Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Finden einer geeigneten Infrastruktur für Datenoperationen in IoT-Umgebungen

Jan Schneider

Studiengang:	Softwaretechnik
Prüfer/in:	PD Dr. rer. nat. habil. Holger Schwarz
Betreuer/in:	Ana C. F. da Silva, M. Sc. Dipl.-Inf. Pascal Hirmer
Beginn am:	1. Februar 2018
Beendet am:	1. August 2018

Kurzfassung

Mit der rasant ansteigenden Anzahl an internetfähigen physischen Objekten nimmt auch die Bedeutung des Internets der Dinge (IoT) zu. Es ermöglicht die Vernetzung von unterschiedlichen Geräten, die mit Sensoren oder Aktuatoren ausgestattet sind und schafft auf diese Weise adaptive und flexible Netzwerke, die ihre Umgebung wahrnehmen und mit ihr interagieren können. Durch diese Entwicklung wächst gleichzeitig auch die Menge an Daten, die innerhalb dieser Netzwerke verarbeitet werden müssen. Um mit den damit verbundenen Problemen umzugehen, wird die Datenverarbeitung häufig in leistungsfähige Cloud-Systeme ausgelagert. Dadurch müssen jedoch alle erfassten Daten zunächst aufwändig in die Cloud übermittelt werden, was zu hoher Netzwerkauslastung und dementsprechend langen Latenzzeiten führt. Dies ist für viele IoT-Systeme inakzeptabel, da sie in Echtzeit auf sich ändernde Bedingungen in ihrer Umgebung reagieren können müssen. In der Folge werden Lösungen benötigt, um die Daten verteilt und mit möglichst kurzen Transportwegen verarbeiten zu können. Allerdings setzen sich IoT-Umgebungen in der Regel aus heterogenen Geräten zusammen, die unterschiedliche Hardware- und Softwareeigenschaften aufweisen, sodass anspruchsvollere Datenoperationen nur auf bestimmten Geräten ausgeführt werden können. Diese Einschränkungen müssen bei der verteilten Ausführung berücksichtigt werden. Um zu beschreiben, wie Daten in einem IoT-System verarbeitet werden, können Datenstrommodelle eingesetzt werden. Im Rahmen dieser Bachelorarbeit werden Konzepte entwickelt und implementiert, die es ermöglichen, Geschäftslogik, die in Form von Datenstrommodellen spezifiziert ist, verteilt in IoT-Umgebungen auszuführen. Dabei wird dynamisch entschieden, auf welchen Geräten die Datenoperationen ausgeführt werden sollen, um sowohl eine korrekte Ausführung der Geschäftslogik zu gewährleisten, als auch die Transportwege der Daten zu minimieren.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Listings

Verzeichnis der Algorithmen

Abkürzungsverzeichnis

DBMS Datenbankmanagementsystem. 27

HTTP Hypertext Transfer Protocol. 86

IoT Internet der Dinge. 17, 23

IPVS Institut für Parallele und Verteilte Systeme der Universität Stuttgart. 55

JSON JavaScript Object Notation. 52

MBP Multi-purpose Binding and Provisioning Platform. 32

MQTT Message Queue Telemetry Transport. 89

REST Representational State Transfer. 35

UML Unified Modeling Language. 102

URL Uniform Resource Locator. 108

1 Einführung und Motivation

Dieses Kapitel enthält eine Einführung über die Thematik, die in dieser Bachelorarbeit behandelt wird und erläutert die dahinterstehende Motivation.

Das Internet der Dinge (IoT) [MSPC12; VF13] ist eine aufkommende und sich breitflächig weiterentwickelnde Technologie, die es ermöglicht, internetfähige Geräte verschiedener Art in Netzwerke zu integrieren und damit zu globalen oder lokalen Infrastrukturen zu verbinden, über die die Geräte miteinander Daten austauschen und interagieren können [MSPC12]. Typischerweise sind diese Geräte mit Sensoren oder Aktuatoren ausgestattet, sodass sie in der Lage sind, ihre Umwelt wahrzunehmen oder zu beeinflussen. Das IoT als wirtschaftliches Geschäftsfeld befindet sich derzeit in einer Phase anhaltenden Wachstums: Laut Gartner, Inc. [Gar17] werden Schätzungen zufolge weltweit derzeit pro Minute über 5000 neue internetfähige Geräte dem IoT hinzugefügt und es wird erwartet, dass bis 2020 mehrere Zehnmilliarden verbundene Geräte in Betrieb sein werden, was voraussichtlich mehreren Geräten pro auf der Erde lebendem Mensch entsprechen wird.

Das IoT kann für vielfältige Anwendungsfälle eingesetzt werden. So ebnet es in der industriellen Produktion den Weg für die sogenannte Industrie 4.0 [LFK+14]. Außerdem ist es dank handlicher, mobiler Geräte wie Smartphones und Wearables und dem immer weiter sinkenden Energie- und Platzbedarf von Netzwerkschnittstellen inzwischen auch alltagsfähig geworden und erhält beispielsweise in Form von Smart Homes [ARA12] Einzug in nahezu alle Bereiche des Lebens. Weitere Anwendungsbereiche finden sich unter anderen in der Stadtentwicklung, Medizintechnik, der Landwirtschaft, dem Energie- und Umweltmanagement und dem Verkehrswesen.

IoT-Umgebungen bestehen in der Regel aus sehr heterogenen Geräten, die wiederum auf unterschiedliche Weisen miteinander vernetzt sind und innerhalb ihrer Umgebung verschiedene Rollen einnehmen [CM12]. So werden IoT-Geräte, die mittels Sensoren physikalische Daten erfassen, häufig in Form hardwarenah zu programmierender Microcontroller realisiert, wie Arduinos[1] oder Raspberry Pis[2], die sich jeweils stark in den angebotenen Schnittstellen, Hardwareressourcen und weiteren Fähigkeiten unterscheiden können. Daneben können beliebige weitere Netzwerkgeräte in eine IoT-Umgebung eingebunden werden, wie Server oder andere Recheneinheiten, die bestimmte Dienste zur Verfügung stellen. Für besonders leistungs- und rechenintensive Prozesse werden darüber hinaus oft Clouds [MG11] in IoT-Umgebungen eingebunden, welche meist bei Drittanbietern gemietet werden und besonders leistungsstarke Backend-Server umfassen.

In Abbildung 1.1 ist ein Beispiel für eine IoT-Umgebung dargestellt, die aus einer Backend-Cloud, zwei Servern, einem Raspberry Pi und einem Temperatursensor, dessen gemessene Daten von einem Arduino-Microcontroller ausgelesen werden können, besteht. Wie anhand der Pfeile in dem Diagramm zu erkennen ist, befindet sich der Raspberry Pi mit den beiden Servern in einem

[1] Microcontroller-Plattform, siehe https://www.arduino.cc
[2] Einplatinencomputer, siehe https://www.raspberrypi.org

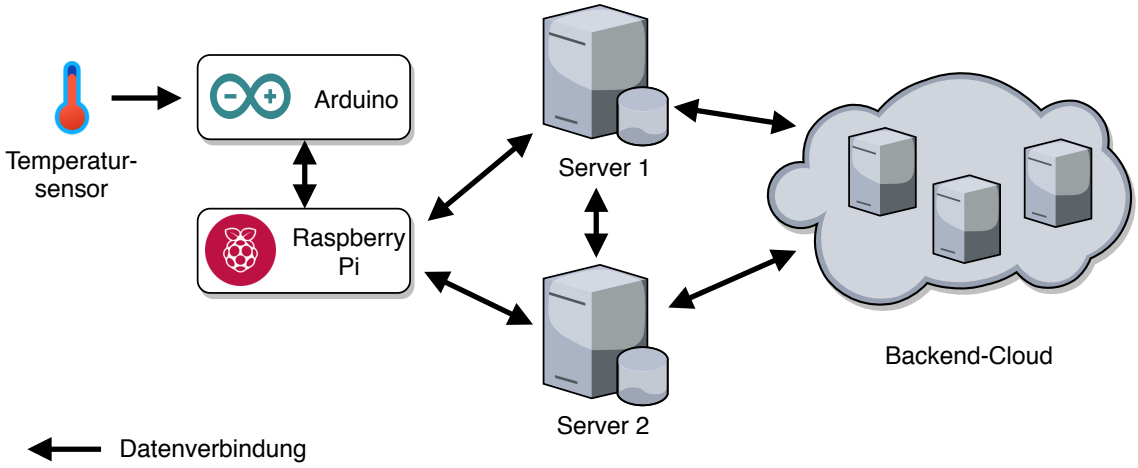

Abbildung 1.1: Schematisches Beispiel für eine IoT-Umgebung

gemeinsamen Netzwerk. Die Backend-Cloud allerdings kann nur von den beiden Servern erreicht werden und nicht von dem Raspberry Pi selbst. Der Arduino-Microcontroller besitzt selbst keine Netzwerkschnittstelle, sondern macht für diesen Zweck von dem Raspberry Pi Gebrauch.

Für viele Anwendungsfälle erscheint es als die bequemste und am einfachsten zu praktizierende Lösung, alle in der jeweiligen IoT-Umgebung erfassten Daten direkt in die Cloud zu transportieren, zentral von einer entsprechenden Software verarbeiten zu lassen und von dort im Anschluss gegebenenfalls vorgesehene Reaktionen auszulösen: Clouds sind leistungsstark, die Anbieter mit ihren großen Rechenzentren kümmern sich selbstständig um den Betrieb und die Wartung und sichern ihren Kunden eine hohe Ausfallsicherheit zu, sodass sich diese nur marginal mit der zugrundeliegenden Infrastruktur auseinandersetzen müssen [WR15].

Doch von einem allgemeinen Blickwinkel aus birgt diese Art der Datenverarbeitung in IoT-Umgebungen auch Probleme und Schwierigkeiten: Der Aufstieg des IoT und die damit verbundene steigende Anzahl gleichzeitig aktiver Geräte geht mit einer insgesamt wachsenden Menge an zu transportierenden und zu verarbeitenden Daten einher. Diese Entwicklung führt zu einer Verschärfung der sogenannten Big-Data-Probleme, welche die Problematiken der effizienten Bewältigung besonders großer Datenmengen formulieren [FHM18]. Durch die Heterogenität der Daten und den hohen Verteilungsgrad innerhalb einer IoT-Umgebung werden für den Datentransport zu einer zentralen Recheneinheit zwangsläufig viele Netzwerkressourcen in Anspruch genommen und damit eine hohe Netzwerkauslastung erzeugt, denn schließlich muss jeder gemessene Wert eines Sensors eigens an die zentrale Verarbeitungsstelle übermittelt werden. Gerade in IoT-Umgebungen ist die Netzwerkinfrastruktur jedoch oft nicht so ausgeprägt, als dass dafür eine ausreichend hohe Bandbreite zur Verfügung stehen würde. In der Folge ist die Datenverarbeitung auf Systemebene mit einer hohen Latenz verbunden, die vielen Anwendungsfällen des IoT, in denen Effizienz und geringe Reaktionszeiten eine entscheidende Rolle spielen, absolut abträglich ist. Auch die Skalierbarkeit eines IoT-Systems wird durch diesen Ansatz beeinträchtigt: Werden der Umgebung weitere IoT-Geräte mit Sensoren hinzugefügt, die ebenfalls ausgewertet und berücksichtigt werden müssen, steigt die Netzwerkauslastung weiter an und verschärft damit das Effizienzproblem. Neben diesen technischen Aspekten ergeben sich weitere Unzulänglichkeiten: Durch die zentralisierte Datenverarbeitung werden alle von der IoT-Umgebung erfassten und potentiell personenbezogenen Daten gemeinsam von einer einzigen Instanz ausgewertet, welche dadurch alleine die Kontrolle

über alle Daten besitzt und sich unabhängig von den an das System gestellten funktionalen Anforderungen aus ein Gesamtbild mit Zusammenhängen zwischen den Daten erstellen kann. Dies führt unweigerlich zu einer Einschränkung der Privatsphäre. Ferner wird bei diesem Ansatz die komplette Kontrolle über die Anwendungen und das System selbst an die Cloud übergeben, was uneingeschränktes Vertrauen der Benutzer der IoT-Geräte in die Cloud erfordert, um die Dienste und Funktionen der IoT-Umgebung in Anspruch nehmen zu können [LME+15]. Die zentralisierte Verarbeitung in der Cloud stellt darüber hinaus auch in Hinsicht auf die Ausfallsicherheit des Systems einen möglichen Schwachpunkt dar: Sollte die Cloud wider Erwarten doch einmal ausfallen oder abgeschaltet werden, wäre das gesamte System nicht mehr funktionsfähig.

Unter Berücksichtigung all dieser Schwierigkeiten zeichnet sich ab, dass die zentralisierte Verarbeitung von Daten keine Universallösung darstellt und insbesondere bei komplexeren Echtzeitsystemen nicht zu empfehlen ist. Als Gegenentwurf zu diesem Ansatz erscheint es stattdessen zweckmäßig, die Verarbeitung von Daten dezentral und möglichst nah an ihrer Quelle durchzuführen: Damit müssten erfasste Sensordaten beispielsweise für eine einfache Plausibilitätsprüfung oder Mittelwertsberechnung nicht erst aufwändig und unter Beanspruchung vieler Netzwerkressourcen in eine Cloud übermittelt werden, denn diese verhältnismäßig einfachen Operationen ließen sich auch problemlos auf leistungsschwächeren Geräten wie Microcontrollern durchführen. In der Regel werden Netzwerkgeräte eher überdimensioniert und besitzen damit zusätzliche Rechenkapazitäten und Ressourcen, die sie zur Verfügung stellen können. Wenn es also möglich ist, derartige Operationen auf Geräten auszuführen, die sich nahe an der Quelle befinden, sollten diese Geräte gegebenenfalls leistungsstärkeren, aber aus Sicht der Topologie des Netzwerks weiter entfernten Geräten vorgezogen werden. Im Gegensatz dazu kann es aber innerhalb einer IoT-Umgebung natürlich auch Datenoperationen geben, wie beispielsweise aufwändige Anfragen auf Datenströmen, die eine eher leistungsstarke Hardwareinfrastruktur erfordern. Für diese müsste dann ein Gerät gefunden werden, das den Ansprüchen der Operation genügt, sich aber trotzdem noch möglichst nahe an der Datenquelle befindet. Es ist somit erstrebenswert, abhängig von der Komplexität und den Leistungsanforderungen der auszuführenden Operationen dafür diejenigen IoT-Geräte zu verwenden, die den Voraussetzungen entsprechen und sich bereits möglichst nahe an den Datenquellen befinden. Auf diese Weise lässt sich die bereits bestehende Netzwerkstruktur der IoT-Umgebung zur Verteilung der Geschäftslogik ausnutzen, sodass möglichst kurze Kommunikationswege entstehen und damit neben dem Erreichen der Vorteile einer dezentralen Ausführung insbesondere die Netzwerkauslastung reduziert wird.

An dem in Abbildung 1.2 dargestellten Beispiel wird dies deutlich. Es ist zweimal dieselbe IoT-Umgebung abgebildet, die aus zwei Sensoren, einem Arduinio-Microcontroller, zwei Raspberry Pis, einem Server und einer Backend-Cloud besteht. In der linken Umgebung wird die Auswertung der Sensordaten zentral in der Cloud vorgenommen; dafür müssen die Daten beider Sensoren zuerst über die Raspberry Pis und den Server zu der Cloud transportiert werden. In der Folge wird eine nicht unerhebliche Netzwerkauslastung und Latenz zwischen allen beteiligten Geräten verursacht, besonders dann, wenn die Frequenz, mit der die Sensordaten erfasst werden, hoch ist. In der rechten Umgebung ist das anders: Dort wird die Auswertung nicht von einer zentralen Instanz, sondern verteilt auf den beiden Raspberry Pis durchgeführt, unter der Prämisse, dass die Infrastrukturen der Raspberry Pis den Leistungsanforderungen der Operationen, die auf den Daten ausgeführt werden sollen, genügen. Die Daten müssen nun nicht mehr erst in die Cloud transportiert werden, sondern nur noch von Sensor 1 zu dem Raspberry Pi 1 und von Sensor 2 über den Arduino zu dem Raspberry Pi 2. Es wird angenommen, dass die Infrastruktur des Arduino-Microcontrollers nicht ausreichend ist, um auf diesem selbst auch Operationen auf den Sensordaten auszuführen;

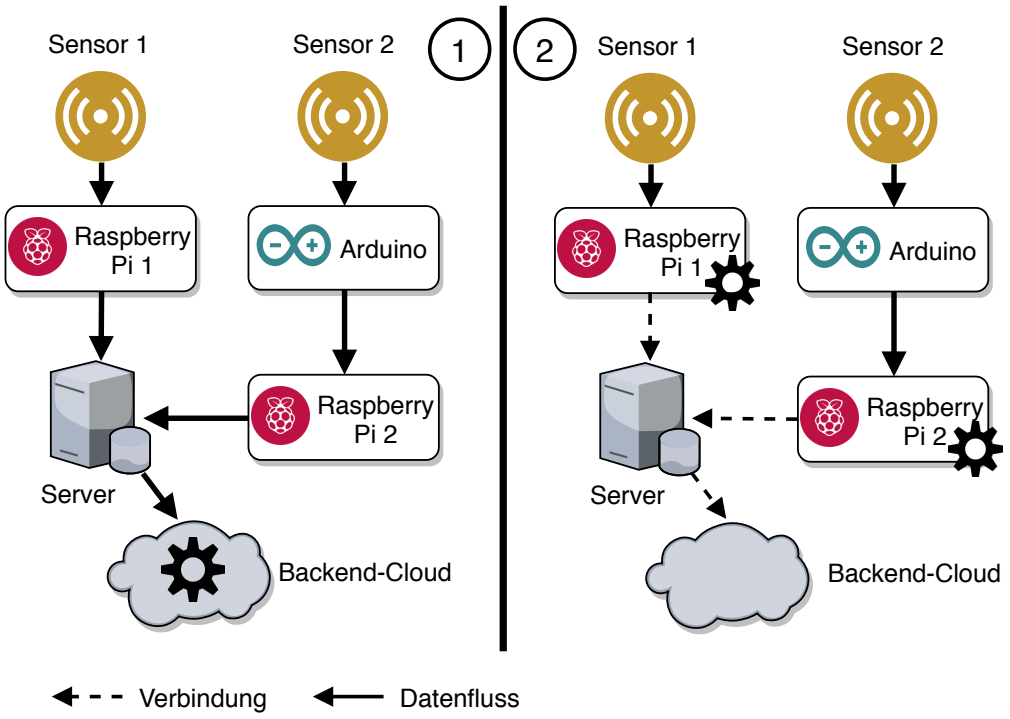

Abbildung 1.2: Beispielhafte Gegenüberstellung einer zentralen (1) und einer verteilten (2) Ausführung

andernfalls könnten die Daten auch schon auf diesem verarbeitet werden. Der Vorteil der rechts dargestellten Umgebung ist offensichtlich: Die Transportwege für die Daten sind wesentlich kürzer, wodurch die Netzwerkauslastung und die Latenz verringert werden. Ferner wird die Ausfallsicherheit erhöht: Sollte ein Teilsystem ausfallen, wie beispielsweise die Backend-Cloud, der Server oder eines der beiden Raspberry-Pis, kann das System der rechts abgebildeten Umgebung eingeschränkt weiterbetrieben werden, während das auf der linken Seite dargestellte System vollständig ausfallen würde.

Diese Bachelorarbeit beschäftigt sich mit der Frage, wie die Geschäftslogik eines Anwendungssystems gemäß den zuvor erläuterten Empfehlungen für eine konkrete IoT-Umgebung automatisiert auf IoT-Geräte verteilt und ausgeführt werden kann.

1.1 Aufbau und Kapitelübersicht

Dieses Dokument gliedert sich in drei Hauptabschnitte, welche die Meilensteine Einarbeitung, Entwurf und Umsetzung der Konzepte repräsentieren. In den Einführungs- und Grundlagenkapiteln wird zunächst ein Überblick über das IoT vermittelt und die Grundlagen von Datenstrommodellen erläutert, anschließend werden die Problemstellung und die damit verbundenen Herausforderungen skizziert. In Kapitel 4 folgt die Abhandlung des konzeptionellen Entwurfs. Dabei wird zu Beginn ein Gesamtüberblick über die entwickelten Lösungen geboten, welche daraufhin im Detail beschrieben werden. Die Beschreibung gliedert sich thematisch in zwei Abschnitte: Im ersten Abschnitt werden zwei Algorithmen zur Verteilung der Geschäftslogik erarbeitet und die Vorbereitungen erläutert,

die dafür getroffen werden mussten. Im zweiten Abschnitt wird das Konzept zur Ausführung der Datenstrommodelle in der IoT-Umgebung vorgestellt. Kapitel 5 beschäftigt sich im Anschluss mit der praxisnahen Umsetzung und Implementierung der Konzepte für eine konkrete IoT-Umgebung. Im darauffolgenden sechsten Kapitel wird auf verwandte Arbeiten verwiesen, die sich mit ähnlichen Problemstellungen auseinandersetzen. Abschließend wird in Kapitel 7 ein Fazit über die Arbeit gezogen, sowie ein Ausblick über mögliche zukünftige Forschung in diesem Themenbereich gegeben.

Die im Rahmen dieser Bachelorarbeit verwendeten Grafiken wurden mit draw.io[3] erstellt.

[3] https://www.draw.io

2 Grundlagen

In diesem Kapitel werden die Grundlagen behandelt, die für die Nachvollziehbarkeit dieser Bachelorarbeit erforderlich sind. Sie umfassen Hintergründe und Detailinformationen zum Internet der Dinge und Datenstrommodellen. Weiterführende Informationen können bei Bedarf den jeweils angegebenen Quellen entnommen werden.

2.1 Internet der Dinge

Bei dem Internet der Dinge (IoT) handelt es sich um einen Sammelbegriff für verschiedene Konzepte, Visionen und Technologien, welcher je nach vorliegendem Anwendungskontext und Domäne unterschiedlich definiert und präzisiert wird. So kann beispielsweise das Wort „Internet“ abhängig von der Anwendungssituation wortwörtlich interpretiert und ausschließlich auf Kommunikationstechniken bezogen werden, die auf dem IP-Protokoll aufsetzen, oder aber als Platzhalter für jegliche Art des technischen Informationsaustausches zwischen Maschinen verstanden werden [FC10].

Gemäß der Internationalen Fernmeldeunion bezeichnet das IoT aus einem allgemeinen Blickwinkel eine globale Infrastruktur von Informationsgesellschaften, die es erlaubt, physische und virtuelle Gegenstände unter Verwendung bestehender und sich entwickelnder Informations- und Kommunikationstechnologien miteinander zu vernetzen und interagieren zu lassen [ITU12].

2.1.1 IoT-Geräte

Das IoT prägt die Vision einer zunehmend vernetzten Welt, in der Gegenstände des täglichen Lebens mit Elektronik und Netzwerkschnittstellen ausgerüstet, in ein Netzwerk integriert und damit zu IoT-Geräten werden können. Sie bilden auf Komponentenebene die Grundlage des IoT und ergänzen damit die bereits bestehende Infrastruktur des Internets. Nach Miorandi et al. [MSPC12] müssen IoT-Geräte unter anderem die folgenden Eigenschaften besitzen:

- Eine physische Beschaffenheit und damit physikalische Eigenschaften
- Grundlegende Möglichkeiten zur Kommunikation innerhalb eines Netzwerks
- Einen eindeutigen Bezeichner, der die Unterscheidung von anderen IoT-Geräten ermöglicht
- Ein aussagekräftiger Name und eine Adresse, die dazu verwendet werden kann, um mit dem jeweiligen IoT-Gerät zu kommunizieren
- Grundlegende Rechenkapazitäten, mit denen Daten verarbeitet werden können

Daneben können IoT-Geräte Sensoren oder Aktuatoren besitzen, um physikalische Daten ihrer Umgebung zu erfassen, beziehungsweise um physisch, mittels Aktuatoren, auf ihre Umgebung einzuwirken. Aus den oben aufgeführten Eigenschaften folgt insbesondere, dass es sich bei IoT-Geräten immer zwingend um real existierende Gegenstände und Objekte handeln muss: Sie können zwar ein digitales Abbild besitzen, ein solches alleine stellt allerdings noch kein eigenständiges IoT-Gerät dar.

Durch den forschungsbedingt weiter sinkenden Energie- und Platzbedarf und die steigende Leistungsfähigkeit elektronischer Bauteile wird es möglich, immer mehr Gegenstände des Alltags als IoT-Geräte zu konstruieren, sodass die Anzahl und Diversität internetfähiger Geräte stetig wächst. Doch nicht nur auf Hardwareebene unterscheiden sich die IoT-Geräte, sondern auch hinsichtlich ihrer Rollen und der Anbindung an das Netzwerk. In Anlehnung an Bonomi et al. [BMZA12] lassen sie sich gemäß ihrer Stellung in der jeweiligen IoT-Umgebung in drei Kategorien unterteilen:

- Edge-Geräte: Edge-Geräte befinden sich aus Sicht der Netzwerktopologie in unmittelbarer Nähe zu den Datenquellen oder -senken des betrachteten Systems und stellen ihrer IoT-Umgebung Daten zur Verfügung oder konsumieren Daten anderer Geräte. Es handelt sich dabei typischerweise um eher leistungsschwache Geräte, die nur mit den nötigsten Kapazitäten ausgestattet sind, um ihre jeweiligen Aufgaben ausführen können.
- Fog-Geräte: Fog-Geräte sind aus Perspektive der Netzwerktopologie weiter als die Edge-Geräte von den Datenquellen und -senken entfernt, befinden sich jedoch meist noch in demselben Intranet. Sie können Daten von den Edge-Geräten beziehen und verarbeiten und damit innerhalb der IoT-Umgebung Dienste nutzen oder anbieten.
- Cloud-Geräte: Cloud-Geräte sind meist leistungsstarke Backend-Server, die typischerweise von Drittanbietern zur Verfügung gestellt werden und sich nicht in demselben Intranet befinden wie die anderen Geräte der IoT-Umgebung. Sie werden zur Durchführung von besonders leistungsintensiven Operationen eingesetzt, der Datentransport zu und von den Cloud-Geräten ist jedoch aufgrund der größeren Netzwerkdistanz in der Regel mit einer höheren Latenz verbunden als zwischen den Edge- oder Fog-Geräten.

Nicht zuletzt durch das IoT wandelt sich die Rolle des Internets: Es ist nicht länger eine reine Netzwerkinfrastruktur, die die Kommunikation zwischen verschiedenen Endbenutzergeräten ermöglicht, sondern die Grundlage für weltweiten Informationsaustausch zwischen virtuellen und realen Objekten, die die Fähigkeit zur Datenverarbeitung und Kommunikation besitzen [MSPC12]. Der Mensch tritt dadurch als Hauptnutzer des Internets in den Hintergrund und überlässt das Feld bisweilen autonom agierenden Geräten, von deren Vernetzung und Zusammenarbeit er dann letztlich profitiert. In Anbetracht dieser Entwicklung wird der Begriff des IoT weitgehend dafür verwendet, um sich auf die folgenden drei Sachverhalte zu beziehen [MSPC12]:

1. Das globale Netzwerk, das auf erweiterten Internettechnologien aufbaut und aus der Vernetzung von IoT-Geräten entsteht
2. Die Menge der unterstützenden Technologien, die zur Realisierung dieser Vision erforderlich sind
3. Die Gesamtheit von Anwendungen und Diensten, die diese Technologien nutzen, um neue wirtschaftliche Geschäftsfelder zu öffnen

Eine Umgebung, die ein oder mehrere IoT-Geräte enthält, wird als *IoT-Umgebung* bezeichnet. Ein Beispiel dafür sind die sogenannten „Smart Homes“ in denen Alltagsgegenstände von Gebäuden wie Hausbeleuchtungen, Heizungen oder Fenster miteinander vernetzt und auf logische Weise verknüpft werden [VF13].

2.1.2 Eigenschaften von IoT-Umgebungen

Durch die Fähigkeiten zur Erfassung der Umgebung, zur Verarbeitung von Daten und zur Kommunikation innerhalb eines Netzwerks können IoT-Geräte potentiell auf vielfältige im Internet verfügbare Dienste zurückgreifen und selbst Teil solcher Dienste werden, denen sie ihre gesammelten Daten und Funktionen zur Verfügung stellen. Auf diese Weise lassen sich vielfältige Anwendungsfälle realisieren, bei denen Entscheidungen auf Basis der Zustände und gesammelten Daten mehrerer IoT-Geräte getroffen und anschließend auch von diesen ausgeführt werden können. Die realen Gegenstände erhalten dadurch eine neue Qualität, indem ihre physischen Funktionen um die flexiblen Fähigkeiten digitaler Objekte erweitert werden [FC10].

Von einem technischen Standpunkt aus setzt sich das IoT damit aus verschiedenen, gegenseitig ergänzenden Technikentwicklungen zusammen, die gemeinsam in eine Menge von nutzbaren Funktionen resultieren [FT08]. Gegebene Anwendungsfälle lassen sich dann unter einer selektiven oder gesamtheitlichen Nutzung dieser Funktionen umsetzen. Nach Roth [Rot16] gehören zu diesen Funktionen unter anderem:

- Kommunikation und Kooperation: Die Geräte im IoT sind in der Lage, sich untereinander oder mit anderen Ressourcen zu vernetzen, um gegenseitig auf Dienste und Daten zuzugreifen.
- Adressierbarkeit: IoT-Geräte können über Namensdienste eindeutig aufgefunden und angesprochen werden.
- Sensorik und Effektorik: IoT-Geräte können ihre Umwelt erfassen und gegebenenfalls auf sie einwirken.
- Lokalisierung: IoT-Geräte kennen ihren physischen Aufenthaltsort oder können auf andere Weise in der realen Welt lokalisiert werden.
- Benutzungsschnittstelle: Menschen haben die Möglichkeit, mit den IoT-Geräten zu kommunizieren und zu interagieren.

In der Regel ist für einen konkreten Anwendungsfall allerdings nur eine Untermenge der aufgeführten Funktionen erforderlich. Beispielsweise kann man eine Heizung und einen räumlich an einer anderen Stelle platzierten Temperatursensor derart als IoT-Geräte konstruieren, dass sie jeweils mit dem Internet verbunden werden und darüber miteinander Daten austauschen können. Die Heizung kann auf diese Weise regelmäßig Temperaturwerte des Sensors abfragen und ausgehend davon die Heizleistung regulieren, dass die Raumtemperatur einen zuvor von dem Benutzer über eine Webanwendung festgelegten Schwellenwert nicht unterschreitet. Eine solche Anwendung macht von einigen der oben genannten Funktionen Gebrauch: Zwischen dem Temperatursensor und der Heizung würde in Form des Austauschs der gemessenen Temperaturwerte Kommunikation stattfinden. Weiter muss der Temperatursensor adressierbar sein, damit die Heizung über das Internet eine Verbindung zu diesem aufbauen und die gemessenen Werte beziehen kann. Der Temperatursensor misst physikalische Werte und stellt damit eine Sensorikfunktion zur Verfügung,

während die Heizung durch das Regulieren der Heizleistung auf ihre Umwelt einwirkt und folglich eine Aktuatorikfunktion anbietet. Zuletzt ist es dem Benutzer in diesem Beispiel möglich, den Schwellenwert für die Raumtemperatur über eine Webanwendung bei der Heizung einzustellen, was eine Benutzungsschnittstelle darstellt. Eine Funktion zur Lokalisierung der IoT-Geräte wird in diesem Beispiel dagegen nicht benötigt.

IoT-Geräte besitzen in der Regel die Möglichkeit, sich flexibel in das jeweilige Netzwerk ein- und auszuklinken; die Verfügbarkeit eines bestimmten IoT-Geräts kann also zu keinem Zeitpunkt garantiert werden. Auf Systemebene kann das IoT deshalb und aufgrund der zuvor aufgeführten Eigenschaften als hochdynamisches und stark verteiltes vernetztes System aufgefasst werden, das aus einer großen Anzahl von IoT-Geräten besteht, die Informationen erzeugen und konsumieren [MSPC12]. Diese Unwägbarkeiten hinsichtlich der Verfügbarkeit von Geräten, Ressourcen und Diensten, die während der Ausführung bestehen, erschweren den Entwurf und die Skalierung von IoT-Ökosystemen: Neue Funktionen und Dienste müssen zur Laufzeit in das Netzwerk eingebracht und verfügbar gemacht werden können, ohne dass sie zuvor zwingend beim Entwurf des Systems berücksichtigt worden sind. Die Systeme und Anwendungen müssen daher besonders flexibel und adaptiv entworfen werden, sodass sie sich weitestgehend autonom verhalten, unmittelbar an neue Gegebenheiten anpassen und selbst organisieren können [MSPC12].

Üblicherweise gehört es auch zu den Zielen einer IoT-Umgebung, möglichst schnell auf Veränderungen in der Umgebung reagieren zu können. Schlägt beispielsweise ein in Form mehrerer IoT-Geräte konzipiertes Feuermeldesystem Alarm, so sollte diese Gefahrensituation von dem jeweiligen IoT-System frühzeitig als solche erkannt werden, damit zeitnah darauf reagiert werden kann und gegebenenfalls entsprechende Gegenmaßnahmen, wie beispielsweise die Aktivierung einer Löschanlage, eingeleitet werden können.

Damit diese Ziele erreicht werden können, müssen Anwendungen in IoT-Ökosystemen eigenständig und in Echtzeit in der Lage sein, die Situation und den Kontext zu erkennen, in dem sich der Benutzer und das System selbst gerade befinden [MSPC12].

2.2 Datenstrommodelle

In Anwendungssystemen und insbesondere auch bei IoT-Systemen werden als Teil der Geschäftslogik im Allgemeinen Daten verarbeitet, um die an das jeweilige System gestellten funktionalen Anforderungen zu erfüllen. Aus einer konzeptionellen Perspektive werden dafür auf einer Menge von Eingabedaten Aktivitäten in einer festgelegten Reihenfolge ausgeführt, die bei vorliegender Korrektheit der Anwendung letztlich in den gewünschten Ausgabedaten resultieren. Die Eingabedaten liegen dabei in Form von Datenströmen vor. Diese Funktionalität eines Systems lässt sich formal durch ein Datenstrommodell beschreiben, welches Aktivitäten eines Systems und die Datenströme zwischen diesen Aktivitäten abbildet.

2.2.1 Datenströme

Die von der Sensorik gemessenen Daten werden den jeweiligen IoT-Geräten in der Regel in Form von kontinuierlichen, potentiell beliebig großen Datenströmen zur Verfügung gestellt. Die einzelnen innerhalb dieser Ströme transportierten Daten können dabei von einem beliebigen Format sein,

auch in Abhängigkeit davon, auf welcher Schicht des OSI-Modells sie betrachten werden. In jedem Fall erschwert dabei das fortwährende, zeitlich variable und möglicherweise unvorhersehbare Eintreffen der Daten deren Verarbeitung [BBD+02]. So ist es zum Beispiel vorstellbar, dass ein Temperatursensor periodisch mit einer zuvor festgelegten Frequenz die Temperatur seiner Umgebung misst und an das IoT-Gerät, an das er angeschlossen ist, weiterleitet. Diese Messwerte resultieren dann aus Sicht des entgegennehmenden Geräts in einem Datenstrom. In Abhängigkeit von gegebenenfalls vorhandenen äußeren Einflüssen und etwaigen technischen Beschränkungen des Sensors, kann die Zeit, bis jeweils ein neuer Messwert am IoT-Gerät eintrifft, dann durchaus variieren und ist nur bedingt vorhersehbar. Das führt dazu, dass Datenströme grundsätzlich auf andere Weise verarbeitet werden müssen als herkömmliche Datenbestände: Die ankommenden Daten können nicht einfach allesamt in ein Datenbankmanagementsystem (DBMS) gespeichert und dann mit Hilfe von diesem verarbeitet werden, denn die traditionellen DBMS sind kapazitiv nicht dafür entworfen, kontinuierlich und schnell Daten aufzunehmen und auf diesen zu operieren [BBD+02]. Gerade im Bereich des IoT sind Effizienz und Reaktionsschnelligkeit jedoch von essentieller Bedeutung (vgl. 2.1.2). Dementsprechend müssen andere Ansätze zur Verarbeitung von Datenströmen gewählt werden.

Gemäß dem Konzept zur Beschreibung von Datenströmen nach Babcock et al. [BBD+02] kann auf alle oder einige Eingabedaten innerhalb des Datenstroms, der von potentiell unbegrenzter Größe sein kann, nicht wahlfrei zugegriffen werden. Weiter hat das System, das den Strom verarbeitet, weder innerhalb des Datenstroms noch über mehrere Datenströme hinweg Kontrolle über die Reihenfolge, in der die einzelnen Daten ankommen. Stattdessen müssen die Daten bei der Verarbeitung des Stroms einzeln und in der Reihenfolge, in der sie angekommen sind, betrachtet werden. Wurde ein Element des Datenstroms durch das Gerät verarbeitet, wird es verworfen und steht danach nicht weiter zum Zugriff zur Verfügung, sofern es nicht explizit im Speicher hinterlegt worden ist. Für das Beispiel des Temperatursensors folgt daraus, dass die am IoT-Gerät ankommenden Temperaturwerte schon direkt bei der Ankunft betrachtet und analysiert werden müssen, da eine Erstsichtung zu einem späteren Zeitpunkt nicht mehr möglich ist. Selbst wenn die eigentliche Verarbeitung und Interpretation erst später erfolgen sollte, muss beim Eingang eines Werts zumindest schon entschieden werden, ob er für die weitere Betrachtung berücksichtigt und dafür explizit gespeichert werden soll.

Die Verarbeitung des Datenstroms gemäß dieser Paradigmen schließt jedoch nicht aus, dass bestimmte Daten des Datenstroms auch zusätzlich auf herkömmliche Weise in relationalen DBMS gespeichert und dann bei den Verarbeitungsprozessen der Datenströme mitberücksichtigt werden können [BBD+02].

2.2.2 Abfragen auf Datenströmen

Es ist wünschenswert, analog zu Datenbankabfragen („Queries“) in relationalen DBMS auch Abfragen auf Datenströmen ausführen zu können. Auf diese Weise können im Beispiel des Temperatursensors leicht alle Werte über einem gewissen Schwellenwert herausgefiltert, Mittelwerte berechnet und andere Operationen durchgeführt werden, die dann bei der Erfassung des vorliegenden Zustands der Umgebung hilfreich sind. Dabei müssen jedoch die Unterschiede beachtet werden, die auf konzeptioneller Ebene zwischen beiden Systemen bestehen.

Grundsätzlich lässt sich zwischen Einmal-Abfragen und kontinuierlichen Abfragen unterscheiden: Einmal-Abfragen, zu denen auch die Abfragen in relationalen DBMS gehören, werden nur einmal zu einem bestimmten Zeitpunkt über eine Momentaufnahme („Schnappschuss“) des Datenbestandes ausgeführt und das Ergebnis an die ausführende Instanz zurückgegeben. Kontinuierliche Abfragen müssen dagegen regelmäßig ausgeführt werden, da im Rahmen des Datenstroms fortlaufend neue Daten eintreffen [BBD+02]. Das Ergebnis einer solchen Abfrage wird über die Laufzeit hinweg erstellt und ist damit nur für die Daten gültig, die bis zum Zeitpunkt der Abfrage bereits im Datenstrom übermittelt worden sind [TGNO92]. Es kann dann entweder gespeichert und bei Ankunft neuer Daten jeweils aktualisiert werden, oder aber selbst in Form eines Datenstroms zurückgegeben werden.

Daneben wird zusätzlich zwischen vordefinierten Abfragen und Ad-Hoc-Abfragen unterschieden. Eine vordefinierte Abfrage ist zumeist eine kontinuierliche Abfrage und wird dem DBMS bereits zur Verfügung gestellt, bevor relevante Daten eingetroffen sind. Ad-Hoc-Abfragen dagegen können entweder Einmal-Abfragen oder kontinuierliche Abfragen sein und werden erst nach dem Start der Datenströme auf dem DBMS erstellt. Sie erschweren den Entwurf eines Systems zur Verarbeitung von Datenströmen, weil die Verarbeitung der Datenströme nicht im Voraus hinsichtlich der Abfragen eingestellt und optimiert werden kann.

Eine Möglichkeit zur Durchführung von Abfragen auf Datenströmen stellt Complex Event Processing dar [CM12].

2.2.3 Aufbau von Datenstrommodellen

Das Architekturmuster Pipes und Filter [Meu95] hat sich für den Aufbau modularer, stark erweiterbarer Anwendungen bewährt und ist deshalb auch als Grundlage zur Erstellung von Datenstrommodellen geeignet [HB17]. Auch die im weiteren Verlauf dieser Bachelorarbeit betrachteten Datenstrommodelle bauen auf Pipes und Filter auf und bestehen allgemein aus vier Basiselementen:

- Operatoren
- Datenquellen
- Datensenken
- Datenströme

In Abbildung 2.1 ist ein Beispiel eines Datenstrommodells dargestellt, das aus zwei solcher Datenquellen, drei Operatoren und zwei Datensenken besteht.

Die Operatoren entsprechen den Filtern und repräsentieren Softwareelemente, die auf Daten Aktivitäten durchführen, wie Rechenoperationen oder Datenfilterungen. Operatoren besitzen immer mindestens einen eingehenden oder einen ausgehenden Datenstrom. Operatoren sind typischerweise Dienste, die einheitliche Schnittstellen für die ein- und ausgehenden Datenströme anbieten [HB17]. Die Datenquellen entsprechen der Eingabe und repräsentieren Softwareelemente, die Daten in das System einspeisen. Typischerweise handelt es sich dabei um Software, die zum Betrieb von Sensoren erforderlich ist, wie beispielsweise Treiber. Allerdings müssen die Daten nicht zwingend Sensoren entstammen, es kommen auch andere Quellen wie beispielsweise Datenbanken in Betracht.

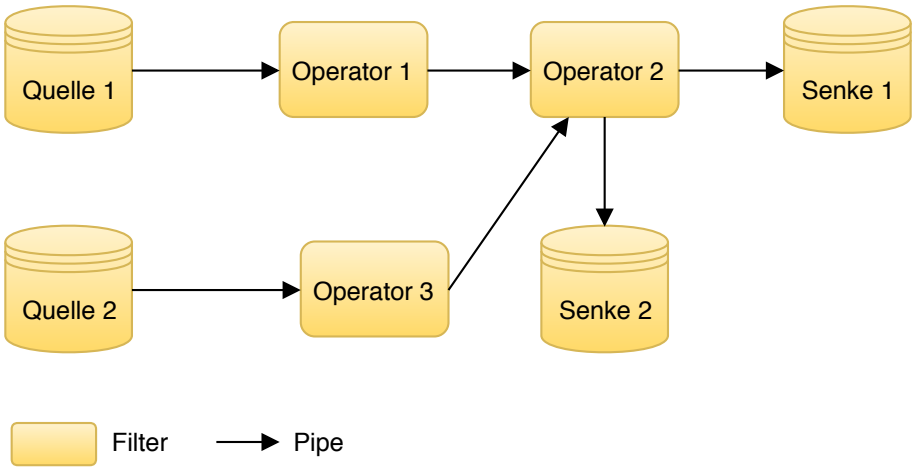

Abbildung 2.1: Beispiel für das auf Pipes und Filter basierende Datenstrommodell

Datenquellen verhalten sich im Datenstrommodell prinzipiell wie Operatoren, verfügen aber über keine eingehenden Datenströme. Datensenken entsprechen der Ausgabe und repräsentieren Softwareelemente, die die vollständig verarbeiteten Daten entgegennehmen. Auch sie verhalten sich wie Operatoren, verfügen aber über keine ausgehenden Datenströme. Die Datenströme selbst, auch Verbindungen genannt, entsprechen den Pipes und repräsentieren die Weiterleitung der Daten von einer Datenquelle oder einem Operator zu einem anderen Operator. Im Datenstrommodell verbinden sie daher entsprechend die beiden betreffenden Elemente miteinander. Datenströme sind gerichtet, die modellierten Verbindungen sind also unidirektional. Datenquellen und Operatoren dürfen nicht über Datenströme mit sich selbst verbunden werden und auch Zyklen sind innerhalb des gesamten Datenstrommodells nicht zulässig. Im Folgenden werden auch Datenquellen und Datensenken unter dem Oberbegriff „Operator“ zusammengefasst.

3 Problembeschreibung

Nachdem die Thematik, die dieser Bachelorarbeit zugrunde liegt, in den vorhergehenden Kapiteln grob umrissen worden ist, wird die Aufgabenstellung in diesem Abschnitt weiter präzisiert. Im Anschluss wird auf die Herausforderungen dieser Arbeit eingegangen.

3.1 Aufgabenstellung

FlexMash ist ein an der Universität Stuttgart entwickeltes Werkzeug zur Modellierung von Datenstrommodellen. Damit ist es möglich, Datenströme zunächst grafisch zu modellieren und anschließend auszuführen [HB17]. In Abbildung 3.1 ist ein Teil der Benutzeroberfläche von FlexMash abgebildet. Mittels Drag and Drop können Datenquellen, Operatoren und Datensenken modelliert und miteinander verbunden werden.

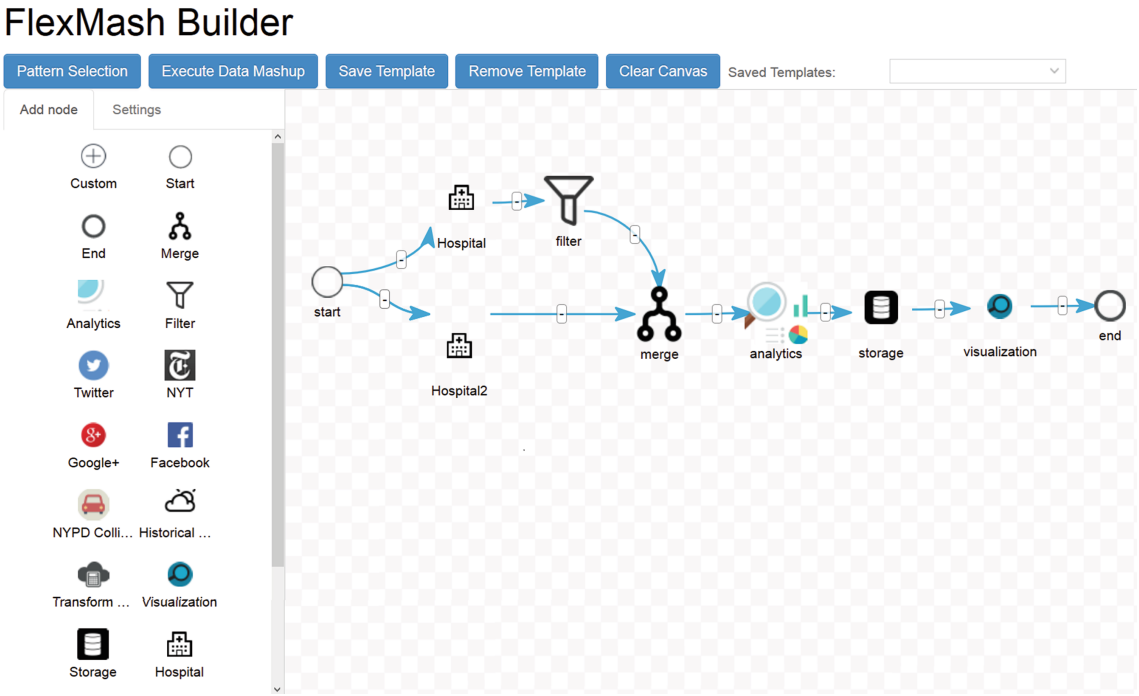

Abbildung 3.1: Screenshot der Benutzeroberfläche von FlexMash

Die Ausführung des auf diese Weise erstellten Datenstrommodells erfolgt in der aktuellen Version von FlexMash in einer einzelnen Laufzeitumgebung, die sich typischerweise in einer Cloud befindet. Ein grundlegendes Ziel des IoT und vieler darauf aufbauender Anwendungssysteme ist es jedoch, adaptive und flexible Umgebungen zu schaffen, die in der Lage sind, so schnell wie möglich auf sich

verändernde Bedingungen in ihrer Umgebung zu reagieren. Wie bereits in Kapitel 1 beschrieben, ist dies mit dem aktuell in FlexMash realisierten Ansatz nicht vereinbar, da bei diesem alle in der IoT-Umgebung produzierten Daten notwendigerweise an die Laufzeitumgebung übermittelt werden müssen, was tendenziell zu starker Netzwerkauslastung und zu hohen Latenzzeiten führt.

Die Hauptaufgabe dieser Bachelorarbeit ist es daher, eine verteilte Ausführung des Datenstrommodells für FlexMash zu realisieren, die anstelle der zentralen Ausführung verwendet werden kann. Das Werkzeug soll damit zukünftig in der Lage sein, vor der Ausführung dynamisch zu entscheiden, auf welchen in der IoT-Umgebung verfügbaren IoT-Geräten die Operationen des Datenstrommodells ausgeführt werden sollen. Operationen, die eher weniger Ressourcen für ihre Berechnungen benötigen, sollen dabei vorzugsweise in der Nähe der Datenquelle ausgeführt werden, idealerweise noch auf dem IoT-Gerät selbst, in das die Quelle, wie beispielsweise der entsprechende Sensor, integriert ist. Operationen, die mehr Ressourcen erfordern, dürfen stattdessen auch in einer aus Perspektive einer Netzwerktopologie weiter entfernten, leistungsfähigeren Umgebung ausgeführt werden, wie auf einem dafür geeigneten Server oder in der Cloud.

Für die Umsetzung der im Rahmen dieser Arbeit entworfenen Konzepte und die Integration dieser in FlexMash steht bereits eine konkrete IoT-Umgebung zu Verfügung. Teil dieser Umgebung ist auch die sogenannte Multi-purpose Binding and Provisioning Platform (MBP)[1]. Sie ermöglicht es erstens, die in der Umgebung befindlichen IoT-Geräte zu binden, um auf ihre Sensoren und Aktuatoren zugreifen zu können und zweitens, Softwareelemente zu provisionieren, die den Operatoren eines Datenstrommodells entsprechen [FHM18]. Unter Bereitstellung passender Shell-Skripte können die mittels der MBP ausgelieferten Operatoren automatisch installiert, gestartet, gestoppt und deinstalliert werden. Dies ermöglicht eine flexible Umverteilung von Operatoren auf andere IoT-Geräte.

3.2 Zielsetzung

Daraus ergeben sich für diese Bachelorarbeit insgesamt drei Ziele mit jeweils mehreren Teilzielen, die im Folgenden detailliert vorgestellt werden. Die Aufgaben, die sich aus diesen Zielen ableiten lassen, können getrennt, aber nicht unabhängig voneinander bearbeitet werden. In den Kapiteln 4 und5, in denen die konzeptionelle Lösung und Umsetzung vorgestellt wird, wird auf diese Ziele erneut eingegangen und jeweils beschrieben, ob und auf welche Weise sie erreicht worden sind.

Ziel 1: Konzeption von Verteilungsalgorithmen

Das erste Ziel dieser Arbeit besteht in der Konzeption von Algorithmen, die in der Lage sind, in FlexMash modellierte Datenstrommodelle auf eine für die Ausführung möglichst geeignete Weise auf die IoT-Geräte der IoT-Umgebung abzubilden. Dabei sollen auch die Anforderungen, die die Operatoren an die Infrastruktur der IoT-Geräte stellen und die korrespondierenden Eigenschaften, die die IoT-Geräte aufweisen, berücksichtigt werden. Diese Problemstellung wird im Folgenden als *Verteilungsproblem* bezeichnet. Damit dieses Problem gelöst werden kann, müssen zunächst einige Vorkehrungen getroffen und Voraussetzungen geschaffen werden. Insbesondere müssen das Ziel der Algorithmen herausgearbeitet und Ein- und Ausgaben spezifiziert werden.

[1] Open-Source-Projekt, verfügbar unter: https://github.com/IPVS-AS/MBP

Teilziel 1.1: Beschreibung des Verteilungsproblems

Bevor die Algorithmen selbst konzipiert werden können, muss das zugrundeliegende Problem, für das die Algorithmen eine Lösung berechnen sollen, im Detail beschrieben werden. Dabei muss auch festgestellt werden, was die Ausgabe der Algorithmen, die Lösung für eine Instanz des Verteilungsproblems, beinhalten muss.

Teilziel 1.2: Formalisierung der Ein- und Ausgabe der Algorithmen

Die Eingaben der Algorithmen sollen formalisiert werden, damit ihre strukturellen Eigenschaften auf formaler Ebene offengelegt werden und von den Algorithmen ausgenutzt werden können. Auch die Ausgabe der Algorithmen, die Lösung für eine konkrete Instanz des Verteilungsproblems, soll formalisiert werden, damit sichergestellt werden kann, dass der Algorithmus nur wohldefinierte Lösungen ausgibt, die von gegebenenfalls nachfolgenden Prozeduren verarbeitet werden können.

Teilziel 1.3: Definition einer gültigen Lösung

Sobald gemäß Teilziel 1.2 die Form der Lösungen der Algorithmen feststeht, soll definiert werden, wann eine solche Lösung als gültig zu erachten ist und die gegebene Instanz des Verteilungsproblems korrekt löst. Eine gültige Lösung ist die Voraussetzung dafür, dass sie als Grundlage für die Ausführung des Datenstrommodells auf IoT-Geräten eingesetzt werden kann.

Teilziel 1.4: Definition der bestmöglichen Lösung

Es ist denkbar, dass es für ein konkretes gegebenes Verteilungsproblem, auf das die Algorithmen angesetzt werden, mehrere unterschiedliche gültige Lösungen gibt. Damit aus diesen, falls vom Anwender gewünscht, die aus seiner Sicht beste Lösung bestimmt werden kann, muss zunächst definiert werden, durch welche Eigenschaften sich eine bestmögliche Lösung von anderen gültigen Lösungen unterscheidet.

Teilziel 1.5: Vergleich zwischen Anforderungen und Eigenschaften

Ein wichtiger Bestandteil der Algorithmen zur Lösung von Verteilungsproblemen ist ein Teilalgorithmus, der in der Lage ist, die an die Infrastruktur einer IoT-Umgebung gestellten Anforderungen mit den tatsächlich in der Praxis vorhandenen Eigenschaften abzugleichen und zu entscheiden, ob ein Operator mit einem Gerät provisioniert werden kann. Dieser Teilalgorithmus kann dann eingesetzt werden, um geeignete Geräte für die Operatoren zu finden.

Teilziel 1.6: Finden einer gültigen Lösung

Sobald gemäß Teilziel 1.3 festgelegt ist, wie eine gültige Lösung aussieht, sollen die Algorithmen so implementiert werden, dass sie für eine gegebene Instanz des Verteilungsproblems eine gültige Lösung ausgeben, wenn eine solche existiert. Dabei soll der Vergleichsalgorithmus aus Teilziel 1.5 verwendet werden.

Teilziel 1.7: Erkennung, dass keine Lösung existiert

Es sind auch Problemstellungen denkbar, in denen keine Lösung existiert, beispielsweise, wenn kein in der IoT-Umgebung verfügbares Gerät die Anforderungen eines bestimmten Operators erfüllt. Wenn keine Lösung existiert, sollen die Algorithmen daher auch keine Lösung finden und das entsprechend signalisieren. Es ist allerdings akzeptabel, dass diese Erkennung in Ausnahmefällen falsch-positive Resultate erzeugt, also dass die

Algorithmen keine Lösung finden, obwohl eine solche theoretisch existiert. Im Idealfall signalisieren die Algorithmen jedoch die Nichtexistenz einer Lösung genau dann, wenn auch wirklich keine Lösung existiert.

Teilziel 1.8: Finden der bestmöglichen Lösung

Unter Einbezug der Definition aus Teilziel 1.4 sollen die Algorithmen in der Lage sein, bei mehreren gefundenen Lösungen die jeweils bestmögliche auszuwählen und auszugeben. Idealerweise ist diese Lösung dann nicht nur unter den gefundenen Lösungen die bestmögliche, sondern von allen tatsächlich existierenden Lösungen.

Ziel 2: Entwurf eines Konzepts zur Ausführung des Datenstrommodells

Das zweite Ziel dieser Arbeit besteht in dem Entwurf eines Konzepts zur Ausführung eines mit FlexMash erstellten Datenstrommodells auf den IoT-Geräten der IoT-Umgebung. Dabei soll eine der von den Verteilungsalgorithmen empfohlenen Verteilungen verwendet werden. Es genügt nicht, nur die Operatoren auf die Geräte abzubilden, es muss auch sichergestellt werden, dass die Operatoren in der Lage sind, die Daten gemäß dem Datenstrommodell untereinander auszutauschen. Somit wird ein Entwurf benötigt, der zum einen eine bestimmte Technologie zum Datentransport vorsieht und zum anderen ein System, das den Datentransport zwischen den Operatoren auch über mehrere IoT-Geräte hinweg ermöglicht.

Teilziel 2.1: Evaluation der Rolle der MBP

Zunächst soll die Rolle der MBP untersucht werden, die in der späteren Umsetzung zur Provisionierung der Operatoren mit IoT-Geräten verwendet werden kann. Dabei soll geprüft werden, welche technischen Möglichkeiten sie bereits zur Verfügung stellt und für welche konkreten Problemstellungen sie damit eingesetzt werden kann. Für alle Probleme, für die die MBP noch keine Lösung anbietet, muss im Rahmen dieses Ziels potentiell eine eigene Lösung entwickelt werden.

Teilziel 2.2: Bestimmung einer geeigneten Kommunikationstechnologie

Es soll eine Technologie gefunden werden, die dafür geeignet ist, Daten zwischen den IoT-Geräten auszutauschen.

Teilziel 2.3: Konzeption der Kommunikation zwischen IoT-Geräten

Damit das Datenstrommodell in dem IoT-System ausgeführt werden kann, muss sichergestellt werden, dass die Operatoren unter Verwendung der in Teilziel 2.2 bestimmten Technologie die Daten getreu dem Datenstrommodell austauschen können. Dabei muss berücksichtigt werden, dass nicht alle Geräte der Netzwerktopologie notwendigerweise miteinander verbunden sind. So kann es vorkommen, dass die Geräte, auf die die Operatoren einer Verbindung aus dem Datenstrommodell ausgeliefert werden, nicht direkt miteinander kommunizieren können. Deshalb soll der Datentransport auch über mehrere involvierte IoT-Geräte hinweg möglich sein.

Ziel 3: Umsetzung der Konzepte

Das dritte und letzte Ziel dieser Arbeit besteht in der Implementierung der im Rahmen der Ziele 1 und 2 erarbeiteten Konzepte für die bestehende IoT-Umgebung. Es soll möglich sein, die Eingaben, die die Verteilungsalgorithmen benötigen, einzulesen und die Algorithmen dann auf diesen Daten auszuführen. Die Ausgabe der Algorithmen soll anschließend als Grundlage verwendet werden, um die Operatoren auf den IoT-Geräten zu installieren und auszuführen. Ferner soll während der Provisionierung implizit auch das in Ziel 2 erarbeitete

Kommunikationskonzept in der IoT-Umgebung eingerichtet werden, sodass die Operatoren wie vorgesehen miteinander kommunizieren können und eine vollumfängliche Ausführung des Datenstrommodells möglich ist. Die bereits in der IoT-Umgebung befindliche MBP kann dabei verwendet werden, um die Operatoren entsprechend mit den Geräten zu provisionieren und, falls erforderlich, Umverteilungen vorzunehmen.

Teilziel 3.1: Digitale Repräsentation der Eingabe der Algorithmen
Die Eingaben, die die Verteilungsalgorithmen zur Berechnung einer möglichen Verteilung benötigen, müssen in der Implementierung eine digitale Repräsentation besitzen. So muss beispielsweise das Datenstrommodell intern in Form von Klassen, Objekten und anderen Programmierparadigmen abgebildet werden. Die Eingaben sollen dabei als Dateien eingelesen werden können. Dafür wird eine entsprechende Importfunktion benötigt, die es ermöglicht, die Dateien einzulesen und auf die internen Datenstrukturen abzubilden.

Teilziel 3.2: Digitale Repräsentation der Ausgabe der Algorithmen
Analog zu den Eingaben, soll auch die Ausgabe der Verteilungsalgorithmen eine digitale Repräsentation besitzen, die dann im weiteren Programmablauf zur Provisionierung verwendet werden kann.

Teilziel 3.3: Umsetzung des Vergleichs von Anforderungen und Eigenschaften
Auch die Anforderungen der Operatoren aus dem Datenstrommodell und die Eigenschaften der IoT-Geräte müssen auf entsprechende Datenstrukturen abgebildet werden. Auf Grundlage von diesen soll dann der Vergleichsalgorithmus aus Teilziel 1.5 umgesetzt werden.

Teilziel 3.4: Implementierung der Verteilungsalgorithmen
Unter Verwendung der internen Repräsentationen aus den Teilzielen 3.1 und 3.2, der Algorithmen aus Teilziel 3.3 und des konzeptionellen Entwurfs aus den Teilzielen 1.6, 1.7 und 1.8, sollen die Verteilungsalgorithmen umgesetzt werden. Nach Abschluss der Implementierung sollen sie in der Lage sein, gegebene Verteilungsprobleme bestmöglich zu lösen oder zu erkennen, wenn keine Lösung existiert.

Teilziel 3.5: Umsetzung der automatischen Provisionierung
Die MBP kann zur Provisionierung der Operatoren mit IoT-Geräten eingesetzt werden und bietet dafür eine REST[2]-Schnittstelle an. Um diese zu nutzen, ist es zunächst erforderlich, einen entsprechenden Klienten zu implementieren, der es erlaubt, die MBP anzusteuern und die notwendigen Operationen wie gewünscht ausführen zu lassen. Der Klient soll möglichst modular, in sich abgeschlossen und nur lose mit den anderen Softwarekomponenten gekoppelt sein, damit er wiederverwendet und gegebenenfalls in anderen Projekten eingesetzt werden kann.

Teilziel 3.6: Umsetzung des Kommunikationskonzepts
Neben der Provisionierung nach Teilziel 3.5 muss auch das zuvor in Ziel 2 entworfene Kommunikationskonzept umgesetzt und in der IoT-Umgebung installiert werden, damit den Operatoren nach der Provisionierung alle benötigten Daten anderer Operatoren zur Verfügung stehen und eine korrekte Ausführung gewährleistet ist.

[2]Representational State Transfer [FT00], Programmierparadigma

Teilziel 3.7: Test der Umsetzung

Wurden alle Konzepte entsprechend der vorhergehenden Teilzielen umgesetzt, muss die Implementierung getestet werden, um zu prüfen, ob sie wie vorgesehen funktioniert. Dafür sollen zum einen Modultests [Joc13] mit einer möglichst hohen Zeilenabdeckung erstellt werden, zum anderen sollen aber auch Integrationstests [Joc13] mit fiktiven Beispielszenarien in der gegebenen IoT-Umgebung durchgeführt werden.

3.3 Herausforderungen

Aus den zuvor beschriebenen Zielen und der Aufgabenstellung ergeben sich einige Herausforderungen, die es im Rahmen des Konzeptes zu bewältigen gilt. Auch auf die Herausforderungen wird in Kapitel 4 erneut eingegangen und aufgezeigt, wie sie überwunden werden konnten.

Herausforderung 1: Variabilität des Datenstrommodells und der IoT-Umgebung

Das Datenstrommodell, auf dem die Algorithmen operieren und die Netzwerktopologie der IoT-Umgebung können potentiell sehr unterschiedlich ausgeprägt sein. So sind Konstellationen mit wenigen oder sehr vielen Operatoren und wenigen oder sehr vielen IoT-Geräten in der Umgebung denkbar. Dies macht es schwer, gesamtheitlich über die Problemstellung zu urteilen oder Eigenschaften herauszuarbeiten, die allen denkbaren Situationen gemein sind. In der Folge entstehen viele Sonderfälle, die bei der Konzeption der Algorithmen und der Ausführung beachtet werden müssen.

Herausforderung 2: Sonderfälle bei Anforderungen und Eigenschaften

Ein Abgleich zwischen den Anforderungen der Operatoren aus dem Datenstrommodell und den tatsächlichen Eigenschaften der IoT-Geräte ist ein Teil der Verteilungsalgorithmen. Hier können allerdings auch einige Sonderfälle auftreten, die beachtet werden müssen: Wie sind beispielsweise die Fälle zu behandeln, wenn für ein Gerät eine Eigenschaft definiert ist, für die es im Datenstrommodell keine entsprechende Anforderung gibt oder wenn bei einem Gerät für eine Anforderung im Datenstrommodell keine entsprechende Eigenschaft definiert ist? Für diese Situationen müssen geeignete Regelungen getroffen werden.

Herausforderung 3: Kumulation von Operatoren

In den Verteilungsalgorithmen muss mit dem Fall umgegangen werden, dass bereits verteilte Operatoren die Eigenschaften von IoT-Geräten kumulativ beeinflussen. Das kann grundsätzlich immer auftreten, wenn ein IoT-Gerät eine bestimmte Eigenschaft besitzt und auf dieses Gerät ein Operator ausgeliefert wird, der eine zu dieser Eigenschaft gehörende Anforderung besitzt. Je nach Ausprägung der Eigenschaft und der Anforderung ist es dann möglich, dass der Operator die Eigenschaft des Geräts während seiner Ausführung beeinflusst. Das führt dazu, dass sich die Eigenschaft aus der Sicht anderer Operatoren, die ebenfalls auf dieses Gerät ausgeliefert werden, verändert. Ein Beispiel dafür ist freier Speicher: Wenn ein Operator die Anforderung formuliert, 300 Megabyte freien Speicher zu benötigen und auf ein Gerät verteilt wird, dass insgesamt 500 Megabyte an freiem Speicher anbietet, dann nimmt der Operator während seiner Ausführung den geforderten Speicher in Anspruch. Dadurch stehen anderen Operatoren, die anschließend auf dasselbe Gerät verteilt werden, nur noch 200 Megabyte an freiem Speicher zur Verfügung. Diese Art der Beeinflussung von Geräteeigenschaften durch Operatoren stellt eine Herausforderung dar, mit der umgegangen werden muss.

Herausforderung 4: Vermeidung eines Single Point of Failure

Ein Ziel des IoT ist es, möglichst ausfallsicher zu sein und die Funktionsfähigkeit und Verfügbarkeit des Gesamtsystems nicht von einer einzelnen Komponente abhängig zu machen. Insofern muss bei der Konzeption der Ausführung darauf geachtet werden, dass nicht versehentlich ein *Single Point of Failure* geschaffen wird, der bei einem Defekt zu einem Ausfall des Gesamtsystems führt.

Herausforderung 5: Lose Kopplung

Die Geräte einer IoT-Umgebung werden so konzipiert, dass zwischen ihnen eine lose Kopplung besteht und sie ohne großen Aufwand durch andere Geräte ersetzt werden können. Dieser Grundsatz muss daher auch im Konzept zur Ausführung der Datenstrommodelle berücksichtigt und das System entsprechend flexibel gehalten werden. So sollte die konzeptionelle Lösung beispielsweise nicht fordern, dass die IoT-Geräte bereits untereinander gegenseitig von ihrer Existenz wissen oder dass sie bereits vom Hersteller aus einen fest integrierten Verweis auf die anderen Geräte besitzen, mit denen sie kommunizieren müssen.

Herausforderung 6: Skalierbarkeit

IoT-Systeme zeichnen sich durch eine hohe Adaptivität und Flexibilität aus. Deshalb muss damit gerechnet werden, dass eine IoT-Umgebung aus nahezu beliebig vielen Geräten bestehen kann und auch für die Datenstrommodelle prinzipiell keine Größenbeschränkungen existieren. Folglich sollte die konzeptionelle Lösung sowohl hinsichtlich der Verteilungsalgorithmen, als auch in Anbetracht des Entwurfs zur Ausführung möglichst gut skalierbar sein und auch mit wachsenden Anwendungsszenarien umgehen können.

Herausforderung 7: Praxisverhalten der IoT-Geräte

Es ist schwierig vorherzusehen, wie sich die IoT-Geräte in der Praxis unter Realbedingungen verhalten. Jedoch muss angenommen werden, dass sie den modellierten Eigenschaften nicht immer genau entsprechen, besonders hinsichtlich Eigenschaften wie Arbeitsspeicher und Rechenauslastung, die in Abhängigkeit von anderen auf dem Gerät ablaufenden Prozessen starken Schwankungen unterworfen sein können. Ein Beispiel dafür stellen Kommunikationsprozesse dar: Wenn eine Vielzahl von Ressourcen in Anspruch genommen werden muss, um die Kommunikation mit anderen IoT-Geräten zu realisieren, dann stehen in der Praxis unter Umständen weniger Ressourcen zur Verfügung, als ursprünglich im Leerlauf des Geräts angenommen worden ist. In der Folge werden Lösungsansätze benötigt, die mit der damit verbundenen Dynamik umgehen können.

4 Konzeption

In diesem Kapitel werden die Lösungskonzepte vorgestellt. Dafür wird zunächst eine ganzheitliche Betrachtung über die entworfenen Konzepte geboten, anschließend werden die Teillösungen, aus denen sich die Gesamtlösung zusammensetzt, jeweils in eigenen Abschnitten im Detail präsentiert. Dabei wird jeweils auch auf die erreichten Ziele und die bewältigten Herausforderungen eingegangen. Die folgenden Teillösungen werden betrachtet:

Anforderungen und Eigenschaften: Bevor der eigentliche Kern der Lösung konzipiert werden kann, müssen dafür zunächst einige Voraussetzungen geschaffen werden. Dazu gehören Konzepte für Anforderungen, die von den Elementen von Datenstrommodellen an die Infrastruktur der IoT-Umgebung gestellt werden und für Eigenschaften, die diese Infrastruktur beschreiben können.

Spezifikation der Eingaben: Die Eingabeartefakte, auf denen die Verteilungsalgorithmen ausgeführt wird, müssen zunächst derart spezifiziert werden, dass sie alle notwendigen Informationen enthalten, die für die Verteilungsentscheidung und die spätere Ausführung des Datenstrommodells benötigt werden.

Entwicklung von Verteilungsalgorithmen: An dieser Stelle werden Algorithmen konzipiert, die es erlauben, Operatoren eines Datenstrommodells auf formeller Ebene auf die Geräte einer IoT-Umgebung abzubilden, unter Berücksichtigung der Anforderungen der Operatoren und der Eigenschaften der Infrastrukturen. Diese Teillösung repräsentiert Ziel 1 der Problembeschreibung.

Konzeption der Ausführung: Im Rahmen dieser Teillösung wird ein System konzipiert, das die Ausführung des Datenstrommodells in der IoT-Umgebung ermöglicht und sicherstellt, dass die Operatoren auf den IoT-Geräten miteinander kommunizieren und Daten austauschen können. Diese Teillösung repräsentiert Ziel 2 der Problembeschreibung.

4.1 Methodischer Ansatz

Zunächst folgt eine Übersicht über die erarbeiteten konzeptionellen Lösungen.

In Abbildung 4.1 ist dargestellt, wie sich die Lösung in den Ablauf von FlexMash eingliedert. Daraus lässt sich auch entnehmen, in welcher Reihenfolge die zur Ausführung des Datenstrommodells benötigten Teilarbeitsschritte durchgeführt werden und auch, welche Artefakte in den einzelnen Schritten verwendet werden und entstehen. Am Anfang steht dabei das Werkzeug FlexMash, mit dessen Hilfe ein Benutzer ein Datenstrommodell erstellen kann. Am Ende des gesamten Prozesses wird das Datenstrommodell in einer gegebenen IoT-Umgebung ausgeführt. Die einzelnen dafür notwendigen Teilschritte werden nachfolgend im Detail beschrieben.

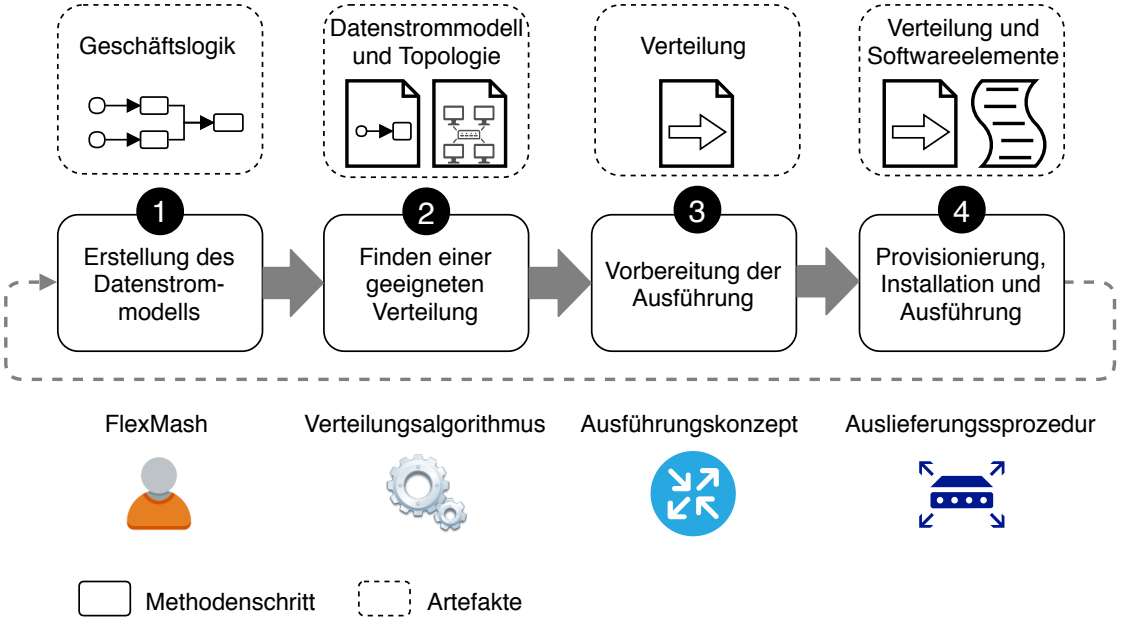

Abbildung 4.1: Methodenschritte und Artefakte der Lösung auf oberster Abstraktionsebene

1. Schritt: Zu Beginn der Methode steht ein Datenstrommodell. Dieses kann in FlexMash modelliert werden und wird für die nachfolgenden Methodenschritte benötigt. Dieser Schritt wird implizit durch FlexMash realisiert und ist damit im Gegensatz zu den Schritten 2, 3 und 4 nicht Gegenstand der konzeptionellen Lösung dieser Bachelorarbeit. Dieser Methodenschritt ist lediglich aus Gründen der Anschaulichkeit ebenfalls in Abbildung 4.1 dargestellt.

2. Schritt: Auf Grundlage des Datenstrommodells und einer Spezifikation der Netzwerktopologie der IoT-Umgebung kann anschließend einer der Verteilungsalgorithmen angewendet werden, um die Operatoren des Datenstrommodells unter Berücksichtigung der Anforderungen in geeigneter Weise auf die IoT-Geräte abzubilden. An dieser Stelle ist zu beachten, dass die Operatoren zu diesem Zeitpunkt noch nicht auf die IoT-Geräte ausgebracht werden. Es wird ausschließlich eine hypothetische Verteilung erzeugt, die die Operatoren auf die IoT-Geräte abbildet. Diese Lösung entspricht dem Resultat dieses Schritts und ist ein Artefakt, das in den nachfolgenden Schritten benötigt wird.

3. Schritt: Das in Schritt 2 erzeugte Verteilungsartefakt ist der Ausgangspunkt, um die Ausführung des Datenstrommodells in der IoT-Umgebung vorzubereiten. Dabei kommt das Ausführungs- und Kommunikationskonzept zum Tragen: Die Daten und Softwareelemente, die zur Ausführung dieser Konzepte benötigt werden, werden in diesem Schritt generiert oder eingelesen und im System hinterlegt, sodass sie im nachfolgenden Schritt, wenn die Operatoren mit den IoT-Geräten provisioniert werden, in der IoT-Umgebung mitinstalliert und -ausgeführt werden können.

4. Schritt: Im letzten Schritt werden die Operatoren aus dem Datenstrommodell mit den IoT-Geräten gemäß der vorgeschlagenen Verteilung provisioniert. Dafür stehen die Softwareelemente, die den Operatoren des Datenstrommodells entsprechen, als Artefakte zur Verfügung und können mit der MBP und dem Modell der Netzwerktopologie ausgeliefert, installiert und ausgeführt werden. Zuvor werden allerdings gegebenenfalls bereits auf den IoT-Geräten in

Betrieb befindliche Operatoren aus früheren Ausführungen angehalten und deinstalliert, um Platz für die neue Verteilung zu machen. Parallel zur Ausbringung der Operatoren wird auch das zuvor vorbereite Ausführungssystem in der IoT-Umgebung installiert und ausgeführt, um den Datentransport zwischen den Operatoren zu ermöglichen. An dieser Stelle ist der Prozess abgeschlossen und kann, wenn der Benutzer ein neues Datenstrommodell in FlexMash erstellt, wieder von vorn gestartet werden.

4.2 Anforderungen und Eigenschaften

Operatoren in Datenstrommodellen können sehr verschiedenartige Ansprüche an die Geräteinfrastruktur stellen, auf der sie ausgeführt werden. Diese können Hardwarevoraussetzungen, Leistungseigenschaften, aber auch die Verfügbarkeit bestimmter Schnittstellen oder Softwaremodule betreffen. IoT-Umgebungen sind heterogen, die darin befindlichen Geräte weisen daher unterschiedliche Eigenschaften auf und potentiell können nicht alle in der Umgebung befindlichen Geräte den Anforderungen eines bestimmten Operators gerecht werden. Daher müssen die *Anforderungen* des Datenstrommodells und die *Eigenschaften* der IoT-Geräte von den Verteilungsalgorithmen bei der Zusammenstellung einer möglichen Verteilung berücksichtigt werden. Beispiele für Anforderungen, die von Operatoren an die ihnen zugrundeliegende Infrastruktur gestellt werden, sind unter anderem das Vorhandensein von ausreichend Arbeitsspeicher, eine Mindestanzahl an Rechenkernen oder die Verfügbarkeit einer MySQL-Datenbank.

Eine Masterarbeit[1], die parallel zu dieser Bachelorarbeit bearbeitet wird, untersucht unterdessen, welche Anforderungen von gegebenen Operationen konkret an die Infrastrukturen gestellt werden und wie die Datenstrommodelle dafür erweitert werden können. Darüber hinaus wird im Rahmen dieser Masterarbeit eine Schnittstelle in FlexMash integriert, mit der die Datenstrommodelle bereits im Werkzeug selbst um Anforderungen ergänzt werden können. Für den weiteren Verlauf dieser Bachelorarbeit wird allerdings bereits ein vorläufiges Konzept für Anforderungen als Arbeitsgrundlage benötigt, auf dessen Basis die Verteilungsalgorithmen entworfen und getestet werden können. Der Entwurf eines solchen Konzepts ist somit auch eine Teilaufgabe dieser Arbeit.

Allerdings können nicht nur die Operatoren der Datenstrommodelle über Anforderungen verfügen: Auch die Verbindungen zwischen den Operatoren, die Datenströme, können Anforderungen an die Verbindungen, über die die Daten bei der Ausführung übermittelt werden. So wäre es beispielsweise denkbar, dass das IoT-System auch personenbezogene Daten verarbeitet, welche nur über verschlüsselte Verbindungen zwischen IoT-Geräten übertragen werden dürfen. Auch Anforderungen an die Bandbreite oder an andere Leistungseigenschaften der Verbindungen sind denkbar.

In Abbildung 4.2 ist ein schematisches Beispiel für ein Datenstrommodell dargestellt, das um Anforderungen ergänzt worden ist. Operator 1 und Operator 2 besitzen jeweils eine Anforderung, Operator 3 dagegen besitzt zwei Anforderungen, die jeweils an die Infrastruktur des Geräts gestellt werden, auf dem sie ausgeführt werden. Auch die Verbindung zwischen Quelle 1 und Operator 1 verfügt über eine Anforderung, die voraussetzt, dass der Datentransport zwischen

[1] Titel: „Enhancing data flow models with computing requirements for distributed IoT environments“, Ausschreibung verfügbar unter https://www.ipvs.uni-stuttgart.de/abteilungen/as/lehre/studentische_arbeiten/masterarbeiten/MA_OperatorPlacement.html

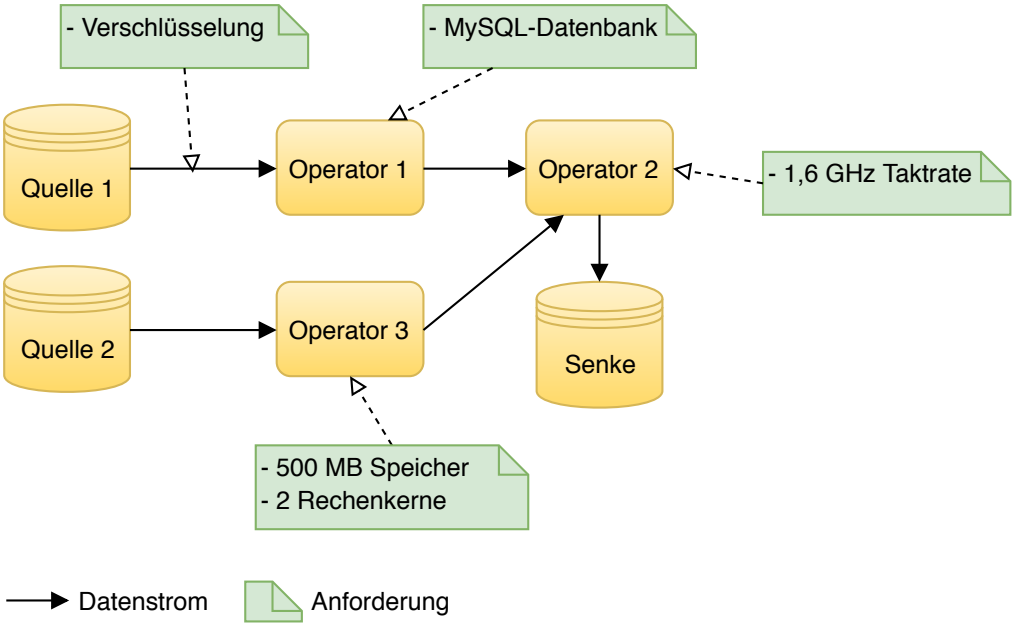

Abbildung 4.2: Beispiel für ein Datenstrommodell mit Anforderungen

den beiden Geräten, auf denen sich Quelle 1 und Operator 1 befinden, nur verschlüsselt erfolgen darf. Es ist allerdings möglich, dass die beiden Geräte, auf die Quelle 1 und Operator 1 in der IoT-Umgebung ausgeliefert werden, im Netzwerk nicht direkt miteinander verbunden sind. Der Datenaustausch muss in diesem Fall indirekt über weitere Geräte erfolgen, die die Daten entsprechend weitervermitteln. Ein solcher Weg im Netzwerk, der von einem Startgerät über möglicherweise mehrere weitere miteinander verbundene Geräte zu einem Zielgerät führt, wird *Netzwerkpfad* genannt. Eine Anforderung einer Verbindung im Datenstrommodell muss von allen Netzwerkverbindungen der IoT-Geräte erfüllt werden, aus denen sich der jeweilige Netzwerkpfad zusammensetzt. Für die Verschlüsselungsanforderung bedeutet dies, dass die Daten im gesamten Netzwerkpfad, über den sie übertragen werden, verschlüsselt werden müssen. Es genügt nicht, dass nur Teilabschnitte des Pfads diese Anforderung erfüllen. In Abbildung 4.3 ist dies anhand von zwei Beispielen für Netzwerkpfade dargestellt: Quelle 1 aus Abbildung 4.2 wird jeweils auf dem Startgerät ausgeführt, Operator 1 auf dem Zielgerät. Da das Medium, von dem Quelle 1 die Daten bezieht, wie beispielsweise ein Sensor, in der Regel fest auf dem Gerät installiert ist, ist das Startgerät in der Abbildung mit einem Sensorsymbol markiert. Die Geräte besitzen keine direkte Verbindung zueinander, sondern müssen die Daten über einen Netzwerkpfad austauschen, der mit Gerät 2 und Gerät 3 zwei weitere Geräte umfasst. Im ersten Beispiel sind alle Teilverbindungen verschlüsselt, die Anforderung nach Verschlüsselung aus Abbildung 4.2 ist somit erfüllt. Im zweiten Beispiel ist die Teilverbindung zwischen dem Gerät 2 und Gerät 3 jedoch nicht gewährleistet. Damit ist die Anforderung nach Verschlüsselung nicht im gesamten Netzwerkpfad erfüllt und aus diesem Grund darf der zweite Pfad in diesem Fall auch nicht zum Datenaustausch verwendet werden.

Es obliegt letztlich den Verteilungsalgorithmen, die in den nachfolgenden Abschnitten dieses Kapitels beschrieben werden, die Verteilung der Operatoren auf die IoT-Geräte so zu wählen, dass die Anforderungen des Datenstrommodells von den Eigenschaften der IoT-Geräte und der Verbindungen zwischen diesen erfüllt werden. Im weiteren Verlauf dieses Abschnitts werden die

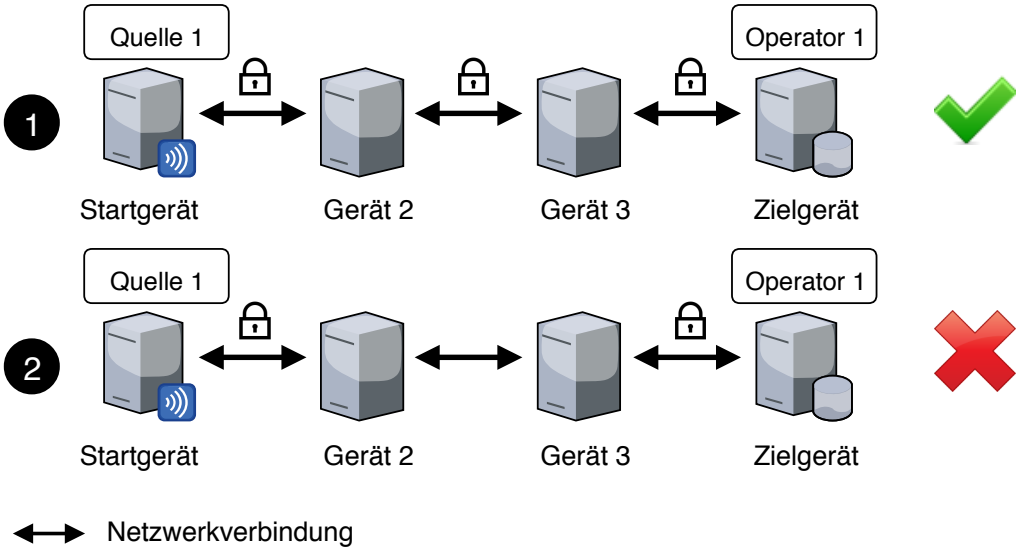

Abbildung 4.3: Beispiele für Netzwerkpfade, die die Verbindungsanforderung erfüllen (1) und nicht erfüllen (2)

Grundlagen dafür gelegt, indem Konzepte für Anforderungen und Eigenschaften eingeführt, die möglichen Beziehungen zwischen diesen definiert und Algorithmen vorgestellt werden, die zum Vergleich von Anforderungen und Eigenschaften eingesetzt werden können.

4.2.1 Ressourcen

Unter einer *Ressource* ist im Rahmen dieser Arbeit ein bestimmtes Hardware- oder Softwaremerkmal eines IoT-Geräts oder einer Verbindung zwischen zwei IoT-Geräten zu verstehen, das quantitativ, nominell oder durch seinen Zustand beschrieben werden kann. Ressourcen bilden die Grundlage für Anforderungen und Eigenschaften. Beispiele für Ressourcen von IoT-Geräten sind:

- Freier Speicherplatz
- Verfügbarer Arbeitsspeicher
- Zugriffsmöglichkeit via SSH

Beispiele für Ressourcen von Verbindungen zwischen jeweils zwei IoT-Geräten sind:

- Verfügbare Bandbreite
- Art der Datenübertragung (Kabelgebunden/Drahtlos)
- Sicherheit der Datenübertragung

4.2.2 Anforderungen

Anforderungen werden benötigt, um die Voraussetzungen zu beschreiben, die Operatoren oder Verbindungen des Datenstrommodells an die Ressourcen der ihnen zugrundeliegende Infrastruktur stellen. Eine Anforderung bezieht sich dabei immer genau auf eine Ressource und besteht aus insgesamt fünf Attributen:

Name: Bei dem Namen handelt es sich um einen eindeutigen, nicht-leeren Bezeichner, der die Ressource spezifiziert, auf die sich die Anforderung bezieht. Er wird zusammen mit dem Datentyp dazu verwendet, um Anforderungen und Eigenschaften zu identifizieren, die sich auf dieselbe Ressource beziehen.

Datentyp: Der Datentyp spezifiziert den Typ, von dem der Wert der Anforderung ist. Mögliche Datentypen sind:

- Boolescher Wert
- Ganzzahl
- Gleitkommazahl
- Zeichenkette

Wert: Der Wert ist von dem Typ, der im Datentyp-Attribut der Anforderung angegeben worden ist. Bei numerischen Datentypen spezifiziert er den Schwellenwert der Anforderung, bei Booleschen Werten und Zeichenketten den Zielwert.

Vergleichsoperator: Der Vergleichsoperator wird verwendet, um eine gegebene Eigenschaft mit der Anforderung zu vergleichen und zu bestimmen, ob die Anforderung von der Eigenschaft erfüllt wird. Bei numerischen Datentypen gibt er an, in welche Richtung des Wertebereichs der Schwellenwert auszulegen ist. Die folgenden Vergleichsoperatoren sind möglich:

= Es wird gefordert, dass der Wert der dazugehörigen Ressource mit dem Wert der Anforderung übereinstimmt.

!= Es wird gefordert, dass der Wert der dazugehörigen Ressource nicht mit dem Wert der Anforderung übereinstimmt.

<, <=, >, >= Es wird gefordert, dass der Wert der dazugehörigen Ressource entweder „kleiner“, „kleiner oder gleich“, „größer“ oder „größer oder gleich“ dem Wert der Anforderung ist. Diese Vergleichsoperatoren stehen nur bei numerischen Datentypen zur Verfügung.

Konsumierend: Dieses Attribut gibt an, ob der Operator oder die Verbindung, die diese Anforderung an die Infrastruktur stellen, die betreffende Ressource bei Ausführung ganz oder teilweise verbrauchen, sodass sie anderen Operatoren oder Verbindungen dann nicht mehr in vollem Umfang zur Verfügung steht. Die möglichen Ausprägungen sind `wahr` und `falsch`.

In Tabelle 4.1 sind fünf Beispiele für Anforderungen aufgelistet. Die erste Anforderung beschreibt, dass der Operator, zu dem diese Anforderung gehört, nur auf IoT-Geräten ausgeführt werden darf, deren Infrastruktur mindestens zwei Rechenkerne umfasst. Da sich die Anzahl der Rechenkerne bei Ausführung des Operators auf einem solchen Gerät nicht ändert, wirkt diese Anforderung nicht-konsumierend: Die Ressource wird während der Ausführung des Operators nicht verbraucht und steht anderen Operatoren, die auf demselben Gerät ausgeführt werden, weiterhin zur Verfügung.

Name	Datentyp	Wert	Operator	Konsumierend
Rechenkerne	Ganzzahl	2	>=	Nein
Speicher	Gleitkommazahl	521.4	>=	Ja
Datenbank	String	MySQL	=	Nein
Verschlüsselung	Bool. Wert	Wahr	=	Nein
Bluetooth	Bool. Wert	Wahr	=	Ja

Tabelle 4.1: Beispiele für Anforderungen

Die zweite Anforderung erhebt einen Anspruch auf 521.4 Megabyte freien Speicher. Zu beachten ist dabei, dass die Einheiten bei diesen Anforderungen nicht spezifiziert werden. Es wird daher automatisch davon ausgegangen, dass für Anforderungen und Eigenschaften, die sich auf dieselbe Ressource beziehen, immer dieselben Einheiten verwendet werden. Da sich bei Ausführung des Operators auf einem Gerät der freie Speicher für weitere Betrachtungen verringert, wirkt diese Anforderung konsumierend. Besitzt das Gerät beispielsweise 800.0 Megabyte an freiem Speicher und wird der Operator mit dieser Anforderung auf dem Gerät ausgeführt, so verbleiben danach noch 278.6 Megabyte an freiem Speicher für nachfolgende Operatoren. Es wird im Rahmen dieser Anforderungen davon ausgegangen, dass bei Deinstallation und Entfernung des Operators von dem Gerät die betreffende Ressource wieder über den vorherigen Eigenschaftswert verfügt, in diesem Fall also wieder über 800.0 Megabyte freien Speicher. Die dritte Anforderung setzt voraus, dass auf dem IoT-Gerät, auf das der Operator installiert werden soll, eine Datenbank vom Typ „MySQL“ installiert ist. Da die Datenbank auch nach Installation und Ausführung des Operators noch zur Verfügung steht, wirkt diese Anforderung nicht-konsumierend. Diese Anforderungen können nicht nur für Operatoren verwendet werden, sondern auch für Verbindungen zwischen den Operatoren im Datenstrommodell, wie die vierte Anforderung in Tabelle 4.1 zeigt. Da ein Datentransport über einen entsprechenden Netzwerkpfad die Verschlüsselung selbst nicht beeinträchtigt, ist diese Anforderung ebenfalls als nicht-konsumierend klassifiziert. Die letzte Beispielanforderung beschreibt, dass der Operator auf dem IoT-Gerät, auf dem er ausgeführt werden soll, eine Bluetooth-Schnittstelle benötigt. Dabei wird angenommen, dass nur ein Operator gleichzeitig diese Schnittstelle nutzen kann. Aus diesem Grund wird die Anforderung als konsumierend eingestuft, da bei Ausführung des Operators die Bluetooth-Schnittstelle keinem anderen Operator mehr zur Verfügung steht.

4.2.3 Eigenschaften

Eigenschaften werden benötigt, um die Charakteristiken zu beschreiben, die IoT-Geräte oder Verbindungen zwischen IoT-Geräten besitzen und bilden das Gegenstück zu den im vorherigen Abschnitt eingeführten Anforderungen. Eine Eigenschaft bezieht sich ebenfalls immer genau auf eine Ressource und besteht aus insgesamt vier Attributen:

Name: Bei dem Namen handelt es sich analog zu den Namen von Anforderungen um einen eindeutigen, nicht-leeren Bezeichner, der die Ressource spezifiziert, auf die sich die Eigenschaft bezieht. Er wird zusammen mit dem Datentyp dazu verwendet, um Anforderungen und Eigenschaften zu identifizieren, die sich auf dieselbe Ressource beziehen.

Datentyp: Der Datentyp spezifiziert den Typ, von dem der Wert der Eigenschaft ist. Es stehen dieselben möglichen Ausprägungen wie bei den Anforderungen zur Verfügung.

Ausgangswert: Der Ausgangswert ist von dem Typ, der im Datentyp-Attribut der Anforderung angegeben worden ist. Bei numerischen Datentypen beschreibt er den Ausgangszustand der Ressource quantitativ, bei Zeichenketten nominell. Bei booleschen Werten drückt er aus, ob eine bestimmte Gegebenheit erfüllt ist oder nicht.

Aktueller Wert: Der aktuelle Wert entspricht standardmäßig dem Ausgangswert und wird nicht explizit spezifiziert. Er wird jedoch beim Vergleich von Anforderungen mit Eigenschaften dazu verwendet, um Zwischenwerte zu speichern, die während der Ausführung beim Verbrauch der betreffenden Ressource durch konsumierende Anforderungen entstehen.

Name	**Datentyp**	**Ausgangswert**
Rechenkerne	Ganzzahl	3
Speicher	Gleitkommazahl	200
Datenbank	String	MongoDB
Bluetooth	Bool. Wert	Falsch
Verschlüsselung	Bool. Wert	Wahr

Tabelle 4.2: Beispiele für Eigenschaften

Analog zu den Anforderungen aus Tabelle 4.1 sind in Tabelle 4.2 Beispiele für mögliche Eigenschaften aufgelistet. Dabei beziehen sich die ersten vier Eigenschaften auf die Infrastruktur von IoT-Geräten, die fünfte Eigenschaft dagegen auf eine Netzwerkverbindung zwischen zwei Geräten in der IoT-Umgebung.

4.2.4 Beziehungen zwischen Anforderungen und Eigenschaften

Damit ein Vergleich von Anforderungen und Eigenschaften möglich wird, müssen Beziehungen zwischen diesen definiert werden. Dafür wird zunächst festgelegt, wann sich eine Anforderung und eine Eigenschaft auf dieselbe Ressource beziehen.

Definition 4.2.1
Sei a eine Anforderung und e eine Eigenschaft. a und e beziehen sich auf dieselbe Ressource genau dann, wenn der Name von a gleich dem Namen von e ist und der Datentyp von a gleich dem Datentyp von e ist. Ist dies der Fall, heißt e *passend* zu a.

Daraus folgt insbesondere, dass sich eine Anforderung und eine Eigenschaft, die entweder nur im Namen oder nur im Datentyp übereinstimmen, nicht auf dieselbe Ressource beziehen und folglich auch nicht passend zueinander sind. Nun kann definiert werden, wann eine Eigenschaft eine Anforderung erfüllt.

Definition 4.2.2
Sei a eine Anforderung und e eine Eigenschaft. wert(a) sei der Wert von a und wert(e) der aktuelle Wert von e. Die Eigenschaft e erfüllt die Anforderung a genau dann, wenn e zu a passend ist und es gilt:

$$\begin{cases} \text{wert}(e) = \text{wert}(a), & \text{für Vergleichsoperator „=“ von } a \\ \text{wert}(e) \neq \text{wert}(a), & \text{für Vergleichsoperator „!=“ von } a \\ \text{wert}(e) < \text{wert}(a), & \text{für Vergleichsoperator „<“ von } a \\ \text{wert}(e) \leq \text{wert}(a), & \text{für Vergleichsoperator „<=“ von } a \\ \text{wert}(e) > \text{wert}(a), & \text{für Vergleichsoperator „>“ von } a \\ \text{wert}(e) \geq \text{wert}(a), & \text{für Vergleichsoperator „>=“ von } a \end{cases}$$

In Tabelle 4.3 sind fünf Beispiele dargestellt, in denen jeweils eine Anforderung und eine Eigenschaft miteinander verglichen werden. Die Datentypen sind aus Gründen der Übersichtlichkeit nicht in der Tabelle aufgeführt, können aber aus den angegebenen Werten abgeleitet werden. Da im ersten Fall die Anforderung und die Eigenschaft unterschiedliche Namen besitzen, passen sie nicht zueinander und die Eigenschaft erfüllt die Anforderung folglich auch nicht. Im zweiten Fall tragen die Anforderung und die Eigenschaft denselben Namen, die angegebenen Werte entsprechen jedoch unterschiedlichen Datentypen, denn die Anforderung verwendet eine Gleitkommazahl, während die Eigenschaft eine Zeichenkette spezifiziert. Folglich passt die Eigenschaft auch nicht zu dieser Anforderung und erfüllt sie dementsprechend auch nicht. In den nachfolgenden Beispielen stimmen Name und Datentyp jeweils überein, deshalb passen die Eigenschaften zu ihren jeweiligen Anforderungen. Im dritten Beispiel verlangt die Anforderung allerdings einen Wert größer oder gleich 500.0, die Eigenschaft gibt aber nur einen niedrigeren Wert an. Aus diesem Grund erfüllt diese Eigenschaft die gegebene Anforderung nicht. Dies ist in den letzten beiden Fällen anders: Dort erfüllt die Eigenschaft jeweils die Anforderung unter dem angegebenen Vergleichsoperator.

Anforderung			**Eigenschaft**		**Beziehungen**	
Name	**Wert**	**Oper.**	**Name**	**Akt. Wert**	**Passend**	**Erfüllend**
Rechenkerne	2	>=	Datenbank	Wahr	Nein	Nein
Speicher	123.4	>=	Speicher	„ROM“	Nein	Nein
RAM	500.0	>=	RAM	434.1	Ja	Nein
RAM	500.0	>=	RAM	889.8	Ja	Ja
OS	„Linux“	!=	OS	„Windows“	Ja	Ja

Tabelle 4.3: Beispiele für Beziehungen zwischen Anforderungen und Eigenschaften

In der Regel besitzen Operatoren oder Datenströme zwischen Operatoren jedoch nicht nur eine Anforderung und auch IoT-Geräte und Netzwerkverbindungen werden meist mit mehreren Eigenschaften modelliert. Aus diesem Grund müssen auch Mengen von Anforderungen und Eigenschaften betrachtet werden.

Definition 4.2.3
Sei A eine Menge von Anforderungen und E eine Menge von Eigenschaften, wobei sich alle Anforderungen in A auf verschiedene Ressourcen beziehen. *E erfüllt A* genau dann, wenn gilt:

$$\forall a \in A : \exists e \in E : e \text{ erfüllt } a$$

Damit eine Menge von Eigenschaften eine Menge von Anforderungen erfüllt, muss es also für jede der Anforderungen in der Anforderungsmenge eine Eigenschaft in der Menge der Eigenschaften geben, die die Anforderung erfüllt. Kann eine Anforderung der Anforderungsmenge nicht erfüllt werden, kann die gesamte Menge nicht erfüllt werden. Aus Definition 4.2.3 folgt damit implizit, dass es für jede Anforderung der Anforderungsmenge auch eine passende Eigenschaft geben muss, damit die Menge der Anforderungen erfüllt werden kann. Andersherum gilt dies nicht: Damit eine Menge von Eigenschaften eine Menge von Anforderungen erfüllen kann, dürfen in der Eigenschaftsmenge auch Eigenschaften enthalten sein, die zu keiner der Anforderungen der Anforderungsmenge passend sind.

4.2.5 Algorithmischer Vergleich

Algorithmus 4.1 Vergleich einer Anforderung mit einer Eigenschaft

```
1:  // Eingabe: Anforderung, Eigenschaft
2:  function SatisfiesRequirement(requirement, capability)
3:      // Passt die Eigenschaft zu der Anforderung?
4:      if (requirement.name ≠ capability.name) and (requirement.type ≠ capability.type) then
5:          return false
6:      end if
7:      // Vergleiche Werte in Abhängigkeit vom Operator
8:      switch requirement.operator do
9:          case "=" :
10:             return capability.value = requirement.value
11:         case "!=" :
12:             return capability.value ≠ requirement.value
13:         case ">" :
14:             return capability.value > requirement.value
15:         case ">=" :
16:             return capability.value ≥ requirement.value
17:         case "<" :
18:             return capability.value < requirement.value
19:         case "<=" :
20:             return capability.value ≤ requirement.value
21:     end switch
22: end function
```

Auf Grundlage der im vorherigen Abschnitt vorgestellten Konzepte können Algorithmen entworfen werden, um Anforderungen ("Requirements") mit Eigenschaften ("Capabilities") zu vergleichen. In Algorithmus 4.1 ist der Pseudocode eines Algorithmus dargestellt, der verifiziert, ob eine Eigenschaft eine gegebene Anforderung erfüllt. Dafür wird zunächst geprüft, ob die Eigenschaft zu der Anforderung passend ist, anschließend wird in Abhängigkeit von dem Operator der Anforderung der Wert der Anforderung mit dem aktuellen Wert der Eigenschaft verglichen.

Algorithmus 4.2 Vergleich einer Menge von Anforderungen mit einer Menge von Eigenschaften

```
1:  // Eingabe: Menge von Anforderungen, Menge von Eigenschaften
2:  function SatisfiesRequirementSet(Requirements, Capabilities)
3:      // Iteriere über alle Anforderungen
4:      for all requirement ∈ Requirements do
5:          satisfied ← false
6:          // Iteriere über alle Eigenschaften
7:          for all capability ∈ Capabilities do
8:              // Erfüllt die Eigenschaft die Anforderung?
9:              if SatisfiesRequirement(requirement, capability) then
10:                 satisfied ← true
11:                 break
12:             end if
13:         end for
14:         if ¬satisfied then
15:             return false
16:         end if
17:     end for
18:     return true
19: end function
```

Algorithmus 4.2 baut auf Algorithmus 4.1 auf und realisiert den bereits zuvor in Definition 4.2.3 formal definierten Vergleich zwischen einer Menge von Anforderungen und einer Menge von Eigenschaften. Dabei werden alle Anforderungen einzeln betrachtet und jeweils mit allen Eigenschaften abgeglichen. Existiert für eine Anforderung keine erfüllende Eigenschaft, bricht der Algorithmus mit dem Ergebnis `falsch` ab, ansonsten gibt er am Ende `wahr` zurück. Dabei wird vorausgesetzt, dass sich die Anforderungen in der Anforderungsmenge paarweise durch den Namen oder den Datentyp unterscheiden, sodass keine Anforderungen enthalten sind, die sich auf dieselbe Ressource beziehen.

4.2.6 Konsumierende Anforderungen

Wie bereits in den vorhergehenden Kapiteln beschrieben, muss auch der Fall berücksichtigt werden, dass ein Operator Ressourcen eines IoT-Geräts oder einer Netzwerkverbindung konsumiert. Dafür wurde in Abschnitt 4.2.2 das Konzept konsumierender Anforderungen eingeführt. Diese benötigen auch eine algorithmische Entsprechung, die es erlaubt, den aktuellen Wert von Eigenschaften gemäß den Werten von Anforderungen anzupassen und auch wieder zurückzusetzen. In Algorithmus 4.3 ist dies für numerische Datentypen realisiert: Der Algorithmus passt den Wert einer Eigenschaft gemäß einer gegebenen Anforderung an, sofern diese konsumierend ist und die Eigenschaft die Anforderung erfüllt. In Abhängigkeit von dem gewählten Vergleichsoperator wird der Wert der Anforderung dabei von dem aktuellen Wert der Eigenschaft entweder subtrahiert oder auf den Wert der Eigenschaft addiert. Für boolesche Datentypen und Zeichenketten funktioniert dies anders: Bei ersteren wird der aktuelle Wert der Eigenschaft invertiert, Zeichenketten werden dagegen durch leere Zeichenketten ersetzt.

Algorithmus 4.3 Konsum einer Eigenschaft durch eine Anforderung

```
1:  // Eingabe: Anforderung, Eigenschaft
2:  procedure ConsumeCapability(requirement, capability)
3:      // Falls Anforderung nicht-konsumierend, breche ab
4:      if ¬ requirement.consuming then
5:          return
6:      end if
7:      // Falls die Eigenschaft die Anforderung nicht erfüllt, breche ab
8:      if ¬SatisfiesRequirement(requirement, capability) then
9:          return
10:     end if
11:     // Addiere bei den Vergleichsoperatoren < und <=, sonst subtrahiere
12:     if requirement.operator ∈ {<, <=} then
13:         capability.value ← capability.value + requirement.value
14:     else
15:         capability.value ← capability.value − requirement.value
16:     end if
17: end procedure
```

Algorithmus 4.4 übernimmt dieselbe Aufgabe für eine Menge von Eigenschaften und Anforderungen. Dabei wird jede Eigenschaft zunächst mit jeder Anforderung der Anforderungsmenge verglichen. Erfüllt die Eigenschaft eine Anforderung und ist die Anforderung konsumierend, wird der aktuelle Wert der Eigenschaft gemäß Algorithmus 4.3 angepasst.

Algorithmus 4.4 Konsum einer Menge von Eigenschaften durch eine Menge von Anforderungen

```
1:  // Eingabe: Menge von Anforderungen, Menge von Eigenschaften
2:  function ConsumeCapabilitySet(Requirements, Capabilities)
3:      // Iteriere über alle Anforderungen
4:      for all requirement ∈ Requirements do
5:          // Iteriere über alle Eigenschaften
6:          for all capability ∈ Capabilities do
7:              // Erfüllt die Eigenschaft die Anforderung?
8:              if SatisfiesRequirement(requirement, capability) then
9:                  // Konsumiere die Eigenschaft durch die Anforderung
10:                 ConsumeCapability(requirement, capability)
11:             end if
12:         end for
13:     end for
14: end function
```

Nun wird ein Algorithmus benötigt, der es ermöglicht, den in Form des aktuellen Werts einer Eigenschaft repräsentierten Konsum einer Ressource wieder rückgängig zu machen. Das ist beispielsweise erforderlich, wenn ein Operator eine konsumierende Anforderung an die Infrastruktur stellt und von dem IoT-Gerät, auf dem er ausgeführt wird, wieder entfernt wird. Dadurch gibt er die Ressourcen wie beispielsweise Speicherplatz, die er zuvor auf dem Gerät in Anspruch genommen hat, wieder frei. Selbstverständlich trifft das auch auf Datenströme zwischen zwei Operatoren zu,

Algorithmus 4.5 Aufhebung des Konsums einer Eigenschaft durch eine Anforderung

```
1:  // Eingabe: Anforderung, Eigenschaft
2:  procedure UndoConsumeCapability(requirement, capability)
3:      // Falls Anforderung nicht-konsumierend, breche ab
4:      if ¬ requirement.consuming then
5:          return
6:      end if
7:
8:      // Je nach Operator wird der Konsum auf unterschiedliche Weise aufgehoben
9:      if requirement.operator ∈ {<, <=} then
10:         // Breche ab, falls der neue Wert den Ausgangswert unterschreiten würde
11:         if (capability.value − requirement.value) < capability.original_value then
12:             return
13:         end if
14:         // Hebe Konsum auf
15:         capability.value ← capability.value − requirement.value
16:     else
17:         // Breche ab, falls der neue Wert den Ausgangswert überschreiten würde
18:         if (capability.value + requirement.value) > capability.original_value then
19:             return
20:         end if
21:         // Hebe Konsum auf
22:         capability.value ← capability.value + requirement.value
23:     end if
24: end procedure
```

die konsumierende Anforderungen an den Netzwerkpfad stellen, über den die Daten transportiert werden. Wird der zuletzt zur Abbildung des Datenstromes verwendete Netzwerkpfad durch einen neuen ersetzt, werden auf dem vorhergehenden Netzwerkpfad die durch den Datentransport in Anspruch genommenen Ressourcen, wie beispielsweise Bandbreite, wieder freigegeben und stehen damit anderen Datenströmen zur Verfügung. Algorithmus 4.5 setzt dies für numerische Datentypen und jeweils eine Anforderung und eine Eigenschaft um: Zuerst wird geprüft, ob die Anforderung wirklich eine konsumierende ist. Falls nicht, bricht der Algorithmus ab. Ansonsten wird der Konsum in Abhängigkeit davon, welcher Vergleichsoperator in der Anforderung spezifiziert worden ist, durch Addition oder Subtraktion des Anforderungswert wieder rückgängig gemacht. Dabei wird insbesondere sichergestellt, dass der neue aktuelle Wert der Eigenschaft niemals den Ausgangswert überschreiten beziehungsweise unterschreiten kann. Die Algorithmen für boolesche Werte und Zeichenketten als Datentypen funktionieren jeweils entsprechend.

Auch hier bedarf es eines Algorithmus, der die Aufhebung des Konsums von Ressourcen für Mengen von Anforderungen und Mengen von Eigenschaften durchführen kann. Dieser ist jedoch analog zu Algorithmus 4.4 zu entwerfen und wird deshalb an dieser Stelle nicht zusätzlich dargestellt.

Die in diesem Abschnitt vorgestellten Konzepte für Vergleiche zwischen Anforderungen und Eigenschaften erfüllen Teilziel 1.5 aus Abschnitt 3.2.

4.3 Spezifikation der Eingaben

Die Verteilungsalgorithmen benötigen zwei Artefakte als Eingaben, um Instanzen des Verteilungsproblems lösen zu können: Zum einen das Datenstrommodell, das von FlexMash erzeugt wird und zum anderen um ein Modell der Netzwerktopologie der betrachteten IoT-Umgebung. In diesem Abschnitt werden die für beide Eingaben erforderlichen Formate spezifiziert.

4.3.1 Datenstrommodelle

Damit die Verteilungsalgorithmen die Anforderungen, die die Elemente des Datenstrommodells an ihre Umgebung stellen, auch tatsächlich berücksichtigen können, müssen die Anforderungen spezifiziert werden. Dies kann durch Annotation der von FlexMash erzeugten Datenstrommodelle erreicht werden. Im Folgenden wird nun zunächst auf das Format eingegangen, das in FlexMash erzeugte Datenstrommodelle besitzen und anschließend erläutert, wie dieses gemäß des in dieser Bachelorarbeit vorgeschlagenen Konzepts um Anforderungen erweitert werden kann.

Format

Datenstrommodelle werden in FlexMash gemäß der JavaScript Object Notation (JSON) [Bra17] spezifiziert. In Listing A.1 ist ein Beispiel für ein durch FlexMash erzeugtes Datenstrommodell dargestellt. Die Operatoren des Datenstrommodells werden dabei dem Array mit dem Schlüssel `nodes` hinzugefügt. Jeder Operator besteht aus vier Attributen, die nachfolgend kurz erläutert werden.

guild: Dieses Attribut gibt den Namen des Operators an und ist innerhalb des Datenstrommodells eindeutig.

serviceId: Dieses Attribut enthält den eindeutigen Namen eines Verbunds von Softwareelementen, der zu diesem Operator gehört. Dabei handelt es sich um die Softwareelemente, die bei der Auslieferung eines Operators auf ein IoT-Gerät installiert und ausgeführt werden sollen.

properties: Das Objekt mit diesem Schlüssel bietet Raum, um das Schema des Datenstrommodells zu erweitern.

target: Dieses Attribut spezifiziert eine Liste, die die Namen aller Operatoren enthält, zu denen der Operator eine Verbindung besitzt. Diese Verbindung ist gerichtet: Der Operator, zu dem die Liste gehört, ist der Ursprungsknoten und die Operatoren in der Liste sind die Zielknoten. Mehrfachverbindungen zwischen denselben Operatoren sind nicht möglich.

Das Datenstrommodell aus Listing A.1 entspricht der grafischen Darstellung in Abbildung 4.4. Es ist anzumerken, dass dieses Modellierungsschema die Operatoren nicht explizit in Datenquellen und Datensenken unterteilt. Diese Klassifizierung kann jedoch anhand der Verbindungen vorgenommen werden, die zwischen den Operatoren modelliert werden: Besitzt ein Operator keine eingehenden Verbindungen, handelt es sich um eine Datenquelle, besitzt er keine ausgehenden Verbindungen, handelt es sich um eine Datensenke.

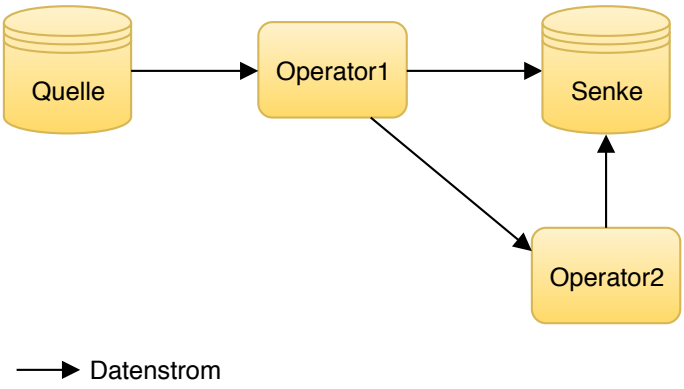

Abbildung 4.4: Grafische Repräsentation des Datenstrommodells aus Listing A.1

Erweiterung

Durch Erweiterung des im vorherigen Abschnitt vorgestellten Schemas von FlexMash können die Operatoren des Datenstrommodells mit Anforderungen annotiert werden. Im Rahmen des Konzepts dieser Arbeit geschieht dies wie folgt: Es wird eine Liste mit dem Schlüssel `requirements` an das Objekt `properties` angefügt, die alle Anforderungen enthält, die der jeweilige Operator an die IoT-Geräte stellt. Die Anforderungen selbst werden als JSON-Objekte spezifiziert, die die in Tabelle 4.4 aufgeführten Attribute erhalten. Diese Modellierung entspricht dem in Abschnitt 4.2.2 vorgestellten Konzepts für Anforderungen.

Schlüssel	Datentyp	Beschreibung
name	Zeichenkette	Name der Anforderung
value	Beliebig	Wert der Anforderung
comparator	Zeichenkette	Vergleichsoperator der Anforderung
consuming	Bool. Wert	Gibt an, ob die Anforderung konsumierend wirkt

Tabelle 4.4: Attribute von Anforderungen im Datenstrommodell

Das Datenstrommodell muss allerdings auch in anderer Hinsicht erweitert werden: Für die Verteilungsalgorithmen ist es wichtig, zu wissen, welchen Geräten in der Netzwerktopologie die im Datenstrommodell spezifizierten Quellen entsprechen. Da die Datenquellen, wie beispielsweise Sensoren, in der Regel festen IoT-Geräten zugeordnet sind und die zum Auslesen der Daten notwendigen Softwareelemente deshalb nur auf dieses spezielle Gerät ausgeliefert werden dürfen, ist eine solche Verknüpfung für die Verteilungsalgorithmen erforderlich. Handelt es sich bei dem betrachteten Operator um eine Datenquelle, kann auch für diese Erweiterung das Objekt mit dem Schlüssel `properties` genutzt werden: Es wird ein Attribut mit dem Schlüssel `deviceId` hinzugefügt, dessen Wert der Name des Geräts ist, auf dem die Softwareelemente der Datenquelle ausgeführt werden sollen. Zu beachten ist, dass diese Erweiterung nur bei Datenquellen erforderlich ist; das Attribut `deviceId` ist somit grundsätzlich optional für JSON-Operator-Objekte.

In Listing 4.1 ist exemplarisch dargestellt, wie die Spezifikation eines Zielgeräts und die Modellierung von Anforderungen im erweiterten Modellierungsschema erfolgt. Das Zielgerät der Datenquelle, zu dessen JSON-Objekt das abgebildete `properties`-Objekt gehört, trägt den Namen `raspberry1`.

```
"properties":{
   "deviceId": "raspberry1",
   "requirements":[
      {
         "name": "ram",
         "value": 1000,
         "comparator": ">=",
         "consuming":true
      },
      {
         "name": "database",
         "value": true,
         "comparator": "=",
         "consuming": false
      }
   ]
}
```

Listing 4.1: Beispiel für die Annotation eines Operators mit Anforderungen (Auszug)

Darüber hinaus werden insgesamt zwei Anforderungen modelliert: Eine konsumierende Anforderung, die einen Anspruch auf eine Mindestmenge an Arbeitsspeicher erhebt und eine zweite, nicht-konsumierende, die die Verfügbarkeit einer Datenbank auf dem IoT-Gerät erfordert.

Im Rahmen dieser Bachelorarbeit wird keine Möglichkeit geschaffen, um auch Anforderungen von Verbindungen zwischen zwei Operatoren in die von FlexMash erzeugten Datenstrommodelle zu integrieren. Der Grund dafür ist, dass die Annotation der Datenstrommodelle bereits in der in Abschnitt 4.2 erwähnten Masterarbeit eingehend untersucht wird und eine tiefergreifende Konzeption in dieser Arbeit unter Umständen zu größeren Inkompatibilitäten zwischen den in beiden Arbeiten vorgeschlagenen Formaten führen würde. Deshalb beschränkt sich die Annotation der Datenstrommodelle zunächst auf die Anforderungen von Operatoren. Die Anforderungen von Verbindungen werden allerdings trotzdem weiterhin in den Konzepten für die Verteilungsalgorithmen mitberücksichtigt.

4.3.2 Modellierung von Netzwerktopologien

Damit die Verteilungsalgorithmen in der Lage sind, die in den Datenstrommodellen annotierten Anforderungen mit den Eigenschaften der IoT-Umgebung zu vergleichen, müssen ihnen diese Eigenschaften bekannt sein. Ferner müssen die Algorithmen den Aufbau und die Struktur der IoT-Umgebung kennen, um entscheiden zu können, welcher Operator mit einem bestimmten IoT-Gerät provisioniert werden soll. Dazu bedarf es eines Modells der Netzwerktopologie, in welchem spezifiziert wird, wie die IoT-Umgebung auf struktureller Ebene aufgebaut ist und welche IoT-Geräte ihr angehören. Auch die Eigenschaften der IoT-Geräte werden darin erfasst. Dieses Modell bildet dann gemeinsam mit dem Datenstrommodell die Eingaben der Verteilungsalgorithmen.

Wie aus Abbildung 4.1 hervorgeht, muss allerdings auch die Prozedur, die die Auslieferung der Operatoren auf die IoT-Geräte vornimmt, wissen, wie die Umgebung aufgebaut ist und wie die Geräte in dieser angesprochen und adressiert werden können. Dazu muss sie die folgenden Merkmale der IoT-Geräte kennen:

- MAC-Adresse des Geräts
- IP-Adresse des Geräts im Netzwerk
- Benutzername auf dem Gerät für SSH-Zugriff

Es liegt nahe, diese Informationen ebenfalls in das Modell der Netzwerktopologie zu integrieren, da auf diese Weise dasselbe Modell sowohl für die Verteilungsalgorithmen, als auch für die Auslieferungsprozedur verwendet werden kann und es in der Folge nur einmal von dem Benutzer angegeben werden muss.

Für das Modell kann auf die IoT-Lite Ontology[2] zurückgegriffen werden. Dabei handelt es sich um eine leichtgewichtige Meta-Ontologie, die es erlaubt, Ressourcen, Entitäten und Dienste des IoT auf hierarchische Weise abzubilden. Sie ist leicht erweiterbar, sodass IoT-Umgebungen nahezu beliebig detailliert modelliert werden können [BEBT16]. Das der Ontologie zugrundeliegende Schema wurde von dem Institut für Parallele und Verteilte Systeme der Universität Stuttgart (IPVS) bereits in Richtung des in dieser Bachelorarbeit benötigten Umfangs erweitert und auch für vorhergehende Projekte im Zusammenhang mit dem IoT eingesetzt. Allerdings fehlen dem Schema noch einige weitere für diese Bachelorarbeit benötigte Elemente, die deshalb als Teil der konzeptionellen Lösung hinzugefügt werden. Diese Erweiterungen werden nachfolgend anhand ihres Elementnamens beschrieben und können im Anhang in Listing A.2, einem Auszug des dem Modell zugrundeliegenden Schemas im JSON-Format, eingesehen werden.

ipvs:macAddress: Ermöglicht es, eine MAC-Adresse für ein IoT-Gerät zu spezifizieren.

ipvs:ipAddress: Ermöglicht es, eine IP-Adresse für ein IoT-Gerät zu spezifizieren.

ipvs:username: Ermöglicht es, einen Betriebssystem-Benutzernamen für ein IoT-Gerät zu spezifizieren, der für den SSH-Zugriff auf das Gerät verwendet werden kann.

ipvs:DeviceCapability: Ermöglicht es, eine Geräte-Eigenschaft zu spezifizieren, welche dem Konzept für Eigenschaften aus Abschnitt 4.2.3 entspricht.

ipvs:CapabilityName: Ermöglicht es, einen Namen für eine Eigenschaft zu spezifizieren.

ipvs:CapabilityValue: Ermöglicht es, einen Wert für eine Eigenschaft zu spezifizieren.

ipvs:hasCapabilities: Ermöglicht es, einem IoT-Gerät eine Liste von Eigenschaften zuzuweisen.

In Listing 4.2 ist ein Beispiel für die Modellierung eines IoT-Geräts in einer IoT-Umgebung mittels dieser Erweiterungen dargestellt. Dieses Gerät trägt den Namen `ipvs:RaspberryPi-1` und ist vom Typ `ipvs:RaspberryPi`, welcher in der durch das IPVS vorgenommenen Erweiterung des Schemas der IoT-Lite Ontology definiert worden ist. Der Name entspricht dabei demselben Namen, der bei Datenquellen auch im Datenstrommodell hinterlegt wird. Das Gerät verfügt über die MAC-Adresse `aa-aa-aa-aa-aa-aa`, die IP-Adresse `192.168.178.1`, den Benutzernamen `superman` und besitzt zwei

[2] Dokumentation verfügbar unter https://www.w3.org/Submission/iot-lite/

```
"@graph":[
   {
      "@id": "ipvs:RaspberryPi-Capability-1",
      "@type": "ipvs:DeviceCapability",
      "ipvs:CapabilityName": "cores",
      "ipvs:CapabilityValue": 2
   },
   {
      "@id": "ipvs:RaspberryPi-Capability-2",
      "@type": "ipvs:DeviceCapability",
      "ipvs:CapabilityName": "database",
      "ipvs:CapabilityValue": true
   },
   {
      "@id": "ipvs:RaspberryPi-1",
      "@type": "ipvs:RaspberryPi",
      "ipvs:macAddress": "aa-aa-aa-aa-aa-aa",
      "ipvs:ipvs:ipAddress": "192.168.178.1",
      "ipvs:username": "superman",
      "ipvs:hasCapabilities": [
         {
            "@id": "ipvs:RaspberryPi-Capability-1"
         },
         {
            "@id": "ipvs:RaspberryPi-Capability-2"
         }
      ]
   }
]
```

Listing 4.2: Beispiel für ein Modell einer Netzwerktopologie mit einem Gerät (Auszug)

Eigenschaften: Die erste beschreibt, dass das Gerät über zwei Rechenkerne verfügt, die zweite, dass auf dem Gerät eine Datenbank installiert ist. Diese Eigenschaften können nun von einem Verteilungsalgorithmus, der über das Modell der Netzwerktopologie verfügt, verwendet werden, um zu prüfen, ob sie den Anforderungen von Operatoren genügen.

Es ist zu beachten, dass es das Modell in seiner aktuellen Version nicht erlaubt, Netzwerkverbindungen zwischen zwei IoT-Geräten explizit zu spezifizieren. Deshalb wird angenommen, dass in einer Instanz des Modells alle unter der Liste mit dem Schlüssel `@graph` aufgeführten IoT-Geräte gegenseitig miteinander verbunden sind, sodass auf struktureller Ebene ein vollständiger Graph entsteht. Aus demselben Grund ist es in diesem Modell auch nicht möglich, Eigenschaften von Verbindungen zwischen zwei IoT-Geräten zu modellieren oder die Verbindungen anhand ihrer Übertragungsdistanz im Netzwerk zu gewichten. Die Konzepte, die im weiteren Verlauf dieses Kapitels vorgestellt werden, machen sich jedoch von diesen Einschränkungen frei und berücksichtigen sowohl Netzwerktopologien, die keine vollständigen Graphen repräsentieren, als auch Eigenschaften

von Netzwerkverbindungen von Geräten. Sollte das Modell für Netzwerktopologien also in Zukunft entsprechend erweitert werden, können die in dieser Arbeit vorgeschlagenen Konzepte weiterverwendet werden.

4.4 Entwicklung von Verteilungsalgorithmen

In diesem Abschnitt werden die Konzepte für Verteilungsalgorithmen vorgestellt, die es erlauben, die Operatoren aus Datenstrommodellen unter Verwendung des Modells der Netzwerktopologie auf möglichst geeignete Weise auf die IoT-Geräte der IoT-Umgebung abzubilden. Dies erfolgt jeweils unter Berücksichtigung der Anforderungen, die die Operatoren an die Infrastruktur der IoT-Umgebung stellen. Im Folgenden wird zunächst das zugrundeliegende Verteilungsproblem im Detail beschrieben und anschließend einschließlich der dazugehörenden Ein- und Ausgaben formalisiert. Darauf aufbauend werden die Konzepte und Eigenschaften zweier möglicher Verteilungsalgorithmen vorgestellt, erklärt und anhand von Pseudocode und Beispielen verdeutlicht.

4.4.1 Verteilungsproblem

Die Verteilungsalgorithmen, deren Konzepte in den nachfolgenden Abschnitten vorgestellt werden, lösen jeweils das Verteilungsproblem für eine gegebene Probleminstanz. Dieses wurde bereits in Abschnitt 3.2 beschrieben und wird an dieser Stelle vertieft.

Die Ausgangssituation dieses Problems bilden ein Datenstrommodell und ein Modell einer Netzwerktopologie, wie sie in Abschnitt 4.3 spezifiziert worden sind. Allerdings wird angenommen, dass die dort beschriebenen Einschränkungen nicht gelten: Das Datenstrommodell spezifiziert damit also Operatoren und Verbindungen zwischen diesen Operatoren, die jeweils Anforderungen besitzen können. Dabei wird vorausgesetzt, dass das Datenstrommodell auf logischer Ebene bereits hinsichtlich Effizienz optimiert worden ist. Das Modell der Netzwerktopologie beschreibt den Aufbau der IoT-Umgebung, einschließlich der darin befindlichen IoT-Geräte und den Netzwerkverbindungen zwischen diesen. Dabei wird nicht vorausgesetzt, dass alle Geräte paarweise direkt miteinander verbunden sind. Weiter spezifiziert das Modell auch die Eigenschaften, die sowohl IoT-Geräte, als auch die Netzwerkverbindungen zwischen diesen Geräten aufweisen. Ferner wird angenommen, dass die Geräteverbindungen im Modell der Netzwerktopologie in Form von *Netzwerkdistanzen* numerisch gewichtet werden. Bei diesen Distanzen handelt es sich um ein Kostenmaß, das auf Grundlage von Netzwerkmetriken wie Latenz und Bandbreite erstellt wird und mit den typischerweise in Routingalgorithmen zum Einsatz kommenden Distanzmaßen [WDA15] verglichen werden kann. Die Netzwerkdistanz ermöglicht es, zu entscheiden, welche Verbindungen der Netzwerktopologie für den Datentransport als geeignet erscheinen. Verbindungen mit kleineren Distanzen werden dabei Verbindungen mit größeren Distanzen grundsätzlich vorgezogen. Die genaue Zusammensetzung und Berechnung der Netzwerkdistanzen wird dem Anwender überlassen, der die Netzwerktopologie modelliert. Auf diese Weise hat er die Möglichkeit, die Operatorverteilung durch die Algorithmen hinsichtlich seiner eigenen Ziele und Anforderungen zu optimieren. Es ist jedoch empfehlenswert, die Netzwerkdistanzen so zu definieren, dass die Verbindungen zwischen Edge-Geräten tendenziell niedrigere Distanzen erhalten als Verbindungen, in denen Fog-Geräte beteiligt sind. Zusätzlich

sollten Verbindungen, die Cloud-Geräten beinhalten, die verhältnismäßig höchsten Distanzen tragen. Dies führt dann wie gewünscht dazu, dass die Algorithmen Verteilungen anstreben, in denen die Daten möglichst nahe an der Quelle verarbeitet werden.

Auf Grundlage dieser Ausgangssituation beschreibt das Verteilungsproblem die Aufgabe, die im Datenstrommodell enthaltenen Operatoren und Datenströme so auf die IoT-Geräte und Netzwerkpfade des Modells der Netzwerktopologie abzubilden, dass gilt:

1. Jeder Operator des Datenstrommodells wird auf ein IoT-Gerät der Netzwerktopologie abgebildet.
2. Jede Datenquelle wird auf das IoT-Gerät abgebildet, dessen Name im Datenstrommodell hinterlegt ist.
3. Jede Verbindung im Datenstrommodell wird unter Aufrechterhaltung der Konsistenz auf einen Netzwerkpfad der Netzwerktopologie abgebildet. Das setzt voraus, dass die Start- und Zielgeräte des Pfads denjenigen Geräten entsprechen, auf denen auch der Start- beziehungsweise der Zieloperator der Verbindung ausgeführt werden.
4. Jedes IoT-Gerät erfüllt die möglicherweise konsumierend-wirkenden Anforderungen der Operatoren, die auf das Gerät abgebildet werden.
5. Jeder Netzwerkpfad erfüllt die möglicherweise konsumierend-wirkenden Anforderungen der Datenströme, die auf den Pfad abgebildet werden.
6. Die Summe der Netzwerkdistanzen der Verbindungen, über die Daten in der IoT-Umgebung transportiert werden, ist möglichst gering.

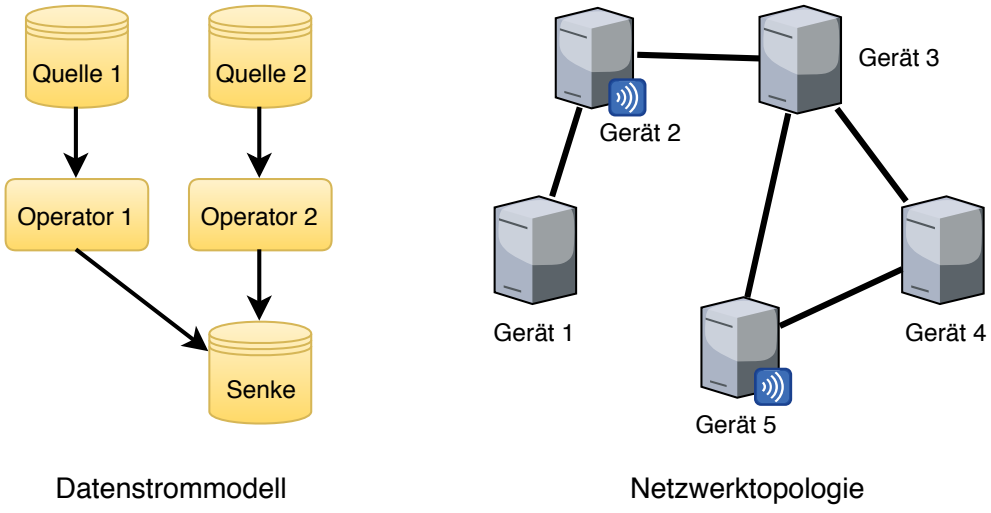

Abbildung 4.5: Ausgangssituation für die Verteilungsalgorithmen

Anhand eines Beispiels kann die Notwendigkeit dieser Forderungen deutlich gemacht werden: In Abbildung 4.5 sind jeweils ein Datenstrommodell und eine IoT-Umgebung, repräsentiert durch ein Modell der Netzwerktopologie, dargestellt, was der Ausgangssituation für die Verteilungsalgorithmen entspricht. Dabei wird angenommen, dass sich der für Quelle 1 erforderliche Sensor auf Gerät 2 der Umgebung befindet, der für Quelle 2 auf Gerät 5. Das Ziel ist es, das Datenstrommodell in der Umgebung vollständig und korrekt auszuführen. Dafür ist es wie in der ersten Forderung beschrieben

notwendig, alle Operatoren des Datenstrommodells auch tatsächlich auf Geräte der IoT-Umgebung abzubilden. Andernfalls werden die Softwareelemente der fehlenden Operatoren im weiteren Verlauf nicht in die IoT-Umgebung ausgeliefert und installiert, was dann die Funktionalität des Systems beeinträchtigt. Da die Datenquellen ihre Daten in der Regel von Sensoren beziehen, welche fest auf bestimmten Geräten der IoT-Umgebung angebracht sind, ist es zudem wie in Abschnitt 4.3.1 beschrieben erforderlich, dass die Softwareelemente der Datenquellen ausschließlich auf die dafür im Datenstrommodell vorgesehenen Geräte ausgeliefert werden. Dies entspricht der zweiten der zuvor vorgestellten Forderungen.

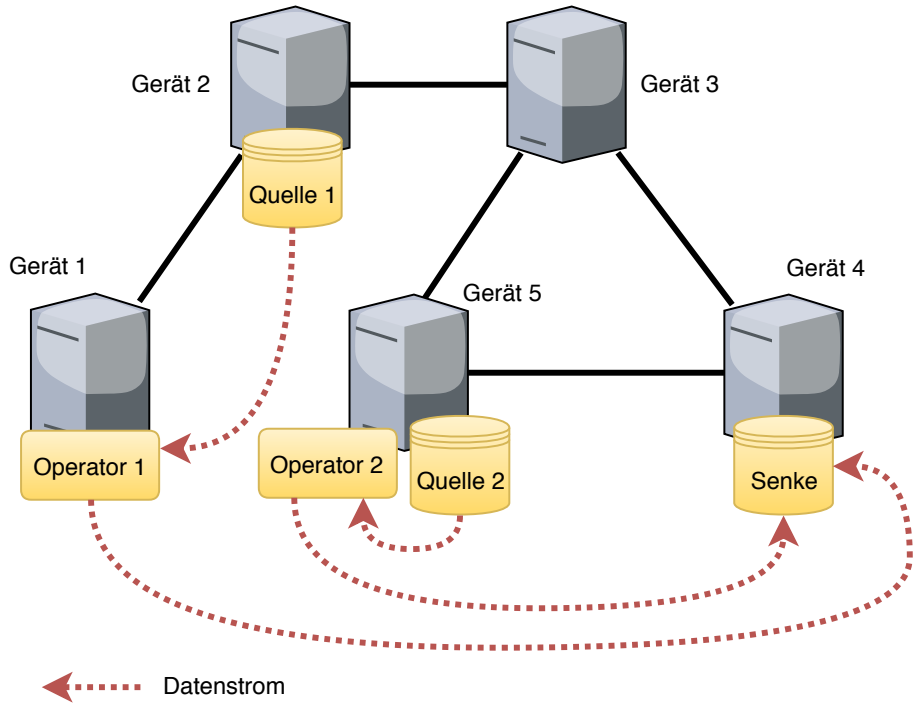

Abbildung 4.6: Mögliche Verteilung der Operatoren auf die IoT-Geräte durch die Verteilungsalgorithmen

In Abbildung 4.6 ist eine mögliche Verteilung der Operatoren auf die Geräte der IoT-Umgebung dargestellt. Die beiden Quellen werden wie vorgesehen auf Gerät 2 und Gerät 5 abgebildet, während Operator 1 auf Gerät 1, Operator 2 ebenfalls auf Gerät 2 und die Senke auf Gerät 4 projiziert werden. Der Datenstrom wird durch die gestrichelten roten Pfeile repräsentiert. Spätestens an dieser Stelle wird ersichtlich, dass es für die Lösung des Verteilungsproblems nicht ausreichend ist, nur die Operatoren des Datenstrommodells abzubilden. Stattdessen müssen auch die Verbindungen zwischen diesen Operatoren auf entsprechende Netzwerkpfade abgebildet werden, damit der Weg festgelegt werden kann, über den die Daten transportiert werden. Andernfalls gibt es in Abbildung 4.6 beispielsweise für den Datenstrom von Operator 2 zur Senke mehrere Möglichkeiten, wie die Daten von Gerät 5 zu Gerät 4 transportiert werden können: Entweder unter Ausnutzung der direkten Verbindung, oder aber über den Umweg unter Einbezug von Gerät 3. Je nach dem, welche Anforderungen diese Verbindung an den Netzwerkpfad stellt, ist es denkbar, dass nur eine der beiden Möglichkeiten oder gar keine verwendet werden kann. Aus diesen Gründen muss analog zu den Operatoren schon vor der tatsächlichen Ausführung des Datenstrommodells durch die Verteilungsalgorithmen entschieden werden, welche Netzwerkpfade zum Datentransport

eingesetzt werden. Die Aufgabe, sicherzustellen, dass die Netzwerkpfade während der Ausführung des Datenstrommodels dann auch wie vorgesehen zum Datentransport genutzt werden, ist Teil des Ausführungskonzepts, das später in dieser Arbeit vorgestellt wird. Offenkundig ist es zur korrekten Ausführung der Geschäftslogik jedoch erforderlich, dass alle im Datenstrommodell enthaltenen Verbindungen unter Aufrechterhaltung der Konsistenz des Datenstrommodells auf Netzwerkpfade der IoT-Umgebung abgebildet werden. Das entspricht der dritten Forderung.

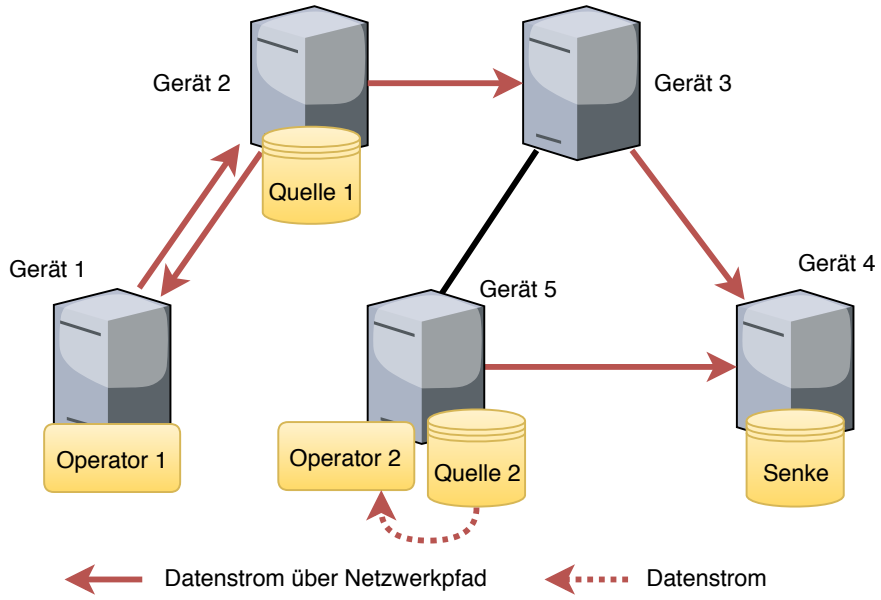

Abbildung 4.7: Mögliche Verteilung der Operatoren und Verbindungen auf die IoT-Umgebung

In Abbildung 4.7 wurde die Verteilung aus Abbildung 4.6 um eine Abbildung der Datenströme auf Netzwerkpfade erweitert. Es ist zu beachten, dass dies nicht die einzige mögliche Lösung für das Verteilungsproblem ist. Da in diesem Beispiel keine Anforderungen und Eigenschaften spezifiziert worden sind, können mit Ausnahme der Quellen alle Operatoren beliebig auf die Geräte verteilt werden und auch für die meisten Datenströme bestehen jeweils mehrere mögliche Netzwerkpfade, die den Datentransport sicherstellen können. Wurden bei den Operatoren oder den Verbindungen zwischen den Operatoren jedoch Anforderungen spezifiziert, dann müssen auch diese in Betracht gezogen werden. Die Verteilungsalgorithmen müssen dann sicherstellen, dass die ausgewählten IoT-Geräte und Netzwerkpfade die Anforderungen der Operatoren und Datenströme jeweils erfüllen können. Wie ein solcher Abgleich insbesondere unter Berücksichtigung des kumulativen Verhaltens konsumierend-wirkender Anforderungen erfolgen kann, wurde bereits in Abschnitt 4.2 ausgeführt. Dies entspricht der vierten und fünften Forderung.

Wenn es wie in diesem Beispiel mehrere Verteilungen gibt, bei denen die IoT-Umgebung jeweils alle Anforderungen des Datenstrommodells erfüllt, ist es erstrebenswert, eine möglichst geeignete Verteilung als Lösung für die jeweilige Probleminstanz auszuwählen. Wie bereits im Verlauf dieser Arbeit beschrieben wurde, ist eine Verteilung als umso besser zu erachten, je näher die Daten an den Quellen verarbeitet werden und je leistungsstärker die Netzwerkverbindungen zwischen jeweils zwei IoT-Geräten sind, über die die Daten transportiert werden. Durch die Einführung der Netzwerkdistanzen als Maß für die Qualität dieser Verbindungen wird es möglich, die Güte einer Lösung quantitativ zu erfassen. Folglich sind grundsätzlich solche Verteilungen anzustreben, bei

denen die Summe der Netzwerkdistanzen möglichst gering ist, die gemäß dem Datenstrommodell zum Datentransport überbrückt werden müssen. Dies entspricht der sechsten Forderung. Zu beachten ist allerdings, dass die Verteilung mit der geringsten Summe an Netzwerkdistanzen nicht zwingend für alle Anwendungsfälle die bestmögliche Lösung darstellt. So sind Szenarien denkbar, in denen es erforderlich ist, nach anderen Kriterien als der Netzwerkdistanz zwischen den involvierten Geräten zu optimieren, wie beispielsweise nach Ausfallsicherheit oder Auslastung der IoT-Geräte. Derartige nichtfunktionale Anforderungen können mit dem in dieser Bachelorarbeit vorgestellten Konzept nicht berücksichtigt werden.

Diese Beschreibung des Verteilungsproblems entspricht dem Teilziel 1.1 aus Abschnitt 3.2 und bildet die Grundlage für die im Anschluss vorgenommene Formalisierung.

4.4.2 Formalisierung

Für die Konzeption der Verteilungsalgorithmen ist es hilfreich, zuerst die Problemstellung einschließlich der erwarteten Ein- und Ausgaben der Algorithmen zu formalisieren. Auf diese Weise werden die bestehenden Strukturen und Beziehungen zwischen den Eingaben und der Lösung offengelegt und ein Rahmen geschaffen, in welchem die Algorithmen validiert und bewertet werden können. Nachfolgend werden die Eingaben, bestehend aus der Netzwerktopologie und dem Datenstrommodell, und die Lösung, die der Ausgabe der Algorithmen entspricht, formalisiert.

Eingaben

Im Folgenden sei R die Mengen aller Anforderungen („Requirements“) und C die Menge aller Eigenschaften („Capabilities“). Anforderungen und Eigenschaften selbst werden an dieser Stelle nicht näher formal definiert, sie beziehen sich jedoch auf die in Abschnitt 4.2 vorgestellten Konzepte. Dabei sei satisfies: $\mathcal{P}(R)\times\mathcal{P}(C) \rightarrow \{\text{wahr}, \text{falsch}\}$ eine auf den in demselben Abschnitt vorgestellten Algorithmen basierende Funktion, die unter Berücksichtigung konsumierender Anforderungen und entsprechender Anpassung der aktuellen Werte der Eigenschaften überprüft, ob eine Menge von Eigenschaften eine Menge von Anforderungen erfüllt. Auf dieser Grundlage wird zuerst die Netzwerktopologie formal eingeführt.

Definition 4.4.1 (Netzwerktopologie)
Eine Netzwerktopologie ist ein Tupel der Form $(D, L, \text{dist}, \text{cap})$ mit:

- $D \neq \emptyset$: Menge aller Geräte (engl.: „Devices“) der Netzwerktopologie
- $L \subseteq \binom{D}{2}$: Menge aller Netzwerkverbindungen (engl.: „Links“) zwischen den Geräten in D
- $\text{dist}: L \rightarrow \mathbb{R}$: Gewichtsfunktion, die den Netzwerkverbindungen ihre jeweilige Netzwerkdistanz zuweist
- $\text{cap}: D \cup L \rightarrow \mathcal{P}(C)$: Abbildung, die den Geräten und Netzwerkverbindungen jeweils eine Menge von Eigenschaften zuweist

sodass (D, L, dist) einen ungerichteten, gewichteten und schlingenfreien Graphen bildet.

Für derartige Netzwerktopologien kann der bereits zuvor eingeführte Begriff des Netzwerkpfads wie folgt formalisiert werden:

Definition 4.4.2 (Netzwerkpfad)
Sei $T := (D, L, \text{dist}, \text{cap})$ eine Netzwerktopologie. Ein Netzwerkpfad in dieser Netzwerktopologie ist eine endliche Folge von Netzwerkverbindungen $(d_0, d_1), (d_1, d_2), \ldots, (d_{n-1}, d_n)$ mit $n \in \mathbb{N}$ sodass gilt:

1. $\forall d_i \in \{d_0, d_1, \ldots, d_{n-1}\} : \{d_i, d_{i+1}\} \in L$
2. $\forall d_i, d_j \in \{d_0, d_1, \ldots, d_n\} : i \neq j \Rightarrow d_i \neq d_j$

Hierbei heißen n die *Länge*, $\text{start}(p) = d_0$ der *Start* und $\text{target}(p) = d_n$ das *Ziel* eines Netzwerkpfads p. Ein Tupel (d_i, d_{i+1}) mit $i \in \{0, 1, \ldots, n-1\}$ heißt *Abschnitt* des Netzwerkpfads. Die Menge aller Netzwerkpfade der Netzwerktopologie T wird mit $\text{PATHS}(T)$ bezeichnet.

Die erste Bedingung für einen Netzwerkpfad stellt sicher, dass alle im Netzwerkpfad vorkommenden Netzwerkverbindungen auch in der Netzwerktopologie eine entsprechende Verbindung besitzen. Aus der zweiten Bedingung folgt, dass ein Netzwerkpfad keine Zyklen oder Schleifen enthalten darf. Nun kann auch das Datenstrommodell definiert werden. Da in diesem spezifiziert wird, welche IoT-Geräte der Umgebung die Softwareelemente der Datenquellen provisionieren sollen, ist es von der Netzwerktopologie nicht vollständig unabhängig.

Definition 4.4.3 (Datenstrommodell)
Sei $(D, L, \text{dist}, \text{cap})$ eine Netzwerktopologie. Ein Datenstrommodell auf dieser Netzwerktopologie ist ein Tupel der Form $(O, S, E, \text{req}, \text{source})$ mit:

- $O \neq \emptyset$: Menge aller Operatoren des Datenstrommodells
- $S \subseteq O$: Menge aller Datenquellen (engl.: „Sources“) des Datenstrommodells
- $E \subseteq (O \times (O \setminus S))$: Menge der Verbindungen (engl.: „Edges“) zwischen den Operatoren in O
- $\text{req} : O \cup E \to \mathcal{P}(R)$: Abbildung, die den Operatoren und Verbindungen jeweils eine Menge von Anforderungen zuweist
- $\text{source} : S \to D$: Abbildung, die die Datenquellen des Datenstrommodells auf die jeweils dafür vorgesehenen Geräte der Netzwerktopologie abbildet

sodass (O, E) einen gerichteten, ungewichteten, schlingenfreien und azyklischen Graphen bildet.

Die Formalisierung der Netzwerktopologie und des Datenstrommodells erfüllt die erste Hälfte des Teilziels 1.2 aus Abschnitt 3.2.

Verteilungsproblem

Auf Basis der Netzwerktopologie und des Datenstrommodells kann nun auch das Verteilungsproblem, das die Verteilungsalgorithmen lösen, formal definiert werden.

Definition 4.4.4 (Verteilungsproblem)
Sei $T := (D, L, \text{dist}, \text{cap})$ eine Netzwerktopologie und $(O, S, E, \text{req}, \text{source})$ ein Datenstrommodell auf dieser Netzwerktopologie. Das *Verteilungsproblem* besteht dann darin, eine *Lösung* (device, path) mit den Abbildungen $\text{device}: O \to D$ und $\text{path}: E \to \text{PATHS}(T)$ zu finden. Sie heißt *gültig*, wenn gilt:

1. $\forall s \in S : \text{device}(s) = \text{source}(s)$
2. $\forall (o_1, o_2) \in E \text{ mit } p := \text{path}((o_1, o_2)) : \text{start}(p) = \text{device}(o_1)$
   $\wedge\ \text{target}(p) = \text{device}(o_2)$
3. $\forall o \in O : \text{satisfies}(\text{req}(o), \text{cap}(\text{device}(n)))$
4. $\forall e \in E \text{ mit } p := \text{path}(e) : \forall a \in p(\mathbb{N}) : \text{satisfies}(\text{req}(e), \text{cap}(a))$

Eine Lösung besteht also aus zwei Abbildungen: Eine, die die Operatoren auf Geräte der Netzwerktopologie abbildet und eine, die die Verbindungen zwischen diesen Operatoren auf Netzwerkpfade in der Netzwerktopologie abbildet. Die erste Forderung dieser Definition stellt sicher, dass die Abbildung der Datenquellen wie im Datenstrommodell definiert auch in der Abbildung der Operatoren der Lösung enthalten ist. Die zweite Forderung trägt dafür Sorge, dass die Struktur und damit die Konsistenz des Datenstrommodells in den Abbildungen erhalten bleibt: Das Gerät, auf das der Startoperator einer Verbindung im Datenstrommodell abgebildet wird, muss der Start des Netzwerkpfads sein, auf das die Verbindung selbst abgebildet wird, während das Gerät, auf das der Zieloperator der Verbindung abgebildet wird, das Ziel dieses Netzwerkpfads darstellen muss. Die Forderungen drei und vier sichern zu, dass die Geräte der Netzwerktopologie und die Netzwerkpfade jeweils die Anforderungen der Operatoren und der Verbindungen erfüllen, auf die sie abgebildet werden. Bei den Netzwerkpfaden müssen dafür alle Abschnitte, aus denen sich der Pfad zusammensetzt, für sich genommen die Anforderungen der Verbindung erfüllen.

Da die Lösungen des Verteilungsproblems die Ausgaben der Verteilungsalgorithmen darstellen, erfüllt Definition 4.4.4 die zweite Hälfte des Teilziels 1.2. Durch die Definition gültiger Lösungen wird darüber hinaus auch Teilziel 1.3 erreicht.

Ein Verteilungsalgorithmus entspricht damit einer Funktion, die ein Datenstrommodell und eine Netzwerktopologie auf eine solche Lösung abbilden. Bei mehreren existierenden Lösungen für eine Instanz des Verteilungsproblem soll es möglich sein, die am geeignetsten erscheinende auszuwählen. Dafür wird an dieser Stelle stufenweise das Maß der Netzwerkdistanz für Lösungen eingeführt.

Definition 4.4.5 (Netzwerkdistanz eines Netzwerkpfads)
Sei p ein Netzwerkpfad innerhalb der Netzwerktopologie $(D, L, \text{dist}, \text{cap})$. Dann berechnet sich die Netzwerkdistanz $\text{dist}(p)$ dieses Pfads wie folgt:

$$\text{dist}(p) = \sum_{\forall a \in p(\mathbb{N})} \text{dist}(a)$$

Die Netzwerkdistanz eines Netzwerkpfads ist also die Summe der Distanzen aller Abschnitte des Pfads gemäß der Gewichtsfunktion der Netzwerktopologie.

Definition 4.4.6 (Netzwerkdistanz einer Lösung)
Sei $s := (\text{device}, \text{path})$ eine Lösung für eine Instanz des Verteilungsproblems, das aus der Netzwerktopologie $(D, L, dist, cap)$ und dem Datenstrommodell $(O, S, E, \text{req}, \text{source})$ besteht. Dann berechnet sich die Netzwerkdistanz $\text{dist}(s)$ dieser Lösung wie folgt:

$$\text{dist}(s) = \sum_{\forall e \in E} \text{dist}(\text{path}(e))$$

Anhand dieses Maßes lässt sich abschließend entsprechend zu Teilziel 1.4 aus Abschnitt 3.2 definieren, was eine Lösung zu der bestmöglichen Lösung macht.

Definition 4.4.7 (Bestmögliche Lösung)
Sei S die Menge aller möglichen Lösungen für eine Instanz des Verteilungsproblems mit Netzwerktopologie T. Eine Lösung $s \in S$ heißt bestmögliche Lösung, falls gilt:

$$\forall t \in S : s \neq t \Rightarrow \text{dist}(t) \geq \text{dist}(s)$$

Diese Definition impliziert, dass es für eine Instanz des Verteilungsproblems prinzipiell mehrere bestmögliche Lösungen geben kann.

4.4.3 Greedy-Algorithmus

An dieser Stelle wird das Konzept des ersten Verteilungsalgorithmus vorgestellt. Dabei handelt es sich um einen Algorithmus der Klasse der Greedy-Algorithmen [CLRS01]. Diesen ist gemein, dass sie in jedem Teilschritt die jeweils lokal beste Lösung auswählen und auf dieser weiterarbeiten, ohne dabei den globalen Kontext des Problems zu betrachten. Das führt dazu, dass Greedy-Algorithmen in der Regel verhältnismäßig effizient sind, aber grundsätzlich nicht die optimale Lösung für Probleme berechnen [SS14].

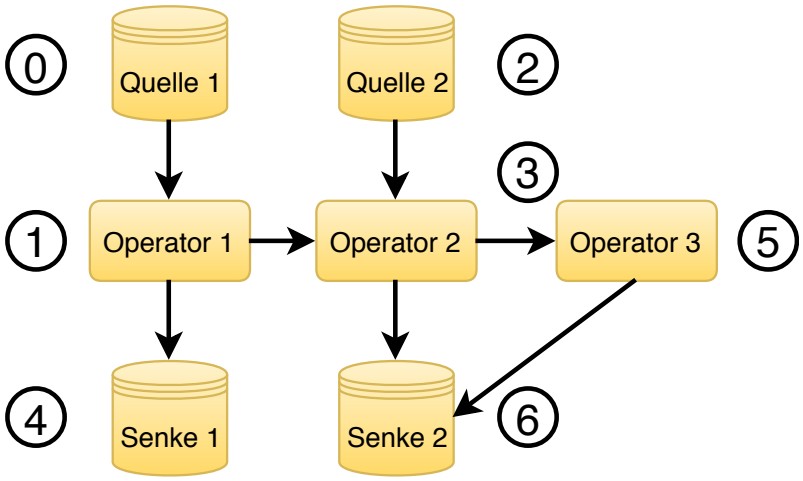

Abbildung 4.8: Topologische Sortierung auf einem Datenstrommodell

Der hier für das Verteilungsproblem vorgeschlagene Greedy-Algorithmus nutzt aus, dass das Datenstrommodell schlingenfrei und azyklisch ist. Dadurch wird es möglich, eine topologische Sortierung [CLRS01] auf dem Graphen des Datenstrommodells durchzuführen und die Operatoren somit unter Berücksichtigung ihrer Abhängigkeiten in eine logische Reihenfolge zu bringen. Dies ist in Abbildung 4.8 anhand eines Datenstrommodells exemplarisch dargestellt. Dabei geben die Zahlen in den Kreisen an, an welcher Position der topologischen Sortierung der jeweilige Operator steht. Die Sortierung ist nicht eindeutig, es gibt für dieses Beispiel mehrere Möglichkeiten die Operatoren anzuordnen. Dabei gilt jedoch immer die Invariante, dass jeder Operator in der Sortierung eine höhere Zahl erhält als alle seine Vorgängeroperatoren. So muss beispielsweise Operator 2 immer eine höhere Zahl besitzen als Operator 1 und Quelle 2, da dies seine direkten Vorgängeroperatoren sind, aber auch eine höhere als Quelle 1, da diese indirekt über Operator 1 auch ein Vorgängeroperator von Operator 2 ist.

Der Algorithmus betrachtet die Operatoren nun gemäß der aus der topologischen Sortierung erhaltenen Reihenfolge, beginnend mit dem ersten Operator. Handelt es sich bei diesem Operator um eine Datenquelle, ist für diesen bereits im Datenstrommodell hinterlegt, auf welches Gerät er abgebildet werden soll. Falls es sich allerdings nicht um eine Datenquelle handelt, muss für den Operator erst ein geeignetes Gerät der Netzwerktopologie gefunden werden. Durch Betrachtung der Operatoren in der Reihenfolge der topologische Sortierung ist durch die zuvor erwähnte Invariante implizit zugesichert, dass alle Vorgänger des Operators im Datenstrommodell bereits vor dem aktuellen Operator betrachtet worden sind und dass für diese folglich jeweils ein geeignetes Gerät zur Ausführung gefunden worden ist. Weil es sich bei diesem Operator nicht um eine Quelle handelt, muss der Operator zudem per Definition mindestens einen Vorgängeroperator besitzen. Diese Eigenschaft nutzt der Algorithmus aus: Um möglichst kurze Kommunikationswege zwischen den Operatoren zu schaffen, untersucht er die Geräte der Netzwerktopologie in der Reihenfolge des Betrags der Netzwerkdistanzen zu denjenigen Geräten, auf denen die Vorgängeroperatoren untergebracht worden sind. Das Gerät, das die geringste Netzwerkdistanz zu einem der Geräte der Vorgängeroperatoren aufweist, wird daher als erstes betrachtet und das Gerät mit der größten Distanz als letztes. Dabei werden die Geräte der direkten Vorgängeroperatoren selbst als erstes untersucht, da die Netzwerkdistanz hier in jedem Fall 0 beträgt. Bei jedem dieser Geräte wird geprüft, ob seine Eigenschaften die Anforderungen des Operators erfüllen. Ist dies nicht der Fall, wird das Gerät mit der nächsthöheren Netzwerkdistanz in Betracht gezogen. In Abbildung 4.9 ist das anhand des Datenstrommodells aus Abbildung 4.8 und einer Netzwerktopologie exemplarisch dargestellt: Quelle 1, Operator 1 und Quelle 2 wurden bereits auf Geräte der Netzwerktopologie abgebildet. Nun ist Operator 2 an der Reihe: Die direkten Vorgängeroperatoren sind Operator 1 und Quelle 2, daher werden die Geräte, auf die diese Operatoren verteilt worden sind, als erstes in Betracht gezogen; das sind in diesem Fall Gerät 1 und Gerät 5. Sollte sich keines von diesen als geeignet erweisen, werden anschließend diejenigen Geräte untersucht, die möglichst nahe an den Geräten der Vorgängeroperatoren liegen. Das ist zunächst Gerät 4 (Distanz 7 zu Quelle 2), dann Gerät 2 (Distanz 9 zu Operator 1) und schließlich Gerät 3 (Distanz 13 von Gerät 1 über Gerät 2). Diese Reihenfolge der Betrachtung der Geräte spiegelt das Prinzip von Greedy-Algorithmen wider: In den Teilschritten wird das jeweils situativ als am geeignetsten erscheinende Gerät ausgewählt und nicht nach dem Gerät gesucht, das zur besten Gesamtlösung führen würde.

Erfüllt das aktuell betrachtete Gerät die Anforderungen des Operators, ist es ein möglicher Kandidat, um den Operator auszuführen. Daraufhin wird geprüft, ob für jeden der in den Operator eingehenden Datenströme in der Netzwerktopologie ein Netzwerkpfad vom Gerät des Vorgängeroperators zu dem aktuell betrachteten Gerät existiert, dessen Eigenschaften den Anforderungen des Datenstroms

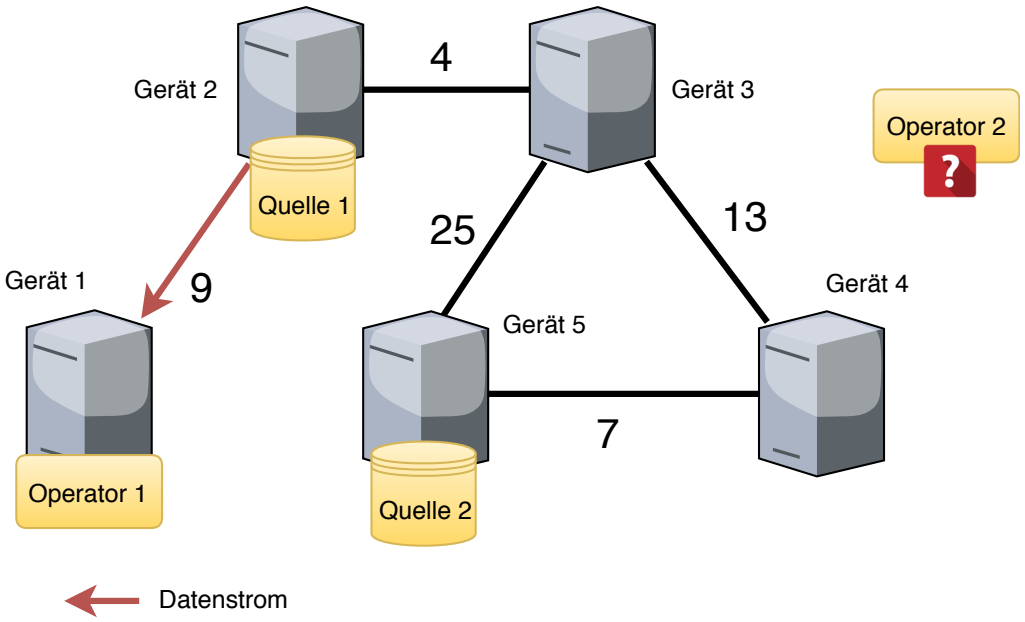

Abbildung 4.9: Beispiel zur Auswahl der Geräte durch den Greedy-Algorithmus

gerecht werden. Dabei werden getreu dem Prinzip von Greedy-Algorithmen bei mehreren existierenden Netzwerkpfaden diejenigen Pfade bevorzugt, die am kürzesten sind. An Abbildung 4.10 wird dies deutlich: Der Algorithmus prüft in diesem Beispiel, ob Operator 2 auf Gerät 4 ausgeführt werden kann. Der Operator selbst erfüllt die Bedingungen; das muss nun auch wie beschrieben für die Netzwerkpfade überprüft werden, die die Verbindung zwischen Quelle 2 und Operator 2 und zwischen Operator 1 und Operator 2 des Datenstrommodels repräsentieren. Für die Verbindung von Quelle 2 nach Operator 2 kommen zwei Netzwerkpfade in Frage: Die direkte Verbindung mit Netzwerkdistanz 7 und der Weg über Gerät 3 mit einer Gesamtdistanz von 38. Unter der Voraussetzung, dass beide Pfade die Anforderungen der Verbindung gemäß dem Datenstrommodell erfüllen, wählt der Greedy-Algorithmus den kürzeren Pfad, welcher der Direktverbindung entspricht. In dem Beispiel in Abbildung 4.10 gibt es auch für die Verbindung von Operator 1 nach Operator 2 zwei verschiedene Wege. Der Greedy-Algorithmus zieht hier ebenfalls, sofern möglich, den kürzeren Pfad vor, welcher von Gerät 1 über Gerät 2, Gerät 3 und schließlich zu Gerät 4 führt.

In dem Fall, dass für eine Verbindung kein zulässiger Netzwerkpfad existiert, verwirft der Algorithmus das aktuell betrachtete Gerät, macht die gegebenenfalls in diesem Teilschritt vorgenommenen Abbildungen und die Anpassungen der Eigenschaftswerte rückgängig und untersucht das nächste Gerät in der Reihenfolge. Gibt es für jedes Gerät der Vorgängeroperatoren einen erfüllenden Netzwerkpfad zu dem gerade betrachteten Gerät, bildet der Algorithmus den Operator auf dieses Gerät und auch die Datenströme auf ihre jeweiligen Netzwerkpfade ab. Anschließend werden die Werte der Eigenschaften des Geräts und die der Abschnitte der Netzwerkpfade auf die Werte der gegebenenfalls konsumierend-wirkenden Anforderungen angepasst. Im Anschluss wird gemäß der Reihenfolge der topologischen Sortierung der nächste Operator betrachtet und entsprechend abgebildet. Wurden schließlich alle Operatoren abgebildet, gibt der Algorithmus die Abbildung der Operatoren und die Abbildung der Datenströme als Lösung aus. Konnte jedoch für einen Operator kein geeignetes Gerät gefunden werden, bricht der Algorithmus ab und gibt aus, dass für die gegebene Probleminstanz keine Lösung berechnet werden konnte.

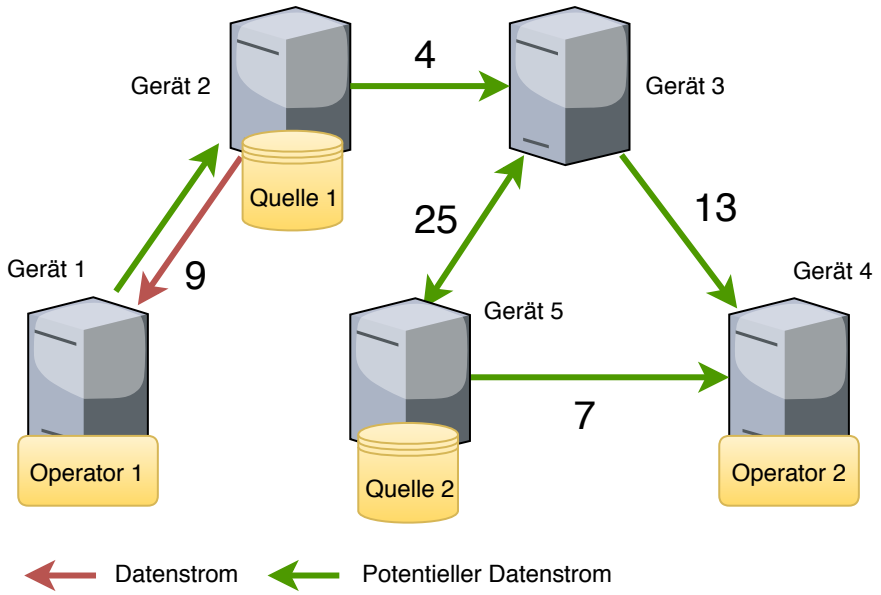

Abbildung 4.10: Beispiel zur Auswahl der Netzwerkpfade durch den Greedy-Algorithmus

Pseudocode

In Algorithmus 4.6 ist der Pseudocode des Greedy-Verteilungsalgorithmus dargestellt. Er verwendet die in Algorithmus 4.7 und Algorithmus 4.8 aufgeführten Hilfsfunktionen.

Ein Datenstrommodell und eine Netzwerktopologie stellen die Eingabe des Algorithmus dar. Zunächst werden in den Zeilen 7 und 8 die Abbildungen `device` und `path` definiert, die die noch zu erweiternde Lösung der Probleminstanz darstellen und am Ende des Algorithmus zurückgegeben werden. In Zeile 11 wird die Funktion GetTopologicalOrder aufgerufen, die auf Grundlage der Operatoren und Verbindungen des Datenstrommodells eine Abbildung order: $\{0, 1, \ldots, |O|\} \to O$ zurückgibt, die der topologischen Sortierung der Operatoren entspricht. Die Funktion, die die topologische Sortierung erzeugt, wird an dieser Stelle nicht explizit definiert. Da dem Datenstrommodell ein Graph zugrunde liegt, können dafür die in der Praxis üblichen Algorithmen zur topologischen Sortierung angewendet werden, wie Kahns Algorithmus [Kah62] oder eine modifizierte Tiefensuche.

Die Schleife, die ab Zeile 13 beginnt, iteriert gemäß der Reihenfolge der topologischen Sortierung über alle Operatoren. In Zeile 16 wird geprüft, ob es sich bei dem aktuellen Operator um eine Quelle handelt. Falls ja wird die Hilfsfunktion TryMapSource aufgerufen, die versucht, den Operator auf das Gerät abzubilden, auf das im Datenstrommodell für diese Quelle verwiesen wird. War dies erfolgreich, wird die Schleife mit dem nächsten Operator fortgesetzt. Falls nicht, kann für das gegebene Problem keine Lösung gefunden werden, da Datenquellen nur auf die im Datenstrommodell hinterlegten Geräte abgebildet werden dürfen.

Die Hilfsfunktion TryMapSource, die in Algorithmus 4.7 angegeben ist, holt die entsprechende Gerätereferenz aus dem Datenstrommodell und prüft in Zeile 6, ob die Eigenschaften dieses Geräts die Anforderungen des übergebenen Operators, der Datenquelle, erfüllen. Falls ja, wird der Operator

mittels der `device`-Abbildung der Lösung hinzugefügt und die aktuellen Werte der Eigenschaften mittels der in Abschnitt 4.2.5 definierten Prozedur ConsumeCapabilitySet den Anforderungen angepasst.

Handelt es sich bei dem betrachteten Operator um keine Datenquelle, wird Algorithmus 4.6 in Zeile 24 fortgesetzt. Dabei wird zunächst die Menge alle Vorgänger im Datenstrommodell ermittelt und nachfolgend die Menge aller Geräte, auf die diese Vorgänger bereits abgebildet worden sind. Mit der Funktion GetClosestDevices wird in Zeile 28 eine Sortierung aller Geräte der Netzwerktopologie hinsichtlich ihrer Entfernung zu den Geräten der Vorgänger vorgenommen und als Funktion closestOrder: $\{0, 1, \ldots, |D|\} \rightarrow D$ zurückgegeben. Dabei referenziert closestOrder(0) das Gerät, das den Vorgängergeräten am nächsten liegt, closestOrder(1) das Gerät, das ihnen am zweitnächsten ist usw. Die Vorgängergeräte selbst sind auch Teil dieser Sortierung und stehen, da ihre Netzwerkdistanz 0 beträgt, ganz am Anfang der Sortierung. Die Funktion GetClosestDevices selbst wird an dieser Stelle nicht näher spezifiziert. In der Praxis eignen sich zur Realisierung Algorithmen zur Bestimmung kürzester Pfade in Graphen, wie beispielsweise der Dijkstra-Algorithmus [Dij59].

Algorithmus 4.6 Greedy-Algorithmus zur Lösung des Verteilungsproblems

```
1:  // Eingabe: Datenstrommodell, Netzwerktopologie
2:  function GreedyDistribution((O, S, E, req, source), (D, L, dist, cap))
3:      M ← (O, S, E, req, source)
4:      T ← (D, L, dist, cap)
5:
6:      // Definiere Abbildungen für die Lösung
7:      device: O → D
8:      path: E → PATHS(T)
9:
10:     // Betrachte Operatoren in O in topologischer Reihenfolge
11:     order ← GetTopologicalOrder(O, E)
12:     i ← 0
13:     while i < |O| do
14:         operator ← order(i)
15:         // Ist aktueller Operator eine Quelle?
16:         if operator ∈ S then
17:             if TryMapSource(operator, device, M, T) then
18:                 continue
19:             else
20:                 return "Keine Lösung gefunden"
21:             end if
22:         end if
23:         // Operator ist keine Datenquelle, ermittle Menge der Vorgänger
24:         PreOperators ← {o ∈ O : ∃(o, operator) ∈ E}
25:         PreDevices ← device(PreOperators)
26:
27:         // Betrachte Geräte in der Reihenfolge der Entfernung zu den Vorgängern
28:         closestOrder ← GetClosestDevices(T, PreDevices)
29:
```

```
30:         foundDevice ← false
31:         j ← 0
32:         while j < |D| do
33:             opDevice ← closestOrder(j)
34:             if TryMapOperator(operator, opDevice, device, path, M, T) then
35:                 foundDevice ← true
36:                 break
37:             end if
38:         end while
39:
40:         // Wurde ein passendes Gerät gefunden?
41:         if ¬ foundDevice then
42:             return "Keine Lösung gefunden"
43:         end if
44:     end while
45:     return (device, path)
46: end function
```

In Zeile 30 wird eine Variable `foundDevice` eingeführt, die speichert, ob für den aktuell betrachteten Operator schon ein geeignetes Gerät zur Ausführung gefunden worden ist. Anschließend wird in Form der in Zeile 28 beginnenden Schleife in der zuvor festgelegten Reihenfolge über die nächstgelegenen Geräte iteriert. Für jedes dieser Geräte wird anschließend in Zeile 34 mit Hilfe der Funktion TryMapOperator versucht, den aktuellen Operator auf diesem Gerät zu platzieren. Gelingt dies für ein Gerät, wird die Variable `foundDevice` auf `true` gesetzt und die Schleife zum Suchen von Geräten abgebrochen. Wird die Schleife jedoch beendet, ohne dass ein passendes Gerät für den Operator gefunden wurde, wird in Form der Zeilen 41 und 42 signalisiert, dass für diese Probleminstanz keine Lösung berechnet werden konnte.

Innerhalb der Funktion TryMapOperator, die in Algorithmus 4.8 abgebildet ist, wird in Zeile 7 geprüft, ob das Gerät `opDevice`, auf dem der Operator `operator` probeweise platziert werden soll, die Anforderungen des Operators erfüllt. Falls das nicht der Fall ist, wird bereits an dieser Stelle

Algorithmus 4.7 Hilfsfunktion TryMapSource für den Greedy-Algorithmus

```
1:  // Eingabe: Operator, Abbildung für Operatoren, Datenstrommodell, Netzwerktopologie
2:  function TryMapSource(operator, device, (O, S, E, req, source), (D, L, dist, cap))
3:      // Verwende das im Datenstrommodell vorgesehene Gerät
4:      opDevice ← source(operator)
5:      // Erfüllt das Gerät die Anforderungen?
6:      if SatisfiesRequirementSet(req(operator), cap(opDevice)) then
7:          ConsumeCapabilitySet(req(operator), cap(opDevice))
8:          device(operator) ← opDevice
9:          return true
10:     else
11:         return false
12:     end if
13: end function
```

abgebrochen. Erfüllt der Operator die Anforderungen, geht es in Zeile 11 mit dem Algorithmus weiter. Dort wird erneut unter Verwendung der Prozedur ConsumeCapabilitySet dafür gesorgt, dass die Werte der Eigenschaften des Geräts `opDevice` an die möglicherweise konsumierend-wirkenden Anforderungen des Operators angepasst werden, wenn dieser auf dem Gerät platziert wird. In Zeile 12 wird der Operator mit dem Gerät `opDevice` formal der Abbildung für Operatoren hinzugefügt, die Teil der Lösung des Algorithmus ist. Dies erfolgt an dieser Stelle allerdings nur probeweise und kann im weiteren Verlauf dieser Funktion wieder revidiert werden.

In Zeile 15 werden erneut die Vorgänger des Operators `operator` ermittelt und gespeichert. Die Menge `PreOperators` enthält somit alle Operatoren, die gemäß dem Datenstrommodell Daten an `operator` übermitteln. Da bereits in Algorithmus 4.6 ausgeschlossen worden ist, dass `operator` eine Datenquelle ist, enthält diese Menge mindestens ein Element und ist somit nicht leer. In Zeile 16 wird eine Menge `mappedEdges` als leere Menge initialisiert. Sie speichert im weiteren Verlauf alle Verbindungen im Datenstrommodell, die im Rahmen dieser Funktion bereits abgebildet worden sind. Innerhalb der Schleife, die in Zeile 18 beginnt, wird einzeln über jeden der Vorgängeroperatoren von `operator` iteriert. In Zeile 20 wird für den aktuellen Vorgänger `preOperator` die Verbindung `edge` im Datenstrommodell ermittelt, die er mit `operator` gemein hat. Weiter wird in Zeile 21 das Gerät bestimmt, auf dem `preOperator` ausgeführt wird. Aufgrund der Betrachtung der Operatoren gemäß der topologischen Sortierung ist sichergestellt, dass alle Vorgängeroperatoren bereits auf Geräte abgebildet worden sind.

Unter Verwendung der Funktion FindShortestMatchingPath wird in Zeile 24 versucht, den kürzesten Netzwerkpfad in der Netzwerktopologie zu finden, der beim Gerät des aktuellen Vorgängeroperators beginnt und bei `opDevice`, dem für Operator `operator` vorgesehenen Gerät, endet. Diese Funktion stellt dabei implizit sicher, dass der Netzwerkpfad die Anforderungen der Verbindung `edge` erfüllt. Auf die Funktion FindShortestMatchingPath wird im Folgenden nicht näher eingegangen, denn auch sie kann unter Verwendung eines Algorithmus für kürzeste Pfade in Graphen verhältnismäßig einfach konstruiert werden. Allerdings muss dabei innerhalb dieses Algorithmus

Algorithmus 4.8 Hilfsfunktion TryMapOperator für den Greedy-Algorithmus

```
1:  // Eingabe: Operator, Gerät, Abbildung für Operatoren, Abbildung für Pfade,
2:  // Datenstrommodell, Netzwerktopologie
3:  function TryMapOperator(operator, opDevice, device, path, (O,S,E,req,source),
    (D,L,dist,cap))
4:      T ← (D,L,dist,cap)
5:
6:      // Erfüllt das Gerät die Anforderungen?
7:      if ¬ SatisfiesRequirementSet(req(operator), cap(opDevice)) then
8:          return false
9:      end if
10:
11:     ConsumeCapabilitySet(req(operator), cap(opDevice))
12:     device(operator) ← opDevice
13:
14:     // Ermittle Vorgängeroperatoren
15:     PreOperators ← {o ∈ O : ∃(o, operator) ∈ E}
```

```
16:     // Prüfe ob für alle Vorgänger erfüllende Netzwerkpfade existieren
17:     mappedEdges ← ∅
18:     for all preOperator ∈ PreOperators do
19:         // Ermittle eingehenden Datenstrom von diesem Vorgänger
20:         edge ← (preOperator, operator)
21:         preDevice ← device(preOperator)
22:
23:         // Versuche erfüllenden Netzwerkpfad zu finden
24:         netPath ← FindShortestMatchingPath(T, preDevice, opDevice, edge)
25:         if netPath = null then
26:             // Kein Pfad gefunden, mache Abbildungen rückgängig
27:             UndoConsumeCapabilitySet(req(operator), cap(opDevice))
28:             device(operator) ← null
29:             for all mapEdge ∈ mappedEdges do
30:                 UndoConsumeCapabilityPath(req(mapEdge), path(mapEdge))
31:                 path(mapEdge) ← null
32:             end for
33:             return false
34:         end if
35:
36:         // Bilde Datenstrom auf Netzwerkpfad ab
37:         ConsumeCapabilityPath(req(edge), netPath)
38:         path(edge) ← netPath
39:         mappedEdges ← mappedEdges ∪{edge}
40:     end for
41:     return true
42: end function
```

bei jedem Teilschritt, im Zuge dessen dem Netzwerkpfad ein weiterer Abschnitt hinzugefügt wird, überprüft werden, ob die Eigenschaften der neue Netzwerkverbindung die Anforderungen der Verbindung `edge` erfüllen. Ist das nicht der Fall, darf der Abschnitt nicht verwendet werden.

Wurde ein kürzester erfüllender Netzwerkpfad gefunden, so geht es in Zeile 37 weiter. Hier werden die aktuellen Werte der Eigenschaften der in dem Netzwerkpfad enthaltenen Abschnitte auf die gegebenenfalls konsumierend-wirkenden Anforderungen der Verbindung `edge` angepasst. Dafür wird eine Prozedur ConsumeCapabilityPath verwendet, die der Prozedur ConsumeCapabilitySet für Netzwerkpfade entspricht. Anschließend wird in den Zeilen 38 und 39 der Datenstrom `edge` auf diesen Netzwerkpfad als Teil der Lösung abgebildet und `edge` der Menge `mappedEdges` hinzugefügt, die für den jeweils aktuellen Operator alle bereits auf Netzwerkpfade abgebildeten Datenströme speichert.

Es ist andererseits auch möglich, dass für den Weg vom Gerät des Vorgängeroperators bis zu dem Gerät `opDevice` in Zeile 24 kein erfüllender Netzwerkpfad gefunden wird. Das wird in Zeile 25 geprüft: Ist das der Fall und der Rückgabewert der Funktion FindShortestMatchingPath dementsprechend `null`, dann müssen alle bereits vorgenommenen Abbildungen und Anpassungen von Eigenschaftswerten rückgängig gemacht werden. Das geschieht in den Zeilen 27 bis 32, unter Zuhilfenahme der Menge `mappedEdges`. Am Ende, in Zeile 33, ist der Ausgangszustand

wiederhergestellt, der vor Aufruf der Funktion TryMapOperator gegeben war. An dieser Stelle wird `false` zurückgegeben, um zu signalisieren, dass das Gerät `opDevice` für `operator` nicht verwendet werden kann.

Ganz zum Schluss, in Zeile 45 von Algorithmus 4.6, wird die Lösung, sofern eine gefunden werden konnte, ausgegeben. Sie setzt sich aus der Abbildung der Operatoren auf die Geräte der Netzwerktopologie und der Abbildung der Verbindungen es Datenstrommodells auf die Netzwerkpfade zusammen.

Eigenschaften

Der Greedy-Verteilungsalgorithmus bearbeitet das Verteilungsproblem bezogen auf die Größe des Datenstrommodells und der Netzwerktopologie in Polynomialzeit und ist damit prinzipiell auch für verhältnismäßig größere Probleminstanzen mit vielen Operatoren, Verbindungen, Geräten, Netzwerkverbindungen und jeweils zahlreichen Anforderungen und Eigenschaften geeignet. Gibt der Algorithmus eine Lösung aus, so ist diese gültig. Es existiert kein Fall, in dem der Algorithmus eine fehlerhafte Lösung ausgibt. Allerdings gibt es Probleminstanzen, in denen keine Lösung gefunden wird, obwohl eine solche existiert, weshalb dieser Algorithmus Teilziel 1.6 aus Abschnitt 3.2 nicht vollständig erfüllt. Ein Beispiel für eine solche Instanz ist in Abbildung 4.11 in Form eines Datenstrommodells und einer Netzwerktopologie dargestellt, welche mit Anforderungen beziehungsweise Eigenschaften annotiert sind. Die Speicheranforderungen von Operator 1 und Operator 2 wirken dabei konsumierend. Quelle 1 muss in diesem Beispiel zwingend auf Gerät 1 abgebildet werden, da nur dort die benötigten Daten verfügbar sind. Die einzige mögliche Lösung für diese Probleminstanz ist die folgende:

- Quelle 1 wird auf Gerät 1 abgebildet
- Operator 1 wird auf Gerät 3 abgebildet
- Operator 2 wird auf Gerät 2 abgebildet
- Die Verbindung zwischen Quelle 1 und Operator 1 wird auf den Pfad von Gerät 1 nach Gerät 3 abgebildet
- Die Verbindung zwischen Quelle 1 und Operator 2 wird auf den Pfad von Gerät 1 nach Gerät 4 abgebildet

Der Greedy-Algorithmus verhält sich hier jedoch anders: Zunächst platziert er ebenfalls Quelle 1 auf Gerät 1. Danach betrachtet er in Abhängigkeit von der topologischen Sortierung entweder Operator 1 oder Operator 2. Falls Operator 1 zuerst betrachtet wird, stellt der Algorithmus fest, dass dieser Operator aufgrund seiner Anforderungen nicht auf das Gerät des Vorgängers (Gerät 1) abgebildet werden kann. Dementsprechend untersucht der Algorithmus anschließend die nächstgelegenen Geräte. Da Gerät 2 näher an Gerät 1 liegt als Gerät 2 und es zudem die Anforderungen des Operators erfüllt, wählt er Gerät 2 für Operator 1 aus und bildet entsprechend auch die Verbindung aus dem Datenstrommodell auf den Netzwerkpfad zwischen Gerät 1 und Gerät 2 ab. Jetzt steht allerdings für den noch fehlenden Operator 2 kein Gerät mehr zur Verfügung, das die Anforderungen erfüllt: Auf Gerät 3 gibt es zwar noch ausreichend Speicher, aber keine Datenbank. Gerät 2 kann nicht verwendet werden, da aufgrund der konsumierend-wirkenden Anforderung von Operator 1, der bereits auf dieses Gerät abgebildet worden ist, nur noch 100 Megabyte an Speicher zur Verfügung

stehen. Für Gerät 1 wurden keine Eigenschaften spezifiziert, somit kann dieses auch nicht Operator 2 provisionieren, da nicht gesichert ist, dass hier ausreichend Speicherplatz zur Verfügung steht. In der Folge wird der Algorithmus an dieser Stelle ausgeben, dass keine Lösung für das Problem gefunden werden konnte, obwohl eine solche existiert. Gibt es für eine Probleminstanz allerdings keine Lösung, so wird dies in jedem Fall von dem Algorithmus erkannt und durch die Ausgabe angezeigt. Aus diesem Grund erfüllt er Teilziel 1.7.

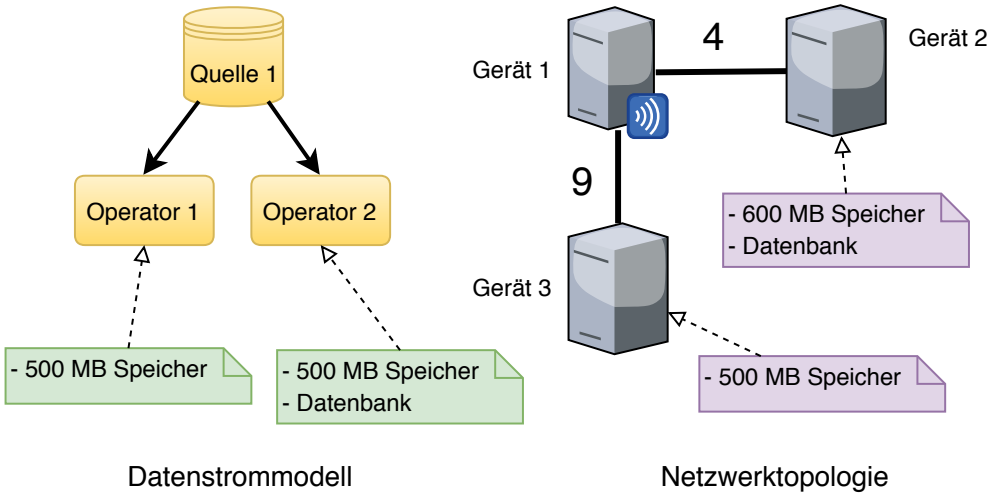

Abbildung 4.11: Probleminstanz, für die der Greedy-Algorithmus fälschlicherweise keine Lösung findet

Findet der Algorithmus eine Lösung, kann nicht davon ausgegangen werden, dass es sich dabei zwingend um die bestmögliche Lösung im Sinne von Definition 4.4.7 handelt. Der Algorithmus wählt die Geräte, auf denen die Operatoren ausgeführt werden sollen und die Netzwerkpfade, über die die Daten transportiert werden sollen jeweils „gierig“ nach der lokal im jeweiligen Teilschritt am geeignetsten erscheinenden Option aus und berücksichtigt dabei nicht, ob dies auch zu der global bestmöglichen Lösung führt. Damit lassen sich analog zu Abbildung 4.11 Probleminstanzen erstellen, in denen der Algorithmus in mindestens einem Teilschritt eine Entscheidung trifft, die das Finden der bestmögliche Lösung im weiteren Verlauf unmöglich macht. Aus diesem Grund kann der Algorithmus Teilziel 1.8 nicht erfüllen.

Zusammenfassend lässt sich festhalten, dass der Greedy-Algorithmus vor allem für solche Anwendungsfälle geeignet ist, in denen es potentiell viele mögliche Lösungen gibt, von denen nicht unbedingt die bestmögliche gefunden werden muss. Dabei dürfen die Datenstrommodelle und Netzwerktopologien auch größeren Umfangs sein. Weniger geeignet ist der Algorithmus für Situationen, in denen es nur sehr wenige mögliche Lösungen und viele Einschränkungen durch Anforderungen gibt, da es dabei passieren kann, dass keine Lösung gefunden wird. Darüber hinaus sollte er nicht für Systeme eingesetzt werden, in denen kurze Kommunikationswege besonders kritisch sind und jeweils die bestmögliche Lösung für das Verteilungsproblem benötigt wird.

4.4.4 Backtracking-Algorithmus

An dieser Stelle wird das Konzept des zweiten Verteilungsalgorithmus vorgestellt. Dabei handelt es sich um einen Algorithmus aus der Klasse der Backtracking-Algorithmen [SS14]. Dieses Algorithmenmuster realisiert eine systematische Suchtechnik, die einen vorgegebenen Lösungsraum vollständig bearbeitet und sich damit für Such- und Optimierungsprobleme eignet. Backtracking-Algorithmen finden zwar in der Regel die bestmögliche Lösung, sofern eine solche existiert, besitzen dafür aber eine hohe Laufzeitkomplexität [SS14].

Der Backtracking-Algorithmus, der an dieser Stelle für das Verteilungsproblem vorgeschlagen wird, besitzt zwei Modi: Im ersten wird die Suche abgebrochen, sobald eine Lösung gefunden worden ist, im zweiten wird die Suche fortgesetzt, bis alle möglichen Lösungen gefunden worden sind. Schließlich wird die beste der gefundenen Lösungen ausgegeben. Um dies zu bewerkstelligen, probiert der Algorithmus systematisch alle möglichen Abbildungen von Operatoren auf Geräte und von Datenströmen auf Netzwerkpfade aus. Je nach Umfang des Datenstrommodells und der Netzwerktopologie kann der Suchraum dafür sehr groß werden. Er ist aber in jedem Fall endlich, sodass eine Terminierung des Algorithmus nach endlicher Zeit garantiert ist.

Zuerst beginnt der Algorithmus damit, die Datenquellen gemäß der im Datenstrommodell hinterlegten Verweise auf die dafür vorgesehenen Geräte abzubilden. Sollte dies bei einer Datenquelle nicht gelingen, weil die Eigenschaften des entsprechenden Geräts nicht die Anforderungen erfüllen, existiert keine Lösung und daher bricht der Algorithmus die Berechnung ab. Im Folgenden kommen zwei Mengen zum Einsatz: Die eine enthält zu Beginn alle Operatoren, die keine Datenquellen sind, die andere Menge enthält alle Verbindungen zwischen den Operatoren des Datenstrommodells. In diesen Mengen wird gespeichert, welche Operatoren und Verbindungen noch auf Geräte beziehungsweise Netzwerkpfade abgebildet werden müssen. Sind beide Mengen leer, erkennt der Algorithmus, dass eine Lösung gefunden worden ist. Die weitere Arbeitsweise des Algorithmus lässt sich in Teilschritte unterteilen: In jedem dieser Teilschritte wird entweder ein Operator oder eine Verbindung abgebildet, wobei jeweils die dazugehörigen Anforderungen und Eigenschaften berücksichtigt werden.

Der Algorithmus beginnt mit den Operatoren: Er nimmt dafür zunächst einen beliebigen Operator aus der Menge der noch abzubildenden Operatoren heraus und vergleicht die Anforderungen nacheinander mit den Eigenschaften aller Geräte der Netzwerktopologie. Wurde ein Gerät gefunden, das die Anforderungen erfüllt, wird der Operator auf dieses abgebildet. Anschließend wird für den nächsten Operator der Operatormenge ein geeignetes Gerät gesucht. Erreicht der Algorithmus an einer Stelle eine sogenannte „Sackgasse“ in der er für einen Operator kein passendes Gerät finden kann, werden die bereits durchgeführten Teilschritte nacheinander wieder rückgängig gemacht und andere Geräte für die vorhergehenden Operatoren gesucht. Dies kann anhand des Beispiels aus Abbildung 4.11, für das der Greedy-Algorithmus keine Lösung findet, verdeutlicht werden. Der Ablauf des Backtracking-Algorithmus für diese Probleminstanz ist in Abbildung 4.12 dargestellt, wobei Quelle 1 bereits gemäß des Verweises im Datenstrommodell bereits auf Gerät 1 abgebildet wurde. Die weiteren Schritte des Algorithmus werden nachfolgend beschrieben.

Schritt 1: Am Anfang befinden sich mit Ausnahme von Quelle 1 alle Operatoren in der Menge der noch abzubildenden Operatoren. In diesem Schritt wird Operator 1 wird aus der Menge entnommen und es wird ein Gerät gesucht, das die Anforderungen des Operators erfüllt. Ein solches wird mit Gerät 2 gefunden. Folglich wird Operator 1 auf Gerät 2 abgebildet und die Werte der Eigenschaften von Gerät 2 entsprechend an die Anforderungen angepasst.

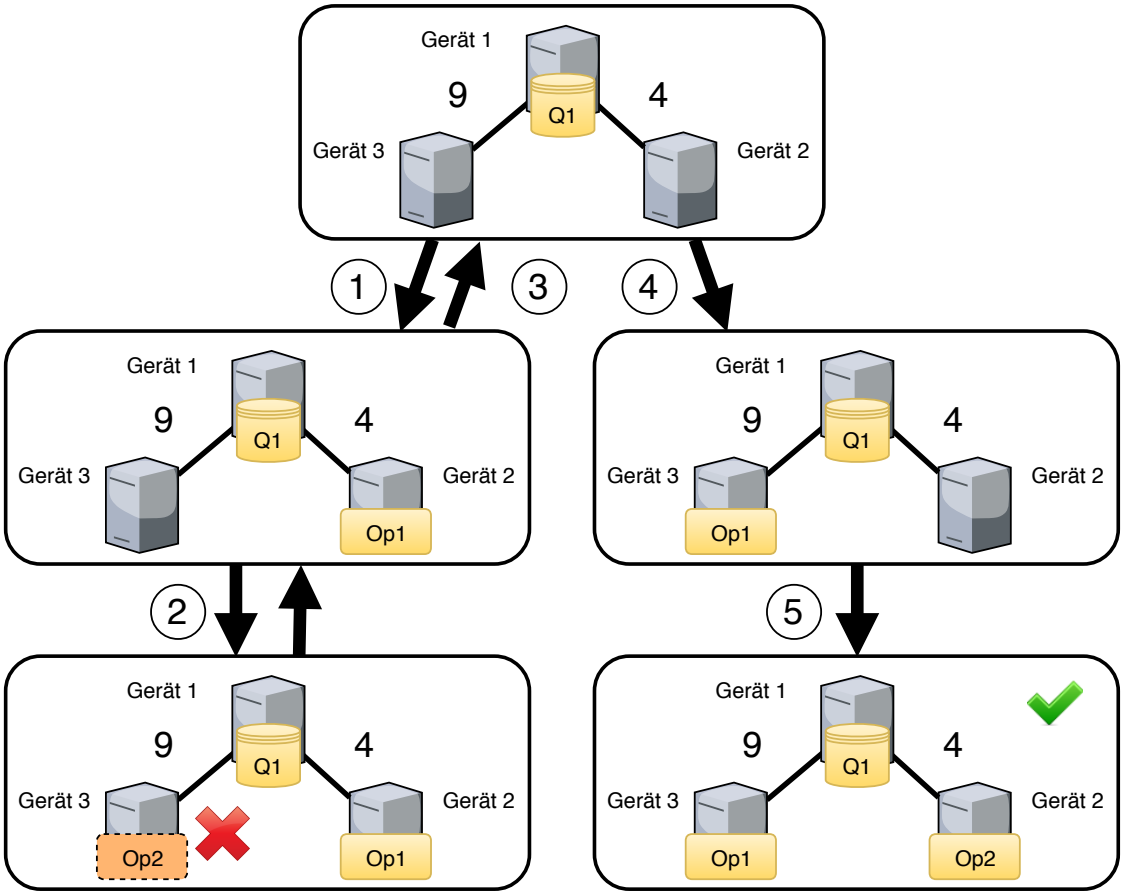

Abbildung 4.12: Schematischer Ablauf des Backtracking-Algorithmus anhand des Beispiels aus Abbildung 4.11

Schritt 2: Operator 2 wird aus der Menge der noch abzubildenden Operatoren entnommen und soll auf ein geeignetes Gerät abgebildet werden. Ein solches kann aber aufgrund der Anforderungen des Operators in keinem der drei Geräte der Netzwerktopologie gefunden werden. Deshalb wird Operator 2 der Operatormenge wieder hinzugefügt.

Schritt 3: Da sich der Algorithmus in einer Sackgasse befindet, wird Schritt 2 vollständig rückgängig gemacht und sowohl die Abbildung, als auch die Anpassung der Eigenschaften von Gerät 2 wieder aufgehoben.

Schritt 4: Da das Abbilden von Operator 1 auf Gerät zu der Sackgasse führte, in der der Algorithmus nicht fortgesetzt werden konnte, wird nun nach einem anderen geeigneten Gerät für Operator 1 gesucht, das in Gerät 3 gefunden wird. Folglich wird Operator 1 auf Gerät 3 abgebildet und die Werte der Eigenschaften angepasst.

Schritt 5: Jetzt kann in Gerät 2 ein passendes Gerät für Operator 2 gefunden werden, das die Anforderungen erfüllt. Der Operator wird auf der Operatormenge entnommen, auf dieses Gerät abgebildet und die Geräteeigenschaften werden wieder entsprechend auf die Anforderungen angepasst.

Während die Abbildungen von Operatoren auf Geräte im Allgemeinen im Laufe des Algorithmus wieder rückgängig gemacht werden können, gilt das nicht für die Datenquellen: Diese bereits zuvor gemäß dem Datenstrommodell vorgenommenen Zuordnungen bleiben während des gesamten Algorithmus erhalten. Gelingt es dem Algorithmus nicht, für alle Operatoren ein geeignetes Gerät zu finden, bricht er ab, sobald alle möglichen Verteilungskombinationen durchprobiert worden sind und vermeldet, dass keine Lösung gefunden werden konnte. Wird jedoch für jeden Operator ein Gerät gefunden, ist die Menge der noch abzubildenden Operatoren nach einer bestimmten Anzahl von Schritten leer. Ab diesem Punkt beginnt der Algorithmus damit, die Verbindungen zwischen den Operatoren auf ähnliche Weise auf Netzwerkpfade in der Netzwerktopologie abzubilden: In jedem Teilschritt entnimmt er eine Verbindung aus der Menge der noch abzubildenden Verbindungen, prüft, auf welche Geräte der Start- und der Zieloperator dieser Verbindung abgebildet worden sind und versucht dann, alle möglichen Netzwerkpfade zwischen diesen beiden Geräten zu finden, die die gegebenenfalls vorhandenen Anforderungen der jeweiligen Verbindung erfüllen. Aus diesen möglichen Netzwerkpfaden wird dann jeweils ein Pfad ausgewählt und der Abbildung der Verbindungen hinzugefügt. Wird für eine Verbindung kein erfüllender Netzwerkpfad gefunden, werden auch hier nacheinander die vorhergehenden Teilschritte wieder rückgängig gemacht und jeweils andere Netzwerkpfade für diese Verbindungen ausprobiert. Dabei können, bei Bedarf, auch Abbildungen von Operatoren auf Geräte wieder rückgängig gemacht und andere Verteilungen durchprobiert werden. Wurden alle Möglichkeiten durchlaufen, ohne dass eine Lösung gefunden werden konnte, terminiert der Algorithmus mit dem Ergebnis, dass keine Lösung existiert. Sind dagegen die beiden Mengen der noch abzubildenden Operatoren und Verbindungen leer, erkennt der Algorithmus, dass eine Lösung gefunden worden ist. Wenn nur eine beliebige Lösung berechnet werden sollte, wird diese nun ausgegeben und der Algorithmus terminiert. Soll dagegen die bestmögliche Lösung gefunden werden, fügt der Algorithmus die Lösung einer Lösungsmenge hinzu, macht den letzten Teilschritt wieder rückgängig und sucht nach weiteren Lösungen. Wurde schließlich auf diese Weise schließlich der komplette Lösungsraum durchsucht, wird die nach Definition 4.4.7 beste Lösung der Lösungsmenge ausgegeben.

Das Vorgehen des Algorithmus entspricht, wie für Backtracking-Algorithmen üblich, einer Tiefensuche in einem Baum. In Abbildung 4.13 ist das exemplarisch dargestellt: Die Ausgangssituation, die Instanz eines gegebenen Verteilungsproblems, bildet die Wurzel des Baums. Ausgehend davon wird der Baum dann in Form von Teillösungen um Kindknoten erweitert, indem zunächst die Operatoren und schließlich auch die Verbindungen zwischen den Operatoren Schritt für Schritt auf Geräte und Netzwerkpfade abgebildet werden. Jede Teillösung repräsentiert dabei einen Abbildungsschritt des Algorithmus. Geht es in einem Schritt nicht weiter, weil für einen Operator oder eine Verbindung kein passendes Gerät oder kein erfüllender Netzwerkpfad gefunden wird, wird im Baum über die Elternknoten wieder so weit nach oben gestiegen und die vorhergehenden Schritte rückgängig gemacht, bis eine andere Teillösung gefunden wird. Diese Teillösung wird dann ebenfalls als Kindknoten dem Baum hinzugefügt und verzweigt diesen damit weiter. Die weiteren Schritte des Algorithmus werden dann auf Grundlage dieses Knotens ausgeführt. Die Blätter dieses Baums bilden auf diese Weise die vollständigen Lösungen des Problems, bei denen alle Operatoren und alle Verbindungen des Datenstrommodels erfolgreich auf die Netzwerktopologie abgebildet werden konnten.

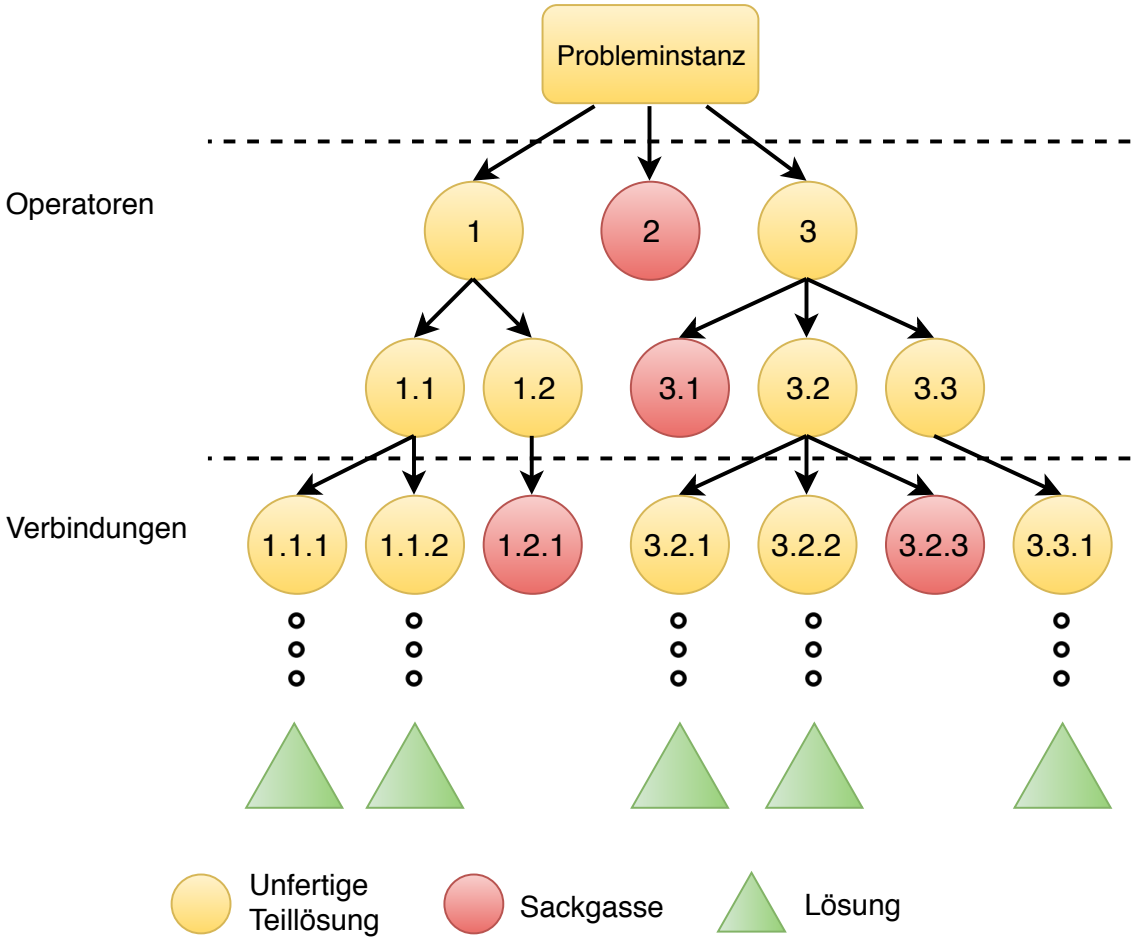

Abbildung 4.13: Beispielhafte Darstellung der Tiefensuche des Backtracking-Algorithmus

Pseudocode

In Algorithmus 4.9 ist der Pseudocode des Backtracking-Verteilungsalgorithmus dargestellt. Er verwendet die in Algorithmus 4.10, Algorithmus 4.11, Algorithmus 4.12 und Algorithmus 4.13 aufgeführten Hilfsfunktionen.

Die Eingabe des Backtracking-Algorithmus besteht aus einem booleschen Wert `best`, der angibt, ob die beste Lösung gesucht werden soll oder ob nach der ersten gefundenen Lösung abgebrochen werden darf, einem Datenstrommodell und einer Netzwerktopologie. In den Zeilen 7 bis 9 in Algorithmus 4.9 wird zunächst ein Tupel definiert, das mit der Abbildung von Operatoren und Geräten und der Abbildung von Verbindungen auf Netzwerkpfade die Lösung des Algorithmus darstellt. Auf diesem Lösungstupel wird im weiteren Verlauf des Algorithmus gearbeitet. Mit Hilfe der Schleife in Zeile 12 wird über alle Datenquellen des Datenstrommodells iteriert, um diese der Abbildung `device` und damit der Lösung hinzuzufügen. Dabei wird in Zeile 15 jeweils geprüft, ob die Geräte wirklich die Anforderungen der Datenquellen erfüllen. Sollte dies nicht der Fall sein, kann durch Zeile 19 signalisiert werden, dass keine Lösung gefunden werden kann, da die Verteilung auf eine geeignete Zuordnung der Datenquellen auf die Geräte durch das

Datenstrommodell angewiesen ist. Zusätzlich zu der Abbildung der Datenquellen werden mittels der Funktion ConsumeCapabilitySet aus Abschnitt 4.2.5 in Zeile 16 die Werte der Eigenschaften der Geräte entsprechend an die gegebenenfalls konsumierend-wirkenden Anforderungen angepasst.

In den Zeilen 23 und 24 werden die bereits im vorherigen Abschnitt angesprochenen Mengen definiert, in denen die Operatoren und Verbindungen gespeichert werden, die im weiteren Verlauf des Algorithmus noch abgebildet werden müssen. Die Lösungsmenge, in der alle gefundenen Lösungen gespeichert werden, wird in Zeile 27 definiert. In Zeile 29 beginnt mit dem Aufruf der Funktion FindSolution der Ablauf des eigentlichen Kerns des Backtracking-Algorithmus, der in Form rekursiver Funktionsaufrufe realisiert ist. Dabei werden jeweils der Parameter `best`, das zuvor definierte Lösungstupel, auf dem nun weitergearbeitet wird, die Operatormenge, die Verbindungsmenge, die Lösungsmenge, das Datenstrommodell und die Netzwerktopologie an die Funktionen übergeben. Bei Erreichen von Zeile 31 des Algorithmus 4.9 ist die rekursive Suche bereits abgeschlossen und es wird geprüft, ob währenddessen eine Lösung gefunden werden konnte. Falls nicht, wird in Zeile 32 eine entsprechende Meldung zurückgegeben, andernfalls wird in Zeile 36 mittels der Funktion GetBestSolution die beste der gefundenen Lösungen ermittelt und zurückgegeben.

Die in Algorithmus 4.10 abgebildete Funktion FindSolution bildet den Kern des Backtracking-Algorithmus und wird rekursiv aufgerufen, um die Tiefensuche nach Lösungen zu realisieren. Ab Zeile 9 wird geprüft, wie weit der Algorithmus bereits fortgeschritten ist und was als nächstes getan werden muss. Sind in der Operatormenge `Operators` noch Operatoren enthalten, müssen diese zunächst auf Geräte abgebildet werden. Dies geschieht durch einen Aufruf der Funktion FindSolutionOperator. Ist stattdessen die Operatormenge bereits leer, aber die Menge der abzu-

Algorithmus 4.9 Backtracking-Algorithmus zur Lösung des Verteilungsproblems

```
 1: // Eingabe: Beste Lösung? (Boolescher Wert), Datenstrommodell, Netzwerktopologie
 2: function BacktrackingDistribution(best, (O, S, E, req, source), (D, L, dist, cap))
 3:     M ← (O, S, E, req, source)
 4:     T ← (D, L, dist, cap)
 5:
 6:     // Definiere Abbildungen für die Lösung und Lösungstupel
 7:     device: O → D
 8:     path: E → PATHS(T)
 9:     s ← (device, path)
10:
11:     // Füge der Lösung alle Quellen gemäß source hinzu
12:     for all sourceOp ∈ S do
13:         opDevice ← source(sourceOp)
14:         // Erfüllt das Gerät die Anforderungen?
15:         if SatisfiesRequirementSet(req(sourceOp), cap(opDevice)) then
16:             ConsumeCapabilitySet(req(sourceOp), cap(opDevice))
17:             device(sourceOp) ← opDevice
18:         else
19:             return "Keine Lösung gefunden"
20:         end if
21:     end for
```

```
22:     // Mengen für die noch nicht abgebildeten Operatoren und Verbindungen
23:     Operators ← (O \ S)
24:     Edges ← E
25:
26:     // Menge für alle gefundenen Lösungen
27:     Solutions ← ∅
28:     // Beginne rekursive Suche nach Lösungen
29:     b ← FindSolution(best, s, Operators, Edges, Solutions, M, T)
30:
31:     if (¬b) or (Solutions = ∅) then
32:         return "Keine Lösung gefunden"
33:     end if
34:
35:     // Ausgabe der bestmöglichen Lösung aus Solutions
36:     return GetBestSolution(Solutions, E, dist)
37: end function
```

Algorithmus 4.10 Hilfsfunktion FindSolution für den Backtracking-Algorithmus

```
1:  // Eingabe: Beste Lösung (Boolescher Wert), Lösungstupel, Menge offener Operatoren,
2:  // Menge offener Verbindungen, Menge aller Lösungen, Datenstrommodell, Netzwerktopologie
3:  function FindSolution(best, (device, path), Operators, Edges, Solutions, (O, S, E, req, source),
    (D, L, dist, cap))
4:      s ← (device, path)
5:      M ← (O, S, E, req, source)
6:      T ← (D, L, dist, cap)
7:
8:      // Prüfe, was noch getan werden muss
9:      if Operators ≠ ∅ then
10:         return FindSolutionOperator(best, s, Operators, Edges, Solutions, M, T)
11:     else if Edges ≠ ∅ then
12:         return FindSolutionEdge(best, s, Operators, Edges, Solutions, M, T)
13:     else
14:         // Fertig, s stellt vollständige Lösung dar
15:         solutionCopy ← ShallowCopy(s)
16:         Solutions ← Solutions ∪ {solutionCopy}
17:         return true
18:     end if
19: end function
```

bildenden Verbindungen noch nicht, dann müssen die Verbindungen als nächstes auf Netzwerkpfade abgebildet werden. Dies geschieht in Zeile 12 durch einen Aufruf der Funktion FindSolutionEdge. In beiden Fällen wird das Ergebnis dieser Funktionen (`wahr` oder `falsch`) direkt zurückgegeben. `Wahr` signalisiert der aufrufenden Funktion, dass die Abbildung des nächsten Operators oder der nächsten Verbindung erfolgreich gewesen ist und auf Basis dieser Lösung im weiteren Verlauf eine gültige Lösung gefunden werden konnte, die in die Lösungsmenge aufgenommen werden kann. `Falsch` als Rückgabewert gibt dagegen an, dass entweder der letzte Operator oder die letzte Verbindung nicht

erfolgreich abgebildet werden konnten und es somit erforderlich ist, mindestens den vorherigen Abbildungsschritt wieder rückgängig zu machen. Sind sowohl die Operatormenge, als auch die Verbindungsmenge leer, dann wurde für alle Operatoren und alle Verbindungen eine geeignete Abbildung in der Netzwerktopologie gefunden und folglich ist in `s` eine vollständige Lösung für das Problem hinterlegt. In diesem Fall betritt die Funktion den Else-Zweig in Zeile 13. Die Lösung wird in Zeile 15 dann in Form eines Aufrufs der Funktion ShallowCopy in die neue Variable `solutionCopy` kopiert, die hinsichtlich Zeigerreferenzen unabhängig von dem in `s` gespeicherten Tupel ist, aber dieselben Werte und Abbildungen enthält. Das ist notwendig, damit im weiteren Verlauf des Algorithmus auf dem Lösungstupel `s` weitergearbeitet werden kann, ohne dass die bereits gespeicherten Lösungen implizit geändert werden. Auf die Funktion ShallowCopy wird an dieser Stelle nicht näher eingegangen, sie kann aber in den meisten Programmiersprachen mit bereits dafür vorgesehenen Bordmitteln implementiert werden. Die Lösung in `solutionCopy` wird in Zeile 16 schließlich der Lösungsmenge hinzugefügt. Da eine Lösung gefunden worden ist, wird `wahr` zurückgegeben.

Die Hauptaufgabe der Funktion FindSolutionOperator aus Algorithmus 4.11 ist es, einen Operator aus der Menge `Operator` zu entnehmen und auf ein geeignetes Gerät abzubilden. In Zeile 8 dieser Funktion wird die Variable `foundSolution` definiert, die speichert, ob bereits eine Lösung gefunden werden konnte. In Zeile 9 wird mittels der Funktion GetOneElement ein Operator der Menge `Operators` entnommen und in `nextOperator` gespeichert. Diese Funktion wird an dieser Stelle nicht näher betrachtet. Wichtig ist allerdings, dass sie deterministisch arbeitet und bei mehreren Aufrufen auf derselben Menge immer dasselbe Element der Menge zurückgibt. Im Folgenden werden Geräte gesucht, die sich zur Ausführung des Operators `nextOperator` eignen. Die Schleife, die in Zeile 11 beginnt, iteriert dafür über alle Geräte der Netzwerktopologie. Auch hier ist es wichtig, dass die Reihenfolge, in der die Geräte betrachtet werden, bei jeder Ausführung dieselbe ist. Andernfalls ist es nicht mehr möglich, Abbildungsschritte später wieder rückgängig zu machen und nach Alternativen zu suchen. In Zeile 12 wird geprüft, ob das aktuell betrachtete Gerät `currentDevice` die Anforderungen des Operators `nextOperator` erfüllt. Ist dem nicht so, wird im Zuge der Schleife das nächste Gerät betrachtet. Ansonsten wird in den Zeilen 14 bis 16 die Abbildung des Operators auf das Gerät vorgenommen, die Werte der Eigenschaften des Geräts angepasst und der Operator aus der Menge der noch abzubildenden Operatoren entfernt. Danach erfolgt der rekursive Aufruf der Funktion FindSolution, um die Tiefensuche auf der aktuellen Lösung fortzusetzen. Wird bei diesem Aufruf `wahr` zurückgegeben, bedeutet dies, dass auf Grundlage der an dieser Stelle vorgenommenen Abbildung im weiteren Verlauf eine Lösung gefunden werden konnte. Dementsprechend wird

Algorithmus 4.11 Hilfsfunktion FindSolutionOperator für den Backtracking-Algorithmus

```
1: // Eingabe: Beste Lösung (Boolescher Wert), Lösungstupel, Menge offener Operatoren,
2: // Menge offener Verbindungen, Menge aller Lösungen, Datenstrommodell, Netzwerktopologie
3: function FindSolutionOperator(best, (device, path), Operators, Edges, Solutions,
   (O, S, E, req, source), (D, L, dist, cap))
4:     s ← (device, path)
5:     M ← (O, S, E, req, source)
6:     T ← (D, L, dist, cap)
7:
8:     foundSolution ← false
9:     nextOperator ← GetOneElement(Operators)
```

```
10:     // Betrache alle Geräte der Netzwerktopologie einzeln
11:     for all currentDevice ∈ D do
12:         if SatisfiesRequirementSet(req(nextOperator), cap(currentDevice)) then
13:             // Füge Abbildung auf das Gerät der Lösung hinzu
14:             ConsumeCapabilitySet(req(nextOperator), cap(currentDevice))
15:             device(nextOperator) ← currentDevice
16:             Operators ← Operators \ {nextOperator}
17:
18:             // Rekursiver Aufruf um die nächste Teillösung zu finden
19:             if FindSolution(best, s, Operators, Edges, Solutions, M, T) then
20:                 foundSolution ← true
21:                 if ¬best then
22:                     return true
23:                 end if
24:             end if
25:
26:             // Keine Lösung gefunden, mache Abbildung rückgängig
27:             UndoConsumeCapabilitySet(req(nextOperator), cap(currentDevice))
28:             device(nextOperator) ← null
29:             Operators ← Operators ∪ {nextOperator}
30:         end if
31:     end for
32:     return foundSolution
33: end function
```

`foundSolution` auf `wahr` gesetzt. Wurde beim Aufruf des Backtracking-Algorithmus der Parameter `best` dagegen auf `falsch` gesetzt, dann ist auch eine beliebige gültige Lösung ausreichend und es muss nicht die bestmögliche Lösung gefunden werden. In diesem Fall bricht der Algorithmus gemäß den Zeilen 21 und 22 an dieser Stelle ab. Soll jedoch die bestmögliche Lösung gefunden werden, fährt der Algorithmus in Zeile 27 fort. Dort geht es auch weiter, wenn der Aufruf der Funktion FindSolution `falsch` zurückgibt, was gleichbedeutend damit ist, dass auf Grundlage der Abbildung des Operators `nextOperator` auf das Gerät `currentDevice` im Zuge des rekursiven Aufrufs keine Lösung gefunden werden konnte. In beiden Fällen muss der letzte Abbildungsschritt rückgängig gemacht werden, entweder um weitere Lösungen zu finden, oder um aus der erreichten Sackgasse herauszukommen. Zunächst wird dafür in Zeile 27 der Konsum der Eigenschaften des betreffenden Geräts durch die konsumierend-wirkenden Anforderungen rückgängig gemacht. Dies geschieht durch die Funktion UndoConsumeCapabilitySet, wie sie in Abschnitt 4.2.5 eingeführt worden ist. In Zeile 28 wird danach die Abbildung des Operators auf das Gerät rückgängig gemacht und in Zeile 29 wird der Operator wieder der Menge noch abzubildener Operatoren hinzugefügt. Danach wird die umgebende Schleife fotzgesetzt, sofern es noch Geräte gibt, die ausprobiert werden können. Am Ende der Funktion wird in Zeile 32 der Inhalt der Variablen `FoundSolution` ausgegeben, die speichert, ob eine Lösung gefunden wurde oder nicht. Durch diesen Rückgabewert erfährt die aufrufende Funktion wiederum, ob eine Lösung gefunden werden konnte und nun in die Lösungsmenge aufgenommen werden kann, oder ob weitere Abbildungsschritte rückgängig gemacht werden müssen.

Die Funktion FindSolutionEdge, die in Algorithmus 4.12 angegeben ist, verfolgt dieselbe Aufgabe wie FindSolutionOperator, allerdings für Verbindungen anstelle von Operatoren. Das Grundgerüst bleibt dabei jedoch dasselbe. In Zeile 8 wird wieder eine Variable `foundSolution` eingeführt, die speichert, ob eine Lösung gefunden wurde und in den Zeilen 9 und 10 wird ein Element `edge` aus der Menge der noch abzubildenden Verbindungen `Edges` entnommen. In Zeile 13 werden durch die Funktion FindMatchingPaths alle möglichen Pfade auf der Netzwerktopologie ermittelt und in der Menge `Paths` gespeichert, die von dem Gerät, auf das der Startoperator der Verbindung `edge` zuvor abgebildet worden ist, zu dem Gerät führen, auf das der Zieloperator der Verbindung `edge` abgebildet worden ist. Da im Backtracking-Algorithmus zuerst die Operatoren auf Netzwerkgeräte abgebildet werden, bevor die Verbindungen betrachtet werden, ist zu diesem Zeitpunkt garantiert bekannt, auf welchen Geräten der Start- und der Zieloperator der betrachteten Verbindung gemäß der Abbildungen des Lösungstupels untergebracht werden sollen. Die verwendete Funktion FindMatchingPaths funktioniert analog zu der im Rahmen des Greedy-Algorithmus vorgestellten Funktion FindShortestMatchingPath mit der Ausnahme, dass bei dieser Funktion die Netzwerkdistanzen nicht betrachtet werden und nicht ein, sondern alle erfüllenden Pfade als Menge ausgegeben werden. Auf die Realisierung der Funktion wird an dieser Stelle nicht weiter eingegangen. Sie kann allerdings parallel zu den Vorschlägen für FindShortestMatchingPath in Abschnitt 4.4.3 realisiert werden, wenn nicht nur der kürzeste der gefundenen Pfade ausgegeben wird, sondern alle. In Zeile 14 wird geprüft, ob überhaupt erfüllende Netzwerkpfade gefunden werden konnten. Falls dem nicht so ist, bricht die Funktion an dieser Stelle mit `false` ab, was der jeweils aufrufenden Funktion signalisiert, dass eine Sackgasse erreicht wurde und mindestens eine der vorhergehenden Verteilungen aus diesem Grund rückgängig gemacht werden muss. Mit der Schleife, die in Zeile 18 beginnt, werden alle gefundenen Pfade nun einzeln betrachtet und jeweils in der Variablen `currentPath` gespeichert. Auch hier ist es wichtig, dass die Reihenfolge, in der die Pfade betrachtet werden, bei jedem Aufruf dieselbe ist. Da bereits durch die Funktion FindMatchingPaths zugesichert wurde, dass die betrachteten Netzwerkpfade die Anforderungen von `edge` erfüllen, kann die Verbindung mittels der Zeilen 20 bis 22 direkt auf den Pfad in `currentPath` abgebildet

Algorithmus 4.12 Hilfsfunktion FindSolutionEdge für den Backtracking-Algorithmus

```
1:  // Eingabe: Beste Lösung (Boolescher Wert), Lösungstupel, Menge offener Operatoren,
2:  // Menge offener Verbindungen, Menge aller Lösungen, Datenstrommodell, Netzwerktopologie
3:  function FindSolutionEdge(best, (device, path), Operators, Edges, Solutions,
    (O, S, E, req, source), (D, L, dist, cap))
4:      s ← (device, path)
5:      M ← (O, S, E, req, source)
6:      T ← (D, L, dist, cap)
7:
8:      foundSolution ← false
9:      (sourceOp, targetOp) ← GetOneElement(Edges)
10:     edge ← (sourceOp, targetOp)
11:
12:     // Ermittle alle passenden Pfade für diese Verbindung
13:     Paths ← FindMatchingPaths(device(sourceOp), device(targetOp), edge)
14:     if Paths = ∅ then
15:         return false
16:     end if
```

```
17:     // Betrachte alle gefundenen Pfade einzeln
18:     for all currentPath ∈ Paths do
19:         // Füge Abbildung auf den Pfad der Lösung hinzu
20:         ConsumeCapabilityPath(req(edge), currentPath)
21:         path(edge) ← currentPath
22:         Edges ← Edges \ {edge}
23:
24:         // Rekursiver Aufruf um die nächste Teillösung zu finden
25:         if FindSolution(best, s, Operators, Edges, Solutions, M, T) then
26:             foundSolution ← true
27:             if ¬best then
28:                 return true
29:             end if
30:         end if
31:
32:         // Keine Lösung gefunden, mache Abbildung rückgängig
33:         UndoConsumeCapabilityPath(req(edge), currentPath)
34:         path(edge) ← null
35:         Edges ← Edges ∪ {edge}
36:     end for
37:     return foundSolution
38: end function
```

werden. Dabei werden die Eigenschaften der Abschnitte des Netzwerkpfads mit der bereits aus dem Greedy-Algorithmus bekannten Funktion ConsumeCapabilityPath angepasst und die Verbindung aus der Menge der noch abzubildenden Mengen entfernt. Analog zu Algorithmus 4.11 wird anschließend in den Zeilen 25 bis 30 die Funktion FindSolution rekursiv aufgerufen, um die Tiefensuche auf der aktuellen Lösung fortzusetzen. Die Abbildung der Verbindung `edge` auf den Netzwerkpfad `currentPath` wird in In den Zeilen 33 bis 35 wieder rückgängig gemacht, sodass im Falle einer Sackgasse oder wenn die bestmögliche Lösung gefunden werden soll, die Schleife mit einem anderen Netzwerkpfad fortfahren kann.

Die Hilfsfunktion GetBestSolution, die in Algorithmus 4.13 dargestellt ist, gibt unter Eingabe der Lösungsmenge, der Menge der Verbindungen des Datenstrommodells und der Distanzfunktion der Netzwerktopologie diejenige Lösung der Menge aus, die gemäß Definition 4.4.7 die beste ist. Zunächst werden in den Zeilen 4 und 5 zwei Variablen eingeführt, die die zum jeweiligen Zeitpunkt beste bekannte Lösung und ihren Distanzwert speichern. Innerhalb der Schleife, die in Zeile 8 beginnt, wird anschließend jede Lösung aus der Lösungsmenge einzeln betrachtet und in der Variablen `s` gespeichert. In den Zeilen 11 bis 14 wird über alle Verbindungen des Datenstrommodells iteriert und dabei für jede Verbindung der Netzwerkpfad ermittelt, auf den die jeweilige Verbindung gemäß der Lösung `s` abgebildet wird. Über die Funktion `dist` der Netzwerktopologie wird dann die Netzwerkdistanz des aktuellen Pfads `path` gemäß Definition 4.4.5 berechnet und auf die Variable `distanceSum` addiert. In den Zeilen 17 bis 20 wird geprüft, ob die gesamte Netzwerkdistanz der Lösung `s` kleiner ist als die der bisher besten bekannten Lösung. Falls ja, werden die Variablen `bestSolution` und `bestDistance` entsprechend aktualisiert. Am Ende der Funktion wird in Zeile 22 die Lösung ausgegeben, die in `bestSolution` hinterlegt ist, was dann der besten der in `Solutions` enthaltenen Lösungen entspricht.

Algorithmus 4.13 Hilfsfunktion GetBestSolution für den Backtracking-Algorithmus

```
1:  // Eingabe: Menge gefundener Lösungen, Menge aller Verbindungen, Distanzfunktion
2:  function GetBestSolution(Solutions, E, dist)
3:      // Variablen für die beste bisher betrachtete Lösung
4:      bestSolution ← null
5:      bestDistance ← ∞
6:
7:      // Betrachte alle Lösungen einzeln
8:      for all s ∈ Solutions mit s ← (device, path) do
9:          distanceSum ← 0
10:         // Betrachte alle Verbindungen und addiere die Distanzen der Pfade
11:         for all edge ∈ E do
12:             path ← path(edge)
13:             distanceSum ← distanceSum + dist(path)
14:         end for
15:
16:         // Vergleiche Distanz mit bester bisher bekannter Lösung
17:         if distanceSum < bestDistance then
18:             bestSolution ← s
19:             bestDistance ← distanceSum
20:         end if
21:     end for
22:     return bestSolution
23: end function
```

Eigenschaften

Im Gegensatz zum Greedy-Algorithmus bearbeitet der Backtracking-Algorithmus das Verteilungsproblem bezogen auf die Größe des Datenstrommodells und der Netzwerktopologie nicht in Polynomialzeit, sondern in Exponentialzeit. Dies wird insbesondere durch die benötigte Tiefensuche und die damit verbundenen rekursiven Funktionsaufrufe verursacht. In der Folge eignet sich der Backtracking-Algorithmus grundsätzlich nur für kleinere Probleminstanzen, große Datenstrommodelle und Netzwerktopologien mit zahlreichen Operatoren und Geräten können dagegen verhältnismäßig lange Rechenzeiten zur Folge haben. Es muss jedoch zwischen den beiden Modi, die der Backtracking-Algorithmus zur Verfügung stellt, unterschieden werden: Soll die bestmögliche Lösung gefunden werden, muss der Algorithmus in jedem Fall den vollständigen Lösungsraum durchsuchen, was grundsätzlich immer zu langen Rechenzeiten führt. Wenn jedoch auch eine beliebige Lösung ausreichend ist, kann die Suche je nach Aufbau, Umfang und Komplexität des Datenstrommodells und der Netzwerktopologie schnell durch den Fund einer Lösung beendet werden. Existiert dagegen keine Lösung, durchsucht der Algorithmus den Lösungsraum auch in diesem Modus vollständig.

Unabhängig von dem Modus, in dem der Backtracking-Algorithmus ausgeführt wird, findet er eine gültige Lösung genau dann, wenn eine solche existiert. Das erfüllt Teilziel 1.6 aus Abschnitt 3.2. Existiert keine Lösung, erkennt der Algorithmus dies in jedem Fall und signalisiert das entsprechend durch seine Ausgabe, wodurch Teilziel 1.7 erreicht wird. Damit unterscheidet er sich vom Greedy-

Algorithmus, der unter Umständen keine Lösung findet, selbst wenn es eine solche gibt. Existiert eine Lösung, so wird der Algorithmus eine der bestmöglichen Lösungen finden, wenn er in dem dafür vorgesehenen Modus ausgeführt wird und nicht terminiert, sobald eine Lösung gefunden worden ist. Damit wird auch Teilziel 1.8 erfüllt. Wenn jedoch eine beliebige Lösung für die jeweilige Instanz des Verteilungsproblems ausreichend ist, so wird er nur in Ausnahme- und Sonderfällen die bestmögliche Lösung finden. Im Durchschnitt ist die Lösung dann schlechter als die, die der Greedy-Algorithmus findet, da im Backtracking-Algorithmus im Gegensatz zum Greedy-Algorithmus keine Optimierungen hinsichtlich der Netzwerkdistanzen vorgenommen werden.

4.5 Konzept der Ausführung

In diesem Abschnitt werden Konzepte für die Ausführung von Datenstrommodellen in der IoT-Umgebung vorgestellt. Als Grundlage dafür dienen die Verteilungsentscheidungen, die von den in Abschnitt 4.4 beschriebenen Algorithmen für gegebene Verteilungsprobleme getroffen werden. Diese Verteilungen bestehen aus einer Abbildung der Operatoren auf Netzwerkgeräte und einer Abbildung der Datenströme auf Netzwerkpfade. Damit das Datenstrommodell gemäß dieser Verteilung in der IoT-Umgebung ausgeführt werden kann, müssen Konzepte für verschiedene Teilprobleme entwickelt werden:

1. **Zuordnung zwischen Verteilungen und IoT-Umgebungen:** Die von den Verteilungsalgorithmen erzeugten Verteilungen verweisen auf Geräte der IoT-Umgebung. Damit die Operatoren des Datenstrommodells so ausgebracht werden können, dass die Konsistenz des Datenstrommodells in der Ausführung erhalten bleibt, muss eine Zuordnung zwischen diesen Verweisen und den physischen Geräten der Umgebung geschaffen werden.

2. **Zuordnung zwischen Operatoren und Softwareelementen:** Die Operatoren werden in Form von Softwareelementen auf die Geräte der IoT-Umgebung ausgebracht. Dafür ist es notwendig, dass die Prozedur, die die Ausbringung durchführt, Zugriff auf die Elemente aller Operatoren hat und diese den jeweiligen Operatoren zuordnen kann.

3. **Ausbringung der Softwareelemente auf Geräte:** Die Softwareelemente müssen auf die Geräte kopiert werden, auf die die Operatoren gemäß der Verteilung ausgebracht werden sollen.

4. **Installation und Ausführung:** Nach Ausbringung der Softwareelemente müssen diese auf dem Gerät installiert und ausgeführt werden.

5. **Terminierung der Ausführung und Deinstallation:** Für den Fall, dass die Ausführung des Datenstrommodells abgebrochen werden oder eine Umverteilung der Operatoren durchgeführt werden soll, muss es möglich sein, die Ausführung der Operatoren auf den Geräten zu terminieren und die Softwareelemente wieder zu deinstallieren.

6. **Direkte Kommunikation zwischen Geräten:** Für die Ausführung des Datenstrommodells ist es erforderlich, dass Geräte der IoT-Umgebung, die über eine gemeinsame Netzwerkverbindung direkt miteinander verbunden sind, kommunizieren und Daten austauschen können.

7. **Kommunikation über Netzwerkpfade:** Die direkte Kommunikation zweier Geräte über eine gemeinsame Verbindung ist für die korrekte Ausführung des Datenstrommodells nicht ausreichend: Die Netzwerkpfade, auf die die Datenströme des Datenstrommodells abgebildet

werden, können auch mehr als zwei Geräteverbindungen umfassen, denn die Start- und Zielgeräte eines Netzwerkpfads müssen in einer IoT-Umgebung nicht notwendigerweise direkt miteinander verbunden sein. Deshalb wird ein Kommunikationskonzept benötigt, das den Datenaustausch zwischen Geräten über beliebig lange Netzwerkpfade und mehrere involvierte Geräte hinweg ermöglicht.

In den nachfolgenden Abschnitten wird zunächst die Rolle der MBP erörtert. Im Anschluss wird eine Kommunikationstechnologie bestimmt, die zur Realisierung der Kommunikation zwischen den Geräten innerhalb der IoT-Umgebung geeignet ist. Abschließend wird ein Konzept vorgestellt, das auf dieser Technologie aufbaut und es den Operatoren in der Ausführung ermöglicht, Daten getreu dem Datenstrommodell auszutauschen.

4.5.1 Evaluation der Rolle der MBP

In diesem Abschnitt wird gemäß dem Teilziel 2.1 aus Abschnitt 3.2 der Funktionsumfang der MBP untersucht, um festzustellen, für welche Problemstellungen sie als Teil der Lösung eingesetzt werden kann.

Die MBP bietet Funktionen an, um Geräte innerhalb einer IoT-Umgebung zu verwalten. Dafür hält sie ein digitales Abbild ihrer Umgebung, in dem die verfügbaren IoT-Geräte aufgeführt sind. Die Erstellung und Konfiguration des digitalen Abbilds muss in der aktuellen Version der MBP manuell vorgenommen werden, was entweder über eine Weboberfläche oder eine REST-Schnittstelle erfolgen kann. Die REST-Schnittstelle ermöglicht dabei eine automatisierte Erstellung dieses Abbilds: So kann die Prozedur, die die Ausführung des Datenstrommodells initiiert, über entsprechende HTTP[3]-Anfragen dem digitalen Abbild selbstständig Geräte hinzufügen, diese umkonfigurieren und auch wieder entfernen. Das Abbild der MBP umfasst allerdings nur die für die MBP erforderlichen Informationen wie Gerätenamen und IP-Adressen und ist weniger detailliert als das Modell, das in Abschnitt 4.3.2 beschrieben wurde. So werden auf Seiten der MBP beispielsweise weder Informationen über die Infrastruktur der Geräte gespeichert, noch werden Beziehungen zwischen verschiedenen Geräten beschrieben. Dennoch eignet sich diese digitale Abbildung, um getreu der Beschreibung des ersten der zuvor aufgeführten Teileprobleme eine Zuordnung zwischen den in den Verteilungen der Algorithmen referenzierten Elemente und der physischen IoT-Umgebung vorzunehmen. Sobald ein Verteilungsalgorithmus eine gültige Verteilung für ein gegebenes Datenstrommodell und ein Modell der Netzwerktopologie berechnet hat, können die darin referenzierten Geräte, auf denen Operatoren ausgeführt werden sollen, dem digitalen Abbild der MBP hinzugefügt werden. Auch die Informationen über die Geräte, die wie in Abschnitt 4.3.2 beschrieben bereits in dem Modell der Netzwerktopologie enthalten sind, wie IP-Adressen, können so in die Geräte des digitalen Abbilds eingetragen werden. In Abbildung 4.14 ist dies schematisch dargestellt: In (A) erhält die Prozedur, die dafür zuständig ist, das Datenstrommodell in der IoT-Umgebung zur Ausführung zu bringen, die von den Algorithmen vorgeschlagene Verteilung als Eingabe. Über die REST-Schnittstelle kann sie die in dieser Verteilung enthaltenen Geräte in (B) als digitale Abbilder der MBP anlegen, zusammen mit den für die MBP erforderlichen Geräteinformationen. Auf Grundlage dieser Abbilder kann die MBP die Geräte der IoT-Umgebung den Abbildern und damit auch den in der Verteilung referenzierten Geräten zuordnen.

[3] Hypertext Transfer Protocol

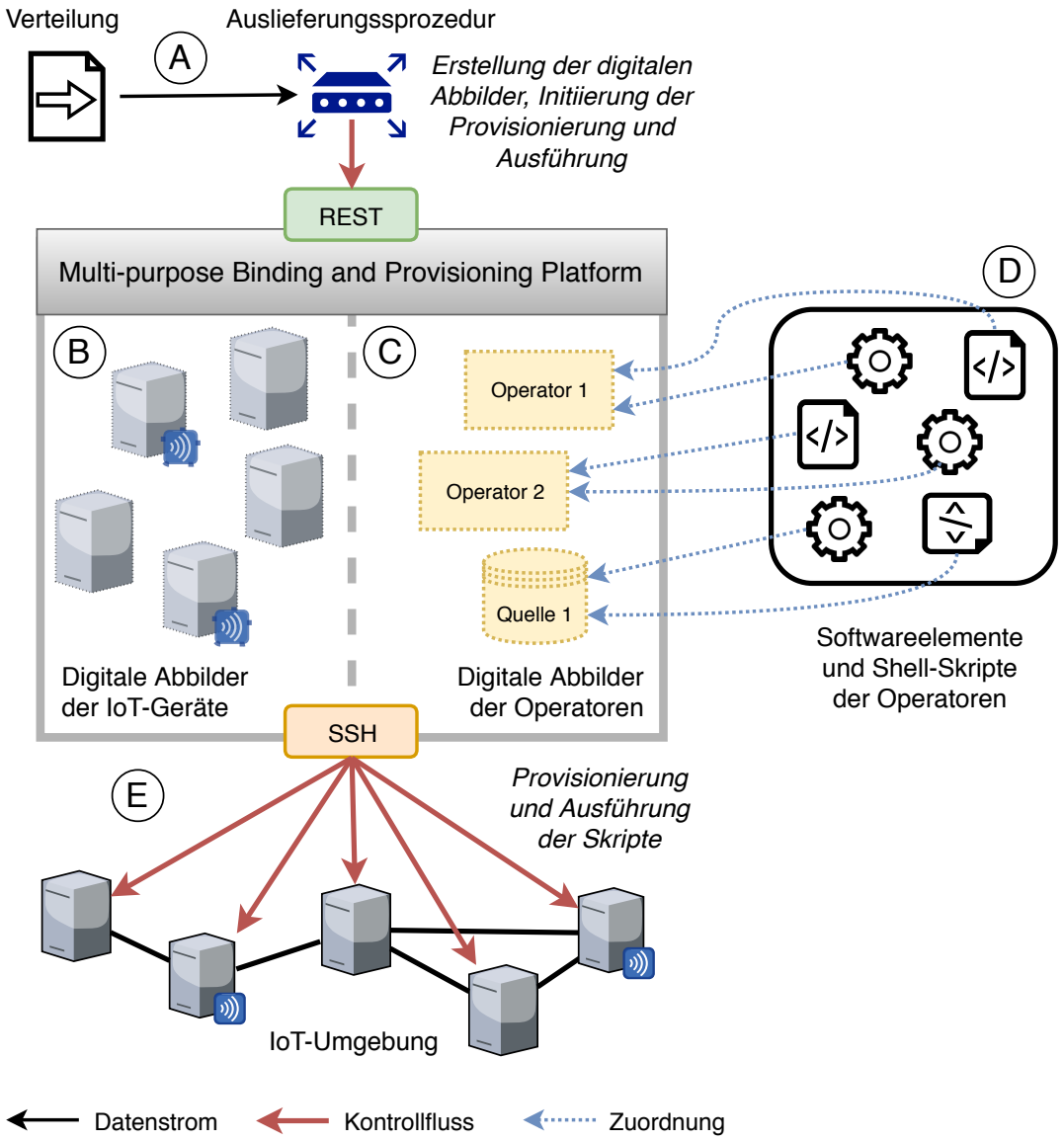

Abbildung 4.14: Übersicht über die Rolle der MBP innerhalb der Ausführung

Die MBP erlaubt es außerdem, digitale Abbilder von Operatoren zu definieren: Hierfür werden in Form von sogenannten *Adaptern* Verweise auf die Softwareelemente hinterlegt, die zur Ausführung des Operators erforderlich sind und auf dem Gerät, auf dem der Operator ausgeführt werden soll, installiert werden müssen. Teil dieser Adapter sind neben der eigentlichen Software auch Shell-Skripte mit festgelegten Dateinamen, die die Installation, den Start, die Terminierung und die Deinstallation dieser Softwareelemente übernehmen können. So ist beispielsweise das Skript mit dem Dateinamen `install.sh` dafür zuständig, die Installation vorzunehmen, während das Skript `stop.sh` die Ausführung abbrechen kann. Die Adapter erlauben auf diese Weise die Zuordnung zwischen Operatoren und Softwareelementen gemäß dem zweiten Teilproblem. Dies ist in Abbildung 4.14 an Stelle (C) dargestellt: Die Auslieferungsprozedur erstellt unter Verwendung der REST-Schnittstelle digitale Abbilder der Operatoren auf der MBP und ordnet diesen die in (D)

dargestellten Dateien zu, zu denen neben den eigentlichen Softwareelementen zur Ausführung der Operatoren auch die entsprechenden Skripte gehören. Bevor diese Dateien allerdings auf der MBP zur Verfügung stehen, müssen sie ihr zunächst hinzugefügt werden, was auch in diesem Fall durch die Auslieferungsprozedur über die REST-Schnittstelle erfolgt.

Die Ansteuerung der in dem digitalen Abbild spezifizierten IoT-Geräte durch die MBP erfolgt über das SSH-Protokoll[4], die dafür notwendigen Informationen wie IP-Adressen und Benutzernamen sind in den Abbildern der jeweiligen Geräte enthalten. Die MBP ist in der Lage, die Dateien, die zu einem Adapter zusammengefasst worden sind, auf ein Gerät der IoT-Umgebung zu kopieren und den Operator auf diese Weise zu provisionieren. Da diese Aktionen ebenfalls über die REST-Schnittstelle der MBP und damit auch automatisiert veranlasst werden können, entspricht dies einer Lösung für das dritte Teilproblem. Über das SSH-Protokoll kann die MBP auch die mitausgelieferten Skripte aufrufen, was es ermöglicht, die zuvor auf das Gerät kopierten Softwareelemente zu installieren und auszuführen. Das es sich um gewöhnliche Shell-Skripte handelt, können mit diesen auf den Geräten auch weitere Softwaremodule bezogen und installiert werden, bevor die eigentliche Software des Operators in Betrieb genommen wird. Benötigt die Software eines Operators beispielsweise eine Java-Umgebung, dann kann diese mit Hilfe des Installationsskripts eingerichtet werden. Bei Bedarf kann die Ausführung eines Operators mit Hilfe dieser Skripte auch wieder angehalten werden und auch die Deinstallation der Softwareelemente ist möglich. Auf diese Weise bietet die MBP Lösungen für das vierte und fünfte Teilproblem an. In Abbildung 4.14 kann dies an Stelle (E) eingesehen werden: Die Auslieferungsprozedur veranlasst über die REST-Schnittstelle, dass die MBP die Softwareelemente aus (D) auf die jeweils dafür vorgesehenen Geräte der IoT-Umgebung kopiert. Die Zuordnung erfolgt dabei durch die digitalen Abbilder der Geräte und der Operatoren. So kann die Auslieferungsprozedur veranlassen, dass ein bestimmter Operator mit einem bestimmten Gerät provisioniert wird. Um die Konsistenz des Datenstrommodells in der Ausführung sicherzustellen, hält sich die Ausführungsprozedur dabei an die Abbildungen, die in der Verteilung vorgesehen sind. Im Anschluss an die Provisionierung führt die MBP die mitausgelieferten Shell-Skripte aus, die die Installation und die Ausführung des Operators initiieren. Wenn die Auslieferungsprozedur die Provisionierung rückgängig macht, beispielsweise, weil eine andere Verteilung des Datenstrommodells vorgenommen werden soll, kann sie die Terminierung und Deinstallation der Operatoren auf vergleichbare Weise über die REST-Schnittstelle der MBP bewirken.

Die MBP stellt außerdem die Möglichkeit zur Verfügung, Sensorwerte aufzuzeichnen und in Form von Diagrammen zu visualisieren. Dafür müssen die gemessenen Werte von den Softwareelementen, die für die Operatoren auf die Netzwerkgeräte ausgeliefert werden, in einem bestimmten Format an einen Nachrichten-Broker der MBP übermittelt werden. Diese Funktion suggeriert eine Lösung für die Teilprobleme 6 und 7: Wenn die Daten aller Sensoren und die Ausgaben der Operatoren auf diese Weise an die MBP übermittelt werden, kann die Kommunikation und die Verteilung der Daten auf die Operatoren, die diese benötigen, prinzipiell auch durch eine entsprechende architektonische Erweiterung der MBP realisiert werden. Ein derartiges Konzept bringt jedoch eine Vielzahl von Problemen mit sich: Dadurch, dass alle Daten zu der MBP und von dort wieder zu den Operatoren auf den IoT-Geräten transportiert werden müssen, entsteht eine hohe Netzwerkauslastung, die das Konzept der Datenverarbeitung in der Nähe der Datenquellen ad absurdum führt. Außerdem bildet die MBP in diesem Fall einen Single Point auf Failure, da bei einem Ausfall der MBP

[4] Secure Shell, Netzwerkprotokoll für verschlüsselte Verbindungen

zwangsläufig die gesamte Kommunikation innerhalb des Systems zum Erliegen kommt. Eine solche Schwachstelle soll aber gemäß Herausforderung 4 vermieden werden. Insofern müssen andere, von der MBP unabhängige Lösungen entworfen werden, um die Kommunikation zwischen den Geräten der IoT-Umgebung zu realisieren. Entsprechende Konzepte dafür werden in den nachfolgenden Abschnitten vorgestellt.

4.5.2 Kommunikationstechnologie

Message Queue Telemetry Transport (MQTT) [BG14] ist ein Nachrichtenprotokoll für die Kommunikation zwischen Endgeräten („Klienten“), das seit 2013 von der Organization for the Advancement of Structured Information Standards (OASIS) als Protokoll für das IoT standardisiert wird [Adv13]. Es basiert auf dem TCP/IP-Protokoll und ermöglicht es, Nachrichten gemäß dem Observer-Entwurfsmuster [Joc13] zu versenden und zu empfangen. MQTT wurde als besonders leichtgewichtiges Protokoll entworfen, das auch auf Geräten mit leistungsschwacher Hardware und in Netzwerken mit geringer Bandbreite und hohen Latenzzeiten eingesetzt werden kann. Um es in der Praxis anwenden zu können, wird mindestens ein Broker benötigt, der Nachrichten von Klienten entgegennimmt, hält und bei Bedarf an andere Klienten weiterleitet. Nachrichten werden dabei mit einem Topic versehen, das die Nachrichten hierarchisch in ihren Kontext einordnet. Die Klienten können diese Topics abonnieren und bekommen dann von dem Broker diejenigen Nachrichten anderer Klienten weitergeleitet, deren Topic dem Abonnement entspricht [BG14].

MQTT wird im Rahmen dieser Bachelorarbeit eingesetzt, um den Datenaustausch zwischen den Geräten einer IoT-Umgebung entlang der durch die Verteilungsalgorithmen vorgeschlagenen Netzwerkpfade zu ermöglichen. Durch die Verwendung von Topics zur hierarchischen Kategorisierung von Nachrichten ermöglicht MQTT die einzelne Adressierung von Operatoren. Das wird insbesondere in dem Fall benötigt, wenn mehrere Operatoren auf demselben Gerät ausgeführt werden: Ohne Klassifizierung der Nachrichten ist unklar, welche der an dem Gerät eingehenden Nachrichten für welchen Operator bestimmt sind. Mit Hilfe von Topics und der eindeutigen Benennung von Datenströmen kann dieses Problem gelöst werden: Jeder Operator abonniert die Topics der Datenströme, deren Daten er für seine Ausführung benötigt. Als Namensformat für die Datenströme werden an dieser Stelle die Namen des Start- und Zieloperators verwendet. Ein Datenstrom, der von dem Operator mit dem Namen `Quelle1` zu dem Operator mit dem Namen `Filter1` geleitet wird, erhält somit den Namen `Quelle1/Filter1`. Der Operator `Filter1` kann nun auf diese Daten zugreifen, indem er das Topic mit diesem Namen beim Broker abonniert.

Abbildung 4.15 verdeutlicht die Funktionsweise anhand eines Beispiels: Oben in der Grafik ist ein Datenstrommodell zu sehen, das aus zwei Quellen und zwei Operatoren besteht. Die Verbindungen zwischen den Operatoren sind mit den Namen nach dem zuvor vorgestellten Schema bezeichnet. Das Datenstrommodell wird nun in einer beliebig strukturierten IoT-Umgebung ausgeführt, die aus drei Geräten besteht; die Operatoren werden dabei so auf die Geräte verteilt, wie es im unteren Bereich der Abbildung dargestellt ist. Die Operatoren `OP1` und `OP2` abonnieren bei dem Nachrichtenbroker jeweils die Topics der Datenströme, die sie für ihre Ausführung benötigen: `OP1` abonniert dementsprechend das Topic `Quelle1/OP1` und `OP2` die Topics `Quelle1/OP2` und `Quelle2/OP2`. Die Datenquellen `Quelle1` und `Quelle2` publizieren ihrerseits die Daten, die sie erhalten, als Nachrichten unter diesen Topics, sodass der Broker sie an die Operatoren auf Gerät 3 weiterleiten kann. Dadurch, dass die eingehenden Datenströme der beiden Operatoren auf Gerät 3 jeweils unterschiedliche Namen tragen und die Operatoren aus diesem Grund auch unterschiedliche Topics abonnieren müssen, ist implizit geregelt,

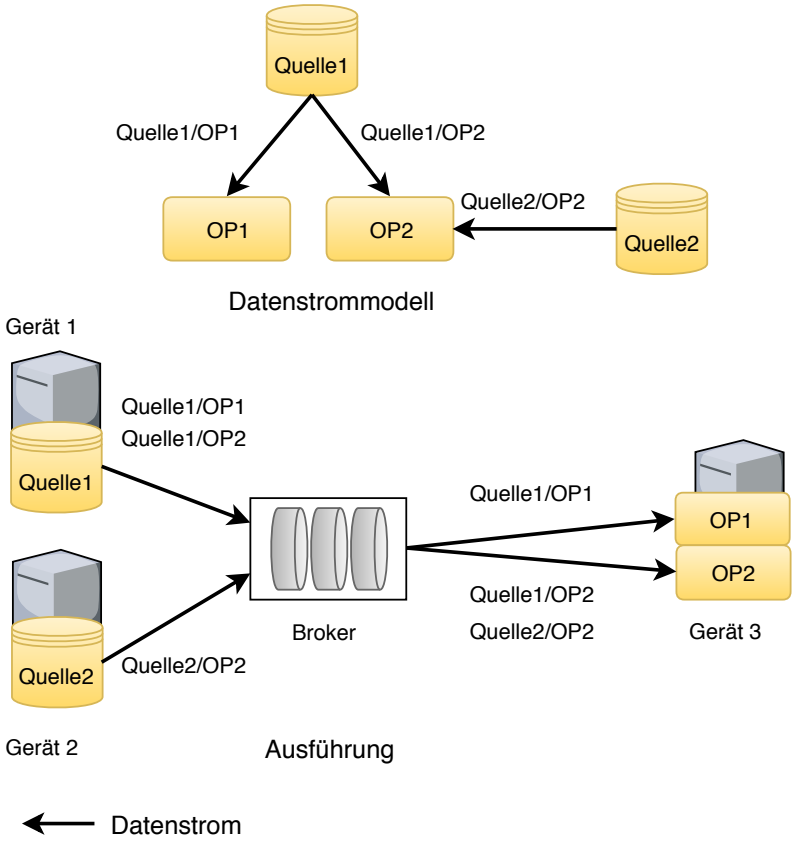

Abbildung 4.15: Beispiel für die Verwendung des MQTT-Protokolls zum Datenaustausch

an welche Operatoren die an Gerät 3 eingehenden Daten weitergeleitet werden müssen. In diesem Beispiel wurden zwei Datenquellen verwendet, die die Daten an den Broker senden. An deren Stelle können aber in Szenarien mit anderen Datenstrommodellen natürlich auch Operatoren treten, die selbst eingehende Kanten besitzen und ihre Ausgaben auf diese Weise an die ihnen im Datenstrommodell nachgestellten Operatoren weiterleiten.

Es genügt allerdings nicht, nur einen Nachrichten-Broker für die gesamte IoT-Umgebung einzusetzen. Ein einziger Broker, an den alle Operatoren Nachrichten senden, erhöht wiederum die Netzwerkauslastung und stellt ebenfalls einen Single Point Of Failure dar, der bei einem Ausfall die Funktion des gesamten Systems beeinträchtigt. Außerdem ist ein System mit nur einem Broker weniger gut skalierbar, da mit zunehmender Größe des Datenstrommodells und der Netzwerktopologie auch die Anzahl an Nachrichten steigt, die übermittelt werden müssen. Aus diesen Gründen muss jedes Gerät der IoT-Umgebung einen eigenen Broker erhalten. Operatoren, die Daten an einen anderen Operator übermitteln müssen, können dann den Broker des Geräts adressieren, auf dem der Zieloperator ausgeführt wird.

In Abbildung 4.16 ist ein Beispiel für eine Ausführung dargestellt, in der wie vorgesehen jedes Gerät einen eigenen Broker besitzt. Die Quelle sendet die von ihr produzierten Daten an die Broker von Gerät 2 und Gerät 3, weil auf diesen Geräten die Operatoren `OP1` und `OP2` ausgeführt werden, die die Daten benötigen. Beide Operatoren abonnieren dementsprechend die dazugehörigen Topics.

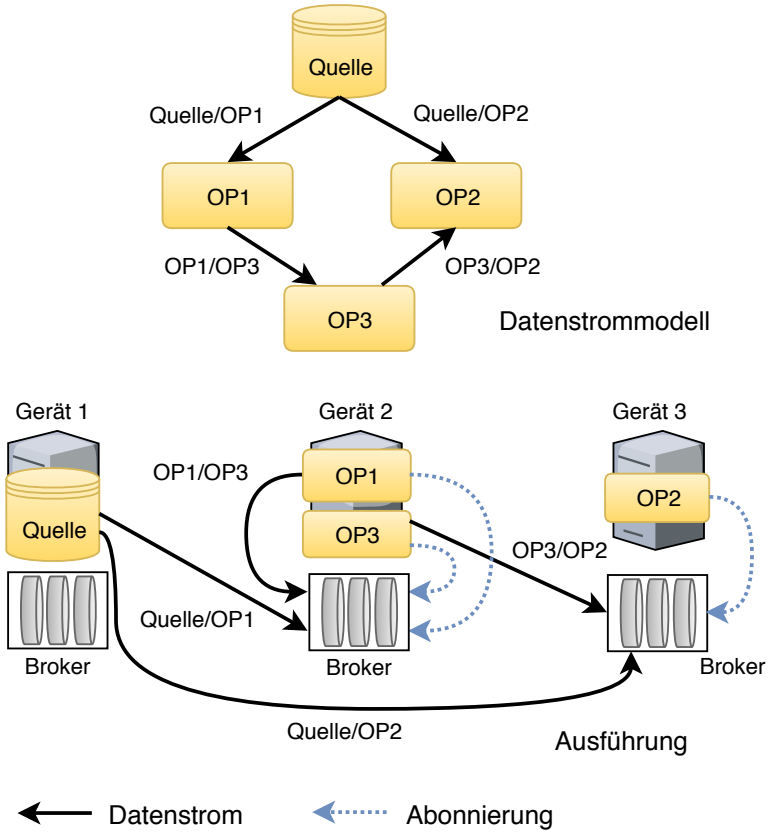

Abbildung 4.16: Beispiel für den Einsatz eines Brokers pro Gerät

Da `OP1` und `OP3` auf demselben Gerät ausgeführt werden, sendet Operator `OP1` die Ausgabedaten an den Broker des eigenen Geräts, von dem `OP3`, der das entsprechende Topic abonniert, dann die Daten erhält.

Bevor das Datenstrommodell in einer solchen Umgebung ausgeführt werden kann, müssen jedoch zuerst auf allen involvierten Geräten derartige Broker installiert werden. Dies kann durch die MBP parallel zur Installation der Operatoren auf den jeweiligen Geräten vorgenommen werden: Dazu müssen dem Shell-Skript, das die Operatorinstallation vornimmt, lediglich die notwendigen Befehle zum Herunterladen und Installieren des entsprechenden Softwarepakets hinzugefügt werden.

Die in diesem Abschnitt erfolgte Auswahl einer Kommunikationstechnologie erfüllt Teilziel 2.2 aus Abschnitt 3.2.

4.5.3 Routingkonzept

Prinzip

Das im vorherigen Abschnitt beschriebene Kommunikationskonzept funktioniert dann, wenn die Geräte, auf denen sich die Start- und Zieloperatoren eines Datenstroms befinden, direkt miteinander verbunden sind. So muss in Abbildung 4.16 Gerät 1 sowohl mit Gerät 2, als auch mit Gerät 3 eine gemeinsame Netzwerkverbindung besitzen. Verbindungen des Datenstrommodells werden aber von

```
{
  "rules": [
    {
      "topic": "Quelle/Operator1",
      "action": "consume"
    },
    {
      "topic": "Quelle/Operator2",
      "action": "forward",
      "target": "192.168.178.2"
    }
  ]
}
```

Listing 4.3: Beispiel für eine Routingtabelle

den Verteilungsalgorithmen auf Pfade abgebildet, die je nach Probleminstanz auch aus mehreren Abschnitten bestehen können. Aus diesem Grund müssen die Daten prinzipiell von dem Gerät des Startoperators über mehrere andere involvierte Geräte hinweg zum Zieloperator transportiert werden können. Um damit umzugehen, wird an dieser Stelle das Konzept von Routingtabellen als Lösung vorgeschlagen. Diese enthalten Regeln, die dem jeweiligen Gerät vorschreiben, wie mit eingehenden Nachrichten in Abhängigkeit von deren Topic umgegangen werden soll.

Diese Routingtabellen werden anhand der von den Verteilungsalgorithmen erzeugten Verteilung erstellt, noch bevor die Operatoren auf die IoT-Umgebung ausgebracht werden. Dies entspricht dem dritten Methodenschritt in Abbildung 4.1. Die berechneten Routingtabellen werden dann zusammen mit den anderen Dateien der Operatoren von der MBP auf die Geräte ausgeliefert. Zur Spezifikation der Routingtabellen wird JSON verwendet. Ein Beispiel einer solchen Tabelle für ein IoT-Gerät ist in Listing 4.3 dargestellt: Darin werden Regeln definiert, welche innerhalb des Arrays mit dem Schlüssel `rules` aufgeführt sind. Die Bedeutungen der einzelnen Attribute einer Regel werden im Folgenden erklärt.

topic: Dieses Attribut gibt das Topic der Nachrichten an, auf die die Regel angewendet werden soll.

action: Dieses Attribut spezifiziert die Aktion, die beim Erhalt einer Nachricht mit dem passenden Topic ausgeführt werden soll. Hier sind zwei Ausprägungen möglich: Die Zeichenkette `consume` als Wert gibt an, dass die Nachricht auf dem Gerät, zu dem die Routingtabelle gehört, ihr Ziel erreicht hat und von dem Operator konsumiert und verarbeitet werden darf. `forward` besagt dagegen, dass die Nachricht an ein anderes Gerät weitergeleitet werden muss.

target: Wenn die Nachricht weitergeleitet werden soll, enthält dieses Attribut die IP-Adresse des Brokers des Zielgeräts.

In dem Beispiel aus Listing 4.3 darf das Gerät, zu dem die Routingtabelle gehört, Nachrichten mit dem Topic `Quelle/Operator1` selbst verarbeiten, da es sich dabei um die Eingabedaten des Operators handelt, die er zur Ausführung benötigt. Nachrichten mit dem Topic `Quelle/Operator2` müssen dagegen an den Broker mit der IP-Adresse `192.168.178.2` weitergeleitet werden, es muss also eine neue Nachricht unter demselben Topic an diese Adresse verschickt werden.

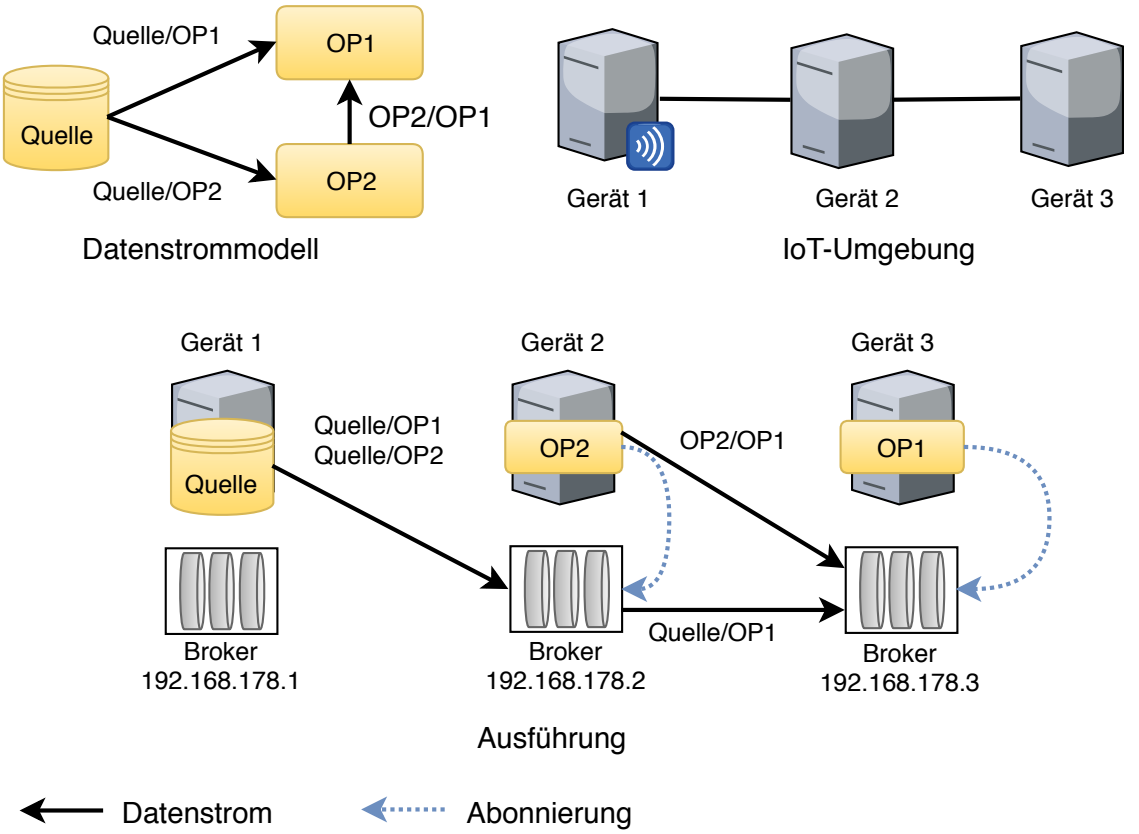

Abbildung 4.17: Beispiel für die Verwendung von Routingtabellen

```
{
  "rules": [
    {
      "topic": "Quelle/OP1",
      "action": "forward",
      "target": "192.168.178.3"
    },
    {
      "topic": "Quelle/OP2",
      "action": "consume"
    }
  ]
}
```

Listing 4.4: Routingtabelle für Gerät 2 aus Abbildung 4.17

```json
{
  "rules": [
    {
      "topic": "Quelle/OP1",
      "action": "consume",
    },
    {
      "topic": "OP2/OP1",
      "action": "consume",
    }
  ]
}
```

Listing 4.5: Routingtabelle für Gerät 3 aus Abbildung 4.17

In Abbildung 4.17 ist ein Beispiel für den Einsatz von Routingtabellen dargestellt. Im oberen Bereich ist ein Datenstrommodell gegeben, das auf der rechts davon abgebildeten IoT-Umgebung ausgeführt werden soll, wie im unteren Bereich der Abbildung dargestellt. Jedes Gerät erhält einen Broker, die jeweils über eine bestimmte IP-Adresse erreichbar sind. Da Gerät 1 nicht direkt mit Gerät 3 verbunden ist, müssen die Daten von der Quelle zu `OP1` auch über Gerät 2 übermittelt werden, was den Einsatz von Routingtabellen erforderlich macht. Die entsprechende Routingtabelle für Gerät 2 ist in Listing 4.4 abgebildet, die für Gerät 3 in Listing 4.5. Da an Gerät 1 keine Nachrichten eingehen, wird auf diesem Gerät keine Routingtabelle benötigt.

Erzeugung von Routingtabellen

Die Routingtabellen müssen automatisch anhand der von den Verteilungsalgorithmen vorgeschlagenen Verteilung erstellt werden, bevor die Geräte der IoT-Umgebung die Operatoren provisionieren. Der Pseudocode in Algorithmus 4.14 ermöglicht dies: Die Eingabe der Funktion CreateRou-

Algorithmus 4.14 Algorithmus zur Erzeugung von Routingtabellen aus einer Verteilung

```
1:  // Eingabe: Verteilung aus Verteilungsalgorithmus, Menge der Geräte
2:  function CreateRoutingTables((device, path), D)
3:      s ← (device, path)
4:
5:      // Ermittle alle Datenströme aus der Definitionsmenge von path
6:      Edges ← D_path
7:
8:      // Erstelle eine Abbildung von Geräten auf Mengen von Regeln
9:      table: D → P(Edges × {CONSUME, FORWARD} × D)
10:     // Initialisiere alle Regelmengen
11:     for all d ∈ D do
12:         table(d) ← ∅
13:     end for
```

```
14:     // Betrachte alle Datenströme einzeln
15:     for all edge ∈ edges do
16:         nPath ← path(edge)
17:         // Beginne mit dem Routing-Eintrag des Zielgeräts
18:         target ← target(nPath)
19:         table(target) ← table(target) ∪ {(edge, CONSUME, target)}
20:         // Betrachte alle Abschnitte des Pfads nPath einzeln
21:         for all (d1, d2) ∈ nPath(ℕ) do
22:             // Ermittle Regelmenge für d1
23:             Rules ← table(d1)
24:             // Wurde Rules für edge schon eine Regel hinzugefügt?
25:             if ∃(a, b, c) ∈ Rules : a = edge then
26:                 continue
27:             end if
28:             // Füge der Regelmenge von d1 eine Weiterleitungsregel hinzu
29:             table(d1) ← table(d1) ∪ {(edge, FORWARD, d2)}
30:         end for
31:     end for
32:     return table
33: end function
```

TINGTABLES besteht zum einen aus einer Verteilung, der Lösung für ein Verteilungsproblem, und zum anderen aus der Menge aller Geräte der betrachteten IoT-Umgebung. In Zeile 6 wird aus der Definitionsmenge der Abbildung `path`, die Teil des übergebenen Lösungstupels ist, die Menge aller in der Verteilung abgebildeten Datenströme ermittelt. In Zeile 9 wird eine Abbildung definiert, die den Netzwerkgeräten der Umgebung jeweils eine Routingtabelle zuordnet. Im Kontext des Algorithmus entspricht eine Routingtabelle einer Menge von Regeln, wobei jeder dieser Regeln ein Tupel ist, das sich aus einem Datenstrom, einer Aktion (`CONSUME` oder `FORWARD`) und einem Zielgerät zusammensetzt. Das geschieht analog zu dem Format für Routingtabellen, das in Listing 4.3 vorgestellt worden ist, mit dem Unterschied, dass an dieser Stelle ein Datenstrom anstelle eines Topics und ein Gerät anstelle einer IP-Adresse referenziert wird. Das Topic lässt sich allerdings leicht aus den Namen der in dem Datenstrom beteiligten Operatoren ableiten und die IP-Adressen der Geräte sind in dem Modell der Netzwerktopologie spezifiziert, sodass nur minimale Anpassungen vorgenommen werden müssen, um die aus diesem Algorithmus erzeugten Regelmengen in der Praxis als Routingtabellen einzusetzen.

In den Zeilen 11 bis 13 von Algorithmus 4.14 werden die Regelmengen für jedes Gerät jeweils als leere Mengen initialisiert. Die eigentliche Erzeugung der Regeln beginnt dann mit der Schleife in Zeile 15: Darin werden alle Datenströme der Verteilung einzeln als `edge` betrachtet und zunächst der Pfad (Zeile 16), auf den der jeweils aktuelle Datenstrom abgebildet wird und dessen Zielgerät (Zeile 18) ermittelt. In Zeile 19 wird die Regelmenge des Zielgeräts um eine Konsumierungsregel erweitert: Da es sich um das Gerät handelt, auf dem die Operatoren ausgeführt werden, die die Daten letztlich verarbeiten, müssen die eingehenden Daten mit dem zu `edge` gehörenden Topic an dieser Stelle konsumiert werden. Mit Hilfe der Schleife, die ab Zeile 21 beginnt, wird jeder Abschnitt des Pfads einzeln betrachtet, damit den Regelmengen der involvierten Geräte die benötigten Weiterleitungsregeln hinzugefügt werden können. In Zeile 23 wird zunächst die Regelmenge des Startgeräts des aktuell betrachteten Abschnitts ermittelt, in Zeile 25 wird dann geprüft, ob

diese bereits einen Eintrag zu dem Datenstrom `edge` enthält. Sollte dies der Fall sein, wird kein weiterer Eintrag hinzugefügt, um einen Konflikt zu vermeiden und stattdessen mit dem nächsten Pfadabschnitt fortgefahren. Andernfalls wird der Regelmenge in Zeile 29 eine Regel hinzugefügt, die besagt, dass Daten, die zu `edge` gehören, an das Zielgerät des aktuell betrachteten Pfadabschnitts weitergeleitet werden sollen.

Am Ende des Algorithmus, wenn alle Datenströme betrachtet worden sind, wird in Zeile 32 die Abbildung `table` ausgegeben. Sie bildet nun jedes in der Ausführung involvierte Gerät auf die Menge der benötigten Routingregeln ab.

Anwendung von Routingtabellen

Die Routingtabellen, die während des Provisionierungsvorgangs mit auf die Geräte ausgeliefert werden, müssen dort natürlich auch für eingehende Nachrichten angewendet werden. Diese Aufgabe kann durch eine eigens dafür vorgesehe Routinganwendung übernommen werden. Eine solche Anwendung wird im Folgenden konzeptionell beschrieben.

Die Routinganwendung ist für alle Geräte der IoT-Umgebung dieselbe und wird ähnlich wie auch der MQTT-Broker bei der Ausbringung der Operatoren im Hintergrund mitinstalliert und ausgeführt. Während der Ausführung des Datenstrommodells führt sie die folgenden Schritte aus:

Schritt 1: Die Routinganwendung liest zunächst die Routingtabelle ein, die ihr auf dem jeweiligen Gerät, auf dem sie installiert worden ist, vorliegt.

Schritt 2: Sie abonniert bei dem Broker ihres Geräts alle Topics, die in der Routingtabelle spezifiziert worden sind.

Schritt 3: Sie verbindet sich mit allen MQTT-Brokern der IoT-Umgebung, für die in der Routingtabelle Ziel-IP-Adressen angegeben worden sind und hält die Verbindungen offen.

Schritt 4: Für jede der abonnierten Nachrichten, die der Routinganwendung nun zugestellt werden, schlägt sie in ihrer Routingtabelle die erforderliche Aktion nach.

- **Fall 1: Konsum** Wenn die empfangene Nachricht auf diesem Gerät konsumiert werden soll, muss sich die Routinganwendung nicht weiter darum kümmern, da der betreffende Operator das dazugehörige Topic dann ebenfalls abonniert hat und ihm die Nachricht damit durch den Broker zugestellt wird.
- **Fall 2: Weiterleitung** Wenn die empfangene Nachricht weitergeleitet werden soll, dann sendet die Routinganwendung dem Broker, dessen IP-Adresse in der Routingtabelle für das Topic dieser Nachricht hinterlegt ist, eine Nachricht mit demselben Inhalt und demselben Topic der empfangenen Nachricht.

Auf diese Weise können die Daten der Operatoren während der Ausführung eines Datenstrommodells über die durch die Verteilung vorgesehenen Netzwerkpfade an die jeweiligen Geräte der Zieloperatoren weitergeleitet werden. Damit wird Teilziel 2.3 aus Abschnitt 3.2 erreicht.

4.5.4 Eigenschaften des Konzepts

Durch die Leichtgewichtigkeit des MQTT-Protokolls und den damit verbundenen niedrigen Hardwarevoraussetzungen wiegt die Heterogenität der Infrastruktur der IoT-Umgebung weniger schwer: MQTT kann somit sowohl auf Geräten mit fundierter Hardwareausstattung, wie Backend-Servern, als auch auf leistungsschwächeren Geräten wie Raspberry Pis eingesetzt werden. Durch die ressourcenschonende Arbeitsweise wird außerdem vermieden, dass die Kommunikation die Eigenschaften des Geräts beeinträchtigt: Wenn beispielsweise ein Gerät in dem Modell der Netzwerktopologie mit der Eigenschaft „Arbeitsspeicher“ und einem fixen Wert spezifiziert wird, dann ist der tatsächlich auf diesem Gerät vorhandene freie Arbeitsspeicher bei einem Kommunikationsprotokoll mit unvorhersehbar hohem Rechenaufwand starken Schwankungen unterworfen. Unter Umständen kann dann nicht sichergestellt werden, dass der in der Eigenschaft angegebene Wert immer zur Verfügung steht, was zur Folge hat, dass einige Operatoren, deren Anforderungen formell durch das Gerät erfüllt werden, trotzdem nicht ordnungsgemäß ausgeführt werden können. Diese Problematik wurde in Herausforderung 7 in Abschnitt 3.3 formuliert und kann durch Verwendung eines leichtgewichtigen Kommunikationsprotokolls zumindest teilweise entschärft werden.

Dadurch, dass jedes Gerät der IoT-Umgebung einen eigenen Broker erhält, wird in Bezug auf Herausforderung 4 aus Abschnitt 3.3 ein Single Point of Failure vermieden. Fällt ein Gerät oder ein Broker aus, können prinzipiell die Operatoren anderer Geräte, die nicht von den ausgefallenen Operatoren abhängig sind, weiterarbeiten, da deren Kommunikation nicht gestört wird. Natürlich hängt dies von dem konkreten Aufbau des Datenstrommodells ab: Wenn es einen Operator gibt, von dem alle anderen Operatoren abhängig sind, dann kann das System bei einem Ausfall trotzdem nicht weiterarbeiten. Im Regelfall jedoch wird durch das an dieser Stelle vorgeschlagene Kommunikationsprinzip die Chance erhöht, dass bei unerwarteten Ereignissen nur Teilsysteme ausfallen.

Herausforderung 5 aus Abschnitt 3.3 verlangt eine lose Kopplung zwischen den Geräten der IoT-Umgebung. Auch das ist bei diesem Ansatz gegeben: Die Geräte müssen einander zunächst nicht kennen, es müssen auch keine festen Kommunikationspfade vorgegeben werden, über die der Datenaustausch erfolgt. Stattdessen muss die IoT-Umgebung lediglich einmal in Form des Modells der Netzwerktopologie spezifiziert werden, was prinzipiell auch automatisch in Form einer Anmeldung erfolgen kann, wenn sich IoT-Geräte in diese Umgebung einklinken. Auf der Basis dieses Modells wird die Verteilung der Operatoren und die Kommunikationspfade zwischen diesen dann dynamisch berechnet und unabhängig von der konkreten Beschaffenheit und Identität der IoT-Geräte in der Umgebung installiert und ausgeführt. Dadurch wird eine lose Kopplung gewährleistet.

Die Skalierbarkeit des Ausführungskonzepts nach Herausforderung 6 lässt sich nur schwer von einem theoretischen Standpunkt aus beurteilen. Ein Grund dafür ist, dass die Wirkungsweise eines solchen Systems stark von der Ausprägung des Datenstrommodells und des Modells der Netzwerktopologie abhängig ist. Je nach dem, wie die Operatoren miteinander verknüpft sind und welche Struktur der IoT-Umgebung zugrunde liegt, kann das System mehr oder weniger gut skalierbar sein. Grundsätzlich lässt sich jedoch feststellen, dass durch den Einsatz von einem Broker pro involviertem Gerät die Arbeitslast, die zum Verteilen der Nachrichten auf die Klienten

erforderlich ist, auf viele Geräte übertragen wird. Dadurch wird die Gefahr reduziert, dass die Broker bei einer Skalierung des Systems durch zu viele Nachrichten überlastet werden und dadurch hohe Latenzzeiten entstehen.

5 Umsetzung

Dieses Kapitel beschreibt die Umsetzung der Konzepte, die in Kapitel 4 vorgestellt wurden. Die Implementierung dieser Lösungen wird in Form einer eigenständigen Softwarebibliothek realisiert, die nach Abschluss dieser Arbeit in das Modellierungswerkzeug FlexMash integriert werden soll.

In diesem Kapitel wird zunächst der Aufbau der Umsetzung erläutert, im Anschluss daran wird eine Übersicht über die Architektur des realisierten Softwaresystems gegeben. Abschließend werden die Tests beschrieben, denen die Implementierung unterzogen worden ist.

5.1 Aufbau

Die Implementierung des Konzepts erfolgt in der Programmiersprache Java[1] in der SDK-Version 1.8. Dies bietet sich an, da FlexMash selbst auch in Java realisiert worden ist und auf diese Weise die spätere Integration dieser Komponente in das Werkzeug erleichtert wird. Als integrierte Entwicklungsumgebung wird die freie Community-Edition von IntelliJ IDEA[2] des Herstellers JetBrains verwendet. Zur Automatisierung und Verwaltung des Build-Prozesses kommt das freie Werkzeug Gradle[3] zum Einsatz.

In den folgenden Unterabschnitten wird auf die Struktur des Projekts und die darin verwendeten Bibliotheken eingegangen.

5.1.1 Projektstruktur

Für die Umsetzung der Konzepte wurde in der Entwicklungsumgebung ein Projekt angelegt. Dessen Ordnerstruktur entspricht grundsätzlich derjenigen, die implizit durch das Werkzeug Gradle angelegt wird, allerdings wurde sie um weitere Verzeichnisse erweitert.

In Abbildung 5.1 die Verzeichnisstruktur des Projekts dargestellt. In dem Order `build` werden die während eines Build-Vorgangs erzeugten Dateien abgelegt und auch das dazugehörige JavaDoc-Dokument befindet sich darin. Der Quellcode der im Rahmen dieser Arbeit entwickelten Bibliothek befindet sich unter `src/main/java` im Hauptpaket `operator_distributor` und ist darin in mehrere Unterpakete gegliedert. Unter `src/tests` sind die Unit-Testfälle spezifiziert, auf die in Abschnitt 5.3 näher eingegangen wird. Der Quellcode der Testfälle ist im Unterordner `operator_distributor` enthalten und gliedert sich analog zu dem Quellcode der Bibliothek in Unterpakete, sodass die Testfälle leicht den zu testenden Klassen aus dem Verzeichnis `main` zugeordnet werden können. In

[1] https://www.java.com/
[2] https://www.jetbrains.com/idea/
[3] https://www.gradle.org/

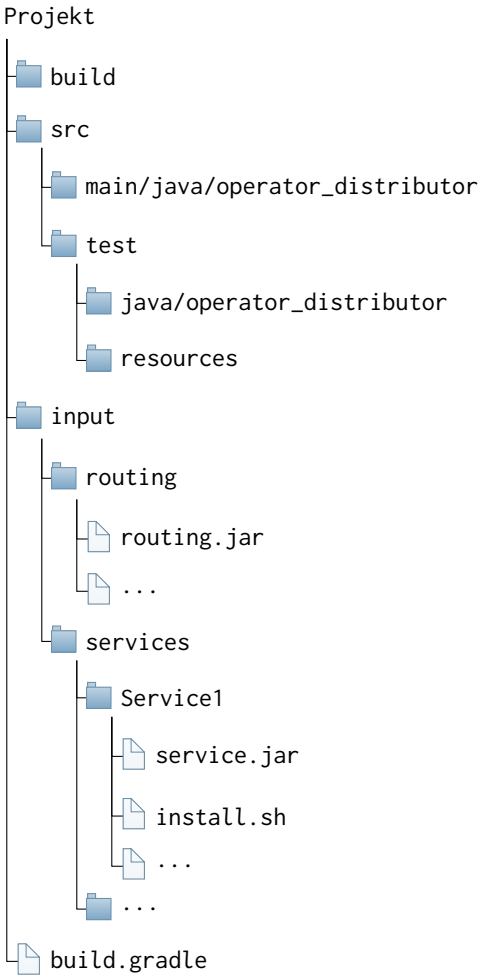

Abbildung 5.1: Verzeichnisstruktur des Projekts zur Umsetzung

dem Unterordner `resources` sind zahlreiche Dateien enthalten, die zur Ausführung der Testfälle erforderlich sind, wie Datenstrommodelle und Modelle der Netzwerktopologien, mit denen das korrekte Einlesen der Dateien überprüft werden kann.

Der Ordner `input` stellt einen Sammelort für alle Eingaben dar, die während der Ausführung des Verteilungsprozesses benötigt werden. In dem Unterordner `routing` werden mit Ausnahme der Routingtabellen alle Dateien gesammelt, die zur Umsetzung des in Abschnitt 4.5.3 vorgestellten Routingkonzepts benötigt werden. Das beinhaltet die Routinganwendung, die in der Lage ist, die empfangenen Nachrichten an die vorgesehenen Adressaten weiterzuleiten, aber auch Shell-Skripte, um diese auf den Geräten zu installieren, auszuführen, anzuhalten und zu deinstallieren. Routingtabellen müssen nicht in diesem Ordner platziert werden, da diese erst zur Laufzeit des Programms anhand der von den Verteilungsalgorithmen vorgeschlagenen Verteilung erzeugt werden. Im Rahmen des Provisionierungsprozesses werden alle Dateien, die in diesem Ordner enthalten sind, über die MBP auf jedes involvierte Gerät der IoT-Umgebung kopiert und die Shell-Skripte entsprechend ihrer Rolle ausgeführt. Im Unterordner `services` werden die Softwareelemente und Skripte hinterlegt, die zu den Operatoren des Datenstrommodells gehören. Dabei erhalten alle Dateien, die zu einem gemeinsamen Operator gehören und damit auf der MBP einen Adapter bilden,

ein eigenes Verzeichnis. Damit die Verzeichnisse den Operatoren zugeordnet werden können, tragen sie die Namen, die im Datenstrommodell wie in Abschnitt 4.3.1 beschrieben bei jedem Operator unter dem Schlüssel `serviceId` angegeben worden sind. Während der Provisionierung werden die in den Ordnern enthaltenen Dateien auf die Geräte kopiert, auf die der entsprechende Operator verteilt worden ist. Im Anschluss wird dann jeweils das dazugehörige Installationsskript ausgeführt. Prinzipiell können auch Datenstrommodelle und Modelle der Netzwerktopologie in dem Ordner `input` gespeichert werden, um dann eingelesen und verarbeitet zu werden. Das ist jedoch nicht zwingend erforderlich, da der Pfad zu diesen Eingaben bei Aufruf der Bibliotheksfunktionen frei angegeben werden kann.

Das Gradle-Skript `build.gradle` im Projektverzeichnis legt grundlegende Projekteinstellungen fest, wie die verwendete Java-Version. Bei Ausführung des Build-Vorgangs wird dieses Skript ausgeführt. In ihm sind auch die Abhängigkeiten der Bibliothek spezifiziert und die Bezugsquellen der verwendeten externen Bibliotheken. Bei Durchführung der Unit-Tests trägt es zudem dafür Sorge, dass die Testdateien aus dem Ordner `src/test/resources` vor der Ausführung der Testfälle an den richtigen Ausführungsort kopiert werden und den Tests zur Verfügung stehen.

5.1.2 Verwendete Bibliotheken

In der Umsetzung werden mehrere Softwarebibliotheken von Drittanbietern verwendet, welche in Tabelle 5.1 mit ihrem Namen, dem dazugehörigen Paketnamen, der Version und der Lizenz aufgeführt sind. Nachfolgend wird der Zweck, für den sie eingesetzt werden, erläutert.

Name	**Paket**	**Version**	**Lizenz**
Apache Commons Validator	commons-validator	1.6	Apache 2.0
Jersey Client	com.sun.jersey	1.19.4	CDDL 1.1, GPL 1.1
JGraphT Core	org.jgrapht	1.2.0	EPL 1.0, LGPL 2.1
JSON	org.json	20180130	JSON-Lizenz
JUnit	junit	4.11	EPL 1.0

Tabelle 5.1: Verwendete Bibliotheken von Drittanbietern

Apache Commons Validator: Diese Bibliothek stellt verschiedene Methoden zur Verfügung, um Benutzereingaben zu validieren. Im Rahmen dieser Arbeit werden damit die Formate der IP- und MAC-Adressen, die in einem Modell einer Netzwerktopologie spezifiziert worden sind, auf Gültigkeit geprüft.

Jersey Client: Jersey Client erlaubt die Realisierung von REST-Klienten, die in der Lage sind, HTTP-Anfragen (Get, Post, Put, Delete) an einen Server zu senden und die Antworten entgegenzunehmen. Die auf dem Server vorhandenen Ressourcen werden dabei intern jeweils als Objekte repräsentiert, was einen objektorientierten Realisierungsansatz und damit ein für dieses Projekt leicht zu integrierendes Programmiermodell ermöglicht. Die Bibliothek wird an dieser Stelle zur Umsetzung eines Klienten eingesetzt, der die MBP über die dafür vorgesehene REST-Schnittstelle gemäß Abschnitt 4.5 ansteuern kann.

JGraphT Core: Bei JGraphT handelt es sich um eine Bibliothek, die Objekte und Methoden für graphentheoretische Operationen zur Verfügung stellt. Damit ist es möglich, verschiedene Arten von Graphen zu modellieren und auf diesen gängige Algorithmen wie die Suche nach kürzesten Wegen auszuführen. Auf diese Weise bildet sie die Grundlage für die digitale Repräsentation des Datenstrommodells und der Netzwerktopologie und ist auch Bestandteil der Verteilungsalgorithmen, beispielsweise um topologische Sortierungen durchzuführen und Pfade zwischen Geräten der Netzwerktopologie zu finden.

JSON: Diese Bibliothek ermöglicht das Parsen von Zeichenketten im JSON-Format und die Transformation dieser in Java-Objekte, auf denen dann weiterführende Operationen ausgeführt werden können. Es ist jedoch auch andersherum möglich, JSON-Zeichenketten aus Objekten zu erzeugen. Die Bibliothek wird an dieser Stelle eingesetzt, um die im JSON-Format spezifizierten Datenstrommodelle und Modelle von Netzwerktopologien einzulesen und auf ihre entsprechenden digitalen Repräsentationen abzubilden. Auch kommt sie zum Einsatz, um die Serverantworten einzulesen, die der Klient zur Ansteuerung der MBP vom Server erhält.

JUnit: JUnit ist eine Bibliothek zur automatisierten Durchführung von Modultests in Java. Sie wird dazu verwendet, um Testfälle zu erstellen, mit denen die Umsetzung während der Implementierung überprüft und bei Änderungen des Quellcodes Regressionstests [Joc13] durchgeführt werden können.

5.2 Architekturentwurf

In diesem Abschnitt wird die Architektur der Bibliothek vorgestellt, die die Konzepte aus Kapitel 4 implementiert.

In Abbildung 5.2 ist ein Komponentendiagramm dargestellt, das auf dem UML2[4]-Standard der Object Management Group[5] basiert und den architektonischen Entwurf der entwickelten Bibliothek beschreibt. Es besteht aus insgesamt sechs Komponenten, wovon sich zwei aus jeweils mehreren Teilkomponenten zusammensetzen. Im Zentrum des Diagramms steht dabei die Komponente `FlexMash`, die das um die neuen Funktionen erweiterte Modellierungswerkzeug repräsentiert. Es ist zu beachten, dass diese Komponente im Gegensatz zu den anderen Komponenten nur aus Gründen der Nachvollziehbarkeit in das Komponentendiagramm aufgenommen worden ist, um zu verdeutlichen, wie die bestehende Version von FlexMash mit der an dieser Stelle entwickelten Bibliothek verknüpft werden kann. FlexMash selbst wird im Verlauf dieser Bachelorarbeit nicht erweitert oder modifiziert - die praktische Integration der Bibliothek ist zum Zeitpunkt des Abschlusses der Arbeit ausstehend. Auf die weiteren in Abbildung 5.2 dargestellten Komponenten wird nachfolgend im Detail eingegangen.

[4] Unified Modeling Language, siehe http://www.uml.org/
[5] Konsortium, siehe http://www.omg.org/

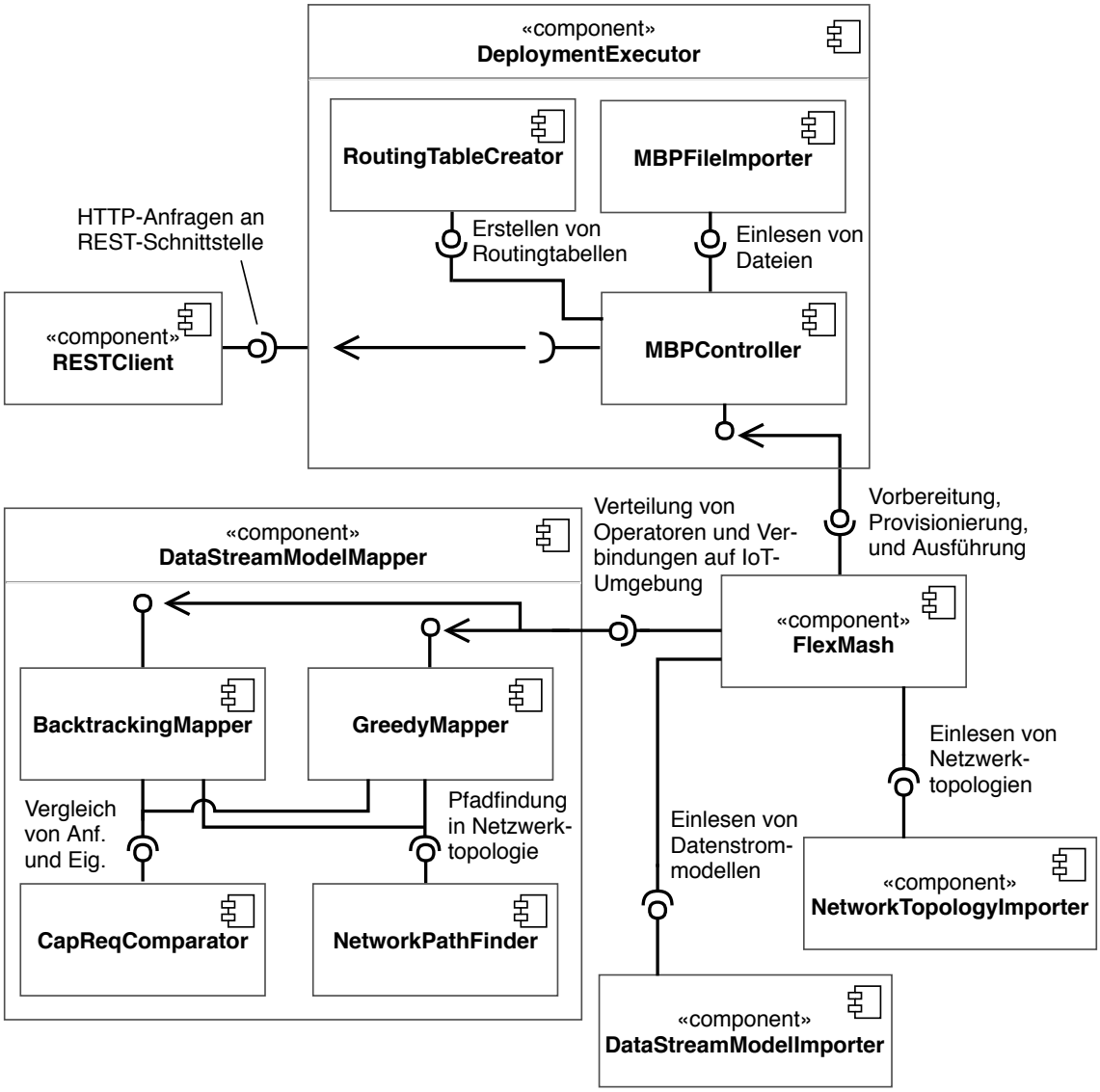

Abbildung 5.2: UML-Komponentendiagramm der Bibliothek

Komponente „DataStreamModelImporter“

Diese Komponente bietet FlexMash eine Schnittstelle an, um Datenstrommodelle einzulesen und auf geeignete Datenstrukturen abzubilden. Diese digitale Repräsentation des Datenstrommodells besteht hauptsächlich aus einer Klasse, die einen gerichteten, ungewichteten Graphen der JGraphT-Bibliothek hält, dem über dafür vorgesehene Methoden in Form von Knoten und Kanten Operatoren und Verbindungen hinzugefügt werden können. Dabei wird implizit darauf geachtet, dass die für das Datenstrommodell geltenden Invarianten eingehalten werden, beispielsweise dass keine Schlingen und Zyklen in den Graphen integriert werden. Für die Operatoren und Verbindungen existieren Klassen, die neben den grundlegenden Eigenschaften wie Namen und IDs bei Operatoren und Start- und Zieloperator bei Verbindungen auch jeweils eine Menge von Anforderungen speichern, die an die Infrastruktur gestellt werden. Die Anforderungen selbst werden dabei durch Anforderungs-Objekte

repräsentiert, die alle der in Abschnitt 4.2.2 beschriebenen Attribute beinhalten. Die Klasse des Datenstrommodells stellt darüber hinaus Methoden zur Verfügung, die es erlauben, rudimentäre Operationen auf dem Graphen auszuführen. So können beispielsweise die Vorgängeroperatoren oder eingehende Kanten eines Operators bestimmt werden.

Damit ein Datenstrommodell durch diese Komponente eingelesen werden kann, wird vorausgesetzt, dass dieses im JSON-Format vorliegt, entweder als Zeichenkette oder als Datei. Im ersten Fall wird bei Aufruf der Komponente die Zeichenkette als Parameter übergeben, im zweiten Fall der Pfad zu der Datei. Die Verarbeitung der Eingabe erfolgt dann mit Hilfe der im vorherigen Abschnitt erwähnten JSON-Bibliothek. Dafür werden zunächst die Operatoren des Datenstrommodells einschließlich ihrer Anforderungen ausgelesen und auf interne Operator-Objekte abgebildet, welche dann dem Datenstrommodell hinzugefügt werden. Im Anschluss werden die Verbindungen zwischen den Operatoren erfasst und ebenfalls in den Graphen des Modells eingetragen. Besitzt ein Operator keine eingehenden Verbindungen, wird er als Datenquelle markiert.

Komponente „NetworkTopologyImporter“

Analog zur Komponente `DataStreamModelImporter` bietet diese Komponente eine Schnittstelle zum Einlesen von Modellen von Netzwerktopologien an. Auch zur digitalen Repräsentation von Netzwerktopologien liegt intern eine Klasse vor, deren Kern durch einen Graphen der JGraphT-Bibliothek gebildet wird. Im Gegensatz zu dem Graphen für Datenstrommodelle handelt es sich dabei allerdings um einen gewichteten, ungerichteten Graph. Diesem können über dafür vorgesehene Methoden Knoten und Kanten in Form von Netzwerkgeräten und Netzwerkverbindungen hinzugefügt werden. Dabei wird auch an dieser Stelle implizit darauf geachtet, dass die Invarianten für Netzwerktopologien eingehalten werden: Zyklen sind bei diesen Graphen im Gegensatz zu den Datenstrommodellen zwar erlaubt, Schleifen jedoch ebenfalls nicht. Die Klasse der Netzwerktopologie stellt außerdem auf JGraphT-basierende Methoden bereit, die es ermöglichen, Operationen auf dem Graphen auszuführen. Dazu gehört unter anderem die Möglichkeit, kürzeste Wege zwischen zwei Geräten im Graphen zu finden.

Die Kantengewichte des Graphen repräsentieren die Netzwerkdistanzen zwischen den IoT-Geräten gemäß der Beschreibung in Abschnitt 4.4.1. Die Distanzen werden in dieser Implementierung jedoch nicht nur als Zahl modelliert, sondern in Form von Objekten, die potentiell beliebig viele unterschiedliche Attribute besitzen können, wie beispielsweise Bandbreite oder Latenz. Als Grundlage dafür dient die Klasse `NetworkDistance`, die von einem Anwender dieser Bibliothek als Oberklasse für eigene Distanzklassen verwendet werden kann. Diese Klasse verfügt über die Methode `getDistance`, die von dem Anwender so zu überschreiben ist, dass sie aus den Attributen seiner Distanzklasse einen zusammenfassenden numerischen Distanzwert berechnet. Die Implementierung dieser Methode obliegt dabei dem Anwender. Auf diese Weise hat er die Möglichkeit, die Verteilung des Datenstrommodells auf die Netzwerktopologie durch die Algorithmen hinsichtlich seiner eigenen Anforderungen zu optimieren.

Für die Netzwerkgeräte und -verbindungen existieren Klassen, deren Attribute diese Elemente beschreiben. So werden für Geräte die ID aus der Netzwerktopologie, die MAC-Adresse, die IP-Adresse und der Benutzername für SSH-Zugriff spezifiziert, für Verbindungen die beiden involvierten Geräte und das dazugehörige Netzwerkdistanz-Objekt. Darüber hinaus kann sowohl den

Geräten, als auch den Verbindungen eine Menge von Eigenschaften im Sinne der Konzeption aus Abschnitt 4.2.3 hinzugefügt werden. Ähnlich wie die Anforderungen werden auch die Eigenschaften durch eigene Objekte repräsentiert.

Um das Modell einer Netzwerktopologie einlesen zu können, ist es wie bei Datenstrommodellen erforderlich, dass es wie in Abschnitt 4.3.2 beschrieben im JSON-Format vorliegt, entweder in Form einer JSON-Zeichenkette oder als Datei. Das Modell der Netzwerktopologie selbst ist jedoch nicht ausreichend, es wird auch die Spezifikation des Schemas benötigt, das dem Modell zugrunde liegt. Das liegt daran, dass im Modell der Netzwerktopologie gemäß der IoT-Lite Ontology nicht nur die Geräte der IoT-Umgebung spezifiziert werden, sondern potentiell auch viele andere Elemente, wie beispielsweise einzelne Hardwarekomponenten, Sensoren oder geographische Koordinaten. Damit unter all diesen Elementen die Definitionen gefunden werden können, die sich auf Geräte beziehen, wird das Schema benötigt, da in diesem die Typen der Geräte festgelegt werden. Auch das Schema muss im JSON-Format vorliegen, entweder als Zeichenkette oder als Datei. Zusammen mit dem Modell der Netzwerktopologie wird es dann beim Aufruf der Komponente übergeben. Beim Einlesen selbst wird zunächst die Schema-Datei nach Definitionen von Typen durchsucht, die direkt oder indirekt von dem Typ `ssn:Device` erben. Dieser Typ repräsentiert den Obertyp alle Geräte, die davon abgeleiteten Typen sind Spezialisierungen dieses Typs und beschreiben damit auch Geräte. Sobald alle diese Typen gefunden worden sind, wird das Modell der Netzwerktopologie nach Gerätespezifikationen durchsucht, die diese Typen aufweisen. Dabei werden nur vollständige Spezifikationen in Betracht gezogen: Sollte bei einer Gerätespezifikation entweder die IP-Adresse, die MAC-Adresse oder der Benutzername für den SSH-Zugriff fehlen, kann es später in der Ausführung nicht verwendet werden und wird daher auch von dieser Komponente während dem Einlesevorgang nicht berücksichtigt. Zur Überprüfung der Gültigkeit des Formats der IP- und MAC-Adressen werden Funktionen aus der Bibliothek Apache Commons Validator verwendet. Für jedes der akzeptierten Geräte wird ein Java-Objekt mit entsprechenden Attributen erzeugt und der digitalen Repräsentation der Netzwerktopologie hinzugefügt. Im Anschluss werden die Eigenschaften, die zu dem jeweiligen Gerät gehören, ebenfalls aus dem Modell der Netzwerktopologie extrahiert und in das entsprechende Objekt eingetragen. Wie in Abschnitt 4.3.2 beschrieben gibt es derzeit keine Möglichkeit, um die Verbindungen zwischen den Geräten der IoT-Umgebung explizit zu spezifizieren. Aus diesem Grund werden, sobald das Einlesen der Geräte aus dem Modell der Netzwerktopologie abgeschlossen ist, alle Geräte im Graphen paarweise miteinander durch Kanten verbunden.

Diese Komponente realisiert gemeinsam mit der Komponente `DataStreamModelImporter` das Teilziel 3.1 aus Abschnitt 3.2.

Komponente „DataStreamModelMapper“

Diese Komponente der Bibliothek bietet FlexMash eine Schnittstelle zur Abbildung der Operatoren und Verbindungen des Datenstrommodells auf die IoT-Geräte und Netzwerkpfade in der Netzwerktopologie an. Wahlweise kann dafür der Greedy-Algorithmus aus Abschnitt 4.4.3 oder der Backtracking-Algorithmus aus Abschnitt 4.4.4 mit der optionalen Möglichkeit zur Findung der bestmöglichem Lösung eingesetzt werden, wodurch Teilziel 3.4 erreicht wird. Diese ist modular aufgebaut und leicht zu erweitern. So verfügen die beiden Verteilungsalgorithmen über die gemeinsame Oberklasse `MappingAlgorithm`, die bereits wichtige Grundfunktionen zur Lösungsfindung realisiert und durch Vererbung die Implementierung weiterer Verteilungsalgorithmen erlaubt. Bei

Verwendung dieser Komponente werden die digitale Repräsentation des Datenstrommodells und des Modells der Netzwerktopologie an die Schnittstelle übergeben, sowie eine Referenz auf den zu verwendenden Algorithmus. Die Ausgabe besteht dann aus einem Lösungsobjekt, das gemäß der Beschreibung in Abschnitt 4.4.1 aufgebaut ist und eine Abbildung von Operatoren auf Geräte und eine Abbildung von Datenströmen auf Netzwerkpfade enthält. In der Klasse der Lösungsobjekte werden darüber hinaus Methoden implementiert, mit denen die Lösungen verändert oder erweitert werden können, sollte dies gewünscht sein. Das gibt einem Benutzer prinzipiell die Möglichkeit, eine erzeugte Lösung nochmals zu überprüfen und gegebenenfalls seinen Vorstellungen anzupassen. Voraussetzung dafür ist, dass zuvor eine entsprechende Benutzerschnittstelle in FlexMash eingerichtet wird. Weitere Methoden der Lösungen ermöglichen es, die Netzwerkdistanz der Lösung zu bestimmen und zu überprüfen, ob das Lösungsobjekt eine gültige Lösung gemäß Definition 4.4.4 repräsentiert. Die beiden Verteilungsalgorithmen geben allerdings bereits standardmäßig gültige Lösungen aus. Wenn sie für eine Probleminstanz keine Lösung finden, wird von dieser Komponente die Ausnahme `NoSolutionFoundException` geworfen, die von der aufrufenden Instanz gefangen werden kann. Die digitale Repräsentation der Lösung erfüllt Teilziel 3.2.

Im Folgenden werden die Teilkomponenten dieser Komponente näher beschrieben.

Teilkomponente „CapReqComparator“

Diese Teilkomponente realisiert den Vergleich zwischen einer Menge von Anforderungen und einer Menge von Eigenschaften, gemäß der Konzepte aus Abschnitt 4.2.5. Das erfüllt Teilziel 3.3. Beide Verteilungsalgorithmen verwenden die Komponente, um zu überprüfen, ob die Eigenschaften eines Geräts oder einer Netzwerkverbindung die Anforderungen bestimmter Operatoren oder Datenströme erfüllen. Dabei werden auch konsumierend-wirkende Anforderungen entsprechend berücksichtigt und die Werte von Eigenschaften bei Bedarf angepasst. Die Operationen, die diesen Vergleichen in der Implementierung zugrunde liegen, enthalten jeweils Fallunterscheidungen für die vier verschiedenen Datentypen, die im aktuellen Konzept von Anforderungen und Eigenschaften unterstützt werden. So werden beispielsweise Anforderungen und Eigenschaften vom Typ „Zeichenkette“ auf andere Weise verglichen als solche vom Typ „Ganzzahl“. Durch eine gemeinsame Interface-Klasse wird dabei sichergestellt, dass die Operationen für jeden Datentypen die gleichen Schnittstellen aufweisen. Dadurch lässt sich das Konzept prinzipiell auf modulare Weise auf beliebig viele weitere Datentypen ausweiten, sofern Bedarf besteht.

Die Schnittstellen, die von dieser Komponente zum Vergleich von Anforderungen und Eigenschaften nach außen hin angeboten werden, akzeptieren auch die Elemente des Datenstrommodells und der Netzwerktopologie als Parameter. Dadurch können der Komponente anstelle von Mengen von Anforderungen und Eigenschaften direkt die Java-Objekte von Operatoren, Verbindungen, Geräte, Netzwerkverbindungen und Netzwerkpfaden übergeben werden. Die innerhalb der Komponente stattfindenden Vergleiche wurden dementsprechend erweitert. Dies ermöglicht eine einfachere Verwendung, da beispielsweise bei Netzwerkpfaden dann bereits implizit alle Abschnitte des Pfads einzeln auf die Erfüllung der Anforderungen überprüft werden können.

Teilkomponente „NetworkPathFinder“

Diese Teilkomponente offeriert Funktionen, um innerhalb der digitalen Repräsentation des Modells der Netzwerktopologie Netzwerkpfade zwischen zwei Geräten zu finden. Sie wird von den beiden Verteilungsalgorithmen verwendet, um Pfade zu finden, auf die die Verbindungen des Datenstrommodells abgebildet werden können. Netzwerkpfade werden an dieser Stelle in Form eigener Objekte repräsentiert, die in Anlehnung an Definition 4.4.2 eine geordnete Liste der Netzwerkverbindungen (Abschnitte) halten, aus denen sie sich zusammensetzen. Die Objekte des Start- und Zielgeräts werden ebenfalls gespeichert. Dies ist erforderlich, um die Richtung des Netzwerkpfads zu ermitteln. Ein Netzwerkpfad kann durch das Hinzufügen neuer Netzwerkverbindungen beliebig erweitert werden - der Zielknoten wird automatisch angepasst. Dabei wird implizit sichergestellt, dass der Netzwerkpfad valide bleibt: So können beispielsweise keine Netzwerkverbindungen hinzugefügt werden, an denen das aktuelle Zielgerät des Netzwerkpfads nicht beteiligt ist, da der Pfad dann nicht mehr durchgängig ist und eine Lücke aufweist. Die Objekte der Netzwerkpfade bieten einige Methoden an, beispielsweise solche, um in der richtigen Reihenfolge über alle Abschnitte des Pfads zu iterieren oder zu prüfen, ob ein bestimmtes Gerät oder eine bestimmte Netzwerkverbindung darin enthalten ist.

Beim Aufruf der Komponente werden ein Start- und ein Zielgerät als Parameter übergeben. Es kann sowohl nach allen gültigen Pfaden vom Start- zum Zielgerät gesucht werden, als auch nach einer bestimmten Anzahl kürzester Pfade, welche dann als Liste geordnet nach ihrer Gesamtdistanz zurückgegeben werden. Die Schnittstelle bietet auch die Möglichkeit, eine ausführbare Methode („Runnable“) zu übergeben, die bei jedem Schritt, in dem dem Netzwerkpfad während der Pfadsuche ein weiterer Abschnitt hinzugefügt wird, überprüft, ob diese neue Netzwerkverbindung verwendet werden darf und dementsprechend einen Booleschen Wert zurückgibt. Dies wird von den Verteilungsalgorithmen dazu verwendet, um mit Hilfe der Komponente `CapReqComparator` zu prüfen, ob die Netzwerkverbindung die Anforderungen des gerade betrachteten Datenstroms aus dem Datenstrommodell erfüllt. Kann kein Pfad gefunden werden, wird die Ausnahme `NoPathFoundException` geworfen. Als Grundlage werden die Methoden genutzt, die von der Klasse der Netzwerktopologie für die Ausführung von Operationen auf dem zugrundeliegenden Graphen angeboten werden.

Teilkomponente „GreedyMapper“

Diese Teilkomponente implementiert den in Abschnitt 4.4.3 vorgestellten Greedy-Algorithmus zur Findung einer Lösung für eine Instanz des Verteilungsproblems unter Verwendung der Komponenten `CapReqComparator` und `NetworkPathFinder`. Bei dieser Implementierung kann über die Methode `setMaxNumberOfNetworkEdges` zusätzlich eingestellt werden, aus wie vielen Abschnitten die Netzwerkpfade zwischen den Geräten der Netzwerktopologie maximal bestehen dürfen, um die Suchzeit bei IoT-Umgebungen mit vielen Geräten zu begrenzen.

Teilkomponente „BacktrackingMapper“

Diese Teilkomponente implementiert den in Abschnitt 4.4.4 vorgestellten Backtracking-Algorithmus zur Findung einer Lösung für eine Instanz des Verteilungsproblems. Dabei wird ebenfalls von den Funktionen der Komponenten `CapReqComparator` und `NetworkPathFinder` Gebrauch gemacht. Unter Verwendung der Methode `setBestSolution` kann eingestellt werden, ob der Algorithmus bereits

nach der ersten gefundenen Lösung abbrechen oder den gesamten Lösungsraum durchsuchen soll, um die bestmögliche Lösung zu finden. Wie schon bei der `GreedyMapper`-Komponente kann auch hier zusätzlich über die Methode `setMaxNumberOfNetworkEdges` festgelegt werden, wie lang die Netzwerkpfade maximal werden dürfen. Mit der Methode `setMaxNumberOfPathsToCheck` kann bei Bedarf außerdem die Anzahl der Netzwerkpfade zwischen zwei Geräten begrenzt werden, die bei der Abbildung von Datenströmen auf Netzwerkpfade in jedem Schritt des Algorithmus betrachtet werden, bevor der vorhergehende Schritt wieder rückgängig gemacht wird. Diese Einstellungen können die Suchzeit bei IoT-Umgebungen mit vielen Geräten begrenzen, allerdings ist dann unter Umständen nicht mehr garantiert, dass tatsächlich die bestmögliche Lösung gefunden wird.

Komponente „RESTClient“

Diese Komponente bietet Funktionen an, um HTTP-Anfragen an Anwendungen zu senden, die über eine REST-Schnittstelle verfügen. Diese Funktionalität wird von der Komponente `MBPController` genutzt, um mit der MBP zu kommunizieren. Bei dem Entwurf wurde darauf geachtet, die Komponente möglichst wiederverwendbar zu gestalten, um sie auch in anderem Kontext einsetzen zu können. Bei der Instanziierung des REST-Klienten wird ein Uniform Resource Locator (URL) übergeben, die auf den Hauptpfad der Anwendung verweist, an den die Anfragen gesendet werden sollen. Ausgehend davon können mit dem REST-Klienten sogenannte REST-Ressourcen angelegt werden. Dabei handelt es sich um Java-Objekte, die die auf dem Server befindlichen Ressourcen wie Geräte oder Operatoren repräsentieren. Jede dieser REST-Ressourcen hält dabei den Pfad, unter dem sie auf dem Server erreichbar ist. Auf den REST-Ressourcen können nun gemäß dem REST-Paradigma `GET`-, `POST`-, `PUT`- und `DELETE`-Anfragen ausgeführt werden, um die auf dem Server befindlichen Ressourcen auszulesen oder zu modifizieren. Dabei kommt die Bibliothek *Jersey Client* zum Einsatz, die die Ausführung der Anfragen übernimmt. Die Antworten des Servers werden jeweils von dem REST-Klienten entgegengenommen, verarbeitet und in Form eines JSON-Objekts der JSON-Bibliothek zurückgegeben. Sollte bei der Ausführung einer Anfrage ein Fehler auftreten, wird in Form einer Unterklasse von `HTTPRequestException` eine Ausnahme geworfen, die den aufgetretenen Fehler beschreibt.

Komponente „DeploymentExecutor“

Diese Komponente bietet FlexMash eine Schnittstelle an, um die Ausführung des Datenstrommodells in der IoT-Umgebung vorzubereiten, die Operatoren mit den Geräten zu provisionieren, das Kommunikationskonzept aus Abschnitt 4.5 in der Umgebung einzurichten und die Ausführung zu initiieren. Als Grundlage dafür dienen die digitale Repräsentation der Netzwerktopologie und das Lösungs-Objekt, das von den Verteilungsalgorithmen erzeugt wurde. Beide Elemente werden bei dem Aufruf der Komponente als Parameter übergeben.

Um mit den digitalen Abbildern der Geräte, Operatoren, Adaptern und Dateien umgehen zu können, die auf Seiten der MBP angelegt und verwaltet werden müssen, wurden dafür innerhalb dieser Komponente jeweils eigene Modell-Klassen angelegt. Instanzen dieser Klassen repräsentieren die auf der MBP befindlichen digitalen Abbilder und tragen deren Eigenschaften in Form von Attributen. Da die MBP Argumente in Form von Zeichenketten im JSON-Format entgegennimmt und ihre Antworten selbst auch in diesem Format ausgibt, wurden diese Klassen derart um JSON-Funktionen erweitert, dass ihre Instanzen auf möglichst einfache Weise mit der MBP synchronisiert werden

können. Zu diesem Zweck verfügen alle Klassen über einen Konstruktor, der es erlaubt, eine Instanz der Klasse aus einem JSON-Objekt zu erzeugen, das das korrespondierende Element auf der MBP beschreibt. Außerdem besitzen sie eine Methode, mit der aus dem jeweiligen Java-Objekt ein entsprechendes JSON-Objekt erzeugt werden kann. Dies vereinfacht die Kommunikation mit der MBP: Soll beispielsweise ein Java-Objekt, das einen Operator repräsentiert, auf der MBP angelegt werden, muss es nur mit der entsprechenden Methode in das JSON-Format konvertiert werden und kann dann bei der entsprechenden HTTP-Anfrage direkt als Parameter angegeben werden. Sollen dagegen alle auf der MBP abgebildeten Operatoren ausgelesen werden, können unter Verwendung des dafür vorgesehenen Konstruktors aus der Server-Antwort direkt die dazugehörigen Java-Objekte erstellt werden, auf denen dann lokal weitergearbeitet werden kann.

Im Folgenden werden die Teilkomponenten dieser Komponente näher beschrieben.

Teilkomponente „RoutingTableCreator“

Diese Teilkomponente wird von der Komponente `MBPController` aufgerufen, um gemäß der Konzepte aus Abschnitt 4.5.3 Routingtabellen zu erzeugen. Die Grundlage dafür bildet die von den Verteilungsalgorithmen generierte Lösung, die dieser Komponente als Parameter übergeben wird. Routingtabellen werden innerhalb dieser Komponente als Java-Objekte abgebildet, die jeweils eine Liste von Routingregeln halten. Diese Regeln werden ebenfalls durch Objekte repräsentiert, die aus einem Datenstrom, einer Routing-Aktion (`CONSUME` oder `FORWARD`) und einem Zielgerät bestehen. Die Objekte der Routingtabellen bieten unter anderem Methoden an, um neue Regeln hinzuzufügen oder bestehende Regeln aus der Tabelle zu entfernen. Dabei wird implizit sichergestellt, dass sich alle Regeln in einer Routingtabelle auf unterschiedliche Datenströme beziehen, um das Hinzufügen widersprüchlicher Regeln zu vermeiden. Jede Routingtabelle erhält außerdem eine Referenz auf das Netzwerkgerät, auf dem sie zum Einsatz kommt.

Der Komponente wird beim Aufruf eine Verteilung übergeben, auf der sie dann Algorithmus 4.14 ausführt. Am Ende des Vorgangs wird dann eine Abbildung an die aufrufende Instanz zurückgegeben, die den beteiligten Netzwerkgeräten aus der IoT-Umgebung ihre jeweiligen Routingtabellen-Objekte zuordnet.

Teilkomponente „MBPFileCreator“

Die Softwareelemente und Shell-Skripte von Operatoren liegen wie in Abschnitt 5.1.1 beschrieben der Komponente `DeploymentExecutor` in Form gewöhnlicher Dateien des Betriebssystems im Verzeichnis `input` vor. Die Aufgabe dieser Teilkomponente ist es, die dort hinterlegten Dateien auszulesen und in Java-Objekte zu transformieren. Diese bestehen aus jeweils zwei Attributen: Das erste speichert den Namen der Datei samt Dateiendung, das zweite den Dateiinhalt. Die Dateien werden auf diese Weise digital abgebildet. Das ist erforderlich, um die Dateien, denen sie entsprechend, anschließend über die Komponente `MBPController` der MBP hinzufügen zu können. Erst dann kann die MBP auf diese Dateien zugreifen und die Dateien zu Adaptern zusammenfassen. In Abbildung 4.14 sind die Dateien, die der MBP bereits verfügbar gemacht worden sind, im Bereich (D) dargestellt. Soll ein Gerät mit diesen Dateien ausgestattet werden, dann legt die MBP auf dem Zielgerät eine neue Datei an, füllt sie mit dem zuvor hinterlegten Dateiinhalt und speichert sie

unter dem angegebenen Dateinamen ab. Die Softwareelemente und Shell-Skripte der Operatoren stehen der MBP nicht in Form von Dateien zur Verfügung, sondern werden lediglich anhand ihres Dateinamens und -inhalts spezifiziert.

Die Schnittstelle, die von der Komponente zur Konvertierung der Dateien angeboten wird, besitzt keine Parameter. Das Auslesen der Dateien aus dem Ordner `input` erfolgt rekursiv, es werden somit auch alle Unterverzeichnisse einbezogen. Für alle auf diese Weise gefundenen Dateien werden dann die entsprechenden Java-Objekte erstellt, die an die aufrufende Instanz zurückgegeben werden.

Als zusätzliche Funktion bietet die Komponente auch eine Schnittstelle an, um Datei-Objekte aus Routingtabellen zu erzeugen. Dafür wird beim Aufruf ein Routingtabellen-Objekt übergeben, welches dann in das digitale Abbild einer Datei überführt wird, das inhaltlich der in Listing 4.3 dargestellten Routingtabelle im JSON-Format entspricht. Das auf diese Weise entstandene Java-Objekt der Datei wird schließlich zurückgegeben. Auch diese Objekte können von dem `MBPController` der MBP als Datei hinzugefügt und zu Adaptern gruppiert werden, sodass die Routingtabellen bei der Provisionierung dann als Dateien auf den Geräten erstellt werden.

Teilkomponente „MBPController“

Der Aufruf der Komponente `DeploymentExecutor` wird an diese Teilkomponente weitergeleitet. Sie übernimmt die Steuerung der MBP und realisiert damit den Programmablauf, der die Ausführung des Datenstrommodels initiiert. Die MBP selbst wird dabei in Form einer Klasse abgebildet. Objekte dieser Klasse erhalten den URL der MBP als Parameter und instanziieren damit den REST-Klienten aus Komponente `RESTClient`. Mit diesem werden dann für Geräte, Adapter und Operatoren jeweils REST-Ressourcen angelegt, die es ermöglichen, neue digitale Abbilder auf der MBP anzulegen oder bestehende abzufragen, zu modifizieren oder zu löschen. Dafür bietet das MBP-Objekt entsprechende Methoden an. Darüber hinaus werden Methoden definiert, um die Verfügbarkeit der MBP unter dem angegebenen URL abzufragen und um die Operatoren mit den Geräten der IoT-Umgebung zu provisionieren. Auf diese Weise wird Teilziel 3.5 aus Abschnitt 3.2 erreicht.

Bei Aufruf beginnt die Komponente damit, auf Grundlage der Verteilung, die als Parameter übergeben wird, die Erstellung der Routingtabellen für die involvierten Geräte zu veranlassen. Diese Aufgabe übernimmt die Komponente `RoutingTableCreator`. Anschließend werden die Geräte und Operatoren der Verteilung in die entsprechenden MBP-Objekte überführt, in das JSON-Format konvertiert und über das MBP-Objekt der MBP als digitale Abbilder hinzugefügt. Die Dateien der Operatoren werden mit Hilfe der Komponente `MBPFileCreator` abgebildet und anschließend ebenfalls der MBP hinzugefügt, wo sie dann zu Adaptern zusammengefasst werden. Dies gilt insbesondere auch für die Routingtabellen und die MQTT-Broker mit den entsprechenden Installationsskripten, die für die Kommunikation der Operatoren untereinander innerhalb der IoT-Umgebung benötigt werden. Die Routinganwendung, die die Routingtabellen liest und auf die an den Geräten eingehenden Nachrichten anwendet, soll an dieser Stelle auch auf die Geräte ausgeliefert werden. Aufgrund des hohen damit verbundenen Implementierungsaufwands konnte sie jedoch im Zuge dieser Bachelorarbeit nicht umgesetzt werden. Die dafür erforderlichen Vorrichtungen wurden allerdings bereits realisiert, sodass die Routinganwendung nach ihrer Fertigstellung direkt in das Provisionierungssystem integriert werden kann. Aus diesem Grund wird Teilziel 3.6 aus Abschnitt 3.2 an dieser Stelle zwar nicht vollständig erfüllt, aber größtenteils.

Nachdem diese Vorbereitungen für die Ausführung abgeschlossen sind, löst diese Komponente zum Schluss auf der MBP für jedes beteiligte Gerät den Provisionierungsvorgang aus, wodurch die Dateien der Operatoren auf die Geräte kopiert und die Installations- und Startskripte im Anschluss ausgeführt werden. Dadurch wird das Datenstrommodell in der IoT-Umgebung in Betrieb genommen.

5.3 Tests

Um gemäß Teilziel 3.7 aus Abschnitt 3.2 überprüfen zu können, ob die Umsetzung wie vorgesehen funktioniert, wurden Testfälle erstellt. Diese umfassen sowohl Modultests, als auch Integrationstests, die nachfolgend beschrieben werden.

5.3.1 Modultests

Zum Test der einzelnen funktionalen Einheiten, aus denen sich die Komponenten zusammensetzen, wurden Testfälle für automatisierte Modultests („Unit-Tests“) entworfen, die unter Verwendung der Java-Bibliothek JUnit ausgeführt werden können. Das Ziel dieser Tests ist es, einzelne Module der entwickelten Bibliothek auf ihre Funktionsfähigkeit zu testen und gleichzeitig Regressionstests zur Verfügung zu stellen, die bei Anpassungen des Quellcodes, beispielsweise im Zuge von Refactorings, Aufschluss darüber geben, ob dadurch die Semantik verändert wurde. Als Metrik zur Qualitätssicherung der Testfälle wird die Zeilenabdeckung verwendet.

Metrik	**Gesamtzahl**	**Abgedeckt**	**Relativ**
Klassenabdeckung:	49	49	100%
Methodenabdeckung:	367	365	99,5%
Zeilenabdeckung:	1944	1898	97,6%

Tabelle 5.2: Testabdeckung des Quellcodes

Insgesamt wurden für die im Rahmen der Umsetzung entwickelte Bibliothek 393 Testfälle erstellt, die eine Zeilenabdeckung von über 97 Prozent des Quellcodes erreichen. Eine detailliertere Übersicht über die erreichte Testabdeckung ist in Tabelle 5.2 dargestellt. Besonders viele Testfälle beziehen sich dabei auf die Verteilungsalgorithmen. Diese werden mehreren sehr unterschiedlichen Testszenarien unterzogen, in denen aus Datenstrommodellen und Modellen von Netzwerktopologien korrekte Verteilungen berechnet werden oder eine Ausnahme geworfen werden soll, falls keine Lösung existiert. Für den Backtracking-Algorithmus existieren zudem mehrere Testfälle, die sicherstellen, dass der Backtracking-Algorithmus bei Verwendung des Modus zur Findung der bestmöglichen Lösung wirklich die optimale Lösung für eine Probleminstanz berechnet.

Auch die Komponente, die Funktionen anbietet, um die MBP zu steuern, wird durch die Testfälle ausgiebig überprüft. Dafür wird die MBP, die im Intranet des IPVS ausgeführt wird, in die automatisierten Tests einbezogen. Ist die MBP auf dem Rechner, auf dem die Testfälle ausgeführt werden, erreichbar, so werden dieser testweise unter anderem digitale Abbilder von Geräten, Operatoren und Dateien hinzugefügt. Mit Hilfe der anderen Methoden dieser Komponente wird dann überprüft, ob das Hinzufügen erfolgreich gewesen ist und die Elemente auf der MBP abgefragt

und ausgelesen werden können. Anschließend werden sie wieder gelöscht, sodass die Testfälle den Zustand der MBP nicht nachhaltig beeinflussen. Auf diese Weise werden alle Funktionen, die die Komponente zur Interaktion mit der MBP zur Verfügung stellt, innerhalb der automatisierten Modultests auf ihre Funktionalität überprüft. Sollte die MBP allerdings nicht erreichbar sein, werden die dazugehörigen Testfälle übersprungen, was dann natürlich auch zu einer insgesamt niedrigeren Testabdeckung führt.

5.3.2 Integrationstests

Mit Hilfe der Integrationstests soll sichergestellt werden, dass die Implementierung tatsächlich auch in der Praxis eingesetzt werden kann und das Zusammenspiel der verschiedenen Komponenten, einschließlich der Anbindung an die MBP, funktioniert. Zur Durchführung der Tests wurden eine virtuelles Maschine und die MBP in einer gemeinsamen IoT-Umgebung der Universität Stuttgart ausgeführt. Es wurde sichergestellt, dass die in dieser Arbeit entwickelte Bibliothek mit der MBP kommunizieren und auf dieser digitale Abbilder von Geräten und Operatoren, sowie Adapter anlegen kann. Die Grundlage dafür bildeten Datenstrommodelle und und Modelle von Netzwerktopologien, die in Form von JSON-Dateien zur Verfügung gestellt wurden. Auch die korrekte Arbeitsweise des Vorgangs, der aus Dateien digitale Abbildungen erzeugt und diese der MBP hinzufügt, um sie zu Adaptern zusammenzufassen, konnte überprüft werden. Für die gegebene virtuelle Maschine wurde zudem die Provisionierung und Ausführung von Softwareelementen eines Operators erfolgreich getestet. Da im Rahmen dieser Arbeit eine Routinganwendung nur konzipiert, aber nicht implementiert wurde, konnte die Funktionsweise des Kommunikationskonzepts in seiner Gesamtheit selbst nicht überprüft werden. Allerdings war der Test von Teilfunktionen dieses Konzepts möglich, wie die Ausbringung und Installation von MQTT-Brokern, die Erzeugung und Auslieferung von Routingtabellen, sowie die Installation, Inbetriebnahme, Terminierung und Deinstallation der Softwareelemente eines Operators durch die dafür vorgesehenen Shell-Skripte.

Zusammenfassend kann festgestellt werden, dass während der Integrationstests keine Ereignisse beobachtet werden konnten, die darauf schließen lassen würden, dass die im Rahmen dieser Arbeit entwickelten Konzepte und deren Umsetzung bei vollumfänglicher Ausführung in der Praxis nicht wie vorgesehen funktionieren würden.

6 Verwandte Arbeiten

In diesem Kapitel werden Arbeiten beschrieben, die zu dieser Bachelorarbeit verwandt sind. Sie behandeln entweder ähnliche Themengebiete oder dienen als Grundlage für die Ergebnisse dieser Arbeit. Der erste Abschnitt dieses Kapitels behandelt dabei Arbeiten, die sich auf Konzepte zur Verteilung von Operatoren eines Datenstrommodells auf IoT-Geräte beziehen, der zweite Abschnitt solche, die thematisch der praktischen Ausführung eines Datenstrommodells in einer IoT-Umgebung zuzuordnen sind.

Den Ausgangspunkt dieser Arbeit stellt das Werkzeug FlexMash dar, das die Modellierung von Datenstrommodellen ermöglicht und im Rahmen dieser Bachelorarbeit konzeptionell erweitert wird. Insofern bildet die Arbeit von Hirmer et al. [HB17] die allgemeine Grundlage für die an dieser Stelle entstandenen Resultate.

6.1 Verteilung von Operatoren

Für die Verteilung von Operatoren eines Datenstrommodells auf Geräte einer IoT-Umgebung wurden in der Literatur bereits zahlreiche unterschiedliche Strategien vorgeschlagen. Diese unterscheiden sich zumeist in der Zielsetzung, die den Konzepten zugrunde liegt. Die in dieser Bachelorarbeit vorgestellten Verteilungsalgorithmen sind lediglich zur Lösung des Verteilungsproblems für eine konkrete IoT-Umgebung konzipiert und haben die Aufgabe, die durch den Datentransport verursachte Netzwerkauslastung zu reduzieren, unter Berücksichtigung der Anforderungen der Operatoren und Datenströme. Im Gegensatz dazu erheben die nachfolgend genannten Arbeiten den Anspruch, das Verteilungsproblem vor einem allgemeineren, projektunabhängigen Hintergrund zu lösen.

Ebenfalls mit dem Ziel, die Netzwerkauslastung zu reduzieren, schlagen Rizou et al. [RDR10] einen heuristischen Algorithmus vor, um Verteilungen zu ermitteln, bei denen die Datenströme zwischen den Operatoren ein möglichst geringes Verzögerungs-Bandbreiten-Produkt [JB88] aufweisen. Dafür wird zunächst eine optimale Lösung in einem kontinuierlichen, über der Latenz der Verbindungen aufgespannten Suchraum berechnet, die anschließend auf das physische Netzwerk abgebildet wird. Der Algorithmus wird dabei nicht auf einer zentralen Instanz ausgeführt, wie es dem in dieser Bachelorarbeit vorgestellten Konzept der Fall ist, sondern verteilt auf den Geräten des jeweiligen Netzwerks. Da dieser Ansatz nicht speziell auf das IoT und dessen Eigenschaften ausgelegt ist erfordert er allerdings, dass sich die in das Netzwerk integrierten Geräte untereinander kennen, um Informationen über die Latenz der Verbindungen austauschen und den Suchraum aufspannen zu können [RDR10]. Dies führt zu einer engeren Kopplung zwischen den Geräten, die im Rahmen der Problemstellung, die dieser Bachelorarbeit zugrunde liegt, nicht erwünscht ist. Weiter berücksichtigt der Vorschlag von Rizou et al. keine Anforderungen, die Operatoren oder Datenströme an die Infrastruktur stellen, auf der sie betrieben werden. Für IoT-Umgebungen ist dies jedoch aufgrund der Heterogenität der zum Einsatz kommenden Geräte zwingend erforderlich.

Die konkrete Definition und Implementierung der Netzwerkdistanzen zwischen zwei IoT-Geräten, nach denen die Algorithmen die Verteilungen optimieren, wird im Rahmen dieser Arbeit dem Benutzer und dessen kontextabhängigen Anforderungen überlassen. Damit können viele verschiedene Anwendungsfälle abgedeckt werden, allerdings keine, bei denen nach bestimmten Leistungskriterien von Operatoren optimiert werden soll. Außerdem sind die Algorithmen nicht in besonderem Maße auf die von dem Benutzer verwendeten (physikalischen) Eigenschaften spezialisiert und können damit unter Umständen existierende Spezialfälle nicht berücksichtigen. Im Gegensatz dazu gibt es in der Literatur neben der bereits erwähnten Arbeit von Rizou et al. [RDR10] weitere Vorschläge für Verteilungsalgorithmen, die auf bestimmte Optimierungskriterien festgelegt und spezialisiert sind. So werden Operatorverteilungen in der Arbeit von Xing et al. [XHÇZ06] beispielsweise nach der Systemauslastung optimiert, Amini et al. [AJS+06] schlagen eine Verteilung auf Basis der Wichtigkeit der Operatoren vor. Auch diesen Ansätzen ist jedoch gemein, dass sie für IoT-Umgebungen ungeeignet sind, da sie keine Möglichkeit bieten, um Anforderungen von Operatoren und Datenströmen zu berücksichtigen.

Cipriani et al. [CSM11] stellen ein Framework namens „M-TOP“ vor, das es ebenfalls ermöglicht, Operatoren eines Datenstrommodells auf Geräte eines Netzwerks abzubilden. Anders als bei den vorherigen Arbeiten können bei diesem Konzept allerdings mehrere Ziele definiert werden, nach denen die Verteilungen optimiert werden sollen. Dies geschieht in Form von Quality of Services: Ein Benutzer hat die Möglichkeit, M-TOP eine Menge von anwendungsspezifischen Quality of Services zu übergeben, für die M-TOP dann Verteilungen ermittelt, die diese Ziele erfüllen. Dabei kommen Heuristiken zum Einsatz, die Verteilungen eliminieren, die zu keinen passenden Lösungen führen. M-TOP erlaubt es außerdem, Operatoren mit Anforderungen und Geräte mit Eigenschaften zu annotieren. Die Grundidee dahinter ist dabei dieselbe wie hinter dem Anforderungs- und Eigenschaftskonzept dieser Arbeit: Damit ein Operator auf einem Gerät platziert werden kann, muss das Gerät auch bei M-TOP zu dem Operator kompatibel sein und dessen Anforderungen erfüllen. Allerdings sind Vergleiche zwischen Anforderungen und Eigenschaften bei M-TOP nur sehr eingeschränkt möglich: Die Werte von gleichnamigen Anforderungen und Eigenschaften werden jeweils nur auf Gleichheit geprüft, es gibt anders als bei der in Abschnitt 4.2.2 vorgestellten Lösung keine Möglichkeit, um andere Vergleichsoperatoren zu verwenden, wie beispielsweise „kleiner“ oder „ungleich“. Außerdem existiert unter M-TOP für Anforderungen von Operatoren kein Konzept, das dem der konsumierend-wirkenden Anforderungen dieser Arbeit entspricht. Damit können keine Anforderungen von Operatoren spezifiziert werden, die die Eigenschaften des Geräts, auf dem sie ausgeführt werden, zur Laufzeit beeinflussen.

In dem Artikel „An Edge-Focused Model for Distributed Streaming Data Applications“ [BHM18] wird eine graphen-basierte Modellierungssprache namens Cresco Application Model (CAM) vorgeschlagen, um verteilte Datenstrom-Anwendungen zu beschreiben. Die damit erzeugten Modelle ähneln den Datenstrommodellen, die in dieser Bachelorarbeit verwendet werden, enthalten aber weitere Informationen über die Eigenschaften und das Verhalten der Operatoren und Datenströme. Darauf aufbauend wird ähnlich zu den Ausführungen in Abschnitt 4.4.1 das Verteilungsproblem für Operatoren formuliert. Um analog zu den Konzepten aus Abschnitt 4.2 zu ermitteln, welche Geräte die zur Ausführung bestimmter Operatoren benötigten Ressouren besitzen, wird in diesem Artikel ein Modell vorgeschlagen, das mit einem Ungleichungssystem arbeitet, in welchem die Ressourcen quantitativ repräsentiert werden. Mit Hilfe von binären Entscheidungsdiagrammen werden aus diesem System dann zulässige Verteilungen berechnet. Darüber hinaus wird auch die Funktionsweise eines Greedy-Algorithmus erläutert, der die Operatoren so zuweist, dass

eine möglichst optimale Auslastung erreicht wird. Der Greedy-Algorithmus, der in dieser Arbeit vorgeschlagen wurde, optimiert die Verteilungen dagegen hinsichtlich der Netzwerkdistanzen, um möglichst kurze Transportwege für Daten zu erreichen.

6.2 Ausführung von Datenstrommodellen

Im Zusammenhang mit der praktischen Ausführung von Datenstrommodellen in IoT-Umgebungen und den daraus resultierenden Problemstellungen und Herausforderungen sind eine Reihe von Arbeiten entstanden, wovon einige die Grundlagen für diese Bachelorarbeit legten.

In dem Artikel „Automating the Provisioning and Configuration of Devices in the Internet of Things“ [HBS+16] setzen sich Hirmer et al. mit dem dynamischen Binden von Netzwerkgeräten und der Provisionierung durch diese auseinander. Um IoT-Geräte einer Netzwerkumgebung automatisch konfigurieren und mit ihnen Dienste provisionieren zu können, wurden eine Systemarchitektur und ein mehrstufige Methode entworfen. Innerhalb dieser wird ein digitales Abbild der IoT-Umgebung angefertigt, die darin enthaltenen digitalen Abbilder der Geräte mit den physischen Geräten verknüpft und Vorkehrungen getroffen, um auf die Geräte zugreifen zu können. Auf Basis dieser Konzepte wurde die MBP entwickelt, die einen zentralen Baustein für die in dieser Bachelorarbeit vorgestellten Lösungen und deren Implementierung bildet.

Franco da Silva et al. stellen in dem Artikel „An Approach for CEP Query Shipping to Support Distributed IoT Environments“ [FHM18] ein generelles Konzept vor, um Daten in IoT-Umgebungen verteilt zu verarbeiten. Dabei wird insbesondere der methodische Ablauf beschrieben, der als Grundlage für den in Abschnitt 4.1 erläuterten Ansatz dient und in dieser Arbeit vertieft, erweitert und implementiert wurde. Darüber hinaus wird die MBP in diesem Artikel eingeführt und ihre Rolle innerhalb der IoT-Umgebung erläutert.

7 Fazit und Ausblick

In diesem Kapitel wird ein abschließendes Fazit über die verfasste Bachelorarbeit gezogen und im Anschluss ein Ausblick auf mögliche zukünftige Arbeiten gegeben.

7.1 Fazit

Ziel dieser Bachelorarbeit war es, Konzepte zu entwickeln und umzusetzen, die es ermöglichen, Geschäftslogik, die in Form von Datenstrommodellen spezifiziert ist, verteilt in IoT-Umgebungen auszuführen. Dabei sollte anhand von Anforderungen dynamisch entschieden werden, auf welchen Geräten die Datenoperationen ausgeführt werden, um sowohl eine korrekte Ausführung der Geschäftslogik zu gewährleisten, als auch die Transportwege der Daten zu minimieren.

Diese Aufgabenstellung konnte durch die im Rahmen dieser Arbeit entstandenen Lösungen größtenteils erfüllt werden. Dafür wurden zuerst Konzepte entwickelt, um Anforderungen von Operatoren und Datenströmen und Eigenschaften von Geräten und Netzwerkverbindungen zu modellieren. Dabei wurde berücksichtigt, dass Anforderungen sehr verschiedene Ausprägungen besitzen und Eigenschaften durch die Ausführung von Operatoren beeinflusst werden können, wofür das Konzept konsumierend-wirkender Anforderungen eingeführt wurde. Auf Grundlage dieser Konzepte wurden Algorithmen eingeführt, um automatisiert testen zu können, ob eine Menge von Eigenschaften eine Menge von Anforderungen erfüllt. Im Anschluss wurden die Schemata von Datenstrommodellen und Modellen von Netzwerktopologien erweitert, sodass in diese sowohl Anforderungen und Eigenschaften, als auch weitere für die Ausführung erforderliche Daten integriert werden können. Um die Entwicklung von Verteilungsalgorithmen zu ermöglichen, die Operatoren auf die Geräte einer IoT-Umgebung verteilen, wurde zunächst das Verteilungsproblem definiert und mitsamt seiner Ein- und Ausgaben formalisiert. Dabei wurde auch festgelegt, wodurch sich bestmögliche Lösungen für Instanzen des Verteilungsproblems auszeichnen. Darauf aufbauend wurden zwei Algorithmen konzipiert, um das Verteilungsproblem zu lösen: Ein Greedy-Algorithmus und ein Backtracking-Algorithmus, die sich jeweils in ihren Eigenschaften unterscheiden. Mit Hilfe des Backtracking-Algorithmus können bestmögliche Lösungen berechnet werden. Um die Datenstrommodelle gemäß der von den Algorithmen berechneten Verteilung in IoT-Umgebungen ausführen zu können, wurde ein Ausführungskonzept entwickelt. Dieses nutzt die MBP, um Softwareelemente von Operatoren mit den IoT-Geräten zu provisionieren. Zur Realisierung der Kommunikation zwischen den Operatoren wurde ein Konzept auf Basis von MQTT-Brokern erarbeitet, über die unter der Verwendung von Topics Daten zwischen den Operatoren ausgetauscht werden können. Damit Daten von Operatoren auch über mehrere Netzwerkgeräte hinweg übermittelt werden können, wurden darüber hinaus Routingtabellen als Teil der konzeptionellen Lösung eingeführt und ein Algorithmus entworfen, der diese aus Operatorverteilungen erzeugen kann.

Diese Konzepte wurden anschließend in der Programmiersprache Java implementiert und getestet. Lediglich die in Abschnitt 4.5.3 beschriebene Routinganwendung konnte aufgrund des großen damit verbundenen Aufwands im Rahmen dieser Arbeit nur konzipiert, aber nicht implementiert werden. Folglich konnten alle der in Kapitel 3 genannten Ziele und Teilziele erreicht werden, mit Ausnahme von Teilziel 3.6, das nur weitestgehend erfüllt wurde.

7.2 Ausblick

Neben den implementierten Konzepten wurden Ideen entwickelt, die aufgrund des großen Aufwands nicht oder nur unvollständig umgesetzt werden konnten, jedoch für zukünftige Arbeiten und Weiterentwicklungen in Betracht gezogen werden können. Diese werden nachfolgend beschrieben.

7.2.1 Implementierung der Routinganwendung

In Abschnitt 4.5.3 wird eine Routinganwendung konzipiert, die es ermöglicht, auf Basis der Routingtabellen den Datenaustausch zwischen Operatoren über mehrere IoT-Geräte hinweg sicherzustellen. Das ist erforderlich, wenn die Netzwerkpfade, auf die die Datenströme in den Verteilungen abgebildet werden, aus mehreren Abschnitten bestehen und eine direkte Kommunikation zwischen den Geräten deshalb nicht möglich ist. Der Algorithmus zur Erzeugung der Routingtabellen und die Auslieferung dieser auf die entsprechenden Geräte wurde in dieser Arbeit implementiert, die Umsetzung der Routinganwendung ist allerdings noch ausstehend und kann für zukünftige Arbeiten in Betracht gezogen werden.

7.2.2 Erweiterung der Schemata der Eingabemodelle

Die in dieser Bachelorarbeit vorgestellten Verteilungsalgorithmen berücksichtigen nicht nur Anforderungen von Operatoren, sondern auch Anforderungen von Datenströmen. Diese werden dann mit den Eigenschaften der Netzwerkverbindungen verglichen, über die ihre Daten transportiert werden sollen. Diese Funktion geht über die ursprüngliche Zielsetzung der Arbeit hinaus und erweitert damit die Menge der Anwendungsfälle der vorgeschlagenen Konzepte. Gemäß der in Abschnitt 4.3 beschriebenen Schemata erlauben allerdings weder die Datenstrommodelle, noch die Modelle der Netzwerktopologien eine Annotation der Datenströme und Netzwerkverbindungen um Anforderungen beziehungsweise Eigenschaften. Für die Modelle der Netzwerktopologien existiert im Gegensatz zu den Datenstrommodellen derzeit noch keine Möglichkeit, um Netzwerkverbindungen zwischen Geräten explizit abzubilden. Dies führt dazu, dass von den in dieser Arbeit vorgestellten Konzepten in der Praxis noch nicht vollumfänglich Gebrauch gemacht werden kann. Daher kann es das Ziel zukünftiger Arbeiten sein, die Schemata dieser Modelle entsprechend zu erweitern und die Implementierung der Einlesekomponenten anzupassen.

7.2.3 Monitoring

Die Entscheidungen der Verteilungsalgorithmen dieser Arbeit basieren auf den Anforderungen von Operatoren und Datenströmen, den Eigenschaften von Netzwerkgeräten und deren Verbindungen und den Netzwerkdistanzen der Verbindungen. Dabei handelt es sich jeweils um statische Charakteristiken, die vor Ausführung der Algorithmen und der Ausbringung der Operatoren manuell in den jeweiligen Modellen festgelegt werden. Das praktische Laufzeitverhalten in der IoT-Umgebung kann jedoch unter Umständen stark von diesen Annahmen abweichen, beispielsweise aufgrund von dynamischen und nur schwer vorhersehbaren Seiteneffekten. Aus diesem Grund ist es sinnvoll, auch Erfahrungswerte aus früheren Ausführungen in die Verteilungsentscheidungen einfließen zu lassen. Dafür werden technische Mittel benötigt, um ein Monitoring der Geräte praktizieren und ihr Verhalten damit während der Ausführung beobachten zu können. Auf Grundlage der auf diese Weise gewonnen Daten kann dann evaluiert werden, welche der von den Verteilungsalgorithmen vorgeschlagenen Operatorverteilungen am ehesten zur zukünftigen Ausführung eines Datenstrommodells geeignet sind. Um dies zu realisieren, können gegebenenfalls Methoden des Maschinellen Lernens [NR12] eingesetzt werden. Der Entwurf derartiger Techniken und Konzepte zur Verbesserung der Verteilungsalgorithmen bedarf weiterer Forschung und kann damit zum Gegenstand zukünftiger Arbeiten in diesem Bereich werden.

Anhang

A.1 Beispiel für ein mit FlexMash erzeugtes Datenstrommodell

```
{
   "nodes":[
      {
         "guiId":"Quelle",
         "serviceId":"Quellen-Software",
         "properties":[ ],
         "target":[
            "Operator1"
         ]
      },
      {
         "guiId":"Operator1",
         "serviceId":"Operator-Software1",
         "properties":[ ],
         "target":[
            "Operator2",
            "Senke"
         ]
      },
      {
         "guiId":"Senke",
         "serviceId":"Senken-Software",
         "properties":[ ],
         "target":[ ]
      },
      {
         "guiId":"Operator2",
         "serviceId":"Operator-Software2",
         "properties":[ ],
         "target":[
            "Senke"
         ]
      }
   ]
}
```

Listing A.1: Beispiel für ein durch FlexMash erzeugtes Datenstrommodell

A.2 Erweiterung des Schemas für Netzwerktopologien

```
{
   "@context":{
      "owl":"http://www.w3.org/2002/07/owl#",
      "rdf":"http://www.w3.org/1999/02/22-rdf-syntax-ns#",
      "rdfs":"http://www.w3.org/2000/01/rdf-schema#",
      "ssn":"http://purl.oclc.org/NET/ssnx/ssn#",
      "xsd":"http://www.w3.org/2001/XMLSchema#",
      "iot-lite":"http://purl.oclc.org/NET/UNIS/fiware/iot-lite#",
      "ipvs":"http://www.ipvs.uni-stuttgart.de/iot-lite#"
   },
   "@graph":[
      {
         "@id":"ipvs:macAddress",
         "@type":"owl:DatatypeProperty",
         "rdfs:domain":{
            "@id":"ssn:Device"
         },
         "rdfs:range":{
            "@id":"xsd:string"
         }
      },
      {
         "@id":"ipvs:ipAddress",
         "@type":"owl:DatatypeProperty",
         "rdfs:comment":"IP address of the device in the common network.",
         "rdfs:domain":{
            "@id":"ssn:Device"
         },
         "rdfs:range":{
            "@id":"xsd:string"
         }
      },
      {
         "@id":"ipvs:username",
         "@type":"owl:DatatypeProperty",
         "rdfs:comment":"Username on the device's operating system which can be used
             in order to connect to the device via SSH.",
         "rdfs:domain":{
            "@id":"ssn:Device"
         },
         "rdfs:range":{
            "@id":"xsd:string"
         }
      },
```

```
44      {
45         "@id":"ipvs:DeviceCapability",
46         "@type":"owl:Class",
47         "rdfs:comment":"Hardware or software capability of the device."
48      },
49      {
50         "@id":"ipvs:CapabilityName",
51         "@type":"owl:DatatypeProperty",
52         "rdfs:comment":"Distinct and unique name (key) of the capability.",
53         "rdfs:domain":{
54            "@id":"iot-lite:DeviceCapability"
55         },
56         "rdfs:range":{
57            "@id":"xsd:string"
58         }
59      },
60      {
61         "@id":"ipvs:CapabilityValue",
62         "@type":"owl:DatatypeProperty",
63         "rdfs:comment":"Value of the capability which is either a boolean, integer,
               double or string.",
64         "rdfs:domain":{
65            "@id":"iot-lite:DeviceCapability"
66         }
67      },
68      {
69         "@id":"ipvs:hasCapabilities",
70         "@type":"owl:DatatypeProperty",
71         "rdfs:comment":"Lists the capabilities which are specified for this device.",
72         "rdfs:domain":{
73            "@id":"ipvs:Device"
74         },
75         "rdfs:range":{
76            "@id":"ipvs:DeviceCapability"
77         }
78      }
79   ]
80 }
```

Listing A.2: Erweiterung des Schemas für Netzwerktopologien

Literaturverzeichnis

[Adv13] O. for the Advancement of Structured Information Standards. *OASIS Message Queuing Telemetry Transport (MQTT) TC*. 2013. URL: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=mqtt (zitiert auf S. 89).

[AJS+06] L. Amini, N. Jain, A. Sehgal, J. Silber, O. Verscheure. „Adaptive Control of Extreme-scale Stream Processing Systems“. In: *Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*. ICDCS '06. Washington, DC, USA: IEEE Computer Society, 2006, S. 71–. ISBN: 0-7695-2540-7. DOI: 10.1109/ICDCS.2006.13. URL: https://doi.org/10.1109/ICDCS.2006.13 (zitiert auf S. 114).

[ARA12] M. R. Alam, M. B. I. Reaz, M. A. M. Ali. „A Review of Smart Homes—Past, Present, and Future“. In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42.6 (Nov. 2012), S. 1190–1203. DOI: 10.1109/tsmcc.2012.2189204 (zitiert auf S. 17).

[BBD+02] B. Babcock, S. Babu, M. Datar, R. Motwani, J. Widom. *Models and Issues in Data Stream Systems*. Technical Report 2002-19. Stanford InfoLab, 2002. URL: http://ilpubs.stanford.edu:8090/535/ (zitiert auf S. 27, 28).

[BEBT16] M. Bermudez-Edo, T. Elsaleh, P. Barnaghi, K. Taylor. „IoT-Lite: A Lightweight Semantic Model for the Internet of Things“. In: *2016 Intl IEEE Conferences on Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld)*. IEEE, Juli 2016. DOI: 10.1109/uic-atc-scalcom-cbdcom-iop-smartworld.2016.0035 (zitiert auf S. 55).

[BG14] A. Banks, R. Gupta. „MQTT Version 3.1. 1“. In: *OASIS standard* 29 (2014) (zitiert auf S. 89).

[BHM18] C. Bumgardner, C. Hickey, V. Marek. „An Edge-Focused Model for Distributed Streaming Data Applications“. In: *PerFoT'18 - International Workshop on Pervasive Flow of Things*. 2018, S. 776–781 (zitiert auf S. 114).

[BMZA12] F. Bonomi, R. Milito, J. Zhu, S. Addepalli. „Fog computing and its role in the internet of things“. In: *Proceedings of the first edition of the MCC workshop on Mobile cloud computing - MCC '12*. ACM Press, 2012. DOI: 10.1145/2342509.2342513 (zitiert auf S. 24).

[Bra17] T. Bray. *The javascript object notation (json) data interchange format*. Techn. Ber. 2017 (zitiert auf S. 52).

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. *Introduction to algorithms second edition*. The MIT Press, 2001 (zitiert auf S. 64, 65).

[CM12] G. Cugola, A. Margara. „Processing Flows of Information: From Data Stream to Complex Event Processing“. In: *ACM Computing Surveys* 44.3 (Juni 2012), S. 1–62. ISSN: 0360-0300. DOI: 10.1145/2187671.2187677. URL: http://doi.acm.org/10.1145/2187671.2187677 (zitiert auf S. 17, 28).

[CSM11] N. Cipriani, O. Schiller, B. Mitschang. „M-TOP“. In: *Proceedings of the 15th Symposium on International Database Engineering & Applications - IDEAS '11*. ACM Press, 2011. DOI: 10.1145/2076623.2076631 (zitiert auf S. 114).

[Dij59] E. W. Dijkstra. „A note on two problems in connexion with graphs“. In: *Numerische Mathematik* 1.1 (Dez. 1959), S. 269–271. DOI: 10.1007/bf01386390 (zitiert auf S. 68).

[FC10] M. Friedemann, F. Christian. „Vom Internet der Computer zum Internet der Dinge“. In: *Informatik-Spektrum* 33.2 (Feb. 2010), S. 107–108. DOI: 10.1007/s00287-010-0417-7 (zitiert auf S. 23, 25).

[FHM18] A. C. Franco da Silva, P. Hirmer, B. Mitschang. „An Approach for CEP Query Shipping to Support Distributed IoT Environments“. In: *Proceedings of the 14th Workshop on Context and Activity Modeling and Recognition (COMOREA)*. IEEE Percom, 2018. Kap. An Approach for CEP Query Shipping to Support Distributed IoT Environments (zitiert auf S. 18, 32, 115).

[FT00] R. Fielding, R. Taylor. *Architectural styles and the design of network-based software architectures*. Bd. 7. University of California, Irvine Doctoral dissertation, 2000. Kap. Chapter 5: Representational State Transfer (REST) (zitiert auf S. 35).

[FT08] E. Fleisch, F. Thiesse. *Internet der Dinge*. In: N. Gronau, E. Weber. *Enzyklopädie der Wirtschaftsinformatik – Online-Lexikon*. Lehrstuhl für Wirtschaftsinformatik (insb. Prozesse und Systeme), 2008. Kap. Technische Grundlagen. URL: http://www.enzyklopaedie-der-wirtschaftsinformatik.de/lexikon/technologien-methoden/Rechnernetz/Internet/Internet-der-Dinge (besucht am 26. 06. 2018) (zitiert auf S. 25).

[Gar17] I. Gartner. *Gartner Says 8.4 Billion Connected "Things"Will Be in Use in 2017, Up 31 Percent From 2016*. Hrsg. von R. van der Meulen. 7. Feb. 2017. URL: https://www.gartner.com/newsroom/id/3598917 (besucht am 01. 07. 2018) (zitiert auf S. 17).

[HB17] P. Hirmer, M. Behringer. „FlexMash 2.0 – Flexible Modeling and Execution of Data Mashups“. In: *Communications in Computer and Information Science*. Springer International Publishing, 2017, S. 10–29. DOI: 10.1007/978-3-319-53174-8_2 (zitiert auf S. 28, 31, 113).

[HBS+16] P. Hirmer, U. Breitenbücher, A. C. F. da Silva, K. Képes, B. Mitschang, M. Wieland. „Automating the Provisioning and Configuration of Devices in the Internet of Things“. Englisch. In: *Complex Systems Informatics and Modeling Quarterly* 9.9 (Dez. 2016), S. 28–43. ISSN: 2255-9922. DOI: 10.7250/csimq.2016-9.02. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=ART-2016-23&engl=0 (zitiert auf S. 115).

[ITU12] ITU. *ITU-T Recommendation Y.2060: Overview of the Internet of things*. Techn. Ber. Internationale Fernmeldeunion, Juni 2012, S. 1. URL: http://handle.itu.int/11.1002/1000/11559 (zitiert auf S. 23).

[JB88] V. Jacobsen, R. Braden. *TCP extensions for long-delay paths*. RFC 1072. Okt. 1988. DOI: 10.17487/RFC1072. URL: https://rfc-editor.org/rfc/rfc1072.txt (zitiert auf S. 113).

[Joc13] H. L. Jochen Ludewig. *Software Engineering*. Dpunkt.Verlag GmbH, 2. Mai 2013. ISBN: 3864900921. URL: https://www.ebook.de/de/product/20483151/jochen_ludewig_horst_lichter_software_engineering.html (zitiert auf S. 36, 89, 102).

[Kah62] A. B. Kahn. „Topological sorting of large networks“. In: *Communications of the ACM* 5.11 (Nov. 1962), S. 558–562. DOI: 10.1145/368996.369025 (zitiert auf S. 67).

[LFK+14] H. Lasi, P. Fettke, H.-G. Kemper, T. Feld, M. Hoffmann. „Industry 4.0“. In: *Business & Information Systems Engineering* 6.4 (Juni 2014), S. 239–242. DOI: 10.1007/s12599-014-0334-4 (zitiert auf S. 17).

[LME+15] P. G. Lopez, A. Montresor, D. Epema, A. Datta, T. Higashino, A. Iamnitchi, M. Barcellos, P. Felber, E. Riviere. „Edge-centric Computing“. In: *ACM SIGCOMM Computer Communication Review* 45.5 (Sep. 2015), S. 37–42. DOI: 10.1145/2831347.2831354 (zitiert auf S. 19).

[Meu95] R. Meunier. „Pattern Languages of Program Design“. In: Hrsg. von J. O. Coplien, D. C. Schmidt. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1995. Kap. The Pipes and Filters Architecture, S. 427–440. ISBN: 0-201-60734-4. URL: http://dl.acm.org/citation.cfm?id=218662.218694 (zitiert auf S. 28).

[MG11] P. Mell, T. Grance. *The NIST definition of cloud computing*. Techn. Ber. 2011. DOI: 10.6028/nist.sp.800-145 (zitiert auf S. 17).

[MSPC12] D. Miorandi, S. Sicari, F. D. Pellegrini, I. Chlamtac. „Internet of things: Vision, applications and research challenges“. In: *Ad Hoc Networks* 10.7 (Sep. 2012), S. 1497–1516. DOI: 10.1016/j.adhoc.2012.02.016 (zitiert auf S. 17, 23, 24, 26).

[NR12] P. Norvig, S. Russell. *Künstliche Intelligenz*. Pearson Studium, 1. Juni 2012, S. 83. 1307 S. ISBN: 3868940987. URL: https://www.ebook.de/de/product/19148036/peter_norvig_stuart_russell_kuenstliche_intelligenz.html (zitiert auf S. 119).

[RDR10] S. Rizou, F. Durr, K. Rothermel. „Solving the Multi-Operator Placement Problem in Large-Scale Operator Networks“. In: *2010 Proceedings of 19th International Conference on Computer Communications and Networks*. IEEE, Aug. 2010. DOI: 10.1109/icccn.2010.5560127 (zitiert auf S. 113, 114).

[Rot16] A. Roth. *Einführung und Umsetzung von Industrie 4.0*. Springer Berlin Heidelberg, 8. Feb. 2016, S. 26–29. URL: https://www.ebook.de/de/product/25702168/einfuehrung_und_umsetzung_von_industrie_4_0.html (zitiert auf S. 25).

[SS14] G. Saake, K.-U. Sattler. *Algorithmen und Datenstrukturen: Eine Einführung mit Java*. dpunkt. verlag, 2014. Kap. Algorithmenmuster: Greedy, S. 213–218, 222–230 (zitiert auf S. 64, 74).

[TGNO92] D. Terry, D. Goldberg, D. Nichols, B. Oki. „Continuous Queries over Append-only Databases“. In: *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*. SIGMOD '92. San Diego, California, USA: ACM, 1992, S. 321–330. ISBN: 0-89791-521-6. DOI: 10.1145/130283.130333. URL: http://doi.acm.org/10.1145/130283.130333 (zitiert auf S. 28).

[VF13] O. Vermesan, P. Friess. *Internet of Things: Converging Technologies for Smart Environments and Integrated Ecosystems*. RIVER PUBL, 2. Juli 2013, S. 39–41. 364 S. ISBN: 8792982735. URL: https://www.ebook.de/de/product/21022366/internet_of_things_converging_technologies_for_smart_environments_and_integrated_ecosystems.html (zitiert auf S. 17, 25).

[WDA15] J. West, T. Dean, J. Andrews. *Network+ guide to networks*. Cengage Learning, 2015, S. 274 (zitiert auf S. 57).

[WR15] L. Wang, R. Ranjan. „Processing distributed internet of things data in clouds“. In: *IEEE Cloud Computing* 2.1 (2015), S. 76–80 (zitiert auf S. 18).

[XHÇZ06] Y. Xing, J.-H. Hwang, U. Çetintemel, S. Zdonik. „Providing Resiliency to Load Variations in Distributed Stream Processing“. In: *Proceedings of the 32Nd International Conference on Very Large Data Bases*. VLDB '06. Seoul, Korea: VLDB Endowment, 2006, S. 775–786. URL: http://dl.acm.org/citation.cfm?id=1182635.1164194 (zitiert auf S. 114).

Alle URLs wurden zuletzt am 31. Juli 2018 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