PatRICIA – a Novel Programming Model for IoT Applications on Cloud Platforms

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

***Abstract*—Cloud computing technologies have recently been intensively exploited for the development and management of large-scale IoT systems, due to their capability to integrate diverse types of IoT devices and to support big IoT data analytics in an elastic manner. However, due to the diversity, complexity and scale of IoT systems, the need to handle large volumes of IoT data in a nontrivial manner, and the plethora of domain-dependent IoT controls, programming IoT applications on cloud platforms still remains a great challenge. To date, existing work neglects high-level programming models and focuses on low-level IoT data and device integration. In this paper, we outline PatRICIA, which aims at providing an end-to-end solution for high-level programming and provisioning of IoT applications on cloud platforms. We present a novel programming model, based on the concept of *intent* and *intent scope*. Further, we introduce its runtime for dealing with the complexity, diversity and scale of IoT systems in the cloud. Our programming model defines abstractions to enable easier, efficient and more intuitive development of *cloud-scale IoT applications*. To illustrate our programming model, we present a case study with real-world applications for controlling and managing electric vehicles.**

I. Introduction

Advances in the Internet of Things (IoT) have provided a global infrastructure of networked physical entities, able to monitor and control their physical status and surrounding environment, as well as to expose themselves via data streams and services over the network [10], [17], [19]. Various enterprise systems, e.g., smart building management system [6] and smart healthcare[14], utilize IoT applications to optimize key tasks of their business processes. Recently, cloud computing has become the key enabler for large-scale IoT systems. Researchers (e.g., [21], [16], [23]) recognize the benefits of exploiting cloud computing for IoT systems, as it could offer better solutions to support IoT applications in terms of device virtualization, provisioning of virtual sensors and actuators, and providing suitable execution infrastructure for resource-intensive IoT applications.

However, to enable the development of cloud-scale IoT applications, we need high-level abstractions and mechanisms, which support scalable and efficient programming with diverse device services and raw data streams on cloud platforms. There are several approaches, which rely on service-oriented computing (SOC) principles to abstract device data- and actuation-points, e.g., [15]. They deal with heterogeneous devices by enabling direct, service-based access to devices and provide mechanisms for service discovery, provisioning and management. However, their support is usually restricted to the device-level services. Recently, several interesting attempts to apply cloud computing technologies in large-scale IoT systems have emerged, e.g., [23], [21]. They mostly focus on data and device integration by utilizing cloud infrastructure and virtualizing individual sensors and actuators as services in the cloud. Although these approaches help simplify the development of the IoT applications, the development process mostly involves composing these device-level services into admissible control sequences or data processing schemes, which makes the development of IoT applications highly complex, and potentially limits the programming scale of such applications on cloud platforms. Concrete abstractions and mechanisms, which enable efficient, more intuitive and scalable development of cloud-based IoT applications still remain underdeveloped.

In this paper we present a novel programming model for IoT applications on cloud platforms. It is one of the fundamental elements of our PatRICIA framework. PatRICIA aims to provide an ecosystem for development and provisioning of cloud-scale IoT applications. The programming model defines high-level programming constructs and operators, which encapsulate domain-specific knowledge (domain model and behavior) and raise the level of programming abstraction, enabling developers to implement cloud-scale IoT applications without worrying about the complexity of low-level device services and raw sensory data streams. This paper contributes our programming model, which enables easier, efficient and more intuitive development of cloud-scale IoT applications.

The remainder of the paper is organized as follows. In Section II, we present the motivating scenario and the key research challenges. Section III outlines the PatRICIA framework and Section IV presents the programming model. Section V describes the implementation and evaluation of our system. Section VI compares our approach with the related research. Finally, Section VII concludes the paper and provides an outlook on future research.

II. Motivation and Research challenges

A. Motivating Scenario

Our work is motivated by a real-world scenario in remote fleet vehicles management in environments, such as golf courses, university campuses and airports. The scenario is based on a case study we conducted in the Pacific Controls Cloud Computing Lab[1] (PC^3L). In the case study we have identified three stakeholders: vehicle manufacturer, distributors and environment managers. To optimize tasks, crucial for their business processes, the stakeholders can utilize IoT applications and services, which remotely, in near real-time monitor and control the underlying physical assets, in this case the fleet. To enable remote access to vehicles' data and control points each vehicle has an on-board device, acting as a gateway to various sensors and actuators installed in the vehicle.

The IoT applications are characterized by a reactive behavior. They receive some (*monitoring*) information, e.g., a change in vehicles' operation and, as a response, perform (*control*) actions on the vehicles. Typically, these applications

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

monitor vehicle's status like maintenance and fault history, battery health, engine status, location, tire pressure and so forth. For example, an application needs to determine if a vehicle is consuming more energy compared to other vehicles in the fleet, i.e., to detect high *energy fault*. To this end, we can utilize power- and odometer from fleet vehicles and correlate this information in order to detect the fault. Further, our applications react to changes, e.g., in vehicle operation, by taking appropriate actions. For example, if an *energy fault* is detected the application can decide to *notify* the driver and the golf course manager via available devices, e.g., a smartphone or to put the vehicle in a *reduced energy* mode, by setting speed, RPM and transmission limits, and wait for the vehicle to return to the base where it can be further examined.

To perform the above-mentioned tasks, among other things, the IoT applications require complex and expensive analytics and have high demand on storage and communication resources. Because these applications connect to and deal with a large number of vehicles, which are distributed across different golf courses, they must be able to handle vast amounts of data efficiently and need to have a global view of the distributed fleet. To support these requirements, it is natural to execute these applications in the cloud, as it has capabilities to connect, provide access, and a unified global view of the geographically distributed fleet. The applications are envisioned to run continuously, but they can be elasticaly scaled down in off-peek times, e.g., during the night, when most of the vehicles remain dormant. In this case, the elastic nature of the cloud can provide advantages in terms of cost reductions and greener IoT computing, e.g., because of reduced energy consumption. Due to the multiplicity of the involved stakeholders with diverse requirements and business models, cloud-scale IoT applications need to support different and customizable usage experiences. Also here the cloud computing is essential, as it potentially offers new, possibly cross-domain, application opportunities and enables flexible business and usage models. Therefore, in this context, the cloud plays a crucial role, as existing enterprise-specific platforms are hardly capable to meet all of these requirements.

B. Research Challenges

We identify several challenges regarding the development of the cloud-scale IoT applications, as a developer needs to: *(RC1)* Work with raw sensory data streams and write complex queries and event processing schemes (**monitor tasks**). *(RC2)*A developer needs a good knowledge about diverse low-level device services and implications of invoking these atomic services, to be able to establish correct dependencies between them and compose them into admissible control sequences (**control tasks**). *(RC3)* The cloud-scale IoT applications execute in very dynamic environments and interact with hundreds or thousands of physical entities. Therefore, monitoring and controlling these entities in a scalable manner is another challenge for developers of IoT applications on cloud platforms, because applications need to dynamically identify **the scope** of their actions, depending on the task-at-hand. *(RC4)* Finally, due to dynamicity of environments, diversity of devices, ad hoc requirements of diverse stakeholders, and hardware or network failures, developing security-, privacy-, safety-, cost- and quality-aware IoT applications is a very challenging task without adequate **runtime mechanisms** to support it.

Figure 1 visualizes the structure of a cloud-scale IoT application. We notice that applications define custom *business logic*, but can utilize similar *monitor and control tasks* and need to apply them on dynamically defined *scopes* to manage the entities in a scalable manner. Therefore, these tasks can be modeled and represented, in such a manner to enable their reuse across different cloud-scale IoT applications. We show that most of the current approaches (see Section VI) that support the development of IoT applications deal with device and data integration and focus mostly on the *Device virtualization layer* or the layers below (see Figure 1), thus, application developers have to deal with much of the complexity, diversity and scale of IoT applications executed on cloud platforms.

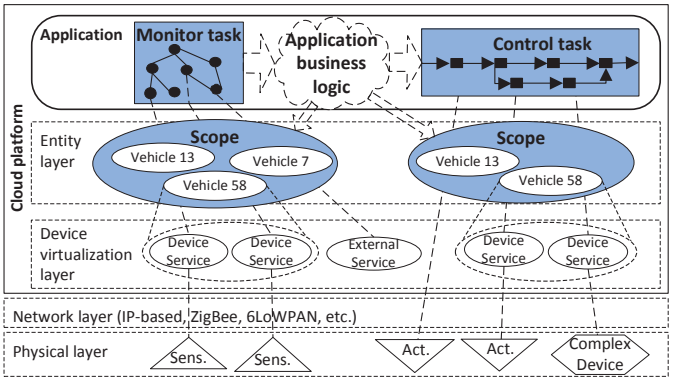

Fig. 1: Example of a cloud-scale IoT application.

III. PatRICIA – a Programming Framework for IoT applications

A. Requirements

Contemporary cloud techniques, e.g., virtualization, elastic scaling, resource and tenant management play a crucial role in highly dynamic and heterogeneous IoT systems. Although, they enable us to virtualize and connect vast amounts of devices, provide a unified view on IoT infrastructure and offer theoretically unlimited processing and storage capabilities, we still need to reduce the complexity and enable scalable development of IoT applications. Therefore, this requires us to rethink the existing development, deployment, execution and provisioning models for IoT applications and infrastructure resources.

The main aim of the **PatRICIA** ***(PR**ogramming **I**ntent-based **C**loud-scale **I**oT **A**pplications)*** framework is to define an ecosystem, which provides an end-to-end solution for cloud-scale IoT applications. This includes providing a programming model and development tools, application execution platform, big IoT data management techniques, and mechanisms to provision, operate and manage IoT infrastructure resources. The core idea of the PatRICIA is to enable the development of value-added IoT applications, which are executed and provisioned on cloud platforms but leverage data from different sensor devices and enable timely propagation of decisions, crucial for business operation, to the edge of the infrastructure.

Programming cloud-scale IoT applications requires different skills and backgrounds, e.g., working with low-level hardware, developing enterprise applications and having knowledge about the domain of interest. Therefore, our framework needs to provide different *logical views* on the development process and enable different developer roles to coherently encapsulate their expertise and focus their development effort. We noticed

that most of the tasks performed by the applications are generic, in the sense, they capture knowledge and industrial best practices in the domain. Therefore, they need to be represented as generic components that can be easily reused. To enable working with these generic tasks in a *cloud-scale manner*, we need to enable automatic task instantiation for developer-defined scopes, e.g., a golf course or the fleet. To keep our programming model stable and easily extensible, we need to enable late runtime binding of the tasks, i.e., decouple task representation from the implementation of its behavior.

Different runtime mechanisms are needed to enable the applications to adapt to changes in quality and costs but also to guarantee safety and security for both users and devices. Further, we need to provide code-distribution techniques, which will allow cloud-scale IoT applications to utilize the edge of the infrastructure (e.g., gateways) as the additional resources, e.g., processing and storage. Application deployment and infrastructure administration need to become fully automated, due to the scale of IoT systems and because domain-specific verticals are increasingly becoming refactored, enabling development of cross-domain cloud-scale IoT applications.

Finally, the cloud-scale IoT applications utilize big IoT data, thus developers require high-level programming techniques that enable scalable and flexible access and analytics of the big IoT data. Therefore, the major requirements for our framework include:

1) Providing a programming model to raise the level of programming abstraction by decoupling domain knowledge implementation from its representation and usage.
2) Providing a cloud-based application execution environment and the supporting runtime mechanisms.
3) Providing development tools, such as testing and staging environments to fully support application development lifecycle.
4) Policy-based automation to enable development of security-, privacy-, safety-, cost- and quality-aware applications.
5) Enabling cloud-scale IoT applications to define and adapt their execution environment, by enabling dynamical, on demand provisioning of IoT infrastructure resources.
6) Providing high-level programming techniques and runtime mechanisms for big IoT data management and analytics.

In this paper we focus primarily on the requirements 1 and 2 and present the design and evaluation of the components to support them.

B. PatRICIA overview

Figure 2 gives an overview of PatRICIA's architecture. Our framework architecture is based on SOA design principles which enable flexible, adaptable and evolvable architecture. The modularity of the framework enables extending the current prototype with future concepts and allows flexible configuring and scaling of individual components atop cloud infrastructure.

Development support layer contains development tools, which are needed to support application development lifecycle and enable provisioning of cloud-scale IoT applications. The programming model is the most important component of this layer as it enables development of cloud-scale IoT applications. *ApplicationManager* is responsible for application configuration, deployment and licensing. Also, this layer integrates a testing environment for the cloud-scale IoT applications.

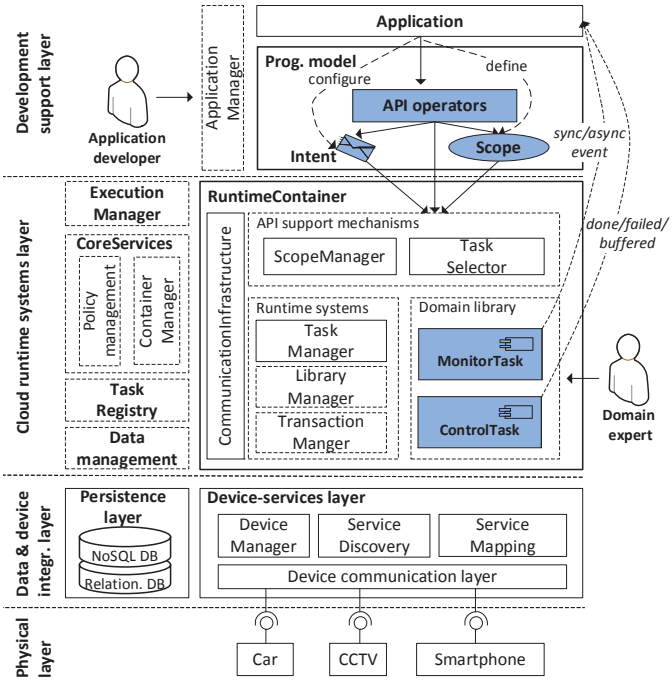

Fig. 2: PatRICIA architecture overview.

Cloud runtime systems layer includes the *RuntimeContainer*, which implements supporting mechanisms for our programming model and provides an execution environment for the applications. The *ExecutionManager* is responsible to monitor applications and the *RuntimeContainer*, and provide mechanisms to elastically scale them during runtime. The *CoreServices* contain a variety of runtime mechanisms and services which are needed by the PatRICIA framework. For example, the *ContainerManager* is used to configure the *RuntimeContainer* and deploy different policies. Further, the *Policy management* component provides mechanisms to specify and enforce these policies, e.g., costs, privacy, security and safety. The *TaskRegistry* is used to store task templates and their metadata. The *Data management* component provides storage, manipulation and analytics mechanisms for the sensory data. It also provides a data quality assurance service, accessible to the runtime container to perform data quality checks.

Data and device integration layer includes *Device-services layer*, which is the IoT device virtualization and management layer of the PatRICIA framework and it underpins monitor and control tasks. It contains *Device communication layer* that implements different connectors, which encapsulate device-specific APIs, communication protocol and enable device-cloud connectivity. The *ServiceMapping* component wraps a physical device and exposes it as a service which defines push, poll, pub and sub methods, providing a communication interface with the devices. *DeviceManager* is responsible for device management, e.g., to detect newly connected devices. The *ServiceDiscovery* component enables discovering and registering device-services. Finally, *Persistence layer* contains NoSQL and relational database, which are used to store the sensory data and other information, needed by the PatRICIA framework.

The PatRICIA framework defines two *logical views* on the application development process and provides support for the

domain expert and application developer roles. Domain experts (Figure 2 right-hand side) use device-services provided by the PatRICIA framework to define domain model and common monitor and control tasks, which form *domain libraries*. In the rest of the paper, however, we mostly focus on the support provided to the application developer role (Figure 2 top-left), by providing the **programming model, runtime container** and a prototype implementation of a vehicle management **domain library**.

IV. Intent-based programing model for cloud-scale IoT applications

A. Basic programming constructs and operators

Our programming model defines constructs and operators, used by developers, to write cloud-scale IoT applications. They enable the developers to work with predefined control and monitor tasks that are provided in the domain library. A control task is any permissible sequence of actuating steps which can be used to control physical devices. Further, a monitor task includes processing, correlation and analysis of sensory data streams to provide meaningful information about the state changes of the underlying environment.

At the application level, we provide explicit representation of these tasks via *Intent*s, i.e., developers write *Intents* to configure and invoke the tasks. When a task is invoked, it is automatically instantiated for the supplied *IntentScope*. Developers use *IntentScope* to delimit the range of an *Intent*. For example, a developer might want to code the expression: "stop all vehicles on hole 1". In this case, "stop" is the desired *Intent*, which needs to be applied on a *IntentScope* that encompasses all vehicles with the location property "hole 1". In our programming model *Intent* and *IntentScope* are first-class entities. This means that they can be stored in variables, used in expressions and passed as parameters to functions.

1) IntentScope: *IntentScope* is an abstraction, which represents a group of physical entities (e.g., vehicles), which share some common properties, e.g., context. More precisely, it is a set of software entities on the cloud platform, which virtualize corresponding physical entities. Thus, *IntentScope*s are determined on the cloud platform, but they enable developers to dynamically delimit physical entities on which an *Intent* will have an effect. In reality there are infinitely many scopes, which can be defined by the applications and can include hundreds of diverse, geographically distributed vehicles. Therefore, we provide mechanisms to dynamically define and work with *IntentScope*s on the cloud platform.

To define an *IntentScope* developers specify properties, which need to be satisfied by the physical entities to be included in the scope. For example, *IntentScope*s can be defined based on a behavior, e.g., "all vehicles exceeding speed limit", a state ("all vehicles with low battery") or a static feature ("all vehicles with a price over ..."). To enable *IntentScope* bootstrapping, we provide a special type of *IntentScope*, which is called *GlobalScope*. It defines the maximal scope for an application and usually contains all physical entities administered by a stakeholder at the given time. Therefore, it is reasonable to assume that the *GlobalScope* is slow-changing over time and it can be configured by a user, e.g., a golf course manager. In PatRICIA the *GlobalScope* is represented as a global variable, which can be directly referenced by an application. Contrary, the minimal *IntentScope*, which can be referenced by an application is a single entity. Our programming model allows *IntentScopes* to be defined explicitly and implicitly. To explicitly define an *IntentScope*, a developer can manually add the entities to the scope by specifying their Ids. Implicit definition of the scope is usually performed by recursively pruning the *GobalScope* and/or combining two or more *IntentScopes*.

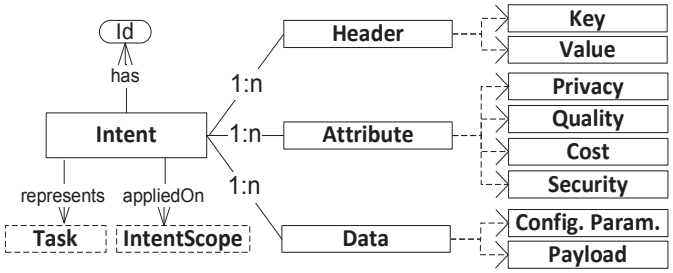

Fig. 3: Intent structure.

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 as 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 as S_{min}. Further, 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: *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*.

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

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, i.e., $E \in \hat{S} \mid E \in S \wedge cond(E) = True$. In this context $cond$ can be $True$ or $False$ depending if an entity satisfies specified property, e.g., if it displays "EnergyFault". To define the $cond$ developers can use any monitor task defined in our domain library and need to provide a parametrized condition expression, e.g, "EnergyFault==True". Further, we provide the common operations on sets, i.e., $\cup, \cap$ and $\setminus$, which have their traditional meaning.

By introducing *IntentScope*s at the application level, we enable development of IoT applications in a scalable manner by shielding the developers from directly referencing the vast number of diverse physical entities and enabling them to dynamically delimit the range of *Intent*s. With this, we address the *(RC3)* from Section II-B.

2) Intent: *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 cloud platform, but enable monitoring and controlling of the physical environments. Based on the information contained in an *Intent*, a suitable task is dynamically *selected, instantiated and executed* on the cloud platform. Our framework translates the *Intent* into a sequence of actuation or data processing steps and maps them on the underlying physical devices (see Section IV-C2). Depending on the task's nature, we distinguish two different types of *Intent*s: *ControlIntent* and *MonitorIntent*. *ControlIntent*s enable applications to provision, operate and manage the low-level components. They abstract

the underlying devices and provide a high-level representation of their functionality. *MonitorIntent*s are used by applications to subscribe for events from the underlying environment and to obtain and provision devices' context.

Figure 3 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 provide information, such as costs, quality, privacy or security requirements. They describe the *Intent* to more detail and are used by the runtime to select the best matching task instance in case there are multiple implementations supporting the *Intent*. Finally, *Intent* can contain data, which is used to configure the tasks and devices or supply additional payload, e.g., a notification message.

To perform an IoT control or to subscribe for relevant events, developers only need to define and configure *Intent*s. This allow them to communicate to the system what needs to be done, instead of worrying how the underlying devices will perform the specific task. Additionally, by supporting dynamic binding of the tasks, we enable development of loosely coupled applications that are independent of the specific task implementation and guaranty stability (e.g., backward compatibility) and enable extensibility of our programming model. Therefore, *Intent*s shield the developers from the complexity of IoT controls and complex data processing, as well as from the diversity of IoT devices and physical environments, addressing research challenges *(RC1& RC2)* .

3) Coupling Intents with IntentScopes: To enable runtime coupling of *Intent*s and *IntentScope*s we need to fully define a validity of *IntentScope*s. First, we examine applicability of an *Intent* on S_{min} (see Section IV-A1). Obviously, this comes down to applying the *Intent* on an entity. Therefore, $apply(I, S_{min}) = apply(I, E) | E \in S_{min}$, where I is an *Intent*. Further, $apply(I, E)$ is true if an *Intent* can be instantiated for the entity and it is determined by the system at runtime, by examining the *mapping and filter* conditions (see Section IV-C3). Therefore, we can apply an *Intent* on S_{min} iff we can apply it on the entity E. Further, because each set S_n can be defined as union of unit sets (S^i_{min}), $S_n = \bigcup_1^n S^i_{min}$, we observe applying an *Intent* can be defined recursively, i.e, $apply(I, S_n) \equiv \bigwedge_1^n apply(I, S^i_{min})$. Therefore, we can apply an *Intent* on an *IntentScope* if we can apply it on its all subsets.

```
Intent eFault = Intent.newMIntent("EnergyFault");
//monitor whole fleet
eFault.setScope(IntentScope.getGlobal());
notify(energyFault,this);//invoke task
//callback function called on event arrival
public void onEvent(Event e){//perform some action}
```

Listing 1: Example usage of MonitorIntent and GlobalScope.

```
IntentScope cs = delimit(IntentScope.getGlobal(),
  Cond.isTrue(eFault)); //eFault defined in Listing1
//Define and configure Intent
Intent eCons = Intent.newCIntent("ReduceEnergy");
eCons.setScope(cs);//set intent scope
eCons.set("speed").value("5");
eCons.set("RPM").value("1100");
send(eCons); //invoke task
```

Listing 2: Example usage of ControlIntent and custom IntentScope.

Now we can show concrete examples of *Intent* and *IntentScope*. Listing 1 depicts a MonitorIntent used to monitor energy consumption and detect potential "EnergyFault" for each vehicle in the fleet. Listing 2 gives an example of *ControlIntent* usage. It shows how to define an *IntentScope* for all vehicles displaying "EnergyFault" and sends *"ReduceEnergy" ControlIntent* to all of them to set the speed limit to 5km/h and to limit engine to 1100rpm.

4) Intent operators: Intent is a passive data structure. Therefore, we need to provide developers with operators to work with the *Intent*s. These operators encapsulate mechanisms to *select, instantiate and execute* tasks, based on the input *Intent*. Consequently, instead of dealing with the individual tasks, a developer is presented with a unified interface to communicate with the runtime systems.

PatRICIA APIs are divided into three categories: *core, system and utility operators*. Due to limited space, we only present the core operators. We define four *core operators*:

1) *send(in ci:ControlIntent, out r:Result)*
2) *notify(in mi:MonitorIntent, in o:CallBackObj)*
3) *poll(in mi:MonitorIntent, out e:Event)*
4) *delimit(in s:IntentScope, in c:Cond, out so:IntentScope)*

The `send` primitive is used to communicate and execute a *ControlIntent*. It accepts the *ControlIntent* as an argument and returns *done* if the *ControlIntent* was executed successfully, *failed* if it is currently impossible to execute the *ControlIntent* and *buffered* if the underlying device is currently busy. When the `send` operator is invoked the container first selects suitable tasks to execute the *ControlIntent* by using intent headers. The task list is further filtered, based on intent attributes, e.g., quality requirements. Here, we use best-effort to find the best matching task implementation. Further, the selected task is configured with *Intent*'s configuration parameters and a payload. Subsequently, the task is instantiated for each entity and finally executed (see Section IV-C4).

The core operators `notify` and `poll` are used to support working with the *MonitorIntents*. The operator `notify` is used by an application to subscribe for events, which are asynchronously delivered to the application. It requires two arguments: a *MonitorIntent*, used to match the appropriate monitor task and a reference to a *callback object*, which gets notified when a new instance of an event becomes available. The `poll` is used to synchronously check the status of the environment, i.e., it will block application's main thread if the required event is currently unavailable. It also requires a *MonitorIntent* as an argument and returns an event (or null) as a result.

The `delimit` operator is API equivalent of $\subseteq_{cond}$, defined in Section IV-A1. It is used to define an *IntentScope* with entities, which satisfy a certain condition. Usually, when an application wants to determine the *IntentScope*, it will start by invoking `delimit` on the *GlobalScope* and further refine it by recursively applying this operator and/or using other operators defined in Section IV-A1.

B. Application structure and lifecycle

Figure 4 depicts a simplified UML diagram showing the structure of our applications. *Application* structure is defined via `onCreate`, `onAppStart` and `onAppExit` methods, which represent hooks used by the runtime to manage applications lifecycle. The `onCreate` method represents application entry point. It provides the application with runtime *Context*,

which provides global information about the application environment. After initialization completion, the container invokes `onAppStart`, which contains application's business logic and from this point on the application is ready to use programing model constructs. Finally, before the application ends, the container invokes `onAppExit` method. This method is used to perform "house keeping", e.g., close any open connections and release any direct references to tasks.

Task (see Figure 4) is defined as an abstract class, which captures general concepts of *MonitorTask*/*ControlTask* and represents the main building block for domain libraries. The *Task* contains metadata, used by the container to bind an *Intent* with the *Task* at runtime. Metadata comprehends *Filter*s, *Mapping*s, *Config* and *Attributes*. The *Filter* specifies a list of *Intents*, supported by the task. The *Mapping* provides information about supported entity types, e.g., vehicle family. The *Config* contains a default configuration of the task. Finally, *Attributes* provide additional information about the task, e.g., provided data quality. Further, *Task*s also have a lifecycle, which is managed by the runtime container. To this end, *Tasks* provide hooks which are used to initialize (`onCreate`), activate (`onStart`) and stop (`onStop`) the task. The `onCreate` contains custom code to initialize a *Task*. The `onStart`, contains processing logic or a sequence of actuation steps. This is the core part of each *Task*, as it contains the specific domain logic, which is executed when an *Intent* invokes the *Task*. We don't define how *Task*s implement the domain logic. For example, monitoring tasks can utilize an event processing framework to implement data processing, but conceptually *Task*s are technology independent. When `onStart` exits, one of the communication methods (`onProcessingDone` or `onActuationDone`) is invoked in order to send response to the application. Finally, when a *Task* instance is no longer needed the container invokes the `onStop` method.

C. Runtime support

The *RuntimeContainer* (see Figure 2) provides an execution environment for the cloud-scale IoT applications. To this end, it implements mechanisms to manage *Task*s and their lifecycle and to dynamically bind *Intent*s with *Task*s. Further, it provides interfaces to register new *Task*s and mediates the communication between the applications and domain libraries.

1) CommunicationInfrastructure: is a communication backbone of our *RuntimeContainer*. It is used to transport *Intent*s and events between applications and the *Task*s. To enable loose coupling between the applications and the domain library tasks it follows pub/sub paradigm. Furthermore, because *Intent*s contain information needed to process and route them, the *CommunicationInfrastructure* must be able to understand *Intent* headers. Therefore, the communication is performed via a partial content-based pub/sub model.

*2) Intent implementation and device-mapping: Intent*s are used to invoke *Task*s. They are processed and executed on the cloud platform, but interact with underlying devices. To perform an *Intent*, PatRICIA instantiates the corresponding *Task* for an entity. It is the responsibility of *Task*s to map the *Intent* to the low-level device services. To this end, they utilize *Devices-services layer* and implement required monitor/control logic.

For example, currently our domain library provides implementations of *Task*s per vehicle family. Each vehicle has a unique Id, which is used to define a messaging topic. All the

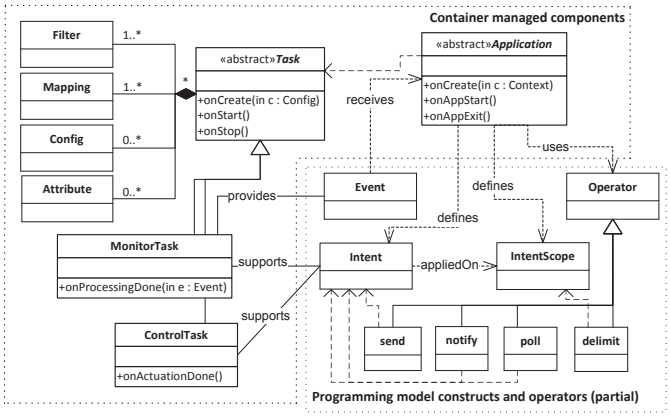

Fig. 4: Simplified UML diagram of application structure.

communication between our library's *Task*s and the vehicle is performed via this topic. The *ServiceMapping* component provides the communication interface and *Device communication layer* provides the required *connector* (a message broker) to communicate with the physical environment. Naturally, as our *Task*s are technology independent, domain libraries can use other mechanisms (*connectors*) to map the *Task*s on the underlying devices.

3) IntentScope and coupling with Intents: ScopeManager implements the *IntentScope* API. It defines a global reference to the *GlobalScope* and implements operators to work with scopes. *GlobalScope* is a singleton, which is initialized with all devices found in the tenant's database. For storing device meta information PatRICIA uses relational databases in the *Persistence layer*. To determine temporary changes in the *GlobalScope*, e.g., devices gone offline, *ScopeManager* communicates with the *DeviceManager*, which implements the Last Will and Testament (LWT) pattern to detect device failure and adapt the *GlobalScope* accordingly.

To support the `delimit` operator *ScopeManager* provides functionality to evaluate the condition expressions and initiates *selection, instantiation and execution* of a *MonitorIntent*, to obtain the value of the specified property (see Section IV-A4). Finally, it provides runtime checks to apply an *Intent* on *IntentScope*. To do this, when an *IntentScope* is added to an *Intent*, the *ScopeManager* resolves the scope and checks if there are suitable *Task*s to support this *Intent* for each entity in the scope by comparing task *filters and mappings*, with intent *headers* (name, Id, entities types, etc.).

4) Intent selection, instantiation and execution: When an application submits a new *Intent*, the *RuntimeContainer* first routes it to the *TaskSelector*, which matches intent *headers* with *Task*'s *filters* and *mappings* to find *Task*s which provide the *Intent* implementation. Afterwards, the *TaskSelector* reads the required (*Intent*) and promised (*Task*) *attributes* and compares them to find the best matching task. Attributes are represented as feature vectors and a multi-dimensional utility function, based on the Hamming distance, is used to perform the matching. Further, *TaskSelector* requests a *Task* instance, by providing its description to the *ScopeManager*, which checks the validity of the coupling and if it is valid, forwards it to the *TaskManager* or otherwise marks the *Intent* as failed. The *TaskManager* instantiates the *Task* via reflections and configures it with intent's *data*. Finally, it triggers the

`onCreate` method on all task instances, to execute any custom initialization code and `onStart` to trigger the execution of the task logic.

V. Implementation & Evaluation

The current prototype of our PatRICIA framework implements the components needed for development and execution of the cloud-scale IoT applications. The same components are framed with solid borderlines in Figure 2. Components outlined with dotted lines will be the subject of our future work.

The PatRICIA framework is implemented in the Java programming language and it is based on WSO2 Stratos[7], which is an open source, full-fledged PaaS solution stack that provides many customizable services, such as identity management, monitoring, logging and multi-tenancy support, necessary for the PatRICIA implementation.

Persistence layer provides a MySQL database to store relational data, e.g., device meta information, and key/value storage (Apache Cassandra [2]) for storing NoSQL sensory data. The PatRICIA chooses how to store the data, depending on its nature. For example, user data, device meta information, etc. is relational and usually requires immediate consistency, thus relational database is used. Contrary, our sensory data is write-intensive and eventual consistency is sufficient most of the times, because the analysis is mostly performed off-line, e.g., by submitting map/reduce jobs.

The *CommunicationInfrastructure* in our *RuntimeContainer* is based on JMS broker (Apache ActiveMQ [1]) and it is used to mediate the communication between the applications and domain library *Task*s. This allows PatRICIA to leverage existing, proven technologies, which provide content-based pub/sub communication between the components, decoupling them and making our programming model higly extensible. The communication is topic-based and the *TaskSelector* uses JMS message selectors to route *Intent*s/events between the applications and the *Task*s. Therefore, *Intent*s are internally modeled as JMS messages. We use message properties to model Intent headers and realize the content-based communication. Currently, at the application level we support XML and attribute/value representation of the *Intents*. To enable *IntentScope* bootstrapping *ScopeManager* defines a global singleton reference to the *GlobalScope*. It is initialized by querying MySQL database and updated when the *DeviceManager* receives LWT message from the JMS broker.

Prototype implementation of the *Domain library* contains a set of *Tasks*, which support *Intent*s used to develop our applications. Library tasks rely on the *Device communication layer* to communicate with the vehicles. It contains its own message broker and *connectors* (one per vehicle family) to mediate the communication. Library tasks communicate with the vehicles over MQTT topic, identified via the vehicle Id. MQTT [5] is a lightweight M2M pub/sub messaging transport, which is a standard for communication with the IoT devices. To implement the *connector*s we used Protocol Buffers [4]. The communication protocol with the vehicles defines a set of vehicle control and status messages, which are marshaled and transported between the vehicles and the applications by the *Device-services layer*. *MonitorTask*s event processing logic is implemented with the Esper [3], a Complex Event Processing framework, i.e., each *MonitorTask* has its own instance of Esper engine and acts as a subscriber to the vehicle topic to receive the low-level status events, e.g., battery voltage. Our *ControlTask*s are implemented as a sequence of low-level messages, which set individual control points in a vehicle and *ControlTask* acts as a publisher of control messages. Therefore, in our deployed scenario vehicle gateways and the cloud behave as both message publisher and subscriber. To this end, each vehicle is equipped with a gateway, which implements a MQTT client and translates the protocol messages to specific points. However, the gateway design is out-of-scope of this paper.

We now demonstrate how PatRICIA can be used to implement our real-world cloud-scale IoT application (see Section II) and use traditional programming model evaluation criteria to evaluate it. The complete source code of the application is shown in Listing 3. Considering *readability and simplicity*, we notice that a developer uses intuitive high-level abstractions (Intent and IntentScope) to write IoT applications, instead of dealing with low-level device-services. Further PatRICIA also provides improvements regarding *reusability* and more *efficient development*. For example, a developer can easily code monitoring of specific fleet vehicles, which fulfill some criteria (lines 7-11). Although, we limit the *expressiveness* to a certain extent, a developer can still easily and intuitively express many common behaviors of could-scale IoT applications (lines 16-19). Finally, *extensibility* of our programming model is guaranteed by deferring the Intent-Task binding to the runtime. This enables adding new concepts (Intents and Tasks) to the model without modifying the existing ones and at the same time guaranties the *backward compatibility* of the applications. Therefore, PatRICIA reduces the *complexity* and enables developers to cope with the *diversity* and the *scale* of the cloud-scale IoT applications.

```
public class ExampleApplication extends Application{
 private Container cont;
 public void onCreate(Context c){
  this.cont = c.getContainerRef();
 }
public void onAppStart(){
 IntentScope s = cont.delimit(IntentScope.getGlobal(),
   Cond.greaterThan("price", "5000"));
 Intent eFault = Intent.newMIntent("EnergyFault");
 eFault.setScope(s);
 cont.notify(eFault, this);//sub. to event
 IntentScope controlS = cont.delimit(s,Cond.isTrue(eFault));
 performIntent(controlS);
}
private void performIntent(IntentScope ts){
 //define Intent and use default configuration
 Intent eCons = Intent.newCIntent("ReduceEnergy");
 eCons.setScope(ts);//set task scope
 cont.send(eCons); //send to all vehicles in ts
}
public void onEvent(Event e){
 performIntent(IntentScope.create(e.getEntityId()));
}
public void onAppExit(){//nothing to do here}
}
```

Listing 3: Example cloud-scale IoT application.

VI. Related work

Most of the current approaches supporting the development of IoT applications deal with device and data integration. Thus, application developers have to deal with the complexity, diversity and scale of IoT applications on cloud platforms.

In [15] the authors focus on abstracting devices as service and enabling two-way communication between enterprise applications and devices via Web Services. Also, approaches uti-

lizing RESTful protocols, CoAP[13] and sMAP[11] exist. For example, [18] focuses on defining a CoAP-based runtime to enable composing IoT services. Most of these approaches focus on abstracting underlying hardware and providing service-based access to a device. Although, they provide some key elements, e.g., service discovery and resource management, they implicitly assume developers have a good understanding of the underlying domain, as raw sensory data streams and low-level device services are directly exposed to them and application development is envisioned by composing the atomic services into admissible control sequences or processing schemes. Compared to these approaches our programming model defines high-level abstractions (*Intent and IntentScope*) enabling development of cloud-scale IoT applications.

Some of the related approaches in ubiquitous computing and context-awareness are [20] and [22]. In [22] authors adopt a definition of task as representation of user's everyday activities. They focus on assisting the users during these activities and managing the resources in smart environments. Although, we share some similarities, regarding task as a generic activity, they don't introduce abstractions with a generic view on scopes, needed to enable development of IoT applications in a scalable manner. Finally, non of the approaches, presented so far are designed for cloud platforms.

In [23], [8], [21], [12] the authors examine applying cloud computing in IoT systems. In [23] the authors focus on sensor-cloud infrastructure, which manages groups of sensors via the cloud, rather than providing sensory data on cloud platform. SenaaS [8] also focuses on providing sensor management and address the issue of combining event streams and services in the cloud. Although, not specifically tailored for cloud, in [9] the authors also examine similar problem and provide a solution in form of event processing containers. OpenIoT framework [21] enables pay-as-you-go sensing as a service and linking sensory data. However, they regard sensor network as a set of networked peripherals and don't focus on a full-fledged cloud platform capable to host third-party IoT applications and services. In [12] the authors focus on implementing virtualization infrastructure to enable sensing and actuating as a service. Although, these approaches increase reusability and fault tolerance of applications, by abstracting and virtualizing individual sensors and actuators and enabling management of these entities in the cloud, for developers the scope of interaction mostly remains a device service.

VII. Conclusion and future work

In this paper, we introduced the PatRICIA framework for programming cloud-scale IoT applications. We presented programming abstractions: *Intent*, *IntentScope*, and a set of runtime mechanisms to support developers in dealing with the complexity and diversity of IoT systems and to enable development of IoT applications in a cloud-scale manner. The set of proposed concepts is not exhaustive, but is sufficient to express many common behaviors of cloud-scale IoT applications. In our programming model, we trade flexibility for a scalable, more intuitive and efficient programming of the cloud-scale IoT applications.

In the future, we will address the remaining requirements elicitated in Section III-A. We plan to extend PatRICIA in several directions: *a)* Enabling the cloud-scale IoT applications to utilize the edge of the infrastructure, by providing suitable code distribution mechanisms; *b)* Giving more control to the applications, by enabling them to define and adapt the execution environments dynamically and on demand. To this end we need to expose the IoT infrastructure resources, e.g, gateways to the cloud-scale applications, develop models to associate costs with these resources, and define clear API to enable the applications to control them; *c)* Enabling policy-based automation of data-quality, security and safety aspects of cloud-scale IoT applications; *d)* Supporting rapid prototyping of cloud-scale IoT applications by utilizing MDD and providing testing environments and the supporting tools.

Acknowledgment

This work is sponsored by Pacific Controls Cloud Computing Lab (PC3L).

References

[1] Apache activemq. http://activemq.apache.org/. 2013.

[2] Apache cassandra. http://cassandra.apache.org/. 2013.

[3] Esper. http://esper.codehaus.org/. 2013.

[4] Google protocol buffers. http://code.google.com/p/protobuf/. 2013.

[5] Mqtt. http://mqtt.org/. 2013.

[6] Pacific controls galaxy. http://pacificcontrols.net/products/. 2013.

[7] Wso2 stratos. http://wso2.com/cloud/stratos/. 2013.

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

[9] S. Appel, S. Frischbier, T. Freudenreich, and A. Buchmann. Eventlets: Components for the integration of event streams with soa. In *SOCA*, 2012.

[10] L. Atzori, A. Iera, and G. Morabito. The internet of things: A survey. *Computer Networks*, 54(15):2787–2805, 2010.

[11] 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.

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

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

[14] D. Gregorczyk, T. Bubhaus, and S. Fischer. A proof of concept for medical device integration using web services. In *SSD*, 2012.

[15] D. Guinard, V. Trifa, S. Karnouskos, P. Spiess, and D. Savio. Interacting with the soa-based internet of things: Discovery, query, selection, and on-demand provisioning of web services. *Services Computing, IEEE Transactions on*, 3(3):223–235, 2010.

[16] M. M. Hassan, B. Song, and E.-N. Huh. A framework of sensor-cloud integration opportunities and challenges. In *ICUIMC*, pages 618–626, 2009.

[17] G. Kortuem, F. Kawsar, D. Fitton, and V. Sundramoorthy. Smart objects as building blocks for the internet of things. *Internet Computing, IEEE*, 14(1):44–51, 2010.

[18] 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.

[19] F. Mattern and C. Floerkemeier. From the internet of computers to the internet of things. In *From active data management to event-based systems and more*, pages 242–259. 2010.

[20] A. Ranganathan, S. Chetan, J. Al-Muhtadi, R. H. Campbell, and M. D. Mickunas. Olympus: A high-level programming model for pervasive computing environments. In *PerCom*, 2005.

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

[22] J. P. Sousa and D. Garlan. Aura: an architectural framework for user mobility in ubiquitous computing environments. 2002.

[23] M. Yuriyama and T. Kushida. Sensor-cloud infrastructure-physical sensor management with virtualized sensors on cloud computing. In *NBiS*, pages 1–8, 2010.