SICS Software-Intensive Cyber-Physical Systems (2020) 35:39–48
https://doi.org/10.1007/s00450-019-00410-z

SPECIAL ISSUE PAPER

A lightweight messaging engine for decentralized data processing in the Internet of Things

Daniel Del Gaudio[1] · Pascal Hirmer[1]

Published online: 26 August 2019

Abstract
Today, Internet of Things applications are available throughout many different domains (manufacturing, health, cities, homes), enabling a high degree of automation to ease people's lives. For example, automated heating systems in a smart home can lead to reduced costs and an increased comfort for the residents. In the IoT, situations can be detected through interpretation of data produced by heterogeneous sensors, which typically lead to an invocation of actuators. In such applications, sensor data is usually streamed to a central instance for processing. However, especially in time-critical applications, this is not feasible, since high latency is an issue. To cope with this problem, in this paper, we introduce an approach for decentralized data processing in the IoT. This leads to decreased latency as well as a reduction of costs.

Keywords Internet of Things · Decentralization · Low latency · Data processing

1 Introduction

The Internet of Things (IoT) is an evolving paradigm that enables new applications, such as *smart homes*, *smart factories*, and *smart cities* [5]. These applications contain heterogeneous devices, equipped with *sensors* and *actuators*, that communicate through standard internet protocols to reach common goals [25]. In IoT applications, data is produced by sensors and is then typically processed by data streaming systems (e.g., using complex event processing) to detect extraordinary events, i.e., situations [7]. Detection then leads to an invocation of actuators. For example, the recognition of unusual patterns in data produced by heat sensors distributed throughout a smart factory might lead to an alert signal or the invocation of a sprinkler system.

In most cases, data is streamed from the sensors to a central execution system for processing. To enable high scalability, availability, and thus, robustness of the execution system, virtualized cloud environments are often used However, since cloud-based environments are usually hosted by external off-premise cloud providers in large data centers, data needs to travel long distances, which leads to high latency for data processing. This is depicted on the left side of Fig. 1.

In this example, the IoT devices *Raspberry Pi 1* and *Arduino* collect data from the sensors *Sensor 1* and *Sensor 2*, which is then shipped to a data processing system in the cloud via *Raspberry Pi 2* and the *Edge Server*. The data processing is then conducted in the cloud environment. The result is transmitted back to *Raspberry Pi 2* where it is used to invoke an actuator. If actuators are invoked based on recognized situations, as described in this scenario, high latency could be an issue, especially in time and security critical applications (e.g., emergency systems). Even if data processing is conducted centralized in a private cloud environment, i.e., on the same premises as the execution environment, communication paths are oftentimes still too long for time-critical or real-time applications, respectively.

To cope with this issue, we propose an approach for decentralized processing of IoT data near the sources (the sensors). This approach is depicted in Fig. 1 on the right. In this example, the data is processed directly on the IoT devices. The result of the *Edge Server* is transmitted to *Raspberry Pi 2*, instead of traveling the detour through the off-premise cloud infrastructure. The key aspect of our approach is that data is processed on the devices themselves and transmitted among them without a central instance, which leads to the following advantages: (i) reduced latency, since data is processed on-premise and near to the sources, (ii) increased flexibility

✉ Daniel Del Gaudio
daniel.delgaudio@ipvs.uni-stuttgart.de

Pascal Hirmer
pascal.hirmer@ipvs.uni-stuttgart.de

[1] Universitätsstraße 38, 70569 Stuttgart, Germany

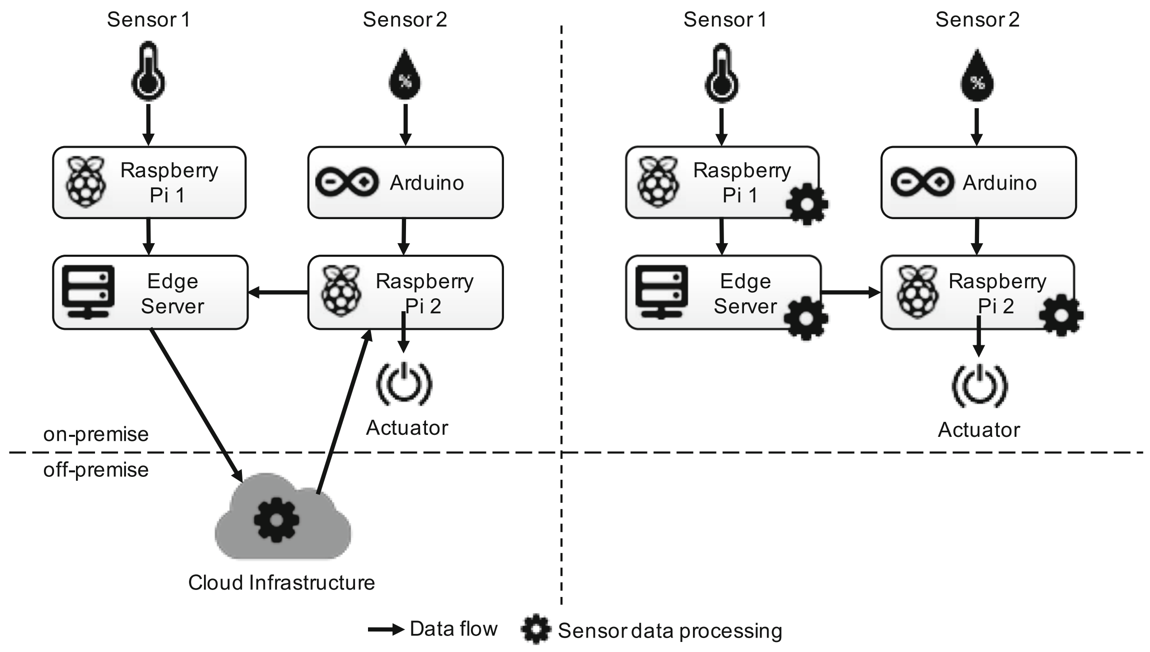

Fig. 1 Comparison of the state of the art (left) and our approach (right)

in regard to device mobility, and (iii) reduced costs, since existing resources available within the IoT environment are utilized.

Our contributions include (i) a holistic method for decentralized data processing in the IoT and (ii) a light-weight embedded IoT messaging engine to implement this method. We validate our approach through a prototypical implementation and comparative runtime measurements.

The remainder of this paper is structured as follows: Sect. 2 describes a motivating scenario that will be referenced throughout this paper. In Sect. 3, an overview of related scientific work is presented. Section 4 introduces the main contribution: an approach for decentralized data processing in the IoT. Section 5 describes the prototypical implementation and evaluates our approach. Finally, Sect. 6 summarizes the paper and gives an outlook on future work.

2 Motivating scenario

In this section, we describe a motivating scenario, which is used to explain our approach throughout this paper. The scenario consists of a conveyor belt that transports items between multiple production machines. Figure 2 depicts the IoT environment of the conveyor belt installation.

IoT Device 1 retrieves the location of each item on the conveyor belt from the distance sensor. If the item needs to be relocated, IoT Device 1 invokes the Item Relocater to put the item to the proper position. IoT Device 1 passes the position data to IoT Device 2 and IoT Device 3. IoT Device 2 then retrieves the data on the RFID tag of the item from the RFID Reader and IoT Device 3 retrieves a picture of it from the Camera. Both devices aggregate the data received from IoT Device 1 and forward it to the Edge Server, since the computation of visual data requires more computing capabilities. The Edge Server computes the path to the correct machine for each item based on the data it received from IoT Device 2 and IoT Device 3. It forwards the path information to the conveyor belts control unit, to lead the item to the correct machine.

The scenario focuses on machine-to-machine (M2M) communication and requires real-time capabilities, thus, demonstrates the strengths of our approach. There are two main goals that need to be achieved: (i) dynamic distribution of items amongst production machines depending on the item's type and condition, and (ii) a timely reaction to wrongly positioned items. The item's type, condition, and position on the conveyor belt is recognized with multiple sensors, such as cameras, distance sensors, and RFID readers. Based on this data, machines need to determine the path for a given item to a specific machine and inform all succeeding machines about it via M2M communication. The aimed solution also requires real-time capabilities because machines need to react to wrongly positioned items in a timely manner. Performing this calculation on a central component, e.g., a virtual machine on the cloud, is expensive and not feasible due to long communication paths. Furthermore, data would need to leave the factory which leads to privacy and security issues. The goal of this paper is to create a solution to process data on local hardware in a decentralized manner to satisfy the requirements of this and other scenarios.

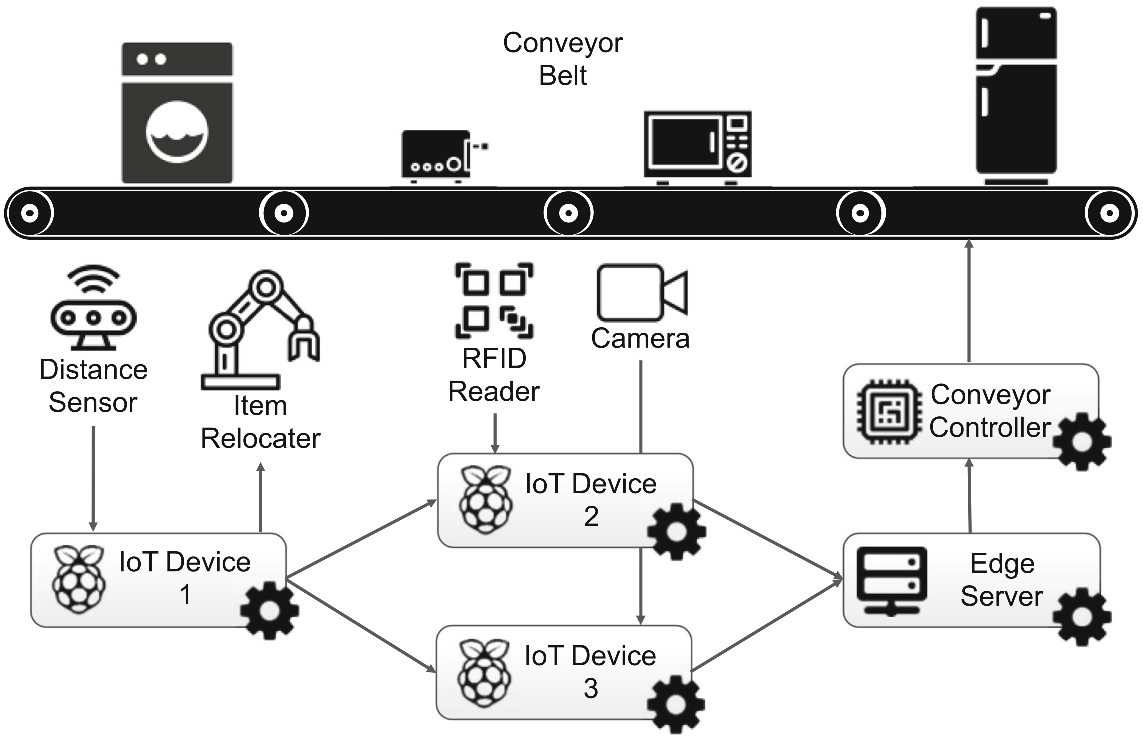

Fig. 2 Data from multiple sensors is used to control the conveyor belt and lead each product to its correct destination

3 Related work

Franco da Silva et al. [8] introduce an approach to model hardware requirements for CEP queries, hardware capabilities of smart devices, and corresponding algorithms to place each query on an appropriate device according to the respective requirements and capabilities. They also introduce the *Multi-purpose Binding and Provisioning Platform* (MBP), a software platform that enables automated binding of devices in an IoT environment as well as software provisioning. It can be used to automatically deploy software on devices and access equipped sensors and actuators. In their work, the authors decide where data processing is conducted through dynamic operator placement. In our work, we extend this approach by focusing on the execution level, i.e., on the communication between these operators to process data in the IoT.

Giang et al. [9] introduce an approach to develop applications for IoT environments in a distributed manner. The programming model, which they call *Distri-buted Data Flow* (DDF), provides application logic expressed as a directed graph, i.e., a flow. Each device processes a part of this flow, including one or multiple nodes. Processing a node creates an output from zero or multiple inputs, which is the input for the next node, or for multiple nodes, in the flow. This supports the execution of applications in a distributed environment, without further division of the application logic. However, Giang et al. do not suggest a solution how devices can communicate with each other without the usage of a central execution system, yet they recognize this as beneficial in many cases.

With SitRS XT, Franco da Silva et al. [7] propose a solution to recognize situations in IoT environments in near real-time. Situation templates specify how data from different sensors has to be aggregated to recognize a specific situation. Situation templates are transformed into executable CEP queries and deployed into an execution engine. A continuous stream of sensor data into the execution engine enables an immediate recognition of changing situations. However, automatically reacting to a situation requires transmitting data back from the off-premise execution engine to on-premise actuators [12]. With our approach, we want to eliminate the latency that comes with this communication overhead and instead utilize computational power of smart devices to evaluate sensor data and recognize situations.

Negasha et al. [15] describe a lightweight service bus to coordinate distributed IoT services. In contrast to Negasha et al., we do not depend on a central instance (i.e., the service bus), the orchestration and communication can be conducted fully decentralized. Hence, we enable direct communication of the IoT devices through our messaging engine without requiring service buses or other central components.

In their article *"An Edge-Focused Model for Distributed Streaming Data Applications"*, Bumgardner et al. [4] describe a graph-based modeling language named *Cresco Application Model (CAM)*, to describe distri-buted data stream applications. The created models are similar to the data stream models used in this paper. Furthermore, Bumgardner et al. aim for an optimal distribution of data stream operators on devices of the environment. To realize this, a greedy algorithm is developed. However, Bumgardner et al. do not focus on the execution of data processing in the IoT environments. This is done by our approach, the distribution algorithm of Bumgardner et al., however, can be used for operator placement in a previous step.

Operator placement is a widely discussed topic in the area of distributed systems (e.g., [2,13,17,21]). Many approaches in this area focus on placement of operators based on available device resources or network capabilities. However, these approaches do not focus on data processing after operator placement as well as redistribution in case of device failures. In this paper, we extend works in the operator placement area with data processing concepts introducing a lightweight messaging engine for device-to-device communication.

Seeger et al. [18,19] present an approach to execute *choreographies* [16] in IoT environments. Interactions between IoT devices are defined via *recipes* [24], which include *ingredients* and *interactions*. IoT devices provide offerings, which describe services offered by it. When a recipe is instantiated, each contained ingredient is mapped to an offering. An interaction describes the data exchange between devices to execute a recipe. Seeger et al. attempt to integrate IoT devices into IoT environments without user interaction, remove single points of failure, and utilize computing resources of network nodes. In contrast to Seeger et al., we do not only focus on modeling data processing in IoT environments but also provide a lightweight messaging engine to implement dynamic real-time IoT applications.

Breitbach et al. [3] propose a multi-level scheduling architecture as well as data and task allocation algorithms to process data in edge computing environments. The authors introduce metrics to decide which devices should perform which tasks and where to store which data. As opposed to Breitbach et al., we mainly focus on data exchange between devices and decoupling of data exchange and data processing. Furthermore, their approach relies on central instances, i.e., brokers, which we tend to avoid.

In summary, in state-of-the-art approaches, there is a lack of concepts that focus on decentralized data processing among distributed IoT devices.

4 Main contribution

In this section, we describe our main contribution, which consists of (i) a lifecycle method, and (ii) a lightweight IoT messaging engine to apply our approach.

4.1 Lifecycle method

Figure 3 depicts our lifecycle method for decentralized data processing in IoT environments. This method consists of five steps that range from the modeling of data processing as data flows to the actual decentralized execution. These steps are explained in detail in the following sections.

4.1.1 Step 1: Data flow modeling

In the first step of our method, a data flow model needs to be created, defining the involved data sources and how data is processed by various operators. In this paper, we define a data flow model as a directed graph, which is based on the pipes

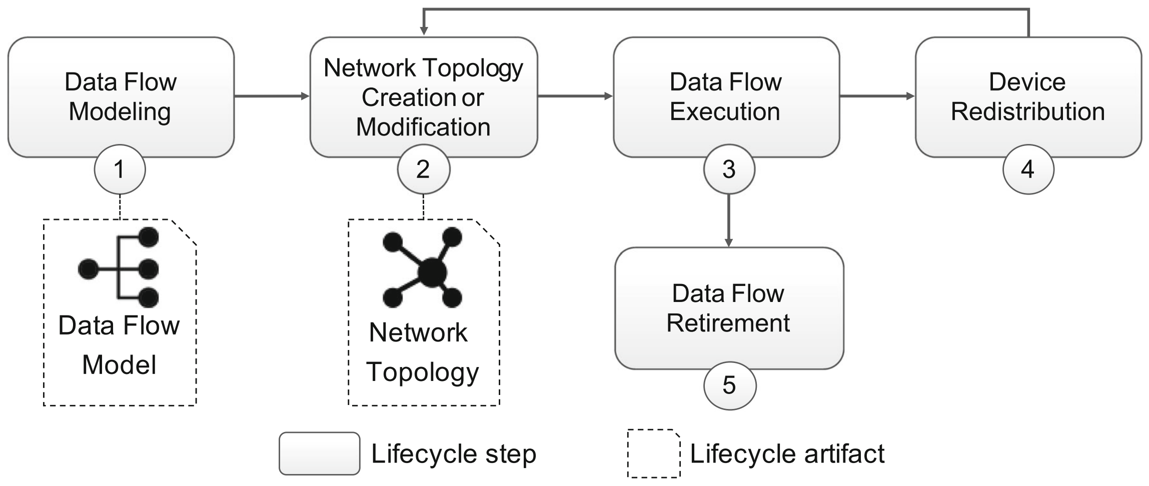

Fig. 3 Life cycle method to enable decentralized data processing

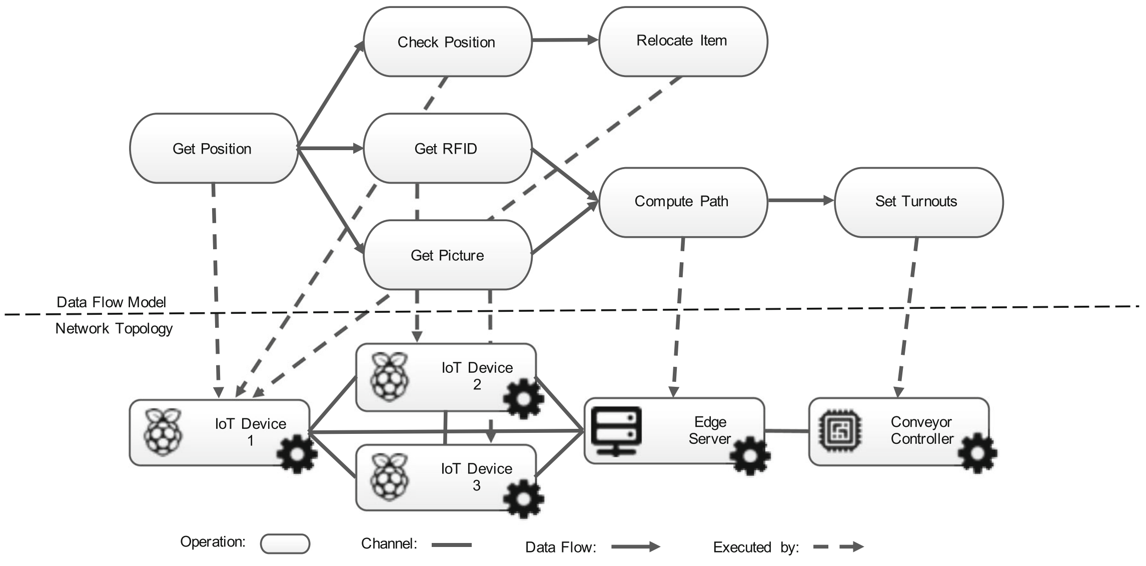

Fig. 4 Mapping of data flow operators to devices in the IoT environment

and filters design pattern [14]. In the pipes and filters pattern, models consist of nodes, the *filters*, and edges, the *pipes*. These pipes and filters form a directed graph. The important aspect of this pattern is that the filters are interchangeable and can be connected with each other arbitrarily through the pipes. To enable this, uniform interfaces and a uniform data transfer format are essential.

In our approach, each filter in the model represents an operation that manipulates data (e.g., filter, aggregation, analysis). These operations can, for example, be performed by an IoT device or a service in an edge cloud. Filters without any incoming pipes are *data sources*, filters without outgoing pipes are *data sinks*. All other filters in the model are describing data operations. In state of the art approaches, data flow models are defined as follows:

Definition 1 A data flow model is a directed graph $DF = (O, E)$ with a set of operations $O = \{o_1, \ldots, o_n\}$ and a set of directed edges $E \subseteq O \times O$. The successor function $suc\colon O \rightarrow O$ is defined as $suc(a) = \{b \in O | (a, b) \in E\}$, denoting the set of operations that follow the operation a according to the data flow model.

On the top of Fig. 4, an example of a data flow model is depicted, which describes data processing in our motivating scenario (Fig. 2). The data flow model starts by retrieving the position of an item on the conveyor belt. After that, it is checked whether the position of the item is correct or not. If the position is considered as incorrect, the item is relocated to a correct position. In parallel, the RFID tag of the item is scanned and a picture of the item is taken by a camera. This data is used to compute the path of the item to the correct machine. Finally, the turnouts are set to lead the item to the correct path.

Each operation in the data flow model must be attached with a unique identifier, the *operation id*, and an *operation name*. The operation name associates the operation with an *operation definition*, which itself is associated with a set of software *artifacts*. To execute an operation on a device, the artifacts, associated with the operation definition, first need to be deployed on it. Then, one of the artifacts, specified in the operation definition, can be executed.

In our previous work [10,11], we already presented detailed approaches to define and create such data flow models, including the open-source graphical modeling tool FlexMash [10]. In this paper, we build on these approaches and extend them to enable decentralized data processing in the IoT.

4.1.2 Step 2: Network topology definition

In the next step, a network topology needs to be created if it is not yet existing. 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. We formally define a network topology as follows:

Definition 2 A network topology is an undirected, connected and weighted graph $NT = (D, C, c, ops)$ with a set of devices $D = \{d_1, \ldots, d_m\}$, a set of undirected edges, i.e., channels $C \subseteq \{\{d_i, d_j\} \mid d_i, d_j \in D\}$, the cost function $c\colon C \rightarrow \mathbb{R}_{\geq 0}$, stating the cost of delivering data over a channel $\{d_i, d_j\}$ from device d_i to device d_j or vice versa, and the function $ops\colon N \rightarrow \mathcal{P}(O)$, indicating that a device d is capable of performing the operations $ops(d)$.

A data flow model is only executable if $\forall o \in O \exists d \in D\colon o \in ops(d)$, i.e., for every operation in the data flow model, there is a device in the network topology which is able to perform it.

On the bottom of Fig. 4, the network topology for the scenario (Fig. 2) is depicted. In this example, IoT Device 1, IoT Device 2, IoT Device 3 and the Edge Server are all able to intercommunicate. The only device that is able to communicate with the Conveyor Controller via a channel is the Edge Server.

4.1.3 Step 3: Data flow execution

The next step provides the main contribution of our method, the execution of the data flow model based on the created network topology. When a device receives incoming data, it performs the associated operation and sends the resulting data to the next device. Data is transmitted between devices in the form of a *message*, including a header and a body, similar to SOAP messages [6]. The message header contains a *next operation id*, a *data flow id* and an optional *correlation id*. The *data flow id* informs devices receiving a message to which data flow model this message belongs. This is important since multiple data flow models can be processed by the same IoT environment.

The *next operation id* indicates which operation in the data flow model must be performed to process the data contained in the body of the message. The *correlation id* is used to identify multiple messages that need to be correlated, e.g., if multiple paths in the data flow model need to be aggregated.

The execution is initiated starting from the data sources. In the scenario (cf. Fig. 4), IoT Device 1 performs the operation *Get Position* by retrieving the position from the Distance Sensor. Then, IoT Device 1 performs *Check Position* and, if the position is considered to be false, also the operation *Relocate Item* by operating the Item Relocater. Then, IoT Device 1 seeks in the network topology for devices that are able to perform the operations *Get RFID* and *Get Picture*, which in this case are only IoT Device 2 for *Get RFID* and IoT Device 3 for *Get Picture*. IoT Device 1 creates two messages, one with *next operation id 2* and one with *next operation id 3*. Both messages get the same *correlation id*, which can be random but must be unique. The body of both messages contains the position data it retrieved from the Distance Sensor. It sends the first message to IoT Device 2 and the second to IoT Device 3. Listing 1 shows an example for the message IoT Device 1 transmits to IoT Device 2.

Listing 1 Example message in JSON representation

```
{
‘‘flow_id’’: ‘‘conveyor_flow’’,
‘‘oiid’’: ‘‘2’’,
‘‘correlation_id’’: ‘‘47’’,
‘‘payload’’: ‘‘y_pos:34’’
}
```

The other devices continue the execution of the data flow model accordingly. When executing the operation *Compute Path*, the Edge Server can correlate the data it receives from IoT Device 2, after executing *Get RFID*, and IoT Device 3, after executing *Get Picture*, by the correlation ids in the messages headers. For example, if the Edge Server first receives a message from IoT Device 2 with the *correlation id 47*, it waits for another message with the same correlation id. When both messages have been received, the Edge Server processes them by executing the operation *Compute Path*. After executing *Compute Path*, the Edge Server sends the resulting path as payload of a new message to the Conveyor Controller, which executes the operation *Set Turnouts* to lead the item to the correct machine.

To handle the consumption, processing and forwarding of messages, we propose a messaging engine which is described in Sect. 4.2.

4.1.4 Step 4: Device redistribution

When a new device is added to or removed from the network topology, other devices in the environment need to be notified so they either consider them for message forwarding or not.

To propagate the information about newly added devices, three different approaches can be applied:

(i) publishing the information of the new device through a central component,
(ii) publishing *hello messages* from the newly added device to other devices, or
(iii) sending a multicast message from the newly added device to all other devices.

Since we aim at creating a solution with only few central components, and thus, less bottlenecks or points of failures, we do not consider the first approach.

For the second approach, each device requires a special interface for hello messages. Hello messages can be forwarded by devices to make sure that each device receives it, similar to the registration in peer-to-peer networks [22]. To avoid cyclic forwarding of hello messages, each hello message needs to carry a list of past recipients of the message. Then, each device can forward the hello message to each device in the topology, except for the ones in the recipient list of the hello message.

The third approach 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.

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

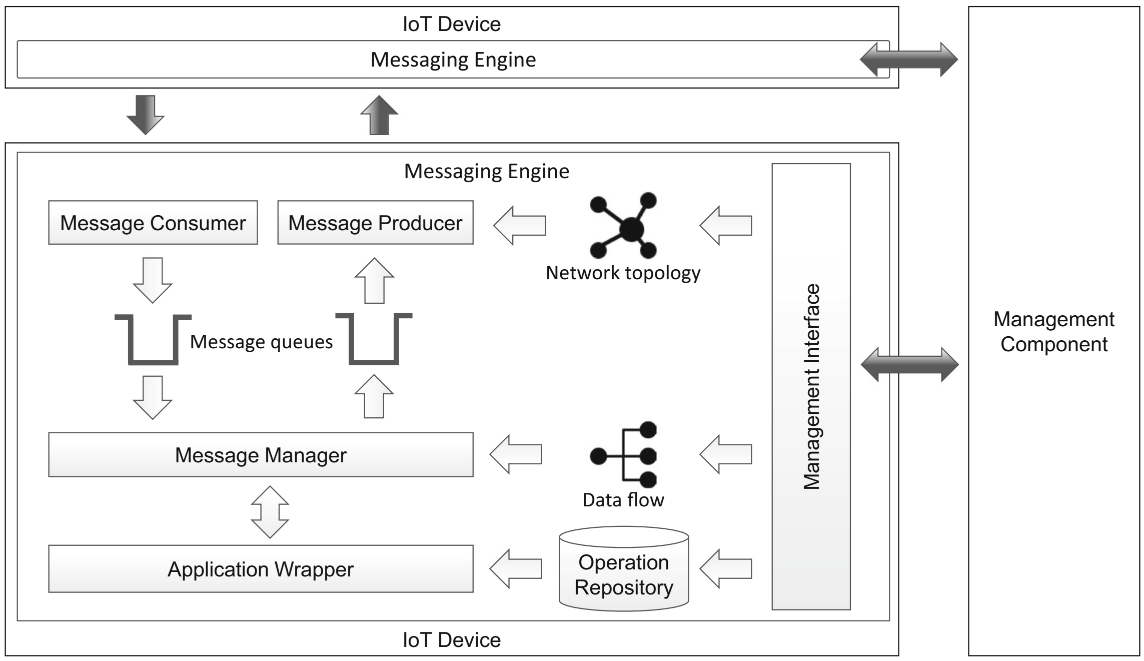

Fig. 5 System architecture of the messaging engine

that the removed device could perform. Nevertheless, this slows down the performance of the execution, since devices try to deliver messages to the removed device.

If the device is removed from the network topology on purpose, all three approaches to propagate information about an added device can also be applied to propagate information about a removed device. If the device 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.

4.1.5 Step 5: Data flow retirement

A data flow can be retired by 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 id*. 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, to prevent that devices process messages of the retired data flow. Stopping each device immediately from processing any message that belongs to a specific data flow, like an emergency halt, could also be done via an interface of each messaging engine of the devices.

4.2 Architecture

In this section, we introduce the architecture of our lightweight messaging engine, which is deployed onto each IoT device to process data as described in the lifecycle method. Figure 5 depicts the architecture of the messaging engine.

Data flow models being processed are stored in the messaging engine, associated with their data flow ids. Changes in the data flow models can be made through a management API of the messaging engine. For example, if messages should be forwarded to a newly inserted operation. Similarly, the network topology is stored in the messaging engine and, if necessary, can be adapted through a management API if changes occur in the network, e.g., when retiring a device. To associate an operation name with an application on the device's file system, the *operation repository* references an operation name to a specific file name (e.g., a script that executes the operation).

All repositories can be accessed via the *management interface*. Messages are received via the *message consumer* and then immediately put into the *incoming message queue*. When the device is able to process a new message, the *message manager* takes the next message from the incoming message queue. In this way, the threat of resource deficit can be reduced, processing each message after another.

The message manager then looks for the data flow id in the message's header and retrieves the stored data flow model. Then, the message manager looks for the operation to be executed using the next operation id in the message's header. The message manager forwards the name of the operation to be executed and the message's body to the *application wrapper*.

The application wrapper calls the application that is associated with the operation name via the operation repository, using the message body, i.e., the data to be processed, as input. As soon as the operation finishes processing, the application wrapper returns the output of the application to the message manager.

In the next step, the message manager assembles a new message with the output of the application as the message body, the same data flow id and correlation ids as the incoming message and the operation id of the operation that is to be performed next in the data flow model. It then puts the new message in the outgoing message queue along with the name of the operation that must be performed next.

The *message producer* collects the next message from the outgoing message queue and looks for all nodes that are able to perform the next operation. We propose two possibilities to determine a recipient device for a message suitable for multiple possible target devices: (i) the message is sent to each possible device successively, until a device accepts the message or (ii) the message producer retrieves information of queued messages and hardware utilization from each possible device and then only sends the message to the device with the most available resources. The first solution leads to less communication overhead but might lead to every outgoing message being sent to the same device. The second solution makes it possible to scale the execution horizontally by adding new devices.

A possible protocol to exchange messages, models and information about devices is the *Constrained Application Protocol* (CoAP) [20]. It is similar to HTTP but more lightweight and based on UDP, which makes it more suitable for constrained IoT devices.

Note that the IoT devices demonstrating our messaging engine (e.g., Raspberry Pis) are quite powerful in comparison to minimal resource devices, such as micro controller boards. In the future, we will consider a compressed variant of our messaging engine that can be run onto devices with minimal resources.

5 Prototype and evaluation

In our prototypical implementation of the messaging engine, implemented in Python, we used the library CoAPthon [23], a library for CoAP interfaces. Figure 6 shows the architecture of our implementation.

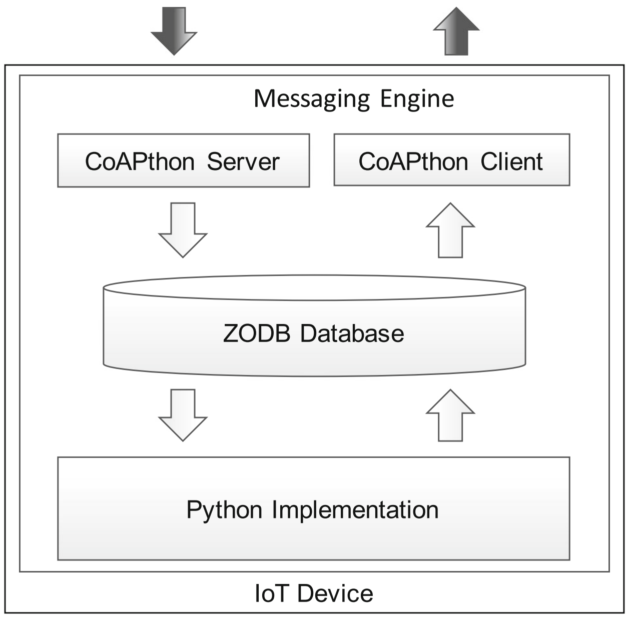

Fig. 6 System Architecture of the prototypical implementation

The management interface, the message consumer and the message producer are implemented as a CoAP server and client using CoAPthon. The repositories and both message queues are implemented using a light-weight but thread-safe NoSQL database called ZODB.[1] All data is transmitted and stored in a JSON format, e.g., as depicted in Listing 1. The message manager and the application wrapper run both in one single thread, which restricts the messaging engine to process only one message at once. CoAPthon creates a new thread for each request, thus, the messaging engine can receive and store multiple messages at once. The message producer runs in its own thread, so the messaging engine can receive multiple messages, process messages, and forward them at the same time.

To evaluate the concept and show the lightweightness of our messaging engine, we measured its overhead and compared it to the traditional approach, invoking IoT operations directly, without such a messaging engine. Table 1 shows the measured values in seconds.

The Messaging Engine and the HTTP script both execute the application echo after receiving a message or an HTTP request and respond with a new message or an HTTP response. Both were running on a virtual machine with one VCPU core, 2 gigabytes of memory and the operating system Ubuntu 16.04.

The difference of the measured times, which is on average 0.16774489 s, needs to be compared to the overhead of

[1] http://www.zodb.org/.

Table 1 Measured times spent to process a message in seconds

Measurement	Messaging engine	HTTP script	
1	0.0336483	0.0074920	
2	0.1962741	0.0078961	
3	0.1845627	0.0054223	
4	0.1952052	0.0060718	
5	0.1925196	0.0077109	
6	0.1845059	0.0048375	
7	0.1846227	0.0074453	
8	0.1884131	0.0066044	
9	0.1882581	0.0077838	
10	0.1963497	0.0056464	
			Difference
Mean	0.17443594	0.00669105	0.16774489
Median	0.1883356	0.00702485	0.18131075

time that is necessary to send the data to a central execution engine and the time the execution engine needs to process the data and to send it to the next device. Since this detour is rather time consuming, the computational overhead of the messaging engine can be disregarded. Thus, we obtained a reduction in latency in comparison to a central execution engine, depending on its location. We increased the flexibility, since devices choose the best successor for an outgoing message. If a device fails, other devices will automatically choose another device that is able to perform the desired operation. If a new device is added, devices will automatically integrate it into the data flow execution. Also, the resources of the devices are utilized, on the one hand to process data, on the other hand to orchestrate the data flow execution itself.

The running messaging engine requires about 23 megabytes of memory and the python files need 116 kilobytes on the file system, thus, leading to a very lightweight implementation. This makes the messaging engine suitable for constrained devices.

6 Summary and future work

In this paper, we introduce an approach for decentralized data processing in the Internet of Things. The main contributions of our paper are a lifecycle method and a lightweight messaging engine, which can be deployed onto various, heterogeneous IoT devices. These messaging engines enable decentralized data processing in the IoT, while data is transferred directly between the involved devices without the need for a central control instance. Messages are distributed dynamically, meaning that the path they take within the IoT environment is not predetermined but depends on available resources of IoT devices or current network capabilities. Through this approach, bottlenecks can be avoided. Furthermore, network traffic and latency can be highly reduced in contrast to shipping all data to a central instance. For management purposes, a central control instance is still necessary, however, only if the network topology or the data flow model change. Especially in time-critical scenarios, such as Smart Manufacturing, applying our approach can lead to a more timely and robust data processing.

We implemented a prototype for the messaging engine and evaluated it by measuring the overhead it produces in contrast to a simple HTTP interface. As our measurements show, the messaging engine is very lightweight and can be run onto resource-limited IoT devices, such as Raspberry Pis or Arduino boards. Our approach uses established standards (e.g., CoAP, JSON) or de-facto standards to ensure its applicability and to make it more future proof.

For future work, we aim at increasing the robustness of our prototype while applying the introduced concepts to the Smart Factory domain. Furthermore, we enable automated deployment of the messaging engine, for example, using the TOSCA [1] standard.

Acknowledgements This work is partially funded by the German Ministry for Economy and Energy in the scope of the project IC4F (01MA17008).

References

1. Binz T, Breitenbücher U, Kopp O, Leymann F (2014) Tosca: portable automated deployment and management of cloud applications. In: Advanced Web Services. Springer, New York, pp 527–549
2. Bonfils BJ, Bonnet P (2004) Adaptive and decentralized operator placement for in-network query processing. Telecommun Syst 26(2):389–409. https://doi.org/10.1023/B:TELS.0000029048.24942.65
3. Breitbach M, Schäfer D, Edinger J, Becker C (2019) Context-aware data and task placement in edge computing environments. In: 2019 IEEE international conference on pervasive computing and communications (PerCom). IEEE, pp 271–282
4. Bumgardner C, Hickey C, Marek V (2018) An edge-focused model for distributed streaming data applications. In: PerFoT'18 - international workshop on pervasive flow of things, pp 776–781
5. Cook DJ, Das SK (2007) How smart are our environments? An updated look at the state of the art. Pervasive Mobile Comput 3(2):53–73
6. Curbera F, Duftler M, Khalaf R, Nagy W, Mukhi N, Weerawarana S (2002) Unraveling the Web Services Web: an introduction to soap, wsdl, and uddi. IEEE Internet Comput 6(2):86–93
7. da Silva ACF, Hirmer P, Wieland M, Mitschang B (2016) Sitrs xt-towards near real time situation recognition. J Inf Data Manag 7(1):4
8. da Silva ACF, Hirmer P, Peres RK, Mitschang B (2018) An approach for cep query shipping to support distributed iot environments. In: 2018 IEEE international conference on pervasive computing and communications workshops (PerCom Workshops). IEEE, pp 247–252
9. Giang N.K, Blackstock M, Lea R, Leung V.C. (2015) Developing iot applications in the fog: a distributed dataflow approach. In: 2015

5th international conference on the Internet of Things (IOT). IEEE, pp 155–162
10. Hirmer P, Behringer M (2017) FlexMash 2.0 - flexible modeling and execution of data mashups, rapid mashup development tools - second international rapid mashup challenge, RMC 2016, Lugano, Switzerland, June 6, 2016, Revised Selected Papers, vol 696, pp 10–29. Springer, New York. https://doi.org/10.1007/978-3-319-53174-8. http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INBOOK-2017-01&engl=0
11. Hirmer P, Mitschang B (2016) FlexMash - flexible data mashups based on pattern-based model transformation, rapid mashup development tools, vol 591, pp 12–30. Springer, New York. https://doi.org/10.1007/978-3-319-28727-0_2. http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INBOOK-2016-01&engl=0
12. Hirmer P, Wieland M, Schwarz H, Mitschang B, Breitenbücher U, Sáez S.G, Leymann F (2016) Situation recognition and handling based on executing situation templates and situation-aware workflows. Computing. https://doi.org/10.1007/s00607-016-0522-9
13. Luthra M, Koldehofe B, Weisenburger P, Salvaneschi G, Arif R (2018) Tcep: adapting to dynamic user environments by enabling transitions between operator placement mechanisms. In: Proceedings of the 12th ACM international conference on distributed and event-based systems, DEBS '18. ACM, New York, pp 136–147. https://doi.org/10.1145/3210284.3210292
14. Meunier R (1995) The pipes and filters architecture. In: Pattern languages of program design. ACM Press/Addison-Wesley Publishing Co, New York, pp 427–440
15. Negash B, Rahmani AM, Westerlund T, Liljeberg P, Tenhunen H (2015) Lisa: Lightweight Internet of Things service bus architecture. Procedia Comput Sci 52:436–443
16. Peltz C (2003) Web services orchestration and choreography. Computer 10:46–52
17. Pietzuch P, Ledlie J, Shneidman J, Roussopoulos M, Welsh M, Seltzer M (2006) Network-aware operator placement for stream-processing systems. In: 22nd international conference on data engineering (ICDE'06), pp 49–49. https://doi.org/10.1109/ICDE.2006.105
18. Seeger J, Deshmukh RA, Bröring A (2018) Dynamic iot choreographies-managing discovery, distribution, failure and reconfiguration. arXiv preprint arXiv:1803.03190
19. Seeger J, Deshmukh RA, Broring A (2018) Running distributed and dynamic iot choreographies. In: 2018 Global Internet of Things summit (GIoTS). IEEE, pp 1–6
20. Shelby Z, Hartke K, Bormann C (2014) The constrained application protocol (coap). Tech. rep
21. Srivastava U, Munagala K, Widom J (2005) Operator placement for in-network stream query processing. In: Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on principles of database systems, PODS '05, pp 250–258. ACM, New York. https://doi.org/10.1145/1065167.1065199
22. Steinmetz R, Wehrle K (2005) Peer-to-peer systems and applications, vol 3485. Springer, New York
23. Tanganelli G, Vallati C, Mingozzi E (2015) Coapthon: easy development of coap-based iot applications with python. In: 2015 IEEE 2nd World Forum on Internet of Things (WF-IoT). IEEE, pp 63–68
24. Thuluva AS, Bröring A, Medagoda GP, Don H, Anicic D, Seeger J (2017) Recipes for iot applications. In: Proceedings of the seventh international conference on the Internet of Things. ACM, p 10
25. Vermesan O, Friess P (2013) Internet of Things: converging technologies for smart environments and integrated ecosystems. River Publishers, Aalborg

Daniel Del Gaudio is a Ph.D. student at the Institute of Parallel and Distributed Systems of the University of Stuttgart since 2018. His research interest lies on data processing and process modeling. Furthermore, he works on database and middleware systems in the domain of Internet of Things.

Pascal Hirmer is a post-doctoral researcher at the Universität Stuttgart in the department of Applications of Parallel and Distributed Systems (IPVS/AS). His research interests are Internet of Things, context-awareness, service provisioning and composition, and cloud computing. His work is published in established journals, book chapters, and conference proceedings.