2017 IEEE Symposium on Service-Oriented System Engineering

A Secure Microservice Framework for IoT

Duo Lu and Dijiang Huang
Arizona State University
Tempe, Arizona
{duolu,dijiang.huang}@asu.edu

Andrew Walenstein
Center for High Assurance
Computing Excellence
BlackBerry
awalenstein@blackberry.com

Deep Medhi
University of Missouri-Kansas City
Kansas City, Missouri
dmedhi@umkc.edu

***Abstract*—The Internet of Things (IoT) has connected an incredible diversity of devices in novel ways, which has enabled exciting new services and opportunities. Unfortunately, IoT systems also present several important challenges to developers. This paper proposes a vision for how we may build IoT systems in the future by reconceiving IoT's fundamental unit of construction not as a "thing", but rather as a widely and finely distributed "microservice" already familiar to web service engineering circles. Since IoT systems are quite different from more established uses of microservice architectures, success of the approach depends on adaptations that enable them to met the key challenges that IoT systems present. We argue that a microservice approach to building IoT systems can combine in a mutually enforcing way with patterns for microservices, API gateways, distribution of services, uniform service discovery, containers, and access control. The approach is illustrated using two case studies of IoT systems in personal health management and connected autonomous vehicles. Our hope is that the vision of a microservices approach will help focus research that can fill in current gaps preventing more effective, interoperable, and secure IoT services and solutions in a wide variety of contexts.**

I. Introduction

Recent advancement of computing and communication technology have contributed to the growing ubiquity of connected devices, whose number is predicted by some analysts to expand to 20 billion [1] to 50 billion [2] in the year 2020. The ITU defines the Internet of Things (IoT) as "a global infrastructure for the information society, enabling advanced services by interconnecting (physical and virtual) things based on existing and evolving interoperable information and communication technologies" [3]. Note carefully that, by this definition, what the IoT *is* a global infrastructure of connected things, and what it *enables* is services.

The IoT is historically and technically an extension of the web-centric Internet. As a result, it may not be surprising to find many of the technical and business structures for IoT systems mirror those of traditional web application systems. The structure of technical solution might be a typical web stack, and the operations and management approach adopted may resemble those of other web-based systems. It may feel comfortable, therefore, for IoT system design to follow similar design patterns as in traditional web development. However, unlike traditional web-based design, the differentiating characteristics of IoT system present important challenges to design, development, deployment and operation. Challenges such as limitations of power, bandwidth and connectivity, unreliability of mobile communications, heterogeneity of the technologies and protocols used, difficulty of ensuring common data access controls, and difficulty of change after deployment.

Many of the challenges can be, and have been, addressed at least partially by existing technologies and development approaches thanks to our collective success in leveraging mature approaches for building more traditional Internet-based solutions. The Internet does, and will still, serve as a backbone for the IoT. However, some success at developing IoT systems using existing methods does not necessarily imply that IoT solutions are as easy-to-build, efficient, flexible, and reliable as we should like. We must also remember that there are many well-known problems in many existing IoT systems, including limited interoperability and dangerous security flaws [4], [5].

Most importantly, prior success with traditional approaches in no way guarantees that we shall be as successful in using the same techniques to build the future IoT system we desire. A future in which technologies and development techniques make it far easier to manage the critical challenges, requirements, and forces; where strong security is woven into the very foundations rather than being an appliqué; and where acquiring and connecting multiple devices from different manufacturers is as easy as buying and installing a light bulb is today. To get to the future IoT quickly, we may be aided by some suitable visions of IoT systems design that can inspire future research and development.

This paper aims to paint one possible vision of the future IoT, whose genesis is in the patterns of success for current systems. The starting point is a reconceptualization of IoT itself: instead of focusing on the *things* as the atomic elements in systems, we argue it can be beneficial to follow the *service-oriented* approach and conceive IoT systems as being built out of *services* [6]. Any IoT node can be abstracted as a smart object providing certain services over the network, and the focus of developers can be raised to the level of data and services rather than on the devices and end-to-end communication. Thus, unlike the ITU definition, in this view, the IoT *is* a network of services, and the connected infrastructure is what the services are implemented upon. It is thus also conceptually distinct from other "Internet-oriented" or "Semantic-oriented" conceptions of IoT [7], [8].

One potential concern with a service-oriented approach to IoT is that traditional heavyweight and centralized service models can be at odds with realistic IoT infrastructures. To

978-1-5090-6320-8/17 $31.00 © 2017 IEEE
DOI 10.1109/SOSE.2017.27

address the current and future challenges of IoT systems development, we propose to apply the so-called "microservices" pattern to IoT systems development. In this view, an IoT system is constructed of fine-grained and self-contained microservices which are independently developed and deployed. However it is not enough to start only with conviction to utilizing microservices patterns, so we combine the microservices pattern with additional complementary patterns that collectively fill in the vision of how IoT systems can be built generally. Specifically, we show how patterns for API gateways, service distribution, service discovery, containment, and access control can be combined to address problems in IoT systems development. The promise and generality of the approach are then illustrated by showing it in action in two different IoT scenarios.

II. From Things to Distributed Microservices

The way in which cloud computing and web service applications are engineered—the techniques and technologies used—have changed substantially since the invention of the Web less than three decades ago, and recently one approach known as "microservice architecture" [9], [10] has gained some notable adherents, including Netflix, Linkedin, and Amazon [11]–[13].

A microservice is a minimal functional software module which is independently developed and deployed. A microservice architecture is a composition of microservices that are deployed and connected through composition mechanisms. Desirable characteristics of microservice-based systems include: decomposition of larger services into small, focused, self-contained services, loose coupling, and clear execution context boundaries. Due to these characteristics, microservices can often be deployed inside portable lightweight execution environments called *containers* [14]. Even the services of cloud infrastructure itself can be developed and deployed in this way [15]. Much of the excitement from switching to a microservices approach comes from their promise for improvements to system and process qualities: improved alignment of system structures to teams and business, easier integration of software components using heterogeneous technologies, easier system evolution, simplified continuous delivery, and improved resilience, and scalability [16], [17].

Assuming that a microservices revolution in cloud-based web services is or will bring about important changes that improve the systems and their development, what would such a revolution look like in IoT, and how should successes from the cloud apply to IoT construed broadly? This is an open question and, we think, a very interesting one. In many ways, there are already numerous connections between IoT to service-oriented systems development approaches and web-based systems.

One obvious connection is the cloud service components of IoT systems. Technologically, IoT systems may build their cloud service components using technology stacks inherited from the traditional Internet and web development. Given this common base, the microservices pattern should be relatively straightforward to apply to any centralized portions of existing IoT systems. And, indeed, Krylovskiy *et. al* [18] drew out parallels to how the microservices pattern could apply to their IoT cloud services, and Butzin *et. al* [19] observed how cloud microservice principles of self-containment apply well to IoT more generally, and how successful microservices pattern might be fruitfully applied to IoT fault management, service coordination, and service version management. Closer to the endpoints, exposing IoT device functions through embedded web service interfaces allows real-world objects to be part of the World Wide Web; this web application layer over devices has been dubbed the "Web of Things (WoT)" [20], [21]. A WoT approach promises to simplify development, and common web-based protocols like REST may become the de facto standard to access many commercial IoT devices, such as Philips Hue [22]. So there are many touchpoints between IoT systems and web engineering approaches in the cloud, as they exist now.

However, it is important to understand that IoT systems, construed broadly enough, have important differences that must be accounted for. It is therefore not a straightforward matter to take solution patterns that work well in cloud-centric or web-centric contexts and apply them just as successfully in more widely to overall IoT solutions.

Consider one future IoT scenario of a community responding to a crisis like a major earthquake occurring within a large city. Early warning of the event is provided by cloud-based mining of data from sensors embedded in roadways, allowing the pre-emptive response of routing autonomous vehicles away from harm. After the quake, an Emergency Response Team (ERT) at the scene reacts according to an alert signaled by the app on a victim's smartphone, which is responding to indications of the owner's possible loss of consciousness and blood from a connected wrist-worn health monitor. The responder's head-worn augmented reality glasses connect to the smart home to permit him entry, and guides him to the victim's location. There he applies life-saving procedures to the victim following instructions from a remote doctor, who is able to read the victim's health information in real time and watch through the responder's body-worn camera. The National Guard, called to the scene, set up the local triage unit that houses the team of remote doctors, provide an emergency cellular service relay to help fill in coverage for the local outage, and clears a path for evacuation by autonomous vehicles guided by the relayed locations of priority emergencies.

Without trying to exhaustively enumerate the many important issues for IoT situations like the above, we will point out several essential concerns not found in typical cloud-focused web-based development: ones that need to be solved well to enable a future IoT like the above. First, the responder's government-issued wearable devices must be able to interact with the various services of the home, and the health monitor must connect to the remote doctor through the smartphone on demand. Neither of these specific interactions should need to be programmed into the system beforehand, and it is perhaps doubtful that such pre-arrangement would be willingly met by the public or by system manufacturers. Second, the victim's

TABLE I
KEY PATTERNS IN THE VISION OF MICROSERVICES-FOCUSED IOT SOLUTIONS

Name	Type	Solution of	Qualities
Microservices	Structural	Decomposition	Team structure alignment (Conway's law); independent development and testing; enabling microservices marketplaces
API Gateway	Structural	Aggregation and distribution	Service API aggregation; event distribution; policy enforcement; network bridging
Distribution	Structural	Locality decomposition	Performance; resilience; scaling
Service discovery	Action	System coupling	Dynamic reconfiguration; extensibility
Containers	Structural	Deployment	Deployment simplicity; security; scaling
Access Control	Structural	Data sharing	Interoperability; scaling; flexibility; secure

health information is highly sensitive but, for the scenario to work, it must be made available to the remote doctor and yet only the location and urgency should be revealed to the National Guard. Without flexible and dynamic access control, and without the devices and networks being able to meet the high security bar of modern health information data protection standards, such a scenario is a complete non-starter. Finally, in the event of an earthquake, key communications systems may be knocked out or overloaded, and the entire communications infrastructure must both be resilient to drops and flexible for routing. SOA, web programming, and microservice architecture also need to manage interoperability of heterogeneous components and protocols, service discovery, security and privacy, and communications disruptions, no matter how the independent, mobile, embedded, and distributed nature of IoT systems shift the balance of concerns to a degree that the approaches taken must be modified.

III. MICROSERVICES PATTERNS FOR IOT

Microservices patterns, like all patterns, are meant not to define a single solution, API, standard, or framework, but rather to identify common patterns found in systems that successfully manage important forces [9]. It is important to keep in mind, therefore, that that a given IoT system might simultaneously involve multiple technical implementations of key elements like containers and event notification protocols—some of these legacy and difficult to change—but these particulars should not stop us from recognizing the pattern in use.

We wish to work towards a vision future in which we are able to efficiently develop IoT systems that are extremely interoparable, flexible, and secure. The structure of the vision consists of a high-level, abstract model of general microservices-based IoT systems, together with patterns of solutions that one might expect to see employed in the design and deployment of any given IoT system following the vision. We argued in Section II that some of the key requirements and forces within IoT systems are exhibited differently than within typical cloud-only, web-centric microservices solutions. Therfore, one should reasonably expect that the sorts of patterns that one might find in use in successful IoT systems will contain common elements to cloud systems, but nevertheless can be instantiated and combined in different ways.

Though doubtlessly many patterns will be used in IoT systems in the future, we focus our vision on the combination of six patterns that we expect to combine constructively to build solutions well. These are listed in Table I which, for each pattern, provides a name to the type of the pattern, what kinds of IoT problems it may be applied to, and some of the desirable system qualities that may be enabled more easily through use of the pattern. A visualization of the overall abstract model of IoT systems using microservices patterns is shown in Figure 1

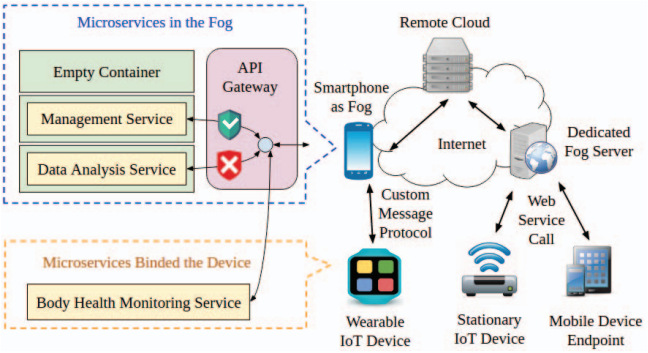

Fig. 1. Abstract model for IoT systems structured using microservices

A. Microservices

We take "microservices"-based IoT systems to be composed of one or more individual self-contained and independent deployable software components i.e., *microservices* interacting with each other by messages abstracted as method calls (e.g. web service call, or message RPC), or through publishing-subscribing events.

For example, consider a hypothetical body weight management IoT system that provides a service to help the user manage her body weight. The system consists of 3 components: a wristband with sensors that measures how many calories the user is burning, an app on her smartphone that asks her to input what she eats and calculates the calorie intake, and a cloud-based software component that offers recommendations for meals and sport activities. Following a microservices pattern, the overall functionality could be separated into a collection of independent microservices, which communicate with each other through well-defined message interface. For example, the wristband may connect to the smartphone through Bluetooth and send body activity intensity information to the app, and the app can connect to the recommendation service through

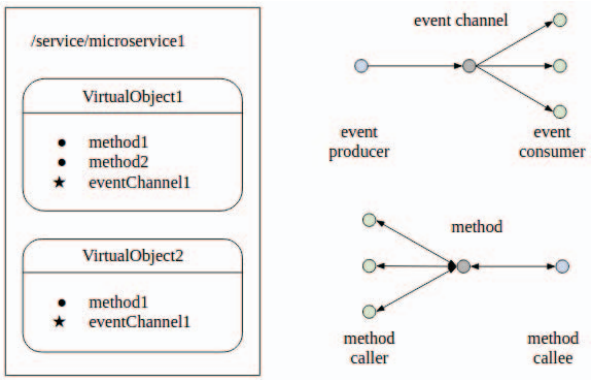

Fig. 2. Virtual object, event channel, and method

platform-level inter-app messaging, or perhaps through a REST API in the case of a recommender service in the cloud. It is completely possible these three microservices are developed by different teams, or even different companies, as long as their interfaces are well maintained.

From the service consumer's point of view, a microservice is nothing but a collection of APIs. To make it easier to discuss the organization of these APIs, we propose an abstract model of the internal structure of a microservice, which can be mapped onto various protocols such as MQTT, CoAP, AMQP, or REST API. In our model, a microservice contains one or more virtual objects, which we take to be akin to objects in object oriented programming (OOP) languages. A virtual object can have two types of service APIs: methods, or event channels. Methods resemble the methods in OOP languages or services in REST APIs: they provide a remote procedure call (RPC). Event channels implements a publisher–subscriber model such that anyone that subscribes to the event channel will be notified when an event is published by an event producer.

Sensors and devices with sensing capability are examples of what might typically be an event producer. The events they publish might usually be intermittent, and their traffic is normally a push style. Actuators and services in the cloud are examples of event consumers and method providers (callee). They are typically continuously online, and their traffic style is typically that of request-response. It may be that one event or method call might generate a cascade of additional events or method calls. A method that does not return anything can be used as an event consumer as long as the method and the event channel are compatible in data format as well as semantics. Virtual objects, event channels and methods can also be dynamically created by a microservice. Figure 2 shows the relationship of virtual object, method, event channel, and the binding with service endpoints.

A common claim of microservice patterns is that they can ease development in several ways [9]. For example, for the developers, it is easy to create simulators and sandbox environments for a real world IoT systems since devices can be abstracted to microservices with well-specified message interface. Additionally, sensor events can be logged and replayed in the sandbox environment to help debugging, testing, and troubleshooting of the microservices under development. Since a self-contained microservice usually does not have external dependencies, the development of a large IoT system can be broken into many small, semi-independent and focused teams.

For the system administrator, each microservice can be independently deployed, and if the method and event channel interfaces are well-defined and common then the connections may be as simple as plugging in light bulbs into standard recepticles. These facts of deployment mean that new components can be incrementally introduced to the system and deployed side-by-side with existing ones. If some microservices are broken or obsolete, they can be simply unplugged and switched out. We envision that there will be a "supermarket" of microservices as well as devices for IoT systems, where the "goods" are containerized ready-to-deploy microservices, such that a system administrator merely needs to "shop" online, like shopping the App Store for her smartphone. For the operator, it is possible to create an IoT operations center that collects the status of all microservices and presents the information on a dashboard. The IoT operations center functionality itself can be implemented as a microservice with required privileges so that it can access the entire system service registry, call any methods, and listen to any events. In this way, the operator is similar to a commander in the NASA mission control center, sitting in her office and manipulating smart devices and microservices as required.

B. API Gateway

An API gateway is an interface that relays calls or requests between two or more microservices. API gateways can aggregate the APIs for multiple microservices into one client interface, and can also distribute or route the calls or requests from one entry point to multiple target microservices. Rather than allowing direct communication among microservices, therefore, an API gateway can be placed in-between them, which can have the effect of adjusting interconnectivity through the aggregation and distribution function of the gateway. Microservice connection is then through registration; when a microservice registers itself to an API gateway, it creates one or more endpoints with unique identity, which can be further bound to an event channel or method through static configuration or dynamic negotiation with the API gateway. Besides the microservices, any user or machine client can also be required to present its identity as a name of identifier to interact with the API gateway to access its associated microservices. The identity management and authentication service might leverage existing systems that managing accounts, such as single-sign-on (SSO) or OAuth [23].

From the API gateway's point of view, each endpoint is uniquely identifiable, where the identity may be represented as a unique name or just an identifier. Such endpoint name or identifier only exist in the model, and the API gateway implementation needs to map it into actual protocol, such as an URL in REST API or CoAP, a queue or exchange in

AMQP, or a topic in MQTT. In a cloud-supported body weight management IoT system as described in the example above, the weight management plan virtual object can be mapped to REST resource URL on the cloud server, which runs the weight management and recommendation microservices, and the the method as well as parameter format on this virtual object can be mapped to the HTTP methods. In typical uses of this approach, the method name is encoded in the JSON payload to augment the limited types of HTTP methods. Such mappings impose additional constraints on our model, such as requiring that consecutive method calls in a session must be "non-sticky".

For addressing across heterogenous IoT devices and networks, virtual objects, methods, and event channels have site-scope unique hierarchical names like URNs. In some cases it may be feasible to create unique IDs using URN style approaches that encodes the device unique identifier (e.g., `snac.asu.edu/device/edb15825-92e2-e7b1-7c97-ddd102258a70` which uses the Bluetooth MAC address for uniqueness). Taking our body weight management IoT system as an example, imagine the API gateway is in the cloud with domain name `snac.asu.edu`; then there can be a virtual object `snac.asu.edu/WeightManagementPlan` with `createPlan` method. After a plan with user profile and weight management goal is created, a virtual object `snac.asu.edu/WeightManagementPlan/plan1` is created with `checkGoal` and `getRecommendation` methods.

Our model is compatible with many message queue protocols as well as REST or CoAP API used in real world IoT systems. For example, the role of our API gateway is similar as an load balancing proxy for web services, or a message broker of MQTT protocol. The virtual object can be mapped to the resource using a RESTful URL, and the method can be mapped to the combination of HTTP methods on that URL and JSON payload. This allows us to apply a microservices pattern on a broader scope than just web based applications. However, some protocols may not be able to provide both method type and event channel type of service APIs. For example, MQTT can only provide publish-subscribe interface and, while CoAP can provide service call natively, publish-subscribe through CoAP will probably be difficult and unnatural to implement. In such scenarios, it is important for the API gateway to support multiple protocols to connect those "IoT islands" to the rest of the system.

In many cases microservices on the devices do not possess publicly reachable IP address, a trait which prevents them being accessed over the Internet outside the domain intranet. Multi-protocol API gateways can bridge the gap. In our model, we call the subsystem made of an API gateway and its associated microservices an IoT site (like website in traditional Internet). The API gateway works as a hub that translating messages from one protocol to another, making microservices in different domain accessible to each other. Meanwhile, the gateway also accept API calls from user or machine clients outside the site and translates them to corresponding messages to intranet API calls. Sometimes an API gateway is able to deliver messages or initiate service call to another API gateway to fulfill complex services spanning over multiple sites.

The existence of API gateway as well as the binding of endpoint to method or event channel makes a natural place to enforce policies. We borrowed the approach of policy enforcement from software defined network (SDN) and, in our model, there are three types of policies: rules to allow or deny the creation or destruction of a certain named service APIs, rules to allow or deny a binding of service API to an endpoint, and constraints on the binding. The first two resemble file system permissions in operating system, and the last one resembles flow policies in SDN, in which the flow is a binding. We explain three examples of policy based on the usage, which are name resolution policy and QoS policy (in this subsection), and access control policy (in a later subsection).

A name resolution policy is a static binding between a service API and an endpoint. This type of policy is useful for integrating microservices of different versions, or for load balancing. Consider again our body weight management IoT system: the developers may construct a new version of the recommendation microservice and deploy it to the cloud. In response, the system administrator needs only change a few policies to bind the methods on the API gateway to the endpoints of new version of the microservice. The replacement is easy if the new version is backward-compatible in syntax and semantics. If it is incompatible, the system administrator can instead create new virtual objects and methods, bind them to the microservice of the new version, and run both versions in parallel until all clients that use the old microservice shift to the new version. For load balancing, the administrator has the option of running multiple instances of the same microservice and to bind them to the same set of virtual objects and methods. Similarly, balance can also be facilitiated by binding multiple event consumers to the same event channel so that events are dispatched among them instead of duplicated to all of them. Name resolution policies and API gateways can combine to decouple the event producer from the consumer (or caller from callee). In an IoT system, microservices might not have fixed location on the network. For example, services on mobile IoT devices might need to attach to different API gateways at different times. Name resolution policies might be used also to redirect requests to other sites if they can work together with service registry mechanism.

A QoS policy is primarily concerned with traffic restriction on a binding (if we regard a binding as a traffic flow). Restrictions could include limiting the number of messages per second or traffic amount in bytes per second. Such policies can help protect the microservice from DoS attacks, or prevent launching DoS from some device or external client. Additionally it provides more flexibility needed to fulfill service level agreements and accounting processes. As with access control policies, QoS policies can also be dynamically generated by a policy controller. Another usage of a QoS policy is to implement so-called "circuit breaker"s [24] in which, if the running status of the microservice shows it is exhausting its

capability, a QoS policy can be dynamically generated to restrict further requests to prevent cascading failures.

C. Distribution

Traditionally, microservices are all located in the cloud. In IoT systems, we expect that microservices can additionally be associated with a device, edge cloud or gateway component, or deployed in the larger Internet cloud. Because of the distribution, API gatways also may be distributed on the edge, including within a personal area network or on a smartphone. These gateways may have various forms, which can be an independent physical machine, or a software component deployed on a router or server.

One of the advantages of distributing microservices and gateways among the cloud is better performance, due to the locality and the data stream management. In many IoT scenarios, the cost of carrying all the device generated data to the centralized remote cloud for processing and decision making outweigh the benefit. For example, some IoT applications may be latency sensitive, or the sheer amount of data generated by the device exceeds available network bandwidth, which requires us to run microservices close to the devices. An API gateway within a smartphone can aggregate data from a connected wearable that provides event data at high data rates, but it might relay only summarized, aggregated, or otherwise "rolled-up" data back to a cloud microservice. Edge cloud and fog computing extends the traditional cloud computing paradigm to the rim of the Internet of Things, which enables new types of applications [25], [26]. In our architecture, edge cloud or fog is a virtualized platform serving as an on-demand execution environment of microservices close to the devices or the data source, which is different from running the microservices on the device itself. Providing an edge cloud or fog platform can also provide better architectural flexibility. For example, microservices can be off-loaded or migrated to the edge cloud or fog from devices to meet changing requirements of service performance.

D. Service Description and Discovery

In an IoT system, integrating microservices requires description of the service APIs in both syntactic and semantic level. Such description will also be served as an important way to search existing services and compose new services by the machine. We propose to use ontology for service description and discovery. Our ontology is made of three parts: the IoT system domain ontology, the service profile, and ontology of microservices in the IoT system, shown in Figure 3. The first part is the domain ontology, which contains the domain knowledge of entities and relations. For example, in a smart building IoT system domain ontology may describe the structure of the building, real world entities such gate, room, and the IoT devices attached to the entity. This part of ontology should follow IoT-Lite ontology [27], or SSN ontology [28], or other domain standard. The second part is service profile, which contains the description of service interface of a domain, including the category, process, parameter, result, condition

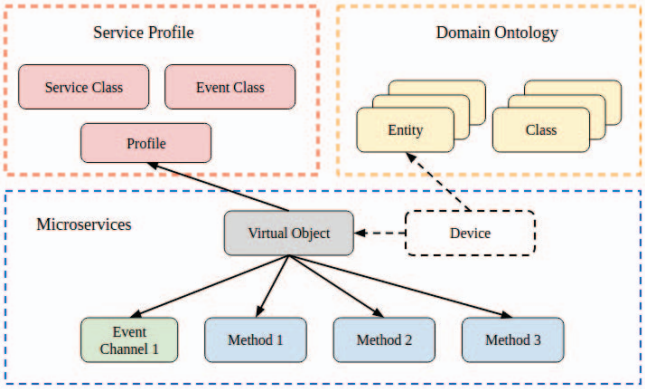

Fig. 3. Ontology for service description and discovery

of a method or an event channel. Additionally service level agreement (SLA) should also be specified as part of the profile. This part of ontology should partially follow WSMO [29], or OWL-S [30], and the service grounding should support commonly used protocols in IoT system other than only web service. The third part is the ontology describing the microservice structure in the IoT system, including the virtual objects, methods, event channels, and their enclosing microservice. This part may also includes deployment related entities, such where is the microservice deployed, on a device or in the cloud, in the fog / edge cloud or remote cloud. The three parts of ontology can be managed by an individual ontology database microservice associated with the API gateway. When new services are registered, corresponding ontology segments are added to this ontology database. Other microservice may query the ontology or run reasoning procedure through this ontology management microservice.

Besides ontology, each microservice should provide a human readable service manifest document for developers. We recommend that each microservice should provide a virtual object of the microservice's name, and the virtual object should have a method called "introspect", which returns a manifest detailing all the virtual objects and service interfaces (with the mapping to various protocols) of the microservice. Such manifest may just contain a plaintext document for human developers, or an annotated XML / HTML to be rendered on browser, or even an annotated document in OWL / RDF to be readable for both human developers and machine programs. In addition, since microservices can be dynamically added to an IoT system, ontology is subject to change. It is helpful to provide a global service registry microservice (possibly co-located with the API gateway) which manages the ontology and manifest to allow service registration and discovery among multiple sites. Each individual microservice can talk to the registry in order to publish its structures and interfaces so that other microservices can make SPARQL query to locate it.

E. Containers

Containers are used frequently in cloud computing for ease of deploy and to provide secure containment between different

tenants. We treat containers generally here as an infrastructure that provides a local context for execution that can be instantiated multiply and separately, and hold something that can be recognized as a microservice. By this we mean to include not only OS-level container solutions like Docker [14], but also typical application-level or platform-level "sandbox" containers like Chromium containers [31] or Android for Work [32].

Containers in IoT systems can serve functions similar in cloud computing in terms of ease of deployment, replication, and scaling. They can permit convenient and efficient deployment of a microservice by packing the microservice along with its dependencies into a single image deployed to the container. From the system administrator's point of view, the deployed container with the microservice is a black box with a few interfaces that can be easily "plugged into" the API gateway. The containerized microservice maintains its own states and stores its data, so that some of the complexity is encapsulated inside the black box. In addition, virtualized execution environments provides isolation and control of the microservices, such as by forcing all traffic to follow the policies enforced on the API gateway. This can ease troubleshooting, online A/B tests, and roll back, and it can simplify overall management. However, when the containers are used on smartphones or other smart devices, the security and isolation of each microservice is the more compelling need.

The use of containers for security, as a pattern, can succeed because of the way that essential security functions of isolation and encryption can be performed comprehensively according to enforced policy rather than requiring the microservice developers to implement them — and relying on them to unfailingly and correctly implement them too. For example, some containers can force all traffic out of the container to be passed through a VPN all data stored locally be encrypted with a container-specific key stored securely in trusted memory. As such, they can establish a level of security for the data-in-transit and data-at-rest. Wrapping microservices and API gateways in containers with known, tested levels of security can greatly simply the problem of certifying or verifying some facts of the system's overall security. As the security of containers improve, the security of the IoT solutions can improve without changing the microservices, and the container solutions are patched to close holes from newly-discovered vulnerabilities, every part of the IoT system can rise with the tide without any changes to the code whatsoever. While this containment may not help stop device manufacturers from shipping devices with other weaknesses (e.g., default passwords in the manual), using containers as a pattern for pervasive security meshes well with a comprehensive microservices approach to service decomposition.

F. Access Control

Access control policy determines whether a dynamic binding between a service API and an endpoint is allowed, which effectively controls whether an endpoint can publish / subscribe an event channel or whether it can make or provide a method call. The policy itself can be dynamically generated, and implementation of such policy usually relies on an additional identity management and authentication microservice as well as a policy controller microservice. For example, in our body weight management system, we may need a microservice in the cloud to authenticate user by password. Before he / she login, any method call on other microservice is denied, while after successful login, a bunch of policies are generated to allow the client to interact with the microservices in the system. Optionally, these policies do not need to be generated at login, but at the time when the client first use a method or an event channel, where the API gateway will temporarily hold it and query the policy controller to dynamically generate a policy on the fly.

IoT device generated data access control is another important service, such as medical data, vehicle sensing data, etc. IoT data is usually generated by physical sensors and stored in a storage node coverage a certain IoT sensing area, e.g., a home, a building, or entire city. The challenges are how to enable data access control when data owners and users are belong to different administrative domain. With the policy controller microservice, more sophisticated access control model can be implemented. For example, access control rules can specify a few condition using the attributes from the ontology, and the controller, which control both service gateway APIs and storage servers, may reason over the rules and ontology to check attributes for both data objects and human subjects to decide whether an action is allowed or denied. In this work, we propose an attribute-based encryption (ABE) [33] approach as one of important building block to build the ABAC model for IoT. With the capability of enforcing access control on microservices, the IoT system will have better security and privacy protections for the data owner.

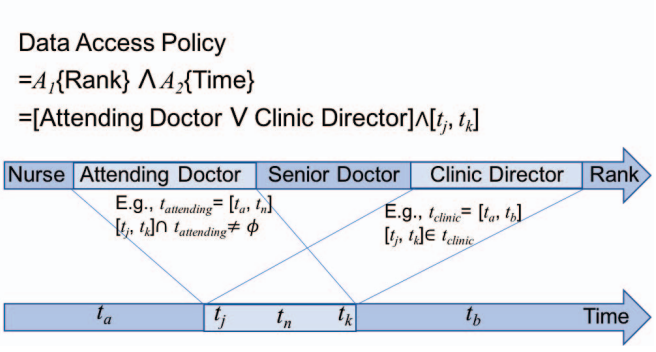

Fig. 4. Two-dimensional attribute ranges in the telemedicine example.

An example is highlighted in Figure 4 for attribute-based medical data access control. The ABE schemes [33] are designed to handle descriptive attributes, however in some cases, the access control policy takes the form of range constraints for the corresponding attributes. An application example is secure data sharing in telemedicine (as shown in Figure 4), where a patient periodically uploads his/her health records to the medical information service delivered by a cloud provider, and healthcare professionals in the

designated clinic can monitor his/her health status based on his/her health records. This patient has a policy that only healthcare professionals with positions higher than Nurse can access his/her health information and that too, only between business hours (time t_j and t_k). Thus, the data access can be specified by a policy $\mathcal{P} = [A_1 \wedge A_2]$, where $A_1 = rank$ and $A_2 = time$ are two attributes, and each attribute has a certain range, where $Rank = \{$Nurse, Attending Doctor, Senior Doctor, Clinic Director$\}$ and $Time = \{t_x | x \in \mathbb{Z}\}$. Correspondingly, a Senior Doctor who has a higher rank can access the data if he or she has been authorized to the time interval that is contained in $[t_j, t_k]$. In our recent work [34], we have developed new comparable ABE solution to realize the presented use case in Figure 4. Based on it, new approaches that can support multiple attribute domains are required, where each domain runs an attribute authority to manage secret keys for a distinct set of attributes in this domain. In addition to the aforementioned comparable attributes and multi-domain properties, edge cloud computing approaches need to investigate to allow offloading intensive encryption and decryption operations to powerful computation nodes close to IoT devices, which significantly reduces the computing overhead for light-weight mobile devices or sensors.

IV. Case Study 1: Personal Health Management

We built a prototype demonstration weight management IoT system using simulated wearable sensors and cloud resources. Its core function is to help the user meet weight management goals by providing exercise recommendations based on a fusion of environmental and personal context information. The excerise recommendations and schedule are customized to the user's weight management goals and contextual constraints. For example, the schedule of a recommended exercise would take into consideration caloric intake, user activity, the available time slots, eating or sleep schedule, and suitability of various exercises to expected weather. Our prototype simulates the local sensors, and the exercise recommendation microservice exposes a few REST API for custom user interaction, and provides a java GUI program (see Figure 5) for demoing, system inspection, manipulation of the stored data, and debugging.

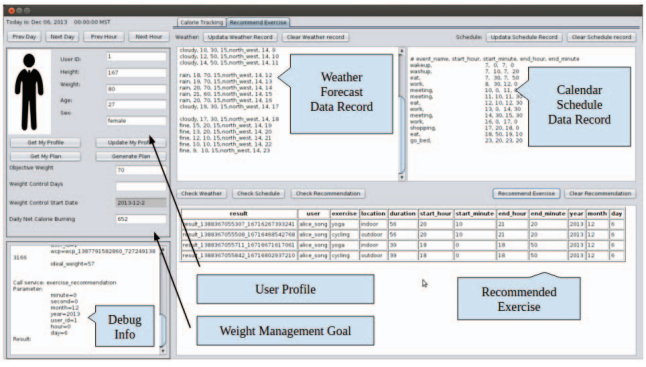

Fig. 5. Body weight management system prototype UI

Microservices architecture. The system architecture, pictured in Figure 6, illustrates the microservices approach by structuring the service components into microservices distributed amongst the cloud, an IoT device, and a smartphone. In the cloud, separate weather forecast and calendar reporting microservices connect to remote resources, like the user's Google calendar, translates the data into the required formats, and provide updates to a separate exercise recommender microservice. On the smartphone a microservice reports calorie intake events collected from the user by direct input. A smart wristband contains a microservice that reports calorie consumption and, since it only connects to the smartphone, the smartphone also acts as a kind of personal edge cloud node with a second microservice that collects calorie consumption information from the wearable and relays them to the cloud. Each of these microservices are viewed as sensors bonded to their own event channel.

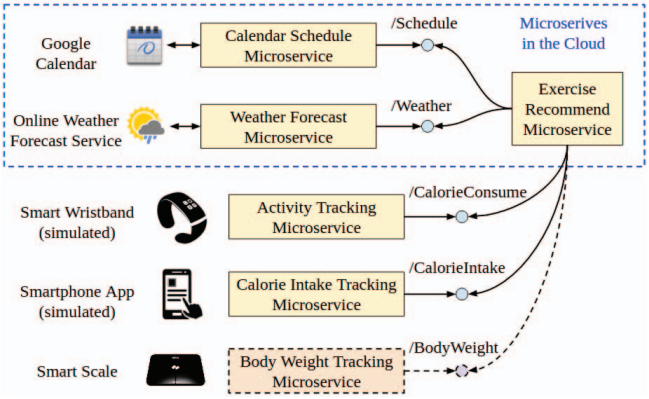

Fig. 6. Microservice architecture of body weight management system

API gateway. The system illustrates the use of the API gateway pattern by combining the weather and calendar microservice for the exercise recommendation microservice, and by aggregating the microservices originating from outside of the cloud. A HTTP API gateway insulates the recommender microservices from the specifics of the individual protocols originally from four different event sources. The gateway approach also makes it possibly to add local management of the event streams, such as by moving the API gateway to an edge cloud node, or the smartphone, to aggregate monitor event data that occurs at a higher frequency than needs to be sent to the cloud. In the case that we wish to include an optional smart body weight scale to further improve the quality of the recommendations (e.g., Withings Body [35]), the change impact is localized and since the adapter for the scale's protocols can be added into the API gateway.

Containers. Our prototype does not utilize any special containers, one could imagine usefully utilizing multiple types of containers in multiple places. In the cloud, a lightweight container such as Docker [14] migh help with deployment and scaling in the case of hosting cloud recommendation services for, say, millions of users. And on the smartphone several microservices need security and privacy isolation from other

applications, and appropriate containers can help ensure the isolation.

V. Case Study 2: Autonomous Vehicles

We built a prototype demonstration system to explore the use of microservice architectures in the context of exploring so-called "platooning" algorithms for autonomous vehicles. Longitudinal platooning means that several vehicles form a physically-local chain that maintain close proximity while traveling down a road [36]. The system employed six VC-trucks within our VC-mobile test bed [37]. Each mobile robot vehicle is an independent IoT subsystem both offering event services to other vehicles and connecting to others'. The event services include both data and control APIs. For example, the vehicles provide an event stream `reportState()` to provide vehicle state, and may provide `join()` and `leave()` so that a lead vehicle can check safety constraints to decide whether to approve vehicles to join.

Platooning requires continuous inter-communication of vehicle motion and location data, including speed and acceleration, and frequent adjustment of motion as needed to maintain a tight platoon. Motion data comes from the cruise control, and location data are collected from ultrasonic range finders. The data are exchanged through vehicle-to-vehicle (V2V) communication over Wifi. Our platooning algorithm uses a platoon leader vehicle, which broadcasts its motion data to the entire cluster, while each other leader vehicle in the chain interchanges motion information with its neighbours behind. Platooning calculation uses a PID controller to calculate desired speeds based on predecessor states, as well as the separation distance to the predecessor, and also uses the leader's states to make adjustements in order to maintain string stability of the whole platoon, and thereby prevent speed oscillation. The platooning gathers sensor readings and also adjusts target driving speed 10 times per second. At a lower level, the vehicle motion controller drives the cruise control to adjust or maintain the vehicle's speed. In our demo, we operate 6 VC-trucks to form a platoon. Starting from zero speed, when the leader starts to move, the platoon successfully achieves a stable state of constant velocity (0.2 m/s) and a preset separation distance (0.7m), as illustrated in Figure 7.

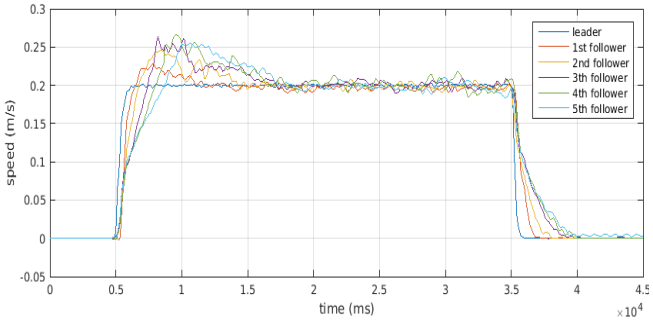

Fig. 7. Stabilization pattern for vehicle speed in the demo

Microservices architecture. We built the system according to a distributed microservices architecture, as illustrated in Figure 8. The services offered by the vehicles are decomposed into microservices that collect coherent but loosely coupled parts of the service in the form of separate platoon and status reporting microservices. In real-world contexts, future automated vehicles might have a variety of other microservices offered by third parties in addition to the these platooning services offered by the manufacturer, and each can be deployed in a different microservice.

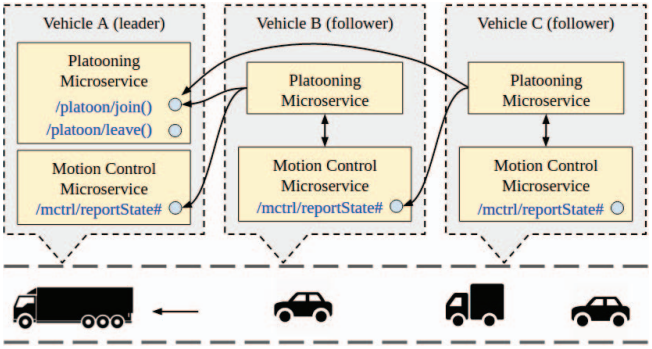

Fig. 8. Communication among Microservices for Vehicle Platooning

API gateways. Each vehicle has its own API gateway that aggregates the services makes it simpler to deploy the microservices differently or manage independent protocol evolution for the microservices. In addition, the API gateway can provide a flexible and fine-grained location for configuring and enforcing access control policies. Since the platooning microservice can control critical safety features like vehicle speed, and since the appropriateness of access may depend upon context not "owned" by the platooning microservice itself, the gateway can be the most suitable location to implement the enforcement.

Containers. Our robot vehicles are outfitted with computers running a hypervizor (KVM [38]), which lets us deploy microservices flexibly. It also provides a security boundary for isolation of the microservices, which is important perhaps particularly for the speed-controlling platooning microservice.

VI. Discussions and Conclusion

In this paper we present a vision of applying microservice architecture in IoT system. We provide a general model of the internal structure of a microservice, and two cases studies under this architecture.

However, microservices are no free lunch or silver bullet. Successful adoption of microservices requires in-depth understanding of the application domain to determine service boundary, which can be very advanced techniques or even arts in complicated real world application scenarios. Moreover, with the evolution of the system, new microservices tends to be layered above the old one and software-only microservices usually rely on the devices, which may gradually lead to more and more service coupling, or even dependency hell. From an

architect's point of view, in many cases the complexity is still there, but just moved from from intra-service domain to inter-service domain. Furthermore, fine grained services tend to suffer from communication overhead more, which diminishes the architectural benefit.

Taking these limitations into consideration, we still believe microservice architecture is promising in the design and construction of an IoT system, mainly due to the inherent characteristics of the IoT system. We summarize the characteristics of IoT system, related design challenges and microservice architecture approaches in TableI. Still, to successfully apply microservices in an IoT system requires many trade off and hard decisions to make. There are many open questions we have not addressed in this paper. For example, whether the API gateway creates a single point of failure as well as a scalability bottleneck? Should we create a standard service level protocol like HTTP to implement the model or should we try to use one model to accommodate different protocols? Can we just use REST API everywhere? Who is in charge of allocating device and service identity, or in another way, giving them names and addresses? How to cooperate among multiple sites? Can we invent a service level "bus" to deliver events and other service requests among multiple API gateways? Then how can we route those messages to the named destinations? In many cases there no good answers, which calls for further research efforts.

ACKNOWLEDGMENT

The authors would like to thank our colleague Zhichao Li and Chao Zhang for their help in the programming of the body weight management and vehicle platooning case study. The research is sponsored by Naval Research Lab (NRL) Grant N00173-15-G017 and National Science Foundation SaTC Grants 1528099 and 1526299.

REFERENCES

[1] "Gartner says 6.4 billion connected "things" will be in use in 2016, up 30 percent from 2015," http://www.gartner.com/newsroom/id/3165317, accessed: 2017-1-30.

[2] D. Evans, "The internet of things how the next evolution of the internet is changing everything (april 2011)," *White Paper by Cisco Internet Business Solutions Group (IBSG)*, 2012.

[3] Recommendation ITU-T Y.2060., "Overview of the internet of things," https://www.itu.int/rec/T-REC-Y.2060-201206-I, 2012, accessed: 2017-1-30.

[4] D. Miorandi, S. Sicari, F. De Pellegrini, and I. Chlamtac, "Internet of things: Vision, applications and research challenges," *Ad Hoc Networks*, vol. 10, no. 7, pp. 1497–1516, 2012.

[5] J. A. Stankovic, "Research directions for the internet of things," *IEEE Internet of Things Journal*, vol. 1, no. 1, pp. 3–9, 2014.

[6] E. Kim, "How 'Internet of Things' startup Jasper became a $1.4 billion company," *Business Insider*, 2014.

[7] L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey," *Computer networks*, vol. 54, no. 15, pp. 2787–2805, 2010.

[8] C. C. Aggarwal, N. Ashish, and A. Sheth, "The internet of things: A survey from the data-centric perspective," in *Managing and mining sensor data*. Springer, 2013, pp. 383–428.

[9] J. Lewis and M. Fowler, "Microservices," 2014.

[10] J. Thönes, "Microservices," *IEEE Software*, vol. 32, no. 1, pp. 116–116, 2015.

[11] "Announcing ribbon: Tying the netflix mid-tier services together," http://techblog.netflix.com/2013/01/announcing-ribbon-tying-netflix-mid.html, accessed: 2017-1-30.

[12] S. Tilkov, "The modern cloud-based platform," *IEEE Software*, vol. 32, no. 2, pp. 116–116, 2015.

[13] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil, "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," in *Computing Colombian Conference (10CCC), 2015 10th*. IEEE, 2015, pp. 583–590.

[14] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.

[15] H. Kang, M. Le, and S. Tao, "Container and microservice driven design for cloud infrastructure devops," in *Cloud Engineering (IC2E), 2016 IEEE International Conference on*. IEEE, 2016, pp. 202–211.

[16] S. Newman, *Building Microservices*. " O'Reilly Media, Inc.", 2015.

[17] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: yesterday, today, and tomorrow," *arXiv preprint arXiv:1606.04036*, 2016.

[18] A. Krylovskiy, M. Jahn, and E. Patti, "Designing a smart city internet of things platform with microservice architecture," in *Future Internet of Things and Cloud (FiCloud), 2015 3rd International Conference on*. IEEE, 2015, pp. 25–30.

[19] B. Butzin, F. Golatowski, and D. Timmermann, "Microservices approach for the internet of things," in *Emerging Technologies and Factory Automation (ETFA), 2016 IEEE 21st International Conference on*. IEEE, 2016, pp. 1–6.

[20] D. Guinard and V. Trifa, "Towards the web of things: Web mashups for embedded devices," in *Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web (MEM 2009), in proceedings of WWW (International World Wide Web Conferences), Madrid, Spain*, vol. 15, 2009.

[21] D. Guinard, V. Trifa, and E. Wilde, "A resource oriented architecture for the web of things," in *Internet of Things (IOT), 2010*. IEEE, 2010, pp. 1–8.

[22] "Philips Hue API," https://developers.meethue.com/philips-hue-api, accessed: 2017-1-30.

[23] "Oauth 2.0, the industry-standard protocol for authorization," https://oauth.net/2/, accessed: 2017-1-30.

[24] F. Montesi and J. Weber, "Circuit breakers, discovery, and api gateways in microservices," *arXiv preprint arXiv:1609.05830*, 2016.

[25] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*. ACM, 2012, pp. 13–16.

[26] M. Chiang and T. Zhang, "Fog and IoT: An overview of research opportunities," *IEEE Internet of Things Journal*, vol. 3, no. 6, pp. 854–864, 2016.

[27] "IoT-Lite Ontology," https://www.w3.org/Submission/iot-lite/, accessed: 2017-1-30.

[28] "Semantic Sensor Network Ontology," https://www.w3.org/TR/vocab-ssn/, accessed: 2017-1-30.

[29] "Web Service Modeling Ontology," https://www.w3.org/Submission/WSMO/, accessed: 2017-1-30.

[30] "Semantic Markup for Web Services," https://www.w3.org/Submission/OWL-S/, accessed: 2017-1-30.

[31] T. C. Project, "Sandbox," Last accessed Feb 2017.

[32] G. Gruman, "Android for work brings container security to google play apps," *Infoworld*, Feb 25, 2015.

[33] J. Bethencourt, A. Sahai, and B. Waters, "Ciphertext-policy attribute-based encryption," in *IEEE Symposium on Security and Privacy*, Washington,DC,USA, 2007, pp. 321–334.

[34] Z. Wang, D. Huang, Y. Zhu, B. Li, and C.-J. Chung, "Efficient attribute-based comparable data access control," *IEEE Transactions on computers*, vol. 64, no. 12, pp. 3430–3443, 2015.

[35] "Withings Body," https://www.withings.com/us/en/products/body, accessed: 2017-1-30.

[36] S. Sheikholeslam and C. A. Desoer, "Longitudinal control of a platoon of vehicles," in *American Control Conference, 1990*. IEEE, 1990, pp. 291–296.

[37] D. Lu, Z. Li, D. Huang, X. Lu, Y. Deng, A. Chowdhary, and B. Li, "Vc-bots: a vehicular cloud computing testbed with mobile robots," in *Proceedings of the First International Workshop on Internet of Vehicles and Vehicles of Internet*. ACM, 2016, pp. 31–36.

[38] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the linux virtual machine monitor," in *Proceedings of the Linux symposium*, vol. 1, 2007, pp. 225–230.