Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Execution of data flow models in distributed IoT environments

Daniel Del Gaudio

Course of Study:	Informatik
Examiner:	Prof. Dr.-Ing. habil. Bernhard Mitschang
Supervisor:	Dr. rer. nat. Pascal Hirmer

Commenced: May 7, 2018

Completed: November 7, 2018

Abstract

The Internet of Things is an emerging technology, driven by combining the physical world with the cyberspace. The IoT enables new approaches such as, smart homes, smart factories and smart cities. An ability of such IoT environments is to immediately react to changing conditions, i.e., situations. Situation recognition can be implemented, for example, by defining and executing data flow models. The state of the art for the execution of data flow models is to utilize an execution engine, typically running in the cloud. Data is transmitted from devices to the engine to be processed. This solution has many disadvantages, like, for example, communication overhead, a single point of failure and long distances for data transfer. Since IoT devices are equipped with processing power themselves, data does not necessarily have to be sent to the cloud, but can be processed on the devices themselves. It can be transmitted directly between devices and does not have to travel the long detour to the cloud and back to the IoT environment. Consequently, a better solution is to execute the data flow model directly on the IoT devices, without using a centralized execution engine. To execute data flow models in distributed IoT environments, in this Master thesis, I propose a lifecycle method with five steps: (i) the modeling of the data flow, (ii) the creation or modification of the network topology, (iii) the execution of the data flow model, (iv) the device redistribution and (v) the retirement of the data flow. As a proof-of-concept and for evaluation purposes, a prototype has been implemented.

Contents

1 Introduction 11
1.1 State of the art approach for the execution of data flow models 11
1.2 Solution and goals of this thesis 12

2 Fundamentals 15
2.1 Data flows 15
2.1.1 Data flow modeling 15
2.1.2 Data flow patterns 16
2.1.3 Complex event processing 16
2.2 Internet of Things 17
2.2.1 Cyber-physical systems 18
2.2.2 Constrained Application Protocol 18
2.3 Peer-to-peer Systems 18
2.4 Publish-subscribe 19

3 Related Work 21

4 Execution of data flow models in distributed IoT environments 25
4.1 Lifecycle method overview 27
4.2 Lifecycle method step 1: Data flow modeling 28
4.3 Lifecycle method step 2: Network topology creation or modification . . . 30
4.4 Lifecycle method step 3: Data flow execution 33
4.4.1 Message format for data exchange 33
4.4.2 Architecture overview 34
4.4.3 Communication protocol for message exchange 37
4.4.4 Message flow 38
4.5 Lifecycle method step 4: Device redistribution 43
4.6 Lifecycle method step 5: Data flow retirement 44
4.7 Optimizations 45
4.7.1 Robustness optimizations 45
4.7.2 Performance optimizations 47

5 Implementation 49
5.1 Infrastructure layer . . . 49
5.1.1 Incoming message repository . . . 49
5.1.2 Outgoing message repository . . . 50
5.1.3 Other repositories . . . 51
5.1.4 Application wrapper . . . 51
5.1.5 Resource monitor . . . 52
5.2 Domain Layer . . . 52
5.2.1 Message and message forwarding manager . . . 52
5.2.2 Services . . . 52
5.3 Interface layer . . . 53
5.3.1 REST interface . . . 53
5.3.2 Hello forwarder . . . 54
5.3.3 MBP adapter . . . 56
5.3.4 Message producer . . . 56
5.4 Special cases . . . 56

6 Evaluation 59

7 Summary and Future Work 63

Bibliography 67

List of Figures

1.1 Execution of a data flow model with a central execution engine 12
1.2 Execution of a data flow model in a distributed IoT environment 13

2.1 Pipes-and-filters example 16
2.2 Publish-subscribe scheme [EFGK03] 19

4.1 Conceptual overview 25
4.2 Life cycle method for the execution of data flow models in distributed IoT environments 27
4.3 UML representation of the data flow meta model 28
4.4 Example of a data flow model 29
4.5 UML diagram of the network topology 30
4.6 Example of a fully connected network topology 31
4.7 Example of a star network topology 32
4.8 Message format for data exchange 33
4.9 Architecture of the messaging engine 35
4.10 Sequential data flow pattern 38
4.11 Parallel split data flow pattern 39
4.12 Exclusive split data flow pattern 40
4.13 Merge with a foregoing split operation 40
4.14 Merge without a foregoing split operation 41
4.15 Merge operation with a gateway node 42
4.16 Subsequent splits 42
4.17 Merge and split in a single node 43
4.18 MBP fallback 46

5.1 Overview of the implementation 50
5.2 Data structure of the node repository 51
5.3 MBP data model 56

List of Listings

2.1 RAPIDE-EPL example [BD15] 17

5.1 Example of a hello resource in JSON representation 54
5.2 Example of a node resource in JSON representation 54
5.3 Example of a message resource in JSON representation 54
5.4 Example of a data flow resource in JSON representation 55
5.5 Example of an operation resource in JSON representation 55
5.6 Example of a health resource in JSON representation 55

1 Introduction

The *Internet of Things* (IoT) [MSDC12; VF13] is an emerging technology, enabled by equipping everyday objects with computational power and networking capabilities. Multiple of interconnected *smart devices* [AAS13] compose *smart environments* or *IoT environments* [HBS+16]. Examples of those are *smart homes*, *smart factories* and *smart cities* [Coc14; Har06; LCW08]. Physical devices transform into *cyber-physical systems*, bridging the gap between the cyberspace and the physical world [BG11; LBK15]. This leads to the emergence of huge amounts of data from smart devices, like sensors, and, thus, to many opportunities, summarized by the term *Big Data* [MBD+12; MCB+11]. The sensor data can be used to make IoT environments adaptive to changing conditions, called *situations* [HWS+16]. These adaptations usually occur automatically, e.g., when the temperature sensors of a machine sense a temperature over a predefined threshold, the machine is automatically shut down. *Data flow models* can be used to define how the data, extracted from *data sources*, has to be processed to evaluate if the desired situation applies. Data flows are modeled by domain experts according to a specific business logic via graphical modeling tools, such as *FlexMash* as introduced by Hirmer and Behringer [HB16].

In the following, the state of the art approach for the execution of data flow models is described and its corresponding problems are demonstrated. Furthermore, a solution for these problems and the goals of this thesis are given.

1.1 State of the art approach for the execution of data flow models

The state of the art for the execution of data flow models is to use a central *execution engine*, e.g., a complex event processing (CEP) engine [CM12]. The data flow model is sent to the execution engine, which is typically running in a cloud, to handle huge data loads [MG+11]. The execution engine executes the data flow model by retrieving the data from each device and processing it in the order of the model. Every chunk of data moves through the execution engine. When the execution of the data flow model is finished, the execution engine returns the result, which is then transmitted back to

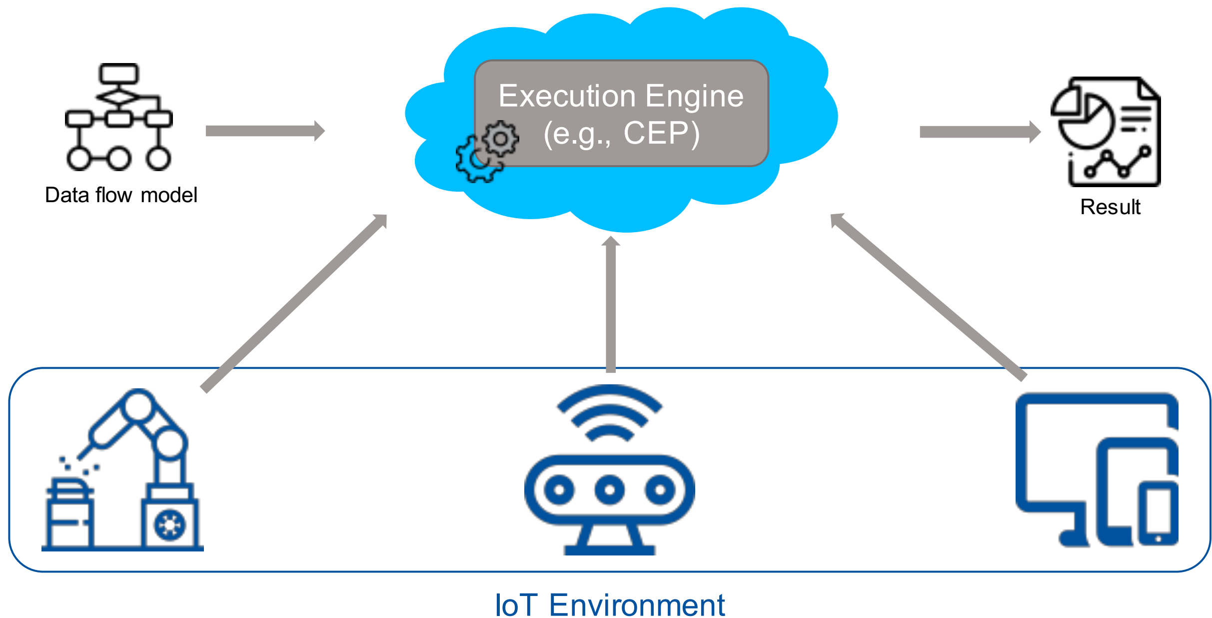

Figure 1.1: Execution of a data flow model with a central execution engine

the IoT environment, e.g., to react to the situation. Figure 1.1 shows the state of the art approach for the execution of data flow models in IoT environments. Since every chunk of data moves through the execution engine, it is a bottleneck in terms of performance and a single point of failure in terms of robustness. Data always needs to take the detour through the execution engine instead of directly traveling to the next device. Since the execution engine is usually running in a distant cloud environment, data has to travel long distances. Also, if the execution engine breaks, the whole execution of the data flow will stop.

Furthermore, this approach is not the intention of the Internet of Things, since every device is equipped with computational power which is not appropriately used when data is processed in a central execution engine. Data does not necessarily have to be moved from the IoT environment to a cloud, since it can also be processed on the devices themselves.

1.2 Solution and goals of this thesis

A better solution to execute a data flow models in distributed IoT environments is to send data directly from one IoT device to another, instead of of sending it through a distant execution engine. The solution is depicted in Figure 1.2. The *communication component* distributes the data flow model over the IoT devices an starts the execution. The data

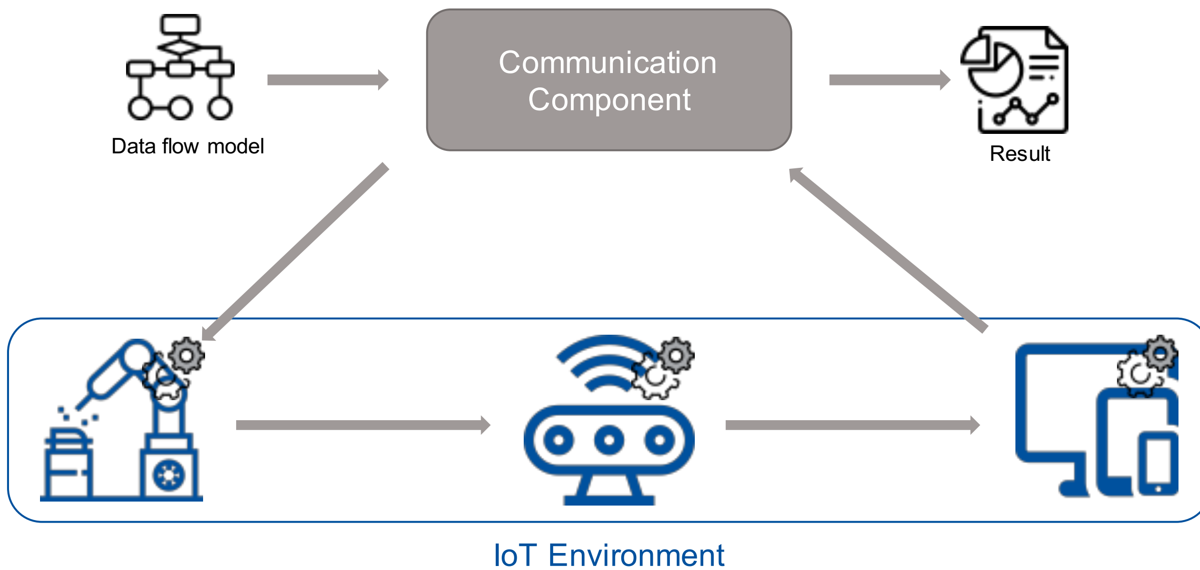

Figure 1.2: Execution of a data flow model in a distributed IoT environment

is processed on each device and sent directly to the next device in the data flow. Data does not have to be sent back to the communication component or an execution engine in the cloud to be processed and then back to the next device in the IoT environment. The result is returned to the communication component. This solution makes use of the computing resources of the devices in the environment and omits the communication overhead between devices and the execution engine. Mechanism for fault handling must be implemented into the devices to ensure robustness.

The execution of data flow models in distributed IoT environments without a central execution engine in a robust and efficient manner is the goal of this thesis.

Structure

The thesis is structured as follows:

Chapter 2 - Fundamentals Necessary fundamentals are described in this Chapter. Those include data flows, the Internet of Things, peer-to-peer systems and publish-subscribe.

Chapter 3 - Related Work This chapter gives an overview of related scientific work. This includes, among others, several publications of the University of Stuttgart.

Chapter 4 - Execution of data flow models in distributed IoT environments Chapter 4 is the main part of this thesis. It describes a concept for the execution of data flow models in distributed IoT environments. Furthermore, it describes possible optimizations to increase robustness and efficiency.

Chapter 5 - Implementation As a proof-of-concept, a prototype has been implemented which is described in this chapter. It is implemented in Python and makes use of the CoAPthon library and the integrated database ZODB.

Chapter 6 - Evaluation An evaluation of the concept of Chapter 4 is given in this chapter. Except from the prototypical implementation, whether the concept is sufficient for real-life systems is regarded.

Chapter 7 - Summary and Future Work Chapter 7 gives a summary of this thesis and potential future work.

2 Fundamentals

In this chapter, fundamentals to comprehend the concepts of this thesis are provided. Section 2.1 describes Information Flow Processing, particularly, the modeling of data flows and the complex event processing. In Section 2.2, the fundamentals of the Internet of Things are described, especially the Constrained Application Protocol. Basic concepts of peer-to-peer systems are described in Section 2.3. The publish-subscribe pattern is explained in Section 2.4.

2.1 Data flows

In their article "Processing flows of information: From data stream to complex event processing," Cugola and Margara [CM12] define Information Flow Processing (IFP), which they distinct in active database systems, data stream management systems, and complex event processing systems. Except from storing data, active database systems also execute rules following the event-condition-action pattern [CM12]. When a specific event occurs and a condition is met, a given action is executed. An event is typically an insertion or an update of a data set. Data stream management systems are also built around persistent storages, like active database systems, except that they process streams of data instead of static data sets. Modeling of data flows is described in Section 2.1.1. Complex event processing systems are explained in Section 2.1.3.

2.1.1 Data flow modeling

Data flows can be modeled using to the pipes-and-filters pattern, introduced by Meunier [Meu95]. Figure 2.1 shows an example of a pipes-and-filters based data processing model. A filter in the pipes-and-filters pattern is a software component, a pipe is a connection between two filters, determining that data gets transfered from the one filter to the other. A filter can be implemented as a service in the sense of Perrey and Lycett [PL03] with a unified interface, for example, *Representational State Transfer* (REST) [FT00]. Data sources and data sinks are special kinds of filters. Data sources are filters that have only outgoing pipes, data sinks have only incoming pipes. Since all the

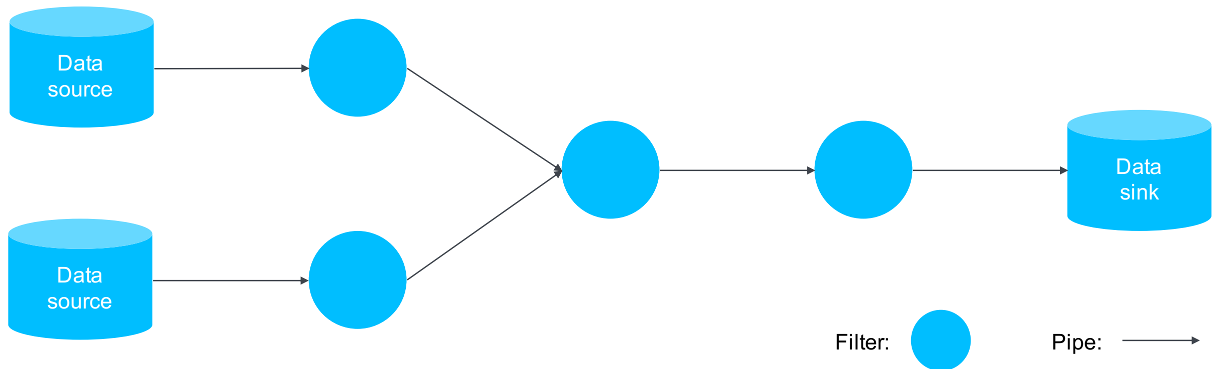

Figure 2.1: Pipes-and-filters example

filters in a data flow model have a unified interface, they can get randomly connected via pipes.

2.1.2 Data flow patterns

Reimann and Schwarz [RS+13] define a set of data flow patterns. They categorize the basic data flow pattern into the *data transfer and transformation pattern* and the *data iteration pattern*, which is, furthermore, divided into the *parallel data iteration pattern* and the *sequential data iteration pattern*. The parallel data iteration pattern consists of a segmentation phase, operation phase and a merging phase. For the segmentation phase, they define the *data splitting pattern* to split the data flow into multiple ones. For the merging phase, they define the *data merge pattern* to integrate the data flows back into a single one. A data iteration is an iteration over a specific dataset. For this thesis, I adopt the sequential data iteration pattern, the data merge pattern and the data splitting pattern. Furthermore, I divide the latter into the parallel data split pattern and the exclusive data split pattern.

2.1.3 Complex event processing

Complex event processing (CEP) is an approach to recognize patterns in distributed message-based systems [LF98]. Cugola and Margara [CM12] describe complex event processing systems as follows. Events in the external world are observed by event observers, also called sources. Event consumers or sinks are notified about events. Between the observers and consumers lies the *complex event processing engine*. The CEP engine filters and combines events from the observers to create higher-level events, also called *composite events* or *situations*, which are then notified to the event consumers.

Listing 2.1 RAPIDE-EPL example [BD15]

```
when (A and B and C) then action_X
```

They see CEP systems as extensions to traditional publish-subscribe systems, because publish-subscribe systems handle only single events at a time [ASS+99], while CEP systems handle composite events. The occurrence of composite events depends on the occurrence of other events, so they can combine the values of multiple sensors via so called *event patterns*. Event patterns are described with event patterns languages which exist in a variety of complexity and difficulty of implementation [Luc02]. Luckham [Luc02] describes four categories of event pattern languages: string pattern matching, single-event, content based matching, multiple-event matching with context and CEP matching. Languages of the latter category often consist of a full set of relational operators, context guards and pattern macros [Luc02]. An example of a CEP matching language is RAPIDE-EPL [Luc02]. Listing 2.1 shows an example expression in RAPIDE-EPL syntax, defining a composite event that occurs if the events *A*, *B* and *C* have occurred [BD15]. The CEP engine is often implemented in a distributed manner, consisting of multiple *event processing agents* (EPA) [ENL11]. An event processing agent (EPA) is defined as a software module that processes events [Luc11]. Multiple EPAs that are interconnected via channels are called an *event processing network* (EPN), which is itself an EPA again [Luc11].

2.2 Internet of Things

Xia et al. [XYWV12] define the Internet of Things (IoT) as the networked interconnection of everyday objects, which are equipped with ubiquitous intelligence. Miorandi et al. [MSDC12] refer to these objects as smart objects with the following properties: a physical embodiment, a minimal set of communication functionalities, a unique identifier, one name or one address, basic computing capabilities and sometimes the ability to sense physical phenomena. Those properties give smart objects the ability to be identifiable, to communicate and to interact. They define eight key system-level features that the Internet of Things needs to support: device heterogeneity, scalability, ubiquitous data exchange through proximity wireless technologies, energy-optimized solutions, localization and tracking capabilities, self-organization capabilities, semantic interoperability and data management and embedded security and privacy-preserving mechanisms [MSDC12]. Miorandi et al. [MSDC12] summarize the three main system-level characteristics of the Internet of Things as follows: anything communicates, anything is identified and anything interacts. These features and characteristics of the IoT leads to many new business opportunities in the fields of smart homes, smart cities, environ-

mental monitoring, health-care, smart business and security and surveillance [MSDC12; VF13].

2.2.1 Cyber-physical systems

Another term for a smart object is *cyber-physical system* [BG11]. Baheti and Gill [BG11] define a cyber-physical system as a system in which physical and computational capabilities are integrated. It is also able to interact with humans through many interfaces. Lee, Bagheri, and Kao [LBK15] propose a five level architecture for cyber-physical systems with two main functional components: connectivity to the physical world and information feedback from the cyber space and intelligent computational capability that constructs the cyber space. So cyber-physical systems connect the cyber space with the physical world.

2.2.2 Constrained Application Protocol

The Constrained Application Protocol [BCS12] (CoAP), defined in [SHB14] by the IETF[1] Constrained RESTful Environments (CoRE) working group, is a communication protocol for environments with constrained resources, like IoT environments. CoAP uses UDP [Pos80] and a simple message layer to retransmit lost messages, instead of TCP [Pos81]. The total header size of the message layer in a typical message is about 10 to 20 bytes [BCS12], making it much more lightweight than HTTP [FGM+99], where messages have a typical header size between 200 to 2000 bytes[2]. On top of the message layer, CoAP has four methods, known by HTTP: GET, PUT, POST, and DELETE. The response codes of CoAP are also similar to the ones of HTTP, but encoded in a single byte.

2.3 Peer-to-peer Systems

If participants of a distributed network architecture share parts of their own hardware resources, like processing power, storage capacity and network link capacities, the network architecture may be called a peer-to-peer (P2P) system [Sch01]. Peer-to-peer can be categorized in structured and unstructured peer-to-peer systems [WS05]. Unstructured per-to-peer systems either rely on a central server that stores the locations

[1] https://www.ietf.org
[2] http://dev.chromium.org/spdy/spdy-whitepaper

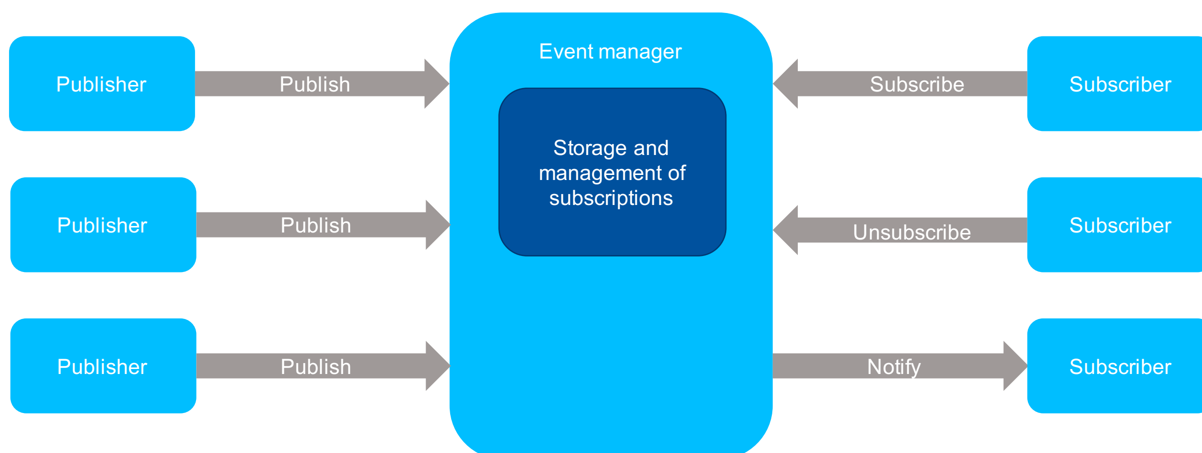

Figure 2.2: Publish-subscribe scheme [EFGK03]

of all resources to serve lookups to clients, or lookups need to be sent to all peers participating in the network. Algorithms to distribute lookups over an unstructured peer-to-peer network are breadth-first search, depth-first search and random walk. Structured peer-to-peer systems use distributed indexing structures, like distributed hash tables, e.g., *Koorde* [KK03], to retrieve data items from the network. Lookups with Koorde take $O(logn)$ hops in a network with at least two neighbors per peer.

2.4 Publish-subscribe

Publish-subscribe is an event-based interaction scheme to exchange information in a loosely coupled and asynchronous manner [EFGK03]. Figure 2.2 depicts the publish-subscribe scheme with its components and interaction elements. *Subscribers* express their interest in an *event* or pattern of events, called a *subscription*. When a *publisher* generates an event, every subscriber gets a *notification* about the event. The message bus that handles subscriptions and notifications ins called the *event manager*.

3 Related Work

In this section, to the concept of this thesis related scientific work is introduced.

With SitRS XT, Franco da Silva et al. [SHWM16] propose a concept for near real-time situation recognition in IoT environments by the use of situation templates and complex event processing [Luc02], based on the work of Hirmer et al. [HWS+15]. The situation template contains the monitored sensors and the conditions that have to apply to recognize the situation. Situation templates are directed, cohesive graphs as introduced by Zweigle et al. [ZHKL09], also called situation aggregation trees. The concept for the execution of data flow models in distributed IoT environments can be used to execute such situation aggregation trees in an efficient and robust manner.

With *FlexMash 2.0*, Hirmer and Behringer [HB16] describe an approach for realizing data flow modeling based on the pipes-and-filters pattern [Meu95]. The data flow model in FlexMash 2.0, called a mashup plan, gets transformed into an executable format, such as a workflow model. Each filter in the mashup plan is implemented by a service offering a REST interface. Hirmer and Behringer describe a method for the execution of data flow models containing six steps: data flow modeling, requirement definition, runtime environment selection, mashup plan transformation, data flow execution and finally result processing. A main concept of Hirmer and Behringer is the execution of the mashup plan in a specific runtime environment, which is automatically chosen, based on the requirement definition of the user [Hir18]. The aim of this thesis is to provide a method to execute data models, such as the mashup plans in FlexMash 2.0, in a distributed IoT environment, without a central runtime environment.

Franco da Silva et al. [SHKM18] describe a concept for CEP query shipping in distributed IoT environments. Their query shipping model is based on the pipes-and-filters pattern and consists of three different types of nodes: data sources, data sinks and query nodes. They also introduce the *Multi-purpose Binding and Provisioning Platform* (MBP) which enables automated binding of IoT devices to access their sensors and actuators and provisioning software on those. They define a life cycle method consisting of five steps, whose initial input is the query shipping model: registration of data sources and data sinks, choice of execution hardware, installation of software components, CEP query shipping, and finally retiring of data processing. Franco da Silva et al. describe a method to keep track of available devices and deploy software on those. They do not introduce

a method to execute a data flow model on those devices, which will be the issue of this thesis. The MBP will be used in this thesis as a platform to register new IoT devices and deploy software on them.

IoT-Lite [BEBT16] is an instantiation of the *semantic sensor network* described by Taylor, Ayyagari, and De Roure [TAD11]. It is a lightweight but extendable semantic model to achieve interoperability and discovery of sensor data in an IoT environment. IoT-Lite is an ontology language to describe sensors with their respective values, meta data and options. IoT-Lite could be extended to act as a modeling language to describe the IoT environment used for the execution of data flow models. This comes with a huge overhead, because the network topology described in Section 4.3 can be modeled much simpler and is not the core of this thesis.

A distributed IoT environment can be classified as a peer-to-peer network [Sch01]. Consequently, many concepts from peer-to-peer networks can be applied to the execution of data flow models in distributed IoT environments, for example, search algorithms, such as breadth-first search, depth-first search or random walk. These search algorithms are based on forwarding messages until the ultimate receiver has been reached. This is much slower in comparison to if the sender already knows the ultimate receiver of a message and sends it directly to him, which is applied in the concept of this thesis. Another concept of peer-to-peer systems are distributed routing tables. Those could be used to route messages to a specific device, but won't be applied in the concept of this thesis, because I assume that every device in the IoT environment is aware of every other device in it. Distributed routing tables could be used as an enhancement for IoT networks that are not fully connected, in contrast to the ones assumed for this thesis.

The Message Queuing Telemetry Transport Protocol[1] (MQTT), standardized by an OASIS committee[2], is a lightweight machine-to-machine communication protocol with a publish-subscribe model. The publish-subscribe pattern is not suitable for the execution of data flow models in distributed IoT environments, because with publish-subscribe, one published message can be consumed by multiple subscribers. This would lead to a duplication of messages and, thus, to a loss of integrity of the execution of the data flow model. Also, the publish-subscribe pattern is based on a central component, the broker, which is contradictory with the concept in this thesis, which is the execution of data flow models without a central component.

Apache Kafka[3] is a distributed streaming platform with a publish-subscribe [EFGK03] approach. Kafka is capable of processing real-time streams of data which are organized

[1] http://mqtt.org
[2] http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html
[3] https://kafka.apache.org

in so called topics. Kafka can be used as a messaging system and its design is focused on horizontal scalability. Using Kafka for the concept of this thesis requires installing Kafka either on a subset of the devices, which would be contrary with a distributed execution of the data flow model, or on each device, which is a huge overhead and contrary to the intention of the design of Apache Kafka. A concept of Kafka that could be used for this thesis is how Kafka stores its messages. Kafka stores every messages immediately in the filesystem in a efficient way.

The IETF CoRE Working Group proposes a concept to register and lookup device's resource descriptions in an IoT environment [She12], the "CoRE resource directory" [SBK13]. However, the document has still the status of a draft and mainly contains definitions of interfaces and guidance for the design of a resource directory. Thus, the concepts of the CoRE resource directory have not been in contemplation for this thesis.

Liu et al. [LLH+13] propose a distributed resource directory architecture for machine-to-machine communication to process resource registration and lookup. The architecture consists of multiple directory peers and light directory peers. A directory peer handle resource description registration and lookup for one or more constrained devices and takes part in a peer-to-peer overlay network that is used to exchange information about resources between different directory and light directory peers. Light directory peers only participate in the overlay network, store and lookup resources, to integrate mobile devices into the overlay network. The concepts of Liu et al. [LLH+13] could be used in the execution of data flow models in distributed environments as an addition to distribute information about devices in the IoT environment. Since a resource lookup costs much more time, in the concept of this thesis, information about resources gets distributed over all devices in the environment before the execution of the data flow model starts.

4 Execution of data flow models in distributed IoT environments

In this chapter, the main contribution of this thesis is described: the execution of data flow models in IoT environments. To execute data flow models in distributed IoT environments without a central execution engine, devices need to be able to communicate and transmit data with each other autonomously. This data, for example, originating from sensors, can be delivered from one device to another via messaging. To make this possible, devices need to be equipped with a *messaging engine*. Figure 4.1 shows an overview of the conceptual approach for the execution of data flow models in IoT environments. Every IoT device consists of a physical part, the sensors and actuators, and a software part, the runtime environment. The underlying layer of the runtime environment is the *application*. It is specific for the device and able to interact with the physical part of the IoT device, e.g., by extracting sensor data or by invoking actuators. The application receives data from the messaging engine, executes an *operation* on the received data (e.g., filtering, aggregation, analysis), and returns the result of the operation. The messaging engine consumes messages from other devices and sends messages to them.

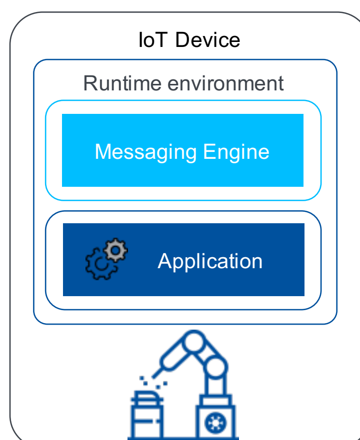

Figure 4.1: Conceptual overview

I define the following requirements for the approach aimed for in this thesis:

Requirement R1: Loose Coupling
The solution must be autonomous in terms of platform technology, location and data format [Kay03]. Devices should be able to communicate in a uniform message format, independently from the platform technology, location or format of the data they want to interchange. Time autonomy, as described by [Kay03], cannot be fulfilled, since messages are sent directly among devices without the use of channels [HW04].

Requirement R2: Persistence and Guaranteed Delivery
Data must not be lost because this can lead to an incorrect execution of the data flow model. A message must always be persistently stored on at least one device. A device may only delete a message when it has successfully been transmitted to the next device.

Requirement R3: Horizontal Scalability
Adding more devices to the IoT environment must lead to an increase in performance. It must not be relevant which device processes a message, as long as the device is able to process the message, which is described in Section 4.2. This way, messages can be distributed among available devices.

Requirement R4: Vertical Scalability
Increasing computational power of devices must lead to increased performance in terms of the execution of the data flow model. The only bottleneck for processing a message is the execution of the application.

Requirement R5: Monitoring It must be possible to monitor the whole system and get an overview of the general status of the execution of the data flow, to be able to manually react to specific situations.

Requirement R6: Robustness
If devices break down, the execution of the data flow model must still proceed. After a breakdown of the whole environment, it must be able to resume the execution at the point of the breakdown without the loss of data.

Section 4.1 describes a life cycle method for the execution of data flow models in distributed IoT environments. The five steps of the life cycle method are described in the following sections. Furthermore, Section 4.7 discusses possibilities to optimize the messaging engine in terms of robustness and performance.

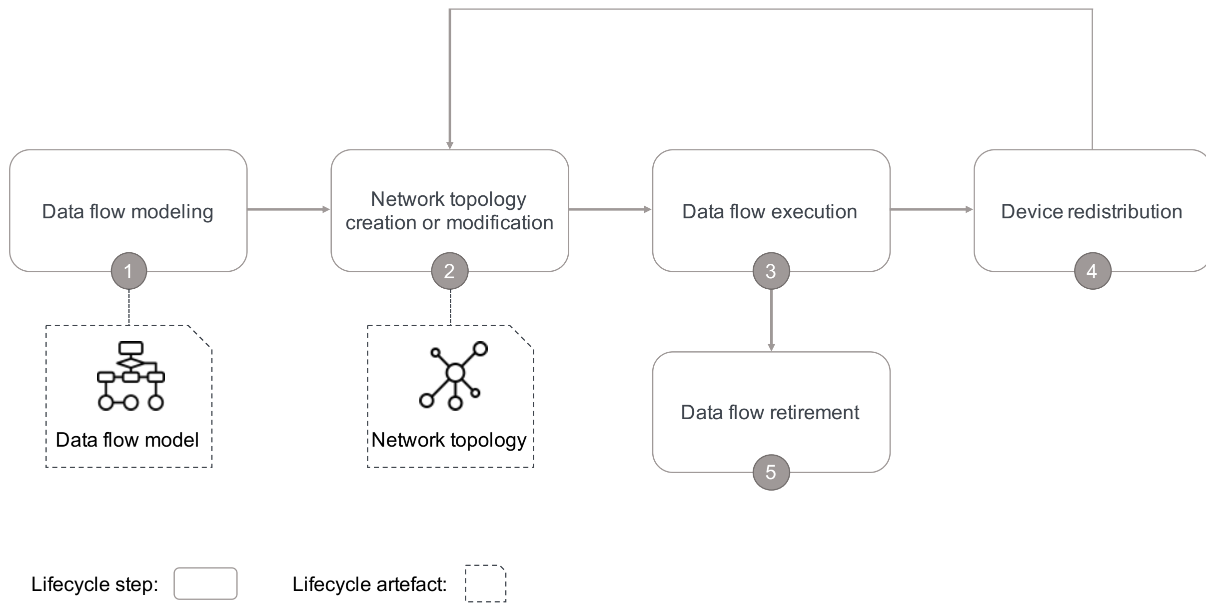

Figure 4.2: Life cycle method for the execution of data flow models in distributed IoT environments

4.1 Lifecycle method overview

I introduce a life cycle method for the execution of data flow models in distributed IoT environments, which is depicted in Figure 4.2. The life cycle begins with modeling the data flow (step 1, Figure 4.2). This can be done by a domain expert using a modeling tool, such as the one described by Hirmer and Behringer [HB16]. The second step is the definition of the network topology, containing all the available devices in the environment and all the operations they are able to perform. The data flow, or parts of it as described in Section 4.4.2, and the network topology then need to be distributed throughout the devices in the environment. To achieve this, a *communication component* is introduced, which is aware of the devices in the environment and is able to communicate with them via a REST interface, as described in Section 4.4.2. The third step is the execution of the data flow model in the IoT environment. The execution is either triggered automatically by one of the devices or manually. Step four is the redistribution of devices. Devices can be added and removed dynamically or some devices might fail and not be responsive anymore. If this happens, the network topology needs to be redefined and the execution of the data flow model can be resumed. In the fifth step, the data flow is retired once it reaches the end of its lifetime. This means that no more data related to the retired data flow will be processed. The individual steps of the lifecycle method are described more detailed in the following sections.

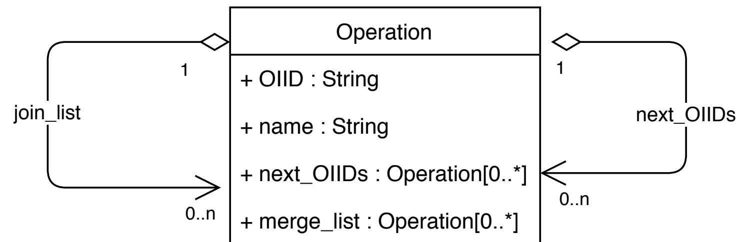

Figure 4.3: UML representation of the data flow meta model

4.2 Lifecycle method step 1: Data flow modeling

The lifecycle of a data flow model execution begins with the modeling of the data flow itself. The data flow model is a directed graph, which is based on the pipes and filters pattern [Meu95], in which nodes in the graph are called *filters* and edges are called *pipes*. Each filter in the model represents an operation that can, for example, be performed by an IoT device. Note that different operations in the data flow model do not necessarily have to be performed by different devices. More precisely, one device can process an arbitrary amount of operators as long as its resource capabilities are sufficient. Furthermore, the same operator can be run on different devices simultaneously to achieve distributed, parallel data processing. Each pipe in the model represents the transfer of data between two operations, i.e., the data flow. In my approach, every node in the data flow model must be annotated with the following information:

- Operation Instance Identifier (OIID)
- Operation Identifier (Operation ID)
- Next Operation Instance Identifiers
- Merge list (optional)

The OIID is a unique identifier for an operation in the data flow model. This is necessary, since every type of operation can have multiple instances in one data flow. Operation ID is an identifier for the operation the device must perform at a specific point in the data flow model. The device maps the Operation ID to a specific application on the device's filesystem. The next operation instance identifiers of an operation determine which operations must be performed next. The merge list is a optional list of foreign OIIDs to indicate that the operation merges multiple messages. The merge list is needed so that the device knows the amount of paths that should be merged, i.e., for how many messages it needs to wait before it can process the messages. Figure 4.3 shows an UML representation of the data flow meta model. Pipes are implicitly defined by the attribute list *next_OIIDs*.

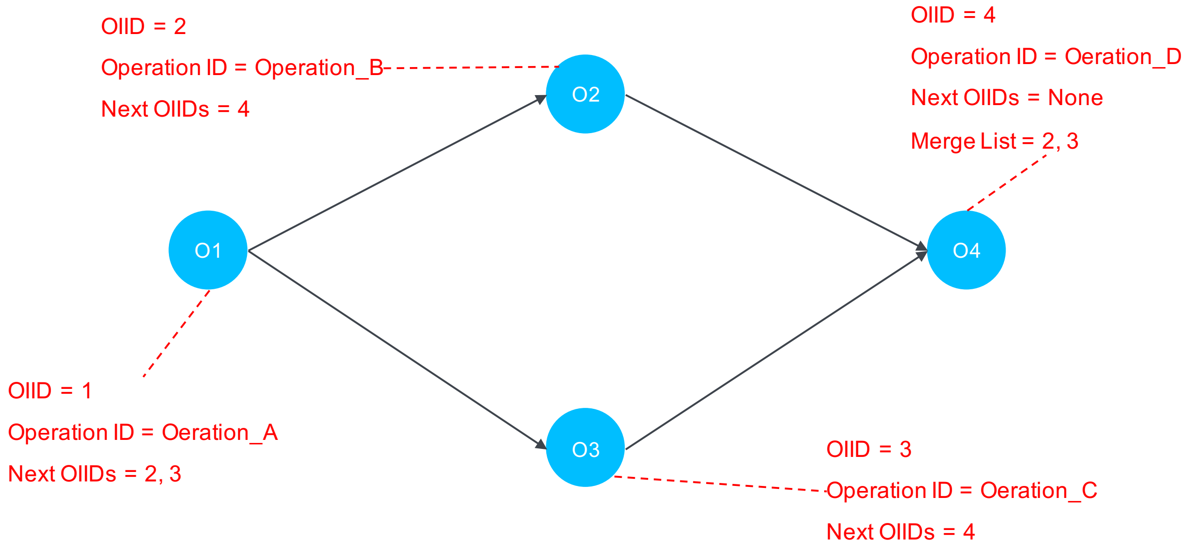

Figure 4.4: Example of a data flow model

Figure 4.4 shows an exemplary data flow model with the necessary information attached to each node. The node *O4* is the only one with a merge list, which denotes that *O4* merges messages from the operations with the OIIDs 2 and 3, i.e., *O2* and *O3*. The attribute *Operation ID* of each operation denotes which actual operation needs to be performed at that point in the data flow model, e.g., *Operation_C* for *O3*. A device performs operation *Operation_A* and sends the results to two other devices to perform the operations *Operation_B* and *Operation_C*. Both of them send their results to the next device *O4* which performs the merging operation *Operation_D*.

The described data flow model can be formalized as a directed graph:

$$DF = (O, E) \tag{4.1}$$

with operations

$$O = o_0, \ldots, o_n \tag{4.2}$$

and edges

$$E \subseteq O \times O \tag{4.3}$$

where every node represents an operation and every edge (o_i, o_j) denotes that after finishing the operation o_i, the resulting data is used as an input for operation o_j. Each operation that has only outgoing edges is called a *data source*, each operation that has

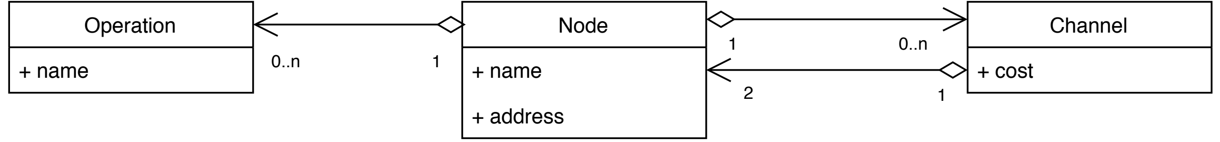

Figure 4.5: UML diagram of the network topology

only incoming edges is called a *data sink*. I also define the *successor* function suc as follows:

$$suc\colon O \to O \tag{4.4}$$

$$suc(a) = \{b \in O|(o,b) \in E\} \tag{4.5}$$

The function maps an operation to all its succeeding operations in the data flow model, i.e., it is the formalization of the Next Operation Identifiers list.

4.3 Lifecycle method step 2: Network topology creation or modification

The network topology is a description of all connected devices in the IoT environment containing the names, addresses and operations they are able to perform. Since all devices that participate in the data flow must be connected to the network, IoT environments can be viewed as fully connected networks. That means, every device in the network is potentially able to communicate with every other device in the network directly. In the context of the network topology, throughout this document, a device is also referred to as a *node*. Figure 4.5 shows the UML representation of the network topology. A fully connected network topology can be modeled as an undirected, connected and weighted graph, which is formalized as follows:

$$NT = (N, C, c, ops) \tag{4.6}$$

with a set of nodes, i.e., devices

$$N = \{n_0, \ldots, n_m\} \tag{4.7}$$

a set of undirected edges or channels

$$C \subseteq \{\{n_i, n_j\} \mid n_i, n_j \in N\} \tag{4.8}$$

and the weight or cost function

$$c\colon C \to \mathbb{R}_{\geq 0} \tag{4.9}$$

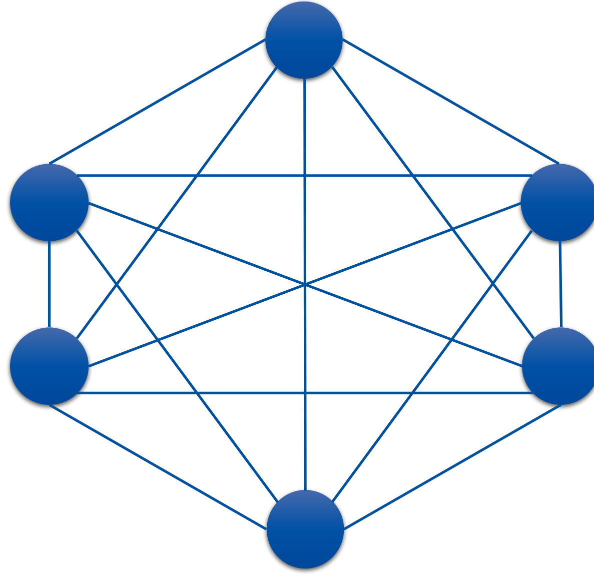

Figure 4.6: Example of a fully connected network topology

which indicates the cost of delivering data over a channel $\{n_i, n_j\}$ from node n_i to node n_j or vice versa. The cost of a given channel depends on multiple factors, e.g., the time the data requires to be delivered over that channel or the bandwidth of the communication medium.

I also define the function ops:

$$ops\colon N \to \mathcal{P}(O) \tag{4.10}$$

That a network topology can be modeled as a complete graph means that for every two nodes exists a path from one node to the other. A path is defined as:

$$P = (n_0, n_1, \ldots, n_m) \tag{4.11}$$

with

$$\forall 0 \leq i \leq m-1\colon \{v_i, v_{i+1}\} \in C \tag{4.12}$$

Consequently, for every two nodes n_i and n_j in N, there is a path from n_i to n_j. And since the graph is undirected, there is also a path from n_j to n_i. Each node represents a device and every channel between two nodes indicates that both devices are able to, without an intermediate device, send messages directly to each other. The function ops maps every node to a subset of the set of operations O, to indicate that a given node n is able to perform the operations $ops(n)$. Figure 4.6 shows an example of a fully connected network topology. Figure 4.7 shows a star network topology example, which is also connected. For a data flow $DF = (O, E)$ to be executable on a given network topology

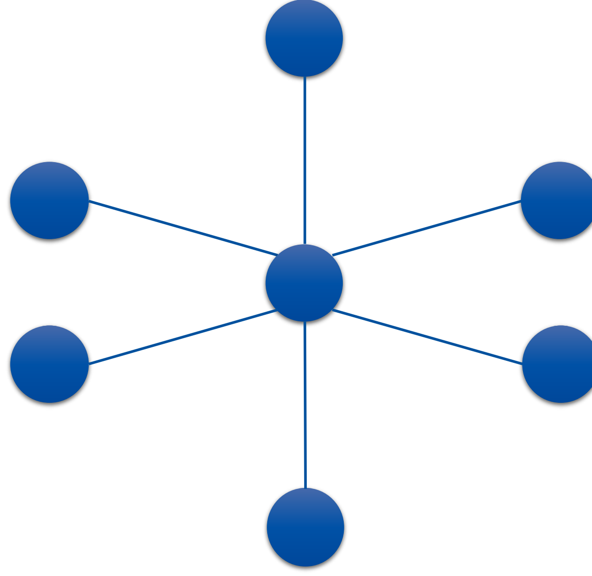

Figure 4.7: Example of a star network topology

$NT = (N, C, c, ops)$, the following rule must be fulfilled:

$$\forall o \in O \exists n \in N : o \in ops(n) \tag{4.13}$$

This means that for every operation in the data flow, there must be at least one node in the network topology that is able to perform it.

Also, for each operation o that a device can perform, it must at least know the address of one or more devices that can perform each operation in $suc(o)$, for each data flow model in whose execution the device can participate. The function $neigh$ states all *neighbor* nodes of a node, i.e., all nodes that the node can exchange messages with. It is defined as

$$neigh : N \to \mathcal{P}(N) \tag{4.14}$$
$$neigh(n) = \{m \in N | \{n, m\} \in C\} \tag{4.15}$$

Consequently, another condition that is required for a data flow $DF = (O, E)$ to be executable on a given network topology $NT = (N, C, c, ops)$ is:

$$\forall n \in N, \forall o \in ops(n), \forall p \in suc(n) \exists m \in N : m \in neigh(n) \vee p \in ops(m) \tag{4.16}$$

For example, in Figure 4.4, the device that performs Operation_A with OIID 1 must know at least one device that is able to perform the operation Operation_B and one that is able to perform the operation Operation_C.

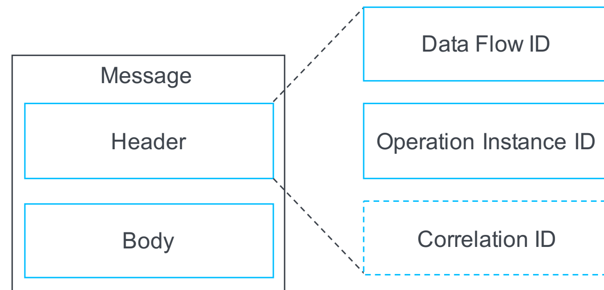

Figure 4.8: Message format for data exchange

4.4 Lifecycle method step 3: Data flow execution

The third step, the execution of the data flow model, is the main part of this thesis, thus, it is described in more depth than the other steps. Besides the description of the step itself, an architecture is presented, which is able to process the data flow models. IoT environments are still constrained environments in terms of computational resources, robustness and failure safety. Thus, there are many additional requirements that need to be fulfilled, compared to conventional messaging systems. Since there is no central control instance, errors during the execution of the data flow are handled by the devices themselves. Thus, robustness is one of the most important requirements for the messaging engine. Another requirement for the messaging engine is flexibility, which enables adding and removing nodes to the network topology. Messages need to be immediately and persistently stored on the disc after receiving and before acknowledging them. The sending node must only remove the message after receiving the acknowledgment. This ensures guaranteed delivery of messages between devices. Since a message can trigger an operation that should only be performed once per message, messages must not be duplicated. Section 4.4.1 describes all the ingredients a message must have to be used for the distributed execution of data flow models as described in this thesis. Section 4.4.2 gives a general overview of all the components of the architecture of the messaging engine and their functionality. Section 4.4.3 contains a brief discussion of the minimal requirements for the protocol two devices can use to deliver messages between each other. Section 4.4.4 describes how messages are being handled by the messaging engine and how different messaging scenarios, or data flow patterns, can be realized by the messaging engine.

4.4.1 Message format for data exchange

To transfer data between devices, messaging is employed. Figure 4.8 shows the information included in a message. The body of a message contains the data that the application

returns after its execution. The header of the message contains the fields data flow ID, operation instance ID and the optional correlation ID, which is used to merge messages as described in Section 4.4.4. The data flow id associates the message with a specific data flow model, since a device can participate in multiple data flows. The operation instance ID (OIID) notifies the receiving device which operation in the data flow model it must perform on the message body, i.e., the data to be processed.

An alternative to the *data flow repository* described in Section 4.4.2 is to store the whole data flow in the message. There are many more messages transfered in the environment than data flow models instantiated, because for every data flow model execution, multiple messages are transfered between devices. So transmitting the data flow model in each message is a big communication overhead that will lead to a performance loss for the execution of the data flow model. Regarding this, transmitting the data flow model inside the messages will not be further discussed in this thesis.

Messages can be structured using typical markup languages like XML[1] or JSON[2], to make them generally understandable, and transmitted via simple communication protocols like TCP or UDP. Details about the protocol to exchange messages between devices are discussed in Section 4.4.3.

4.4.2 Architecture overview

Figure 4.9 shows an overview of the different components of the messaging engine. The architecture has been divided into the three layers *interface layer*, *domain layer* and *infrastructure layer*. The interface layer contains components that communicate with other devices or provide interfaces to retrieve internal information of the device. The domain layer mainly processes messages and the infrastructure layer stores all the data and provides the interface to the device itself and the applications. The layers and their components are described more precisely in the following.

Interface layer

The *message consumer* represents the endpoint for other devices to transmit messages. The *REST interface* component offers interfaces for deploying, retrieving, updating and deleting data flow models, operations and nodes. It is also the endpoint for deploying *hello* messages and triggering the hello mechanism of a device, as described in Section 4.5. Furthermore, the REST interface serves the current health status of the device.

[1] https://www.xml.com
[2] https://www.json.org

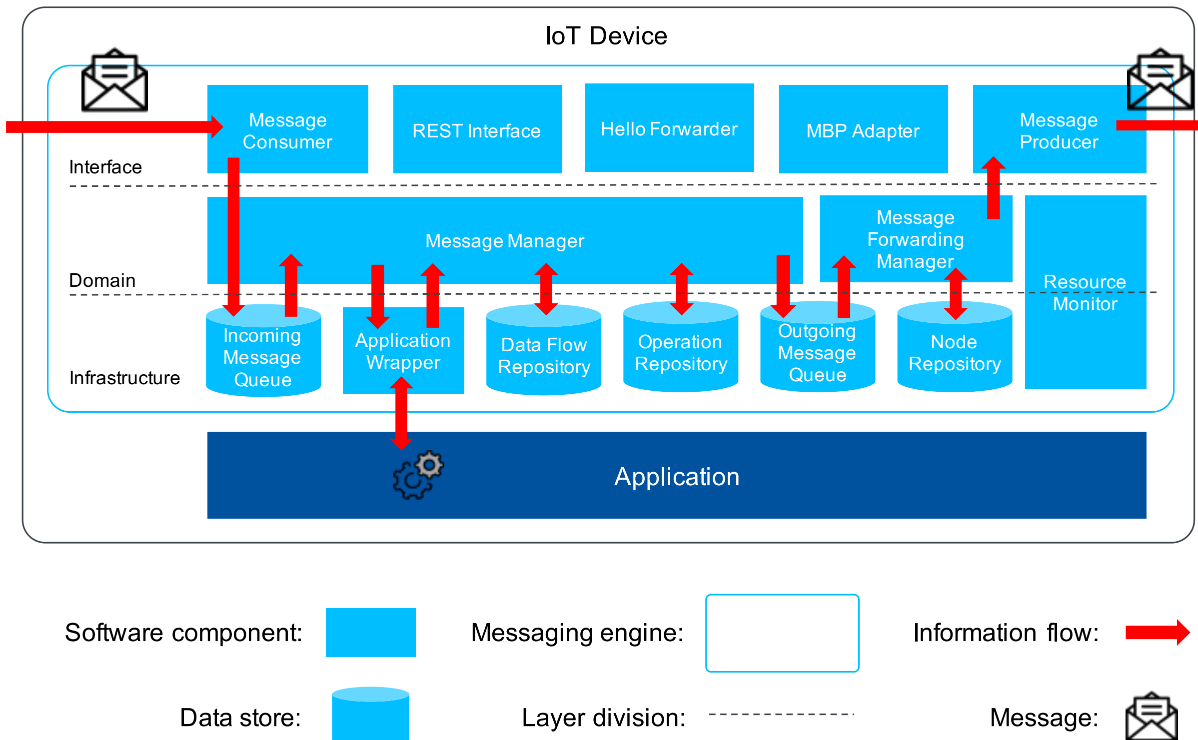

Figure 4.9: Architecture of the messaging engine

The *hello forwarder* adds the device name to every received hello message and forwards it to all other devices it has stored in the node repository. The *message producer* component sends messages it receives from the message forwarding manager to one or multiple given devices. The *MBP adapter* is able to communicate with a given MBP instance, described in Chapter 3. A possible technology to implement the REST interface and the message consumer and producer is the Contrained Application Protocol (CoAP) [SHB14], which will be further described in Section 4.4.3. The message producer communicates with a given message consumer by its IP address and a standardized port.

The protocol, which the messaging engine uses to transport messages between each other, must include responses like *acknowledgment* and *error*. Publish-subscribe [EFGK03] protocols like MQTT[3] are not suitable for the communication between devices, since they are based on a reliable message broker, whose non-existence is the essence of the concept of this thesis. The message producer and the message consumer can also be implemented as CoAP client and server, which is described in Section 4.4.3.

[3]http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html

Domain layer

The *message manager* receives messages from the incoming message repository, retrieves necessary information from the data flow and operation repository, forwards all to the application wrapper and puts the result of the wrapper into the outgoing message repository in form of a new message. The *message forwarding manager* takes messages out of the outgoing message repository, takes possible successor nodes from the node repository and forwards everything to the message producer.

The message producer and the message forwarding manager should run in separate threads to ensure that they do not block each other. Otherwise, messages cannot be processed while other messages are being transfered and vice versa. The message producer must also run in a separate thread than the message consumer, so that messages can be consumed while others are being processed. Depending on the specific application and the physical resources of a device, multiple instances of the message manager can run in parallel to increase throughput.

On the domain layer part of the *resource monitor*, it measures the capacity of the repositories and serves the resource and health information of the device to the interface layer. How the message manager and the message forwarding manager process the header of a message is described in Section 4.4.4.

Infrastructure layer

The *incoming message queue* acts like a first-in-first-out queue for messages that have been received by the message consumer. If the operation, that must be performed on a message, is a merge operation, the message is stored separately, until all messages have been received that need to be merged. The *outgoing message queue* is mostly similar to the incoming message queue. It stores messages that have been processed by the message manager until they could be delivered by the message producer.

The *data flow repository* of each device is used to store the operation IDs of the succeeding operations of the operation the device performs in a given data flow model. One solution is to store the whole data flow model in the data flow repository of each device. This leads to a huge storage overhead, but makes it more simple to distribute the information over the devices, since ever device simply receives the whole data flow model. Another solution is to only store the succeeding operations of a device's operation in its data flow repository. This makes it necessary for the communication component to traverse the data flow model and send the respective nodes to each device separately. Since every device can be a part of multiple data flows, each data flow needs to be associated with the data flow ID. This makes it feasible to implement the data flow repository as a

key-value store [HHLD11] and use the whole data flow ID as the key to the data flow model or just the list of succeeding operations. The data flow ID can also be used to update a data flow model, while messages that are still stored on the devices should be processed according to the former data flow. Otherwise, the new data flow, or data flow segments, can simply be posted to every device with the same ID than the old data flow.

The *node repository* stores a list with a subset of all operations that appear in the data flows that the device has stored. It should have at least one device stored for each operation that follows the one of the operations the device must performs in the data flow, to assure that the data flow execution can continue. To associate operation identifiers with a specific application on the file system of the device, the *operation repository* maps operation IDs on URIs[4].

The *application wrapper* wraps the functionality of the application by receiving a message body and an application URI from the message manager and returns the resulting data. The infrastructure layer part of the resource monitor measures the available resources of the device in terms of memory, disc and CPU utilization.

4.4.3 Communication protocol for message exchange

The communication protocol to transmit messages between two devices should have at least the following operations:

- PUT
- ACKNOWLEDGE
- REJECT

These three operations are necessary to assure that communicating devices can make sure that messages do not get lost or duplicated. If a device receives a message via the PUT operation and is able to store the message in its incoming message queue in terms of free capacity, it responds with the ACKNOWLEDGE operation. The sending device can then safely delete the message from its outgoing message queue. If the receiving device is not able to consume the message, it responds with the REJECT operation, so the sending device knows that it has to look for another device that can act as a successor in the data flow model. Regarding the CoAP protocol, the PUT operation can be mapped to CoAPs "Put" request. The ACKNOWLEDGE operation can be mapped to CoAPs "2.01

[4] https://tools.ietf.org/html/rfc8089

Figure 4.10: Sequential data flow pattern

Created" response, and the REJECT operation can be mapped to CoAPs "5.03 Service Unavailable" response. Similar mappings could be done to the HTTP protocol.

4.4.4 Message flow

From the data flow pattern described in Section 2.1.2, introduced by Reimann and Schwarz [RS+13], the four most common have been chosen to discuss how the messaging engine handles them: sequential data flow, parallel split, exclusive split and merge. The information flow inside the messaging engine is illustrated by the red arrows in Figure 4.9.

Sequential

The most simple case of message processing is the sequential case. It is characterized by one incoming and one outgoing message per operation. Messages can simply be received, processed and passed to the next device. Figure 4.10 shows the sequential data flow pattern.

Incoming messages are received by the message consumer. It stores the messages directly into the incoming message queue, then sends an acknowledgment back to the sender. When the device is ready to perform an operation, the message manager takes the next message out of the incoming message queue and looks for the data flow in the data flow repository with the data flow ID in the message header. In the encountered data flow model, it looks for the node with the same OIID as the one in the message header. Thus, the message manager can figure out which operation must be performed. It searches in the operation repository for the specific application that performs the operation and then triggers the application wrapper to execute it. After the execution, the result is returned to the message manager via the application wrapper.

The message manager creates a new message based on the result and puts it into the outgoing message queue. It sets the OIID of the new message to the one of the next operation in the data flow model and the data flow ID to the one of the corresponding incoming message. If the incoming message had a correlation ID, it sets the correlation ID of the outgoing message to the same one.

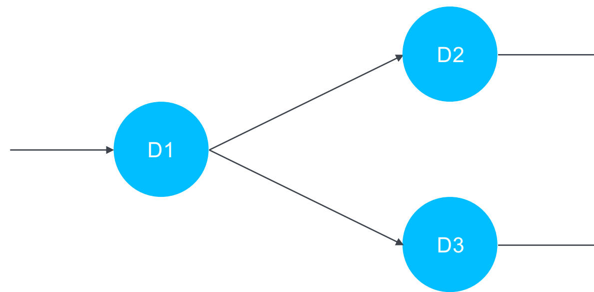

Figure 4.11: Parallel split data flow pattern

The message forwarding manager takes the next message out of the outgoing message queue. Then it seeks in the node repository for one or multiple nodes that are able to perform the operation that is associated with the OIID of the outgoing message in the data flow model. It creates a list, containing the encountered nodes, and transmits the message and the list to the message producer. The message producer then tries to send the message to one of the nodes in the node list.

Parallel split

A device that performs a split operation in the data flow model consumes one message and produces two or more messages. That means that the application in the runtime environment of the device needs to return its result in a specific format, so that the application wrapper is able to distinguish the different data sets and the message manager can create one message for each data set. The application also needs to specify which message belongs to which succeeding operation, for example, by aligning the order of its output messages to the order of the next operations in the data flow model. Another use case for the parallel split is that every parallel outgoing message should go to the same operation but to an appropriate amount of different devices to achieve performance and horizontal scalability [DG08]. To make this possible, the message producer must make sure to choose as much devices for the same succeeding operation as possible. Figure 4.11 shows an example of the parallel split pattern. Device *D1* sends two messages, which can have the same content, to each of the devices *D2* and *D3*.

Exclusive split

Exclusive split means that the message content, the outcome of the application, or a combination of both, decides which operation must be executed next. This makes it necessary for parts of the messaging engine to understand parts of the semantics of the message body. Figure 4.12 shows an example of the parallel split pattern. *D1* decides,

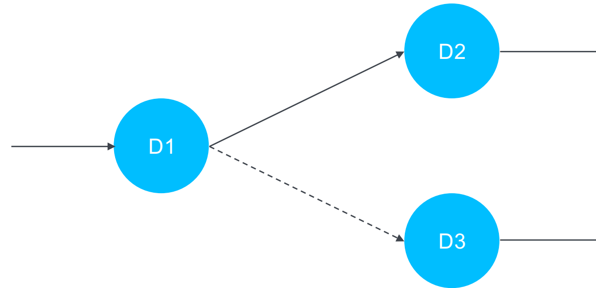

Figure 4.12: Exclusive split data flow pattern

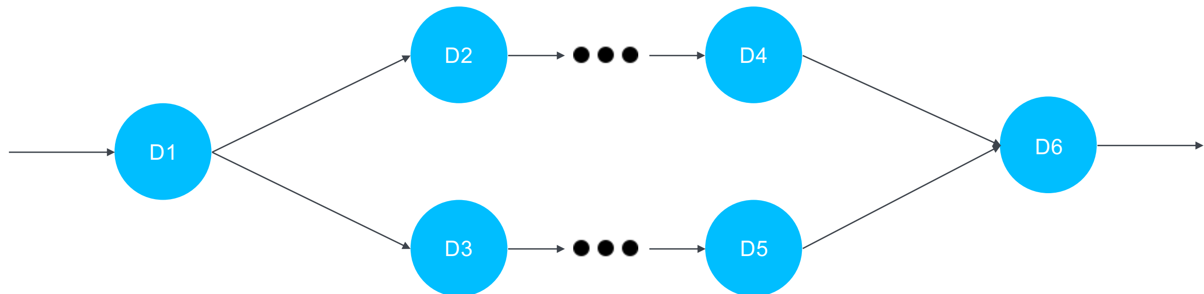

Figure 4.13: Merge with a foregoing split operation

based on some predefined criteria, to send a message to *D2* and not to *D3*. The exclusive split pattern has not been described in Section 4.2, thus, it will not be further discussed in this thesis.

Merge

A merge operation merges multiple incoming messages. There is a distinction between a merge with a preceding split operation, and a merge without one. The merge with a preceding split can be achieved by adding a correlation ID to each message after the splitting operation. Figure 4.13 shows a data flow model with a merge and a foregoing split. *D1* equips both messages, resulting from a single splitting operation, with the same correlation ID. Every device between *D1* and *D6*, that is not supposed to perform a merging operation, simply passes the correlation ID. When *D6* receives a message with a correlation ID, it stores the message separately, until the other message with the same correlation ID arrived. Then it puts both messages together in the incoming message queue, so the merge operation can be performed.

The merge without a foregoing split is much more difficult to handle and cannot be definitely solved. It can occur if the data flow has more than one starting points. Figure 4.14 shows a data flow model with a merge without a foregoing split. *D1* and *D2* are both data sources and produce messages. There is no implicit correlation between the

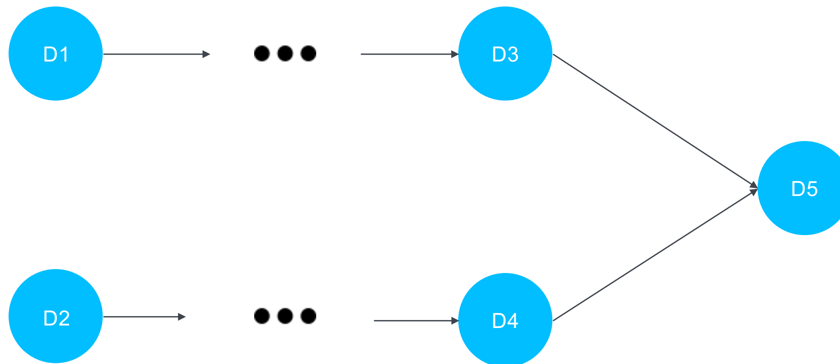

Figure 4.14: Merge without a foregoing split operation

messages produced by *D1* and *D2* as if the messages were produced by a single split operation. Also the times when *D1* and *D2* produce messages must not always be the same. One option to perform such a merge, is to match each message with the next messages that are sent through other channels. For example, if *D5* in Figure 4.14 receives one message from *D3* and then three messages from *D4*, it merges all the three messages from *D4* with the one from *D3*, resulting in three new outgoing messages. Another approach is to define a time window and merge all messages that arrive within the given time window. A third option is to merge messages syntactically, for example, averaging all values in the payload of the messages. This approach is rather inappropriate for the solution proposed in this thesis, since the messaging engine should be generic in terms of message content.

Since every operation in a data flow model can be performed by multiple devices, it could happen that two messages with the same correlation ID are being delivered to different devices that perform the merge operation. This leads to both messages never being merged, since both devices will always wait for the message that is buffered on the mutual other device.

One way to tackle this problem is to give only one device the ability to perform a specific merge operation. Since this solution leads to a single point of failure and a bottleneck in terms of execution performance, it is not an ideal solution. It also takes away the possibility to horizontally scale the execution of the data flow by adding new nodes.

Another approach is to add more than one device to the network topology that can perform the merge operation and let each device periodically ask the others for the correlation IDs of the messages in their incoming message queue. This makes the execution horizontally scalable, but also leads to a huge communication overhead between nodes that perform the same merging operation.

A third solution is to add a single additional *gateway node* to the topology that receives all the messages for the specific merge operation. The gateway node then distributes

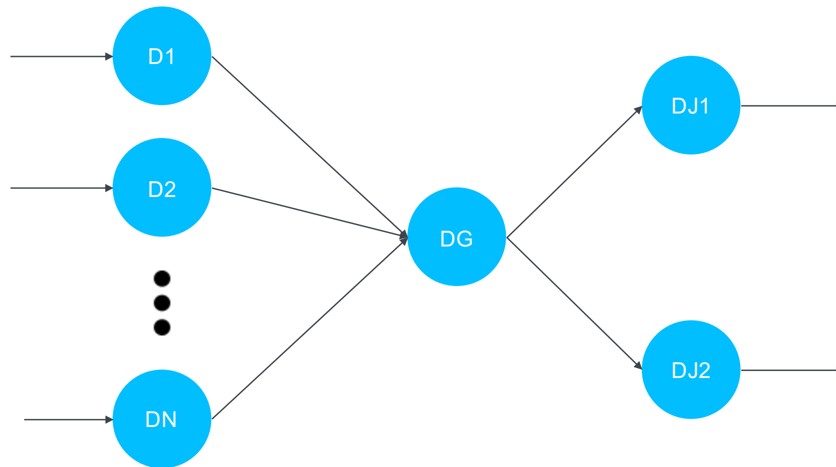

Figure 4.15: Merge operation with a gateway node

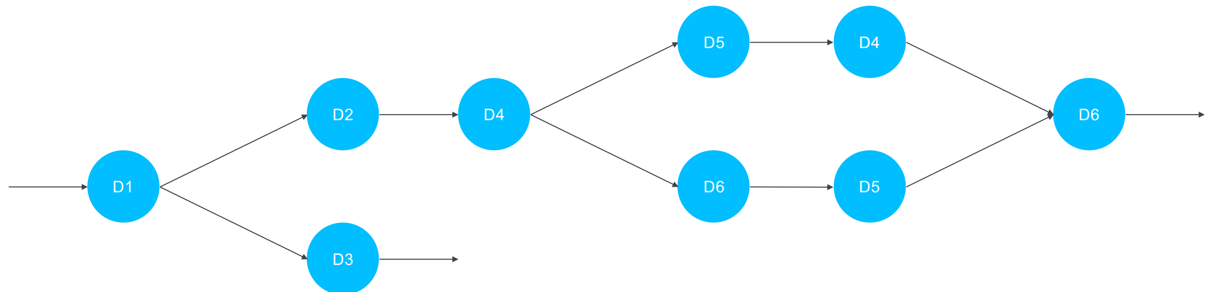

Figure 4.16: Subsequent splits

the messages over all devices that can perform the merge operation. Figure 4.15 shows the conceptual approach of a merging operation with multiple merging nodes and a preceding gateway node. The gateway node *DG* stores messages until it has received every message that is necessary for one merging operation. It then forwards the messages with the same correlation ID to one of the merging devices *DJ1* or *DJ2*. Since this is not an expensive operation in comparison to a merging operation itself, the gateway node is unlikely to be a bottleneck for the performance of the data flow execution.

To handle multiple splitting nodes that follow each other, like, in the data flow example in Figure 4.16, multiple correlation IDs must be added to a message. A list of correlation IDs is added to the messages header, which is treated as a first-in-first-out queue. This way, in the data flow in Figure 4.16, first *D1* adds a correlation ID and then *D4*. *D9* pops the newest message from the correlation ID queue in the messages it receives, which is the correlation ID that has been added by *D4*. So the messages will be merged correctly by *D9* and also all succeeding merging nodes.

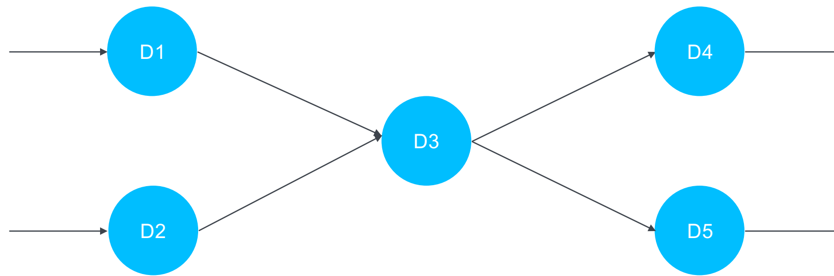

Figure 4.17: Merge and split in a single node

Combinations

A combination of a merge and a split operation in one node, like in the example data flow in Figure 4.17, can also be realized by the messaging engine, without further extensions. An operation in the data flow model can be attached with multiple outgoing nodes and also a merge list. The operation that device *D3* in Figure 4.17 implements could have the following configuration:

- OIID = 3
- Operation ID = Operation_3
- Merge list = 1, 2
- Next OIIDs = 4, 5

4.5 Lifecycle method step 4: Device redistribution

When a new device, which does not necessarily have to be a physical device but can also be, e.g., a virtual machine, is added to or removed from the network topology, the other devices need to be notified about it. To propagate the information about newly added nodes, two approaches can be applied:

1. Publishing the new node via the REST interface on every device by the communication component.
2. Publishing one node via the REST interface of the newly added node and triggering its *hello* mechanism.
3. Sending a multicast message from the newly added device to all other devices.

For the second approach, each device needs an interface for *hello messages*, which can be integrated into the REST interface. To avoid cyclic message forwarding of hello messages, each hello message needs to carry a list of past recipients of the message. Then, every device can forward the hello message to every device in its node repository, except for the ones in the recipient list of the hello message. The third solution only requires adding the new device to the network, which makes it the most appropriate solution. The second and the third solution can also be combined to prevent failures if a device is not reached by the multicast. Multicasting is natively supported by CoAP[5].

When a device is removed from the network topology, other devices will not be able to transmit messages to it anymore, which will not stop the data flow execution as long as there is another device that is able to perform the operations that the removed device could perform. Nevertheless, this slows down the performance of the execution, since devices always try to deliver messages to the removed node. If the device is removed from the network topology on purpose, all three approaches to propagate information about an added node can also be applied to propagate information about a removed node. If the node is removed from the network topology because the device failed, it will not always be able to distribute the information about its own failure. This could be done by the communication component or other devices, when they recognize that the device has failed, by being consistently unable to deliver messages to it, or periodically checking the device's health status.

4.6 Lifecycle method step 5: Data flow retirement

A data flow can be retired by immediately stopping each device from processing messages of the specific data flow, or by only stopping each data source from producing messages of the data flow. In the second approach, one must wait until no more messages, belonging to the stopped data flow, are being processed in the IoT environment. To notice if there are still messages in the environment that belong to a specific data flow, each data source could add a unique message identifier (message ID). So every outgoing message from each data source can be compared with each received message on each data sink. If each message, by means of the message ID, that has been produced by a data source, has been received by a data sink, no more messages are being processed in the environment. Splitting operations need to be considered specifically. Then, the data flow model can be deleted from each device via the REST interface, to prevent that any device no more processes any message of the retired data flow. Stopping each device immediately from processing any message that belongs to a specific data flow, like an

[5]https://tools.ietf.org/html/rfc7390

emergency halt, could also be done via a specific endpoint in the REST interface of each messaging engine.

4.7 Optimizations

Since the execution of data flow models can be very time and failure critical, approaches to improve robustness and performance are covered in this section. Section 4.7.1 discusses robustness optimizations regarding guaranteed delivery, persistence, failure handling and logging. Section 4.7.2 describes performance optimizations in terms of successor node selection and memory management.

4.7.1 Robustness optimizations

The three main topics in terms of robustness optimizations that are discussed are guaranteed delivery, persistence and failure handling.

Guaranteed delivery and persistence

Loosing a message can lead to non performing operations that might be necessary for the successful execution of the data flow model. A message can get lost in the process of transmitting it from one device to another, or during buffering and processing a message on a device. Since IoT environments tend to be more vulnerable for failures and breakdowns, and there is no central instance that keeps the messages, persistence of messages inside the devices is very important. After a breakdown, the IoT environment should be able to continue the execution of the data flow at the point where the breakdown happened, without loosing a single message. This can be achieved by equipping every device with a persistent memory and storing each message persistently before sending the acknowledgment back to the sender after receiving a message. The sender of the message must not delete the message from its memory until it received the acknowledgment. Another mechanism to increase robustness in terms of message persistence is to periodically backup messages, that are stored on each device, on a central component. If then the whole environment breaks down, the messages can be redeployed to each device and the data flow execution can continue from the last backup point.

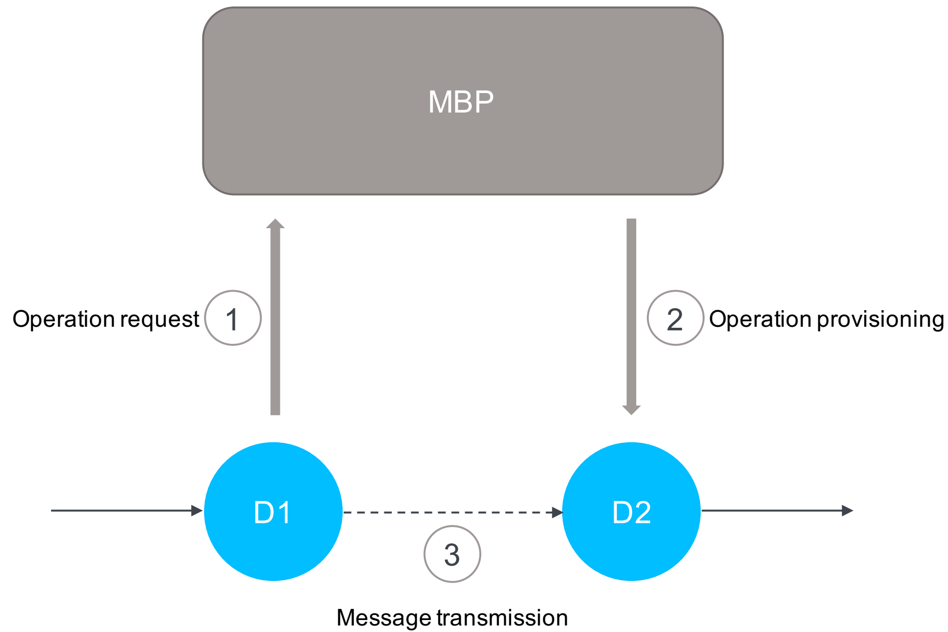

Figure 4.18: MBP fallback

Failure Handling

During the execution of data flow models in distributed IoT environments, multiple failures can occur. Devices, that are necessary for the execution of a data flow model, might not be available. This means that if a device tries to deliver a message to another device listed in its node repository, none of them are responding. This leads to a stop of the whole data flow execution. The first approach to increase robustness in terms of failure handling is by adding multiple nodes to the network that are able to perform a single operation. This makes it possible for the devices to choose between multiple successors if not all of them are available, as described in Section 4.4. The second approach can be applied by a device if no device is available that comes into question as a recipient for an outgoing message. Then, the only solution is to ask the MBP, as described by Franco da Silva et al. [SHKM18], to provision the operation on a new device and add it to the network topology. The approach is depicted in Figure 4.18. Device *D1* cannot find an available device to transmit a message to. It consequently invokes the MBP to provision the desired operation on another device *D2*. After the operation has been provisioned, the execution of the data flow model continues.

Logging

Since there is no central controlling component, there is no central point in the environment that keeps track of the events occurring in all devices. To enable a central monitoring of the whole environment, devices must ship their logs to a central point. The logs could also be served via the REST interface of the messaging engine. Shipping the logs to a central point can be done only once in, e.g., every 24 hours to reduce traffic.

4.7.2 Performance optimizations

Two critical issues in terms of performance are the selection of a successor node and the memory management.

Successor node selection

After performing the operation that is associated with a message, the messaging engine must send the result to another device that is able to perform the next operation in the data flow model. Ideally, for every operation that follows one of the operations a device can perform, the device has a list of multiple other devices, that are able to perform these following operations. This forces the device to decide to which device it sends its results. The outcome of this decision has impact on the performance of the execution of the whole data flow. There are basically two different methods to choose a recipient for a message:

First fit The message producer tries to send the message to the first device in the list. If the devices returns an error, it goes on with the next device in the list. This is repeated until a device returns an acknowledge.

Best fit The message producer retrieves the health information, containing the free resources, via the REST API of every device in the list. Then it sends the message to the device with the most free resources.

The first fit method needs lesser communication iterations than the best fit method. On the other hand, it could lead to sending all messages to the same device, which is an unbalanced use of resources. The best fit method makes the devices distribute load evenly over devices that can perform the same operations. This makes the execution of the data flow horizontally scalable. Performance can be increased by adding multiple devices that are able to perform the same operations. Another parameter that must be regarded for the best fit method is the cost of the channel to each node. This can be

calculated by measuring the round-trip time of each health request, which will not be sufficient as discussed in Section 4.3. Since devices do not store the network topology as the graph described in Section 4.3, but only store the address and operations of each other device, it is difficult to also store the cost of each channel. This exceeds the scope of this thesis and, thus, will not be further discussed.

Memory management

Since guaranteed delivery and persistence of messages are necessary requirements of the messaging engine, memory management is a big issue for performance enhancement. Messages should not get lost when a device fails, so they cannot simply be held in volatile memory. Storing each message on the disc before and after performing its corresponding operation is expensive and, thus, should be the aim of performance optimizations.

A possible optimization is to write messages directly to the filesystem instead of using a database management system. One way to do this is the one Apache Kafka utilizes, as introduced in Chapter 3. It divides its storage into partitions which are further split into segments. On the filesystem, a partition is a directory and each segment is a file inside a partition directory. Messages get written incrementally to the newest segment in each partition parallel until a size limit is reached. Then it creates a new segment. To find the next message in a first-in-first-out manner, an offset is stored for each segment. This method can be adopted to implement the incoming and outgoing message queue of the messaging engine.

5 Implementation

In this chapter, an implementation of the concepts introduced in Chapter 4 is described, serving as proof-of-concept. Since the implemented messaging engine can run on most constrained devices and there are many libraries for it, the chosen programming language to implement the prototypical messaging engine is Python[1] in the version 3.6. The most constrained hardware that the prototype has been tested on was a Raspberry Pi[2]. Figure 5.1 depicts the basic architecture of the implementation in the environment it has been tested in. The implementation is called the *Python IoT Messaging Engine* (PIME). The environment consists of Raspberry Pis and several virtual machines, each running an instance of PIME. PIME is implemented in a three-layered architecture: the infrastructure layer, based on the embedded database system ZODB[3], the service layer and the interface layer, based on CoAPthon Tanganelli, Vallati, and Mingozzi [TVM15]. The three layers of PIME are described in Section 5.1, Section 5.2 and Section 5.3. Parts of the concept of this thesis that are not implemented in the prototype are outlined in Section 5.4.

5.1 Infrastructure layer

To implement the repositories, the embedded database system ZODB has been used. ZODB is a lightweight key-value store, which is capable of transaction processing. This is important, since multiple concurrent threads inside the domain layer are reading and writing data simultaneously, as described in Section 5.2.

5.1.1 Incoming message repository

The incoming message repository has four internal data structures to store messages: *messages*, a queue, for incoming messages that are waiting to be processed, *messages_in_work*

[1] https://www.python.org
[2] https://www.raspberrypi.org
[3] http://www.zodb.org/en/latest/index.html

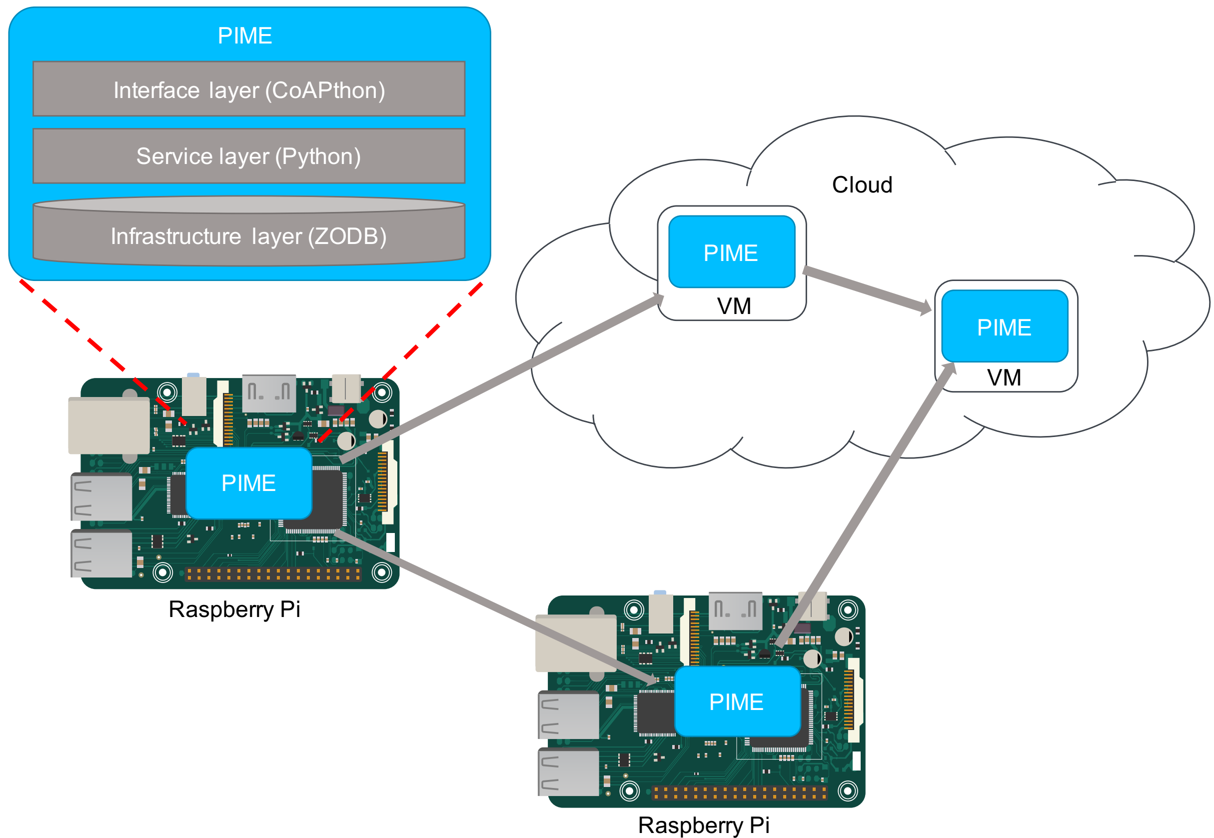

Figure 5.1: Overview of the implementation

for messages that are being processed, *correlation_messages* for messages that need to be joined with other messages that have not been received yet, and *messages_failed* for messages that could no be processed, for example, because the operation is missing. When the message manager takes a new message from the messages queue, a copy of it gets stored in messages_in_work to make sure it doesn't get lost by a failure during its processing. A message is only removed from messages_in_work when it has successfully stored in the outgoing message repository after its processing. When each message for a merging operation has been received, the message gets deleted from correlation_messages and enqueued in the message queue inside a single list. If a new operation is posted via the REST interface, all messages in messages_failed are checked for the new operation and processed if the respective operation has been posted.

5.1.2 Outgoing message repository

The outgoing message repository has similar storing points as the incoming repository, namely: *messages*, *messages_in_sending* and *messages_failed_sending*. Messages inside

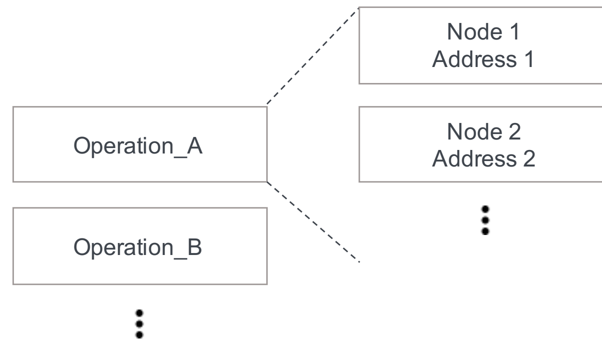

Figure 5.2: Data structure of the node repository

the messages queue of the outgoing message repository are taken out from the message forwarding manager to deliver them to a succeeding node. Messages that could not be delivered to a succeeding node, for example, because there was none available, get stored in messages_failed_sending. After a preconfigured period of time, all messages in messages_failed_sending get rescheduled to the outgoing messages queue, to retry their delivery. While a message is being delivered, a copy of it is stored to messages_in_sending to make sure it doesn't get lost during a breakdown.

5.1.3 Other repositories

The flow repository stores data flow models by their flow ID. The node repository stores, for each operation, a list of nodes which can perform it, as depicted in Figure 5.2. The operation repository stores the URI of the application for each operation the node can perform.

5.1.4 Application wrapper

The application wrapper has only the one method *start_operation*, which receives one message or a list of messages, an application path, and one or multiple OIIDs. If it received a list of messages, it concatenates its payloads. If it received a single message, it takes the message's payload and starts the application with the single or concatenated payload as a command line parameter. The application wrapper expects a JSON data structure from the application via its standard output. If the application returns a JSON array, it creates a new outgoing message for each JSON object in the array. If the application returns a single JSON object, it creates one message with the object as payload. The application wrapper then returns the JSON array or object back to the message manager.

5.1.5 Resource monitor

The resource monitor has only a single function which returns the health status as true or a false, of the device by using a simple metric. It measures the remaining memory and disc capacities with the psutil[4] library and compares both with a given threshold, which can be adjusted when starting the application. When the threshold is met, it replies with false, otherwise with true.

5.2 Domain Layer

The main parts of the domain layer are the message manager and the message forwarding manager. Both of them run in a separate thread and act like polling consumers [HW04]. The domain layer also consists of several service classes, providing the functionality of the infrastructure layer.

5.2.1 Message and message forwarding manager

When the message manager has finished processing a message, it requests the next message in the message queue of the incoming message repository. The message forwarding manager acts in a similar way by waiting until there is a message in the outgoing message queue. Each manager runs in a separate thread and constantly requests the next message from the corresponding repository, until it receives one. The domain layer also manages the periodical rescheduling of failed messages inside the incoming and outgoing message repositories.

5.2.2 Services

The domain layer has also multiple classes that act as services for the repositories. Those encompass the *data_flow_service*, the *message_service*, the *node_service*, the *operation_service* and the *resource_service*. All of these classes serve the basic functionality of the respective repositories, except for the resource_service, it is the implementation of the service layer part of the resource monitor. The message_service is the interface for both, the incoming_message_repository and the outgoing_message_repository. It also handles the separate storing of messages that need to be merged.

[4]https://pypi.org/project/psutil/

5.3 Interface layer

The REST interface and the message consumer has been implemented using CoAP [SHB14], since CoAP is a suitable protocol for the IoT [BCS12]. The message consumer has also been implemented as part of the REST interface.

The Python library used in the implementation is CoAPthon[5] by Tanganelli, Vallati, and Mingozzi [TVM15].

5.3.1 REST interface

The messaging engine has the following interfaces:

Data flow To post and delete data flow models to the device. Listing 5.4 shows an example of a data flow in its JSON representation. Each subelement of "flow" represents a node in the data flow model, The key of the subelement is the OIID of the according node.

Health To retrieve the health status of the device, consisting of a general flag to state if the device is able to consume messages and the amount of messages in its queues. Listing 5.6 shows an example of a health resource in its JSON representation.

Hello To post a hello message to the device, as the example depicted in Listing 5.1. The array *recipients* represents a list of each node that already received the hello message, to make sure that each device receives each hello message only once.

Message The message resource is the implementation of the message consumer. Listing 5.3 shows an example of a message in JSON representation. Messages are trasnsmitted via CoAPs POST operation.

Node The node resource can be used to post, get and delete nodes to or from the device. Listing 5.2 shows an example of a node resource in JSON representation. The array *operations* is a list of each operation the node is able to perform. The *name* of the node must be unique in the IoT environment, as well as the *address*.

Operation The operation resource can be used to post, get and delete operations to or from the device. Listing 5.5 shows an example of an operation resource in its JSON implementation. The value of the field *application* is the path to the application that the application wrapper must call to perform the operation.

[5] https://github.com/Tanganelli/CoAPthon

Listing 5.1 Example of a hello resource in JSON representation


```
{
      "recipients": [
      "vm1",
      "vm2"
      ],
      "node": {
            "name": "vm3",
            "address": "127.0.0.1",
            "operations": [
                  "sleep_1",
                  "sleep_5"
            ]
      }
}
```


Listing 5.2 Example of a node resource in JSON representation


```
{
      "address": "192.168.209.164",
      "name": "vm3",
      "operations": [
            "operation_1",
            "operation_2"
      ]
}
```


Trigger hello A CoAP POST request with an empty payload triggers the hello mechanism of the device, which is executed by the hello forwarder which is described in Section 5.3.2.

5.3.2 Hello forwarder

The *hello forwarder* creates a hello resource and sends it to each node in the node repository when the hello mechanism is triggered. It also responds to the original sender of a hello message with a post to the senders node interface with its own node

Listing 5.3 Example of a message resource in JSON representation


```
{
      "flow_id": "parallel_flow_1",
      "payload": "Hallo;Welt",
      "oiid": "1"
}
```

Listing 5.4 Example of a data flow resource in JSON representation

```
{
      "flow_id": "parallel_flow_1",
      "flow": {
            "1": {
                  "operation": "test_app_split",
                  "next_oiid_list": ["2", "3"]
            },
            "2": {
                  "operation": "test_app",
                  "next_oiid": "4"
            },
            "3": {
                  "operation": "test_app_vm3",
                  "next_oiid": "4"
            },
            "4": {
                  "operation": "test_app_join",
                  "next_oiid": "none",
                  "join_list": ["2", "3"]
            }
      }
}
```

Listing 5.5 Example of an operation resource in JSON representation

```
{
      "operation_name": "operation_1",
      "application": "/home/ubuntu/test_operation.py"
}
```

description. Thus, the sender of a hello message automatically knows each node the message has passed. When a device receives a hello message, the hello forwarder adds the name of its own node to the message's recipient list and forwards the message to each node in the node repository except for those which are already in the recipient list.

Listing 5.6 Example of a health resource in JSON representation

```
{
      "available": "yes",
      "message_count": 12
}
```

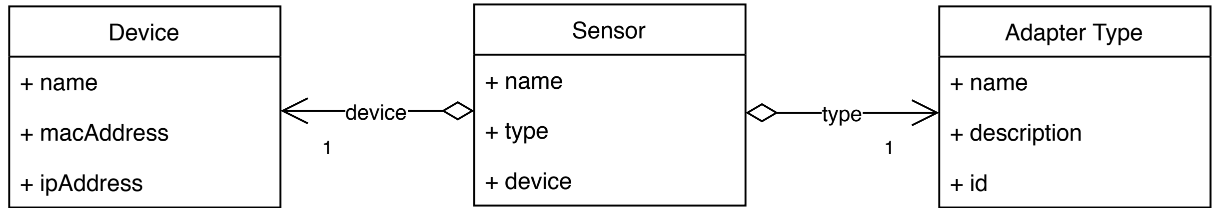

Figure 5.3: MBP data model

5.3.3 MBP adapter

Since the functionality to request an operation on a new device is not directly implemented in the MBP, most of its logic has been implemented inside the MBP adapter. Figure 5.3 shows the UML representation of the necessary part of the data model of the MBP. The MBP adapter first retrieves a list of sensors, a list of devices and a list of adapter types from the MBP. An adapter type in the MBP can be seen as a operation in this thesis, since each adapter type in the MBP is associated with one or multiple files, that can be applications which can perform operations in the sense of this thesis. The MBP adapter then seeks for the minimum busy device in terms of number of sensors running on a device. Then it looks for the ID of an adapter type with the same name as the operation that should be deployed, so it can create a new sensor from the wanted adapter type. After the sensor has been created, it can deploy it on the device that has been picked.

5.3.4 Message producer

The message forwarder can either forward a single message to the node that is able to execute the desired operation and has the most available resources, or send a list of messages to as many different nodes as possible. The message producer has also been implemented using CoAPthon. He retrieves the health resource via the REST interface of each qualified node in the node repository and then posts the message via the message resource of the REST interface of the node with the most free resources. A node is qualified if he is able to perform the operation that is next in the data flow model.

5.4 Special cases

Not all data flow patterns that have been introduced in Section 4.4.4 have also been implemented in the prototype. These comprise the exclusive split and the merge with

a gateway node. The gateway node is an extension of the basic concept to increase performance in a merging data flow. The exclusive split makes it necessary for PIME to understand the semantics of the data, which is not described conceptually.

6 Evaluation

The prototypical implementation proves that the execution of data flow models in distributed IoT environments without a central execution engine is possible by utilizing the concept described in this thesis.

The following paragraph evaluates whether the concept described in Chapter 4 satisfies the requirements defined in Chapter 4:

Requirement R1: Robustness
: Section 4.7.1 describes how robustness in terms of persistence and guaranteed delivery can be increased, and how messages can be restored after a breakdown. In Section 4.7.1, a concept to handle failures that occur when no device is available that is qualified for a successor in terms of the data flow is described. Both methods described in Section 4.7.2 select a succeeding device regarding the health status of a device. Thus, messages are only transmitted to devices that are able to consume them. These concepts significantly increase robustness, but do not eliminate the fact that data is solely handled by IoT devices and not by a central component, e.g., running in the cloud.

Requirement R2: Loose Coupling
: In Section 4.4.1, a generic message format to exchange data between devices is described. The architecture includes a uniform REST interface and message producers and consumer which abstract the message transmission from the underlying platform technology. The application wrapper abstracts the messaging engine from the underlying application. This makes the concept autonomous in terms of platform technology and message format. Location autonomy is not achieved because devices need to know the physical address of another device to communicate with them. Consequently, Requirement R2 is not fully satisfied.

Requirement R3: Persistence and Guaranteed Delivery
: Messages that are received via the message consumer are immediately stored on persistent memory. The protocol defined in Section 4.4.3 makes sure that a device stores an outgoing message at least until it receives an acknowledgment. This leads to a guaranteed delivery of messages between devices. Furthermore, Section 4.7.1 describes how messages can be recovered after loosing persistent memory, which fully satisfies requirement R3.

Requirement R4: Horizontal Scalability
Section 4.4.4 describes a concept for the parallel split pattern, where messages get evenly distributed over available devices. Furthermore, Section 4.7.2 describes how messages can be evenly distributed over all devices in the IoT environment which can perform the same operation. Consequently, adding more devices to the environment that can perform the same operation leads to a more efficient execution of the data flow model.

Requirement R5: Vertical Scalability
Receiving, processing and forwarding of messages are decoupled from each other by using separate threads and queues. Thus, increasing processing power of a device makes receiving, storing and forwarding messages faster. Processing a message depends on the specific application and possibly on the physical part of the device and, thus, cannot generally be made more efficient by increasing the processing power of the device.

Requirement R6: Monitoring Via the REST interface of the messaging engine described in Section 4.4.2, current health information of devices can be retrieved. In Section 4.7.1, a method the ship logging information of the devices to a central component is described. This makes it possible to monitor the whole environment from a central component, without loosing benefits of the execution without a central execution engine.

The weak point of the concept introduced in this thesis is robustness. The approaches to increase robustness for the execution that have been regarded in Section 4.7.1, such as a periodical backup of messages, the MBP fallback, and the central logging point, increase robustness fundamentally but may not be sufficient in a real-life production system.

However, whether the execution of a data flow model is more efficient by the means of this thesis or in the conventional approach has not been proven yet. Also, whether the robustness of the solution will suffice a real life application has not been tested. If each node has enough successor nodes for each operation in its node repository the execution of the data flow model without a central execution engine can be more efficient than the execution with one, because at least the communication overhead of sending the data via the execution engine can be omitted. Another issue that has not been considered in this thesis is security and privacy, which is still a challenge in the domain of IoT [SRGC15]. The lack of a central execution engine may make the environment more vulnerable for attacks.

Not every data flow pattern described by Reimann and Schwarz [RS+13] has been analyzed in Section 4.4.4 in terms of its feasibility with the execution engine. Nevertheless, the basic patterns can be handled by the execution engine and it could be extended for

the remaining ones. The only problem will be to make the messaging engine understand the semantics of the data, which is necessary to realize patterns like the exclusive split.

The real benefits of the concepts of this thesis will emerge when IoT environments will be much more self-organizing than now [AT13]. This would help with the problem of finding the right successor and, thus, increase the robustness.

7 Summary and Future Work

In this thesis, a concept for the execution of data flow models in distributed IoT environments is introduced, which is based on messaging and refrains from a central execution engine. To achieve this, a lifecycle method is proposed which consists of the the five steps data flow modeling, network topology creation or modification, data flow execution, device redistribution and data flow retirement.

The data flow is modeled as a directed graph in the pipes-and-filters style, where each node, i.e. filter, stands for an operation. a network topology is an undirected, connected and weighted graph with a node for every device in the environment and an edge between to nodes if the respective devices are able to communicate with each other. The data flow model and the network topology is distributed over the devices to ensure the execution of the data flow model.

To realize the execution of the data flow model, an architecture for a messaging engine is proposed. It is a three-layered architecture which most important parts are a message consumer, a message producer, an incoming and an outgoing message queue and an application wrapper. The application wrapper executes an application on the IoT device that performs the operation according to the data flow model. The result of the application is transformed into a new message and enqueued into the outgoing message queue. The message producer sends the outgoing message to the message consumer of another device that is able to perform the next operation of the data flow model. The message consumer puts messages directly into the incoming message queue. Every device in the IoT environment is equipped with the messaging engine, so the devices are able to send message to each other. A message consists of a header and a body, whereby the body is the data that is used for the execution of the data flow model. Several data flow patterns are regarded in terms of their feasibility with the given messaging engine: sequential data flow, parallel split, exclusive split and merge.

When new devices are added to the environment or devices are removed from it, the device redistribution takes place. Other devices need to be informed about the redistribution what can be achieved via the communication component or milticasting mechanisms. When the data flow is no longer needed, it is retired.

Optimizations to increase robustness and performance are also regarded, those compass guaranteed delivery and persistence, failure handling, logging, successor node selection

and memory management. Guaranteed delivery of messages can be ensured if every consumed message is persistently stored before replying with an acknowledging response. Since there is no central execution engine that can be monitored, shipping logs from the devices to a central component is useful to track events of the whole environment. Two fundamental strategies to choose a target node for a message if multiple nodes are available are evaluated: the first fit and the best fit method. Since message need to be consistently stored to ensure guaranteed delivery, improvements in memory management are useful to increase performance, like incrementally storing messages directly on the filesystem.

An implementation to prove the functionality of the concept is also described. It is implemented in Python and makes use of a CoAP library and the ZODB database, an integrated key-value database. The implementation has been tested in a distributed environment consisting of Raspberry Pis and virtual machines.

Future work

A possible future work is the implementation of the remaining data flow patterns described by Reimann and Schwarz [RS+13], which have not been implemented in the prototype. This makes the implementation more suitable for real-life production systems.

Also, a testing environment can be arranged to measure whether the execution without a central execution engine is strikingly more efficient than the execution with one.

The performance of the implementation described in this thesis can be compared to the performance of common execution engines, for example BPEL [Sta07] engines like OW2 Orchestra[1] or the IBM WebSphere Process Server[2]. The concept can be evaluated and extended in terms of its real-time capabilities by regarding the "The 8 requirements of real-time stream processing" by Stonebraker, Çetintemel, and Zdonik [SÇZ05].

Furthermore, ways to ensure privacy and security can be determined, which is still an issue in the Internet of Things, since constrained devices in distributed environments are more vulnerable [RZL13].

Finding a way to connect the messaging engine with the semantics of the message and sensor data, for example by introducing a extension system or a meta model would be useful to implement data flow patterns like the exclusive split.

[1] https://projects.ow2.org/view/orchestra/
[2] https://www-01.ibm.com/software/integration/wps/

Equipping IoT devices with a discovery process [ABB17] or self-organizing mechanisms [AT13; HWBM16a; HWBM16b] would be useful to increase robustness and usability.

A concept to consider channel cost, also regarding distance, roundtrip time and communication technology can be elaborated to enhance the performance of the execution of data flow models. Mechanisms to optimizing the execution in terms of performance by using feedback from succeeding devices can also be implemented into the messaging engine.

Bibliography

[AAS13] C. C. Aggarwal, N. Ashish, A. Sheth. “The internet of things: A survey from the data-centric perspective.” In: *Managing and mining sensor data*. Springer, 2013, pp. 383–428 (cit. on p. 11).

[ABB17] M. Aziez, S. Benharzallah, H. Bennoui. “Service discovery for the Internet of Things: Comparison study of the approaches.” In: *Control, Decision and Information Technologies (CoDIT), 2017 4th International Conference on*. IEEE. 2017, pp. 0599–0604 (cit. on p. 65).

[ASS+99] M.K. Aguilera, R.E. Strom, D.C. Sturman, M. Astley, T.D. Chandra. “Matching events in a content-based subscription system.” In: *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*. ACM. 1999, pp. 53–61 (cit. on p. 17).

[AT13] A.P. Athreya, P. Tague. “Network self-organization in the internet of things.” In: *Internet-of-Things Networking and Control (IoT-NC), 2013 IEEE International Workshop of*. IEEE. 2013, pp. 25–33 (cit. on pp. 61, 65).

[BCS12] C. Bormann, A.P. Castellani, Z. Shelby. “Coap: An application protocol for billions of tiny internet nodes.” In: *IEEE Internet Computing* 16.2 (2012), pp. 62–67 (cit. on pp. 18, 53).

[BD15] R. Bruns, J. Dunkel. *Complex event processing: komplexe Analyse von massiven Datenströmen mit CEP*. Springer-Verlag, 2015 (cit. on p. 17).

[BEBT16] M. Bermudez-Edo, T. Elsaleh, P.M. Barnaghi, K.L. Taylor. “IoT-Lite: A Lightweight Semantic Model for the Internet of Things.” In: *UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld*. 2016, pp. 90–97 (cit. on p. 22).

[BG11] R. Baheti, H. Gill. “Cyber-physical systems.” In: *The impact of control technology* 12.1 (2011), pp. 161–166 (cit. on pp. 11, 18).

[CM12] G. Cugola, A. Margara. “Processing flows of information: From data stream to complex event processing.” In: *ACM Computing Surveys (CSUR)* 44.3 (2012), p. 15 (cit. on pp. 11, 15, 16).

[Coc14] A. Cocchia. “Smart and digital city: A systematic literature review.” In: *Smart city*. Springer, 2014, pp. 13–43 (cit. on p. 11).

[DG08] J. Dean, S. Ghemawat. "MapReduce: simplified data processing on large clusters." In: *Communications of the ACM* 51.1 (2008), pp. 107–113 (cit. on p. 39).

[EFGK03] P. T. Eugster, P. A. Felber, R. Guerraoui, A.-M. Kermarrec. "The many faces of publish/subscribe." In: *ACM computing surveys (CSUR)* 35.2 (2003), pp. 114–131 (cit. on pp. 19, 22, 35).

[ENL11] O. Etzion, P. Niblett, D. C. Luckham. *Event processing in action*. Manning Greenwich, 2011 (cit. on p. 17).

[FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee. *Hypertext transfer protocol–HTTP/1.1*. Tech. rep. 1999 (cit. on p. 18).

[FT00] R. T. Fielding, R. N. Taylor. *Architectural styles and the design of network-based software architectures*. Vol. 7. University of California, Irvine Doctoral dissertation, 2000 (cit. on p. 15).

[Har06] R. Harper. *Inside the smart home*. Springer Science & Business Media, 2006 (cit. on p. 11).

[HB16] P. Hirmer, M. Behringer. "FlexMash 2.0–Flexible Modeling and Execution of Data Mashups." In: *International Rapid Mashup Challenge*. Springer. 2016, pp. 10–29 (cit. on pp. 11, 21, 27).

[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 (Dezember 2016), pp. 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 (cit. on p. 11).

[HHLD11] J. Han, E. Haihong, G. Le, J. Du. "Survey on NoSQL database." In: *Pervasive computing and applications (ICPCA), 2011 6th international conference on*. IEEE. 2011, pp. 363–366 (cit. on p. 37).

[Hir18] P. Hirmer. "Anforderungsbasierte Modellierung und Ausführung von Datenflussmodellen." In: (2018) (cit. on p. 21).

[HW04] G. Hohpe, B. Woolf. *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley Professional, 2004 (cit. on pp. 26, 52).

[HWBM16a] P. Hirmer, M. Wieland, U. Breitenbücher, B. Mitschang. "Automated Sensor Registration, Binding and Sensor Data Provisioning." Englisch. In: *Proceedings of the CAiSE'16 Forum, at the 28th International Conference on Advanced Information Systems Engineering (CAiSE 2016)*. Vol. 1612. CEUR Workshop Proceedings. Ljubljana, Slovenia: CEUR-WS.org, June 2016, pp. 81–88. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2016-22&engl=0 (cit. on p. 65).

[HWBM16b] P. Hirmer, M. Wieland, U. Breitenbücher, B. Mitschang. "Dynamic Ontology-based Sensor Binding." Englisch. In: *Advances in Databases and Information Systems. 20th East European Conference, ADBIS 2016, Prague, Czech Republic, August 28-31, 2016, Proceedings*. Vol. 9809. Information Systems and Applications, incl. Internet/Web, and HCI. Prague, Czech Republic: Springer International Publishing, Aug. 2016, pp. 323–337. ISBN: 978-3-319-44039-2. DOI: 10.1007/978-3-319-44039-2. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2016-25&engl=0 (cit. on p. 65).

[HWS+15] P. Hirmer, M. Wieland, H. Schwarz, B. Mitschang, U. Breitenbücher, F. Leymann. "SitRS-a situation recognition service based on modeling and executing situation templates." In: *Proceedings of the 9th symposium and summer school on service-oriented computing*. 2015, pp. 113–127 (cit. on p. 21).

[HWS+16] P. Hirmer, M. Wieland, H. Schwarz, B. Mitschang, U. Breitenbücher, S. G. Sáez, F. Leymann. "Situation recognition and handling based on executing situation templates and situation-aware workflows." Englisch. In: *Computing* (Oktober 2016), pp. 1–19. DOI: 10.1007/s00607-016-0522-9. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=ART-2016-12&engl=0 (cit. on p. 11).

[Kay03] D. Kaye. *Loosely coupled: the missing pieces of Web services*. RDS Strategies LLC, 2003 (cit. on p. 26).

[KK03] M. F. Kaashoek, D. R. Karger. "Koorde: A simple degree-optimal distributed hash table." In: *International Workshop on Peer-to-Peer Systems*. Springer. 2003, pp. 98–107 (cit. on p. 19).

[LBK15] J. Lee, B. Bagheri, H.-A. Kao. "A cyber-physical systems architecture for industry 4.0-based manufacturing systems." In: *Manufacturing Letters* 3 (2015), pp. 18–23 (cit. on pp. 11, 18).

[LCW08] D. Lucke, C. Constantinescu, E. Westkämper. "Smart factory-a step towards the next generation of manufacturing." In: *Manufacturing systems and technologies for the new frontier*. Springer, 2008, pp. 115–118 (cit. on p. 11).

[LF98] D. C. Luckham, B. Frasca. "Complex event processing in distributed systems." In: *Computer Systems Laboratory Technical Report CSL-TR-98-754. Stanford University, Stanford* 28 (1998) (cit. on p. 16).

[LLH+13] M. Liu, T. Leppanen, E. Harjula, Z. Ou, A. Ramalingam, M. Ylianttila, T. Ojala. "Distributed resource directory architecture in Machine-to-Machine communications." In: *Wireless and Mobile Computing, Networking and Communications (WiMob), 2013 IEEE 9th International Conference on*. IEEE. 2013, pp. 319–324 (cit. on p. 23).

[Luc02] D. Luckham. *The power of events*. Vol. 204. Addison-Wesley Reading, 2002 (cit. on pp. 17, 21).

[Luc11] D. Luckham. "Event Processing Glossary-Version 2.0, Event Processing Technical Society." In: (2011) (cit. on p. 17).

[MBD+12] A. McAfee, E. Brynjolfsson, T. H. Davenport, D. Patil, D. Barton. "Big data: the management revolution." In: *Harvard business review* 90.10 (2012), pp. 60–68 (cit. on p. 11).

[MCB+11] J. Manyika, M. Chui, B. Brown, J. Bughin, R. Dobbs, C. Roxburgh, A. H. Byers. "Big data: The next frontier for innovation, competition, and productivity." In: (2011) (cit. on p. 11).

[Meu95] R. Meunier. "The pipes and filters architecture." In: *Pattern languages of program design*. ACM Press/Addison-Wesley Publishing Co. 1995, pp. 427–440 (cit. on pp. 15, 21, 28).

[MG+11] P. Mell, T. Grance, et al. "The NIST definition of cloud computing." In: (2011) (cit. on p. 11).

[MSDC12] D. Miorandi, S. Sicari, F. De Pellegrini, I. Chlamtac. "Internet of things: Vision, applications and research challenges." In: *Ad hoc networks* 10.7 (2012), pp. 1497–1516 (cit. on pp. 11, 17, 18).

[PL03] R. Perrey, M. Lycett. "Service-oriented architecture." In: *Applications and the Internet Workshops, 2003. Proceedings. 2003 Symposium on*. IEEE. 2003, pp. 116–119 (cit. on p. 15).

[Pos80] J. Postel. *User datagram protocol*. Tech. rep. 1980 (cit. on p. 18).

[Pos81] J. Postel. "Transmission control protocol specification." In: *RFC 793* (1981) (cit. on p. 18).

[RS+13] P. Reimann, H. Schwarz, et al. “Datenmanagementpatterns in Simulationsworkflows.” In: *BTW*. 2013, pp. 279–293 (cit. on pp. 16, 38, 60, 64).

[RZL13] R. Roman, J. Zhou, J. Lopez. “On the features and challenges of security and privacy in distributed internet of things.” In: *Computer Networks* 57.10 (2013), pp. 2266–2279 (cit. on p. 64).

[SBK13] Z. Shelby, C. Bormann, S. Krco. “CoRE resource directory.” In: (2013) (cit. on p. 23).

[Sch01] R. Schollmeier. “A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications.” In: *Peer-to-Peer Computing, 2001. Proceedings. First International Conference on*. IEEE. 2001, pp. 101–102 (cit. on pp. 18, 22).

[SÇZ05] M. Stonebraker, U. Çetintemel, S. Zdonik. “The 8 requirements of real-time stream processing.” In: *ACM Sigmod Record* 34.4 (2005), pp. 42–47 (cit. on p. 64).

[SHB14] Z. Shelby, K. Hartke, C. Bormann. *The constrained application protocol (CoAP)*. Tech. rep. 2014 (cit. on pp. 18, 35, 53).

[She12] Z. Shelby. *Constrained RESTful environments (CoRE) link format*. Tech. rep. 2012 (cit. on p. 23).

[SHKM18] A. C. F. da Silva, P. Hirmer, R. Koch Peres, 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 at Percom*. 2018 (cit. on pp. 21, 46).

[SHWM16] A. C. F. da Silva, P. Hirmer, M. Wieland, B. Mitschang. “SitRS XT-Towards Near Real Time Situation Recognition.” In: *Journal of Information and Data Management* 7.1 (2016), p. 4 (cit. on p. 21).

[SRGC15] S. Sicari, A. Rizzardi, L. A. Grieco, A. Coen-Porisini. “Security, privacy and trust in Internet of Things: The road ahead.” In: *Computer networks* 76 (2015), pp. 146–164 (cit. on p. 60).

[Sta07] O. Standard. “Web services business process execution language version 2.0.” In: *URL: http://docs. oasis-open. org/wsbpel/2.0/OS/wsbpel-v2. 0-OS. html* (2007) (cit. on p. 64).

[TAD11] K. Taylor, A. Ayyagari, D. De Roure. “SEMANTIC SENSOR NETWORKS.” In: (2011) (cit. on p. 22).

[TVM15] G. Tanganelli, C. Vallati, E. Mingozzi. “CoAPthon: Easy development of CoAP-based IoT applications with Python.” In: *Internet of Things (WF-IoT), 2015 IEEE 2nd World Forum on*. IEEE. 2015, pp. 63–68 (cit. on pp. 49, 53).

[VF13] O. Vermesan, P. Friess. *Internet of things: converging technologies for smart environments and integrated ecosystems*. River Publishers, 2013 (cit. on pp. 11, 18).

[WS05] K. Wehrle, R. Steinmetz. “Peer-to-Peer systems and applications.” In: *LNCS* 3485 (2005) (cit. on p. 18).

[XYWV12] F. Xia, L. T. Yang, L. Wang, A. Vinel. “Internet of things.” In: *International Journal of Communication Systems* 25.9 (2012), pp. 1101–1102 (cit. on p. 17).

[ZHKL09] O. Zweigle, K. Häussermann, U.-P. Käppeler, P. Levi. “Supervised learning algorithm for automatic adaption of situation templates using uncertain data.” In: *Proceedings of the 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human*. ACM. 2009, pp. 197–200 (cit. on p. 21).

All links were last followed on November 05, 2018.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature