Interfacing the Internet of a Trillion Things

Bradford Campbell[†], Pat Pannuto[†], and Prabal Dutta
Electrical Engineering and Computer Science Department
University of Michigan
Ann Arbor, MI 48109
{bradjc,ppannuto,prabal}@umich.edu

ABSTRACT

Meaningful, reusable applications built on top of ubiquitous and networked devices will be slow to materialize as long as device APIs vary widely, communication protocols are not standardized, and programming support is limited and inconsistent. When even feature-identical devices present different APIs and application creators are burdened with managing the variability, the promise of the swarm of devices will go unrealized. We start addressing this issue by providing a model for devices, based on input and output ports, that allows for a set of common interfaces to represent a range of devices. Further, we provide a solution to the bootstrapping problem, providing a general means to bridge the adoption gap for a new API for the Internet of Things. We borrow both the name, *accessor*, and several key design concepts from a recent proposal by Latronico et. al, for our interface layer that wraps currently non-conforming devices with the standard interface. We show how a small, straightforward to write (and read) JavaScript file can convert diverse devices into common interfaces that are conducive for creating applications.

We realize our system with three environments that can execute accessors, Python, Java, and Node.js, a range of accessors for both IoT and legacy devices, and a browser-based application for interacting with devices using our proposed interfaces. We show how the same accessor mechanism can form synthetic devices with higher-level interfaces and we outline how our system can be extended to support authentication, accessor control, and cloud storage support.

[†]Co-primary authors.

SWEC'15, April 13, 2015, Seattle, WA, USA

1. INTRODUCTION

Networked, ubiquitous devices, from smart appliances to automatic deadbolts, from power meters to adaptive light bulbs, are rapidly increasing in number to form the swarm. These devices are, and will be, little more than decoration, however, without a reliable and useful way to communicate and interact with them. The promised benefits of the swarm—comprehensive sensing and monitoring, predictive device responsiveness, and personalized infrastructure—will be slow to materialize if "connected" devices are locked behind incompatible communication protocols, inconsistent APIs, and incongruent data models. If devices can only communicate with their own cloud and every pair of devices needs a unique adapter to talk to one another, the swarm will be completely networked but not connected to anything.

Addressing this issue will require standardizing the interfaces, communication protocols, and public APIs of devices. Much as the OpenFlow [16] protocol enabled Software Defined Networking research to thrive, when communications and interfaces are standardized, higher-level exploration prospers. This standardization process for the Internet of Things is a gargantuan task, however, covering many layers of the stack, hundreds of manufacturers, and a wide suite of functionality. While we do not expect the Internet of Things to converge on a unified interface soon, we would like to build applications on top of the swarm today that do not have to be re-written as the standardization process occurs.

To enable applications, bridge the gulf between devices, and begin the transition towards standardization, we propose defining standard interfaces for devices by modeling each device as a canonical "black box" with input and output ports. Ports have a name, direction, type, optional unit, and description. A port-based model for devices is intuitive, maps well to simple and possibly pictorial representations of devices, and provides a clean abstraction for distinct devices to expose the "same" port. Conceptually, interacting with a device requires learning which ports the device has and sending data to or requesting data from the correct port for a particular application. To ease port discovery and understanding, we group *ports* into a set of standard *interfaces* that devices can instantiate. These interfaces form the foundation of our proposed standard API.

Currently, device APIs do not necessarily follow this model, and those that do, do not share a standard underlying protocol for reading and writing ports. The specifics of this protocol, whether it be based on HTTP, CoAP [21], the Simple Thing Protocol [5], MQTT [3], WebSockets [11], or some other application layer protocol, are outside of the scope of this paper. We expect that a standard will emerge, but our system is not constrained to the eventual protocols. To allow for building applications today, however, we provide a method, based on the previously defined concept of "accessors" [14], of wrapping non-conforming devices. Accessors in our system are snippets of JavaScript code that provide the port-based interface while converting port reads and writes to the underlying native API of a particular device. If a device is updated with support for the future standard, the accessor is simply dropped and the application continues to execute unmodified.

Accessors are executed inside of an accessor runtime that is responsible providing the standard, ports-based interface of devices to applications in a native way given its programming language and execution environment. Runtimes minimize the burden of writing accessors by requiring the JavaScript accessors to focus on interacting with the device's API and not any complicated internal runtime APIs. Runtimes also allow using accessors that include other accessors as dependencies. This allows the same infrastructure to provide more desirable, higher-level interfaces such as "watch a movie" instead of "turn on the TV" and "turn on the DVD player."

Standardization processes often require momentum in order to gain wide traction. Our system and the use of accessors as a "shim" layer until device APIs standardize allows for the port-based model to gain momentum, encouraging new devices to conform in the future. As devices standardize, the runtime simply discards the JavaScript accessor.

To further explore the runtime architecture, accessor design, and default port interfaces, we implement a prototype of our proposed infrastructure. This comprises multiple runtime environments and allows us to evaluate the complexity of creating an accessor and the effectiveness of our system.

2. RELATED WORK

Providing a consistent and composable interface for ubiquitous devices is critical for device usability and creating applications as the devices become more numerous. Another system that targets these goals is the Thing System [6]. The Thing System provides a web server that presents a common GUI for interacting with a range of devices. Each supported device has a corresponding JavaScript wrapper that adapts the device interface into the Thing System server. Further, they have defined two data communication protocols, the Thing Sensor Reporting Protocol [7] and the Simple Thing Protocol [5]. While the Thing System is motivated by the same general problem, our approach differs in several key aspects. First, we focus on simplifying the task of writing the per-device JavaScript wrapper by eliminating any framework-level API that the wrapper must be compatible with. This reduces the number of lines of code in a wrapper for the same device by half or more. Second, our system provides a mechanism for easily creating synthetic devices with more useful, higher-level interfaces from existing devices. Third, our system is designed as a kernel that can be embedded in other applications such as smartphone apps or web services.

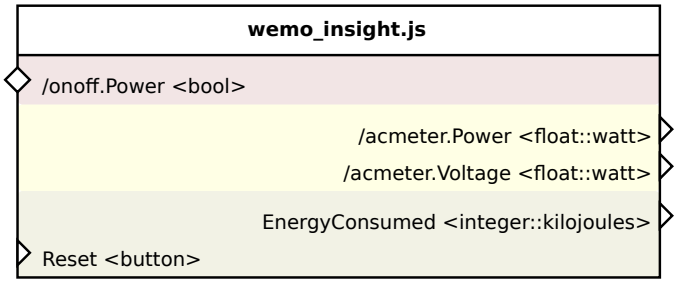


Figure 1: Block model for the WeMo Insight. The Insight is both an AC switch, supporting the `/onoff` interface, and an AC meter, supporting the `/acmeter` interface. The Insight also supports an odometer-like "Total Energy Consumed" which has no standard interface. The Insight accessor exposes custom ports to present the device-specific features.

Defining a device-level data model for classes of devices is essential for building composable and shared applications. Bluetooth [1] has a suite of well defined services [2] that a device can support that specify data descriptions and types. This ontology allows applications to interface seamlessly with different peripherals that support the same services. While Bluetooth has a much different focus than our system, we could leverage these interface specifications in our device definitions.

Accessors [14] is an existing project that we borrow the concept of JavaScript wrappers for devices from. While the original project focuses on wrapping devices in order to explore models of computation and interaction, we leverage accessors as a method to build applications on top of.

3. STANDARD INTERFACE MODEL

To provide a consistent view of a wide range of devices, we propose modeling all device interactions as a read or a write to a well-defined port. As an example, Figure 1 shows a representation of an AC relay device that also meters power (e.g. The WeMo Insight [8]). It has one input/output port ("Power") that when written to turns the attached load on or off and when read from returns the current power state of the load. The remaining ports are strictly output ports and only support reading the current power draw, daily energy consumption, and instantaneous RMS voltage.

Ports for a device are well defined with types and optional units. Types range from primitive types to higher-level, but still simple, types such as color. Units are primarily used with numeric types. For instance, they allow a monetary port to specify that it reports in USD. Types and units aid in connecting ports between devices to ensure data compatibility.

The port-based model allows for a clean representation of a range of devices. Specifying control and query actions as ports aligns well with traditional interaction modes such as physical buttons, online web forms, GUI elements, and status displays. Other potential device models may suitably capture interaction patterns with devices. We advocate for the port based model for three reasons. First, describing physical devices as a box with inputs and outputs is intuitive. Second, the box-and-port model adapts well to describing interactions between devices pictorially, which may be a useful future feature. Third, two devices, such as two different light bulbs, may expose the same port. Applications could interact with both bulbs knowing only about the port, and not about details of the device.

3.1 Device Interfaces

Interfaces in our system are groups of ports that share some common relation. Interfaces are specified in a namespaced hierarchy to allow for logical grouping. This allows interfaces to not only specify ports and how to interact with the device, but also provides some descriptive information about a device that implements them. For example, a device that provides the pcrlighting/onoff interface describes itself as a type of light while also specifying that it provides a "Power" port.

These interfaces, as in all interface design, must be carefully composed. Ideally they should be widely reusable to facilitate application design. With common interfaces, applications can be implemented on top of the interfaces instead of specific devices. For instance, an application that turns the lights off in a room when the occupants leave that is implemented on top of the lighting interface will work without modification in rooms containing different smart light bulbs.

Each device is not restricted to a single interface, in fact, we expect most devices to implement multiple interfaces for different slices of their functionality. For instance, a power meter with an included relay would provide both the "OnOff" interface and the "PowerMeter" interface. Further, we expect that there will be device- and vendor-specific interfaces for features that are specific to only certain devices. But, much in the way that instruction set architectures have evolved, we hope that interfaces will standardize in the future.

3.2 Interaction

Once a common interface model has been established for devices there must be a standard communication protocol for interacting with the interfaces. While specifying a specific protocol is outside of the scope of this paper, we anticipate that HTTP or CoAP may be a good fit as interfaces and ports may map well to URLS.

4. ACCESSORS

Our proposed standard interface presents a bootstrapping issue. There is no incentive to program against a standard interface if no devices support it and there is no incentive for devices to support a standard interface if no applications use it. Accessors solve this issue by acting as a shim layer. They wrap device-specific implementations to present an API that adheres to the standard interface.

Accessors are not restricted to simple protocol translations, however. An accessor holds state and can perform arbitrary computation. With this, accessors can translate device-centric interfaces into semantically meaningful interfaces for synthetic devices.

4.1 Accessor Design

Accessors are designed to be easy to write and comprehend. The expectation is that a large number of accessors will need to be written, one for every networked device currently on the market. This implies that accessor authors are unlikely to be experts in the accessor programming environment or even programming at all.

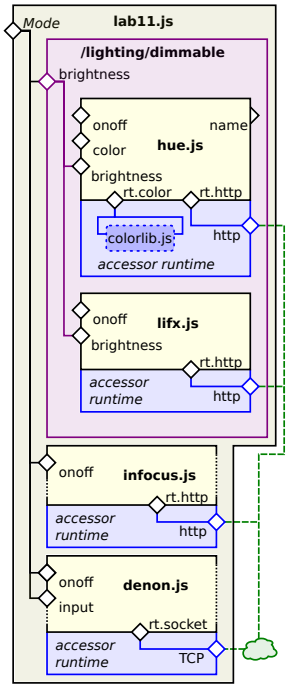
To maximize accessibility and minimize maintenance overhead, an accessor is a single JavaScript file. JavaScript is no longer a browser-only language. It has been shown to be both quickly accessible to novice programmers [15, 20] and viable for advanced applications [17]. An accessor is a single JavaScript file. Accessors express their capabilities imperatively. They `provide_interface()`s for standard interfaces and `create_port()`s for device-specific features. The design explicitly rejects any human-authored metadata files for an accessor, as this creates cognitive load to share state across multiple files.

4.2 Lifecycle and Execution Model

When an accessor is first loaded its `init` method is executed to allow the accessor to verify it can connect to the device and to load any data structures as necessary. From there, all interaction with the accessor occurs by the runtime calling the accessor's port functions. Each input (or inout) port has a corresponding function in the accessor. When an application instructs the runtime to set a port, the runtime calls the port function with the new value of the port. The accessor is responsible for correctly transmitting that value to the device. If the runtime removes the accessor, its optional `wrapup` function is called, allowing the accessor to perform any necessary cleanup.

4.3 Masking JavaScript Runtime Variability

Pure JavaScript has no I/O capability. JavaScript runtime environments (e.g. browsers, Node.js) provide APIs for web requests or other I/O. To abstract JavaScript runtime variations, we introduce an accessor runtime API. The API includes common interfaces such as HTTP and Berkeley sockets. Additionally, accessor authors can import external JavaScript libraries (e.g. a colorspace conversion library) and the accessor runtime will ensure they are available when the accessor is run. By scoping all I/O under our accessor runtime API, we can encapsulate and embed accessors. This enables us to create first-class bindings between the JavaScript accessor and other languages.



(a) Block and Port Model of lab11

```
$ cat lab_control.py                                  | Lab Control Application
 1 import accessors                                   | Load the accessor library
 2 lab11 = accessors.get_accessor_by_location(        | Load the lab11 accessor (creates the object in (a))
                      '/usa/48109/uofm/lab11.json')   | Returns a Python object with ports as attributes
 3 def watch_movie():
 4   lab11.mode = "Movie"                             | Set and fire the Mode port

$ cat lab11.js                                        | Accessor for Lab (A Synthetic Device)
 5 var lights, proj, receiver;                        | References to other accessors
 6
 7 function init() {                                  | Called when accesor is loaded (line 2)
 8   proj  = load_dep('/display/', get_param('infocus'));  | The parameters are URIs for accessors with the
 9   audio = load_dep('/receiver', get_param('denon'));    | right configuration for our projector and receiver
10   lights = load_dep('/lighting/dimmable',          | Runtime creates an aggregate of all accessors
                       '/usa/48109/uofm/lab11');      | with dimmable interfaces at that location
11   create_port("inout", "Mode", {                   | We create a custom port unique to this accessor.
       type: 'select',                                | For a select-type port, the runtime will enforce that
       options: ['Empty', 'Work', 'Music', 'Movie']   | only valid values are passed (line 4). Custom ports
     });                                              | behave the same as ports from standard interfaces
12 }
13
14 function Mode(mode) {                               | This function is invoked by line 4
15   if (mode == "Movie") {
16     lights.brightness = 0.25;                      | This will apply to all the lights
17     proj.power = true;
18     audio.power = true;
19     audio.input = 'Media PC';                      | Input is implicit given Movie Mode
20   }
21   ...                                              | snipped
22 }
```

(b) Snippet of our lab control applet and the composite accessor it uses.

Figure 2: Composite Accessor Example. To show the ease of programming on top of accessors and the power of composite accessors, we show a simple lab control applet. The `lab_control.py` applet uses the composite accessor `lab11`, which integrates the lighting, projector, and audio receiver into a single synthetic device. `lab11` uses absolute knowledge about the room and its exact devices—lines 8 and 9 reference the exact projector and audio receiver—as well as abstract knowledge provided by the runtime—line 10 requests all of the dimmable lights in the room—to instantiate itself.

4.4 Synthesizing Devices by Composition

The utility of accessors extends beyond acting as shims for legacy devices. By composing accessors, users can create abstract devices, such as a smart home. An accessor can express a dependency on a specific device—*My Oven*—or a collection—*All devices that support the* `/light/lighting` *interface at 123 First St., Ann Arbor, MI*.

Accessors that compose other accessors effectively create new "devices". These new devices can implement standard interfaces, e.g. `/smarthome/lights`. Because an accessor is simply JavaScript, higher-level devices can easily write logic to abstract variations in device capability, e.g. map a request to dim lights onto simpler binary lights using a user-set threshold or expose interfaces with semantic meaning such as "watch movie" instead of `Lights.Dim`, `TV.Power`, `TV.Input`, and `DVD.Power`. Composite accessors provide a mechanism to map from a device-centric view—*turn on stereo*—to a semantic view—*play music*—and can act as a key enabler towards pervasive computing.

5. IMPLEMENTATION

An instance of our web runtime is available at accessors.io and currently includes about a dozen accessors, including multiple devices in the same class[1] and a composite accessor that creates a synthetic device that represents our lab. Figure 2 includes a snippet of the composite accessor and demonstrates how a minimal Python program can use a high-level accessor to realize complex semantic interfaces, e.g. "Watch a movie".

5.1 Accessor Host Server

As a first step, the accessor host server "compiles" all of the accessors to an intermediate representation in JSON. This compilation pass first validates the accessor. It ensures that an accessor implements every port in an interface it claims to provide, checks that any dependencies are valid, and is expanding to ensure that runtime APIs are used correctly. The output of the compilation is a JSON blob that includes metadata about the accessor, its runtime requirements, external libraries, interfaces provided, ports created, some additional metadata, and the original accessor JavaScript. Accessor runtimes download this intermediate representation when they load a new accessor. This allows runtimes to ensure that all dependencies are met before initializing an accessor.

The current implementation has a single, centralized accessor host server. Device-specific accessors, e.g. a Phillips Hue accessor, are placed in a wide tree hierarchy, organized by vendor. Accessors can require parameters, e.g. a `bridge_url` for the Hue.

[1]The ACme++ [9] and a WeMo Insight [8] both support the `/onoff` and the `/power/ac_meter` interfaces.

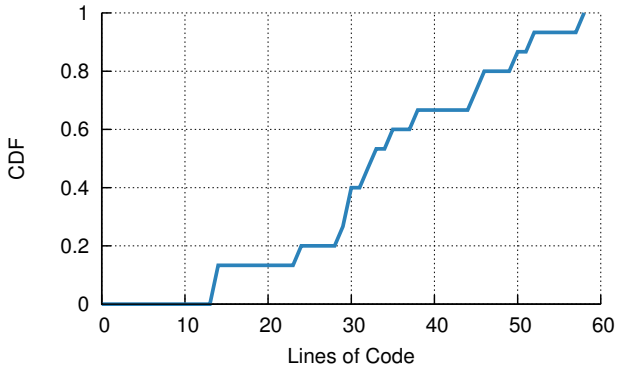


Figure 3: CDF of lines of code for 15 accessors. As a rough measure of the overhead for authoring accessors, we consider their size. Our largest accessor is only 58 lines of JavaScript, suggesting that writing accessors is accessible to a diverse audience of potential coders.

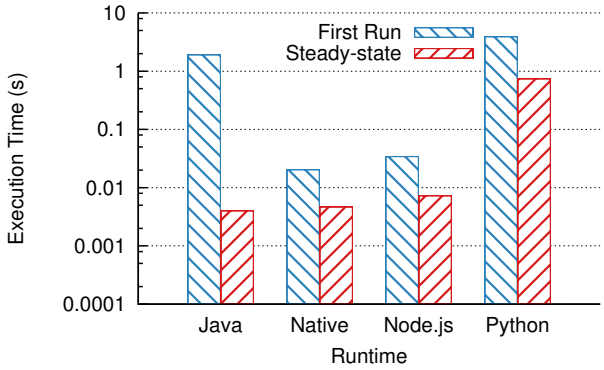


Figure 4: Accessor Runtime Overhead. To measure runtime overhead, we run a loop of HTTP requests to `localhost` using accessors from various runtimes. Running accessor JavaScript code in non-JavaScript-based runtimes requires an embedded a JavaScript engine (Java) or communicating with an external JavaScript engine (Python). For non-JavaScript runtimes, loading the JavaScript engine adds a one-time warmup penalty. For the non-embedded Python case, interprocess communication adds significant overhead to the continued execution.

The accessor host server includes a location-oriented hierarchy that holds parameters for specific instantiations of devices, e.g. `/usa/48109/uofm/lab11/hue_pat.json`. This file contains parameters for the Hue bulb ID and bridge URL for the Hue at Pat's desk and a pointer to the accessor for the device at `/devices/Phillips/hue.json`. The synthetic lab11 composite accessor also resides in the location tree on the accessor host server. Runtimes treat all objects in the location tree as complete accessors, and will resolve indirections such as the device reference for Pat's Hue transparently.

5.2 Accessor Runtimes

We implement three accessor runtimes. One implementation is written in JavaScript on Node.js, which has native support for executing JavaScript [4]. The second runtime we develop in Java using the Nashorn engine from Java 8 to execute JavaScript [18]. Finally, we create a Python runtime. Python has no means to directly execute JavaScript, so we use python-bond, a library that bridges Python and an instance Node.js [10].

Additionally, we create a webserver and RPC server to build a "browser runtime". Accessor code does not actually execute in the browser. We initially attempted an in-browser accessor runtime, but rejected the effect as browsers are too sandboxed of an environment to support the accessor runtime API. Browsers support only websockets, requiring a support server to proxy other protocols such as UDP or TCP; the same-origin policy impedes the ability to support devices that neglect the `Access-Control-Allow-Origin` header—which both the audio receiver and project in our lab omit—, requiring yet another proxy for HTTP requests. Instead, we build an RPC server with a REST API. This REST API maps the ports for accessors to `GET`, `PUT`, and `POST` for reading, setting, and firing ports respectively. The webserver uses the metadata from the compiled accessor to build a GUI on-demand for accessors. This accessor webserver is available at accessors.io.

6. INITIAL EVALUATION AND INSIGHTS

Our proposed standard interface and accessor environment are very young—we have written a total of fifteen accessors. However, this preliminary exploration has been very informative and shaped many of our implementation design choices. Here we attempt to capture that reasoning and present preliminary evidence for why we believe our architecture can scale far beyond fifteen accessors.

6.1 Accessor Creation Overhead

Essential to encouraging traction and adoption of our system is minimizing the barrier to creating and maintaining accessors for devices. This encompasses reducing the amount of framework overhead present in accessors and providing a comprehensive standard library to use when writing accessors. Towards this reduction, a key element in our design is the single-file, self-describing accessor. Our initial design required users to explicitly list ports and interfaces in a separate metadata section. With only the preliminary set of accessors, the authors continually failed to keep the metadata and implementation in sync, motivating our decision to programmatically extract all metadata from the accessor source.

To quantify the complexity of accessors we examine the number of lines of JavaScript source code present in the fifteen accessors we have created so far. Figure 3 shows the CDF of the files. On average, our accessors are 33 lines long, and our longest is 58. While this evaluation does not replace a user study or a more thorough evaluation of the accessor creation complexity, it does provide some initial insight into the relative simplicity of creating a JavaScript accessor.

6.2 JavaScript Execution Overhead

Runtime environments must be able to execute JavaScript to use accessors, however, most programming languages and runtimes do not have existing JavaScript engines built in. To create runtimes, then, one must either compile the JavaScript to the runtime's native language or execute the accessor in a separate JavaScript environment. Figure 4 compares the overhead of the three runtimes described in Section 5.2. We run a small accessor app that makes continuous HTTP requests to localhost. We also include a native JavaScript applet that directly issues the same HTTP requests as a baseline.

The most significant runtime overhead is loading the JavaScript engine for non-native JavaScript runtimes, which leads to the high cost for the first run. Most JavaScript engines include analyses that optimize running code, accounting for the more modest improvements in native JavaScript. With Nashorn, Java is able to run JavaScript directly on the Java Virtual Machine, resulting in near-zero context switch overhead between runtimes. As Java is generally more performant, the Java runtime is actually able to outperform even the native JavaScript. Python, on the other hand, requires continuous interprocess communication between the Python and Node.js engines. The python-bond library communications between processes by reading and writing serialized JSON to stdin/stdout, a particularly inefficient mechanism, resulting in about $100\times$ runtime overhead.

The high performance of the native Node.js and the integrated Java solution, coupled with the diverse and growing number of embedded JavaScript components and engines [13, 19], lead us to believe that choosing JavaScript as the intermediate language is the correct choice. The current performance of the Python implementation is largely an implementation detail, and already new libraries such as PyV8 [12] are emerging to run JavaScript from Python with much higher efficiency.

7. FUTURE DIRECTIONS

Our prototype device interface and accessor system presents several areas for future work and exploration.

7.1 Authentication

The accessor system we propose provides a method for users to access data and control devices, but does not provide a mechanism for validating that the users should be able to access those devices. We intentionally do not build authentication into the accessor host server as this does not provide a method for revoking access. A user can cache an accessor and execute it later, even if the user could not re-request the accessor from the server. Therefore, authentication must exist between the executing accessor and the end device. This, however, requires the accessor to understand the identity of the user and perform the possibly complex authentication procedure itself, burdening the accessor creator.

A possible solution is to allow the accessor runtime to perform the authentication on behalf of the accessor and then have it provide the accessor with a token that it can use in its requests. This approach is feasible if specifying the authentication scheme and authentication parameters can be done in a concise way for a range of devices, that is, that there are only a handful of authentication schemes used in practice that can be consistently parametrized. Surveying currently used authentication mechanisms and integrating them into accessors is left as future work.

7.2 Composing Accessors

Composing accessors can be a very powerful tool for creating complex interactions with the physical world without a prohibitively high barrier to entry. Our system includes one form of composed accessors in the form of accessors that use other accessors as dependencies. However, another useful composition may be an event-based model where a change in the output of one accessor triggers an event in another accessor. Take, for example, a composition of the "RoomTemperature" and "Hue" accessors. A user may wish to illuminate the Hue when the room temperature exceeds a certain threshold. Composing accessors in this manner requires an environment where this logic can be specified and the long-running execution can occur. How to enable this type of composition of accessors is left as future work.

7.3 Authorizing Device Communication

Once devices *can* communicate, there needs to be a mechanism for determining if they *should* communicate. While devices may initially be trusted, bugs or malicious code should not be able to cause devices to interact in a manner the user does not expect. The port based definition of devices allows for one natural method to restrict communication. The management environment discussed in Section 7.2 can issue a pair of cryptographic keys for the communicating devices that are assigned to the relevant ports. Those devices will now only listen to messages for specific ports that are encrypted with the correct keys. Any attempts by a misbehaving device to control a device it is not allowed to will be ignored.

7.4 Seamless Cloud Interaction

Accessors and the standard device model provide two natural mechanisms for leveraging cloud resources with device interactions. First, certain low-capability devices that are constrained by energy-harvesting power supplies or limited network connectivity can be proxied in the cloud. That is, a cloud endpoint would provide the port interface on behalf of the device, and all interactions would be handled by the cloud instead of the actual device. Second, specified ports could be handled by the cloud instead of the device. For instance, in a power metering example, the power meter can easily handle a current power query, but a port that provides historical power data over some time range may be much easier to implement with a cloud service that is collecting the historical data. A mechanism similar to an HTTP redirect issued by the device would likely make the hand-off seamless.

8. CONCLUSIONS

The Internet of Things is missing a standard interface. This complicates building applications which have to compensate for the variability in devices, are limited to only specific devices, and have to be updated when new devices are added. We propose a model, based on defining devices as blocks with input and output ports, for devices that allows for a common interface across devices that applications can be built on. To enable this functionality today, before devices adopt this model, we leverage the concept of accessors, or JavaScript wrappers around devices that convert the device-specific interface into our proposed model. These accessors are simple to write and can be removed when the device supports the standard interface. Our implementation of accessors allows new applications to benefit from this interface while providing a platform for addressing issues such as authentication, access control, and device composition. Building useful applications is a primary goal of the swarm, and our system is a fundamental step towards achieving that goal.

9. ACKNOWLEDGMENTS

This research was conducted with Government support under and awarded by DoD, Air Force Office of Scientific Research, National Defense Science and Engineering Graduate (NDSEG) Fellowship, 32 CFR 168a This work was supported in part by the TerraSwarm Research Center, one of six centers supported by the STARnet phase of the Focus Center Research Program (FCRP), a Semiconductor Research Corporation program sponsored by MARCO and DARPA. This material is based upon work partially supported by the National Science Foundation under grant CNS-1350967, and generous gifts from Intel and Texas Instruments.

10. REFERENCES

[1] Bluetooth. https://www.bluetooth.org.
[2] Bluetooth Services. https://developer.bluetooth.org/gatt/services/Pages/ServicesHome.aspx.
[3] MQTT. http://mqtt.org.
[4] Node.js. http://nodejs.org/.
[5] Simple Thing Protocol. http://thethingsystem.com/dev/Simple-Thing-Protocol.html.
[6] The Thing System. http://thethingsystem.com/index.html.
[7] Thing Sensor Reporting Protocol. http://thethingsystem.com/dev/Thing-Sensor-Reporting-Protocol.html.
[8] Belkin. WeMo Insight Switch. http://www.belkin.com/us/support-product?pid=01t80000003JS3FAAW, 2014. Part #: F7C029fc.
[9] B. Campbell. ACme++. http://lab11.eecs.umich.edu/pcb.html#ACme++, Jan. 2014.
[10] Y. D'Elia. Python Bond. http://www.thregr.org/~wavexx/software/python-bond/.
[11] I. Fette and A. Melnikov. The WebSocket Protocol. RFC 6455 (Proposed Standard), Dec. 2011.
[12] L. Flier. PyV8. https://code.google.com/p/pyv8/.
[13] A. Hidayat. Esprima. http://esprima.org/.
[14] E. Latronico, E. A. Lee, M. Lohstroh, C. Shaver, A. Wasicek, and M. Weber. A vision of swarmlets. *Internet Computing, IEEE*, Jan. 2015.
[15] Q. H. Mahmoud, W. Dobosiewicz, and D. Swayne. Redesigning introductory computer programming with HTML, JavaScript, and Java. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '04, pages 120–124, New York, NY, USA, 2004. ACM.
[16] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, Mar. 2008.
[17] T. Mikkonen and A. Taivalsaari. Using JavaScript as a real programming language. Technical report, Mountain View, CA, USA, 2007.
[18] OpenJDK. Nashorn. http://openjdk.java.net/projects/nashorn/.
[19] Pur3 Ltd. Espruino. http://www.espruino.com/.
[20] D. Reed. Rethinking CS0 with JavaScript. In *Proceedings of the Thirty-second SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '01, pages 100–104, New York, NY, USA, 2001. ACM.
[21] Z. Shelby, K. Hartke, and C. Bormann. The Constrained Application Protocol (CoAP). RFC 7252 (Proposed Standard), June 2014.