2018 IEEE International Conference on Industrial Internet (ICII)

A Modular and Adaptive Architecture for Building Applications with Connected Devices

Pat Pannuto and Prabal Dutta
Electrical Engineering and Computer Sciences
University of California, Berkeley
{ppannuto,prabal}@berkeley.edu

Wenpeng Wang and Bradford Campbell
Computer Science
University of Virginia
{ww2cg,bradjc}@virginia.edu

***Abstract*—**
Smart and connected devices offer enormous potential to enable context-aware, localized, and multi-device orchestrations that could substantially increase the reach and utility of computing. The growth of these applications has been hampered, however, as devices, their data, and their control have been largely sequestered to their own vendor-specific APIs, clouds, and applications—a largely stove-piped state of affairs. Where barriers between devices have been pierced, the connections often occur between vendor clouds, affecting the latency, privacy, and reliability of the original application, while simultaneously increasing complexity. Locally executing applications have not materialized as devices with incompatible communication protocols, inconsistent APIs, and incongruent data models rarely communicate. We claim that what is needed to unlock the application potential is an architecture tailored to facilitating applications composed of networked devices.

Our proposed architecture addresses this by providing a port-based abstraction for devices using a small wrapper layer. This device abstraction provides a consistent view of devices, and embeddable runtimes provide existing applications straightforward access to devices. The architecture also supports device discovery, shared interfaces between devices, and an application specification interface that promotes creating device-agnostic applications capable of operating even when devices change. We demonstrate the efficacy of our architecture with two application case studies that highlight the abstraction layers between applications and devices and employ the embeddability of our system to add new functionality to existing systems.

I. Introduction

Low power sensing systems, wearables, smart devices, and other connected devices—both from the research community and from the commercial realm—have traditionally formed homogeneous networks of identical devices or devices from the same origin [13], [14], [29], [32]. This homogeneity does not prevent these devices from achieving their intended goals, but typically they use applications that are single purpose or directed by the original vendor, often using a vendor-specific mobile application, vendor-supplied cloudlet, or cloud service. While this has been a successful model, as devices become more prevalent, implementation techniques become better understood, and standards become widely implemented, a shift is beginning to emerge: from building *devices* to building *applications*.

The motivation behind the shift is clear: opening lines of communication across vendor boundaries will enable richer experiences and applications than are currently possible. The beginnings of this trend are materializing as a range of devices, such as smart deadbolts, washing machines, and lighting, advertise compatibility with Nest [3] to immediately increase their functionality and appeal to customers who already own the thermostat [9]. Other possible applications motivate this as well and are infeasible without cross-vendor device interaction. One can envision applications such as automatic machinery lockout based on localization of employees, end-to-end asset environmental metrics and management, or even a mixed-reality view of assembly lines that overlays health and status.

While conceiving these applications can be straightforward, reasoning about how to build them is less so and raises many challenges and requirements. 1) There must exist a common methodology for interacting with devices. Conceptually, devices often provide simple functionality ("how many steps have I taken"), but programmatically expressing this is more difficult. 2) Device communication protocols vary, and some interface logic will be required. It should be minimized, however, and not require a custom adapter between each pair of devices. 3) The applications must execute inside of some context. Browser-based web applications, for instance, execute mainly in the cloud, but it is less clear where low power device-to-device applications should execute. 4) While this genre of applications is inherently local, the cloud may still be useful, though it raises questions concerning reliability and privacy.

Current approaches designed to work with today's devices suffer from many of these issues. Application support from manufacturers is often tied to vendor gateways or clouds, meaning that device data for applications must be routed through them. This leads to applications which are cumbersome to scale, often exhibit high execution latency due to round trips to the cloud, reduce privacy by routing all data through the cloud, and are exposed to failure due to network outages. So-called vendor-agnostic systems such as IFTTT [1] and Zapier [10] connect vendor clouds together, still relying on each vendor's vertical device-to-cloud silo, and expose device information to yet another cloud service. Newer device-level or locally executing approaches, such as AllJoyn [11] and the Thing System [7], require complete participation from devices or are monolithic approaches that support limited execution scenarios. In the enterprise context, this absence of interoperability manifests as limitations on real options for IT investment and has slowed adoption [25].

978-1-5386-7771-1/18/$31.00 ©2018 IEEE
DOI 10.1109/ICII.2018.00009

We advocate for a new architecture for structuring, organizing, and implementing applications of connected devices. This architecture attempts to address the shortfalls of current approaches by providing a core "kernel" abstraction around devices and focusing on facilitating interactions between devices. Each device provides and extends common interfaces for device classes, so, for instance, all thermostats can be accessed with the same base API. We carefully design the abstraction layers to encourage flexibility when devices change, execution environments change, or applications change.

The architecture is structured in three main layers: device representation, device communication abstraction, and application interface. We simplify devices and encourage consistency by modeling devices as a collection of ports and a bundle of internal state. Device control and feedback occurs through the ports, and the cache of device state simplifies application design. This model is implemented for devices with small snippets of code that abstract the low-level specifics of a device. Abstraction code is designed to be transparently embedded inside applications. A system library handles finding, fetching, loading, and executing the snippets, and allow the snippets to execute in a variety of contexts: on a user's device, in a cloudlet, in the cloud, or directly on a device. Applications can be described independent of the mechanism they execute on top of. Additionally, applications can be created independently of specific devices, by relying on shared and well-specified device interfaces for classes of common devices.

The balance between structure and flexibility of the architecture makes it well suited for building applications of connected devices. The known interfaces, format and content of the code snippets, and system library interface provide structure that expands application functionality while eliminating duplicated code, encouraging rapid development of device-specific code snippets, and simplifying application development. Other aspects of the system, including the ability to write arbitrary code to interact with device-specific APIs and protocols, and to execute the snippets on multiple platforms, provide the adaptation layer to handle the realities of this application space. Expecting all devices to natively conform to the same protocols and APIs, relying on a common gateway to communicate with all devices, or restricting application logic to the cloud are all approaches that are unlikely to be successful. We argue a successful architecture must be able to adapt to present and future variability while providing a consistent framework to build applications against.

To test and explore our architecture we provide a prototype implementation of each of the layers. As every device requires a code wrapper, we make an extensive effort to minimize the burden of authoring these snippets. Further, our implementation includes supporting infrastructure for hosting, grouping, displaying, and debugging device specific code snippets. We provide the system library that implements several execution environments for running the snippets. We close with two case studies, examining two applications that leverage different aspects of our infrastructure to provide concrete examples of the advantages provided by our architecture. The first creates localized responsive lighting using power meter sensors. The application is specified once and implemented twice to demonstrate flexibility as devices change. The second shows how the infrastructure can be integrated into existing applications and how device discovery and generic device ports ease the burden of writing portable applications.

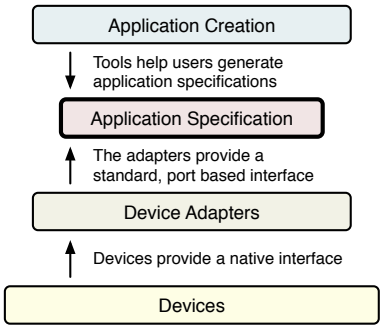


Fig. 1: Layered application architecture. An application specification describes a means of achieving a desired behavior. Specifications are runnable, instantiatable constructs and achieve the end goal of realizing devices interacting and operating in an intelligent manner. The adapter layer provides a means of abstracting the functional behavior of devices from details of their exact interfaces and implementations. Teasing out application creation from specification decouples the means of achieving desired behavior from an abstract concept of what should happen. We advocate for these abstractions as the right balance of modularity and expressivity for creating meaningful applications with connected devices.

II. Overview

Today, low power and locally networked devices employ a range of interaction paradigms to enable user control, cloud-based control, and device-to-device communication. The breadth of patterns is an artifact of enabling a wide range of applications that leverage different aspects of local gateways, user devices, cloud services, and end devices. A common theme throughout the patterns is the presence of an adapter, or some gateway or software library that implements the communication protocol supported by a given device. These adapters are often built into smartphone apps, cloud services, and gateways in a device and application specific fashion.

Our architecture supports the range of application-enabling interaction patterns but focuses on defining abstractions between the relevant components that allow for reusable adapters. We claim that a well-specified abstraction layer for devices will enable modular and reusable adapters that are critical for building meaningful applications with connected devices. These adapters can then run in a range of contexts depending on what best suits the environment and application: on a smartphone, on a laptop, in a local cloudlet, in a remote cloud, or even directly on a device itself.

The architecture's main purpose is to support and enable applications. Our system is a collection of support structures

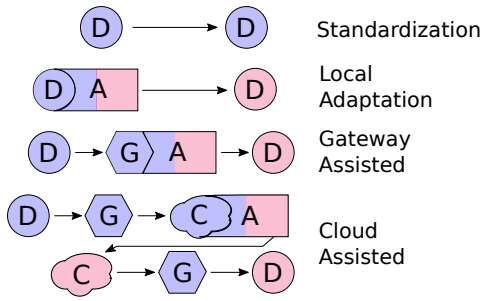


Fig. 2: Heterogeneous interactions require adaptations. Applications involving communication between heterogeneous devices require either global standardization or an adaptation layer, on the source device, the destination device, a local gateway (cloudlet), or in the cloud. Our architecture simplifies these interaction patterns by providing standardization for these adaptations, allowing the adaptation of seamlessly move to an appropriate execution context based on device or network capability and application requirements.

that both aid creating portable applications and provide a framework for new devices to easily integrate into applications. The abstractions provided by the architecture, shown in Figure 1, allow application descriptions to be decoupled from application implementations, creating a clean separation between the myriad possible methods for constructing applications: visual block editors, natural language processors, or online learning algorithms, for example, and the abundance of possible implementation strategies: directly linking devices, coordinating with a cloudlet, or distributing computation on available resources, for example. The ultimate system goal is to facilitate authoring applications the are constrained to a specific behavior but not to currently available resources or a specific set of devices.

III. Background & Related Work

Our system builds on the foundation laid out by Latronico et al. who first introduced *accessors*, an actor-model representation of networked devices [24]. We adopt the principle of their accessor building block, although we modify the implementation to allow addressing some of the open questions from the original accessor work, including search and discovery of accessors, versioning, static analysis, and automatic resource management.

A. Device Abstractions

For heterogeneous devices to interoperate, either all networked devices must adhere to a global (future-aware!) application layer standard or more realistically somewhere some adaptation code must execute. Figure 2 shows the typical placement of these adaptations, at one or both of the end devices, on a supporting gateway or cloudlet, or in the cloud. There are several competing device abstraction strategies that impose different expectations on where these adaptations are implemented and as a consequence what types of device-to-device interactions are possible. We ultimately select and extend accessors as they afford the most flexibility for the instantiation of adaptations and device interaction patterns.

The Thing System provides a web server that presents a common GUI for interacting with a range of devices [7]. To integrate devices, a dedicated Thing System server runs JavaScript libraries for each device, on a local cloudlet. Alternatively, devices can implement the Thing Sensor Reporting Protocol [8] or the Simple Thing Protocol [6] to directly connect to the central server, or Steward. In all cases, however, the goal is to integrate the device into the Steward. The Steward federates all communication and is responsible for all device discovery.

AllJoyn pushes towards a more distributed approach [11]. Conceptually, AllJoyn takes the established DBus protocol for inter-process communication and extends it to inter-device communication. As a message bus, AllJoyn does enable direct device-to-device discovery and communication, however, the system requires buy-in from all devices on the bus, and only devices on the same logical bus can communicate. While nodes can proxy, scaling AllJoyn in practice remains an open question.

MTConnect is an open standard specially designed to meet the requirements of the manufacturing industry [2]. It can enhance the data acquisition capabilities from equipment in manufacturing facilities. It can also expand the use of data driven decision making in manufacturing operations and aims to shift software applications atop manufacturing equipment towards a plug-and-play environment, reducing the integration cost of software systems. However, one major limitation for MTConnect is the read-only method allowing monitoring of the asset [12], which provides the manufacture no capability for control of the asset. Another limitation to MTConnect is the non-adoption of PMC data and tooling in the specification. The analysis of PMC and tooling data is highly critical to various visualization and optimization processes. Ultimately, MTConnect's structure limits its scope for use in practice.

OPC Unified Architecture (OPC-UA) is a platform independent service-oriented architecture that provides machine to machine communication for industrial automation [5]. It focuses on communicating with industrial equipment and systems for data collection and control. The architecture supports multi-platform implementations, scalability, multi-threaded operation, security, timing controls, and chunking of big datagrams. However, the major drawback of the architecture is the resulting complexity that makes it difficult to develop reusable client applications, those that are independent of the specific implementation of each server. This suggests that OPC-UA may not achieve its complete interopability promise in practice. This can be seen in factories and infrastructures where each project integrates independently using various PLC technologies delivered with differing and limited implementations of OPC-UA.

From the research community, the BOSS building control system [18] leverages sMAP [17] as its principle abstraction mechanism. In this design, the burden for adhering to the system abstraction interface is pushed onto the participating devices or dedicated, pre-deployed proxy servers translating on behalf of inflexible devices. HomeOS [21] takes the opposite approach, with a centralized "Device Functionality Layer" that coalesces devices onto a single "PC" abstraction similar to the Steward from the Thing System.

B. The Accessor Abstraction

In contrast to these designs, the accessor abstraction shim acts as a standalone kernel. This makes the decision of where and how to integrate much more flexible. Applications can instantiate adapters in their native runtime to communicate directly with devices that have no knowledge of our ecosystem. Local cloudlets or intelligent gateways can run shims to act as proxies for devices. This differentiation is key to the flexibility facilitated by our design. Pushing abstractions to the device like AllJoyn and sMAP requires the ability to either change code on devices or instantiate proxies on their behalf. Placing abstractions in the cloud like the Thing System or IFTTT necessarily centralizes device control and information. Enforcing standards like MTConnect or OPC-UA requires buy-in and limits opportunities for revision or enhancement. By creating a shim layer that exposes a clean interface not tied to any application framework, our model supports significantly greater flexibility than previous device connectivity frameworks.

To provide a consistent view of a wide range of devices, accessors model all device interactions as a write or subscription to well-defined ports or (newly) a read to a device attribute. Device ports are defined by their direction characteristics relative to the device itself, either "input" or "observe", or both. Ports that support the input direction allow for control of the device, and observe ports allow the device to generate events. Attributes allow for the state of the device to be queried.

This model allows for a clean representation of a range of devices. The ports are designed for push operation, where computation and control can occur spontaneously in reaction to events. This facilitates real-time applications and interactions. Human interface devices, environmental monitors, event detection sensors, and other event-generating devices can all publish data on their output ports at appropriate times. These events can then feed to other devices on their inputs, and cause applications to execute when there is useful computation and communication to be done. Attributes are designed for a pull model and allow for queries on devices. Asking for the current temperature, if the light is on, or what the current power draw of a house is are all operations that map to reading attributes.

Attributes are an extension not included in the original accessor design. Broadening the model beyond just ports by adding attributes makes it better suited for characterizing the breadth of devices. Implementing queries with ports is cumbersome for accessor authors, as it requires an input port to ask the query, and an output port for the response. Decoupling this is counterintuitive, and the environment the device model is executing in must determine how to send the output to the requester, which is often difficult to implement. Instead, allowing device models to expose attributes which can simply be read greatly simplifies interacting with devices in practice.

IV. Design

We describe the design of our application architecture, Figure 3, starting from conceptual abstractions for devices, adding the components required to enable applications, and building to our comprehensive system for multi-device applications.

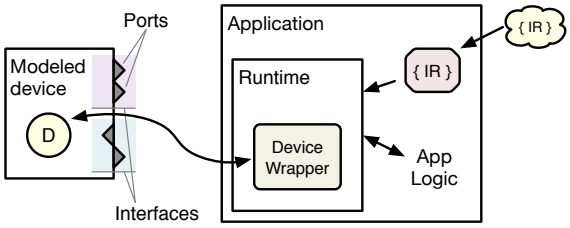


Fig. 3: Architecture overview. Devices are modeled abstractly using a port and attribute representation, and these are logically grouped into interfaces. An application leverages the device's model by fetching the intermediate representation of a device-specific wrapper. The wrapper executes inside of a runtime which is embedded inside of the application. This architecture allows applications to be structured around a conceptual view of devices and to push the complexities and specifics of interacting with devices into the wrapper layer.

A. Device Interfaces and Interface Ontology

Accessors allow ports and attributes to be specified individually on a device-by-device basis, however, this provides little standardization between devices. To allow for better grouping of devices, ports and attributes are grouped into interfaces, and devices can provide interfaces instead of individual ports. This difference allows applications that are using this system to choose and program against devices based on the interfaces they provide rather than the specific device. Applications or their runtimes can then fanout commands to all matching devices instead of needing to enumerate all devices explicitly.

Defining a standard set of interfaces requires defining an ontology for interfaces and their ports. The design of this ontology is critical for the utility and usability of our proposed design as these interfaces are the principle mechanism that application authors will interact with. Additionally, the interfaces are only useful if they are capable of encapsulating the functionality of multiple devices. If every device has a custom interface, then the system has essentially been reduced to writing applications for specific devices again.

To balance these design goals, we principally define our interface tree as a wide, shallow structure. While simple single-purpose devices may fall neatly in one interface, many devices will cover multiple interfaces. The shallow tree maximizes the flexibility of interfaces by maximizing the number of devices that will fit into a given top-level category. Depth in the tree allows for specifying more advances features, e.g. all smart lights can turn on or off but only some can change color.

To simplify the burden of mapping devices to all available interfaces, we design interfaces to be inheritable to allow for sharing ports between interfaces. Sharing ports allows for more advanced device grouping without creating excess ports. As an example, consider two likely interfaces, `/onoff` for devices that can be turned on and off, and `/lighting/light` for devices which are lights. To allow lights to be both grouped in the lighting group and in the group of all devices that can be power toggled, the `/lighting/light` interface

Well-known methods that authors implement

`init`	Called when the wrapper is loaded
`cleanup`	Called when the wrapper is unloaded
`<Port>.input`	Called when an input port is written to
`<Port>.output`	Called when an attribute is read
`<Port>.subscribe`	Called to register a callback to subscribe to this port

Framework `init` API

`provide_interface`	Implement a standard interface
`create_port`	Create device-specific port
`create_attribute`	Create device-specific attribute

Framework Runtime API

`get_parameter`	Get a configuration parameter
`load_library`	Third-party library support
`load_dependency`	Load a sub-wrapper

Built-in libraries

`{a}rt.{log,time,color,encode,http,coap,socket,amqp,mqtt,...}`

TABLE I: System framework. To minimize learning overhead and maximize code reuse, we carefully target a minimum framework API. Authors need only declare ports and attributes and support their actions. Our implementation provides an extensive runtime (`rt`) library and a `load_library` mechanism for third-party extensions.

inherits from the the `/onoff` interface. Each light that provides the `/lighting/light` implicitly provides the `/onoff` interface. This eliminates the need to author both a `/onoff/Power` port and a `/lighting/light/Power` port, while permitting applications to transparently use either. The inheritance mechanism supports multiple inheritance as interfaces can extend multiple other interfaces.

Beyond the standard interfaces, devices may also choose to add additional ports. These device-specific ports provide a mechanism for device or vendor specific extensions. With sufficient popularity, these custom ports provide a pathway for the introduction of new standard interfaces.

B. Device Wrappers

A device wrapper is our realization of the accessor design paradigm. Device wrappers are designed to be easy to author, and our system enables this with a simple framework for these wrappers. In contrast to the original accessor design, which requires authors to synthesize both the specification and device-specific code, we design wrappers to automatically infer device specifications from the code that implements the wrapper. Wrappers are authored in JavaScript, chosen because it has been shown to be both quickly accessible to novice programmers [26], [31] and viable for advanced applications [27]. The architecture itself does not require that the wrappers be in JavaScript, however, and other languages could be used in the future.

Each device wrapper includes function implementations for initialization, each port, and each attribute. The `init` method is called first to specify the interfaces and device-specific ports and attributes the device provides, as well as to run any device-specific setup or connection logic. For each port and attribute, the wrapper includes functions for when they are read from or written to. In contrast to more managed systems such as

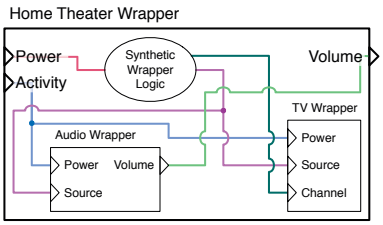


Fig. 4: Device wrapper for a synthetic "Home Theater" device. By including the wrappers for an audio device and a television with some additional logic, a home theater interface can be created without reimplementing the logic for communicating with the contained devices. This example demonstrates how a synthetic device can be created to provide a useful interface.

AllJoyn or the Thing System that require authors to integrate devices into the framework, these function implementations are the only author responsibilities. Table I summarizes the complete framework presented to authors.

1) Specialization: A wrapper is designed to encompass a specific device or product, but often it is desirable to have specialization for individual instances of a given device, such as the bridge address of a smart bulb. We introduce the concept of parameters for accessors, where device-specific parameters are coupled with the generic wrapper to create a device instance, which is a unique adapter for a specific device. Decoupling parameters from ports disentangles writes intended to modify the underlying devices from writes intended to modify the device adapter. This permits devices to advertise a generic interface for control with lightweight parameters, as opposed to requiring either a centralized database of all device instances or requiring adapter instantiators to ascertain device-specific parameters via an unknown third-party mechanism.

2) Synthetic Devices: Wrappers are not limited to a one-to-one ratio with devices. Often, higher level interfaces may be more useful than interfaces directly on top of devices. For instance, consider a "Home Theater" wrapper, as shown in Figure 4, which abstracts multiple audio-visual devices into a single device. This wrapper is created by adding a layer of control logic above the audio device and television wrappers and correctly mapping ports. Instead of reimplementing the audio and TV wrappers, they can be included directly in the home theater wrapper. This method allows the architecture to support synthetic devices in cases where higher-level interfaces are more suitable than device-level interfaces.

C. Wrapper Intermediate Representation

While we emphasize minimizing overhead for device wrapper creators, parsing source code to ascertain device capabilities burdens developers who wish to use the wrappers in an application. To mitigate this issue, our design adds an intermediate representation that creates a machine (and human) parsable document that both includes the wrapper code and details the ports, attributes, interfaces, and other properties of the device.

Furthermore, this intermediate representation is generated by a compiler that is able to validate the correctness of device wrappers and detect and explain common errors to wrapper creators. This compilation step additionally simplifies the design of downstream users of the device wrapper as they can omit many correctness checks. This intermediate representation allows application environments to easily understand commonalities between devices, properties of each port, and meta information about devices that may be useful to an application.

D. Runtimes

Once a wrapper exists that describes how to interact with a device, it must actually execute in some context. Runtimes provide execution environments that run the code snippets and provide a context-specific interface to the associated device. The abstraction layer from the intermediate representation allows for many runtimes across many programming languages and execution environments. The objective of each runtime is to present the device wrapper in a native way for the given environment that a developer would find natural.

Runtimes also provide the device wrapper a standard library to use when communicating with devices. This provides support for HTTP, CoAP, UDP, BLE, and other communication protocols. The intermediate representation includes a list of all libraries used, allowing the runtime to ensure that all required resources are available in advance.

E. Device Discovery

For devices to be included in applications they must be enumerated. Manual device registration may be effective for small numbers of devices or for accurately grouping devices by location or user, but in other contexts manually listing devices may be infeasible. Dynamic applications, for instance, that intend to operate with a class of devices present in any space where the application is run may wish to discover devices dynamically. Also, devices themselves may wish to discover other nearby devices that provide a particular service the original device cannot provide.

Device discovery and the corresponding wrapper discovery are complimentary problems. Existing discovery technologies, such as Zeroconf (mDNS+DNS-SD) and UPnP [15], [16], [34], announce the presence of a device and possibly some services exposed by the device. Advertising the wrapper instead announces both the presence of a device and its services based on the interfaces it provides. Critically, it also announces *how* to interact with the device. Our architecture leverages existing device discovery protocols for advertising devices supported by this architecture to take advantage of existing tools.

F. Applications

Once devices can be modeled, grouped, described, discovered, and accessed, they can be connected, in an abstract sense, to create interesting applications. The main property of applications within the architecture is that application specification is independent of application implementation, as illustrated in Figure 5. That is, applications can be described in a generic way and that application description in conjunction with locally available devices and computational resources can later be executed. This well-defined abstraction layer between what an application should do and how it executes is critical for developing a successful ecosystem around this architecture. Above the abstraction layer, there may be many methods for describing applications, such as block diagrams, converting speech to commands, writing pseudo-code, and interpreting existing actions to automatically create applications. Allowing these methods to be explored without including the burden of application execution aids application creation. Below the abstraction layer, how an application executes may change as devices are added, updated, or removed, connectivity options change, or local computational resources change. Decoupling the application specification from immediately available resources allows the system to adapt over time and to run the application in the most reliable and efficient manner.

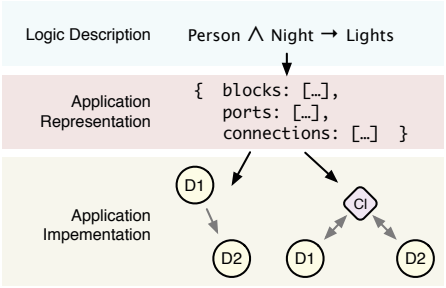


Fig. 5: Application creation structure. The abstraction layers in the architecture allow applications to be considered at three levels. The first level specifies the conceptual representation of an application, and this example uses a logic statement. The next level is a standard representation of the application that can be instantiated. The final level are possible implementations.

At a high level, applications in our architecture are described by connecting the ports of devices to each other. For instance, a simple conceptual application may be "when I enter my office turn the air conditioner on" and a very simple version of this application might be described as connecting the "Door opened" output of a door sensor to the "Enable" input port of the air conditioner. In practice, applications will be more intricate than this simple example and will require additional logic between devices, data processing at various points, external data inputs, and other features beyond just devices. Despite this, we argue, applications are fundamentally composed of interconnected device ports with some possible intermediate logic.

G. Standardization Vision

Device wrappers and their runtimes are necessary to create standard interfaces and reusable applications out of widely varying devices. However, a standard for device communication may emerge and become prevalent, particularly if interoperability becomes commercially desirable or advantageous. Based on our architecture, we advocate for a a REST style interface based on CoAP [33]. This meshes with our architecture for

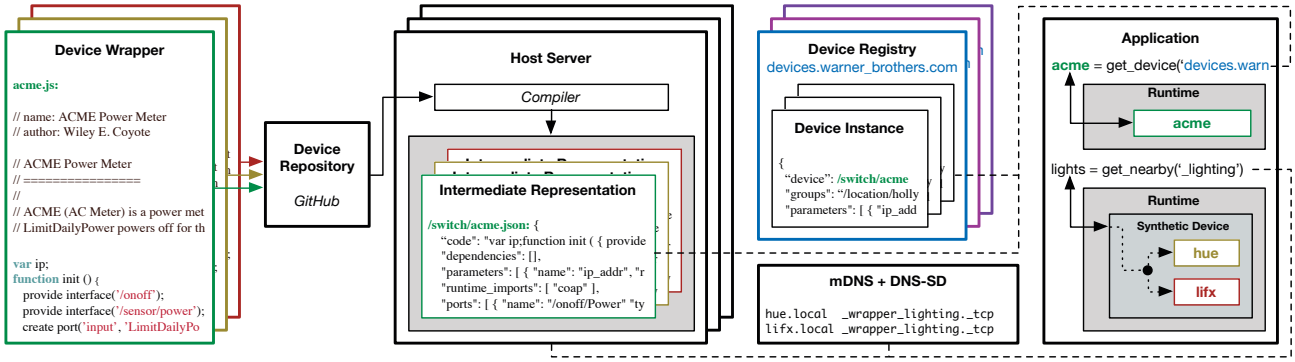


Fig. 6: Implementation overview. A device wrapper is a JavaScript file that imperatively describes the interfaces and ports of a device and provides code for using them. Users author these device wrappers and commit them to a single, global repository. They are then compiled into an intermediate representation that validates correctness and extracts rich metadata for automated tools to leverage. A Host Server is a generic server that hosts compiled wrappers and can be queried for available wrappers. Device instances—the combination of a generic wrapper and parameters for a specific instance of a device—integrate with Zeroconf (mDNS + DNS-SD) for automated discovery or are served from static, pre-configured device registries. Applications load instances into a runtime, a compatibility library layer between the native application execution environment (e.g. Python) and the framework that wrappers are programmed against.

two reasons. First, URIs map well to devices, interfaces, and ports. As an example, a controllable light might have the URI "http://192.168.1.2/lighting/light/Power" which is the URI for turning the light on and off inside of the "lighting/light" interface. Second, the CoAP REST commands map to port types well. GET corresponds to an attribute, POST to an input port, and GET with the observe option corresponds to subscribing to a port. Finally, CoAP is designed to be sufficiently lightweight to be viable for resource-constrained devices, which represent a growing proportion of intelligent, networked devices. While our architecture is explicitly designed so as not to require devices to support any eventual standard, it is also aims to encourage devices themselves to buy in, and drive towards a system with minimal to no need for device wrappers.

If this standard, or a different similar one based on ports, did emerge, our architecture is designed to gracefully support the change without requiring changes to applications. The runtime layer simply drops the device wrapper and maps application interactions directly on to the standard interface.

V. Implementation

As a teaser to motivate our ecosystem, we present here in full a Python applet that turns off the lights when everyone leaves the room:

```
1 # /usr/bin/env python
2 import device_runtime as rt
3 room = rt.get_nearby('/occupancy', limit=1)
4 lights = rt.get_by_location(room['_location'], '/lighting')
5 room.Empty.subscribe(lambda : lights.Power = False)
```

In line 3 a synthetic device made up of all nearby devices that implement the `/occupancy` interface is created automatically by the runtime. The `get_nearby` constructor adds `_location` metadata, which is used in line 4 to ask the runtime to create another synthetic device that collects all of the nearby devices that implement the `/lighting` interface. Finally, line 5 attaches a function to the `Empty` output port of the `room`, which writes to the `Power` port of the `lights` whenever a new event arrives.

The remainder of this section presents our implementation from bottom-up, beginning with how the device wrappers are developed and compiled. We next consider discovery and distribution, the infrastructure that powers the `get_nearby` and `get_by_location` methods. Finally, we explore the mechanics of implementing a runtime and differing mechanisms for bridging runtimes and native execution environments. Figure 6 gives an overview of how all the components of the system fit together.

A. A Wrapper and IR

Recall that a wrapper is simply JavaScript code that is then compiled into an intermediate representation (IR), which is a JSON document containing the wrapper code and parsed metadata. This compilation also ensures adherence to declared interfaces and performs parameter validation and other static checks for correctness. Figure 7 shows an example of a complete wrapper and part of its compiled IR.

B. Host Server

This host server provides a repository of available device wrappers that other pieces of the infrastructure use to find and download the device wrappers. Currently, we implement this as a public GitHub repository and a webserver that keeps an up-to-date copy of each device IR available. The webserver also presents a browsable view of the available device wrappers, as well as any compilation errors that could prevent wrappers from working correctly, as shown in Figure 8. A suitable management and distribution policy for the repository is left to future work, but we are inspired by systems such as Homebrew [23] that succeed with a similar model.

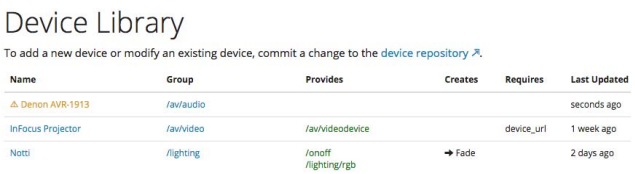
```
// name: ACME Power Meter
// author: Wiley E. Coyote

// ACME Power Meter
// ================
//
// ACME is a power meter with a relay for load shedding.
// LimitDailyPower powers off if limit is exceeded.

var ip;

function init () {
  provide_interface('/onoff');
  provide_interface('/sensor/power');
  create_port('input', 'LimitDailyPower',
        {type: 'integer', min: 100, units: 'watthour'});
  ip = get_parameter('ip_addr');
}

// The asynchronous runtime (art) returns promises
onoff.Power.input = async function (state) {
  await art.coap.post('coap://['+ip+']/Power/'+state);
}

onoff.Power.read = async function () {
  return await art.coap.get('coap://['+ip+']/Power/');
}

sensor.power.Power.subscribe = async function (cb) {
  await art.coap.observe('coap://['+ip+']/Watts', cb);
}

// Create a virtual port required by the interface
sensor.power.PowerChange.subscribe = async function (cb) {
  var threshold = get_parameter('power_threshold', 10);
  await sensor.power.Power.subscribe(function (watts) {
    var power = onoff.Power.read();
    if ((watts < threshold) && power) {
      cb(false);
    } else if ((watts > threshold) && !power) {
      cb(true);
    }  });
}

LimitDailyPower.input = async function (l) {
  await art.coap.post('coap://['+ip+']/DailyLimit'+l);
}
```

(a) ACME Wrapper: `acme.js`

```
{
    "code": "var ip;function init () \{ <snipped>",
    "dependencies": [],
    "parameters": [ {
            "name": "ip_addr"
            "required": true,          (line 17: no default value)
        }, {
            "name": "power_threshold",
            "required": false,         (line 34: provides default value)
            "default": "10",
        } ],
    "runtime_imports": [ "coap" ],     (lines 21,25,29: use art.coap)
    "ports": [ {
            "name": "/onoff/Power"     (/onoff interface definition)
            "type": "bool",
            "display_name": "Power",
            "direction": "input",
        }, <snipped>
    ],
    "implements": [
        {                              (line 13: provides /onoff)
            "interface": "/onoff",
            "provides": [ [ "/onoff.Power", "PowerControl" ] ],
            "ports": [ "onoff.Power" ]
        }, <snipped>
    ],
    <snipped>
}
```

(b) ACME Wrapper IR: `acme.json`

Fig. 7: Wrapper example. The ACME Power Meter is an AC meter and load-controlling switch. Additionally, it adds a novel vendor feature, a daily power cutoff. It does not natively report when the attached load switches on or off, so the wrapper emulates the required `PowerChange` port. The compiled wrapper embeds the original code as an element. It resolves interfaces to list all ports and identifies all aliased interfaces. The IR also extracts all parameters and external dependencies. This verbose, compiled form enables powerful queries across wrappers, permits runtimes to ensure all parameters are specified and dependencies satisfied, and facilitates the creation of on-demand user interfaces.



(a) Library of devices.

(b) Wrapper compilation error.

Fig. 8: Host Server. The host server creates a web interface for exploring the wrappers in the repository. Users can find wrappers by device name, interfaces provided, or other metadata. For wrappers with compilation errors, the host server provides a clear explanation of the error. Additionally, the Notti smart light [35] gives an example of a device with a custom port in addition to the standard interface, as this light adds a custom "fade out" feature.

```
HOST          SERVICE                              PORT
acme.local    _wrapper_sensor_power._udp           5683
   TXT: wrapper_path=/sensor/meter/ACME

hue.local     _wrapper_lighting_hue._tcp           4999
   TXT: wrapper_path=/lighting/hue/huesingle
        username=MyHueUser
```

Fig. 9: DNS Service Discovery records for devices with wrappers. Interfaces are advertised as services with the service type marked by "_wrapper". The DNS infrastructure handles naming the device and port, and the TXT field is used for the wrapper path and parameters.

C. Device Discovery

We extend the Multicast DNS (mDNS) [16] and DNS Service Discovery (DNS-SD) [15] discovery services with a protocol to enable clients to find devices that have available wrappers. With DNS-SD, services that a device provides are encoded as dot separated strings, such as "_ipp._tcp" for a printer.

Devices that have a corresponding wrapper can advertise their interfaces as services. To comply with the DNS-SD protocol, the service is prepended with "_wrapper" to denote that there is a corresponding wrapper. For example, a Hue light may advertise the "_wrapper_lighting_Hue" service. Inside of the TXT record in the service advertisement, the light includes the key "wrapper_path" which allows the application to fetch the correct wrapper. Any other parameters the device requires are also included in the TXT section. Figure 9 shows the records devices in our architecture may advertise.

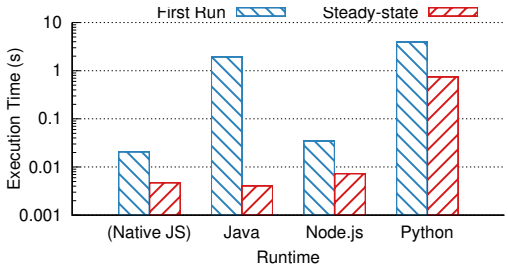


Fig. 10: Runtime overhead. To estimate runtime overhead, we run a loop of HTTP requests to `localhost` using wrappers from various runtimes compared against an equivalent native Node.js app. Running wrapper JavaScript code in non-JavaScript-based runtimes requires an embedded a JavaScript engine (Java) or communicating with an external JavaScript engine (Python). For embedded runtimes (Java), loading the JavaScript engine adds a one-time warmup penalty. For the non-embedded (Python) case, inter-process communication adds significant overhead to the continued execution. In steady-state, requests from the Java-based runtime actually run slightly faster than the native Node.js app.

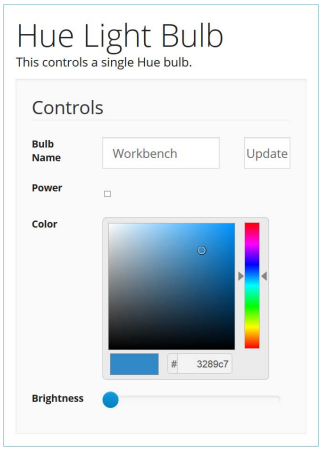


Fig. 11: Auto-generated GUI for a Phillips Hue. The web runtime automatically generates device GUIs from devices. Advanced UI elements are inferred from port types. The `Color` port renders as a color picker because the port is of type `color`. The `Brightness` port renders as a slider because it is an `integer` type port with a `min` and `max`. This entire UI element is auto-generated using only the wrapper IR.

D. Runtimes

The responsibility of the runtime is to provide an execution environment for device wrappers, while making each device available as a native object in the host application language.

1) Hosted Runtimes: We implement three runtimes, enabling native application development in Node.js, Java, and Python. The Node.js runtime has native support for executing JavaScript [4]. The Java runtime uses the Nashorn scripting engine to execute JavaScript [28]. Nashorn executes JavaScript directly on the Java virtual machine, facilitating high performance and easing the passing of data between the Java and JavaScript environments. Unfortunately, Nashorn does not yet fully support ECMAScript 6. Traceur is a tool that "transpiles" ECMAScript 6 code to valid ECMAScript 5 code, at the cost of about 30% performance overhead [22]. The Host Server will transpile a wrapper on-demand if an ECMAScript 5 version is requested. Python has no means to directly execute JavaScript, so we use python-bond, a library that bridges Python and an instance of Node.js via RPC calls [20].

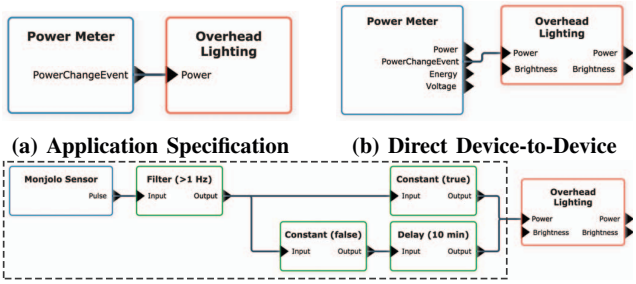
The majority of the runtime code is shared across implementations, executing in the JavaScript context. The IR enables runtimes to pre-load parameters, libraries, and dependencies before executing the device wrapper. Only the top box of functions from Table I need to be shimmed at runtime.

To provide some understanding of the tradeoffs between the different runtimes and execution environments, we benchmark I/O performance across the three runtimes. Figure 10 compares the overhead of the native (Node.js), embedded and transpiled (Java + Nashorn), and remote (Python + python-bond + Node.js) runtimes. We run a small 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. Loading the JavaScript engine imposes a heavy startup cost on the non-native runtimes. For our I/O-heavy workload, however, the runtime overhead of Traceur is unsurprisingly not a large penalty. Indeed, the Java runtime slightly outperforms Node.js for this microbenchmark. The RPC calls in the Python runtime impose a heavier runtime burden. Upon further examination, this effect is amplified by the relatively naïve and inefficient RPC mechanism employed by the python-bond library.

2) Proxy Runtime: We initially attempted a browser-based runtime, but rejected the effort as browsers are too sandboxed of an environment to support many of the communication protocols used by devices. Browsers support only websockets, requiring a support server to proxy other protocols such as UDP or TCP. The browser-enforced same-origin policy prevents support for devices that neglect the `Access-Control-Allow-Origin` header—which only one of the dozen commercial devices we tested set—, requiring a proxy for HTTP requests as well.

Instead, we build an RPC webserver with a HTTP REST API. This maps a `PUT` to `input`, a `GET` to `read`, and uses websockets for subscribing to ports. The webserver uses the metadata from the IR to build a GUI on-demand, like the Hue GUI in Figure 11. Complex UI elements such as color pickers, sliders, and drop-down lists are inferred from port types.

More generally, this RPC server can act as a proxy runtime for any device. This is a powerful step towards a standardization vision, especially with the advent of local cloudlets and intelligent gateways. One could imagine gateways automatically running a proxy runtime for all connected devices. In this way, even if no device ever implemented our standard API, every networked device would transparently adhere to the API.



(a) Application Specification **(b) Direct Device-to-Device**

(c) Cloudlet Based

Fig. 12: Describing an application using a block editor. An application, shown in (a) and conceptually described as "when devices in a space are drawing power, turn on the lights", is implemented in two ways using a block-connecting approach. In (b), the chosen power meter and lights have ports that match the description, allowing a direct connection to satisfy the specification. In (c), a different power meter ("Monjolo") is used that sends pulses proportional to the metered load. The pulse stream must be filtered and fed into a type conversion block ("Constant") to generate an "On" signal. A "Delay" block is used to automatically turn off the lights. As the dashed area generates "PowerChangeEvent"s, the application requirements are now satisfied. Because this application performs additional processing, it is run in a cloudlet. Implementing the same application description in multiple ways demonstrates how the architecture is able to adapt to changing devices.

VI. Case Studies

To demonstrate how our architecture and system implementation help create applications, we implement two device-centric applications. As performance and correctness metrics are difficult to define for this architecture, we do not evaluate in the traditional sense but attempt to explain how the architecture aids creating these applications and others in the future.

A. Responsive Lighting Application

Commercially available "smart lights" surpass conventional lighting by allowing remote control and programmatic access, but often still rely on manual intervention for determining when to illuminate or switch states. Our first application attempts to add automatic control to smart lighting by integrating inputs from other sensors, such as power meters. Conceptually, the application can be specified as:

> When devices in a localized area are drawing power, turn the lights in that area on, and when then devices are not, turn the lights off.

This results in an intuitive and responsive application where lights respond to activity and not direct control.

To create this application we use a browser-based visual block editing interface written on top of jsPlumb [30] to describe how components interact. Figure 12a shows the block representation of the high-level application description. It encompasses two devices, a power meter and the lights. The functionality requires a "PowerChangeEvent" port to be connected to the "Power" port of the lighting. This describes a power meter that, when the attached load turns on or off, will send a message to another device, in this case lighting, to turn it on and off. This depiction is what we consider to be the application description layer, as described in Section IV-F, and relies on the port abstraction to model how the devices should interact.

Given the application description, the system is able to implement the application in two ways. First, as shown in Figure 12b, two devices matched the required ports and were eligible to directly communicate to execute the application. At the system configuration level, the power meter is considered to be able to run the lighting device's wrapper to control its state. In practice, the power meter can be configured to send an event to a particular destination when the load changes state.

The second implementation, shown in Figure 12c, uses nested applications to implement the same functionality, and demonstrates how the system can adapt to device diversity. Instead of the same true power meter, sensing is performed by a rudimentary power meter known as a Monjolo sensor which uses energy-harvesting principles to pulse packets at a rate roughly proportional to the load being metered [19]. This provides sufficient power metering for the application, but due to its energy-harvesting operating principles, cannot determine when the load has turned off. Therefore, the sensor is wrapped with additional logic, shown inside of the dashed box, to add a delay to turn the light back off when the pulses cease, and to do type conversion. This causes the dashed box portion to become its own application with the correct ports to satisfy the power metering requirement of the original application goal. The block editing environment understands that the additional logic requires runtime computation and therefore runs this implementation in a local cloudlet.

The responsive lighting application is enabled by many aspects of the architecture. The application itself is specified in a wholly device-independent manner, facilitating diverse implementations. Devices are modeled with ports, and possible implementation strategies can be easily validated by the block editing tool (or any other system) based on matching ports. The two implementations further highlight the flexibility of the architecture. For the severely resource-limited energy harvesting power meter, a local cloudlet runs a device wrapper to interface with the Monjolo device's Pulse output and then presents an interface that satisfies the remainder of the application. The other case eschews wrappers entirely and demonstrates how a standard for device communication can enable direct device-to-device communication. In this way no additional servers are required to support applications and no third parties ever learn of events in the system. Empirically, occupants in our preliminary deployment quickly adapted to rely on the responsive lighting environment provided by both implementations.

B. Security and Safety on the Factory Floor

A critical issue for manufacturers is to ensure the security and safety of their factories. An application using smart devices may be desirable for this issue, but in industrial settings a

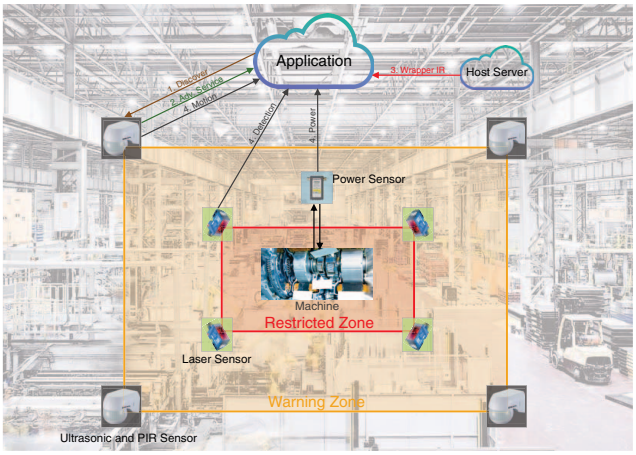


Fig. 13: Factory security and safety application. This app leverages the system infrastructure to dynamically discover peripheral sensors, download the correct wrappers, and execute the application logic on a nearby gateway. As wrappers can be discovered and downloaded dynamically, the application works in other zones or environments without modifications.

key challenge for multi-device applications is interoperability across devices from different vendors. To demonstrate this and to propose a solution enabled by our architecture, we examine a particular example: ensuring worker safety by defining a restricted zone around dangerous equipment. The application can be specified as:

> When the potentially dangerous equipment is not drawing power and sensors detect a worker nearby, turn on the light.
> When the equipment is on and sensors in the warning zone detects motion, turn on a warning light.
> If sensors in the restricted zone detect a worker, shut down the machine.

This requires defining a restricted zones near the equipment, reliably detecting if the equipment is running, and leveraging multiple sensor types to to ensure reliable worker detection

This application has two key challenges: it must minimize the possibility of false positives, and it must be capable of working with different types of sensors from different vendors. The solution provided by our architecture is shown in Figure 13. A power meter monitors the power supply of the machine to determine if the machine is on based on whether it is drawing power. Laser trip sensors monitor the restricted zone, while ultrasonic and PIR sensors monitor the warning zone. Devices that have device wrappers advertise the interfaces they support as services. To find nearby sensors, the application queries for all devices that support the desired services. The application then iterates through those responses and downloads the correct wrappers from the host server. It can then use the intermediate representation to select the right port from the interface and the wrapper to query its value and generate different responses based on the inputs.

The application emphasizes several aspects of the architecture. In this case we make no assumption of the vendors or their interfaces for the hetergeneous array of sensors, breaking the traditional communication boundaries. Coordinating between different types of sensors minimizes the false positive problem that often occurs on single sensors improving the reliability of the system. Moreover, by providing the application the ability to fetch the wrapper on-demand based on the immediately available sensors – in contrast to a statically defined application and sensor configuration –, the same application can execute in multiple zones with different types of sensors without any modification. This makes it easier to divide buildings into zones based on logical divisions rather than specific sensor installations. Furthermore, this security and safety application can be applied to different companies with same needs without requiring customization. For existing buildings with old platforms, this application is isolated and does not intervene with extant systems, thus it will not have any adverse affect on any of the current systems running in the building.

These case studies illustrate two points on the spectrum of device-centric applications, and help demonstrate how our architecture and implementation can directly simplify creating applications in two very different realms. While these examples do not capture the breadth of possible applications, they provide intuition and motivation for the possibilities an application architecture can promote. While previous systems have realized various capabilities presented by this design—device abstraction, cloud-to-cloud interaction, device-to-device interaction, and device discovery—it is the union of these capabilities that marks the key novelty of this new architecture. With support for current devices and the ability to adapt to future devices, this architecture encourages adoption by end users, manufacturers, and application creators, and could finally enable truly modular and adaptive applications.

VII. Discussion

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

A. Authentication

The 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 host server as this does not provide a method for revoking access. A user can cache a wrapper and execute it later, even if the user could not re-request the wrapper from the server. Therefore, authentication must exist between the executing wrapper and the end device. This, however, requires the wrapper to understand the identity of the user and perform the possibly complex authentication procedure itself, burdening the wrapper creator.

A possible solution is to allow the runtime to perform the authentication on behalf of the wrapper and then have it provide the wrapper 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 parameterized. Surveying currently used authentication mechanisms and integrating them into the architecture is left as future work.

B. 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. A management environment 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.

C. Seamless Cloud Interaction

Device wrappers 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 (or equivalently a local gateway/cloudlet) 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.

VIII. Conclusions

Our contribution in this work is a coherent architecture for networked devices. We identify the key abstraction layers between an abstract application concept—"turn the lights off when I leave the room"—and the details of exactly how and when to send what bits to which devices. This starts with defining ports and attributes for devices and modeling all interaction with the device as interactions with this interface. It builds to presenting the interface to applications with device wrappers, or small adapters that encapsulate the device-specific API. These wrappers share interfaces among devices, further raising the level of abstraction when designing applications. It ends with application frameworks that help create and execute applications on top of the networked devices. This architecture facilitates communication between previously incompatible devices to enable the applications that a fully-connected world promises.

IX. Acknowledgments

This work was supported in part by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, and in part by TerraSwarm, an SRC program sponsored by MARCO and DARPA. Additionally, this material is based upon work supported by the National Science Foundation under grant number CNS-1824277 and the NSF/Intel CPS Security Program under grant CNS-1822332.

References

[1] IFTTT. https://ifttt.com/.
[2] MTConnect. https://www.mtconnect.org.
[3] Nest Thermostat. https://store.nest.com/product/thermostat/.
[4] Node.js. http://nodejs.org/.
[5] OPC-UA. https://opcfoundation.org/about/opc-technologies/opc-ua/.
[6] Simple Thing Protocol. http://thethingsystem.com/dev/Simple-Thing-Protocol.html.
[7] The Thing System. http://thethingsystem.com/index.html.
[8] Thing Sensor Reporting Protocol. http://thethingsystem.com/dev/Thing-Sensor-Reporting-Protocol.html.
[9] Works with Nest. https://nest.com/works-with-nest/.
[10] Zapier. https://zapier.com/.
[11] Allseen Alliance. AllJoyn. https://allseenalliance.org.
[12] S. Atluru and A. Deshpande. Data to information: Can mtconnect deliver the promise. *Transactions of NAMRI/SME*, 37:197 – 204, 2009.
[13] Belkin. WeMo Insight Switch. http://www.belkin.com/us/support-product?pid=01t80000003JS3FAAW, 2014. Part #: F7C029fc.
[14] Y. Cheng, X. Li, Z. Li, S. Jiang, Y. Li, J. Jia, and X. Jiang. AirCloud: A cloud-based air-quality monitoring system for everyone. SenSys'14.
[15] S. Cheshire and M. Krochmal. DNS-Based Service Discovery. RFC 6763 (Proposed Standard), Feb. 2013.
[16] S. Cheshire and M. Krochmal. Multicast DNS. RFC 6762 (Proposed Standard), Feb. 2013.
[17] S. Dawson-Haggerty, X. Jiang, G. Tolle, J. Ortiz, and D. Culler. sMAP: A simple measurement and actuation profile for physical information. SenSys'10, pages 197–210, 2010.
[18] S. Dawson-Haggerty, A. Krioukov, J. Taneja, S. Karandikar, G. Fierro, N. Kitaev, and D. Culler. BOSS: Building operating system services. NSDI'13. USENIX, 2013.
[19] S. DeBruin, B. Campbell, and P. Dutta. Monjolo: An energy-harvesting energy meter architecture. SenSys'13, 2013.
[20] Y. D'Elia. Python Bond. www.thregr.org/~wavexx/software/python-bond.
[21] C. Dixon, R. Mahajan, S. Agarwal, A. Brush, B. Lee, S. Saroiu, and P. Bahl. An operating system for the home. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 337–352, San Jose, CA, 2012. USENIX.
[22] Google. Traceur compiler. https://github.com/google/traceur-compiler.
[23] Homebrew. The missing package manager for OS X. http://brew.sh.
[24] E. Latronico, E. A. Lee, M. Lohstroh, C. Shaver, A. Wasicek, and M. Weber. A vision of Swarmlets. *Internet Computing, IEEE*, 2015.
[25] I. Lee and K. Lee. The Internet of Things (IoT): Applications, investments, and challenges for enterprises. *Business Horizons*, 58(4):431 – 440, 2015.
[26] Q. H. Mahmoud, W. Dobosiewicz, and D. Swayne. Redesigning introductory computer programming with HTML, JavaScript, and Java. SIGCSE'04, 2004.
[27] T. Mikkonen and A. Taivalsaari. Using JavaScript as a real programming language. Technical report, Mountain View, CA, USA, 2007.
[28] OpenJDK. Nashorn. http://openjdk.java.net/projects/nashorn/.
[29] Phillips. Hue. http://www2.meethue.com, 2018.
[30] S. Porritt. jsPlumb toolkit. http://github.com/sporritt/jsplumb.
[31] D. Reed. Rethinking CS0 with JavaScript. SIGCSE'01, 2001.
[32] A. Saifullah, S. Sankar, J. Liu, C. Lu, R. Chandra, and B. Priyantha. Capnet: A real-time wireless management network for data center power capping. RTSS'14, Dec 2014.
[33] Z. Shelby, K. Hartke, and C. Bormann. The Constrained Application Protocol (CoAP). RFC 7252 (Proposed Standard), June 2014.
[34] UPnP Forum. UPnP device architecture 1.1. http://www.upnp.org/specs/arch/UPnP-arch-DeviceArchitecture-v1.1.pdf, Oct 2008.
[35] Witti. Notti. http://wittidesign.com/en/notti/.