The Operating System of Things

Christopher Streiffer
cdstreif@cs.duke.edu

Ali Razeen
alrazeen@cs.duke.edu

November 10, 2017

1 Introduction

Numerous groups have proposed different IoT (Internet of Things) hub platforms to help users easily deploy and manage the IoT devices in their homes and other environments [23, 14, 16, 19]. These platforms are supposed to hide the complexity of the IoT devices and derive utility from them via apps that monitor the installed sensors and perform automated actions. However, none of the existing platforms have emerged as a clear winner in terms of market share and even though IoT, as a whole, has seen some traction among users, it is still in its infancy compared to adoption of other technologies such as smartphones [10, 17].

We believe that a successful IoT hub platform should provide the following properties: (i) it should be easy for developers to write IoT apps, (ii) it should be easy for users to install an arbitrary mix of apps, and (iii) users should retain control over their devices and know that their preferences and safety are not going to be compromised. It should not be possible for an app to interfere with other apps and send conflicting commands to a shared IoT device, thereby inconveniencing a user, and it should not be possible for apps to violate a user-specified invariant in the IoT environment. To the best of our knowledge, we do not know of an existing hub platform that provides the above properties; they typically restrict how apps are written [14, 16, 9, 6], lack a comprehensive permissions framework that lets user manage an app's access to a device [3, 12, 19, 24], or do not provide an easy way of ensuring user-preferences are violated [23].

In this proposal, we discuss the design of a new hub platform that meets the above requirements. Our platform is inspired by the abstractions used in Software-Defined Networks (SDNs). In SDNs, the data plane and the control plane are decoupled, thereby allowing both to operate independently. For example, when a switch receives a packet from a new sender, it notifies the controller, which then computes how the packet should be handled and installs forwarding rules on the switch. Subsequently, further packets from the same sender are forwarded automatically without the involvement of the controller. The communication between the switches and the controllers is defined by OpenFlow [18], an event-driven protocol. When the underlying network state changes, switches send events to the controller. The controller will forward these events to the network apps running on it. In response to these events, apps may send commands back to the switches.

This event-driven paradigm is central in our hub platform. As with OpenFlow, the hub will send events to apps whenever the state of the IoT environment changes. When the hub receives command from apps, it will perform a series of checks to ensure that the different commands do not conflict with each other, and that the commands do not violate user preferences. If a command passes these checks, it will be sent to the IoT device to perform the actuation task. Otherwise, the command is rejected and the app responsible for the command is notified. Apps may only communicate to the devices via the hub and will not have direct access to them.

The above design meets the criteria we outlined for a successful hub platform: (i) the event-driven paradigm is flexible and imposes minimal restrictions on how developers write their apps, (ii) the hub can inspect and control commands from apps to ensure that users retain control over their devices and that apps do not interfere with each other.

The rest of this proposal is organized as follows. We describe in detail the design and programming model of our platform in Section 2 and Section 3, respectively. We discuss some additional features

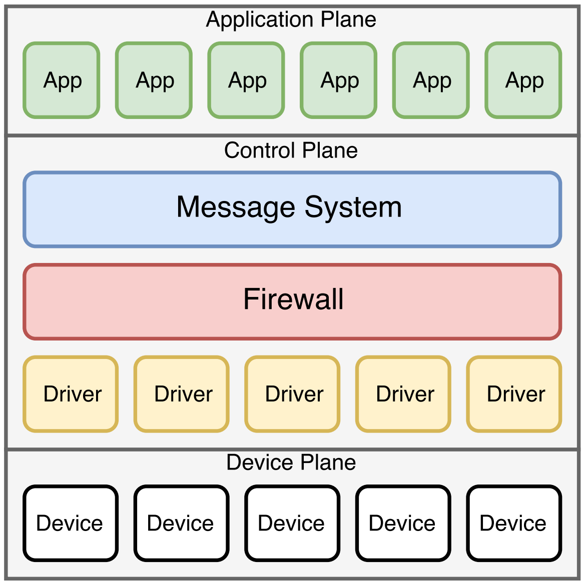

Figure 1: A block diagram of our proposed controller.

we believe are important to the commercial success of our platform in Section 4. Finally, we survey related work in Section 5.

2 System Design

Our proposed platform provides a simple, but powerful model for developing and deploying applications within smart environments. In Figure 1, we show a block diagram view of our controller. At a high level, it consists of three layers – the *application plane*, *control plane*, and *device plane*.

The *control plane* is responsible for handling all communication between the devices and applications, which reside in the *device* and *application* planes respectively. It is entirely event-driven and it uses a combination of periodic polling and push notifications from the devices to keep track of their states. When a device's state changes, the controller broadcasts a state update event to all running applications.

If an application responds to an event with a command, the controller first validates the command. If the command is accepted, then the controller updates its view of the application's state and sends the command to the device. For example, if an application sends a command to turn on a light bulb, the controller will update the application's state with the tuple (A_i, L_i, S_{on}), where A_i represents the application, L_i represents the light, and S_{on} represents the state of the light. This application state is used internally by the controller to compute the proposed state of an application whenever it sends a new command, which is then used for conflict detection and resolution.

Overall, this design provides a powerful abstraction for providing fine-grained security and cleanly separates the three different layers. We now describe each of them in greater detail.

2.1 Device Plane

The *device plane* consists of the IoT devices running within the system. In our design, each device connects to the controller over whichever protocol is best suited for the device e.g. Bluetooth, 802.11, etc. From an application's perspective, the connection between a device and the controller is persistent, and the controller is responsible for handling transient failures.

We note that the device state may be updated either through physical interactions with the device, or through the programmable interface within our system. A physical interaction could range from a power failure to an individual directly interacting with the device i.e. turning off the smart device. We assume that the communication channel between a device and the controller is bidirectional, and that a device can respond to state poll messages from the controller. If supported, the device may also push its state updates to the controller.

2.2 Application Plane

The *application plane* is responsible for running all programs installed within the system. Programs may be written with any programming language e.g. Java, C++, Python, Go, etc. as long as they use our event-driven API. We will provide libraries for each language to minimize the burden of using our API. The programming model for our system will be discussed in more detail in Section 3.

To allow further flexibility, applications are not restricted in the location where they must be executed, and can be run off-device if necessary. For instance, a program which utilizes a deep learning framework can be run on edge infrastructure to gain access to greater computational power. As data analytics have proven to be a driving force behind smart environments, we believe that this design is necessary in order to support smarter applications.

2.3 Control Plane

The *control plane* is composed of three layers – driver, firewall, and message handler. The driver exposes an interface for communicating directly with IoT devices. The firewall handles all communication between the driver and message handler, and acts as a filter to ensure system invariants are not violated and to perform conflict resolution between conflicting application states. The message handler is responsible for maintaining communication with applications, ensuring message delivery, and transmitting commands to lower layers.

Driver Layer: The driver exposes an interface that allows for the controller to communicate with the IoT devices supported by our system. Each IoT device will have an accompanying driver which implements the protocols to communicate with the device. A driver also exposes a common interface to the firewall to allow for communication to be standardized across each device. Further, a driver is responsible for converting generic commands into device-specific operations. This allows for our programming framework to be device agnostic such that developers can design their applications to operate on generic types e.g. Smart Light.

Firewall Layer: The firewall has two primary responsibilities – ensuring system invariants are never violated and performing conflict resolution. The firewall allows for users to implement both coarse-grained and fine-grained policies. The coarse-grained policies consist of defining access control between applications and devices. For instance, the user may specify that an application responsible for controlling the heating should not interact with a smart door lock.

The fine-grained policies allow for users to define device states which should never be violated. These are typically system invariants which should never be violated. Further, we provide a powerful matching engine to allow users to define these invariants based on sophisticated system state, if required.

The second responsibility of the firewall is to perfrom conflict resolution. Because the controller keeps track of application state and device state, the firewall can detect conflicts between updates. Suppose the user has a power-saving application and lighting-management application running within their system. Now suppose the power-saving application issues a command to a turn off a smart light. At the same time, the lighting-management application issues a command to turn the light on. Because the controller keeps track of application state, it will detect a conflicting action between the two applications. In this instance, the firewall will send a notification to the user asking her to resolve the conflict. Further, the system will ask the user if they would like to define a default rule for handling this conflict, or to assign a priority to one of the applications.

Therefore, the flow for conflict resolution consists of (i) checking system invariants defined by the user, (ii) checking application priority, and (iii) asking the user to directly resolve the conflict. In order to facilitate ease-of-use, we provide an administrative interface to the user which allows for them to construct these policies in an intuitive manner.

Messaging Layer: The messaging layer is responsible for ensuring the delivery of messages between the applications and the lower layers. Although all communication is carried out via messages, our system is also capable of supporting applications which require data streams from IoT devices.

3 Programming Model

The programming model of our platform conforms to the following rules:

1. All communication between devices and applications are based on strictly-defined messages.
2. Applications may use filters to subscribe to specific event types.
3. Applications may react to events from the controller by sending commands to the controller.

The controller will send events to an application based on the devices it can access. An application with an access to a light bulb will receive a *state update* message if the user manually turns off the light bulb. However, it will not receive events about other devices. In addition to device state updates, the controller also provides other types of events. For example, to simplify application development, the controller provides a notion of time and makes available the *time change* event. This event will be delivered to an application based on the requested granularity (once every hour, day, etc.). If the controller receives a command from an application, it will respond with a *command status* message, to indicate if the command was rejected or accepted. We believe this event-driven model to be simple and it imposes minimal restrictions on how apps are written.

We expect some IoT devices to stream data at a high rate. To support such devices without the overhead of generating one message per data item, the controller can send a *device data* event that contains within it a device descriptor, which may be used by applications to read data from the device. This reduces the overhead of reading streaming data without ceding the controller's control over the device; if necessary, the controller can always invalidate the descriptor and prevent the application from reading any more data.

If a user revokes an application's access to a device, the controller will send *device offline* message to the application, and will reject all future commands from the application addressed to that device. We believe that this will incentivize application developers to write robust applications, since the API makes it clear that they may lose access to a device, and that the commands they send may not be accepted.

4 Additional Features

We additionally provide two software components for increasing the functionality of our system. The first component is an emulator which can be used to test and debug applications within virtual smart environments. The second component is an application store to connect developers and users.

4.1 Emulator

The primary purpose of the emulator is to create virtual environments for testing and debugging applications. We believe that this will be useful for both application developers and end-users, and describe some use-cases below:

Development: Developers may use our emulator to write applications and test out platform without having to buy the IoT hardware.

Debugging: Because our system is entirely event and command driven, we can provide a deterministic total-ordering of messages processed by the controller. Because all traces of events and commands are logged by the controller, we can recreate the state of the system leading up to a crash using the emulator.

Integration Testing: We believe that it will be highly beneficial for users to be able to test applications within their smart environment before installing them. The emulator can, in conjunction with the installed applications and devices, show users how the application will work with the existing smart environment.

4.2 Application Store

We provide an app store to connect users and developers. The main purpose of the store is to allow users to find applications tailored to their smart environments. Users will be able to query the store to discover applications that they can run with their existing devices. The application store can also suggest new applications or new devices to buy. Finally, the store provides developers with financial incentives to write useful applications.

5 Related Work

A large number of IoT systems have been proposed in the academic literature, the open-source community, and by companies selling proprietary systems. In this section, we broadly survey these existing systems and discuss how they relate to our proposed system.

Controller platforms

A lot of the existing controller platforms have goals similar to ours. They run in the centralized hub of an IoT deployment, mediate accesses to the IoT devices, and provide a single interface over the devices they manage. This interface allows users to access their devices without worrying about hardware-specific details (such as how the devices are connected to the hub and what communication protocol they use). The existing systems typically also provide a dashboard to allow users to easily monitor and manage their IoT devices. In these aspects, our proposed system and these existing systems share a common goal: make it easy for users to manage their IoT devices.

The differences between our proposed system and the existing systems are better understood by recounting the key features of our system. First, the firewall plays a central role in managing conflicting IoT commands from different apps and in ensuring user-specified invariants are not violated. Second, our event-driven programming model is flexible and imposes few restrictions on how developers write their apps; it does not require them to use a particular programming language nor does it restrict apps from using external services (such as the cloud). Third, the permissions framework in our system allows users to control which apps have accesses to which devices, and allows users to easily revoke an app's access to a device. We believe that all three features have to be present to (i) enable developers to easily write IoT apps and publish them to a marketplace, (ii) allow users to download arbitrary third-party apps without worrying about whether the apps may conflict with each other, and (iii) let users retain control over their IoT environment. An added benefit of our event-driven paradigm, in conjunction with the firewall and the permissions framework, is that developers will be trained to write resilient apps from the beginning, since the API makes it clear that an app may lose its access to a device and that the commands they send to the controller may be rejected.

We have not found an existing system that combines all three features and quite notably, we did not find a system with a concept that handles all of our firewall's tasks. We now discuss these existing systems by comparing them to the key features of our system.

HomeOS [23] is remarkably close to our proposed system. It uses Datalog to express access control rules and provide a permissions framework and a simple app-priority scheme to mediate conflicts caused by commands from different apps to the same shared device. However, user-specified invariants are not a first-class citizen in HomeOS, and therefore, it may be difficult to avoid unpleasant outcomes.

Consider the following scenario: a user has a space heater in her bedroom and has an app that turns on the space heater whenever she is in the room and turns it off otherwise. She might have another app that controls the thermostat in the living room and powers on the heating system when the temperature falls below a threshold. She also has an invariant to state that the heating system and the space heater should not both be turned on at the same time.

In our system, this invariant is maintained without requiring any coordination between the apps. If the user is in her bedroom, the commands from the thermostat app to turn on the heating system are rejected. It is not clear how HomeOS will handle such requirements in their current formulation of their datalog rules, which only encodes "that resource r can be accessed by users in group g, using module m, in the time window from T_s to T_e, on day of the week d, with priority pri and access mode a."

Another key difference between our system and HomeOS is in the programming framework. We believe that the event-driven paradigm of our API, which is heavily inspired by OpenFlow, is powerful and yet flexible. It allows developers to write their apps in any language, as long as they can communicate with our controller. Similar to OpenFlow, this also introduces a layer of separation between apps and the controller platform and allows both to evolve independently. In HomeOS, apps have to be written in C# since HomeOS uses C# language features (such as `System.AddIn`) to load apps as modules into the OS. HomeOS also restricts an app's access to the Internet for the sake of user privacy. We do not impose this restriction as developers may wish to write apps that use cloud services to perform analytics or access APIs from other service providers. They may even write apps that run in the cloud. In our system, before users install an app, we display whether the app uses the Internet and where it runs, and leave it to them to decide whether they trust that app.

In Eclipse SmartHome [6], OpenHAB [14] and OpenRemote [16], apps have to be specified as rules within the included rule engine. A rule is an action that executes when the triggers associated with it fires. For example, the user can specify the rule: "turn on the lights in this room if someone walks into it." In these systems, apps are constrained by the rule engine and it is difficult, for example, to write an app that access cloud services.

In contrast to rule engines, DeviceHive [3], Machina.io [12], The Thing System [19] and Zetta [21] provide a flexible API to access the IoT devices. Developers may use web-based protocols in their apps to communicate with these controllers and control the IoT devices. However, these systems lack a permissions framework. Once the user grants an app access to a device, it is difficult for the user to revoke that access.

The controller platforms discussed so far (including our proposed system) are primarily focused on IoT environments with a single hub controlling a multitude of devices. In contrast to this, DSA [4] and Kaa [11] support distributed IoT networks, where multiple IoT environments, each with a separate hub, are logically connected together. DSA and Kaa enable the collection of data from distributed IoT environments for analytics purposes, and also provide an API to control the various distributed devices. Distributed IoT environments may be used in agricultural settings (monitoring many geographically-dispersed farms), industrial settings (monitoring factories), or retail environments (monitoring large shops).

Although we are interested in developing a system that also works in distributed IoT environments, we leave it to future work. That said, DSA and Kaa suffer from some of the same limitations as the controller platforms seen earlier: they both do not have a firewall to handle conflicting commands and ensure user-level invariants, and their permissions framework is inflexible.

Programming frameworks

Different programming frameworks have been proposed to write IoT apps. These are orthogonal to our system and are, in theory, compatible with the event-driven API provided by our system.

Node-RED [13] and Calvin [2] are frameworks for programming IoT apps. They are based on the flow-based programming [7] paradigm where programs are expressed in terms of their data flows. That is, instead of writing an IoT app as a monolithic program in a control flow language such as C or Java, developers express their apps in terms of a directed graph. Each node in this graph represents either a (i) data source, (ii) data sink, or (iii) an operation on the data, and the edges represent the flow of data between nodes.

As these frameworks abstract away the details of how the app executes in practice, it makes it possible for a developer to experiment with different deployment strategies without requiring extensive changes to the app. For example, a developer could initially choose to deploy the whole app in the cloud. At a later time, she can choose to execute parts of the app graph directly on the IoT devices. She just has to ensure that the nodes connected in the graph have valid communication channels. This is not as easily achieved in traditional programming languages.

IFTTT (If-This-Then-That) [9] is a trigger-action rule engine for writing simple IoT apps. It works with a large number of commercial IoT devices and online services, such as Facebook and Twitter. This is similar to the rule engine in OpenHAB and OpenRemote, but it runs in the cloud and is not tied to a particular controller. IFTTT does not provide any conflict-avoidance mechanism and it has been shown that it is possible for users to combine multiple rules and inadvertently cause a violation in terms of personal privacy, security, and etc. [28].

Visualization tools

A number of systems have been developed primarily to visualize data from IoT sensors. Bug Labs [1] provides Dweet [5], a simple messaging abstraction for publishing data to the cloud in key-value pairs, and Freeboard [8], a web-based visualization dashboard for data collected via Dweet. OpenIoT [15] is system for collecting data from IoT sensors into the cloud for monitoring and visualization. OpenIoT also semantically annotates the data collected from sensors using the W3C SSN (Semantic Sensor Network) specification [20]. These visualization tools are orthogonal to our proposed system. Users may, if they *choose to do so*, install apps to publish their data to the cloud and visualize them with existing tools.

Research proposals

Although none of the controller platforms we surveyed combines the features we believe are crucial for a successful IoT system, some of the problems we aim to address have been discussed in the academic literature.

In a recent proposal, Balaji et al. highlight the importance of modeling and enforcing constraints an IoT environment [22], which are dictated by physical and human laws, and by user preferences. This is equal to the firewall in our proposed system. They briefly propose capturing the constraints in a state-space model and capturing the dependencies between a physical environment and the IoT devices in it in a graph. They suggest that this will help in checking if an actuating decision will violate a constraint. Unfortunately, they stop short of showing a concrete solution and leave it as an open problem for further research. In our system, constraints (or invariants), are captured and enforced by the firewall.

There has also been work in the addressing the problem of conflicting IoT apps. DepSys is one of the first systems to propose a solution to conflicting apps in a smart home environment [27]. In DepSys, developers have to specify how their app affects the physical environment, the different operations an app may make, and the priority of each operation. We consider this approach as too burdensome to the developer. SIFT is another system to detect conflicts by smart home apps [25]. In SIFT, developers write their apps in if-then rules, similar to IFTTT and other rule engines. SIFT will use model checking to test the apps against policies specified by users to look for policy violations. This approach is unsatisfactory as it severely constraints how developers may write their apps.

Ma et al. studied conflicts in a city-wide IoT environment ("smart cities") [26] and discuss the reasons why conflicts happen. They also propose a watchdog service to detect and resolve conflicts. Their watchdog uses a variety of inputs to detect conflicts, including DepSys, and resolves conflicts by involving human decision makers in the loop and machine learning. Although we are trying to address single-hub IoT environments to begin with, it is possible to re-use the techniques proposed by Ma et al. For example, it is possible to use machine learning in our firewall to quickly address conflicts during runtime.

Lee et al. showed that the permissions scheme used in existing IoT platforms to grant apps access to devices suffers from a number of shortcomings [24]. Because of its coarse-grained nature, apps will either have no access to a device or have complete control over it. While this is clearly undesirable in adversarial settings, it is inadequate even in the absence of malicious apps since multiple apps having complete control over a shared device may result in conflicting control over the device. To remedy this, they propose FACT, a fine-grained access control system for IoT devices where each sensing and actuating functions supported by a device are protected by individual permissions.

They provide the example of a smart door lock that has two sensing states, *battery life* and *locked status*, and supports one actuating command, *lock*. In this example, a battery monitoring app should only request permissions to read the *battery life* of the lock but not permissions to either read the *locked status* or send the *lock* command. FACT is complementary to our proposed system and it is possible to use some of the same techniques to implement our permissions framework.

References

[1] Bug Labs. http://buglabs.net/.

[2] Calvin. https://github.com/EricssonResearch/calvin-base.

[3] DeviceHive. https://devicehive.com/.

[4] Distributed Services Architecture. http://www.iot-dsa.org/.

[5] Dweet. https://dweet.io/.

[6] Eclipse SmartHome. https://www.eclipse.org/smarthome/.

[7] Flow-based Programming. http://www.jpaulmorrison.com/fbp/.

[8] Freeboard. https://freeboard.io/.

[9] IFTTT helps your apps and devices work together - IFTTT. https://ifttt.com.

[10] Internet of Things Institute - Why the IoT is in its infancy. http://www.ioti.com/iot-engineering/welcome-commodore-64-days-iot.

[11] Kaa. https://www.kaaproject.org/.

[12] Machina.io. https://macchina.io/.

[13] Node-RED: Flow-based programming for the Internet of Things. https://nodered.org.

[14] OpenHAB. http://www.openhab.org.

[15] OpenIoT. http://www.openiot.eu.

[16] OpenRemote. http://www.openremote.com.

[17] Rise of IoT - Internet of Things. https://www.huffingtonpost.com/entry/rise-of-iot-internet-of-things_us_59b373dee4b0bef3378ce052.

[18] SDN / OpenFlow — Flowgrammable. http://flowgrammable.org/sdn/openflow/.

[19] The Thing System. http://thethingsystem.com.

[20] W3C SSN. `https://www.w3.org/2005/Incubator/ssn/ssnx/ssn`.

[21] Zetta. `http://www.zettajs.org`.

[22] B. Balaji, B. Campbell, A. Levy, X. Li, A. Mayberry, N. Roy, V. N. Swamy, L. Yang, V. Bahl, R. Chandra, and R. Mahajan. Modeling Actuation Constraints for IoT Applications. In *arXiv:1701.01894*, January 2017.

[23] C. Dixon, R. Mahajan, S. Agarwal, A. Brush, B. Lee, S. Saroiu, and P. Bahl. An Operating System for the Home. In *Proceedings of NSDI '12*, April 2012.

[24] S. Lee, J. Choi, J. Kim, B. Cho, S. Lee, H. Kim, and J. Kim. FACT: Functionality-centric Access Control System for IoT Programming Frameworks. In *Proceedings of SACMAT '17*, June 2017.

[25] C.-J. M. Liang, B. F. Karlsson, N. D. Lane, F. Zhao, J. Zhang, Z. Pan, Z. Li, and Y. Yu. SIFT: Building an Internet of Safe Things. In *Proceedings of IPSN '15*, April 2015.

[26] M. Ma, S. Preum, W. Tarneberg, M. Ahmed, M. Ruiters, and J. Stankovic. Detection of Runtime Conflicts among Services in Smart Cities. In *Proceedings of SMARTCOMP '16*, May 2016.

[27] S. Munir and J. A. Stankovic. DepSys: Dependency Aware Integration of Cyber-Physical Systems for Smart Homes. In *Proceedings of ICCPS '14*, April 2014.

[28] M. Surbatovich, J. Aljuraidan, L. Bauer, A. Das, and L. Jia. Some Recipes Can Do More Than Spoil Your Appetite: Analyzing the Security and Privacy Risks of IFTTT Recipes. In *Proceedings of WWW '17*, April 2017.