Enabling Synergy in IoT: Platform to Service and Beyond

Michael P Andersen*, Gabe Fierro†, David E. Culler‡
Electrical Engineering and Computer Science, UC Berkeley
*Email: *m.andersen@cs.berkeley.edu, †gtfierro@cs.berkeley.edu, ‡culler@cs.berkeley.edu,*

***Abstract*—To enable a prosperous Internet of Things, devices and services must be extensible and adapt to changes in the environment or user interaction patterns. These requirements manifest as a set of design principles for each of the layers in an IoT ecosystem, from hardware to cloud services. This paper gives concrete guidelines learned from building a full-stack Synergistic IoT platform.**

***Keywords*-Internet of Things; Wireless sensor networks; Sensor motes;**

I. Introduction

The Internet of Things that we imagine involves far more than the mere ability of many miniature computational devices embedded in the fabric of everyday life to communicate. We expect that these devices will be specialized in ways reflecting the 'thing' they are a part of, that distinctive ensembles of connected things will provide rich functionality as natural-to-use applications and services, that space and proximity matter, as they dictate context and deliniate boundaries of applicability, trust, and authority, and that all of this will leverage the deep storage and processing resources of the cloud, as well as its potentially global visibility. This is a fundamentally heterogeneous world, and yet we imagine seamless, nearly spontaneous interactions among diverse collections of things working together - in a word, *synergy*.

And yet, the prelude to the IoT we see all around us today stands in stark contrast to this conception. While the smartphone is ubiquitous and wearable devices are everywhere, almost invariably for one to work with the other an application for the particular thing must be preloaded onto the phone and the two devices must be explicitly paired using a particular, common link and protocol. The situation is no better with Zigbee or z-wave 'things', essentially each requiring a product-specific gateway and unable to interact with the phones and wearables of the BLE universe - despite immense effort to develop detailed application profiles. WiFi scarcely improves the situation, despite inheriting the ability for a device from any vendor to communicate with any other; we still need, for example, a dedicated application for the phone to interact with the thermostat - or resort to interacting with its web-accessible avatar. Certainly the phone serves as an intermediary in many ways, possessing PAN, LAN, and WAN links, but strangely this largely means isolated vendor-specific stacks pass through it.

This situation led us to develop a complete IoT system in which to explore the issues of synergy at various levels, as illustrated by Figure 1. Simply applying the techniques associated with "the Internet" is not enough. We do need end-to-end communication amongst devices using distinct physical links, but the proximity and relationship of those devices matter, so decoupling through bridges and routers is not enough. We do need devices to access content provided by other devices in a uniform manner, but the path identifying that information is largely determined by the relationships between the providers and their context. Notification, events, and APIs are the norm, rather than `GET`-ting documents. Applications might be viewed as mash-ups of physically dependent services, but there also needs to be a principled way to assemble those ensembles from context.

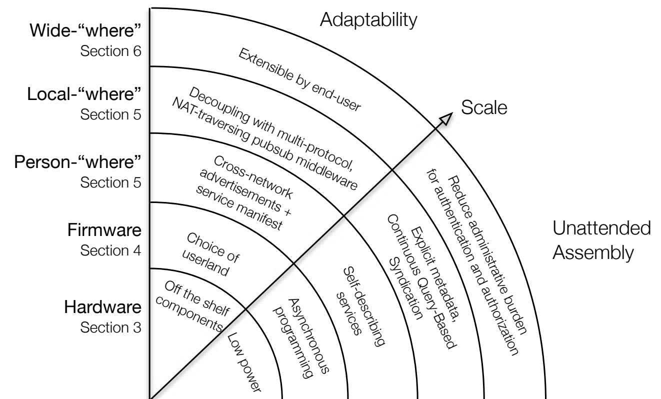


Figure 1: An effective IoT solution requires addressing adaptability and unattended assembly and operation at every tier.

This paper seeks to bring to the fore the key issues encountered in the quest to achieve synergy in the IoT. These issues arise at every level and they have some commonality. Enclosing complex, highly specialized behavior behind a simple interface is key. Service descriptions must be more than a specification to permit vendor interoperability; they should permit bootstrapping from context to avoid pre-configuration and should be accessible independent of the link between things, yet should not prohibit peer-to-peer interaction. Relationships should be extracted from metadata, and hence things should be notified as such metadata changes. "Middleboxes" have important roles to

play, including bridging, adaptation and routing amongst heterogeneous technologies, but also establishing points of presence and boundaries of trust.

We begin with a very brief description of our exploratory system to provide a concrete framing of the study. The remainder is organized in layers: first hardware, then firmware, then three scopes of interaction which we term person-where, local-where and wide-where - expanding conventional notions of PAN, LAN, WAN to include the interactions and services that occur in those domains.

II. Synergy

The system architecture developed to explore synergy in IoT is shown schematically in Figure 2.

Hardware: At the device level, we have introduced a new platform that brings together embedded wireless networking, wearables and "Maker" developments, typified by IEEE 802.15.4, BLE and Arduino peripherals, respectively. (See [6] for a complete description.) This platform is built around the *Storm* module, designed in 2013, with a Cortex-M4, 802.15.4 radio, and flash, mounted on an Arduino-compliant carrier, *Firestorm* (Figure 3), which provides BLE communication with a Cortex-M0+ SoC, and several sensors. This design point was intended to capture what embedded IoT might converge to, and, indeed, now several system-on-chip offerings provide ample flash, powerful MCUs and 802.15.4 or Bluetooth radios. Storm is extremely low power (2.3 μ in sleep with RTC active) and supports many peripherals (63 GPIO pins). This adds to the commercial ecosystem of wearables, embedded Linux boxes (Raspberry Pi's and Beaglebones), BLE tags, and such.

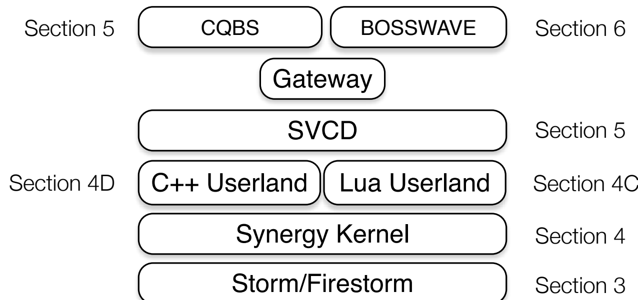


Figure 2: An IoT system architecture

Firmware: An extension of TinyOS [12] was developed for this platform that utilizes the newly available protection mechanisms (MPUs) to establish a clear user/system boundary in the domain of networked things lighter than a Pi. A syscall interface and language runtime was developed that exposes low-power, event-driven execution to user level and two language environments were created - Lua and C++. This allows instantiation of a breadth of user-programmable 'things' not represented in the commercial landscape.

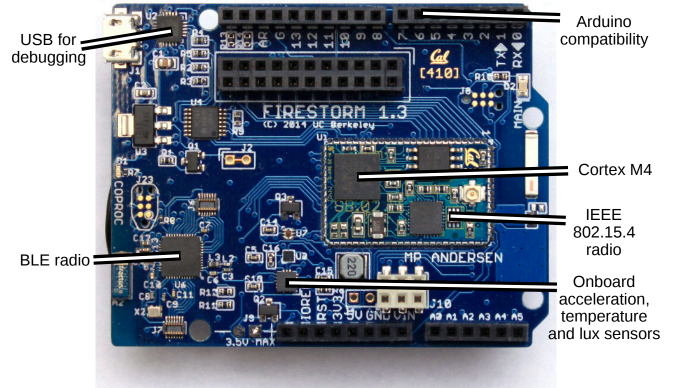


Figure 3: The Firestorm platform

Person-where: In this setting, we developed a self-describing service tier to enable automated assembly of purpose-driven ensembles at the scope of the individual person, which subsumes phones interacting with things in the environment and wearables interacting with spaces and things. Such assembly must not require pre-configured applications and bindings. Things project their API and services in a manner that allows the phone to bootstrap itself into the context, using complete descriptions in the cloud. This scope retains a sense of individual management.

Local-where: In a similar manner, collections of things can assemble into federated ensembles to provide services and interfaces that expand that of the individual devices. This relies on discovery of context and interfaces through queries to consistent metadata. Services and enclosing applications are associated with place, rather than person, and must operate unattended, requiring automated notification of changes as represented in the metadata. Local tier routing and computing resources bridge and translate heterogeneous elements.

Wide-where: Beyond simply tunneling local interactions to resources hosted in the cloud, broader-scale ensembles containing devices owned by multiple parties should also be able to be assembled into meaningful services and applications. To support this, a web of trust infrastructure is created; this establishes the namespace in which interactions occur, establishment of identity and delegation of trust.

III. Hardware

The hardware of an IoT device sets the stage for the capabilities of the entire stack. A device must be made adaptable by unifying communication modes: device-to-device, device-to-internet and device-to-person. It must be extensible and foster innovation by facilitating the reuse of available sensors and actuators. And it must do this without compromising on lower power operation - the tenet of unattended battery powered devices.

A. Communications

To create a true Internet of Things, we need features from multiple wireless communication protocols. Bluetooth Low Energy is the predominant technology in off-the-shelf smart devices as it is intended for connecting a human to a smart device. Looking forward, however, protocols like IEEE 802.15.4 are better suited for device-to-device interaction, having seen decades of research into automatic, unattended IP mesh network formation. BLE has only recently seen research into carrying IP traffic and peer-to-peer (as opposed to peripheral-to-phone) communication [19][14].

In addition, connecting an IoT ensemble to the Internet poses different challenges. Neither BLE nor IEEE 802.15.4 are known for ubiquitous Internet connectivity (although both *can* do so, by introducing new devices into the environment such as a gateway or mobile phone). The best protocol for this functionality is 802.11.x (WiFi). WiFi with Internet access is readily available in many of the places that IoT devices would reside.

At the time of writing, it does not seem like any of these protocols is in a position to subsume the others. It is only by utilizing all three that we can construct unattended IoT device ensembles.

Fortunately, this requirement to support multiple protocols is becoming increasingly less difficult: recent trends in SoC design and "turnkey" software abstractions have decreased the difficulty of supporting BLE and, to a lesser extent, 802.15.4. Improvements in lower power MCU (microcontroller) design and fabrication means that these dedicated peripheral controllers do not significantly increase the power budget of the platform.

B. Extensibility

For IoT devices to reach ubiquity, the barrier to entry must be low. In the embedded space, one of the ways this can be achieved is through re-use, lowering investment cost and therefore the cost of failure. This is exemplified by the Maker community where many good ideas can be prototyped using an ensemble of off-the-shelf controllers, sensors and actuators. The Firestorm successfully preserves this ability to rapidly innovate by maintaining pin-compatibility with Arduino shields, but still providing a platform that is useful for pilot-stage use by offering production-grade energy efficiency. This combination allows for rapid assembly of smart, Internet-connected devices without the expense and associated risk involved in hardware design and fabrication.

C. Low power

The whole-circuit design is also influenced by the goal of low power. There are often conflicts between the desire for adaptability or extensibility and the concerns relating to unattended use. In particular, sensors and debugging support can significantly increase power consumption if care is not taken.

As an example of where this disjunction compromises a platform, consider the Arduino Zero. Here, the MCU is very low power, and could enable piloting IoT devices based on the platform, with sleep currents of $< 10\ \mu A$. Unfortunately, the desire for easy debugging and support for high power peripherals raises the minimum sleep current by three orders of magnitude to 1.5 mA, reducing the maximum battery life on a pair of AA batteries from more than ten years to a month. With consideration of the interactions between the applications running on the platform, the peripheral components and the MCU, it is possible to design a platform that obtains low power operation while preserving ideal usability characteristics. The Firestorm has an idle current of 9.6 μA while still possessing USB debugging, multiple sensors and a power rail that can drive 800 mA - more than enough even for peripherals like WiFi radios. This is done by extending the mechanisms for low power that exist within the MCU - gateable power and clock domains - to the board as a whole.

IV. Firmware

The architecture and programming paradigms used in the firmware can be chosen to complement the hardware design and projected use cases of the platform.

A. Curtailing complexity

As low-power MCUs become increasingly capable, much of the complexity in IoT device development can be hidden from firmware engineers by software abstraction layers. Ironically, this complexity is primarily a result of the increase in device capabilities. For example, in order to achieve low power operation on modern, highly capable microcontrollers, the designers offer fine-grained control of peripherals and clock domains, so that only the features currently being used contribute to the power consumption. This highly hardware-specific power control must be mastered by any battery powered IoT device, a burden that increases the difficulty of application development. The solution to this, as used by the Firestorm, is to isolate this complexity in a kernel layer so that applications do not need to manage it explicitly.

As another example, advances in radio technology have allowed for the creation of low power software-defined-radios such as the CC2650 that can communicate with both 802.15.4 and BLE. The downside is that the software becomes more complex. To mitigate this, Texas Instruments provides the software control in ROM that executes on a separate MCU within the SoC, and the application interacts with a higher-level API. Similarly, the NRF51822 BLE SoC used on the Firestorm ships with a "soft-device" that provides a similar level of abstraction, implementing the Bluetooth stack and simplifying the interface to it.

These trends need to be carried through wherever possible in IoT framework development. By isolating complexity in

reusable self-contained modules (whether this be hardware or software), we lower the barrier to entry. Some embedded operating systems achieve this by providing libraries that are linked to at compile time, but this restricts the application programming model. By leveraging the memory protection unit (MPU) and dual stack pointers present on Cortex-M microcontrollers, the application and supporting kernel can be fully decoupled. This approach is clearly not new, but it is only recently possible to take this approach on embedded platforms while still remaining in the μA energy budget.

Communication between the application and the kernel on the Firestorm uses a syscall ABI. This allows the userland to be implemented in any language, and allows for preemption of the userland without any explicit cooperation from the application code.

B. Meeting the hardware half-way

When designing firmware for low power hardware, the biggest challenge is resolving the disjunction between the programming pattern used for application development and the inherent behavior of the hardware. For example, the most commonly used embedded programming language, C, is efficient at expressing sequential logic with blocking calls. Unfortunately, if a sequence of embedded device operations is expressed in this way, it leads to inefficient CPU usage and power. For example, if the analog to digital converter is exposed via a blocking API with the CPU spinning until the sampling operation is complete, it is obvious that more power will be consumed than if the CPU is allowed to execute other code or enter a low power mode while the sample is being acquired. If the event-based nature of the hardware is captured in the C application, it leads to fragmented logic. The code initiating the sample will occur in a different place than the code handling completion of the sample, which may be in an interrupt handler or a different callback function.

There are two means of elegantly resolving the conflict between the desire to have simple, readable code and the desire to make efficient use of the hardware. The first option is to use lightweight threads that are suspended during blocking calls to asynchronous operations. This is the method used by some IoT operating systems such as RIOT-OS [7]. The second option is to use closures to handle the completion events. Closures can be defined in the same place as the asynchronous operation initiation, with code that appears sequential. This method is widely used in Javascript based frameworks.

There are quite a few languages that have the necessary primitives to support closures as a means of expressing asynchronous events. C does not, but C++11, Lua, Rust, Javascript and Python do. We explore two generic use cases which fall on opposite ends of the application spectrum. Lua is a highly dynamic interpreted language that, as a consequence, trivially supports modification of code while it is running. For rapid development and use cases where the application logic is changing frequently (such as devices modifying other device behavior) languages like Lua are a good choice. Conversely, for long-lived applications that do not require runtime changes to the code, C++ is a good choice as it is resource-efficient and more deterministic than Lua.

C. A C++ userland

While many IoT applications can benefit from the agility of dynamic languages such as Lua, discussed below, C-type languages remain the staple of production embedded applications because of their predictability. We argue that using C++11 for application development offers significant advantages over existing C based approaches.

The principle advance that has led to C++ being a interesting embedded language was the introduction of lambda functions, which are essentially closures. Traditionally, an asynchronous operation in C is expressed as a split-phase pair. When a function is invoked, a callback is passed, along with a context object that is used by the callback to demultiplex the invoker and take appropriate action. The difficulty is that as the same callback may arise from multiple different sources, this demultiplexing quickly becomes unmanageable. For example, the `spi_complete` operation would need to determine where the SPI operation was invoked, and how to advance the state machine to proceed. By using closures, each invocation of the SPI operation can provide a *unique* callback in the scope of the invocation, with the context object constructed transparently as the variables captured by the lambda. The result is code that is far more legible.

This problem of demultiplexing asynchronous operations is not new; it has also been solved by making such operations appear synchronous, using threads. The difficulty here is that threads are very resource intensive. Often the stacks must be overprovisioned as they cannot be expanded at runtime given the lack of a virtual memory system. Furthermore, the entire memory for the stack is unavailable for use by other parts of the system while the thread is active. In contrast, the closure model only keeps the *referenced* variables in memory, and the stack does not persist from one closure invocation to the next (assuming closures are executed using a task queue). This means that application developers can create many more chains of asynchronous operations using closures than they could using threads, with the same resource footprint. The disadvantage of callback approaches is that it is syntactically less elegant than the thread based approach: long chains of asynchronous events can cause what is colloquially known as *callback hell* - very deep levels of nested functions. In our experience, this is well worth the increase in resource efficiency, and can be mitigated by refactoring.

One use of the C++ userland on Firestorm was for firmware on the smart chair discussed in Section VI. The challenge is that these chairs are deployed in remote locations, and must operate unattended for many months. The firmware implements reliable communication, a flash filesystem, control logic and other pieces of functionality, all involving highly asynchronous logic. The use of C++ closures greatly reduces the complexity in comparison to C and nesC designs, and therefore increases the reliability.

D. Lua userland

In addition to the production-grade C++ userland, we also constructed a Lua userland. While a Lua application uses more resources than a C++ application, it has several advantages. Applications can be prototyped interactively on the device via a serial or TCP shell. Individual symbols and variables can be added or replaced at runtime without rebooting the application. Programs can be modified at runtime to adapt to devices appearing in their ensemble.

As an example, this dynamic userland facilitates the creation of an infrastructure for measuring routing and network performance in a wireless mote deployment, bringing together past work on Active Networks [16], code dissemination [11], and network testbeds [20]. The ability to replace Lua symbols and functions over the network at runtime lets every mote in the deployment perform as an Active Network node that can dynamically constrain the routing topology, alter traffic generation, and choose which metrics to collect and report without having to manually program each individual mote between experiments.

In our realization, a distributed collection of Firestorm motes all run the TinyOS kernel with a Lua-based userland (see Figure 4) running a short program that defines a traffic generator, a reporting function, and initialization and termination conditions for the network experiment. As seen in Figure 5, this code can be sent over IP, either link-local or globally routed, or over Bluetooth. The Bluetooth link can be used to diagnose a broken 802.15.4 link and pull out the necessary debug information. This is particularly useful when diagnosing routing loops that would otherwise island the node and prevent real-time reporting from happening. We have used this platform extensively to study the applicability of the RPL [21] standard for IoT devices.

V. Person-where

The assembly of local, purpose-driven ensembles presents challenges in how to conduct discovery and account for heterogeneity in devices and services. At scale, these compositions cannot rely on pre-configured applications or the intervention of a human operator. Instead, devices must bootstrap themselves into an application by discovering nearby self-describing services.

The powerful, asynchronous userland environment enables the self-assembly of dynamic, cross-network appli-

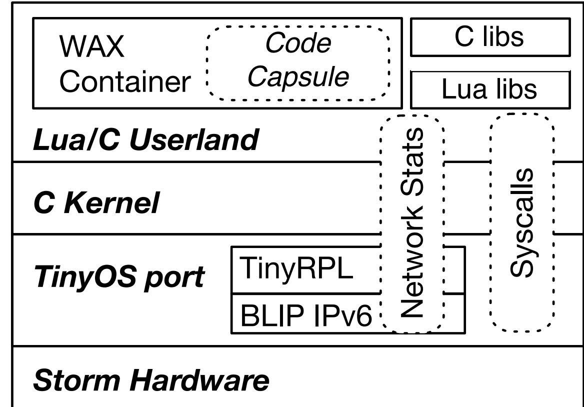


Figure 4: The full "Wireless Active Networks" (WAX) hardware/software stack for network experimentation. Kernel syscalls take advantage of the synergy between hardware and firmware to expose low-level metrics on the radio stack that would otherwise be difficult to incorporate.

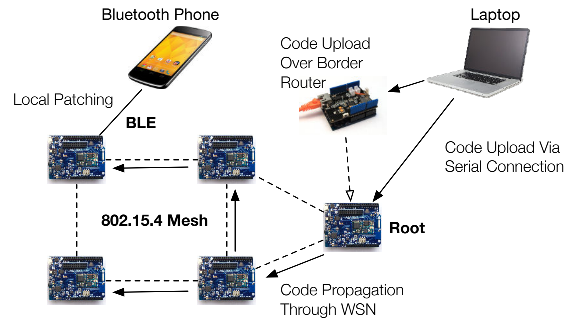


Figure 5: Modifying existing code over the network

cations within a personal area. The key functionality is a brokerless, local publish/subscribe discovery mechanism that is symmetric over IPv6/802.15.4 and BLE. We focus our discussion on devices interacting within a broadcast-domain, and defer our discussion of discovery and service composition in a local-area network to the next section.

A. Discovery

For the most part, current discovery mechanisms for networked things follow one of two patterns:

1) Pairwise master-client binding within a broadcast domain, usually using BLE (e.g. for wearables)
2) Service invocation in a local-area network, usually over an IP network

Despite often supporting both patterns, commercially-available platforms tend to partition functionality into one or the other. The Firestorm platform explores symmetrically exposing functionality both as a human-interfacing device as well as in connectionless machine-to-machine interactions. To provide for effective discovery of devices and invocation

SVCD Function	Description
`init()`	Initializes sockets and begins advertisements
`add_service(svc_id)`	Add a new service with the given identifier
`add_attr(svc_id, attr_id, write_fn)`	Attach a callback function for writes to the specified attribute
`notify(svc_id, attr_id, value)`	Notify subscribers on the given attribute of a new value
`subscribe(targetip, svc_id, attr_id, on_notify) -> subscription_id`	Subscribe to changes on an attribute by the specified provider
`unsubscribe(subscription_id)`	Unsubscribe from the indicated subscription
`advert_received(payload, src_ip)`	Triggered when a service advertisement is heard from a device identified by the `src_ip`. Payload includes list of services and attributes provided

Figure 6: API for cross-network service description and utilization. The implementation transparently uses both BLE and 802.15.4 system calls and less than 250 lines of Lua code

of services within a broadcast domain, a platform must attend to self-describing services, adaptability to usage patterns and event subscriptions.

In general, self-describing devices should communicate their capabilities as well as contextual information ("metadata") which allows a discovery process to determine the relationship between itself and a device. In personal-scale ensembles, which reside within a broadcast domain, the hearing of a service advertisement is enough to establish proximity and determine a relationship. Ensembles should be able to form themselves using only the service descriptions contained in heard advertisements.

Secondly, with a characteristic-oriented service advertisement framework, a service can be written for symmetric use over both BLE and 802.15.4, allowing a single definition of a service to adapt as users and devices change their interaction patterns.

A third feature that greatly simplifies application development is the ability to subscribe to changes in a device's attributes or characteristics. While this is possible using polling techniques (or natively using BLE GATT notifications), we wish to provide the application programmer with a unified API for creating and handling subscriptions across networks.

The SVCD framework integrates these design points into a simple, asynchronous API (Figure 6). Current discovery mechanisms are insufficient for meeting our objectives for one or more of the following reasons:

- discovery does not include a list of available services and characteristics, requiring some intermediary to supply this information (UPnP [13], DNS-SD [8])
- discovery cannot be performed in a peer-to-peer manner by devices, requiring an external coordinator (ZigBee [5])
- discovery mechanisms are not limited to a broadcast domain or otherwise lack sufficient contextual information for devices to determine relevance (UPnP, DNS-SD, Bonjour)
- services do not provide real-time subscriptions or notifications of data changes (ZigBee, DNS-SD)

B. *SVCD: synergistic service discovery*

Our solution presents a unified API, SVCD, for implementing self-describing services that advertise simultaneously over BLE and 802.15.4. To maintain compatibility with mobile phones, we retain use of the GATT for advertising over BLE, and use structured link-local multicast packets for advertising over 802.15.4.

An instance of SVCD is identified by a unique 2-byte ID derived from the MAC address of the mote running the instance. Each instance advertises a set of services; each service is composed of a set of read or read/write attributes. Services and attributes are indexed by unique 2-byte identifiers recorded in a GitHub-hosted central manifest file that lists full human- and machine-readable descriptions of the family of known services. Placing the manifest on GitHub means that mobile phones can easily discover and make use of local services discovered via BLE, even providing human-readable descriptions of the services to the end user. Most importantly, this can be accomplished without the phone application being explicitly programmed with knowledge of all possible services and attributes.

Figure 7 is an example of a service description contained in the service manifest file. The use of succinct identifiers and service/attribute grouping means there is a clean representation for both BLE and 802.15.4. The service/attribute distinction mirrors the GATT's service/characteristic structure and can be directly represented as such. For the 802.15.4 implementation, services and attributes are encapsulated in MessagePack [1], an efficient binary serialization protocol. The upshot is that even a size-restricted advertisement can contain a complete description of all services and attributes offered by a device. This is in contrast to UPnP, in which advertisements do not contain an actual description of ser-

```
"pm.storm.svc.fsSensors": {
  "id": "0x300f",
  "name": "FireStorm sensing profile",
  "desc": "Sensing profile for FireStorm",
  "attributes": {
    "pm.storm.attr.fsSensors.temperature": {
      "id": "0x401b",
      "name": "Temperature Reading",
      "format": [
        ["s8", "C", "Temperature in Celsius" ]
      ]
    },
    "pm.storm.attr.fsSensors.occupancy": {
        ...
    }
  }
}
```

Figure 7: Example of a service advertising on-board sensors for the Firestorm platform. The `format` field informs the data type, unit and description of the arguments or readings on an attribute.

vices, but rather a URL pointer to where those descriptions reside.

The API, as seen in Figure 6, provides a simple but powerful platform for constructing applications over discovered services. It unifies the creation of services and attributes across both 802.15.4 and BLE and provides facilities for subscribing to heard devices (`subscribe`) and publishing to subscribers (`notify`).

Asynchrony simplifies the API: service discovery and subscription are inherently event-based, and advertisements and invocation are usually handled asynchronously via timers or callbacks. In this way, the SVCD API can make use of the low-power features of the underlying hardware. What's more, the system is resilient to the inevitable moving, replacing and re-programming of service providers. This type of robustness to change is essential in a true Internet of Things, as we will explore in Section VI.

C. Applications

This framework simplifies IoT device creation. Consider a thermal comfort application leveraging the cooperation between hardware, firmware and personal-area services: at a high level, discovered temperature sensors drive the actuation of an off-the-shelf space heater towards maintaining a temperature setpoint (Figure 8).

First, leveraging an Arduino Relay Shield from the Maker community simplifies retrofitting the "dumb" space heater with actuation capabilities. This shield is attached to a Firestorm running SVCD, which communicates with a set of Firestorm-based temperature sensors distributed throughout a space exposed as a discoverable temperature service (Figure 7) using just a few lines of Lua code.

The application running on the space heater Firestorm advertises a setpoint attribute that takes a target temperature

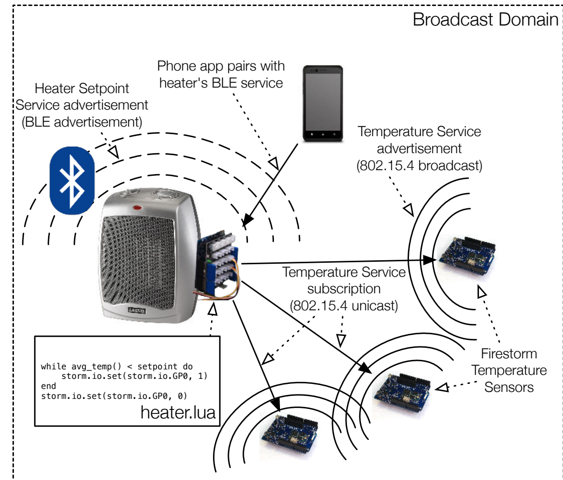


Figure 8: Example of composing locally-discovered services into an ensemble application using a mobile phone, 802.15.4 and BLE.

as input. Each of these advertised values (temperature and setpoint) are incorporated into the central manifest.

A user's phone discovers the setpoint attribute advertised by the space heater and prompts the user to input a desired temperature. Asynchronously, the heater listens to service advertisements, discovers the set of temperature sensors and subscribes to their values. This demonstrates the benefit of broadcast-domain discovery: the found sensors are known to be relevant to the application because they would only be heard if their measurements were physically relevant. The space heater's Firestorm averages these temperature sensors and actuates the heater if the measured area temperature is lower than the user-provided setpoint.

VI. Local-where

Composing applications over ensembles of devices in an area larger than a broadcast domain raises several challenges:

- The context of discovered devices and services cannot be assumed and must be explicitly managed. This predicates the need for a discovery service that can leverage rich metadata describing the set of available devices and services.
- Accounting for metadata must be complemented with a method for accounting for change in that metadata. Devices and environments inevitably change; an effective discovery service must provide a continually consistent view of which resources match an application's request.
- The number of applications and devices to account for is larger without the restriction of a broadcast domain. There is a need for a layer of indirection to reduce the load on low-power, low-bandwidth, duty-cycled embedded devices.

- More advanced applications will require historical data access as well as real-time streaming. It is intractable for embedded devices to provide these services directly.

This family of concerns can be addressed with the introduction of a local server that handles device and service discovery, management of device metadata, data archival and access, and a continuous query-based syndication (CQBS) mechanism for real-time data consumption via a multiprotocol broker. This central component is referred to as the *CQBS archiver*.

In accounting for these new challenges, two principles carry over from personal-area ensembles:

1) Devices and services need to be self describing, and
2) devices, services and encompassing applications need to operate unattended.

A. Metadata-driven discovery

Device descriptions require a principled representation that supports granular discovery of relationships between devices and services, rather than solely on the names of which interfaces a device provides.

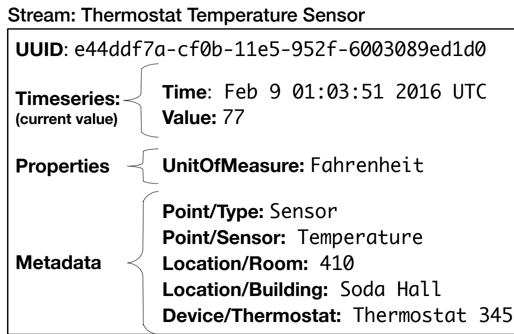


Figure 9: A stream example of a thermostat's temperature sensor

Each device advertises itself to the CQBS archiver as a set of streams: a *stream* (Figure 9) is a virtual representation of a specific sensor or actuator channel, indexed by a universally unique identifier (UUID). Each stream has two associated structures produced by the device and stored at the archiver:

- A timeseries: a single progression of `<timestamp, value>` pairs. A value is the state of the stream at the provided timestamp. Values are not limited to numerical data.
- A metadata set: a bag of structured key-value pairs describing the context of the device and the details of the stream (i.e. what it senses or actuates)

Metadata describes the context of a device and its services, which may include parameters such as location, ownership and role[1] of the device as well as attributes of each sensor or actuator, including engineering units, data type and write properties. Applications discover devices and services by expressing to the CQBS archiver a SQL-like query that operates on this metadata; these queries describe the *relationships* between devices and services.

The challenge here is that metadata often changes: for example a device may be moved, a modular sensor platform may be altered, or the installation environment may change. The method of continuous query-based syndication, implemented at the archiver, addresses this by dynamically updating the results of an application's discovery query and informing the application in real-time. Devices notify the archiver when their metadata changes, enabling the archiver to maintain a consistent view of the state and context of all devices and services.

For applications to operate unattended, they must be able to a) express a robust definition of the set of devices and services it needs, and b) maintain a consistent view of that set even as devices and their context change. Continuously evaluated metadata-driven queries allow applications to be informed of changes to the set of services they use.

B. Continuous query-based syndication

An application initializes a continuous subscription by sending a query to the archiver, which evaluates it and returns the initial set of matching devices and services, but persists the query. As metadata updates arrive at the archiver (either via a device's update or an administrative command), the archiver locates the affected queries and reevaluates them. If the results of the query have changed, the difference is sent to each application subscribed to that query.

An example can be found in the simplified deployment in Figure 10. An application wants to discover the set of temperature sensors in room 410 in building Soda Hall, which is expressed in the illustrated query.

CQBS delivers updates to the set of streams captured by a metadata query, so that a subscriber always has an up-to-date view of the resources it is using. This is in contrast to systems that only provide static queries over sensor stream metadata [15] or provide query-based subscriptions that do not update the publisher-consumer binding as metadata changes [9].

These concerns remain largely unaddressed by modern systems concerned with the composition of services over networked things. These systems – including CORBA [17], Jini [18], AllJoyn [3] and Iotivity [10] – generally offer limited discovery capabilities that do not identify how the implementer of an interface is related to other resources required by an application. In other words, this approach assumes that the application or application developer has enough prior contextual information on the set of discovered resources to disambiguate which are relevant to the application. The detection of changed metadata is usually relegated to periodic advertisements and client- or server-side timeouts, which can still result in stale data, particularly

[1] E.g. lighting, heating, printing, visualization, etc

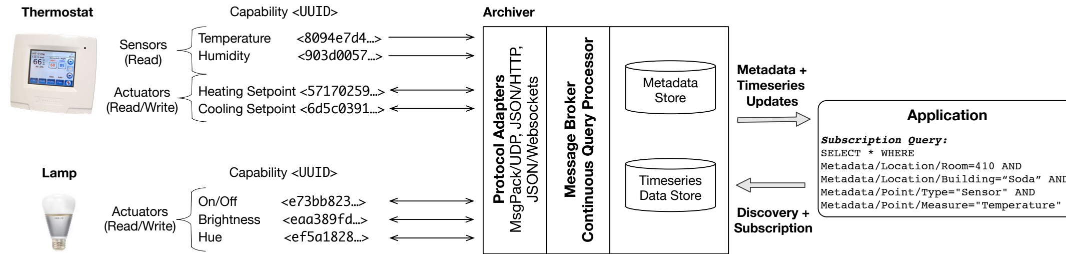


Figure 10: The archiver is the central component in a local-area IoT deployment. Devices register themselves with the archiver using rich metadata to describe their context. Applications can then discover these devices by expressing queries over the metadata. Applications can also access historical and real-time (CQBS) data from the devices by querying or subscribing to the archiver. `SELECT *` indicates that the application wants to receive all metadata for each matching stream.

in the case of real-time data-streaming applications.

C. Discovery of modular interfaces

IoT middleware often groups too much functionality into a single interface meaning that the mapping of device to interface is not straightforward: a device may not offer all functions an interface expects, or it may offer more. As the heterogeneity of devices grows, a device may only be adequately "covered" by one or more (incomplete) interfaces. As an example, consider the the AllJoyn Lighting Service [4] which advertises binary control of lighting as well as brightness, hue and *power consumption*. Because AllJoyn only supports discovery on identity (names) and interface names [2], applications can encounter two problems. Firstly, an application cannot query specifically for a brightness or hue capability, and there is no guarantee that those functions are even offered by the underlying lighting device. Secondly, an application interested in power consumption *must know to query for the lighting interface* in order to get the power monitoring services. In practice, this requires all applications to have knowledge of all features of all possible interfaces. This argues for more granular descriptions of device capabilities.

For example, consider how a thermostat would be represented as streams to the CQBS archiver: a thermostat's "sensors" would include its temperature and humidity readings and its "actuators" would include heating and cooling setpoints and fan settings. Each of these capabilities would be described independent from each other and in a standard way; this allows applications to discover individual streams that provide the specific functionalities required, rather than searching for all available interfaces that may or may not provide them.

D. Local-area applications

The utility of CQBS can be demonstrated in the construction of an IoT application that integrates a collection of smart, networked chairs with a building's HVAC system. The chairs possess a family of sensors – occupancy, temperature and humidity – and actuators – heating strips and cooling fans – that can mitigate user discomfort. On a per-room basis, the application samples the temperature readings from the chairs and combines this information with whether occupants are using the heating or cooling features of the chairs. If all occupants in a room have enabled the heat setting on their chairs, then the room is likely too cold. The application can then adjust the HVAC settings for that room to increase user comfort.

The challenge of integrating mobile devices like chairs with static infrastructure in a building is accounting for stale metadata. Chairs often move from office to office, and building components such as thermostats may be replaced. If the application is "hardcoded" to a set of chair sensors and to a set of HVAC endpoints, any change in those devices would invalidate the control decision. Using CQBS, the application can subscribe to the relationship between a room and the chairs it contains, and use notifications of changed metadata to adjust its control loop.

VII. WIDE-WHERE

When creating larger scale ensembles, or ensembles that contain devices owned by multiple parties, there are additional concerns over those discussed above. We can summarize these as:

- How is identity managed at scale?
- How are devices, services and their interfaces named and referred to? How do we group these?
- Where are these references stored, and where would a user or service look to find other devices and services?
- Given a reference, how does that get resolved to an actual IP endpoint to send traffic to?
- Many IoT devices and services are not directly reachable from the Internet (NAT for example). How would these devices and services communicate?
- How can we fully decouple producers from consumers?
- How would secure transport mechanisms such as SSL work at such a large scale? Can we lower the administrative burden?
- How would permissions for operations on IoT devices and services work? Can we lower the management

overhead?

BOSSWAVE, or **BW**, is an exploration of the problem, and one possible solution. Initially for ensembles of services and devices in smart buildings (hence Building Operating System Services Wide Area Verified Exchange), this system is an extension of the principles in the previous two sections.

A. Identity

When considering security and identity, we must consider that an IoT system consists of more than just devices and services, but also the people that own and manage them. All of these components require a form of identity to be able to describe a security policy. A scalable definition of identity must be easily verifiable, but also easy to create. An SSL certificate, for example, does not quite meet these criteria as it requires the cooperation of a hierarchy of certificate authorities to generate. The correct CA certificates also need to be available in order to verify an identity.

The approach used by BW to solve this problem is to define identity using a small ECDSA public key and define any person, device or service in possession of the corresponding private key as *equivalent* to the *entity* that the key identifies. Identity can then be verified using signatures, which does not require the cooperation of any third parties.

B. Namespaces and interfaces

In order for an interface to be referenced by other services, it needs to be named. For management and discovery purposes it is also convenient to be able to group services into *namespaces*. For example, an organization would create a namespace and group devices and services within that namespace so that searches for services and devices based on metadata-defined relationships have a natural bound.

To establish resource names, BW uses URIs where the first element of the URI is the public key of the entity that *owns* the namespace. This definition of a namespace eliminates the resolution step required to determine which entity owns the namespace and can grant permissions on it. This is in contrast to SSL certificates which only prove that some trusted authority *believes* a domain belongs to an identity. Here the namespace name is *defined* by the identity owning it.

To obtain an IP address of a server to communicate with given a URI, BW uses a DNS resolution on the namespace key (this could also be a lookup in a DHT). The resulting SRV record contains the IP address of the BW server that routes traffic for that namespace and that server's public key. The whole record is signed by the namespace's private key, i.e. the URI contains all the information required to verify that the DNS record is correct, so the record cannot be faked.

C. Communication patterns

In the Internet of Things, it is advantageous to have middleware relaying messages between devices so that firewalls and NAT appliances do not prevent devices or services from communicating and so that increased consumer load does not affect producers. The BW *router* is responsible for this relaying. Devices and services connect to a BW router running locally (or nearby) that they trust completely, communicating using a simple plaintext protocol that is designed to be easily implementable in a variety of languages. As part of the initial handshake, the client transmits its private key to the trusted router. The local trusted router then performs the necessary cryptographic operations to sign outgoing messages and verify incoming messages. The namespace resolution process described above yields the information for a BW server that will route traffic and store state. This is called the *designated router* for the namespace. Based on the resource's namespace, the local router will forward the message to the appropriate designated router. The designated router must be globally reachable, but a client's local trusted router can be behind a NAT.

To provide secure transport, preventing snooping of traffic and impersonation of the designated router, SSL is used for communication between BW routers. To avoid the administrative (and financial) overhead of obtaining CA signed certificates, self-signed SSL certificates are used and made secure by adding an additional signature made using the router's private key. In this way, the designated router cannot be impersonated or subjected to man-in-the-middle attacks as the trusted local router knows what the designated router's public entity key is in advance (it is in the DNS record that is signed by the namespace entity).

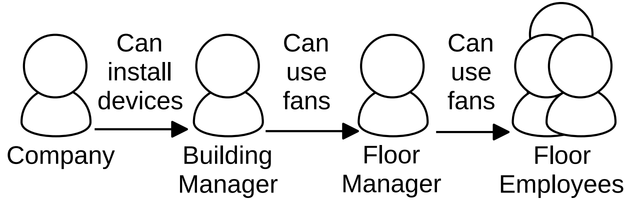
D. Permissions

While permissions in the Internet of Things can potentially be very complex, it is worth considering the simplest possible permission model that scales well, both technically and administratively:

- Permissions are granted directly to entities, not intermediate forms of identity that require runtime resolution
- Permissions are granted and revoked by the same people who manage the devices and services that the permissions affect
- Permissions are verifiable with minimal trust of external parties and no communication to external servers

BW obtains these by basing its permission system on the de-facto method of permission management between humans: peer to peer requesting and granting with delegation to trusted intermediaries.

To make this clear, consider the following scenario. A company's building manager installs some overhead fans in a shared office space, and tells the floor manager that she is allowed to control the speed. The floor manager then tells the other employees in that space that they should feel free to turn on the fans whenever they feel uncomfortable. The exchange of permissions looks as follows:



In BW, the permissions are granted in exactly the same way. Each of the "people" in the above figure translates to an entity. The company, from which all the permissions originate, administers a namespace `bw://`K_C`/`. Each arrow in the diagram is declaring that a source entity trusts a destination entity with certain permissions on a *resource*. These *declarations of trust* or DOTs are formally a tuple of the granter's key K_{from}, the grantee's key K_{to}, a resource identifier (URI), and a set of permissions P. To make the tuple unforgeable, we append a signature over $(K_{from}, K_{to}, URI, P)$ created using the granter's private key. As these DOTs are tamper proof, we can store them in a DHT or any other public repository.

When an employee in the room sends a message to turn on a fan, the BW message includes a standalone proof of permissions in the headers. This proof is a chain of DOTs where the grantor of each DOT matches the grantee of the previous DOT. The permissions granted by the chain are the intersection of the permissions granted by each individual DOT. For a BW router or a message recipient to consider a chain valid, it must begin with the namespace entity, and end with the entity that signed the message.

This permissions model satisfies the design principles set out above. As our notion of identity allows for non-interactive verification of message origin via signatures, granting permissions directly to identities is very effective in reducing complexity and vulnerability. By leveraging chains of DOTs, we create a decentralized web of trust model that allows the individuals making human trust decisions to directly create the digital representations of that trust. Furthermore, the web of trust model does not rely on third parties or central servers to manage permissions - messages contain self-standing proofs of trust and can be verified without contacting external servers or maintaining a database of certificate authorities. This characteristic is useful for IoT devices, as it means that we do not incur expensive network round trips in order to verify incoming messages.

E. Persistent data

In addition to handling messages intended for real-time consumption, BW also allows clients to persist messages to a URI that can then be queried later. This is used, for example, to store contextual information about devices and services. The *persist* and *query* permissions are also granted via DOTs. In order for a service or device to find other services and devices, it builds a DOT chain giving it query permissions on URIs matching a certain pattern (denoted by the DOTs). It can then query the BW router for persisted messages within that URI set, or subscribe to those URIs. Within each interface in the namespace, metadata regarding the interface such as where the sensor is located, is stored using persist messages. As an example, an application could be built that utilizes HVAC data from multiple companies' buildings to determine if a given building is consuming more or less heating energy than buildings of similar construction. Once appropriate permissions on each of the building namespaces is acquired (via personal interactions), the relevant streams of information can be located by querying for persisted metadata. The application can verify that all the metadata and data it is basing decisions on originated from trusted principles as the DOT chains are present in the persisted objects.

VIII. Conclusion

A True IoT device is one that embodies the spirit of the Internet - a heterogeneous network infrastructure connecting dissimilar machines running a diversity of software. The success of the Internet and the modern World Wide Web is in no small part due to a careful design that leverages these diverse technologies, being mindful of their strengths and weaknesses, to create something more than the sum of its parts.

This spirit, when applied to the Internet of *Things* drives us to architect systems that embrace and utilize the differences between available technologies, and mediate the often conflicting requirements of each layer of the stack comprising an IoT ensemble. Throughout the stack, a core principle has been that of enabling unattended interoperability between devices.

At a hardware level, the myriad of connectivity options must be considered complementary, rather than competing, to achieve interoperability between devices, services and people. Furthermore, interaction with the thriving industry of off-the-shelf sensors, actuators and development platforms such as those from the Maker movement is central to creating the environment of low-risk innovation required to fuel the success of the Internet of Things at a meaningful scale.

At the firmware layer, embedded programming can borrow from event-based software architecture developments, leveraging patterns that better mirror the asynchronous nature of the hardware beneath it, and the services above it. This results in more understandable and more power-efficient code, a key requirement for unattended battery-powered devices.

At a person-scale, the use of a dynamic application programming environment enables an adaptable service infrastructure that can borrow security from the implicit trust inherent in physical proximity to allow devices to discover and use each other without human interaction. These capabilities allow a smart device to leverage nearby devices

based on the functionality they advertise, rather than their identity.

By leveraging a central coordinating archiver, ensembles can scale across a local area. The addition of more powerful device descriptions in the form of metadata allows for ensembles to be predicated on arbitrary device context such as its function, location or user-defined group. This context can be manipulated externally by a human or a service, making the system more resilient to changes over time. Ensembles defined by context can adapt unattended as new devices are added and old ones are retired.

Finally, scalable, secure ensembles can be constructed across a wide scale by utilizing a decentralised pub/sub system that manages permissions in a manner isomorphic to human management of permissions - peer to peer. This structure maintains the advantages present in the local-where solution, such as ensembles based on relations, not identity, but addresses the problems that arise when principals come from multiple administrative domains and there is no longer complete trust between all individuals. Scalability in the IoT goes beyond computational power, and involves the administrative load that devices place on people. Unattended operation implies not just the device itself, but the security configuration as well. A web-of-trust security model minimizes the overhead of converting human permissions into technical ones, and allows unattended isolation of devices from untrusted traffic, even as the set of authorized parties changes.

Acknowledgments

This material is based upon work supported by the Fulbright Scholarship Program and National Science Foundation under grants CPS-1239552. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] MsgPack. http://msgpack.org/index.html, 2015.

[2] A. Alliance. Advertisement and Discovery. https://allseenalliance.org/framework/documentation/learn/core/system-description/advertisement-discovery, 2015.

[3] A. Alliance. AllJoyn: proximity based peer-to-peer technology. https://www.alljoyn.org, 2015.

[4] A. Alliance. Getting Started with the AllJoyn™ Lighting Service Framework 15.04 . https://wiki.allseenalliance.org/_media/tsc/lighting/getting_started_alljoyn_lighting_service_framework_15.04_lighting_controller_service.pdf, 2015.

[5] Z. Alliance. Zigbee specification, 2006.

[6] M. P. Andersen, G. Fierro, and D. Culler. System design for a synergistic, low power mote/ble embedded platform. In *Information Processing in Sensor Networks, 2016. IPSN 2016*. IEEE, 2016.

[7] E. Baccelli, O. Hahm, M. Günes, M. Wählisch, and T. C. Schmidt. Riot os: Towards an os for the internet of things. In *Computer Communications Workshops (INFOCOM WKSHPS), 2013 IEEE Conference on*, pages 79–80. IEEE, 2013.

[8] S. Cheshire and M. Krochmal. DNS-based service discovery. Technical report, 2013.

[9] S. Dawson-Haggerty, X. Jiang, G. Tolle, J. Ortiz, and D. Culler. sMAP: a simple measurement and actuation profile for physical information. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, pages 197–210. ACM, 2010.

[10] L. Foundation. IoTivity. https://www.iotivity.org, 2015.

[11] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 81–94. ACM, 2004.

[12] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, et al. Tinyos: An operating system for sensor networks. In *Ambient intelligence*, pages 115–148. Springer, 2005.

[13] B. Miller, T. Nixon, C. Tai, M. D. Wood, et al. Home networking with universal plug and play. *Communications Magazine, IEEE*, 39(12):104–109, 2001.

[14] Z. Shelby, J. Nieminen, T. Savolainen, M. Isomaki, B. Patil, and C. Gomez. IPv6 over BLUETOOTH(R) Low Energy. IETF RFC 7668, Oct. 2015.

[15] A. Sheth, C. Henson, and S. S. Sahoo. Semantic sensor web. *Internet Computing, IEEE*, 12(4):78–83, 2008.

[16] D. L. Tennenhouse and D. J. Wetherall. Towards an active network architecture. In *DARPA Active Networks Conference and Exposition, 2002. Proceedings*, pages 2–15. IEEE, 2002.

[17] S. Vinoski. CORBA: integrating diverse applications within distributed heterogeneous environments. *Communications Magazine, IEEE*, 35(2):46–55, 1997.

[18] J. Waldo. The Jini architecture for network-centric computing. *Communications of the ACM*, 42(7):76–82, 1999.

[19] H. Wang, M. Xi, J. Liu, and C. Chen. Transmitting ipv6 packets over bluetooth low energy based on bluez. In *Advanced Communication Technology (ICACT), 2013 15th International Conference on*, pages 72–77. IEEE, 2013.

[20] G. Werner-Allen, P. Swieskowski, and M. Welsh. Motelab: A wireless sensor network testbed. In *Proceedings of the 4th international symposium on Information processing in sensor networks*, page 68. IEEE Press, 2005.

[21] T. Winter, P. Thubert, A. Brandt, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, J. Vasseur, and R. Alexander. RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks. RFC 6550 (Proposed Standard), Mar. 2012.