Optimizing Elastic IoT Application Deployments

Michael Vögler, Johannes M. Schleicher, Christian Inzinger, and Schahram Dustdar, *Fellow, IEEE*

Abstract—Applications in the Internet of Things (IoT) domain need to integrate and manage large numbers of heterogenous devices. Traditionally, such devices are treated as external dependencies that reside at the edge of the infrastructure and mainly transmit sensed data or react 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 only simple sensors or transmitters, but provide limited execution environments. This opens up an opportunity to utilize this previously untapped processing power in order to offload parts of the application logic directly to these edge devices. To effectively exploit this new type of device, the design of IoT applications needs to change to explicitly consider devices that are deployed in the edge of the infrastructure. This will not only increase the overall flexibility and robustness of IoT applications, but also reduce costs by cutting down expensive communication overhead. Therefore, to allow the 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 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. Using our framework, applications can furthermore evolve their deployment topologies at runtime in order to react on environmental changes, such as changing request loads. Our framework supports different IoT application topologies and we show that our solution elastically provisions application deployment topologies using a cloud-based testbed.

Index Terms—Internet of Things, Cloud Computing, Application Deployment, Topology Optimization

1 Introduction

Internet of Things (IoT) applications are expected to manage and integrate an ever-increasing number of heterogeneous devices to sense and manipulate their environment. Increasingly, such devices do not only serve 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 this accrued execution capabilities 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]. This explicit consideration of 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 actual request load.

In addition to the traditional design considerations for cloud applications, IoT cloud applications must be designed to cope with issues arising from geographic distribution of edge devices, network latency and outages, as well as regulatory requirements. We argue that edge devices must be treated as first-class citizens when designing IoT cloud applications and the traditional notion of cloud resource elasticity [3] 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 environments and the fact that they are deployed at customer premises to integrate and connect to local sensors and actors requires special consideration when provisioning components on IoT gateways. By carefully deciding when to deploy certain 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 at the infrastructure edge 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 necessary whenever a new application needs to be deployed, but can also be caused by environmental changes, such as changing

- *M. Vögler, J. M. Schleicher, and S. Dustdar are with the Distributed Systems Group, TU Wien, Austria. Email: {voegler, schleicher, dustdar}@dsg.tuwien.ac.at.*
- *C. Inzinger is with the s.e.a.l. - software evolution & architecture lab, University of Zurich, Switzerland. Email: inzinger@ifi.uzh.ch*

Manuscript received December 27, 2015; revised August 7, 2016.

request patterns, changes in the physical edge infrastructure (e.g., adding/removing sensors or IoT gateways), evolutionary changes in application business logic throughout its lifecycle, or evolving non-functional requirements.

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 deployed IoT gateways. DIANE is furthermore continuously monitoring the available edge infrastructure and can autonomously optimize application deployment topologies in reaction to changes in the application environment, such as significant changes in request load, network partitions, or device failures.

A preliminary version of this approach was presented in [6], where we introduce the fundamental concepts of the DIANE framework, along with a mechanism for a priori generation and subsequent provisioning of optimized deployment topologies. In this work, we extend the framework with a two-fold optimization mechanism that enables the evolution of application deployment topologies at runtime in reaction to changes in their execution environment. Furthermore, we provide a detailed discussion of the prototype implementation and significantly extend the evaluation of our framework.

The remainder of this paper is structured as follows: In Section 2 we outline specific requirements that need to be addressed. In Section 3 we introduce the DIANE framework to dynamically create application deployment topologies for large-scale IoT cloud systems, and present our approach for optimizing deployments at runtime in Section 4. We provide detailed evaluations in Section 5 and Section 6, discuss relevant related research in Section 7, followed by a conclusion and an outlook on future research in Section 8.

2 Requirements

The emergence of the IoT in combination with the advent and rapid adoption of the smart city paradigm give rise to a domain of edge devices that are pervasively deployed in large numbers around the globe. As outlined previously, the convergence of cloud computing and IoT paradigms, and especially the evolution of IoT gateways to include constrained execution environments, allows for systems with ever changing deployment topologies due to various evolving factors. Specifically, vital aspects of the smart city domain, like Building Management Systems (BMS) that need to deal with billions of devices, or Traffic Control Systems (TCS) that depend on optimal resource utilization in order to handle large amounts of sensor data, need to be able to optimize their deployment topologies both during deployment and at runtime in order to enable optimal resource utilization. To allow for dynamic generation of optimal deployment topologies for such applications, a solution must meet the following requirements: 1) It needs to enable *optimal utilization of edge devices* with 2) the ability to *dynamically move application logic to these devices*. 3) Furthermore, it shall allow for *deployment topologies to evolve during runtime* and 4) needs to respect *non-functional requirements* that arise in this context.

3 The DIANE Framework

In order to address the previously identified 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.

3.1 IoT Application Design and Implementation

To dynamically generate deployment topologies for IoT applications, the design and implementation of such applications have to follow the microservices architecture approach [4], which enables developers to build flexible applications whose components can be independently evolved and managed. Therefore, each component of an application has to be self-contained, able to run separately, and facilitate loosely coupled communication for interacting with other components. In addition to this application design approach, we are using MADCAT [7] for describing the overall application and its components. MADCAT allows for the 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.

`Technical Units` are used to describe application components by considering abstract architectural concerns and concrete deployment artifacts to capture technology decisions that depend on the actual implementation. To manage multiple possible TUs to realize a specific application component, MADCAT employs decision trees that assist developers of such applications in creating 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

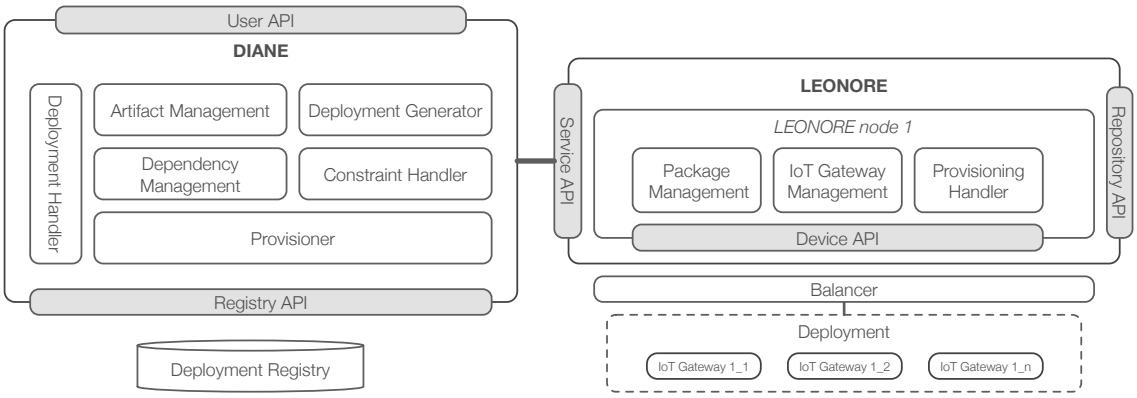


Fig. 1: DIANE and LEONORE – Overview

A TU starts with a `context` to specify 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 that is described by the TU. 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. In order to create an executable, `build` specifies an `assembly` that describes the location within a repository and the name of the executable. Furthermore, `build` defines `steps` that need to be executed to create the executable. Next, `execute` defines the necessary steps for running the executable. In addition to the execution steps, `configuration` stores a possible runtime configuration (e.g., environment variables) that is needed for execution. To allow configuration items to map to other application components, `dependencies` reference TUs of other application components. Finally, the TU enables developers to provide relevant `constraints` that help users of the application to decide on a suitable deployment infrastructure.

For each TU an operations manager can create one or more `Deployment Units` (DUs). In essence, a DU describes how an associated TU can be deployed on concrete infrastructure. To create a specific DU the provider uses the information contained in the TU and its 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": [...]
}
```

Like a TU, a DU also has a `context`, `type`, and `name`. Next, `technicalUnits` allow referencing 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.

By using TUs and corresponding DUs it is possible to completely describe an IoT application. To finally provision an application deployment, DIANE uses TUs, DUs and concrete infrastructure knowledge to generate `Deployment Instances` (DIs). DIs represent concrete deployments on actual machines of the infrastructure, by considering 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"}]}
}
```

Again, 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 by the `application` attribute. It contains the `name` and the `version` of the application component. Finally, runtime configurations required by the component are resolved by the framework and represented in `environment`.

3.2 DIANE Framework

The framework that allows generating IoT application deployment topologies and deals with 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 its overall design follows the microservices architecture principle. In the following, we introduce the main components of DIANE and discuss the integration with LEONORE [1] for provisioning edge devices. Finally, we describe the concrete process of generating and provisioning application deployment topologies.

To keep track of deployments and their relation to TUs and DUs, DIANE provides a `Deployment Registry`. The registry stores units and deployments using a tree structure that represents the relations among them. By managing TUs and corresponding 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 component separately.

In order to provision an IoT application deployment topology with DIANE, the user of the framework has to invoke the `User API` by providing the following required information: (i) TUs, (ii) corresponding DUs, and (iii) optional artifacts that are needed by the deployment (e.g., executables) but cannot be resolved automatically by the framework, such as 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.

In addition to the discussed units, the framework also requires corresponding application artifacts. Therefore, the `Artifact Management` component receives artifacts, resolves all references, and creates an artifact package that is transferred to LEONORE. Each created artifact package contains an executable, a version, and the commands to start and stop the artifact.

To generate DIs, the `Deployment Generator` resolves the dependencies among the provided TUs and DUs by using the `Dependency Management`. The management component returns a tree structure that represents dependencies among units. In addition, the generator handles possible deployment constraints that are specified in the DUs by invoking the `Constraint Handler`. The invoked handler returns a list of infrastructure resources that comply with the specified constraints. Before generating DIs, the generator needs to resolve application runtime configurations (e.g., application properties) in the TUs. This is done by delegating the configuration resolving process to the constraint handler, which provides a temporary configuration. Finally, the generator creates the actual DIs by mapping DUs to concrete machines and updating possible links in the temporary configuration that correspond to infrastructure properties (e.g., IP address of a machine).

Since units in our approach reference each other, the `Dependency Management` is responsible for resolving these 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 appends 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.

To find suitable machines for the deployment of application components, DUs allow defining 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). Whereas, software constraints define requirements that correspond to the application component respectively its deployment (e.g., should this component be replicated and if so on how many machines). In order to provide a list of suitable machines the `Constraint 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 are not needed in case software constraints only demand for a certain number of machines.

For actually provisioning the final DIs the `Provisioner` component is used. The component receives generated DIs and the respective 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 node and updates the deployment registry.

3.3 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. For our approach, we will facilitate and extend LEONORE to provision IoT application deployment topologies on edge devices managed and provisioned by DIANE. 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.

The `IoT Gateway` is a generic representation of an IoT device that especially 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, and an agent to communicate with LEONORE. To allow for the seamless integration of DIANE with LEONORE, we extend the provided APIs and create 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. To provision application components along with corresponding artifacts, DIANE uses LEONORE's service API to supplement these artifacts with additional metadata (e.g., name, version, executables). The `Package Management` component stores the provided information along with the artifacts in a repository. 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. The `IoT Gateway Management` handles this

information by adding it to a repository and assigning a handler that is responsible for managing and provisioning the respective gateway. 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. Since LEONORE deals with large-scale IoT deployments that potentially generate significant load, the framework elastically scales using dynamically provisioned 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. This requires an initial capacity planning step to determine the number of devices that can be reliably provisioned using one LEONORE node. The framework then commissions an initial set of LEONORE nodes using a $N + 1$ strategy with one active node and one hot standby. If all active nodes are fully loaded, the balancer spins up a new node and queues incoming requests. Similarly, the balancer will decommission nodes when load decreases.

3.4 Provisioning of IoT Application Deployment Topologies

The provisioning of IoT application deployment topologies is started when DIANE receives a request to deploy a specific IoT application or application component. The overall process comprises the following steps: (1) 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). Next, the deployment manager is responsible for handling deployment requests and forwarding them to the artifact manager. (2) 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. (3) After resolving the artifacts, the artifact manager invokes the service API to transfer the artifacts to LEONORE. (4) LEONORE receives the artifacts to subsequently pack and store them in its internal repository. (5) For each TU and DU the deployment handler does the following: (6) Forward the list of TUs and DUs to the dependency management component to resolve dependencies and relations among the units. (7) Resolve possible infrastructure constraints that are defined in the DUs by using the constraint handler. (8) The constraint handler gathers all managed machines and their corresponding context (e.g., IP, name, runtime) from LEONORE. (9) According to specified constraints the handler returns a set of machines that are suitable for deploying a specific DU. (10) Invoke the constraint handler again to generate runtime configurations that are specified in the TU, and generate DIs using the gathered suitable machines and runtime configurations. (11) Finally, 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.

4 Application Deployment Optimization

After presenting the overall approach and the respective realization in the previous section, we now discuss an extension for optimizing the application deployment toplogy at runtime. In the approach presented so far, we only consider the initial deployment of application topologies and its respective components. However, since IoT applications have to deal with varying loads during operation, we need a mechanism that allows for adapting application topologies at runtime in order to provide the necessary performance and flexibility. Furthermore, this would also enable applications to fully utilize the available processing power of the edge infrastructure. To address these requirements, we extend DIANE to add a two-fold optimization approach, and apply the introduced notion of offloading business logic to the infrastructure edge to DIANE itself.

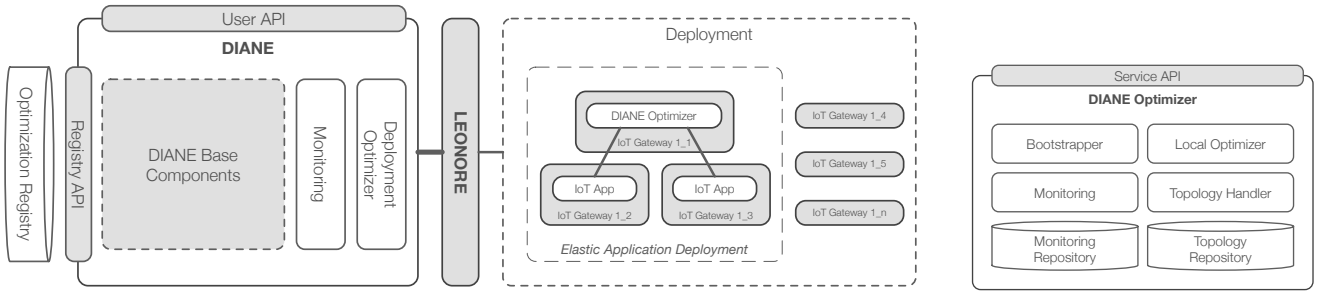
4.1 Elastic Application Deployment

To allow for the optimization of application topologies at runtime, we introduce the notion of an `Elastic Application Deployment`. In contrast to our initial approach that only deploys application components on a set of pre-defined edge devices, we now extend the provisioning mechanism to allow operators to define a hot pool of devices. On theses additional devices, application components are provisioned, but remain idle until they get started. Therefore, this hot pool will be used for optimizing applications, e.g., by scaling application components up or down depending on the application load. In essence, the elastic application deployment consists of a set of devices, which host deployed and running application components, and an additional pool of devices that are provisioned with redundant application components that are initially idle. To manage this new form of deployment, we introduce DIANE Optimizers that get provisioned by DIANE and are running on actual edge devices.

4.2 MADCAT Unit Extensions

In order to enable DIANE to start adapting the topology of a running application, we need an approach that allows the acquisition of runtime information of this application. This information should comprise both, details about the facilitated deployment on the infrastructure (e.g., currently used number of edge devices), as well as application-specific performance metrics like current request load. Based on this information, DIANE can then decide on the best optimization strategy and how to apply the strategy appropriately.

Therefore, we extend our application description approach, which is based on the MADCAT methodology. First, we introduce so called `endpoint` attributes in a DU. An endpoint represents a URL where application-specific performance metrics can be acquired. Since we want to provide an extensible approach, the defined endpoint can either be provided by the application itself or by an external monitoring tool. Furthermore, to support multiple performance metrics, an application can have a list of endpoints that can be used by DIANE for gathering runtime information. To identify endpoints, each provided endpoint has a unique name within a DU.



(a) DIANE extended (b) DIANE Optimizer

Fig. 2: DIANE extensions – Overview

Next, based on monitoring information, we need a mechanism to define certain criteria that allow for deciding if an application topology needs to be adapted. Consequently, we extend the overall MADCAT methodology to introduce Optimization Units. An `Optimization Unit` (OU) is used to describe two types of rules that can be used for optimizing an application deployment. First, `application-rules` define criteria for application-specific performance metrics. Second, `infrastructure-rules` define criteria that are targeted towards the used deployment infrastructure. An example of an OU can be seen in Listing 4.

Listing 4: Optimization Unit

```
{
  "@context": "http://madcat.dsg.tuwien.ac.at/",
  "@type": "OptimizationUnit",
  "name": "BMS",
  "technicalUnits": [{"name": "BMS/Control"}],
  "application-rules": [
    {"name": "response", "endpoint": "@BMS/Control.
        endpoints.response", "contract": "UNDER", "value":
        "3"}],
  "infrastructure-rules": [
    {"name": "cpu", "contract": "MIN"}],
  "action-policies": [{"name": "ScalingPolicy"}]
}
```

Listing 4 describes an application rule that defines that the response time that can be measured from the given endpoint should be under 3 seconds. Next, an infrastructure rule is defined that demands that the application deployment running on the infrastructure should keep the consumed processing power minimal. The difference between these two types of rules is that the former requires monitoring the application itself by using the defined endpoints, whereas the latter requires in-depth knowledge about the used infrastructure resources.

Next, an OU provides an `action-policies` attribute that references either pre-defined or custom-built action policies based on the MONINA language [8], [9]. These policies define a set of actions to be used for optimizing the application whenever any application rules are violated. For example, an action policy can define that in order to react to increased load, the application deployment needs to be scaled up by using more available machines, or scale down if performance metrics indicate that the current load can be managed with a smaller deployment.

By using the described unit extensions, operators can now define how a deployed application can be monitored and under which circumstances its deployment should be optimized.

4.3 Server-side Extension

To enable the optimization of deployed application topologies, we extend DIANE by adding several new components, which are depicted in Figure 2a. In the following, we describe them in more detail.

We extend the `User API` to allow operators to upload OUs that define criteria for triggering the optimization of an application's deployment. Next, operators can use the user API to define custom action policies for describing how an application can be optimized. Since we demand that applications deployed with DIANE follow the microservice architecture approach, optimizing the deployment of an application is relatively easy by evolving the deployment topology. For example, a simple approach to deal with increased load that demands more processing power, is to scale up the application deployment by using additional resources. Uploaded OUs and defined action policies are stored in the optimization registry.

We introduce a `Monitoring` component to collect runtime measurements from deployed and running applications that usually reside in the same deployment infrastructure as our framework (e.g., cloud). Based on the details defined in an OU, the monitoring component creates application-specific listeners for the given endpoints to acquire performance measurements from the application in a configurable interval. The collected data is then forwarded to the deployment optimizer, which takes care of further processing.

To optimize the deployment of an application based on defined rules, we introduce a separate `Deployment Optimizer` component. The optimizer receives collected data from the monitoring component and then analyzes the data based on the defined rules and thresholds in the corresponding OU. When the optimizer detects that the application no longer meets the defined criteria it provides the following two optimization modes:

1) *Blackbox Mode*: In blackbox mode, DIANE optimizes the application deployment by treating the deployment infrastructure as black box, which means that the deployment optimizer has no specific knowledge about the used edge devices and their respective resources. In

this mode, the deployment optimizer can only optimize for application-rules.

2) *Whitebox Mode*: In whitebox mode, the deployment optimizer has full knowledge of the used deployment infrastructure and can therefore also optimize for infrastructure-rules.

In order to enable these optimization modes, we present the DIANE Optimizer, which can be deployed in the edge infrastructure. The DIANE Optimizer monitors and controls an elastic application deployment that allows for optimizing the deployment topology of an application by either starting currently idle application components, or stopping unnecessary components.

To allow DIANE to facilitate the DIANE Optimizer, the optimizer needs to be associated with an application and then deployed in the edge infrastructure. This is done using the following approach: (i) When an OU is uploaded by an operator via the service API, DIANE extracts which application and respective components are affected. (ii) Next, the respective DIs are analyzed to gather the used deployment in the infrastructure. (iii) To form an elastic application deployment based on the defined action policies, the deployment generator is used to generate a fresh set of DIs that is provisioned, but not yet started to form a pool of idle components to allow for the evolution of the application topology. (iv) Then, the constraint handler is used for finding a suitable machine for running the DIANE Optimizer, and the provisioner is used for deploying the optimizer on the selected machine. (v) Finally, once the optimizer registers itself with DIANE, it is provided with the deployment topology of the application, as well as the provisioned but not yet started DIs that can be used for optimizing the application deployment.

To keep track of uploaded OUs, corresponding action policies, and deployed DIANE Optimizers, we add an `Optimization Registry`. In this repository, for each application that is handled by DIANE, we store defined OUs and corresponding action polices. In addition, for each DIANE Optimizer deployment, we store the ID of the optimizer as well as the machine in the infrastructure that is hosting the optimizer. The combination of optimizer ID, and machine IP and ID allows DIANE to uniquely identify the optimizer deployment.

4.4 DIANE Optimizer

The DIANE Optimizer enables the optimization of an application topology by monitoring the actual deployment infrastructure, which provides valuable insights on the infrastructure performance. The DIANE Optimizer is specifically catered to be lightweight in terms of memory consumption and CPU usage, so that it can be executed on machines residing in the edge infrastructure that only provide a fraction of the processing power of cloud resources. The architecture of the DIANE Optimizer is depicted in Figure 2b. In the following, we outline the basic components of a DIANE Optimizer.

Once a DIANE Optimizer is deployed in the edge infrastructure, the `Bootstrapper` component of the optimizer is responsible for registering the deployment with DIANE. Based on this information, the server-side framework can keep track of deployed optimizers. Furthermore, during the registration process the optimizer receives the list of machines representing the current deployment of the application, as well as a hot pool of machines where application components are already provisioned, but not yet started. These lists are then forwarded to the topology handler for further processing.

To form an elastic application deployment the `Topology Handler` first extracts the devices that represent the current application deployment based on the provided information from the bootstrapper. This topology representation is then enriched with the current hot pool of application components and then updated in the `Topology Repository`. Based on this stored topology, the DIANE Optimizer knows which devices are currently used by the application and is also able to optimize the overall application topology by starting idle or stopping running components.

To gather valuable insights from the used deployment infrastructure, the DIANE Optimizer uses a dedicated `Monitoring` component. According to the stored application topology, the monitoring extracts the respective machines. In order to acquire performance measurements from these machines, the DIANE Optimizer facilitates the LEONORE profiler that is pre-installed on the machines to extract performance data like used CPU and consumed memory. Therefore, whenever the application topology is updated (e.g., new machines are added) the monitoring component contacts each machine of the deployment to register an endpoint where the machines, respectively their LEONORE profilers, publish the profiled monitoring information in a configurable interval. The published performance profiles of the machines are then grouped by machine and stored for later analysis in the local `Monitoring Repository`. The repository is implemented as a local cache using available RAM and/or disk resources if available, which allows for fast read and write access, while still considering the resource-constrained nature of the underlying infrastructure. To save memory, the cache only keeps the most recent profiles. Furthermore, since the collection of data is happening in the edge infrastructure the overall communication costs are considerably low.

In the current version, a DIANE Optimizer does not automatically decide when to optimize its corresponding elastic application topology. Therefore, it provides a `Service API` that allows DIANE to trigger a deployment evolution. Whenever DIANE decides that based on a defined application rule the application deployment has to be optimized, it finds the responsible DIANE Optimizer and invokes the service API by providing infrastructure rules and action policies that need to be respected. Next, the request is forwarded to the local optimizer, which is then responsible for choosing suitable optimization actions and executing them accordingly.

Once DIANE triggers an optimization by invoking the DIANE Optimizer, the `Local Optimizer` performs the following steps in order to process the request: (i) Analyze the given application policy to identify a set of possible deployments that need to be updated for optimizing the application topology. (ii) If infrastructure rules are defined, the set of possible deployments is filtered by using gathered

monitoring information. For example, if an application rule describes that the used CPU of the deployment has to be kept minimal, the optimizer will use the performance profiles stored in the monitoring repository to choose a small deployment that can deal with the load while only consuming a fraction of the provided total resources. (iii) If no application rules are defined, the set is reduced by picking deployments naïvely. (iv) After the set of deployments that need to be updated is finalized, the application policy is executed. This means that the application deployment topology is optimized by either starting idle or stopping running application components. (v) Finally, the topology handler is notified to store the evolved application deployment in the topology repository.

In case DIANE detects that a DIANE Optimizer is not responding anymore, the server-side framework restarts or redeploys the machine the optimizer is deployed on.

4.5 Optimizing an Elastic Application Deployment

The process of optimizing an elastic application deployment is initiated by an operator that defines an OU and corresponding action policies. To describe the overall process let us consider that we want to scale up an application deployment to a maximum of 20 machines (action policy) whenever the response time of the application is over a defined threshold (application rule). Furthermore, during scale up the deployment should be kept minimal in terms of used CPU (infrastructure rule). After describing these requirements, the operator uploads the OU and the action policy to DIANE. Based on the input, DIANE creates an elastic application deployment and deploys a DIANE Optimizer. Next, the monitoring component starts collecting response time measurements from the defined endpoints of the application. Once DIANE detects that the response time of the application violates the defined threshold in the OU, it invokes the respective DIANE Optimizer by providing the defined scale up action policy and infrastructure rule. Then, the DIANE Optimizer decides that based on the provided input and gathered performance profiles of the machines, it is sufficient to scale up the application deployment by only using 2 additional devices and queues further scale up requests from DIANE until these devices are fully utilized. In case no infrastructure rules are defined by the operator, the overall approach follows the same steps as above, except that no infrastructure information is used by the DIANE Optimizer and the deployment is scaled up by using a naïve approach (e.g., 5 devices for each scale up request).

Using explicit infrastructure knowledge (whitebox mode) allows the DIANE Optimizer to optimize the application deployment topology more efficiently compared to an approach that only uses pre-defined or naïve adaptation steps (blackbox mode).

5 Evaluation – IoT Application Deployment and Execution

To evaluate our approach we implemented a demo 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 microservices architecture approach. Next, we created a test setup in the cloud using CoreOS[2] to virtualize edge devices as Docker[3] containers. We reuse 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 demo application and the created evaluation setup, present different evaluation scenarios, and analyze the gathered results.

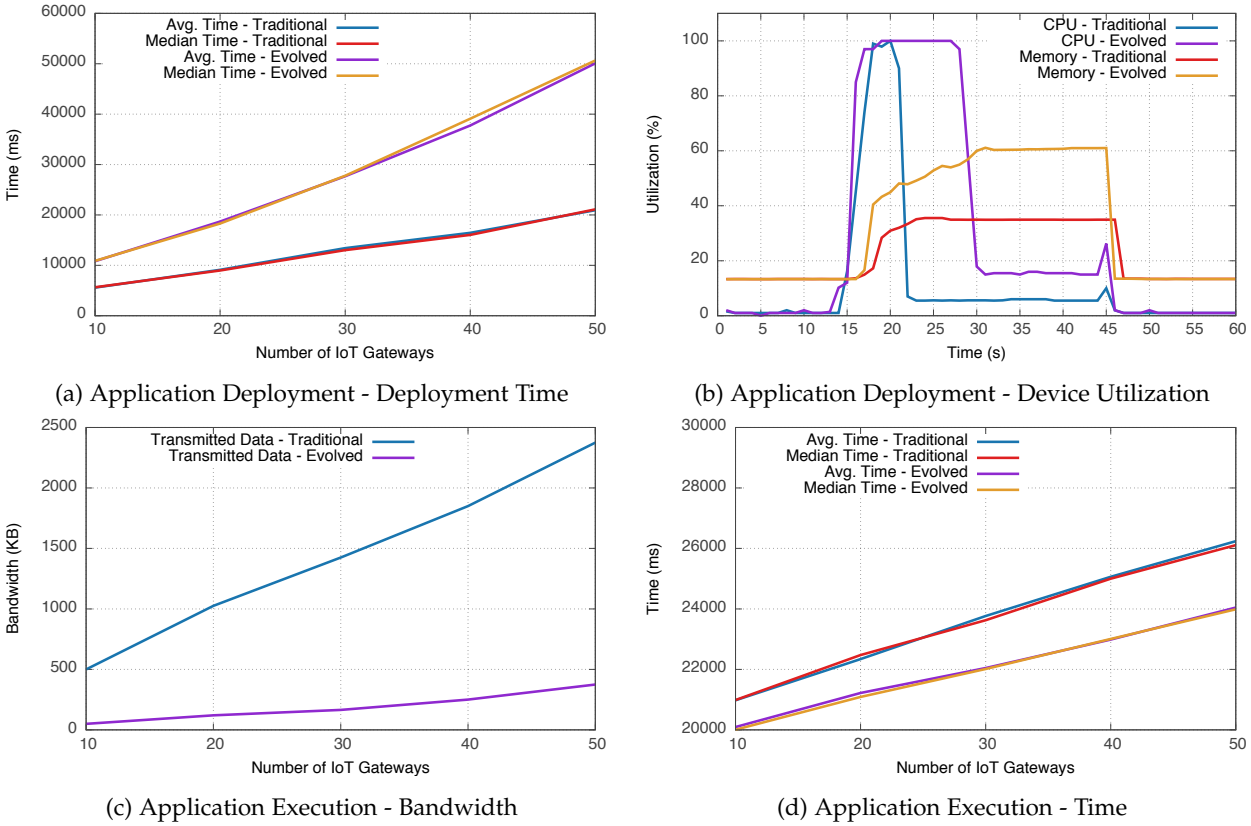
5.1 BMS Demo Application

Currently, IoT applications are designed and implemented as layered architectures [10]. This means that 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 business logic [11]. According to this layered approach, business logic only runs in the application layer and the IoT infrastructure is provisioned with appropriate software, sends data, and reacts on its environment [12]. However, in practice more and more IoT devices provide constrained execution environments that can be used for offloading parts of the business logic. To compare these two deployment approaches we develop an application for a building management system that consists of the following components: (1) An `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. (2) A `Temperature Management` (management) represents 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. (3) Finally, the `Building Controller` (control) is the top level component and decides for each handled management component the directive it has to execute. In the traditional deployment topology that follows the common IoT application deployment model, the unit component is deployed on devices in the IoT infrastructure, and both the processing and control components are executed on a platform in the cloud. We refer to this deployment as *traditional application topology*. In contrast, in a contemporary deployment topology, some of the processing logic is offloaded onto devices in the IoT infrastructure, which we refer to as *evolved application topology*.

5.2 Setup

For the evaluation of our framework we create an IoT testbed in our private OpenStack[4] cloud. We reuse a Docker image that was created for LEONORE to virtualize and mimic a physical gateway in our cloud. To run several of

2. https://coreos.com
3. https://www.docker.com
4. http://www.openstack.org



(a) Application Deployment - Deployment Time

(b) Application Deployment - Device Utilization

(c) Application Execution - Bandwidth

(d) Application Execution - Time

Fig. 3: Evaluation Results – IoT Application Deployment & Execution

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. As foundation of our setup, an *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 IoT gateway-specific framework components of LEONORE are pre-installed in the containers. On top of the testbed, the LEONORE framework is distributed over 2 VMs using Ubuntu 14.04. The first VM hosts the balancer and uses the m1.medium flavor, whereas the second VM uses the m2.medium flavor (5760MB Ram, 3 VCPUs and 40GB Disk space) and is deployed with a LEONORE node. On top, DIANE is hosted in one VM using Ubuntu 14.04 with the m1.medium flavor. Finally, the platform components of the BMS demo 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 the two presented 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 in the cloud, since the management component is deployed on the devices (evolved application topology). In both scenarios the unit component is deployed and running on the devices in the IoT infrastructure.

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

5.3 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.

5.3.1 Deployment Time

Figure 3a shows the overall time that is needed for creating 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 3a 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 requires deploying twice as many application components and corresponding artifacts, however, we argue that 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.

5.3.2 Gateway Resource Utilization

Figure 3b 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 deployment via our framework, which provisions the application deployments and starts the execution. Then, the deployments run 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 leaves room for using 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 the traditional application topology, the load on the device has almost doubled, and except for the high initial CPU load peak, the overall utilization of the device is still acceptable and reasonable.

5.4 IoT Application Execution

In the second experiment we collect runtime information from the BMS application to compare both deployment topologies. In order to do that, we deploy both topologies with our framework on an increasing number of 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 DIANE. Depending on the BMS application deployment topology, the management component is either executed in the platform (i.e., the cloud) or on each device.

5.4.1 Bandwidth Consumption

Figure 3c 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 measurements and reacting to 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.

5.4.2 Execution Time

Figure 3d 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 the devices reduces the execution time by 7%, since application 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 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 to 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.

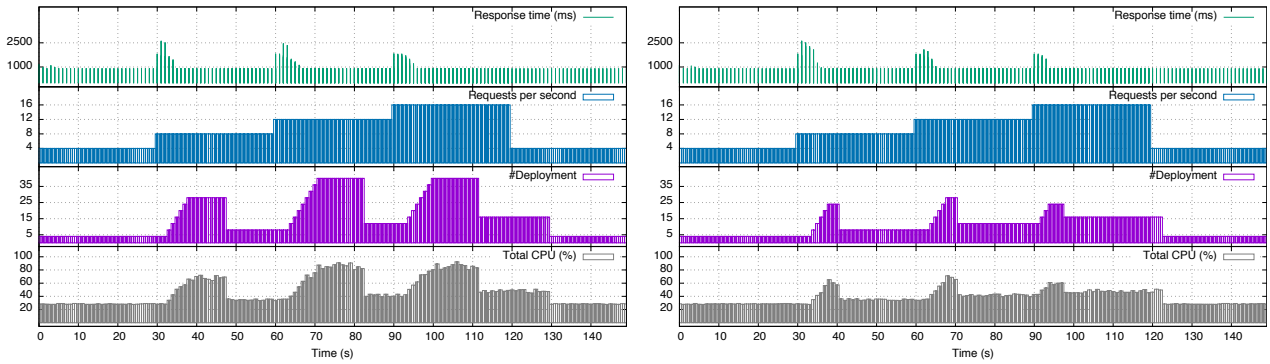
6 Evaluation – Elastic Application Deployment

To evaluate our application deployment optimization mechanism we implemented a smart city demo application and reused the test setup presented in Section 5.2. In the remainder of this section we give an overview of the developed smart city demo application, discuss the concrete evaluation setup, present different evaluation scenarios, and analyze the gathered results.

6.1 Smart City Demo Application

For this experiments we use a demo application that implements the concept of Autonomous Intersection Management[6], which enables autonomous cars in a smart city environment. In our scenario we want to handle large numbers of cars, which requires smart city operators to optimize the deployment topology of such intelligent control systems by using any kind of available processing power. To analyze this approach, we develop a simple traffic control application that manages incoming requests sent from autonomous cars. The incoming requests need to be processed by the application to calculate if a car's intended path is valid (i.e., safe to use). Since the autonomous cars generate a huge load, the application supports scaling the computational logic across infrastructure boundaries. Therefore, the application is separated into two components. A possibly replicated processing component that provides the calculation logic. On top, a central platform component that receives requests by autonomous cars and forwards them to the underlying processing components. Furthermore, to analyze application performance, it provides specific endpoints to acquire metrics like request load and response time.

6. http://www.cs.utexas.edu/~aim/



(a) Blackbox (b) Whitebox

Fig. 4: Evaluation Results – IoT Application Topology Optimization (Step Load Pattern)

6.2 Setup

In order to evaluate the introduced application deployment optimization using the DIANE Optimizer, we reuse the setup presented in Section 5.2. However, for this evaluation, we exchange the VM hosting the components of the BMS IoT application, with a new VM using Ubuntu 14.04 and the m1.small flavor to host the platform component of the smart city demo application. In order to evaluate and compare the different optimization modes, the processing component of the smart city demo application is deployed and executed on the devices in the IoT infrastructure.

6.3 IoT Application Topology Optimization

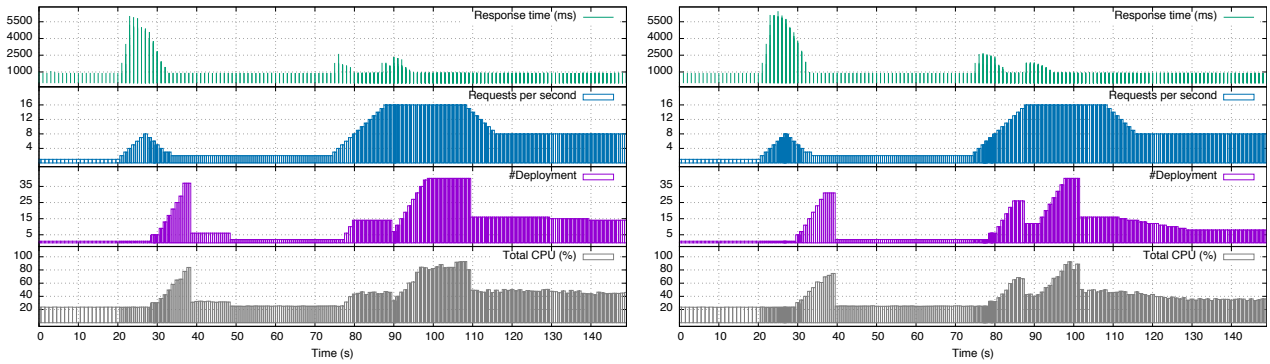
In the following experiments we use DIANE to optimize the deployment topology of the smart city demo application by scaling it across the available IoT infrastructure. We create an OU that defines the allowed threshold for the response time of the application and that the used application deployment should keep the CPU usage across the infrastructure to a minimum. Furthermore, we also define a policy for scaling up the deployment when the response time is over the defined threshold, as well as a policy for scaling down the application by stopping unused infrastructure devices. Additionally, for the experiments we assume that an elastic application deployment was already formed by using a total of 40 machines, plus one additional machine for hosting the DIANE Optimizer.

Next, for comparing the two different optimization modes (blackbox and whitebox) we use different patterns for generating load on the application. In the first scenario, we use a load pattern that simulates a stepwise increase and decrease in requests. In the second scenario, we use a pyramid-like load pattern for sending requests to the application. For the blackbox optimization mode, the deployment topology of the application is scaled without using the provided infrastructure rule, whereas for the whitebox mode we facilitate gathered knowledge about the infrastructure to provide an optimized scaling approach according to the infrastructure rule.

6.3.1 Scenario 1: Step Load Pattern

Figure 4 illustrates the evaluation results for the first scenario. The x-axis shows the temporal course of the evaluation in seconds. In the 'requests per second' section we see that we begin the evaluation by sending 4 concurrent requests per second to the application and increase the load stepwise every 30 seconds to see if DIANE is able to scale up the application. Finally, at 120 seconds we reduce the load to 4 requests per second to see if DIANE is also able to scale down the application. In the 'response time' section we see the response time for each incoming request. The 'deployment' section illustrates the number of facilitated edge devices by the deployment. Finally, the 'total CPU' section represents how much of the total available CPU is used by the application deployment at the given time.

By comparing Figure 4a, which represents the blackbox optimization mode, and Figure 4b, which shows the result for using the whitebox optimization mode, we notice that for the first interval of requests the response time of the application is almost constant for both approaches. At 30 seconds, when the request load doubles we notice that in both cases the response time rises. For both modes, at approximately 34 seconds DIANE starts scaling up the application by invoking the DIANE Optimizer, since the response time of the application violates the provided threshold. However, by looking at the results, we notice several differences during the deployment optimization process. The blackbox mode uses a naïve approach that scales up the deployment until the response time is no longer violated. This, in combination with a lot of queued up requests, leads to the fact that the blackbox mode uses a lot of infrastructure resources for a relatively long time before they are released again. In comparison, in the whitebox mode the DIANE Optimizer uses gathered monitoring information from the deployment infrastructure and only scales up the application when the currently used resources are fully utilized. This allows the application to handle the queued up requests with a smaller deployment in shorter time. For the following two increases in requests per second at 60 and 90 seconds, we see that the framework is also able to detect and analogously handle them. Finally, at 120 seconds, we notice that the load drops, which is detected by the whitebox mode almost immediately, due to the fact that DIANE Optimizer constantly receives information about the used resources. In comparison, the blackbox mode needs significantly more time to detect the changed load by monitoring the application and therefore uses resources for a longer



(a) Blackbox (b) Whitebox

Fig. 5: Evaluation Results – IoT Application Topology Optimization (Pyramid Load Pattern)

period. After comparing both modes using the stepwise load pattern, we can conclude that both approaches allow for optimizing the application deployment according to the provided OU. However, by using gathered knowledge of the infrastructure deployment, the whitebox mode is able to evolve the application topology by using less resources and therefore reduces the total overall CPU utilization by approximately 15%. In addition, we also notice that in total the whitebox mode produces approximately 25% less response time violations compared to the blackbox approach.

6.3.2 Scenario 2: Pyramid Load Pattern

Figure 5 illustrates the evaluation results for the second scenario. We compare the blackbox optimization mode (Figure 5a) and the whitebox approach (Figure 5b) using a pyramid-like load pattern. We notice that for the first 20 seconds the response time of the application for both modes is stable. At 20 seconds the first pyramid load pattern starts increasing the load on the application. We see that it takes a considerable amount of time until DIANE triggers the scale up of the application deployment. Compared to the first scenario, we see that both optimization modes are struggling with this type of load pattern and provide almost identical results. However, by comparing both results, we notice that for the first pyramid-like increase and drop in load, the blackbox mode performs better in terms of violated response times compared to the whitebox approach. This can be explained by the fact that the extremely fast load change does not allow the whitebox mode to utilize gathered infrastructure information. In addition, by looking at the deployment size we see that the whitebox mode uses a smaller deployment for a longer time, compared to the blackbox mode. For the next load increase at 75 seconds we see that the blackbox mode uses one small and one big scale up, in terms of deployment size, to compensate for the response time violations, which leads to a high deployment utilization. In contrast, the whitebox mode is able to use the infrastructure resources more efficiently by using more machines for the first scale up, and an additional scale up for a shorter period of time. Therefore, for the second pyramid-like load change, the whitebox mode uses in total less resources, but again generates more response time violations. After comparing both modes when using the pyramid-like load pattern, we can conclude that on the one hand the whitebox mode in total uses approximately 5% less resources in terms of utilized total CPU. However, on the other hand the blackbox mode produces approximately 30% less response time violations.

To summarize the results, we see that both proposed optimization approaches allow for evolving the application deployment topology at runtime. However, by comparing the results of both scenarios we see that choosing an optimal optimization approach depends on various factors, such as the expected load on the application, and the tradeoff between application performance violations (i.e., response time) and cost benefit by using less infrastructure resources.

7 Related Work

In the literature the overall terminology of IoT is well-defined [11], [12]. However, the characterization of IoT applications is not that clear. First, IoT applications can be defined as applications that hide the underlying IoT infrastructure by introducing an abstraction layer [13], [14], [15] and on top of that layer execute business logic in the cloud [16]. Second, there are distributed applications that consist of an enterprise application for managing underlying devices, and simple application parts that reside in components that are deployed in the edge infrastructure and allow for sensing as well as reacting to their environment [17], [18]. Both approaches have in common that devices, which are deployed in the IoT infrastructure, are defined as external dependencies. Hence, these devices are not considered as an integral part when designing and developing an application. In order to address this issue, recent approaches explicitly respect IoT devices as part of the application that require efficient management in order to provide scalable as well as flexible IoT applications [19], [20]. However, none of the approaches discussed so far consider provisioning and deploying parts of the application on resource-constrained devices that provide limited execution environments [21], which would help facilitating this untapped processing power for building robust and adaptable applications. For the actual deployment of applications, there exists only a limited amount of prior work (e.g., [22], [23], [24], [25]) in the literature that deal with the location-aware placement of cloud application components. In contrast to our approach,

these approaches do not support placing application components on constrained edge infrastructures in order to allow for improving the deployment topology of an application.

Additionally, since our approach also allows for optimizing an application deployment topology, we also have to consider relevant work in this research topic. There is a significant body of work on optimization algorithms for adapting deployments of cloud applications. Among others (e.g., [26], [27], [28]), Emeakaroha et al. [29] present a scheduling heuristic for cloud applications that considers several SLA objectives. The approach provides a mechanism for load balancing the execution of an application across available cloud resources, as well as a feature for automatically leasing additional cloud resources on demand. Wada et al. [30] propose an evolutionary deployment optimization for cloud applications. By introducing a multi objective genetic algorithm, the authors are able to optimize the application deployment to satisfy SLAs under conflicting quality of service objectives. Frey et al. [31] introduce CDOXplorer, a simulation-based genetic algorithm for optimizing the deployment architecture and corresponding runtime configurations of cloud applications. By applying techniques of the search-based software engineering field, CDOXplorer analyzes the fitness of a simulated set of possible application configurations, in order to allow for optimizing the overall application. In contrast to our work, none of the approaches presented so far, considers application topologies that are deployed on edge devices, and therefore can be seen as supplemental approaches to DIANE's notion of IoT deployments.

Next to algorithms, several approaches emerged in the literature that are specifically targeted at adapting application deployments in the cloud. For example, CloudScale [32] is a middleware for building applications that are deployed on and running in the cloud. By using a transparent approach, CloudScale enables the development of cloud applications like regular programs without the need for explicitly dealing with the provisioning of cloud resources. In order to scale applications, CloudScale provides a declarative deployment model that enables operators to define requirements and corresponding policies. Menasce et al. [33] present Sassy, a framework that enables applications to be self-adaptive and self-optimizing. Based on a self-architecting approach, Sassy provides a near-optimal application deployment by considering both quality of service and functional requirements. Compared to our approach, all these platforms have in common that they transparently adapt the application topology by optimizing the underlying cloud deployment. However, by only focussing on one specific type of infrastructure (i.e., the cloud), these platforms do not provide a generic approach that can also be used for optimizing application deployments on edge infrastructures as proposed in this paper.

8 Conclusion

In order to sense and manipulate their environment, applications in the Internet of Things (IoT) are required to integrate and manage a large number of heterogenous devices, which traditionally serve as simple sensors and actuators. Recently, however, devices emerged that in addition to basic sensing and actuating features, also provide constrained execution environments with limited processing, memory, and storage capabilities. To exploit this untapped processing power, applications can offload parts of their business logic onto edge devices. This offloading of application components not only increases the robustness of the overall application deployment, but also allows for cutting down costs by reducing expensive cloud to edge communication overhead. The consideration of edge devices is especially important for IoT applications that are deployed in the cloud, as the cloud allows applications to react to changing requirements by elastically adapting their overall 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 latencies need to be explicitly considered in the design of IoT cloud applications. Furthermore, applications need to be designed as clearly separated components that can be deployed independently. This application design approach enables the flexible provisioning of applications whose deployment topology evolves by dynamically offloading components to edge devices. To support this, we introduced DIANE, an approach that dynamically generates 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 allow for flexible provisioning of application components on both, cloud infrastructure, as well as edge devices deployed in the IoT infrastructure. In addition, DIANE provides an optimization approach that allows for evolving application deployment topologies at runtime to enable applications to autonomously react to environmental changes (e.g., changing request patterns).

In our ongoing work, we plan to extend DIANE to address further challenges. We plan to further adapt our MADCAT unit methodology to allow for more detailed descriptions of application topologies and enable local coordination of topology changes among edge devices. We will further investigate ideal intervals for data collection as well as fault tolerance and mitigation strategies for all DIANE components. 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 [34].

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 *Proc. Int. Symp. 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," *Comm. ACM*, vol. 53, no. 4, pp. 50–58, 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., 2015.
[5] M. Vögler, F. Li, M. Claeßens, J. M. Schleicher, S. Nastic, and S. Sehic, "COLT Collaborative Delivery of lightweight IoT Applications," in *Proc. Int. Conf. IoT as a Service*, ser. IoTaaS'14. Springer, 2014, p. to appear.

[6] M. Vögler, J. M. Schleicher, C. Inzinger, and S. Dustdar, "DIANE - Dynamic IoT Application Deployment," in *Proc. Int. Conf. Mobile Services, Special Track - Services for the Ubiquitous Web*. IEEE, 2015, pp. 298–305.
[7] 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 *Proc. Int. Symp. Service-Oriented System Engineering*, ser. SOSE'14. IEEE, 2014, pp. 13–22.
[8] C. Inzinger, W. Hummer, B. Satzger, P. Leitner, and S. Dustdar, "Generic event-based monitoring and adaptation methodology for heterogeneous distributed systems," *Software: Practice and Experience*, vol. 44, no. 7, pp. 805–822, 2014.
[9] M. Vögler, J. M. Schleicher, C. Inzinger, and S. Dustdar, "Ahab: A Cloud-based Distributed Big Data Analytics Framework for the Internet of Things," *Software: Practice and Experience*, p. to appear, 2016.
[10] D. Agrawal, S. Das, and A. El Abbadi, "Big data and cloud computing: new wine or just new bottles?" *Proc. VLDB Endowment*, vol. 3, no. 1-2, pp. 1647–1648, Sep. 2010.
[11] S. Li, L. D. Xu, and S. Zhao, "The internet of things: a survey," *Information Systems Frontiers*, pp. 1–17, Apr. 2014.
[12] L. Da Xu, W. He, and S. Li, "Internet of Things in Industries: A Survey," *IEEE Trans. Ind. Informat.*, vol. 10, no. 4, pp. 2233–2243, 2014.
[13] 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.
[14] P. Patel, A. Pathak, T. Teixeira, and V. Issarny, "Towards application development for the internet of things," in *Proc. Middleware Doctoral Symp.*, ser. MDS'11. ACM, 2011, pp. 5:1–5:6.
[15] H. Ning and Z. Wang, "Future Internet of Things Architecture: Like Mankind Neural System or Social Organization Framework?" *IEEE Commun. Lett.*, vol. 15, no. 4, pp. 461–463, 2011.
[16] F. Li, M. Vögler, S. Sehic, S. Qanbari, S. Nastic, H.-L. truong, and S. Dustdar, "Web-Scale Service Delivery for Smart Cities," *IEEE Internet Comput.*, vol. 17, no. 4, pp. 78–83, 2013.
[17] W. Colitti, K. Steenhaut, N. De Caro, B. Buta, and V. Dobrota, "REST Enabled Wireless Sensor Networks for Seamless Integration with Web Applications," in *Proc. Int. Conf. Mobile Adhoc and Sensor Systems*, ser. MASS'11. IEEE, 2011, pp. 867–872.
[18] Q. Zhu, R. Wang, Q. Chen, Y. Liu, and W. Qin, "IOT Gateway: Bridging Wireless Sensor Networks into Internet of Things," in *Proc. Int. Conf. Embedded and Ubiquitous Computing*, ser. EUC'10, 2010, pp. 347–352.
[19] S. S. Yau and A. B. Buduru, "Intelligent Planning for Developing Mobile IoT Applications Using Cloud Systems," in *Proc. Int. Conf. Mobile Services*, ser. MS'14, 2014, pp. 55–62.
[20] F. Li, M. Vögler, M. Claessens, and S. Dustdar, "Towards Automated IoT Application Deployment by a Cloud-Based Approach," in *Proc. Int. Conf. Service-Oriented Computing and Applications*, ser. SOCA'13, 2013, pp. 61–68.
[21] A. Sehgal, V. Perelman, S. Kuryla, and J. Schonwalder, "Management of resource constrained devices in the internet of things," *IEEE Commun. Mag.*, vol. 50, no. 12, pp. 144–149, 2012.
[22] R. Buyya, R. N. Calheiros, and X. Li, "Autonomic Cloud computing: Open challenges and architectural elements," in *Proc. Int. Conf. Emerging Applications of Information Technology*, ser. EAIT'12, 2012, pp. 3–10.
[23] S. Radovanovic, N. Nemet, M. Cetkovic, M. Z. Bjelica, and N. Teslic, "Cloud-based framework for QoS monitoring and provisioning in consumer devices," in *Proc. Int. Conf. Consumer Electronics*, ser. ICCE'13, 2013, pp. 1–3.
[24] H. Qian and M. Rabinovich, "Application Placement and Demand Distribution in a Global Elastic Cloud: A Unified Approach," in *Proc. Int. Conf. Autonomic Computing*, ser. ICAC'13. USENIX Assoc., 2013, pp. 1–12.
[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] J. Z. W. Li, M. Woodside, J. Chinneck, and M. Litoiu, "CloudOpt: Multi-goal optimization of application deployments across a cloud," in *Proc. Int. Conf. Network and Service Management*, ser. CNSM'11. IFIP, 2011, pp. 1–9.
[27] P. Leitner, W. Hummer, B. Satzger, C. Inzinger, and S. Dustdar, "Cost-Efficient and Application SLA-Aware Client Side Request Scheduling in an Infrastructure-as-a-Service Cloud," in *Proc. Int. Conf. Cloud Computing*, ser. CLOUD'12. IEEE, 2012, pp. 213–220.
[28] W. Yuan, H. Sun, X. Wang, and X. Liu, "Towards Efficient Deployment of Cloud Applications through Dynamic Reverse Proxy Optimization," in *Proc. Int. Conf. High Performance Computing and Communications & Int. Conf. Embedded and Ubiquitous Computing*. IEEE, 2013, pp. 651–658.
[29] V. C. Emeakaroha, I. Brandic, M. Maurer, and I. Breskovic, "SLA-Aware Application Deployment and Resource Allocation in Clouds," in *Proc. Computer Software and Applications Conference Workshops*, ser. COMPSACW'11. IEEE, 2011, pp. 298–303.
[30] H. Wada, J. Suzuki, Y. Yamano, and K. Oba, "Evolutionary deployment optimization for service-oriented clouds," *Softw. Pract. Exper.*, vol. 41, no. 5, pp. 469–493, 2011.
[31] S. Frey, F. Fittkau, and W. Hasselbring, "Search-based genetic optimization for deployment and reconfiguration of software in the cloud," in *Proc. Int. Conf. on Software Engineering*, ser. ICSE'13, 2013, pp. 512–521.
[32] P. Leitner, B. Satzger, W. Hummer, C. Inzinger, and S. Dustdar, "CloudScale - a Novel Middleware for Building Transparently Scaling Cloud Applications," in *Proc. Symp. on Applied Computing*, ser. SAC'12. ACM, 2012, pp. 434–440.
[33] D. a. Menascé, H. Gomaa, S. Malek, and J. P. Sousa, "Sassy: A framework for self-architecting service-oriented systems," *IEEE Softw.*, vol. 28, no. 6, pp. 78–85, 2011.
[34] J. M. Schleicher, M. Vögler, C. Inzinger, and S. Dustdar, "Towards the internet of cities: A research roadmap for next-generation smart cities," in *Proc. Intl. Workshop on Understanding the City with Urban Informatics*. ACM, 2015, pp. 3–6.

Michael Vögler is a researcher at the Distributed System Group at TU Wien. His research interests are cloud computing, service-oriented architectures, distributed systems, and IoT.

Johannes M. Schleicher is a PhD student at the Distributed System Group at TU Wien. His research interests are cloud computing, distributed systems and smart cities.

Christian Inzinger is a postdoctoral researcher at the software evolution and architecture lab (s.e.a.l.) at University of Zurich. His main research focus is on helping developers write better cloud applications and his work is mainly concerned with architectures for cloud applications, software evolution, and fault management in distributed elastic systems.

Schahram Dustdar is a full professor of computer science with a focus on Internet technologies and heads the Distributed Systems Group at TU Wien. He is an ACM Distinguished Scientist and recipient of the IBM Faculty award. He is an Associate Editor of IEEE Trans. on Services Computing, ACM Trans. on the Web, and ACM Trans. on Internet Technology and on the editorial board of IEEE Internet Computing. He is the Editor-in-Chief of Computing (Springer).