Towards Deviceless Edge Computing: Challenges, Design Aspects, and Models for Serverless Paradigm at the Edge

Stefan Nastic and Schahram Dustdar

1 Introduction

Recently, Cloud Computing, Edge Computing, and the Internet of Things (IoT) have been converging ever stronger, sparking creation of very large-scale, geographically distributed systems [1, 2]. Such systems intensively exploit Cloud Computing models and technologies, predominantly by utilizing large and remote data centers, but also nearby Cloudlets [3, 4] to enhance resource-constrained Edge devices (e.g., in terms of computation offloading [5–7] and data staging [8]) or to provide an execution environment for cloud-centric IoT/Edge applications [9, 10].

Serverless computing is an emerging paradigm, typically referring to a software architecture where application is decomposed into "triggers" and "actions" (or functions), and there is a platform that provides seamless hosting and execution of developer-defined functions (FaaS), making it easy to develop, manage, scale, and operate them. This complexity mitigation is mainly achieved by incorporating sophisticated runtime mechanisms into serverless or FaaS platforms. Hence, such platforms are usually characterized by fully automating many of the management and operations processes. Therefore, serverless computing can be considered as the next step in the evolution of Cloud platforms, such as PaaS, or more generally of the utility computing.

While originally designed for centralized cloud deployments, the benefits of serverless paradigm become especially evident in the context of Edge Computing [11]. This is mainly because in such systems, traditional infrastructure and application management solutions are tedious, ineffective, error-prone, and ultimately very costly. Luckily, some of the existing serverless techniques, such

S. Nastic (✉) · S. Dustdar
Distributed Systems Group, TU Wien, Vienna, Austria
e-mail: nastic@dsg.tuwien.ac.at; dustdar@dsg.tuwien.ac.at

© The Author(s) 2018

V. Gruhn, R. Striemer (eds.), *The Essence of Software Engineering*,
https://doi.org/10.1007/978-3-319-73897-0_8

as sandboxed execution of polyglot tenant-provided code, can be applied on Edge without substantial modifications. The most common approach to runtime execution environments is to utilize Linux containers (such as Docker). Unfortunately, due to inherently different nature of Edge infrastructure, for example, in terms of available resources, network, geographical hyper-distribution, very large scale, etc., fundamental architecture and design assumptions behind cloud-based serverless computing need to be reexamined and specifically tailored for the Edge infrastructure in order to realize *Deviceless Edge Computing*. Some of the main research challenges of the emerging Deviceless Computing include:

- *Resource pooling and rapid elasticity*. Traditional serverless platforms utilize commodity infrastructure, small footprint, and short execution duration, combined with statistical multiplexing of a large number of heterogeneous workloads over time [12]. Elasticity at the Edge implies challenges not present in the Cloud, mostly due to different nature of the infrastructure, the topology of network connectivity, and locality-awareness.
- *Security*. Unlike serverless platforms which often operate in secured environments, the Edge is exposed to various attacks, requiring much better protection and isolation for the individual hosts, tenants, and applications.
- *Automated provisioning and management at scale*. Due to dynamicity, heterogeneity, geographical distribution, and the sheer scale of the Edge infrastructure, traditional management and provisioning approaches are hardly feasible in practice. Thus, novel techniques, which will provide a uniform view and interaction with both Cloud and Edge, are needed [13].
- *Scheduling on loosely coupled and scarce Edge resources.* Scheduler is one of the core components in cloud-based serverless computing. However, at the Edge, application scheduling, orchestration, and configuration management cannot be done in an easy and predictable manner (e.g., by Deviceless platform runtime mechanisms) due to the limited nature of Edge resources and their inherent volatility.
- *Deviceless application development.* In Deviceless paradigm we trade explicit device management for slightly complex application business logic. This means that the development context of such applications needs to grow beyond writing custom business logic to also consider the involved Edge resources and their capabilities, but on a higher level of abstraction, for example, in code.
- *Edge-centric governance.* Due to inherently different nature of Edge-based systems, traditional governance approaches need to be reevaluated and particularly designed to be suitable in the new Edge context. In particular, governance objectives (law, compliance, etc.) are not easily mapped to concrete operations processes (e.g., querying sensory data streams or adding/removing devices). Additionally, making the governance approaches feasible in deviceless paradigm requires full automation of such operational governance processes.

In this chapter, we continue our line of research towards realizing the novel paradigm of Deviceless Edge Computing, by extending the previously defined concepts [11] and by building on our existing work in the area of Edge Computing and

IoT, which serve as the main enablers of Deviceless Computing. In particular, we propose a reference architecture for the Deviceless Edge Computing. Furthermore, we analyze the main aspects of realizing the Deviceless Computing paradigm from two main points of view: (1) required support for application development, in terms of programming models (Sect. 4), and (2) required runtime support for deviceless applications, in terms of main deviceless platform mechanisms (Sect. 5).

The remainder of the chapter is organized as follows: Sect. 2 presents the state of the art. Section 3 introduces a reference architecture of a Deviceless Edge Platform. In Sect. 4 we present our programming model for developing deviceless functions. Section 5 introduces the provisioning model and a middleware for provisioning Deviceless Edge applications. Finally, Sect. 6 concludes the chapter and gives an outlook of future research.

2 Related Work

Recently, the serverless computing paradigm has been rapidly emerging in the IT industry, since its appearance in AWS Lambda[1] in 2014. Major public Cloud providers have introduced comparable FaaS offerings—Azure Functions,[2] Google Cloud Functions.[3] In addition to commercial offerings, several open-source initiatives have emerged, including Apache OpenWhisk[4] (originally developed by IBM, now under incubation at ASF jointly with Adobe and additional companies), as well as several projects developed in the open by various vendors such as Iron Functions,[5] Fission,[6] and Kubeless.[7]

In spite originating as a special case of Cloud computing, the FaaS/serverless paradigm has since evolved to also become applicable beyond the traditional Cloud data centers. For example, the PubNub BLOCKS[8] offering enhances their real-time data stream capabilities running on a network of Edge data centers (e.g., used in IoT applications to stream events and logs between the Edge and the Cloud), with the ability to invoke custom handlers (provided by the application developer) on the data path. Similarly, Amazon Lambda@Edge[9] allows to run custom Javascript handlers on web traffic going through their CloudFront (CDN) facilities. Moreover, there are recent attempts to expand applicability of "serverless" even further, to

[1] http://aws.amazon.com/lambda/.

[2] http://azure.microsoft.com/en-us/services/functions/.

[3] http://cloud.google.com/functions/.

[4] http://openwhisk.org/.

[5] http://open.iron.io/.

[6] http://fission.io/.

[7] http://github.com/bitnami/kubeless.

[8] http://goo.gl/IIjkZi.

[9] http://docs.aws.amazon.com/lambda/latest/dg/lambda-edge.html.

IoT gateways and devices—both commercial (e.g., Amazon Greengrass[10]) and exploratory (e.g., OpenWhisk). However, most of these attempts are at early stages, and architectural and design assumptions behind such approaches need to be reevaluated, for example, to address the challenges described in Sect. 1, in order for the serverless paradigm to be fully adopted in Edge computing environments, as opposed to being an extension of Cloud (e.g., in CDN).

The core principle of the serverless paradigm includes fully automated orchestration, lifecycle management, and scheduling of both user-defined functions and underlying resource pools. Recently, different approaches have emerged that focus on utilizing principles of Service-Oriented Architecture (SOA), dynamic service orchestration, and Cloud computing techniques, in order to facilitate execution of data processing applications (e.g., with cloud offloading), but also provisioning and management of vast Edge infrastructure. For example, in [14] the authors introduce sensor-cloud infrastructure that virtualizes physical sensors as services and provides management and monitoring mechanisms for the virtual sensors. However, their support for provisioning and orchestration of virtual sensors is based on static templates, which are not intended for dynamic reconfigurations and optimizations required in Deviceless platform. OpenIoT framework [15] utilizes semantic Web technologies and CoAP to enable web of things and linked sensory data. They mostly focus on discovering, linking, and orchestrating Internet-connected objects. Further, the authors focus on developing a virtualization infrastructure to enable sensing and actuating as a service on the Cloud. They propose a software stack which includes support for management of device identification, selection, and aggregation. In [16] 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 and a range of mechanisms to support SaaS applications. These approaches provide valuable insights, advances, and a solid baseline to underpin the Deviceless Edge Computing paradigm.

3 Deviceless Edge Platform

3.1 Approach

The main objective of our approach is to provide a full stack platform for supporting executing and automatically operating Deviceless applications across Cloud and Edge in a unified manner. The key role of the distributed *Deviceless Edge Platform* is to facilitate automated management of the underlying resource pool and optimal placement of applications/functions in order to support the envisioned deviceless

[10] http://aws.amazon.com/greengrass/.

execution model. This approach allows for combining the benefits of the Edge (lower response time, ability to manage heterogeneous data) with the computational and storage capabilities of the Cloud. For example, time-sensitive data, such as life-critical vital signs, can be analyzed at the Edge close to where they are generated instead of being transported to the Cloud for processing. Alternatively, selected data can be forwarded to the Cloud for a further, more powerful analysis and long-term storage.

3.2 *Platform Usage and Architecture Overview*

Figure 1 shows a high-level view of the platform and main *top-down control process* (left) and *application execution and results delivery process* (right). The proposed deviceless paradigm is particularly suitable for managing different granularity of user-defined business logic functions *bottom-up*. This means that the Edge focuses on local views (e.g., per Edge gateway) while the Cloud supports global

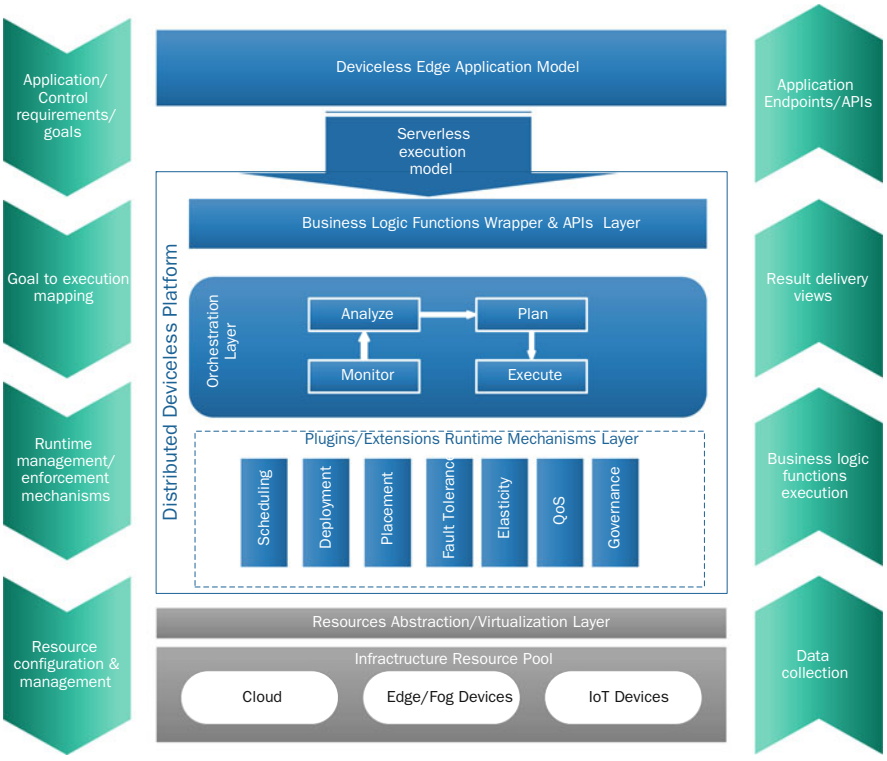

Fig. 1 Deviceless platform architecture

views, for example, combining and analyzing data from different Edge devices, regions, or even domains. For example, in the case of data analytics applications, data is collected from the underlying devices and delivered to the applications via consumption APIs. More importantly, the application business logic such as data analytics can be performed on Edge nodes, Cloud nodes, or both, and the results can be delivered from any of the nodes directly, based on the desired view. Moreover, the *top-down control process* allows decoupling of application requirements (*What*) from concrete realization of those requirements (*How*). This allows developers to simply define the application behavior and business logic and application goals (e.g., regarding provisioning) instead of dealing with the complexity of different management, orchestration, and optimization processes. Moreover, Fig. 1 shows the Deviceless platform's core architecture:

- *Business Logic Wrapper and APIs Layer* focuses on executing and managing user-provided functions, for example, delivering required data to the function and creating results endpoints. To this end, it wraps the user-defined functions in executable artifacts such as Linux Containers and relies on the underlying layers to perform concrete runtime actions and execution steps.
- *The Orchestration Layer* is responsible for interpreting and executing user-defined functions, requirements, and configuration models. This layer acts as a "gluing" component bringing together application's configuration model, business logic functions, and platform's runtime mechanisms. Therefore, the Orchestration Layer receives the application configuration directives, in terms of high-level objectives such as to optimize network latency. It interprets and analyzes these goals and decides how to orchestrate the underlying resources, as well as the user-defined functions, by invoking the underlying runtime mechanisms. To this end, this layer contains micro (Edge-based) and macro (Cloud-based) orchestration and control loops. For example, it can utilize the Scheduling and the Placement mechanisms to determine the most suitable node (Cloud or Edge) for executing a function in order to reduce the network latency.
- *The Runtime Mechanisms Layer* is an extensible plug-ins layer, providing mechanisms to support executing the actions initiated by the Orchestration Layer. The Deployment, the Scheduling, the Elasticity, and basic reasonable defaults for the Quality of Service (QoS) are the core runtime mechanisms. More precisely, the platform has to determine the minimally required elastic resources, provision them, deploy, and then schedule and execute analytics functions, which will satisfy the QoS requirements. On the other hand, the Governance, the Placement, the Fault Tolerance, and the extended QoS mechanisms are optional. For example, in some cases, the sensory data, used by an application, could be confidential and some geographical regions should be excluded. Placing the computation (functions) closer to the data and deciding whether to use Cloud or Edge resources could improve the QoS. Additionally, having a k-fault-tolerant platform that can mitigate the risks of failures to acceptable level further improves the QoS.

In the remainder of the chapter, we particularly focus on two key aspects of Deviceless Edge platform: its programming support for deviceless applications and its support for application management and operation.

4 Programming Support for Deviceless Edge Computing

The main purpose of our programming model is to provide a programmatic view on the whole application ecosystem, that is, the full stack from the infrastructure to software components and services. The main principle behind our programming model is *everything as code*. This includes providing support for writing deviceless functions' business logic, as well as representing the underlying infrastructure components (e.g., gateways) at the application level and enabling developers to programmatically determine their deployment and provisioning. Figure 2 shows a component diagram with the logical structure of Deviceless Edge applications. The main components of such application include custom business logic components, that is, user-defined functions; resource provisioning and deployment logic (custom or stock component provisioning); and operational governance logic. In the remainder of this section, we mainly focus on the programming support for deviceless functions. More details on programmatic provisioning and governance can be found in [17].

4.1 Programming Support for Deviceless Edge Functions

In our programming model, we consider a unified notion of deviceless functions. However, we provide versatile abstractions, which enable expressing the functions' business logic depending on the nature of their respective interactions with Edge or Cloud resources. Figure 3 shows a simplified UML diagram of the programming

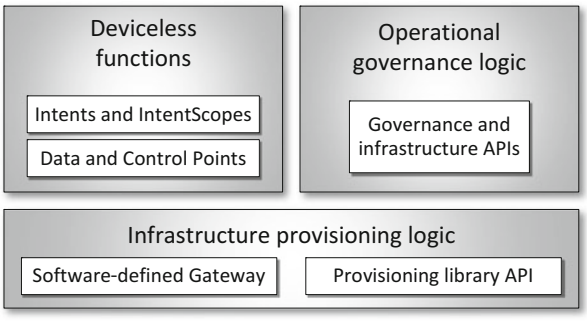

Fig. 2 Overview of deviceless application structure

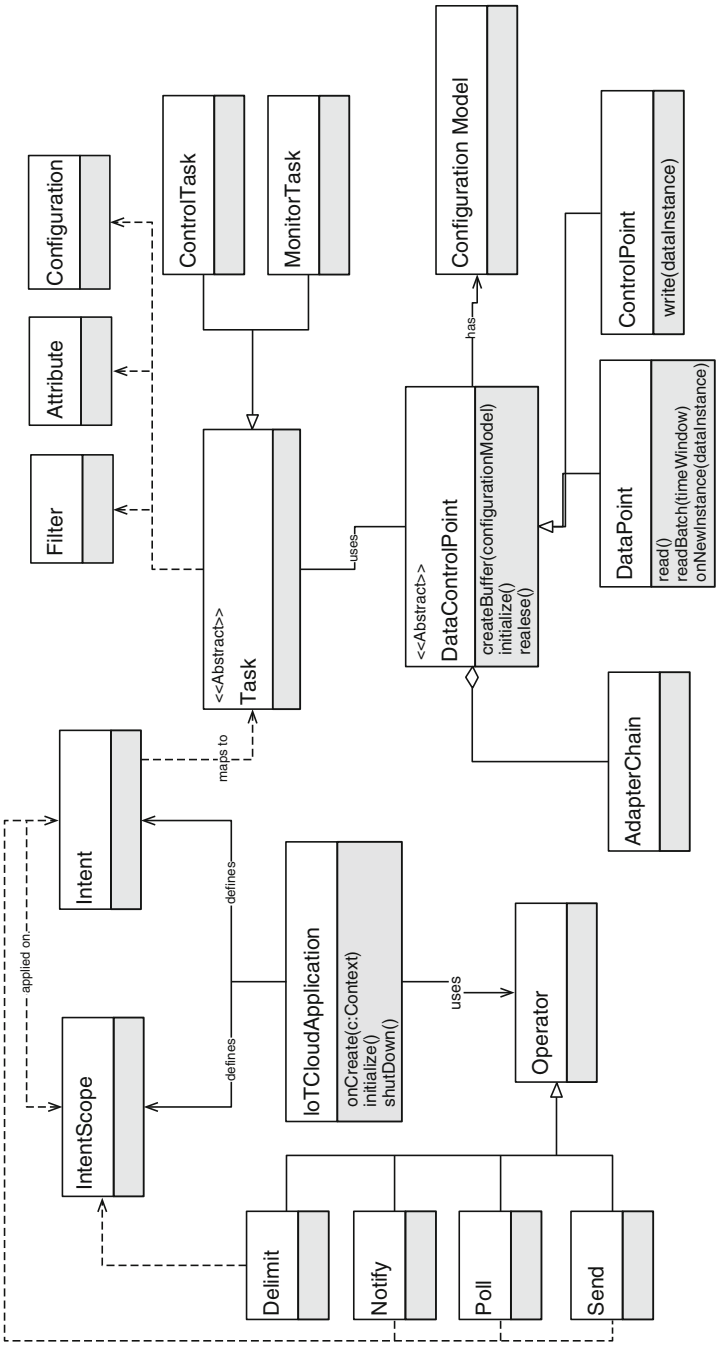

Fig. 3 Simplified UML of programming model for Deviceless edge functions

model. Its key abstractions are *Data and Control Points* and *Intents*. Deviceless functions can be executed in Edge devices to implement control and monitor tasks. For example, a monitoring task includes processing, correlation, and analysis of sensory data streams. Data and Control Points are provided to support such a task development. Deviceless functions executed in the Cloud usually define virtual service topologies by referencing the tasks. At the application level, we provide explicit representation of these tasks via *Intents*, that is, developers write *Intents* to dynamically configure and invoke the tasks. Further, developers use *IntentScope*s to delimit the range of an *Intent*. For example, a developer might want to code the expression: "stop all vehicles on golf course X." In this case, "stop" is the desired *Intent*, which needs to be applied on an *IntentScope* that encompasses all vehicles with the location property "golf course X."

4.2 Intents and IntentScopes

Intent is a data structure describing a specific task which can be performed in a physical environment. In reality, *Intent*s are processed and executed on the Deviceless platform, but enable monitoring and controlling of the physical environments by triggering corresponding deviceless functions. Based on the information contained in an *Intent*, a suitable task/function is dynamically selected, instantiated, and executed. Depending on the task's nature, we distinguish between two different types of *Intent*s: *ControlIntent* and *MonitorIntent*. *ControlIntent*s enable applications to operate and invoke the low-level components, that is, provide a high-level representation of their functionality. *MonitorIntent*s are used by applications to subscribe for events from the sensors and to obtain devices' context.

Figure 4 shows the *Intent* structure and its most relevant parts. Each *Intent* contains an ID, used to correlate invocation response with it or apply additional actions on it. Additionally, it contains a set of headers, which specify meta-information needed to process the *Intent* and bind it with a suitable task during the runtime. Among other things, headers carry *Intent's* name and a reference to an *IntentScope*. Further, an *Intent* can contain a set of attributes, which are used by the runtime to select the best matching task instance in case there are multiple

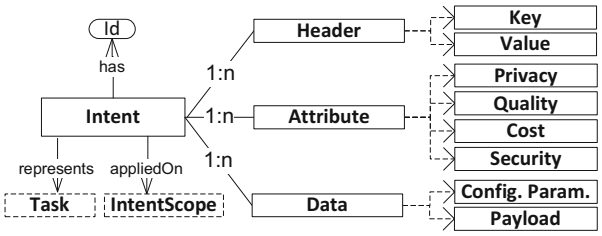

Fig. 4 Intent structure

Intent implementations available. Finally, *Intent* can contain data, which is used to configure the tasks or supply additional payload. Generally, *Intents* allow developers to communicate to the system what needs to be done instead of worrying how the underlying devices will perform the specific task.

Our programming model also allows developers to define *IntentScopes*. IntentScopes need to be defined explicitly and implicitly, that is, developers can explicitly add entities to the scope by specifying their IDs or recursively prune the *GlobalScope*. Formally, we use the well-known set theory to define *IntentScope* as a finite, countable set of entities (set elements). The *GlobalScope* represents the universal set, denoted by S^{max}; therefore, $\forall S(S \subseteq S^{max})$, where S is an *IntentScope*, must hold. Further, for each entity E in the system general membership relation $\forall E(E \in S | S \subseteq S^{max})$ must hold. Therefore, an entity is the unit set, denoted by S_{min}. Empty set $\emptyset$ is not defined; thus, applying an *Intent* on it results with an error. Finally, a necessary condition for an *IntentScope* to be valid is as follows: *IntentScope* is valid iff it is a set S, such that $S \subseteq S^{max} \wedge S \not\equiv \emptyset$ holds. Equation (1) shows operations used to define or refine an *IntentScope*. The most interesting operation is $\subseteq_{cond}\ S$. It is used to find a subset ($\hat{S}$) of a set S, which satisfies some condition, that is, $E \in \hat{S} \mid E \in S \wedge cond(E) = True$.

$$S = S_{min} | S^{max} | \subseteq_{cond} S \mid S \cup S \mid S \cap S \mid S \setminus S \tag{1}$$

4.3 Data and Control Points

Generally, the main motivation for introducing the Data and Control Points is to enable developing deviceless functions that encapsulate a domain-specific task. Hence, they are used to develop domain libraries of deviceless functions. In this context, a domain library contains a set of reusable functions that are responsible to encapsulate domain-specific knowledge, most notably domain model and common behaviors, in a reusable manner. For example, a building automation expert developer could develop a domain library to facilitate development of higher-level functionality for building management systems. To this end, Data and Control Points represent and enable management of data and control channels (e.g., device drivers) to the low-level sensors/actuators in an abstract manner. Generally, they mediate the communication with the connected devices (e.g., digital, serial, or IP based), enable application-specific customizations of the channels, and also implement communication protocols for the connected devices, for example, Modbus, CAN, or I^2C.

The *DataControlPoint* (Fig. 3) is an abstract class which provides main operators and lifecycle management hooks for the Data and Control Points. Both DataPoints and ControlPoints inherit from this component and encapsulate the specialized behavior for reading sensory data (DataPoints) and preforming the actuations (ControlPoints). In general, the DataControlPoint allows the developers to perform concurrent reads and writes, regardless of whether the low-level drivers support

sequential or concurrent reads and writes. In this way, the applications have an impression of exclusive usage of the available devices. Another important feature of DataControlPoint is that they enable developers to configure custom behavior of underlying devices. For this purpose, each DataControlPoint can have a ConfigurationModel associated with it. For example, an application 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.

The most important concept supporting the DataControlPoint is the VirtualBuffers, which are provided and managed by the Deviceless Edge Platform. In general, such buffers enable virtualized access to and custom configurations of underlying sensors and actuators. They act as multiplexers of the data and control channels, thus enabling the device applications to have their own view of and define custom configurations for such channels. To this end, the VirtualBuffers wrap the device drivers and share a common behavior with them. For example, they can be initialized, shut down, and released. Both buffers and drivers lifecycle are managed by the platform. Finally, to support application-specific configurations such as sensor poll rates, filters, or scalers, each virtual buffer can have an AdapterChain. Adapter chains reference different Adapters, which are specified and parametrized via DataControlPoint's ConfigurationModel. Any raw sensing value is passed through such adapter chain before being delivered to a DataPoint.

5 Provisioning Support for Deviceless Edge Computing

In this section, we shift focus from deviceless functions and application level support to the core Deviceless Edge Platform components. In particular, we discuss resource provisioning in Deviceless Edge Computing, as it is the cornerstone for resource pooling and rapid elasticity at the Edge. Moreover, provisioning component (middleware) is a crucial enabler for deviceless paradigm, because it decouples the developers and their applications from the underlying devices. In the following, we discuss our deviceless provisioning model and the middleware.

5.1 Software-Defined Gateways

Software-defined gateways (SDGs) are the core abstraction in deviceless provisioning model. Their main purpose is to support virtualizing Edge compute resources, for example, IoT devices, in order to provide isolated and managed execution environments for deviceless functions.

To achieve this, SDGs encapsulate functional aspects (e.g., communication capabilities or sensor poll frequencies) and non-functional aspects (e.g., quality attributes, elasticity capabilities, costs, and ownership information) of the Edge resources and expose them to the deviceless platform (provisioning middleware).

The functional, provisioning, and governance capabilities of the units are exposed via *well-defined APIs*, which enable provisioning and controlling the SDGs at runtime, for example, start/stop. Our conceptual model also allows for composing and interconnecting SDGs, in order to dynamically deliver the Edge resources and capabilities to the applications. The runtime provisioning and configuration is performed by specifying late-bound policies and configuration models. Naturally, the SDGs support mechanisms to map the virtual resources with the underlying physical infrastructure. However, this is out of the scope of this chapter. Finally, some of the most important features of SDGs include:

- They provide software-defined API, which can be used to access, configure, and control the units, in a unified manner.
- They support fine-grained internal configurations, for example, adding functional capabilities like different communication protocols, at runtime.
- They can be composed at higher level, via dependency units, creating virtual topologies that can be (re)configured at runtime.
- They enable decoupled and managed configuration (via late-bound policies) to provision the units dynamically and on-demand.
- They have utility cost functions that enable pricing the Edge resources as utilities.

Figure 5 gives the architectural view of SDGs and depicts the most important components of software-defined gateways. In the figure, the double line shows virtual boundaries of the SDG prototypes. Our provisioning model does not require building custom SDGs from scratch. Instead, it provides SDG prototypes and defines mechanisms (implemented by the middleware) 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

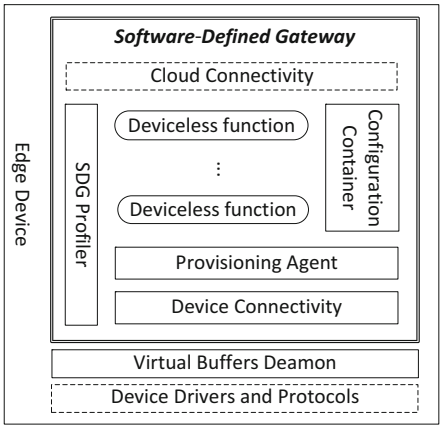

Fig. 5 Software-defined gateway (SDG) architecture

prototypes are used to bootstrap higher-level SDG functionality. It is important to mention that SDG prototypes do not propose a novel virtualization solution. Instead, they rely on proven techniques, namely, kernel-supported virtualization approaches, which offer a number of lightweight execution environments/drivers such as LXCs, libvirt-sandbox, or even chroot. Such environments are generally referred to as containers that can be used 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 are a number of components (cf. Fig. 5) which are preinstalled in each SDG prototype in order to support such APIs.

5.2 *Deviceless Provisioning Middleware*

Figure 6 gives a high-level architecture overview of our middleware. Generally, the provisioning middleware is designed based on the microservices architecture and it is distributed across the Cloud and Edge devices. The main components

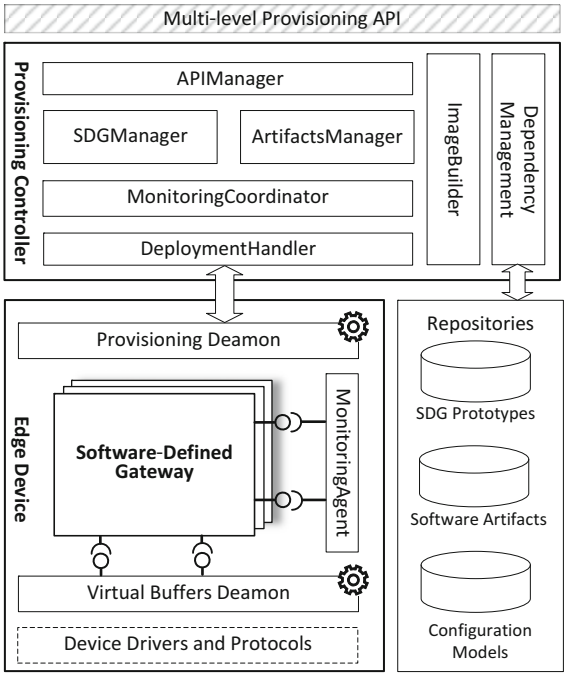

Fig. 6 Architecture overview of the deviceless provisioning middleware

of the provisioning middleware include (1) the *Software-Defined Gateways*, (2) the *Provisioning and Virtual Buffers Daemons* that run in Edge devices, and (3) the *Provisioning Controller* which runs in the Cloud. Previously, we have briefly discussed the SDGs; in the remainder of this section, we mainly focus on describing the Provisioning Controller component and point an interested reader to our earlier publication [13], where we discuss the Provisioning and Virtual Buffers Daemons in great detail.

The Provisioning Controller (Fig. 6, top) provides a mediation layer that enables the Deviceless Edge Platform to interact with the Edge infrastructure 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. This enables our Provisioning Controller to support elastically scalable execution of provisioning workflows, since we can dynamically spin up additional instances of microservices under heavy load and scale out the Provisioning Controller to support a 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 *APIManager* is to manage the *Multilevel Provisioning API*, that is, it encapsulates the middleware provisioning capabilities in well-defined APIs and handles all API calls from user-defined provisioning workflows. Although our middleware provides multilevel 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, that is, *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 delivering them to the Edge devices. 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 user-defined functions 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 *DependencyManagement service* to resolve the required 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 infrastructure, since otherwise each Edge device would have to perform the same set of actions, for example, downloads.

6 Conclusion

The chapter introduced a novel vision of the Deviceless Edge Computing paradigm. In order to clarify some of the most important aspects of this emerging paradigm, we have analyzed the key challenges associated with Deviceless Edge Computing and presented a generic reference architecture of a Deviceless Platform. Moreover, we have presented *Intent-based programming model* and an approach for *automated provisioning* of the Edge infrastructure, based on *Software-Defined Gateways*. We discussed how these two approaches facilitate two main challenges: *deviceless application development* and *automated provisioning and management at scale*, respectively.

As we have discussed, the presented approaches significantly reduce the complexity related to development and runtime management (e.g., provisioning, deployment, and configuration management) of deviceless applications. However, there is still a long road ahead to fully realize the vision of the Deviceless Edge Computing. In the future, we plan to continue our line of research, by focusing on addressing the most important research challenges such as (1) enabling resource pooling and rapid elasticity, at the Edge, (2) scheduling deviceless functions execution on loosely coupled and scarce Edge resources, and (3) addressing the key governance and security issues related with deviceless applications. To this end, we plan to focus on "filling the gaps" in the proposed reference architecture, by developing the required models and platform mechanism.

Acknowledgment This work is sponsored by Joint Programming Initiative Urban Europe, ERA-NET, SMART-FI project under project No. 6683255.

References

1. Amazon: Amazon Web Services IoT. https://aws.amazon.com/iot/. Accessed June 2016
2. Sundar Pichai (Google Official Blog): Building the next evolution of Google. https://googleblog.blogspot.co.at/2016/05/io-building-next-evolution-of-google.html. Accessed June 2016
3. Satyanarayanan, M., Bahl, P., Caceres, R., Davies, N.: The case for VM-based cloudlets in mobile computing. Pervasive Comput. **8**(4), 14–23 (2009)
4. Bahl, V.: Cloud 2020: emergence of micro data centers (cloudlets) for latency sensitive computing (keynote). In: Middleware 2015 (2015)
5. Cuervo, E., Balasubramanian, A., Cho, D.-K., Wolman, A., Saroiu, S., Chandra, R., Bahl, P.: 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, New York (2010)
6. Chun, B.-G., Ihm, S., Maniatis, P., Naik, M., Patti, A.: Clonecloud: elastic execution between mobile device and cloud. In: Conference on Computer Systems. ACM, New York (2011)
7. Messer, A., Greenberg, I., Bernadat, P., Milojicic, D., Chen, D., Giuli, T.J., Gu, X.: Towards a distributed platform for resource-constrained devices. In: Proceedings 22nd International Conference on Distributed Computing Systems, 2002, pp. 43–51. IEEE, New York (2002)

8. Stuedi, P., Mohomed, I., Terry, D.: Wherestore: location-based data storage for mobile devices interacting with the cloud. In: MCS (2010)
9. Distefano, S., Merlino, G., Puliafito, A.: Sensing and actuation as a service: a new development for clouds. In: NCA, pp. 272–275 (2012)
10. Nastic, S., Sehic, S., Voegler, M., Truong, H.-L., Dustdar, S.: Patricia - a novel programing model for iot applications on cloud platforms. In: SOCA (2013)
11. Glikson, A., Nastic, S., Dustdar, S.: Deviceless edge computing: extending serverless computing to the edge of the network (2017)
12. Breitgand, D., Glikson, A., et al.: Sla-aware resource over-commit in an IaaS cloud. In: CNSM'12
13. Nastic, S., et al.: A middleware infrastructure for utility-based provisioning of IoT cloud systems. In: The First IEEE/ACM Symposium on Edge Computing (2016)
14. Yuriyama, M., Kushida, T.: Sensor-cloud infrastructure-physical sensor management with virtualized sensors on cloud computing. In: NBiS (2010)
15. Soldatos, J., Serrano, M., Hauswirth, M.: Convergence of utility computing with the internet-of-things. In: IMIS, pp. 874–879 (2012)
16. Hassan, M.M., Song, B., Huh, E.-N.: A framework of sensor-cloud integration opportunities and challenges. In: ICUIMC (2009)
17. Nastic, S., Truong, H.-L., Dustdar, S.: SDG-Pro: a programming framework for software-defined IoT cloud gateways. J. Internet Serv. Appl. **6**(1), 1–17 (2015)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.