A Life Cycle Method for Device Management in Dynamic IoT Environments

Daniel Del Gaudio, Maximilian Reichel and Pascal Hirmer
Institute for Parallel and Distributed Systems, University of Stuttgart, Universitätsstraße 38, Stuttgart, Germany

Keywords: Internet of Things, Discovery, Device Integration, Decentralization.

Abstract: In the Internet of Things, interconnected devices communicate with each other through standardized internet protocols to reach common goals. By doing so, they enable building complex, self-organizing applications, such as Smart Cities, or Smart Factories. Especially in large IoT environments, newly appearing devices as well as leaving or failing IoT devices are a great challenge. New devices need to be integrated into the application whereas failing devices need to be dealt with. In a Smart City, newly appearing actors, for example, smart phones or connected cars, appear and disappear all the time. Dealing with this dynamic is a great issue, especially when done automatically. Consequently, in this paper, we introduce A Life Cycle Method for Device Management in Dynamic IoT Environments. This method enables integrating newly appearing IoT devices into IoT applications and, furthermore, offers means to cope with failing devices. Our approach is evaluated through a system architecture and a corresponding prototypical implementation.

1 INTRODUCTION

The *Internet of Things* (IoT) is an evolving paradigm, in which interconnected devices communicate with each other through standardized internet protocols to reach common goals (Vermesan and Friess, 2013). Usually, such IoT devices are attached with sensors and actuators to monitor the environment or influence it (Granell et al., 2020). Famous examples for IoT applications are Smart Homes, Smart Factories, or Smart Cities (Harper, 2006; Lucke et al., 2008; Su et al., 2011).

IoT environments are very dynamic, which means that devices enter and leave these environments regularly. This, however, leads to the great challenge of adapting to the new infrastructure. More precisely, failing devices could lead to errors in the IoT applications or, for example, to monitoring inaccuracy due to a decreased coverage of the environment with sensors. In contrast, newly appearing devices can increase the benefit of the IoT application, for example, by new sensors and actuators, computing power or better coverage of the environment. In a Smart City, e.g., newly entering connected cars need to be considered in traffic control, whereas leaving cars can be ignored.

The goal of the IoT is managing the environments possibly autonomous with as little human interaction as possible. When a new device enters the environment or a device leaves it, it does not make sense to involve a human actor, such as a technician to adjust the application accordingly. Especially in large scenarios, for example, in Smart Cities, the overhead would be tremendous.

Hence, IoT applications should be able to involve newly appearing devices or to cope with leaving or failing devices automatically without any human interaction. To solve this issue, in this paper, we introduce A Life Cycle Method for Device Management in Dynamic IoT Environments. Our approach includes (i) a meta model to describe the behavior of an IoT environment, (ii) a meta model to describe the structure of an IoT environment, (iii) the lifecycle method to integrate new devices into existing environments regarding their capabilities and the environment's requirements, and (iv) a system architecture to implement the method. Our approach is evaluated through a prototypical implementation of this architecture.

In our approach, the goal is to keep everything as decentralized as possible. More precisely, the number of central components should be kept as low as possible since the vision of the IoT focusses on direct machine-to-machine communication. Consequently, only a single central components is required in our approach, which copes with integrating new devices into IoT environments. The devices, however, still operate in a completely distributed and decentralized

Gaudio, D., Reichel, M. and Hirmer, P.
A Life Cycle Method for Device Management in Dynamic IoT Environments.
DOI: 10.5220/0009340900460056
In *Proceedings of the 5th International Conference on Internet of Things, Big Data and Security (IoTBDS 2020)*, pages 46-56
ISBN: 978-989-758-426-8
Copyright © 2020 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

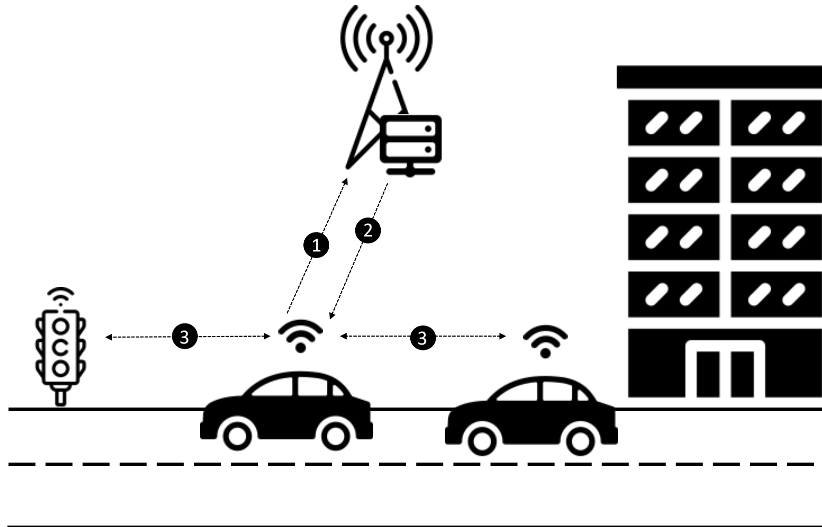

Figure 1: Motivational Scenario.

manner. Note that integrating a new device into an environment is difficult to do decentralized, since one must have an overview of the whole environment to delegate an appropriate task. Hence, we decided to handle this centrally.

The remainder of this paper is structured as follows: Section 2 describes a scenario we use to motivate our contribution. Section 3 introduces related work that copes with the integration of IoT devices in running IoT environments. Next, in Section 4 the meta models are described that form the foundation for our paper. Section 5 then describes our main contribution: a lifecycle method for device management in dynamic IoT environments. Section 6 then evaluates the approach and Section 7 concludes the paper and gives an outlook to future work.

2 MOTIVATION

In this chapter, we introduce a motivating scenario, which serves as means to explain our approach and, furthermore, to serve as a basis for evaluation purposes. The domain of this motivating scenario is Smart Cities. In this domain, a large number of devices exist, such as Smart Cars or Smart Transportation in general, Wearables worn by pedestrians, or stationary edge servers. These devices are highly dynamic and heterogeneous. More precisely, a large amount of devices enter and leave the environment constantly. Manual registration of these devices is impossible due to the large amount.

The devices in our scenario intercommunicate dependant on geographical proximity, user preferences, or use cases. Furthermore, not only one IoT application is employed in Smart Cities but many different ones, partly using the same devices (e.g., Smart Parking, Smart Transportation, Smart Grids). In this scenario, we will specifically focus on automated traffic management of Smart Cars.

The goal in such environments is making it possible for devices to interact with each other in a highly dynamic manner. For example, when a smart car enters a city with a specific service for traffic management, this service needs information about the position of the car in order to include it. However, usually the car owner does not want to share his general position for privacy reasons. If the car had the appropriate software and logic installed, it could preprocess its position data to blur the position and only share information that is required for traffic management. Furthermore, the traffic management system gets relieved since data arrives already preprocessed. The great challenge in this scenario is integrating newly entering cars into the traffic management application so that the car is registered, the required software is deployed, and the car starts sending the appropriate data. For obvious reasons, the whole process needs to be done fully automatically.

With the contribution of this paper, we aim at solving the following challenges of this scenario: (i) discovery and registration of IoT devices, in this scenario Smart Cars, (ii) automated deployment of software and application logic on heterogeneous IoT devices, (iii) data processing on the IoT devices to increase privacy and scalability, and (iv) seamless device communication according to the specific application.

The introduced scenario is depicted in Figure 1.

In this figure, first, the smart car registers itself at the nearest Road Side Unit, which is depicted on the middle top. In a second step, the Road Side Unit deploys the appropriate application logic and software components on the car. In this scenario and in the scope of this paper, we assume that the IoT devices allow such communication and are willing to collaborate with the involved IoT applications.

Finally, once the appropriate application logic is deployed, the car is able to communicate with the other cars and Road Side Units in the Smart City according to the specific regulations, which is the third step in Figure 1.

3 RELATED WORK

In previous work (Del Gaudio and Hirmer, 2019), we introduced a lightweight messaging engine to enable device-to-device communication in the IoT. This messaging engine handles information exchange between devices in a dynamic, scalable, and robust manner. Data processing is defined by a processing model. Data is exchanged in the form of messages containing a header and a payload with the actual data. In this previous work, the messaging engine still lacks mechanisms to cope with the dynamic we aim for in this paper. Hence, in this paper, we plan to enhance this messaging engine so it can cope with newly added or failing devices and enable a more dynamic device-to-device communication.

Seeger et al. (Seeger et al., 2019) propose an approach to process data in IoT environments in a distributed manner, which is similar to ours. They model data processing based on choreographies (Peltz, 2003). They also introduce a concept for failure detection, where devices monitor themselves to detect failures. In contrast to our work, Seeger et al. specifically focus on device failures and their recovery in the scope of their IoT choreographies. Newly appearing devices, however, are not discovered and considered automatically. Moreover, we aim at creating a more generic approach, which is not only valid specifically for choreographies but for all kinds of models.

Franco da Silva et al. and Hirmer et al. (Hirmer et al., 2016; Franco da Silva et al., 2020; Franco da Silva et al., 2019) propose the new IoT platform MBP as well as means to map operators onto distributed IoT devices and execute them. However, they do not provide any mechanisms for coping with newly appearing devices or device failures.

Kodeswaran et al. (Kodeswaran et al., 2016) introduce Idea – a system for efficient failure management in smart IoT environments. The core of the Idea approach is a central system, which provides means for monitoring failures of sensors and, if a failure occurs, a scheduling component for maintenance. The work focuses mostly on Smart Home applications. In contrast to Kodeswaran et al., we not only cope with failing devices but also integrate new devices into the IoT applications. Furthermore, our approach is more decentralized, whereas the Idea framework describes a central system.

Similar to Kodeswaran et al., Kapitanova et al. (Kapitanova et al., 2012) introduce a system to handle non-stopping failures in Smart Home environments. Failure detection is done by machine learning algorithms. In contrast, we aim at a more traditional, lightweight monitoring approach and, furthermore, we also provide means to handle newly added devices, not only the ones that fail.

Very similar to our approach is the research area of discovery in the IoT. Datta et al. (Datta et al., 2015) categorize related work in the area of discovery into the following areas: distributed and peer-to-peer discovery services, centralized architectures, CoAP-based service discovery, semantic-based discovery, search engines for resource directory, and utilization of ONS and DNS.

Fredj et al. (Fredj et al., 2014) propose a semantic-based service discovery using ontologies. A semantic model that can be used to achieve discovery in such a manner is IoT Lite (Bermudez-Edo et al., 2016). We use a similar, however, more lightweight approach that introduces a meta model that represents the IoT devices of an application that is used for discovery purposes. By doing so, discovery can be enabled more efficiently in contrast to working with heavyweight ontology models.

CoAP-based discovery mechanisms can make use of the resource discovery (*/.well-known/core*) interface of a CoAP server, through which provided services of the server can be retrieved (Shelby et al., 2014; Shelby et al., 2013). Cirani et al. (Cirani et al., 2014) propose an architecture for peer-to-peer-based autonomous resource and service discovery in the IoT. The architecture utilizes a central IoT gateway and the CoAP resource discovery interface. In our paper, we decided to focus on a more generic approach for device discovery and integration that is not specifically tailored to CoAP.

In conclusion, our thorough literature review of related work shows that even though there are already many works focusing either on device failures or on device discovery and integration, there is yet no combined approach, which provides the flexibility, dynamics and genericity we aim for in this paper.

4 METAMODELS

For heterogeneous devices being able to intercommunicate, we need standards to describe applications in terms of communication.

In this section, we introduce two meta models, which build the foundation for our lifecycle method: (i) a data processing model, specifying the business logic of an IoT application (Section 4.1), and (ii) a structural description of IoT environments, including devices, sensors, actuators, network, and communication (Section 4.2).

These models are necessary because data processing should be decoupled from the actual devices that execute data processing operations, which leads to a clear separation of concerns. Hence, devices can be added, removed, and exchanged without having to adapt the processing model. Furthermore, an application developer can model an IoT application via the processing model without further knowledge of the structure of the IoT environment in which the application will be executed. Consequently, both models are also fully decoupled and, thus, interchangeable. We describe details of these models in the following.

4.1 Processing Model

Interactions between devices in IoT environments can be described by directed graphs according to the pipes and filters pattern (Meunier, 1995), which we refer to as the *processing model* in this paper. In our experience, and as shown in related works (e.g., (Del Gaudio and Hirmer, 2019; Franco da Silva et al., 2019)), the pipes and filters pattern is appropriate to describe the behavior of an IoT environment.

Nodes in the processing model, i.e., the filters, are referred to as *operations* and are not bound to specific devices. Instead, they are associated with a set of *requirements*, making it possible to dynamically choose the right IoT device for the operator.

We define the processing model as a tupel $DF = (O, E, R, req)$ with the set of operations O, the set of directed edges $E = \{e = (o_i, o_j) | o_i, o_j \in O\}$, stating that operation o_i must be performed right before operation o_j, the set of requirements R, and the function $req : O \rightarrow \mathcal{P}(R)$, which links each operation to a set of requirements. An example for a processing model in JSON representation looks as follows:

```
{
  "flow_id": "TrafficLightFlow",
  "flow": {
    "1": {
      "operation": "SendPosition",
      "requirements": ["PositionSensor"],
      "next_oiid": "2"
    },
    "2": {
      "operation": "SetSignals",
      "requirements": ["TrafficSignals"],
      "next_oiid": "3"
    },
    "3": {
      "operation": "StopCar",
      "requirements":
        ["AccelerationController"],
      "next_oiid": none
    },
  }
}
```

In this example, `flow_id` defines a unique name for each processing model. Each sub-element of `flow` represents an operation. `next_oiid` indicates which operation must be executed after `SendPosition` and so forth. The processing model is depicted in Figure 2. The first operation `SendPosition` is executed by `SmartCar1` by sending its position data to `SmartTrafficLight`. It uses the data to execute `SetSignals` and send the result to `SmartCar2`, in order that it can timely react to the change of signals and stop the car.

In the following sections, this processing models is used to define the business logic (more precisely, the data flow) of the IoT applications.

4.2 Structural Description of IoT Environments

Additionally to the behavior of the IoT applications, we need a model to specify the structure of IoT environments. This involves all included devices and connections between them. For this model, we propose an undirected and weighted graph $E = (D, L, C, cap, w)$ with a set of devices $D = \{d_1, \ldots, d_n\}$, that exist in the environment, a set of links $L = \{l = \{d_i, d_j\} | d_i, d_j \in D\}$, stating that device d_i can communicate with device d_j and vice versa, and a set of capabilities C. The function $cap : D \rightarrow \mathcal{P}(C)$ links each device with a set of capabilities in order to match them with the requirements of the processing model. The function $w :\rightarrow \mathbb{Q}$ associates each link with a specific weight. The weight of a link indicates the costs of delivering a message via the link. The weight of a link can be very dynamic and must be monitored during runtime.

The structural description needs to be changed by devices being added and removed, which is the main contribution of this paper. This model in JSON representation is shown in the following example:

```
{
    "device": {
```

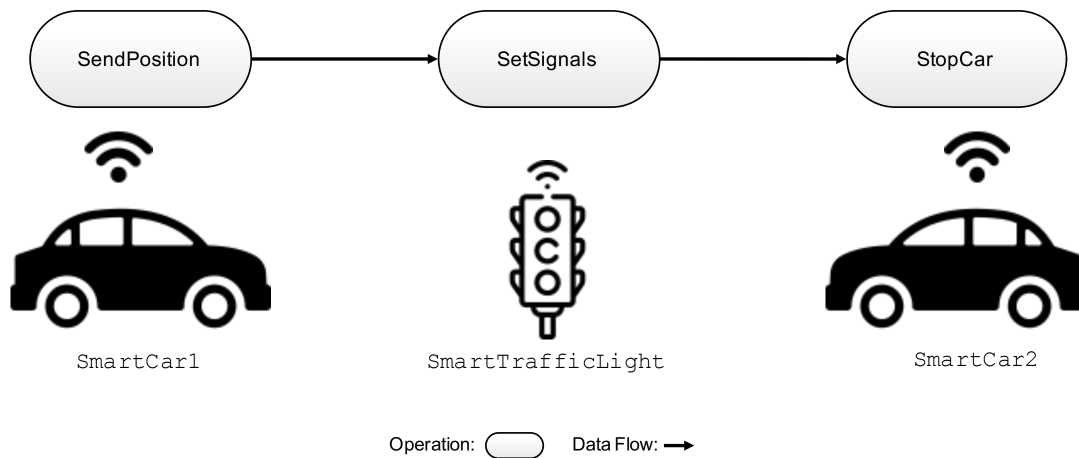

Figure 2: Traffic Light Processing Model.


```
        "name": "SmartCar1",
        "capabilities": [
            "AccelerationController",
            "PositionSensor",
            "HighProcessingPower" ],
        "address": "192.168.56.11",
        "mac": "f0-f4-85-f5-41-e6-53",
        "credentialGroup": "group1"
    }
    "device": {
        "name": "SmartTrafficlight",
        "capabilities": [
            "TrafficSignals",
            "MediumProcessingPower" ],
        "address": "192.168.56.22",
        "mac": "00-53-b7-ff-21-24-34",
        "credentialGroup": "group1"
    }
    "link_1_2": {
        "weight" : 2.3
        "devices": [
            "SmartCar1",
            "SmartTrafficlight"]
    }
}
```


5 LIFE CYCLE METHOD FOR DEVICE MANAGEMENT

We propose a generic life cycle method with six steps to integrate a new IoT device into an existing environment, which is depicted in Figure 3. The lifecycle method describes how one device can be integrated into an existing IoT environment to work with the other devices together. It describes the cycle terms of entering, leaving, and reentering an IoT environment of a single device. Furthermore, we propose a system architecture that implements the life cycle, which is depicted in Figure 4.

The cycle is started by the runtime agent on the device discovering the *runtime management*, representing our system for device integration (step 1). In the second step, the device can register itself at an registration service by sending a *registration message*. The registration service contains a database with the following data: (i) information about each device that is registered, (ii) information about each operation that can potentially be deployed and executed in the environment, and (iii) information about which operation is deployed on which device.

In the next step, the runtime management then determines the proper set of operations to deploy on the new device, as defined by the processing model of the IoT application (step 3). After that, it deploys the associated software and configure it using the designated processing models (step 4).

The next step 5 represents the data processing, i.e., including the new IoT device into the execution of the running processing models of the IoT environment. The management runtime is now only responsible for monitoring the new device and is not involved in the machine-to-machine coordination. Hence, decentralized data processing can still be ensured, which improves the robustness due to no single point of failure. Finally, in the last step 6 of our lifecycle method, the device is removed from the environment, either intentionally or by failure.

The details of these steps are described in the following sections.

5.1 Step 1: Server Discovery

The first step of our lifecycle method is the server discovery. In order for a device to be integrated in an

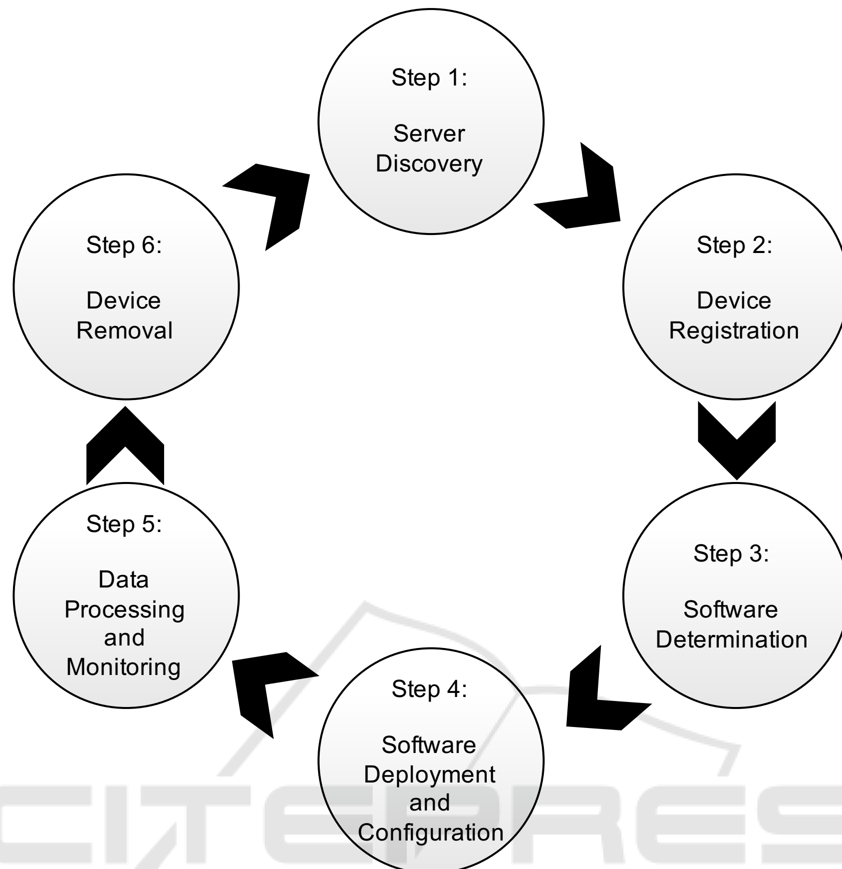

Figure 3: Life Cycle Method.

SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS

IoT application, it must register itself at the registration component, i.e., the registration server.

We evaluated three possibilities how a device can find the registration server in a network: (i) the address of the registration component is preconfigured on the device, (ii) the device sends a broadcast or multicast message into the network hoping that the registration server receives it and sends a response, or (iii) the server discovery is implemented into the network via a DHCP or DNS server.

We do not further consider the first solution, since preconfiguration of devices omits the dynamicity we aim for in this paper. For example, when a device enters a different environment, the address of the registration server might be different than the one that is preconfigured. Furthermore, the address of the registration server could change anytime.

The third solution requires specifically configured DNS or DHCP servers and is, thus, also not further regarded. Consequently, we prefer the second solution, since it can be implemented by our system itself without any critical dependencies. The runtime agent on the devices sends broadcast messages into the network and the registration server responds on receiving the message. If the network does not allow broadcasting, the other solutions can be considered as a backup.

5.2 Step 2: Device Registration

The second step of our lifecycle method comprises the device registration. After a new device discovered the address of the registration server (step 1), it registers itself by sending a *registration message* to the registration server. An example of such a message for our motivating scenario is depicted in the following:

```
{
    "device": {
        "name": "SmartCar2",
        "capabilities": [
            "AccelerationController",
            "PositionSensor",
            "HighProcessingPower"
        ],
        "address": "192.168.56.11",
        "mac": "12-43-b5-ae-fd-44-35",
        "credentialGroup": "group2"
    },
```

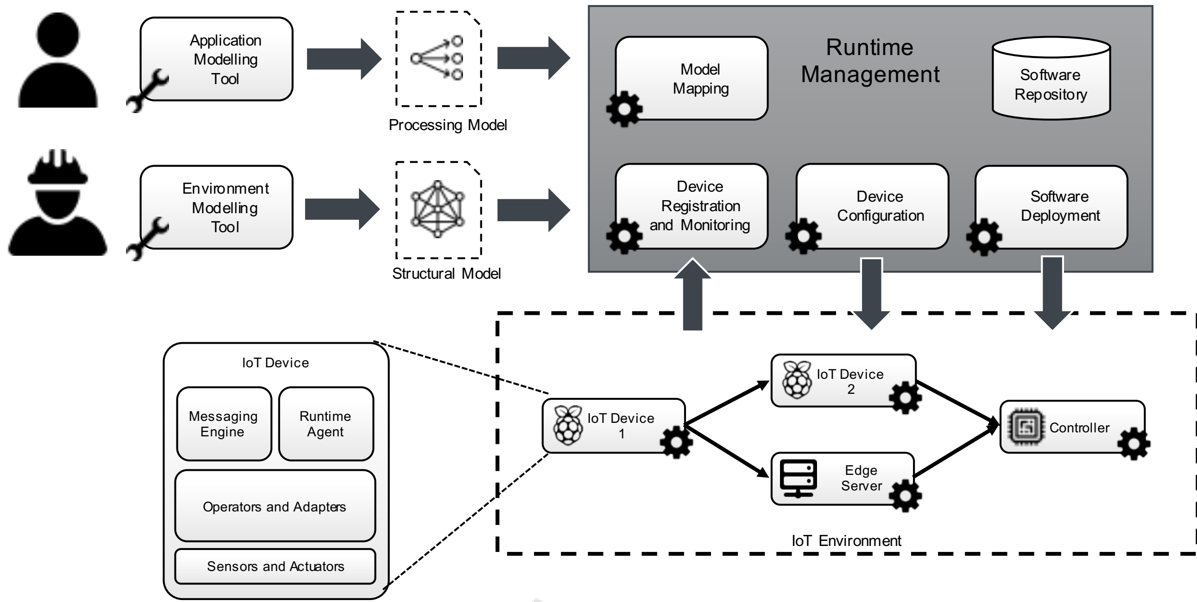

Figure 4: Architecture to Apply Our Lifecycle Method for Device Integration.

```
    "options": {
        "ttl": 3600
    }
}
```

The element `device` defines device-specific metadata. `name` is the self-given name for the device that does not necessarily have to be unique. The subelement `capabilities` is used to choose appropriate operations for the device, `address` defines the IP address, `mac` the MAC address, and `credentialGroup` is used to define the credentials to access the device, for example, via SSH. Furthermore, the element `options` is used for the registration process itself, defining important properties, such as `ttl`, determining the time-to-life for the registration as explained in Section 5.5. The registration server stores the information about the device based on the registration message. After step 2, the device is registered and can be further processed.

5.3 Step 3: Software Determination

As the device is registered at the registration server, the server must now determine a suitable set of operations, implementing specific business logic of an IoT application, and deploy them on the device. This is done in step 3 of our lifecycle method.

We determined three aspects to consider for choosing suitable operations: (i) the capabilities of the device in terms of sensors, actuators, and processing power, (ii) the requirements of available operations in terms of sensors, actuators, and processing power, and (iii) the specific characteristics of the environment the IoT applications are provided in.

In our architecture (cf. Figure 4), the *Model Mapping* component is responsible to map the capabilities of the devices onto the most useful set of applications, regarding all applications that are available.

We define the function $fulfills(c,r)$ for each requirement $r \in R$ and each capability $c \in C$, which evaluates to 1 if capability c fulfills requirement r. Therefore, we define the function $executableBy(o,d)$ for each operation $o \in O$ and each device $d \in D$ as

$$executableBy(o,d) = \begin{cases} 1, & \forall r \in req(o) \exists c \in cap(d): \\ & fulfills(c,r) = 1 \\ 0, & else \end{cases} \quad (1)$$

stating that o is executable by d in terms that for each of requirement d, o offers a fulfilling capability.

For a new device d, we can now seek for the subset of operations

$$O_d = \{o \in O | executableBy(o,d) = 1\}, \quad (2)$$

which contains all operations that are executable by d and, thus, fulfills aspects (i) and (ii).

In terms of aspect (iii), many different characteristics must be considered when choosing which operation is the most ”useful” for an environment. We consider an operation as the most useful for an environment if it is one that is not yet deployed on a device at all, expanding the application’s coverage of the IoT environment.

Furthermore, for choosing suitable operations to be run of the devices, we evaluated two processing patterns that need to be handled separately:

- **Parallelizable Operations:**
  Parallelizable operations can be scaled horizontally by deploying multiple instances of them. Hence, when deploying such operations, it needs to be considered whether scaling is required by the specific IoT application. Furthermore, if all available operations are already deployed, we only look for parallelisable operations.
- **Unique Operations:**
  A unique operation must be unique inside the environment, for example, because all data that is processed by the operation must be consolidated in one instance. Unique operations must be considered separately when choosing the operations to be deployed.

To manage the operations in the environment, we must keep track of operations that are already running, the workload of each operation, and the utilization of resources of the devices. The set of operations that are executed by a given device d in the context of any processing model is determined by $executedBy(d)$. In order to realize this, a sophisticated monitoring of the IoT environment is essential. However, this is not the focus of this paper. Hence, we refer to existing work in IoT monitoring, for example, provided by (Lazarescu, 2013).

5.4 Step 4: Software Deployment and Configuration

In the fourth step, the chosen operations need to be deployed on the new device. For each operation the environment is able to perform, the associated software components must be stored in the *software repository* which is part of the registration server, as shown in our system architecture (cf. Figure 4). These software components can range from simple scripts to sophisticated software applications, whereas only lightweight components can be deployed onto resource-restricted IoT hardware. This decision has already been made in the previous step Software Determination.

The Software Deployment component of our architecture extracts the necessary software components from the software repository, connects to the device, and starts installing the software and all required dependencies. This deployment step can be either implemented manually using, for example, SSH connections, or in a more sophisticated manner. One approach for a more robust software deployment strategy is offered by the OASIS standard TOSCA (OASIS, 2013a; OASIS, 2013b). An open-source implementation of TOSCA is, for example, provided by OpenTOSCA (Binz et al., 2014). By using a standard-based approach, such as TOSCA, the deployment strategy is more future-proof than a non-standard-based approach.

After all applications have been deployed, the messaging engine on the devices (responsible for machine-to-machine communication) must be configured accordingly. The messaging engine needs the operations it can perform, the section of the processing model it participates in, and the node information of devices it must interact with.

Furthermore, messaging engines on other devices that need to interact with the new devices need the information about the newly appeared device, more specifically, which operations it provides and how they can interact with it. A simpler but less scalable solution would be to inform each device in the environment about the newly appeared device (for example, through broadcasting) and, in this process, hand over the necessary information about the new device. However, since IoT devices often tend to be constrained in terms of memory capacity and to minimize network traffic, we do not consider this simple solution.

In our approach, to compute all relevant devices that need to communicate with a newly added device d, the runtime management traverses the processing model and looks for each operation in $executedBy(d)$. Every device that executes a predecessor of an operation in $executedBy(d)$ must be notified about d. Vice versa, d must be notified about each device, that executes a successor operation of each operation in $executedBy(o)$. By doing so, only the devices need to be notified, which are required to interact with the newly added device. This reduces the network traffic that would have been produced by broadcast messages and leads to a more tailored solution.

5.5 Step 5: Data Processing and Monitoring

After deployment of the required software components, the newly added device is ready to process data. It can receive, process, and forward data according to the processing models. Our system is now also able to monitor the devices in the environment. To realize this, each runtime agent sends health messages to the server periodically. The frequency highly depends on the use case, i.e., how critical device failures influence the health of the application.

When the server receives no health message from a device for a specified period of time, it assumes

that the device has left the environment accidentally, for example due to an occurring device failure. In case the device leaves the environment voluntarily, it deregisters itself. This will be discussed in more depth in step 6. The time period is specified in `ttl` in `options` in the registration message as introduced in Section 5.2. This time period can be individual for each device, since IoT environments tend to be highly heterogeneous. In order to prevent unnecessary network traffic, choosing the right time period in which devices send their heartbeat messages has to be considered wisely.

Furthermore, for device health monitoring, important metrics are the weight of links (cf. Section 4) and the resource utilization of devices. Monitoring information can be used to detect bottlenecks and react accordingly, for example, by deploying additional software on devices in order to scale the application.

When the data processing is interrupted because a device is not capable of finding an appropriate receiver for data, it can perform a request to the device registration and monitoring component, which will then respond with the address of another device hosting the desired operation or deploy the operation on one.

5.6 Step 6: Device Removal

There are two ways a device is removed from the environment: on purpose, for example, when a Smart Car leaves the city area, or accidentally, for example, when the network connection is lost or when the device's hardware fails.

When a device leaves the environment on purpose, e.g. when it should be exchanged, it should be able to backup all its data on the registration server. When a new device enters the environment that is able to perform one of the operations the leaving device performed, the registration server can initialize the new device with the backup data. By doing so, it can be ensured that no data is left. Of course, this data needs to be attached with a timespan until it gets stale. Especially streaming data in the IoT get stale very quickly, thus, loses its usefulness after a short amount of time.

When a device leaves the environment accidentally, loss of data can only be prevented by persistently storing all data on the device hoping that it reenters. If this is not the case, however, all data that was stored on the device at the time of the failure is lost. One solution to cope with this issue are periodical backups. However, it needs to be considered that these backups lead to a significant overhead regarding computing and network resources. Hence, this tradeoff needs to be carefully considered for each application separately.

After the last step of our lifecycle method, the device leaves the environment. At any time, the device can re-enter and the whole method is re-initialized. Note that if a device already contains the required software components when re-entering, the method can be significantly accelerated. Thus, it also considers previous iterations.

6 PROTOTYPE AND DISCUSSION

To evaluate our concept, we implemented a prototype of the runtime management and the runtime agent, running on the device itself (cf. Figure 4).

The agent application is implemented in Python to guarantee a certain degree of lightweightness because this agent is usually deployed onto resource-limited IoT hardware. In contrast, the runtime management components, running in a scalable cloud environment, is implemented in Java.

These components communicate via CoAP, a lightweight HTTP-like protocol. In order to implement the CoAP client and server applications, we used the Python implementation CoAPthon (Tanganelli et al., 2015) for the client and the Java implementation Californium (Kovatsch et al., 2014) for the server. Furthermore, we use MongoDB[1] as database to store information about registered devices on the server. Furthermore, all communication payload is serialized in JSON.

To simulate the traffic management scenario described in Section 2, we created an IoT environment with multiple mobile and stationary devices attached to an OpenStack-managed[2] private cloud platform running on an IBM Pureflex computing cluster with 12 compute nodes. As IoT hardware, we used Raspberry Pis[3] attached to different sensors and actuators that are hosting the agent. A virtual machine in the private cloud is hosting the runtime management components. Using such a flexible infrastructure enables easy scaling of the runtime management components and prevents them from becoming a single-point-of-failure.

In order to set up our runtime management component, a few simple steps need to be conducted, thus, making it easy to set up in different scenarios by different stakeholders. First the runtime management needs to be deployed on an application server running, e.g., on a virtual machine in the cloud or even on

[1] https://www.mongodb.com
[2] https://www.openstack.org
[3] https://www.raspberrypi.org

an IoT device. Second, the runtime agent must be installed on each IoT device that is involved in the scenario. Furthermore, each device needs a minor preconfiguration to be able to share its capabilities with the runtime management. For many IoT devices, we recommend automating this installation and configuration step, using for example, TOSCA or other well-known deployment tools, such as Ansible[4].

As mentioned before, all additional software necessary for a specific scenario should be stored in the software repository and gets installed automatically by the runtime management according to the processing model for the scenario and the devices capabilities in step 4 of our lifecycle method.

Our prototype shows that our concept can cope with the challenges listed in Section 2. Devices are automatically registered when they enter the area of our environment and connect to the Wi-Fi (i). Software is automatically deployed on each device according to the processing model (ii). Data is (pre-)processed on the devices by the deployed software and is sent to the other devices according to the processing and structural models ((iii) and (iv)).

Modelling distributed applications with the processing model in Section 4.1 and the environment with the structural model in Section 4.2 decouples application development from executing environments and, thus, creates dynamic IoT environments with interchangeable devices. Data processing can be scaled horizontally by adding more devices to the environment, since parallelizable operations are deployed automatically and load is balanced amongst them.

7 CONCLUSION

In this paper, we present A Life Cycle Method for Device Management in Dynamic IoT Environments. Using this method, newly appearing devices can be seamlessly integrated into IoT applications without the need for manual, time-consuming steps. In addition, we introduce concepts that allow coping with failing devices or voluntarily leaving ones. Our lifecycle method builds on meta models, describing data processing and the IoT infrastructure landscape. Based on these models, newly appearing devices can be found, registered, necessary software can be installed and they can be integrated for data processing in an IoT application. Finally, the device can be retired either voluntarily or when it fails. Even in case of a failure, we can support IoT applications in providing a robust way of data processing so that applications do not fail when single devices do.

We implemented a prototype for our lifecycle method in order to provide a proof-of-concept. In the future, we aim at applying this prototype to more complex scenarios in order to show the strengths of our approach to an even greater extend.

[4]https://www.ansible.com/

ACKNOWLEDGEMENTS

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

REFERENCES

Bermudez-Edo, M., Elsaleh, T., Barnaghi, P., and Taylor, K. (2016). Iot-lite: a lightweight semantic model for the internet of things. In *2016 Intl IEEE Conferences on Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld)*, pages 90–97. IEEE.

Binz, T., Breitenbücher, U., Kopp, O., and Leymann, F. (2014). Tosca: portable automated deployment and management of cloud applications. In *Advanced Web Services*, pages 527–549. Springer.

Cirani, S., Davoli, L., Ferrari, G., Léone, R., Medagliani, P., Picone, M., and Veltri, L. (2014). A scalable and self-configuring architecture for service discovery in the internet of things. *IEEE Internet of Things Journal*, 1(5):508–521.

Datta, S. K., Da Costa, R. P. F., and Bonnet, C. (2015). Resource discovery in internet of things: Current trends and future standardization aspects. In *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, pages 542–547. IEEE.

Del Gaudio, D. and Hirmer, P. (2019). A lightweight messaging engine for decentralized data processing in the internet of things. *SICS Software-Intensive Cyber-Physical Systems*.

Franco da Silva, A. C., Hirmer, P., and Mitschang, B. (2019). Model-based operator placement for data processing in iot environments. In *2019 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 439–443. IEEE.

Franco da Silva, A. C., Hirmer, P., Schneider, J., Ulusal, S., and Tavares Frigo, M. (2020). Mbp: Not just an iot platform. In *Proceedings of the 18th Annual IEEE Intl. Conference on Pervasive Computing and Communications*.

Fredj, S. B., Boussard, M., Kofman, D., and Noirie, L. (2014). Efficient semantic-based iot service discovery mechanism for dynamic environments. In *2014 IEEE 25th Annual International Symposium on Personal,*

Indoor, and Mobile Radio Communication (PIMRC), pages 2088–2092. IEEE.

Granell, C., Kamilaris, A., Kotsev, A., Ostermann, F. O., and Trilles, S. (2020). *Internet of Things*, pages 387–423. Springer Singapore, Singapore.

Harper, R. (2006). *Inside the smart home*. Springer Science & Business Media.

Hirmer, P., Breitenbücher, U., da Silva, A. C. F., Képes, K., Mitschang, B., and Wieland, M. (2016). Automating the Provisioning and Configuration of Devices in the Internet of Things. *Complex Systems Informatics and Modeling Quarterly*, 9:28–43.

Kapitanova, K., Hoque, E., Stankovic, J. A., Whitehouse, K., and Son, S. H. (2012). Being smart about failures: Assessing repairs in smart homes. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, UbiComp '12, pages 51–60, New York, NY, USA. ACM.

Kodeswaran, P. A., Kokku, R., Sen, S., and Srivatsa, M. (2016). Idea: A system for efficient failure management in smart iot environments. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '16, pages 43–56, New York, NY, USA. ACM.

Kovatsch, M., Lanter, M., and Shelby, Z. (2014). Californium: Scalable cloud services for the internet of things with coap. In *2014 International Conference on the Internet of Things (IOT)*, pages 1–6. IEEE.

Lazarescu, M. T. (2013). Design of a wsn platform for long-term environmental monitoring for iot applications. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 3(1):45–54.

Lucke, D., Constantinescu, C., and Westkämper, E. (2008). Smart factory-a step towards the next generation of manufacturing. In *Manufacturing systems and technologies for the new frontier*, pages 115–118. Springer.

Meunier, R. (1995). The pipes and filters architecture. In *Pattern languages of program design*, pages 427–440. ACM Press/Addison-Wesley Publishing Co.

OASIS (2013a). Topology and Orchestration Specification for Cloud Applications.

OASIS (2013b). TOSCA Primer. Online.

Peltz, C. (2003). Web services orchestration and choreography. *Computer*, (10):46–52.

Seeger, J., A. Deshmukh, R., Sarafov, V., and Bröring, A. (2019). Dynamic iot choreographies. *IEEE Pervasive Computing*, 18:19–27.

Shelby, Z., Bormann, C., and Krco, S. (2013). Core resource directory.

Shelby, Z., Hartke, K., and Bormann, C. (2014). The constrained application protocol (coap). Technical report.

Su, K., Li, J., and Fu, H. (2011). Smart city and the applications. In *2011 international conference on electronics, communications and control (ICECC)*, pages 1028–1031. IEEE.

Tanganelli, G., Vallati, C., and 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)*, pages 63–68. IEEE.

Vermesan, O. and Friess, P. (2013). *Internet of things: converging technologies for smart environments and integrated ecosystems*. River publishers.

SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS