Generic Driver Injection for Automated IoT Application Deployments

Karoline Saatkamp, Uwe Breitenbücher, Frank Leymann, and Michael Wurster

Institute of Architecture of Application Systems,
University of Stuttgart, Germany
{saatkamp, breitenbuecher, leymann, wurster}@iaas.uni-stuttgart.de

Karoline Saatkamp, Uwe Breitenbücher, Frank Leymann, and Michael Wurster. 2017. Generic Driver Injection for Automated IoT Application Deployments. In *Proceedings of the 19th International Conference on Information Integration and Web-based Applications & Services, Salzburg, Austria, December 4-6, 2017* (iiWAS'17), 320-329. https://doi.org/10.1145/3151759.3151789

BibTeX
```
@inproceedings{Saatkamp2017_Driver,
  author    = {Saatkamp, Karoline and Breitenb{\"u}cher, Uwe and
               Leymann, Frank and Wurster, Michael},
  title     = {Generic Driver Injection for Automated IoT
               Application Deployments},
  booktitle = {Proceedings of the 19\textsuperscript{th}
               International Conference on Information Integration
               and Web-based Applications \& Services},
  year      = {2017},
  pages     = {320--329},
  publisher = {ACM},
  doi       = {10.1145/3151759.3151789}
}
```

Generic Driver Injection for Automated IoT Application Deployments

Karoline Saatkamp, Uwe Breitenbücher, Frank Leymann, and Michael Wurster
Institute of Architecture of Application Systems, University of Stuttgart, Germany
[lastname]@iaas.uni-stuttgart.de

ABSTRACT

In the domain of IoT a major objective is the interconnection of a variety of devices with higher level applications. Therefore, several IoT middleware systems have been developed. These IoT integration middleware systems are heterogeneous, e.g., in terms of the supported transport protocols. Thus, IoT environments often differ due to the utilized middleware. As a result, by deploying applications in different environments the communication clients on the application side have to be adjusted manually. This leads to a greater development effort for each deployment and hampers the application's portability. In this paper, we present a generic driver injection concept to enable the development of portable IoT applications and the automated deployment in different environments without manual adaptation efforts. For this, a programming model and a deployment modeling concept are introduced. We demonstrate the feasibility of our approach with a TOSCA-based prototype.

CCS CONCEPTS

• **Computer systems organization**; • **Software and its engineering → Software notations and tools**;

KEYWORDS

IoT Application Deployment, Drivers, Programming Model, TOSCA

ACM Reference format:
Karoline Saatkamp, Uwe Breitenbücher, Frank Leymann, and Michael Wurster. 2017. Generic Driver Injection for Automated IoT Application Deployments. In *Proceedings of the 19th International Conference on Information Integration and Web-based Applications & Services, Salzburg, Austria, December 4–6, 2017 (iiWAS '17),* 10 pages.
https://doi.org/10.1145/3151759.3151789

1 INTRODUCTION

In the last years, the importance of the Internet of Things (IoT) has increased significantly enabled by the latest developments in object identification, sensing, and computation [1, 9]. The main objective is to gather and process information by a variety of sensors and to change the physical world by actuators, if necessary [9]. To enable the interconnection of a large number devices and higher level applications IoT integration middleware is used for establishing the communication between them. Multiple IoT middleware systems such as Eclipse Mosquitto[1] or FIWARE Orion[2] are already available. Each has its own characteristics, fits for different requirements, and is highly heterogeneous in terms of, e.g., the supported communication protocols [10, 16, 19]. Several components form the entire IoT environment [10]: The sensors and actuators are attached to devices, which serve as bridges between hardware and software components. The IoT integration middleware connects devices and higher level applications. Because of the heterogeneity of middleware systems, IoT environments differ greatly from each other. Therefore, the device software and the higher level applications have to be tailored to the capabilities of the used middleware to enable integration [8]. Often a manual adaptation of the source code is required, which is costly, time-consuming, and error-prone. Thus, the development of the applications as well as the deployment are affected if applications are used in different environments.

In this paper, we present a generic driver injection concept to enable the development of portable IoT applications and the automated deployment of the entire IoT environment with different IoT middleware systems. Other approaches address the deployment of devices [21] or provide modeling approaches based on the TOSCA standard [8, 15, 22] or self-defined models [13] for IoT environments to enable their automated deployment. However, the integration with different IoT middleware systems and thus the portability is not tackled. For this, we introduce a programming model that facilitates the selection of appropriate drivers for applications depending on the used middleware during deployment time. These drivers enable the communication between the applications and the IoT middleware. Based on a middleware-independent modeling concept deployment scenarios can be defined without a specific middleware. Depending on the environment the deployment model is completed and the suitable drivers are selected in an automated manner. Thus, the adaptation effort for the application deployment in different IoT environments shifts from a manual application's implementation adaptation to a model adjustment. This decreases the development and deployment effort and improves the portability of IoT applications. Because TOSCA is a widely used standard for describing application deployments in a portable manner, we validate the feasibility of our approach with a TOSCA-based prototype [18].

The remainder of this paper is organized as follows: Section 2 introduces fundamentals and motivates our approach. Section 3 describes the general approach in detail, while Section 4 maps the approach to TOSCA, and Section 5 validates our approach based on our prototypical implementation. Finally, Section 6 discusses related work and Section 7 concludes the paper.

[1]https://mosquitto.org/
[2]https://fiware-orion.readthedocs.io/en/master/index.html

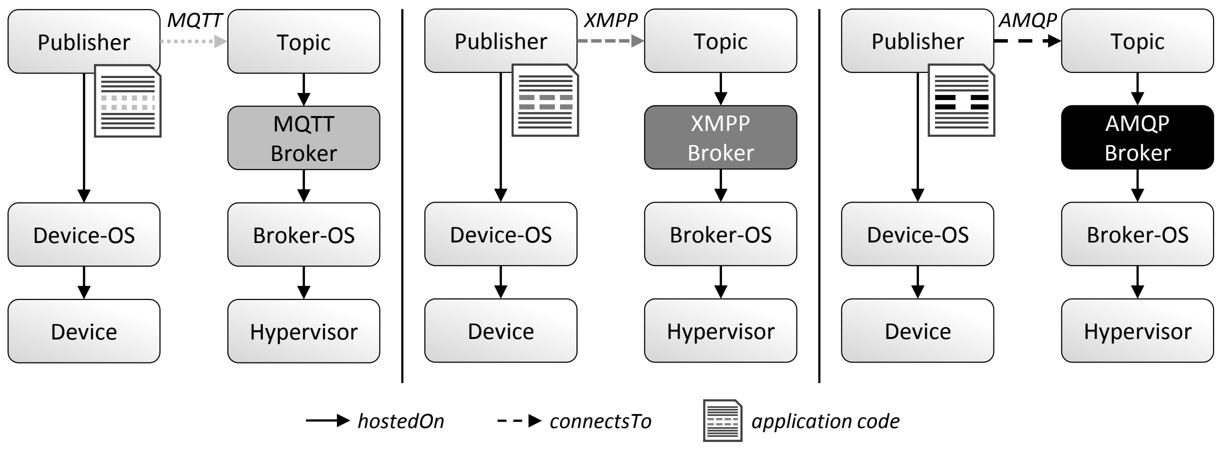


Figure 1: Impact of different IoT middleware systems on the communication and application's implementation

2 FUNDAMENTALS AND MOTIVATION

In cloud computing several deployment technologies and standards are available to ease the deployment of applications in cloud environments. Well known deployment and configuration technologies such as Chef[3] or Ansible[4] and container approaches such as Docker[5] are widely used. A common standard to describe cloud application deployments in a vendor independent and portable manner is the *Topology and Orchestration Specification for Cloud Applications (TOSCA)*, which is an OASIS standard enabling the automated deployment and management of applications [4, 18]. For this, cloud applications are described by deployment models containing a topology, which represents the structure of the application. This topology specifies the components of the application and the relations between them. Such relations can express, for example, that a component is *hostedOn* or *connectsTo* another component.

Different works demonstrated how TOSCA can be used to automate the deployment of the entire IoT environment [8, 15]. However, TOSCA models that describe such deployments typically specify the used IoT middleware to which devices and the corresponding applications connect. Thus, (i) the software on the devices as well as (ii) all applications that interact with these devices are bound to the used middleware. As a result, when the devices and applications are used with a different environment and middleware, respectively, complex code adjustments are required to bound them to an other middleware [8]. For example, Figure 1 depicts three topologies presenting the same IoT scenario: an application publishing messages to a topic. For a better understanding this simplified scenario with just one publisher is chosen. Everything that applies to the publisher also applies to subscribers and further publishers.

Each topology consists of two stacks: (i) an IoT middleware stack and (ii) an IoT device stack. The device stack represents the device, e.g., a Raspberry Pi, with an operating system and an application running on the device. Such an application can be, e.g., a Python script which reads and publishes the data from a sensor connected to the device. The application code snippet added to the *Publisher* component illustrates the application's implementation. The middleware stack shows the IoT middleware in form of a message broker hosted on a virtual machine. The topic is used by the application to publish, e.g., sensor data.

The choice of a particular message broker affects the supported transport protocol, e.g., if MQTT, XMPP, or AMQP messages can be handled by the broker. This in turn means that the part of the application code responsible for the communication with the IoT middleware has to be changed accordingly to use the required communication client on the application side. This is necessary to establish a connection and to be able to publish or subscribe to a topic. The adaptation of the implementation is needed because the processing of outgoing and incoming messages depends on the transport protocol. Thus, not only a model but also an implementation adaptation is required if the IoT middleware changes as not all IoT middleware systems support the same protocols.

The implementation adaptation effort limits not only the reuse of applications in other environments but also impedes the change of the IoT middleware in an existing environment (technology lock-in). For each deployment of the application with a middleware the application is not designed for, a lot of manual effort is necessary to adjust the application code. The matching of components to a different environments can be already automated by topology completion algorithms [11, 20]. However, the adaptation of the communication behavior is still a challenge. To enable an automated adaptation, we propose to shift the adjustment effort from an application's implementation adaptation to a deployment model adjustment to tackle these issues.

In order to achieve the mentioned objective, we introduce (i) a programming model to implement the application independently of the used IoT middleware and to enable the injection of drivers according to the chosen middleware. Moreover, we present (ii) a concept for the middleware-independent modeling of the application's deployment and (iii) an approach for the automated model adjustment depending on the chosen IoT middleware. These generic driver injection concept enables the development and deployment of IoT applications in a portable manner.

[3]https://www.chef.io/chef/
[4]https://www.ansible.com/
[5]https://www.docker.com/

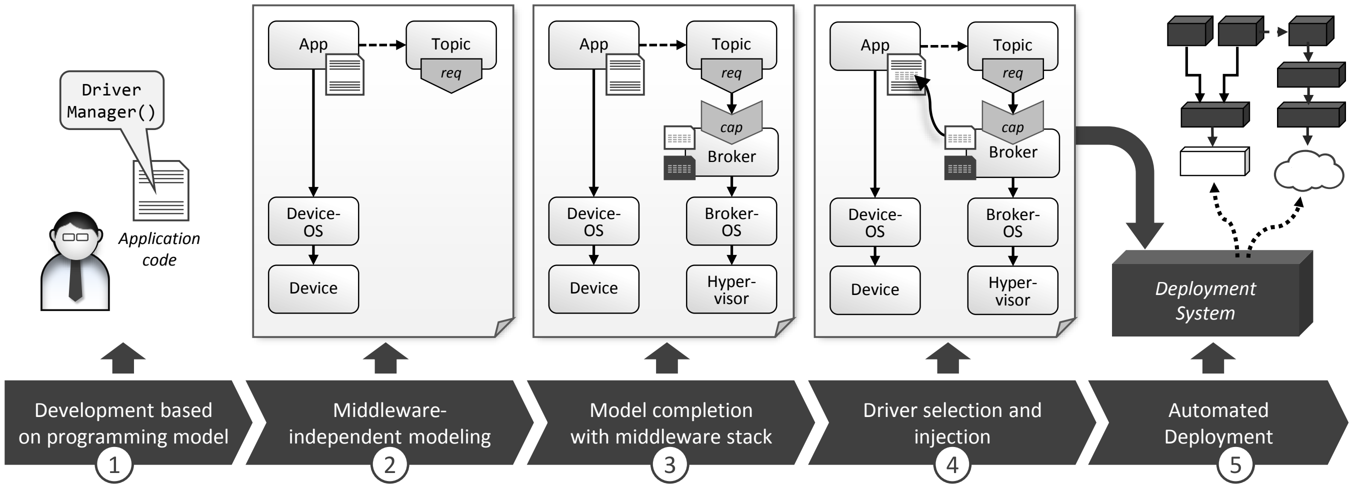

Driver Manager()
Application code
App
Topic
req
cap
Broker
Device-OS
Device
Broker-OS
Hyper-visor
Deployment System
Development based on programming model
1
Middleware-independent modeling
2
Model completion with middleware stack
3
Driver selection and injection
4
Automated Deployment
5

Figure 2: Overview of the generic driver injection concept

3 GENERIC DRIVER INJECTION CONCEPT

To deal with the mentioned challenges, we introduce a programming model in Section 3.2 to encapsulate the communication capability between applications and the different IoT middleware systems. Furthermore, we present a modeling concept for an automated deployment in different IoT environments in Section 3.3.

3.1 Overview

An overview of the generic driver injection concept is depicted in Figure 2. All relevant steps from the development to the deployment of an application in an IoT environment are covered. Our concept bases on five steps: First of all, the application is developed in step 1. The *programming model* contains predefined *Driver Manager* libraries for different programming languages such as Python or Java. Under use of these libraries for the development of applications, drivers can be selected and used to establish the communication with a specific middleware.

In the second step, a *middleware-independent model* is defined. As already seen in Figure 1, the application code is part of the model but the communication specific code is removed. Additionally, the Topic component is tagged with a requirement *req*. It indicates that the model is incomplete and not deployable. Based on the exposed requirement an available middleware with a suitable capability has to be attached in step 3. Each broker comes with a set of *Drivers* in different programming languages which can be used by the Driver Manager to establish a connection. In the fourth step, the driver in the appropriate programming language is selected and linked to the application code that the deployment works properly. In the last step, the deployment model is deployed as specified: the application code and the driver are deployed at the device. The driver is used by the Driver Manager of the application to establish the connection. Thus, IoT applications can be developed in a portable manner and the deployment model can be tailored middleware-dependent for an automated deployment. In the following, the middleware-independent modeling, the model adjustment, as well as the programming model are presented more detailed.

3.2 Programming Model

The goal of our programming model is to completely encapsulate the communication between an IoT application and the middleware and is used for the development of the application in step 1. By using the predefined *Driver Manager* library for the implementation of the application a connection can be established to each middleware providing a suitable driver. Because this programming model is intended for the IoT domain, we focus on publishing and subscribing to topics. Nevertheless, the concept of the programming model is generic and feasible for other domains as well.

Figure 3 depicts the concept of our programming model. We distinguish between *development time* (upper half) and *deployment time* (lower half). Starting with the development time, at the very top of the figure a set of *Driver Manager* libraries are shown. These libraries are predefined for different languages and versions, e.g., Java 8 or Python 3. For the IoT domain they expose the methods *publish()* to publish and *subscribe()* to subscribe to a topic. However, the concrete implementation of these methods depends on the broker and can not be predefined as the supported transport protocol and other middleware-specific information can vary.

An application developer uses the Driver Manager library to implement his application. Furthermore, matching *drivers* for the middleware systems implementing interfaces of the Driver Manager are implemented, for example, by the middleware provider or other experts. These drivers are the concrete implementation of the communication behavior between an application with this specific middleware. If no middleware driver is present, e.g., during the development of the application, a stub implementation of the driver interface is available as a fallback to have a runnable code. For each middleware, drivers in different programming languages can be available. For example, a MQTT broker provides drivers in Java 8 and Python 3 implementing the methods required to establish a MQTT connection to the broker and to publish or subscribe to a topic. The drivers are created just once for each middleware and can be reused for each application.

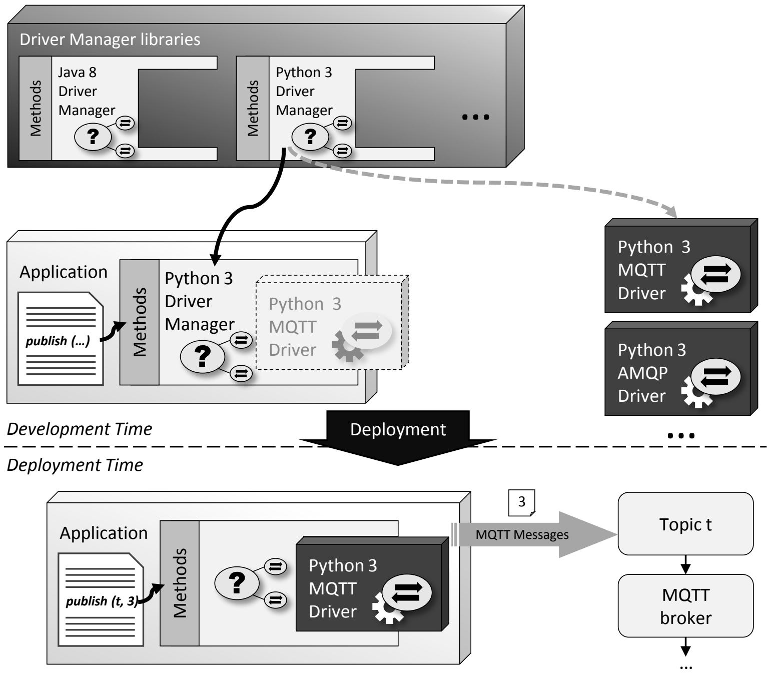


Figure 3: Programming Model

During deployment time the application and a suitable driver are provisioned. The selection of the driver is covered in Section 3.3. For establishing the connection, the correct driver has to deployed in a way, that the Driver Manager are able to use the driver for connecting the application to the broker. Depending on the underlying programming language, different mechanisms are required to achieve this. For example, in case of an application as JAR (Java Archive) file, the dependency to the driver can be solved by referencing the driver class available on the *classpath* to run the application. In case of Python the driver has to be stored in a known directory at the target host. Other information, like the topic or IP address of the broker, are provided, e.g., by a configuration file.

Using this programming model the communication behavior is encapsulated and has not be considered during the application development. For each broker, drivers implementing the specific communication behavior can be provided. Depending on the selected middleware, the driver can be injected and a connection can be established. No more manual adaptations of the application code are required in case of changing middleware.

3.3 Middleware-Independent Modeling and Model Adjustment

In this section, we introduce the middleware-independent modeling concept to model deployment models for IoT environments independently of the IoT middleware. Furthermore, we describe the adaptation steps during modeling and deployment time (steps 2 to 4) and the deployment in step 5 in detail. For this purpose, we show a detailed deployment model and its adjustment in Figure 4.

The depicted topology in Figure 4 is more detailed compared to the topology seen in Figure 1. The types of components and relations are defined. In this example a Python application *Publisher* is used, which requires a Python 3 runtime at the hosting device. This dependency is modeled by the relation of type *dependsOn*. The used device is a Raspberry Pi with a Raspbian Jessie operating system. Additionally, *requirements*, *capabilities*, and *executables* can be added to components. In this example, a requirement *MessageBroker* is attached to the Topic. A requirement can be fulfilled by a capability as exposed by the MQTT broker. All executables, e.g., JARs or Python scripts are also contained in the topology model. Figure 4 illustrates the steps 2 to 5 (cf. Figure 2), described more detailed in the following.

3.3.1 Middleware-independent modeling (step 2). The IoT environment is modeled independently of the middleware as shown at the left side of Figure 4. In this example only one publishing application connected to a topic is shown to simplify the scenario. However, multiple applications connected to the topic are feasible, i.e., also complex deployment models are portable. Each component that publishes or subscribes to a topic has a required driver type as indicator for the required programming language of the driver assigned. In this example, the *Publisher* component has a *Python 3 Driver* type assigned. The automated driver specification is achieved by type inheritance explained in step 4.

The Topic component has an open requirement attached. An open requirement means, that no outgoing relation to a matching capability is assigned yet. This middleware-independent model is therefore incomplete and not deployable. It has to be completed by a specific middleware stack. Depending on the available IoT middleware in a specific environment the model can be completed.

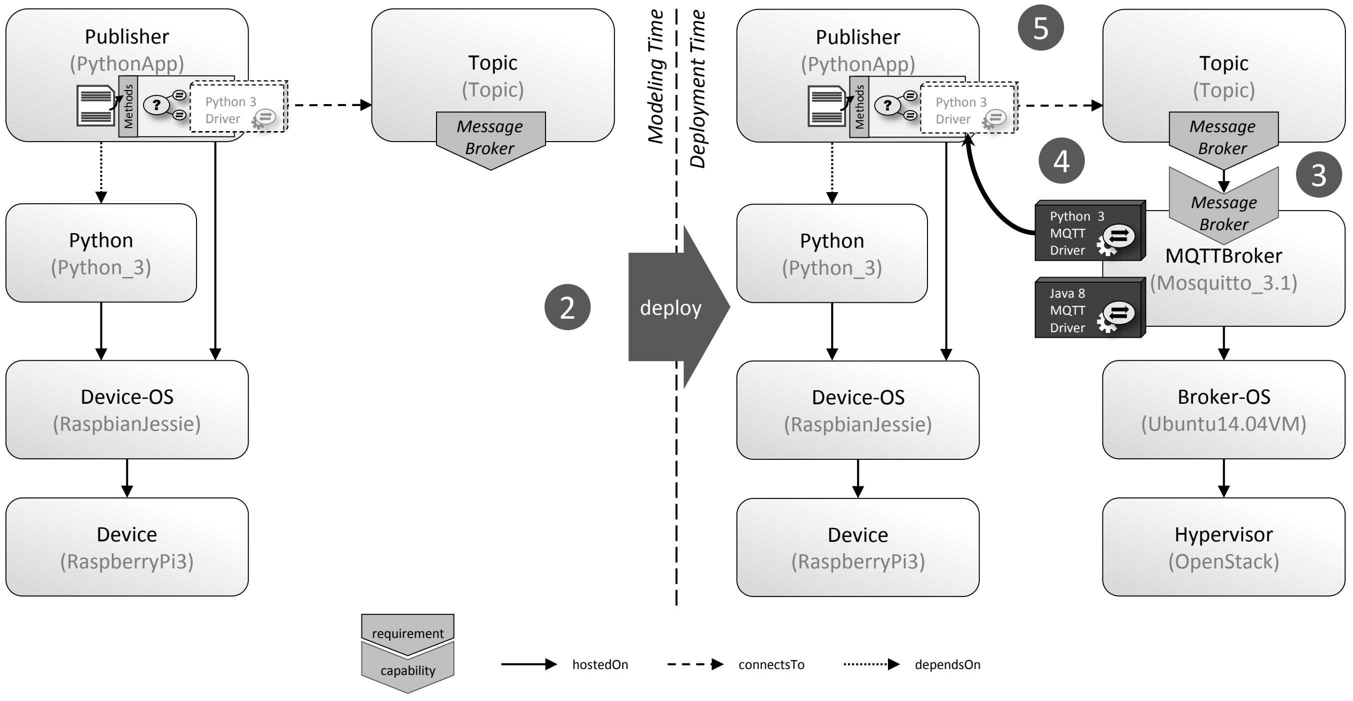


Figure 4: Middleware independent model (left) and model adjustment steps during deployment time (right)

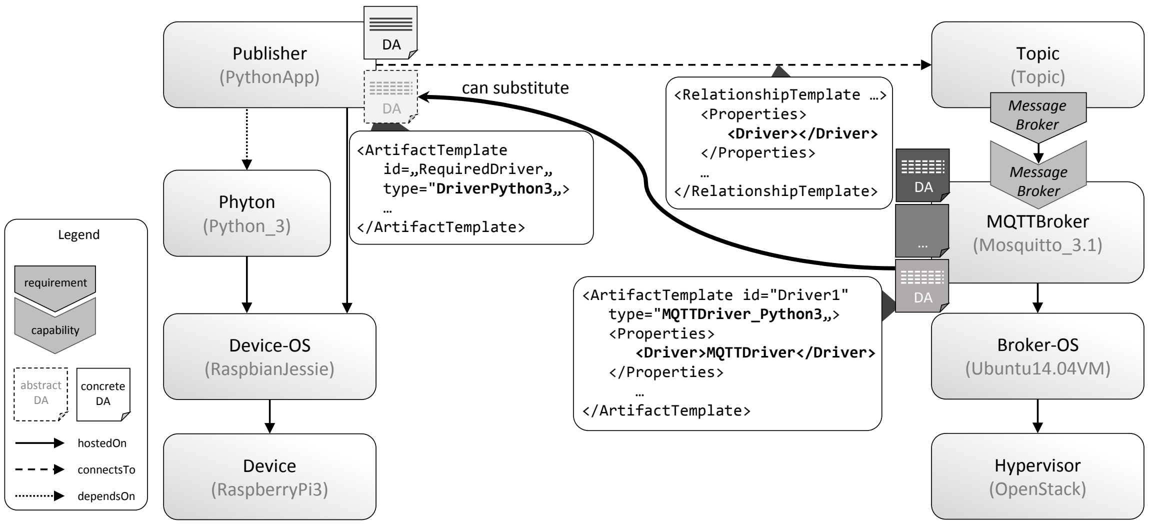
3.3.2 Model completion with middleware stack (step 3). In this step, a suitable middleware stack has to be selected and added to the topology. For the open requirement *Message Broker* an available middleware stack with the corresponding capability has to be found. This can be done either manually by the modeler or automated by, e.g., a topology completion approach [11, 20]. The relation between the requirement and the capability is realized as *hostedOn* relation. The middleware comes with the specific drivers for the supported programming languages. The drivers base on the programming model described in the previous section. This step is executed either during modeling time in case of a manual completion or during deployment time in case of an automated completion.

3.3.3 Driver selection and injection (step 4). During deployment time, a driver is selected in this step. For each component with a required driver type, the middleware stack the component is connected to is considered. The outgoing relations of the application component are the indicators to detect the connected middleware stacks. Based on the required driver type of the application and the middleware-specific driver types, the right driver is selected. The middleware-specific drivers implement the driver interface as necessary for the communication with the middleware. As depicted in Figure 4 two driver artifacts are attached to the *MQTTBroker*: one driver of type *Python 3 MQTTDriver* and one driver of type *Java 8 MQTTDriver*. The first one is a specialized type of the *Python 3 Driver* and therefore selected in this example.

The driver is added to the application because the application code as well as the driver have to be deployed in a way that the Driver Manager, which is part of the application's implementation, can use the driver for establishing the communication with the middleware. Depending on the language of the application, different mechanisms are used to bind the driver. For example, by setting Java's *classpath* while starting the Java application or using a specific directory to store the driver artifact, the Driver Manger is able to reference the driver. Not only one but also multiple drivers can be bound to the Driver Manager in case an application has to communicate with multiple middleware systems. That means, for each IoT middleware the application shall be connected to, a driver is selected and the mapping between selected driver and middleware is used to establish the connection by use of the driver.

3.3.4 Deployment (step 5). After the deployment model is completed and the drivers are selected and injected, the IoT environment can be finally automatically deployed as defined. For this purpose, a declarative runtime is used which does not require explicit management plans for the instantiation but interpret the topology to infer the deployment logic [5]. A detailed description of a system architecture is given in Section 4.3. Based on the topology model in our example the application logic as well as the driver are deployed. Afterwards, the application can publish data to the topic.

The basis of this presented modeling and model adjustment concept is an appropriate development of the used applications. The application has to be designed for the use of drivers to enable the communication with IoT middleware systems. The application development must base on the programming model described in Section 3.2, otherwise the injection of drivers is not possible.



```
<ArtifactTemplate
   id=„RequiredDriver„
   type="DriverPython3„>
   …
</ArtifactTemplate>
```

```
<RelationshipTemplate …>
   <Properties>
      <Driver></Driver>
   </Properties>
   …
</RelationshipTemplate>
```

```
<ArtifactTemplate id="Driver1"
   type="MQTTDriver_Python3„>
   <Properties>
      <Driver>MQTTDriver</Driver>
   </Properties>
      …
</ArtifactTemplate>
```

Figure 5: TOSCA-based topology model

3.4 Limitations of the Approach

The presented approach bases on the assumption, that the introduced programming model is used to implement the IoT applications and the middleware drivers. In case an application does not fulfill this requirement, the application can either not be used for the deployment or has to be adapted accordingly. The same applies to the middleware. However, a subsequent adjustment of an application or implementation of middleware drivers is required only once. Afterwards the approach can be used for a recurrent deployment in different environments.

Furthermore, the approach is valid for all programming languages for which mechanisms exist to run an application using external libraries or scripts. Programming languages such as Python, Java, or C++ enable reflection or dynamic linking at runtime. For languages that can handle only static linking, such as Go, our concept is not applicable. Nevertheless, there is a workaround to obtain the same result as with our approach: The drivers are deployed as independent software components and a communication is established through inter-process communication (IPC) techniques between the IoT application and the driver, which forwards the received data to the connected middleware. The result is the same but the solution differs.

4 VALIDATION BASED ON TOSCA

The generic driver injection concept presented in Section 3 can be applied to TOSCA, a standard to describe cloud application deployments. In this section, we present (i) a TOSCA-based solution of our concept and (ii) a system architecture for the automation of the model adjustment steps. On the one hand, TOSCA is chosen to demonstrate that our concept can be mapped to an existing standard, and on the other hand to make use of the existing toolchain for our prototype: the modeling tool Eclipse Winery™ [14] and the TOSCA runtime OpenTOSCA container [3]. Our prototypical implementation is shown in Section 5.

4.1 TOSCA Fundamentals

TOSCA is a standard to describe cloud application deployments by topologies. A topology specifies the components of an application and the relationships between them [18]. The components are specified as *Node Templates* and the relations as *Relationship Templates*. The semantic of Node as well as Relationship Templates are defined by the used *Node Types* and *Relationship Types*, respectively.

For the deployment of the components often executables have to be installed and operations are executed. All executables, e.g., JARs, WARs, and shell scripts, are called *artifacts*. Two kinds of artifacts are distinguished: *Deployment Artifacts (DA)* representing the application logic and *Implementation Artifacts (IA)* representing executables required to deploy the component, e.g., install scripts. DAs can be attached to Nodes and IAs to Nodes and Relationships, depending on where the artifact shall be executed. Similar to Node Types, *Artifact Types* specify the type of the artifacts. All types in TOSCA can be declared as abstract. This means, at latest during the deployment the artifact of an abstract type has to be substituted by an artifact using a specialized derived Artifact Type. Each TOSCA type can inherit from another type and, thus, inheritance hierarchies can be established. Additionally, for each type *Properties* can be defined to add additional information to Templates, e.g., the topic name or IP address.

The basis elements to enable an automated topology completion are *Requirements* and *Capabilities*. Each defined *Requirement Type* has a *required Capability Type* assigned. Based on this, a matching between Requirements and Capabilities can be realized. After a matching is found, it can be solved by a Relationship Template. The type of the Relationship Template depends on the semantic of the Capability. These are the required TOSCA elements to realize the presented generic driver injection model. For the automated deployment a declarative TOSCA runtime is required which derives the deployment logic from the topology itself and does not need explicit deployment plans [5].

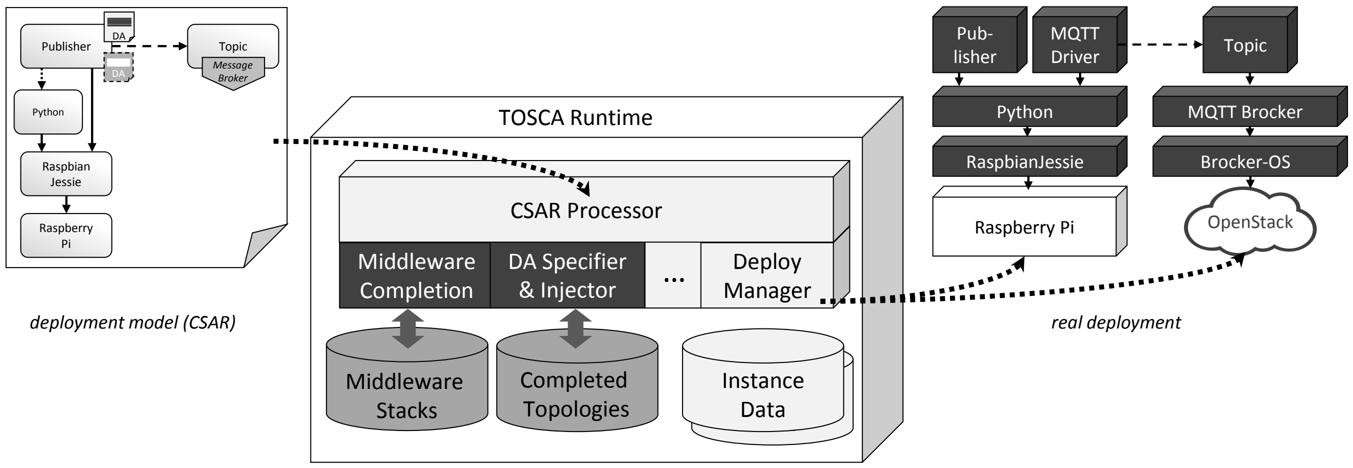


Figure 6: Simplified system architecture of an extended TOSCA runtime

4.2 TOSCA-Based Realization

The presented middleware-independent modeling as well as the model adjustment can be realized with TOSCA. In Figure 5 a TOSCA topology model is depicted. First of all in step 2 (cf. Figure 2), the device stack (left stack) and the topic connected by a Relationship Template of type *connectsTo* or any derived type with the Python application are modeled. The middleware completion (step 3) can be done manually by adding the desired middleware components to the model or by existing automated matching mechanisms [11, 20]. The automated completion bases on a matching between Requirements and Capabilities attached to Node Templates.

Two DAs are attached to the Node Template *Publisher*: a concrete DA representing the application logic and an abstract DA representing the driver. The abstract DA uses an abstract Artifact Type *DriverPython3*. The abstract driver DA serves as a placeholder for the concrete drivers of the connected middleware stacks. As mentioned before, at latest during the deployment all DAs referencing abstract artifacts have to be substituted. For this, all Deployment Artifacts referencing artifacts of an abstract Artifact Type are recognized in step 4. By analyzing the outgoing relationships of the Node Templates with attached abstract DAs, the middleware stacks can be discovered. Based on the mandatory inheritance hierarchy used for the Artifact Types the suitable concrete DA can be found. The abstract driver DAs are of an abstract Artifact Type which depends on the programming language of the application. In the example in Figure 5, it is a Python 3 application and therefore an artifact of the abstract Artifact Type *DriverPython3* is attached.

The concrete driver DAs use specialized derived Artifact Types, e.g., *MQTTDriver_Python3* or *AMQPDriver_Python3*. Based on the underlying type convention that for each programming language one abstract Artifact Type for drivers exists, the suitable concrete DAs can be uniquely identified. If an application is connected to multiple topics of different middleware systems, the abstract DA can be replaced by multiple concrete DAs. The abstract DA just serves as placeholder and indicator to start the DA specification process. Especially for multiple drivers it is important to know which driver is used for which connection. For this, a property *Driver* attached to each connectsTo Relationship Template is used referencing the driver used to establish this concrete connection between the application and the topic.

Additionally to the DAs, IAs are required for the deployment in step 5 to install the application logic and the drivers in a way, the Driver Manager can use the drivers for establishing the connection and for the communication itself. Such IAs are required for the lifecycle operations of an application [18]. They are attached to the Node Template itself or the Node Type Implementation. These IAs are needed, e.g., for installing the application logic and drivers and for establishing the connection to the topic. However, all IAs are hidden in Figure 5 to reduce complexity.

4.3 System Architecture

As already mentioned in previous sections, a declarative runtime for the automated deployment is required. To facilitate (i) the model completion with middleware stacks and (ii) the driver selection and injection such a declarative runtime has to be extended. Therefore, in this section, we present a system architecture for a TOSCA runtime that supports our introduced approach. The system architecture is depicted in Figure 6 in a simplified manner to focus on the components important for our approach. Multiple other components such as the Artifact Manager or Instance Manager are also important for the deployment but not depicted for a better clarity. A detailed view on the components of a TOSCA runtime, required to instantiate an application are given in the TOSCA Primer [17].

The two main components to realize our approach are the *Middleware Completion* and the *DA Specifier & Injector* component using two repositories: the *Middleware Stack repository* provides the predefined middleware stacks used for the automated completion based on open requirements and the *Completed Topology repository* stores the completed topologies with the injected drivers.

If an incomplete TOSCA topology is given to the TOSCA runtime as a Cloud Service Archive (CSAR), the *CSAR Processor* component unpacks the CSAR to make the stored files available for the *Deploy Manager* and to analyze the contained topology model. In case the topology is incomplete, the CSAR is forwarded to the Middleware Completion component that checks the open requirements and browses the Middleware Stack repository for a matching middleware stack. When a matching topology fragment is found, the fragment is attached to the corresponding requirement.

After the completion process, the DA Specifier & Injector component identifies all Node Templates with attached abstract DAs. By analyzing the outgoing relationships the relevant middleware stacks for each Node Template are discovered. The concrete DAs supporting the right programming language are chosen and the abstract DAs are replaced. This completed topology is stored in the Completed Topologies repository and the Deploy Manager deploys the topology model accordingly and stores the instance data in the respective repository. On the right side of Figure 6 the deployed real IoT environment from the running example are depicted. On a Raspberry Pi and on an OpenStack the modeled components are deployed. Additionally, the *MQTT Driver* provided by the middleware is deployed on the same host as the application. Thus, with the extended TOSCA runtime the middleware completion as well as the DA selection and injection can be executed.

5 PROTOTYPE AND APPLICATION

For the concept validation, we prototypically implemented our approach by extending the existing TOSCA-based ecosystem OpenTOSCA with the modeling tool *Eclipse Winery*[6] and the TOSCA runtime *OpenTOSCA Container*[7]. The TOSCA modeling tool Winery can be used to model topologies and to export them as CSARs [14]. The OpenTOSCA Container is a TOSCA-compliant runtime that can process CSARs and deploy the applications accordingly [3]. Figure 7 illustrates the enriched OpenTOSCA Container. Components, which are not in focus of this work are skipped but can be looked up in Binz et al. [3]. The Container Repository extends the existing Container, whereby its source code bases on the source code of the Winery. The OpenTOSCA Container uses the Container Repository for the model adjustment during deployment time.

[6]https://github.com/eclipse/winery
[7]https://github.com/OpenTOSCA

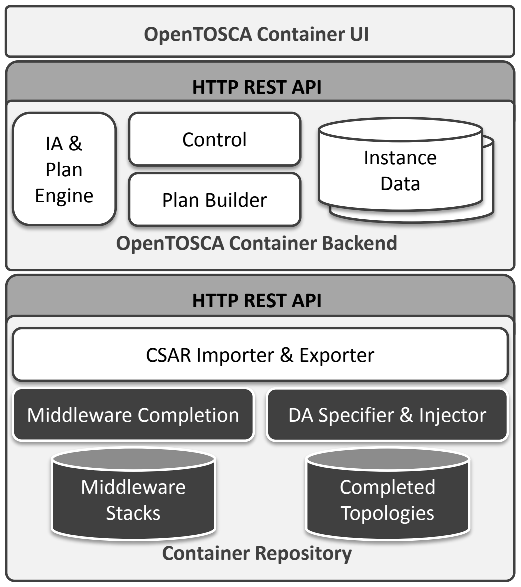


Figure 7: Enriched OpenTOSCA ecosystem

The middleware-independent model can be modeled with the Winery and exported as CSAR. This *OpenTOSCA Container UI* can be used to upload the CSAR to the Container. The *Control* component interprets the included topology and checks if it contains open requirements. In case the topology is incomplete, the OpenTOSCA Container forwards the CSAR to the Container Repository for the completion. The *Middleware Completion* component matches, in the first step, the open requirements of the incomplete topology to topology fragments from the *Middleware Stacks* repository. In the Middleware Stack repository the topology fragments are stored as TOSCA topology templates in a specific namespace that identifies these topologies as basis for the middleware completion. After the completion, the *DA Specifier & Injector* component detects the abstract DAs and replaces them by concrete DAs as described in previous sections. This completed and specified topology is exported from the Container Repository to the OpenTOSCA container, which starts again the deployment of the topology: the *Plan Builder* generates the plans based on the topology and the *IA & Plan Engine* processes the plans and required IAs for the instantiation.

This prototype is used to demonstrate how to apply the introduced approach to a concrete use case scenario. We want to show, that (i) a MQTT broker (Eclipse Mosquitto[8]) and a AMQP broker (RabittMQ[9]) can be used interchangeable based on the generic driver injection model and that (ii) this can be done for a Python as well as a Java application. The topology model for this test scenario is illustrated in Figure 8. A Python application hosted on a Raspberry Pi publishes data to a topic. The Java application running on a virtual machine subscribes to this topic to process the data. The marked middleware stack is interchangeable and not part of the middleware-independent model.

In a first step, the Driver Manager as described in Section 3.2 is implemented in Python and Java. The libraries are used to implement the Python *Publisher* application and the Java *Subscriber* application. The middleware-independent model is described appropriate by adding two DAs to each application Node Template. For the Python application the DA is of type *DriverPython3* and for the Java application of type *DriverJava8*. Both are abstract Artifact Types and required for a later DA specification and injection. Furthermore, the available middleware stacks with the two different brokers have to be prepared as topology models. For each broker, a topology is modeled containing the broker itself with its hosting environment. A Capability is added to the broker Node Template to enable an automated completion. The broker specific driver interface implementations are added to the Node Template as concrete DAs: one for Python and one for Java applications. The Artifact Types are derived from the abstract types attached to the Publisher and the Subscriber, respectively. These middleware fragments are stored in the Middleware Stacks repository.

After the development and modeling phase the deployment is executed. First of all, the Middleware Completion component searches

[8]https://mosquitto.org/
[9]https://www.rabbitmq.com/

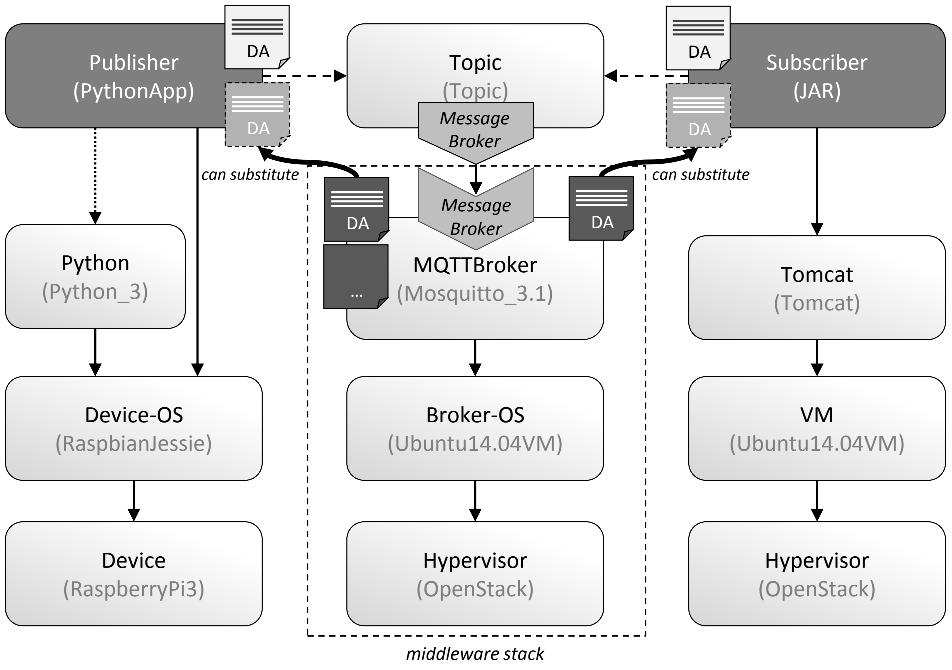


Figure 8: Case study topology model

for a matching topology fragment for the open requirement *Message Broker* in the topology model. For a matching capability, the correct Relationship Type to connect the requirement with the capability is determined based on the inheritance hierarchy of the capability. In our case, a *hostedOn* relationship is chosen. In the next step, the DA Specifier & Injector component determines all abstract DAs contained in the topology model. For the abstract DAs attached to the Publisher Node Template and Subscriber Node Template, concrete DAs are detected at the Broker Node Template which replace the abstract ones on both sides. The deployment of the use case with the Eclipse Mosquitto as well as the RabbitMQ middleware stack was successful. This demonstrates the feasibility of our approach with a common used standard like TOSCA.

6 RELATED WORK

Although different approaches exist dealing with the automated deployment of IoT applications [8, 13, 15, 22], the portability between different IoT environments and the automated deployment adjustment to enable the integration with the different middleware systems are not tackled. Li et al. [15] show how TOSCA can be used in general for modeling IoT applications. With the LEONORE framework an extension of this work is presented to enable a large-scale provisioning of IoT applications in a heterogenous environment [21]. Preinstalled components are required on each IoT gateway (i.e. device) and the portability as well as integration with higher level applications is not addressed.

Franco da Silva et al. [8] focus on the deployment of the whole IoT environment with TOSCA and on the difference of the deployment models with different IoT middleware systems. The provisioning of the same IoT scenario with different middleware is realized by a remodeling of the TOSCA deployment model and an adjustment of the application code. The approach implies that (i) the application code is accessible and (ii) experts familiar with the application code are available. Both prerequisites are often not met, especially if software provided by third parties are used. But also if they are satisfied, it is a great development effort, especially for complex applications. These are the issues we tackle with our approach.

Hur et al. [13] address the deployment across different IoT platforms by using a semantic service description ontology to get a common understanding of heterogeneous devices and service deployments. However, the integration of applications and middleware systems is not tackled.

Zimmermann et al. [22] introduce an approach to ease the interaction of components by expressing the application operations as application interfaces in TOSCA and to enable the communication via service bus. The use of a service bus always generates an overhead, which is not practical in many IoT environments.

Well known are the enterprise integration patterns which documents best practice solution for the integration of applications [12]. The messaging endpoint of an application is responsible to send and receive messages. Such an endpoint is at least a messaging gateway. The messaging gateway encapsulates the communication behavior from the application logic. As presented in Guth et al. [10] such a gateway is often part of the IoT architecture to enable the communication between the devices and the IoT integration middleware. Our driver concept implements this gateway pattern.

Different gateways to transform the transport protocol between IoT devices, middleware, and higher-level applications are already introduced [2, 6]. Desai et al. [6] introduce a Semantic Gateway as a Service for the translation between messaging protocols to enable the interoperability and independence between different protocols. For this, a multi-protocol proxy is developed to bridge between the communication clients on the device side and the message broker. However, the broker is part of the proxy and not exchangeable. An extensible intelligent gateway is presented by A-Fuqaha et al. [2] able to transform different protocols and to take QoS aspects into account. Instead of protocol translations for the different protocols, we provide a programming model to enable the development of drivers for arbitrary middleware systems.

The technical realization of our approach bases on the Service Locator Pattern [7]. The Driver Manager is the central entity that deals with the dependencies to the drivers required to establish the connections between a software component and a specific middleware. Thus, the dynamical binding of drivers during deployment time is possible. Compared to the other approaches, our concept enables not only the interoperability independent of the communication protocol because of the middleware-dependent driver injection, but also the portability and automated deployment and integration of the devices and higher level applications in different IoT environments with different middleware systems.

7 CONCLUSION

In this paper, we presented our generic driver injection concept to enable the development of portable IoT applications, which can be deployed in different IoT environments with different IoT integration middleware systems. This concept eliminates the manual adaptation effort to adjust the communication client on the application side for the communication with the respective middleware. For this, we introduced a middleware-independent modeling approach for deployment models. The IoT scenario with the devices and applications are modeled independently of the used IoT middleware. For each IoT environment this middleware-independent model is completed by the available middleware. Based on the introduced programming model, the injection of communication drivers dependent on the selected middleware is facilitated. These drivers encapsulate the communication between IoT applications and a specific middleware. Each middleware comes with a set of drivers, which can be used by the IoT applications to establish the connection the middleware. Thus, the communication behavior between applications and message brokers is encapsulated in the driver which are provided by the middleware itself.

We presented furthermore the execution of the model completion with a specific middleware, the driver selection and injection, and the deployment of the IoT environment in an automated manner. We validated this approach with a TOSCA-based prototype and demonstrated the feasibility by using the prototype for a concrete use case scenario. In this paper, we focused on a specific IoT programming model. However, it is a generic concept which can be applied to different domains. The extension of the approach to other domains is part of future works.

ACKNOWLEDGMENTS

This work is partially funded by the BMWi project *SmartOrchestra* (01MD16001F) and *IC4F* (01MA17008G).

REFERENCES

[1] Ala Al-Fuqaha, Mohsen Guizani, Mehdi Mohammadi, Mohammed Aledhari, and Moussa Ayyash. 2015. Internet of things: A survey on enabling technologies, protocols, and applications. *IEEE Communications Surveys & Tutorials* 17, 4 (2015), 2347–2376.

[2] Ala Al-Fuqaha, Abdallah Khreishah, Mohsen Guizani, Ammar Rayes, and Mehdi Mohammadi. 2015. Toward better horizontal integration among IoT services. *IEEE Communications Magazine* 53, 9 (2015), 72–79.

[3] Tobias Binz, Uwe Breitenbücher, Florian Haupt, Oliver Kopp, Frank Leymann, Alexander Nowak, and Sebastian Wagner. 2013. OpenTOSCA – A Runtime for TOSCA-based Cloud Applications. In *Proceedings of the 11th International Conference on Service-Oriented Computing (ICSOC 2013)*. Springer, 692–695.

[4] Tobias Binz, Gerd Breiter, Frank Leymann, and Thomas Spatzier. 2012. Portable Cloud Services Using TOSCA. *IEEE Internet Computing* 16, 03 (May 2012), 80–85.

[5] Uwe Breitenbücher, Tobias Binz, Kálmán Képes, Oliver Kopp, Frank Leymann, and Johannes Wettinger. 2014. Combining Declarative and Imperative Cloud Application Provisioning based on TOSCA. In *International Conference on Cloud Engineering (IC2E 2014)*. IEEE, 87–96.

[6] Pratikkumar Desai, Amit Sheth, and Pramod Anantharam. 2015. Semantic Gateway as a Service Architecture for IoT Interoperability. In *Proceedings of the IEEE International Conference on Mobile Services (MS)*. IEEE, 313–319.

[7] Martin Fowler. 2004. Inversion of control containers and the dependency injection pattern. (Jan. 2004). https://martinfowler.com/articles/injection.html

[8] Ana Cristina Franco da Silva, Uwe Breitenbücher, Pascal Hirmer, Kálmán Képes, Oliver Kopp, Frank Leymann, Bernhard Mitschang, and Roland Steinke. 2017. Internet of Things Out of the Box: Using TOSCA for Automating the Deployment of IoT Environments. In *Proceedings of the 7th International Conference on Cloud Computing and Services Science (CLOSER 2017)*. SciTePress, 358–367.

[9] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. 2013. Internet of Things (IoT): A vision, architectural elements, and future directions. *Future Generation Computer Systems* 29, 7 (2013), 1645–1660.

[10] Jasmin Guth, Uwe Breitenbücher, Michael Falkenthal, Frank Leymann, and Lukas Reinfurt. 2016. Comparison of IoT Platform Architectures: A Field Study based on a Reference Architecture. In *2016 Cloudification of the Internet of Things (CIoT)*. IEEE, 1–6.

[11] Pascal Hirmer, Uwe Breitenbücher, Tobias Binz, Frank Leymann, et al. 2014. Automatic Topology Completion of TOSCA-based Cloud Applications. In *GI-Jahrestagung*. GI, Vol. P-251. GI, 247–258.

[12] Gregor Hohpe and Bobby Woolf. 2004. *Enterprise integration patterns: Designing, building, and deploying messaging solutions.* Addison-Wesley Professional.

[13] Kangho Hur, Sejin Chun, Xiongnan Jin, and Kyong-Ho Lee. 2015. Towards a Semantic Model for Automated Deployment of IoT Services across Platforms. In *Proceedings of the IEEE World Congress on Services (SERVICES*. IEEE, 17–20.

[14] Oliver Kopp, Tobias Binz, Uwe Breitenbücher, and Frank Leymann. 2013. Winery – A Modeling Tool for TOSCA-based Cloud Applications. In *Proceedings of the 11th International Conference on Service-Oriented Computing (ICSOC 2013)*. Springer, 700–704.

[15] Fei Li, Michael Vögler, Markus Claessens, and Schahram Dustdar. 2013. Towards Automated IoT Application Deployment by a Cloud-Based Approach. In *Proceedings of the 6th International Conference on Service-Oriented Computing and Applications (SOCA))*. IEEE, 61–68.

[16] Julien Mineraud, Oleksiy Mazhelis, Xiang Su, and Sasu Tarkoma. 2016. A gap analysis of Internet-of-Things platforms. *Computer Communications* 89 (2016), 5–16.

[17] OASIS. 2013. *Topology and Orchestration Specification for Cloud Applications (TOSCA) Primer Version 1.0.* Organization for the Advancement of Structured Information Standards (OASIS).

[18] OASIS. 2013. *Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0.* Organization for the Advancement of Structured Information Standards (OASIS).

[19] Mohammad Abdur Razzaque, Marija Milojevic-Jevric, Andrei Palade, and Siobhán Clarke. 2016. Middleware for internet of things: a survey. *IEEE Internet of Things Journal* 3, 1 (2016), 70–95.

[20] Karoline Saatkamp, Uwe Breitenbücher, Oliver Kopp, and Frank Leymann. 2017. Topology Splitting and Matching for Multi-Cloud Deployments. In *Proceedings of the 7th International Conference on Cloud Computing and Services Science (CLOSER 2017)*. SciTePress, 247–258.

[21] Michael Vögler, Johannes Schleicher, Christian Inzinger, Stefan Nastic, Sanjin Sehic, and Schahram Dustdar. 2015. LEONORE - Large-Scale Provisioning of Resource-Constrained IoT Deployments. In *IEEE Symposium on Service-Oriented System Engineering*. IEEE, 78–87.

[22] Michael Zimmermann, Uwe Breitenbücher, and Frank Leymann. 2017. A TOSCA-based Programming Model for Interacting Components of Automatically Deployed Cloud and IoT Applications. In *Proceedings of the 19th International Conference on Enterprise Information Systems (ICEIS)*, Vol. 2. SciTePress, 121–131.