Institute of Architecture of Application Systems

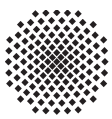
Deployment of Distributed Applications Across Public and Private Networks

Kálmán Képes, Uwe Breitenbücher, Frank Leymann,
Karoline Saatkamp, Benjamin Weder

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

BibTeX:

```
@inproceedings {Kepes2019_DistDeployment,
  Author = {K{\'a}lm{\'a}n K{\'e}pes and Uwe Breitenb{\"u}cher and
            Frank Leymann and Karoline Saatkamp and Benjamin Weder},
  Title = {{Deployment of Distributed Applications Across Public and Private
            Networks}},
  Booktitle = {Proceedings of the 23rd IEEE International Enterprise
               Distributed Object Computing Conference (EDOC)},
  Publisher = {IEEE},
  Pages = {236--242},
  Month = oct,
  Year = 2019,
  issn = {2325-6354},
  doi = {10.1109/EDOC.2019.00036}
}
```

Universität Stuttgart
Germany

Deployment of Distributed Applications across Public and Private Networks

Kálmán Képes, Uwe Breitenbücher, Frank Leymann, Karoline Saatkamp, Benjamin Weder
Institute of Architecture of Application Systems, University of Stuttgart, Germany
{kepes, breitenbuecher, leymann, saatkamp, weder}@iaas.uni-stuttgart.de

***Abstract*—The growing usage of software and hardware in our everyday lives has lead to paradigms such as Cloud Computing, Edge Computing, and the Internet of Things. The combination of these paradigms results in distributed and heterogeneous target environments: components of an application often need to be deployed in different environments such as clouds, private data centers, and small devices. This makes the deployment of distributed applications a complex and error-prone challenge as deployment systems have to (i) support cloud deployments, (ii) determine the location of physical resources, (iii) cope with security mechanisms preventing inbound communication, and (iv) use hardware-constrained devices. In this paper, we present an approach for the automated deployment of distributed applications on heterogeneous target environments consisting of public and private clouds, and devices. We especially tackle the issue of deploying components in environments having restricted inbound communication capabilities. We prototypically implemented and compared our approach based on a smart home scenario using TOSCA and the OpenTOSCA Ecosystem.**

***Index Terms*—distributed application deployment; heterogeneous infrastructures; orchestration; automation; TOSCA**

I. Introduction

The growing usage of software and hardware in our everyday lives gave birth to paradigms such as Cloud Computing (CC) [1], Edge Computing (EC) [2], and the Internet of Things (IoT) [3]. Although CC enabled companies to cut costs by eliminating the need to buy and maintain their own data centers, the need to fulfill requirements such as low latency or mobility support arose. Therefore, the paradigm of EC was developed to move compute resources to the edge of the network [2]. With IoT, software and hardware resources are distributed even further, e.g., in smart home devices, factories, or even cars. In IoT scenarios, a heterogeneous landscape of sensors and actuators is installed to monitor and actuate within environments, respectively. The data generated by these components must often be aggregated on available devices or edge clouds to realize low latency [2]. For long-term analyses, the data is then stored and processed in the cloud.

However, deploying such applications that consist of application components hosted in different, heterogeneous environments is a highly complex challenge - especially as a manual deployment is not possible because it requires immense technical expertise and it is too time-consuming, error-prone, and costly [4]. Additionally, the deployment system itself must be able to cope not only with different environments, but also with application components and available resources. Therefore, application deployments must be automated.

For automating the deployment of applications several technologies have been developed in recent years [5], [6]. All of them use different execution models that refer to how the orchestration of a deployment is realized [5]: In the centralized execution model, a manager performs all tasks to deploy components by pushing its commands to resources, e.g., by sending request messages to an API or executing scripts via Secure Shell protocol. In contrast, in the de-centralized execution model all participating resources, such as virtual machines and devices have to install agents that poll their commands from a central manager.

However, in IoT scenarios, such as Smart Home and Industry 4.0, the involved private networks of smart homes or factories are protected by firewalls that prevent inbound communication and thus the centralized orchestration model is not sufficient, since API or SSH calls cannot pass the firewalls [7]. Besides, some of the participating devices (e.g. Raspberry Pi or Arduino) in such scenarios are only equipped with minimal computing capacity and hundreds of these devices types can be involved. Thus, installing agents on each device that request commands from a central manager in order to avoid inbound communication is costly and for some devices even not possible due to restricted computing and storage capabilities. As a result, neither a fully centralized nor a fully de-centralized model employing agents is sufficient.

In this paper, we present an approach to tackle these issues, by enabling to determine the location of a participating resource and to execute operations locally in its respective environment, without the need of network access from an central perspective or the need for installing an agent on each managed resource. The basis for this approach is a *temporary orchestrator* coordinating the sequence of deployment tasks for the entire application. The actual execution of a task is performed either by the temporary orchestrator itself or a *temporary executor* which is running in a participating environment in the fashion of a manager or agent to manage local resources, e.g., virtual machines or devices. We implemented our approach based on the TOSCA standard [8] and demonstrate its feasibility with a smart home scenario. Moreover, we made a comparison with widely used technologies.

The remainder of the paper is as follows: Section II describes fundamentals and a motivating scenario. We present our approach in Section III and describe the prototypical implementation and comparison in Section IV. The related work is discussed in Section V and we conclude in Section VI.

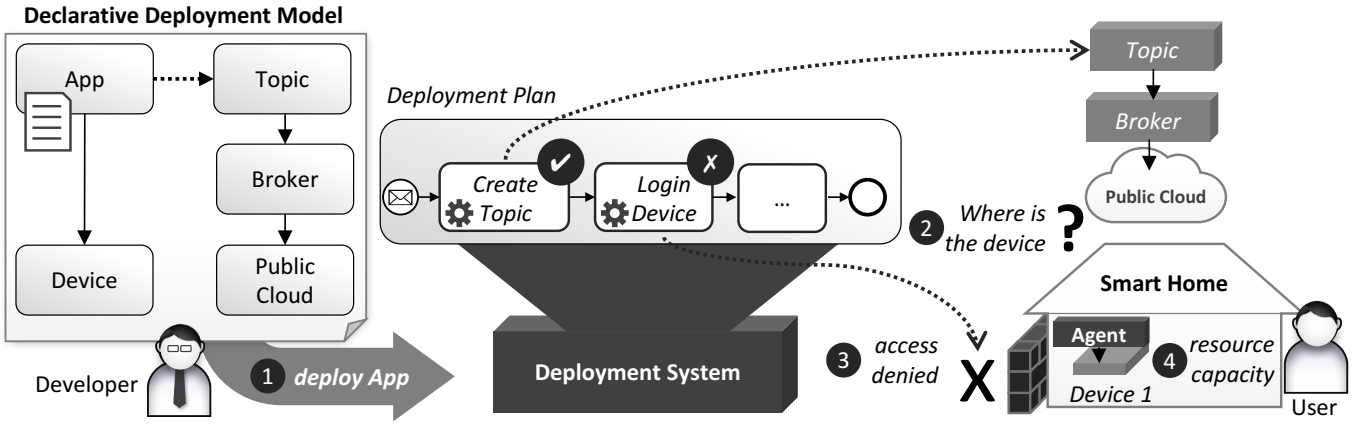


Fig. 1. Distributed application deployment with a (de-)centralized deployment system causes selection and access problems.

II. Motivating Scenario and Fundamentals

In the following, we introduce *declarative deployment models* [9] as the basis for our approach and a motivating scenario showing challenges of distributed deployments in restricted and heterogeneous environments.

A. Declarative Deployment Models

A declarative deployment model [9] (see also Breitenbücher [10] and Saatkamp et al. [11]) describes the structure of an application to be deployed (e.g. see left in Figure 1). It specifies the application's *components* and their *relations* among each other [12]. Components can be infrastructure components, such as IoT devices (e.g. a Raspberry Pi) or virtual machines (e.g. on AWS), middleware components (e.g. a Messaging Broker) or application-specific components (e.g. a Python app). The relations specify the relationship between two components, e.g., that a Broker shall run on AWS. To specify the semantics of components and relations, these elements have an associated *component type* and *relation type*, respectively. A device component, e.g., can be of type Raspberry Pi or Arduino. For relations, types such as *hostedOn* or *connectsTo* can be defined, where the first indicates that a component must be hosted on another, or a component has to connect to another component via communication channels. To enable the configuration of components they expose so-called *properties* that allow application developers to set configuration parameters, e.g., as key-value pairs, such as, the port of a message broker or the ID of a device. Additionally, a component type can expose *deployment operations* that allow a deployment system to create instances of components by invoking these, e.g., to create a virtual machine instance by a *createVM* operation of a cloud or creating a topic by using a *Create Topic* operation of the Broker component in Figure 1. In addition, each deployment operation is associated with a (software) artifact that implements the operations functionality in a reusable manner to be executed in proper execution environments [12], e.g., as a Java-based web service on an application server or a shell script on a virtual machine.

B. Motivating Scenario

Figure 1 depicts our motivating scenario illustrating problems that occur when deploying distributed applications across environments restricting inbound communication. An application developer wants to deploy an application where its components have to be deployed in heterogeneous environments. This is done by a deployment system by giving it a declarative deployment model that specifies the structure of the application (see left in Figure 1). The model consists of the needed components, their exposed deployment operations, their relations, and software artifacts that implement components and deployment operations. The deployment system processes the given declarative deployment model and generates and executes a sequence of deployment tasks, i.e., a deployment plan (see middle in Figure 1) to start and configure instances of the modeled components and relations (for details see [13]). In the example in Figure 1, a new topic has to be created at the Broker by executing the *Create Topic* operation. The operation is implemented as a web service which is deployed by the deployment system in a runtime for executing the operation. If the task is executed the web service is called with the respective parameters, e.g., the URL and the topic name, to send an API call to the Broker. However, the App component that sends data to the Broker from the device is not deployable on the device because of the Smart Home firewall preventing inbound communication (see 3 in Figure 1). This can be overcome by installing agents on the resources to poll commands from a central manager, but is not sufficient for our IoT scenario that uses a hardware-constrained device (see 4 in Figure 1). Moreover, if the infrastructure component does not have a publicly accessible address, the deployment system cannot even connect to the device (see 2 in Figure 1). Hence, these challenges arise while executing the deployment plan: (i) the deployment system has to determine where the resources are located, e.g., in our case the device, (ii) it requires access to these resources which are protected by security mechanisms, e.g., in our case firewalls, and additionally (iii) it has to be able to manage heterogeneous resources from virtual machines to hardware-constrained IoT devices.

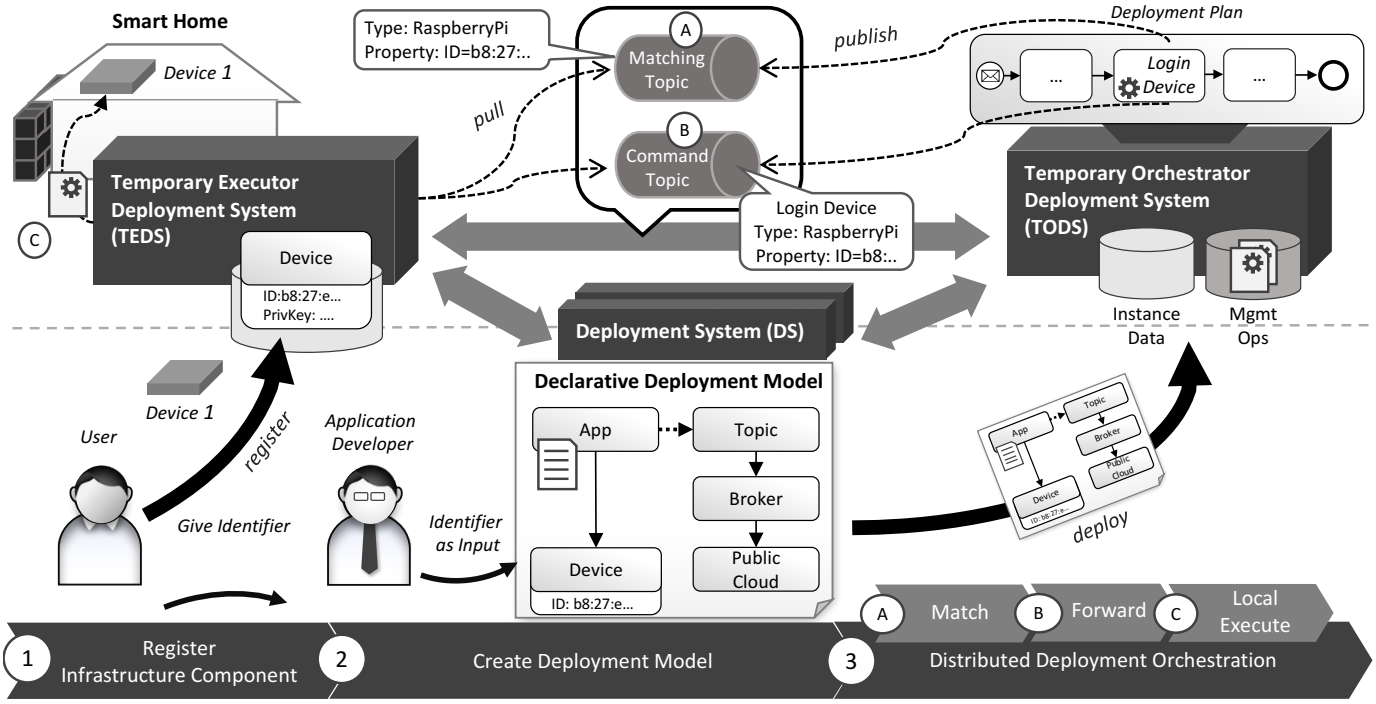


Fig. 2. Overview approach for a distributed deployment across closed networks.

III. Distributed Deployment Approach

To deal with the mentioned challenges, we present an approach to enable the deployment of distributed applications across network boundaries and without the need to install agents on each managed component, starting with an overview in Section III-A. In Section III-B, we introduce the matching of instance data to identify the location of infrastructure components used for the deployment. In Section III-C, we present a system architecture supporting our proposed approach.

A. Overview

Figure 2 gives an overview of our *deployment approach*. It covers all steps from the registration of new infrastructure components available in a certain network to the automated distributed deployment of an application. Since middleware and application-specific components can only be deployed on known infrastructure components, they have to be registered at the respective *deployment system (DS)* in the first step (step 1). One DS manages all components accessible in a certain network. Such components can be devices, such as Raspberry Pis or MICAs, hypervisors, such as a vSphere, or public clouds, such as AWS or Azure. For the registration, the component type and its specific properties, e.g., MAC- or IP-address and credentials, are required. The *user* on the one hand is responsible for the components in a certain network, e.g., in the smart home. The *application developer* on the other hand develops applications that consist of components distributed across several networks, where parts of the application must be deployed directly on the devices, e.g., to collect and aggregate data, and other parts are deployed on private or public clouds. The developer creates a declarative deployment model for the automated provisioning (step 2). In our motivating scenario in Figure 2, a Python-App shall be deployed on a Raspberry Pi with the *ID* containing its MAC-address. The Python app publishes data to a Topic created on a MQTT-Broker running on a public cloud, e.g., to send measured temperature data. Since the device is located in a network that restricts inbound communication, a DS which is located in the same network is required for the application deployment in step three. The DS which starts the deployment takes the role of the *temporary orchestrator DS* (TODS) and starts a deployment plan for instantiating the modeled application. The execution order can be derived from the deployment model, e.g., as in [13]. For each deployment operation, first, the location of its execution have to be determined (step A) and, second, the deployment artifact implementing the operation have to be forwarded to the responsible DS which will take the role of the *temporary executor DS* (TEDS, see step B) before it can be locally executed (step C). For determining the execution location in step A, the TODS identifies the infrastructure components in the deployment model and tries to match these to its registered instances using the *model-instance matching* concept, described in detail in Section III-B. If no matching instance can be locally detected, the component information is forwarded to the other DSs using the Matching Topic to determine a DS which has a matching instance registered. In the example in Figure 2, the device component in the deployment model matches to the instance registered in the DS in the smart home on the left. After successful matching, the command to execute the deployment operation is forwarded to

the respective TEDS using the Command Topic (step B). The TEDS polls the deployment artifact and executes it depending on the artifact (step C): If it is, for example a WAR, it is deployed and executed in the deployment operation runtime (see Section III-C) of the TEDS. If it is a script, e.g., a shell-script, it is loaded and executed directly on the device.

Our approach enables to deploy and manage distributed applications across protected networks that restrict inbound communication by dynamically assigned roles of TODS and TEDSs that manage the respective restricted environments. Based on that, no significant altering of security mechanisms is needed, as there is only the need for a communication channel between the participating DSs. Additionally, there is no need to install agents on each single resource and thus, the pre-installation effort of agents is reduced and also devices with limited computing and storage capacity can be managed.

B. Model-Instance Matching

To achieve a distributed execution of deployment operations an approach to determine *where* to execute *which* operation is needed. In this section, we provide a *location detection algorithm* to determine the execution location of deployment operations required for step 3A of our approach. In Algorithm 1 its two procedures are shown: The *location detection* procedure gets the application component c_m for which the deployment operation is executed. For example, the component that exposes the *Create Topic* deployment operation in Figure 1 is the *Broker* component in the deployment model. In addition, the application deployment model d, and the set of all components C_i in the instance data database of the respective TODS or TEDS is given as input. The algorithm is executed in two steps: First, the component at the bottom of the application stack of component c_m, which we call the *infrastructure component* is determined, and then the infrastructure component is checked against all infrastructure components in the instance data database.

Let C_d be the set of all components in d. Then, we have to find the component in this set that is connected to c_m by a path of hostedOn relations and which has no outgoing hostedOn relations, which means it is on the bottom of the stack of c_m (lines 2-3). Therefore, we define $successors^{+\text{hostedOn}}(c_i)$ as the subset of components in C_d, which are direct or transitive hostedOn successors of a component $c_i \in C_d$. Thus, the infrastructure component needs to be a direct or transitive successor of c_m and the set of its own direct and transitive successors needs to be empty. This condition could be fulfilled for multiple components, if a component is allowed to have multiple outgoing hostedOn relations. However, we restrict our approach to components without multiple outgoing hostedOn relations, as the semantic of a component hosted on more than one other component is unclear, leading to a unique infrastructure component for each component.

Afterwards, the infrastructure component is matched against all components in the instance data by comparing its component type and properties (lines 12-15). A matching is found, if a component in the instance data has the same type and a subset of matching properties. We allow to match against a subset to enable matching only against the unique identifier (e.g. MAC address) of an infrastructure component. If a matching is found by a DS, it is responsible for executing the deployment operation on component c_m (line 5). Otherwise the execution of the deployment operation can be delegated to another DS, which has matching instance data (lines 7-8). Each participating DS tries to find a matching instance by executing the instance data matching algorithm. Once a responsible DS has been identified, the deployment operation is delegated.

Algorithm 1 Location Detection Algorithm

```
1:  procedure LocationDetection(c_m, d, C_i)
2:      c_infra := c_i ∈ C_d : (c_infra ∈ successors^+hostedOn(c_m)
3:              ∧ successors^+hostedOn(c_infra) = ∅)
4:      if InstanceDataMatching(c_infra, C_i) then
5:          return own location
6:      else
7:          publish message c_infra on matching channel
8:          receive & return location from matching channel
9:      end if
10: end procedure
11: procedure InstanceDataMatching(c_infra, C_i)
12:     if (∃c_i ∈ C_i :(properties(c_infra) ⊆ properties(c_i)
13:      ∧ type(c_infra) = type(c_i))) then
14:         return true
15:     end if
16:     return false
17: end procedure
```

C. Distributed Deployment Operation Execution

Figure 3 shows the system architecture of our distributed deployment approach. The system consists out of multiple cooperating DS. They manage the components, devices, and applications in their own network. If an application is instantiated by the system, one of the DSs has the role of the TODS responsible for executing and coordinating the overall deployment plan. Only the TODS is in charge of running the deployment plan of the application. The other distributed DSs are responsible for the execution of deployment operations, which are called by the deployment plan, but cannot be performed at the TODS, and therefore, have the role of TEDSs. However, the role assignment is not static and every DS of the distributed deployment system can play the role as TODS or TEDS, as long as, the TODS are accessible by all other participating TEDSs.

In addition to the *plan runtime*, which is used to execute deployment plans, a DS consists out of a *management bus*, a *deployment operation runtime*, and two databases. The *artifact database* stores all deployment artifacts which are contained in uploaded deployment models, like scripts or WAR files. Depending on the artifact type they are executed directly on the infrastructure component or need an additional runtime inside the deployment system. This runtime is provided by the deployment operation runtime, an extensible component

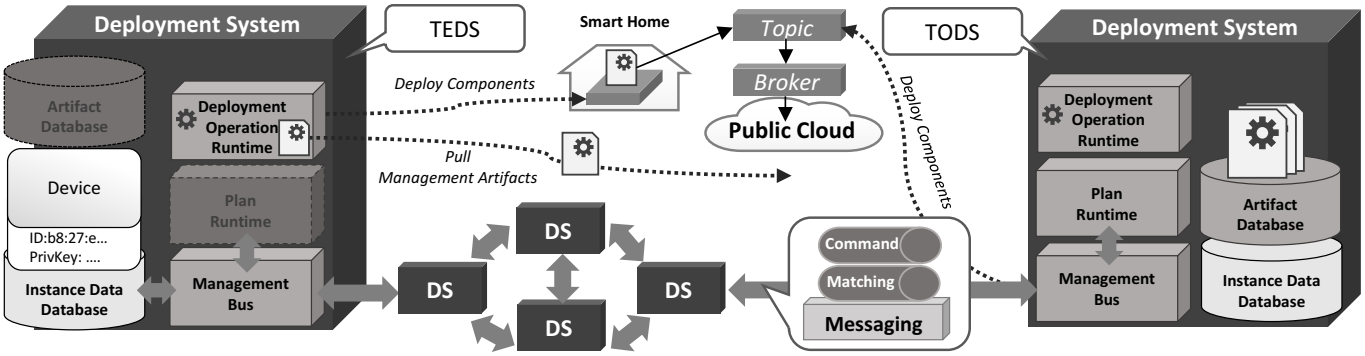


Fig. 3. System architecture of the proposed distributed deployment system showing the communcation, components and roles of the participants.

which is able to execute deployment operation, e.g., implemented as web services. The management bus provides the communication capabilities for all components inside a DS as well as between other DSs. Finally, the *instance data database* is used to store instance information, e.g., identifiers and current properties of running infrastructure and application components, which is used by the location detection algorithm to identify such instances and, therefore, the location where to execute deployment operations (see Section III-B).

When the deployment plan needs to execute a deployment operation, it calls the management bus of the TODS with the operation name and the needed parameters. After receiving the operation call, the management bus runs the location detection algorithm. Therefore, it first performs instance data matching on local component instances to determine whether the deployment operation can be executed in its local environment. If this is the case, the operation call can be forwarded to the local deployment system. After successful execution, it returns to the plan runtime and the next task in the deployment plan is triggered. However, if the local instance data matching of a TODS is not successful, the management bus publishes a *matching request* containing the component type and the properties of the infrastructure component to be resolved by the location detection procedure. Exchanging these requests is done via topics on a Messaging middleware (see Messaging component in Figure 3). The messaging middleware should support the publish-subscribe pattern to decouple the DSs, which in return eases the extensibility of the system itself. Each subscribed DS receives the matching request and performs instance data matching locally and returns a response if it finds a matching component. After receiving the response the TODS knows, a DS is able to execute the current deployment operation and publishes a *command request* with the deployment operation and the location of the component's artifacts in the artifact database. The location is used by the executing deployment operation runtime to poll the deployment artifact. As the communication is started from inside the protected network only the polling of operations and artifacts must be achieved. In case the TODS does not receive a response from one of the TEDS within time, the operation cannot be executed and the deployment aborts.

IV. Prototype and Comparison

In this section we describe the implementation of our approach. It is based on the OASIS standard *Topology and Orchestration Specification for Cloud Applications* (TOSCA) [8] and the *OpenTOSCA ecosystem* [14].

TOSCA enables to model cloud applications in a standardized, vendor-independent and portable manner and is also applicable for modeling IoT applications [15]. Applications are modelled as so called *Service Templates*, which contain *Topology Templates* describing their application topology. A Topology Template maps to the application structure of our application deployment models and consists of *Node Templates* corresponding to application components and *Relationship Templates* representing their relationships. Node Templates and Relationship Templates are instances of Node- and Relationship Types which specify their semantics, and therefore, map to component and relation types. Besides the description of *Interfaces*, which correspond to deployment operations, TOSCA also enables to define *Properties*, which can be used to configure a template when it is instantiated. Deployment operations of types are implemented by so called *Implementation Artifacts*, such as scripts or web services.

Our prototypical implementation extends the *OpenTOSCA container*, a TOSCA-compliant runtime that is able to process TOSCA models and create TOSCA Plans in BPEL [16] enabling, e.g., to start and stop applications. To show the feasibility of the presented concepts, the extension enables to setup several cooperating OpenTOSCA containers and to execute BPEL plans on TODS, invoking the needed deployment operations on itself or other TEDS. The communication between the OpenTOSCA container instances is realized over the management bus and the *Message Queuing Telemetry Transport* (MQTT) protocol. Additionally, we implemented the location detection algorithm in the systems management bus. If a local execution is not possible, the management bus is able to publish matching and command messages via MQTT to delegate the execution to the other available OpenTOSCA containers. For validating the feasibility of our approach, we realized the deployment model as presented in Figure 2.

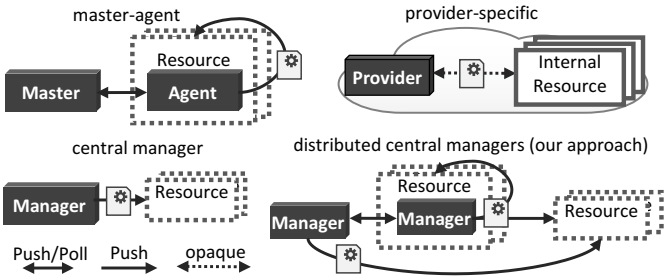


Fig. 4. Overview of execution model and communication pattern.

TABLE I
COMPARISON OF DEPLOYMENT SYSTEM ARCHITECTURES

Exec. Model/Comm. Pattern	Technologies
master-agent/poll	Puppet, Chef,Juju, CFEngine, Cloudify
central manager/push	Ansible, HEAT, Terraform, SaltStack, Cloudify
master-agent/push	Kubernetes, Docker Compose, SaltStack
provider-specific/-	CloudFormation, Azure Resource Manager
dist. central managers/push & poll	Our approach

A. Comparison with widely used technologies

We conducted a comparison based on the different execution models and communication patterns (see Figure 4) between different deployment technologies and our own approach (see detailed results in Table I). The comparison is based on previous work [5] [17]. In upper left corner of Figure 4 the *master-agent* execution model executes deployment operations by sending commands to registered agents running on the used resources, which executes the respective software artifacts to achieve the desired deployment tasks. The central manager execution model directly executes the software artifacts on the used (external) resources (see lower left corner in Figure 4). In the provider-specific execution model the provider executes software artifacts on the internal resources of the provider (see upper right corner in Figure 4). In our approach, we use the so-called distributed central manager (see lower right corner in Figure 4) were a set of central managers are either able to directly communicate with the used resources or passing the deployment operation to another manager to execute it.

Most of the compared technologies use a master-agent execution model (see in Table I), while other technologies either employed a centralized manager execution model (Note that some can employ multiple models, e.g., SaltStack). The main problem with the approaches are either the connectivity through firewalls (in case of push communication) or the restricted capacities of managed resources to run an agent (in case of master-agent execution models). Our approach overcomes these problems by employing a network of multiple managers that can take the role of a central manager (temporary orchestrator/TODS) and other the role of agents/local managers (temporary executor/TEDS) during runtime dynamically. This enables to use the system flexibly, such as, using only a single manager to enable centralized management, or registering other managers behind firewalls to enable the deployment through firewalls without agents.

Another aspect of our comparison was the communication pattern of the technologies, i.e, if a technology uses either the push or poll pattern. Technologies that employ a central-manager execution model also use a push communication pattern, while those which use a master-agent execution model use a poll-based communication pattern. The exception here are Docker Compose and Kubernetes which use a master-agent execution model and push commands to agents. Additionally, some systems (e.g. Cloudify) enable to push and poll the commands, i.e., push is being used when there is only a single central manager and poll when agents are in use. Our approach mixes push and poll as well, as each manager publishes command messages to potential participants which poll these message and depending whether they are responsible for the needed resources push deployment commands to these.

V. RELATED WORK

Arcangeli et al. [6] present a survey on research works to enable the automated deployment of distributed applications. From the perspective of accessing the infrastructure behind firewalls and installing agents, most approaches investigated by Arcangeli et al. rely on bootstrapping the infrastructure with their specific agents, decreasing their usage on hardware constrained infrastructure components or the usage in secured networks when the agents receive their commands via push pattern. Our approach enables fully distributed execution while maintaining the ability to either use a manager to control multiple infrastructure components or employ a manager as an agent on each resource itself, with the drawback that infrastructure components have to be registered at the responsible manager. This task is currently executed manually, but can be automated, e.g, using discovery standards such as UPnP in case of IoT devices. For distributing applications across multiple environments [11] presents an approach to annotate components in deployment models with target location labels to indicate the desired distribution. However, the matching is based on model adaptations instead of model-instance matching. Several approaches deal with the placement of services [18], [19] and their deployment distributed across cloud, edge, and devices [20]–[24]. Wen et al. [18] present a fog orchestrator for the optimal placement of IoT applications. They focus on the planning aspect rather then the distributed deployment across network boundaries. Also Brogi et al. [19] focus on the placement decision. Skarlat et al. [20], [21] introduce FrogFrame, a framework for IoT service placement, deployment, and execution in the fog. The fog controller serves as central control and orchestration node that forwards commands to fog nodes that manage several fog cells. However, this distributed deployment system requires management nodes on each manageable device (i.e. it follows a fixed master-agent architecture) and can only handle Docker containers as artifacts. The DIANE framework developed by Vögler et a. [23] enables

the flexible deployment of IoT applications in a declarative manner. However, the framework relies on agents on each of the gateways which will be used as infrastructure components, which in turn restricts the usage of the system for embedded low computing and storage devices.

VI. Conclusion

In this paper, we presented problems occurring when trying to enable the distributed deployment over heterogeneous environments. One problem is the access and location of environments as security mechanisms may prevent proper access for deployment. Another problem is the heterogeneity of applications using different types of resources from virtual machines hosted on the cloud to IoT devices hosted in Smart Homes. Our distributed deployment approach enables (i) to determine the location of the required resources in the distributed protected environments and (ii) to execute the deployment operations locally to install and configure the application components on the respective resources. In contrast to other approaches, there is no need for pre-installed agents on each resource easing the integration of larger amounts of infrastructure resources and furthermore, also small devices with less computing power can be managed. One participant in the system takes the role of the temporary orchestrator deployment system (TODS), orchestrating the deployment of the overall application by involving other participants, having the role of the temporary executor deployment system (TEDS), which are able to execute the needed deployment tasks in the respective environments.

In the future we want to further evaluate our approach in an Industry 4.0 scenario where multiple production machines and robots have to be managed from an edge cloud. One of the problems within the mentioned scenario is that different partners that must start resources on their private infrastructure use internal, proprietary APIs which they do not want to share in a global application model. Therefore, we plan to adapt our approach from an orchestrated to a choreographed execution, by splitting the deployment plans according to the locality of environments and infrastructure.

Acknowledgment

This work was partially funded by the DFG project DiStOPT (252975529) and the BMWi project *Industrial Communication for Factories* – IC4F (01MA17008G).

References

[1] B. P. Rimal, E. Choi, and I. Lumb, "A taxonomy and survey of cloud computing systems," in *2009 Fifth International Joint Conference on INC, IMS and IDC*, Aug 2009, pp. 44–51.

[2] R. Roman, J. Lopez, and M. Mambo, "Mobile edge computing, fog et al.: A survey and analysis of security threats and challenges," *Future Generation Computer Systems*, vol. 78, pp. 680 – 698, 2018.

[3] L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey," *Computer Networks*, vol. 54, no. 15, pp. 2787 – 2805, 2010.

[4] U. Breitenbücher, K. Képes, F. Leymann, and M. Wurster, "Declarative vs. Imperative: How to Model the Automated Deployment of IoT Applications?" in *Proceedings of the 11th Advanced Summer School on Service Oriented Computing*. IBM Research Division, 2017, pp. 18–27.

[5] D. Weerasiri, M. C. Barukh, B. Benatallah, Q. Z. Sheng, and R. Ranjan, "A Taxonomy and Survey of Cloud Resource Orchestration Techniques," *ACM Computer Surveys*, vol. 50, no. 2, pp. 26:1–26:41, May 2017.

[6] J.-P. Arcangeli, R. Boujbel, and S. Leriche, "Automatic deployment of distributed software systems: Definitions and state of the art," *Journal of Systems and Software*, vol. 103, pp. 198 – 218, 2015.

[7] A. Luoto and K. Systä, "Fighting network restrictions of request-response pattern with MQTT," *IET Software*, vol. 12, no. 5, pp. 410–417, 2018.

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

[9] C. Endres, U. Breitenbücher, M. Falkenthal, O. Kopp, F. Leymann, and J. Wettinger, "Declarative vs. Imperative: Two Modeling Patterns for the Automated Deployment of Applications," in *Proceedings of the 9th International Conference on Pervasive Patterns and Applications*. Xpert Publishing Services (XPS), 2017, pp. 22–27.

[10] U. Breitenbücher, "Eine musterbasierte Methode zur Automatisierung des Anwendungsmanagements," Dissertation, Universität Stuttgart, Fakultaet Informatik, Elektrotechnik und Informationstechnik, 2016.

[11] K. Saatkamp, U. Breitenbücher, O. Kopp, and F. Leymann, "Topology splitting and matching for multi-cloud deployments," in *Proceedings of the 7th International Conference on Cloud Computing and Services Science (CLOSER 2017)*. SciTePress, Apr. 2017, pp. 247–258.

[12] M. Wurster, U. Breitenbücher, M. Falkenthal, C. Krieger, F. Leymann, K. Saatkamp, and J. Soldani, "The Essential Deployment Metamodel: A Systematic Review of Deployment Automation Technologies," *arXiv:1905.07314 [cs.SE], To appear: Software-Intensive Cyber-Physical Systems (SICS)*, 2019.

[13] U. Breitenbücher, T. Binz, K. Képes, O. Kopp, F. Leymann, and J. Wettinger, "Combining Declarative and Imperative Cloud Application Provisioning based on TOSCA," in *International Conference on Cloud Engineering (IC2E 2014)*. IEEE, Mar. 2014, pp. 87–96.

[14] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, and S. Wagner, "OpenTOSCA - A Runtime for TOSCA-based Cloud Applications," in *Proceedings of the 11th International Conference on Service-Oriented Computing (ICSOC 2013)*. Springer, Dec. 2013, pp. 692–695.

[15] A. C. Franco da Silva, U. Breitenbücher, P. Hirmer, K. Képes, O. Kopp, F. Leymann, B. Mitschang, and R. Steinke, "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, Apr. 2017.

[16] OASIS, *Web Services Business Process Execution Language (WS-BPEL) Version 2.0*, Organization for the Advancement of Structured Information Standards (OASIS), 2007.

[17] T. Delaet, W. Joosen, and B. Vanbrabant, "A Survey of System Configuration Tools," in *Proceedings of the 24th International Conference on Large Installation System Administration (LISA 2010)*. USENIX, Nov. 2010.

[18] Z. Wen, R. Yang, P. Garraghan, T. Lin, J. Xu, and M. Rovatsos, "Fog orchestration for internet of things services," *IEEE Internet Computing*, vol. 21, no. 2, pp. 16–24, Mar 2017.

[19] A. Brogi, S. Forti, and A. Ibrahim, "How to best deploy Your Fog applications, probably," in *2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC)*, May 2017, pp. 105–114.

[20] O. Skarlat, S. Schulte, M. Borkowski, and P. Leitner, "Resource Provisioning for IoT Services in the Fog," *2016 IEEE 9th Conference on Service-Oriented Computing and Applications (SOCA)*, pp. 32–39, 2016.

[21] O. Skarlat, V. Karagiannis, T. Rausch, K. Bachmann, and S. Schulte, "A framework for optimization, service placement, and runtime operation in the fog," in *2018 IEEE/ACM 11th International Conference on Utility and Cloud Computing (UCC)*, Dec 2018, pp. 164–173.

[22] C. Wöbker, A. Seitz, H. Mueller, and B. Bruegge, "Fogernetes: Deployment and management of fog computing applications," in *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*, April 2018, pp. 1–7.

[23] M. Vögler, J. M. Schleicher, C. Inzinger, and S. Dustdar, "DIANE - Dynamic IoT Application Deployment," in *2015 IEEE International Conference on Mobile Services*, June 2015, pp. 298–305.

[24] R. Jain and S. Tata, "Cloud to Edge: Distributed Deployment of Process-Aware IoT Applications," in *2017 IEEE International Conference on Edge Computing (EDGE)*, 2017, pp. 182–189.