UNIVERSITÄT DUISBURG-ESSEN
FAKULTÄT FÜR INGENIEURWISSENSCHAFTEN
ABTEILUNG INFORMATIK UND ANGEWANDTE KOGNITIONSWISSENSCHAFT

Bachelorarbeit

Implementation und Evaluation eines Algorithmus für Abhängigkeitsbewahrend Updatekonfigurationen in Smart Home Systemen

Koray Uzun
Matrikelnummer:3081690
Angewandte Informatik (Bachelor)

Fachgebiet Verteilte Systeme, Abteilung Informatik
Fakultät für Ingenieurwissenschaften
Universität Duisburg-Essen

12. September 2021

Erstgutachter: Prof. Dr-Ing. Torben Weis
Zweitgutachter: Prof. Dr. Gregor Schiele
Zeitraum: 21. Juni 2021 - 20. September 2021

Abstract

Smart Home Systeme sind Netzwerke elektronischer Geräte mit dem Ziel die Wohn- und Lebensqualität ihrer Nutzer im eigenen Heim zu erhöhen. Diese Systeme unterliegen, wie jede andere Technologie auch, einem ständigen Wandel. Während viele elektronische Geräte, seien es Smartphones, Tastaturen oder auch Kopfhörer meist nach einigen wenigen Jahren ausgetauscht werden, ist die Erwartungshaltung gegenbüber Smart Home Systemen eine andere. Aufgrund der hohen Anschaffungskosten erwarten Käufer eine hohe Lebensdauer. Gleichzeitig müssen sich die Systeme den Gegebenheiten der Zeit anpassen. Neue Funktionen oder Sicherheitslücken erfordern ein ständiges updaten der einzelnen Geräte in einem System. Mit jedem Update geht jedoch ein großes Risiko einher, da Hersteller nicht garantieren können, dass ein Update keine Schäden im System anrichtet. Dies liegt daran, dass Updates lediglich für das betroffene Gerät auf mögliche Fehlerquellen überprüft werden, nicht jedoch das Zusammenspiel des Updates mit anderen Geräten. Aufgrund der Diversität an Smart Home Systemen und Geräten wäre dies für den Hersteller gar nicht umsetzbar. Daher bedarf es einer Möglichkeit Updates präventiv zu überprüfen, um so größere Schäden zu meiden und die Langlebigkeit von Smart Home Systemen gewährleisten zu können. Diese Arbeit beschäftigt sich mit der Implementation und Evaluation eines Algrorithmus, welcher Abhängigkeiten zwischen Geräten untersucht und basierend auf diesen Abhängigkeiten Updatekonfigurationen ermittelt, welchen den größten Nutzen für den Nutzer haben. Der Algorithmus basiert auf dem Paper "An Algorithm for Dependency-Preserving Smart Home Updates" (Peter Zdankin et. al) [1].

Contents

Chapter 1

Einführung

In den letzten Jahren erfreuen sich Smart Home Systeme immer größerer Beliebtheit. Laut Verified Market Research wird der globale Smart Home Markt im Jahr 2019 auf bereits 80 Milliarden Dollar geschätzt. Prognostiziert werden 207.88 Milliarden Dollar im Jahr 2027, was einer einer jährlichen Wachstumsrate von 13.52 Prozent entsprechen würde[2]. Diese Zahlen spiegeln den Wunsch der Menschen nach digitaler Vernetzung im eigenen Heim wider, um das eigene Heim komfortabler und sicherer zu gestalten [3]. Um den Ansprüchen der Nutzer gerecht zu werden, entwickeln Anbieter immer neuere und bessere Techonologien, wodurch die Systeme immer umfangreicher und komplexer werden. Während 2016 durchschnittlich 5.8 Geräte über ein Smart Home System vernetzt waren, waren es 2018 bereits durchschnittlich 8.1 Geräte [4]. Es ist zu vermuten, dass sich dieser Trend vorerst so weiterentwickeln wird und somit Smart Home Systeme immer größer und komlexer werden. Wie jede andere Technologie auch müssen Smart Home Systeme regelmäßig geupdatet werden, um zum Beispiel Sicherheitslücken zu schließen oder neue Funktionalitäten hinzuzufügen. Ein Smart Home System besteht aus vielen verschiedenen Komponenten, weswegen es nicht möglich ist das System als Ganzes upzudaten. Vielmehr werden einzelne Komponenten eines Systems unabhängig voneinander aktualisiert. Problem hierbei ist, dass Software Updates potenzielle Fehlerquellen beinhalten können, wie fehlerhafte Implementationen. Dies sind jedoch Aspekte die bereits vom Entwickler weitestgehend vermieden werden. Viel Problematischer ist das Zusammspiel verschiedener Geräte und den dazugehörigen Updates. Ein an sich fehlerfreies Update kann bei der Installtion in einem Smart Home System, trotz "fehlerfreier" Programmierung Probleme verursachen. Mit Problemen ist in diesem Kontext gemeint, dass ein Update die Kommunikation zwischen zwei Geräten in einem System unterbrechen kann. Was ist wenn zum Beispiel ein Update eine Funktion eines Geräts so verändert, dass ein anderes Gerät nicht mehr darauf zugreifen kann. Durch solche Änderungen können Abhängigkeiten zwischen Geräten zerstört werden, wodurch ganze Teile eines Smart Home Systems ausfallen können. Solch ein Ausfall bedeutet für den Nutzer meist einen großen finanziellen Schaden, da Updates oft nicht rückgängig gemacht werden können und somit Ersatz beschafft werden muss[1]. Aktueller Stand im Bereich Smart Home ist, dass es keine Möglichkeit gibt Updates vor ihrer Installation zu überprüfen, so dass man als Nutzer vollkommen auf die vielen Entwickler angewiesen ist. Aus diesen

Gründen sind präventive Maßnahmen notwendig, um die Langlebigkeit von Smart Home Systemen gewährleisten zu können.

1.1 Ziel der Arbeit

Eine Steigerung der Lebensqualität ist das Kernziel von Smart Home Systemen. Dies erreicht man zum Beispiel durch die Konfiguration von automatisierten Prozessen. Dadurch können Nutzer Zeit, Energie und in gewissen Belangen auch Geld sparen. Laut Umfragen geben 57 Prozent der Nutzer von Smart Home Systemen in Amerika an jeden Tag durchschnittlich 30 Minuten Zeit einzusparen [5]. Zusätzlich lassen sich nach Berechnungen des Fraunhofer Instituts für Bauphysik durch Automatisierungen bis zu ca. 40 Prozent Heizkosten sparen [6]. Solche Vorteile zeigen, dass sich die anfangs teure Anschaffung eines Smart Home Systemes auf lange Zeit rentieren kann. Sicherlich ist dies der Idealfall, aber dennoch sind die genannten Zahlen Indikatoren dafür, dass Smart Home Systeme viele Potenziale bieten. Es hängt am Ende von vielen Faktoren ab, ob sich die Anschaffung eines Smart Homes lohnt und objektiv ist dies nicht zu beurteilen. Was jedoch objektiv gesagt werden kann, ist dass in den meisten Fällen Nutzer aufgrund der teuren Anschaffungskosten eines Smart Homes davon ausgehen, dass ihr System viele Jahre erhalten bleibt. Zum Beispiel gibt es Smart Home Systeme, welche beim Bau eines Hauses direkt mit geplant werden, sodass man, wie zum Beispiel bei KNX Systemen alle Geräte über ein permanent installiertes Bus System verbinden kann [1]. Selbstverständlich gibt es auch günstigere Varianten, aber dennoch ist die generelle Erwartungshaltung, dass ein funktionierendes System auch über viele Jahre hinweg weiterhin funktionieren wird. Daher ist das Ziel dieser Arbeit die Implementation und Evaluation eines Algrorithmus, welcher Abhängigkeiten zwischen Geräten untersucht und basierend auf diesen Abhängigkeiten Updatekonfigurationen ermittelt, welchen den größten Nutzen für den Nutzer haben. Mit diesem Algorithmus wäre es für Smart Home Nutzer möglich Updates bereits vor der Installation zu überprüfen, um so sicherzustellen, dass sie keine Ausfälle verursachen. Die Anwendung dieses Algorithmus sollte für den Nutzer möglichst einfach sein. So könnte man zum Beispiel ein Smartphone nutzen, welches als zentrales Gerät im System dient. Auf diesem zentralen Gerät wäre es dann möglich den Algorithmus laufen zu lassen und sich über das Smartphone dann für die passende Updatekonfiguration zu entscheiden. Smartphones bieten sich besonders an, da sie eine intuitive Bedienung für den Nutzer ermöglichen und im Normalfall jeder Nutzer eines Smart Homes ein Smartphone besitzt. So umgeht man das Problem, dass es, wie bereits erwähnt, etliche Smart Home Anbieter mit verschiedener Software und Geräten gibt.

1.2 Aufbau der Arbeit

Kapitel 2 befasst sich mit aktuellen Technologien im Smart Home Bereich und bildet eine Grundlage für das Verständnis der restlichen Arbeit. Die Literatur auf der diese Arbeit aufbaut wird vorgestellt und zusätzlich werden die Themen Langlebigkeit und Dependency Management im Software Bereich untersucht. Im dritten Kapitel 3 wird das Design und die Implementation des Algorithmus behandelt. Kapitel 4 umfasst eine Evalutation des Algorithmus hinsichtlich Performance und Designentscheidungen. Das letzte Kapitel fasst die Ergebnisse der Arbeit zusammen und thematisiert mögliche Verbesserungen der Arbeit.

Chapter 2

Related Work

In diesem Kapitel wird das Thema Langlebigkeit im Bereich Software Engineering thematisiert. Infolgedessen werden Smart Home Systeme unter dem Aspekt der Langlebigkeit genauer beleuchtet und das Thema Dependency Management aufgegriffen, wobei bereits bestehende Ansätze/Lösungen diesbezüglich genauer betrachtet werden. Zusätzlich wird der aktuelle Stand des Dependency Managements im Bereich Smart Home untersucht.

2.1 Smart Homes

Smart Home Systeme sind grundlegend nichts anderes als die Verknüpfung elektronischer Geräte. Durch die hohe Anzahl der Anbieter für Smart Home Systeme existieren jedoch viele Architekturen, die teilweise sehr unterschiedlich funktionieren . Im Rahmen dieser Arbeit ist es nicht notwendig die verschiedenen Smart Home Architekturen genauer zu beleuchten, dennoch schadet es nicht sich einen groben Überblick über die aktuellen Technologien zu verschaffen. Grob können die verschiedenen Architekturen darin unterscheiden werden, ob sie eine Verknüpfung zu einer Cloud besitzen. Cloud gebundene Architekturen bieten dem Nutzer meist eine einfachere Installation und Wartung, indem zum Beispiel Updates automatisch installiert werden. Gleichzeitig bringen sie aber auch potenzielle Gefahren mit sich, da die Abhängigkeit zu einer Cloud Smart Home Systeme verwundbar machen kann. Wird ein externer Service, wie zum Beispiel die Cloud abgeschaltet, führt dies bei Cloud-gebundenen Archtiekturen oft zu Ausfällen im System[7]. Des Weiteren sind Geräte, die mit einer Cloud verbunden sind anfälliger gegenüber Angriffen von Außenstehenden, die sich Zugriff auf das System beschaffen wollen. Nicht Cloud-gebundene Architekturen haben diese Nachteile nicht. Es ist jedoch meist schwieriger eine solche Architektur aufzubauen, da zum Beispiel, wie im Falle von OpenHab[?] viel manuelle Konfiguration benötigt wird. Zusätzlich bieten nicht Cloud-gebundene Architekturen mehr Sicherheit, da sie keine starken Abhängigkeiten zu anderen Dienstleistern benötigen. Natürlich könnte man Smart Home Systeme noch weiter spezifizieren, indem man einzelne Komponenten genauer beleuchet und verschiedene Smart Home Anbieter miteinander vergleicht, jedoch ist dies, wie bereits erwähnt, im

Rahmen dieser Arbeit nicht notwendig. Der zu entwickelnde Algorithmus soll später theoretisch Platform- und Architekturunabhängig anwendbar sein. Zu diesem Zweck werden im Folgenden einige Aspekte bezüglich Smart Homes definiert und vorausgesetzt. Zdankin beschreibt in seiner Arbeit Smart Home Systeme als:

> *"A smart home has a set of devices that are connected through a platform. Each device has a certain software version and a set of available updates. Each software version has a set of available predefined services. Devices can use services of other devices which creates dependencies. An update configuration is one of the finite states of the nondeterministic finite automaton (NFA) that can be constructed by using the configurations as states and connecting them using individual updates as transitions." (Zdankin et al., 2020, Requirements and Mechanisms for Smart Home Updates [8])*

Um Missverständnisse zu vermeiden folgen nun zusätzlichen Begriffserklärungen. Die folgenden Elemente in Abbildung 2.1 sind für ein Smart Home System also essentiell:

- Device (Gerät): Geräte sind die Komponenten, die die eigentliche Funktionalität anbieten. Sie besitzen eine aktuelle Firmware Version mit entsprechenden Diensleistungen(Services) und potenzielle Updates. Zusätzlich können sie Abhängigkeiten zu anderen Geräten haben, indem sie auf deren Dienstleistungen zugreifen.
- Firmware Version: Die Firmware Version gibt an, auf welcher Version ein Gerät sich befindet und gibt somit an welche Dienstleistungen ein Gerät anbietet.
- Service (Dienstleistungen): Eine Dienstleistung ist eine Funktionalität, die ein Gerät anbietet. Zum Beispiel bietet ein Thermostat die Dienstleistung "Temperatur messen" an.
- Update: Eine Aktualisierung der Firmware eines Geräts. Durch ein Update können sich angebotenen Diensleistungen eines Geräts ändern.
- Platform: Die Komponente, die alle Geräte miteinander vernetzt.
- Dependency (Abhängigkeiten): Eine Abhängigkeit zwischen zwei Geräten besteht, wenn ein Gerät Zugriff auf die Dienstleistungen eines anderen Geräts benötigt.

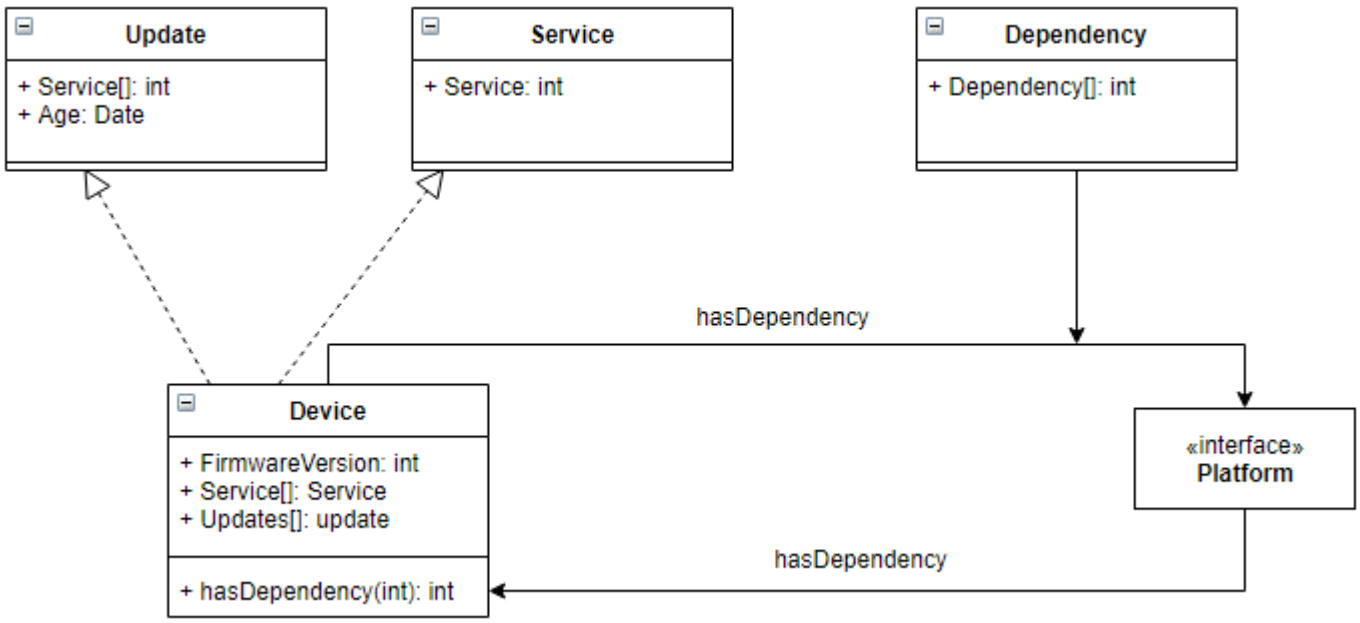

Figure 2.1: Grundlegende Komponenten eines Smart Home Systems

2.1.1 Prescriptive vs Descriptive Standards

Für die Kommunikation der Geräte untereinander existieren viele Kommunikationsprotokolle wie zum Beispiel Bluetooth, Wi-Fi oder auch XMPP. Diese sind bereits im großen Umfang in Verwendung und stark erforscht, weshalb es keiner weiteren Untersuchung dieser Protokolle im Rahmend dieser Arbeit bedarf. Kommunikation allein recht jedoch nicht, die Geräte müssen nicht nur untereinander kommunizieren, sondern sie müssen sich auch verstehen. Dafür ist ein Standard notwendig, welcher die Funktionen/Dienstleistungen der Geräte beschreibt. Die zwei führenden Ansätze im Bereich IoT/Smart Homes sind deskriptive und preskriptive Standards. Zdankin beschreibt die beiden Standards in seiner Arbeits folgendermaßen:

"A standard for service definitions that only regulates how a device can describe itself is a descriptive standard, because it allows vendors to describe their devices in their own terms. A recently finalized descriptive standard is the Web Of Things [Ka20; Ko20]. If devices need to communicate across different descriptive standards, a translation must be considered. However, McCool et al. have stated security concerns about purely descriptive approaches, such as scanning for door locks with known physical weaknesses [MR18] (Zdankin et al., 2020, Requirements and Mechanisms for Smart Home Updates [8]).

TODO: Wie muss ich die Quellenangaben im Zitat zitieren?

A prescriptive standard does not only regulate the how but also the what of self-description. These standards prescribe how devices and services should be defined and vendors can use these terms for unambiguity. By using the same terms to define services, translation is not required anymore. A prescriptive standard defines all possible devices and all their services they could implement. However, a device can implement a subset of these services only. This restriction has benefits as well because it ensures compatibility across device vendors. Widely used examples are the smart home standards of Amazon, Apple, and Samsung6." (Zdankin et al., 2020, Requirements and Mechanisms for Smart Home Updates [8])

In der Realität bieten sich hybride Standards vermutlich am meisten an, da sowohl deskriptive als auch preskriptive Standards Vor- und Nachteile besitzen.

2.1.2 Update Planing

Oft werden Smart Home Geräte automatisch über eine aktive Internetverbindung geupdatet oder auch durch andere Geräte wie zum Beispiel einem Hub. Dies bietet dem Nutzer mehr Komfort, da kein manueller Aufwand für Updates ensteht. Solches "Remote Updating" sollte aber eigentlich vermieden werden, da dadurch zum Beispiel schädliche Software installiert werden kann. Außerdem müsste der externe Diensleister, der das Updaten der Geräte übernimmt, viele Informationen über das System sammeln, um so ideale Updatekonfigurationen zu finden. Das Preisgeben aller Informationen sollte nicht leichtsinnig getan werden, da es viele potenzielle Gefahren birgt. Allein die Information, wann eine automatisierte Heizung anfängt zu heizen kann potenziellen Kriminellen Informationen darüber geben, wann jemand zu Hause ist oder wann ein Haus leer steht. Idealerweise sollten Nutzer daher das Updaten lokal selbst übernhemen. Dies bedeutet selbstverständlich mehr manuelle Konfiguration, vor jedem Update, da der Nutzer gefragt werden sollte, ob er ein Update installieren möchte. Diese Frage kann ein Nutzer ohne weitere Informationen gar nicht plausibel entscheiden. Es wäre also von Vorteil, wenn es für den Nutzer eine Übersicht über alle möglichen Updates geben würde, in der er sich dann für passende Updates entscheiden kann. Für einen solchen Zweck bietet sich Smartphone als zentrales Gerät an um eine Übersicht über alle im System vorkommenden Geräte zu erstellen. Dazu benötigt man eine Möglichkeit die Informationen über die Services und Updates eines Geräts leicht abfragen zu können. Eine solche Abfrage müsste in der Realität für jedes Gerät individuell angepasst werden. Dies ist jedoch nicht umsetzbar, weswegen es nötig ist einen einheitlichen Weg zu finden. Zdankin schlägt vor die Metadaten der Geräte zu diesem Zweck zu nutzen. Geräte besitzen üblicherweise Metadaten, welche Informationen über Software Version, Speicherbedarf usw. enthalten. Diesen Metadaten könnte man zusätzlich Informationen über Dienstleistungen und Updates hinzufügen, sodass man einen schnellen Überblick über alle Geräte innerhalb eines Netzwerks erhalten kann (Zdankin et al., 2020, Requirements and Mechanisms for Smart Home Updates [8]).

2.2 Langlebigkeit

Im Bereich Software Engineering ist Langlebigkeit ein bekanntes Problem. Es wurden bereits Untersuchungen über die durchschnittliche Lebensdauer von Software durchgeführt mit dem Ergebnis, dass kleine Projekte im Bereich von unter 100.000 Zeilen Code wird eine Lebensdauer von circa 6-8 Jahren haben. Dabei ist zu beobachten, dass mit steigender Größe eines Projekts die erwartete Lebensdauer im Durchschnitt ebenfalls steigt [9]. Dies hängt vermutlich mit den hohen Kosten zusammen, die für größere Projekte aufkommen, weswegen man daran interessiert ist lange einen Nutzen von der Software zu haben. Gründe für das Altern von Software sind vielseitig. So kann zum Beispiel die Wartung von Software einfach zu teuer werden, so dass es sich lohnt neue Software zu entwickeln oder eine Entwicklungsumgebung wird nicht mehr unterstützt und es lohnt sich nicht die Software zu portieren. Andere Gründe können Sicherheitslücken oder auch fehlerhafte Updates sein. Die German Research Foundation hat ein Projekt namens Design for Future ins Leben gerufen, um das Problem der Langlebigkeit von Software zu untersuchen. Das Projekt beschreibt sein Ziel darin fundamentale neue Ansätze im Bereich langlebiger Software zu finden, um Probleme mit Legacy-Software oder der Adaption von Software auf neue Plattformen zu lösen und eine kontinuierliche Weiterentwicklung von Software Systemen gewährleisten zu können [10]. Im Bereich Smart Home existieren keine solche Programme. Das Problem der Langlebigkeit wird in diesem Kontext unterschätzt oder einfach ignoriert. Aufgrund der Komplexität der Systeme besitzen sie viele Schwachstellen die zu einem frühzeitigen Ausfall des Systems führen können. Dazu gehören:

Threat
Discontinued External Services
Breaking Updates
New class of devices
Trade Conflict
Abandoned Protocol
Forced Incompatibility
Second Hand Smart Homes

Figure 2.2: Gefahren für die Langlebigkeit von Smart Home Systemen[1]

1

In dieser Arbeit wird hauptsächlich die Gefahr der "Breaking Updates" thematisiert. "Breaking Updates" können Abhängigkeiten im System zerstören und sind somit Schäden gegen die man als Nutzer nichts unternehmen kann. Eine Möglichkeit wäre es Updates generell nicht zu installieren, wodurch Nutzer auf neue Funktionalitäten verzichten müssten, während gleichzeitig das Risiko steigt, dass Sicherheitslücken im System aufkommen. Updates also grundlegend zu ignorieren ist nicht empfehlenswert, weswegen es einer

[1] Zdankin et al, 2020, An Algorithm for Dependency-Preserving Smart Home Updates

Möglichkeit bedarf Updates vor ihrer Installation zu überprüfen. Eine andere Gefahr ist zum Beispiel, dass ein externer Service, der für das Smart Home System notwendig ist, offline gehen kann (Discontinued External Service).[1] Der Unteschied zu den "Breaking Updates" besteht darin, dass man als Käufer eines Cloud-gebundenen Smart Home Systems diese Gefahr aktiv und willentlich eingeht. Die ist bei "Breaking Updates" nicht der Fall. Außerdem lässt sich das Ausmaß der Gefahr eines "discontinued Services" weitesgehend vermeiden, indem man bei etablierten Anbietern einkauft oder komplett auf die Abhängigkeit zu einem externen Service verzichtet. Auch dies ist bei "Breaking Updates" nicht möglich.

2.3 Dependency Managment

Unter Dependency Management versteht man im Allgemeinen das Strukturieren von Abhängigkeiten verschiedener Systeme/Programme. Oft können Programme nur in Abhängigkeit von anderen Programmen starten. Dies stellt kein Problem dar, solange die Abhängigkeiten in ihrer Anzahl überschaubar bleiben. Realität ist jedoch, dass Programme meist von mehreren Programmen abhängig sind und diese Prorgramme wiederum abhängig von anderen Programmen. Bei zu vielen Abhängigkeiten verliert man als Entwickler und auch als Nutzer schnell den Überblick. Dieses Wirrwarr an Abhängigkeiten bezeichnet man umgangssprachlich als "Dependency Hell", denn sobald man einmal den Überblick verloren hat, wird es äußerst schwierig weitere Änderungen am System vorzunehmen.

2.3.1 Probleme

Es folgt eine kleine Zusammenfassung, der am häufigsten auftretenden Formen der "Dependency Hell" [11]:

- Long Chain of Dependencies: Ein Programm A ist abhängig von einem Programm B. Das Programm B wiederum ist abhängig von einem Programm C. Möchte man nun Programm A nutzen benötigt man zusätzlich Programm B und damit auch Programm C. Solche lange Ketten können Konflikte verursachen. (Siehe Conflicting Dependencies)
- Conflicting Dependencies: Sei ein Programm A abhängig von einem Programm B1.1, ein anderes Programm B ist abhängig von Programm B1.2. Programm B kann aber nicht gleichzeitig mit der Version 1.1 und der Version 1.2 installiert sein. Dies bedeutet, Programm A und B können nicht gleichzeitig in einem System laufen.
- Circular Dependencies: Sei ein Programm A abhängig von einer spezifischen Version von Programm B. Programm B wiederum ist abhängig von einer spezifischen Version von Programm A. Beide Programme können nun nicht geupdatet werden, da sie die Abhängigkeiten kaputt verletzen würden.
- Version Lock: Durch einen Version Lock ist man dazu gewungen beim updaten einer Software alle abhängigen Programme ebenfalls upzudaten. (semver.org)
- Version Promiscuity: Werden Abhängigkeiten zu locker betrachtet können zukünftige Updates als kompatibel angesehen werden, obwohl sie es nicht sind.

Für die genannten Probleme existieren bereits viele Ansätze, um gegen sie vorzugehen oder sie zu vermeiden. Dazu gehören [11]:

2.3.2 Lösungen

- Semantic Versioning [12]: Semantic Versioning versioniert Software nach dem Format "MAJOR.MINOR.PATCH", um Probleme wie Version Lock oder Version Promiscuity zu vermeiden. Dafür benötigt man eine Schnittstelle der man die Änderungen an der Software mitteilt. Bei Änderungen, die die API betreffen und nicht abwärtskompatibel sind muss die MAJOR Nummer erhöht werden. Bei Änderungen, die abwärtskompatibel sind, muss die MINOR Nummer erhöht werden. Änderungen, die die API nicht betreffen erhöhen die Patch Nummer. Es gibt noch viele weitere Spezifikationen bezüglich Semantic Versioning, das grobe Konzept sollte jedoch verstanden sein.
- Package Manager: Package Manager sind Tools die sich um das installieren, updaten und konfigurieren von Programmen automatisch kümmern, wodurch eine manuelle Konfiguration nicht mehr nötig ist.
- Portable Applications: Portable Applikationen umgehen die Probleme, die Abhängigkeiten mit sich bringen. Sie funktionieren selbstständig, indem alle nötigen Komponenten bereits vorhanden sind.

Die Auflistungen sollen zeigen, dass Dependency Management ein allgegenwärtiges Thema ist, welches in den meisten Bereichen bereits viel Aufmerksamkeit bekommt. Dependency Management umfasst natürlich noch viele weitere Probleme und dazugehörige Lösungen und Lösungsansätze, im Bereich Smart Home exisitiert jedoch nahezu keine Forschungsgrundlage. Als Nutzer kann man nur hoffen, dass neue Updates keine Schäden anrichten, da es keine Möglichkeit gibt sich davor zu schützen.

2.4 Fazit

Im Bereich Software Engineering wird Langlebigkeit viel Aufmerksamkeit geschenkt. Smart Home Systeme sind selbstverständlich Teil des Software Engineerings, aber schaut man sich den aktuellen Stand der Forschung der Langlebigkeit explizit im Bereich der Smart Home Systeme an, wird deutlich das diesbezüglich kaum Forschungsgrundlagen bestehen. Dabei werden Smart Home Systeme als Luxusgüter angesehen von denen man aufgrund der hohen Preise eine lange Lebensspanne erwartet. Um dies zu gewährleisten werden generelle Ansätze des Software Engineerings bezüglich Langlebigkeit genutzt, ausreichend ist dies jedoch nicht. Ohne vernünftiges Dependency Management kann nicht gewährleistet werden, dass Updates keine Ausfälle in Smart Home Systemen verursachen. Dies sieht man am Beispiel des Logitech Harmony Hub's [13]. Logitech hatte eine neue Firmware Version für das Hub veröffentlicht, welche Sicherheitslücken schließen und Bugs fixen sollte. Nachdem Update kam es vermehrt zu Ausfällen zwischen dem Hub

TODO: umschreiben

und third-party Geräten. Den Nutzern waren die Hände gebunden, sie konnten nur darauf hoffen, dass Logitech die Probleme wieder behebt. Dies hat Logitech in diesem Fall auch getan, dennoch zeigt dieses kleine Beispiel, wie leichtsinnig Updates von Nutzern installiert werden. Eine andere Möglichkeit besitzt man auch als Nutzer eines Smart Homes nicht wirklich, da es wie eingangs erwähnt, zur Zeit keine Möglichkeiten gibt Updates vor der Installation zu überprüfen.

Chapter 3

Design und Implementation

Dieses Kapitel beschreibt das Design der Implementation des Algorithmus und erklärt Entscheidungen, die im Laufe der Implementation getroffen wurden.

3.1 Oberflächliches Design

Zunächst folgt eine grobe Beschreibung aller Schritte des Algorithmus, um so als Leser genauere Entscheidungen im Laufe der Arbeit besser nachvollziehen zu können. Der Algorithmus basiert auf dem Paper "An Algorithm for Dependency-Preserving Smart Home Updates" (Peter Zdankin et al, 2020) [1]. Ziel ist es ist, wie Bereits in Kapitel 1.1 erwähnt, Abhängigkeiten zwischen Geräten zu untersuchen und basierend auf diesen Abhängigkeiten Updatekonfigurationen zu ermitteln, welchen den größten Nutzen für den Nutzer haben.

1. Im ersten Schritt wird eine Übersicht über die aktuelle Updatekonfiguration erstellt. Eine Updatekonfiguration ist eine Zusammenfassung aller Geräte und ihrer aktuellen Firmwareversion. Dadurch weiß man, welche Dienstleistungen die Geräte aktuell anbieten. Zusätzlich wird eine Liste aller möglichen Updates eines Geräts erstellt.

2. Der zweite Schritt ist eine Optimierung bei der versucht wird die Anzahl der interessanten Updates zu verringern. Dazu werden dominierte Updates herausgefiltert und im weiteren Verlauf nicht mehr in Betracht gezogen. Wie genau dies geschieht und welche Updates dominiert sind, wird in einem späteren Kapitel erläutert.

3. Im dritten Schritt wird mit Hilfe der Übersicht über alle Geräte und Updates ein Updatekonfigurationsgraph erstellt. Dieser Graph kann in der Theorie als nichtdeterministischrn endlicher Automat realisiert werden, bei dem jeweils ein Zustand eine Updatekonfiguration darstellt. In Abbildung 3.1 ist ein Teil eines Graphes dreier Geräte mit jeweils zwei Updates zu sehen. Der Start-Zustand zeigt, dass sich Gerät 1 auf Version 0, Gerät 2 auf Version 2 und Gerät 3 auf Version 4

befindet. Die anderen Zustände bilden die anderen Möglichkeiten auf die man die Geräte updaten kann, wodurch andere Updatekonfigurationen entstehen. [1]

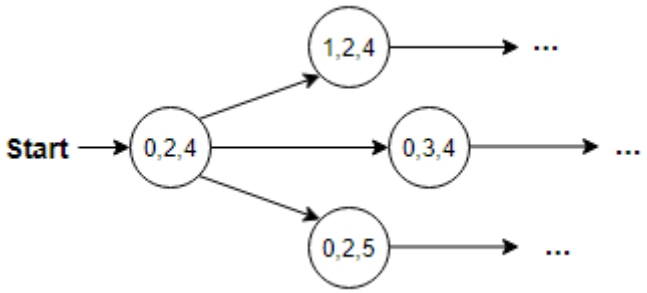

Figure 3.1: Ausschnitt eines Updatekonfigurationgraphs[1]

4. Im vierten Schritt werden alle Updatekonfigurationen dahingehen überprüft, ob sie eine Abhängigkeit verletzen. Ist dies der Fall wird diese Konfiguration aus dem Updatkonfigurationsgraphen entfernt.

5. Im fünften Schritt werden alle nicht Pareto-optimalen Konfigurationen herausgefiltert. Pareto-optimal wird im späteren Verlauf definiert.

6. Im sechsten Schritt werden die Updatekonfigurationen in Subnetzwerke unterteilt. Besteht zum Beispiel keinerlei Abhängigkeit zwischen zwei Gruppen von Geräten in einem System, kann man diese Gruppen in Subnetzwerke unterteilen. Für diese Gruppen kann man später unabhängig von anderen Gruppen die passende Updatekonfiguration auswählen.

7. Im letzten Schritt wird jeder Updatekonfiguration ein Rating zugewiesen mit dem Ziel dem Nutzer die Entscheidung für die passende Updatekonfiguration zu erleichtern.

3.2 Konkretes Design

Basierend auf den Erkenntnissen aus den Kapiteln 1 und 2 können nun Designentscheidungen bezüglich des Algorithmus getroffen werden. Ziel ist es den Prototypen eines zukunftsfähigen Tools zu entwickeln, welcher Smart Home Nutzern die Möglichkeit bietet Updates vor ihrer Installation zu überprüfen. Dabei zielt das Design darauf ab, unnötige architekturspezifische Regularien zu vermeiden, um so möglichst simpel gestaltet werden zu können. Nichtsdestotrotz sollte es möglichst einfach sein den Algorithmus

[1] Zdankin et al, 2020, An Algorithm for Dependency-Preserving Smart Home Updates

in verschiedene Smart Home Architekturen zu integrieren. Um solch eine einfache Integration zu gewährleisten wird die Implementation daher in Java sehr modular gehalten. Java bietet sich dafür als objektorientierte Programmiersprache sehr an. Des Weiteren wird der Algorithmus nicht auf der Basis eines realen Smart Home Systems aufgebaut. Dies wäre kontraproduktiv, da man als Entwickler eher dazu neigen würde eine systemspezifische Implementation zu entwickeln. Anstatt also ein reales Smart Home Systems als Grundlage zu nehmen werden im Folgenden andere Möglichkeiten genauer beleuchtet. Eine Möglichkeit ist es eine Liste mit Geräten, aktueller Firmware Version, möglichen Updates und allen dazugehörigen Dienstleistungen zu erstellen. Fügt man dieser Liste dann noch Abhängigkeiten hinzu erhält man alle wichtigen Informationen über ein Smart Home System, die man benötigt. Vorteil einer solchen Liste ist, dass es sehr einfach wäre sie zu erstellen. Für die spätere Evaluation des Algorithmus wäre solch eine statische Liste jedoch kontraproduktiv, da man den Algorithmus somit immer am gleichen System testen würde. Dadurch könnte man keine validen Aussagen über die Performance des Algorithmus treffen. Aus diesem Grund wird ein Generator genutzt, welcher in Abhängigkeit von verschiedenen Parametern Smart Home Systeme kreiert. So können die Anzahl der Geräte, die Anzahl der möglichen Updates pro Gerät, die Anzahl der Dienstleistungen pro Gerät oder auch die Anzahl der Abhängigkeiten zwischen den Geräten variiert werden. Dies ermöglich den später implementierten an Algorithmus an verschiedenen Systemen zu evaluieren, um so ein gutes Fazit über Effiziens und Lauftzeit treffen zu können.

Generator

Das Design eines solchen Generators lässt sich in Java sehr leicht gestalten. Der Generator erzeugt Geräte, welche in Form von Objekten gespeichert werden. Diese Objekte enthalten Informationen über die Version, die Dienstleistungen und die möglichen Updates des Geräts. Zusätzlich wird noch das Alter der Updates für spätere Zwecke gespeichert. Idealerweise sollten Updates nach den Spezifikationen des Semantic Versioning definiert werden, da es sich dabei um eine weit verbreitete Art der Versionierung handelt und die Implementation sich damit an aktuelle Standards hält. Um jedoch im Rahmen dieser Arbeit die Implementierung zu vereinfachen wird auf diesen Standard verzichtet und Updates werden im ganzzahligen Bereich von 0 an hochgezählt. Dementsprechend kommt nach dem Update 0 das Update 1 und so weiter.

```
public class Device {
    int version;
    ArrayList<Integer> services;
    ArrayList<ArrayList<Integer>> updates;
    ArrayList<ArrayList<LocalDate>> updateAge;
}
```

Wie bereits in Kapitel 2.1.1 erwähnt unterscheiden sich Smart Home Geräte hinsichtlicher ihrer Definition. Da ohne klare Definitionen eine Kommunikation zwischen den Geräten nicht möglich wäre, wird davon ausgegangen, dass alle Geräte und Dienstleistungen unter einem preskriptiven Standard definiert sind. Die Geräte können also untereinander kommunzieren ohne, dass eine Übersetzung nötig ist, wie es unter einem deskriptiven Standard der Fall wäre. Rein für die Implementierung des Algrorithmus ist es von Vorteil sich an preskriptiven Standards zu orientieren, da durch die Homogenität der Geräte und die redundanzfreie Architektur die Komplexität des Algorithmus gemindert werden kann.

Implementation

Nachdem die Geräte erstellt wurden, müssen Abhängigkeiten zwischen diesen Geräten kreiert werden, um ein Smart Home System zu simulieren. Eine Abhängigkeit herrscht zwischen zwei Geräten, wenn ein Gerät Zugriff auf die Dienstleistungen eines anderen Geräts benötigt. In Abbildung 3.2 sieht man die Abhängigkeit zwischen einer Heizung und einem Thermostat. Die Heizung greift periodisch auf die Dienstleistung des Thermostats zu, indem es das Thermostat nach der aktuellen Temperatur fragt. Liegt die Temepratur nun unter einer bestimmten Schwelle würde die Heizung automatisch anspringen und so gewährleisten, dass immer eine Mindesttemperatur vorhanden ist.

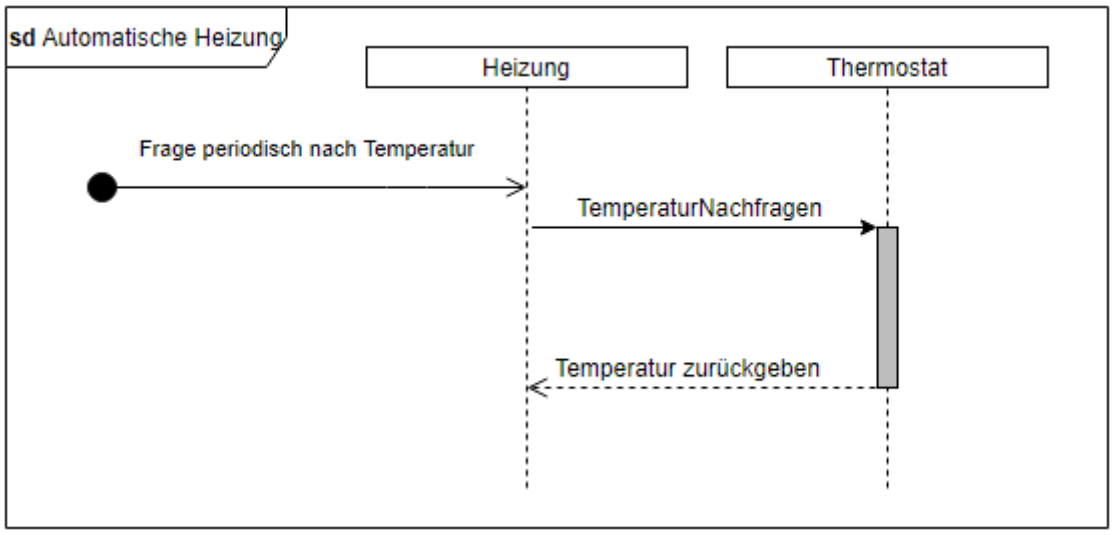

Figure 3.2: Abhängigkeit zwischen zwei Geräten

Die zwei Komponenten Geräte und Abhängigkeiten reichen bereits aus, um ein theoretisches Smart Home System zu bilden, welches für die Anwendung des Algorithmus völlig ausreicht. Weitere Voraussetzungen sind, dass innerhalb eines solchen Systems keine Updates automatisch installiert werden und es möglich ist, wie in Kapitel 2.1, beschrieben, sich eine Übersicht über alle Geräte und deren Informationen zu beschaffen. In realen Systemen müssten diese Informationen in den Metadaten der Geräte gespeichert werden, um sie leicht abfragen zu können. Da Metadaten bei den meisten Geräten bereits existieren, sollte es für die Entwickler dieser Geräte kein großes Problem darstellen die Metadaten um diese Informationen zu ergänzen. Genauere Details, wie zum Beispiel, ob die Kommunikation über eine Cloud läuft sind nicht interessant und werden dementsprechend vernachlässigt. Nun steht die Basis für die Implementation des Algorithmus. Im ersten Schritt werden die Updates aller Geräte untersucht. Ziel dieses Schrittes ist es, dominierte Updates zu finden und diese zu löschen, um so Rechenpower einzusparen. Updates sind dominiert, wenn ein anderes Update exisitiert, welches mindenstens die gleichen Dienstleistungen anbietet und zusätzlich aktueller ist.

Version	Services
1.0	1,2
1.1	1,2
2.0	1,3
2.1	1,3
3.0	3,4

Figure 3.3: Gerät mit 5 Versionen/Firmwareaktualisierungen und den dazugehörigen Dienstleistungen[1]

[2] In Abbildung 3.3 ist ein Gerät mit 5 verschiedenen Versionen zu sehen. In diesem Beispiel ist die Version 1.0 von der Version 1.1 dominiert, da Version 1.1 die gleichen Dienstleistungen wie Version 1.0 anbietet und gleichzeitig aktueller ist. Das Gleiche gilt für Version 2.0 und 2.1. Für dieses Gerät bleiben also nur noch drei potenzielle Updates, nämlich Version 1.1, 2.1 und 3.0 übrig. Diese Optimierung an allen Geräten im System durchzuführen spart eine Menge Rechenzeit. Im nächsten Schritt wird nämlich mit Hilfe des kartesischen Produkts aller restlichen, nicht-dominierten Updates ein Updatekonfigurationsgraph erstellt. Dieser Graph beinhaltet alle möglichen Updatekonfigurationen, die im System installiert werden könnten. Seien also in einem System 10 Geräte mit je 2 Updates bedeutet dies, dass es $2^(10) = 1024$ verschiedene Updatekonfigurationen geben würde. Bei 3 Updates pro Gerät wären es schon $3^(10) = 59049$ verschiedene Konfigurationen. Dies macht deutlich, dass jedes im ersten Schritt als dominiert erkanntes Update die Laufzeit des Algorithmus erheblich verringern kann. Wie bereits erwähnt kann der Updatekonfigurationsgraph in der Theorie als nichtdeterministischer endlicher

[2]Zdankin et al, 2020, An Algorithm for Dependency-Preserving Smart Home Updates

Automat realisiert werden. In Java wird dieser Automat in Form einer mehrdimensionalen Arrayliste dargestellt. Gegeben seien 2 Geräte, zum Beispiel eine Heizung und ein Thermostat Die Heizung ist momentan auf Version 0 mit den Services {1,2 } und besitzt keine Updates. Das Thermostat ist momentan auf Version 0 mit den Services {5,7 } und kann auf Version 1 mit den Services {6,8 } geupdatet werden. Der Updatekonfigurationsgraph sieht in Java dann folgendermaßen aus:

TODO: Noch ne abbildung mit NFA

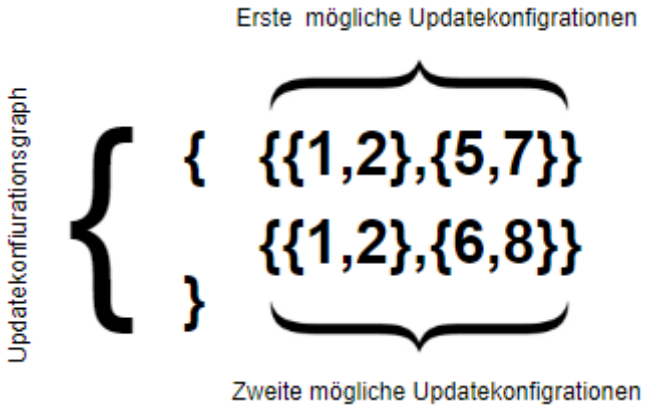

Figure 3.4: Updatekonfigurationsgraph in Form einer List in Java

Nun wird der Updatekonfigurationsgraph nach ungültigen Konfigurationen durchsucht. Eine Konfiguration ist ungültig, wenn bestimmte Dienstleistungen fehlen, die eigentlich aufgrund einer Abhängigkeit nötig wären. Schauen wir uns wieder Abbildung ? an und legen dabei fest, dass die Heizung Zugriff auf die Dienstleistung mit der Nummer 7 vom Thermostat benötigt. Dies würde bedeuten, dass die Konfiguration {{1,2},{5,7} } gültig und die Konfiguration {{1,2},{6,8}} ungültig ist. Alle ungültigen Konfigurationen können ignoriert werden, wodurch die Performance des Algorithmus wieder gesteigert werden kann. Nachem bereits dominierte Updates entfernt wurden, werden im nächsten Schritt dominierte Updatekonfigurationen herausgefiltert, um so wieder die Performance des Algorithmus zu steigern. Eine Updatekonfiguration ist dominiert, wenn eine weitere Konfiguration existiert, in der alle Updates entweder gleich alt oder neuer sind. In Abbildung ? ist der Auschnitt eines Updatekonfiurationsgraphen von drei Geräten mit jeweils 3 Updates zu sehen, wobei die Updates nach ihrer Aktualtiät von links nach rechts sortiert sind. Bei Gerät1 also ist das Update {{1,3} aktueller als Update {{1,2} und das Update {{1,4} ist aktueller als Update {{1,3}. Die drei Zustände stehen für je eine Updatekonfiguration, die Pfeile sollen andeuten, dass es möglich ist von einer Updatekonfiguration auf eine andere upzudaten. Bei genauerer Betrachtung der einzelnen Updatekonfigurationen ist zu erkennen, dass die grüne Konfiguration von der blauen dominiert wird, von der roten jedoch nicht. In der blauen Konfiguration besitzen nämlich alle Geräte eine aktuellere Version, in der roten Konfiguration hingegen ist die Version

des zweiten Geräts eine ältere ({{6,7} ist älter als {{6,8}). Aufmerksamen Lesern wird außerdem auffallen, dass die rote Konfiguration ebenfalls von der blauen dominiert wird, denn auch hier bestzen alle Geräte eine aktuellere Version.

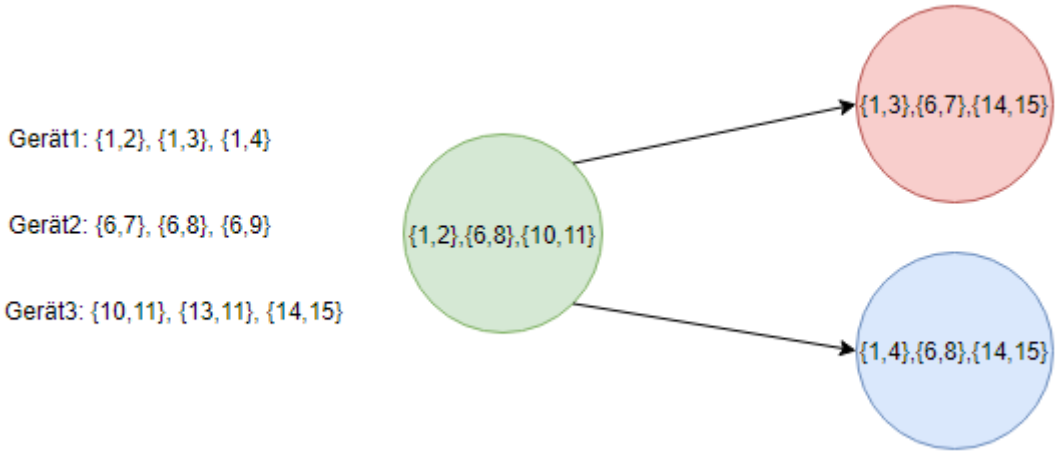

Figure 3.5: Dominiere Updatekonfigurationen

Durch das Filtern dominierter Updatekonfiguration wird die Anzahl potenzieller Konfigurationen signifikant verringert, wodurch es für den Nutzer einfacher wird sich für eine passende Konfiguration zu entscheiden. Um diese Entscheidung noch zusätzlich zu vereinfachen werden im folgenden Schritt Subnetzwerke erstellt, für die unabhängig von anderen Subnetzwerken eine Updatekonfiguration ausgewählt werden kann. Es ist nicht unüblich, dass in einem Smart Home System Gruppen vorhanden sind die unabhängig von anderen Gruppen kommunizieren. Mögliche Gruppen zwischen denen keine Kommunikation notwendig ist, können zum Beispiel Fernseher+Soundboxen und Backofen+(Mikrowelle?) sein. Hier bietet es sich logischerweiserweise an die möglichen Updatekonfigurationen unabhängig voneinander zu betrachten, da so für den Nutzer ein einfacherer Überblick verschafft werden kann. Ein Subnetzwerk besteht also aus allen Geräten, die durch eine Abhängigkeit in Verbindung stehen. Diese Verbindung muss keine direkte sein. Seien zum Beispiel zwei Geräte A und B durch eine Abhängigkeit verbunden und gleichzeitig sei Gerät A zusätzlich mit einem Gerät C aufgrund einer Abhängigkeit verbunden, dann sind die Geräte B und C ebenfalls auf indirektem Weg miteinander verbunden, obwohl keine direkte Abhängigkeit zwischen ihnen herrscht. Somit sind alle drei Geräte A, B und C Teil eines Subnetzwerkes. Der letzte Schritt besteht darin Bewertungen für die übrig gebliebenen Updatekonfigurationen zu erstellen. Zu diesem Zweck werden den einzelnen Dienstleistungen ein Wert zugeschrieben. Unter einem preskirptiven Standard wäre es theoretisch für Hersteller möglich Daten zu sammeln und so zu ermitteln, welche Dienstleistungen am häufigsten verwendet werden. Ein Standard

TODO: Plagiat!

könnte dann zusätzlich zur Definition einer Dienstleistung eine Wertung abgeben, je nach dem wie häufig diese Dienstleistung im Gegensatz zu anderen verwendet wird. Nutzerstatistiken allein reichen nicht aus um eine valide Wertung abzugeben, da Dienstleistungen existieren, welche trotz seltener Nutzung extrem wichtig sein können. Zum Beispiel sollte der Alarm eines Feuermelders nicht aufgrund der seltenen Nutzung dieser Dienstleistung, als weniger wichtig bewertet werden. Daher bedarf es weiterer Metriken, um eine valide Aussage über den Wert einer Dienstleistungen treffen zu können.

3.3 Implementierter Code

Die mitgegebene Implementation des beschriebenen Codes besteht aus 4 Klassen:

- Device.java: Klasse zum Erstellen der Geräte als Objekte.
- Generator.java: Generator zum erstellen verschiedener Smart Home Systeme
- SmartHome.java: Klasse in der das Smart Home System gespeichert wird und an der der Algorithmus ausgeführt wird.
- Algorithm.java: Die Implementierung des Algorithmus wie er beschrieben wurde.

Die Klasse Device beinhaltet einen Konstruktor zum Erstellen der Geräte. Der Generator nutzt diesen Konstruktor und erstellt mit der Funktion createDevice Geräte. Zwischen den Geräten werden anschließend mit der Funktion createDependencies Abhängigkeiten erzeugt werden. Beim erstellen der Geräte und der Abhängigkeiten wird auf auf die Klasse Random aus dem "java.util" package zurückgegriffen, um den Geräten zufällige Dienstleistungen und Abhängigkeiten zuzuweisen. Die Anzahl der Geräte und der Abhängigkeiten hängt von manuell ausgewählten Parametern in der Klasse SmartHome ab.

```
public class SmartHome extends GeneratorFinal{
...
    static int nrOfDevices = 10;
    static int nrOfUpdatesPerDevice = 3;
    static int nrOfServicesPerDevice = 10;
    static int nrOfDependencies = nrOfDevices*2;
...
}
```

Die Klasse Algorithmus beinhaltet den eingentlich Code des beschriebenen Algorithmus. Im ersten Schritt werden mit der Methode isDominated alle dominierten Updates entfernt und die übrig gebliebenen Updates in die Funktion cartesianProduct als Parameter eingefügt, um so den Updatekonfigurationsgraphen zu erstellen. Danach werden mit der Funktion deleteBreakingConfigurations alle ungültigen Konfigurationen aus dem Updatekonfigurationsgraphen entfernt. Dafür erstellt die Funktion für jede Konfiguration

eine Arrayliste mit allen in dieser Konfiguration vorkommenden Dienstleistungen und überprüft dann, ob alle notwendigen Dienstleistungen in der List vorkommen. Wenn dies nicht der Fall ist wird die Konfiguration markiert und anschließend aus dem Updatekonfigurationsgraphen entfernt. Als nächstes entfernt die Funktion paretoOptimal allle dominiert Updatekonfigurationen wie in Kapitel 3.2.2 beschrieben. Die Funktionen createSubnetworks und floodFill erstelllen anschließend Subnetzwerke basierend auf den Abhängigkeiten zwischen den Geräten. Ein Subnetzwerk besteht dann nur noch aus Geräten zwischen denen direkt oder indirekt eine Abhängigkeit besteht. Im letzten Schritt erstellt die Funktion rating für jede Dienstleistung eine Wertung. Die Werte der einzelnen Dienstleistungen werden dann pro Subnetzwerk summiert, um sie vergleichen zu können.

Chapter 4

Evaluation

Im folgenden Kapitel wird die Implementation hinsichtlich ihrer Performance untersucht, wobei der Algorithmus an unterschiedlichen Systemen getestet wird, die mit Hilfe des Generators erstellt werden.

4.1 Theoretische Performance

Ein Ziel dieser Evaluation ist es herauszufinden, ob der Algorithmus an aktuellen Smart Home Systemen eine akzeptable Laufzeit vorweist. Bereits ohne Messungen vorgenommen zu haben, können bei genauerer Betrachtung des Algorithmus Vermutungen zur Komplexität einzelner Schritte aufgestellt werden. Zum Berechnen des Updatekonfigurationsgraphen muss das kartesische Produkt zwischen allen Geräten und den dazugehörigen Updates gebildet werden. Sei ein System mit n Geräten gegeben und pro Gerät existieren s Updates, dann entstehen beim Erstellen des kartesischen Produkts s^n mögliche Konfigurationen. Zur Vereinfachung wird davon ausgegangen, dass alle Geräte in einem System die gleiche Anzahl an Updates besitzten. Die Komplexität dieses Schrittes wäre also vereinfacht O(s^n). Für ein "reales" Smart Home System, in dem die Geräte unterschiedlich viele Updates besitzen, läge die Komplexität bei O(|Gerät1|*|Gerät2|*|Gerät3|*...). Aufgrund dieses exponentiellen Wachstums ist zu vermuten, dass der Rechenaufwand mit jedem weiteren Gerät und jedem Update stark ansteigt. Bei einem Smart Home System mit 10 Geräten und jeweils 5 Updates pro Gerät entstehen ohne weitere Optimierungen bereits $5^(10) = 9765625$ verschiedene Konfigurationen. Sollte der Trend sich fortsetzen, dass die durchschnittliche Anzahl an Geräten pro Smart Home System weiterhin ansteigt, könnte es notwendig sein zusätzliche Optimierungsschritte zu entwickeln, um so den Rechenaufwand zu verringern. Im Idealfall ergeben die Messungen, dass der Rechenaufwand des Algorithmus so gering ist, dass die Laufzeit auch bei großen Systemen mit zum Beispiel 30 Geräten vertretbar ist. So kann die Zukunftstauglichkeit des Alogrithmus bei immer größer werdenden Smart Home Systemen gewährleistet werden. Ein weiterer rechenintensiver Schritt ist das Löschen aller ungültigen Updatekonfigurationen. Zu diesem Zweck muss für jede potenzielle Updatekonfiguration jeweils

eine Liste mit allen angebotenen Dienstleistungen erstellt werden, die dann für weitere Berechnungen des Algorithmus genutzt werden. Für die aus dem vorigen Beispiel $5^(10) = 9765625$ Konfigurationen müssten also 9765625 Listen erstellt werden, die alle einzeln durchgegangen werden, um so ungültige Updatekonfigurationen zu finden. Auch für diesen Teilschritt ist somit eine hoher Rechenaufwand zu erwarten. Während andere Berechnungsschritte ebenfalls zu Laufzeit des Algorithmus beitragen, ist davon auszugehen, dass sie um einen großen Faktor weniger Rechenintensiv sind und dementsprechend weniger ins Gewicht fallen. Die in diesem Abschnitt aufgestellten Hypothesen werden in den folgenden Abschnitten überprüft.

4.2 Messungen

In diesem Kapitel werden mit Hilfe eines Generators verschiedene Smart Home Systeme kreiert, an denen anschließend die Laufzeit des Algorithmus gemessen wird. Die Laufzeit des Algorithmus wird stark von der Größe der erstellten Smart Home Systemen abhängig sein. Es stehen folgende Parameter zur Verfügung, um die Größe eines Smart Home Systems festzulegen:

- #Geräte: Legt die Anzahl der Geräte im System fest
- #DienstleistungenProGerät: Legt die Anzahl der Dienstleistungen fest, die ein Gerät anbietet.
- #UpdatesProGerät: Legt die Anzahl der Updates pro Gerät fest (Aus Implementationsgründen besitzen alle Geräte die gleiche Anzahl an möglichen Updates)
- #Abhängigkeiten: Legt die Anzahl der Abhängigkeiten im System fest.

Als erstes soll die Frage beantwortet werden, ob der Algorithmus an Systemen mit aktuell durchschnittlicher Größe anwendbar ist. Zu diesem Zweck wird mit Hilfe des Generators ein durchschnittliches Smart Home System generiert. Zunächst muss jedoch geklärt werden wie ein solches Durchschnittssystem aussieht. Laut Statista[8] waren in amerikanischen Haushälten im Jahr 2020 durchschnittlich 10 Geräte miteinander verbunden. Die durchschnittliche Anzahl an Updates pro Gerät zu ermitteln, ohne empirische Daten vorliegen zu haben, stellt sich als schwierig dar. Um diesen Parameter nicht willkürlich zu würfeln, wird sich an der Anzahl der Updates von Apples Iphones orientiert. Seit 2013 erhalten Iphones durchschnittlich 10 Updates pro Jahr[14]. Dabei muss beachtet werden, dass sich die Upates über das Jahr verteilen und somit kein Gerät von einen auf den anderen Tag 10 neue Updates erhält. Dennoch kann man nicht davon ausgehen, dass Nutzer täglich überprüfen, ob ein neues Update zur Verfügung steht, so dass sie sich anhäufen. Um täglich manuelle Konfiguration zu vermeiden, wird davon ausgegangen, dass ein Nutzer regelmäßig alle 3 Monate, also ingesamt 4 mal im Jahr, seine Geräte updatet. So sollte ein Gerät durchschnittlich zwischen 2 und 3 (10 Updates

pro Jahr / 4 mal updaten pro Jahr = 2.5) möglichen Updates besitzen. Nun stellt sich noch die Frage wie viele Dienstleistungen ein Gerät durchschnittlich anbietet und wie viele Abhängigkeiten zwischen den Geräten herrschen. Auch dies ist ohne empirische Daten schwierig zu beurteilen. Bei Smart Home Geräten handelt es sich meist um simple Geräte, die nur wenigen Zwecken dienen. So kann zum Beispiel ein Thermostat nur die Temperatur messen, eine Heizung heizen, eine Sicherheitskamera aufnehmen und vielleicht einen Alarm auslösen. Die meisten Geräte bieten also nur wenige Dienstleistungen. Diese Annahme beruht jedoch auf eigenen Erfahrungen und könnte somit zu gering geschätzt sein. Daher wird für das Erstellen der Geräte davon ausgegangen, dass sie durchschnittlich 5 Dienstleistungen anbieten, um so die Größe eines Systems nicht zu unterschätzen. Aufgrund der geringen Anzahl an Dienstleistungen pro Gerät gehen wir ebenfalls davon aus, dass die Anzahl der Abhängigkeiten, die ein Gerät besitzt ebenfalls gering ist. Viele Geräte funktionieren auch vollkommen selbstständig besitzen und beseitzen keinerlei Abhängigkeit. Für die Anzahl der Abhängigkeiten wird daher festgelegt, dass ein Gerät durchschnittlich 2 Abhängigkeiten besitzt.

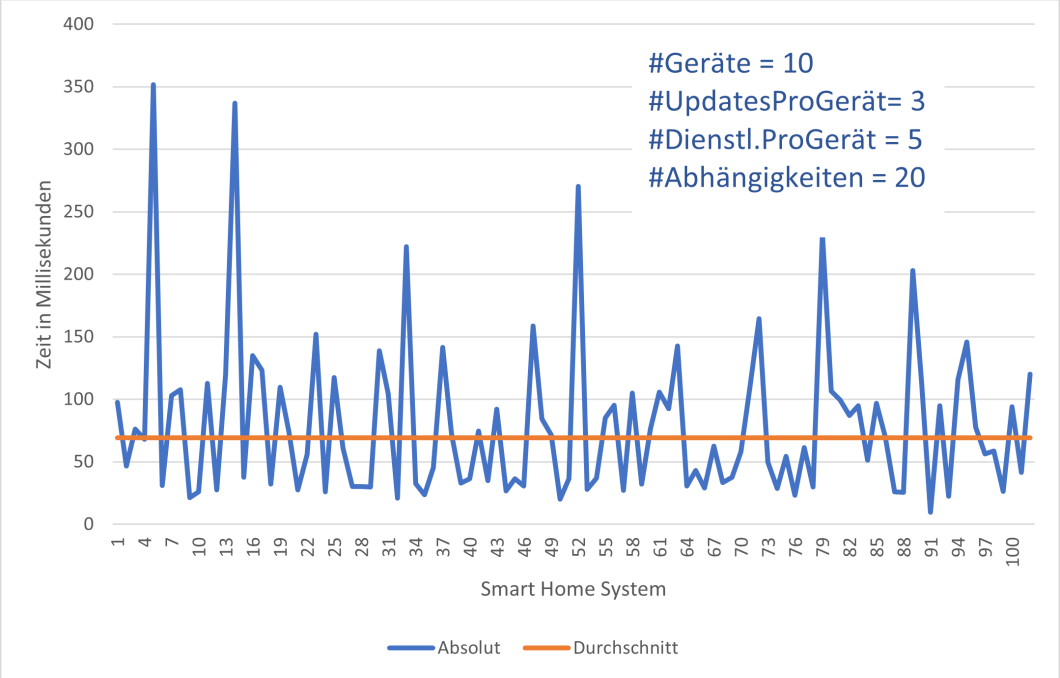

Figure 4.1: Laufzeitmessungen an einem durchschnittlichen System

Anhand der Abbildung 4.1 ist zu erkennen, dass die Berechnungen im Bereich von Milisekunden durchlaufen sind. Die durschnittliche Laufzeit des Algorithmus liegt bei 79 Millisekunden mit Ausreißern bei 350 und 21 Millisekunden. In einem Buch zum

Thema Usability Engineering hat Jakob Nielsen verschiedene Grenzwerte zur Reaktionszeit von Webseiten festgelegt. Diese Werte können als Indikator genutzt werden, um die Laufzeit des Algorithmus zu beurteilen. Sie können nicht 1 zu 1 auf die Laufzeit des Algorithmus übertragen werden, da der Alogrithmus in einem anderen Kontext als eine Webseiten zu betrachten ist. Während Webseiten täglich von Nutzern besucht werden, wird der Algorithmus nur einige male im Jahr verwendet werden. Dementsprechend ist davon auszugehen, dass die Laufzeiten für Nutzer zumutbar sind, solange sie unterhalb den folgenden Grenzen liegen:

- 0.1 Sekunden: Prozesse unter 0.1 Sekunden nimmt ein Nutzer als ohne Verzögerung wahr.
- 1.0 Sekunden: Der Nutzer nimmt eine kurze Verzögerung wahr, die meist als nicht störend interpretiert wird.
- 10 Sekunden: Für bis zu 10 Sekunden wartet ein Nutzer durchschnittlich auf eine Reaktionen. Verzögert sich eine Reaktion um mehr als 10 Sekunden wechselt ein Nutzer zu anderen Dingen bis eine Reaktion erscheint.

Des Weiteren fällt auf, dass der Algorithmus bei gleich bleibenden Parametern starke Schwankungen aufweist. Dies liegt daran, dass trotz gleich bleibender Parameter die Komplexität eines Smart Homes variieren kann. Überträgt man dies auf die reale Welt ist dies einfacher zu verstehen, da es logisch erscheint, dass zwei Systeme trotz gleicher Größe nicht automatisch gleich komplex sind. In einem System können viele Updates dominiert sein, während dies im anderen System nicht der Fall ist. Dies führt dazu, dass die Optimierungen an beiden Systemen unterschiedlich erfolgreich ausfallen, so dass in einem System mehr Berechnungen durchgeführt werden müssen.

4.2.1 Laufzeit der Teilschritte

In diesem Abschnitt werden die Lauftzeiten der einzelnen Teilschritte untersucht, um so Problemstellen in der Implementation ausfindig zu machen. Abbildung 4.2 zeigt die Laufzeit der einzelnen Schritte, die in Kapitel 3 beschrieben wurden. Die Messungen wurden wie im vorigen Kapitel an einem durchschnitllichen Smart Home System durchgeführt. Die genauen Parameter sind der Abbildung zu entnehmen. Es ist zu erkennen, dass der Schritt "deleteBreakingConigurations" den Großteil der Laufzeit ausmacht. Die Laufzeit aller anderen Schritte summiert sich auf nur 34 Prozent der gesamten Laufzeit aus. Die Hypothesen aus Abschnitt 4.1 haben sich teilweise bewahrheitet. Das Erstellen des Updatekonfigurationsgraphen nimmt nicht so viel Zeit in Anspruch wie angenommen. Um so mehr stimmt, dass das Löschen der ungültigen Konfigurationen sehr rechenintensiv im Vergleich zu den anderen Schritten ist. Möchte man den Alogrithmus optimieren, dann sollte diesem Schritt besondere Aufmerksamkeit geschenkt werden. Die anderen Teilschritte fallen wie erwartet nicht nicht ins Gewicht und können dementsprechend vernachlässigt werden.

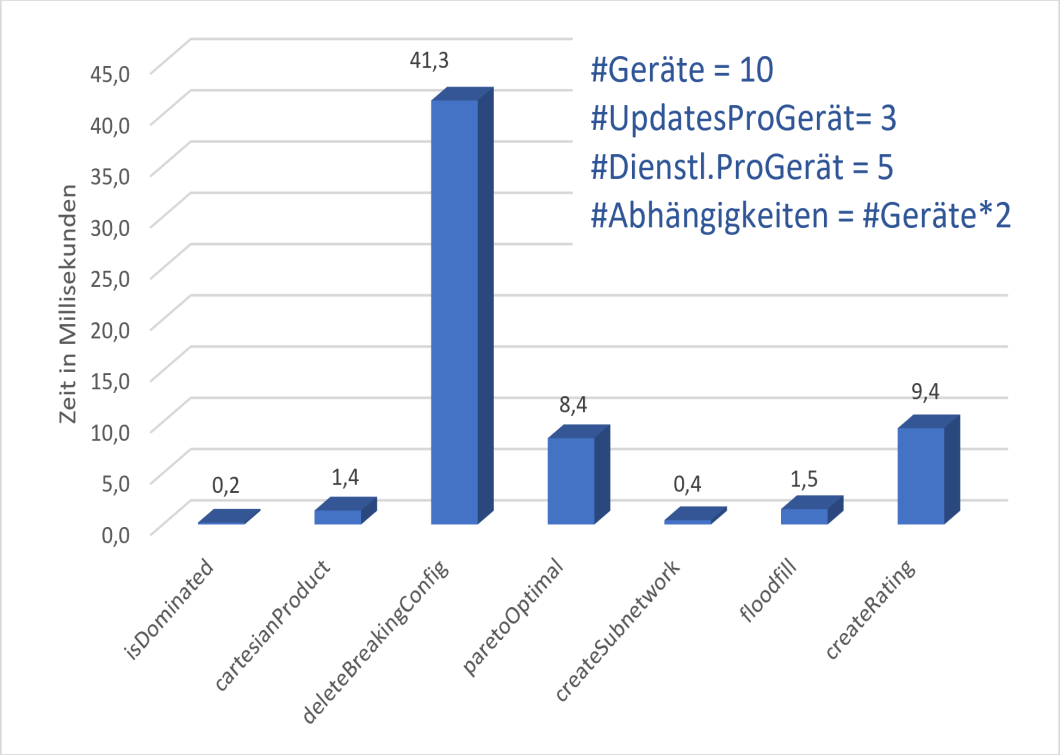

Figure 4.2: Laufzeit einzelner Teilschritte

4.2.2 Einfluss einzelner Parameter

Im weiteren Verlauf werden die Parameter des Smart Home Generators verändert und die damit verbundenen Laufzeitschwankungen gemessen. So kann ermittelt werden welchen Einfluss die einzelnen Parameter auf die Laufzeit haben. Der erste Parameter, der unabhängig von den anderen untersucht wird, ist die Anzahl der Geräte in einem Smart Home System. Dafür wird die die Entwicklung der Laufzeit bei steigender Anzahl an Geräten im System gemessen. Um den Einfluss der anderen Parameter möglichst gering zu halten, sind sie, wie in Abbildung 4.3 zu sehen, eingestellt. In der Abbildung ist zu erkennen, das mit steigender Anzahl an Geräten die Laufzeit des Algorithmus ansteigt. Im Bereich von 1-8 Geräten hat das Hinzufügen eines Geräte eine durchschnittliche Laufzeitsteigerung von circa 15 Prozent zur Folge. Ab 9-13 Geräten ergeben die Messungen beim Hinzufügen eines Geräts eine Laufzeitsteigerung von circa 30 Prozent. Ab 14 Geräten erhöht sich sich die Laufzeit mit jedem hinzugefügten Gerät um jeweils circa 60 Prozent. Die Ergebnisse zeigen, dass das Hinzufügen eines Geräts bei kleinen Systemen (unter 13 Geräten) geringe Laufzeiterhöhungen zur Folge haben. Je größer ein System ist, desto stärker erhöht sich die sich die Laufzeit mit jedem zusätzlich hinzugefügtem Gerät. Die Messungen wurden für bis zu 20 Geräte in einem System durchgeführt. Aufgrund der Kurve des Graphen ist zu erwarten, dass der Graph bei noch größeren Systemen immer stärker steigen würde.

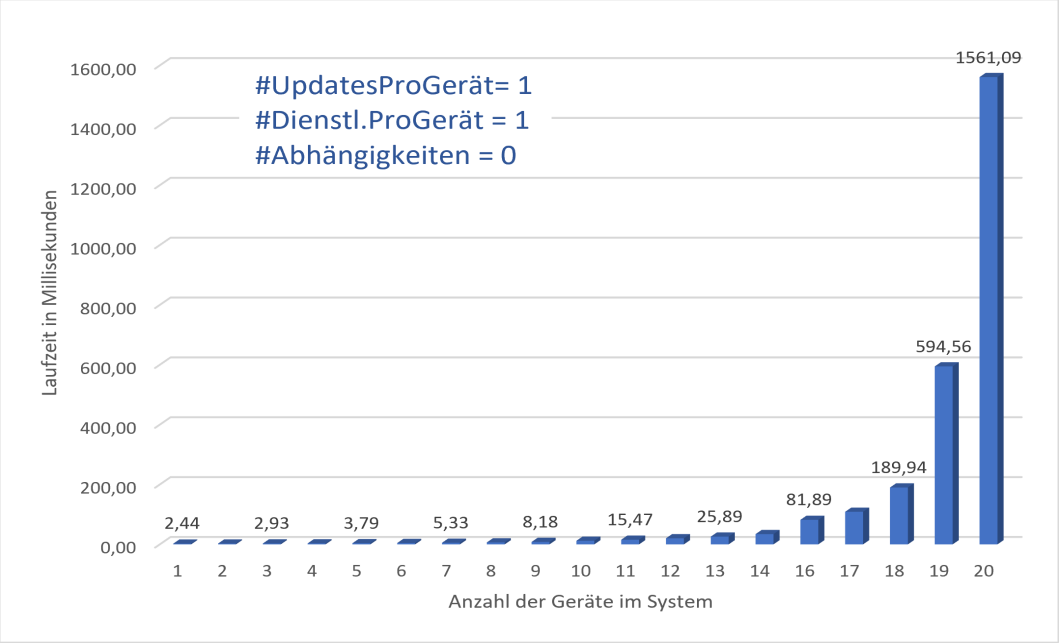

Figure 4.3: Laufzeit beim Erhöhen der Anzahl der Geräte im System

In Abbildung 4.3 handelt es sich um theoretische Ergebnisse, um den unabhängigen Einfluss des Steigerns eines einzelnen Parameters zu messen. In einem realistischem Smart Home System ist davon auszugehen, dass mit steigender Anzahl an Geräten auch die Anzahl der Abhängigkeiten im System steigt. Außerderm liegt die Anzahl der Updates pro Gerät in einem realistischem Smart Home System, wie bereits erwähnt zwischen 2 und 3. Um also praxisnähere Ergebnisse zu ermitteln, zeigen die Messungen in Abbildung 4.4 die Laufzeit des Algorithmus an realistischeren Systemen. Es ist zu erkennen, dass das Erhöhen der Anzahl der Geräte einen Ähnlichen Effekt hat wie zuvor . Der Unterschied besteht darin, dass eine starke Laufzeitsteigerung bereits bei weniger Geräten beginnt und noch stärker ansteigt. Aufgrund des starken Anstiegs war es nicht möglich die Anzahl der Geräte wie in Abbildung 4.3 auf bis zu 20 zu steigern. Die durchschnittliche Laufzeiterhöhung pro hinzugefügtem liegt bei 1-6 Geräten bei circa 30 Prozent (doppelt so viel wie in den Messungen zuvor). Ab 7-13 Geräten liegt die Steigerung bei circa 80 Prozent und bei der Erhöhung von 13 auf 14 Geräten hat sich die Laufzeit um 440 Prozent erhöht. Somit kann man zusammenfassen, dass die Laufzeit des Algorithmus bei realistischen Systemen sehr stark ansteigt, wenn die Anzahl der Geräte in einem System steigt.

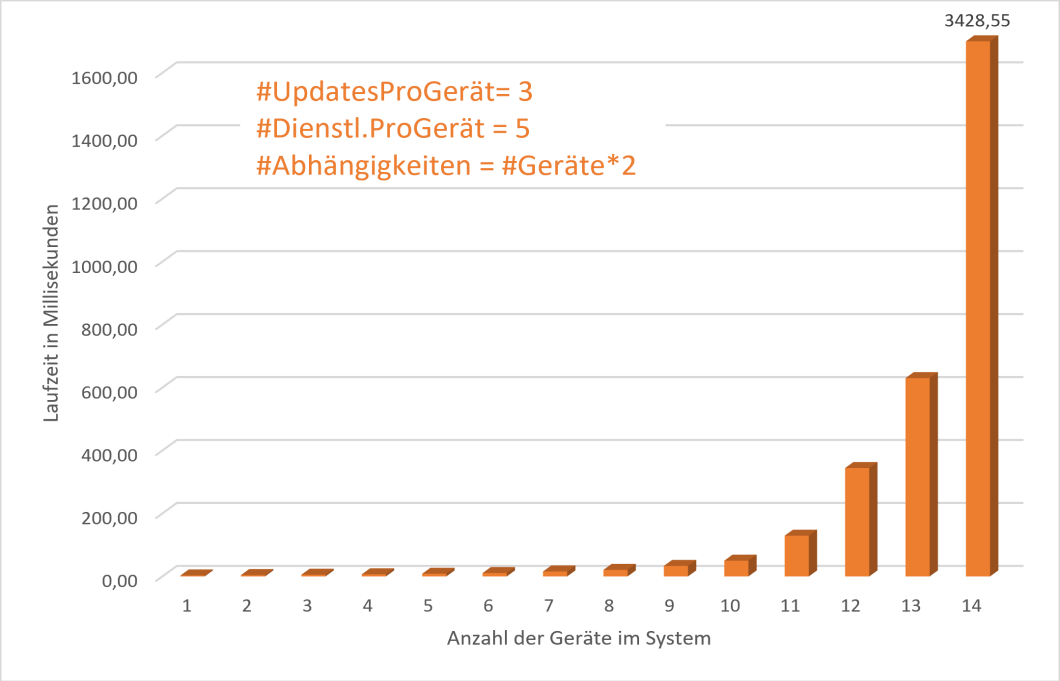

Figure 4.4: Laufzeit beim Erhöhen der Anzahl der Geräte in einem realistischem System

Nun folgen weitere Messungen an Systemen bei denen die Anzahl der Updates pro Gerät erhöht werden. Es wird wieder versucht den Einfluss der anderen Parameter auf die Laufzeit möglichst gering zu halten. Die genauen Einstellungen sind der Abbildung 4.5 zu entnehmen. Die Abbildung zeigt, dass bei der Verdopplung der Updates pro Gerät die Laufzeit sich immer um durchschnittlich 60 Prozent erhöht. Runtergerechent bedeutet dies, dass jedes Update die Laufzeit um durchschnittlich 6 Prozent erhöht. Es handelt sich wieder um ein exponentielles Wachstum, was im Vergleich zu Abbildung 4.3 jedoch langsamer ansteigt. Die Messungen zeigen insgesamt, dass das Verdoppeln der Geräte einen stärkeren Einfluss auf die Laufzeit hat, als das Verdoppeln der Updates im System. Dies könnte daran liegen das beim Bilden des kartesischen Produkts, wie in Kapitel 4.1 erwähnt, die Anzahl der Geräte im Exponenten steht und die Anzahl der Updates in der Basis. Ein Gerät hinzuzufügen lässt den Updatekonfigurationsgraphen dementsprechend schneller wachsen, als das Hinzufügen eines Updates.

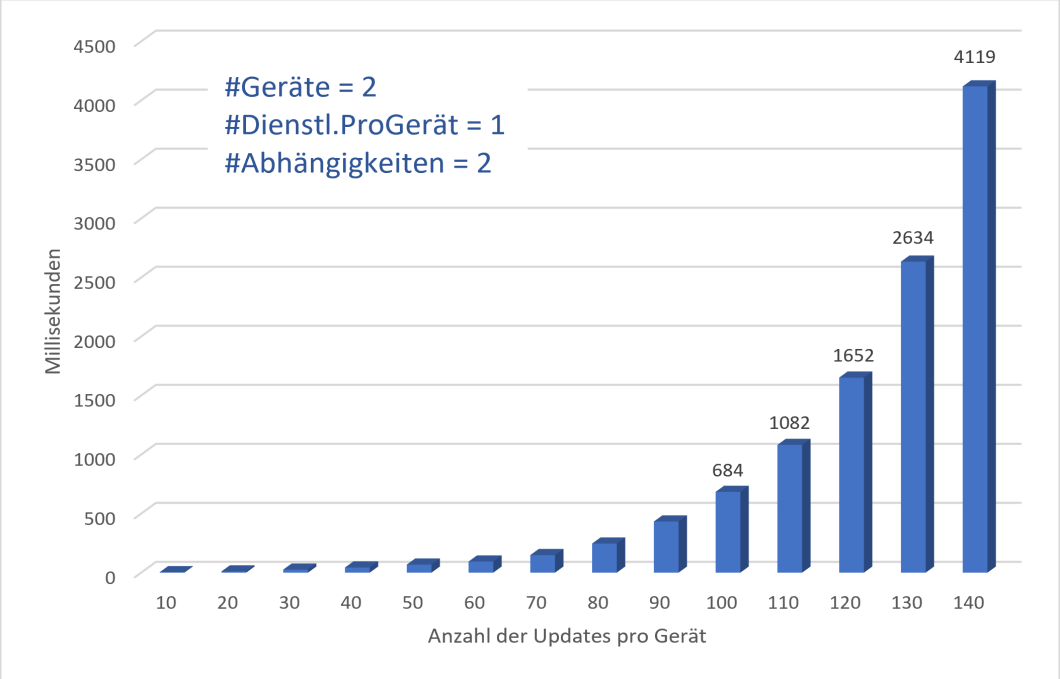

Figure 4.5: Laufzeit beim Erhöhen der Anzahl der Updates pro Gerät im System

In Abbildung 4.5 wurden die Messungen wieder an einem unrealistischen System vorgenommen, um den Einfluss des Steigerns eines einzelnen Parameters zu messen. Also werden Messungen an einem realen System durchgeführt. Aufgrund der Implementation des Generators besitzen alle Geräte die gleiche Anzahl an Updates. Dementsprechend ist es nicht möglich einem System mit 10 Geräten 1 Update hinzuzufügen. Für zukünftige Messungen wäre es von Vorteil den Generator entsprechend zu ändern, so dass dies möglich ist. Es ist also nicht möglich die Anzahl der Updates um jeweils 1 zu erhöhen, sondern für alle Geräte wird jeweils 1 Update hinzugefügt. Dementsprechend ist zu sehen, dass die Laufzeit an einem System mit 10 Geräten extrem ansteigt im Gegensatz zu einem System mit 5 Geräten. Bei beiden Konfiguration ist ein exponentielles Wachstum zu erkennen.

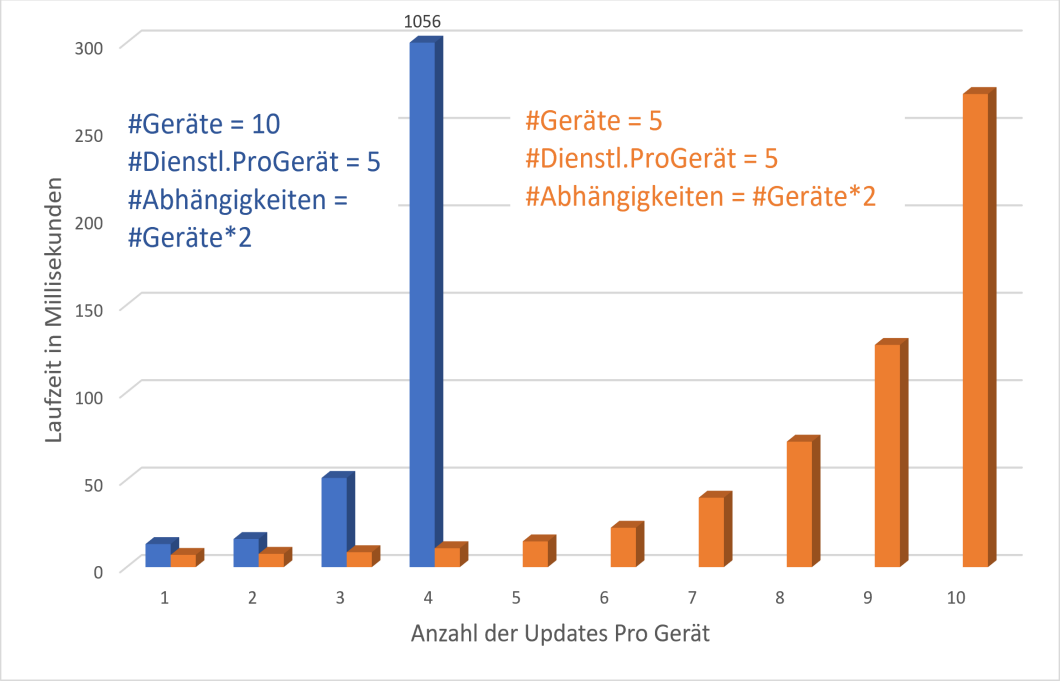

Figure 4.6: Laufzeit beim Erhöhen der Anzahl der Updates pro Gerät in einem realistischen System

4.2.3 Einfluss der Parameter im Zusammenspiel

Als Letztes wird gemessen welchen Einluss die beiden Parameter #Geräte und #UpdateProGerät auf die Laufzeit haben, wenn man sie gleichzeitig erhöht. In Abbildung 4.7 zeigen die Messungen, dass die Laufzeit ab 7 Geräten mit je 7 Updates pro Gerät extrem ansteigt. Die Algorithmus weist als auch hier ein sehr starkes wachstum ab einer bestimmten Grenze vor.

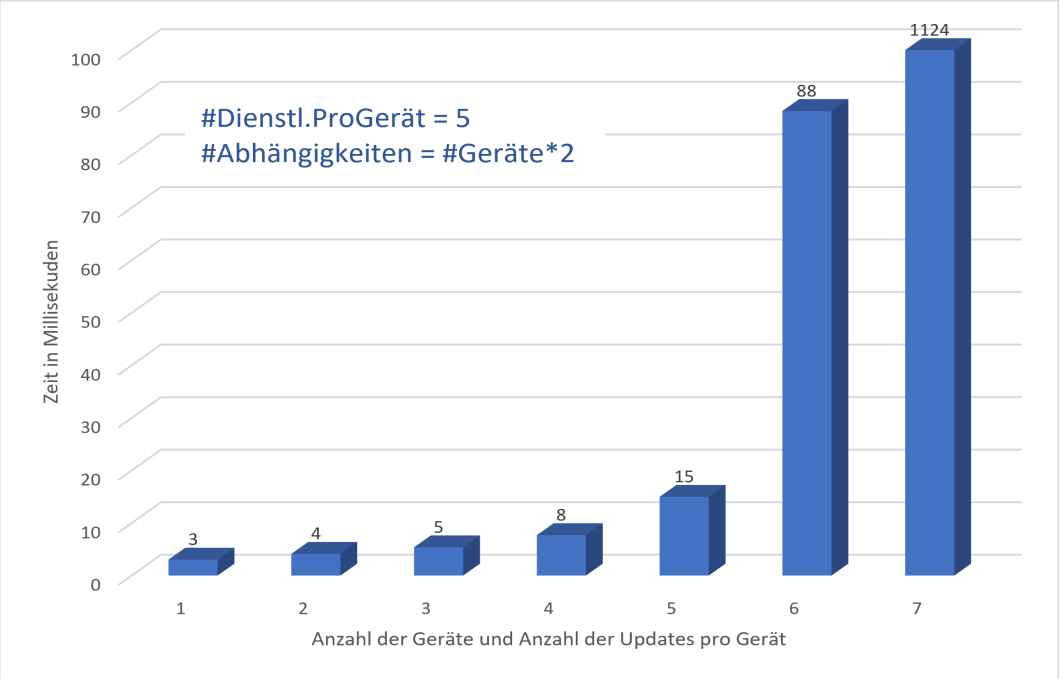

Figure 4.7: Caption

4.3 Specs

Die Laufzeit des Alogrithmus ist von vielen Faktoren abhängig. Um die Ergebnisse besser beurteilen zu können folgt eine Liste der verwendeteten PC Konfiguration:

- Betriebssystem: Windows 10 Home
- CPU: AMD Ryzen 3 3200g (4 Kerne, Taktfrequenz 3.6 GHz)
- RAM: 16 GB DDR4 3000 MHz
- Speichermedium: SANDISK Ultra 3D NVMe SSD (Lesen: 2400 MB/s, Schreiben: 1750 MB/s)

Moderne Smartphone besitzen leistungsstarke Prozessoren. Das meist verkaufte Smartphone im Jahr 2021 ist das Apple Iphone 12 mit einem[15] A14 Apple Bionic Prozessor und 4 GB LPDDR4X (4266 MHz) RAM. Der Prozessor besitzt 2 Hochleistungskerne mit Taktfrequenzen bis zu 3 GHz und 4 weiteren Kernen mit Taktfrequenzen bis zu 1,82 GHz [16]. Die Komponenten des Iphone 12 sind nicht wesentlich Leistungsschwacher als die des verwendeteten Computers. Der Algorithmus sollte dementsprechend auf einem modernen Smartphone ebenfalls zügig durchlaufen sein. Verbesserungen der Implementation bezüglich Nebenläufigkeit würden die Performance zusätzlich steigern, so dass der Algorithmus auch auf leistungsschwächeren Smartphones funktionieren sollte.

4.4 Ergebnisse

Die Evaluationsschritte zeigen, dass der Algorithmus zum aktuellen Stand der Smart Home Systeme für viele Menschen ein wichtiges Produkt sein könnte. Nutzern wäre es möglich mit einer simplen Anwendung auf dem Handy die Updates eines Smart Home System zu steuern. Die Ergebnisse sind jedoch mit Vorsicht zu betrachten, da die Eintstellungen der Parameter auf vielen Annahmen beruhen für die es noch keine ausreichende Forschungsgrundlage gibt. Für zukünftige Messungen wäre es von Vorteil empirische Daten über Smart Home Systeme zu sammeln, um so mit Hilfe des Generators möglichst realitätsnahe Systeme zu kreieren. Die Evaluation hat auch gezeigt, dass die Zukunftstauglichkeit des Algorithmus nicht garantiert ist und weitere Optimierungen vermutlich notwendig sein werden, um vertretbare Laufzeiten bei immer größer werdenen Smart Home Systemen zu erreichen. Durch die modulare Implementierung sollte es möglich Änderungen am Alogrithmus vorzunehmen und ihn so zu verbessern. Ein potenzieller Ansatz den man verfolgen könnte ist die Reihenfolge des Alogrithmus zu ändern. So könnte man im ersten Schritt nach Subnetzwerken suchen und den Rest des Algorithmus für jedes Subnetzwerk unabhängig von einander laufen lassen. Dadurch müsste der Algorthmus öfter ausgeführt werden, jedoch ist zu vermuten, dass die Ausführungszeit auf vielen kleinen Subnetzwerken geringer sein wird, als auf einem großen System.

Chapter 5

Conclusion

Der Fokus dieser Arbeit liegt auf der Implementation und Evaluation eines Algorithmus für Abhängigkeitsbewahrende Updatekonfigurationen in Smart Home Systemen. Dieser Algorithmus ist in der Theorie ein Werkzeug, das vielen Smart Home Nutzern mehr Kontrolle und Sicherheit über ihr System bieten würde. Die Ergebnisse der Evaluation haben gezeigt, dass das Design des Algorithmus erfolgreich in einer Implementation umgesetzt wurde. Steigt die Größe der Smart Home Systeme jedoch an, kann es zu Problemen bezüglich der Laufzeit und des Speicherbedarfs kommen. Dementsprechend sind weitere Optimierungen bezüglich des Designs und der Implementation nötig, um die zukunftsfähigkeit des Algorithmus gewährleisten zu können. Insgesamt liegt das Ziel dieser Arbeit darin die Langlebigkeit von Smart Home Systemen gewährleisten zu können. Es gibt viele weitere Aspekte, die einen Einfluss auf die Langlebigkeit eines Smart Home Systems haben. Einige dieser Aspekte wurden in dieser Arbeit angerissen, jedoch nicht genauer thematisiert. In Folgearbeiten könnten diese Aspekt genauer untersucht werden. Des Weiteren muss angemerkt werden, dass die Ergebnisse der Evaluation unter Vorbehalt zu betrachten sind, da es sich bei den Messungen lediglich um theoretische Ergebnisse handelt, die sehr von der Implementation des Generators abhängig sind. Dementsprechend wäre es für zukünftige Arbeiten von Vorteil empirische Daten zu sammeln, um den Generator so konfigurieren zu können, so dass die generierten Systeme möglichst realistisch sind. Um noch genauere Ergebnisse zu erzielen wäre es nötig den Algorithmus an realen Smart Home Systemen zu testen. Dafür muss von Seiten der Hersteller ein Standard eingeführt werden, um einen einheitlichen Zugriff auf die Informationen eines Smart Home Systemen zu haben. Ein Vorschlag diesbezüglich ist in Kapitel 2.1.2 zu finden. Ohne einen solchen Standard wäre es nicht möglich den Algorithmus Plattformunabhängig zu realisieren. Außerdem wurden viele Detailaspekte in dieser Arbeit außer Acht gelassen. So wurde nicht festgelegt,

Bibliography

[1] Peter Zdankin, Matthias Schaffeld, Marian Waltereit, Oskar Carl, Torben Weis, "An Algorithm for Dependency-Preserving Smart Home Updates" (2021). https://ieeexplore.ieee.org/document/9431040

[2] Global Smart Home Market Size By Technologies (Cellular Network Technologies, Protocols And Standards), By Product (Lighting Control, Security And Access Control, HVAC Control), By Geographic Scope And Forecast https://www.verifiedmarketresearch.com/product/global-smart-home-market-size-and-forecast-to-2025/ (Visited on 23.08.2021)

[3] https://de.wikipedia.org/wiki/Smart_Home (visited on 18.08.2021)

[4] Expertenbeitrag von Amanuel Dag, Country Director bei CONTEXT https://www.homeandsmart.de/smart-home-status-quo-2019 (visited on 18.08.2021)

[5] https://policyadvice.net/insurance/insights/smart-home-statistics/) (visited on 18.08.2021)

[6] Matthias Kersken, Herbert Sinnesbichler, Hans Erhorn, "Analyse der Einsparpotenziale durch Smarthome- und intelligente Heizungsregelungen" (Oktober 2018) https://www.ibp.fraunhofer.de/content/dam/ibp/ibp-neu/de/dokumente/sonderdrucke/bauphysik-gertis/6-einsparpotenziale-intelligente-heizungsregelung.pdf (visited on 03.09.2021)

[7] Peter Zdankin, Matthias Schaffeld, Marian Waltereit, Oskar Carl, Torben Weis, "An Algorithm for Dependency-Preserving Smart Home Updates" (2020), 2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops) https://ieeexplore.ieee.org/abstract/document/9156165

[8] Peter Zdankin, Matthias Schaffeld, Marian Waltereit, Oskar Carl, Torben Weis, "Requirements and Mechanisms for Smart Home Updates" (october 2020) https://www.researchgate.net/publication/344441146_Requirements_and_Mechanisms_for_Smart_Home_Updates

[9] https://mitosystems.com/software-evolution/ (visited on 03.09.2021)

[10] F. Doesburg, F. Cnossen, W. Dieperink, W. Bult, A. M. de Smet, D. J. Touw, and M. W. Nijsten, "Improved usability of a multi-infusion setup using a centralized control interface: A task-based usability test," PLOS ONE, vol. 12, no. 8, pp. 1–10, 08 2017. [Online]. Available: `https://doi.org/10.1371/journal.pone.0183104` (visited on 03.09.2021)

[11] `https://en.wikipedia.org/wiki/Dependency_hell` (visited on 03.09.2021)

[12] `https://semver.org/` (visited on 03.09.2021)

[13] `https://arstechnica.com/gadgets/2018/12/logitech-firmware-update-breaks-locally-cont` (visited on 28.08.2021)

[14] `https://venturebeat.com/2018/02/28/apples-ios-update-frequency-has-increased-51-unde` (visited on 06.09.2021)

[15] `https://www.businessinsider.de/insider-picks/technik/die-meistverkauften-smartphones-2021` (visited on 12.09.2021)

[16] `https://de.wikipedia.org/wiki/IPhone_12` (visited on 12.09.2021)

Versicherung an Eides Statt

Ich versichere an Eides statt durch meine untenstehende Unterschrift,

- dass ich die vorliegende Arbeit - mit Ausnahme der Anleitung durch die Betreuer - selbstständig ohne fremde Hilfe angefertigt habe und
- dass ich alle Stellen, die wörtlich oder annähernd wörtlich aus fremden Quellen entnommen sind, entsprechend als Zitate gekennzeichnet habe und
- dass ich ausschließlich die angegebenen Quellen (Literatur, Internetseiten, sonstige Hilfsmittel) verwendet habe und
- dass ich alle entsprechenden Angaben nach bestem Wissen und Gewissen vorgenommen habe, dass sie der Wahrheit entsprechen und dass ich nichts verschwiegen habe.

Mir ist bekannt, dass eine falsche Versicherung an Eides Statt nach §156 und §163 Abs. 1 des Strafgesetzbuches mit Freiheitsstrafe oder Geldstrafe bestraft wird.

Duisburg, 12. September 2021
(Ort, Datum)

(Vorname Nachname)