Institut für Architektur von Anwendungssystemen

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Topologie-basierte und verteilte Provisionierung von IoT Anwendungen

Benjamin Weder

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. Dr. h. c. Frank Leymann
Betreuer/in:	M.Sc. Kálmán Képes
Beginn am:	15.07.2018
Beendet am:	10.01.2019

Kurzfassung

Das Paradigma des Internet of Things (IoT) beschreibt die Vernetzung von beliebigen Dingen innerhalb des Internets. Mit dem zunehmenden Erfolg des IoT Paradigmas steigt die Anzahl verbundener Dinge bzw. Geräte kontinuierlich an. Für Anbieter von IoT Anwendungen entstehen damit neue Herausforderung bei der Provisionierung und Wartung solcher Anwendungen. Eine manuelle Provisionierung und Wartung ist aufgrund der großen Anzahl an Komponenten und ihrer Heterogenität eine komplexe, fehleranfällige und teure Aufgabe. Deshalb werden Techniken und Technologien benötigt, um diese Aufgaben zu automatisieren. Im Anwendungsbereich des Cloud Computing kann die Provisionierung von Anwendungen durch sogenannte Deployment Systeme vollständig automatisiert werden. Ein Einsatz von Deployment Systemen in IoT Szenarien ist ebenfalls möglich, allerdings kann die Heterogenität von Geräten und die Verteilung von Anwendungen über verschiedene private Netzwerke dazu führen, dass die Provisionierung scheitert. So können zum Beispiel Firewalls den Zugriff auf Geräte verhindern, auf denen Teile einer Anwendung provisioniert werden sollen.

Im Rahmen dieser Arbeit wurde deshalb ein Konzept entwickelt, um die automatische Provisionierung von verteilten IoT Anwendungen mit einem Deployment System zu ermöglichen. Dazu wird gezeigt, wie ein verteiltes Deployment System die Probleme, die bei der Provisionierung von verteilten IoT Anwendungen mit einem zentralisierten Deployment System auftreten, umgehen kann. Weiterhin wird ermittelt, auf welche Weise die verteilten Knoten eines solchen Deployment Systems kommunizieren können und wie anhand der Topologie einer Anwendung bestimmt werden kann, welcher Knoten für die Provisionierung von welchen Teilen der Anwendung verantwortlich ist. Das entwickelte Konzept wurde prototypisch im OpenTOSCA Container, einem auf dem TOSCA-Standard basierenden, automatischen Deployment System, implementiert und anschließend validiert.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

Verzeichnis der Algorithmen

Abkürzungsverzeichnis

API Application Programming Interface.

ARM Advanced RISC Machines.

BPEL Business Process Execution Language.

BPMN Business Process Model and Notation.

CSAR Cloud Service Archive.

DA Deployment Artifact.

DSL Domain Specific Language.

FTP File Transfer Protocol.

IA Implementation Artifact.

IoT Internet of Things.

JMS Java Message Service.

MQTT Message Queue Telemetry Transport.

NIST National Institute of Standards and Technology.

OASIS Organization for the Advancement of Structured Information Standards.

QoS Quality of Service.

REST Representational State Transfer.

SSH Secure Shell.

TCP Transmission Control Protocol.

TOSCA Topology and Orchestration Specification for Cloud Applications.

URI Uniform Resource Identifier.

VMM Virtual Machine Monitor.

WAR Web Application Archive.

XML Extensible Markup Language.

1. Einleitung

In den letzten Jahren wurde eine immer größere Anzahl an neuen Anwendungen für das *Internet of Things (IoT)* entwickelt [SBH+17]. Dabei sind Anwendungsbereiche sowohl im öffentlichen, als auch im privaten Raum vorzufinden. So können IoT Anwendungen zum Beispiel in der *Industrie 4.0* oder in *Smart Homes* eingesetzt werden. Für IoT Anwendungen werden üblicherweise viele Geräte mit angeschlossenen Sensoren und Aktuatoren benötigt, um beispielsweise die Temperatur in einem Smart Home zu überwachen und zu steuern [RMPC16]. Die manuelle Bereitstellung und Wartung aller Hardware- und Softwarekomponenten wird damit sehr aufwendig, fehleranfällig und teuer [NSL+14]. Diese Probleme werden durch die Heterogenität der Netzwerke und Geräte im Kontext des IoT noch weiter verstärkt. Deshalb muss eine Möglichkeit gefunden werden, die Bereitstellung und Wartung von IoT Anwendungen möglichst vollständig zu automatisieren.

Im Bereich des *Cloud Computing* wird eine automatische Bereitstellung und Wartung von Anwendungen bereits durch verschiedene Verfahren ermöglicht [VRCL08]. Es ist zum Beispiel realisierbar, Cloud Anwendungen mittels des *TOSCA Standards* portabel und interoperabel zu definieren. Anschließend können diese durch ein auf TOSCA basiertes Deployment System, einer sogenannten *TOSCA-Runtime*, automatisch bereitgestellt werden [SBH+17]. Der TOSCA Standard kann ebenfalls verwendet werden, um Anwendungen im Kontext des IoT zu definieren. Bei der automatischen Bereitstellung und Wartung der IoT Anwendungen können jedoch Probleme auftreten, die bei Cloud Anwendungen nicht vorhanden sind. Dies ist insbesondere der Fall, wenn es sich um verteilte IoT Anwendungen handelt, bei denen Teile der Anwendung in mehreren unterschiedlichen Netzwerken ausgeführt werden sollen. So können zum Beispiel Firewalls die automatische Bereitstellung erschweren oder ganz verhindern. Ebenso kann die Effizienz der Bereitstellung durch die Verteilung und die Heterogenität von Netzwerken und Geräten deutlich beeinträchtigt werden.

Das Ziel dieser Masterarbeit ist es daher, die Probleme der automatischen Provisionierung von verteilten IoT Anwendungen zu analysieren. Darauf aufbauend soll ein Konzept entwickelt werden, mit dem verteilte IoT Anwendungen dennoch automatisch und effizient bereitgestellt bzw. gewartet werden können. Außerdem soll eine prototypische Implementierung des Konzepts umgesetzt werden. Als Basis der Implementierung dient dabei der *OpenTOSCA Container* [Ope], eine Open Source TOSCA-Runtime, die an der Universität Stuttgart entwickelt wurde. Anhand der prototypischen Implementierung soll abschließend eine Validierung des entwickelten Konzepts durchgeführt werden.

1.1. Motivierendes Szenario

Abbildung 1.1 stellt beispielhaft eine verteilte IoT Anwendung dar, für die eine automatische Provisionierung einer beliebigen Anzahl von Anwendungsinstanzen wünschenswert wäre, um den großen Aufwand der manuellen Bereitstellung zu vermeiden. Für die Darstellung wurde die Topologie der Anwendung, also der schematische Aufbau mit allen beteiligten Komponenten, mit dem TOSCA Standard (siehe Abschnitt 2.6) modelliert. Der TOSCA Standard bietet eine XML- bzw. YAML-basierte Sprache zur Beschreibung von Anwendungen, aber derzeit keine standardisierte grafische Notation [BBK+12]. Um eine schnelle Erfassung von Anwendungen zu ermöglichen und grundlegende Informationen auch an Menschen ohne fundierte Kenntnisse von TOSCA übermitteln zu können, wird jedoch eine solche Notation benötigt. Für Abbildung 1.1 und alle weiteren Abbildungen von TOSCA Topologien in dieser Arbeit, wird deshalb die von Breitenbücher et al. [BBK+12] vorgestellte grafische Notation *Vino4TOSCA* verwendet.

Die in Abbildung 1.1 dargestellte Anwendung ist im Zuge des BMWi Projekts *Smart Orchestra*[1] als möglicher Anwendungsfall entstanden. Das Ziel des Smart Orchestra Projekts ist, eine konfigurierbare *Smart Service Plattform* für IoT Anwendungen bzw. Systeme zu entwickeln [LAB+18]. Die Service Plattform verfügt dabei unter anderem über einen *Markplatz*, über den Nutzer Informationen zu den angebotenen Services abrufen können. Wenn sich ein Nutzer entscheidet einen Service zu bestellen, soll dieser automatisch von der Service Plattform bereitgestellt werden. Dazu muss diese eine sogenannte *Provisioning Engine* enthalten, die in der Lage ist, das automatische Deployment von Smart Services durchzuführen. Im Smart Orchestra Projekt wird als Provisioning Engine die Open Source TOSCA-Runtime *OpenTOSCA Container* (siehe Abschnitt 2.7) verwendet. Deshalb muss es ermöglicht werden, dass die präsentierte Anwendung mit dem OpenTOSCA Container in heterogenen und verteilten IoT Umgebungen bereitgestellt werden kann.

Grundsätzlich lässt sich die Anwendung in drei Bestandteile unterteilen. Dies wird anhand der drei Stapel in der Topologie der Anwendung deutlich. Der linke Teil der Anwendung besteht aus einem Raspberry Pi mit einem angeschlossenen Sensor zur Messung der Luftfeuchtigkeit und der Temperatur. Außerdem wird auf dem Raspberry Pi ein Python Adapter ausgeführt, der die Messwerte des Sensors ausliest und auf zwei unterschiedlichen Topics eines *Fiware Kontext Brokers*[2] veröffentlicht (siehe Publish/Subscribe in Abschnitt 2.2). Der verwendete Broker muss dabei nicht durch die Anwendung provisioniert werden, sondern ist bereits Teil der Smart Service Plattform [LAB+18]. Für den mittleren Teil der Anwendung muss eine virtuelle Maschine mit Ubuntu als Betriebssystem in einer *OpenStack Cloud*[3] erstellt werden. Auf der virtuellen Maschine wird ein Python Skript ausgeführt, das die Sensorwerte aus den beiden Topics ausließt und daraus einen Index berechnet, der die Wahrscheinlichkeit eines Schimmelbefalls angibt. Dieser Index wird auf einem dritten Topic veröffentlicht. Der letzte Bestandteil der Anwendung auf der rechten

[1] https://smartorchestra.de/

[2] https://fiware-orion.readthedocs.io/en/master/

[3] https://www.openstack.org/

Seite beinhaltet, wie der linke Teil, ein Raspberry Pi, auf dem ein Python Skript ausgeführt wird. Dieses Skript liest den Indexwert auf dem Topic periodisch aus und erzeugt ein Warnsignal, sobald dieser einen kritischen Wert übersteigt. Die gesamte Anwendung kann also beispielsweise dazu verwendet werden, einen Raum in einem Smart Home zu überwachen und den Bewohner bei einer erhöhten Schimmelgefahr zu warnen.

1.2. Aufbau der Arbeit

Die vorliegende Arbeit ist folgendermaßen gegliedert:

Kapitel 2 - Grundlagen:

Dieses Kapitel führt die für das Verständnis der Arbeit nötigen Grundlagen ein.

Kapitel 3 - Verwandte Arbeiten:

Hier werden verwandte Arbeiten aus dem Bereich des verteilten Anwendungs- und Netzwerkmanagement präsentiert.

Kapitel 4 - Problemstellung und Anforderungen:

In diesem Kapitel werden die in der Arbeit behandelten Problemstellungen und die daraus resultierenden Anforderungen an das Lösungskonzept beschrieben.

Kapitel 5 - Konzept der verteilten Provisionierung von IoT Anwendungen:

Dieses Kapitel stellt das im Zuge dieser Arbeit entwickelte Konzept zur verteilten Provisionierung von IoT Anwendungen vor.

Kapitel 6 - Implementierung:

In diesem Kapitel werden Details der prototypischen Implementierung des entwickelten Konzepts vorgestellt. Die Grundlage der Implementierung bildet der OpenTOSCA Container. Es wird beschrieben, wie die Architektur des OpenTOSCA Containers zur Umsetzung des Konzepts angepasst werden muss. Außerdem werden einige technische Details präsentiert.

Kapitel 7 - Validierung des Konzepts und der Implementierung:

Dieses Kapitel beschäftigt sich mit der Validierung des entwickelten Konzepts und der prototypischen Implementierung. Dazu werden die Anforderungen überprüft und es wird beschrieben, wie das motivierende Szenario durch das Konzept umgesetzt werden kann.

Kapitel 8 - Zusammenfassung und Ausblick:

Im letzten Kapitel erfolgt eine kurze Zusammenfassung der Kernthemen dieser Arbeit. Abschließend werden in einem Ausblick Anknüpfungspunkte für mögliche Weiterentwicklungen vorgestellt.

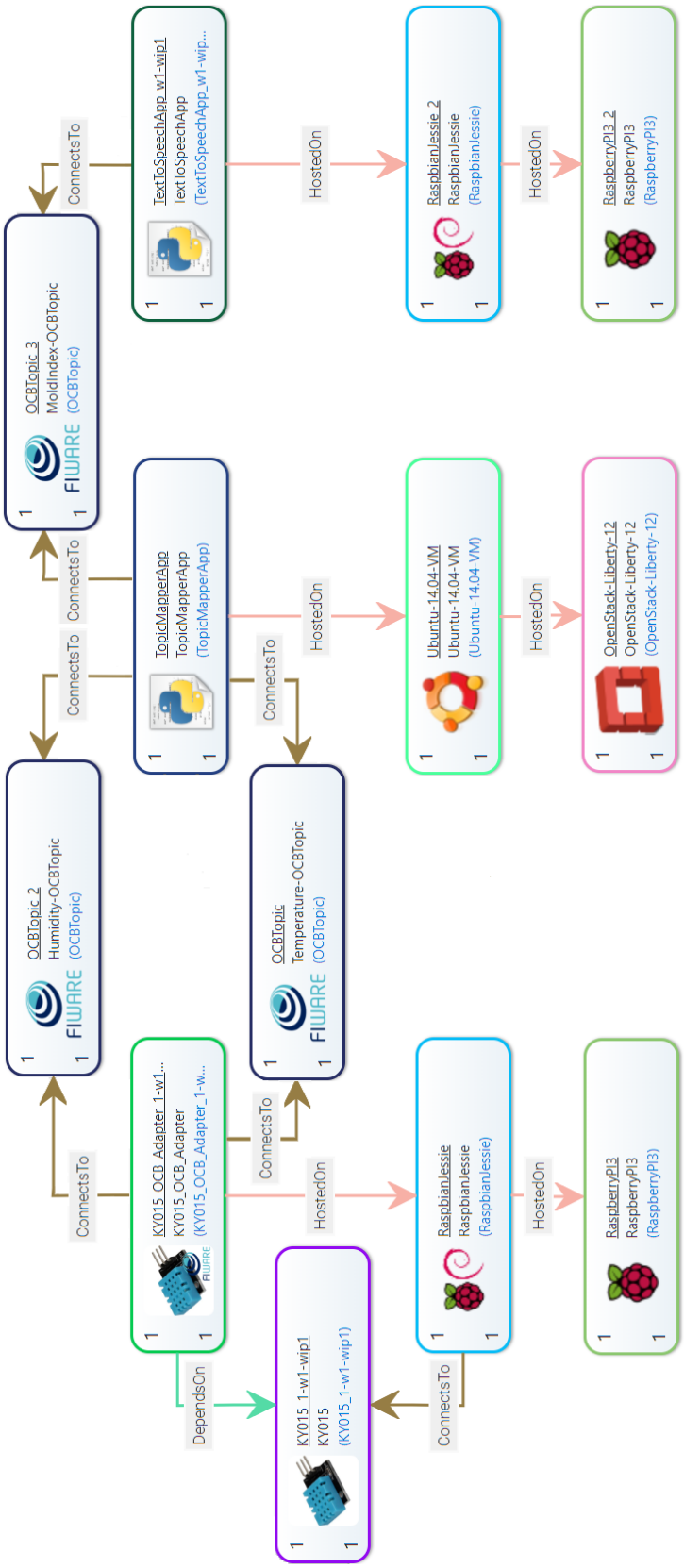

Abbildung 1.1.: Anwendungsfall aus dem BMWi Projekt Smart Orchestra

2. Grundlagen

In diesem Kapitel werden die Grundlagen vorgestellt, die für das Verständnis der folgenden Kapitel elementar sind. Im ersten Abschnitt werden der Begriff des *Internet of Things (IoT)* und Besonderheiten von IoT Anwendungen beschrieben. Anschließend erfolgt die Einführung des *Publish/Subscribe* Interaktionsmodells und des *Message Queuing Telemetry Transport (MQTT)* Protokolls, das auf diesem Modell basiert. In den nächsten Abschnitten werden das *OSGi Framework* und das Integrations-Framework *Apache Camel* für die Programmiersprache Java betrachtet. Danach erfolgt die Erläuterung des Konzepts des *Cloud Computing* und des *TOSCA Standards*, der es erlaubt, Cloud Anwendungen portabel und interoperabel zu definieren. Nachfolgend wird das Open Source Projekt *OpenTOSCA* vorgestellt, das an der Universität Stuttgart entwickelt wurde und als Basis für die prototypische Implementierung des Konzepts dieser Arbeit dient. Abschließend finden eine Betrachtung der Open Source Software *Docker* statt, die zur Containervirtualisierung verwendet werden kann.

2.1. Internet of Things

Das Paradigma des *Internet of Things (IoT)* erfreut sich schon seit einigen Jahren eines großen Interesses der Wissenschaft und Industrie [SGFW10]. Der Begriff wird aber auch zunehmend in der breiten Öffentlichkeit diskutiert und dabei meistens in Bezug auf Anwendungen in *Smart Homes* oder der *Industrie 4.0* verwendet. Grundlage des IoT Paradigmas ist die Verbindung der physischen und der virtuellen Welt [SBK+16]. Dabei spricht man von sogenannten *cyber-physischen Systemen*. Um die physische Welt zu manipulieren, beinhalten solche Systeme *Smart Devices*, wie zum Beispiel *Raspberry Pis*, mit angeschlossenen Aktuatoren und Sensoren [SBH+17]. In einem Smart Home können zum Beispiel Sensoren zur Messung der Raumtemperatur und Aktuatoren zur Steuerung der Klimaanlage verwendet werden. Zusätzlich zu den Smart Devices besteht der virtuelle Teil eines cyber-physischen Systems aus Software zur Extraktion, Aggregation und Evaluation von Sensordaten [SBK+16]. Im genannten Smart Home Beispiel kann die Software also dazu verwendet werden, die Daten mehrerer Sensoren zu analysieren und, bei einer Überschreitung eines Schwellenwertes, die Klimaanlage mittels eines Aktuators zu starten.

IoT Umgebungen bestehen üblicherweise aus drei Bausteinen [SBH+17]: *Smart Devices* mit Sensoren und Aktuatoren, *IoT Middleware* und der *IoT Anwendung*. Abbildung 2.1 zeigt den schematischen Aufbau einer IoT Umgebung und die Kommunikation zwischen den verschiedenen Bausteinen. Smart Devices sind physische Geräte mit installierter Software zur

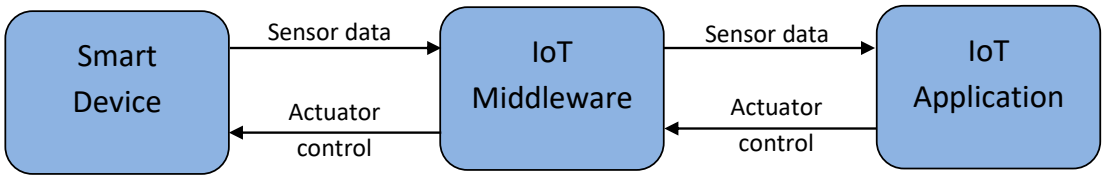

Abbildung 2.1.: Bausteine einer IoT Umgebung (angelehnt an [SBH+17])

Auslesung von Sensordaten, zur Steuerung von Aktuatoren und zur Kommunikation mit der IoT Middleware. Die IoT Middleware ist das Verbindungsstück zwischen den Smart Devices und der IoT Anwendung. Sie ermöglicht die Kommunikation, ohne dass sich Anwendung und Device dabei direkt kennen müssen und bietet damit eine Abstraktionsebene. Somit ist es möglich, Anwendungen und Smart Devices leicht auszutauschen bzw. neue Smart Devices zu einer bestehenden Anwendung hinzuzufügen. Ein Beispiel für eine IoT Middleware ist der Open Source Message Broker *Mosquitto*[1], der das *MQTT Protokoll* verwendet, das in Abschnitt 2.2 vorgestellt wird [SBK+16]. Die IoT Anwendung verbindet sich mit der IoT Middleware, um über sie Sensordaten zu erhalten und Befehle an die Aktuatoren zu senden [SGFW10]. Beispiele für IoT Anwendungen sind die bereits genannte Anwendung zur Steuerung einer Klimaanlage oder ein Dashboard, das die Daten aller Sensoren in einer Fabrikhalle extrahiert und den Arbeitern Abweichungen von der Norm anzeigt.

Nachfolgend werden einige der wichtigsten Eigenschaften von IoT Umgebungen bzw. Anwendungen nach Razzaque et al. [RMPC16] vorgestellt:

- *Heterogene Smart Devices*: In IoT Umgebungen wird häufig eine große Anzahl unterschiedlicher Devices eingesetzt. Dabei unterscheiden sich die Devices unter anderem anhand der verwendeten Protokolle, der vorhandenen Hardwarekapazitäten, wie beispielsweise der Prozessor- oder Speicherkapazität, der bereitgestellten Funktionalität und dem Hersteller.
- *Beschränkte Ressourcen*: Viele Smart Devices haben nur sehr limitierte Prozessor-, Speicher- und Netzwerkkapazitäten. Ein Grund dafür ist, dass trotz der großen Anzahl an benötigten Devices die Kosten gering gehalten werden müssen. Außerdem müssen Smart Devices häufig batteriebetrieben werden. Zu große Prozessorkapazitäten würden jedoch zu einem zu hohen Stromverbrauch führen.
- *Große und dynamische Netzwerke*: Netzwerke in IoT Umgebungen beinhalten eine größere Menge an Geräten als die meisten konventionellen Netzwerke. Damit kann eine sehr große Anzahl an Events im Netzwerk entstehen, die geeignet behandelt werden müssen, um keine Überlastung zu verursachen. Eine weitere Schwierigkeit ist die Dynamik des Netzwerks. In IoT Netzwerken werden häufig neue Geräte

[1] https://www.mosquitto.org/

hinzugefügt oder alte Geräte entfernt. Zusätzlich verbinden sich viele Geräte aufgrund der begrenzten Kapazität ihrer Batterie nur periodisch mit dem Netzwerk, um Strom zu sparen. Damit kann die Anzahl der Geräte im Netzwerk noch stärker schwanken.

- *Verteilte Systeme*: Anwendungen im IoT sind hochgradig verteilt. Diese Verteilung findet zum einen lokal statt, da zum Beispiel eine große Zahl an unterschiedlichen Devices innerhalb einer Fabrik eingesetzt wird. Zum anderen können IoT Anwendungen aber auch global verteilt sein, indem beispielsweise Teile in der Cloud (siehe Abschnitt 2.5) gehostet werden und nur die Smart Devices lokal betrieben werden.
- *Context-awareness*: Die Sensoren einer IoT Umgebung generieren große Mengen an Daten, die zuerst analysiert und interpretiert werden müssen, bevor daraus wertvolle Informationen entstehen können. Um die Interpretation zu ermöglichen, speichern IoT Umgebungen Kontextinformationen zusammen mit den Sensordaten. Diese Kontextinformationen können zum Beispiel die Position des Sensors und die Zeit, zu der die Daten erhoben wurden, beinhalten.
- *Hohe Sicherheitsanforderungen*: Das Internet of Things benötigt für viele Anwendungen eine globale Konnektivität und Zugreifbarkeit. Das bedeutet, dass jeder zu jeder Zeit von jedem Ort aus auf das IoT zugreifen kann. Damit bietet sich eine große Angriffsfläche für IoT Anwendungen. Um Missbrauch und Angriffen vorzubeugen, müssen Sicherheitsmechanismen in IoT Anwendungen eingebaut werden, die die Komplexität der Anwendungen und Netzwerke noch zusätzlich steigern.

Durch die heterogenen Geräte, die großen und dynamischen Netzwerke, die verteilten Anwendungen und die benötigten Sicherheitsmechanismen ist die Provisionierung und Wartung von IoT Umgebungen eine sehr komplexe Aufgabe [NSL+14]. Das manuelle Durchführen dieser Aufgabe ist zeitaufwendig, fehleranfällig und teuer [LVCD13]. Deshalb sollten möglichst große Teile der Provisionierung und Wartung automatisiert durchgeführt werden [SBH+17]. Im Idealfall muss lediglich die physische Installation der Smart Devices manuell vorgenommen werden. Alle weiteren Schritte können anschließend automatisiert umgesetzt werden.

Eine vollständige Automatisierung kann bereits in vielen Anwendungsbereichen erreicht werden [SBH+17]. Dazu können automatische Deployment Systeme, wie der in Abschnitt 2.7 vorgestellte *OpenTOSCA Container*, verwendet werden. Solche Systeme können jedoch zum Beispiel an Firewalls scheitern, welche die Smart Devices vor unbefugten Zugriffen aus fremden Netzwerken schützen sollen. Das Ziel dieser Arbeit ist es, ein Konzept zu entwickeln um diese Problematiken im Kontext des IoT zu lösen und den Einsatz von automatisierten Deployment Systemen somit in einem noch breiteren Anwendungsbereich zu ermöglichen.

2.2. Publish/Subscribe und MQTT

Beim *Publish/Subscribe Interaktionsmodell* werden die Sender von Nachrichten *Publisher* und die Empfänger *Subscriber* genannt [SBK+16]. Subscriber registrieren sich, bevor sie Nachrichten erhalten können, bei einer Menge an *Topics*, für die sie sich interessieren. Diese Registrierungen werden als *Subscriptions* bezeichnet. Ab dem Moment der Subscription erhält der Subscriber asynchron jede Nachricht, die auf dem Topic veröffentlicht wird. Topics und ihre zugehörigen Subscriptions werden von *Message Brokern* verwaltet [SBH+17]. Jedes Topic bezieht sich dabei auf einen bestimmten Themenbereich. Bei einer Nachrichtenanwendung kann ein Topic beispielsweise Politik und ein anderes Sport lauten. Topics können außerdem in *Hierarchien* angeordnet werden [LP+03]. Das Sport Topic kann zum Beispiel zwei Subtopics Fußball und Basketball besitzen. Wenn eine Nachricht auf dem Subtopic veröffentlicht wird, wird diese sowohl den Subscribern auf diesem Topic, als auch auf allen Topics weiter oben in der Hierarchie zur Verfügung gestellt. Damit wird es Subscribern erlaubt, feingranularere Subscriptions durchzuführen bzw. die Menge an Subscriptions zu verringern. Ein Subscriber, der sich lediglich für Fußball interessiert, kann sich also nur für dieses Topic registrieren, wohingegen ein anderer Subscriber sich für Fußball und Basketball interessieren könnte und deshalb eine Subscription auf das Topic Sport durchführt.

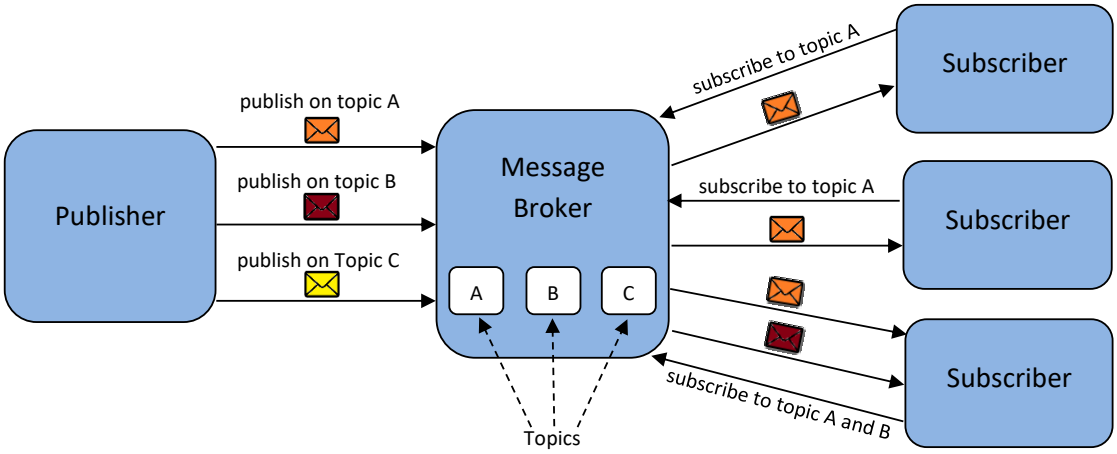

Abbildung 2.2.: Publish/Subscribe Interaktionsmodell (angelehnt an [HW04])

Abbildung 2.2 zeigt ein Beispiel für die Interaktionen beim Publish/Subscribe Modell. In der Mitte ist der Message Broker mit den drei Topics A, B und C zu sehen. Je nach Art des Brokers müssen diese Topics explizit angelegt werden oder werden implizit erstellt, sobald die erste Subscription bzw. die erste Veröffentlichung zum Topic vom Broker empfangen wird [SBH+17]. Die oberen beiden Subscriber registrieren sich lediglich für das Topic A. Der untere Subscriber erstellt dagegen Subscriptions für die Topics A und B. Im Beispiel veröffentlicht der Publisher zu jedem der drei Topics des Brokers je eine Nachricht. Man kann sehen, dass die Nachricht auf Topic A (orange Nachricht) vom Broker,

basierend auf den Subscriptions, an alle drei Subscriber ausgeliefert wird. Die Nachricht auf Topic B (rote Nachricht) wird dagegen nur vom letzten Subscriber empfangen und die Nachricht auf Topic C (gelbe Nachricht) erhält gar kein Subscriber. Anhand dieses Beispiels lässt sich der grundlegende Unterschied zwischen Punkt-zu-Punkt Verbindungen und dem Publish/Subscribe Prinzip feststellen. Nachrichten können bei Publish/Subscribe von 0 bis n Empfängern erhalten werden, im Gegensatz zu genau einem Empfänger, wie es bei Punkt-zu-Punkt Verbindungen üblich ist.

Das Verwenden des Publish/Subscribe Interaktionsmodells ermöglicht es, Anwendungen, die Daten senden müssen und Anwendungen, die diese Daten empfangen, bis zu einem gewissen Grad zu entkoppeln [HW04]. Die sendende Anwendung muss lediglich den Message Broker und das Topic kennen und benötigt kein Wissen über die empfangende Anwendung. Die empfangende Anwendung benötigt ebenfalls kein Wissen darüber, welche Anwendungen auf dem Topic Nachrichten veröffentlichen. Es besteht jedoch eine zeitliche Abhängigkeit, da der Subscriber nur dann Nachrichten erhält, wenn er gerade aktiv ist. Wenn er durch einen Fehler also kurzfristig ausfällt, verpasst er Nachrichten, die während dieser Zeit veröffentlicht werden. Eine zweite Abhängigkeit besteht im Nachrichtenformat. Der Subscriber muss das Format verstehen, das vom Publisher veröffentlicht wird. Um diese beiden Abhängigkeiten aufzulösen, können die von Gregor Hohpe in seinem Buch „*Enterprise Integration Patterns*“ [HW04] definierten Pattern verwendet werden. Das Pattern „*Durable Subscriber*“ ermöglicht es Subscribern zu definieren, dass Nachrichten, während der Zeit, in der sie die Verbindung zum Broker verlieren, gespeichert werden sollen. Die Nachrichten werden dann ausgeliefert, sobald der Subscriber wieder eine Verbindung zum Broker herstellt. Mit dem Pattern „*Message Translator*“ kann dagegen das Nachrichtenformat auf dem Weg der Nachricht vom Publisher zum Subscriber transformiert werden und somit die Abhängigkeit vom Nachrichtenformat aufgelöst werden.

Das *Message Queuing Telemetry Transport (MQTT)* Protokoll ist ein von der *Organization for the Advancement of Structured Information Standards (OASIS)* standardisiertes, leichtgewichtiges Publish/Subscribe Protokoll [SBK+16]. Durch die leichtgewichtige Art, ist das MQTT Protokoll für Geräte mit beschränkten Ressourcen und für unzuverlässige Netzwerke mit niedrigen Bandbreiten geeignet. Aufgrund dieser Eigenschaften wird es häufig in IoT Umgebungen eingesetzt [AGM+15]. Technisch baut das MQTT Protokoll auf dem *TCP* Protokoll auf und gehört damit zur Anwendungsschicht des *OSI-Modells* [Ker92]. Zum Zeitpunkt dieser Arbeit ist die aktuelle MQTT Version, die von den meisten Implementierungen umgesetzt wird, *v3.1.1*. In dieser Version erlaubt das MQTT Protokoll keine benutzerdefinierten Header Felder [OASa]. Dies hat zur Folge, dass Metadaten, die in anderen Protokollen als Header übertragen werden können, bei MQTT vom Sender in den Body der Nachrichten eingefügt werden müssen und der Empfänger diese anschließend wieder extrahieren muss. Im Mai 2018 wurde von OASIS die MQTT Version *v5.0.0* angekündigt, die unter anderem dieses Problem behebt und benutzerdefinierte Header Felder zulässt [OASb]. Da diese Version jedoch aktuell von den meisten Anbietern von Bibliotheken oder Brokern noch umgesetzt werden muss, kann sie für die prototypische Implementierung dieser Arbeit nicht verwendet werden.

Je nachdem, in welcher Anwendung das MQTT Protokoll eingesetzt werden soll, kann zwischen drei *Quality of Service (QoS)* Leveln gewählt werden [LKHJ13]. Die Level unterscheiden sich anhand ihrer Garantie für die Nachrichtenübertragung und ihrer Performance. Dabei bietet QoS Level 0 die beste Performance, dafür aber keinerlei Garantie, dass die Nachricht beim Empfänger ankommt. QoS Level 1 bietet dem Nutzer dagegen, für eine leicht verschlechterte Performance, eine sogenannte „at-least-once“ Garantie. Dies bedeutet, dass bei QoS Level 1 die Nachricht auf jeden Fall einmal den Empfänger erreicht. Es ist jedoch möglich, dass dabei Duplikate entstehen und der Empfänger die Nachricht somit mehrfach erhält. QoS Level 2 weist die schlechteste Performance der drei Level auf. Dafür wird jedoch garantiert, dass die Nachricht exakt einmal beim Empfänger ankommt.

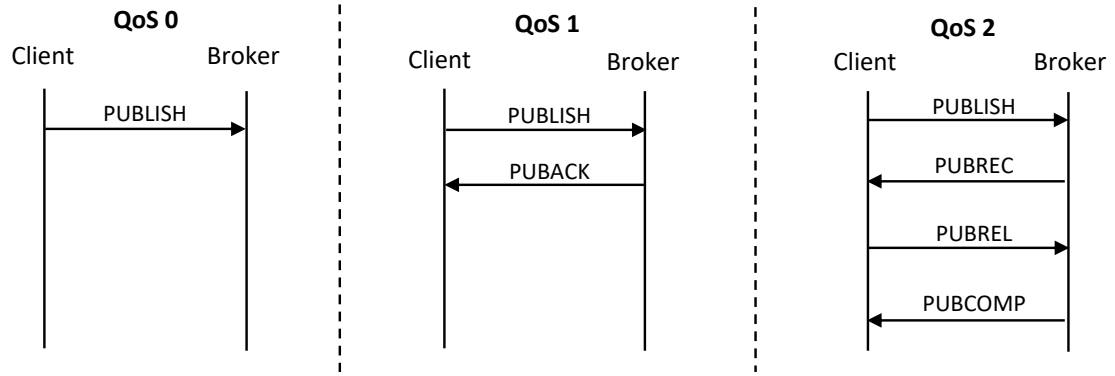

Abbildung 2.3.: Nachrichtenaustausch bei verschiedenen QoS Leveln (nach [LKHJ13])

Abbildung 2.3 zeigt den Nachrichtenaustausch, der zur Realisierung der verschiedenen QoS Level benötigt wird. Bei QoS Level 0 wird die Nachricht lediglich einmal übertragen, ohne sie zu bestätigen oder zwischenzuspeichern [LKHJ13]. Für QoS Level 1 dagegen speichert der Sender die Nachricht, bis er eine Bestätigung für sie erhalten hat. Dazu sendet der Empfänger nach dem Erhalt einer Nachricht ein „PUBACK“, das die ID der Nachricht beinhaltet. Wenn der Sender eine Zeit lang keine Bestätigung für eine Nachricht erhält, überträgt er diese erneut. Bei Verlust eines „PUBACK“ kann es daher dazu kommen, dass der Empfänger die Nachricht zweimal erhält und verarbeitet. Dies wird bei QoS 2 durch einen sogenannten *4-way Handshake* vermieden [LKHJ13]. Der Sender speichert die Nachricht für diesen QoS ebenfalls vor dem Senden. Zusätzlich speichert der Empfänger beim Erhalt einer Nachricht ihre ID. Damit kann vermieden werden, dass der Empfänger die Nachricht beim erneuten Übertragen, aufgrund des Verlustes des „PUBREC“, nochmal verarbeitet. Beim Empfang des „PUBREL“ kann der Empfänger die ID der Nachricht löschen und sendet als Antwort ein „PUBCOMP“. Wenn das „PUBCOMP“ verloren geht, erhält der Empfänger erneut ein „PUBREL“. Da die ID, auf die sich das „PUBREL“ bezieht, allerdings schon gelöscht wurde, kann das „PUBCOMP“ nochmal übertragen werden. Sobald der Sender das „PUBCOMP“ erhält, kann er ebenfalls alle Informationen über die Interaktion löschen und es ist sichergestellt, dass die Nachricht exakt einmal verarbeitet wurde.

2.3. OSGi Framework

In diesem Abschnitt wird das *OSGi Framework* eingeführt [OSG]. Es dient als Software-Plattform für den *OpenTOSCA Container*, der in Abschnitt 2.7 als Teil des *OpenTOSCA Ökosystems* vorgestellt wird und die Grundlage der Implementierung dieser Arbeit bildet. Das Ziel des OSGi Frameworks ist es, eine Plattform für die Programmiersprache Java bereitzustellen, die es ermöglicht, Anwendungen und Services leicht zu modularisieren und Abhängigkeiten zwischen Modulen dynamisch zur Laufzeit aufzulösen. Die Referenzimplementierung der OSGi Spezifikation, die auch in dieser Arbeit verwendet wird, ist *Equinox*, das von der Eclipse Foundation verwaltet wird [Ecl]. Neben Equinox existiert eine Reihe weiterer Implementierungen der OSGi Spezifikation, die aber häufig nur Teile des Standards umsetzen.

Die Grundeinheit zur Modularisierung einer Anwendung sind sogenannte *Bundles*, die im OSGi Framework installiert, gestartet, gestoppt und deinstalliert werden können [LNH03]. Jedes Bundle stellt dabei eine in sich möglichst geschlossene Softwarekomponente dar, die dem Nutzer oder anderen Bundles eine bestimmte Funktionalität anbietet [OSG]. Ein Bundle ist ein *Java Archive (JAR)*, bestehend aus den Java Klassen zur Implementierung der bereitgestellten Funktionalität, einer Manifest Datei und möglichen weiteren Ressourcen, wie beispielsweise Konfigurationsdateien [LNH03]. Die Manifest Datei beinhaltet die Metadaten für das Bundle, wie zum Beispiel den Bundle-Name, die Bundle-Version und Abhängigkeiten zu anderen Bundles. Eine Besonderheit des OSGi Frameworks ist, dass unterschiedliche Versionen desselben Bundles gleichzeitig verwendet werden können. Damit wird es zum Beispiel ermöglicht, ein Update eines Bundles stufenweise auszurollen.

Wenn ein Bundle anderen Bundles Teile seiner Funktionalität anbietet, spricht man dabei im OSGi Kontext von *Services* [LNH03]. Ein Service ist ein Java Objekt, dass per Interface definiert wurde. Services können vom Bundle zusammen mit den Interfaces, die sie implementieren, bei der OSGi *Service Registry* registriert werden. Anschließend können die Services von anderen Bundles durch Queries bei der Service Registry entdeckt und verwendet werden [OSG]. Durch die Trennung von Interfaces und konkreten Implementierungen kann die Implementierung später leichter angepasst werden, ohne dabei andere Bundles zu beeinflussen, da diese lediglich vom Interface abhängen und die Implementierung über die Service Registry laden können. Dies ermöglicht es sogar, dass Implementierungen zur Laufzeit des Frameworks dynamisch ausgetauscht werden können [LNH03].

Ein weiterer Vorteil des OSGi Frameworks ist, dass für ein einzelnes Interface mehrere Implementierungen in der Service Registry registriert werden können. Dies erlaubt es, ein Plug-In System, basierend auf einem Interface, aufzubauen. So kann beispielsweise ein Interface mit zwei Methoden erstellt werden. Dabei stellt die eine Methode eine Funktionalität für eine bestimmte Art von Ressourcen bereit und die andere Methode gibt die Art von Ressourcen zurück, auf die die erste Methode anwendbar ist. Wenn ein Bundle das Plug-In System nutzen möchte, kann es anschließend über die Service Registry alle verfügbaren Plug-Ins abrufen, über die zweite Methode bestimmen, welches Plug-In für eine aktuelle Ressource das Richtige ist und dann die erste Methode des Plug-Ins mit der Ressource aufrufen.

In Abbildung 2.3 ist die mögliche Interaktion von zwei Bundles mit der Service Registry grafisch dargestellt. Bundle 1 und 2 wurden im OSGi Framework installiert und gestartet. Bundle 1 bietet Funktionalität für andere Bundles an und registriert deshalb einen Service bei der Service Registry. Bundle 2 möchte diesen Service nutzen und sendet dazu eine Query an die Service Registry, um den Service zu entdecken. Die Service Registry liefert daraufhin die von Bundle 1 registrierte Implementierung an Bundle 2. Bundle 2 kann diese nun solange verwenden bis sie von Bundle 1 abgemeldet wird. Dies passiert beispielsweise, wenn Bundle 1 gestoppt und aus dem OSGi Framework deinstalliert wird.

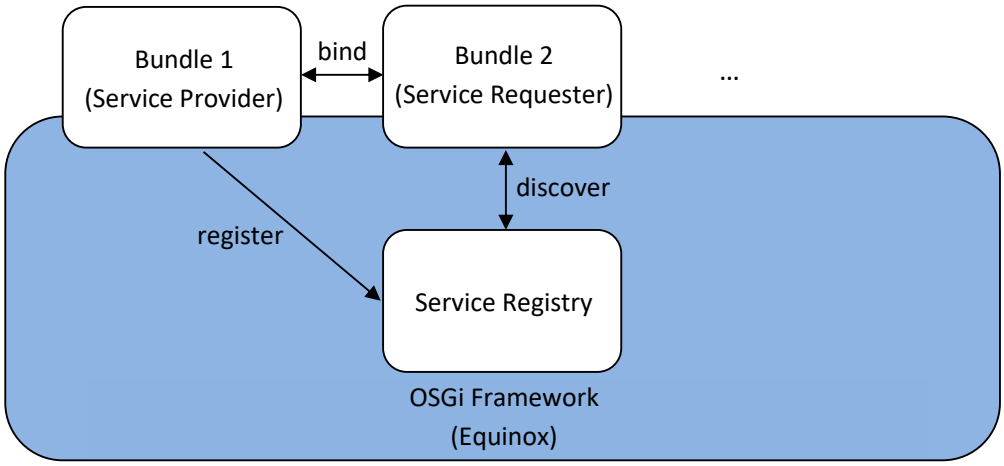

Abbildung 2.4.: OSGi Framework Service Discovery (angelehnt an [LNH03])

2.4. Apache Camel

Apache Camel ist ein Open Source Integrationsframework, das die Integration verschiedener Systeme erleichtern soll [IA18]. Die Grundfunktionen von Camel sind das Routing und die Konvertierung von Nachrichten. Camel bietet dabei eine Vielzahl an verschiedenen unterstützten APIs und Protokollen. Außerdem ist das Komponentenmodell von Camel leicht erweiterbar, wodurch neue Protokolle zum Framework hinzugefügt werden können. Damit ermöglicht es Camel, Systeme und Services mit völlig unterschiedlichen verwendeten Protokollen und Datenformaten auf eine einheitliche Art und Weise zu integrieren. Eine Besonderheit von Camel ist, dass nahezu alle, der von Gregor Hohpe und Boby Woolf in ihrem Buch „*Enterprise Integration Patterns*“ [HW04] beschriebenen Pattern, wie zum Beispiel „*Recipient List*“, „*Content-Based Router*“ oder „*Aggregator*“, direkt umgesetzt wurden. Da diese sehr häufig für die Integration mehrerer Systeme benötigt werden, müssen sie also nicht selbst implementiert werden, sondern können direkt mittels eines Camel Befehls verwendet werden [IA18].

Im Folgenden werden die wichtigsten Komponenten, Konzepte und Begriffe von Camel kurz beschrieben und voneinander abgegrenzt.

Exchange: Die grundlegende Einheit, die in Camel über Daten verfügt und die zur Kommunikation zwischen Systemen verwendet wird, bezeichnet man als *Message* [IA18]. Eine Camel Message besteht aus einer Menge an *Headern* und einem *Body*. Ein Header ist ein Key/Value-Paar, das Metadaten über die Message enthält. Der Body einer Message ist dagegen vom Typ *java.lang.Object* und kann somit ein beliebiges Java Objekt umfassen. Zum Austausch von Messages und zum Routing werden in Camel sogenannte *Exchanges* verwendet. Die vereinfachte Struktur eines Camel Exchanges ist in Abbildung 2.5 dargestellt. Jedes Exchange besteht aus einer *Exchange ID*, die das Exchange eindeutig identifiziert und einer Menge an *Properties*. Properties beinhalten, wie Header auf Message Ebene, Metadaten über das Exchange. Außerdem wird für jedes Exchange ein *Message Exchange Pattern (MEP)* ausgewählt. Das MEP definiert, ob es sich um Kommunikation in nur eine Richtung oder eine Request/Reply Interaktion handelt. Je nach MEP verfügt das Exchange nur über eine *In Message* oder zusätzlich über eine *Out Message*. Für den Fall, dass während des Routings ein Fehler auftritt, enthalten Exchanges ein *Exception* Feld, aus welchem Fehler ausgelesen werden können.

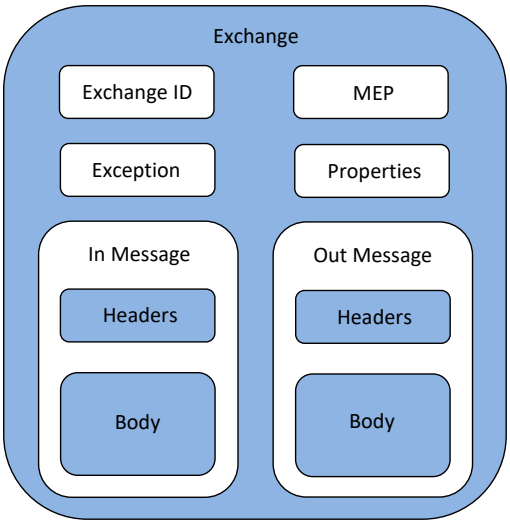

Abbildung 2.5.: Vereinfachte Struktur eines Camel Exchange Objekts (nach [IA18])

Endpoint: Die Endpunkte eines Kommunikationskanals heißen bei Camel *Endpoints* [IA18]. Endpoints können zum Senden und Empfangen von Daten bzw. Exchanges verwendet werden. Jeder Endpoint wird in Camel durch einen *Uniform Resource Identifier (URI)* identifiziert und konfiguriert. Eine URI besteht dabei aus einem *Schema*, einem *Kontextpfad* und aus *Optionen*. Durch das Schema wird definiert, welche Art von Endpoint durch die URI identifiziert wird. Verschiedene Endpoint Arten sind zum Beispiel FTP und MQTT Endpoints. Der Kontextpfad spezifiziert den Kontext innerhalb einer Art von Endpoints. Für einen FTP Endpoint kann dies zum Beispiel der Hostname und Port eines FTP-Servers sein. Optionen können zur weiteren Konfiguration von Endpoints verwendet werden.

Listing 2.1 Beispiel: Camel Endpoint

```
1 mqtt:endpoint?host=tcp://192.168.1.5&subscribeTopicName=test.topic
```

Listing 2.1 beschreibt einen Camel MQTT Endpoint. Dies wird durch das Schema der URI definiert (hier „mqtt“). Beim Kontextpfad (hier „endpoint“) sind MQTT Endpoints ein Spezialfall, da dieser lediglich ein Name ist und keine semantische Bedeutung hat. Die Optionen (hier „host=tcp://192.168.1.5&subscribeTopicName=test.topic“) dagegen definieren, zu welchem MQTT Broker eine Verbindung aufgebaut und auf welches Topic eine Subscription durchgeführt werden soll. Der Endpoint kann also genutzt werden, um alle Nachrichten, die beim MQTT Broker unter der Adresse 192.168.1.5 und dem Topic „test.topic“ veröffentlicht werden, zu empfangen.

Component: *Components* werden in Camel verwendet, um Endpoints zu erzeugen [IA18]. Jede Component ist mit einem eindeutigen Namen assoziiert. Dieser Name wird als Schema in URIs von Endpoints verwendet, um die korrekte Component zu identifizieren, die zur Erzeugung des Endpoints benötigt wird. Das Schema „mqtt“ in der URI eines Endpoints würde also bedeuten, dass die MQTT Component von Camel zur Erzeugung des Endpoints verwendet werden soll. Camel erlaubt es, neue Components zu erstellen, wodurch neue Arten von Endpoints erzeugt werden können, falls diese noch nicht unterstützt werden.

Route: Camel *Routes* können genutzt werden, um Routing- und Konvertierungsregeln für Nachrichten bzw. Exchanges zu definieren [IA18]. Jede Route startet mit einem Endpoint, von welchem Daten konsumiert und als Exchanges in die Route eingefügt werden. Anschließend durchlaufen die Exchanges, je nach Anwendung, eine Reihe von Verarbeitungsschritten. Diese Schritte werden als *Processor* definiert und im nächsten Abschnitt vorgestellt.

Listing 2.2 Beispiel: Camel Route in Java DSL

```
1 from("mqtt:endpoint?host=tcp://192.168.1.5&subscribeTopicName=test.topic")
2   .to("jms:test.queue");
```

Listing 2.2 zeigt eine simple Camel Route mit lediglich einem Verarbeitungsschritt. Die Route konsumiert Nachrichten vom bereits beschriebenen MQTT Endpoint und sendet diese an einen JMS Endpoint. Der JMS Endpoint veröffentlicht die Nachrichten in der JMS Queue mit dem Namen „test.queue“. Zur Definition der Route in Listing 2.2 wurde die Camel *Java DSL* verwendet. Camel bietet jedoch weitere DSLs, wie zum Beispiel, die *XML DSL*, mit der die gleiche Route definiert werden kann.

Processor: Ein Camel *Processor* ist ein Knoten, der einen Verarbeitungsschritt in einer Route durchführt [IA18]. Ein Processor kann beispielsweise eine Nachricht von einem Endpoint konsumieren, eine Nachricht konvertieren oder diese an einen Endpoint senden. Camel beinhaltet Processors für die verschiedenen *Enterprise Integration Patterns*. Außerdem ist es aber auch möglich, eigene Processors zu erstellen und zu verwenden. Eine Camel

Route kann als Graph von Processors gesehen werden, wobei immer die *Out Message* des vorherigen Processors als *In Message* des nachfolgenden Processors verwendet wird. Abbildung 2.6 stellt eine Camel Route grafisch dar. Der erste Processor einer Route hat eine spezielle Rolle und wird als *Consumer* bezeichnet. Dieser empfängt die Daten von einem Endpoint und sendet sie als Exchange an den zweiten Processor (siehe „from“ im Beispiel der Java DSL). Alle weiteren Processors können das Exchange verändern, es an einen Endpoint senden oder andere Events durch den Empfang des Exchanges auslösen. Sobald das Exchange den letzten Processor einer Route verlässt, entscheidet das MEP, ob das bearbeitete Exchange als Antwort zurückgesendet wird oder keine Antwort nötig ist.

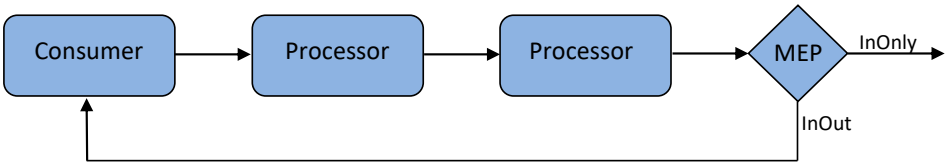

Abbildung 2.6.: Camel Route mit Processor Komponenten (nach [IA18])

2.5. Cloud Computing

Cloud Computing ist ein Paradigma, das ein weiterhin wachsendes Interesse in der Industrie erfährt [KPM]. Nach den Daten des *Bitcom Cloud Monitors 2018* [KPM] setzen bereits zwei von drei Unternehmen in Deutschland Cloud Computing ein und über die Hälfte derer, die es noch nicht verwenden, planen den Einsatz in der Zukunft. Dabei profitieren die Unternehmen vor allem von einer flexibleren IT-Infrastruktur und der Möglichkeit, Anwendungen unkomplizierter und schneller zu skalieren.

Für den Begriff Cloud Computing gibt es zahlreiche unterschiedliche Definitionen [VRCL08]. Eine in Wissenschaft und Industrie weitgehend akzeptierte Definition für Cloud Computing wurde von der US-Standardisierungsbehörde *National Institute of Standards and Technology (NIST)* verfasst [MG+11]. Die NIST Definition beinhaltet *fünf grundlegende Charakteristiken*, *drei Service Modelle* und *vier Deployment Modelle*. Sowohl die Service Modelle als auch die Deployment Modelle sind für das grundsätzliche Verständnis von Cloud Computing weniger relevant und werden deshalb nicht genauer betrachtet. Weitere Informationen dazu finden sich in der NIST Definition [MG+11].

Nachfolgend werden die fünf grundlegenden Charakteristiken von Cloud Computing nach der NIST Definition vorgestellt [MG+11]:

- *On-demand self-service*: Nutzer können Cloud Ressourcen, wie zum Beispiel Server, Speicher oder Services, auf Abruf automatisiert provisionieren. Dabei ist keine menschliche Interaktion auf Seiten des Cloud Providers nötig.

- *Broad network access*: Der Zugriff auf Cloud Ressourcen geschieht über das Netzwerk und mithilfe üblicher Protokolle.
- *Resource pooling*: Cloud Provider bieten ihre IT Ressourcen einer großen Anzahl an Nutzern an. Damit können sie Schwankungen im Ressourcenbedarf einzelner Nutzer ausgleichen und die Hardware effizient ausnutzen.
- *Rapid elasticity*: Ressourcen können elastisch provisioniert und freigegeben werden. Somit können Anwendungen sehr schnell bei steigender oder sinkender Last skaliert werden. Für den Nutzer erscheinen die vorhandenen Cloud Ressourcen damit häufig als unbegrenzt.
- *Measured service*: Die Nutzung von Cloud Ressourcen wird vom Cloud System aufgezeichnet. Damit hat sowohl der Provider als auch der Nutzer einen genauen Überblick über die verwendeten Ressourcen und die Dauer der Nutzung. Häufig wird die Nutzung in einem *pay-per-use* Modell abgerechnet, wobei der Nutzer pro Ressource und pro Zeiteinheit bezahlt.

Das Cloud Computing Paradigma kann mit dem Paradigma des Internet of Things kombiniert werden [SBH+17]. Damit wird es ermöglicht, IoT Umgebungen skalierbar und interoperabel zu gestalten. Eine Möglichkeit ist zum Beispiel, sowohl die IoT Middleware, als auch die IoT Anwendung in der Cloud auszuführen. Somit können diese sehr schnell und günstig provisioniert werden. Um die Provisionierung kompletter IoT Umgebungen zu automatisieren, kann der TOSCA Standard verwendet werden, der im folgenden Abschnitt vorgestellt wird.

2.6. TOSCA

Die *Topology and Orchestration Specification for Cloud Applications (TOSCA)* ist ein OASIS-Standard für die Beschreibung von Cloud Anwendungen [OASd]. TOSCA bietet eine XML- bzw. YAML-basierte Sprache, mit der Anwendungen unabhängig von bestimmten Technologien oder Anbietern definiert werden können. Damit wird eine hohe Portabilität und Interoperabilität der Anwendungen erreicht und ein sogenannter *Vendor-Lock-In* vermieden [BBK+14]. Unter einem Vendor-Lock-In versteht man die Abhängigkeit von einem bestimmten Anbieter und dessen proprietären Technologien bzw. APIs, wodurch ein Wechsel des Anbieters sehr teuer und zeitaufwendig ist. Auch wenn TOSCA für die Beschreibung von Cloud Anwendungen entwickelt wurde, kann es ebenfalls genutzt werden, um IoT Anwendungen bzw. Umgebungen zu modellieren [SBH+17]. Da der TOSCA-Standard sehr umfangreich ist, werden in diesem Abschnitt lediglich die Grundlagen von TOSCA und für diese Arbeit besonders relevante Teile des Standards beschrieben. Weitere Informationen zu TOSCA finden sich zum Beispiel in der TOSCA Spezifikation [OASd] oder anderen Veröffentlichungen zu TOSCA [BBLS12] [BBK+14].

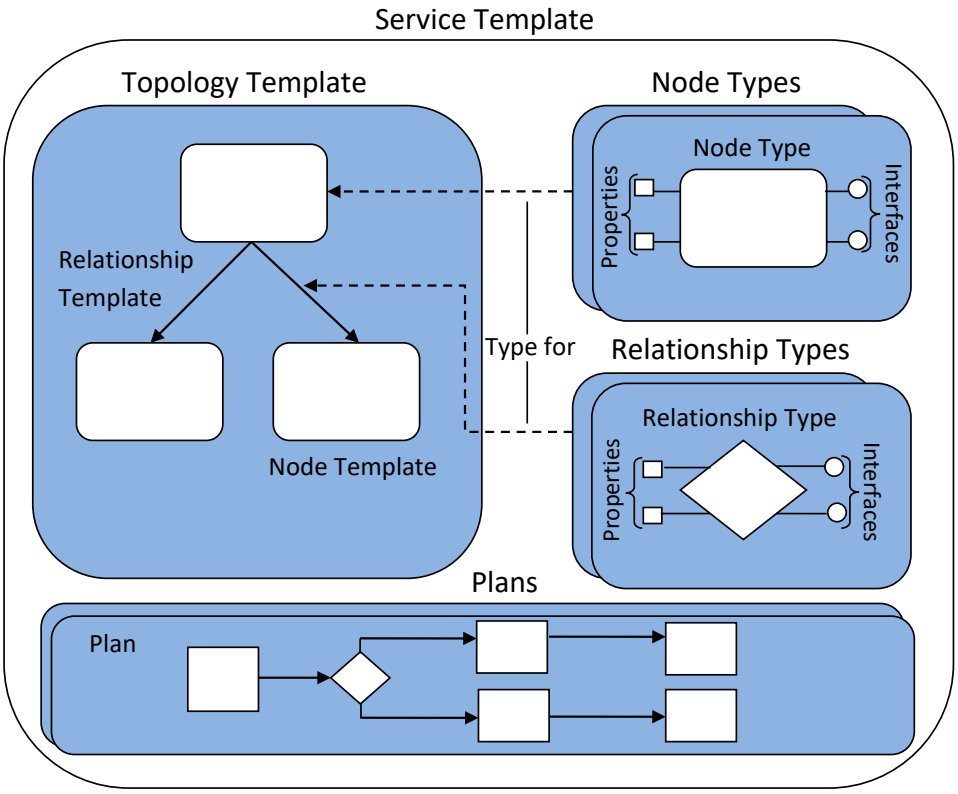

Abbildung 2.7.: Inhalt und Aufbau eines Service Templates (nach [OASd])

Alle für einen Cloud-Service benötigten Informationen werden bei TOSCA in einem sogenannten *Service Template* definiert [OASd]. Die Struktur eines Service Templates ist in Abbildung 2.7 dargestellt. Ein Service Template besteht aus zwei Hauptbestandteilen: Einem *Topology Template* und einer Menge an *Plänen*. Außerdem beinhaltet es jeweils eine Menge an *Node Types* und *Relationship Types*. Das Topology Template beschreibt die Topology, also die Struktur bzw. den Aufbau der modellierten Anwendung [BBK+14]. Es ist ein gerichteter Graph, bestehend aus *Node Templates* (Knoten) und *Relationship Templates* (Kanten). Node Templates repräsentieren dabei die Komponenten der Anwendung. Ein Node Template kann also zum Beispiel ein Betriebssystem oder einen Web Server darstellen. Relationship Templates repräsentieren dagegen Beziehungen zwischen Komponenten bzw. Node Templates und können zum Beispiel definieren, dass eine Komponente auf der anderen gehostet wird oder zwei Komponenten eine Verbindung zueinander aufbauen.

Jedes Node und Relationship Template eines Topology Templates stammt von einem abstrakten Typ ab, einem *Node Type* bzw. *Relationship Type* [OASd]. Dieses Typsystem wird in TOSCA verwendet, um die Wiederverwendbarkeit von Komponenten zu erhöhen. Node Types beschreiben die *Properties* und *Interfaces* der zugehörigen Node Templates. Ein Interface definiert dabei eine Menge an Operationen zusammen mit ihren Eingabe- und Ausgabeparametern. Properties können bei einem Node Type für ein Betriebssystem

zum Beispiel der Benutzername und das Passwort für den Zugriff sein. Ein Interface dieses Typs könnte dagegen Operationen definieren, die es ermöglichen, ein Skript auf dem Betriebssystem auszuführen oder ein bestimmtes Programm zu installieren. Relationship Types definieren ebenfalls die Properties und Interfaces von Relationship Templates. Die Properties können dabei zum Beispiel Informationen beinhalten, die für die Herstellung einer Verbindung zwischen zwei Node Templates benötigt werden.

Die vierte Komponente eines Service Templates, neben dem Topology Template, den Node Types und den Relationship Types, sind die *Pläne* [BBLS12]. Pläne werden genutzt, um die Managementaspekte einer Anwendung zu beschreiben. Zu diesen Managementaspekten gehört zum Beispiel das Instanziieren einer Anwendung, das Skalieren einer Komponente der Anwendung zur Laufzeit oder das Terminieren einer Instanz der Anwendung. TOSCA führt dazu keinen neuen Standard für Pläne ein, sondern erlaubt die Nutzung bestehender Standards wie *BPMN* oder *BPEL*, um Pläne als Workflow Modelle zu beschreiben [OASd]. Pläne können die von den Node und Relationship Types definierten Operationen nutzen und diese auf den konkreten Node Templates bzw. Relationship Templates aufrufen.

Die Node und Relationship Types liefern zwar die Definition von Operationen mitsamt ihren Parametern, eine konkrete Implementierung aber beinhalten sie nicht [OASd]. Der TOSCA Standard definiert dafür *Artifact Types*, die wiederverwendbare Typen für die verschiedenen Software-Artefakte innerhalb einer Topologie darstellen. Unterschieden wird bei TOSCA zwischen *Implementation Artifacts (IAs)* und *Deployment Artifacts (DAs)* [LVCD13]. Implementation Artifacts dienen als Implementierung der Operationen von Node bzw. Relationship Types. So kann beispielsweise ein Implementation Artifact vom Artifact Type WAR die Operationen eines Betriebssystem Node Types implementieren. Die WAR Datei würde dann unter anderem eine Operation bereitstellen, bei der eine SSH Verbindung zum Betriebssystem hergestellt wird, um anschließend ein übergebenes Skript auszuführen. Zur Ausführung von Implementation Artifacts wird üblicherweise eine externe Management Infrastruktur benötigt [BBLS12]. Deployment Artifacts implementieren dagegen die Anwendungsfunktionalität und können an Node Templates angehängt werden. Sie werden benötigt, um Node Templates zu instanziieren [BBK+14]. Ein Node Template vom Typ Web Anwendung kann zum Beispiel ein Deployment Artifact angehängt haben, das die PHP Dateien der Web Anwendung beinhaltet, die zur Instanziierung auf den unterliegenden Web Server übertragen werden müssen.

Um den Austausch von TOSCA Definitionen zu ermöglichen, werden sogenannte *Cloud Service Archives (CSARs)* verwendet [OASd]. Eine CSAR ist eine ZIP Datei, die Topology Templates, Artefakte, Typen, Pläne und alle sonstigen benötigten Dateien beinhaltet [BBK+14]. CSARs sind *self-contained*. Das bedeutet, sie umfassen alle Dateien, die für das Deployment einer Anwendung benötigt werden und haben keine externen Abhängigkeiten. Durch die Standardisierung sind CSARs portabel und erlauben es, die enthaltene Anwendung auf jeder beliebigen TOSCA-Runtime zu provisionieren. Ein Beispiel für eine solche TOSCA-Runtime ist das Open Source Projekt *OpenTOSCA Container* (siehe Abschnitt 2.7).

Für das Deployment von modellierten Anwendungen bietet TOSCA zwei verschiedene Ansätze: Den *imperativen* und den *deklarativen Ansatz* [BBK+14]. Für den imperativen Ansatz werden sowohl die Topologie der Anwendung modelliert, als auch alle benötigten Management Pläne erstellt und in die entstehende CSAR eingefügt [BKLW17]. Die TOSCA-Runtime führt bei diesem Ansatz die inkludierten Pläne aus und interpretiert die Topologie nicht. Der imperative Ansatz hat den Vorteil, dass damit beliebige Deployment Logik ausgeführt werden kann und deshalb auch das Deployment von sehr komplexen Anwendungen möglich ist. Dafür wird von den Anwendungsentwicklern jedoch ein sehr großes Expertenwissen benötigt. Außerdem ist die Entwicklung fehleranfällig und zeitintensiv. Beim deklarativen Ansatz dagegen beinhaltet die CSAR keine erstellten Management Pläne [BKLW17]. Stattdessen interpretiert die TOSCA-Runtime das bereitgestellte Topologie Template. Diese Interpretation basiert auf von TOSCA definierten Konstrukten, wie zum Beispiel dem standardisierten *Lifecycle Interface* (siehe [OASc]). Durch den deklarativen Ansatz wird die Anwendungsentwicklung stark vereinfacht und kann eventuell sogar nur mittels eines grafischen Modellierungswerkzeugs, wie zum Beispiel *Winery* (siehe Abschnitt 2.7), durchgeführt werden. Der Nachteil des Ansatzes ist, dass der Deployment Prozess damit nicht beliebig angepasst werden kann.

2.7. OpenTOSCA

OpenTOSCA[2] ist ein Ökosystem für die Cloud-Beschreibungssprache TOSCA, das an der Universität Stuttgart entwickelt wurde und sowohl ein Modellierungswerkzeug für TOSCA, als auch eine TOSCA-Runtime beinhaltet [BBH+13]. In diesem Abschnitt werden die verschiedenen Komponenten des Ökosystems aufgelistet und es wird kurz die jeweilige Funktion erläutert. Genauer betrachtet wird dabei die Architektur und die Funktionalität der TOSCA-Runtime *OpenTOSCA Container*, da diese im Zuge dieser Arbeit erweitert wird.

Winery: Bei *Winery* handelt es sich um ein Modellierungswerkzeug für TOSCA [KBBL13]. Winery kann genutzt werden, um verschiedene TOSCA Elemente mittels einer Web Oberfläche zu erstellen, anstatt diese direkt in XML-Syntax definieren zu müssen. Zu diesen Elementen gehören beispielsweise Node Types und Relationship Types. Die erstellten TOSCA Elemente können anschließend in einem *Topology Modeler* mittels drag-and-drop zu einem Topology Template kombiniert werden, das die Topologie eines Service Templates definiert. Damit wird das Modellieren einer Anwendung stark vereinfacht und beschleunigt. Nach Beendigung der Modellierung erlaubt es Winery, modellierte Service Templates als CSARs zu exportieren. Diese CSARs können anschließend mittels einer TOSCA-Runtime automatisch provisioniert werden.

Plan Engine: Um Pläne, die in einer CSAR vorhanden sind oder basierend auf einer Topologie erzeugt werden, aufrufen zu können, müssen diese in einer Laufzeitumgebung für Pläne bereitgestellt werden [BBH+13]. Da der OpenTOSCA Container (siehe unten) derzeit

[2] http://www.iaas.uni-stuttgart.de/OpenTOSCA/

nur BPEL Pläne unterstützt, wird lediglich eine Laufzeitumgebung für BPEL Pläne benötigt [Ope]. Eine solche Laufzeitumgebung für BPEL Pläne ist zum Beispiel Apache Ode[3]. Im Kontext von OpenTOSCA wird die Plan Laufzeitumgebung *Plan Engine* genannt.

Implementation Artifact Engine: Die *Implementation Artifact Engine* bzw. *IA Engine* wird verwendet, um Implementation Artifacts (IAs) einer CSAR auszuführen [BBH+13]. IAs verfügen über die Management Logik für Node bzw. Relationship Types und können von Plänen aufgerufen werden. Je nach Art der IAs wird eine andere Laufzeitumgebung als IA Engine benötigt. Im OpenTOSCA Ökosystem werden derzeit die IA Arten *WAR* und *Skript* unterstützt [Ope]. Zu Skript Artefakten zählen dabei verschiedene Technologien, wie zum Beispiel *Ansible*, *Chef* oder *Shell Scripts*. Da Skript Artefakte direkt innerhalb einer Topologie ausgeführt werden müssen, benötigen diese keine externe Laufzeitumgebung. Für WAR Artefakte wird eine solche Laufzeitumgebung jedoch benötigt, weshalb ein speziell konfigurierter Tomcat[4] Server, als IA Engine, Teil des OpenTOSCA Ökosystems ist.

OpenTOSCA UI: Die *OpenTOSCA UI* ist das User Interface des OpenTOSCA Ökosystems [Ope]. Durch die UI kann zum Beispiel eine CSAR in den OpenTOSCA Container geladen oder eine Instanz einer Anwendung erzeugt werden. Intern verwendet die UI die REST API des OpenTOSCA Containers und der Winery, um die vom Nutzer ausgewählten Aktionen auszuführen.

OpenTOSCA Container: Der *OpenTOSCA Container* ist eine Open Source TOSCA-Runtime [BBH+13]. Die Architektur des Containers ist in Abbildung 2.8 dargestellt. Zusätzlich sind die anderen Komponenten des OpenTOSCA Ökosystems, mit denen der Container interagieren kann, oberhalb des Containers abgebildet.

Am unteren Rand des Architekturbildes sind die Datenbanken des Containers als Zylinder dargestellt. Dort werden unter anderem die *Modelle*, *Endpunkte* und *Instanzdaten* abgespeichert [BBH+13]. Unter den Modellen versteht man dabei die TOSCA Definitionen, die aus hochgeladenen CSARs extrahiert werden. Endpunkte werden für Pläne und Implementation Artifacts gespeichert, die in der Plan Engine bzw. IA Engine deployt wurden und somit über den Endpunkt aufrufbar sind. In den Instanzdaten wird zum einen abgelegt, welche Instanzen derzeit im Container ausgeführt werden bzw. erzeugt wurden [OASe]. Zum anderen werden dort auch Properties der Instanzen gespeichert und aktualisiert. Eine solche Property kann zum Beispiel die IP-Adresse einer virtuellen Maschine sein, die beim Erstellen der Instanz erzeugt wurde und anfangs nicht bekannt war.

Die unterschiedlichen Komponenten des OpenTOSCA Containers sind als Rechtecke mit abgerundeten Ecken abgebildet [BBH+13] [Zim16]. Die *Control-Komponente* ist dabei für die Steuerung aller Abläufe im Container und für die Koordination aller anderen Komponenten zuständig. In der *Core-Komponente* befinden sich die Logik zur Verarbeitung von TOSCA Definitionen und die Schnittstellen zu den Datenbanken, über welche die anderen Komponenten auf die Daten zugreifen können. Der *Plan Builder* sorgt dafür,

[3] http://ode.apache.org/
[4] http://tomcat.apache.org/

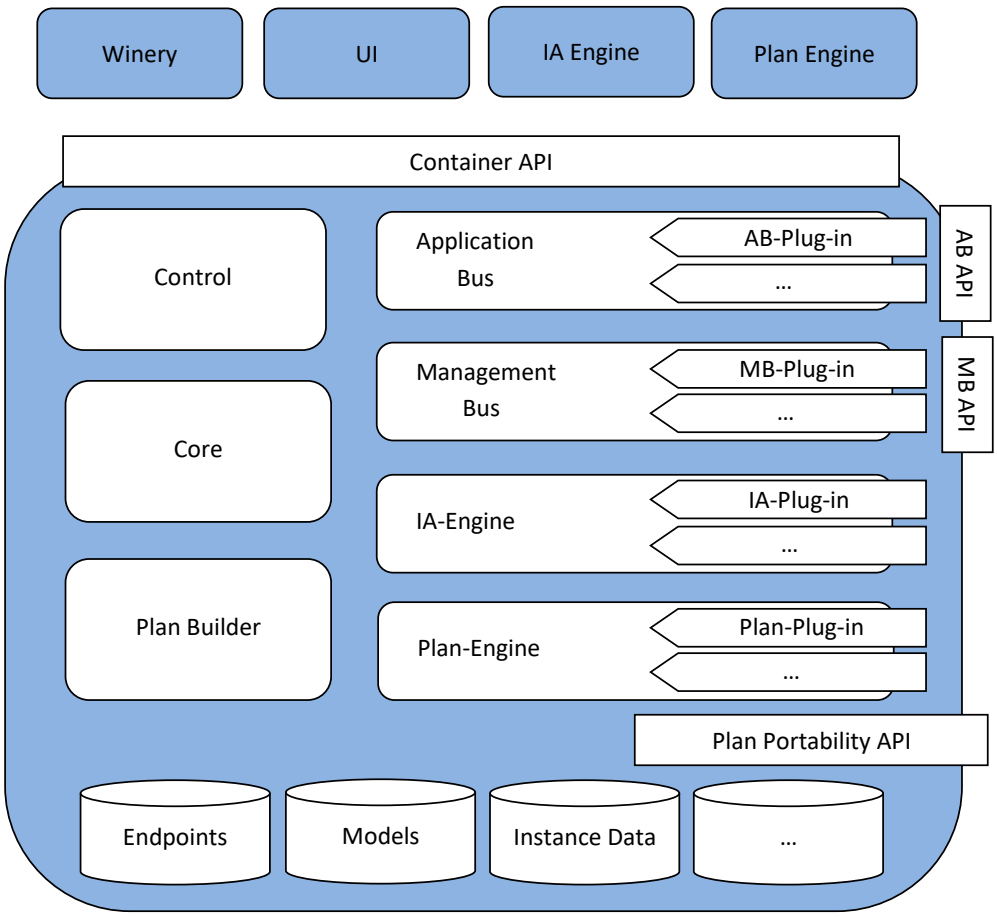

Abbildung 2.8.: OpenTOSCA Architektur (nach [Zim16] und [BBH+13])

dass der OpenTOSCA Container, neben einer imperativen TOSCA-Runtime, auch eine deklarative Runtime ist [Kép13]. Falls im Container eine CSAR hochgeladen wird, die über keine Pläne verfügt, wird der Plan Builder aktiviert. Dieser erzeugt daraufhin anhand der Topology Templates in der CSAR die Management Pläne.

Die vier weiteren Komponenten des Containers sind der *Application Bus*, der *Management Bus*, die *IA-Engine* und die *Plan-Engine* [Zim16] [Ope]. Diese sind jeweils mit einem Plug-In System (Pfeilform) ausgestattet, um die unkomplizierte Erweiterung zu ermöglichen. Die *IA-Engine* und die *Plan-Engine* Komponenten des Containers sind dabei nicht mit den externen Komponenten IA Engine und Plan Engine des OpenTOSCA Ökosystems (siehe oben) zu verwechseln. Die Komponenten des Containers beinhalten jeweils die Logik, um mit den externen Komponenten zu kommunizieren. Die IA-Engine Komponente verfügt also zum Beispiel in einem Plug-In über die Funktionalität, um auf einem Tomcat Server WAR Artefakte zu deployen. In der Plan-Engine Komponente wird dagegen ein Plug-In verwendet, das das Verwalten von BPEL Plänen auf Apache Ode ermöglicht. Durch das Plug-In System können leicht neue Arten von Plänen bzw. IAs im Container hinzugefügt werden.

Der *Management Bus* bietet eine einheitliche Schnittstelle zum Aufruf von IAs und Plänen, die in der IA bzw. Plan Engine deployt sind [Zim16] [Ope]. Dafür hat er Plug-Ins für verschiedene Aufrufarten. Es existiert also zum Beispiel jeweils ein Plug-In um *REST* und *SOAP* Services aufrufen zu können. Die Aufgabe des *Application Bus* ist es dagegen, die Kommunikation zwischen verschiedenen Anwendungen, die vom OpenTOSCA Container verwaltet werden, zu ermöglichen.

Neben den erläuterten Komponenten enthält das Architekturbild in Abbildung 2.8 noch die verschiedenen APIs des OpenTOSCA Containers, die als Rechtecke dargestellt sind [BBH+13]. Die *Container API* ist das zentrale REST Interface des Containers, mit welchem über die Control-Komponente verschiedene Aktionen im Container ausgelöst werden können. Die Container API wird beispielsweise von der OpenTOSCA UI verwendet. Außerdem besitzen sowohl der Management Bus, als auch der Application Bus jeweils eine API, über die die Komponenten direkt aufgerufen werden können. Die *Plan Portability API* kann von Plänen verwendet werden, um auf die Topologie und Instanzdaten der zugehörigen Instanz zuzugreifen [BBH+13].

2.8. Docker

Bei der Open Source Software *Docker* handelt es sich um eine Plattform, die zur Entwicklung und Ausführung von Anwendungen verwendet werden kann [Tur14]. Docker basiert auf der Technik der *Containervirtualisierung*. Dabei ist die Containervirtualisierung eine leichtgewichtige Alternative zur *Hypervisor-basierten Virtualisierung*, wie sie zum Beispiel in der Cloud häufig eingesetzt wird [CMF+16]. Bei der Hypervisor-basierten Virtualisierung wird auf einem Host Betriebssystem ein *Virtual Machine Monitor (VMM)* installiert. Dieser ermöglicht es, mehrere virtuelle Maschinen mit komplett unabhängigem Betriebssystem auf der zugrunde liegenden Hardware auszuführen. Die Containervirtualisierung verwendet dagegen das Host Betriebssystem und erzeugt lediglich für jeden Container einen eigenen Userspace. Container haben im Vergleich zu virtuellen Maschinen damit den Vorteil, weniger Ressourcen zu benötigen, da nur ein Betriebssystem verwendet wird. Außerdem können Container deutlich schneller gestartet werden als virtuelle Maschinen. Die Nachteile, die dafür in Kauf genommen werden müssen, sind eine geringere Isolation und damit mehr potentielle Sicherheitslücken, sowie ein schwierigerer Zugriff auf physische Hardware, wie zum Beispiel Sensoren [CMF+16].

Abbildung 2.9 zeigt die Architektur der Docker Plattform. Die drei Hauptbestandteile sind der *Docker Client*, der *Docker Host* und die *Docker Registry* [Docb]. Der Docker Host ist der Host, auf dem die Containervirtualisierung durchgeführt werden soll [Tur14]. Er beinhaltet den *Docker Daemon* und eine Menge an *Containern* und *Images*. Jede Anwendung, die mittels Docker ausgeführt wird, ist ein eigenständiger Container, der alles beinhaltet, das zur Ausführung benötigt wird. Images sind dagegen die Baupläne von Containern und können verwendet werden, um neue Container zu instanziieren. Der Docker Daemon bietet eine API an, mit der beispielsweise neue Container erstellt, alte Container gelöscht oder

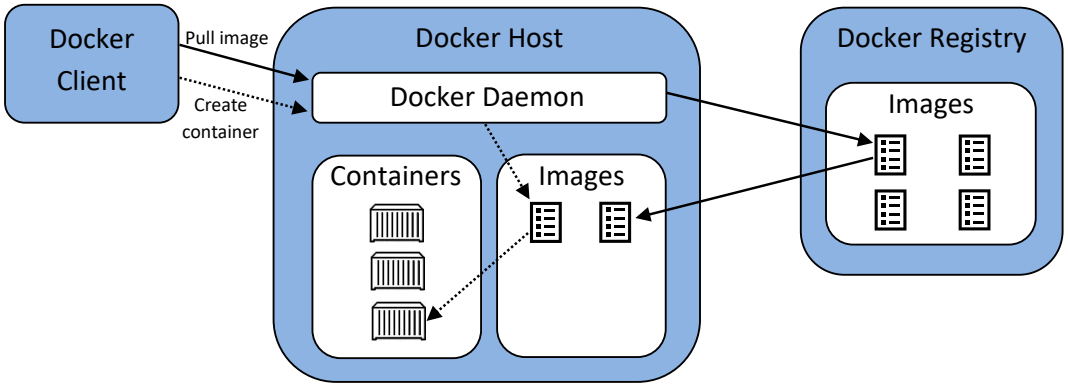

Abbildung 2.9.: Docker Plattform Architektur (angelehnt an [Docb])

neue Images heruntergeladen werden können. Docker Registries sind Speicherplätze für Docker Images und können von Docker Daemons zum Download von Images genutzt werden. Es existieren sowohl öffentliche Registries, wie die offizielle Docker Registry[5], als auch private Registries [Docb]. Mit Hilfe des Docker Clients können Nutzer mit dem Docker Daemon kommunizieren. Der Client kann lokal auf dem Docker Host vorliegen oder von einem anderen Host aus zur Steuerung des Docker Daemons verwendet werden.

Da viele Anwendungen zu komplex sind, um in einem einzelnen Docker Container ausgeführt werden zu können, werden diese üblicherweise in mehrere Images bzw. Container unterteilt und somit modularisiert [Tur14]. Mit Docker ist das Starten solcher Anwendungen jedoch eine komplexe Aufgabe, da jeder benötigte Container einzeln hochgefahren werden muss. Dabei bestehen eventuell Abhängigkeiten, wodurch die Container in einer bestimmten Reihenfolge gestartet werden müssen. *Docker Compose* ist ein Werkzeug, das es erlaubt, mehrere Docker Container in einer einzelnen Datei zu definieren und zu konfigurieren [Doca]. Anschließend können mit nur einem Docker Compose Befehl alle Container gestartet werden. Da in der Docker Compose Datei auch Abhängigkeiten definiert werden können, ist das Starten komplexer Anwendungen damit problemlos und schnell möglich. Mit Docker Compose ist es zum Beispiel realisierbar, das komplette OpenTOSCA Ökosystem (siehe Abschnitt 2.7) mit nur einer Docker Compose Datei zu definieren und zu starten.

[5] https://hub.docker.com/

3. Verwandte Arbeiten

Dieses Kapitel stellt verwandte Arbeiten aus den Bereichen des verteilten Anwendungs- und Netzwerkmanagement vor. Außerdem werden Arbeiten betrachtet, die die Verbindung zwischen dem IoT und *Fog Computing* erforschen. Die Idee des Fog Computing ist, Services ähnlich wie in der Cloud anzubieten, diese aber näher zum Endnutzer zu bringen und somit beispielsweise die Latenz zu verringern [CZ16]. Damit sind Ressourcen und Services stärker über verschiedene, teilweise private, Netzwerke verteilt als in der Cloud. Die in dieser Arbeit untersuchten Probleme der verteilten Provisionierung müssen diesbezüglich ebenfalls gelöst werden.

Balter et al. stellen in ihrer Arbeit [BBB+09] das Konzept der *Olan Configuration Language (OCL)* vor. Bei OCL handelt es sich um eine Beschreibungssprache für Software-Architekturen, die speziell für die Definition von verteilten Anwendungen entwickelt wurde. Eine Anwendung wird dabei als Menge von *Software-Komponenten* definiert, die durch Kommunikations-Komponenten, den sogenannten *Connectors*, verbunden werden. Das Ziel der Trennung von Software- und Kommunikations-Komponenten ist, eine größere Flexibilität zu erreichen, da die Kommunikation unabhängig von der eigentlichen Anwendungsfunktionalität konfiguriert werden kann. Die Connectors haben außerdem die Aufgabe, die Heterogenität der Software-Komponenten zu verbergen, indem sie zum Beispiel unterschiedliche Datenformate konvertieren können. Nachdem eine verteilte Anwendung vollständig mit OCL definiert wurde, kann die Beschreibung an den *OCL Compiler* übergeben werden (siehe Abbildung 3.1). Dieser generiert aus der Beschreibung die Klassen für die Software-Komponenten bzw. Connectors und ein *Configuration Machine Script* und speichert alle generierten Artefakte in einem verteilten Repository. Das Configuration Machine Script enthält die gesamte Logik, die für die Bereitstellung der Anwendung ausgeführt werden muss. Wenn eine Anwendungsinstanz erzeugt werden soll, wird die *Olan Configuration Machine* aufgerufen. Diese führt das Configuration Machine Script aus und nutzt dazu spezielle Libraries, die in der Lage sind, Software-Komponenten und Connectors zu instanziieren. Ein Vorteil des Konzepts ist, dass damit heterogene Software-Komponenten zu verteilten Anwendungen kombiniert und auch Legacy-Anwendungen eingebunden werden können. Dazu müssen lediglich die Libraries der Olan Configuration Machine geeignet erweitert werden. Außerdem ermöglicht OCL, eine beliebige Anzahl an Anwendungsinstanzen vollautomatisch bereitzustellen. Ein Nachteil des Konzepts ist dagegen, dass das Configuration Machine Script lediglich zentralisiert ausgeführt wird, wodurch bei verteilten IoT Anwendungen Firewall und Performance Probleme auftreten können. Außerdem beschäftigt sich OCL lediglich mit der initialen Bereitstellung von verteilten Anwendungen und nicht mit dem Management aktiver Instanzen. Im Kontext von IoT Anwendungen muss jedoch häufig eine Neukonfiguration oder ein Update von Komponenten durchgeführt werden, was OCL derzeit nicht unterstützt.

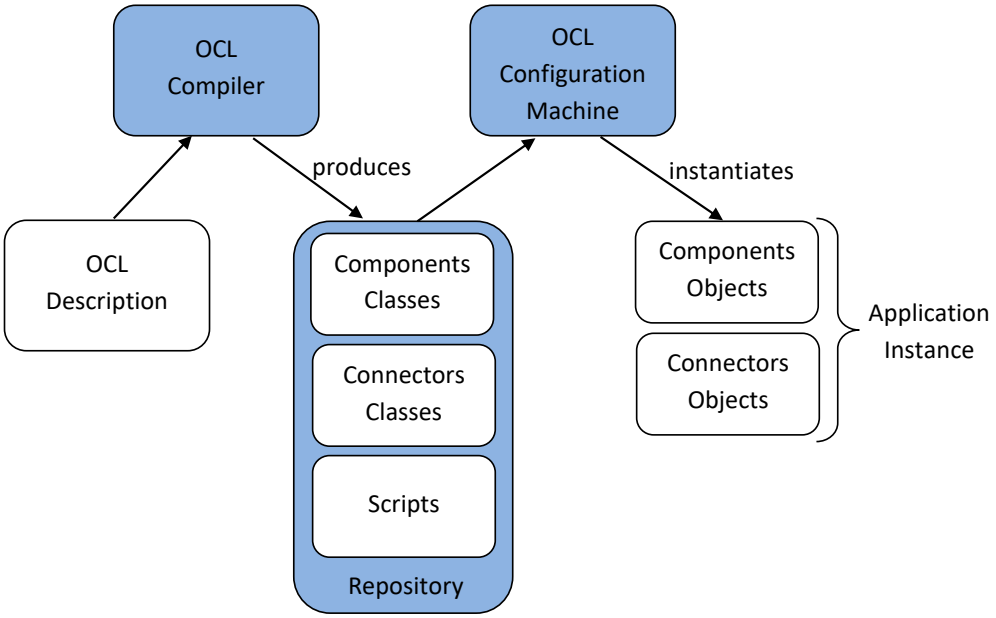

Abbildung 3.1.: Bereitstellung einer Anwendung mit OCL (nach [BBB+09])

Albrecht et al. präsentieren in ihren Arbeiten *Plush*, ein Framework für das Management verteilter Anwendungen [Alb07][ABD+07]. Plush bietet generische und einheitliche Abstraktionen, um verteilte Anwendungen in völlig unterschiedlichen Anwendungsszenarios und Umgebungen verwalten zu können. Damit hat Plush gegenüber anderen Ansätzen, die lediglich auf eine bestimmte Art von Anwendungen in einer spezifischen Umgebung abzielen, einen entscheidenden Vorteil. Das Plush Framework erlaubt es, verteilte Anwendungen zu spezifizieren, automatisch bereitzustellen, zu überwachen und zur Laufzeit zu steuern. Es deckt also den gesamten Lebenszyklus der Anwendungen, von der Definition bis zum finalen Herunterfahren, ab. Um ein verteiltes Management von Ressourcen in unterschiedlichen Netzwerken zu ermöglichen, muss das Plush Framework selbst verteilt ausgeführt werden. Dazu wird eine Client/Server Architektur verwendet, wobei bei jeder verwalteten Ressource ein Client ausgeführt werden muss. Der Plush Server, der auch als *Controller* bezeichnet wird, wird dagegen üblicherweise auf dem Computer des Nutzers gestartet. Die Aufgabe des Controllers ist, Eingaben vom Nutzer entgegenzunehmen und entsprechende Kontrollnachrichten an die Clients zu senden. Diese reagieren auf die Nachrichten, indem sie zum Beispiel eine Ressource bereitstellen oder neu konfigurieren. Um die Kommunikation des Controllers mit allen Clients zu ermöglichen, wird im Plush Framework ein Overlay Netzwerk erstellt, das alle Komponenten verbindet. Für das Overlay Netzwerk erlaubt Plush zwei unterschiedliche Topologien: Die *Sternstruktur* und die *Baumstruktur*. Abbildung 3.2 zeigt ein Beispiele für beide Arten solcher Overlay Netzwerke. Auf der linken Seite ist die Sternstruktur dargestellt, bei der alle Clients direkt mit dem Controller verbunden werden. Diese Struktur bietet die beste Performance, da Nachrichten immer nur zu einem direkten Nachbarn gesendet werden müssen. Dafür ist die Skalierbarkeit eingeschränkt, weil der Controller bei einer großen Menge an Clients

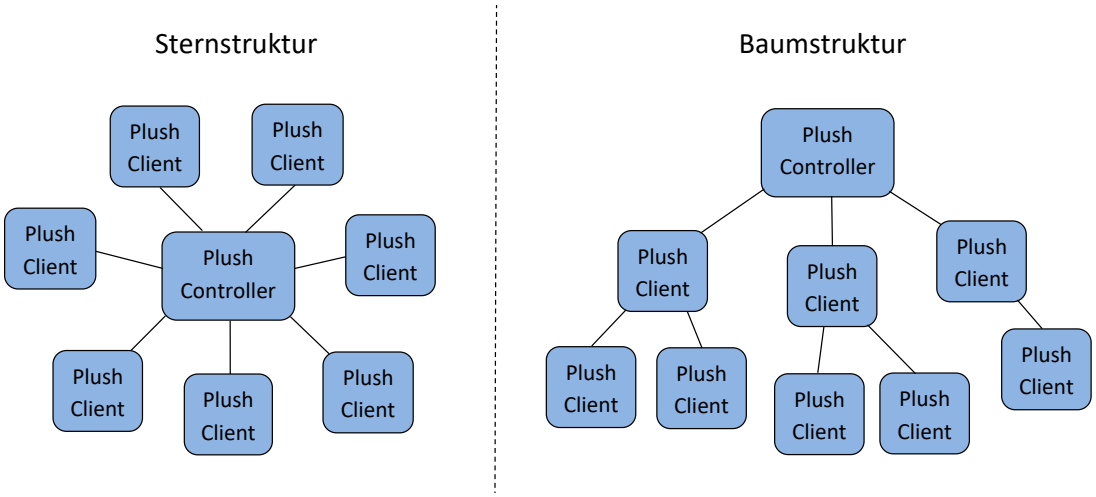

Abbildung 3.2.: Topologien des Plush Overlay Netzwerks (nach [Alb07])

zu einem Flaschenhals werden kann. Auf der rechten Seite ist dagegen die Baumstruktur abgebildet. Dort können Clients indirekt über andere Clients mit dem Controller verbunden werden. Damit kann eine bessere Skalierbarkeit erreicht werden, da der Controller entlastet wird. Die Performance verschlechtert sich jedoch, weil Nachrichten des Controllers von Clients weitergeleitet werden müssen, was zusätzliche Zeit benötigt. Die Architektur des Plush Controllers und der Plush Clients unterscheidet sich nur minimal und besteht aus drei großen Bestandteilen: Dem *User-Interface*, der *Anwendungsspezifikation* und den *Kernfunktionseinheiten*. Durch das User-Interface wird es den Nutzern von Plush erlaubt, mit den anderen beiden Bestandteilen der Architektur zu interagieren. So kann der Nutzer über das User-Interface beispielsweise eine Anwendung beschreiben und anschließend instanziieren oder die Beschreibung einer laufenden Anwendung ändern, um über den Plush Controller eine Rekonfiguration der Anwendung auszulösen. Die Definition von Anwendungen erfolgt in Plush über eine speziell entwickelte *Application Specification Language (ASL)*. Mit dieser ASL können Anwendungen in fünf verschiedene Blöcke unterteilt werden, die anschließend separat beschrieben werden. So existieren zum Beispiel Blöcke, die den Prozess innerhalb der Anwendung, den Datenfluss und die erforderlichen Komponenten beschreiben. Die Anwendungsspezifikation in der Plush Architektur wird benötigt, um aus einer Definition in der ASL die internen Objekte zu generieren, die für die Bereitstellung und Wartung der repräsentierten Anwendung verwendet werden. Deshalb besteht die Anwendungsspezifikation aus je einem Bestandteil pro Block der ASL, der das Parsing der jeweiligen Definitionen übernimmt. Die Kernfunktionseinheiten haben dagegen die Aufgabe, die Kommunikation zwischen dem Controller und den Clients zu ermöglichen. Außerdem überwachen diese die Ressourcen einer Anwendung und sind für die Koordination aller benötigten Schritte zur Bereitstellung bzw. zum Management einer Anwendung zuständig. Eine Besonderheit von Plush ist, dass es in den Kernfunktionseinheiten eine automatische Fehlererkennung und -behebung für die Infrastruktur und die Anwendung beinhaltet, womit bestimmte Fehlerarten ohne Eingriff des Nutzers behoben werden können.

Chiang et al. untersuchen in ihrer Arbeit [CZ16], wie das neue Paradigma des *Fog Computing* definiert wird und auf welche Weise es für das Internet of Things genutzt werden kann. Mit dem Erfolg des Cloud Computing Paradigmas wurden Berechnungs-, Speicher- und Netzwerkressourcen zunehmend von Cloud Providern in großen Datenzentren bereitgestellt. Durch die große Verfügbarkeit, die schnelle Skalierbarkeit und das einfache Management der Ressourcen werden die Anforderungen vieler Anwendungen nahezu ideal abgedeckt. Für Anwendungen im IoT stellen Chiang et al. allerdings fest, dass viele neue Anforderungen entstehen, die mit der Cloud schwierig zu lösen sind. So kann eine IoT Anwendung beispielsweise Daten mit Sensoren messen, diese weiterverarbeiten und aus den Resultaten eine geeignete Steuerung von Aktuatoren ableiten. Die Weiterverarbeitung der Daten könnte in die Cloud ausgelagert werden, da ein einzelnes IoT Device eventuell zu wenig Ressourcen für die Verarbeitung besitzt. Somit müssen die Daten jedoch aus der IoT Umgebung in die Cloud geladen, dort verarbeitet und anschließend wieder zur IoT Umgebung gesendet werden. Diese Übertragungen erzeugen eine große Latenz, die für einige Anwendungen nicht tolerierbar ist. Außerdem kann als Anforderung existieren, dass die Sensordaten die IoT Umgebung aus Sicherheitsgründen nicht verlassen dürfen. Eine Verarbeitung in der Cloud ist in einem solchen Fall deshalb ausgeschlossen. Stattdessen sollte die Verarbeitung nach Chiang et al. mittels Fog Computing durchgeführt werden. Als Fog Computing definieren sie dabei eine Architektur, die Ressourcen näher zum Endnutzer bringt. Im obigen Beispiel können statt der Cloud, Ressourcen anderer IoT Devices in der gleichen IoT Umgebung verwendet werden. Hierbei werden die gesamten Ressourcen aller IoT Devices einer IoT Umgebung virtualisiert und bilden gemeinsam einen Fog.

Skarlat et al. beschreiben in ihrer Arbeit [SSBL16] ein Framework, mit dem für IoT Anwendungen Ressourcen sowohl in der Cloud, als auch im Fog automatisch provisioniert werden können. Das Ziel des Frameworks ist, IoT Anwendungen in beliebigen Fog und Cloud Umgebungen möglichst optimal auszuführen. Um zu bestimmen, welche Ressourcen für eine optimale Ausführung der Anwendung verwendet werden sollten, werden verschiedene Metriken genutzt. So fließen beispielsweise die Verzögerung bei der Ausführung, die Auslastung der Ressourcen und die Kosten in die Bewertung ein. Die optimale Konfiguration kann somit durch das Lösen eines Optimierungsproblems berechnet werden. Abbildung 3.3 stellt die Architektur des Frameworks dar. Die zentrale Komponente der Architektur ist die *Cloud-Fog Control Middleware*. Diese ist für die Provisionierung aller Ressourcen in der Cloud und das Zuweisen der Aufgaben an die Ressourcen zuständig. Ressourcen in der Cloud werden dabei hauptsächlich für sehr ressourcenintensive Aufgaben, ohne strenge Anforderungen an eine minimale Verzögerung, verwendet. Außerdem koordiniert die Cloud-Fog Control Middleware auch die gesamte Fog Umgebung, die in sogenannte *Fog Colonies* unterteilt ist. Jede Fog Colony wird von einem *Fog Orchestration Control Node* verwaltet. Der Fog Orchestration Control Node ist dabei für eine Menge an *Fog Cells* innerhalb der Fog Colony zuständig. Fog Cells sind Software-Komponenten, die auf IoT Devices ausgeführt werden und als Zugriffspunkt dienen, um diese steuern zu können. Zusätzlich kann ein Fog Orchestration Control Node eine Menge anderer Fog Orchestration Control Nodes und deren Fog Colonies kontrollieren. Somit erlaubt es das Framework eine Baumstruktur aufzubauen, sodass alle Fog Colonies von der Cloud-Fog

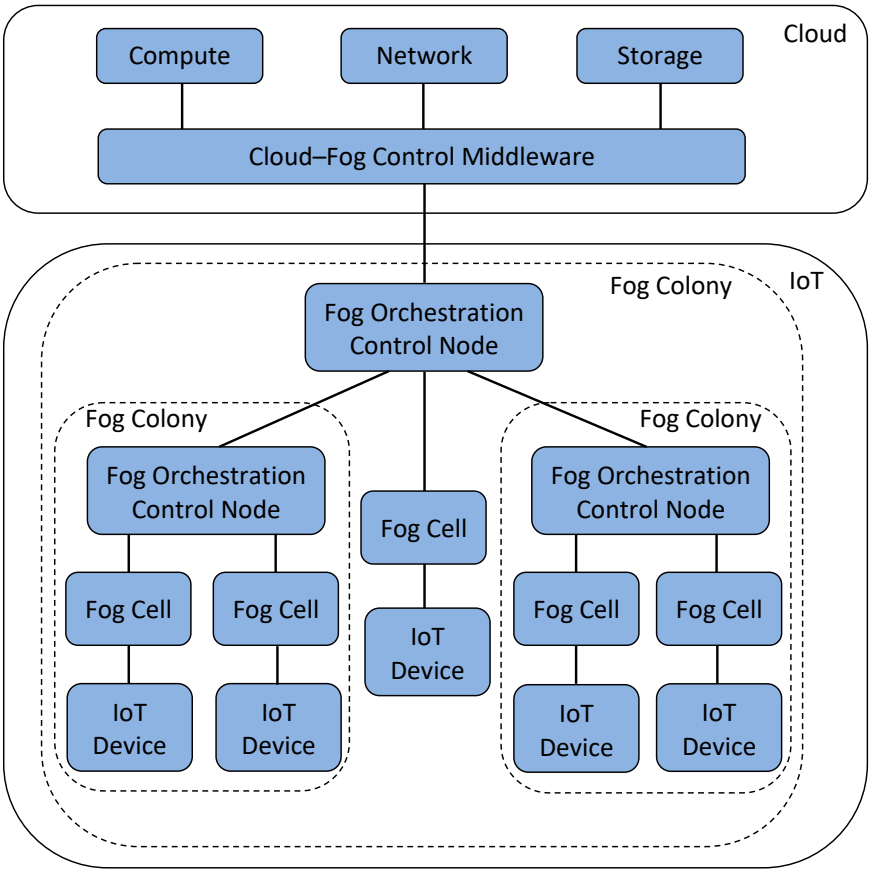

Abbildung 3.3.: Architektur des Cloud/Fog Provisionierungs-Frameworks [SSBL16]

Control Middleware erreicht werden können. Wenn Ressourcen für eine Anwendung provisioniert werden sollen, wird also zunächst die Cloud-Fog Control Middleware aufgerufen, die die erforderlichen Ressourcen in der Cloud bereitstellt. Außerdem leitet sie die Befehle zur Bereitstellung der Ressourcen im Fog an die verschiedenen Fog Orchestration Control Nodes weiter. Diese stellen die Ressourcen in den lokalen Fog Cells bereit und senden die Befehle über die Baumstruktur an die Fog Orchestration Control Nodes weiter unten. Falls nur Ressourcen im Fog und nicht in der Cloud bereitgestellt werden müssen, erlaubt es das Framework auch, eine Fog Colony autonom auszuführen. Dann ist die Wurzel der Baumstruktur also nicht die Cloud-Fog Control Middleware, sondern der Fog Orchestration Control Node der autonomen Fog Colony. Diese kann jedoch ebenfalls weitere Fog Colonies verwenden und steuern. Eine Evaluierung des Frameworks, die Skarlat et al. in ihrer Arbeit vorgenommen haben, ergibt, dass Verzögerungen bei der Ausführung von Aufgaben um bis zu 39% reduziert werden können. Diese Reduzierung wird hauptsächlich erreicht, da die Übertragung von Daten in die Cloud nur noch für sehr rechenintensive Aufgaben vorgenommen werden muss. Ein weiteres Ergebnis ist, dass die Ressourcen der IoT Devices durch die Verwendung im Fog deutlich besser ausgelastet werden und somit Kosten für Ressourcen in der Cloud gespart werden können.

4. Problemstellung und Anforderungen

In diesem Kapitel werden die Problemstellung und anschließend die zu erfüllenden Anforderungen an das in dieser Arbeit entwickelte Konzept zur automatisierten und verteilten Provisionierung von IoT Anwendungen erläutert. Dazu wird zunächst beschrieben, aus welchen Gründen die Provisionierung von Anwendungen, die als TOSCA Service Template modelliert wurden, nach dem im TOSCA Primer [OASc] vorgeschlagenen Konzept in IoT Umgebungen zu Problemen führen kann. Darauf aufbauend werden Anforderungen abgeleitet, die an das in dieser Arbeit entwickelte Konzept gestellt werden.

4.1. Problemstellung

Dieser Abschnitt erläutert die Problemstellung, die dieser Arbeit zugrunde liegt. In Abschnitt 4.1.1 werden die Abläufe vorgestellt, die nach der Definition des TOSCA Primers üblicherweise in einer TOSCA-Runtime vorzufinden sind [OASc]. Dabei wird besonderes Augenmerk auf die Abläufe beim Hochladen einer CSAR und beim Erzeugen einer Instanz einer Anwendung gelegt. Je nach Implementierung können die Abläufe oder die Namen der einzelnen beteiligten Komponenten von der Definition abweichen. Die grundsätzlichen Abläufe sollten jedoch in jeder TOSCA-konformen Runtime ähnlich sein. Abschnitt 4.1.2 erläutert anschließend beispielhaft zwei der Probleme, die bei der Verwendung der vorgestellten Abläufe in IoT Szenarien auftreten können.

4.1.1. Abläufe in einer TOSCA-Runtime

Um die Provisionierung einer Anwendung automatisiert mit der Hilfe einer TOSCA-Runtime durchführen zu können, muss die Anwendung zunächst modelliert werden [OASc]. Dazu kann ein TOSCA Modellierungswerkzeug, wie zum Beispiel Winery [KBBL13], verwendet werden. Das Endprodukt der Modellierung ist eine CSAR Datei, die über alle benötigten Definitionen, wie zum Beispiel das Service Template der Anwendung, verfügt. Außerdem enthält die CSAR auch alle Software-Artefakte, wie DAs und IAs, die zur Provisionierung der Anwendung benötigt werden. Anschließend kann die CSAR Datei in eine TOSCA-Runtime hochgeladen werden. Der damit ausgelöste Prozess innerhalb der TOSCA-Runtime ist in Abbildung 4.1 als BPMN Diagramm dargestellt.

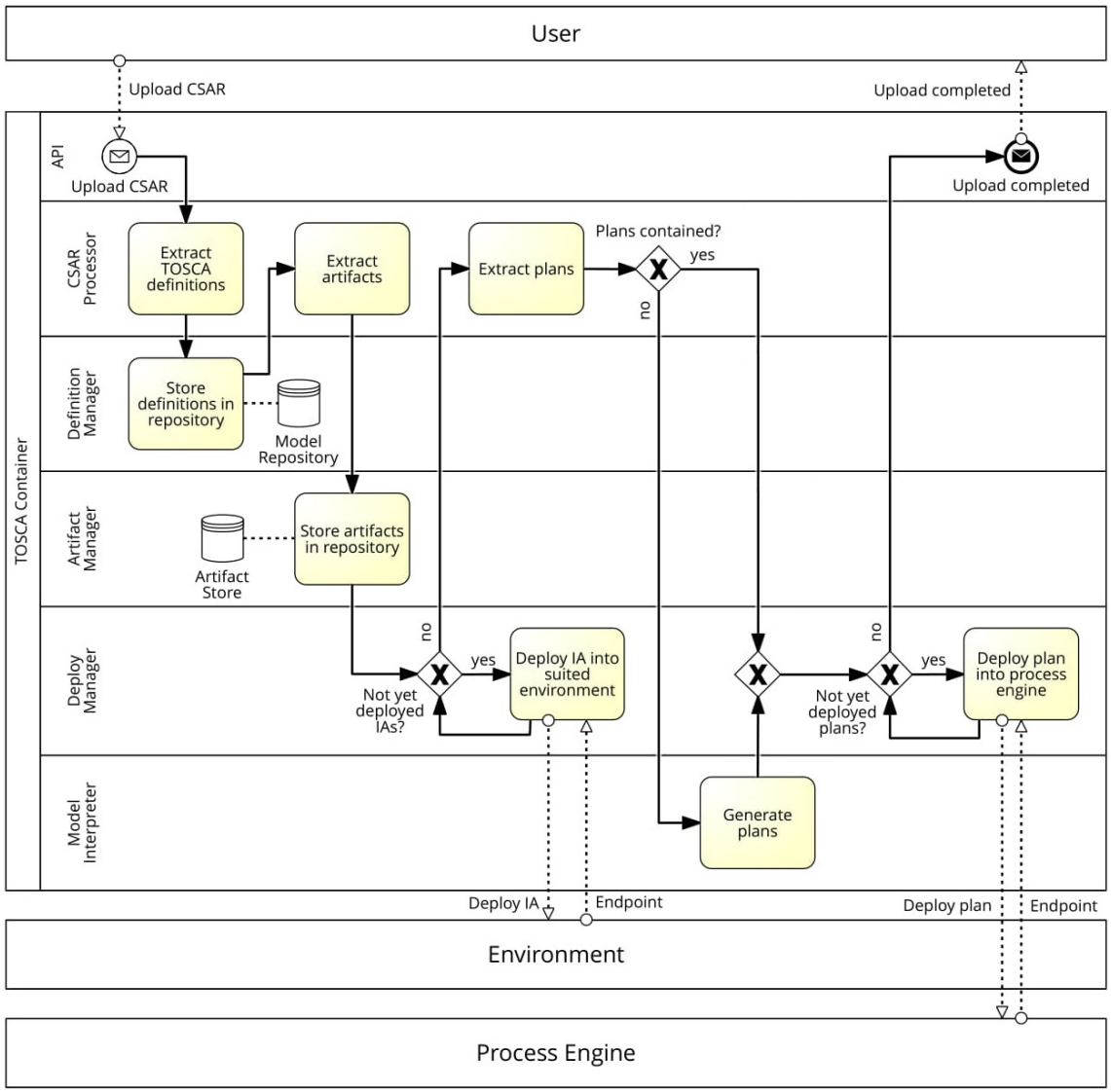

Abbildung 4.1.: CSAR Deployment in einer TOSCA-Runtime

Zunächst lädt der Nutzer die CSAR über eine *API* in die TOSCA-Runtime bzw. den *TOSCA Container* hoch [OASc]. Dabei kann eine grafische Benutzeroberfläche verwendet werden oder der Upload direkt über eine Konsole durchgeführt werden. Die API übergibt die CSAR an den *CSAR Processor*, der den Inhalt der CSAR extrahiert und alle weiteren Aktionen koordiniert. Der CSAR Processor leitet die TOSCA Definitionen der CSAR an den *Definition Manager* weiter, der diese in einem Repository speichert, um später den Zugriff auf die Definitionen zu ermöglichen. Als nächstes speichert der *Artifact Manager* die Artefakte (IAs und DAs) der CSAR auf eine geeignete Weise. Skripte können beispielsweise in einem lokalen Dateisystem gespeichert werden, Docker Images in einer Docker Registry, usw. Um die IAs, welche die Operationen der Node und Relationship Types der CSAR implementieren, bei der Instanzerzeugung aufrufen zu können, müssen diese in einem geeigneten *Environment* bereitgestellt werden. WAR Artefakte können zum Beispiel auf einem Tomcat

Server deployt werden. Aus einem Docker Image kann hingegen auf einem Docker Host ein Container erstellt werden. Die Aufgabe der Bereitstellung der IAs übernimmt der sogenannte *Deploy Manager*. Die Environments liefern dem TOSCA Container die Endpunkte der IAs zurück, über die diese aufgerufen werden können. Anschließend überprüft der CSAR Processor, ob die CSAR bereits Management Pläne für die Anwendung beinhaltet. Falls dies nicht der Fall ist, wird die Topologie der Anwendung an den *Model Interpreter* übergeben, der die benötigten Management Pläne aus der Topologie generiert. Der Model Interpreter wird jedoch nur für TOSCA-Runtimes benötigt, die neben dem imperativen Ansatz auch den deklarativen Ansatz unterstützen, da nur dann eine Generierung von Plänen nötig ist. Im letzten Schritt werden die in der CSAR enthaltenen oder generierten Pläne vom Deploy Manager auf einer *Process Engine* deployt. Beim Deployment der Pläne müssen die Endpunkte der benötigten IAs bereits an die Pläne gebunden werden. Dies bedeutet, dass das Deployment der IAs zwangsläufig immer vor dem Deployment der Pläne durchgeführt werden muss, da die IA Endpunkte ansonsten noch nicht vorhanden wären. Nachdem der Deploy Manager das Plan Deployment abgeschlossen hat, wird der Nutzer über die API benachrichtigt, dass der Upload der CSAR erfolgreich beendet wurde.

Im Anschluss an das erfolgreiche Deployment einer CSAR ist das automatisierte Erstellen von Instanzen der durch die CSAR repräsentierten Anwendung möglich [OASc]. Abbildung 4.2 stellt den Ablauf der Instanzerzeugung als BPMN Diagramm dar. Der Nutzer startet die Provisionierung einer Instanz, indem er eine entsprechende Anfrage an die TOSCA Container API sendet. Die Anfrage muss den Namen der CSAR beinhalten, welche die zu erzeugende Anwendung repräsentiert. Außerdem muss der Nutzer alle Parameter übergeben, die zum Aufruf des *Build Plans* benötigt werden [OASe]. Ein solcher Parameter kann zum Beispiel die IP- oder MAC-Adresse eines Raspberry Pis sein, auf dem eine Anwendung provisioniert werden soll. Die Anfrage wird innerhalb der TOSCA-Runtime an den *Instance Manager* übergeben, der ein neues Instanz Objekt erzeugt und in einer Datenbank speichert. Dieses Objekt kann zum Beispiel die übergebenen Parameter, die Zeit der Instanzerzeugung und den aktuellen Zustand der Instanz beinhalten. Somit kann der Nutzer jederzeit die Daten über alle vom TOSCA Container verwalteten Instanzen abrufen.

Als nächstes ruft der Instance Manager den Build Plan der Anwendung in der Process Engine auf [OASc]. Dazu muss er den Endpunkt des während der Verarbeitung der CSAR bereitgestellten Plans nachschlagen und eine Nachricht mit den Parametern des Nutzers an den Endpunkt senden. Die Process Engine startet den Plan und führt die enthaltenen *Tasks* aus. Ein Task kann beispielsweise der Aufruf einer Operation eines Node Types auf dem korrespondierende IA sein. Außerdem kann ein Task auf Daten des TOSCA Containers zugreifen und zum Beispiel DAs herunterladen oder Instanzdaten updaten. Da TOSCA vorhandene Plan Standards wie BPMN und BPEL verwendet, kann ein Task jedoch auch beliebige andere Aktionen ausführen. Nach der Terminierung des Plans, sendet die Process Engine eine Nachricht mit potentiellen Ausgabe-Parametern an den TOSCA Container. Dieser führt, falls nötig, ein Update der Instanzdaten durch und sendet dem Nutzer die Bestätigung, dass die gewünschte Instanz erfolgreich erstellt wurde.

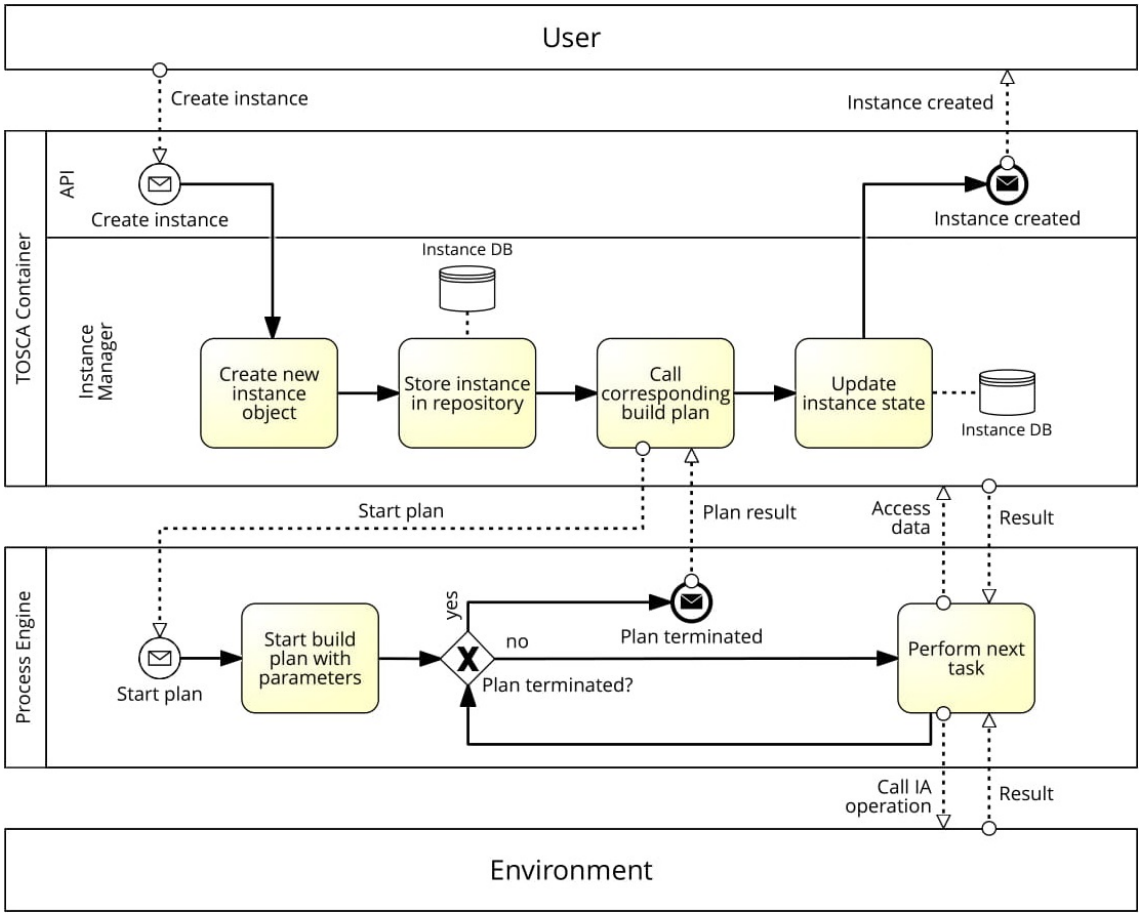

Abbildung 4.2.: Instanzerzeugung in einer TOSCA-Runtime

4.1.2. Probleme in IoT Szenarien

Durch die Implementierung der in Abschnitt 4.1.1 vorgestellten Abläufe in einer TOSCA-Runtime, wird die automatische Provisionierung von Cloud Anwendungen ermöglicht [OASc]. Somit kann ein aufwendiges, fehleranfälliges und teures manuelles Instanziieren von Cloud Anwendungen vermieden werden [SBK+16]. Für den Einsatz in IoT Szenarien mit heterogenen Netzwerken und einer großen Anzahl an Geräten mit geringen Ressourcen ergeben sich jedoch eine Reihe von Problemen, die eine effiziente Provisionierung von IoT Anwendungen verhindern. In diesem Abschnitt werden zwei dieser Probleme beispielhaft vorgestellt. Dabei wird davon ausgegangen, dass die TOSCA-Runtime in einem getrennten Netzwerk, außerhalb der IoT Umgebung, in der die Anwendung bereitgestellt werden soll, vorhanden ist. Die TOSCA-Runtime kann also in der Cloud oder einem Datenzentrum ausgeführt werden. Dies entspricht einem häufigen Anwendungsfall, weil die TOSCA-Runtime somit von einem externen Anbieter bereitgestellt und bedient werden kann. Besitzer eines Smart Homes sind häufig keine IT Spezialisten und können bzw. wollen ihre Anwendungen deshalb nicht selbst verwalten, da dies trotz der Vereinfachung durch die TOSCA-Runtime keine triviale Aufgabe ist. Damit können diese also dem externen

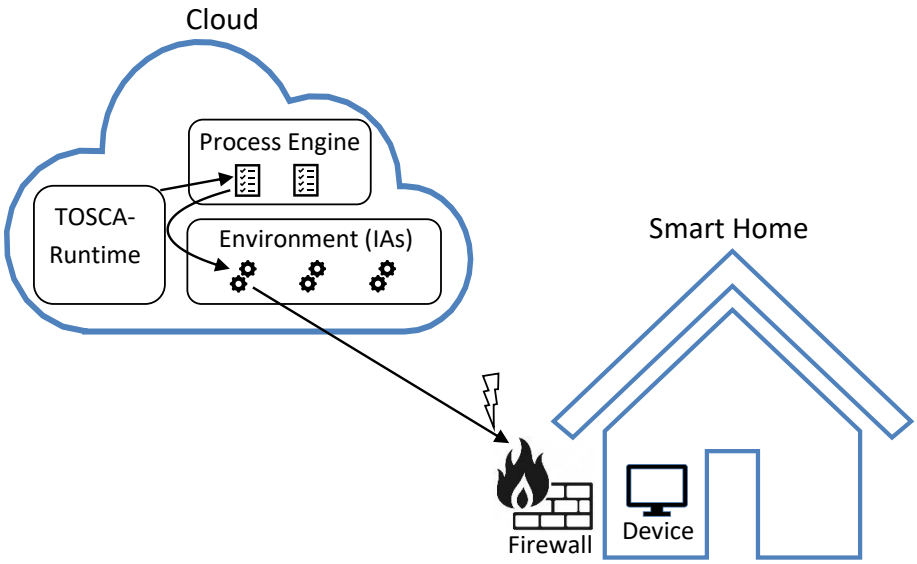

Abbildung 4.3.: Problem: Firewalls in IoT Umgebungen

Anbieter die Provisionierung und das Management ihrer Anwendungen übertragen. Dieser kann dann die Smart Homes vieler verschiedener Kunden durch eine zentrale, über das Internet zugängliche, TOSCA-Runtime steuern.

In Abbildung 4.3 ist das erste betrachtete Problem dargestellt. Links oben in der Abbildung ist die TOSCA-Runtime, die Process Engine mit zwei Plänen und das Environment mit drei bereitgestellten IAs zu sehen. Die Komponenten werden alle in der Cloud ausgeführt, um eine gute Skalierbarkeit und den Zugriff von jedem Ort aus zu ermöglichen. Rechts unten ist das Smart Home eines Kunden des TOSCA-Runtime Betreibers abgebildet. Im Smart Home ist ein Device, zum Beispiel ein Raspberry Pi, vorhanden, auf welchem Anwendungen ausgeführt werden können. Wenn der Kunde beim Anbieter eine neue Anwendung bestellt, möchte der Anbieter diese mittels der TOSCA-Runtime automatisch auf dem Device des Kunden bereitstellen. Dazu erzeugt ein Mitarbeiter in der TOSCA-Runtime eine neue Instanz der Anwendung und verwendet dafür die Parameter, die das Device des Kunden identifizieren und für die gewünschte Konfiguration der Anwendung sorgen. Daraufhin ruft die TOSCA-Runtime den Build Plan der Anwendung in der Process Engine auf [OASe]. Für die Provisionierung verwendet der Build Plan die IAs, die im Environment bereitgestellt werden. Ein solches IA kann zum Beispiel einen SSH Zugriff auf dem Device des Kunden durchführen, um dort bestimmte Programme herunterzuladen oder Befehle auszuführen. Dieser Zugriff vom IA auf das Device des Kunden ist jedoch üblicherweise nicht möglich, da das Netzwerk im Smart Home über eine Firewall verfügt und diese das Device vor Zugriffen von außen schützt. Die automatische Provisionierung der Anwendung schlägt deshalb fehl. Eine Möglichkeit, dieses Problem zu umgehen, wäre, die Firewall für den Zugriff von außen zu öffnen. Da dies jedoch ein Sicherheitsrisiko darstellt, ist dieses Vorgehen keine Alternative. Ein weiteres Problem dieser Möglichkeit

ist, dass der Kunde eventuell nicht selbst in der Lage ist, die Firewall neu zu konfigurieren. Deshalb müsste ein Techniker des Anbieters diese Aufgabe übernehmen. Dies steht aber im Widerspruch zur Automatisierung des gesamten Prozesses.

Abbildung 4.4 stellt ein zweites Problem grafisch dar, das einer effizienten Provisionierung von IoT Anwendungen mittels einer TOSCA-Runtime im Weg steht. Zur Beschreibung des Problems wird davon ausgegangen, dass das oben beschriebene Problem der aktiven Firewalls gelöst werden konnte und ein Zugriff von der TOSCA-Runtime auf die Devices des Kunden möglich ist. Im Vergleich zu Abbildung 4.3 besitzt der Kunde zwei Devices, auf denen er die gleiche Anwendung ausführen möchte. Dies kann zum Beispiel sinnvoll sein, wenn der Kunde die Heizung seines Smart Homes automatisch steuern lassen möchte. Dabei können in mehreren Räumen Sensoren zur Messung der Raumtemperatur installiert werden, die eine bessere Überwachung der Temperatur ermöglichen, als ein einzelner Sensor. Das Problem, das bei der Erzeugung mehrerer Instanzen im selben Smart Home entsteht, ist die mehrfache Übertragung derselben Dateien in das gleiche Netzwerk. Solche Dateien können zum Beispiel DAs sein, die auf die Devices übertragen werden müssen. Da in IoT Umgebungen häufig nur begrenzte Ressourcen zur Verfügung stehen, wird durch dieses Verfahren unnötig verfügbare Bandbreite verbraucht. Eine bessere Lösung wäre es, beim Erstellen der ersten Instanz die Datei zu übertragen und im Netzwerk zwischenzuspeichern. Damit muss bei der Provisionierung der zweiten Instanz die Datei nur noch im lokalen Netzwerk transferiert werden.

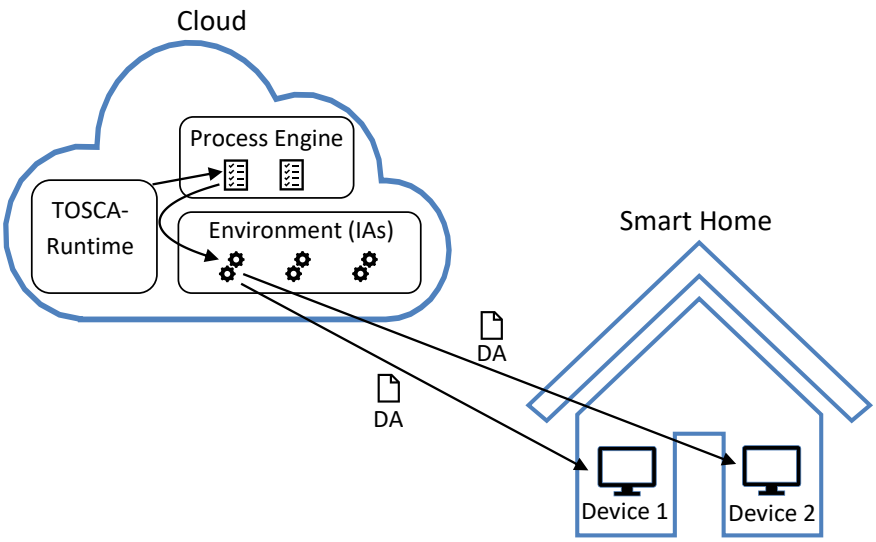

Abbildung 4.4.: Problem: Verschwendung von verfügbarer Bandbreite

4.2. Anforderungen

In diesem Abschnitt werden die Anforderungen an diese Arbeit definiert. Dabei werden Anforderungen an das Gesamtkonzept zur verteilten Provisionierung von TOSCA-basierten Anwendungen in IoT Umgebungen vorgestellt. Zusätzlich werden Anforderungen an die prototypische Implementierung des Konzepts und die Integration in das OpenTOSCA Ökosystem aufgelistet.

- **Anforderung 1 (A1):** Das Konzept soll die automatische und verteilte Provisionierung von mit TOSCA modellierten Anwendungen in IoT Umgebungen mit heterogenen Netzwerken und Geräten mit geringen Ressourcen ermöglichen.
- **Anforderung 2 (A2):** Das Management von IoT Anwendungen in verschiedenen IoT Umgebungen soll über ein einzelnes zentrales Interface durchführbar sein.
- **Anforderung 3 (A3):** Informationen zur Identifizierung von Geräten, die Teil der Topologie einer Anwendung sind, müssen dynamisch zur Zeit der Instanzerzeugung übergeben werden können, um eine Wiederverwendbarkeit von Anwendungen zu ermöglichen.
- **Anforderung 4 (A4):** Probleme, die durch aktive Firewalls in IoT Umgebungen entstehen, sollen durch das Konzept gelöst werden, ohne dass eine neue Konfiguration von Firewalls benötigt wird.
- **Anforderung 5 (A5):** Die Effizienz der Provisionierung mehrerer gleicher Anwendungen in einer IoT Umgebung soll gesteigert werden, indem große Dateien lediglich einmal in das heterogene Netzwerk der IoT Umgebung übertragen werden müssen.
- **Anforderung 6 (A6):** Die Modellierung soll ohne neue Modellierungskonstrukte erfolgen und es soll somit erlaubt werden, dass Cloud und IoT Anwendungen konzeptionell auf die gleiche Weise modelliert werden können.
- **Anforderung 7 (A7):** Entwicklung einer prototypischen Implementierung des Konzepts und Integration in das OpenTOSCA Ökosystem.
- **Anforderung 8 (A8):** Die Performance der Provisionierung von Anwendungen mittels OpenTOSCA soll sich durch die Integration des Konzepts nicht verschlechtern.

Anforderung **A1** gibt der Arbeit den grundsätzlichen Rahmen. Es sollen IoT Anwendungen mittels TOSCA modelliert und anschließend automatisch bereitgestellt werden können. Dabei muss das Konzept die IoT spezifischen Besonderheiten, wie die geringen verfügbaren Ressourcen, behandeln. Zusätzlich soll das verteilte Management von IoT Anwendungen in unterschiedlichen IoT Umgebungen ermöglicht werden und dabei über ein zentrales Interface steuerbar sein (**A2**). Dies ist für Anbieter von IoT Anwendungen bedeutsam, da diese die Anwendungen aller Kunden zentral verwalten können, ohne dabei einen Einsatz von Technikern beim Kunden zu benötigen. So kann zum Beispiel eine neue Version einer Anwendung mit nur wenigen Klicks bei allen Kunden automatisiert bereitgestellt werden.

Für die Provisionierung von Anwendungen werden bestimmte Informationen benötigt, die zum Beispiel die Hardware identifizieren, auf der die Anwendung ausgeführt werden soll. Eine TOSCA-Runtime darf nicht die Anforderung stellen, dass diese Informationen beim Upload der CSAR bereits verfügbar sind (**A3**). Dies würde bedeuten, dass die Informationen bereits zur Modellierungszeit in der CSAR hinterlegt werden müssten und die CSAR deshalb nur zur Provisionierung einer ganz bestimmten Instanz verwendbar wäre [OASe]. Die Wiederverwendbarkeit von CSARs würde damit also verloren gehen. Deshalb muss es möglich sein, die benötigten Informationen erst bei der Erzeugung einer Instanz an die TOSCA-Runtime zu übergeben.

Die Anforderungen **A4** und **A5** definieren, dass die in Abschnitt 4.1.2 beschriebenen Probleme durch das Konzept gelöst werden sollen. Eine TOSCA-Runtime, die dem in dieser Arbeit entwickelten Konzept folgt, soll also in der Lage sein, Anwendungen in IoT Umgebungen zu provisionieren, auch wenn diese von einer Firewall geschützt werden (**A4**). Dabei ist es wichtig, dass hierzu keine neue Konfiguration der Firewall benötigt wird. Außerdem soll die TOSCA-Runtime die Effizienz der Provisionierung mehrerer gleicher Anwendungen in einer IoT Umgebungen steigern, indem große Dateien, wie zum Beispiel DAs, lediglich einmal in die jeweilige Umgebung übertragen werden müssen (**A5**).

A6 beschreibt, dass sich die Modellierung von Cloud und IoT Anwendungen konzeptionell nicht unterscheiden soll. Das bedeutet, die Anwendungs-Modellierer müssen kein neues Vorgehen erlernen, sondern können lediglich andere Node und Relationship Types für die Modellierung von IoT Anwendungen einführen und verwenden. Insbesondere darf deshalb keine Erweiterung des TOSCA Standards mit neuen Konstrukten nötig sein, um zusätzliche benötigte Informationen darstellen zu können.

Die Anforderungen **A7** und **A8** beziehen sich auf die prototypische Implementierung des entwickelten Konzepts. Ein Ziel der Arbeit ist es, das entwickelte Konzept in das OpenTOSCA Ökosystem (siehe Abschnitt 2.7) einzubinden (**A7**). Deshalb bildet der OpenTOSCA Container, der im Ökosystem als TOSCA-Runtime fungiert, die Grundlage für die Implementierung des Konzepts. Die grundsätzlichen Abläufe im OpenTOSCA Container orientieren sich an den in Abschnitt 4.1.1 vorgestellten Abläufen, die im TOSCA Primer definiert sind. Durch die Einbindung des entwickelten Konzepts, ändern sich diese Abläufe an einigen Stellen. Dabei muss beachtet werden, dass dadurch die Performance der Provisionierung von Cloud Anwendungen möglichst wenig beeinträchtigt wird (**A8**).

5. Konzept der verteilten Provisionierung von IoT Anwendungen

Dieses Kapitel präsentiert ein Konzept zur verteilten Provisionierung von IoT Anwendungen basierend auf TOSCA Topologien. Dazu werden die einzelnen Bestandteile des Konzepts zunächst separat betrachtet. Abschnitt 5.1 beschreibt als Erstes, warum ein verteiltes System aus mehreren kommunizierenden TOSCA-Runtimes die Probleme der Provisionierung von IoT Anwendungen beheben kann. Außerdem wird definiert, auf welche Weise die Kommunikation zwischen den TOSCA-Runtimes am besten stattfindet. Anschließend wird in Abschnitt 5.2 erläutert, weshalb das Deployment von Implementation Artifacts für die Verwendung einer verteilten TOSCA-Runtime verzögert werden muss und nicht beim Deployment der CSAR durchgeführt werden kann. Als letzten Bestandteil des Konzepts betrachtet Abschnitt 5.3 die Frage, wie entschieden werden kann, welche TOSCA-Runtime des verteilten Systems die Provisionierung einer bestimmten Anwendung bzw. eines Knotens der Anwendung durchführen muss. In Abschnitt 5.4 werden die einzelnen Bestandteile schließlich zum Gesamtkonzept zusammengesetzt und es wird vorgestellt, wie die Abläufe in den TOSCA-Runtimes angepasst werden müssen, um das Konzept umzusetzen.

5.1. Verteilte TOSCA-Runtime

In diesem Abschnitt wird beschrieben, wie eine *verteilte TOSCA-Runtime* die Probleme, die bei der Provisionierung von Anwendungen in IoT Umgebungen auftreten können, beheben kann. Anschließend wird analysiert, wie die Kommunikation zwischen den einzelnen TOSCA-Runtime Knoten am besten stattfindet. Dabei müssen verschiedene Abwägungen durchgeführt werden. Zum Beispiel, ob die Kommunikation Push- oder Pull-basiert abläuft, ob sie synchron oder asynchron sein soll und welches konkrete Protokoll zur Umsetzung verwendet werden kann.

Unter einer verteilten TOSCA-Runtime versteht man, dass mehrere TOSCA-Runtime Knoten parallel genutzt werden und diese in verschiedenen Netzwerken positioniert sind. Die einzelnen Knoten können anschließend miteinander kooperieren und repräsentieren damit für den Nutzer eine einzelne, verteilte TOSCA-Runtime. Bei der Art der Zusammenarbeit der TOSCA-Runtime Knoten kann zwischen zwei verschiedenen Ansätzen unterschieden

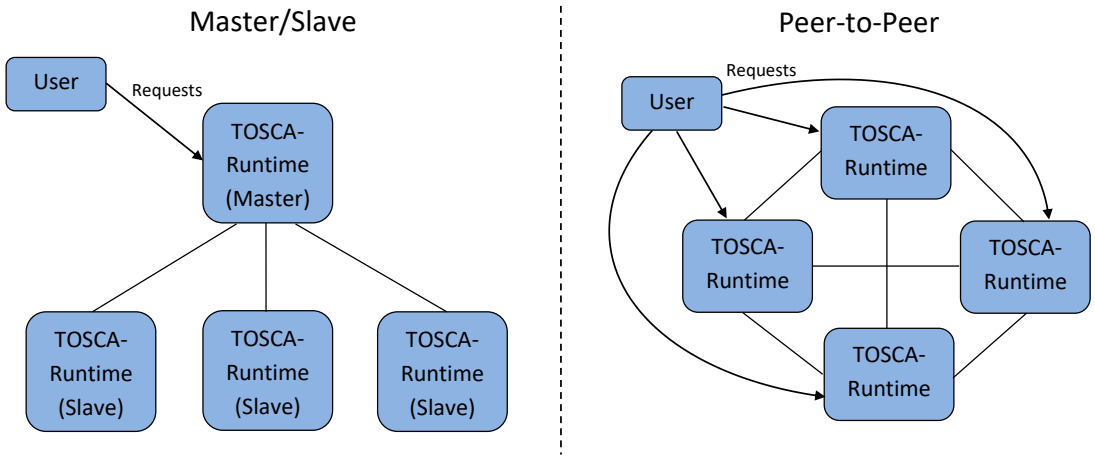

Abbildung 5.1.: Architekturvergleich: Master/Slave vs. Peer-to-Peer (angelehnt an [KM])

werden: *Master/Slave* [Bus95] und *Peer-to-Peer* [SF02]. Abbildung 5.1 stellt die Unterschiede zwischen den beiden Architekturen grafisch dar. Bei der Master/Slave Architektur nimmt lediglich ein Knoten die Anfragen von Nutzern entgegen [Bus95]. Dieser Knoten wird als *Master* bezeichnet. Alle anderen Knoten werden *Slaves* genannt und können dem Master helfen, Teile der Anfrage des Nutzers zu bearbeiten. Bei einer Peer-to-Peer Architektur sind hingegen alle Knoten gleichberechtigt [SF02]. Das bedeutet, dass jeder Knoten in der Lage ist, Anfragen von Nutzern entgegenzunehmen und die Dienste anderer Knoten für die Erfüllung der Anfragen in Anspruch zu nehmen.

Die beiden unterschiedlichen Architekturen bieten jeweils eine Reihe von Vor- und Nachteilen [Bus95][SF02]. Master/Slave hat eine geringere Komplexität, da das gesamte Management des verteilten Systems über einen zentralen Knoten durchgeführt wird [Bus95]. Außerdem ist der Kommunikationsaufwand in Master/Slave Systemen häufig geringer. Der Grund hierfür ist, dass beim Master/Slave System nur der Master Anfragen an Slaves sendet und diese die Anfragen beantworten. Bei Peer-to-Peer Systemen dagegen kommunizieren und koordinieren sich alle Knoten des verteilten Systems [SF02]. Der große Nachteil der Master/Slave Architektur ist die Anfälligkeit des Masters für Fehler [Bus95]. Wenn dieser aufgrund eines Fehlers ausfällt, ist das gesamte verteilte System nicht mehr verfügbar. Ein Peer-to-Peer System kann dagegen üblicherweise trotz des Ausfalls eines einzelnen Knotens problemlos weiterarbeiten [SF02]. Ein weiterer Vorteil von Peer-to-Peer Systemen ist die häufig bessere Performance, da nicht alle Anfragen durch einen einzelnen Knoten geleitet werden müssen und dieser somit nicht zum Flaschenhals des Systems werden kann. Aufgrund der Vor- und Nachteile beider Architekturen kann die Wahl der geeigneten Architektur je nach Anwendungsfall der verteilten TOSCA-Runtime unterschiedlich ausfallen. Deshalb schreibt das Konzept keine der beiden Architekturen vor, sondern ermöglicht es, beide Architekturen umzusetzen. Wie dies erreicht werden kann, wird am Ende dieses Abschnitts genauer diskutiert.

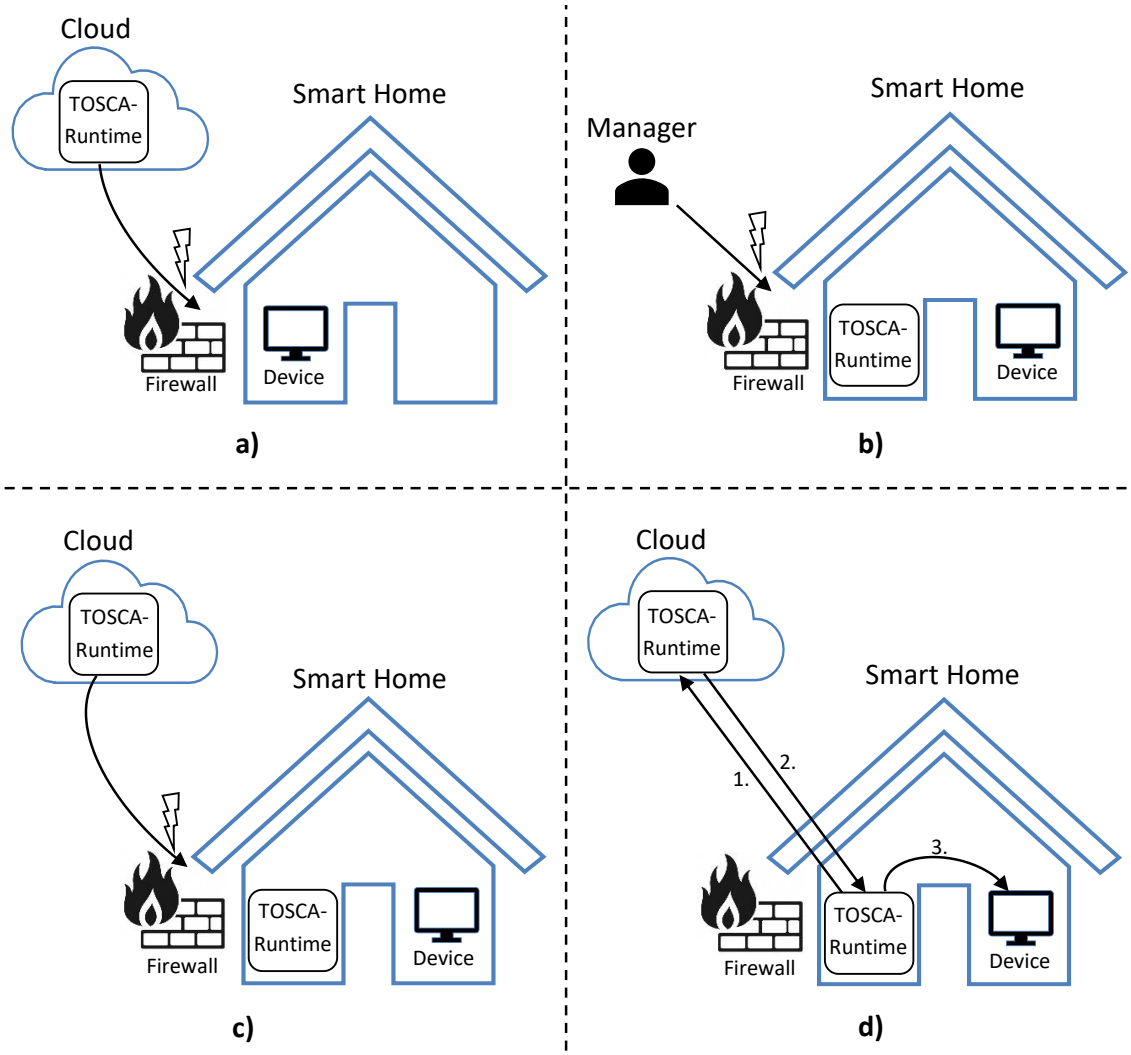

Abbildung 5.2.: Lösung des Firewall Problems durch eine verteilte TOSCA-Runtime

Zunächst wird betrachtet, warum eine verteilte TOSCA-Runtime die in Abschnitt 4.1.2 beschriebenen Probleme bei der Provisionierung von IoT Anwendungen beheben kann. Abbildung 5.2 a) stellt das Problem bei aktiven Firewalls erneut dar. Für eine bessere Übersichtlichkeit wird hier nur die TOSCA-Runtime abgebildet und nicht die Process Engine und das Environment. Diese sind jedoch trotzdem immer zusätzlich zu jedem TOSCA-Runtime Knoten vorhanden. Aufgrund der aktiven Firewall im Smart Home ist die TOSCA-Runtime nicht in der Lage, auf das Device zuzugreifen. Somit ist keine Provisionierung von Anwendungen möglich. Ein Lösungsansatz, um das Problem zu beheben ist, die TOSCA-Runtime im Smart Home auszuführen. Entsprechend hat die TOSCA-Runtime Zugriff auf die Devices innerhalb des Smart Homes. Dieser Ansatz ist in Abbildung 5.2 b) dargestellt. Das Problem des Ansatzes ist, dass damit der Zugriff auf die TOSCA-Runtime nur innerhalb des Smart Homes möglich ist. Sofern der Besitzer des Smart Homes alle seine Anwendungen selbst verwalten möchte, ist dies kein Problem

und Ansatz b) kann angewendet werden. In vielen Fällen aber sollen die Anwendungen von einem Manager verwaltet werden, der sich außerhalb des Smart Homes befindet und keinen Zugriff auf die TOSCA-Runtime im Smart Home hat.

Abbildung 5.2 c) verwendet zur Lösung des Firewall Problems eine verteilte TOSCA-Runtime. Ein Knoten wird dabei in der Cloud ausgeführt und erlaubt den Zugriff von externen Managern. Ein zweiter Knoten wird dagegen im Smart Home ausgeführt und hat somit Zugriff auf die Devices im Smart Home. Um das Problem vollständig zu lösen, muss der Knoten in der Cloud in der Lage sein, dem Knoten im Smart Home Befehle zur Provisionierung von Teilen der Anwendung zu übermitteln. Bei Ansatz c) wird dies versucht, indem die TOSCA-Runtime in der Cloud die Befehle Push-basiert sendet. Dabei verhindert jedoch erneut die Firewall die Kommunikation. Dies wird in Abbildung 5.2 d) gelöst, indem eine Pull-basierte Kommunikation verwendet wird. Der TOSCA-Runtime Knoten im Smart Home fragt also beim Knoten in der Cloud nach neuen Befehlen und erhält diese als Antwort. Da die Kommunikation aus dem geschützten Bereich gestartet wird, kann sie trotz aktiver Firewall stattfinden [Opp97].

Durch eine verteilte TOSCA-Runtime kann auch das Problem behoben werden, dass große Dateien mehrfach in das Netzwerk der IoT Umgebung übertragen werden müssen. Abbildung 5.3 stellt die Lösung dafür grafisch dar. Die TOSCA-Runtime in der Cloud überträgt die Datei beim Erstellen der ersten Instanz der Anwendung zur TOSCA-Runtime im Smart Home. Diese kann die Datei lokal speichern und somit vermeiden, dass sie bei der Erzeugung einer zweiten Instanz der Anwendung erneut übertragen werden muss. Der Einfachheit halber wurde das Problem der aktiven Firewalls hier ignoriert, da dies wie oben beschrieben gelöst werden kann.

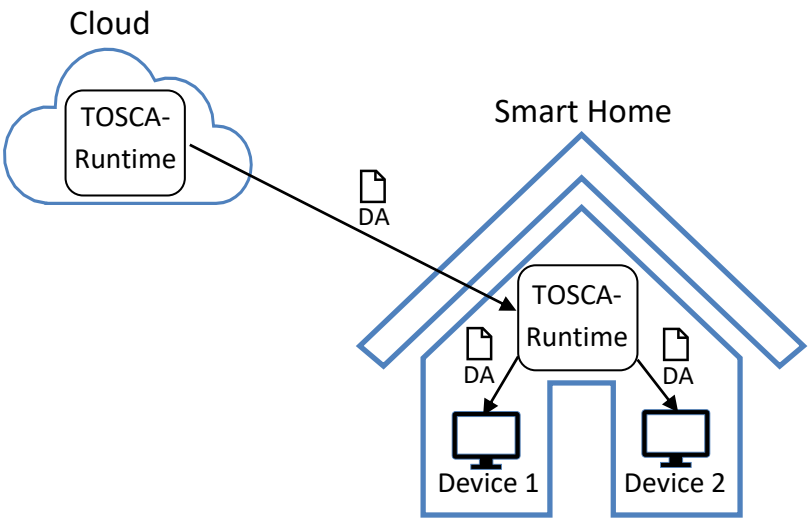

Abbildung 5.3.: Lösung des Problems der verschwendeten Bandbreite

Mit der Implementierung und Ausführung einer verteilten TOSCA-Runtime, können die Probleme der automatischen Provisionierung von Anwendungen in IoT Umgebungen also gelöst werden. Es muss jedoch noch definiert werden, auf welche Art die TOSCA-Runtime Knoten untereinander kommunizieren können und welche Anpassungen bei Implementierungen einer TOSCA-Runtime vorgenommen werden müssen, damit diese als verteilte TOSCA-Runtimes verwendet werden können. In Abbildung 5.2 c) und d) wurde bereits gezeigt, dass die Kommunikation zwischen den einzelnen TOSCA-Runtime Knoten Pull-basiert stattfinden sollte, um Firewall Probleme zu vermeiden. Die Pull-basierte Kommunikation kann synchron oder asynchron durchgeführt werden. Bei der synchronen Kommunikation senden die TOSCA-Runtime Knoten periodisch Anfragen an die anderen Knoten, ob neue Befehle für sie verfügbar sind. Für die asynchrone Kommunikation registrieren sie dagegen einen Callback und werden anschließend sofort informiert, sobald neue Befehle verfügbar sind. Da eine synchrone Kommunikation einen großen Aufwand darstellt, wenn nur wenige Befehle ausgetauscht werden, sollte eine asynchrone Kommunikation verwendet werden. Eine weitere Frage ist, welches Protokoll zur Umsetzung der Kommunikation verwendet werden sollte. Die Verwendung eines Publish/Subscribe Protokolls (siehe Abschnitt 2.2) bietet hierbei gegenüber einem Point-to-Point Protokoll den Vorteil, dass die TOSCA-Runtime Knoten nicht alle anderen Knoten des verteilten Systems kennen müssen, um einen zuständigen Knoten zu adressieren. Stattdessen können sie eine Anfrage auf einem Topic veröffentlichen. Der zuständige TOSCA-Runtime Knoten verarbeitet dann die Anfrage beim Erhalt selbständig, während alle anderen Knoten die Anfrage ignorieren. Ein geeignetes Protokoll ist deshalb das MQTT Protokoll, welches das Publish/Subscribe Konzept umsetzt und außerdem durch die leichtgewichtige Implementierung ideal für IoT Umgebungen ist, auf die das entwickelte Konzept abzielt. Die Verwendung anderer Protokolle ist jedoch ebenfalls denkbar.

Auf welche Weise die Prozesse innerhalb einer TOSCA-Runtime angepasst werden müssen, um die beschriebene Kommunikation umzusetzen, wird in Abschnitt 5.4 erläutert. Damit ein TOSCA-Runtime Knoten Befehle von einem anderen Knoten erhalten kann, benötigt dieser jedoch lediglich die Adresse des MQTT Brokers und das Topic, auf welchem die Befehle veröffentlicht werden. Dabei kann entweder ein globaler MQTT Broker verwendet werden, der von mehreren Knoten zur Veröffentlichung von Befehlen genutzt wird, oder jeder Knoten kann einen eigenen Broker einsetzen. Je nachdem, welche Architektur für die verteilte TOSCA-Runtime erreicht werden soll, können den einzelnen Knoten die passenden Broker Adressen übergeben werden. Soll eine Master/Slave Architektur erzielt werden, wird allen Knoten, außer dem Master, lediglich die Adresse des Brokers des Masters überreicht. Wenn dagegen eine Peer-to-Peer Architektur im verteilen System erreicht werden soll, erhält jeder Knoten die Adressen der Broker aller anderen Knoten. Dabei muss beachtet werden, dass keiner der MQTT Broker von einer Firewall geschützt ist, die die Kommunikation verhindern könnte.

5.2. Verzögertes Implementation Artifact Deployment

Wenn in einer verteilten TOSCA-Runtime ein Knoten für die Provisionierung und Verwaltung eines Teils einer Anwendung zuständig ist, bedeutet dies, dass auch alle Implementation Artifacts (IAs) des Teils der Anwendung im Environment dieses TOSCA-Runtime Knotens deployt werden müssen. Die IAs stellen die Management Infrastruktur für die Anwendung dar und werden deswegen in dem Knoten benötigt, der für die Provisionierung und das Management verantwortlich ist. Somit ist die Entscheidung, welcher Knoten für einen Teil einer Anwendung zuständig ist, auch gleichzeitig die darüber, in welchem Knoten die IAs bereitgestellt werden müssen. In einer TOSCA-Runtime werden die IAs nach den Abläufen des TOSCA Primer beim Upload einer CSAR direkt deployt (siehe Abschnitt 4.1.1) [OASc]. Deshalb muss bereits beim Upload einer CSAR entschieden werden, welcher TOSCA-Runtime Knoten beim Erzeugen einer Anwendung für die Provisionierung welcher Teile zuständig ist. Abbildung 5.4 stellt dieses Problem dar. Der Nutzer hat eine CSAR in den TOSCA-Runtime Knoten in der Cloud hochgeladen, wobei es unklar ist, ob der Knoten in der Cloud oder einer der beiden Knoten in den Smart Homes für einen Teil der Anwendung verantwortlich ist. Ohne diese Information ist es aber unmöglich, die zugehörigen IAs an der richtigen Stelle bereitzustellen. Eine Möglichkeit, dieses Problem zu umgehen, wäre, alle IAs aus jeder CSAR in allen verteilten TOSCA-Runtime Knoten zu deployen. Zum einen würde dieses Verfahren zu unnötigen Übertragungen von Daten

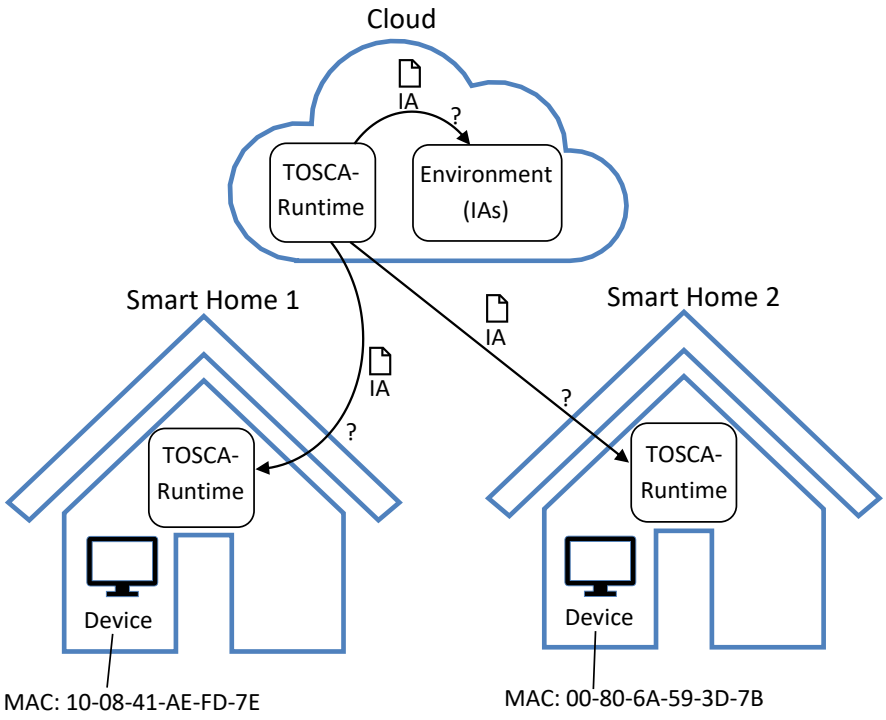

Abbildung 5.4.: Problem der IA Verteilung

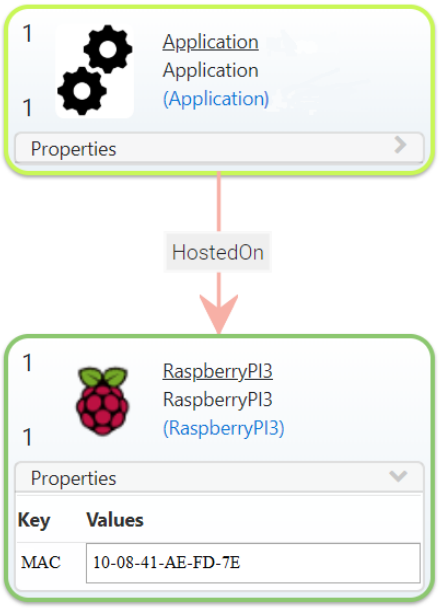

Abbildung 5.5.: Identifizierung von Hardware durch Properties

und zur Verschwendung von Ressourcen in den TOSCA-Runtimes führen. Zum anderen entstehen dadurch Probleme, wenn ein neuer TOSCA-Runtime Knoten in das verteilte System eingefügt wird. Der neue Knoten kann entweder nicht für die Provisionierung von Instanzen bereits hochgeladener CSARs verwendet werden oder es müssen alle IAs zunächst zu ihm Übertragen werden, wodurch eine große Verzögerung beim Start des Knotens entstehen kann. Deshalb ist dieses Verfahren nicht anwendbar.

Es stellt sich also die Frage, wie beim Upload einer CSAR entschieden werden kann, welcher Knoten für welche Teile einer Anwendung zuständig ist. Die Zuständigkeit hängt damit zusammen, auf welcher Hardware bzw. auf welchen Devices die Anwendung provisioniert werden soll. Wenn die Anwendung in Abbildung 5.4 auf dem Device mit der MAC-Adresse „10-08-41-AE-FD-7E“ ausgeführt werden soll, ist der TOSCA-Runtime Knoten in Smart Home 1 zuständig. Wird dagegen das Device mit MAC-Adresse „00-80-6A-59-3D-7B“ verwendet, ist der Knoten in Smart Home 2 verantwortlich. Informationen, welche die verwendete Hardware identifizieren können, werden in TOSCA als Properties an den Node Templates definiert. Es könnte also zum Beispiel eine Topologie aus einem Raspberry Pi und einer darauf ausgeführten Anwendung bestehen (siehe Abbildung 5.5). Dann kann das Raspberry Pi Node Template eine Property mit der MAC-Adresse des physischen Devices beinhalten, wodurch die Identifizierung der Hardware möglich ist. Solche Informationen sollten jedoch nicht bereits zur Modellierungszeit in der Topologie hinterlegt werden, da die erstellte CSAR ansonsten nur zur Provisionierung einer Anwendung auf einer bestimmten Hardware verwendet werden kann und damit nicht wiederverwendbar ist. Stattdessen sollten die Informationen beim Erzeugen einer Instanz einer Anwendung an die TOSCA-Runtime übergeben werden [BBH+13]. Damit kann dieselbe CSAR für die Provisionierung von beliebig vielen Instanzen auf beliebiger Hardware genutzt werden. Eine direkte Folge

daraus ist jedoch, dass die Entscheidung über die Zuständigkeit erst beim Erzeugen einer Instanz einer Anwendung getroffen werden kann. Deshalb muss das IA Deployment bei der Verwendung einer verteilten TOSCA-Runtime vom Zeitpunkt des Uploads einer CSAR bis zum Zeitpunkt der Instanzerzeugung verzögert werden.

Durch das Verzögern des IA Deployments können Probleme entstehen, die beim direkten Deployment während des CSAR Uploads nicht auftreten. Eine naheliegende Annahme ist es, dass sich die Zeit zur Erzeugung einer Instanz einer Anwendung damit verlängert, da das IA Deployment zusätzliche Zeit benötigt. Abschnitt 7.3 enthält eine Studie, die die Zeiten der Instanzerzeugung mit und ohne verzögertes IA Deployment am Beispiel der TOSCA-Runtime *OpenTOSCA Container* (siehe Abschnitt 2.7) vergleicht. Es zeigt sich, dass die Zeiten nur gering voneinander abweichen und beim Erzeugen der zweiten Instanz einer Anwendung nahezu identisch sind, da die IAs dann bereits deployt sind und wiederverwendet werden können. Ein zweites Problem ist, dass die IA Endpunkte durch das verzögerte Deployment beim Deployment der Pläne noch nicht bekannt sind. Damit können die Tasks der Pläne die IAs nicht direkt aufrufen. Deshalb muss ein Management Bus als zusätzliche Komponente in die TOSCA-Runtime integriert werden [Zim16]. Anstelle der IA Endpunkte, können die Pläne dann den Endpunkt des Management Busses aufrufen. Dieser kümmert sich dann um das Deployment der IAs bzw. das Nachschlagen des Endpunktes und das Weiterleiten des Aufrufs. Zusätzlich kann der Management Bus weitere Funktionalitäten wie das Übersetzen des Nachrichtenformates oder das Updaten von Parametern anbieten.

Mit der Verzögerung des Deployments muss auch der Zeitpunkt des Undeployments von IAs überdacht werden. Der übliche Zeitpunkt dafür ist während des Undeployments der CSAR [OASc][Ope]. Dies bedeutet also, die IAs werden erst gelöscht, sobald der Nutzer entscheidet, dass eine Erzeugung von weiteren Instanzen, der durch die CSAR repräsentierten Anwendung, mittels der TOSCA-Runtime nicht gewünscht ist. Somit werden die Ressourcen für die IAs im Environment während der kompletten Zeit, in der die CSAR in der TOSCA-Runtime aktiv ist, belegt [OASe]. Insbesondere in IoT Szenarien, in denen Knoten der verteilten TOSCA-Runtime eventuell auf IoT Devices ausgeführt werden, die nur über wenig Ressourcen verfügen, kann dies einen Nachteil darstellen. Ein früheres Undeployment der IAs hat hingegen zur Folge, dass, falls danach erneut eine Instanz der Anwendung erzeugt werden soll, die IAs erneut deployt werden müssen. Die Zeit der Instanzerzeugung verlängert sich also eventuell und es werden zusätzliche Ressourcen für das Deployment benötigt. Deshalb kann die Abwägung, ob ein früheres Undeployment sinnvoll ist, je nach Anwendungsfall unterschiedlich ausfallen. Dabei muss jedoch eine Unterscheidung zwischen *intrinsischen* und *extrinsischen IAs* durchgeführt werden (siehe Abbildung 5.6) [OASe]. Intrinsische IAs werden direkt auf das Device kopiert, auf dem die Anwendung provisioniert werden soll. Ein intrinisches IA kann also beispielsweise ein Shell Script sein, das einen Web Server auf einem Betriebssystem installiert. Da es unwahrscheinlich ist, dass dasselbe intrinsische IA auf demselben Device mehrmals hintereinander ausgeführt wird, sollten intrinische IAs direkt nach der Ausführung wieder entfernt werden, um Ressourcen zu sparen. Extrinsische IAs dagegen werden auf einer externen Infrastruktur ausgeführt und können damit für das Management mehrerer Instanzen auf unterschiedlichen Devices verwendet werden. Deshalb ist hier die oben beschriebene Abwägung zu treffen. Im Zuge

des Konzepts dieser Arbeit wurde als Zeitpunkt des Undeployments von extrinsischen IAs, das Terminieren der letzten Instanz der CSAR in der TOSCA-Runtime festgelegt. Damit wird die Wahrscheinlichkeit für ein mehrmaliges IA Deployment in kurzer Zeit gering gehalten. Gleichzeitig werden Ressourcen eventuell früher freigegeben, als beim alten Ansatz. Zukünftigen Arbeiten könnten das Konzept dieser Arbeit um einen Algorithmus erweitern, mit dem besser vorhergesagt werden kann, wann ein extrinsisches IA nicht mehr benötigt wird. Somit wäre eine noch bessere Ausnutzung der vorhandenen Ressourcen möglich.

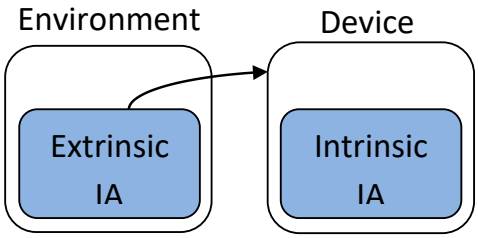

Abbildung 5.6.: Unterscheidung zwischen extrinsischen und intrinsischen IAs

5.3. Entscheidung über die Zuständigkeit der verteilten TOSCA-Runtime Knoten

In Abschnitt 5.2 wurde beschrieben, dass in TOSCA, zur Identifizierung von konkreter Hardware, Properties verwendet werden können. Um das Konzept einer verteilten TOSCA-Runtime umsetzen zu können, muss anhand dieser Properties bei der Erzeugung einer Anwendungsinstanz entschieden werden, welcher TOSCA-Runtime Knoten für welche Node bzw. Relationship Templates der Anwendung zuständig ist. Die Properties eines Service Templates bzw. eines konkreten Node oder Relationship Templates haben jedoch keine standardisierte Semantik [OASd]. Stattdessen können Properties beliebige Informationen beinhalten. Für eine verteilte TOSCA-Runtime ist damit unklar, wie sie die Informationen nutzen kann, um eine Entscheidung über die Verteilung der Komponenten der Anwendung zu treffen. Dabei sind verschiedene Ansätze denkbar.

Zum Beispiel kann eine Property mit einer bestimmten Semantik eingeführt werden, die für die Entscheidung verwendet wird. Dafür könnte eine *IP-Property* definiert werden und zusätzlich könnte für jeden TOSCA-Runtime Knoten beim Start ein bestimmter *IP-Bereich* festgelegt werden, für den der Knoten verantwortlich ist. Somit wäre immer der TOSCA-Runtime Knoten für ein Node Template zuständig, der über die IP-Property des Node Templates in seinem IP-Bereich verfügt. Der Nachteil dieses Vorgehens ist, dass dafür jedes Node Template bzw. mindestens ein Node Template eines Topologie Stapels die IP-Property enthalten muss. Damit werden die Modellierer sehr stark eingeschränkt und es können Fälle auftreten, in denen keine sinnvolle IP-Property definiert werden kann. Dies ist zum Beispiel dann der Fall, wenn durch die Anwendung eine virtuelle Maschine in der Cloud erstellt

wird, da die IP der virtuellen Maschine erst nach der Erzeugung zugewiesen wird. Ein weiterer Nachteil ist, dass durch diesen Ansatz die Portabilität von Anwendungen verloren gehen kann, da unterschiedliche Implementierungen einer verteilten TOSCA-Runtime unterschiedliche Properties als Entscheidungsmerkmal verwenden können.

Ein zweiter Ansatz kann sein, in jede erstellte Topologie spezielle *Tags* oder *Labels* einzufügen [SBKL17][OASd]. Diese Tags könnten zum Beispiel direkt definieren, auf welchem TOSCA-Runtime Knoten welcher Teil der Anwendung verwaltet werden soll. Dabei muss jedoch ebenfalls zunächst ein Schema standardisiert werden, um die Portabilität von Anwendungen beizubehalten. Außerdem können Tags lediglich zur Modellierungszeit der Anwendung definiert werden, wodurch die Entscheidung über die Verteilung ebenfalls bereits dort getroffen werden muss. Dies würde die Wiederverwendbarkeit der Anwendung einschränken.

Im Konzept dieser Arbeit wird stattdessen ein Ansatz verfolgt, der auf *Instanzdaten* basiert. Der Vorteil dieses Ansatzes ist, dass damit sowohl die Portabilität, als auch die Wiederverwendbarkeit von CSARs und den darin enthaltenen Anwendungen erhalten werden kann. Außerdem können CSARs für die Nutzung in zentralisierten TOSCA-Runtimes und verteilten TOSCA-Runtimes gleich modelliert werden. Dies bedeutet, dieselbe CSAR kann in IoT Szenarien, in denen eine verteilte TOSCA-Runtime benötigt wird und anderen Szenarien ohne verteilte TOSCA-Runtime verwendet werden. Unter Instanzdaten versteht man die Menge aller Daten, die in einer TOSCA-Runtime über in der Erstellung befindliche, erstellte oder bereits wieder gelöschte Instanzen verschiedener CSARs gespeichert werden [OASc]. Die Instanzdaten beinhalten zum Beispiel den Zustand aller Instanzen und die aktuellen Werte der Properties der verschiedenen Node und Relationship Templates einer Instanz [OASe].

Die Grundidee dieses Konzepts ist es, ein sogenanntes *Property Matching* durchzuführen, wenn eine Entscheidung über die Verteilung eines Node bzw. Relationship Templates getroffen werden muss. Dabei wird für ein Node Template das unterste Node Template seines Topologie Stapels gesucht. Dieses Node Template wird als *Infrastructure Node Template* bezeichnet. Um das Infrastructure Node Template zu ermitteln, können die ausgehenden *hostedOn*, *deployedOn* und *dependsOn* Relationship Templates verfolgt werden, bis das letzte Node Template erreicht ist. Wenn das Infrastructure Node Template für ein Relationship Template gesucht wird, kann dagegen zunächst das Source Node Template, also das Node Template, das der Ausgangspunkt der Relationship ist, ermittelt werden und danach dasselbe Verfahren angewendet werden. Anschließend wird das Property Matching zwischen dem Infrastructure Node Template und allen Node Templates in den Instanzdaten der TOSCA-Runtime durchgeführt. Dabei wird ein Matching erzielt, wenn der Node Type beider Node Templates derselbe ist und beide Node Templates exakt die gleichen Properties besitzen. Sollte die TOSCA-Runtime ein Matching erzielen, bedeutet dies, dass sie für das Node bzw. Relationship Template, für welches das Matching ausgeführt wurde, zuständig ist. Damit müssen alle zugehörigen IAs im Environment der TOSCA-Runtime deployt werden. Wird dagegen kein Matching erzielt, kann bei allen anderen TOSCA-Runtime Knoten des verteilten Systems eine Anfrage gestellt werden, ob diese für das Node bzw. Relationship Template zuständig sind. Dazu wird eine Nachricht auf dem MQTT Broker der

TOSCA-Runtime veröffentlicht, die den Node Type und die Properties des Infrastructure Node Templates für das Matching enthält. Anschließend kann die TOSCA-Runtime auf eine Antwort warten und, falls sie diese bekommt, das Deployment der IAs beim entsprechenden TOSCA-Runtime Knoten durchführen. Wenn sie dagegen in einem festgelegten Timeout keine Antwort erhält, kann die Instanzerzeugung entweder mit einem Fehler abgebrochen werden, oder es kann ein lokales Deployment der IAs als Standard durchgeführt werden.

Um den Ansatz des Property Matchings umsetzen zu können, müssen bei jedem TOSCA-Runtime Knoten des verteilten Systems Instanzdaten für die Devices bzw. die Hardware vorliegen, für die sie zuständig sind. Wenn also ein TOSCA-Runtime Knoten für ein Raspberry Pi mit der IP 192.168.1.1 zuständig ist, muss in seinen Instanzdaten ein Node Template vom Node Type Raspberry Pi und mit der IP als Property existieren. Eine Möglichkeit ist es, die Instanzdaten für die vorhandene Hardware an jedem Knoten manuell anzulegen. Da dies jedoch einen zusätzlichen Aufwand darstellt, ist es wünschenswert, die Erzeugung der Instanzdaten zu automatisieren. Dies kann beispielsweise im Zuge einer sogenannten *Device Discovery* durchgeführt werden [BSMS01][DBN14]. Dabei kann jede TOSCA-Runtime anhand verschiedener Protokolle ermitteln, welche Devices und Hardware in ihrem Netzwerk vorhanden sind und anschließend automatisch die passenden Instanzdaten anlegen. Ein solches Device Discovery Verfahren ist jedoch nicht Teil dieser Arbeit und kann als zukünftige Erweiterung umgesetzt werden.

In Algorithmus 5.1 wird das beschriebene Verfahren zur Entscheidung der Verteilung von Node und Relationship Templates über die verschiedenen TOSCA-Runtime Knoten als Algorithmus dargestellt. Dabei wird davon ausgegangen, dass die Funktion *DeploymentDistributionDecision* in der TOSCA-Runtime, in der die CSAR hochgeladen wurde, für jedes Node und Relationship Template aufgerufen wird, sobald eine Instanz der CSAR erzeugt werden soll. Somit sind die Zuständigkeiten klar und die IAs für die Instanzerzeugung können deployt und anschließend aufgerufen werden.

Der Algorithmus erhält als Eingabe das Node bzw. Relationship Template, für das die Verteilungsentscheidung getroffen werden soll (Zeile 1). Falls es sich bei der Eingabe um ein Relationship Template handelt, wird das Source Node Template für die weitere Verarbeitung bestimmt (Zeilen 2-4). Anschließend wird das Infrastructure Node Template ermittelt, indem entlang der Infrastruktur Kanten die Topologie nach unten iteriert wird (Zeilen 5-14). Mit dem Infrastructure Node Template kann das lokale Instanzdaten Matching durchgeführt werden (Zeile 15). Das bedeutet, es werden der Node Type und die Properties des Infrastructure Node Templates bestimmt (Zeilen 29-30) und mit dem Node Type und den Properties aller Instanzdaten verglichen (Zeilen 31-36). Wenn dabei ein Matching erzielt wird, gibt der Algorithmus den Hostnamen des lokalen TOSCA-Runtime Knotens zurück (Zeile 16). Das bedeutet, dass in diesem Fall die zugehörigen IAs lokal deployt werden. Wurde dagegen kein Matching erzielt, wird mit dem Instanzdaten Matching in den anderen Knoten der verteilten TOSCA-Runtime fortgefahren. Dazu wird eine Anfrage mit dem Infrastructure Node Template für alle anderen TOSCA-Runtime Knoten über MQTT veröffentlicht (Zeile 18). Anschließend wartet der Algorithmus auf eine Antwort der anderen TOSCA-Runtime Knoten (Zeile 19). Dabei wird ein Timeout gestartet, nachdem davon

Algorithmus 5.1 Verteilungsalgorithmus

```
1:  function deploymentDistributionDecision(template)
2:      if template instanceOf RelationshipTemplate then
3:          template ← getSourceNodeTemplate(template)
4:      end if
5:      repeat
6:          foundNewTemplate ← false
7:          for all relationshipTemplate ∈ getOutgoingRelations(template) do
8:              if isInfrastructureEdge(relationshipTemplate) then
9:                  foundNewTemplate ← true
10:                 template ← getTargetNodeTemplate(relationshipTemplate)
11:                 break
12:             end if
13:         end for
14:     until ¬foundNewTemplate
15:     if matchingInstanceData(template) then
16:         return localhost
17:     else
18:         sendRequestViaMQTT(template)
19:         response ← receiveResponseWithTimeout(timeout)
20:         if response ≠ null then
21:             return getHostNameFromMessage(response)
22:         else
23:             return localhost
24:         end if
25:     end if
26: end function
27:
28: function matchingInstanceData(template)
29:     type ← getNodeType(template)
30:     properties ← getProperties(template)
31:     for all instance ∈ getNodeTemplatesFromInstanceData() do
32:         if type = getNodeType(instance) and
33:             properties = getProperties(instance) then
34:             return true
35:         end if
36:     end for
37:     return false
38: end function
```

ausgegangen wird, dass das Matching bei allen anderen Knoten ebenfalls nicht erfolgreich war. Das Timeout kann je nach Kapazität der Hardware, auf der die TOSCA-Runtime Knoten ausgeführt werden, kürzer oder länger gewählt werden, da davon die Berechnungszeit für das Matching abhängt. Wenn der Algorithmus innerhalb des Timeouts eine Antwort empfängt, wird der Hostname des antwortenden TOSCA-Runtime Knotens extrahiert und vom Algorithmus zurückgegeben (Zeile 21). Somit müssen die zum Node bzw. Relationship Template gehörenden IAs bei der antwortenden TOSCA-Runtime deployt werden. Wenn der Timeout ohne Antwort abläuft, hat die TOSCA-Runtime keine Information darüber, wo die IAs bereitgestellt werden müssen. Deshalb wird als Standardausgabe der lokale Hostname zurückgegeben (Zeile 23). Damit kann jedoch nicht garantiert werden, dass die Instanzerzeugung erfolgreich durchgeführt werden kann.

5.4. Übersicht über das Gesamtkonzept

In den vorigen Abschnitten dieses Kapitels wurde gezeigt, dass die Nutzung einer verteilten TOSCA-Runtime die Probleme der automatischen Provisionierung von IoT Anwendungen beheben kann. Außerdem wurde beschrieben, warum zur Umsetzung einer verteilten TOSCA-Runtime das IA Deployment verzögert werden muss. Weiterhin wurde ein Algorithmus vorgestellt, mit dem entschieden werden kann, welcher TOSCA-Runtime Knoten für welche Teile einer konkreten Anwendungsinstanz verantwortlich ist. Um das Konzept zu vervollständigen, wird in diesem Abschnitt präsentiert, wie die Abläufe in einer TOSCA-Runtime angepasst werden müssen, um sie als verteilte TOSCA-Runtime einsetzen zu können.

Abbildung 5.7 zeigt den neuen Ablauf beim Deployment einer CSAR in einem Knoten der verteilten TOSCA-Runtime als BPMN Diagramm. Im Vergleich zum alten Ablauf (siehe Abbildung 4.1) wurde lediglich das Deployment der Implementation Artifacts aus dem Ablauf entfernt. Das bedeutet, dass nach wie vor, nach dem Upload der CSAR, die TOSCA Definitionen extrahiert und in einem Repository gespeichert werden. Daraufhin werden die Artefakte aus der CSAR in geeigneten Speichermöglichkeiten abgelegt. Ein Deployment der IAs findet anschließend jedoch nicht statt, da der zuständige TOSCA-Runtime Knoten noch nicht ermittelt werden kann (siehe Abschnitt 5.2). Stattdessen werden als nächstes die Pläne aus der CSAR extrahiert bzw. im deklarativen Fall generiert. Abschließend werden die Pläne in der Process Engine deployt. Da die Endpunkte der IAs zu diesem Zeitpunkt nicht bekannt sind, können diese auch nicht an die Tasks der Pläne gebunden werden. Stattdessen wird an alle Tasks, die ein IA aufrufen, der Endpunkt der TOSCA-Runtime übergeben. Somit wird die TOSCA-Runtime aktiviert, sobald ein IA aufgerufen werden soll und kann sich anschließend um die Verteilungsentscheidung und das Deployment im korrekten TOSCA-Runtime Knoten kümmern. Damit ist das CSAR Deployment abgeschlossen und der Nutzer wird darüber informiert.

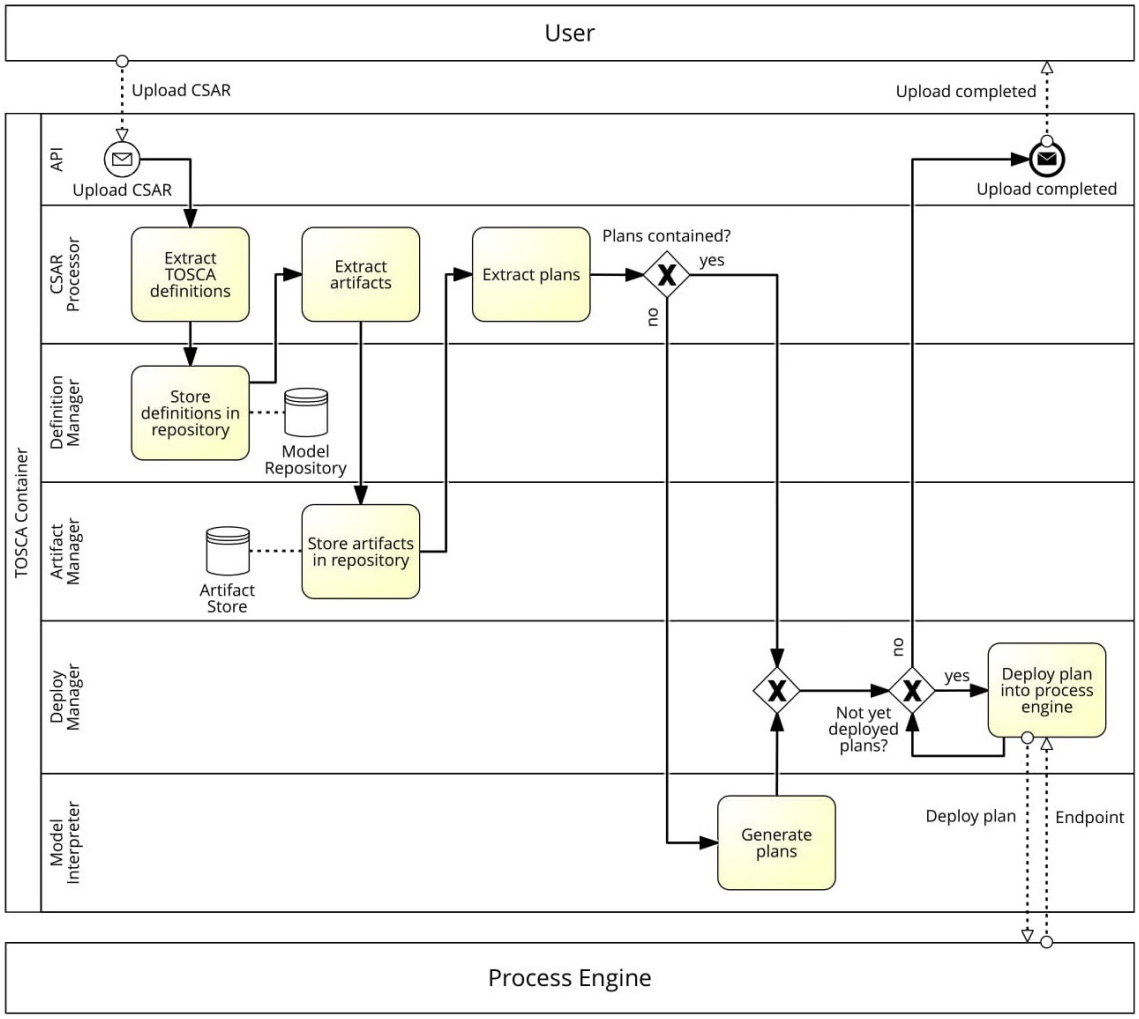

Abbildung 5.7.: Konzept: CSAR Deployment in einer TOSCA-Runtime

Im Gegensatz zum Deployment einer CSAR, bei der noch keine Kommunikation zwischen verschiedenen TOSCA-Runtime Knoten nötig ist, wird bei der Erzeugung von Anwendungsinstanzen die Kommunikation und Kooperation benötigt. Deshalb ändert sich der Ablauf für die Instanzerzeugung bei einer verteilten TOSCA-Runtime deutlich stärker als der des CSAR Deployments. Abbildung 5.8 stellt den neuen Prozess der Instanzerzeugung als BPMN Diagramm dar. Der Ablauf wird durch einen Befehl des Nutzers über die API gestartet. Dabei übergibt der Nutzer alle benötigten Parameter an den TOSCA Container. Anschließend erzeugt der TOSCA Container in der Datenbank ein Instanzobjekt und startet den korrespondierenden Build Plan in der Process Engine. Bis zu diesem Zeitpunkt ist der Ablauf noch gleich wie bei einer zentralisierten TOSCA-Runtime. Anstatt auf die Terminierung des Build Plans zu warten, betritt der TOSCA Container jetzt allerdings einen Zustand, in dem er auf verschiedene, von der Process Engine gesendete, Nachrichten reagieren kann. Dies wird durch das Event-basierte Gateway dargestellt. Der Ablauf in der Process Engine ist ebenfalls exakt gleich wie bei einer zentralisierten TOSCA-Runtime. Der Unterschied

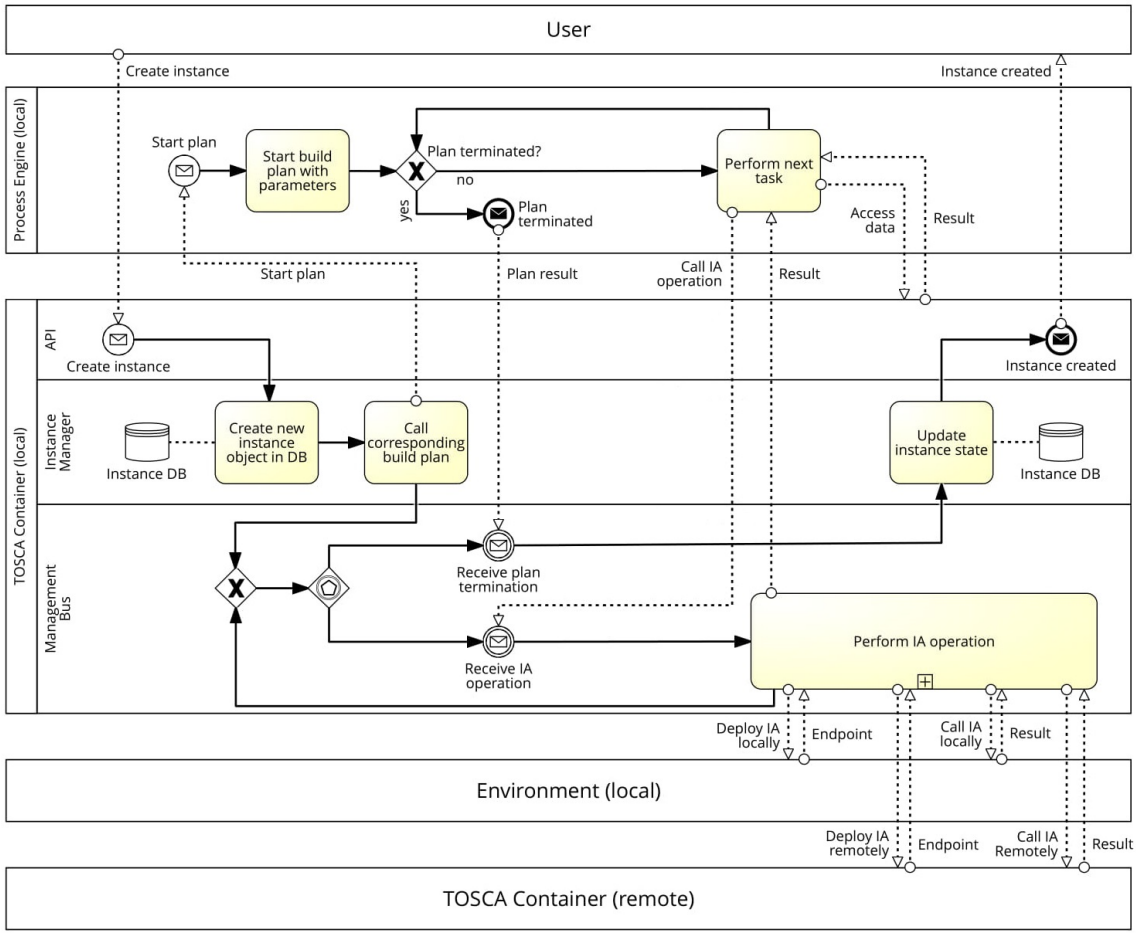

Abbildung 5.8.: Konzept: Instanzerzeugung in einer TOSCA-Runtime

ist allerdings, dass beim Deployment des Plans anstelle der Endpunkte der verschiedenen IAs ein Endpunkt des Management Busses des TOSCA Containers übergeben wurde. Das bedeutet, wenn ein Plan ein IA aufrufen möchte, sendet er den Aufruf zusammen mit den Parametern an den Management Bus, statt direkt an das IA. Wenn der Management Bus einen Aufruf für ein IA erhält, trifft er die Entscheidung, welcher TOSCA Container für das IA zuständig ist. Danach sorgt er für das geeignete Deployment und ruft schließlich das IA im lokalen Environment oder, mittels Zusammenarbeit der Management Busse verschiedener Container, in einem entfernten Environment auf. Dieser Prozess wurde aus Gründen der besseren Übersichtlichkeit in Abbildung 5.8 als zugeklappter Subprozess dargestellt. Weitere Details dazu finden sich in Abbildung 5.9 und dem folgenden Absatz. Nachdem alle Tasks des Build Plans erfolgreich abgeschlossen wurden, terminiert der Plan und sendet dem Management Bus eine Nachricht mit den Ausgabeparametern. Daraufhin werden die Instanzdaten im TOSCA Container geupdatet und der Nutzer erhält die Informationen über die neu erstellte Anwendungsinstanz.

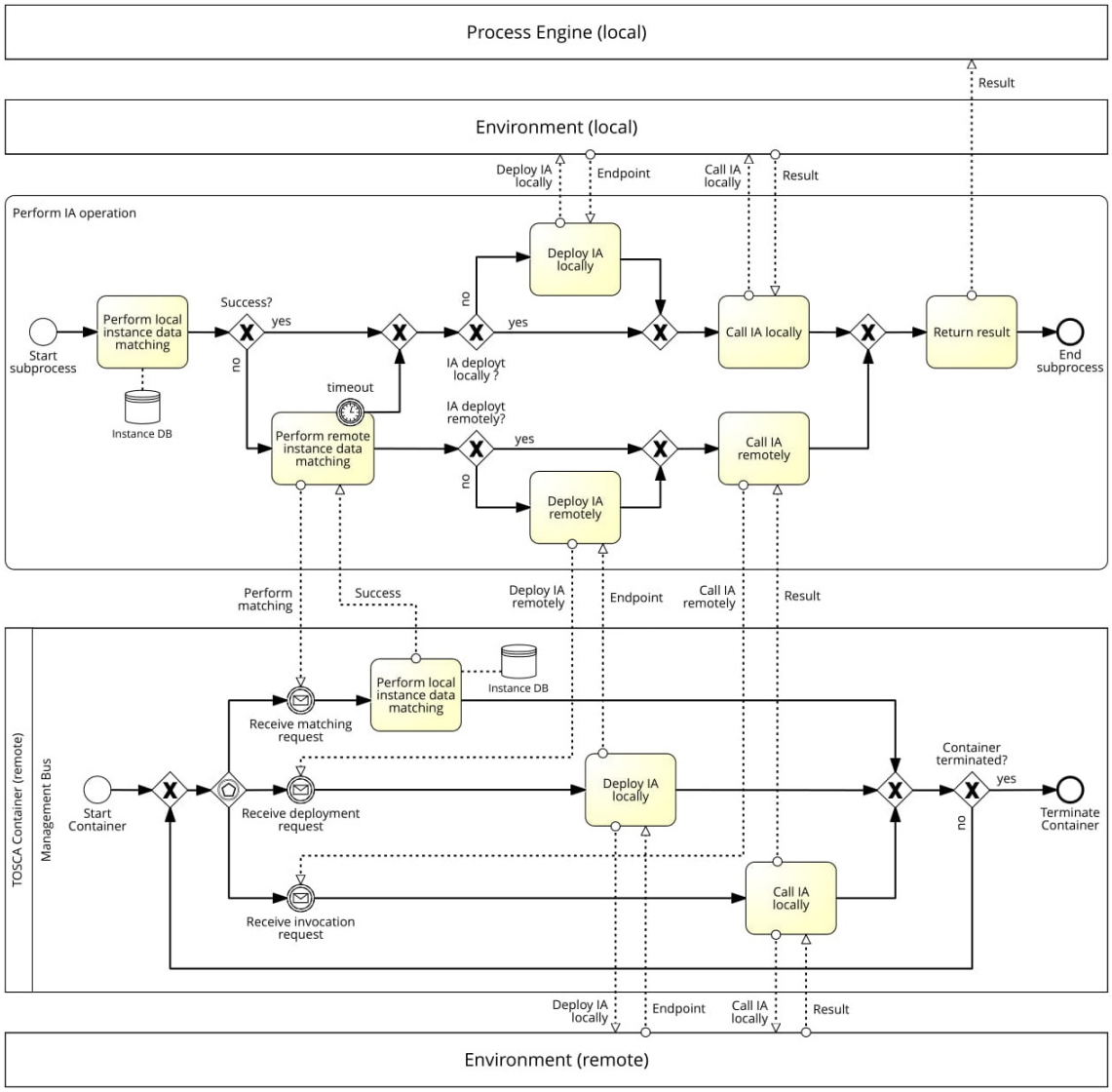

Abbildung 5.9.: Konzept: Subprozess mit IA Aufrufen

In Abbildung 5.9 ist der Subprozess, der zum Aufruf von IA Operationen ausgeführt wird, dargestellt. Jeder TOSCA Container, der an der verteilten TOSCA-Runtime teilnimmt, startet beim Hochfahren einen Thread, der die Anfragen der anderen Container bearbeitet. Dies kann man beim remote Container in der Abbildung sehen, der nach dem Start, mittels des Event-basierten Gateways, auf Nachrichten der anderen Container in der verteilten TOSCA-Runtime wartet. Jeder Nachrichtenaustausch zwischen zwei TOSCA Containern findet dabei über einen MQTT Broker statt, auf den beide TOSCA Container Zugriff haben, um Firewall Probleme zu vermeiden (siehe Abschnitt 5.1). Der Broker wurde jedoch aus Gründen der besseren Übersichtlichkeit nicht in den Ablauf integriert. Nachdem der Subprozess für den Aufruf eines IAs gestartet wurde, wird zunächst das lokale Instanzdaten Matching durchgeführt. Falls dies erfolgreich ist, wird das IA lokal verwaltet (oberer Teil des Subprozesses). Wenn das Matching nicht erfolgreich ist, wird das Instanzdaten

Matching auf den anderen Containern ausgeführt. Dazu wird eine Nachricht mittels MQTT veröffentlicht. Die empfangenden Container führen das Matching aus und senden, falls es erfolgreich war, eine Antwort zurück. Anschließend warten die Container wieder auf neue Anfragen. Wenn innerhalb eines Timeouts keine Antwort eintrifft, wird das IA ebenfalls lokal verwaltet. Dazu wird überprüft, ob das IA bereits im lokalen Environment deployt wurde und falls nicht, wird das Deployment durchgeführt. Anschließend wird der Aufruf an das IA weitergeleitet und das Ergebnis an die Process Engine gesendet. Damit ist der Aufruf des Subprozesses beendet. Für den Fall, dass das IA von einem anderen TOSCA Container verwaltet wird, wird ebenfalls zunächst überprüft, ob das IA in diesem Container bereits deployt wurde. Dies kann durch die Daten der lokalen Endpunkte-Datenbank geschehen, ohne mit dem anderen Container kommunizieren zu müssen. Falls das IA noch deployt werden muss, wird es mittels MQTT an den anderen Container gesendet, der es in seinem Environment deployt. Anschließend kann das IA aufgerufen werden, indem der Subprozess die Eingabeparameter an den Management Bus des verantwortlichen Containers sendet. Nach erfolgreichem Aufruf, erhält der lokale TOSCA Container die Ausgabeparameter und leitet sie, wie beim lokalen Aufruf, an die Process Engine weiter, wodurch der Subprozess beendet wird. Der Thread des remote Containers bleibt dagegen weiter aktiv und wartet auf neue Anfragen, bis der gesamte TOSCA Container terminiert wird.

6. Implementierung

In diesem Kapitel wird die prototypische Implementierung des in dieser Arbeit entwickelten Konzepts zur verteilten Provisionierung von IoT Anwendungen vorgestellt. Die Grundlage der Implementierung bildet das OpenTOSCA Ökosystem und konkret die TOSCA-Runtime OpenTOSCA Container. Der Großteil der Implementierung zielt darauf ab, den OpenTOSCA Container zu einer verteilten TOSCA-Runtime weiterzuentwickeln. Dies bedeutet, es muss die Logik umgesetzt werden, die zur Kommunikation zwischen einzelnen OpenTOSCA Container Knoten benötigt wird. Außerdem muss die Verteilungsentscheidung anhand des Instanzdaten Matching implementiert werden. Neben den Erweiterungen des OpenTOSCA Containers soll das gesamte OpenTOSCA Ökosystem auf typischen IoT Devices, wie zum Beispiel Raspberry Pis, ausführbar gemacht werden. Somit muss in einer IoT Umgebung kein extra Server eingefügt werden. Das OpenTOSCA Ökosystem kann vielmehr direkt auf den IoT Devices ausgeführt werden. Ein Problem das hierbei auftritt, sind die unterschiedlichen Prozessorarchitekturen, die eine Ausführung erschweren können.

6.1. Neue OpenTOSCA Container Architektur

Um die in dieser Arbeit entwickelten neuen Abläufe innerhalb einer TOSCA-Runtime (siehe Abschnitt 5.4) im OpenTOSCA Container umzusetzen, muss die Architektur angepasst werden. Abbildung 6.1 stellt die neue Architektur des OpenTOSCA Containers dar. Im Vergleich zur alten Architektur (siehe Abbildung 2.8) wurde jeweils eine Komponente entfernt, eine Komponente hinzugefügt und eine Komponente um ein zweites Plug-In System erweitert.

Das Deployment der IAs wird im neuen Konzept vom Management Bus übernommen. Deshalb wird dieser um ein zweites Plug-In System erweitert. Das alte Plug-In System (rechts) beinhaltet Plug-Ins für den Aufruf verschiedener IA bzw. Plan Arten, beispielsweise mittels SOAP oder REST und wird unverändert beibehalten. Im neuen Plug-In System (links) sind dagegen Plug-Ins für das Deployment der IAs enthalten. Diese ähneln den Plug-Ins der IA-Engine sehr stark und können zum Beispiel für das Deployment von WAR oder Script Artefakten verwendet werden. Allerdings müssen die Plug-Ins das Datenformat und den Aufrufmechanismus des Management Busses unterstützen. Das bedeutet insbesondere, dass die Kommunikation zwischen dem Bus und den Plug-Ins mittels Camel (siehe Abschnitt 2.4) durchgeführt werden muss.

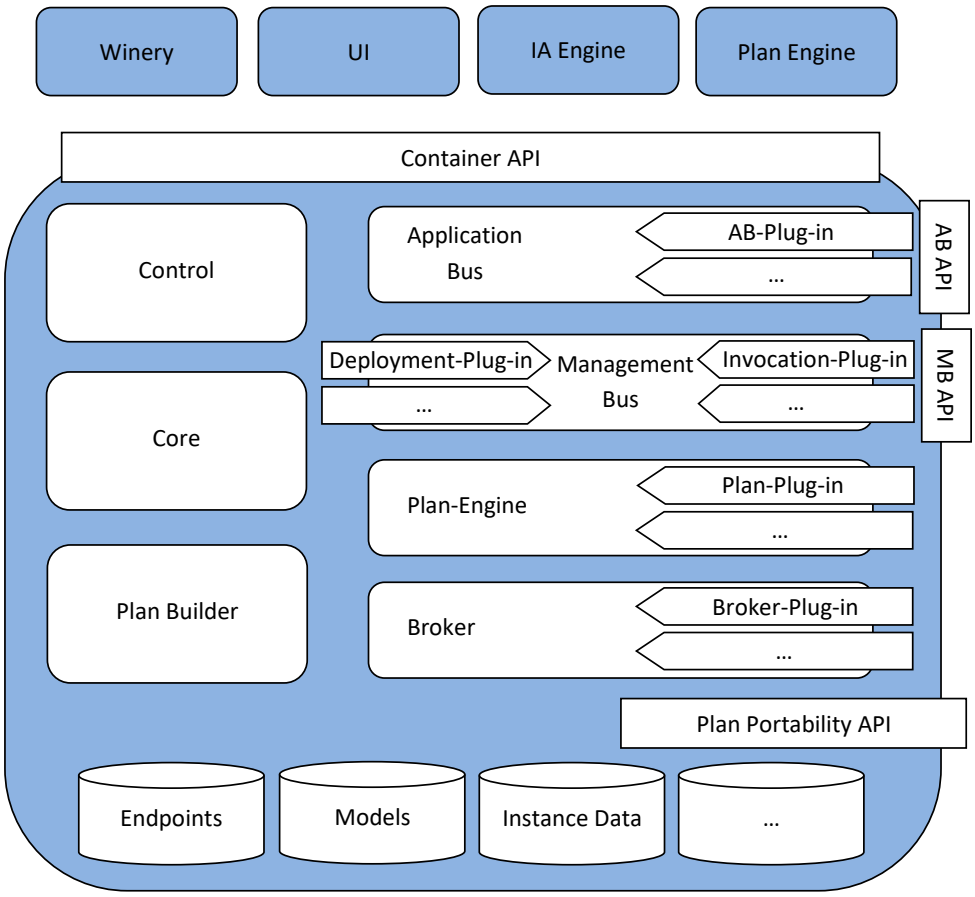

Abbildung 6.1.: Neue OpenTOSCA Container Architektur

Da das Deployment der IAs in der neuen Architektur vom Management Bus vorgenommen werden kann, wird die IA-Engine als Komponente nicht mehr benötigt und kann aus dem OpenTOSCA Container komplett entfernt werden. Bei der neu hinzugefügten Komponente handelt es sich um die *Broker-Komponente*. Die Broker-Komponente kann genutzt werden, um Broker verschiedener Arten innerhalb des OpenTOSCA Containers zu starten. Von der Broker-Komponente erzeugte Broker können zum Beispiel für die Kommunikation zwischen verschiedenen OpenTOSCA Container Knoten oder für die Kommunikation von mittels OpenTOSCA provisionierten Anwendungen verwendet werden. Derzeit ist es lediglich möglich einen MQTT Broker innerhalb des OpenTOSCA Containers zu starten. Dieser wird für die Kommunikation zwischen verschiedenen OpenTOSCA Container Knoten eingesetzt, falls diese als verteilte TOSCA-Runtime verwendet werden sollen. Die gesamte Logik für die Verteilungsentscheidung und die Kommunikation zwischen OpenTOSCA Container Knoten wird jedoch ebenfalls im Management Bus angesiedelt, sodass hierfür keine zusätzliche Komponente benötigt wird.

6.2. Kommunikation zwischen OpenTOSCA Container Knoten

Damit die in Abschnitt 5.1 und Abschnitt 5.4 beschriebene Kommunikation zwischen Knoten einer verteilten TOSCA-Runtime im OpenTOSCA Container umgesetzt werden kann, wird mindestens ein aktiver MQTT Broker benötigt. Grundsätzlich sind drei verschiedene Ansätze zur Bereitstellung des Brokers denkbar. Es könnte erstens davon ausgegangen werden, dass ein global zugreifbarer Broker vorhanden ist bzw. dass dieser manuell erstellt wird. Die Adresse sowie die Zugangsdaten kann somit jedem OpenTOSCA Container beim Start übergeben werden. Eine zweite Möglichkeit ist es, die OpenTOSCA Docker Compose Datei[1] so zu erweitern, dass dort zusammen mit den anderen Docker Containern immer ein Container mit einem MQTT Broker gestartet wird. Letzlich ist es möglich, einen MQTT Broker direkt als OSGi Bundle innerhalb des OpenTOSCA Container Projekts einzubinden. Dazu kann Moquette[2] verwendet werden, eine Open Source Java Implementierung eines MQTT Brokers. Somit würde beim Start jedes OpenTOSCA Containers immer ein lokaler Broker erstellt, der für die Kommunikation verwendet werden kann.

Die manuelle Bereitstellung des Brokers ist aufgrund des zusätzlichen Aufwandes und des nötigen Wissens nicht geeignet. Bei der Erstellung des Brokers mittels Docker Compose ergibt sich das Problem, dass der Broker nur dann vorhanden ist, wenn das OpenTOSCA Ökosystem mittels Docker gestartet wird. Da auch andere Konfigurationen denkbar sind, ist dies ebenfalls keine brauchbare Option. Deshalb wurde die dritte Option umgesetzt, bei der Moquette in das OSGi Projekt eingebunden wird. Dazu wurde ein Bundle erstellt, das seinerseits beim Start des OpenTOSCA Containers einen Broker erzeugt, der auf einem konfigurierbaren Port arbeitet. Zusätzlich ist es möglich, einen Benutzernamen und ein Passwort für die Authentifizierung beim Broker zu definieren. Listing 6.1 zeigt einen Ausschnitt der *Config.ini* Datei, über die der OpenTOSCA Container konfiguriert werden kann. Im Ausschnitt sind alle Optionen mit Bezug zum internen MQTT Broker abgebildet. Bei Verwendung der dargestellten Konfiguration würde der Broker also den Standard Port für MQTT 1883 verwenden und den Zugriff mit dem Benutzernamen und Passwort „admin“ zulassen.

Listing 6.1 Konfiguration des Moquette MQTT Brokers

```
1 org.opentosca.container.broker.mqtt.port=1883
2 org.opentosca.container.broker.mqtt.username=admin
3 org.opentosca.container.broker.mqtt.password=admin
```

Die gesamte Kommunikation zwischen OpenTOSCA Container Knoten wird über die Management Bus Komponenten abgewickelt (siehe Abschnitt 6.1). Da der Managment Bus auf Camel (siehe Abschnitt 2.4) basiert, müssen für die Kommunikation zwischen

[1] https://github.com/OpenTOSCA/opentosca-docker
[2] https://projects.eclipse.org/projects/iot.moquette

den Knoten ebenfalls geeignete Camel Routes definiert werden. Dabei wird eine Route benötigt, um Anfragen und Antworten an andere Knoten zu senden. Als Zweites muss eine Route erstellt werden, die zum Empfangen von Antworten auf eigene Anfragen genutzt wird. Außerdem muss für jeden anderen Knoten, von dem Anfragen empfangen werden sollen, eine zusätzliche Route definiert werden. Der Master Knoten in einer Master/Slave Architektur würde also nur zwei Routes benötigen. Wird dagegen eine Peer-to-Peer Architektur mit drei Knoten verwendet, braucht jeder Knoten fünf aktive Camel Routes. Die drei verschiedenen Arten von Routes werden im Folgenden genauer vorgestellt.

Listing 6.2 stellt die Konfiguration der Camel Route zum Senden von Anfragen und Antworten an andere OpenTOSCA Container Knoten dar. In den Zeilen 1-4 wird zunächst der Endpoint definiert, über den die Nachrichten mittels MQTT veröffentlicht werden sollen. Dass es sich beim Endpoint um einen MQTT Endpoint handelt, kann durch das Schema der URI erkannt werden (siehe Abschnitt 2.4). Anhand der Optionen der URI wird die URL des MQTT Brokers als Host, das MQTT Topic, der Benutzername und das Passwort für die Authentifizierung, sowie der benötigte Quality of Service (QoS) konfiguriert. Dabei werden für die MQTT URL und das Topic Platzhalter in der URI definiert, die, sobald ein Exchange durch die Route gesendet wird, dynamisch durch Header Felder des Exchanges ersetzt werden. Somit kann dieselbe Route für eine beliebige Anzahl an MQTT Brokern verwendet werden, anstatt immer eine neue Route mit fest definierter URL und Topic erstellen zu müssen. Für den Benutzernamen und das Passwort wurden lokale Variablen eingefügt, deren Zuweisungen zur besseren Lesbarkeit nicht dargestellt sind. Diese Variablen enthalten jedoch den Benutzernamen und das Passwort, die beide in der Config Datei des OpenTOSCA Containers definiert sind (siehe oben). Falls die verschiedenen Broker, die in der verteilten TOSCA-Runtime eingesetzt werden, unterschiedliche Zugangsdaten verwenden, könnten die Zugangsdaten ebenfalls per Platzhalter eingefügt werden und später dynamisch ersetzt werden. Der Einfachheit halber wird jedoch davon ausgegangen, dass für alle Broker dieselben Zugangsdaten verwendet werden sollen. Als QoS wird „ExactlyOnce“ definiert, was QoS Level 2 nach der MQTT Spezifikation entspricht (siehe Abschnitt 2.2). Das bedeutet, alle Nachrichten, die gesendet werden, werden auch garantiert exakt einmal ausgeliefert [OASa]. Dies ist bedeutsam, da verlorene Nachrichten als Antwort auf eine Anfrage zum Instanzdaten Matching beispielsweise problematisch sind. Der anfragende Knoten geht damit davon aus, dass kein anderer Knoten passende Instanzdaten vorliegen hat und gibt entweder einen Fehler zurück oder nimmt als default ein lokales Deployment der IAs vor, was eventuell zu den Firewall Problemen führen kann, die das neue Konzept vermeiden soll.

Von Zeile 6 bis 14 wird anschließend die eigentliche Camel Route konfiguriert. Zeile 6 definiert dazu als erstes den Endpoint, der den Eintrittspunkt von Camel Exchanges in die Route definiert. Dieser verwendet das Schema *„direct“* und den Kontextpfad „SendMQTT“. Endpoints mit „direct“ als Schema können verwendet werden, um lokale Endpoints im Camel Context, der Laufzeitumgebung für Camel Routes, zu erstellen, die über einen eindeutigen Namen identifiziert werden [IA18]. Somit ist es zum Beispiel möglich, lokale Routes miteinander zu verknüpfen oder per Java Code erstellte Exchanges über den Endpoint in eine Route einzufügen bzw. Exchanges aus einer Route zu empfangen. Wenn eine Nachricht per MQTT veröffentlicht werden soll, können also alle Daten in ein Exchange

Listing 6.2 Camel Route zum Senden von Anfragen und Antworten

```
1  String producerEndpoint = "mqtt:send?host=${header.MQTTBROKER_URL}"
2    + "&userName=" + username + "&password=" + password
3    + "&publishTopicName=${header.MQTT_TOPIC}"
4    + "&qualityOfService=ExactlyOnce";
5
6  from("direct:SendMQTT")
7    .process(outgoingProcessor)
8    .doTry()
9      .marshal(jaxb)
10     .recipientList(simple(producerEndpoint))
11   .endDoTry()
12   .doCatch(Exception.class)
13     .log(LoggingLevel.ERROR, LOG, noMarshalling)
14   .end();
```

eingefügt und anschließend über den „direct“ Endpoint an die Route übergeben werden. In Zeile 7 werden die Exchanges vom *OutgoingProcessor* verarbeitet. Der OutgoingProcessor liest die Header Felder der Exchanges und fügt sie in den Body der beinhalteten Nachricht ein. Dies ist nötig, da MQTT in der Version v3.1.1 keine benutzerdefinierten Header Felder zulässt und die Informationen aus den Headern ansonsten verloren gehen würden (siehe Abschnitt 2.2). Anschließend wird in Zeile 9 ein Marshalling der Nachricht im Exchange durchgeführt, um das XML-basierte Datenformat zu erhalten, das zur Kommunikation zwischen OpenTOSCA Container Knoten verwendet und in Abschnitt 6.3 vorgestellt wird. Wenn das Marshalling erfolgreich war, kann das Exchange in Zeile 10 durch den oben beschriebenen MQTT Endpoint veröffentlicht werden und die Bearbeitung der Route ist abgeschlossen. Für den Fall, dass beim Marshalling ein Fehler auftritt, da zum Beispiel Daten nicht konvertierbar sind, wurde von Zeile 8 bis 11 ein „Try“-Block eingefügt, bei dem mögliche Exceptions abgefangen und an den „Catch“-Block von Zeile 12 bis 14 übergeben werden. Im Fehlerfall wird eine entsprechende Fehlermeldung in das Log des OpenTOSCA Containers geschrieben (Zeile 13).

In Listing 6.3 ist die Camel Route zum Empfangen von Antworten anderer OpenTOSCA Container Knoten abgebildet. Der Endpoint, mit dem Exchanges in die Route eingefügt werden (Zeile 1-3), ist dabei nahezu gleich definiert wie der Endpoint, mit dem die Exchanges die vorige Route verlassen. So sind Benutzername, Passwort und QoS exakt gleich konfiguriert. Lediglich bei der Broker URL und beim Topic unterscheiden sich die Endpoints, da hier anstatt Header Felder des Exchanges ebenfalls lokale Variablen verwendet werden, die die Konfiguration des lokalen Moquette MQTT Brokers wiedergeben. Der Endpoint und damit die Route (Zeile 7) empfängt also alle Nachrichten, die auf dem lokalen MQTT Broker unter einem vordefinierten Antwort-Topic veröffentlicht werden. In allen Anfragen, die ein OpenTOSCA Container sendet, wurde das *„Return Address“* Pattern umgesetzt [HW04]. Das bedeutet, die Anfragen enthalten die Adresse des Brokers und

Listing 6.3 Camel Route zum Empfangen von Antworten anderer Knoten

```
1  String consumerEndpoint = "mqtt:response?host=" + url
2    + "&userName=" + username + "&password=" + password
3    + "&subscribeTopicNames=" + topic + "&qualityOfService=ExactlyOnce";
4
5  String producerEndpoint = "direct:Callback-${header.CORRELATION_ID}";
6
7  from(consumerEndpoint)
8    .doTry()
9      .unmarshal(jaxb)
10     .process(incomingProcessor)
11     .choice()
12       .when(header(correlationHeader).isNotNull())
13         .recipientList(simple(producerEndpoint))
14       .endChoice()
15       .otherwise()
16         .log(LoggingLevel.WARN, LOG, noCorrelation)
17       .endChoice()
18   .endDoTry()
19   .doCatch(DirectConsumerNotAvailableException.class)
20     .log(LoggingLevel.ERROR, LOG, noDirectException)
21   .doCatch(Exception.class)
22     .log(LoggingLevel.ERROR, LOG, noMarshalling)
23   .end();
```

das Topic, auf dem Antworten auf die Anfragen veröffentlicht werden sollen. Somit muss der empfangende OpenTOSCA Container Knoten diese nicht kennen und die Knoten des verteilten Systems können besser entkoppelt werden. Daraus folgt, dass der Name des Topics für die Route beliebig gewählt werden kann, wenn dieser Name ebenfalls als „Return Address“ in allen Anfragen verwendet wird.

Nach dem Empfang eines Exchanges wird ein Unmarshalling durchgeführt und die Header Felder, die vor der Übertragung über MQTT aus den Exchange Headern in den Body übertragen wurden, werden von einem *IncomingProcessor* für die weitere Verarbeitung wieder in die Exchange Header kopiert (Zeilen 9 und 10). Wenn dabei ein Fehler auftritt, wird dieser wie bei der vorigen Route in das Log eingetragen (Zeile 22). Anschließend wird überprüft, ob in den Headern des Exchanges eine *Correlation ID* definiert wurde (Zeile 12) und falls nicht, wird ein Fehler im Log vermerkt (Zeile 16). Unter einer Correlation ID versteht man eine ID, die vom Sender einer Anfrage generiert und in die Anfrage kopiert wird [HW04]. Der Empfänger der Anfrage muss diese ID in seine Antwort einfügen. So hat der Sender der Anfrage die Möglichkeit, den Kontext, der zum Erzeugen der Anfrage geführt hat, zu laden und die Antwort entsprechend zu verarbeiten. Correlation IDs werden vor allem bei Verwendung von asynchroner Kommunikation eingesetzt, um mehrere parallele Anfragen unterscheiden zu können. Wenn im Exchange eine Correlation ID definiert wurde,

verlässt das Exchange über den in Zeile 5 definierten Endpoint die Route (Zeile 13). Dieser Endpoint nutzt das „direct“ Schema und einen Kontextpfad, der aus dem Wort „Callback“ und der Correlation ID aus den Headern des Exchanges besteht.

Das bedeutet, wenn ein OpenTOSCA Container Knoten eine Anfrage an die anderen Knoten stellen möchte, generiert er zunächst eine Correlation ID. Anschließend kann ein Exchange mit der Anfrage erzeugt und die Correlation ID in die Header des Exchanges kopiert werden. Als nächstes muss ein Konsument erstellt werden, der am „direct“ Endpoint mit einem Kontextpfad nach obigem Schema auf Antworten wartet. Dann kann das Exchange über die Route aus Listing 6.2 gesendet werden. Wenn ein anderer Knoten eine Antwort auf die Anfrage abschickt, wird die Route aus Listing 6.3 aktiviert und die Antwort wird vom Konsument am „direct“ Endpoint empfangen und kann weiterverarbeitet werden. Ist an einem „direct“ Endpoint kein Konsument vorhanden, wenn ein Exchange dort eintrifft, wird eine „DirectConsumerNotAvailableException“ generiert. Dies kann zum Beispiel der Fall sein, wenn eine Antwort zu spät eintrifft und der Konsument wegen eines Timeouts bereits gelöscht wurde. Sofern dieser Fall eintritt, wird die Exception von der Route abgefangen und ein entsprechender Eintrag im Log erzeugt (Zeile 20).

Die dritte Art von Camel Routes wird benötigt, um Anfragen anderer OpenTOSCA Container Knoten zu empfangen und um diese bearbeiten zu können. In Listing 6.4 ist dargestellt, wie Routes dieser Art konfiguriert werden. Der Endpoint zum Empfangen der Anfragen (Zeile 1-3) ist exakt gleich definiert, wie der Endpoint zum Empfangen der Antworten aus der vorigen Route. Jedoch werden den Variablen für die URL und das Topic hier die Werte des MQTT Brokers des entfernten OpenTOSCA Container Knotens zugewiesen, von dem die Route Anfragen entgegennehmen soll. Konfiguriert werden können die Daten der anderen Knoten über die Config Datei des OpenTOSCA Containers. Das Unmarshalling und die Bearbeitung der Exchanges durch den IncomingProcessor wird ebenfalls gleich durchgeführt, wie bei der vorigen Route (Zeilen 11 und 12). Jedoch wird die Bearbeitung der Anfragen in dieser Route parallelisiert, um einen besseren Durchsatz zu erreichen, falls mehrere Anfragen gleichzeitig eingehen. Die Parallelisierung wird durch das Schlüsselwort „threads“ eingeleitet (Zeile 9). Dabei bedeuten die Zahlen, die dem Befehl als Parameter übergeben werden, dass die minimale Größe des Thread Pools 2 und die maximale Größe 5 beträgt. In der vorigen Route ist dies nicht nötig, da bereits über die API des OpenTOSCA Containers, pro zu erstellender Instanz, ein Thread erzeugt wird und die Anfragen für die Provisionierung einer Instanz ohnehin nur sequentiell abgearbeitet werden können.

Die Zeilen 13 bis 28 stellen die Routing Logik für die eingehenden Anfragen dar. Dazu inspiziert die Route ein spezielles Header Feld des Exchanges, das den Namen der Operation der Anfrage enthält. Derzeit werden vier Operationen unterstützt: das Deployment bzw. Undeployment eines IAs, der Aufruf eines IAs und das Instanzdaten Matching. Falls ein Container eine Anfrage erhält, bei der keine der unterstützten Operationen vorliegt, wird dies im Log vermerkt (Zeile 27). Wenn dagegen eine der definierten Operationen angefordert wird, erfolgt lokal der Aufruf einer entsprechende Java Methode mit dem Exchange als Parameter. Dazu kann ein Camel Endpoint mit dem Schema „*bean*“ verwendet werden. Ein solcher Endpoint ist in Listing 6.4 beispielhaft für das IA Deployment abgebildet (Zeile 5 und 6). Für die anderen Operationen sind die Endpoints zur besseren Lesbarkeit

nicht dargestellt. Diese können aber nach dem gleichen Schema definiert werden. Der Kontextpfad des Endpoints beinhaltet den Namen, mit der die benötigte Java Bean in einer Registry gefunden werden kann. Da der OpenTOSCA Container ein OSGi Projekt ist, kann dazu die OSGi Registry verwendet werden (siehe Abschnitt 2.3). Als Name der Bean muss somit lediglich der Name der Java Klasse und als Präfix der Name des Pakets, in der sie enthalten ist, angegeben werden. Durch das Feld „method“ in den Optionen der URI kann schließlich die Methode ausgewählt werden, die auf der Java Bean ausgeführt werden soll.

Listing 6.4 Camel Route zum Empfangen von Anfragen anderer Knoten

```
String consumerEndpoint = "mqtt:request?host=" + url
  + "&userName=" + username + "&password=" + password
  + "&subscribeTopicNames=" + topic + "&qualityOfService=ExactlyOnce";

String deploymentEndpoint = "bean:org.opentosca.bus...RequestReceiver"
  + "?method=invokeIADeployment";

from(consumerEndpoint)
  .threads(2, 5)
  .doTry()
    .unmarshal(jaxb)
    .process(incomingProcessor)
    .choice()
      .when(header(remoteOperationHeader).isEqualTo(INVOKE_IA_DEPLOYMENT))
        .to(deploymentEndpoint)
      .endChoice()
      .when(header(remoteOperationHeader).isEqualTo(INVOKE_IA_UNDEPLOYMENT))
        .to(undeploymentEndpoint)
      .endChoice()
      .when(header(remoteOperationHeader).isEqualTo(INVOKE_IA_INVOCATION))
        .to(invocationEndpoint)
      .endChoice()
      .when(header(remoteOperationHeader).isEqualTo(INVOKE_INSTANCE_DATA_MATCHING))
        .to(instanceMatchingEndpoint)
      .endChoice()
      .otherwise()
        .log(LoggingLevel.WARN, LOG, invalidOperation)
      .endChoice()
  .endDoTry()
  .doCatch(Exception.class)
    .log(LoggingLevel.ERROR, LOG, noMarshalling)
  .end();
```

6.3. Datenmodell für die Kommunikation

In diesem Abschnitt wird das im verteilten OpenTOSCA Container zur Kommunikation verschiedener Knoten verwendete Datenmodell vorgestellt. Das Datenmodell wird in den Camel Routes des Management Busses genutzt, um die Daten in den Exchanges über MQTT senden zu können (siehe Abschnitt 6.2). Deshalb muss das Datenmodell zum einen ermöglichen, dass Informationen über das Deployment bzw. den Aufruf von IAs oder das Instanzdaten Matching ausgetauscht werden können. So müssen zum Beispiel Parameter für einen IA Aufruf übertragen werden können, damit der Aufruf am entfernten Knoten erfolgreich durchgeführt werden kann. Zum anderen ist es nötig, innerhalb des Datenmodells Namen und Werte der Header Felder eines Exchanges einfügen zu können. Da über MQTT lediglich der Body der Nachricht des Exchanges gesendet werden kann, weil es keine benutzerdefinierten Header Felder gibt, können die Informationen aus den Exchange Headern auf diese Weise dennoch erhalten werden. Um die Informationen aus den Headern nach der Übertragung wieder möglichst leicht auslesen zu können, sollten diese im Datenmodell klar vom eigentlichen Inhalt der Nachricht getrennt werden.

Listing 6.5 zeigt das XML Schema des Datenmodells. Das Wurzelelement der XML-basierten Nachrichten stellt die *CollaborationMessage* dar (Zeile 49). Im zugehörigen Type (Zeile 42) ist definiert, dass die CollaborationMessage bis zu zwei weitere Elemente beinhaltet: Eine *HeaderMap* und ein optionaler *Body* (Zeilen 44 bis 46). Die HeaderMap wird verwendet, um die Header der Camel Exchanges zu speichern. Dazu kann sie eine beliebige Anzahl an *KeyValuePairs* enthalten (Zeilen 11 bis 15). Jedes KeyValuePair besteht dabei aus zwei Elementen vom Type String, die den Namen des korrespondierenden Header Feldes bzw. den Wert speichern können (Zeilen 5 bis 10). Der Body wird dagegen benötigt, damit die eigentliche Anfrage übertragen werden kann. Deshalb verfügt der Body entweder über ein *InstanceDataMatchingRequest* Element oder ein *IAInvocationRequest* Element, um in diesen die Daten für das Instanzdaten Matching bzw. den IA Aufruf zu hinterlegen. Wie in Abschnitt 6.2 beschrieben, werden derzeit vier Operationen an entfernten OpenTOSCA Container Knoten unterstützt. Für das IA Deployment bzw. Undeployment wird jedoch kein Body Element benötigt, da hierfür nur der Name des IAs und eine URL, an der das IA verfügbar ist, übertragen werden müssen. Diese Informationen werden im Management Bus generell als Metadaten in den Headern gespeichert [Zim16]. Somit ist für diese Operationen kein extra Body Element nötig. Aus diesem Grund ist das Body Element im XML Schema als optional definiert (Zeile 46).

Für die Durchführung des Instanzdaten Matching benötigen entfernte OpenTOSCA Container Knoten den Node Type und die Properties des Node Templates als Parameter (siehe Abschnitt 5.3). Diese können in einem *InstanceDataMatchingRequest* Element transferiert werden (Zeilen 21 bis 27). Dabei ist für den Node Type der QName erforderlich, der diesen identifiziert. Da bei den Properties des Node Templates eine beliebige Anzahl definiert werden kann, werden diese ebenfalls als KeyValueMap repräsentiert, wobei jedes Paar den Namen und den Wert einer Property enthält. Wird dagegen ein IA Aufruf mit einer Nachricht angefordert, müssen die Parameter für den Aufruf übertragen werden. Für die Parameter

Listing 6.5 XML Schema der Nachrichten zwischen OpenTOSCA Container Knoten

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
 xmlns:ns="http://collaboration.org/schema"
 targetNamespace="http://collaboration.org/schema">
 <xs:complexType name="KeyValueType">
  <xs:sequence>
   <xs:element name="Key" type="xs:string" />
   <xs:element name="Value" type="xs:string" />
  </xs:sequence>
 </xs:complexType>
 <xs:complexType name="KeyValueMap">
  <xs:sequence>
   <xs:element maxOccurs="unbounded" name="KeyValuePair" type="ns:KeyValueType" />
  </xs:sequence>
 </xs:complexType>
 <xs:complexType name="Doc">
  <xs:sequence>
   <xs:any minOccurs="0" maxOccurs="1" processContents="skip" />
  </xs:sequence>
 </xs:complexType>
 <xs:complexType name="InstanceDataMatchingRequest">
  <xs:sequence>
   <xs:element minOccurs="1" maxOccurs="1" name="NodeType" type="xs:QName" />
   <xs:element minOccurs="1" maxOccurs="1" name="Properties"
        type="ns:KeyValueMap" />
  </xs:sequence>
 </xs:complexType>
 <xs:complexType name="IAInvocationRequest">
  <xs:choice>
   <xs:element minOccurs="1" maxOccurs="1" name="Params" type="ns:KeyValueMap" />
   <xs:element minOccurs="1" maxOccurs="1" name="Doc" type="ns:Doc" />
  </xs:choice>
 </xs:complexType>
 <xs:complexType name="BodyType">
  <xs:choice>
   <xs:element minOccurs="0" maxOccurs="1" name="InstanceDataMatchingRequest"
        type="ns:InstanceDataMatchingRequest" />
   <xs:element minOccurs="0" maxOccurs="1" name="IAInvocationRequest"
        type="ns:IAInvocationRequest" />
  </xs:choice>
 </xs:complexType>
 <xs:complexType name="CollaborationMessage">
  <xs:sequence>
   <xs:element minOccurs="1" maxOccurs="1" name="HeaderMap"
        type="ns:KeyValueMap" />
   <xs:element minOccurs="0" maxOccurs="1" name="Body" type="ns:BodyType" />
  </xs:sequence>
 </xs:complexType>
 <xs:element name="CollaborationMessage" type="ns:CollaborationMessage" />
</xs:schema>
```

Listing 6.6 Beispiel: Aufruf des Instanzdaten Matching auf den anderen Knoten

```
1  <CollaborationMessage xmlns="http://collaboration.org/schema">
2   <HeaderMap>
3    <KeyValuePair>
4     <Key>CORRELATION_ID</Key>
5     <Value>21bca67f-8ec9-45f0-9c06-d3416e7debef</Value>
6    </KeyValuePair>
7    <KeyValuePair>
8     <Key>MQTTBROKER_URL</Key>
9     <Value>tcp://192.168.178.25:1883</Value>
10   </KeyValuePair>
11   <KeyValuePair>
12    <Key>MQTT_TOPIC</Key>
13    <Value>opentosca/container/collaboration/request</Value>
14   </KeyValuePair>
15   <KeyValuePair>
16    <Key>REMOTE_OPERATION</Key>
17    <Value>invokeInstanceDataMatching</Value>
18   </KeyValuePair>
19   <KeyValuePair>
20    <Key>REPLYTO_TOPIC</Key>
21    <Value>opentosca/container/collaboration/response</Value>
22   </KeyValuePair>
23  </HeaderMap>
24  <Body>
25   <InstanceDataMatchingRequest>
26    <NodeType xmlns:ns1="http://opentosca.org/nodetypes">
27        ns1:RaspberryPI3</NodeType>
28    <Properties>
29     <KeyValuePair>
30      <Key>MACAddress</Key>
31      <Value>10-08-B1-E6-28-D3</Value>
32     </KeyValuePair>
33    </Properties>
34   </InstanceDataMatchingRequest>
35  </Body>
36 </CollaborationMessage>
```

eines IA Aufrufs kann im OpenTOSCA Container entweder eine Java HashMap oder ein XML Element verwendet werden [Zim16]. Deshalb kann ein *IAInvocationRequest* beide Arten von Parametern umfassen (Zeilen 28 bis 33). Im XML Schema kann die Java HashMap auf eine KeyValueMap und das XML Element auf ein „any“ Element abgebildet werden.

Listing 6.6 stellt ein Beispiel für eine Nachricht dar, die zwischen Knoten eines verteilen OpenTOSCA Containers übertragen werden kann und die auf dem oben beschriebenen Schema basiert. Von Zeile 2 bis 23 ist die HeaderMap zu sehen, die alle Header Felder beinhaltet, die im Exchange vor der Übertragung über MQTT definiert waren. Dazu zählen die URL des Brokers und das Topic, auf dem die Nachricht veröffentlicht wird (Zeilen 7 bis 14), da diese Informationen von der Camel Route zum Senden von Nachrichten benötigt werden (siehe Listing 6.2). Außerdem wird zwischen Zeile 3 und 6 die Correlation ID der Nachricht übertragen, über die der Kontext der Nachricht und der Empfänger Thread

beim Erhalt einer Antwort identifiziert werden können. Der „REMOTE_OPERATION“ Header wird beim Empfänger der Nachricht verwendet, um die angeforderte Operation auszuwählen (Zeilen 15 bis 18). Die dargestellte Nachricht ist also dazu bestimmt, das Instanzdaten Matching für den Inhalt des Bodies der Nachricht durchzuführen. Als letzten Header enthält die Nachricht das „REPLYTO_TOPIC“ (Zeilen 19 bis 22). Durch dieses Header Feld wird das „Return Address“ Pattern umgesetzt (siehe Abschnitt 6.2). Der empfangende Knoten kann also seine Antwort auf diesem Topic veröffentlichen, ohne vorher mit einem fest definierten Topic konfiguriert zu werden. Die Zeilen 24 bis 35 zeigen den Body der Nachricht. Das Body Element enthält ein InstanceDataMatchingRequest Element, da die Nachricht, wie durch die Operation definiert, das Instanzdaten Matching auslösen soll. Für das Instanzdaten Matching müssen der Node Type und die Properties des Node Templates an die anderen Knoten des verteilten OpenTOSCA Containers gesendet werden. In der konkreten Nachricht soll das Matching für ein Raspberry Pi als Hardware durchgeführt werden (Zeile 27). Um die Hardware eindeutig zu identifizieren, wird in diesem Fall mit der MAC-Adresse nur eine einzelne Property verwendet (Zeilen 28-33).

6.4. Ausführen von OpenTOSCA auf Raspberry Pis

Um von den Vorteilen des neu entwickelten Konzepts in IoT Umgebungen, wie zum Beispiel der Umgehung von Firewalls, profitieren zu können, muss in jedem Netzwerk, in dem Anwendungen verwaltet werden sollen, ein Knoten der verteilten TOSCA-Runtime gestartet werden. Wenn also beispielsweise drei Smart Homes gemanaged werden sollen, können ein Knoten in der Cloud als zentraler Zugriffspunkt und drei Knoten in den verschiedenen Smart Homes genutzt werden. Bei der Verwendung einer zentralisierten TOSCA-Runtime muss dagegen lediglich der Knoten in der Cloud ausgeführt werden. Damit ist dort nur eine virtuelle Maschine bzw. ein Server nötig. Durch die Verteilung der TOSCA-Runtime ist also deutlich mehr Infrastruktur bzw. Hardware erforderlich, auf der die verteilten Knoten ausgeführt werden. Gleichzeitig kann jedoch nicht davon ausgegangen werden, dass in den Smart Homes neben den Smart Devices für die IoT Anwendungen weitere Infrastruktur wie Server oder Computer vorhanden sind, die dauerhaft angeschaltet sind und für die Ausführung eines Knotens der verteilten TOSCA-Runtime verwendet werden können. Das bedeutet, dass die Ausführung direkt auf einem der Smart Devices ermöglicht werden muss. Aufgrund der Heterogenität der Smart Devices und den häufig geringen Ressourcen, kann dies jedoch problematisch sein.

Im motivierenden Szenario dieser Arbeit und auch in realen Smart Home Szenarien können diese Smart Devices zum Beispiel Raspberry Pis sein. Für die Validierung des Konzepts anhand des motivierenden Szenarios ist es also nötig, das OpenTOSCA Ökosystem auf einem Raspberry Pi auszuführen. Das Hauptproblem, das hierbei auftritt, ist die *ARM* Prozessorarchitektur, die sich grundlegend von der weit verbreiteten *x86* Prozessorarchitektur unterscheidet [RH09]. So ist es beispielsweise nicht möglich, Programme, die auf der einen Prozessorarchitektur basieren, problemlos auf der Anderen auszuführen.

Wenn das OpenTOSCA Ökosystem in einer Produktionsumgebung eingesetzt wird, sollte es üblicherweise mittels Docker Compose gestartet werden. Deshalb wird eine Docker Compose Konfiguration benötigt, die einen automatischen Start auf Geräten mit einer ARM Prozessorarchitektur ermöglicht. Da alle Docker Images der Komponenten des OpenTOSCA Ökosystems auf der x86 Architektur basieren, musste für jede Komponente ein neues Dockerfile für die ARM Architektur erstellt werden. Außerdem wurde auch eine neue Docker Compose Datei erzeugt, die die neuen Images konfiguriert. Somit ist es möglich, das OpenTOSCA Ökosystem auf einem Raspberry Pi, ebenso wie auf einer virtuellen Maschine, mit nur wenigen Befehlen hochzufahren. Weitere Details dazu können im zugehörigen Git Repository gefunden werden[3].

Ein weiteres Problem, das bei der Ausführung des OpenTOSCA Ökosystems auf einem Smart Device auftritt, ist die lange Antwortzeit aufgrund der geringen Ressourcen. Beim Start des OpenTOSCA Ökosystems mittels Docker Compose werden derzeit insgesamt 8 Docker Container gestartet, wodurch die Ressourcen eines Raspberry Pis komplett ausgelastet sind. Zu den Docker Containern zählt beispielsweise das Modellierungswerkzeug Winery. Da es sehr unwahrscheinlich ist, dass diese Winery Instanz zur Modellierung verwendet wird, sollte sie im Normalfall auf einem Smart Device nicht gestartet werden. Stattdessen ist es sinnvoll, nur die Docker Container zu erzeugen, die auch tatsächlich benötigt werden. Im Extremfall sind dies lediglich die beiden Docker Container mit der IA Engine und dem OpenTOSCA Container. Die beiden Docker Container reichen zum Betrieb eines Knotens der verteilten TOSCA-Runtime aus, wenn an diesem Knoten keine CSARs hochgeladen werden und der Knoten lediglich auf Befehle anderer Knoten reagieren soll. Muss dagegen der Upload von CSARs an diesem Knoten ermöglicht werden, ist zusätzlich die Plan Engine bereitzustellen, damit die Pläne in den CSARs ausgeführt werden können. Dies ist ebenso der Fall, wenn in zukünftigen Erweiterungen neben verteilten IAs auch verteilte Pläne verwendbar sein sollen.

[3] https://github.com/OpenTOSCA/opentosca-docker-arm

7. Validierung des Konzepts und der Implementierung

Dieses Kapitel beinhaltet eine Validierung des in dieser Arbeit entwickelten Konzepts und der darauf basierenden prototypischen Implementierung. In Abschnitt 7.1 werden zunächst die in Abschnitt 4.2 definierten Anforderungen überprüft und bewertet, ob diese nicht, teilweise oder ganz erfüllt wurden. Danach wird in Abschnitt 7.2 gezeigt, wie die Anwendung des motivierenden Szenarios (siehe Abschnitt 1.1) mit dem neuen Konzept provisioniert werden kann. Abschließend präsentiert Abschnitt 7.3 eine empirische Studie, in der die Zeiten der Instanzerzeugung mit und ohne Integration des neuen Konzepts anhand der TOSCA-Runtime OpenTOSCA Container verglichen werden.

7.1. Überprüfung der Anforderungen

In diesem Abschnitt werden die Anforderungen an das entwickelte Konzept und die Implementierung validiert. Die Ergebnisse der Validierung sind in Tabelle 7.1 zusammengefasst. In der ersten Spalte wurde die Nummer der Anforderung aus Abschnitt 4.2 mit einer kurzen Zusammenfassung der Anforderung eingefügt. Die zweite Spalte zeigt das Ergebnis der Validierung. Zwei Haken stehen dabei für eine voll erfüllte Anforderung. Ein einzelner Haken bedeutet dagegen, dass die Anforderung zwar größtenteils erfüllt wurde, dass aber für eine volle Erfüllung eventuell noch weitere Verbesserungen des Konzepts oder der Implementierung nötig sind.

Anforderung	**Erfüllt?**
A1 („Verteilte Provisionierung von IoT Anwendungen“)	✓✓
A2 („Management über ein zentrales Interface“)	✓✓
A3 („Dynamische Identifizierung von Hardware“)	✓✓
A4 („Beheben von Problemen durch Firewalls“)	✓✓
A5 („Vermeidung von Mehrfachübertragungen“)	✓
A6 („Einheitliche Modellierung“)	✓✓
A7 („Prototypische Implementierung“)	✓✓
A8 („Keine Verschlechterung der Performance“)	✓

Tabelle 7.1.: Validierung der Anforderungen an die Arbeit

Das Hauptziel des Konzepts ist, die automatische und verteilte Provisionierung von mit TOSCA modellierten IoT Anwendungen zu ermöglichen. Dies konnte durch die Entwicklung einer verteilten TOSCA-Runtime (siehe Abschnitt 5.1) erreicht werden. Da in der verteilten TOSCA-Runtime Knoten in verschiedenen Netzwerken positioniert werden können, ist es möglich, Anwendungen in unterschiedlichen IoT Umgebungen bereitzustellen. Wenn dabei eine Master/Slave Architektur für die verteilte TOSCA-Runtime gewählt wird, kann das Interface des Masters als zentrales Interface zum Management aller Anwendungen genutzt werden. Damit sind die Anforderungen *A1* und *A2* vollständig erfüllt.

Um eine Wiederverwendbarkeit von CSARs zu ermöglichen, muss es erlaubt sein, Daten für die Identifizierung von Hardware dynamisch zum Zeitpunkt der Instanzerzeugung an die TOSCA-Runtime zu übergeben (*A3*). Dies wurde realisiert, indem die Entscheidung über die Verteilung und das IA Deployment bis zur Erzeugung von Anwendungsinstanzen verzögert wurde (siehe Abschnitt 5.2). Außerdem führt die Abwicklung der kompletten Kommunikation zwischen Knoten der verteilten TOSCA-Runtime über MQTT Broker dazu, dass Probleme, die durch Firewalls entstehen, umgangen werden können (*A4*). Dabei muss jedoch darauf geachtet werden, dass entweder global verfügbare Broker verwendet werden, oder, in einer Master/Slave Architektur, mindestens alle Knoten Zugriff auf den Broker des Masters haben.

Durch das Zwischenspeichern von großen Dateien, wie DAs und IAs, bei den verteilten Knoten der TOSCA-Runtime, erlaubt das entwickelte Konzept die Vermeidung von Mehrfachübertragungen und somit eine effizientere Provisionierung (*A5*). Bei der prototypischen Implementierung wurde dieses Speichern für DAs jedoch bisher nicht umgesetzt. Der Grund hierfür ist, dass die verteilte TOSCA-Runtime auf typischen IoT Devices mit geringen Ressourcen ausgeführt werden soll. Damit ist eine Abwägung zwischen den benötigten Ressourcen für das Speichern und den benötigten Ressourcen für das erneute Übertragen zu treffen. Zukünftige Erweiterungen könnten ein Konzept entwickeln, das ermittelt, wann ein Speichern und wann ein erneutes Übertragen das effizientere Vorgehen ist. Bei IAs wird das Deployment dagegen nur einmal durchgeführt. Damit ist bis zum Undeployment, aufgrund des Löschens der letzten Instanz, die das IA verwendet, keine erneute Übertragung nötig. Anforderung *A5* wurde deshalb bisher nur teilweise erfüllt.

Anforderung *A6* fordert, dass keine neuen Modellierungskonstrukte für TOSCA eingeführt werden. Die Modellierung von Cloud und IoT Anwendungen wird somit konzeptionell auf dieselbe Weise durchgeführt und Modellierer müssen keine neuen Verfahren erlernen. Dies wurde erreicht, indem die Verteilungsentscheidung auf der Grundlage des Instanzdaten Matching getroffen wird (siehe Abschnitt 5.3) Damit wird keine Erweiterung des TOSCA Standards benötigt und die Modellierung von Anwendungen mit und ohne verteilte Provisionierung wird exakt gleich durchgeführt.

Das entwickelte Konzept wurde prototypisch implementiert und in den OpenTOSCA Container integriert (*A7*). Durch die Integration des Konzepts wurde die Performance bei der Provisionierung von Anwendungsinstanzen, wenn keine Verteilung benötigt wird, geringfügig verschlechtert (*A8*). Abschnitt 7.3 enthält weitere Informationen zur Performance vor und nach der Integration des Konzepts. Die Verschlechterung betrifft, aufgrund des

verzögerten IA Deployments, vor allem die Erzeugung der ersten Instanz einer CSAR. Da das verzögerte Deployment der IAs jedoch nicht vermieden werden kann, wenn dieselbe TOSCA-Runtime auch verteilt einsetzbar sein soll (siehe Abschnitt 5.2), muss die geringe Verschlechterung der Performance in Kauf genommen werden. Weil die Performance bei der Erzeugung weiterer Instanzen jedoch nahezu unverändert bleibt, ist die Anforderung *A8* dennoch größtenteils erfüllt. Eine mögliche Erweiterung wäre, zwei verschiedene Modi (zentralisiert und verteilt) in der TOSCA-Runtime einzuführen und im Falle einer zentralisierten TOSCA-Runtime das Deployment der IAs bereits beim Deployment der CSAR durchzuführen. So kann die Performance auch beim Erstellen der ersten Instanz einer Anwendung beibehalten werden.

7.2. Validierung des motivierenden Szenarios

Um die Verwendbarkeit des entwickelten Konzepts anhand eines realen Anwendungsfalls zu zeigen, wird in diesem Abschnitt präsentiert, wie die Anwendung des in Abschnitt 1.1 vorgestellten motivierenden Szenarios mit Hilfe eines verteilten OpenTOSCA Containers automatisch bereitgestellt werden kann. Die Anwendung, deren Topologie in Abbildung 1.1 dargestellt ist, besteht aus drei unterschiedlichen Bestandteilen. Der linke Teil der Anwendung ist für die Generierung von Sensordaten, beispielsweise in einem Smart Home, zuständig. In der Mitte der Topologie ist dagegen die Middleware dargestellt, mit der die Daten verarbeitet und weitergeleitet werden können. Diese wird auf einer virtuellen Maschine in der Cloud ausgeführt. Der rechte Teil der Anwendung wird genutzt, um aus den gewonnen Daten Steuerungsbefehle abzuleiten und über Aktuatoren umzusetzen. Dieser Teil der Anwendung könnte, wie der erste Teil, ebenfalls in einem Smart Home ausgeführt werden. Dabei können beide Teile im selben Smart Home oder in unterschiedlichen Smart Homes positioniert sein. Um für die Validierung eine möglichst verteilte Anwendung zu erhalten, wird davon ausgegangen, dass die beiden Teile der Anwendung in verschiedenen Smart Homes ausgeführt werden sollen.

Abbildung 7.1 zeigt das Setup der Hardware, die für die Provisionierung der Anwendung verwendet werden soll. Es sind zwei Smart Homes mit jeweils einem Raspberry Pi dargestellt. Diese sollen für die Ausführung des linken bzw. rechten Teils der Anwendungstopologie verwendet werden. Für jedes Raspberry Pi wird zusätzlich die MAC-Adresse angegeben, über die dieses identifiziert werden kann. Diese Informationen müssen der TOSCA-Runtime beim Start der Provisionierung als Properties übergeben werden. Außerdem ist eine OpenStack[1] Cloud abgebildet, die zur Provisionierung der IoT Middleware genutzt wird. Eine zentralisierte TOSCA-Runtime kann bei der Bereitstellung, wie in Abschnitt 4.1.2 beschrieben, an den Firewalls der Smart Home Umgebungen scheitern. Stattdessen soll deshalb eine verteilte TOSCA-Runtime bzw. ein verteilter OpenTOSCA Container zum Einsatz kommen. Dazu wird je ein Knoten in beiden Smart Homes und

[1] https://www.openstack.org/

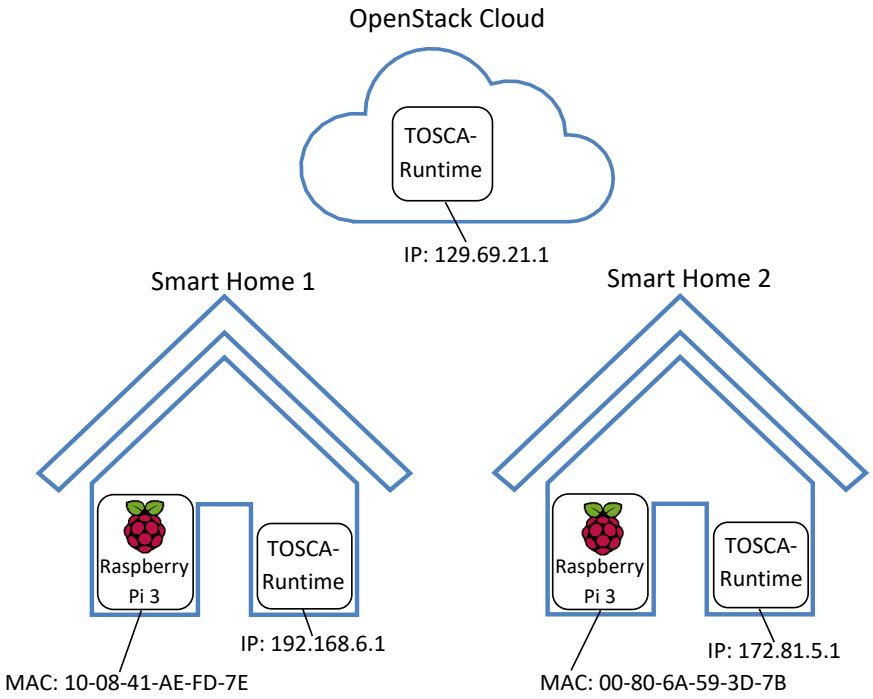

Abbildung 7.1.: Aufbau der verwendeten Hardware für das motivierende Szenario

in der Cloud positioniert. Um eine geeignete Konfiguration zu ermöglichen, ist für jeden OpenTOSCA Container Knoten die IP-Adresse angegeben, über die dieser erreicht werden kann. Für die Provisionierung wird angenommen, dass diese vom OpenTOSCA Container Knoten in der Cloud aus gestartet wird. Da in diesem Fall die Kontrolle nur von einem Knoten ausgeht, kann als Architektur der verteilten TOSCA-Runtime eine Master/Slave Architektur verwendet werden (siehe Abschnitt 5.1). Das bedeutet, lediglich die Slaves in den Smart Homes müssen Informationen über den MQTT Broker des Masters besitzen. Der Master dagegen benötigt keinerlei Informationen über andere OpenTOSCA Container.

Listing 7.1 zeigt einen Ausschnitt der für die Verwendung eines verteilten OpenTOSCA Containers bedeutsamen Konfigurationsparameter des Master Knotens. In Zeile 1 wird definiert, dass am Master Knoten ein MQTT Broker auf dem Port 1883 gestartet werden soll. Nach dem Hochfahren des OpenTOSCA Containers ist dieser also über die URL „tcp://129.69.21.1:1883“ erreichbar. Die Zeilen 2-3 geben anschließend an, dass die Authentifizierung am Broker mit dem Benutzernamen und Passwort „admin“ durchgeführt wird. Um dem OpenTOSCA Container Knoten mitzuteilen, dass eine Verwendung als verteilte TOSCA-Runtime vorgesehen ist, wird in Zeile 5 der „*Collaboration Mode*“ aktiviert. Durch diese Option wird es ermöglicht, dass alle Abläufe die mit der Zusammenarbeit von verschiedenen Knoten zu tun haben, deaktiviert werden können. Somit wird eine größere Effizienz erreicht, wenn lediglich eine zentralisierte TOSCA-Runtime erforderlich ist. In den Zeilen 6 und 7 können andere OpenTOSCA Container Knoten angegeben werden, von

denen der Knoten Anfragen empfangen soll. Da es sich jedoch um den Master handelt, sind hier keine Einträge nötig. Bei einer Peer-to-Peer Architektur würden hingegen die Hostnames und Ports der MQTT Broker beider Knoten in den Smart Homes angegeben.

Listing 7.1 Ausschnitt der Konfiguration des OpenTOSCA Container Master Knotens

```
1 org.opentosca.container.broker.mqtt.port=1883
2 org.opentosca.container.broker.mqtt.username=admin
3 org.opentosca.container.broker.mqtt.password=admin
4
5 org.opentosca.container.collaboration.mode=true
6 org.opentosca.container.collaboration.hostnames=
7 org.opentosca.container.collaboration.ports=
```

In Listing 7.2 ist ein Ausschnitt der Konfigurationsparameter der beiden Slave Knoten in den Smart Homes dargestellt. Durch die Zeilen 1-3 wird der lokale MQTT Broker gleich konfiguriert, wie beim Master. Da die Slaves keine Anfragen für andere Knoten veröffentlichen, benötigen sie nicht unbedingt einen lokalen MQTT Broker. Zumindest der Benutzername und das Passwort muss aber definiert werden, da diese auch zur Identifizierung bei anderen Brokern verwendet werden. In den Zeilen 6 und 7 unterscheidet sich die Konfiguration der Slaves vom Master. Dabei wird in Zeile 6 die IP-Adresse des Masters angegeben und in Zeile 7 der Port, an dem der dortige MQTT Broker arbeitet. Wenn Anfragen von mehr als nur einem anderen Broker empfangen werden sollen, kann unter „Hostnames“ und „Ports“ eine kommaseparierte Liste mit den Daten der verschiedenen Broker eingefügt werden.

Listing 7.2 Ausschnitt der Konfiguration der OpenTOSCA Container Slave Knoten

```
1 org.opentosca.container.broker.mqtt.port=1883
2 org.opentosca.container.broker.mqtt.username=admin
3 org.opentosca.container.broker.mqtt.password=admin
4
5 org.opentosca.container.collaboration.mode=true
6 org.opentosca.container.collaboration.hostnames=129.69.21.1
7 org.opentosca.container.collaboration.ports=1883
```

Nach der Durchführung der aufgezeigten Konfiguration, können die verschiedenen OpenTOSCA Container Knoten gestartet werden. Für den automatischen Start aller benötigten Komponenten des OpenTOSCA Ökosystems, ist beispielsweise eine Verwendung von Docker Compose denkbar. Im nächsten Schritt müssen die Instanzdaten angelegt werden, damit der verteilte OpenTOSCA Container bestimmen kann, welcher Knoten für welche Node und Relationship Templates der Anwendung zuständig ist. Dieser Vorgang wird in diesem Kapitel manuell vorgeführt. Mit zukünftigen Erweiterung sollte dies jedoch automatisiert werden, sodass stattdessen direkt die Anwendung provisioniert werden kann. Da beim Instanzdaten Matching das Infrastructure Node Template, also das unterste Node

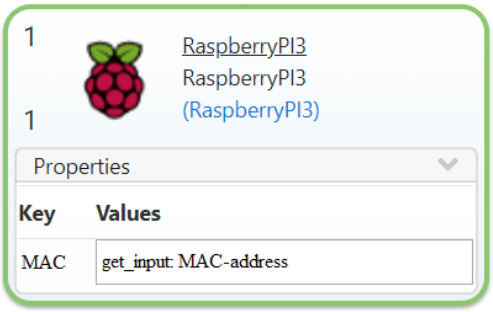

Abbildung 7.2.: Topologie zur Erzeugung der Instanzdaten in den Slaves

Template jedes Stapels der Anwendungstopologie, verglichen wird, muss dieses zunächst ermittelt werden. Für die beiden Anwendungsteile, die in den Smart Homes verwaltet werden sollen, ist das Infrastructure Node Template jeweils vom Node Type „RaspberryPI3“. Deshalb wird für die Erzeugung der Instanzdaten eine CSAR mit einer Topologie erstellt, die lediglich ein Node Template von diesem Node Type enthält. Die Topologie ist in Abbildung 7.2 dargestellt. Der Node Type „RaspberryPI3“ definiert mit der MAC-Adresse nur eine Property. Diese Property enthält in der Abbildung den Wert „get_input: MAC-address“. Damit wird definiert, dass die Property vom OpenTOSCA Container bei der Erzeugung einer Instanz abgefragt wird, und dem Nutzer dabei der Name „MAC-address“ angezeigt werden soll. Somit kann die CSAR für das Anlegen von Instanzdaten für beliebig viele konkrete Raspberry Pis verwendet werden. Die CSAR kann nun in die beiden OpenTOSCA Container Knoten in den Smart Homes hochgeladen werden. Anschließend wird dort eine Instanz mit der MAC-Adresse „10-08-41-AE-FD-7E“ bzw. „00-80-6A-59-3D-7B“ erzeugt (siehe Abbildung 7.1). Der Build Plan terminiert sofort, weil für eine Topologie mit lediglich einem Raspberry Pi ohne zu installierender Anwendung, keine Logik zur Provisionierung ausgeführt werden muss. Damit sind die beiden OpenTOSCA Container Knoten in den Smart Homes fertig eingerichtet. Da der Topologie Stapel mit der IoT Middleware vom OpenTOSCA Container Knoten in der Cloud verwaltet werden soll, könnten hier ebenfalls geeignete Instanzdaten erzeugt werden. Für den OpenTOSCA Container wurde als Standard für die Bereitstellung von IAs, wenn keine passenden Instanzdaten gefunden werden, jedoch eine lokale Bereitstellung umgesetzt. Das bedeutet, dadurch, dass die CSAR im OpenTOSCA Container Knoten in der Cloud hochgeladen wird, sind die Instanzdaten dort optional. Somit ist auch der Master Knoten vollständig konfiguriert und die Provisionierung der Anwendung kann gestartet werden.

Um die Provisionierung des motivierenden Szenarios zu starten, muss zunächst die CSAR der Anwendung in den OpenTOSCA Container Knoten in der Cloud hochgeladen werden. Anschließend kann eine Instanz der Anwendung erzeugt werden. Dabei muss der Nutzer die zusätzlichen, nötigen Parameter an den OpenTOSCA Container übergeben. Für diese Anwendung könnten die Parameter unter anderem die MAC-Adressen der beiden verwendeten Raspberry Pis und die Zugangsdaten für die OpenStack Cloud sein. Mit den übergebenen Parametern führt der verteilte OpenTOSCA Container den Build Plan der Anwendung aus. Während der Tasks des Plans kommunizieren die OpenTOS-

CA Container untereinander, um die unterschiedlichen Bestandteile der Anwendung an den verschiedenen Standorten bereitzustellen. Nach der Terminierung des Plans kann die Anwendung des motivierenden Szenarios verwendet und, bei Bedarf, durch Management Pläne verändert oder terminiert werden.

7.3. Empirische Studie zur Dauer der Instanzerzeugung

Dieser Abschnitt stellt eine empirische Studie zur Dauer der Instanzerzeugung bei der Verwendung verschiedener Konzepte in einer TOSCA-Runtime vor. Als TOSCA-Runtime Implementierung wurde dafür der OpenTOSCA Container (siehe Abschnitt 2.7) genutzt. Durch die Studie sollen zum Beispiel Erkenntnisse gewonnen werden, ob das in dieser Arbeit entwickelte Konzept zu einer Verschlechterung der Performance führt und wie sich die Verteilung von Knoten einer verteilten TOSCA-Runtime über verschiedene Netzwerke auf die Dauer der Instanzerzeugung auswirkt. Die Ergebnisse der Studie sind in Tabelle 7.2 zusammengefasst.

Um mit der Studie ein breiteres Spektrum an Anwendungen abzudecken, wurden die Tests anhand von zwei unterschiedlichen CSARs durchgeführt. Dabei stellt eine CSAR eine eher komplexe Anwendung dar und die andere CSAR enthält eine sehr simple Anwendung. Die komplexere Anwendung wird in der ersten Spalte von Tabelle 7.2 als *CSAR Nr. 1* bezeichnet und ist in Abbildung 7.3 dargestellt. Zur Provisionierung der Anwendung wird zunächst eine virtuelle Maschine mit Ubuntu 14.04 als Betriebssystem in einer VSphere Cloud[2] erstellt. Sobald die virtuelle Maschine erfolgreich hochgefahren ist, wird darauf ein MySQL[3] Datenbank Management System, zusammen mit den erforderlichen Programmen, installiert und eine Datenbank für die Anwendung erzeugt. Die Erzeugung der Datenbank beinhaltet auch das Anlegen des benötigten Schemas in der relationalen Datenbank. Nachfolgend installiert die TOSCA-Runtime eine Docker Engine (siehe Abschnitt 2.8) auf der virtuellen Maschine. In der Docker Engine wird ein Container mit der PHP Anwendung MyTiniTodo[4] gestartet. Im letzten Schritt kann eine Verbindung zwischen dem MyTiniTodo Docker Container und der MySQL Datenbank hergestellt werden, indem die MyTiniTodo Anwendung so konfiguriert wird, dass sie ihre Daten in der Datenbank speichert.

In Abbildung 7.4 ist die Topologie von *CSAR Nr. 2* dargestellt. Die Topologie besteht lediglich aus drei Node Templates. Zwei Node Templates stellen ein Raspberry Pi mit seinem Betriebssystem dar und das dritte Java in der Version 9. Wenn eine Instanz der Anwendung erstellt wird, muss die TOSCA-Runtime also eine Verbindung zu einem Raspberry Pi herstellen und dort Java 9 durch das Paketsystem installieren. Im Vergleich zu CSAR Nr. 1 werden für die Provisionierung deutlich weniger Management Operationen

[2] https://www.vmware.com/de/products/vsphere.html
[3] https://www.mysql.com
[4] http://www.mytinytodo.net

CSAR Nr.	Verwendetes Konzept	Anzahl Durch-läufe	∅[a] Dauer CSAR Deployment	σ[b] CSAR Deployment	∅ Dauer 1. Instanz	σ 1. Instanz	∅ Dauer Instanzen 2-5	σ Instanzen 2-5
1	Altes Konzept	20	1:33:54	00:04:10	4:13:93	00:04:68	4:06:15	00:07:60
1	Neues Konzept ohne Verteilung	20	1:05:19	00:04:45	5:08:67	00:06:56	4:09:98	00:09:84
1	Neues Konzept mit lokaler Verteilung	20	1:05:25	00:03:70	5:24:78	00:04:70	4:25:73	00:04:67
1	Neues Konzept mit globaler Verteilung	20	1:05:60	00:03:15	5:44:39	00:02:73	4:30:91	00:01:92
2	Altes Konzept	20	0:38:78	00:00:90	2:53:93	00:01:83	2:52:77	00:01:10
2	Neues Konzept ohne Verteilung	20	0:25:70	00:00:40	3:06:50	00:01:14	2:55:20	00:01:05
2	Neues Konzept mit lokaler Verteilung	20	0:25:45	00:00:44	3:11:33	00:00:50	2:58:10	00:00:78
2	Neues Konzept mit globaler Verteilung	20	0:25:52	00:00:40	3:13:37	00:00:57	2:58:10	00:00:76

Tabelle 7.2.: Vergleich der Dauer des CSAR Deployments und der Instanzerzeugung bei der Verwendung verschiedener Konzepte anhand des OpenTOSCA Containers

[a] Durchschnitt
[b] Standardabweichung

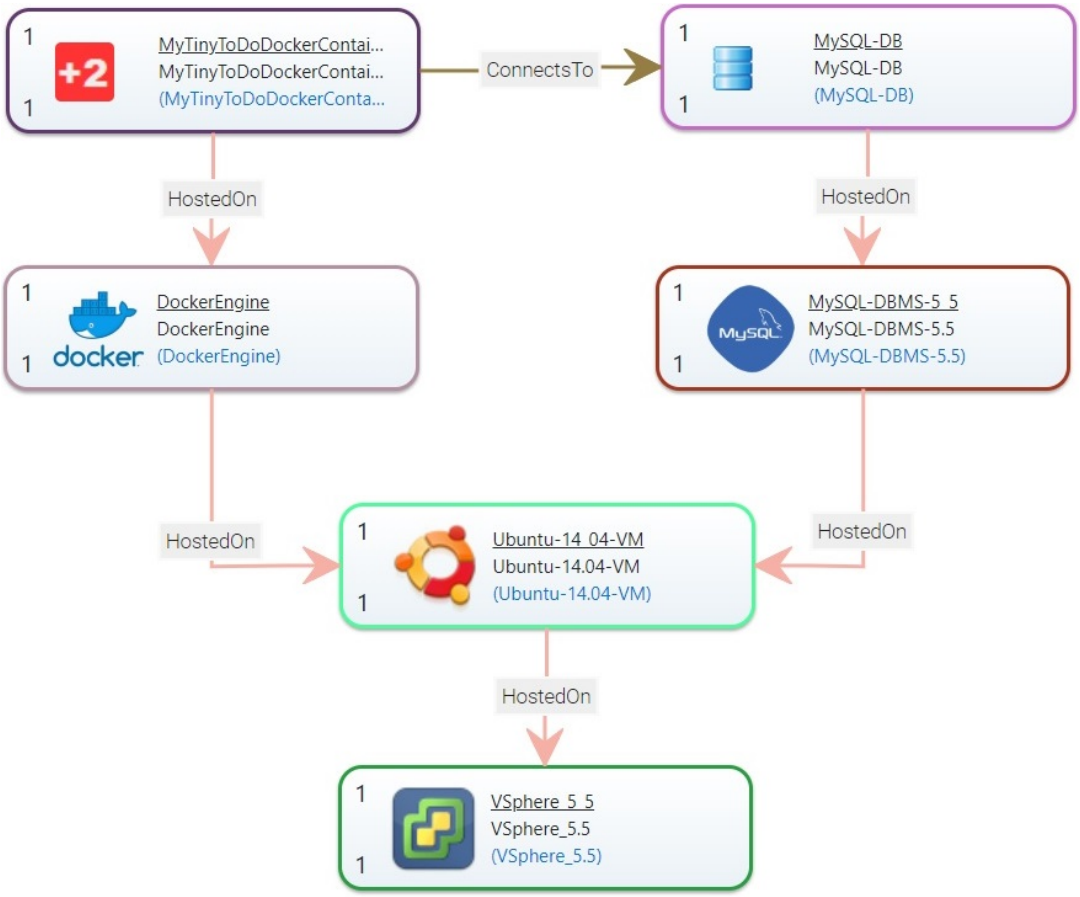

Abbildung 7.3.: CSAR Nr. 1: MyTinyTodo + MySQL auf VSphere

aufgerufen und damit auch weniger IAs von der TOSCA-Runtime deployt und verwendet. Deshalb sollten die Zeiten für CSAR Nr. 2, unabhängig vom verwendeten Konzept, generell geringer ausfallen, als für CSAR Nr. 1.

Die zweite Spalte in Tabelle 7.2 gibt das verwendete Konzept an. Für die Studie wurden insgesamt vier verschiedene Ansätze evaluiert. Der erste Ansatz ist das *alte Konzept*, für das die Abläufe in Abschnitt 4.1.1 vorgestellt wurden. Die Messungen für diesen Ansatz dienen als Referenzwerte für das neu entwickelte Konzept. Beim zweiten Ansatz handelt es sich um das *neue Konzept ohne Verteilung*. Hier wurden also die neu entwickelten und in Abschnitt 5.4 präsentierten Abläufe in der TOSCA-Runtime umgesetzt. Es wurde aber nach wie vor eine zentralisierte TOSCA-Runtime verwendet. Das *neue Konzept mit lokaler Verteilung* stellt den dritten überprüften Ansatz dar. Lokale Verteilung bedeutet, dass eine verteilte TOSCA-Runtime eingesetzt wird, wobei die einzelnen Knoten jedoch im selben Netzwerk positioniert sind. Dabei ist ein anderer Knoten für die Provisionierung zuständig, wie der, in den die CSAR hochgeladen wurde. Somit ist eine Kommunikation zwischen den Knoten nötig. Beim letzten Ansatz handelt es sich um das *neue Konzept mit globaler Verteilung*. Im Gegensatz zum dritten Ansatz sind die Knoten der verteilten TOSCA-Runtime hier also nicht im selben Netzwerk, sondern global verteilt.

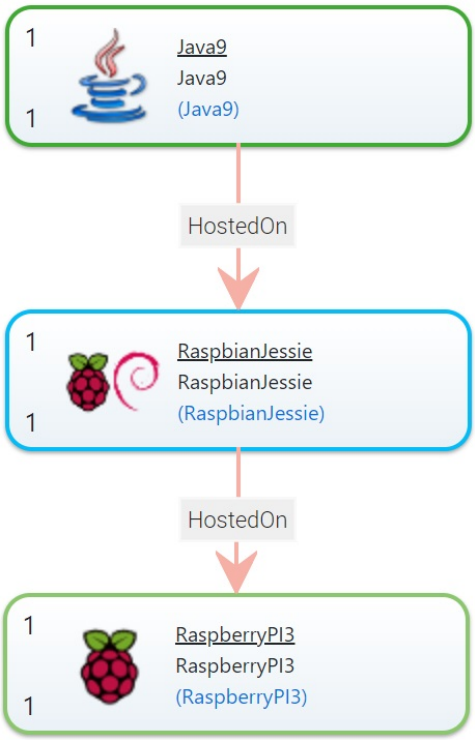

Abbildung 7.4.: CSAR Nr. 2: Java 9 auf Raspberry Pi 3

Die Spalten vier bis neun der Tabelle beinhalten die Mittelwerte der Messwerte, die in der Studie aufgezeichnet wurden bzw. deren Standardabweichung. Für jede Kombination aus CSAR und Konzept wurden jeweils 20 Durchläufe ausgeführt. Ein Durchlauf bedeutet dabei das Deployment der CSAR und anschließend die Erzeugung von fünf Instanzen der Anwendung. Genaue Messwerte zu den einzelnen Durchläufen finden sich in Anhang A. Für das CSAR Deployment (Spalte vier) und das Erstellen der ersten Instanz (Spalte sechs) wurde jeweils der Durchschnitt der 20 Durchläufe berechnet und in Tabelle 7.2 eingefügt. Spalte acht beinhaltet dagegen den Durchschnitt für die Erzeugung der Instanzen zwei bis fünf aller Durchläufe. Der Grund für die Trennung der ersten und der weiteren Instanzen ist, dass das IA Deployment beim neuen Konzept durch die Erstellung der ersten Instanz ausgelöst wird und diese damit einen Spezialfall bildet. Die Spalten fünf, sieben und neun enthalten die Standardabweichung der Messwerte der vorherigen Spalten und geben Aufschluss darüber, wie sehr sich die aufgezeichneten Messwerte unterscheiden. Alle Zeiten innerhalb der Tabelle sind im Schema Minuten:Sekunden:Millisekunden angegeben.

Für die Messungen der Studie wurde das OpenTOSCA Ökosystem mittels Docker Compose (siehe Abschnitt 2.8) auf virtuellen Maschinen ausgeführt. Bei den Ansätzen eins bis drei sind die benötigten virtuellen Maschinen in der VSphere Cloud des Instituts für Architektur von Anwendungssystemen der Universität Stuttgart bereitgestellt worden. Ansatz eins und zwei benötigen dabei jeweils nur eine virtuelle Maschine für die zentralisierten TOSCA-Runtimes. Bei Ansatz drei werden dagegen zwei virtuelle Maschinen verwendet und damit eine verteilte TOSCA-Runtime mit zwei Knoten konfiguriert. Für Ansatz vier ist es nicht

möglich, nur virtuelle Maschinen in der VSphere Cloud zu erstellen, da diese ansonsten im selben Rechenzentrum und damit nicht global verteilt wären. Stattdessen wurde eine virtuelle Maschine in VSphere und die Zweite in Amazons EC2[5] Cloud provisioniert, um so eine global verteilte TOSCA-Runtime zu erhalten. Alle virtuellen Maschinen verwenden dasselbe Betriebssystem und haben die gleiche Hardware Spezifikation, die im Folgenden beschrieben wird. Beim Betriebssystem handelt es sich um Ubuntu 16.04.01 LTS in der 64 Bit Version. Die virtuellen Maschinen verwenden zwei CPUs vom Modell Intel Xeon E5-2690 v2 mit einer Taktfrequenz von 3.00 GHz. Als Arbeitsspeicher stehen 2048 MB zur Verfügung und die Festplattengröße beträgt 64 GB.

Aus den Daten der Studie geht hervor, dass die Zeit für das CSAR Deployment bei der Verwendung des neuen Konzepts deutlich reduziert wird. Im Vergleich zum alten Konzept werden bei CSAR Nr. 1 ca. 28s und bei CSAR Nr. 2 ca. 13s weniger benötigt, was einer Ersparnis von über 30% entspricht. Der Grund hierfür ist, dass die IAs der CSARs erst beim Erstellen der ersten Instanz deployt werden (siehe Abschnitt 5.2) und diese Zeit somit beim neuen Konzept entfällt. Betrachtet man die Zeiten des CSAR Deployments hingegen bei den verschiedenen Verteilungen unter Verwendung des neuen Konzepts, sieht man, dass diese unabhängig von der Verteilung nahezu gleich bleiben. Da aktuell nur IAs verteilt ausgeführt werden und nicht Pläne (siehe Abschnitt 5.1), spielt die Verteilung während des CSAR Deployments noch keine Rolle und die geringen Differenzen sind auf Schwankungen bei den Messungen zurückzuführen.

Die Erzeugung der ersten Anwendungsinstanz dauert im neuen Konzept ohne Verteilung dagegen deutlich länger als im Alten, weil hier das Deployment der IAs durchgeführt werden muss. Für einen aussagekräftigen Vergleich ist es sinnvoll, die Zeiten des CSAR Deployments und der Erzeugung der ersten Instanz zu addieren, da dann in beiden Konzepten das IA Deployment enthalten ist. Wenn eine CSAR hochgeladen wird, um direkt eine Instanz zu erzeugen, entspricht dies außerdem der relevanten Antwortzeit der TOSCA-Runtime. Bei CSAR Nr. 1 benötigt das neue Konzept ca. 26s bzw. knapp über 7% länger. Weil die IAs beim neuen Konzept immer direkt vor einem Operationsaufruf bereitgestellt werden, muss dafür jedes Mal überprüft werden, ob das Environment (z.B. ein Tomcat Server) verfügbar ist, um anschließend eine Verbindung herzustellen. Beim alten Konzept wird dies dagegen nur einmal durchgeführt. Anschließend können alle IAs auf einmal deployt werden (siehe Abschnitt 4.1.1). Durch die vielen nötigen Management Operationen schlägt sich dieser Overhead bei CSAR Nr. 1 in der Performance nieder. Zur Provisionierung von Instanzen von CSAR Nr. 2 werden dagegen nur wenige Operationen verwendet. Die benötigte Zeit beim neuen Konzept verringert sich damit sogar um eine halbe Sekunde. Wenn weitere Instanzen einer Anwendung erzeugt werden sollen, ist der Unterschied in den Zeiten mit 2-3s sehr gering. Die erhöhte Zeit bei der Verwendung des neuen Konzepts ist darauf zurückzuführen, dass die Skript Artefakte direkt nach der Ausführung wieder gelöscht werden, um Ressourcen auf IoT Devices zu sparen (siehe Abschnitt 5.2) und somit einige zusätzliche Befehle nötig sind.

[5] https://aws.amazon.com/de/ec2

Sobald eine verteilte TOSCA-Runtime eingesetzt wird und nicht der Knoten für die Provisionierung einer Instanz zuständig ist, bei dem die CSAR deployt wurde, müssen die IAs zwischen den Knoten übertragen werden. Deshalb dauert die Erzeugung der ersten Instanz bei lokaler Verteilung weitere 16s bzw. 5s länger als ohne Verteilung. Anhand des Unterschiedes zwischen CSAR Nr. 1 und 2 ist ersichtlich, dass weniger zu übertragende IAs auch zu einer deutlich geringeren Verzögerung führen. Auch die weiteren Instanzen benötigen etwas länger, da die Knoten der verteilten TOSCA-Runtime hier ebenfalls kommunizieren müssen. Die Verzögerung ist aber deutlich geringer, weil nur kleine Datenmengen übertragen werden. Bei globaler Verteilung der Knoten der TOSCA-Runtime wirkt sich die langsamere Kommunikation über das Internet auf die Zeit der Instanzerzeugung negativ aus. Dabei ist die Auswirkung auf die erste Instanz, aufgrund der großen Datenmengen beim Übertragen der IAs, deutlich stärker, als bei den weiteren Instanzen. Da CSAR Nr. 2 nur einen entfernten Aufruf zur Provisionierung benötigt, ist die Verzögerung bei den weiteren Instanzen sogar so gering, dass diese in der Toleranz der Messungen verschwindet und die Zeiten gleich sind, wie bei lokaler Verteilung.

Insgesamt lässt sich also sagen, dass das neu entwickelte Konzept die Instanzerzeugung, vor allem bei der ersten Instanz, verlängert. Wenn die erste Instanz direkt nach dem Upload einer CSAR erstellt werden soll, wirkt sich die Verzögerung jedoch durch das schnellere CSAR Deployment deutlich geringer aus. Bei allen weiteren Instanzen hält sich der Overhead mit unter 2% in akzeptablen Grenzen. Sobald das neue Konzept in verteilten Umgebungen eingesetzt wird, verschlechtert sich die Performance der TOSCA-Runtime zusätzlich etwas. Da sich dies durch die erforderliche Kommunikation allerdings nicht vermeiden lässt und damit viele Probleme zentralisierter TOSCA-Runtimes (siehe Abschnitt 4.1.2) gelöst werden können, sollte dieser Nachteil in vielen Anwendungsgebieten tolerierbar sein.

Einschränkend muss beachtet werden, dass sich alle Zeiten der Studie auf den OpenTOSCA Container mit Entwicklungsstand vom 17.10.2018 beziehen. Das bedeutet, für das alte Konzept wurde die Version des Containers zu diesem Datum unverändert verwendet. Beim neuen Konzept stellt dagegen diese Version die Grundlage dar, auf welcher aufbauend alle Änderungen und Erweiterungen durchgeführt wurden. Zukünftige Änderungen oder Erweiterungen am OpenTOSCA Container können die Zeiten erheblich verlängern oder verkürzen. Wenn dieselben Änderungen auf beide Konzepte angewendet werden, sollten die Unterschiede jedoch ähnlich bleiben und die Grundaussagen der Analyse der Ergebnisse treffen weiter zu.

8. Zusammenfassung und Ausblick

Die kontinuierlich steigende Anzahl an entwickelten IoT Anwendungen und deren zunehmende Verbreitung in privaten und öffentlichen Räumen, stellen neue Herausforderungen an die Anbieter der Anwendungen. Eine manuelle Bereitstellung und Wartung der Anwendungen bei jedem Kunden ist durch die große Menge an Hardware- und Softwarekomponenten und deren Heterogenität eine komplexe, fehleranfällige und teure Aufgabe. Stattdessen ist es wünschenswert, diese Aufgaben möglichst vollständig zu automatisieren, sodass lediglich die benötigte Hardware physisch installiert werden muss.

Der TOSCA Standard wurde entwickelt, um Cloud Anwendungen portabel und interoperabel zu beschreiben. Nach der Modellierung können diese Anwendungen mittels einer TOSCA-Runtime automatisch provisioniert werden. Auch wenn TOSCA speziell für Cloud Anwendungen entworfen wurde, ist es ebenfalls möglich, damit IoT Anwendungen zu definieren. Somit können diese auch automatisch bereitgestellt werden. In IoT Umgebungen können die üblicherweise in TOSCA-Runtimes umgesetzten Konzepte allerdings zu Problemen führen, die im Cloud Umfeld nicht auftreten. So ist es beispielsweise möglich, dass Firewalls den Zugriff auf IoT Devices verhindern und dadurch die komplette automatische Bereitstellung fehlschlägt.

In dieser Arbeit wurde ein Konzept entwickelt, das die Probleme von TOSCA-Runtimes in IoT Umgebungen beheben soll. Bei der Analyse der Probleme konnte festgestellt werden, dass eine verteilte TOSCA-Runtime, bestehend aus mehreren verteilten Knoten in verschiedenen Netzwerken, gegenüber einer zentralisierten TOSCA-Runtime einige Vorteile bietet. So können zum Beispiel Firewalls umgangen werden, indem ein Knoten global zugreifbar in der Cloud ausgeführt wird und die anderen Knoten in den lokalen IoT Netzwerken positioniert werden. Um Befehle zur Provisionierung zu empfangen, können die lokalen Knoten aus dem geschützten Bereich heraus beim globalen Knoten Anfragen stellen, wodurch die Firewalls umgangen werden können. Anschließend können sie die Provisionierung von Teilen der Anwendung, für die sie zuständig sind, vornehmen. Somit kann das Management von Anwendungen über den globalen Knoten trotz aktiver Firewalls ermöglicht werden.

Damit eine verteilte TOSCA-Runtime eingesetzt werden kann, muss ein Prozess bzw. Algorithmus entworfen werden, mit dem entschieden werden kann, welcher Knoten für welche Teile einer Anwendung zuständig ist. Dabei muss darauf geachtet werden, keine unnötigen Erweiterungen am TOSCA Standard vorzunehmen, die möglicherweise die Portabilität oder Interoperabilität von Anwendungen einschränken. Deshalb basiert der Ansatz dieser Arbeit auf dem sogenannten Instanzdaten Matching. Es werden also bei jedem Knoten der verteilten TOSCA-Runtime Instanzdaten für alle lokal zugreifbaren Ressourcen bzw.

Devices angelegt. Beim Erzeugen einer Anwendung können anschließend die Node Templates, die die Hardware einer Anwendung identifizieren, mit den Instanzdaten der verteilten TOSCA-Runtime Knoten verglichen werden. Damit ist immer der Knoten für einen Stapel der Topologie zuständig, der die passenden Instanzdaten zur jeweiligen Hardware vorliegen hat. Passend sind Instanzdaten dabei, wenn sowohl der Node Type der Node Templates, als auch alle vorhandenen Properties übereinstimmen. Da Properties für zu erzeugende Anwendungsinstanzen üblicherweise erst dann vollständig vorhanden sind, wenn die Provisionierung über die API der TOSCA-Runtime ausgelöst wird, kann die Entscheidung über die Zuständigkeit auch erst zu diesem Zeitpunkt getroffen werden. Eine Folge daraus ist, dass im neu entwickelten Konzept das Deployment der IAs einer Anwendung vom Zeitpunkt des CSAR Deployments auf den Zeitpunkt der Instanzerzeugung verschoben werden muss.

Das entwickelte Konzept wurde prototypisch im OpenTOSCA Container, einer an der Universität Stuttgart entwickelten TOSCA-Runtime, umgesetzt. Dabei musste die Architektur leicht angepasst und vor allem Erweiterungen eingefügt werden, die die Kommunikation zwischen OpenTOSCA Container Knoten betreffen. Ein weiterer wichtiger Punkt war es, die Ausführbarkeit des OpenTOSCA Containers bzw. des OpenTOSCA Ökosystems auf typischen IoT Devices zu ermöglichen, was durch die unterschiedlichen Prozessorarchitekturen erschwert wird. Dadurch können Knoten des OpenTOSCA Containers auch in IoT Netzwerken ausgeführt werden, in denen nur IoT Devices vorhanden sind. Anhand der Validierung des motivierenden Szenarios konnte gezeigt werden, wie eine verteilte Anwendung in heterogenen Netzwerken mit dem Prototyp automatisch bereitgestellt werden kann. Außerdem wurde eine Studie durchgeführt, die Aufschluss darüber gibt, wie stark sich die Verwendung einer verteilten TOSCA-Runtime und die daraus resultierende, nötige Kommunikation auf die Performance auswirkt. Das Ergebnis zeigt, dass sich die Performance etwas verschlechtert. Dieser Nachteil wird jedoch durch die vielen Vorteile einer verteilten TOSCA-Runtime aufgewogen.

Ausblick

Das im Laufe der Arbeit entwickelte Konzept zielt darauf ab, die Bereitstellung und Wartung von IoT Anwendungen möglichst vollständig zu automatisieren. Die Automatisierung wird bereits weitestgehend erreicht. Das benötigte manuelle Anlegen der Instanzdaten für die Verteilungsentscheidung verhindert aber derzeit eine komplette Automatisierung. Eine zukünftige Erweiterung sollte deshalb zunächst diesen Prozess ebenfalls automatisieren. Dazu kann zum Beispiel eine *Device Discovery* Komponente in die TOSCA-Runtime eingebunden werden [BSMS01]. Diese Komponente könnte über verschiedene Netzwerk- und Discovery-Protokolle ermitteln, welche Geräte in ihrem Netzwerk vorhanden sind und für diese automatisch geeignete Instanzdaten anlegen.

Eine weitere sinnvolle Erweiterung wäre die Verteilung von Plänen in der verteilten TOSCA-Runtime. Im aktuellen Konzept wird die Management Infrastruktur und damit die IAs zwar zu den TOSCA-Runtime Knoten bei den jeweiligen Devices gebracht, die Pläne werden

aber nach wie vor zentralisiert ausgeführt. Das bedeutet, der komplette Plan wird in der Process Engine der TOSCA-Runtime abgearbeitet, bei der die CSAR hochgeladen wurde. Für jeden Aufruf einer Operation an einem Node Type bzw. eines IAs muss die Process Engine also den lokalen TOSCA-Runtime Knoten aktivieren, der dann das Deployment und den Aufruf zusammen mit den anderen Knoten des verteilten Systems ausführt. So werden die Probleme bei der Provisionierung einer verteilten Anwendung, wie zum Beispiel aktive Firewalls, gelöst. Die Performance kann aber noch weiter verbessert werden. Dazu muss der Build Plan so generiert werden, dass für jeden Teil einer Anwendung, der von einem anderen TOSCA-Runtime Knoten verwaltet wird, ein extra Plan entsteht. Dann kann dieser lediglich einmal übertragen und dann lokal ausgeführt werden, anstatt bei jeder Operation eine Interaktion der Knoten der verteilten TOSCA-Runtime zu benötigen.

Da die Funktionalität für das Instanzdaten Matching im Konzept dieser Arbeit bereits beschrieben und umgesetzt wurde, kann darauf in zukünftigen Arbeiten ein Ansatz für die sogenannte *Service Komposition* aufgebaut werden [MM04]. Bei der Service Komposition werden aus bestehenden Anwendungen neue Anwendungen generiert, die eine neue, bisher nicht vorhandene Funktionalität anbieten. Außerdem kann es damit ermöglicht werden, vorhandene Soft- und Hardware Komponenten wiederzuverwenden. Wenn zum Beispiel in einer Anwendung eine virtuelle Maschine mit einem Tomcat Server erzeugt wird, um anschließend eine WAR Datei darauf zu übertragen, kann der Tomcat Server von einer anderen Anwendung mit einer anderen WAR Datei ebenfalls verwendet werden. Somit kann die Erzeugung einer neuen virtuellen Maschine und die Installation des Tomcat Servers bei der zweiten Anwendung wegfallen, wodurch sich die Zeit der Instanzerzeugung sehr stark verringert. Interessante Fragen, die im Zuge eines solchen Ansatzes auftreten, sind zum Beispiel, wann vorhandene Soft- und Hardware wiederverwendet werden kann, ohne die Performance der Anwendungen zu beeinträchtigen und wie Sicherheitsanforderungen für unterschiedliche Anwendungen, trotz der Wiederverwendung, erfüllt werden können.

A. Messwerte der empirischen Studie

In diesem Abschnitt werden die genauen Messwerte, die den Mittelwerten aus Tabelle 7.2 zugrunde liegen, präsentiert. Dabei wurde für jede Zeile der Mittelwerttabelle, also für jede Kombination aus CSAR und Konzept, eine extra Tabelle erstellt. Diese Tabellen beinhalten jeweils 20 Zeilen für die 20 durchgeführten Testläufe. Jede Zeile enthält dabei die Zeit für das CSAR Deployment und für die Erstellung von fünf Instanzen in diesem Durchlauf. Die Angabe der Zeiten erfolgt immer im Format Minuten:Sekunden:Millisekunden.

Durchlauf Nr.	CSAR Deployment	1. Instanz	2. Instanz	3. Instanz	4. Instanz	5. Instanz
1	1:39:42	4:13:43	3:57:37	4:07:59	4:46:02	3:58:59
2	1:35:18	4:00:71	4:11.59	4:06:91	4:07:75	4:02:19
3	1:38:29	4:18:85	4:03:17	4:00:02	3:59:23	4:03:30
4	1:22:97	4:17:36	4:07:90	4:04:83	3:56:98	4:11:54
5	1:33:75	4:16:91	4:43:17	4:03:55	4:01:87	4:09:65
6	1:41:08	4:14:83	4:05:91	4:03:23	4:07:88	4:04:17
7	1:27:20	4:09:70	4:08:51	4:08:91	4:01:27	4:10:90
8	1:31:91	4:19:22	4:05:34	3:59:71	4:08:66	4:09:94
9	1:35:73	4:08:41	4:01:77	4:05:19	4:03:51	4:00:98
10	1:30:65	4:13:00	4:05:78	4:10:91	3:58:88	4:05:69
11	1:33:55	4:13:66	4:00:79	4:03:31	4:05:76	3:55:99
12	1:33:67	4:21:30	4:08:39	4:11.69	4:07:41	4:03:65
13	1:30:91	4:18:11	4:11:78	4:05:57	3:59:92	4:09:67
14	1:35:17	4:09:78	4:03:81	4:08:71	4:07:85	4:02:39
15	1:29:85	4:11:53	4:02:57	4:02:90	4:07:64	4:05:11
16	1:33:80	4:16:62	4:10:96	4:06:23	4:05:58	4:03:87
17	1:32:99	4:09:76	4:03:54	4:22:15	4:03:17	4:00:60
18	1:38:31	4:17:23	4:04:41	4:01:77	4:06:21	3:58:45
19	1:31:50	4:11:78	4:07:90	4:06:88	4:02:31	4:04:08
20	1:34:76	4:16:31	3:59:52	4:17:82	4:08:41	4:02:96

Tabelle A.1.: CSAR Nr. 1, altes Konzept

Durchlauf Nr.	CSAR Deployment	1. Instanz	2. Instanz	3. Instanz	4. Instanz	5. Instanz
1	1:08:39	5:06:63	4:26:46	4:02:11	3:47:59	3:52:73
2	1:14:52	5:21:35	4:02:17	4:12:46	3:44:53	4:34:51
3	1:02:80	5:13:17	4:15:62	4:22:15	3:51:81	4:13:33
4	1:14:61	5:15:50	4:07:22	4:11:55	4:14:71	4:24:23
5	1:07:93	5:13:67	4:19:86	4:08:90	4:09:94	4:12:33
6	0:59:32	5:05:79	4:13:22	4:16:37	4:14:62	4:12:28
7	1:03:11	5:08:49	4:02:35	4:13:96	4:18:15	3:55:88
8	1:06:76	5:11:11	4:15:17	4:16:76	3:53:63	4:09:42
9	0:59:82	5:07:89	3:51:87	4:00:69	4:11:21	4:05:92
10	1:02:99	5:14:15	3:55:66	4:13:71	4:15:65	4:04:11
11	1:10:00	5:11:07	4:13:21	4:20:53	4:09:71	4:23:63
12	1:08:34	4:49:48	4:09:45	4:22:84	3:51:61	4:19:74
13	1:03:22	5:15:36	4:16:75	4:15:31	4:17:22	4:19:73
14	1:02:67	4:59:62	4:17:48	4:04:31	4:13:80	3:59:30
15	1:02:55	5:14:21	4:19:72	4:15:45	4:09:72	4:13:83
16	1:00:97	5:16:29	4:18:47	4:13:83	4:05:69	4:16:64
17	1:04:03	5:10:91	4:09:15	4:13:85	3:47:53	4:07:42
18	0:59:56	4:51:69	4:08:17	4:07:49	3:50:59	4:14:31
19	1:06:33	5:09:54	4:14:71	4:13:85	4:09:61	3:54:83
20	1:05:82	4:57:38	4:01:38	4:14:41	4:18:97	4:09:55

Tabelle A.2.: CSAR Nr. 1, neues Konzept ohne Verteilung

Durchlauf Nr.	CSAR Deployment	1. Instanz	2. Instanz	3. Instanz	4. Instanz	5. Instanz
1	1:08:56	5:31:84	4:33:84	4:24:77	4:27:83	4:29:11
2	1:05:91	5:17:47	4:39:30	4:22:52	4:29:73	4:19:55
3	0:58:77	5:32:11	4:22:61	4:19:53	4:26:48	4:25:37
4	1:03:90	5:22:81	4:30:90	4:28:11	4:31:22	4:27:61
5	1:10:76	5:24:27	4:19:76	4:24:74	4:19:27	4:17:55
6	1:09:32	5:20:09	4:27:57	4:33:15	4:23.26	4:26:79
7	1:05:60	5:21:81	4:30:31	4:22:83	4:27:43	4:20:07
8	1:10:98	5:30:65	4:25:74	4:24:43	4:27:82	4:31:00
9	1:04:54	5:28:90	4:26:42	4:29:37	4:19:87	4:18:77
10	1:07:11	5:26:33	4:28:33	4:27:83	4:32:40	4:22:41
11	0:59:52	5:24:69	4:19:18	4:22:04	4:25:66	4:24:98
12	1:00:97	5:33:02	4:36:78	4:22:76	4:25:46	4:30:15
13	1:08:04	5:18:88	4:29:47	4:28:58	4:21:99	4:25:52
14	1:06:12	5:26:91	4:27:56	4:31:02	4:28:56	4:27:79
15	1:02:53	5:25:43	4:28:74	4:15:51	4:23:84	4:26:92
16	1:01:57	5:28:11	4:23:16	4:26:54	4:20:84	4:17:36
17	1:03:66	5:19:73	4:30:24	4:20:12	4:23:70	4:24:00
18	1:05:89	5:20:08	4:20:78	4:19:93	4:27:84	4:23:62
19	1:00:54	5:22:66	4:24:44	4:20:73	4:24:03	4:28:56
20	1:10:72	5:19:81	4:29:73	4:23:11	4:37:20	4:24:34

Tabelle A.3.: CSAR Nr. 1, neues Konzept mit lokaler Verteilung

Durchlauf Nr.	CSAR Deployment	1. Instanz	2. Instanz	3. Instanz	4. Instanz	5. Instanz
1	1:10:55	5:41:96	4:29:65	4:28:82	4:32:87	4:31:09
2	1:08:95	5:39:73	4:32:15	4:32:68	4:30:25	4:29:93
3	0:59:41	5:45:31	4:33:09	4:29:78	4:30:52	4:29:29
4	1:08:60	5:47:60	4:28:70	4:35:24	4:30:21	4:30:55
5	1:05:13	5:47:93	4:33:50	4:28:76	4:29:92	4:30:36
6	1:09:62	5:43:11	4:30:88	4:27:76	4:31:82	4:32:99
7	1:03:17	5:46:23	4:29:90	4:29:52	4:33:84	4:32:29
8	1:04:09	5:45:06	4:33:43	4:30:77	4:31:23	4:29:01
9	1:05:67	5:44:43	4:30:13	4:32:67	4:28:83	4:29:26
10	1:08:99	5:49:02	4:32:86	4:30:70	4:29:61	4:30:54
11	1:06:42	5:43:76	4:31:05	4:29:76	4:31.90	4:28:77
12	1:00:80	5:39:83	4:33:63	4:32:87	4:30:55	4:30:32
13	1:03:76	5:44:46	4:29:87	4:31:65	4:28:76	4:30:11
14	1:05:24	5:43:71	4:27:90	4:30:08	4:29:49	4:31:31
15	1:06:50	5:45:32	4:30:44	4:31:77	4:32.21	4:30:20
16	1:04:33	5:42:18	4:31:32	4:29:83	4:30:01	4:29:42
17	1:09:04	5:38:94	4:38:07	4:33:19	4:32:52	4:30:44
18	0:59:82	5:45:67	4:30:65	4:31:64	4:29:03	4:38:23
19	1:07:45	5:46:08	4:29:21	4:31.34	4:30:06	4:27:60
20	1:04:47	5:47:43	4:31:70	4:30:94	4:29:56	4:30:20

Tabelle A.4.: CSAR Nr. 1, neues Konzept mit globaler Verteilung

Durchlauf Nr.	CSAR Deployment	1. Instanz	2. Instanz	3. Instanz	4. Instanz	5. Instanz
1	0:38:92	2:52:11	2:55:17	2:54:40	2:50:83	2:52:76
2	0:38:71	2:55:90	2:52:75	2:53:64	2:53:06	2:52:16
3	0:39:16	2:54:76	2:53:45	2:52:65	2:50:97	2:52:33
4	0:38:44	2:53:61	2:52:87	2:50:72	2:51:32	2:51:60
5	0:36:80	2:54:24	2:55:08	2:53:63	2:53:46	2:52:41
6	0:37:55	2:52:65	2:53:44	2:51:83	2:52:56	2:51:99
7	0:39:67	2:56:37	2:55:60	2:53:78	2:54:11	2:53:08
8	0:39:78	2:52:44	2:53:76	2:52:21	2:52:88	2:51:24
9	0:38:70	2:53:00	2:52:56	2:52:13	2:54:04	2:51:78
10	0:40:03	2:52:48	2:50:38	2:51:91	2:51:26	2:51:33
11	0:38:34	2:54:67	2:54:70	2:52:43	2:52:98	2:53:11
12	0:38:25	2:50:86	2:51:43	2:53:01	2:52:72	2:52:30
13	0:39:79	2:53:70	2:53:40	2:53:79	2:52:17	2:52:45
14	0:38:63	2:52:35	2:54:43	2:52:80	2:51:42	2:52:07
15	0:37:11	2:58:46	2:53:68	2:54:63	2:50:44	2:52:46
16	0:40:10	2:53:44	2:53:64	2:52:33	2:51:98	2:52:13
17	0:38:28	2:57:31	2:52:46	2:54:89	2:53:24	2:53:41
18	0:39:66	2:52:86	2:53:67	2:53:55	2:52:20	2:52:08
19	0:38:73	2:53:95	2:52:51	2:53:95	2:52:17	2:53:00
20	0:38:91	2:53:43	2:52:48	2:52:36	2:53:44	2:52:75

Tabelle A.5.: CSAR Nr. 2, altes Konzept

Durchlauf Nr.	CSAR Deployment	1. Instanz	2. Instanz	3. Instanz	4. Instanz	5. Instanz
1	0:25:64	3:06:79	2:54:87	2:55:81	2:54:03	2:53:53
2	0:26:21	3:05:55	2:57:44	2:55:67	2:54:05	2:54:68
3	0:25:43	3:08:01	2:53:90	2:54:64	2:53:22	2:54:05
4	0:25:70	3:06:81	2:54:35	2:54:74	2:54:21	2:54:42
5	0:25:91	3:06:73	2:55:68	2:55:34	2:54:13	2:54:02
6	0:25:88	3:06:34	2:56:87	2:55:79	2:55:43	2:54:27
7	0:26:08	3:07:04	2:55:76	2:55:43	2:54:26	2:53:08
8	0:24:99	3:05:99	2:53:69	2:54:12	2:54:38	2:55:07
9	0:25:87	3:07:24	2:56:78	2:57:32	2:56:83	2:55:23
10	0:26:32	3:09:13	2:55:82	2:56:30	2:56:32	2:55:71
11	0:25:10	3:07:34	2:57:66	2:56:42	2:55:79	2:55:53
12	0:25:92	3:05:56	2:55:98	2:56:72	2:54:63	2:54:33
13	0:26:34	3:06:11	2:56:81	2:54:78	2:55:32	2:55:23
14	0:25:21	3:04:78	2:53:56	2:54:53	2:54:48	2:55:04
15	0:25:56	3:07:35	2:55:57	2:54:47	2:54:13	2:55:94
16	0:24:96	3:06:60	2:56:46	2:55:39	2:55:67	2:53:82
17	0:25:77	3:05:42	2:55:18	2:56:26	2:54:72	2:54:90
18	0:25:91	3:04:83	2:53:99	2:54:78	2:54:23	2:54:86
19	0:25:45	3:04:65	2:57:21	2:56:42	2:56:17	2:54:69
20	0:25:69	3:07:77	2:56:55	2:55:52	2:55:70	2:55:36

Tabelle A.6.: CSAR Nr. 2, neues Konzept ohne Verteilung

Durchlauf Nr.	CSAR Deployment	1. Instanz	2. Instanz	3. Instanz	4. Instanz	5. Instanz
1	0:25:81	3:11:03	2:58:31	2:58:03	2:58:43	2:57:85
2	0:24:98	3:10:67	2:59:11	2:58:32	2:58:92	2:57:55
3	0:25:05	3:11:17	2:58:78	2:58:21	2:57:97	2:58:04
4	0:26:13	3:11:43	2:59:66	2:58:21	2:57:33	2:57:45
5	0:25:15	3.11:87	2:58:90	2:55:34	2:58:08	2:58:23
6	0:24:86	3:10:98	2:58:02	2:57:65	2:57:81	2:58:09
7	0:25:49	3:12:06	2:59:42	2:57:24	2:57:02	2:58:25
8	0:25:36	3:11:54	3:00:44	2:58:01	2:57:61	2:57:23
9	0:25:63	3:11:32	2:57:93	2:58:18	2:57:66	2:58:14
10	0:26:25	3:12:10	2:58:20	2:58:16	2:57:57	2:58:09
11	0:25:11	3:11:36	2:59:12	2:57:63	2:57:82	2:58:44
12	0:25:43	3:11:22	2:58:62	2:55:32	2:57:99	2:58:22
13	0:25:88	3:10:76	2:59:01	2:58:24	2:57:45	2:58:19
14	0:25:25	3:10:34	2:57:33	2:58:02	2:57:81	2:58:64
15	0:24:54	3:11:42	2:59:13	2:58:74	2:58:32	2.57:77
16	0:25:90	3:11:67	2:58:46	2:57:82	2:56:21	2:58:06
17	0:25:42	3:11:80	2:58:37	2:58:59	2:59:30	2:58:23
18	0:25:83	3.12:08	2:57:60	2:57:34	2:58:91	2:58:54
19	0:25:77	3:10:54	2:57:05	2:58:20	2:58:08	2:59:12
20	0:25:06	3:11:21	2:58:24	2:58:27	2:57:66	2:58:34

Tabelle A.7.: CSAR Nr. 2, neues Konzept mit lokaler Verteilung

Durchlauf Nr.	CSAR Deployment	1. Instanz	2. Instanz	3. Instanz	4. Instanz	5. Instanz
1	0:25:32	3:13:64	2:58:62	2:57:99	2:57:51	2:58:67
2	0:25:96	3:12:55	2:59:60	2:58:03	2:57:71	2:57:66
3	0:25:17	3:13:09	2:57:23	2:58:10	2:58:16	2:58:22
4	0:26:00	3:13:43	2:58:74	2:57:98	2:59:76	2:57:84
5	0:25:03	3:13:12	2:58:08	2:59:62	2:58:74	2:58:32
6	0:24:76	3:12:87	2:56:72	2:57:65	2:58:44	2:57:06
7	0:25:53	3:14:04	2:58:77	2:58:50	2:57:41	2:58:21
8	0:25:24	3:13:31	2:58:67	2:57:81	2:58:72	2:57:90
9	0:25:63	3:12:92	2:57:56	2:56:88	2:57:52	2:58:00
10	0:25:35	3:13:45	2:58:39	2:58:46	2:57:11	2:57:51
11	0:26:12	3:12:69	2:59:02	2:59:12	2:58:66	2:58:77
12	0:25:73	3:13:77	2:56:79	2:58:10	2:57:45	2:58:33
13	0:25:94	3:14:56	2:58:55	2:59:27	2:58:53	2:58:62
14	0:25:44	3:13:44	2:56:34	2:56:80	2:57:41	2:57:78
15	0:24:76	3:13:73	2:58:26	2:57:06	2:58:67	2:58:01
16	0:25:81	3:13:47	2:57:18	2:58:47	2:59:09	2:58:11
17	0:25:18	3:14:15	2:58:20	2:57:42	2:58:63	2:58:27
18	0:25:94	3:14:06	2:58:25	2:55:62	2:58:90	2:57:64
19	0:25:82	3:12:61	2:57:21	2:58:88	2:58:51	2:58:03
20	0:25:73	3:12:43	2:58:33	3:00:54	2:57:47	2:57:80

Tabelle A.8.: CSAR Nr. 2, neues Konzept mit globaler Verteilung

Literaturverzeichnis

[ABD+07] J. R. Albrecht, R. Braud, D. Dao, N. Topilski, C. Tuttle, A. C. Snoeren, A. Vahdat. „Remote Control: Distributed Application Configuration, Management, and Visualization with Plush“. In: *Large Installation System Administration (LISA) Conference* (2007), S. 1–19 (zitiert auf S. 42).

[AGM+15] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, M. Ayyash. „Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications“. In: *IEEE Communications Surveys & Tutorials*. IEEE (2015), S. 2347–2376 (zitiert auf S. 25).

[Alb07] J. R. Albrecht. „Distributed Application Management“. Diss. University of California, San Diego, (2007) (zitiert auf S. 42, 43).

[BBB+09] R. Balter, L. Bellissard, F. Boyer, M. Riveill, J.-Y. Vion-Dury. „Architecturing and Configuring Distributed Application with Olan“. In: *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer (2009), S. 241–256 (zitiert auf S. 41, 42).

[BBH+13] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, S. Wagner. „OpenTOSCA – A Runtime for TOSCA-based Cloud Applications“. In: *11^{th} International Conference on Service-Oriented Computing*. Springer LNCS (2013) (zitiert auf S. 35–38, 61).

[BBK+12] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann, D. Schumm. „Vino4TOSCA: A Visual Notation for Application Topologies Based on TOSCA“. In: *OTM 2012, Part I* (2012), S. 416–424 (zitiert auf S. 18).

[BBK+14] U. Breitenbücher, T. Binz, K. Képes, O. Kopp, F. Leymann, J. Wettinger. „Combining Declarative and Imperative Cloud Application Provisioning based on TOSCA“. In: *Proceedings of the IEEE International Conference on Cloud Engineering (IEEE IC2E)*. IEEE Computer Society (2014), S. 87–96 (zitiert auf S. 32–35).

[BBLS12] T. Binz, G. Breiter, F. Leymann, T. Spatzier. „Portable Cloud Services Using TOSCA“. In: *IEEE Internet Computing*. IEEE Computer Society (2012), S. 80–85 (zitiert auf S. 32, 34).

[BKLW17] U. Breitenbücher, K. Képes, F. Leymann, M. Wurster. „Declarative vs. Imperative: How to Model the Automated Deployment of IoT Applications?“ In: *Proceedings of the 11^{th} Advanced Summer School on Service Oriented Computing*. IBM Research Division (2017), S. 18–27 (zitiert auf S. 35).

[BSMS01] M. S. Borella, G. M. Schuster, J. J. Mahler, I. Sidhu. *Protocol and Method for Peer Network Device Discovery*. US Patent 6,269,099. Juli 2001 (zitiert auf S. 65, 100).

[Bus95] F. Buschmann. „The Master-Slave Pattern“. In: *Pattern Languages of Program Design*. ACM Press/Addison-Wesley Publishing Co. (1995), S. 133–142 (zitiert auf S. 56).

[CMF+16] A. Celesti, D. Mulfari, M. Fazio, M. Villari, A. Puliafito. „Exploring Container Virtualization in IoT Clouds“. In: *2016 IEEE International Conference on Smart Computing (SMARTCOMP)*. IEEE (2016), S. 1–6 (zitiert auf S. 38).

[CZ16] M. Chiang, T. Zhang. „Fog and IoT: An Overview of Research Opportunities“. In: *IEEE Internet of Things Journal*. IEEE (2016), S. 854–864 (zitiert auf S. 41, 44).

[DBN14] S. K. Datta, C. Bonnet, N. Nikaein. „An IoT Gateway Centric Architecture to Provide Novel M2M Services“. In: *2014 IEEE World Forum on Internet of Things (WF-IoT)*. IEEE (2014), S. 514–519 (zitiert auf S. 65).

[Doca] Docker. *Docker Compose Dokumentation*. https://docs.docker.com/compose/ (zitiert auf S. 39).

[Docb] Docker. *Docker Webseite*. https://docs.docker.com (zitiert auf S. 38, 39).

[Ecl] Eclipse Foundation. *Equinox*. http://www.eclipse.org/equinox/ (zitiert auf S. 27).

[HW04] G. Hohpe, B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, (2004) (zitiert auf S. 24, 25, 28, 77, 78).

[IA18] C. Ibsen, J. Anstey. *Camel in Action*. Manning Publications Co., (2018) (zitiert auf S. 28–31, 76).

[KBBL13] O. Kopp, T. Binz, U. Breitenbücher, F. Leymann. „Winery – A Modeling Tool for TOSCA-based Cloud Applications“. In: *International Conference on Service-Oriented Computing*. Springer (2013), S. 700–704 (zitiert auf S. 35, 47).

[Kép13] K. Képes. „Konzept und Implementierung einer Java-Komponente zur Generierung von WS-BPEL 2.0 BuildPlans für OpenTOSCA“. (2013) (zitiert auf S. 37).

[Ker92] H. Kerner. *Rechnernetze nach OSI*. Addison-Wesley, (1992) (zitiert auf S. 25).

[KM] A. Kelly, D. McCreary. *Webinar - Making Sense of NoSQL*. https://www.slideshare.net/Couchbase/webinar-making-sense-of-nosql-applying-nonrelational-databases-to-business-needs (zitiert auf S. 56).

[KPM] KPMG AG und Bitkom Research GmbH. *Cloud-Monitor 2018*. https://www.bitkom.org/Presse/Anhaenge-an-PIs/2018/180607-Bitkom-KPMG-PK-Cloud-Monitor-2.pdf (zitiert auf S. 31).

[LAB+18] A. Liebing, L. Ashauer, U. Breitenbücher, T. Günther, M. Hahn, K. Képes, O. Kopp, F. Leymann, B. Mitschang, A. C. F. da Silva, R. Steinke. „The SmartOrchestra Platform: A Configurable Smart Service Platform for IoT Systems“. In: *Papers from the 12th Advanced Summer School on Service-Oriented Computing (SummerSoC 2018)*. IBM Research Division (2018), S. 14–21 (zitiert auf S. 18).

[LKHJ13] S. Lee, H. Kim, D.-k. Hong, H. Ju. „Correlation Analysis of MQTT Loss and Delay According to QoS Level“. In: *International Conference on Information Networking (ICOIN)*. IEEE (2013), S. 714–717 (zitiert auf S. 26).

[LNH03] C. Lee, D. Nordstedt, S. Helal. „Enabling Smart Spaces with OSGi“. In: *IEEE Pervasive Computing*. IEEE (2003), S. 89–94 (zitiert auf S. 27, 28).

[LP+03] Y. Liu, B. Plale et al. „Survey of Publish Subscribe Event Systems“. In: *Computer Science Dept, Indian University* (2003) (zitiert auf S. 24).

[LVCD13] F. Li, M. Vogler, M. Claeßens, S. Dustdar. „Towards Automated IoT Application Deployment by a Cloud-based Approach“. In: *2013 IEEE 6th International Conference on Service-Oriented Computing and Applications (SOCA)*. IEEE (2013), S. 61–68 (zitiert auf S. 23, 34).

[MG+11] P. Mell, T. Grance et al. „The NIST Definition of Cloud Computing“. In: *National Institute of Standards and Technology* (2011) (zitiert auf S. 31).

[MM04] N. Milanovic, M. Malek. „Current Solutions for Web Service Composition“. In: *IEEE Internet Computing*. IEEE (2004), S. 51–59 (zitiert auf S. 101).

[NSL+14] S. Nastic, S. Sehic, D.-H. Le, H.-L. Truong, S. Dustdar. „Provisioning Software-defined IoT Cloud Systems“. In: *2014 International Conference on Future Internet of Things and Cloud (FiCloud)*. IEEE (2014), S. 288–295 (zitiert auf S. 17, 23).

[OASa] OASIS. *MQTT Version 3.1.1 Plus Errata 01. (Dez. 2015)*. URL: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/errata01/os/mqtt-v3.1.1-errata01-os-complete.html (zitiert auf S. 25, 76).

[OASb] OASIS. *MQTT Version 5.0. (Mai 2018)*. URL: http://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html (zitiert auf S. 25).

[OASc] OASIS. *Topology and Orchestration Specification for Cloud Applications Primer Version 1.0. (Jan. 2013)*. URL: http://docs.oasis-open.org/tosca/tosca-primer/v1.0/tosca-primer-v1.0.html (zitiert auf S. 35, 47–50, 60, 62, 64).

[OASd] OASIS. *Topology and Orchestration Specification for Cloud Applications Version 1.0. (Nov. 2013)*. URL: http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html (zitiert auf S. 32–34, 63, 64).

[OASe] OASIS. *TOSCA Simple Profile in YAML Version 1.2. (Juli 2018)*. URL: http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.2/TOSCA-Simple-Profile-YAML-v1.2.pdf (zitiert auf S. 36, 49, 51, 54, 62, 64).

[Ope] OpenTOSCA Team. *OpenTOSCA Ökosystem - Github*. https://github.com/OpenTOSCA (zitiert auf S. 17, 36–38, 62).

[Opp97] R. Oppliger. „Internet Security: Firewalls and Beyond“. In: *Communications of the ACM*. ACM (1997), S. 92–102 (zitiert auf S. 58).

[OSG] OSGi Alliance. *Specifications - OSGi Alliance*. https://www.osgi.org/developer/specifications/ (zitiert auf S. 27).

[RH09] K. Roberts-Hoffman, P. Hegde. „ARM Cortex-A8 vs. Intel Atom: Architectural and Benchmark Comparisons“. In: *Dallas: University of Texas at Dallas* (2009) (zitiert auf S. 84).

[RMPC16] M. A. Razzaque, M. Milojevic-Jevric, A. Palade, S. Clarke. „Middleware for Internet of Things: A Survey“. In: *IEEE Internet of Things Journal*. IEEE (2016), S. 70–95 (zitiert auf S. 17, 22).

[SBH+17] A. C. F. da Silva, U. Breitenbücher, P. Hirmer, K. Képes, O. Kopp, F. Leymann, B. Mitschang, R. Steinke. „Internet of Things Out of the Box: Using TOSCA for Automating the Deployment of IoT Environments“. In: *Proceedings of the 7th International Conference on Cloud Computing and Services Science (CLOSER)*. SciTePress Digital Library (2017), S. 358–367 (zitiert auf S. 17, 21–24, 32).

[SBK+16] A. C. F. da Silva, U. Breitenbücher, K. Képes, O. Kopp, F. Leymann. „OpenTOSCA for IoT: Automating the Deployment of IoT Applications based on the Mosquitto Message Broker“. In: *Proceedings of the 6th International Conference on the Internet of Things*. ACM (2016), S. 181–182 (zitiert auf S. 21, 22, 24, 25, 50).

[SBKL17] K. Saatkamp, U. Breitenbücher, O. Kopp, F. Leymann. „Topology Splitting and Matching for Multi-Cloud Deployments“. In: *Proceedings of the 7th International Conference on Cloud Computing and Services Science (CLOSER)*. SciTePress (2017), S. 247–258 (zitiert auf S. 64).

[SF02] D. Schoder, K. Fischbach. „Peer-to-Peer“. In: *Wirtschaftsinformatik*. Springer (2002), S. 587–589 (zitiert auf S. 56).

[SGFW10] H. Sundmaeker, P. Guillemin, P. Friess, S. Woelfflé. „Vision and Challenges for Realising the Internet of Things“. In: *Cluster of European Research Projects on the Internet of Things, European Commision* (2010), S. 34–36 (zitiert auf S. 21, 22).

[SSBL16] O. Skarlat, S. Schulte, M. Borkowski, P. Leitner. „Resource Provisioning for IoT Services in the Fog“. In: *2016 IEEE 9th Conference on Service-Oriented Computing and Applications (SOCA)*. IEEE (2016), S. 32–39 (zitiert auf S. 44, 45).

[Tur14] J. Turnbull. *The Docker Book: Containerization is the new Virtualization*. James Turnbull, (2014) (zitiert auf S. 38, 39).

[VRCL08] L. M. Vaquero, L. Rodero-Merino, J. Caceres, M. Lindner. „A Break in the Clouds: Towards a Cloud Definition“. In: *ACM SIGCOMM Computer Communication Review*. ACM (2008), S. 50–55 (zitiert auf S. 17, 31).

[Zim16] M. Zimmermann. „Konzept und Implementierung einer Komponente zur Kommunikation TOSCA-basierter Anwendungen“. (2016) (zitiert auf S. 36–38, 62, 81, 83).

Alle URLs wurden zuletzt am 08. 01. 2019 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift