A Middleware Infrastructure for Utility-based Provisioning of IoT Cloud Systems

Stefan Nastic, Hong-Linh Truong, and Schahram Dustdar
Distributed Systems Group, TU Wien, Austria
Email: {*lastname*}@*dsg.tuwien.ac.at*

***Abstract*—Recently, we have witnessed numerous benefits of exploiting Cloud Computing models and technologies in the context of the Internet of Things and Edge Computing. However, utility-based provisioning paradigm, one of the most important properties of Cloud Computing, is yet to be realized in emerging IoT Cloud systems. In this paper, we introduce a novel middleware, which provides comprehensive support for multi-level provisioning of IoT Cloud systems. The main features of our middleware include: i) A generic, light-weight resource abstraction mechanism, which enables application-specific customization of Edge devices; ii) Support for automated provisioning of Edge resources and application components in a logically centralized manner, via dynamically managed APIs; and iii) Flexible provisioning models that enable self-service, on-demand consumption of the Edge resources. We evaluate our middleware using real-life applications in the domain of building management systems.**

I. Introduction

Recently, Cloud Computing and the Internet of Things (IoT) have been converging ever stronger, sparking creation of very large-scale, geographically distributed systems, called *IoT Cloud systems* [1]–[3]. IoT Cloud systems intensively exploit Cloud Computing models and technologies, predominantly by utilizing large and remote data centers, but also nearby Cloudlets [4], [5] to enhance resource-constrained Edge devices (e.g., in terms of computation offloading [6]–[8] and data staging [9]) or to provide an execution environment for cloud-centric IoT applications [10], [11].

One of the main advantages of Cloud Computing is reflected in its support for self-service, on-demand resource consumption, where users can dynamically allocate appropriate amount of infrastructure resources (e.g., computing or storage) required by an application [12], [13]. To date, we have witnessed numerous benefits of this utility-based provisioning model in terms of more flexible and cheaper IT operations [14]. Therefore, it would be natural to expect that this flagship property of Cloud Computing would be inherited by IoT Cloud, as well. Unfortunately, this is still not the case, because current approaches dealing with IoT Cloud provisioning mostly focus on providing virtualization solutions for the Edge devices, such as IoT gateways [15]–[17]. Although device virtualization is one of the preconditions for utility-based provisioning, such approaches are usually intended to support a specific task, e.g., data integration or data-linking, and largely rely on rigid provisioning models. This inherently prevents consuming IoT Cloud infrastructure resources as generic compute or storage utilities and requires rethinking existing support for: i) representing the IoT Cloud infrastructure resources, ii) managing their configuration and deployment models, as well as iii) composing low-level resource components into usable infrastructures, capable to support novel application requirements.

In this paper, we continue our line of research towards utility-based provisioning of IoT Cloud systems by introducing a novel provisioning middleware for IoT Cloud. Our middleware builds on the previously introduced concepts and frameworks [18], [19], extending them with comprehensive support for scalable multi-level provisioning of IoT Cloud systems. The main features of our middleware include: i) A generic, light-weight resource abstraction mechanism, based on software-defined gateways, which enables application-specific customization of Edge devices; ii) Support for automated provisioning of Edge resources and application components in a logically centralized manner, via dynamically managed APIs; and iii) Flexible provisioning models that enable self-service, on-demand consumption of the Edge resources.

The remainder of the paper is organized as follows: Section II presents a motivating scenario, background and main research challenges. In Section III, we introduce our middleware and discuss its architecture in detail. Section IV outlines the main runtime mechanism for multi-level provisioning. Section V describes experimental results and outlines current prototype implementation. In Section VI, we discuss the related work. Finally, Section VII concludes the paper and gives an outlook of our future research.

II. Motivation & Background

A. Scenario

Let us consider a case of Building Management System (BMS), which provides applications to manage various building facilities, such as HVAC systems, elevators and emergency alarms. SAPP is a BMS application developed in collaboration with our industry partners[1]. It is written in Sedona [20] and it is responsible to monitor environmental conditions within buildings and to regulate the HVAC facilities accordingly. Figure 1 shows a high-level system architecture of SAPP application. The solid arrows show typical propagation of sensory data within the SAPP. The dashed elements represent IoT Cloud infrastructure resources and the dashed arrows

[1] http://pcccl.infosys.tuwien.ac.at/

978-1-5090-3322-5/16 $31.00 © 2016 IEEE
DOI 10.1109/SEC.2016.35

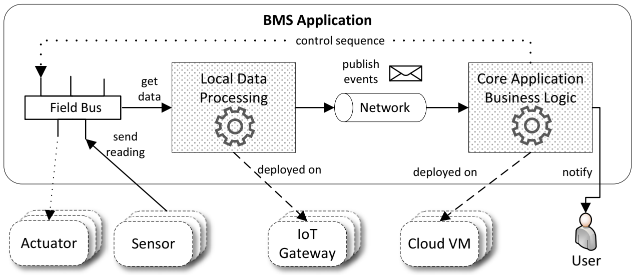

Fig. 1. Overview of SAPP architecture.

illustrate deployment of main application components. SAPP is highly distributed and executes atop both the Edge devices (e.g., IoT gateways) and the Cloud infrastructure. Generally, the Edge part of application's business logic is responsible for obtaining sensory readings from building facilities and performing initial data processing. The processed data is then transmitted over the network (Wi-Fi or 3G) to the Cloud for further processing. Finally, SAPP's Cloud services invoke actions, such as to notify a user or to perform remote actuation, e.g., regulate water pressure.

The BMS is a large-scale, geo-distributed system responsible for managing hundreds of buildings. It executes atop complex IoT Cloud infrastructure that includes: (i) various Edge devices, such as sensors, actuators and gateways, which are installed throughout the buildings, (ii) network elements, and (iii) cloud services, e.g., for complex event processing, NoSQL data storage, and streaming data analysis. Since BMS service provider is usually not the owner of physical infrastructure, e.g., Edge devices installed in buildings, the IoT Cloud infrastructure is consumed by BMS as a utility and it needs to be dynamically provisioned to satisfy BMS requirements. In this context, we define infrastructure provisioning in a broader sense [21], particularity involving a set of activities performed by developers and operations managers to prepare infrastructure resources and system/application components, bringing the system to a state where it is usable for the end user.

Unfortunately, most of contemporary approaches dealing with IoT Cloud provisioning [15]–[17] provide only partial solutions in terms of available tools, frameworks and middleware. These approaches do not fully account for the inherent properties of IoT Cloud infrastructures, such as heterogeneity, geographical distribution, and the sheer scale of such infrastructures. As a result, system integrators and operations managers have to rely on provisional solutions, which require combining multitude of provisioning techniques, such as manual, script- and service-based provisioning. Additionally, many of these approaches implicitly assume manual logging into Edge devices or even physical on-site presence, making them hardly feasible in practice. The issue is further exacerbated due to a strong dependence of IoT Cloud applications on specific properties of the underlying Edge devices (e.g., available sensors) and novel resource features, which also need to be considered during application provisioning. Therefore, consuming IoT Cloud infrastructure as a utility and provisioning even a simple application such as our SAPP is a challenging task.

B. Background

Previously we have introduced a conceptual model for software-defined IoT Cloud systems [18]. The core concept of the provisioning model is *software-defined IoT unit*. The software-defined IoT units describe IoT Cloud resources (e.g., virtual sensors), their runtime environments (e.g., IoT gateways) and capabilities (e.g., communication protocols or data point controllers). Such units are used to encapsulate infrastructure resources and to abstract their provisioning in software. To this end, they expose well-defined APIs and can be composed at different levels, creating virtual runtime infrastructures for IoT Cloud applications. The main purpose of such software-defined IoT Cloud infrastructures is to enable *utility-based provisioning of IoT Cloud resources* by providing a uniform view on the entire resource pool, as well as by allowing IoT Cloud applications to customize and consume those resources dynamically and on-demand.

In [19], we have presented a cloud-based provisioning framework for provisioning IoT Cloud applications. The framework provides a scalable provisioning controller, which is responsible for managing, deploying and installing application components in Edge devices. Further, the introduced framework provides mechanisms to manage the Edge devices, e.g., to register new devices and detect disconnected devices. One of the distinguishing features of IoT Cloud systems is infrastructure virtualization layer and there is a number of existing approaches that deal with Edge devices virtualization, exposing such devices to the upper layers on different levels of abstraction. However, suitable tools and frameworks that provide support for managing the virtualized IoT Cloud resources remain largely underdeveloped. Therefore, a comprehensive *provisioning middleware* is required in order to provide a uniform representation of the underlying (virtual) infrastructure resources, as well as to enable utility-based delivery and consumption of such resources in a logically centralized manner.

C. Research Challenges and Provisioning Middleware Requirements

Utility-based provisioning is a well-established and proven concept in Cloud Computing [12], [22]. Among other things it requires: on-demand, self-service usage models; enabling ubiquitous access to a shared pool of configurable resource, which can be customized to exactly meet application requirements; as well as autonomous and automated allocation of the consumed resources. However, given the previously described properties of IoT Cloud, realizing these features in the context of IoT Cloud systems is a non-trivial task creating a number of challenges that need to be addressed.

One of the main challenges is to support *on-demand, self-service usage model*, because it requires support for uniform interactions with the large-scale, heterogeneous IoT Cloud resource pool. This could potentially be achieved by virtualizing and encapsulating the IoT Cloud resources into well-defined APIs and allowing users to access such resources on multiple levels of abstraction. However, in this case provisioning middleware needs to provide support for a non-trivial task of managing such virtual resources, their APIs and mediating all the communication with heterogeneous devices.

Assuming that IoT Cloud resources are accessible in a uniform manner, another challenge is to enable the users to *automatically provision IoT Cloud resources*. However, strong dependencies of IoT Cloud applications on specific properties of the underlying devices and novel resource features intrinsically prevent from consuming IoT Cloud infrastructure as traditionally generic compute or storage utilities. This requires providing comprehensive provisioning support on multiple levels such as infrastructure-, platform- and application-level. One way to achieve this is to utilize provisioning workflows [23]. The main advantage of the workflow approach is that it supports nested provisioning workflows, which are well suited for multi-level provisioning. However, to support execution of the provisioning workflows for a large resource pool the middleware needs to enable elastically scalable execution of provisioning tasks.

Enabling ubiquitous access to the large, geographically-distributed resource pool is yet another challenge since it demands a *logically centralized interaction* with underlying devices. However, since the underlying devices are inherently dispersed, the middleware needs to be distributed across the resource-constrained devices, thus optimized for such constrained execution environments. Moreover, to support customizing such resources, the middleware needs to support *management of application components and configuration models*, but it also needs to provide suitable mechanisms to dynamically deliver them to the Edge devices.

III. IoT Cloud provisioning middleware

The main purpose of our provisioning middleware is to facilitate implementing and executing provisioning workflows in IoT Cloud systems, by addressing the previously-described challenges. To this end, the middleware provides a set of runtime mechanisms and does most of the "heavy lifting" to support the users in implementing and executing provisioning workflows in large-scale software-defined IoT Cloud systems, without worrying about scale, geographical distribution and heterogeneity of those systems. The support for the multi-level provisioning is thoroughly discussed in Section IV. At the moment it is important to note that IoT Cloud provisioning involves two main tasks: i) *allocating and deploying Software-Defined Gateways (SDG)*, which are special type of aforementioned software-defined IoT units and ii) *customizing software-defined gateways with application-specific artifacts*.

Figure 2 gives a high-level architecture overview of our middleware. Generally, the provisioning middleware is designed based on the microservices architecture [24] and it is distributed across the Cloud and Edge devices. The main components of the provisioning middleware include: i) the *Software-Defined Gateways*, ii) the *Provisioning and Virtual Buffers Daemons* that run in Edge devices and iii) the *Provisioning Controller* which runs in the Cloud. In the remainder of this section, we discuss these components in more detail.

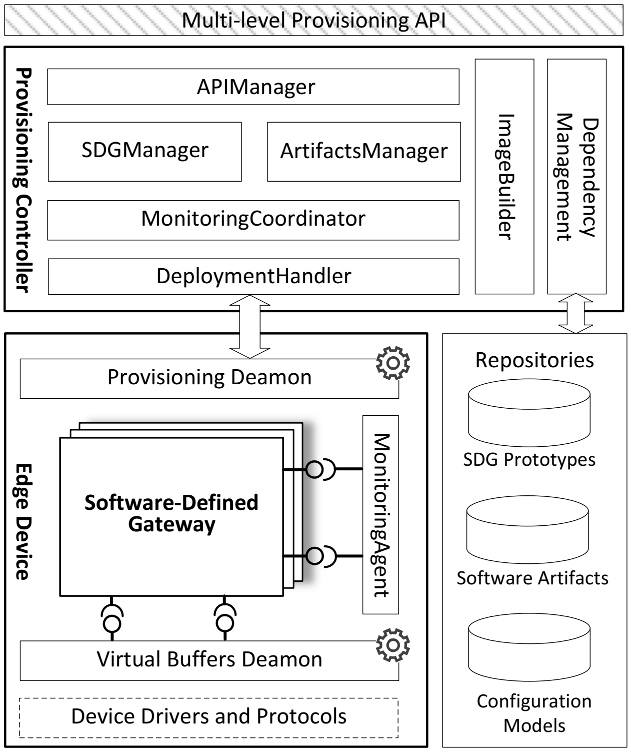

Fig. 2. Architecture overview of the provisioning middleware.

A. Software-defined Gateways

Software-defined gateways (SDGs) are one particular type of our software-defined IoT units. Their main purpose is to support virtualizing IoT Cloud compute resources, most notably Edge devices, in order to provide isolated and managed application execution environments. Our middleware does not support building custom SDGs from scratch. Instead it provides SDG prototypes and mechanisms to customize them, based on application-specific requirements. At their core the SDG prototypes define an isolated runtime environment for the SDGs and application-specific components. The main purpose of the SDG prototypes is to provide isolated namespaces, as well as limit and isolate resource usage, such as CPU and memory. Therefore, the SDG prototypes are used to bootstrap higher-level SDG functionality. In Figure 3 the double line shows virtual boundaries of the SDG prototypes. It is important to mention that SDG prototypes do not propose a novel virtualizaton solution. Instead they rely on proven techniques, namely kernel-supported virtualization approaches, which offer a number of light-weight execution environments/drivers such as LXCs, libvirt-sandbox or even chroot. Such environments are generally referred to as containers that can be used

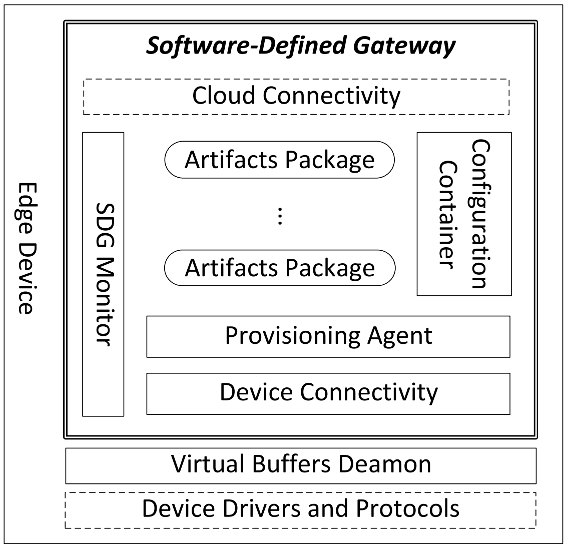

Fig. 3. Software-defined gateway architecture.

to "wrap" the SDGs. Conceptually, virtualization choices do not pose any limitations, because by utilizing well-defined APIs, our SDGs can be dynamically configured, provisioned, interconnected and deployed, at runtime. The SDG prototypes are hosted in the IoT Cloud and enriched with functional and provisioning capabilities, which are exposed via the well-defined APIs. There is a number of middleware components (cf. Figure 3), which are preinstalled in each SDG prototype in order to support such APIs. Next, we discuss these components in more detail.

1) Artifact Packages: Generally, IoT Cloud applications consist of different application components and supporting files (e.g., libraries and binaries), which we refer to as application-specific artifacts. Such artifacts are deployed, configured and executed inside software-defined gateways. Our provisioning middleware does not make any assumptions about application model or concrete artifact implementations. However, in order to enable automated artifacts provisioning, it requires them to be packaged as shown in Figure 4. There are two important things to mention here. First, the Artifact Package needs to contain a set of provisioning directives with all the necessary instructions, such as installing and uninstalling the package. When a provisioning workflow submits a provisioning directive, the middleware maps the request to a concrete implementation of the provisioning directive. To support implementing such directives, in our previous work we have provided a light-weight provisioning DSL [25]. Second, the packages contain meta-information such as artifacts' hardware requirements and exposed APIs. The specification of the APIs is optional, but it is needed by the middleware if an application decides to delegate management of its configuration models, as we discuss subsequently.

2) Configurations Container: In order to support centrally managed configuration models and dynamic feature composition, besides managing application binaries our provisioning middleware is responsible to maintain application-specific configurations. As shown in Figure 4, application configuration models are treated as special components of artifact packages. By decoupling the configuration models from the functional artifacts, we can treat them as any software-defined IoT unit. Each SDG is equipped with a *Configurations Container* that is responsible to store configuration models, actively listen for configuration changes (when new or updated configuration models are registered) and apply the new configuration directives, e.g., by restarting an OS service.

In order to support a full fledged, dynamic feature composition, the configuration container can act as a plug-in system, based on the inversion of control principles. It provides mechanisms to bind application artifacts, based on supplied configurations or to redefine them when the configurations change. The container initially binds functional artifacts based on the configuration models and continuously listens for configuration changes applying them on the affected functional artifacts accordingly. The runtime changes are achieved by invalidating affected parts of the existing dependency tree and dynamically rebuilding them, based on new configuration directives. This feature is especially useful for managing communication protocols, which are provided by *Cloud and Device Connectivity* components (cf. Figure 3). However, to support dynamic feature composition, our middleware requires application artifacts to be wrapped in well-defined APIs, which are known to the provisioning container. Since this imposes some limitations, this feature is optionally provided by our middleware. The main advantage of this approach is that it allows for updating only the configuration models without updating the entire artifact package, thus enabling flexible customizations and dynamic configuration changes without runtime interrupts as well as significantly reducing communication overhead.

3) Provisioning Agent: All packages that are not preinstalled in Edge devices have to be provisioned by the middleware during runtime. For this purpose, our middleware provides a light-weight *Provisioning Agent*, which is preinstalled inside SDGs. The agent continuously runs in each SDG and manages local artifact packages. The main responsibility of the provisioning agent is to periodically inspect the Provisioning Controller's (cf. Figure 2) update queue, download the artifact packages and execute directives referenced in provisioning workflows. Additionally, the agent acts as a local interpreter of the provisioning directives specified via our provisioning

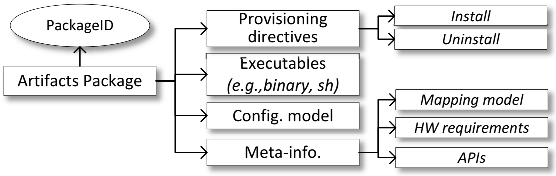

Fig. 4. Artifacts package structure.

DSL [25]. Finally, the agent is responsible to handle various requests initiated by the Provisioning Controller. This is achieved by triggering the required actions in SDGs such as creating a snapshot of the current device state via the SDGMonitor and uploading the snapshot to the Controller. The SDGMonitor is discussed in more detail later in this section.

4) Device Connectivity: The SDGs are deployed to Edge devices with limited privileges, in the sense that they are not permitted to directly access the hardware. An obvious reason for such limitation is security, but also resource contentions and customization requirements, since we can have multiple SDGs executing simultaneously in the same Edge device. In order to enable applications to access the underlying devices, e.g., sensors, SDGs offer the *Device Connectivity* component. The main part of the device connectivity component is an SDG endpoint, which exposes the devices to SDGs and enables service-based interaction with them. The SDG endpoint is a single point of interaction with the underlying *Virtual Buffers Daemon* (cf. Figure 3) and at the moment, it is defined up to the transport layer. For this reason the device connectivity component provides a pluggable connectivity layer, which is by default preconfigured with our custom, REST-like application-level protocol. In the current prototype we also support CoAP and MQTT communication protocols, but the device connectivity can be easily extended by plugging in other application-level protocols, such as sMAP [26].

B. Edge Device Middleware Support

In order to support management of SDGs in Edge devices, our middleware provides light-weight components that are preinstalled and continuously run inside the Edge devices. The most important components are the Virtual Buffers Daemon and the Provisioning Daemon, shown in Figure 2 on the right-hand side.

1) Virtual Buffers Daemon: We have discussed how our software-define gateways can be used to virtualize Edge devices compute resources. However, since SDGs run with reduced privileges, the middleware also needs to virtualize accessing the low-level devices such as sensors and actuators. To this end it provides the Virtual Buffers Daemon (VBD). The main purpose of the VBD is to mediate the communication with the devices connected to a field bus (e.g., via Modbus, CAN, SOX/DASP, I^2C or IP-based) and to provide a virtually exclusive access to such device. In general, the VBD acts as a multiplexer of the data and control channels, enabling the SDGs to have their own view of and define custom configurations for such channels. For example, a software-defined gateway can configure sensor poll rates, activate a low-pass filter for an analog sensory input or configure unit and type of data instances in the stream.

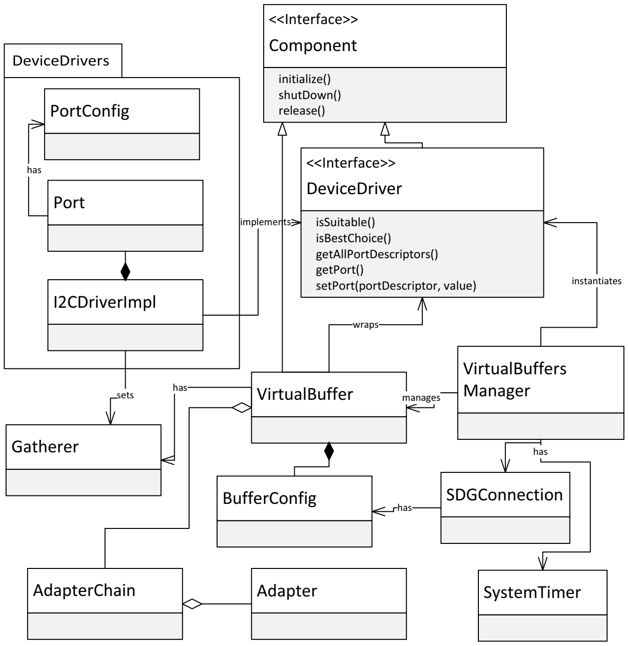

Fig. 5. Simplified UML diagram of Virtual Buffers Daemon.

Figure 5 depicts a simplified UML diagram of the VBD's most important components. The main concept behind VBD is the VirtualBuffer. Generally, the main goal of the virtual buffers is to provide virtual representation of sensors and actuators. They wrap the DeviceDrivers and share common behavior with them, which is inherited through the Component Interface. For example, they can be initialized, shutdown and released. Both the buffers and the drivers lifecycle is managed by the VirtualBuffersManager. Moreover, a virtual buffer is associated with a set of Gatherers and can contain an optional AdapterChain. Generally, a gatherer is a higher level representation of a port. For example, in case of a sensing device the gatherer represents the most recent value of the hardware interface. To support SDG-specific configurations such as sensor poll rate, filters or scalers, each virtual buffer can have an AdapterChain associated with it. Adapter chains reference different Adapters, which are specified and parametrized via the BuferConfig. For example, a raw sensing value is passed through the adapter chain before being delivered to a SDG. Finally, the VBD is responsible to instantiate and maintain an open communication channel with software-defined gateways (via the SDGConnection) and keep track of the mappings among the SGDs and their VirtualBuffers.

2) Provisioning Daemon: So far, we have tacitly assumed that SDGs are readily available in Edge devices. However, this is naturally not the case and the SDGs need to be dynamically allocated, instantiated and deployed to the Edge devices. These tasks are shared responsibility of the Provisioning Daemon and the Provisioning Controller.

At its core the Provisioning Daemon has a light-weight httpd server that enables a bidirectional communication between the Provisioning Controller and the Edge devices (i.e. SDGs). It is designed as a pluggable component, which relies on the existing support for managing shared hosting domains (i.e., containers) such as Docker, LXD or virsh. The most important components of the Provisioning Daemon are an invocation mapper and a set of plug-in components called Connectors.

Among other things, the invocation mapper is responsible to handle the provisioning directives form the controller and map them to the corresponding Connector, as well as to obtain the required SDG prototypes form the Repositories (cf. Figure 2) and locally manage their images. The connectors act as wrappers of the underlying mechanisms for managing SDGs, exposing them to the invocation mapper via uniform APIs. Therefore, to use a different virtualization solution for SDGs, one only needs to develop the needed connector and register it with the invocation mapper.

Moreover, the Provisioning Daemon mediates the communication with the SDG provisioning agents. To support this, it manages local network interfaces of SDGs and behaves like a transparent proxy for all inbound communication. Regarding outbound communication the Provisioning Daemon treats monitoring responses in a particular manner. It intercepts the monitoring information delivered by SDGMonitors and enriches it with the current device state information, which is delivered by the MonitoringAgent (cf. Figure 2). The *MonitoringAgent* is used to collect meta information about the SDGs, but also to continuously monitor the underlying system via available interfaces in order to provide dynamic device information. To this end, it executes a sequence of runtime monitoring actions to complete the dynamic device state-snapshot. For example, such actions include: currently available disk space, available RAM, firewall settings, environment information, list of processes and daemons, as well as a list of currently installed and running SGDs. The created snapshots are transmitted to the Provisioning Controller periodically or on request. The device snapshot is also used by the aforementioned invocation mapper in order to determine if a new SDG can be instantiated and deployed to an Edge device. This is particularly important, since most of the current virtualization management solutions only provide a rudimentary support in this regard.

C. Cloud-based Provisioning Controller

The Provisioning Controller (cf. Figure 2 left-hand side) is the cloud counterpart part of our middleware. It provides a mediation layer that enables the users to interact with IoT Cloud in a conceptually centralized fashion, without worrying about geographical distribution and heterogeneity of the underlying Edge devices. Internally, the Provisioning Controller comprises several microservices: *APIManager, MonitoringCoordinator, SDG- and ArtifactsManager, DeploymentHandler and DependencyManagement service*. These microservices are self-contained units, which communicate over REST APIs and can be individually deployed on different cloud VMs. This enables our Provisioning Controller to support elastically scalable execution of provisioning workflows (cf. Section V), since we can dynamically spin up additional instances of microservices under heavy load and scale out the Provisioning Controller to support large number of connected Edge devices. Due to space limitations, in continuation we only describe the most important microservices of the Provisioning Controller.

The main responsibility of the *APIManger* is to manage the *Multi-level Provisioning API*, i.e., it encapsulates the middleware provisioning capabilities in well-defined APIs and handles all API calls from user-defined provisioning workflows. Although our middleware provides multi-level provisioning support, this distinction is only relevant to the middleware internal components, since the APIManager hides all such details from the users, who effectively observe only simple API calls and corresponding responses. Therefore, the APIManager is responsible to resolve incoming requests, map them to the respective handlers, i.e., *SDGManager* or *ArtifactsManager* (depending on the request type), and deliver results to the calling workflow. Among other things, the actions performed by these managers involve selecting requested SDGs or artifacts by querying the corresponding SDG- and Artifacts-Repository, building the package images and deliver them to the Edge devices. In Section IV, we describe this process in more detail.

All device state-snapshots are maintained by the MonitoringCoordinator, which manages static device meta-information and periodically sends monitoring request to the *MonitoringAgent* in order to obtain runtime snapshots of current device state. Finally, since the majority of application artifacts and SDG images are not readily available in Edge devices, the *DeploymentHandler* is responsible to deliver them to the Edge devices (i.e., *Provisioning Daemons*) or SDGs (i.e, *Provisioning Agents*) at runtime. The DeploymentHandler relies on the *DependencyManagemet service* to resolve the required artifact dependencies and *ImageBuilder* to prepare (package and compress) them into deployable images. Resolving the dependencies on the cloud is particularly useful, because it saves a lot of processing and networking, from the perspective of whole IoT Cloud infrastructure, since otherwise each Edge device would have to perform the same set of actions, e.g., downloads.

IV. Runtime mechanisms for multi-level provisioning in IoT Cloud

A. Runtime execution of provisioning workflows

In order to provision (a part of) the SAPP application (cf. Section II) a user might design a provisioning workflow shown in Figure 6 (top). Individual actions of such workflow usually reference specific provisioning capabilities, exposed via the middleware APIs, and rely on the middleware to support their execution. Usually, the main execution thread of the provisioning workflows (denoted by the solid lines in our SAPP provisioning workflow) represents provisioning directives for the infrastructure-level, such as to deploy a SDG of a specific type on an IoT gateway or spin-up a cloud-based Message Queue Broker, e.g., MQTT Broker. The sub-workflows (denoted by dashed lines in the same example) are mainly used to specify application-level provisioning directives. Generally, application-level provisioning involves deploying, configuring and starting application artifacts at the Edge. In the case of our SAPP application this involves customizing the SDGs with the application-specific artifacts, i.e., the SAPP's local monitoring service and its configuration models. Further, since the SAPP application is written in

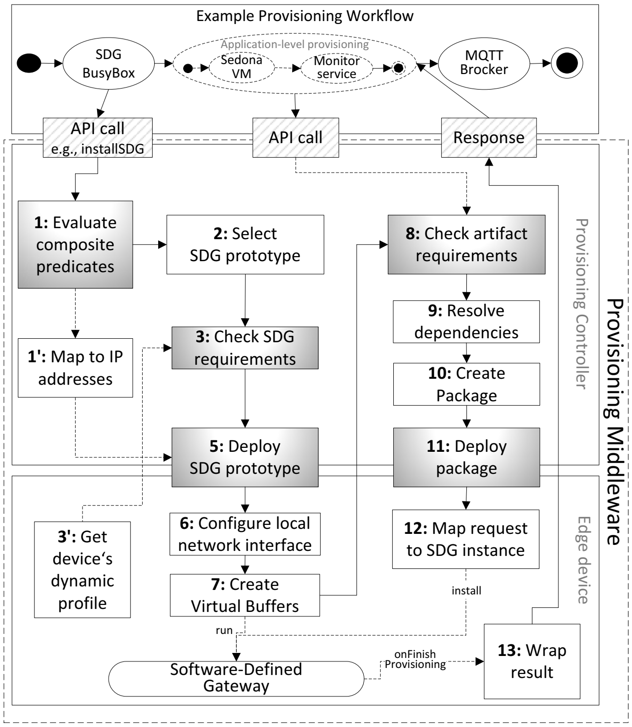

Fig. 6. Runtime execution of a provisioning workflow.

Sedona [20], the provisioning workflow also needs to deploy Sedona VM, which is a light-weight runtime, typically used in building management systems.

Figure 6 also depicts a simplified sequence of steps performed by the middleware when executing a provisioning workflow. For the sake of clarity, we omit several steps and mainly focus on showing the most common interaction, e.g., we assume no errors or exceptions occur and we do not show interaction with the Repositories. A provisioning workflow requests an application artifact or a SDG by specifying their respective IDs (currently consisting of a name and a version number) and a specific Edge device ID. Next, the workflow invokes a specific API, e.g., to install or uninstall the artifact. At this point the middleware attempts to execute the specified provisioning directive. The steps 1 to 7 in Figure 6 depict the most important actions performed by our middleware in order to support an infrastructure-level provisioning request, e.g., to deploy, instantiate and start a SDG in an Edge device. Therefore, the middleware performs the following actions: i) The *APIManager* initially evaluates the composite predicates (described later in this section) in order to determine a set of devices on witch the SGD will be deployed; ii) The *SDGManager* selects device compatible SDG prototype and registers it with the *DeploymentHandler*; iii) The *MonitoringCoordinator* together with *MonitoringAgent* checks the SDG against current device-state snapshot; iv) The *DeploymentHandler* transfers the SDG prototype image to the *Provisioning Deamon*; v) The *ProvisioningDeamon* configures the SDG's local network interface (based on the supplied mapping model), starts the SGD and registers the new SDG instance with the *Virtual Buffers Deamon*; vi) Finally, the *Virtual Buffers Deamon* allocates a set of dedicated virtual buffers and creates a dedicated SDGConnection handler. At this point the SDG instance is running in the Edge device and it is performing internal initialization actions such as starting the Configuration Container, the Provisioning Agent and its local SDG Monitor. After the final initializations the SDG transmits its initial device state to the controller and it is ready to handle application-level provisioning requests.

To support an application-level provisioning request the provisioning middleware performs the following actions (steps 8 to 13 in Figure 6): i) Similarly to the step 3 each application artifact is checked against current SDG-state snapshot, delivered by the *SDG Monitor*; ii) The *Dependency Management Service* resolves runtime dependencies of the artifact; iii) The *PackageManager* builds a deployable image and registers it with the *DeploymentHandler*; iv) Similarly to the step 5 the *DeploymentHandler* deliveres the image to the *Provisioning Deamon*; v) Finally, the *Provisioning Deamon* transparently forwards the image to the SDG's *Provisioning Agent*, which installs the package locally in the SDG. In the remainder of the section we describe the most important runtime mechanisms in more detail.

B. Evaluating composite predicates

While describing the main steps of the provisioning process, we have mostly focused on the steps performed for a single device and a single SDG. However, usually the provisioning workflows are meant to provision multiple devices, e.g., that share some common properties or belong to the same organization. Therefore, the same provisioning logic should be applicable regardless of specific devices. In this context, it is particularly important to support designing generic provisioning workflows, in the sense that such workflows should be defined independently of the Edge devices, e.g., without referencing device IDs. One of the main preconditions for this is to support the users to dynamically delimit the range of provisioning actions. In our middleware this is achieved by allowing the users to specify the required device properties, as a set of composite predicates. Such predicates reference device or SDG meta information and are used to filter out only the matching devices, which meet the specified criteria. These predicates are specified by the users and delivered to the middleware in a provisioning request as POST parameters.

To bootstrap delimiting the range of a provisioning action, our middleware maintains a set of available devices for a particular user. The current prototype always considers all the connected devices, since at the moment there is only a limited support for managing the device identities and the access control. However, this is not a conceptual drawback and there are many available solutions, which can be used to provide this functionality (as discussed in Section II). The predicates are applied on this set, filtering out all resources that do not match the provided attribute conditions. The middleware uses the resulting set of resources to initiate the provisioning actions with *SDG- and AtrifactsManager*. These managers are also responsible to provide support for gathering results delivered by the *ProvisioningDeamons* and the *ProvisioningAgents*, once the provisioning action is completed (cf. Figure 6 step 13). This is needed since after the resources are selected, provisioning actions are performed in parallel and the results are asynchronously delivered to provisioning workflows.

C. Artifacts and SDGs prototypes runtime validation

Since we are dealing with resource-constrained devices, before deploying a SDG or application artifact the middleware needs to verify that the component can be installed on a specific device, e.g, that there is enough disk space available. This happens during step 3 (Check SDG requirements) and step 8 (*Check artifact requirements*). To this end, the *MonitoringCoordinator* first queries the Repositories. Besides the artifact binaries and SDG prototypes, the repositories store corresponding meta-information, such as required CPU instruction set (e.g., ARMv5 or x86), disk space and memory requirements. After obtaining the meta-information our middleware starts building the current device state-snaphshot. This is done in two stages. First, the device features catalog is queried to obtain relevant static information, such as CPU architecture, kernel version and installed userland (e.g., BusyBox [27]) or OS. Second, the *MonitoringCoordinator* in coordination with the *MonitoringAgent* and *SDGMonitor* executes a sequence of runtime profiling actions to complete the dynamic device state-snapshot. For example, the profiling actions include: currently available disk space, available RAM, firewall settings, environment information, list of processes and daemons, and list of currently installed capabilities. Finally, when the dynamic device snapshot is completed, it is compared with the SDG's/artifact's meta information in order to determine if they are compatible with the device. In this context, the middleware performs in a similar fashion to a fail-safe iterator, in the sense that it works with snapshots of device states. For example, if something changes on the device side, during step 3 or step 8, it cannot be detected by the middleware and in this case its behavior is not defined. Since we assume that all the changes to the underlying devices are performed exclusively by our middleware, this is a reasonable design decision. Other errors, such as failure to install an artifact, in a specific SDG, are caught by the middleware and delivered as notifications to the provisioning workflow, so that they do not interrupt its execution. With this approach the middleware is capable to make autonomous decisions about the provisioned resource. This is one of the main preconditions for supporting automated execution of provisioning workflows, but also for enabling on-demand, self-service provisioning model, since our middleware does not make any implicit assumptions such as user awareness of device properties nor it requires them to manually interact with the underlying devices.

D. Provisioning models

In the following we discuss the provisioning models currently supported by the middleware prototype and discus some possible optimizations. After the *MonitoringCoordinator* determines an SDG/package is compatible with Edge devices, the middleware needs to create a SDG or Artifact image and deliver it to these devices (steps 5 and 11 in Figure 6). This process requires the middleware to make the following decisions: what to deliver to the devices, how to deliver it and where to host the image. Therefore, the image delivery process is structured along these three main phases.

1) Delivery models: In the first phase, the middleware needs to chose whether to deliver a complete image or only a download script. In the first case the *ImageBuilder* creates a SDG or an Artifact image, which is essentially a compressed Artifact Package or SDG prototype. This image is then registered with the *DeploymentHandler* by a corresponding manager, which transfers the whole image to the *ProvisioningDeamon*. In the second case the process is performed in a similar fashion, but in addition to the image the *ImageBuilder* also generates a download script. The main part of this script is an URL of the location where the actual image resides. Instead of the whole image, only this script is sent to the *ProvisioningDeamon*, so it can download and install the image. Since both of these approaches have their advantages [28], the middleware leaves it to the users to make a decision, i.e., to select the most suitable approach and pass it as a configuration parameter in the provisioning request.

2) Deployment models: In the second phase the *DeploymentHandler* deploys the image (or the download script) to the device. We support two different deployment strategies. The first strategy is poll-based, in the sense that the image is placed in a queue and remains there for a specified period of time (TTL). Both *ProvisioningDeamons* and *ProvisioningAgents* periodically inspect the queue for new provisioning requests. When a request is available, the device can poll the new image when it is ready, e.g., when the load on it is not too high. Although a provisioning workflow can specify the image priority in the queue, if a device is busy over longer period of time, e.g., there is not enough disk space to install a SDG, this can lead to a request starvation, blocking the execution of the provisioning workflow. For this reason our middleware also supports a push-based deployment. In this case, instead of waiting in the queue, an image is immediately pushed on a device. This gives a greater control to the provisioning framework, but since the previously described image runtime validation performs in a fail-safe manner, the push-based deployment can lead to an undesired behavior. Therefore, when using this strategy a provisioning workflow should also provide compensation actions, to return the device in the previous state. Naturally, these two strategies can be used to create hybrid deployment strategies, such as using the pool-based approach for SDG prototypes and the push-based approach for application artifacts, because pushing artifacts is particularly useful for security updates of hot fixes in SDGs.

3) Placement models: Finally, the middleware decides where to host the image. This largely depends on a specific deployment strategy, but also on the delivery model. For example, for push-based deployment the *DeploymentHandler* stores the images in-memory, also the download scripts are always kept in-memory, but in case of pool-based strategy, images are usually hosted in middleware local *Repositories*. However, it is not difficult to imagine more complex provisioning models, which can be put in place in order to optimize the provisioning process, e.g., to save bandwidth. For example, to achieve this, our middleware could easily utilize proven technologies such as Content Delivery Networks (CND), Cloudlets or micro data centers. One way of accomplishing this is to deliver a download script to a set of Edge devices and push an image to a Cloudlet, residing in the proximity (single-hop) of these device. The *ProvisioningDeamon* could then use the poll-based approach to obtain the image.

V. Implementation & Evaluation

A. Prototype implementation

In the current prototype, the middleware Provisioning Controller cloud-based microservices are implemented in Java (based on Java SE) and Scala programming languages. The middleware agents and the ProvisioningDeamon are implemented as Shell and Python scripts (based on light-wight httpd server). The VirtualBuffersDeamon is also implemented in Java (based on Java SE Embedded), but in this case we have created a lightweight compact profile JVM runtime [29] specifically tailored for constrained devices. The total disk size of the JVM, VirtualBuffersDeamon and all its dependencies is little over $15Mb$. The complete source code and supplement materials providing more details about current middleware implementation are publicly available in Git Hub[2].

B. Middleware performance

In the following experiments we show two main performance aspect of our provisioning middleware: support for: i) *scalable execution of the provisioning workflows* (hundreds of Edge devices) and at the same time ii) *middleware suitability for constrained devices* in terms of resource consumption, i.e., its memory and CPU usage.

1) Applications: In the experiments we used two real-life applications from a Building Management System, developed in collaboration with our industry partners[3]. For our experiments, it is important to note that the first application (SAPP) is written in Sedona [20] and it size is approximately $120Kb$, including the SVM and the application (.sab, .sax, .scode and Kits files). The second application (JAPP) is JVM-based (compact profile2) and its size including all binaries, libraries and the JVM is around $14Mb$.

Additionally, for the experiments we have developed a SDG prototype, based on BusyBox, which is a very lightweight Linux user land. The SDG prototype is specifically built for Docker's libcontainer virtualization environment and is approximately $1.4Mb$ in disk size (without applications).

2) Experiment setup: In order to evaluate middleware performance regarding resource usage, we built 15 physical gateways (cf. Figure 7) and installed them throughout our department. The getaways are based on Raspberry Pi 2, with ARMv7 CUP and 1Gb of RAM. They run Raspbian Linux 8 (based on Debian "Jessi") on Linux Kernel 4.1.

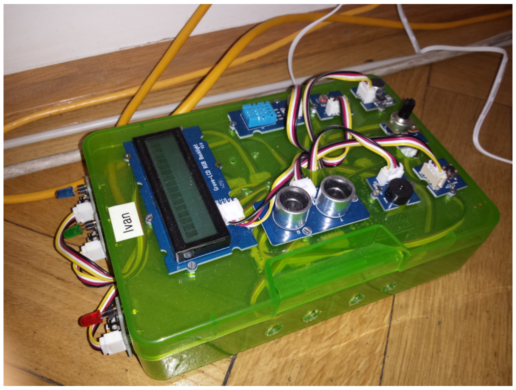

Fig. 7. An example of our gateways for Building Management Systems.

In order to evaluate how our middleware behaves in a large-scale setup, we created a virtualized IoT cloud testbed based on CoreOS [30]. In our testbed we use Docker containers to mimic physical gateways in the cloud. These containers are based on a snapshot of a real-world gateway, developed by our industry partners. For the experiments, we deployed a CoreOS cluster on our local OpenStack cloud. The cluster consists of 4 CoreOS 444.4.0 VMs (with 4 VCPUs and 7GB of RAM), each running approximately 200 Docker containers. The Provisioning Controller and the Repositories are also deployed in our Cloud on 3 Ubuntu 14.04 VMs (with 2VCPUs and 3GB of RAM).

Finally, since the physical gateways are attached to our department network, in order to connect them to the cloud network (but avoid potential security risks), we have created a network overlay based on Wave routers [31].

3) Experiments: **Middleware resource consumption at the Edge.** Initially, we show the performance of middleware most important components that continuously run in edge devices, namely the *ProvisioningDeamon* and the Virtual-

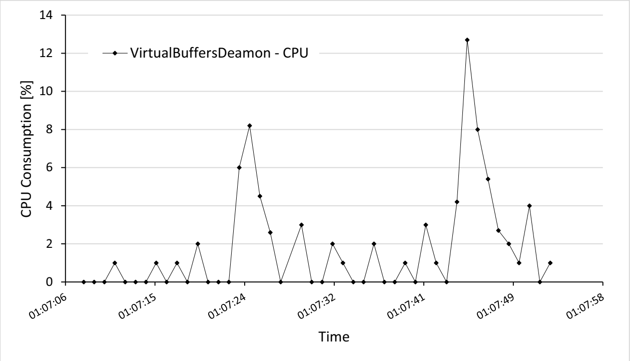

Fig. 8. CPU consumption of the VirtualBuffersDeamon.

[2] https://github.com/tuwiendsg/SDI
[3] http://pcccl.infosys.tuwien.ac.at/

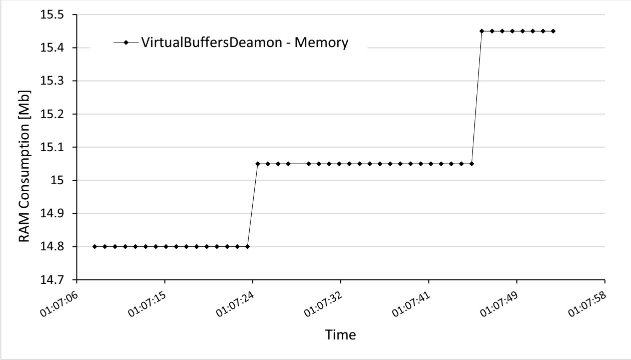

Fig. 9. Memory consumption of the VirtualBuffersDeamon.

BuffersDeamon. The *MonitoringAgent* is not considered in our experiments, since it only periodically executes to create device-state snapshots, thus it does not have statistically significant impact on the performance. Further, it is important to mention that the runtime overhead of middleware components running in the SDGs is almost negligible, since it is less than $1Mb$. The main goal of the following experiments is to demonstrate the validity of our approach w.r.t. resource-constrained devices, since we do not claim that it outperforms related approaches, which provide functionality that partially overlaps with our middleware.

Figure 8 and Figure 9 respectively show the CPU and memory usage of the VirtualBuffersDeamon, over a period of time. There are several important things to notice here. When there are no SDGs (applications) running in the gateway the deamon is mainly idle, i.e., it only periodically polls the underlying drivers for device status and on average its CPU consumption is less than 2%. This can be observed in Figure 8, before the first peak. The two peaks represent SDG deployments for the two applications. The first peak happens when the Sedona-based application is deployed and the second peak signals the deployment of Java-based application. Since SAPP requires smaller number of sensors than JAPP, the deamon needs to allocate and configure less virtual buffers, thus the difference in the two peaks. However, in both cases the maximum CPU usage of the deamon is below 14% and it lasts only a few seconds. For the same scenario we have measured the deamon's memory usage. Figure 9, shows the total memory of deamon's JVM process (with heap memory, Perm Size and stack). Initially, we notice that in the idle state the deamon consumes little bit under $15Mb$ of RAM (the initial heap size is configured to a minimum of $1Mb$), what can be considered a low memory footprint. We also observe that memory consumption behaves in a similar manner to CPU consumption. This is represented by the two distinct jumps in memory usage (cf. Figure 9). The increase in memory usage is due to newly allocated virtual buffers, adapters (heap) and SDGConnections (stack). The reason for the difference being the same as above. Finally, we notice a monotonic growth of memory usage, the reason for this is that the Deamon does not trigger garbage collection, since both SDGs are ruining and using the buffers, however after an application is stopped the deamon releases its buffers. Therefore, the performance of the VirtualBuffersDeamon can be seen as suitable for resource-constrained devices.

Figure 10 and Figure 11 show the CPU and memory usage of the ProvisioningDeamon (and the used Connector for the underlying virtualization solution). In this case we only consider infrastructure-level provisioning requests, i.e., configuring and starting SDGs, since only this type of requests are explicitly handled by the ProvisioningDeamon. In Figure 10, we notice that in general our provisioning deamon utilizes the CPU resources scarcely, namely its CPU usage is mostly around 1%. This is due to the fact that most of the time the deamon idle, it only periodically checks for new requests from the Provisioning Controller and sends a hart bit. The dramatic spikes in CPU usage happen only during the SDG deployment (we launched 4 SDGs on the gateway during the experiment), since this includes expensive network and computation operations, i.e., downloading SDG prototype, configuring it and starting it. However, the later two operations are performed by the Connectors which execute the commands and quickly terminate. Figure 11 shows the memory usage of the provisioning deamon for the same experiment. One can notice that during the experiment the memory usage of the provisioning deamon was always below $30Mb$ and more importantly shortly after an SDG is started the deamon releases the unused (Connector's) memory. Therefore, middleware Edge components in total require under $45Mb$ of memory and consume around 2% of CPU on average. We believe that this is reasonable resource utilization suitable for resource-constrained devices.

Scalable execution of provisioning workflows. The reason why we put an emphasis on the scalability of our middleware is that it is one of the key precondition for consistent realization of provisioning workflows across a large resource pool. For example, if the execution of provisioning workflows were to scale exponentially with the size of the resource pool, theoretically it would take infinitely long to have a consistent infrastructure baseline for the the whole system, given a sufficiently large resource pool.

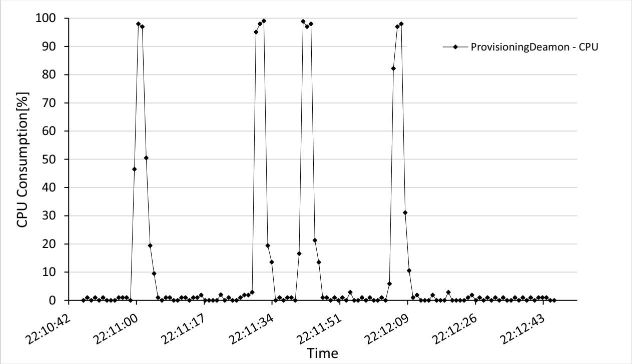

Fig. 10. CPU consumption of the ProvisioningDeamon.

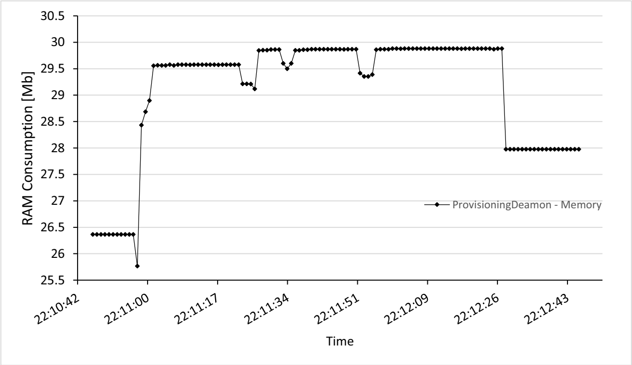

Fig. 11. Memory consumption of the ProvisioningDeamon.

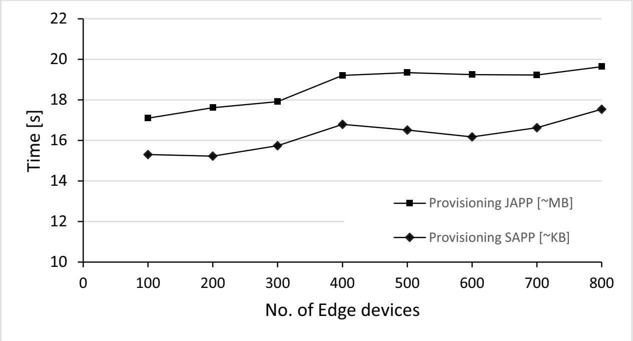

Fig. 12. Average execution time of provisioning workflows for JAPP and SAPP aplications.

The experiment presented in Figure 12 shows execution times (averaged results of 30 repetitions) of the JAPP and SAPP provisioning workflows. We can see that the middleware mechanisms for workflow execution (Section IV-A) scale within $O(nlog(n))$ for relatively large number of Edge devices (up to 800 gateways), which can be considered a satisfactory result. We also notice that computational overheads of the provisioning agents and deamons have no statistically significant impact on the results, since they are distributed among the underlying devices. Additionally, after the device number reaches 400 gateways there is a drop in the capability provisioning time. The reason for this is that the middleware load balancer spins-up additional instances of the DeploymentHandler, SDGManager and ArtifactsManager, naturally reducing provisioning time for subsequent requests. Finally, the provisioning mechanism behaves in a similar fashion for both application. The reason for this is that all gateways are in the same network, what can be seen as an equivalent to provisioning a complex of collocated buildings.

VI. Related Work

Over the last years, advancing the convergence of Edge (IoT) and Cloud computing has been receiving a lot of attention. This has resulted in a number of approaches which lay a cornerstone for realizing the utility-based provisioning in IoT Cloud. For example, different approaches deal with leveraging more powerful resources such as remote, fully-fledged Clouds or smaller Cloudlets and micro data centers, which are located in the proximity (single hop away) of the Edge, to enhance resource-constrained (mobile) devices. Such approaches, also referred to as cyber-foraging systems [32], mainly focus on specific tasks such as computation offloading [6], [7], [33] or data offloading (data staging) [9], [34]–[36]. Although, they offer valuable insights about moving cloud computing closer to the Edge, as well as about smart resource utilization, management and allocation, contrary to our approach they mainly emphasize on algorithms (e.g., solvers), energy efficiency, performance (e.g., of processing or networking) and supporting architectures for the aforementioned tasks.

Other approaches which mainly adopt a cloud-centric view, mostly aim at virtualizing Edge devices, predominantly sensors and actuators, on cloud platforms. In [10] the authors focus on developing a virtualized infrastructure to enable sensing and actuating as a service on the cloud. They propose a software stack that includes support for management of device identification and device services aggregation. In [15], the same authors discus a utility-oriented paradigm for IoT, explicit claiming the resource virtualization and abstraction as their main goal. In [16] the authors introduce sensor-cloud infrastructure that virtualizes physical sensors on the cloud and provides management and monitoring mechanisms for the virtual sensors. In [37] the authors develop an infrastructure virtualization framework for wireless sensor networks. It is based on a content-based pub/sub model for asynchronous event exchange and utilizes a custom event matching algorithm to enable delivery of sensory events to subscribed cloud users. SenaaS [38] mostly focuses on providing a cloud semantic overlay atop physical infrastructure. It defines an IoT ontology to mediate interaction with heterogeneous devices and data formats, exposing them as event streams to the upper layer cloud services. Similarly, the OpenIoT framework [17] focuses on supporting IoT service composition by following service-oriented paradigm. It mainly relies on semantic web technologies and CoAP [39] to enable web of things and linked sensory data. Such approaches address issues such as discovering, linking and orchestrating internet connected objects and IoT services. Also there are various commercial solutions such as Xively [40], Carriots [41] and ThingWorx [42], which allow users to connect their sensors to the Cloud and enable remote access to and management of such sensors. The aforementioned approaches mainly focus on providing different virtualization, device interoperability and semantic-based data integration techniques for IoT Cloud. Therefore, such approaches conceptually underpin our middleware, since virtualizing Edge devices is a main precondition towards realizing utility-based provisioning paradigm in IoT Cloud systems. Although, some of the above-described solutions (e.g., [10], [16], [17]) provide support for provisioning and management of virtual sensors and actuators, their support is often based on tightly-coupled provisioning models, e.g., static templates. Moreover, such approaches are usually meant to support specific data-centric tasks, mostly focusing on integrating various data formats, providing data-linking solutions and supporting communication protocols. Contrary, to these approaches our middleware provides support for multi-level provisioning and consuming both IoT and Cloud resources as general-purpose utilities.

Putting more focus on the network virtualization, programming and management, two prominent approaches have recently appeared, namely fog computing and software-defined networking. Advances in software-defined networking (SDN) [43]–[45] have enabled easier management and programming of the intermediate network resources, e.g., routers, mostly focusing on defining the networking logic, e.g., injecting routing rules into network elements. In [46] the authors present a concept of fog computing and define its main characteristics, such as location awareness, reduced latency and general QoS improvements. They focus on defining a virtualized platform that includes the Edge devices and enables running custom application logic atop different resources throughout the network. Although the general idea of fog computing shares similarities with our approach, there is still a number of challenges to realize its full vision [47]. Further, current advances in fog computing mainly revolve around virtualization, management and programmatic control of the network elements. Although provisioning of network resources is not the focus of our middleware, these approaches can be seen as complementary to our own approach, since the network resources are an integral part of IoT Cloud infrastructures.

Finally, since the utility-based provisioning paradigm originated from cloud computing, it is natural that cloud computing has provided numerous tools and frameworks to support the utility-based provisioning. The relevant approaches are centered around infrastructure automation and configuration management solutions such as OpsCode Chef [48], BOSH [49] and Puppet [50] as well as deployment topology orchestration approaches such as OpenStack Heat [51], AWS CloudFormation [52] and OpenTOSCA [53]. The main reasons why these solutions cannot be simply reused in the context of IoT Cloud systems are that they mostly assume unlimited amount of available resources; they do not account for intrinsic dependance of application business logic on underlying devices; they are usually not suited for constrained environments and they often rely on features provided only by fully-fledged OS, e.g., configuration management approaches often hand off dependency resolution to OS package managers.

VII. Conclusion and future work

In this paper, we introduced a provisioning middleware that enables developing generic, multi-level provisioning workflows and supports automated and scalable execution of such workflows in IoT Cloud systems. We showed how our middleware supports on-demand, self-service resource consumption by providing flexible provisioning models and support for uniform, logically centralized provisioning of Edge devices, application artifacts and their configuration models. We introduced provisioning support for software-defined gateways to enable application-specific customization of Edge devices through well-defined APIs, while preserving the benefits of proven virtualization mechanism. The initial results of our experiments are promising, since they showed that our middleware enables scalable execution of provisioning workflows across relatively large IoT cloud resource pool and at the same time its overhead in terms of resource consumption is suitable for resource-constrained devices.

In the future, we plan to improve middleware resource allocation mechanism that currently only considers static device properties. This will be accomplished by extending our middleware in several directions to: Support smarter resource allocation, e.g., optimized placement of applications on Edge devices; Provide more dynamic and finer-grained resource monitoring in order to support pay-as-you-go model, possibly in market-like fashion [54]; Enable elasticity aspects for IoT Cloud systems; And integrate the middleware with governance framework for IoT Cloud [55]. Finally, we plan to extend our middleware to address security issues such as access control and to introduce monetary compensation model for used IoT resources, based on Blockchain technology.

Acknowledgment

This work is sponsored by Joint Programming Initiative Urban Europe, ERA-NET, under project No. 5631209.

References

[1] Amazon, "Amazon Web Services IoT." URL: https://aws.amazon.com/iot/. [Online; accessed Jun.-2016].

[2] Sundar Pichai (Google Official Blog), "Building the next evolution of Google." URL: https://googleblog.blogspot.co.at/2016/05/io-building-next-evolution-of-google.html. [Online; accessed Jun.-2016].

[3] FIWARE Foundation, "FIWARE Success Strories." URL: https://www.fiware.org/success_stories. [Online; accessed Jun.-2016].

[4] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *Pervasive Computing*, vol. 8, no. 4, pp. 14–23, 2009.

[5] V. Bahl, "Cloud 2020: Emergence of micro data centers (cloudlets) for latency sensitive computing (keynote)," in *Middleware 2015*, 2015.

[6] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "Maui: making smartphones last longer with code offload," in *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pp. 49–62, ACM, 2010.

[7] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: elastic execution between mobile device and cloud," in *Conference on Computer systems*, ACM, 2011.

[8] A. Messer, I. Greenberg, P. Bernadat, D. Milojicic, D. Chen, T. J. Giuli, and X. Gu, "Towards a distributed platform for resource-constrained devices," in *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on*, pp. 43–51, IEEE, 2002.

[9] P. Stuedi, I. Mohomed, and D. Terry, "Wherestore: Location-based data storage for mobile devices interacting with the cloud," in *MCS*, 2010.

[10] S. Distefano, G. Merlino, and A. Puliafito, "Sensing and actuation as a service: a new development for clouds," in *NCA*, pp. 272–275, 2012.

[11] S. Nastic, S. Sehic, M. Voegler, H.-L. Truong, and S. Dustdar, "Patricia - a novel programing model for iot applications on cloud platforms," in *SOCA*, 2013.

[12] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility," *Future Generation computer systems*, vol. 25, no. 6, pp. 599–616, 2009.

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

[14] F. Leymann, “Cloud Computing: The Next Revolution in IT,” in *52th Photogrammetric Week ’09*, pp. 3–12, 2009.

[15] S. Distefano, G. Merlino, and A. Puliafito, “A utility paradigm for iot: The sensing cloud,” *Pervasive and mobile computing*, vol. 20, pp. 127–144, 2015.

[16] M. Yuriyama and T. Kushida, “Sensor-cloud infrastructure-physical sensor management with virtualized sensors on cloud computing,” in *NBiS*, 2010.

[17] J. Soldatos, M. Serrano, and M. Hauswirth, “Convergence of utility computing with the internet-of-things,” in *IMIS*, pp. 874–879, 2012.

[18] S. Nastic, S. Sehic, D.-H. Le, H.-L. Truong, and S. Dustdar, “Provisioning Software-defined IoT Cloud Systems,” in *FiCloud’14*, 2014.

[19] M. Voegler, J. M. Schleicher, C. Inziger, S. Nastic, S. Sehic, and S. Dustdar, “Leonore – large-scale provisioning of resource constrained iot deployments,” in *SOSE*, 2015.

[20] Tridium, “Sedona Virtual Machine.” URL: http://www.sedonadev.org/. [Online; accessed Jan.-2016].

[21] N. Wang, D. C. Schmidt, A. Gokhale, C. D. Gill, B. Natarajan, C. Rodrigues, J. P. Loyall, and R. E. Schantz, “Total quality of service provisioning in middleware and applications,” *Microprocessors and Microsystems*, vol. 27, no. 2, pp. 45–54, 2003.

[22] F. Leymann and D. Roller, “Production workflow: concepts and techniques,” 2000.

[23] A. Keller and R. Badonnel, “Automating the provisioning of application services with the bpel4ws workflow language,” in *Utility Computing*, pp. 15–27, Springer, 2004.

[24] Martin Fowler, “Microservices - a definition of this new architectural term.” URL: http://martinfowler.com/articles/microservices.html. [Online; accessed Jan.-2015].

[25] S. Nastic, H.-L. Truong, and S. Dustdar, “Sdg-pro: a programming framework for software-defined iot cloud gateways,” *Journal of Internet Services and Applications*, vol. 6, no. 1, pp. 1–17, 2015.

[26] S. Dawson-Haggerty, X. Jiang, G. Tolle, J. Ortiz, and D. Culler, “smap: a simple measurement and actuation profile for physical information,” in *SenSys*, pp. 197–210, 2010.

[27] BusyBox, “BusyBox: The Swiss Army Knife of Embedded Linux.” URL: https://busybox.net/about.html. [Online; accessed Jan.-2016].

[28] G. Procaccianti, P. Lago, and G. A. Lewis, “A catalogue of green architectural tactics for the cloud,” in *Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA), 2014 IEEE 8th International Symposium on the*, pp. 29–36, IEEE, 2014.

[29] Oracle, “Java SE Embedded 8 Compact Profiles Overview.” URL: http://www.oracle.com/technetwork/java/embedded/resources/tech/compact-profiles-overview-2157132.html. [Online; accessed Jan.-2016].

[30] CoreOs, “CoreOS - a Linux for Massive Server Deployments.” URL: http://coreos.com/. [Online; accessed Mar.-2015].

[31] Waveworks, “Wave router.” URL: https://github.com/weaveworks/weave. [Online; accessed Mar.-2016].

[32] G. Lewis, S. Echeverría, S. Simanta, B. Bradshaw, and J. Root, “Tactical cloudlets: Moving cloud computing to the edge,” in *Military Communications Conference (MILCOM), 2014 IEEE*, pp. 1440–1446, IEEE, 2014.

[33] K. Kumar and Y.-H. Lu, “Cloud computing for mobile users: Can offloading computation save energy?,” *Computer*, vol. 43, no. 4, pp. 51–56, 2010.

[34] T. Armstrong, O. Trescases, C. Amza, and E. de Lara, “Efficient and transparent dynamic content updates for mobile clients,” in *Proceedings of the 4th international conference on Mobile systems, applications and services*, pp. 56–68, ACM, 2006.

[35] J. Flinn, S. Sinnamohideen, N. Tolia, and M. Satyanarayanan, “Data staging on untrusted surrogates.,” in *FAST*, vol. 3, pp. 15–28, Citeseer, 2003.

[36] A. Zaslavsky, C. Perera, and D. Georgakopoulos, “Sensing as a service and big data,” *arXiv preprint arXiv:1301.0159*, 2013.

[37] M. M. Hassan, B. Song, and E.-N. Huh, “A framework of sensor-cloud integration opportunities and challenges,” in *ICUIMC*, 2009.

[38] S. Alam, M. Chowdhury, and J. Noll, “Senaas: An event-driven sensor virtualization approach for internet of things cloud,” in *NESEA*, 2010.

[39] B. Frank, Z. Shelby, K. Hartke, and C. Bormann, “Constrained application protocol (coap),” *IETF draft, Jul*, 2011.

[40] xively.com, “Xively.” URL: http://xively.com. [Online; accessed Jan.-2016].

[41] carriots.com, “Carriots–IoT Application Platform.” URL: https://www.carriots.com. [Online; accessed Jan.-2016].

[42] thingworx.com, “ThingWorx.” URL: http://thingworx.com. [Online; accessed Jan.-2016].

[43] B. Koldehofe, F. Dürr, M. A. Tariq, and K. Rothermel, “The power of software-defined networking: line-rate content-based routing using openflow,” in *MW4NG’12*, 2012.

[44] K. Kirkpatrick, “Software-defined networking,” *Communications of the ACM*, vol. 56, no. 9, pp. 16–19, 2013.

[45] H. Kim and N. Feamster, “Improving network management with software defined networking,” *Communications Magazine, IEEE*, vol. 51, no. 2, pp. 114–119, 2013.

[46] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, “Fog computing and its role in the Internet of Things,” in *MCC workshop on Mobile cloud computing*, pp. 13–16, 2012.

[47] S. Yi, C. Li, and Q. Li, “A survey of fog computing: concepts, applications and issues,” in *Proceedings of the 2015 Workshop on Mobile Big Data*, pp. 37–42, ACM, 2015.

[48] OpsCode, “Chef.” URL: http://opscode.com/chef. [Online; accessed Feb.-2016].

[49] BOSH, “BOSH.” URL: http://docs.cloudfoundry.org/bosh/. [Online; accessed Feb.-2016].

[50] Puppet Labs, “Puppet.” URL: http://puppetlabs.org. [Online; accessed Feb.-2016].

[51] Open Stack Orchestration, “Heat Project.” URL: https://wiki.openstack.org/wiki/Heat. [Online; accessed Feb.-2016].

[52] AWS, “CloudFormation.” URL: https://aws.amazon.com/cloudformation/. [Online; accessed Feb.-2016].

[53] OpenTOSCA, “OpenTOSCA.” URL: http://www.iaas.uni-stuttgart.de/OpenTOSCA/. [Online; accessed Feb.-2016].

[54] M. Vögler, F. Li, M. Claeßens, J. M. Schleicher, S. Sehic, S. Nastic, and S. Dustdar, “Colt collaborative delivery of lightweight iot applications,” in *Internet of Things. User-Centric IoT*, pp. 265–272, Springer, 2015.

[55] S. Nastic, M. Voegler, C. Inziger, H.-L. Truong, and S. Dustdar, “rtGovOps: A Runtime Framework for Governance in Large-scale Software-defined IoT Cloud Systems,” in *Mobile Cloud 2015*, 2015.