2015 IEEE International Conference on Mobile Services

DIANE – Dynamic IoT Application Deployment

Michael Vögler, Johannes M. Schleicher, Christian Inzinger, and Schahram Dustdar
Distributed Systems Group, Vienna University of Technology, 1040 Vienna, Austria
{lastname}@dsg.tuwien.ac.at

***Abstract*—Applications in the Internet of Things (IoT) domain need to manage and integrate huge amounts of heterogenous devices. Usually these devices are treated as external dependencies residing at the edge of the infrastructure mainly transmitting sensed data or reacting to their environment. Recently however, a fundamental shift in the basic nature of these devices is taking place. More and more IoT devices emerge that are not simple sensors or transmitters, but provide limited execution environments. This opens up a huge opportunity to utilize these previously untapped processing power in order to offload custom application logic directly to these edge devices. To effectively exploit this new type of device the design of IoT applications needs to change to also consider devices that are deployed on the edge of the infrastructure. The deployment of parts of the application's business logic on the device will not only increase the overall robustness of the application, but can also reduce communication overhead. To allow for flexible provisioning of applications whose deployment topology evolves over time, a clear separation of independently executable application components is needed. In this paper, we present DIANE, a framework for the dynamic generation of optimized deployment topologies for IoT cloud applications that are tailored to the currently available physical infrastructure. Based on a declarative, constraint-based model of the desired application deployment, our approach enables flexible provisioning of application components on edge devices deployed in the field. DIANE supports different IoT application topologies and we show that our solution elastically provisions application deployment topologies using a cloud-based testbed.**

I. Introduction

Current Internet of Things (IoT) applications need to manage and integrate an ever-increasing number of heterogeneous devices to sense and manipulate their environment. Increasingly, such devices do not only act as simple sensors or actors, but also provide constrained execution environments with limited processing, memory, and storage capabilities. In the context of our work, we refer to such devices as *IoT gateways*. By exploiting the functionality offered by IoT gateways, applications can offload parts of their business logic to the edge of the infrastructure to reduce communication overhead and increase application robustness [1]. Explicitly considering edge devices in IoT application design is especially important for applications deployed on cloud computing [2] infrastructure. The cloud provides access to virtually unlimited resources that can be programmatically provisioned with a pay-as-you-go pricing model, enabling applications to elastically adjust their deployment topology to match their current resource usage and according cost to the current request load.

IoT cloud applications therefore must be designed to cope with issues arising from geographic distribution of edge devices, network latency and outages, as well as regulatory requirements, in addition to the traditional design considerations for cloud applications. Hence, edge devices must be treated as first-class citizens when designing IoT cloud applications and the traditional notion of resource elasticity [3] in cloud computing needs to be extended to include such heterogeneous IoT gateways deployed at the infrastructure edge, enabling interaction with the physical world. To allow for the flexible provisioning of applications whose deployment topology changes over time due to components being offloaded to IoT gateways, applications need to be composed of clearly separated components that can be independently deployed. The microservices architecture [4] recently emerged as a pragmatic implementation of the service-oriented architecture paradigm and provides a natural fit for creating such IoT cloud applications. We argue that future large-scale IoT systems will use this architectural style to cope with their inherent complexities and allow for seamless adaptation of their deployment topologies. Uptake of the microservice architecture will furthermore allow for the creation of IoT application markets (e.g., [5]) for practitioners to purchase and sell domain-specific application components.

IoT gateways can be considered an extension of the available cloud infrastructure, but their constrained execution environment and the fact that they are deployed at customer premises to integrate and connect to local sensors and actors requires special consideration when provisioning application components on IoT gateways. By carefully deciding when to deploy certain application components on gateways or cloud infrastructure, IoT cloud applications can effectively manage the inherent cost-benefit trade-off of using edge infrastructure, leveraging cheap communication within edge infrastructure while minimizing expensive (and possibly slow or unreliable) communication to the cloud, while also considering processing, memory, and storage capabilities of available IoT gateways. It is important to note that changes in application deployment topologies will not only be triggered whenever a new application needs to be deployed, but can also be caused by environmental changes, such as changing customer request patterns, changes in the physical infrastructure at the edge (e.g., adding/removing sensors or gateways), or evolutionary changes in application business logic throughout its lifecycle.

In this paper, we present DIANE, a framework for dynamically generating optimized deployment topologies for IoT cloud applications tailored to the available physical infrastructure. Using a declarative, constraint-based model of the desired application deployment, our approach enables flexible provisioning of application components on both, cloud infrastructure, as well as IoT gateways deployed in the field.

The remainder of this paper is structured as follows: In Section II we introduce the DIANE framework to address the identified problems in dynamically creating application deployment topologies for large-scale IoT cloud systems. We provide a detailed evaluation in Section III, discuss relevant

978-1-4673-7284-8/15 $31.00 © 2015 IEEE
DOI 10.1109/MS.2015.49

related research in Section IV, followed by a conclusion and an outlook on future research in Section V.

II. Approach

To address the previously defined requirements, we present DIANE, a framework for the dynamic generation of deployment topologies for IoT applications and application components, and the respective provisioning of these deployment topologies on edge devices in large-scale IoT deployments. The overall architecture of our approach is depicted in Figure 1 and consists of the following top-level components: (i) DIANE, and (ii) LEONORE. In the following, we describe these components in more detail and discuss the design and implementation of IoT applications.

A. IoT Application

In order to allow dynamic generation of deployment topologies for IoT applications in our framework, the design and implementation of such applications have to follow the micro services architecture approach [4]. This design approach enables developers to build flexible and scalable applications, whose components can be independently evolved and managed. Each component of an application is self-contained, can be run separately and uses loosely coupled communication for interacting with other components. Following this application design approach we are using the MADCAT [6] methodology to describe the overall application and it's components. In essence, MADCAT enables the structured creation of applications by addressing the complete application lifecycle, from architectural design to concrete deployment topologies provisioned and executed on actual infrastructure. For our approach we focus on Technical Units (TUs) and Deployment Units (DUs) to describe applications and their components.

1) Technical Unit (TU): Technical Units are used to describe application components by using both the abstract architectural concerns and concrete deployment artifacts to capture technology decisions, which depend on the actual implementation. Since there are usually multiple possible TUs available to realize a specific application component, MADCAT employs decision trees to assist developers of such applications in realizing TUs. An example of a TU can be seen in Listing 1. We are using the JSON-LD[1] format to store and transfer MADCAT units.

Listing 1: Technical Unit

```
{
  "@context": "http://madcat.dsg.tuwien.ac.at/",
  "@type": "TechnicalUnit",
  "name": "BMS/Unit",
  "artifact-uri": "...",
  "language": "java",
  "build": {
    "assembly": {"file": "unit.jar"},
    "steps": [{"step": 1, "tool": "maven", "cmd": "
        mvn clean install"}]
  },
  "execute": [{"step": 1, "tool": "java", "cmd": "
      java -jar @build.assembly.file"}],
  "configuration": [{"key": "broker.url", "value": "
      @MGT.broker.url"}],
  "dependencies": [{"name": "MGT", "technicalUnit":
      {"name": "BMS/Management"}}],
  "constraints": {"type": "...","framework": "Spring
       Boot","runtime": "JRE 1.7","memory": "..."}
}
```

[1] http://json-ld.org

Each TU starts with a `context` that specifies the structure of the information and a specific `type`. The `name` uniquely identifies the TU and should refer to the application name and the specific component the TU describes. The `artifact-uri` defines the repository that stores the application sources and artifacts. The `language` field describes the used programming language and an optional version. For creating a runnable executable `build` specifies an `assembly` that defines the location within the repository and the name of the executable. Furthermore `build` describes a list of `steps` that need to be executed to create the executable. `Execute` defines the necessary steps to run the executable. In addition to the execution steps, `configuration` stores a possible runtime configuration (e.g., environment variables) that is needed for execution. Since some of the configuration items might map to other application components, `dependencies` reference TUs of other application components. Finally, the TU includes relevant `constraints` provided by the developer to help users of the application to decide on a suitable deployment infrastructure.

2) Deployment Unit (DU): For each TU one or more deployment units (DUs) are created by an infrastructure provider or operations manager. A DU describes how the TU can be deployed on concrete infrastructure. In order to create a specific DU the provider uses the information contained in the TU and the knowledge about the owned infrastructure. Listing 2 shows an example DU created for the TU above.

Listing 2: Deployment Unit

```
{
  "@context": "http://madcat.dsg.tuwien.ac.at/",
  "@type": "DeploymentUnit",
  "name": "BMS/Unit",
  "technicalUnits": [{"name": "BMS/Unit"}],
  "constraints": [{
      "hardware": [{"type": "...", "os": "...", "
          capabilities": [{"name": "JRE", "version":
           "1.7"}], "memory": "..."}],
      "software": [{"replication": [{"min": "all"}]}
          ]
  }],
  "steps": [...]
}
```

A DU has a `context`, `type` and `name`. Next, `technicalUnits` reference the TUs that are deployed using this specific DU. Based on the information provided in the TU (e.g., constraints) the infrastructure provider defines `constraints` for `hardware` and `software` that are used to decide on suitable infrastructure resources for executing an application component. Finally, `steps` list the necessary deployment steps.

3) Deployment Instance (DI): Based on the TUs and corresponding DUs it is possible to completely describe an IoT application. Now, to finally provision an application deployment, the framework uses TUs, DUs and concrete infrastructure knowledge to generate deployment instances (DIs).

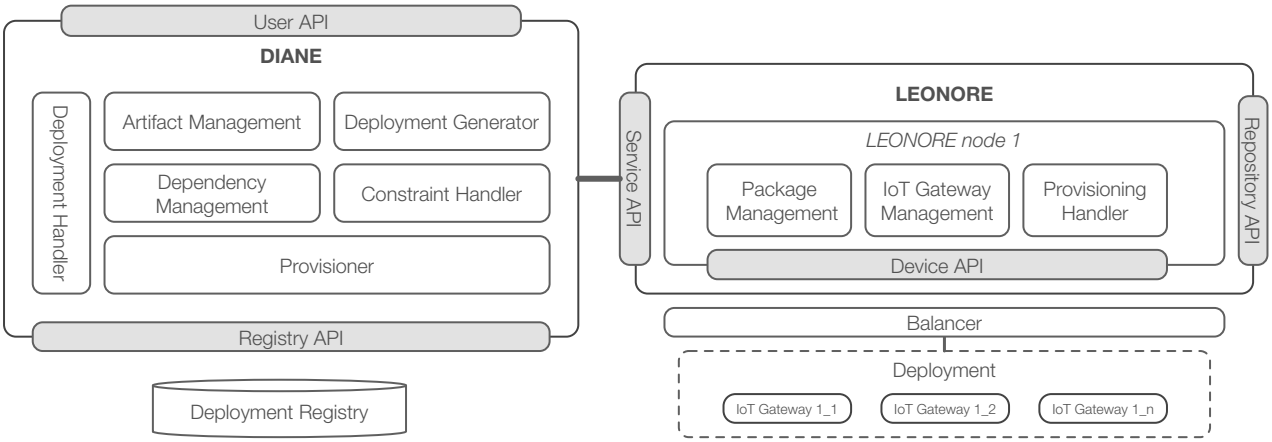


Fig. 1: Framework – Overview

DIs represent concrete deployments on actual machines of the infrastructure, taking into consideration the defined software and hardware constraints. An example of a DI using the DU and TU from above can be seen in Listing 3.

Listing 3: Deployment Instance

```
{
  "@context": "http://madcat.dsg.tuwien.ac.at/",
  "@type": "DeploymentInstance",
  "name": "...",
  "machine": {"id": "...", "ip": "..."},
  "application": {"name": "BMS/Unit", "version": "1.
      0.0", "environment": [{"key": "broker.url", "
      value": "failover:tcp://10.99.0.40:61616"}]
  }
}
```

A DI has a `context`, `type` and `name`. The `machine` field stores data about the concrete machine that is provisioned with an application component. Runtime information, needed for executing the application component, is represented in `application`. It contains the `name` and the `version` of the application component. Runtime configurations required by the component are resolved by the framework and represented in `environment`.

B. DIANE Framework

The enabling framework for generating IoT application deployment topologies and the provisioning of these deployments on edge devices in large-scale IoT deployments is depicted on the left hand side of Figure 1. DIANE is a scalable and flexible cloud-based framework and is developed following the micro services architecture design. In the following, we introduce the main components of DIANE, discuss the integration with LEONORE [1] for provisioning edge devices and discuss the concrete process of generating and provisioning application deployment topologies.

1) Deployment Registry: To keep track of deployments and their relation to TUs and DUs, the framework uses a `Deployment Registry`. The registry stores the units and deployments using a tree structure to represent the relations among them. By keeping track of TUs and DUs the framework can provide application deployment provisioning at a finer granularity. This means that it is possible with DIANE to provision an application deployment topology in one batch, but also provision each deployment separately.

2) Deployment Handler: To provision an IoT application deployment topology with DIANE the user of the framework has to invoke the `User API` and provide the following required information: (i) TUs, (ii) corresponding DUs, and (iii) optional artifacts that are needed by the deployment (e.g., executables) that can not be resolved automatically by the framework like private repositories that are not publicly accessible. Since the focus of our work is on generating and provisioning DIs, a user of the framework is responsible for creating the required MADCAT units and necessary application artifacts. The `Deployment Handler` is responsible for handling user interaction and finally triggers the provisioning of application deployments.

3) Artifact Management: In addition to the MADCAT units, the framework also needs corresponding application artifacts. The `Artifact Management` component receives the artifacts, resolves all references and creates an artifact package that gets transferred to LEONORE. Each package contains an executable, a version, and the commands to start and stop the artifact.

4) Deployment Generator: To generate DIs, the `Deployment Generator` resolves the dependencies among the provided TUs and DUs by using the `Dependency Management` component. The management component returns a tree structure to represent the dependencies among the units. Next, the generator deals with possible deployment constraints that are specified in the DUs by invoking the `Constraint Handler`. The handler returns a list of infrastructure resources that comply with the specified constraints. Before the actual generation of DIs, the generator needs to handle application runtime configurations (e.g., application properties) in the TUs. Therefore, the generator delegates the configuration resolving to the Constraint Handler and receives a temporary configuration. Finally, the generator

creates the actual DIs by mapping the DUs to concrete machines and updating possible links in the temporary configuration that correspond to infrastructure properties (e.g., IP address of a machine).

5) Dependency Management: Since MADCAT units reference each other, the `Dependency Management` is responsible for resolving unit dependencies. For representing the dependencies among the units the management component creates a tree structure. The process of dependency resolution first creates for each TU a new root node. After creating the root nodes it checks if a TU has a reference to another TU and if so creates a new leaf node linking to the respective root node. Next, it checks the provided DUs and adds them to the respective TU node as a leaf. In case a reference cannot be resolved based on the provided units, it queries the `Deployment Registry`. The final product of this process is a tree topology, where each root node represents a TU and the leaves are the corresponding DUs or a reference to another TU.

6) Constraint Handler: In order to provide a suitable deployment of application components on machines, DUs provide deployment constraints. In our approach we distinguish hardware and software constraints. Hardware constraints deal with actual infrastructure constraints e.g., operating system or the installed capabilities of a machine. Software constraints define requirements that correspond to the application component and its deployment e.g., should this component be replicated and if so on how many machines. To provide a list of suitable machines the handler retrieves a list of all known machines and their corresponding metadata from LEONORE. Then, based on the defined constraints in the DU, it filters out the ones that do not fit or not needed in case software constraints only demand for a certain amount of machines.

7) Provisioner: For actually provisioning the final DIs the `Provisioner` component is used. The component receives the generated DIs and the topology of TUs, DUs and their dependencies. The provisioner then traverses the topology and for each TU and DU combination, it deploys the corresponding DIs by invoking LEONORE, adds the DIs to the respective DU as leaf and updates the deployment registry.

C. LEONORE

LEONORE [1] is a service oriented infrastructure and toolset for provisioning application packages on edge devices in large-scale IoT deployments. LEONORE creates installable application packages, which are fully prepared on the provisioning server and specifically catered to the device platform to be provisioned. It allows for both, push- and pull-based provisioning of devices. For our approach we will facilitate and extend LEONORE to provision IoT application deployment topologies on edge devices managed and provisioned by LEONORE. A simplified architecture of LEONORE and connected IoT deployments is depicted on the right hand side of Figure 1. In the following we describe the most important components that are involved when provisioning an IoT application.

1) IoT Gateway: The `IoT Gateway` is a general and generic representation of an IoT device, which considers the resource constrained nature and limitations of these devices. The IoT gateway uses a container for executing application packages, a profiler to monitor the status of the system, an agent to communicate with LEONORE, and a connectivity layer that supports different communication protocols and provisioning strategies.

2) Service API: To allow for the seamless integration of DIANE with LEONORE we extended the provided APIs and created a general `Service API`. This interface allows: (i) to query LEONORE for currently managed devices and their corresponding metadata, (ii) to add additional application artifacts that are needed for building application packages, and (iii) to provision application deployment topologies represented as DIs.

3) Package Management: To provision application components along with the corresponding artifacts, DIANE adds these artifacts and additional metadata (e.g., name, version, executables) via the service API. The `Package Management` component stores the provided information along with the artifacts in a repository.

4) IoT Gateway Management: In order to keep track of connected IoT gateways, LEONORE uses the following approach: During gateway startup, the gateway's local provisioning agent registers the gateway with LEONORE by providing its device-specific information (e.g., id). The `IoT Gateway Management` handles this information by adding it to a repository and assigning a handler that is responsible for handling and provisioning the respective gateway.

5) Provisioning Handler: The `Provisioning Handler` is responsible for the actual provisioning of application packages. The handler decides on an appropriate provisioning strategy, triggers the building of gateway-specific packages and executes the provisioning strategy. Depending on the strategy the IoT gateway either queries the framework for packages or updates get automatically pushed to the gateway.

6) Balancer: Since LEONORE deals with large-scale IoT deployments that potentially create significant load, the framework scales by using so-called LEONORE nodes. These nodes comprise all components that are required for managing IoT gateways. To distribute the gateways evenly on available nodes a `Balancer` is used to assign gateways to available nodes that are then responsible for handling any further interaction with the respective IoT gateways. In case all available nodes are fully loaded, the balancer spins up a new node and queues incoming requests. Similarly, the balancer will decommission nodes when load decreases.

D. Provisioning of IoT Application Deployment Topologies

The overall process of provisioning IoT application deployment topologies is started when DIANE receives a request to deploy a specific IoT application or application package. The process comprises the following steps: (i) in order to generate the deployment topology of an application or application component with DIANE the user provides an optional list of artifacts and a mandatory list of MADCAT units (i.e., TUs and DUs). The deployment manager is then responsible for handling deployment requests and forwards the request to the artifact manager; (ii) the artifact manager resolves artifacts

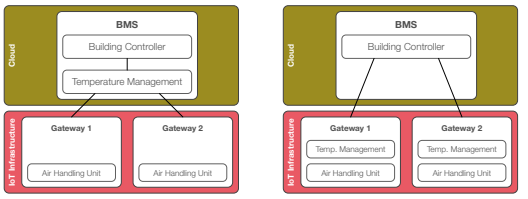
according to the provided information in the TUs by either loading them from a specified repository or using the provided artifacts; (iii) after resolving the artifacts, the artifact manager transfers the artifacts to LEONORE by invoking the Service API; (iv) LEONORE receives the artifacts, packs and stores them in an internal repository; (v) for each TU and DU the deployment handler does the following: (vi) forward the list of TUs and DUs to the dependency management component to resolve dependencies and relations among the units; (vii) resolve possible infrastructure constraints defined in the DUs with the help of the constraint handler; (viii) the constraint handler gathers all managed machines and their corresponding context (e.g. IP, name, runtime) from LEONORE; (ix) according to the specified constraints the handler returns a list of machines that are applicable for deployment of a specific DU; (x) invoke the constraint handler again to generate runtime configurations that are specified in the TU; (xi) generate DIs using the gathered suitable machines and runtime configurations; (xii) for each DI the handler invokes the provisioner that stores the DI and corresponding DUs and TU in the deployment registry, deploys the DI by invoking the service API of LEONORE, which then takes care of provisioning the application deployment on the actual infrastructure.

III. Evaluation

To evaluate our approach we implemented a sample IoT application based on a case study conducted in our lab in cooperation with a business partner in the building management domain. In this case study we identified the requirements and basic components of commonly applied applications in this domain. Based on this knowledge we developed an IoT application for managing and controlling air handling units in buildings, where the design and implementation follows the micro services architecture approach. Next, we created a test setup in the cloud using CoreOS[2] to virtualize edge devices as Docker[3] containers. We use LEONORE's notion of IoT gateways as representation of edge device in our experiments.

In the remainder of this section we give an overview of the developed IoT application, the concrete evaluation setup, present different evaluation scenarios and analyze the gathered results.

A. BMS - IoT Application



(a) Traditional Application Topology

(b) Evolved Application Topology

Fig. 2: IoT Application Topology

[2]https://coreos.com

[3]https://www.docker.com

Currently, applications in IoT are designed and implemented as layered architectures [7], where the bottom layer consists of deployed IoT devices, a middleware that provides a unified view of the deployed IoT infrastructure, and an application layer that executes the business logic [8]. According to this layering, business logic only runs in the application layer and the IoT infrastructure is provisioned with appropriate software, transmits data and reacts to received control information [9]. However, in practice most of the IoT devices provide constrained execution environments that can be used to offload parts of the business logic onto these devices. To compare these two deployment approaches we developed an application for a building management system that consists of the following components: (i) `Air Handling Unit` (unit) is deployed on an IoT device, reads data (e.g., temperature) from a sensor, transmits the data to and reacts on control commands received from the upper layer; (ii) `Temperature Management` (management) is the processing component of the application and gathers the status information of the units. It receives high level directives from the upper layer and based on the processed unit data and the received directives, forwards appropriate control commands to the unit; and (iii) `Building Controller` (control) is the top level component and decides for each handled management component the directive it has to execute. The basic deployment topology that follows the traditional IoT application deployment model is depicted in Figure 2a. In the figure we see that the unit component is deployed on devices in the IoT infrastructure and that both the processing (management) and control components are executed on a platform in the cloud. We refer to this deployment as *traditional application topology*. In contrast, Figure 2b depicts a deployment that offloads some of the processing (management) onto devices, which we refer to as *evolved application topology*.

B. Setup

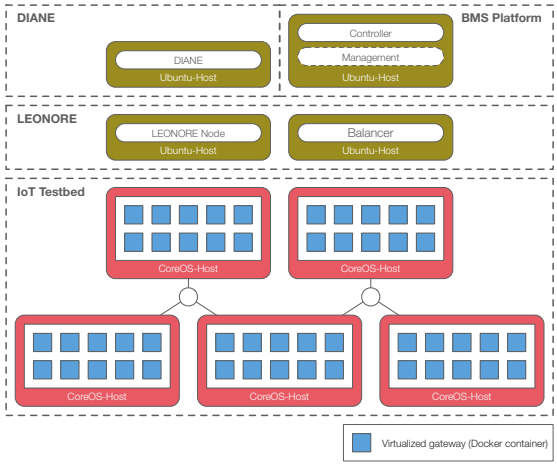
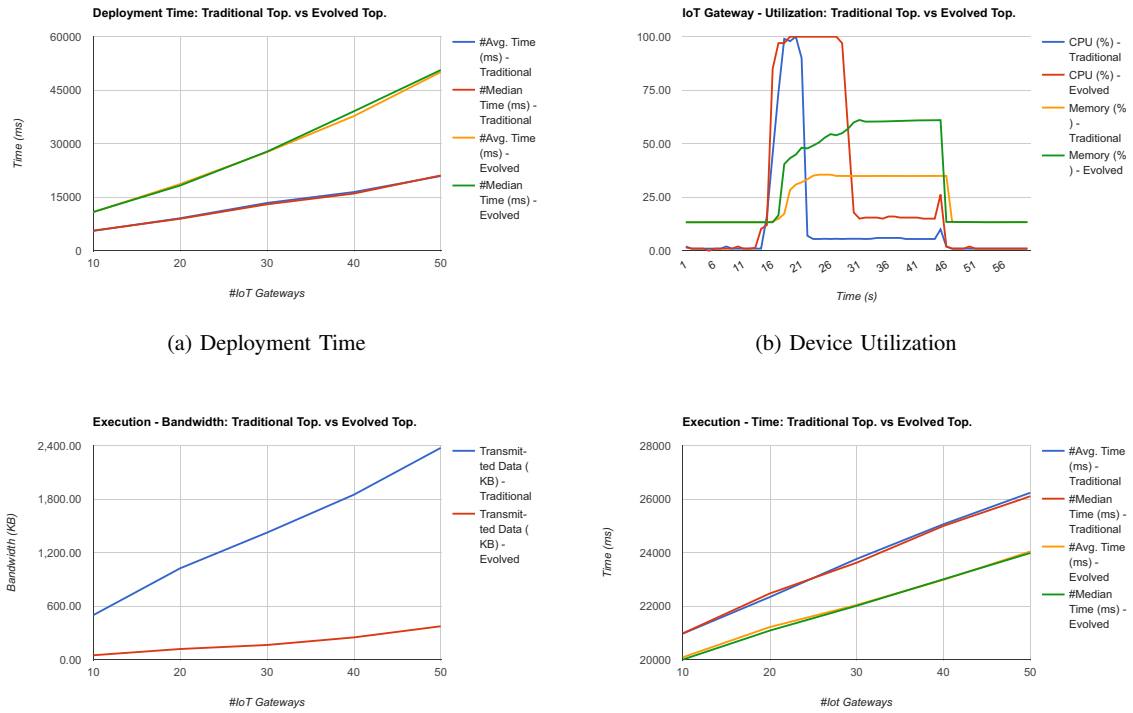


Fig. 3: Evaluation - Setup

For the evaluation of our framework we created an IoT testbed in our private OpenStack[4] cloud. We reuse a Docker

[4]http://www.openstack.org



(a) Deployment Time

(b) Device Utilization

(c) Application Execution - Bandwidth

(d) Application Execution - Time

Fig. 4: Evaluation Results

image that was created for LEONORE to virtualize and mimic a physical gateway in our cloud. To run several of these virtualized gateways, we use CoreOS clusters and fleet[5], a distributed init system, for handling these clusters. Based on fleet's service unit files, we dynamically generate according fleet unit files and use them to automatically create, run and stop virtualized gateways. Figure 3 depicts the overall setup that we use for our experiments. At the bottom, the *IoT Testbed* consists of a CoreOS cluster of 5 virtual machines, where each VM is based on CoreOS 607.0.0 and uses the m1.medium flavor (3750MB RAM, 2 VCPUs and 40GB Disk space). The gateway-specific framework components of LEONORE are pre-installed in the containers. In the middle, the LEONORE framework is distributed over 2 VMs using Ubuntu 14.04. The first VM hosts the balancer and uses the m1.medium flavor. The second VM hosts a LEONORE node and uses the m2.medium flavor (5760MB Ram, 3 VCPUs and 40GB Disk space). On top, DIANE is hosted in one VM using Ubuntu 14.04 with the m1.medium flavor.

The platform components of the BMS IoT application are deployed on a separate VM using Ubuntu 14.04 and the m1.small flavor (1920MB Ram, 1 VCPUs and 40GB Disk space). In order to evaluate and compare both deployment topologies of the application, the BMS platform initially comprises controller and management (traditional application topology), and is then reduced to only host the controller, since the management component is deployed on the devices (evolved application topology). In both scenarios the unit component is deployed and executed on the devices in the IoT infrastructure.

[5]https://github.com/coreos/fleet

C. IoT Application Deployment

In the first experiment we measure the time that is needed for dynamically creating application deployments for the two BMS IoT application deployment topologies and provisioning of these deployments on IoT devices. In the second experiment we compare the device resource utilization when executing the provisioned application deployments.

1) Deployment Time: Figure 4a shows the overall time that is needed for the creation and provisioning of application deployments on an increasing number of devices. The time measurement begins when DIANE is invoked and ends when DIANE reports the successful deployment. To deal with possible outliers and provide more accurate information we executed each measurement 10 times and calculated both the average and median time. In Figure 4a we see that for the traditional application topology the framework provides a stable and acceptable overall deployment time. In comparison the deployment of the evolved application topology takes in total almost twice as long, but also provides a stable deployment time. Taking into account that the evolved application topology needs to deploy twice as many application components and corresponding artifacts, this increase is reasonable since the

limiting factor is the actual provisioning of devices as we create application packages that have more than doubled in size.

2) Gateway Resource Utilization: Figure 4b depicts the cpu and memory utilization of one device when provisioning and executing the two IoT application deployment topologies. The Figure shows that initially there is no application component running on the device. After 15 seconds we initiate the provisioning via our framework, which provisions the application deployments and starts the execution. Then the deployment runs for 30 seconds. Afterwards the framework stops the execution. When provisioning the traditional application topology, we clearly see that the cpu utilization has a short high peak due to the startup of the deployment. However, after this high peak the overall utilization of the device is low and would allow to use this untapped processing power to offload business logic components on the device. To illustrate the feasibility of this claim we also provision and execute the evolved application topology on the device. We see that in comparison to traditional application topology, the load on the device is almost twice as high, but except for the high initial cpu load peak, the overall utilization of the device is still acceptable and reasonable.

D. IoT Application Execution

In the second experiment we have an in-depth look at the BMS application and compare both deployment topologies. In order to do that, we deploy both topologies with our tool on an increasing number on devices. However, now we measure bandwidth consumption and execution time when invoking the application's business logic. The measurement begins by invoking the control component of the application to specify a virtual set-point temperature on each device, where each unit component on the device has the same initial temperature reading. To provide reliable results, we execute each measurement 10 times and freshly provision the devices after each measurement with our tool. Depending on the BMS application deployment topology the management component is either executed in the platform (i.e., the cloud) or on each device.

1) Bandwidth Consumption: Figure 4c shows the average bandwidth consumption that results from invoking the business logic of the two application deployment topologies. We see that the traditional application topology causes a significant amount of data transmission between platform and IoT infrastructure. As a result the transmitted data produces a high load on the network and consumes a lot of bandwidth. This behavior is obvious, since the complete business logic is executed on the platform and devices are only sending and reacting on control messages. In contrast, the evolved application topology produces less traffic and therefore consumes on average only 13% of the bandwidth. This is due to the offloading of the processing (management) component to each device, which therefore drastically reduces the transmitted data between platform and IoT infrastructure.

2) Execution Time: Figure 4d shows the time that is needed for executing the previously described business operation of the BMS application for the two application deployment topologies. We see that for both topologies the application scales well and provides reasonably fast results. However, we notice that the offloading of the processing components on devices reduces the execution time by 7%, since component interaction within a device is faster than the interaction between device and platform.

After presenting and evaluating the gathered experiment results, we can deduce the following: DIANE, the framework for the dynamic creation of IoT application deployment topologies and actual provisioning of these, is capable of dealing with different application topologies and changes in the IoT infrastructure. The framework scales well with increasing size of application deployment topologies and does not add additional overhead to the overall time that is needed for provisioning the IoT infrastructure. Note that for very large deployments the use of multiple coordinated LEONORE nodes is required. Furthermore, depending on the scenario, it is feasible to offload application components from a cloud platform on devices in the IoT infrastructure. Examples of such scenarios are applications that generate a significant amount of traffic between the platform and the IoT infrastructure and therefore justify the additional deployment overhead.

IV. Related Work

The overall terminology of the Internet of Things is well-defined in the literature [8], [9]. In contrast, the definition of applications in IoT comprises applications that are solely deployed in a platform in e.g., the cloud [10] and are executed on top of an resource abstraction layer [11], [12], [13], [14], and distributed applications, where basic components are embedded in devices that are deployed on the edge of the infrastructure for sensing and reacting on the environment and a corresponding enterprise application for managing these devices [15], [16]. The above approaches have in common that they see devices deployed in the IoT infrastructure as external dependencies and therefore do not consider them in the design and development as integral part of the application. Recently, approaches emerged that also consider IoT devices as part of the application that need to be managed efficiently in order to develop and deploy flexible and scalable IoT applications [17], [18], [19], [20], [21]. However, none of the approaches discussed so far, considers deploying and provisioning parts of the application on edge devices that provide constrained execution environments [22] in order to facilitate this untapped processing power and build more robust and adaptable applications. Regarding the deployment, there is only limited amount of prior work on location-aware placement of cloud application components [23], [24], [25], [26], [27], but these approaches are not designed to handle placement decisions for constrained edge infrastructure to improve application deployment topologies.

V. Conclusion

To sense and manipulate the environment, applications in the Internet of Things (IoT) need to manage and integrate a large number of heterogeneous devices for sensing and manipulating the environment. Recently, it emerges that such devices, apart from the most basic sensors and actuators, also provide limited execution environments that are constrained in their processing, memory and storage capabilities. In order to exploit this untapped processing power provided by these IoT devices, parts of an application's business logic can be

offloaded to the edge of the infrastructure, which not only increases the robustness of the application but can also reduce communication overhead. Especially for IoT applications deployed on cloud computing infrastructures the consideration of edge devices is important, since the cloud enables applications to react to the current request load by elastically adjusting their deployment topology. Therefore, in addition to the traditional design considerations for cloud applications, specific issues like the geographical distribution of edge devices and the resulting network latency need to be considered in the design of IoT cloud applications. Furthermore, applications need to be built from clearly separated components, which can be deployed independently, to allow for flexible provisioning of applications whose deployment topology evolves by offloading components to edge devices. This calls for an approach to dynamically generate optimized deployment topologies for IoT cloud applications, which are tailored to the currently available physical infrastructure. DIANE uses a declarative, constraint-based model of the desired application deployment, to enable the flexible provisioning of application components on both, cloud infrastructure, as well as IoT devices deployed in the IoT infrastructure.

In our ongoing work, we plan to extend DIANE to address further challenges. We will integrate non functional elasticity dimensions (e.g., costs) to further optimize deployment topologies [3] and enable local coordination of topology changes between edge devices. Additionally, we plan to further adapt and extend the MADCAT unit methodology to allow for more detailed descriptions of application topologies. Furthermore, we will integrate our framework with our overall efforts in designing, deploying, and managing complex, large-scale IoT applications to provide a comprehensive tool set for researchers and practitioners.

REFERENCES

[1] M. Vögler, J. M. Schleicher, C. Inzinger, S. Nastic, S. Sehic, and S. Dustdar, "LEONORE - Large-Scale Provisioning of Resource-Constrained IoT Deployments," in *Proceedings of the 9th International Symposium on Service-Oriented System Engineering*, ser. SOSE'15. IEEE, 2015, pp. 78–87.

[2] M. Armbrust, I. Stoica, M. Zaharia, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, and A. Rabkin, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, Apr. 2010.

[3] S. Dustdar, Y. Guo, R. Han, B. Satzger, and H.-L. Truong, "Programming Directives for Elastic Computing," *IEEE Internet Computing*, vol. 16, no. 6, pp. 72–77, 2012.

[4] S. Newman, *Building Microservices*. O'Reilly Media, Inc., Feb. 2015.

[5] M. Vögler, F. Li, M. Claeßens, J. M. Schleicher, S. Nastic, and S. Sehic, "COLT Collaborative Delivery of lightweight IoT Applications," in *Proceedings of the 2014 International Conference on IoT as a Service*, ser. IoTaaS'14. Springer, 2014, p. to appear.

[6] C. Inzinger, S. Nastic, S. Sehic, M. Vögler, F. Li, and S. Dustdar, "MADCAT - A Methodology for Architecture and Deployment of Cloud Application Topologies," in *Proceedings of the 8th International Symposium on Service-Oriented System Engineering*, ser. SOSE'14. IEEE, 2014, pp. 13–22.

[7] D. Agrawal, S. Das, and A. El Abbadi, "Big data and cloud computing: new wine or just new bottles?" *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 1647–1648, Sep. 2010.

[8] S. Li, L. D. Xu, and S. Zhao, "The internet of things: a survey," *Information Systems Frontiers*, pp. 1–17, Apr. 2014.

[9] L. Da Xu, W. He, and S. Li, "Internet of Things in Industries: A Survey," *IEEE Transactions on Industrial Informatics*, vol. 10, no. 4, pp. 2233–2243, 2014.

[10] F. Li, M. Vögler, S. Sehic, S. Qanbari, S. Nastic, H.-L. truong, and S. Dustdar, "Web-Scale Service Delivery for Smart Cities," *Internet Computing, IEEE*, vol. 17, no. 4, pp. 78–83, 2013.

[11] M. Kovatsch, "Firm firmware and apps for the Internet of Things," in *Proceedings of the 2nd Workshop on Software Engineering for Sensor Network Applications*, ser. SESENA'11. ACM, 2011, pp. 61–62.

[12] D. Guinard, I. Ion, and S. Mayer, "In Search of an Internet of Things Service Architecture: REST or WS-*? A Developers Perspective," in *Mobile and Ubiquitous Systems: Computing, Networking, and Services*. Springer, 2012, vol. 104, pp. 326–337.

[13] H. Ning and Z. Wang, "Future Internet of Things Architecture: Like Mankind Neural System or Social Organization Framework?" *IEEE Communications Letters*, vol. 15, no. 4, pp. 461–463, 2011.

[14] P. Patel, A. Pathak, T. Teixeira, and V. Issarny, "Towards application development for the internet of things," in *Proceedings of the 8th Middleware Doctoral Symposium*, ser. MDS'11. ACM, 2011, pp. 5:1–5:6.

[15] Q. Zhu, R. Wang, Q. Chen, Y. Liu, and W. Qin, "IOT Gateway: Bridging Wireless Sensor Networks into Internet of Things," in *Proceedings of the IEEE/IFIP 8th International Conference on Embedded and Ubiquitous Computing*, ser. EUC'10, 2010, pp. 347–352.

[16] W. Colitti, K. Steenhaut, N. De Caro, B. Buta, and V. Dobrota, "REST Enabled Wireless Sensor Networks for Seamless Integration with Web Applications," in *Proceedings of the 8th International Conference on Mobile Adhoc and Sensor Systems*, ser. MASS'11. IEEE, 2011, pp. 867–872.

[17] F. Li, M. Vögler, M. Claessens, and S. Dustdar, "Towards Automated IoT Application Deployment by a Cloud-Based Approach," in *Service-Oriented Computing and Applications (SOCA), 2013 IEEE 6th International Conference on*, 2013, pp. 61–68.

[18] S. Nastic, S. Sehic, M. Vögler, H. L. Truong, and S. Dustdar, "PatRICIA – A Novel Programming Model for IoT Applications on Cloud Platforms," in *Service-Oriented Computing and Applications (SOCA), 2013 IEEE 6th International Conference on*, 2013, pp. 53–60.

[19] R. Khan, S. U. Khan, R. Zaheer, and S. Khan, "Future internet: The internet of things architecture, possible applications and key challenges," *Proceedings - 10th International Conference on Frontiers of Information Technology, FIT 2012*, pp. 257–260, 2012.

[20] J. a. Stankovic, "Research Directions for the Internet of Things," *Internet of Things Journal, IEEE*, vol. 1, no. 1, pp. 3–9, 2014.

[21] S. S. Yau and A. B. Buduru, "Intelligent Planning for Developing Mobile IoT Applications Using Cloud Systems," in *Mobile Services (MS), 2014 IEEE International Conference on*, 2014, pp. 55–62.

[22] A. Sehgal, V. Perelman, S. Kuryla, and J. Schonwalder, "Management of resource constrained devices in the internet of things," *Communications Magazine, IEEE*, vol. 50, no. 12, pp. 144–149, 2012.

[23] I. Narayanan, A. Kansal, A. Sivasubramaniam, B. Urgaonkar, and S. Govindan, "Towards a Leaner Geo-distributed Cloud Infrastructure," in *Proceedings of the 6th USENIX Workshop on Hot Topics in Cloud Computing*, ser. HotCloud'14. USENIX Association, 2014.

[24] R. Buyya, R. N. Calheiros, and X. Li, "Autonomic Cloud computing: Open challenges and architectural elements," in *Proceedings of the Third International Conference on Emerging Applications of Information Technology*, ser. EAIT'12, 2012, pp. 3–10.

[25] P. Mayer, J. Velasco, A. Klarl, R. Hennicker, M. Puviani, F. Tiezzi, R. Pugliese, J. Keznikl, and T. Bureš, "The Autonomic Cloud," in *Software Engineering for Collective Autonomic Systems*. Springer, 2015, pp. 495–512.

[26] H. Qian and M. Rabinovich, "Application Placement and Demand Distribution in a Global Elastic Cloud: A Unified Approach," in *Proceedings of the 10th International Conference on Autonomic Computing*, ser. ICAC'13. USENIX Association, 2013, pp. 1–12.

[27] S. Radovanovic, N. Nemet, M. Cetkovic, M. Z. Bjelica, and N. Teslic, "Cloud-based framework for QoS monitoring and provisioning in consumer devices," in *Consumer Electronics ?? Berlin (ICCE-Berlin), 2013. ICCEBerlin 2013. IEEE Third International Conference on*, 2013, pp. 1–3.