2014 International Conference on Future Internet of Things and Cloud

Provisioning Software-defined IoT Cloud Systems

Stefan Nastic, Sanjin Sehic, Duc-Hung Le, Hong-Linh Truong, and Schahram Dustdar
Distributed Systems Group, Vienna University of Technology, Austria
Email: {*lastname*}@*dsg.tuwien.ac.at*

***Abstract*—Cloud computing is ever stronger converging with the Internet of Things (IoT) offering novel techniques for IoT infrastructure virtualization and its management on the cloud. However, system designers and operations managers face numerous challenges to realize IoT cloud systems in practice, mainly due to the complexity involved with provisioning large-scale IoT cloud systems and diversity of their requirements in terms of IoT resources consumption, customization of IoT capabilities and runtime governance. In this paper, we introduce the concept of *software-defined IoT units* – a novel approach to IoT cloud computing that encapsulates fine-grained IoT resources and IoT capabilities in well-defined APIs in order to provide a unified view on accessing, configuring and operating IoT cloud systems. Our software-defined IoT units are the fundamental building blocks of software-defined IoT cloud systems. We present our framework for dynamic, on-demand provisioning and deploying such software-defined IoT cloud systems. By *automating provisioning* processes and supporting *managed configuration models*, our framework simplifies provisioning and enables flexible runtime customizations of software-defined IoT cloud systems. We demonstrate its advantages on a real-world IoT cloud system for managing electric fleet vehicles.**

I. Introduction

Cloud computing technologies have been intensively exploited in development and management of the large-scale IoT systems, e.g., in [11], [16], [18], because theoretically, cloud offers unlimited storage, compute and network capabilities to integrate diverse types of IoT devices and provide an elastic runtime infrastructure for IoT systems. Self-service, utility-oriented model of cloud computing can potentially offer fine-grained IoT resources in a pay-as-you-go manner, reducing upfront costs and possibly creating cross-domain application opportunities and enabling new business and usage models of the IoT cloud systems.

However, most of the contemporary approaches dealing with IoT cloud systems largely focus on data and device integration by utilizing cloud computing techniques to virtualize physical sensors and actuators. Although, there are approaches providing support for provisioning and management of the virtual IoT infrastructure (e.g, [8], [16], [18]), the convergence of IoT and cloud computing is still at an early stage. System designers and operations managers face numerous challenges to realize large-scale IoT cloud systems in practice, mainly because these systems impose diverse requirements in terms of granularity and flexibility of IoT resources consumption, custom provisioning of IoT capabilities such as communication protocols, elasticity concerns, and runtime governance. For example, modern large-scale IoT cloud systems heavily rely on the cloud and virtualized IoT resources and capabilities (e.g., to support complex, computationally expensive analytics), thus these resources need to be accessed, configured and operated in a unified manner, with a central point of management. Further, the IoT systems are envisioned to run continuously, but they can be elastically scaled in/down in off-peek times, e.g., when a demand for certain data sources reduces. Due to the multiplicity of the involved stakeholders with diverse requirements and business models, the modern IoT cloud systems increasingly need to support different and customizable usage experiences. Therefore, to utilize the benefits of cloud computing, IoT cloud systems need to support virtualization of IoT resources and IoT capabilities (e.g., gateways, sensors, data streams and communication protocols), but also enable: i) encapsulating them in a well-defined API, at different levels of abstraction, ii) centrally managing configuration models and automatically propagating them to the edge of infrastructure, iii) automated provisioning of IoT resources and IoT capabilities.

In this paper, we introduce the concept of software-defined IoT units – a novel approach to IoT cloud computing that encapsulates fine-grained IoT resources and IoT capabilities in a well-defined API in order to provide a unified view on accessing, configuring and operating IoT cloud systems. Our software-defined IoT units are the fundamental building blocks of software-defined IoT cloud systems. They enable consuming IoT resources at a fine granularity, allow for policy-based configuration of IoT capabilities and runtime operation of software-defined IoT cloud systems. We present our framework for dynamic, on-demand provisioning of the software-defined IoT cloud systems. By automating main aspect of provisioning processes and supporting centrally managed configuration models, our framework simplifies provisioning of such systems and enables flexible runtime customizations.

The rest of the paper is structured as follows: Section II presents a motivating scenario and research challenges; Section III describes main principles and our conceptual model of software-defined IoT systems; Section IV outlines main provisioning techniques for software-defined IoT systems; Section V introduces design and implementation of our prototype, followed by its experimental evaluation; Section VI discusses the related work; Finally, Section VII concludes the paper and gives an outlook of the future research.

II. Motivation

A. Scenario

Consider a scenario about fleet management (FM) system for small-wheel electric vehicles deployed worldwide, on different golf courses. The FM is an IoT cloud system comprising golf cars' on-board gateways, network and the cloud infrastructure. The main features provided by the on-board device include: a) vehicle maintenance (fault history, battery health, crash history, and engine diagnostics), b) vehicle tracking (position, driving history, and geo-fencing), c) vehicle info (charging status, odometer, serial number, and service notification), d) set-up (club-specific information, maps, and fleet information). Vehicles communicate with the cloud via 3G, GPRS or Wi-Fi network to exchange telematic and diagnostic data. On the cloud we host different FM subsystems and services to manage the data. For example: a) *Realtime vehicle status*: location, driving direction, speed, vehicle fault alarms; b) *Remote diagnostics*: equipment status, battery health and timely maintenance reminders; c) *Remote control*: overriding on-board vehicle control system in case of emergency; d) *Fleet management*: service history and fleet usage patterns.

In the following we highlight some of the requirements and features of the FM system that we need to support:

- The FM subsystems and services are hosted in the cloud and heavily rely on the virtualized IoT resources, e.g., vehicle gateways and their capabilities. Therefore, we need to enable encapsulating and accessing IoT resources and IoT capabilities, via an uniform API.
- The FM system has different requirements regarding communication protocols. The fault alarms and events need to be pushed to the services (e.g, via MQ Telemetry Transport (MQTT) [1]), when needed vehicle's diagnostics should be synchronously accessed via RESTfull protocols such as CoAP [10] or sMAP [7]. The remote control system requires a dedicated, secure point-to-point connection. Configuring these capabilities should be decoupled from the underlying physical infrastructure, in order to allow dynamic, fine-grained customization.
- The FM system spans multiple, geographically distributed cloud instances and IoT devices that comprise FM's virtual runtime topologies. These topologies abstract a portion of the IoT cloud infrastructure, e.g., needed by specific subsystem, thus they should support flexible configuring to allow for on-demand provisioning.

The limited support for fine-grained provisioning at higher levels leads to tightly coupled, problem specific IoT infrastructure components, which require difficult and tedious configuration management on multiple levels. This inherently makes provisioning and runtime operation of IoT cloud systems a complex task. Consequentially, system designers and operations managers face numerous challenges to provision and operate large-scale IoT cloud systems such as our FM.

B. Challenges

RC1 – The IoT cloud services and subsystems provide different functionality or analytics, but they mostly rely on common physical IoT infrastructure. However, to date the IoT infrastructure resources have been mostly provided as coarse grained, rigid packages, in the sense that the IoT systems, e.g., the infrastructure components and software libraries are specifically tailored for the problem at hand and do not allow for flexible customization and provisioning of the individual resource components or the runtime topologies. This inherently hinders self-service, *utility-oriented delivery and consumption* of the IoT resources at a finer granularity level.

RC2 – *Elasticity*, although one of the fundamental traits of the traditional cloud computing, has not yet received enough attention in IoT cloud systems. Elasticity is a principle to provision the required resources dynamically and on demand, enabling applications to respond to varying load patterns by adjusting the amount of provisioned resources to exactly match their current needs, thus minimizing resources over-provisioning and allowing for better utilization of the available resources [9]. However, IoT cloud systems are usually not tailored to incorporate elasticity aspects. For example, new types of resources, e.g., data streams, delivered by IoT infrastructure are still not provided elastically in IoT cloud systems. Opportunistic exploitation of constrained resources, inherent to many IoT cloud systems further intensifies the need to provision the required resources on-demand or as they become available. These challenges prevent current IoT systems from fully utilizing the benefits cloud's elastic nature has to offer and call for new approaches to incorporate the elasticity capabilities in the IoT cloud systems.

RC3 – Dependability is a general measure of dynamic system properties, such as availability, reliability, fault resilience and maintainability. Cloud computing supports developing and operating dependable large-scale systems atop commodity infrastructure, by offering an abundance of virtualized resources, providing replicated storage, enabling distributed computation with different availability zones and diverse, redundant network links among the system components. However, the challenges to build and *operate dependable large-scale IoT cloud systems* are significantly aggravated because in such systems the cloud, network and embedded devices are converging, thus creating very large-scale hyper-distributed systems, which impose new concerns that are inherently elusive with traditional operations approaches.

RC4 – Due to dynamicity, heterogeneity, geographical distribution and the sheer scale of IoT cloud, traditional management and provisioning approaches are hardly feasible in practice. This is mostly because they implicitly make assumptions such as physical on-site presence, manually logging into devices, understanding device's specifics, etc., which are difficult, if not impossible, to achieve in IoT cloud systems. Thus, novel techniques, which will provide an unified and conceptually centralized view on system's configuration management are needed.

Therefore, we need novel models and techniques to provision and operate the IoT cloud systems, at runtime. Some of the obvious requirements to make this feasible in the very large-scale, geographically distributed setup are: (i) We need tools which will automate development, provisioning and operations (DevOps) processes; (ii) Supporting mechanisms need to be late-bound and dynamically configurable, e.g., via policies; (iii) Configuration models need to be centrally managed and automatically propagated to the edge of the infrastructure.

III. Principles and building blocks of software-defined IoT systems

A. Principles of Software-Defined IoT

Generally, software-defined denotes a principle of abstracting the low-level components, e.g., hardware, and enabling their provisioning and management through a well-defined API [14]. This enables refactoring the underlying infrastructure into finer-grained resource components whose functionality can be defined in software after they have been deployed.

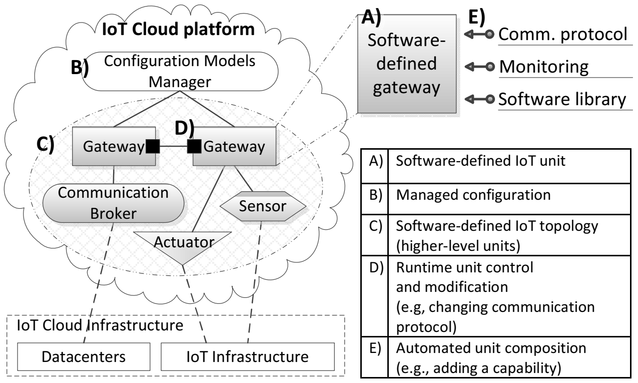
Software-defined IoT systems comprise a set of resource components, hosted in the cloud, which can be provisioned and controlled at runtime. The IoT resources (e.g., sensory data streams), their runtime environments (e.g., gateways) and capabilities (e.g., communication protocols, analytics and data

TABLE I. SUMMARY OF MAIN PRINCIPLES AND ENABLERS OF SOFTWARE-DEFINED IOT SYSTEMS

Research challenges	**High-level principles**	**Enablers**
• Flexible customization	• API encapsulation of IoT resources and capabilities	• Software-defined IoT units
• Utility-oriented delivery and consumption	• Fine-grained resources consumption	• Software-defined IoT topology (complex units)
• Self-service usage model	• Policy-based specification and configuration	• Centrally managed configuration models and policies
• Support for elasticity concerns	• Automated provisioning	• Automated units composition
• Operating dependable large-scale IoT cloud systems	• Cost awareness	• Runtime unit control and modification
• Central point of management	• Runtime elasticity governance	

point controllers) are described as *software-defined IoT units*. *Software-defined IoT units* are software-defined entities that are hosted in an IoT cloud platform and abstract accessing and operating underlying IoT resources and lower level functionality. Generally, *software-defined IoT units* are used to encapsulate the IoT resources and lower level functionality in the IoT cloud and abstract their provisioning and governance, at runtime. To this end, our *software-defined IoT units* expose well-defined API and they can be composed at different levels, creating virtual runtime topologies on which we can deploy and execute IoT cloud systems such as our FM system. Therefore, main principles of software-defined IoT systems include:

- API Encapsulation – IoT resources and IoT capabilities are encapsulated in well-defined APIs, to provide a unified view on accessing functionality and configurations of IoT cloud systems.
- Fine-grained consumption – The IoT resources and capabilities need to be accessible at different granularity levels to support agile utilization and self-service consumption.
- Policy-based specification and configuration – The units are specified declaratively and their functionality is defined programmatically in software, using the well-defined API and available, familiar software libraries.
- Automated provisioning – Main provisioning processes need to be automated in order to enable dynamic, on-demand configuring and operating software-defined IoT systems, on a large-scale (e.g, hundreds gateways).
- Cost awareness – We need to be able to assign and control costs of delivered IoT resources and capabilities in order to enable their utility-oriented consumption.
- Elasticity support – They should support elasticity governance [9], by exposing runtime control of elastic capabilities through well-defined API.



A)	Software-defined IoT unit
B)	Managed configuration
C)	Software-defined IoT topology (higher-level units)
D)	Runtime unit control and modification (e.g, changing communication protocol)
E)	Automated unit composition (e.g., adding a capability)

Fig. 1. Main enablers of software-defined IoT cloud systems

Table I summarizes how we translate the aforementioned high-level principles into concrete enablers. For example, to allow for flexible system customization, we need to enable fine-grained resource consumption, well-defined API encapsulation and provide support for policy-based specification and configuration. These principles are enabled by our software-defined IoT units and support for centrally managed configuration. Figure 1 gives high-level graphical overview of the main building blocks and enabling techniques, needed to support the main principles of software-defined IoT systems. Subsequently, we describe them in more detail.

B. Conceptual Model of Software-defined IoT Units

Figure 2 illustrates the conceptual model of our software-defined IoT units. The units 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 IoT resources and expose them in the IoT cloud. The functional, provisioning and governance capabilities of the units are exposed via *well-defined APIs*, which enable provisioning and controlling the units at runtime, e.g., start/stop. Our conceptual model also allows for composing and interconnecting software-defined IoT units, in order to dynamically deliver the IoT resources and capabilities to the applications. The runtime provisioning and configuration is performed by specifying late-bound policies and configuration models. Naturally, the software-defined IoT units support mechanisms to map the virtual resources with the underlying physical infrastructure.

To technically realize our unit model we introduce a concept of *unit prototypes*. They can be seen as resource containers, which are used to bootstrap more complex, higher-level units. Generally, they are hosted in the cloud and enriched with functional, provisioning and governance capabilities, which are exposed via software-defined APIs. The unit prototypes can be based on OS-level virtualization, e.g., VMs, or more finer-grained kernel supported virtualization, e.g., Linux containers. Conceptually, virtualization choices do not pose any limitations, because by utilizing the well-defined API, our unit prototypes can be dynamically configured, provisioned, interconnected, deployed, and controlled at runtime.

Given our conceptual model (Figure 2), by utilizing the *provisioning API*, the unit prototypes can be dynamically coupled with late-bound runtime mechanisms. These can be any software components (custom or stock), libraries or clients that can be configured and whose binding with the unit prototypes is differed to the runtime. For example, the mechanisms can be used to dynamically add communication capabilities, new functionality or storage to our software-defined IoT units. Therefore, by specifying policies, which are bound later during runtime, system designers or operations managers can flexibly manage unit configurations and customize their capabilities, at *fine granularity levels*. Our conceptual model also allows for composing the software-defined IoT units at higher levels. By selecting dependency units, e.g., based on

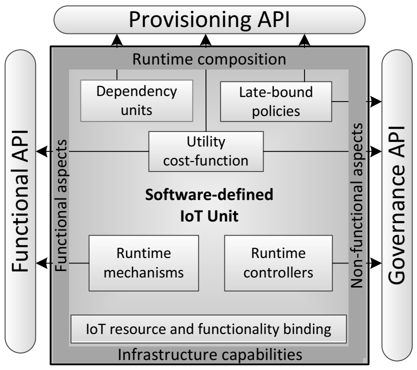


Fig. 2. Conceptual model of software-defined IoT units.

their costs, analytics or elasticity capabilities, and linking them together, we can dynamically build more complex units. This enables flexible *policy-based specification and configuration* of complex relationships between the units. Therefore, by carefully choosing the granularity of our units and providing configuration policies we can *automate the units composition process* at different levels and in some cases completely defer it to the runtime. This makes the provisioning process flexible, traceable and repeatable across different cloud instances and IoT infrastructures, thus reducing time, errors and costs.

The runtime *governance API*, exposed by the units, enables us to perform runtime control operations such as starting or stopping the unit or change the topological structure of the dependency units, e.g., dynamically adding or removing dependencies at runtime. Therefore, one of the most important consequences of having software-defined IoT unit is that the functionality of the virtual IoT infrastructure can be (re)defined and customized after it has been deployed. New features can be added to the units and the topological structure of the dependency units can be customized at runtime. This enables automating provisioning and governance processes, e.g., by utilizing the governance API and providing monitoring at unit level, we can enable *elastic horizontal scaling* of our units.

Therefore, most important features of software-defined IoT units which enable the general principles of software-defined IoT (see Section III-A) are:

- 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, e.g, 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 IoT resources as utilities.

C. Units Classification

Depending on their purpose and capabilities, our software-defined IoT units have different granularity and internal topological structure. Therefore, conceptually we classify them into: (i) *atomic*, (ii) *composed* and (iii) *complex software-defined IoT units*. Depending on their type, the units require specific runtime mechanisms and expose specific provisioning API. Next we describe each unit type in more detail.

The *atomic software-defined IoT units* are the finest-grained software-defined IoT units, which are used to abstract the core capabilities of an IoT resource. They provide software-defined API and need to be packaged portably to include components and libraries, that are needed to provide desired capabilities. Figure 3 depicts some examples of the atomic software-defined units. We broadly classify them into functional and non-functional atomic software-defined IoT units, based on the capabilities they provide. Functional units encapsulate capabilities such as communication or IoT compute and storage. Non-functional units encapsulate configuration models and capabilities such as elasticity controllers or data-quality enforcement mechanisms. Therefore, the atomic units are used to identify fine-grained capabilities needed by an application. For example, the application might require the communication to be performed via a specific transport protocol, e.g., MQTT or it might need a specific monitoring component, e.g., Ganglia[1]. Classifications similar to the one presented in Figure 3 can be used to guide the atomic units selection process, in order to easily identify the exact capabilities, needed by the application.

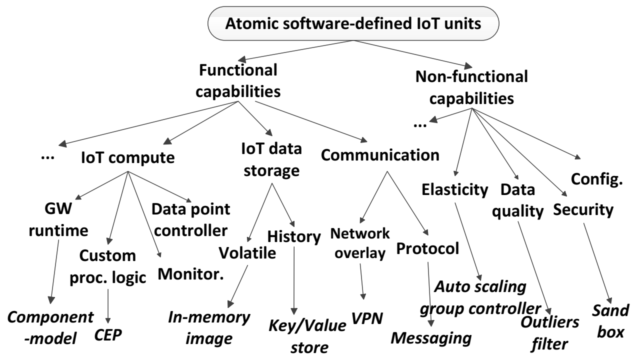


Fig. 3. Example classification of atomic software-defined IoT units.

The *composed software-defined IoT units* have multiple functional and non-functional capabilities, i.e., they are composed of multiple atomic units. Similarly to the atomic units they provide well-defined API, but require additional functionality such as mechanisms to support declaratively composing and binding the atomic units, at runtime (Section IV-B). Example of composed unit is a software-defined IoT gateway.

The *complex software-defined IoT units* enable capturing complex relationships among the finer-grained units. Internally, they are represented as a topological network, which can be configured and deployed, e.g., on the cloud. They define an API and can integrate (standalone) runtime controllers to dynamically (re)configure the internal topology, e.g., to enable elastic horizontal scaling of the units. Finally, they rely on runtime mechanism to manage the references, e.g., IP addresses and ports, among the dependency units.

We notice that the software-defined API and our units offer different advantages to the stakeholders involved into

[1]http://ganglia.info/

designing, provisioning and governing of software-defined IoT systems. For example, IoT infrastructure providers can offer their resources at fine-granularity, on-demand. This enables specifying flexible pricing and cost models and allows for offering the IoT resources as elastic utilities in a pay-as-you-go manner. Because our units support *dynamic and automated composition* on multiple levels, consumers of IoT cloud resources can provision the units to exactly match their functional and non-functional requirements, while still taking advantage of the existing systems and libraries. Further, system designers and operations managers, use late-bound policies to specify and configure the unit's capabilities. Because we treat the functional and configuration units in a similar manner (see Section IV-B), configuration models can be stored, reused, modified at runtime and even shared among different stakeholders. This means that we can support *managed configuration models*, which can be centrally maintained via configuration management solutions for IoT cloud, e.g., based on OpsCode Chef[2], Bosh[3] or Puppet[4].

IV. Provisioning software-defined IoT cloud systems

A. Automated composition of software-defined IoT units

Generally, building and deploying software-defined IoT cloud systems includes creating and/or selecting suitable software-defined IoT units, configuring and composing more complex units and building custom business logic components. The deployment phase includes deploying the software-defined IoT units together with their dependency units and required (possibly standalone) runtime mechanisms (e.g., a message broker). In this paper we mostly focus on provisioning reusable stock components such as gateway runtime environments or available communication protocols. Developing custom business logic components is out of scope of this paper and we address it elsewhere [15].

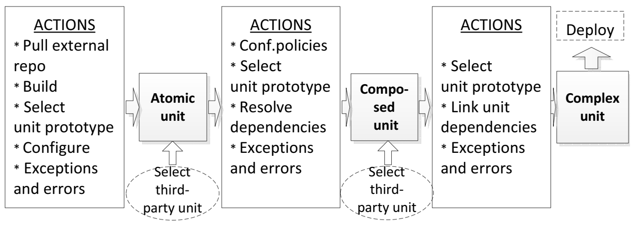


Fig. 4. Automated composition of software-defined IoT units.

Figure 4 illustrates most important steps to compose and deploy our IoT units. There are three levels of configuration that can be performed: (i) Building/selecting atomic units; (ii) Configuring composed units; (iii) Linking into complex units. Each of the phases includes selecting and provisioning suitable unit prototypes. For example, the unit prototypes can be based on different resource containers such as VMs, Linux Containers (e.g., Docker) or OSGi runtime.

The atomic units are usually provided as stock components, e.g., by a third-party, possibly in a market-like fashion. Therefore, this phase usually involves selecting and configuring stock components (e.g., Sedona[5] or Niagara^{AX}[6] execution environments). Classifications similar to the one presented in Figure 3 can be used to guide the atomic units selection process. In case we want to perform custom builds of the existing libraries and frameworks, there are many established build tools which can be used, e.g., for Java-based components, Apache Ant or Maven.

On the second level, we configure the composed units, e.g., a software-defined IoT gateway. This is performed by adding the atomic units (e.g., runtime mechanisms and/or software libraries) to the composed unit. For example, we might want to enable the gateway to communicate over a specific transport protocol, e.g., MQTT and add a monitoring component to it, e.g., a Ganglia agent. To perform this composition seamlessly at runtime, additional mechanisms are required. We describe them in Section IV-B.

Third level includes defining the dependencies references between the composed units, which ”glue together” the complex units. These links specify the topological structure of the desired complex units. For example, to this end we can set up a virtual private network and provide each unit with a list of IP addresses of the dependency units. In this phase, we can use frameworks (e.g., TOSCA-based, OpenStack Heat, Amazon CloudFormation, etc.) to specify the runtime topological structure of our units and utilize mechanisms (e.g., Ubuntu CloudInit[7]) to bootstrap the composition, e.g, pass the references to the dependency units.

B. Centrally managed configuration models and policies

An important concept behind software-defined IoT cloud systems is the late-bound runtime policies. Our units are configured declaratively, via the policies by utilizing the exposed software-defined API, without worrying about internals of the runtime mechanisms, i.e, the atomic units. To enable seamless binding of the atomic units we provide a special unit prototype, called *bootstrap container*. The bootstrap container acts as a plug-in system, which provides mechanisms to define (bind) the units based on supplied configurations or to redefine them when configuration policies are changed. For example, runtime changes of the units are achieved by invalidating affected parts of the existing dependency tree and dynamically rebuilding them, based on the new configuration directives. Therefore, the units can be simply ”droped in” and our bootstrap container (re)binds them together, at runtime without rebooting system.

We decouple the configuration models (late-bound policies) from the functional units. Therefore, we can treat configuration policies as any software-defined IoT unit, which adheres to the general principles of software-defined IoT (Section III-A). By encapsulating the configuration policies in separate units, we can manage them at runtime via centralized configuration managements solutions for IoT cloud. Our framework provides mechanisms to specify and propagate the configuration models to the edge of IoT cloud infrastructure (e.g., gateways) and our bootstrap container enforces the provided directives. To this end, our bootstrap container initially binds functional and configuration units and continuously listens for configuration changes and applies them on the affected functional units, accordingly. To enable performing runtime modifications without

[2]http://opscode.com/chef
[3]http://docs.cloudfoundry.org/bosh/
[4]http://puppetlabs.org
[5]http://www.sedonadev.org/
[6]http://www.niagaraax.com/
[7]http://help.ubuntu.com/community/CloudInit/

worrying about any side-effects we require the configuration actions to be idempotent. The usual approach to achieve this is to wrap the units as OS services. Among other things the late-bound policies and our mechanisms for managed configuration enable flexible customization and dynamic configuration changes, at runtime.

V. Prototype and experiments

A. Prototype implementation

The main aim of our prototype is to enable developers and operations managers to dynamically, on-demand provision and deploy software-defined IoT systems. This includes providing software-defined IoT unit prototypes, enabling automated unit composition, at multiple levels and supporting centralized runtime management of the configuration models.

In Section III we introduced the conceptual model of our software-defined IoT units. To technically realize our units, we utilize the concept of virtual resource containers. More precisely, we provide different *unit prototypes* that can be customized and/or modified during runtime by adding required runtime mechanisms encapsulated in our atomic units. The unit prototypes provide resources with different granularity, e.g., VM flavors, group quotas, priorities, etc., and boilerplate functionality to enable automated provisioning of custom software-defined IoT units.

Figure 5 provides a high-level overview of the framework's architecture. Our framework is completely hosted in the cloud and follows a modular design which guarantees flexible and evolvable architecture. The current prototype is implemented atop OpenStack [2], which is an open source Infrastructure-as-a-Service (IaaS) cloud computing platform. *Presentation layer* provides an user interface via Web-based UI and RESTful API. They allow a user to specify various configuration models and policies, which are used by the framework to compose and deploy our units in the cloud. *Cloud core services layer* contains the main functionality of the framework. It includes the *PolicyProcessor* used to read the input configurations and transform it to the internal model defined in our framework. *Units management services* utilize this model for composing and managing the units. The *InitializationManager* is responsible for configuring and composing more complex units. It translates the directives specified in configuration models into concrete initialization actions on the unit level. In our current

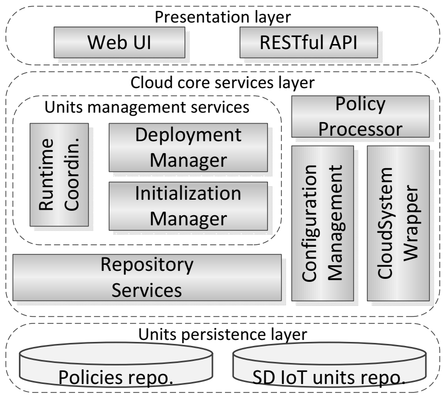


Fig. 5. Framework architecture overview.

implementation, the core of the *InitalizationManager* is an OpsCode Chef client, which is passed to the VMs during initialization via Ubuntu cloud-init. *InitalizationManager* also provides mechanisms for configuration management. The *DeploymentManager* is used to deploy the software-defined IoT units in the cloud. Our prototype relies on SALSA[8], a deployment automation framework we developed. It utilizes the API exposed by the *CloudSystemWrapper* to enable deployment across various cloud providers, currently implemented for OpenStack cloud. *DeploymentManager* is responsible to manage and distribute the dependency references for the complex units (Section III-C). *Units persistence layer* provides functionality to store and manage our software-defined units and policies.

B. Experiments

1) Scenario analysis: We now show how our prototype is used to provision a complex software-defined IoT unit, which provides functionality for the real-life FM location tracking service (Section II-A). The service reports vehicle location in near real-time on the cloud. To enable remote access, the monitored vehicles have an on-board device, acting as a gateway to its data and control points. To improve performance and reliability, the golf course provides on-site gateways, which communicate with the vehicles, provide additional processing and storage capabilities and feed the data into the cloud. Therefore, the physical IoT infrastructure comprises network connected vehicles, on-board devices and local gateways.

Typically, to provision the FM service system designers and operations manager would need to directly interact with the rigid physical IoT infrastructure. Therefore, they at least need to be aware of its topological structure and devices' capabilities. This means that the FM service also needs to have understanding of the IoT infrastructure, instead of being able to customize the infrastructure to its needs. Due to inherent inflexibility of IoT infrastructure, its provisioning usually involves long and tedious task such as manually logging into individual gateways, understanding gateway internals or even on site presence. Therefore, provisioning even a simple FM location tracking service involves performing many complex tasks. Due to a large number of geographically distributed vehicles and involved stakeholders IoT infrastructure provisioning requires a substantial effort prolonging service delivery and increasing costs. Subsequently, we show the advantages our units (Section III-B) and the provisioning techniques (Section IV) have to offer to operations managers and application designers in terms of: a) *Simplified provisioning* to reduce time, costs and possible errors; b) *Flexibility* to customize and modify the IoT units and their runtime topologies.

To enable the FM system we developed a number of atomic software-define IoT units[9] such as: a software-defined sensor that reports vehicle location in realtime, messaging infrastructure based on Apache ActiveMQ[10], software-defined protocol based on MQTT and JSON, the bootstrap container based on the Spring framework[11], and corresponding configuration units. The experiments are simulated on our OpenStack (Folsom) cloud and we used Ubuntu 12.10 cloud image (Memory:

[8]https://github.com/tuwiendsg/SALSA/

[9]https://github.com/tuwiendsg/SDM

[10]http://activemq.apache.org/

[11]http://projects.spring.io/spring-framework/

2GB, VCPUS: 1, Storage: 20GB). To display location changes we develop a Web application which displays changes of vehicles' location on Google Maps.

2) Simplified provisioning: To demonstrate how our approach simplifies provisioning of the virtual IoT infrastructure, we show how a user composes the FM complex software-defined IoT unit, using our framework. Figure 6 shows the custom deployment of the topological structure of the FM vehicle tracking unit, deployed in the cloud. The unit contains two gateways for the vehicles it tracks, a web server for the Web application and a message broker that connects them.

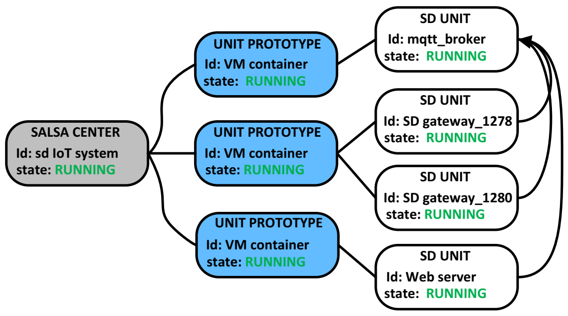


Fig. 6. Topological structure of FM vehicle tracking unit (a screen shot).

In order to start provisioning the complex unit, system designer only needs to provide a policy describing the required high-level resources and capabilities required by the FM service. For example, Listing 1 shows a snippet from the configuration policy for FM location tracking unit, that illustrates specifying a software-defined gateway, for the on-board device.

```
...
<tosca:NodeTemplate id="SD-Gateway"
  name="car_1278" type="vm">
  <tosca:Properties>
   <MappingProperties>
    <MappingProperty type="vm">
     <property name="instanceType">m1.small</property>
     <property name="provider">openstack@dsg</property>
     <property name="baseImage">ami-00000163</property>
    </MappingProperty>
   </MappingProperties>
  </tosca:Properties>
  <tosca:Requirements>
   <tosca:Requirement name="MQTT-broker-IP" type="String"
   id="brokerIp_Requirement"/>
  </tosca:Requirements>
  <tosca:DeploymentArtifacts>
   <tosca:DeploymentArtifact artifactType="chef"
   artifactRef="deployClient"/>
  </tosca:DeploymentArtifacts>
</tosca:NodeTemplate>
...
```

Listing 1. Partial TOSCA-like complex unit description.

The policy describes gateway's initial configuration and the cloud instance where it should be deployed. Additionally, it defines a dependency unit, i.e. the MQTT broker and specifies vehicle's Id that can be used to map it on the underlying device, as shown in [15]. Our framework takes the provided policy, spawns the required unit prototypes and provides them with references to the dependency units. At this stage the virtual infrastructure comprises solely of unit prototypes (VM-based). After performing the high-level unit composition and establish the dependencies between the units, the user continues composing on the finer granularity level. By applying the top-down approach we enable differing design decisions and enable early automation of known functionality, to avoid over-engineering and provisioning redundant resources.

In the next phase, the user provisions individual unit prototypes. To this end, he/she provides policies specifying desired finer-grained capabilities. Listing 2 shows example capabilities, that can be added to the gateway. To enable asynchronous pushing of the location changes it should communicate over the MQTT protocol. Listing 3 shows a part of Chef recipe used to add MQTT client to the gateway. Our framework fetches the atomic units, that encapsulate the required capabilities, from the repository and composes them automatically, relying on the software-defined API and our bootstrap container.

```
{"run_list":
  ["recipe[bootstrap_container]",
  "recipe[mqtt-client]",
  "recipe[protocol-config-unit]",
  "recipe[sd-sensor]"]
}
```

Listing 2. Run list for software-defined gateway.

```
include_recipe 'bootstrap_container::default'
remote_file "mqtt-client-0.0.1-SNAPSHOT.jar" do
 source "http://128.130.172.215/salsa/upload/files/..."
 group "root"
 mode 00644
 action :create_if_missing
end
```

Listing 3. Chef recipe for adding the MQTT protocol.

Therefore, compared to the traditional approaches, which require gateway-specific knowledge, using proprietary API, manually logging in the gateways to set data points, our *automated units composition* (Section IV-A), based on declarative unit configuration policies, simplifies the provisioning process and makes it traceable and repeatable. Our units can easily be shared among the stakeholders and composed to provide custom functionality. This enables system designers and operations managers to rely on the existing, established systems, thus reducing provisioning time, potential errors and costs.

3) Flexible customization: To exemplify the flexibility of our approach let us assume that we need to change configuration of the FM unit to use CoAP instead of MQTT. This can be due to requirements change (Section II-A), reduced network connectivity or simply to reuse the unit for a golf course with different networking capabilities. To customize the existing unit, an operations manager only needs to change the `"recipe[protocol-config-unit]"`unit (Listing 2) and provide an atomic unit for the CoAP client. This is a nice consequence of our late-bound runtime mechanisms and support for *managed configuration models*, provided by our framework. We treat both functional and configuration units in the same manner and our bootstrap container manages their runtime binding (Section IV-B). Compared to traditional approaches that require addressing each gateway individually, firmware updates or even modifications on the hardware level, our framework enables flexible runtime customization of our units and supports operation managers to seamlessly enforce configuration baseline and its modifications on a large-scale.

VI. Related Work

Recently, many interesting approaches enabling convergence of IoT and cloud computing appeared. For example, in [5], [8], [11], [16], [18] the authors mostly deal with IoT infrastructure virtualization and its management on cloud platforms, [6], [13] utilize the cloud for additional computation resources, in [17], [19] the authors mostly focus on utilizing cloud's storage resources for Big-IoT-Data and [12], [16] integrating IoT devices with enterprise applications based on SOA paradigm. Due to limited space in the following we only mention related work exemplifying approaches that deal with IoT infrastructure virtualization and management.

In [11] 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. In [18] the authors introduce sensor-cloud infrastructure that virtualizes physical sensors on the cloud and provides management and monitoring mechanisms for the virtual sensors. However, their support for sensor provisioning is based on static templates that, contrary to our approach, do not support dynamic provisioning of IoT capabilities such as communication protocols. SenaaS [5] mostly focuses on providing a cloud semantic overlay atop physical infrastructure. They define an IoT ontology to mediate interaction with heterogeneous devices and data formats, exposing them as event stream services to the upper layers. OpenIoT framework [16] utilizes semantic web technologies and CoAP [10] to enable web of things and linked sensory data. They mostly focus on discovering, linking and orchestrating internet connected objects, thus conceptually complementing our approach. In [8] 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, all of which can be seen as enablers of our approach. Also there are various commercial solutions such as Xively [4] and ThingWorx [3], which allow users to connect their sensors to the cloud and enable remote access and management.

Most of these approaches focus on different virtualization techniques for IoT devices and data format mediation. They also enable some form of configuration, e.g., setting sensor poll rates. These approaches can be seen as complementary to our own, as device virtualization sets the corner stone for achieving software-defined IoT systems. We rely on the advances in convergence of IoT and cloud, introduced by the previous work and extend it with novel concepts for abstracting and encapsulating IoT resources and capabilities, exposing them via software-defined API on the cloud and enabling fine-grained provisioning. Therefore, our approach can be seen as a natural step in evolution of IoT cloud systems.

VII. Conclusion

In this paper, we introduced the conceptual model of software-defined IoT units. To our best knowledge this is the first attempt to apply software-defined principles on IoT systems. We showed how they are used to abstract IoT resources and capabilities in the cloud, by encapsulating them in software-defined API. We presented automated unit composition and managed configuration, the main techniques for provisioning software-defined IoT systems. The initial results are promising in the sense that software-defined IoT system enable sharing of the common IoT infrastructure among multiple stakeholders and offer advantages to IoT cloud system designers and operations managers in terms of simplified, on-demand provisioning and flexible customization. Therefore, we believe that software-defined IoT systems can significantly contribute the evolution of the IoT cloud systems.

In the future we plan to continue developing the prototype and extend it in several directions: a) Providing techniques and mechanisms to support runtime governance of software-defined IoT systems; b) Enabling our software-defined IoT systems to better utilize the edge of infrastructure, e.g., by providing code distribution techniques between cloud and IoT devices; c) Enabling policy-based automation of data-quality, security and safety aspects of software-defined IoT systems.

Acknowledgment

This work is sponsored by Pacific Controls Cloud Computing Lab (PC3L[12])

References

[1] Mq telemetry transport. http://mqtt.org/. March 2014.

[2] Openstack. http://www.openstack.org/. March 2014.

[3] Thingworks. http://thingworx.com. Last accessed: March 2014.

[4] Xively. https://xively.com. Last accessed: March 2014.

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

[6] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*. ACM, 2011.

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

[8] S. Distefano, G. Merlino, and A. Puliafito. Sensing and actuation as a service: a new development for clouds. In *NCA*, pages 272–275, 2012.

[9] S. Dustdar, Y. Guo, B. Satzger, and H.-L. Truong. Principles of elastic processes. *Internet Computing, IEEE*, 15(5):66–71, 2011.

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

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

[12] M. Kovatsch, M. Lanter, and S. Duquennoy. Actinium: A restful runtime container for scriptable internet of things applications. In *Internet of Things*, pages 135–142, 2012.

[13] K. Kumar and Y.-H. Lu. Cloud computing for mobile users: Can offloading computation save energy? *Computer*, 43(4):51–56, 2010.

[14] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. ACM, 2010.

[15] 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.

[16] J. Soldatos, M. Serrano, and M. Hauswirth. Convergence of utility computing with the internet-of-things. In *IMIS*, pages 874–879, 2012.

[17] P. Stuedi, I. Mohomed, and D. Terry. Wherestore: Location-based data storage for mobile devices interacting with the cloud. In *ACM Workshop on Mobile Cloud Computing & Services*. ACM, 2010.

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

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

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