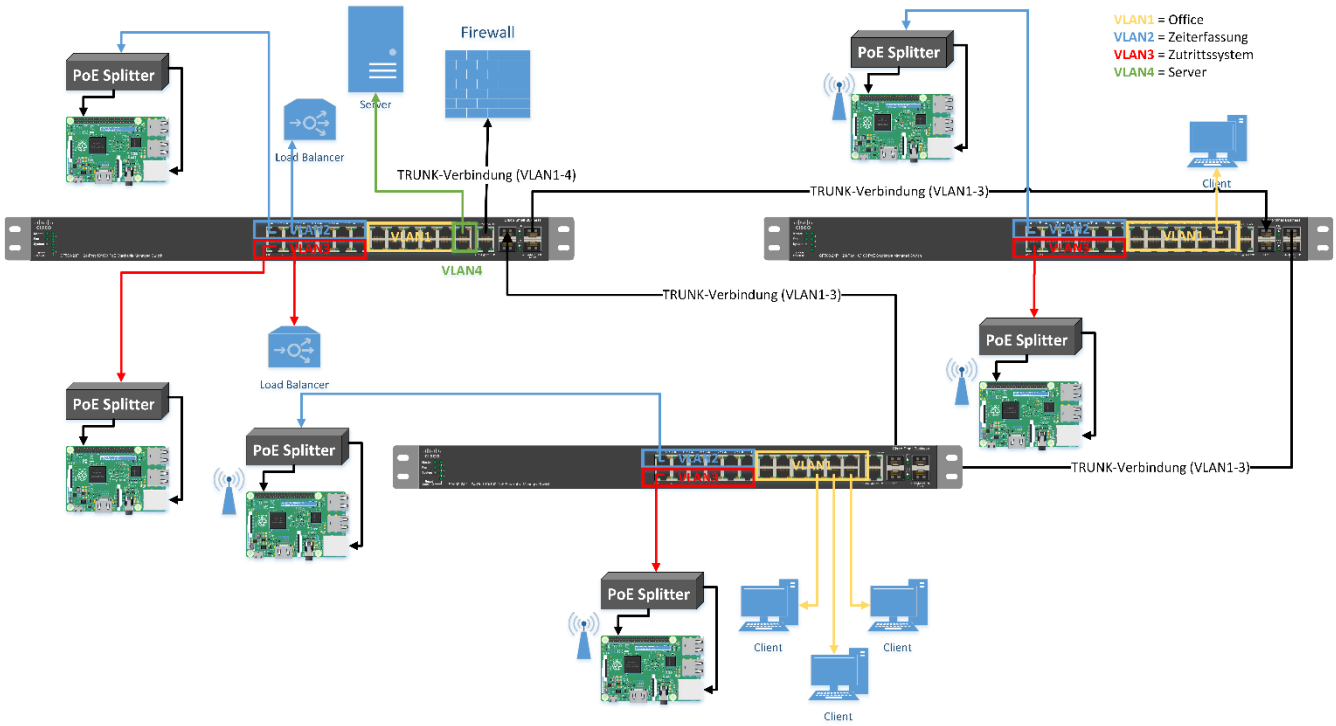

Abbildung 20: Grundlegender Aufbau des Konzepts

Sofern die beiden VLANs 2 und 3 nicht über frei zugängliche Netzwerkports erreicht werden können, sondern diese in versperrten Datencentern untergebracht sind (inklusive der angeschlossenen Clients/Server) kann auf weitere Sicherheitsvorkehrungen verzichtet werden, da sich physisch niemand in das Netzwerk einklinken kann. Anderenfalls müsste, wie in Kapitel 4.1.1 beschrieben, mit 802.1X gearbeitet werden, sodass nur bekannte Geräte ins Netzwerk inkludiert werden können.

Die Clients wurden neben den bereits inkludierten GPIO-Ports mit einer USB-To-TTL Schnittstelle ausgestattet um die Anbindung eines RFID-Lesegeräts über das Universal Asynchronous Receiver Transmitter (UART) Protokoll zu ermöglichen.

Nachdem nun der grundlegende Hardwareaufbau veranschaulicht wurde, wird im nächsten Abschnitt auf die verwendete Software näher eingegangen.

4.4.2 Software

Dieses Kapitel dient dazu die verwendete Software näher zu beschreiben, sodass der in dieser Arbeit implementierte Prototyp nachvollziehbar ist und eine eigene Implementierung anhand der beschriebenen Merkmale durchgeführt werden kann.

Als Betriebssystem für die Umsetzung des Servers wurde auf die in Kapitel 2.7.1 beschriebene Linux-Distribution Raspbian zurückgegriffen, da diese die in dieser Arbeit angeführten Funktionen unterstützt und bereits diverse Frameworks zur Verfügung stehen um die benötigten Bestandteile einfacher implementieren zu können. Weiters bietet dieses System Hardwarebeschleunigung, welche diverse Abläufe beschleunigen kann. Raspbian ist außerdem das einzige angeführte Betriebssystem, welches das Booten über Netzwerk unterstützt – Näheres hierzu siehe Kapitel

2.6.4. Durch die Wahl des Netzwerkboots ist es somit möglich alle Server und Clients zentral zu verwalten, sodass bei etwaigen Änderungen nur die Images an zentraler Stelle getauscht werden müssen und ein Neustart der Hardware initialisiert werden muss. Andere Lösungen, wie das in Kapitel 2.7.3 und 2.7.4 beschriebene Windows 10 IoT beziehungsweise Android Things unterstützen diese Möglichkeit nicht und befinden sich außerdem noch in einem Beta-Status. Das Grundkonzept von Windows 10 IoT ist darüber hinaus nicht mit dem Konzept der Microservices in Einklang zu bringen, da es nicht möglich ist mehrere Apps gleichzeitig auszuführen. Dies würde bedeuten, dass bei Austausch oder Hinzufügen eines Microservices temporär alle am Server laufenden Dienste offline genommen werden müssen. Windows 10 IoT würde somit nur als Client dienen können, welcher die angebotenen Microservices konsumiert. Aufgrund der in Kapitel 3 beschriebenen Plattformunabhängigkeit von Microservices wäre dieser Einsatz jedoch problemlos möglich sollte dies eine Applikation erfordern, welche nur für Windows 10 IoT existiert.

Für die Implementierung der einzelnen Komponenten des Server/Client-Systems wurde auf die Programmiersprache Python zurückgegriffen, welche standardmäßig von Raspbian unterstützt wird. Die Wahl fiel darauf, da bereits diverse Frameworks für die Implementierung in dieser Sprache existieren. Aufgrund der Programmiersprachenunabhängigkeit von Microservices wäre es jedoch möglich gewesen viele andere Sprachen für die Implementierung zu verwenden – Näheres hierzu siehe Kapitel 3.1.

Als Lösung für die Server/Client-Kommunikation wurde auf die in Kapitel 3.3.3 näher beschriebene Variante des Apache Thrift zurückgegriffen, da dieser wie beschrieben viele Vorteile in sich vereint und ebenso weitestgehend programmiersprachenunabhängige Entwicklung erlaubt. Als Framework diente in dieser Arbeit die Apache Thrift Implementation ThriftPy, welche ein angenehmeres Implementieren der Funktionen ermöglicht, da es wiederkehrende Generierungsvorgänge der Thrift-Dateien unnötig macht (Yu, 2016). Für die Übermittlung der Daten wurde auf das TBinaryProtocol zurückgegriffen, welches es erlaubt auch auf dem vergleichsweise leistungsschwachen ARM-Prozessor des Raspberry Pi 3 Model B eine praxistaugliche Übertragungsrate zu erreichen. Der Transport selbst wurde mittels TBufferedTransport realisiert, um im Vergleich zum sonst gebräuchlichen TSocket die gleichzeitige Inanspruchnahme der Microservices zu ermöglichen. Aufgrund der Verwendung der ThriftPy Implementierung ist es jedoch problemlos möglich andere Transport-Protokolle und Mechanismen innerhalb weniger Minuten zu realisieren.

Um die Problematik der fehlenden Systemzeit zu lösen wurde auf das in Kapitel 2.5 beschriebene NTP zurückgegriffen, welches dafür sorgt, dass die Server und Clients immer über das korrekte Datum und die korrekte Uhrzeit Bescheid wissen.

Wie bereits in den vorhergehenden Kapiteln beschrieben, wurde als Datenbank auf insgesamt drei MySQL-Instanzen zurückgegriffen, welche in jeweils einem eigenen Docker-Container ausgeführt werden – Näheres bezüglich Docker wurde in Kapitel 4.2 erläutert. Für den Zugriff auf die gespeicherten Informationen und deren Bearbeitung diente das MySQLdb-Modul für Python. Die Docker-Instanzen der MySQL-Datenbanken, sowie die erzeugten Docker-Instanzen der in Python implementierten Microservices wurden als Docker Swarm ausgeführt, was – wie in Kapitel 4.2 beschrieben – das spätere Hinzufügen und Entfernen von Rechnern zum Zwecke der

Lastverteilung ermöglicht und somit die aus den Kapiteln 3.4, 3.5, 4.1.1 und 4.1.2 beschriebenen Erfordernisse erfüllt.

Als Client-Betriebssystem diente ebenfalls das für die Server verwendete Raspbian, welches um eigene Applikationen erweitert wurde, sodass der Konsum der Microservices ermöglicht und die Interaktion mit den RFID-Chips realisiert werden konnte. Hierbei wurde aufgrund der nicht notwendigen Lastverteilung an den einzelnen Clients auf die Docker-Integration verzichtet und die Applikation somit am physischen Host ohne weitere Zwischenebene ausgeführt. Der Zugriff auf die RFID-Chips wurde hierbei mittels einer modifizierten Version von pynfc durchgeführt (Auty, 2014), welches wiederum auf libnfc als Framework für den Zugriff zurückgreift (nfc-tools/libnfc, 2017). Diese Python-Applikation diente dazu das in Kapitel 2.3 beschriebene Data-on-Tag Verfahren umzusetzen, bei welchem die eindeutige ID des RFID-Chips ausgelesen wird. Im Anschluss daran wurde diese ID an die Microservices-Client-Applikation übergeben, welche ihrerseits diese eindeutige ID an den entsprechenden Microservice zur Überprüfung weitergereicht hat.

Abschließend werden nun im nächsten Kapitel die gesammelten Erkenntnisse zusammengefasst und das Ergebnis dieser Arbeit präsentiert.

5 ZUSAMMENFASSUNG UND ERGEBNIS

In diesem Kapitel erfolgt eine Zusammenfassung der in dieser Arbeit erarbeiteten Erfordernisse für einen Einsatz von Raspberry Pis in Verbindung mit Microservices in Unternehmen. Die Praxistauglichkeit wurde hierbei durch die prototypische Implementierung eines Zutrittssystems und einer Zeiterfassung mittels RFID-Chips getestet. Je nach späterem Einsatzbereich ist es erforderlich die Komponenten in ein eigenes, von Dritten getrenntes, Netzwerk auszulagern. Diese Auslagerung kann mittels eines eigenen VLAN geschehen. Um den Ausfall einzelner Komponenten kompensieren zu können sollte hierbei auf RTSP, sowie eine USV zurückgegriffen werden. Beim Einsatz von VLANs erfolgt der Datenaustausch zwischen den Diensten ungehindert im selben Netzwerk, während Dritte keine Möglichkeit besitzen auf diesen Datenverkehr Einfluss zu nehmen. In Unternehmen ist es jedoch häufig notwendig, dass auch einzelne Abteilungen Zugriff auf ein solches Netzwerk erhalten um dieses zum Beispiel zu warten oder um Auswertungen durchführen zu können. Hierfür bietet es sich an eine Firewall zu verwenden, welche den Netzwerkverkehr in beide Richtungen beschränkt und die es somit ermöglicht nur ausgewählten Bereichen diesen Zugriff zu gewähren.

In diesem Fall sollte jedoch auf eine verschlüsselte Übertragung der Informationen Wert gelegt werden, sodass Dritte weiterhin keine Möglichkeit besitzen diese Daten unerlaubterweise einzusehen oder zu manipulieren. Für die unverschlüsselte sowie verschlüsselte Übertragung kann auf Apache Thrift zurückgegriffen werden, das hierfür eine ausreichend hohe Übertragungsrate bietet und plattformunabhängig ist, sowie für eine große Anzahl an Programmiersprachen zur Verfügung steht.

Aufgrund des in der Arbeit beschriebenen technologischen Unterschieds zwischen x86 und ARM-Prozessoren ist es nicht ohne weiteres möglich herkömmliche Software, welche auf x86-Prozessor-basierten Systemen lief, auf der ARM-Plattform auszuführen. Der Quellcode muss für die Ausführung auf ARM-Prozessoren explizit hierfür compiliert werden, sofern der Code nicht bereits in Form von zum Beispiel Bytecode für die Ausführung innerhalb einer virtuellen Maschine vorliegt.

Ein mittels x86-Prozessor ausgestattetes System verfügt zwar meist über mehr Leistung, verbraucht hierfür jedoch auch bedeutend mehr Strom. Die Verwendung eines Raspberry Pi hat hierbei nicht nur den Vorteil, dass dieser notfalls über eine Powerbank oder PoE betrieben werden kann, sondern auch, dass dieser über diverse Input/Output-Schnittstellen verfügt, welche die Realisierung von Hardwarelösungen erlaubt, die mit herkömmlichen Servern nicht ohne zusätzliche Hardware möglich wäre. Durch die Verwendung von PoE ist es somit auch möglich eine rudimentäre Fernwartung des Raspberry Pi zu realisieren. Bei etwaigen Softwarefehlern kann somit aus der Ferne ein Neustart initialisiert werden, auch wenn der Client nicht mehr auf Netzwerkanfragen reagiert.

Durch die Verwendung von PXE kann in Verbindung mit dem Betriebssystem Raspbian außerdem zentral auf sämtliche gespeicherten Informationen des Betriebssystems Einfluss genommen werden, während mittels SSH eine weitere Form der Fernwartung angeboten wird.

Um eine Lastverteilung und weitergehende Ausfallsicherheit zu gewährleisten ist es möglich die verwendeten Applikationen innerhalb eines Docker-Containers auszuführen, welcher mittels Docker Swarms auf mehrere Hosts zur parallelen Ausführung verteilt wird. Hierbei haben durchgeführte Tests in der Vergangenheit gezeigt, dass eine Ausführung mehrerer Tausend Docker-Container parallel auf einem einzelnen Raspberry Pi möglich ist, ohne dass dieser seine Arbeit einstellt. Für das erarbeitete Konzept mussten jedoch nur insgesamt drei Docker-Container für die verteilte Datenbank ausgeführt werden, während die in Python programmierten Microservices direkt unter Raspbian liefen. Die Tests haben hierbei gezeigt, dass bereits ein einzelner Raspberry Pi dazu in der Lage ist die hierfür notwendige Leistung zu erbringen, ohne dass die Reaktionszeit auf ein Niveau fällt, welches zu Kommunikationsschwierigkeiten zwischen den Komponenten führen könnte. Neben der bereits erwähnten Verteilung der Microservices auf mehrere Hosts zur parallelen Ausführung, sollte jedoch auch am Client auf möglicherweise auftretende Ausfälle der Microservices Rücksicht genommen werden. Das bedeutet, dass dieser für diese Fälle einen Cache implementiert haben sollte, welcher etwaige Buchungen zwischenspeichert bis diese an den Server übertragen werden konnten. Eine solche Zwischenspeicherung ist bei einem Zutrittssystem jedoch nur teilweise sinnvoll, da verspätete Freigaben eines Zutritts ein erhebliches Sicherheitsrisiko darstellen würden. Um auch in diesem Fall auf ein Backup-System zurück greifen zu können, empfiehlt es sich bisher bekannte Berechtigungen für den Client temporär lokal abzulegen. Eine Freigabe aufgrund dieser gespeicherten Daten sollte jedoch nur für einen begrenzten Zeitraum erfolgen, um eventuell bereits gesperrte Personen, von denen der Client aufgrund eines technischen Gebrechens keine Kenntnis besitzt, rechtzeitig vom Zutritt abzuhalten. Damit auch längere Ausfälle einzelner Hardwarekomponenten zu keiner dauerhaften Zutrittsverweigerung aller Personen führen, ist es zum Beispiel möglich einzelne Master-RFID-Chips fix im Quellcode der Clients zu hinterlegen. Für den Fall des Verlusts eines solchen Chips kann der entsprechende Teil des Programmcodes zentral ausgetauscht werden, um im Anschluss daran durch einen per Script initialisierten Neustart aller Clients automatisch auf diese verteilt zu werden, da diese ihre Konfiguration von einer zentralen Stelle abrufen. Entscheidet man sich jedoch dazu den gesamten Betriebssystem-Datenträger an diese zentrale Stelle auszulagern würde ein Ausfall selbiger oder des Netzwerks unweigerlich dazu führen, dass der Client seine Arbeit einstellt. Um auch diese Eventualität ausschließen zu können müsste man bei den Clients somit auf einen lokalen Datenträger, zum Beispiel in Form einer eMMC, zurückgreifen.

Im Falle des verwendeten RFID-Chips sollte im praktischen Einsatz darauf geachtet werden moderne Chips zu verwenden, welche über keine bekannten Sicherheitslücken verfügen. Weiters sollte nicht nur die reine eindeutige ID eines Chips zur Authentifizierung herangezogen werden, sondern ein mittels Public/Private-Key verschlüsselter Bereich des Chips, da diese Informationen, im Gegensatz zur ID, nicht für jede Person frei zugänglich sind.

Abschließend erfolgte ein Vergleich der Ausführungszeit für Microservice-basierte Python-Applikationen zwischen dem Raspberry Pi 3 Model B und einem auf x86-Prozessoren basierten System auf welchem, zur besseren Vergleichbarkeit, Debian als Betriebssystem installiert wurde. Da – wie in Kapitel 2.7.1 beschrieben – Raspbian ebenfalls auf diesem System basiert sorgt dies für eine ähnliche Ausgangssituation auf beiden Testsystemen. Der eingesetzte PC verfügte über einen i5-Prozessor mit 2,4 GHz Basis-Taktfrequenz sowie 4 GB Arbeitsspeicher und ein SSD-Laufwerk. Als Client diente bei allen Tests ein Raspberry Pi 3 Model B, welcher die angebotenen Microservices konsumierte. Tabelle 2 veranschaulicht die gefundenen Ergebnisse. Es wurden insgesamt jeweils 10 Messungen durchgeführt, von denen anschließend der Mittelwert herangezogen wurde.

	Raspberry Pi 3 Model B – Dauer in Sekunden	Zugriffe pro Sekunde	x86-basierter PC – Dauer in Sekunden	Differenz Raspberry Pi zu PC in Prozent
10.000 Zugriffe inklusive Dockerinstanzen, SQL-Zugriff und Terminal-ausgabe	44,528	224,578	355,258	-87%
10.000 Zugriffe inklusive Dockerinstanzen und SQL-Zugriff	40,639	246,069	351,139	-88%
10.000 Zugriffe inklusive Dockerinstanzen	10,625	941,176	6,921	+54%
10.000 Zugriffe	10,985	910,332	7,191	+53%

Tabelle 2: Test der Zugriffs-Performance mittels Thrift

Man sieht hierbei, dass der Raspberry Pi 3 Model B beim Zugriff auf SQL-Daten weitaus performanter war als sein Konkurrent. Aufgrund der mehrfach durchgeführten Messung kann ein Messfehler oder andere Programme, welche im Hintergrund gelaufen sein könnten ausgeschlossen werden. Der Performance-Unterschied wird hierbei vermutlich an der optimierten MySQL-Docker-Instanz von hypriot (hypriot/rpi-mysql, 2017) für den Raspberry Pi liegen, während der x86-basierte PC eine Standard MySQL-Docker-Instanz erhielt.

Weiters wurde ein Test durchgeführt um zu überprüfen wie lange das Auslesen der eindeutigen ID einer RFID-Karte am Client benötigt. Tabelle 3 veranschaulicht die gefundenen Ergebnisse. Es wurden insgesamt 10 Messungen durchgeführt, von denen anschließend der Mittelwert herangezogen wurde. Würde man – wie zuvor beschrieben – nicht nur auf die ID zugreifen, sondern die Authentifizierungsmethoden für die gesicherten Speicherbereiche verwenden, welche in Kapitel 2.3 näher beschrieben wurden, würden diese Messwerte aufgrund des höheren Rechen- und Übertragungsaufwandes höher ausfallen.

	Dauer der Lesevorgänge in Sekunden	Lesevorgänge pro Sekunde
10.000 Lesevorgänge inklusive Terminalausgabe	388,396	26
10.000 Lesevorgänge ohne Terminalausgabe	380,289	26

Tabelle 3: Test der Leseperformance mittels RFID

Die in dieser Arbeit aufgestellte Arbeitshypothese H0 ist somit widerlegt, da es möglich ist mit einer auf ARM-Prozessoren basierenden Architektur kostengünstig und redundant Microservices zur Verfügung zu stellen ohne dabei wesentliche Performance Einbußen bei den angeführten Beispieldiensten in Kauf nehmen zu müssen. Ein einzelner Raspberry Pi 3 Model B als Server ist dazu in der Lage in jeder Sekunde mehr als die 8-fache Datenmenge zu verarbeiten, welche ihm ein Client mittels RFID-Ausleseverfahren zur Verfügung stellen kann. Bei etwaiger Optimierung durch weniger SQL-Zugriffe oder der Verwendung einer hardwarenäheren Programmiersprache sind hierbei noch weitere Verbesserungen zu erwarten.

Die gestellte Forschungsfrage kann somit als beantwortet angesehen werden, da der Einsatz von Microservices auf ARM-Prozessor-basierten Architekturen durch das verwendete Betriebssystem und dessen Wartungsmöglichkeiten, die dafür compilierte Software, die eingesetzte Webservice-Architektur, sowie der eingesetzten Hardware beschränkt wird.

ABKÜRZUNGSVERZEICHNIS

3DES	Triple Data Encryption Standard
ADB	Android Debug Bridge
AES	Advanced Encryption Standard
API	Application Programming Interface
APT	Advanced Packaging Tool
ARM	Advanced RISC Machines
Bash	Bourne-again shell
BPDU	Bridge Protocol Data Unit
CISC	Complex Instruction Set Computer
CMT	Cooperative Multitasking
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DDR-SDRAM	Double Data Rate Synchronous Dynamic Random Access Memory
DES	Data Encryption Standard
DHCP	Dynamic Host Configuration Protocol
DTD	Dokumenttypdefinition
DTE	Data Terminal Equipment
eMMC	embedded Multimedia Card
FPS	Frames pro Sekunde
FPU	Floating Point Unit
FTP	File Transfer Protocol
FTPS	File Transfer Protocol over SSL
GPG	GNU Privacy Guard
GPIO	General Purpose Input/Output
GPU	Graphics Processing Unit
HATEOAS	Hypermedia as the Engine of Application State
HLL	High-Level Languages
HTTP	Hypertext Transport Protokol
HTTPS	Hypertext Transfer Protocol Secure
IDL	Interface Definition Language
IoT	Internet of Things
IP	Internet Protocol
IP	Internet Protocol
JSON	JavaScript Object Notation
mA	Milliampere
MicroSD	Micro Secure Digital
MLC	Multi-Level Cell
MSTP	Multiple Spanning Tree Protocol

NTP	Network Time Protocol
PaaS	Platform-as-a-Service
PD	Powered Device
PFS	Perfect Forward Secrecy
PGP	Pretty Good Privacy
PoE	Power-over-Ethernet
PSE	Power Sourcing Equipment
RAM	Random Access Memory
REST	Representational State Transfer
RFID	Radio Frequency Identification
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
RPC	Remote Procedure Call
RPMB	Replay-Protected Memory Block
RSTP	Rapid Spanning Tree Protocol
RTC	Real Time Clock
SD	Secure Digital
SDK	Software Development Kit
SLC	Single-Level Cell
SMTP	Simple Mail Transfer Protocol
SMTPS	Simple Mail Transfer Protocol Secure
SNTP	Simple Network Time Protocol
SOAP	Simple Object Access Protocol
SoC	System on a Chip
SSD	Solid State Drives
SSH	Secure Shell
STP	Spanning Tree Protocol
TCP	Transmission Control Protocol
TCP	Transmission Control Protocol
TFTP	Trivial File Transfer Protocol
TSL	Transport Layer Security
TTL	Time to Live
UART	Universal Asynchronous Receiver Transmitter
UDP	User Datagram Protocol
UID	Unique Identifier
URI	Uniform Resource Identifier
USB	Universal Serial Bus
USV	Unterbrechungsfreie Stromversorgung
UWP	Universal Windows Application API
VFD	Voltage and frequency dependent from main supply
VFI	Voltage and frequency independent from main supply

VI	Voltage independent from main supply
VPU	Vector Processing Unit
WLAN	Wireless Local Area Network
XML	Extensible Markup Language
XSD	XML Schema Definition

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

LISTINGS

LITERATURVERZEICHNIS

Abbott, M. L., & Fisher, M. T. (2009). *The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise.* Pearson Education.

Alasdair, A. (28. Februar 2016). *Meet the New Raspberry Pi 3 — A 64-bit Pi with Built-in Wireless and Bluetooth LE*. Abgerufen am 8. Oktober 2016 von Make: DIY Projects and Ideas for Makers: http://makezine.com/2016/02/28/meet-the-new-raspberry-pi-3/

Albitz, P., & Liu, C. (2002). *DNS und BIND.* O'Reilly Germany.

Annett, R. (19. November 2014). *What is a Monolith?* Abgerufen am 11. Juni 2017 von Coding the Architecture: http://www.codingthearchitecture.com/2014/11/19/what_is_a_monolith.html

Assmann, B. (26. Februar 2013). *SSL offloading impact on web applications.* Abgerufen am 11. Juni 2017 von HAProxy: https://www.haproxy.com/blog/ssl-offloading-impact-on-web-applications/

Auty, M. (3. August 2014). *ikelos/pynfc.* Abgerufen am 5. Juni 2017 von GitHub: https://github.com/ikelos/pynfc

Badach, A., & Rieger, S. (2013). *Netzwerkprojekte: Planung, Realisierung, Dokumentation und Sicherheit von Netzwerken.* Carl Hanser Verlag GmbH Co KG.

Barrett, D., & Silverman, R. (2001). *SSH, The Secure Shell: The Definitive Guide.* O'Reilly.

Bauer, K. (2003). *Automating UNIX and Linux Administration.* Apress.

Bauer, P. (2013). *The Raspberry Pi Computer.* Peter Bauer.

Becher, M. (2009). *XML: DTD, XML-Schema, XPath, XQuery, XSLT, XSL-FO, SAX, DOM.* W3l GmbH.

Bell, C. (2016). *Windows 10 for the Internet of Things.* Apress.

Benz, B. (30. Oktober 2006). Die Technik der Flash-Speicherkarten. *c't 23/2006*, S. 136.

Berners-Lee, T., Fielding, R., & Masinter, L. (Januar 2005). *RFC 3986 - Uniform Resource Identifier (URI): Generic Syntax.* Abgerufen am 5. Februar 2017 von The Internet Engineering Task Force (IETF®): https://tools.ietf.org/html/rfc3986

Bertacco, V., & Legay, A. (2013). *Hardware and Software: Verification and Testing: 9th International Haifa Verification Conference, HVC 2013, Haifa, Israel, November 5-7, 2013, Proceedings* (Bd. 8244). Springer.

Beutelspacher, A., Neumann, H. B., & Schwarzpaul, T. (2012). *Kryptografie in Theorie und Praxis: Mathematische Grundlagen für elektronisches Geld, Internetsicherheit und Mobilfunk.* Springer-Verlag.

Bichler, K., Riedel, G., & Schöppach, F. (2013). *Kompakt Edition: Lagerwirtschaft: Grundlagen, Technologien und Verfahren.* Springer-Verlag.

Böttcher, A. (2007). *Rechneraufbau und Rechnerarchitektur.* Springer-Verlag.

Bourke, T. (2001). *Server Load Balancing.* O'Reilly Media, Inc.

Boyd, C., & Mathuria, A. (2013). *Protocols for Authentication and Key Establishment.* Springer Science & Business Media.

Brown, E. L. (2006). *802.1X Port-Based Authentication.* CRC Press.

Burghardt, M. (2013). *Web Services: Aspekte von Sicherheit, Transaktionalität, Abrechnung und Workflow.* Springer-Verlag.

Canavan, J. E. (2001). *Fundamentals of Network Security.* Artech House.

Carneiro, C., & Schmelmer, T. (2016). *Microservices From Day One: Build robust and scalable software from the start.* Apress.

Chen, C., Novick, G., & Shimano, K. (16. Dezember 2006). *RISC Architecture*. Abgerufen am 23. Oktober 2016 von RISC vs. CISC: https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/

CISCO. (April 2013). *Administration Guide for Cisco TelePresence Software Release 1.10.* Abgerufen am 12. Juni 2017 von CISCO: http://www.cisco.com/c/en/us/td/docs/telepresence/cts_admin/1_10/admin/guide/cts_admin/ctsadmin_cfg.html

CISCO. (20. Januar 2016). *Cisco Universal Power Over Ethernet - Unleash the Power of your Network White Paper.* Abgerufen am 11. Juni 2017 von CISCO: http://www.cisco.com/c/en/us/products/collateral/switches/catalyst-4500-series-switches/white_paper_c11-670993.html

Clark, M. P. (2003). *Data Networks, IP and the Internet: Protocols, Design and Operation.* John Wiley & Sons.

Coy, W. (2013). *Aufbau und Arbeitsweise von Rechenanlagen: Eine Einführung in Rechnerarchitektur und Rechnerorganisation für das Grundstudium der Informatik.* Springer-Verlag.

c't Redaktion. (25. August 2016). Der neue Bootloader. *c't Raspberry Pi (2016): Praxiswissen und Know-how für eigene Projekte*, S. 17.

Daigneau, R. (2011). *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services.* Addison-Wesley.

Dallas, K. (2. Februar 2015). *Windows 10 Coming to Raspberry Pi 2*. Abgerufen am 2. Januar 2017 von Windows Developer: https://blogs.windows.com/buildingapps/2015/02/02/windows-10-coming-to-raspberry-pi-2/

Das, L. B. (2010). *The X86 Microprocessors: Architecture And Programming (8086 To Pentium).* Pearson Education India.

Dean, T. (2009). *Network+ Guide to Networks.* Cengage Learning.

Dear, R. (5. März 2014). *Embedded Computing Design*. Abgerufen am 23. Oktober 2016 von ARM vs x86: Is x86 dead?: http://embedded-computing.com/articles/arm-x86-x86-dead/

Delgado, L. (27. Februar 2016). *Micro-services architecture: a view on scalability.* Abgerufen am 11. Juni 2017 von LinkedIn: https://www.linkedin.com/pulse/micro-services-architecture-view-scalability-luis-delgado-mba-msc

Dembowski, K. (2015). *Raspberry Pi - Das technische Handbuch: Konfiguration, Hardware, Applikationserstellung.* Springer-Verlag.

Eckert, C. (2014). *IT-Sicherheit: Konzepte - Verfahren - Protokolle.* Walter de Gruyter GmbH & Co KG.

E-Control. (7. Juli 2016). *Ausfall- und Störungsstatistik – Ergebnisse 2015.* Abgerufen am 17. April 2017 von E-Control: https://www.e-control.at/documents/20903/388512/Ausfall-+und+Stoerungsstatistik+2015_Stromausfallsdauer_E-Control.pdf

Ericsson. (Juni 2016). *Ericsson Mobility Report.* Abgerufen am 6. Oktober 2016 von Ericsson: https://www.ericsson.com/res/docs/2016/ericsson-mobility-report-2016.pdf

Familiar, B. (2015). *Microservices, IoT and Azure: Leveraging DevOps and Microservice Architecture to deliver SaaS Solutions.* Apress.

Fielding, R. (2000). *Architectural Styles and the Design of Network-based Software Architectures.* Abgerufen am 30. Januar 2017 von UNIVERSITY OF CALIFORNIA, IRVINE: http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf

Fowler, S. J. (2016). *Production-Ready Microservices: Building Standardized Systems Across an Engineering Organization.* O'Reilly Media, Inc.

Furber, S. B. (2000). *ARM System-on-chip Architecture.* Pearson Education.

Garman, J. (2003). *Kerberos: The Definitive Guide: The Definitive Guide.* O'Reilly Media, Inc.

Gay, W. (2014). *Mastering the Raspberry Pi.* Apress.

Geier, J. (2008). *Implementing 802.1X Security Solutions for Wired and Wireless Networks.* John Wiley & Sons.

Geisler, F. (2014). *Datenbanken: Grundlagen und Design.* mitp Verlags GmbH & Co. KG.

Glover, B., & Bhatt, H. (2006). *RFID Essentials.* O'Reilly Media, Inc.

Goasguen, S. (2015). *Docker Cookbook.* O'Reilly Media, Inc.

Gulbransen, D. (2002). *Using XML Schema.* Que Publishing.

Gunther, O., Fabian, B., & Evdokimov, S. (2011). *RFID and the Internet of Things: Technology, Applications, and Security Challenges.* Now Publishers Inc.

Harrington, W. (2015). *Learning Raspbian.* Packt Publishing Ltd.

Hartl, H. (2008). *Elektronische Schaltungstechnik: mit Beispielen in PSpice.* Pearson Deutschland GmbH.

Hartpence, B. (2012). *Praxiskurs Routing und Switching.* O'Reilly Germany.

Herzog, A. (15. September 2015). *Docker*. Von Docker: https://blog.docker.com/2015/09/update-raspberry-pi-dockercon-challenge/ abgerufen

Hofmann, M., Schnabel, E., & Stanley, K. (2017). *Microservices Best Practices for Java.* IBM Redbooks.

Holla, S. (2015). *Orchestrating Docker.* Packt Publishing Ltd.

Horan, B. (2013). *Practical Raspberry Pi.* Apress.

Hruska, J. (26. Februar 2016). *How do SSDs work?* Abgerufen am 29. Januar 2017 von Extreme Tech: https://www.extremetech.com/extreme/210492-extremetech-explains-how-do-ssds-work

Hüwe, S. (2010). *Raspberry Pi für Windows 10 IoT Core: Der praktische Einstieg für Anwender und Entwickler.* Carl Hanser Verlag GmbH Co KG.

hypriot. (Januar 2017). *hypriot/rpi-mysql.* Abgerufen am 18. Juni 2017 von Docker Hub: https://hub.docker.com/r/hypriot/rpi-mysql/

IDC. (31. Mai 2016). *Gradual Change in Server Microprocessor Market; IDC Expects Competition and Evolving Workloads to Change Supply Ecosystem in 2017*. Von IDC Analyze the Future: http://www.idc.com/getdoc.jsp?containerId=prUS41419716 abgerufen

International Energy Agency. (2014). *More Data, Less Energy: Making Network Standby More Efficient.* Abgerufen am 6. Oktober 2016 von International Energy Agency: http://www.iea.org/publications/freepublications/publication/MoreData_LessEnergy.pdf

Jaeger, T. (2008). *Operating System Security.* Morgan & Claypool Publishers.

Javvin Technologies. (2005). *Network Protocols Handbook.* Javvin Technologies Inc.

Johansson, L. (3. Dezember 2014). *What is message queueing?* Von CloudAMQP: https://www.cloudamqp.com/blog/2014-12-03-what-is-message-queuing.html abgerufen

Joos, T. (2006). *Planungsbuch Microsoft-Netzwerke: der Praxisleitfaden für Unternehmen.* Pearson Deutschland GmbH.

Joyner, J. (2015). *Microservices Architecture For Beginners: Build, Integrate, Test, Monitor Microservices Successfully.* Speedy Publishing LLC.

Kaul, O. B. (29. November 2011). *Skalierbarkeit.* Abgerufen am 11. Juni 2017 von Uni Hannover - Software Engineering: http://www.se.uni-hannover.de/pub/File/kurz-und-gut/ws2011-labor-restlab/RESTLab-Skalierbarkeit-Oliver-Beren-Kaul-kurz-und-gut.pdf

Kopparapu, C. (2002). *Load Balancing Servers, Firewalls, and Caches.* John Wiley & Sons.

Krafft, M. F. (2005). *The Debian System: Concepts and Techniques.* No Starch Press.

Labs, L., & Spier, A. (26. November 2016). Kleine Karten, groß und schnell. *c't 25/2016*, S. 110-115.

Langer, J., & Roland, M. (2010). *Anwendungen und Technik von Near Field Communication (NFC).* Springer-Verlag.

Laurent, S. S., Johnston, J., & Dumbill, E. (2001). *Programming Web Services with XML-RPC.* O'Reilly Media, Inc.

Lawton, S. (29. Mai 2014). *Advantages Of Flash In The Data Center: Not Just A Flash In The Pan.* Abgerufen am 29. Januar 2017 von Tom's IT Pro: http://www.tomsitpro.com/articles/flash-data-center-advantages,2-744-2.html

Li, Q. C., & Yao, C. (2003). *Real-Time Concepts for Embedded Systems.* CRC Press.

Love, R. (2010). *Linux Kernel Development* (Bd. 3). Pearson Education.

Lowy, J. (2005). *Programming .NET Components: Design and Build .NET Applications Using Component-Oriented Programming* (Bd. 2). O'Reilly Media, Inc.

Matthias, K., & Kane, S. P. (2015). *Docker: Up & Running.* O'Reilly Media, Inc.

Membrey, P., & Hows, D. (2015). *Learn Raspberry Pi 2 with Linux and Windows 10* (Bd. 2). Apress.

Membrey, P., Plugge, E., & Hows, D. (2012). *Practical Load Balancing: Ride the Performance Tiger.* Apress.

Merten, M. (1. April 2016). Überreife Himbeere: Wie es mit dem Raspberry Pi weitergeht. *c't 8/2016*, S. 148-149.

Micheloni, R., Marelli, A., & Eshghi, K. (2012). *Inside Solid State Drives (SSDs).* Springer Science & Business Media.

Microsoft Corporation. (Dezember 2016). *Release Notes for Windows 10 IoT Core Build Number 14986*. Abgerufen am 4. Januar 2017 von Microsoft Developer: https://developer.microsoft.com/en-us/windows/iot/docs/ReleaseNotesInsiderPreview

Mills, D. L. (September 1985). *RFC 958 - Network Time Protocol (NTP)*. Abgerufen am 31. Oktober 2016 von The Internet Engineering Task Force: https://tools.ietf.org/html/rfc958

Mills, D. L. (Januar 2006). *RFC 4330 - Simple Network Time Protocol (SNTP) Version 4 for IPv4, IPv6 and OSI*. Abgerufen am 31. Oktober 2016 von The Internet Engineering Task Force: https://tools.ietf.org/html/rfc4330

Mitchell, B. (22. März 2017). *What Is a DNS Cache?* Abgerufen am 11. Juni 2017 von lifewire: https://www.lifewire.com/what-is-a-dns-cache-817514

Morrenth, F. (2014). *Sichere Kommunikation verteilter Applikationen über XML Web Services mit WS-Security.* diplom.de.

Morris, K. (2016). *Infrastructure as Code: Managing Servers in the Cloud.* O'Reilly Media, Inc.

Mueller, S., & Soper, E. M. (8. Juni 2001). *Microprocessor Types and Specifications*. Abgerufen am 7. Oktober 2016 von informIT: the trusted technology learning source: http://www.informit.com/articles/article.aspx?p=130978&seqNum=6

Nadareishvili, I., Mitra, R., McLarty, M., & Amundsen, M. (2016). *Microservice Architecture: Aligning Principles, Practices, and Culture.* O'Reilly Media, Inc.

Nemeth, E., Snyder, G., & Hein, T. (2007). *Linux-Administrations-Handbuch.* Pearson Deutschland GmbH.

Nemeth, E., Snyder, G., & Hein, T. R. (2006). *Linux Administration Handbook* (Bd. 2). Addison-Wesley Professional.

Neubig, B., & Briese, W. (1997). *Das große Quarzkochbuch.* Franzis-Verlag Feldkirchen.

Newcomer, E. (2002). *Understanding Web Services: XML, WSDL, SOAP, and UDDI.* Addison-Wesley Professional.

Newman, S. (2016). *Building Microservices.* O'Reilly Media, Inc.

nfc-tools/libnfc. (18. Mai 2017). Abgerufen am 5. Juni 2017 von NFC Tools: http://nfc-tools.org/index.php?title=Main_Page

Noonan, W., & Dubrawsky, I. (2006). *Firewall Fundamentals.* Pearson Education.

NXP. (21. Februar 2011). *Mainstream contactless smart card IC for fast and easy solution development.* Abgerufen am 9. Oktober 2016 von NXP: http://cache.nxp.com/documents/short_data_sheet/MF1SPLUSX0Y1_SDS.pdf?pspll=1

NXP. (21. Februar 2011). *NXP.* Von http://www.nxp.com/documents/data_sheet/MF1S503x.pdf abgerufen

Oppliger, R. (2009). *SSL and TLS: Theory and Practice.* Artech House.

Patterson, D. A., & Sequin, C. H. (1981). RISC I: A Reduced Instruction Set VLSI Computer. *ISCA '81 Proceedings of the 8th annual symposium on Computer Architecture*, S. 443-457.

Piekarski, W. (13. Dezember 2016). *Announcing updates to Google's Internet of Things platform: Android Things and Weave.* Abgerufen am 15. Januar 2017 von Android Developers Blog: https://android-developers.googleblog.com/2016/12/announcing-googles-new-internet-of-things-platform-with-weave-and-android-things.html

PowerShell Team. (18. August 2016). *PowerShell on Linux and Open Source!* Abgerufen am 15. Januar 2017 von Windows PowerShell Blog: https://blogs.msdn.microsoft.com/powershell/2016/08/18/powershell-on-linux-and-open-source-2/

Prunicki, A. (Juni 2009). *Apache Thrift.* Von Software Engineering Tech Trends: http://jnb.ociweb.com/jnb/jnbJun2009.html abgerufen

Prunicki, A. (Juni 2009). *Apache Thrift.* Abgerufen am 6. Februar 2017 von Software Engineering Tech Trends: https://www.ociweb.com/resources/publications/sett/june-2009-apache-thrift/

Puglisi, S. (2015). *RESTful Rails Development: Building Open Applications and Services.* O'Reilly Media, Inc.

Rahm, E. (1994). *Mehrrechner-Datenbanksysteme: Grundlagen der verteilten und parallelen Datenbankverarbeitung.* Addison-Wesley.

Rakowski, K. (2015). *Learning Apache Thrift.* Packt Publishing Ltd.

Raspberry Pi (Trading) Ltd. (Oktober 2016). *RPI CM Datasheet.* Von RaspberryPi.org: https://www.raspberrypi.org/documentation/hardware/computemodule/RPI-CM-DATASHEET-V1_0.pdf abgerufen

Raspberry Pi Foundation. (Februar 2015). *Raspberry Pi - Teach, Learn, and Make with Raspberry Pi*. Von https://www.raspberrypi.org/products/raspberry-pi-2-model-b/ abgerufen

Raspberry Pi Foundation. (Februar 2016). *Raspberry Pi - Teach, Learn, and Make with Raspberry Pi*. Von https://www.raspberrypi.org/products/raspberry-pi-3-model-b/ abgerufen

Ray, C. (2009). *Distributed Database Systems.* Pearson Education India.

Ray, E. T. (2003). *Learning XML: Creating Self-Describing Data.* O'Reilly Media, Inc.

Raýman, G. (2006). *Praxisbuch Objektorientierung: von den Grundlagen zur Umsetzung.* Galileo Press.

Reiter, E. (März 2011). Minimyth – ein Diskless Client für MythTV. *LinuxUser*.

Research and Markets. (April 2016). *Internet of Things (IoT) Market by Software Solution (Real-Time Streaming Analytics, Security, Data Management, Remote Monitoring, & Network Bandwidth Management), Platform, Service, Application Domain, and Region - Global Forecast to 2021*. Abgerufen am 6. Oktober 2016 von Research and Markets - The world's largest market research store: http://www.researchandmarkets.com/research/gsjxb5/internet_of

Richardson, C. (19. Mai 2015). *Introduction to Microservices.* Abgerufen am 22. Januar 2017 von NGINX: https://www.nginx.com/blog/introduction-to-microservices/

Richardson, C. (2016). *Pattern: Client-side service discovery.* Von Microservice Architecture: http://microservices.io/patterns/client-side-discovery.html abgerufen

Richardson, C. (2016). *Pattern: Server-side service discovery.* Von Microservice Architecture: http://microservices.io/patterns/server-side-discovery.html abgerufen

Richardson, C. (9. April 2017). *Monolithic Architecture pattern.* Abgerufen am 11. Juni 2017 von Microservice Architecture: http://microservices.io/patterns/monolithic.html

Robbins, A. (2005). *Unix in a Nutshell: A Desktop Quick Reference - Covers GNU/Linux, Mac OS X,and Solaris* (Bd. 4). O'Reilly Media, Inc.

Romanian Energy Regulatory Authority. (31. August 2016). *National Report 2015.* Abgerufen am 30. April 2017 von Autoritatea Naţională de Reglementare în Domeniul Energiei: https://www.google.at/url?sa=t&rct=j&q=&esrc=s&source=web&cd=7&ved=0ahUKEwjZ8qrh8czTAhVEVxQKHfj7DSIQFghEMAY&url=http%3A%2F%2Fwww.anre.ro%2Fdownload.php%3Ff%3Dgq%252BEgw%253D%253D%26t%3Dvdeyut7dlcecrLbbvbY%253D&usg=AFQjCNGnQoSzFxEXxiroX2dEl__eZTiEOQ&sig2=1

Rost, M., & Wefel, S. (2016). *Sensorik für Informatiker: Erfassung und rechnergestützte Verarbeitung nichtelektrischer Messgrößen.* Walter de Gruyter GmbH & Co KG.

Russ, M. (2016). *Antifragile Software.* Leanpub.

Schiffmann, W. (2006). *Technische Informatik 2: Grundlagen der Computertechnik.* Springer-Verlag.

Schmidt, M. (25. August 2016). Das passende Betriebssystem wählen. *c't Raspberry Pi (2016): Praxiswissen und Know-how für eigene Projekte*, S. 148.

Schwartz, A., Wolff, E., & Heusingfeld, A. (2016). *Microservices: Der Hype im Realitätscheck.* entwickler.Press.

Shanks, E. (18. November 2013). *Disk Latency Concepts.* Abgerufen am 29. Januar 2017 von The IT Hollow: http://theithollow.com/2013/11/18/disk-latency-concepts/

Sheffer, Y., Holz, R., & Saint-Andre, P. (Mai 2015). *Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS).* Abgerufen am 11. April 2017 von Internet Engineering Task Force (IETF) - RFC 7525: https://tools.ietf.org/html/rfc7525

Shroff, G. (2010). *Enterprise Cloud Computing: Technology, Architecture, Applications.* Cambridge University Press.

Skonnard, A. (März 2003). *Understanding SOAP.* Abgerufen am 11. Juni 2017 von Microsoft Developer Network: https://msdn.microsoft.com/en-us/library/ms995800.aspx

Skonnard, A. (März 2003). *Understanding XML Schema.* Abgerufen am 11. Juni 2017 von Microsoft Developer Network: https://msdn.microsoft.com/en-us/library/aa468557.aspx

Snell, J., Tidwell, D., & Kulchenko, P. (2001). *Programming Web Services with SOAP: Building Distributed Applications.* O'Reilly Media, Inc.

Stokes, J. (August 1999). *RISC vs. CISC: the Post-RISC Era*. Abgerufen am 9. Oktober 2016 von ars technica: http://archive.arstechnica.com/cpu/4q99/risc-cisc/rvc-3.html

Studer, B. (2010). *Netzwerkmanagement und Netzwerksicherheit: ein Kompaktkurs für Praxis und Lehre.* vdf Hochschulverlag AG.

Teixeira, S. (10. August 2015). *Hello, Windows 10 IoT Core*. Abgerufen am 4. Januar 2017 von Building Apps for Windows: https://blogs.windows.com/buildingapps/2015/08/10/hello-windows-10-iot-core/

Thompson, R. B., & Thompson, B. F. (2003). *PC Hardware in a Nutshell.* O'Reilly Media, Inc.

Tiernan, R. (19. November 2014). *Qualcomm CEO Mollenkopf Sees Big Opportunity in Server Chips*. Abgerufen am 9. Oktober 2016 von Tech Trader Daily: http://blogs.barrons.com/techtraderdaily/2014/11/19/qualcomm-analyst-day-ceo-mollenkopf-reflects-on-strong-2014-despite-china/

Tseng, H.-W., Grupp, L., & Swanson, S. (5. Juni 2011). *Understanding the Impact of Power Loss on Flash Memory.* Abgerufen am 3. Oktober 2016 von University of California, San Diego: https://cseweb.ucsd.edu/~swanson/papers/DAC2011PowerCut.pdf

Turnbull, J. (2014). *The Docker Book: Containerization is the new virtualization.* James Turnbull.

Upton, E. (8. September 2016). *Ten millionth Raspberry Pi, and a new kit*. Abgerufen am 8. Oktober 2016 von RaspberryPi.org: https://www.raspberrypi.org/blog/ten-millionth-raspberry-pi-new-kit/

Upton, E., & Halfacree, G. (2014). *Raspberry Pi User Guide* (Bd. 2). John Wiley & Sons.

Upton, E., Duntemann, J., Roberts, R., Mamtora, T., & Everard, B. (2016). *Learning Computer Architecture with Raspberry Pi.* John Wiley & Sons.

Upton, L. (11. Juni 2014). *Raspberry Pi Foundation*. Von Raspberry Pi Foundation: https://www.raspberrypi.org/blog/raspberry-pi-at-buckingham-palace-3-million-sold/ abgerufen

USB Implementers Forum, Inc. (27. April 2000). *Universal Serial Bus Specification Revision 2.0.* Abgerufen am 3. 10 2016 von USB.org: http://www.usb.org/developers/docs/usb20_docs/usb_20_091216.zip

Vaudenay, S. (2005). *A Classical Introduction to Cryptography: Applications for Communications Security.* Springer Science & Business Media.

Walk, E., Schuermann, J., Schmidt, E., Oehlmann, H., Bienert, R., & Kovács, N. (2012). *RFID-Standardisierung im Überblick.* Beuth Verlag.

Wang, W. (20. Dezember 2016). *Google 'Android Things' - An Operating System for the Internet of Things.* Abgerufen am 15. Januar 2017 von The Hacker News - Security in a serious way: http://thehackernews.com/2016/12/google-android-things-os.html

Webber, J., Parastatidis, S., & Robinson, I. (2010). *REST in Practice: Hypermedia and Systems Architecture.* O'Reilly Media, Inc.

Werner, H. (2008). *Supply Chain Management - Grundlagen, Strategien, Instrumente.* Springer DE.

Westcott, D. A., & Coleman, D. D. (2015). *CWNA Certified Wireless Network Administrator Official Deluxe Study Guide: Exam CWNA-106.* John Wiley & Sons.

Whitney, T., Jacobs, M., Weston, S., & Satran, M. (8. Februar 2017). *Intro to the Universal Windows Platform.* Abgerufen am 11. Juni 2017 von Windows Dev Center: https://docs.microsoft.com/en-us/windows/uwp/get-started/universal-application-platform-guide

Windeck, C. (8. Dezember 2016). *Windows 10 "Cellular PC": ARM-Chips führen x86-Code aus.* Abgerufen am 11. Juni 2017 von Heise online: https://www.heise.de/newsticker/meldung/Windows-10-Cellular-PC-ARM-Chips-fuehren-x86-Code-aus-3565914.html

Wolff, E. (2015). *Microservices: Grundlagen flexibler Softwarearchitekturen.* dpunkt.verlag.

Wolff, E. (2016). *Microservices: Flexible Software Architecture.* Addison-Wesley Professional.

Wootton, B. (8. April 2014). *Microservices - Not a free lunch!* Abgerufen am 11. Juni 2017 von High Scalability: http://highscalability.com/blog/2014/4/8/microservices-not-a-free-lunch.html

Yener, M., & Dundar, O. (2016). *Expert Android Studio.* John Wiley & Sons.

Yiu, J. (2011). *The Definitive Guide to the ARM Cortex-M0.* Elsevier.

Yu, L. (26. August 2016). *ThriftPy — ThriftPy 0.3.9 documentation.* Abgerufen am 4. Juni 2017 von ThriftPy: https://thriftpy.readthedocs.io/en/latest/

Zeppenfeld, K., & Finger, P. (2009). *SOA und WebServices.* Springer-Verlag.

Zheng, M., Tucek, J., Qin, F., & Lillibridge, M. (12. Februar 2013). *Understanding the Robustness of SSDs under Power Fault.* Abgerufen am 3. Oktober 2016 von https://www.usenix.org/system/files/conference/fast13/fast13-final80.pdf

Zimmermann, J., & Spurgeon, C. E. (2014). *Ethernet: The Definitive Guide* (Bd. 2). O'Reilly Media, Inc.